VU Software Engineering 2
VU-Leiter: Prof. Uwe Zdun
Gruppenarbeit Task 2 – Design Patterns in Practice 1

TASK 2 – DEFENSIVE PROGRAMMING

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

Defensive Programming wird im Task-2-Projekt u.a. in Form von **Exception Handling** verwendet. Bei Fehlern, die nicht ignoriert werden sollen/können, wird an der jeweiligen Stelle eine Exception geworfen, die dann durch die passenden catch-Blöcke aufgefangen wird. Wenn Exceptions auftreten, werden außerdem entsprechende Informationen ausgegeben bzw. für den Programmierer angezeigt, damit klar ist, welches Problem/welche Exception aufgetreten ist.

Die folgenden Code-Beispiele zeigen die Verwendung von Exceptions im Projekt.

Code-Beispiel 1:

```
public void readCSV(){

    BufferedReader bufferedReader = null;
    try {

        bufferedReader = new BufferedReader(new FileReader(csvFilePath));
        String text;
        int firstLine = 0;
        int size = 0;
        while ((text = bufferedReader.readLine()) != null) {
            if (firstLine == 0){
                firstLine = 1;
            }else if (firstLine == 1){
                title = text.split(";");
```

```
                data = new Object[getCountOfLine()][title.length];

                firstLine = 2;
            }
            else{
                data[size]= text.split(";");
                size++;
            }

        }

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (bufferedReader != null) {
            try {
                bufferedReader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    setTableWithTitle();
    title = getTitelList(title.length);
}
```

Im Code-Beispiel 1 aus der Klasse CSVManager.java wird Exception Handling in der Methode readCSV verwendet. Von dem angegebenen Dateipfad soll eine CSV-Datei unter Verwendung eines BufferedReader gelesen werden. Dafür wird zunächst ein try-Block verwendet, in dem Exceptions auftreten können, während die Datei ausgelesen wird. Es kann sein, dass die CSV-Datei am angegeben Pfad nicht gefunden werden kann. Daher wird in diesem Fall eine FileNotFoundException geworfen, die dann im catch-Block für FileNotFoundExceptions gefangen wird und die Meldung wird für den Programmierer ausgegeben über die printStackTrace Methode. Genauso erfolgt die Behandlung der IOExceptions. Der finally-Block wird immer ausgeführt nach dem try-Block – auch wenn eine unvermutete Exception auftritt. Daher kann im finally-Block auch eine Art „Code Clean-Up“ durchgeführt und Vorgänge, damit keine werden, was bedeutet, dass wie z.B. hier in Code-Beispiel 1 das geöffnete BufferedReader-Objekt geschlossen wird (mit bufferedReader.close()), falls dies im try-Block nicht passiert ist oder die Ausführung unterbrochen wurde.

Code-Beispiel 2:

```
public void writeCSV(char separators){

    try {
        Writer writer = new FileWriter(csvFilePath);
        StringBuilder builder = new StringBuilder();
        for (int i = 0; i  < title.length; i++){
            builder.append(title[i] + separators);
        }
        builder.append("\n");
        for (int i = 0; i < data.length; i++){
            for (int f = 0; f < data[0].length; f++){
                builder.append(data[i][f] + ";");
            }
            builder.append("\n");
        }

        writer.write(builder.toString());
        writer.close();

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

}
```

Im Code-Beispiel 2 aus der Klasse CSVManager.java wird Exception Handling in der Methode writeCSV-Methode verwendet. Es kann sein, dass im try-Block eine IOException auftritt, wenn die CSV-Datei aus ihren einzelnen String-Teilen zusammengesetzt wird. Falls eine IOException auftritt, wird diese im entsprechenden catch-Block gefangen und es wird eine Ausgabe über die Methode printStackTrace gemacht, damit der Programmierer sieht, was geschehen ist.

Code-Beispiel 3:

```
btnsaveAsODF.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        int returnVal = fileChooserOpen.showSaveDialog(null);

        if (returnVal == JFileChooser.APPROVE_OPTION){
            String path=fileChooserOpen.getSelectedFile().getAbsolutePath();
            if (path.toLowerCase().endsWith(".ods") && model != null){
                final File file = new File(path);
                  try {
                    SpreadSheet.createEmpty(model).saveAs(file);
                } catch (FileNotFoundException e1) {
                    e1.printStackTrace();
                } catch (IOException e1) {
                    e1.printStackTrace();
                }
            }else{
                JOptionPane.showMessageDialog(frame,
                        "The file name must be '*.ods'",
                        "Inane error",
                        JOptionPane.ERROR_MESSAGE);
            }
        }

    }
});
```

Im Code-Beispiel 3 aus der Klasse Main.java im Package at.ac.univie.view kann eine Exception während Dateiauswahlprozess auftreten. Es kann sein, dass im try-Block die Datei nicht gefunden werden kann anhand des Dateipfades. Dann wird eine FileNotFoundException geworfen, die im entsprechenden catch-Block behandelt wird. Es kann auch sein, dass eine IOException auftritt – auch diese wird im entsprechenden catch-Block gefangen und behandelt.
Im Sinne des Defensive Programming wird in dieser Methode auch darauf geachtet, dass, wenn man den button (btnsaveAsODF) anklickt, eine Datei mit Endung .ods (i.e. Open Document Spreadsheet Dateityp) zum Speichern eingegeben wurde. Falls dies nicht der Fall ist, wird dem User eine Fehlermeldung angezeigt, dass der Dateiname die Endung .ods haben muss, damit die Speicherung als Open Document Spreadsheet durchgeführt werden kann.

Im Sinne des Defensive Programming kann es an geeigneter Stelle auch vorteilhaft sein, bei der Fehlerbehandlung einen **neutralen („harmlosen“) Wert zurückzuliefern** bzw. die **angefragte Operation nicht durchzuführen**, wenn dies keinen Sinn macht oder schwere Fehler in Programmausführung zur Folge hätte.

Code-Beispiel 4:

```
public class ArithmeticOperationsDivision implements IOperations{

    @Override
    public void execute(String eingabe, JTable table) {
        int operPosition = eingabe.indexOf("/");
        double first = Double.parseDouble(eingabe.substring(1, operPosition).replace(',', '.'));
        double second = Double.parseDouble(eingabe.substring(operPosition+1, eingabe.length()).replace(',', '.'));
        if (second!=0)
            table.getModel().setValueAt(((first/second)+"").replace('.', ','), table.getSelectedRow(), table.getSelectedColumn());
        else
            table.getModel().setValueAt(first, table.getSelectedRow(), table.getSelectedColumn());

    }

}
```

Im oben stehenden Code-Beispiel 4 aus der Klasse ArithmeticOperationsDivision.java im Package at.ac.univie.logik.strategyPattern kann es sein, dass der Benutzer eine Division durch Null eingibt. In diesem Fall wäre keine Ergebnisberechnung möglich und die Programmausführung würde einen Fehler produzieren. Daher wird hier zur Fehlerbehandlung keine Division durchgeführt, wenn der zweite Wert (Variable double second) den Wert Null hat. Der Divisionsvorgang wird nicht durchgeführt und an der Stelle der angefragten Divsion durch Null wird der erste Wert (Variable double first) gesetzt. Alternativ könnte hier z.B. auch statt der Division durch Null nur der Wert Null zurückgeliefert werden und dies in der betroffenen Zelle gesetzt werden, damit kein Fehler entsteht und die Ausführung ohne Unterbrechung weitergeht.

Im Projekt wurden auch **Debugging-Hilfen** verwendet, damit das Verhalten der betracheteten Methoden während dem Programmieren analysiert werden kann. Diese Debugging-Hilfen werden, nachdem die Methode fertiggestellt wurde, entweder als Kommentar erhalten oder ganz entfernt aus dem Quellcode.

Code-Beispiel 5:

```
    double summe = 0;
    if (FromCellRowPosition != TO_CELLRowPosition)
    {

        for(int i = FromCellRowPosition; i <= TO_CELLRowPosition; i++){
            summe += Double.parseDouble(((String) table.getModel().getValueAt(i,TO_CELLColumnPosition)).replace(',', '.'));
        }

    }else if(FromCellColumnPosition != TO_CELLColumnPosition){
        for(int i = FromCellColumnPosition ; i <= TO_CELLColumnPosition; i++){
            summe += Double.parseDouble(((String) table.getModel().getValueAt(FromCellColumnPosition,i)).replace(',', '.'));
        }

    }
    table.getModel().setValueAt((summe + "").replace('.', ','), table.getSelectedRow(), table.getSelectedColumn());
    System.out.println("Summe: " + summe);
}
```

In der Methode execute der Klasse SUM.java aus Code-Beispiel 5 wurde während dem Programmieren die Debugging-Hilfe System.out.println(“Summe: “ + summe); (siehe Ende des Code-Abschnitts) verwendet, damit man überprüfen kann, ob sich die Methode richtig verhält. Diese Ausgabe kann während dem Programmieren nützlich sein. Nach der Fertigstellung der Methode wird diese der Debugging-Hilfe dienende Ausgabe aber wieder gelöscht bzw. als Kommentar belassen. So wie hier in der Klasse SUM.java, wurden auch für die Funktionen COUNT und MEAN entsprechende Debugging-Hilfen erstellt, die bei der Analyse des Verhaltens der execute-Methoden unterstützend waren.

Code-Beispiel 6:

```
public void changeData (int row, int col, Object newData){
    data[row][col] = newData;
    //System.out.println(data[row][col]);
}
```

Im Code-Beispiel 6 wird die Methode changeData der Klasse CSVManager.java gezeigt. Diese enthält eine als Debugging-Hilfe benützte Ausgabe, die anzeigte, welcher Inhalt tatsächlich durch das übergebene Obejct newData in der Instanzvariable data an der Stelle der entsprechenden Zelle gesetzt wurde. Diese zusätzliche Ausgabe zur Code-Analyse wurde nach Fertigstellung der Programmierung wieder entfernt.