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

TASK 2 – 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

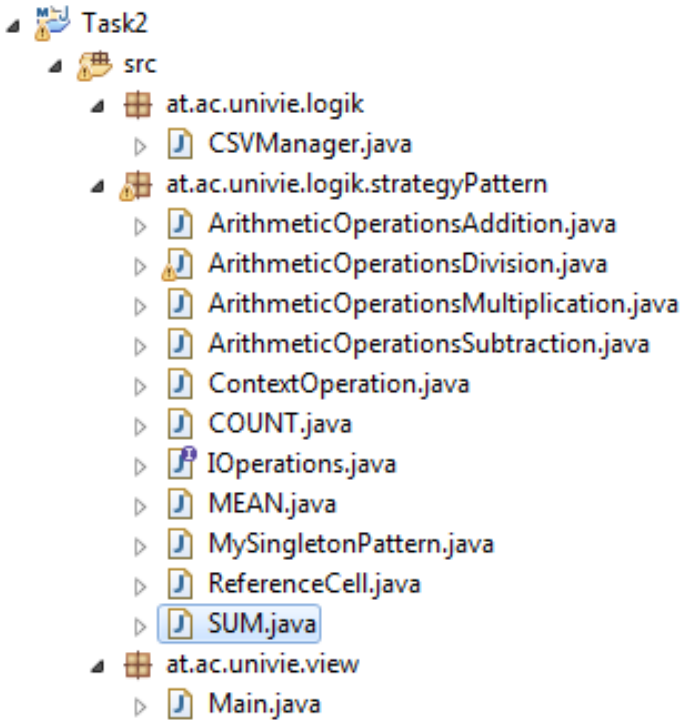
In Bezug auf Coding Practices wurden insbesondere die Namenskonventionen für Methoden und Klassen, die Kommentierung und die Empfehlungen für die Formatierung und logische Organisation des Codes umgesetzt. Mit Hilfe dieser eingesetzten Coding Practices soll die Lesbarkeit, Erweiterbarkeit und Wartbarkeit des Codes erleichtert werden.

Im folgenden Bereich wird anhand von einigen Beispielen aus der Implementierung von Task2 dargestellt und diskutiert, welchen Nutzen die Coding Practices in diesem konkreten Fall haben und wie sie umgesetzt wurden.

Die Coding Practices der einheitlichen Datei-Struktur, logischen Organisation der Dateien und der passenden Namensgebung der Source-Code-Files wurde im Projekt umgesetzt. Die untenstehende Abbildung zeigt die Organisation der Projektteile in Packages. Im Package at.ac.univie.logik befindet sich der CSVManager, der das Handling von CSV-Dateien durchführt. Im Package at.ac.univie.logik.strategyPattern befinden sich die für das Strategy Pattern verwendeten Komponenten, wie z.B. das Interface IOperations, das die execute-Methode anbietet, die verschiedene Implementierungsformen besitzt. Im Package at.ac.univie.view befindet sich die Klasse Main, die das Application Window aufgerufen und dem Benutzer das Fenster zur Verfügung stellt.

Geplante Erweiterungen der Datei-Struktur sind ein Package namens at.ac.univie.model (in dem sich die für die Speicherung von Formeln in .ods-Dateien notwendigen Elemente befinden) und eine Erweiterung des Package at.ac.univie.view um die Klassen für die Einbindung der verschiedenen Chart-Typen (unter Verwendung des Observer Pattern). Schließlich ist angestrebt, dass das Projekt im Sinne des Model-View-Controller (bzw. Controller erweitert um Logik) zusätzlich zu den Patterns Strategy und Observer klar strukturiert ist.

Die Namensgebung der jeweiligen Source-Files und Packages wurde dementsprechend gewählt.
Abbildung der Datei- und Packagestruktur (vorläufig):



In Bezug auf die Namenskonventionen im Rahmen der Coding Practices ist es wichtig, dass aus dem Name der Klassen, Methoden, Variablen und Packages klar ersichtlich ist, was die jeweilige Komponente des Projekts macht. Daher sollte ein Name mit einer für die Programmierer verständlichen Bedeutung gewählt werden. Außerdem sollten die gewählten Namen aussprechbar, suchbar und nicht extrem vage sein. Klassen- und Objektname sollten Nomen oder Nominalphrasen sein, Methoden sollten ein Verb oder eine Verbphrase enthalten und Getter sowie Setter sollten entsprechend den Konventionen (z.B. getFile, setFile) mit get/set plus jeweilige Variablen-/Objekt-Bezeichnung benannt werden. Untenstehend werden einige Programmteile aus dem Projekt auf diese Kriterien hin analysiert und diskutiert.

Code-Beispiel 1:

In diesem Code-Beispiel wurde für die Klasse der Name CSVManager gewählt, da diese Klasse das gesamte Handling für CSV-Dateien macht. Als Instanzvariablen wurden csvFilePath, also der Dateipfad des CSV-Files als

String, der Titel (bzw. die Spaltenüberschriften) als String-Array und data, also der Daten-Teil gewählt. Die Instanzvariablen haben jeweils eine leicht erkennbare Bedeutung, sind leicht aussprechbar und suchbar. Für die Methode readCSV ist die Bedeutung ebenfalls leicht nachvollziehbar und sie entspricht dem für Methoden empfohlenen Aufbau, da sie ein Verb enthält, das das Nomen CSV näher definiert. Mittels dieser Methode wird eine CSV-Datei anhand ihres Dateipfades (csvFilePath-String) eingelesen.

```
public class CSVManager {
    private String csvFilePath;
    private String[] title;
    private Object[][] data;

    /**
     * Dem Konstruktor CSVManager wird der Parameter csvFilePath (als String)
     * csvFilePath gesetzt.
     * @param csvFilePath - Pfad der CSV-Datei
     */
    public CSVManager(String csvFilePath){
        this.csvFilePath = csvFilePath;
    }

    /**
     * Die Methode readCSV liest eine CSV-Datei mit Hilfe ihres Dateipfades (s
     * erstellt die Tabelle.
     */
    public void readCSV(){

        BufferedReader bufferedReader = null;
        try {
```

Code-Beispiel 2:

In diesem Beispiel wird die Methode changeDate auf die oben erwähnten Kriterien hin analysiert. Die Namensgebung dieser Methode erfolgte im Einklang mit den Empfehlungen für Methode – sie enthält ein Verb, das das Nomen näher definiert. Außerdem ist der Sinn von changeData leicht erkennbar. Die Methode verändert ein bestimmtes Datenelement, das anhand der Integers row und col, also Zeilen- und Spaltenposition aufgefunden werden kann. Außerdem ist der Methodenname leicht aussprechbar und auffindbar.

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

Code-Beispiel 3:

Für die untenstehende Klasse ArithmeticOperationsAddition ist anhand des Namens leicht die zugehörige Bedeutung erkennbar. Die Klasse führt die arithmetische Operation der Addition durch. Dieser Klassenname wurde den Empfehlungen für die Namensgebung entsprechend gewählt, da ein Nomen verwendet wurde. Der Name ist außerdem leicht aussprechbar und suchbar. In der Klasse wird die Methode execute des Interface IOperations überschrieben. Die Methode lässt anhand ihrer Bedeutung leicht erkennen, dass die Ausführung der Addition in diesem Bereich stattfindet. Auch den weiteren Empfehlung der leichten Aussprechbarkeit, Lesbarkeit und Suchbarkeit entspricht die Methode. Die an diese Methode übergebenen Parameter eingabe und table haben ebenfalls eine klare Bedeutung. Sie bezeichnen die vom Benutzer gemachte Eingabe und die Tabelle, in der die Operation ausgeführt wird.

```
public class ArithmeticOperationsAddition implements IOperations{

    /**
     * Die Methode execute nimmt die Parameter eingabe und jtable entgegen und berechnet die
     * arithmetische Operation der Addition.
     * @param eingabe
     *      Eingabe-String der arithmetischen Operation der Addition (z.B. =3+1)
     * @param table
     *      JTable-Objekt, in dem die Eingabe gemacht wurde
     */
    @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(',', '.'));
        table.getModel().setValueAt(((first+second)+"").replace('.', ','), table.getSelectedRow(), table.getSelectedCol
    }

}
```

Code-Beispiel 4:

Für den untenstehenden Button wurde ein Name gewählt, der leicht verständlich, aussprechbar und suchbar ist. Dem Name nach wird beim Klick auf den Button die CSV-Datei gespeichert (i.e. die CSV-Datei wird mittels csvManager-Instanz geschrieben).

```
buttonSaveCSVFile.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        csvManager.writeCSV(';');
        JOptionPane.showMessageDialog(frame, "The CSV-File changed and saved!!");
    }
});
```

Code-Beispiel 4:

In for-Schleifen, wie im untenstehenden Code-Beispiel, ist es erlaubt, dass Namen wie i als Laufvariable verwendet werden, da damit nur die Schleife durchlaufen wird – dies wäre bei anderen Code-Stücken allerdings nicht empfohlen laut Coding Practices. Innerhalb der Schleife ist die Bedeutung der restlichen Variablen leicht erkennbar und gut lesbar. Die Tabelle wird anhand der Zeilenanzahl durchlaufen und es werden Werte gesetzt.

```
for (int i = 0; i < table.getRowCount(); i++) {
    table.setValueAt((i + 1) + "", i, 0);
}
```

Code-Beispiel 5:

In for-Schleifen, wie im ebenfalls im untenstehenden Code-Beispiel, ist es erlaubt, dass Namen wie i verwendet werden. Die übrigen Variablen in diesem Code-Abschnitt wie z.B. count, summe, FromCellRowPosition, TO_CELLRowPosition haben eine klare Bedeutung, sind gut leserlich und aussprechbar und können leicht gesucht werden.

```
int count = 0;
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(',', '.'));
        count++;
    }

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

Das Kommentieren dient der internen Dokumentation und Orientierung für jene, die den Code lesen (d.h. vor allem für technisches Personal). Daher sollte der Code übersichtlich gehalten werden und die Kommentare sollten erklären, was der Code macht und wie er funktioniert. Der Code sollte in einem übersichtlichen Umfang sein, d.h. nicht zu lang und nicht zu kurz, und sollte die für das Verständnis wichtigen Teile umreißen. Offensichtliche Kommentare wie z.B. für Laufvariablen sind nicht notwendig. Diese Empfehlungen bezüglich Coding Practices wurde auch im vorliegenden Projekt so gut wie möglich umgesetzt. Als Kommentarformen wurden vor allem javadoc-Kommentare (der Form /** javadoc-comment */) geschrieben, damit ein javadoc vom gesamten Projekt erstellt werden kann und bei der Abgabe mitgeliefert werden kann. Zusätzlich wurden an einigen Stellen zur Verdeutlichung bestimmter Code-Zeilen noch normale Kommentare der Form // Comment oder der Form /* Comment */ geschrieben.
Ein weiterer Punkt, der in Bezug auf Coding Practices beachtet werden sollte, sind die logische Organisation und die Formatierung des Codes. Die Formatierung und Einrückung sollte konsistent sein über alle Files hinweg, da der Code dann leichter und schneller für alle Beteiligten nachvollziehbar ist.
Dazu werden nun einige Beispiel aus dem Code des Projekts diskutiert.

Code-Beispiel 6:

Über der Methode readCSV der Klasse CSVManager wurde ein javadoc-Kommentar verfasst, der erklärt, was die Methode genau macht. Da kein @param oder @return im javadoc-Kommentar angeführt wurde, ist darauf zu schließen, dass es keine Parameter gibt, und dass der return-Typ void ist.

```
/**
 * Die Methode readCSV liest eine CSV-Datei mit Hilfe ihres Dateipfades (siehe Instanzvariable csvFilePath) ein und
 * erstellt die Tabelle.
 */
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;
```

Das Code-Stück der Methode readCSV ist außerdem logisch passend organisert. Die Einrückungen wurden der Abfolge entsprechend gewählt. Zunächst wird der BufferedReader angelegt und der try-Block eröffnet. Die Inhalte im try-Block sind dann um eine Einrückung nach rechts verschoben, damit erkennbar ist, dass sie innerhalb des try-Blocks ausgeführt werden. Innerhalb der im unteren Bereich der Abbildung gezeigten while-Schleife sieht man, dass die Inhalte der while-Schleife wiederum organisatorisch um eine Einrückung mehr nach rechts verschoben sind, da sie innerhalb vom while-Abschnitt ausgeführt werden.

Code-Beispiel 7:

Der javadoc-Kommentar vor der Methode writeCSV beschreibt anschaulich, was die Methode macht. Hinsichtlich der Formatierung und logischen Organisation wurden white spaces genutzt, um dem Code eine leicht verständlich Ordnung und Struktur zu geben. Der Methodenteil, der hier abgebildet ist, zeigt die logische Aufteilung in den try-Block, in dem eine weitere Einrückung nach rechts gemacht wurde. Innerhalb des try-Blocks gibt es zwei for-Schleifen, in denen jeweils der Code wieder eingerück wird nach rechts. Danach folgt der catch-Block, der sich auf derselben Einrückungsebene wie der try-Block befindet, und wiederum in sich für den Code eine Einrückung weiter nach rechts hat.

```
/**
 * Die Methode writeCSV schreibt bzw. speichert eine CSV-Datei an dem angegebenen CSV-Dateipfad.
 * Alle Bestandteile der CSV-Datei werden hier gemaess der CSV-Syntax in das File geschrieben.
 */
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();
    }
```

Code-Beispiel 8:

Der javadoc-Kommentar vor der Klasse ArithmeticOperationsAddition beschreibt, was die Klasse macht, und dass sie das Interface IOperations implementiert. Alle Inhalte innerhalb der Klasse werden dann weiter nach rechts eingerückt. Direkt über der Methode execute, die die Methode aus dem Inteface IOperations überschreibt, wird die Funktionsweise der Methode kurz erklärt. Außerdem werden die beiden Parameter, die der Methode übergeben werden, näher erklärt. Innerhalb der Methode ist wieder ersichtlich, dass eine weitere Einrückung stattgefunden hat.

```java
/**
 * Die Klasse ArithmeticOperationsAddition implementiert das Interface IOperations und
 * bietet die Implementierung der Methode execute fuer die Addition.
 * @author Cordula Eggerth
 *
 */
public class ArithmeticOperationsAddition implements IOperations{

    /**
     * Die Methode execute nimmt die Parameter eingabe und jtable entgegen und berechnet die
     * arithmetische Operation der Addition.
     * @param eingabe
     *      Eingabe-String der arithmetischen Operation der Addition (z.B. =3+1)
     * @param table
     *      JTable-Objekt, in dem die Eingabe gemacht wurde
     */
    @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(',', '.'));
        table.getModel().setValueAt(((first+second)+"").replace('.', ','), table.getSelectedRow(), table.getSelectedCc
    }

}
```

Code-Beispiel 9:

Der javadoc-Kommentar vor der Klasse ReferenceCell erklärt ebenfalls, was die Klasse macht, und dass sie das Interface IOperations implementiert. Innerhalb der Klasse findet wieder eine weitere Einrückung statt, um zu zeigen, dass der Code dort eine Stufe tiefer liegt. Der javadoc-Kommentar direkt über der Methode execute beschreibt diese und bietet auch nähere Informationen über die Parameter eingabe und table, die an die Methode beim Aufruf übergeben werden. Innerhalb der execute-Methode wird wieder eine weitere Einrückung gemacht, um die logische Organisation des Codes im Rahmen der Coding Practices übersichtlich und gut lesbar zu gestalten. Die white spaces und Klammerungen in diesem Beispiel werden außerdem dazu genützt, die

Formatierung so leserlich wie möglich zu machen und für den Programmierer die Abfolge leichter verständlich zu machen.

```java
/**
 * Die Klasse ReferenceCell implementiert das Interface IOperations und stellt die Implementierur
 * fuer die Methode execute zur Verfuegung. Der Wert einer Reference Cell wird im Feld der Eingab
 * gesetzt.
 */
public class ReferenceCell implements IOperations{

    /**
     * Es wird ueber die Methode execute ermittelt, was der Wert der referenzierten Zelle ist.
     * Dann wird der Wert der Zelle, in der die Eingabe gemacht wurde, auf den Wert, der sich
     * in der referenzierten Zelle  befindet, gesetzt.
     * @param eingabe
     *      String der Eingabe in die Zelle, wo der Referenzwert gesetzt werden soll (z.B. =B3)
     * @param table
     *      JTable-Objekt der Tabelle, in der die Eingabe gemacht wird
     */
    @Override
    public void execute(String eingabe, JTable table) {
        MySingletonPattern mySingletonPattern = MySingletonPattern.getInstance();
        eingabe = eingabe.replace("=","");
        int splitPos = mySingletonPattern.getSplitPosition(eingabe);
        int choseRow = mySingletonPattern.getStringPosition(eingabe.substring(0, splitPos))+1;
        int choseCol = Integer.parseInt(eingabe.substring(splitPos, eingabe.length()))-1;
        String value = (String) table.getModel().getValueAt(choseCol, choseRow);
        table.getModel().setValueAt(value, table.getSelectedRow(), table.getSelectedColumn());

    }

}
```

Code-Beispiel 10:

Die Namensgebung der Methode isCellEditable folgt den Coding Practices für Methoden vom Rückgabe-Typ boolean, denn diese sollten mit „is“ beginnen. Außerdem sollte eine Methode so kurz und verständlich wie möglich geschrieben sein, was hier auch berücksichtigt wird. Der javadoc-Kommentar vor der Methode isCellEditable beschreibt, was in der Methode gemacht wird – es wird hier also ermittelt, ob die Zelle editiert werden darf oder nicht. Die Parameter rowIndex (Zeilenindex) und columnIndex (Spaltenindex) habe eine klare Bedeutung, und sind leicht lesbar und suchbar im Sinne der in den Coding Practices vorgeschlagenen Namensgebung. Die Parameter werden im javadoc-Kommentar jeweils auch im Bereich @param näher erklärt.

```java
/**
 * Die Methode isCellEditable gibt an, ob die betrachtete Zelle editierbar ist oder nicht.
 * @param rowIndex
 *       Zeilenindex der betrachteten Zelle
 * @param columnIdex
 *       Spaltenindex der betrachteten Zelle
 */
public boolean isCellEditable(int rowIndex, int columnIndex) {
    if(columnIndex!=0)
        return true;
    return false;
```

Code-Beispiel 11:

Der javadoc-Kommentar vor der Methode findCells erklärt, was die Methode macht und beschreibt ihre Parameter in den Bereich @param näher. Die Einrückung erfolgt innerhalb der Methode um einen Schritt nach rechts. Innerhalb der Methode ist auch erkennbar, dass z.B. die Berechnung von int splitPosFROM_CELL über die Methode getSplitPosition(FROM_CELL) in eine eigene Methode ausgelagert wird, die die Split-Position für die übergebene Zelle berechnet, damit der Code innerhalb der Methode findCells logisch übersichtlich und zusammengehörend ist.

```java
/**
 * Die Methode findCeils berechnet die Zeilen- und Spaltenposition der FromCell und ToCell jeweils.
 * @param eingabe
 *      Eingabe des Benutzers als String
 * @param callElement
 *
 */
public void findCeils(String eingabe, String callElement){
    int index = eingabe.indexOf(':');
    String FROM_CELL =  eingabe.substring(callElement.length(), index);
    String TO_CELL = eingabe.substring(index+1, eingabe.lastIndexOf(')'));
    int splitPosFROM_CELL = getSplitPosition(FROM_CELL);
```