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

TASK 2 – DESIGN PATTERNS

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

BERÜCKSICHTIGUNG DER KEY DESIGN PRINCIPLES:

Anmerkung: Die Dokumentation zu Verwendung der „Key Design Principles“ erfolgt an dieser Stelle gemäß der Antwort der Tutorin im Diskussionsforum auf das Posting und die Frage der Platzierung dieser Dokumentation (Frage gestellt von Cordula Eggerth).

Hinsichtlich der Key Design Principles wird zunächst die Modularisierung des Projekts angestrebt, indem mehrere Packages, wie z.B. logik, logik.strategyPattern oder view erstellt werden. In diesen Packages wurden die entsprechenden zusammengehörigen Klassen und Interfaces platziert. Außerdem wurde innerhalb von den Klassen darauf Wert gelegt, dass die Methoden die Key Design Principles des Information Hiding und der Kapselung erfüllen. Ersteres bezieht sich u.a. auf den Zugriff auf die Instanzvariablen über Get- und Set-Methoden, wie untenstehend dargestellt anhand des Beispiels der Get-Methoden für zwei Instanzvariablen aus der Klasse CSVManager.java:

```
/**
 * Get-Methode fuer die Instanzvariable title.
 * @return title (als String)
 */
public String[] getTitel(){
    //title = new String[title.length];
    //getTitelList(5);
    return title;
}

/**
 * Get-Methode fuer die Instanzvariable data.
 * @return data (als Object[][])
 */
public Object[][] getData(){
    return data;
}
```

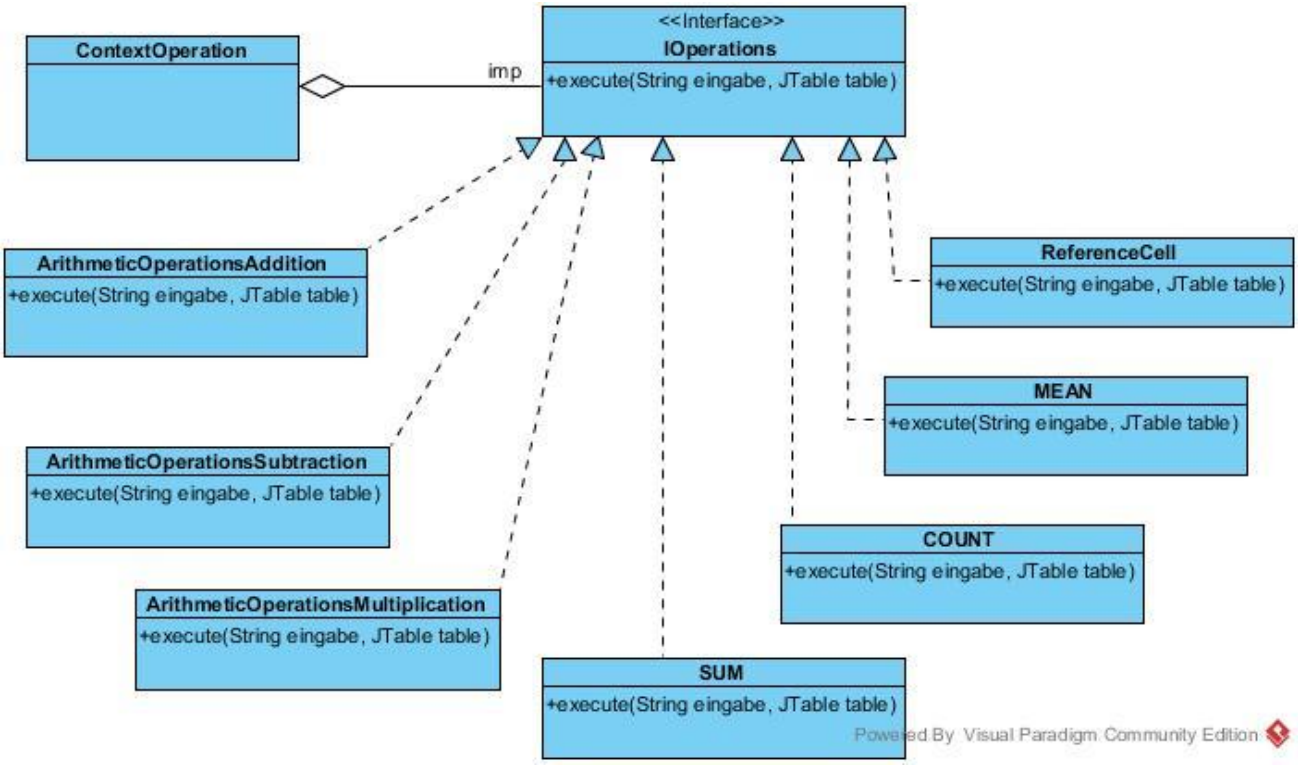
Gemäß dem Prinzip des Information Hiding soll jedes Modul seine innere Zusammensetzung vor direkten Zugriffe von außen schützen, und im Sinne der Encapsulation (bzw. Kapselung) soll es entsprechende Schnittstellen besitzen, damit es mit der Außenwelt interagieren kann. Es sollte also möglichst wenig von den inneren Details preisgegeben werden. Wichtig ist also, dass die interne Bindung (i.e. Kohäsion) zwischen Modulen hoch ist, und dass die externe Bindung (i.e. Coupling) zwischen Modulen niedrig ist, und dass die Kommunikation nur über gut definierte Schnittstellen erfolgt. Dies wird im Projekt umgesetzt, indem die Klassen intern auf die Instanzvariablen zugreifen können und die Instanzvariablen von außen nur über Get- und Set-Methoden zugänglich sind. Zudem ist die Schnittstelle für die Verwendung der Klassen.
Das Key Design Principles der Separation of Concerns wurde insofern beachtet, dass das Projekt hierarchisch unterteilt ist in Projekt, dann darunterliegend die Ebene der Packages und wieder darunterliegend die Ebene der Interfaces und Klassen. Innerhalb von den Klassen hat jede Methode ihren klar abgegrenzten Bereich, für den sie zuständig ist.

ANWENDUNG VON DESIGN PATTERNS:

Strategy Pattern:

Im Projekt wurde das Strategy Pattern (mit Hilfes des Interface IOperations) verwendet. Das Strategy Pattern dient dazu, dass man verschiedene Algorithmen auswechselbar machen kann, und diese über ein einheitliches Interface ansprechen kann. Das Strategy-Interface der allgemeinen Definition des Patterns (siehe Folien Kapitel „Design Patterns 1“ von Prof. Zdun) wurde im Projekt Interface IOperations gennant. Es stellt die Schnittstelle für die Algorithmen dar, die je nach passender Situation ausgewechselt werden können und den Client, der im Fall des Projekts die ContextOperation Klasse ist, unabhängig vom jeweiligen Algorithmus werden lässt.
Die untenstehende Abbildung zeigt das UML-Diagramm der Verwendung des Strategy Patterns im Projekt. Der „Client“ ist die ContextOperation. Sie verwendet eine bzw. mehrere verschiedene Arten von Algorithmen für arithmetische Operationen, Reference Cells oder auch Funktionen (Sum, Count, Mean). Die jeweiligen ArithmeticOperations-Klassen, ReferenceCell-Klasse und

Funktionen-Klassen implementieren das Interface IOperations, und können darüber nahtlos im Client verwendet werden, je nach.



Verwendetes Interface IOperations (siehe untenstehendes Code-Stück):

Dieses Interface beinhaltet die Methode execute, die von allen Berechnungsalgorithmen überschrieben wird.

```
3  import javax.swing.JTable;
4
5  /**
6   * Das Interface IOperations dient dazu, die jeweiligen Operationen auf den Zellen der Tabelle, abzukapseln und
7   * je nach Bedarf, eine der Operationen auszufuehren.
8   */
9  public interface IOperations {
0      /**
1       * Die Methode execute nimmt die Parameter eingabe in der betroffenen Zelle und den JTable table entgegen und fuehrt dann
2       * die jeweils gewuenschte Operation aus.
3       * @param eingabe
4       *       Benutzer-Eingabe in die Zelle (als String)
5       * @param table
6       *       JTable, in dem die Eingabe gemacht wird
7       */
8      public void execute(String eingabe, JTable table);
9  }
```

Verwendung der ContextOperation in der Klasse Main.java (siehe untenstehendes Code-Stück):

In der Funktion fireTableCellUpdated wird die Eingabe entgegengenommen und es wird geprüft, um welche Eingabe es sich handelt. Gibt es am Anfang der Eingabe ein „=“ Zeichen, dann werden neue Operationen angelegt und die Operation wird durchgeführt. Wenn kein „=“ Zeichen am Anfang der Eingabe steht, wird die Methode changeData des CSVManager.java aufgerufen und die Daten werden nur geändert, aber es wird keine Berechung (Operation) gemacht.

im Main.java:

```
@Override
public void fireTableCellUpdated(int row, int column) {
    String eingabe = this.getValueAt(row, column).toString();
    if (eingabe.startsWith("="))
    {
        ContextOperation imp = null; // imp steht fuer implementation
        if (eingabe.toLowerCase().startsWith("=sum("))
        {
            imp = new ContextOperation(new SUM());
        }else if (eingabe.toLowerCase().startsWith("=mean(")){
            imp = new ContextOperation(new MEAN());
        }else if (eingabe.toLowerCase().startsWith("=count(")){
            imp = new ContextOperation(new COUNT());
        }else if(eingabe.contains("+")){
            imp = new ContextOperation(new ArithmeticOperationsAddition());
        }else if(eingabe.contains("-")){
            imp = new ContextOperation(new ArithmeticOperationsSubtraction());
        }else if(eingabe.contains("*")){
            imp = new ContextOperation(new ArithmeticOperationsMultiplication());
        }else if(eingabe.contains("/")){
            imp = new ContextOperation(new ArithmeticOperationsDivision());
        }else{
            imp = new ContextOperation(new ReferenceCell());
        }
        imp.execute(eingabe, table);
    }
    else
        csvManager.changeData(row, column, this.getValueAt(row, column));
    super.fireTableCellUpdated(row, column);
}
```

Verwendung der ArithmeticOperationsAddition.java Klasse als Beispiel für eine ConcreteStrategy Klasse (siehe untenstehendes Code-Stück):

Als Beispiel für alle mit Berechnungen (Operations) betrauten Klassen, wird hier ArithmeticOperationsAddition vorgestellt. Diese Klasse setzt den Teil ConcreteStratgy Klasse des Strategy Patterns um.

in ArithmeticOperationsAddition.java:

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

    /**
     * Die Methode execute nimmt die Parameter eingabe und jtable entgegen und berechnet d
     * 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) {
        MySingletonPattern mySingletonPattern = MySingletonPattern.getInstance();
        mySingletonPattern.checkArithmeticComponents(eingabe, "+", table);
    }

}
```

In der oben dargestellten Klasse ArithmeticOperationsAddition wird das Interface IOperations implementiert und davon die Methode execute überschrieben. Die Methode execute macht schließlich die Berechnung der Addition.

Die Verwendung der Klasse ContextOperation.java (siehe untenstehendes Code-Stück):

In ContextOperaion gibt es die Instanzvariable operations vom Typ IOperations. Diese ruft allgemein die execute-Methode auf.

in ContextOperation.java:

```
public class ContextOperation {
    /**
     * Instanzvariable: operations (Datentyp IOperations)
     */
    private IOperations operations;

    /**
     * Konstruktor: Instanzvariable operations wird hier gesetzt
     * @param operations
     *      durchzufuehrende operation (als IOperations)
     */
    public ContextOperation(IOperations operations){
        this.operations = operations;
    }

    /**
     * Die Methode execute fuehrt eine Operation fuer die eingabe, die
     * der Benutze in der Tabelle gemacht hat, aus.
     * @param eingabe
     *      Eingabe des Benutzers als String
     * @param table
     *      Tabelle table, in der die Operation ausgefuehrt wird (als JTable-Objekt)
     */
    public void execute(String eingabe, JTable table ){
        operations.execute(eingabe, table);
    }
```

Vorteile des Strategy Patterns:

Im Strategy Pattern können Familien von verwandten Algorithmen definiert werden, die dann je nach Kontext in der jeweils passenden Ausführung benützt werden. Dies findet z.B. im vorliegenden Projekt über das Interface IOperations statt, das von verschiedenen Ausführungen der Berechnungsoperations-algorithmen implementiert wird.

Die Vewendung des Interface IOperations ist hier eine gute Alternative zur Vererbung, weil trotzdem verschiedene Arten von Algorithmen und Verhalten der Methoden durchgeführt werden können.

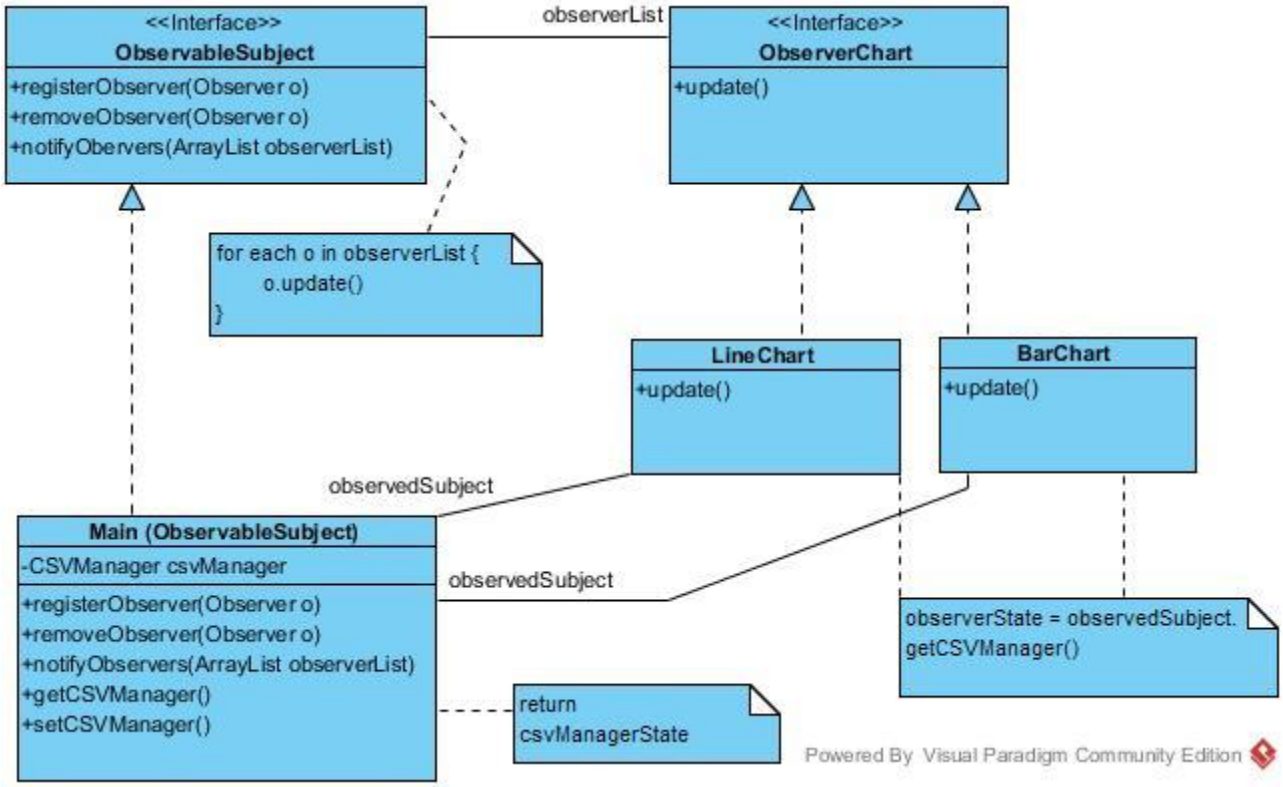
Durch die Verwendung des Strategy Pattern können auch verschiedene Algorithmen derselben Verhaltensweise angeschlossen werden.

Ein Nachteil des Strategy Patterns kann sein, dass der Client (als die Main-Klasse) die verschiedenen Berechnungsarten kennen muss, damit diese verwendet werden können. Allerdings war dies kein allzu großes Problem im Projekt. Außerdem steigt auch die Anzahl der Objekte durch die Verwendung des Strategy Patterns, und dadurch die Komplexität.

Observer Pattern:

Im Projekt wurde das Observer Pattern durchdacht. Das Observer Pattern dient dazu, dass man die Präsentationslogik (also die Views bzw. GUI) von den Application-Daten (also dem Modell) trennen kann. Im vorliegende Projekt wurde dies anhand des Beispiels der LineChart und BarChart Klassen, die als Views agieren, durchdacht und ausprobiert. Die beiden Charts kennen einander aber nicht, und können trotzdem durch die Implementierung des Interface Observer auf dasselbe ObservableSubject gesetzt werden. Wenn der Benutzer nun Änderurngen in einem Teil vornimmt, werden alle Observer benachrichtigt – also in unserem Fall alle Charts, dass die Informationen sich geändert haben, und dass die Charts nun upgedatet werden müssen. Wenn also das ObservableSubject sich ändert, können gemäß Observer-Pattern alle registrierten Oberver benachrichtigt werden über notifyObservers benachrichtigt und automatisch upgedatet werden. Die Verwendung des Observer Patterns im Projekt wird im untenstehenden Klassendiagramm dargestellt.
Man kann mittels Observer Pattern zusätzlich zum LineChart und BarChart unabhängig von der existierenden Observeranzahl noch weitere hinzufügen, ohne dass diese die bestehenden Observer stören. Die Kopplung zwischen den Observern und dem ObservableSubject ist minimal und „abstrakt“, da keine enge Verbindung zwischen den beiden besteht. Wenn die Methode notify des ObservableSubjects aufgerufen wird, wird nicht nur ein bestimmter Observer (bzw. View) informiert, sondern es werden alle Observer benachrichtigt. Problematisch kann es allerdings sein, wenn unerwartet Updates stattfinden, und weil die Observer keine detaillierte Information darüber bekommen, was tatsächlich geändert wurde.
Im untenstehenden Diagramm wird im Projekt das Interface ObservableSubject dargestellt. Dieses Interface wird von Main (ObservableSubject) implementiert, das die Methoden des Interface überschreibt und den zu beobachtenden Zustand des csvManagers enthält. Main enthält auch eine ArrayList<Observer>,

die alle registrierten Observer enthält. Das Interface ObserverChart ist das Interface für die Observerklassen (LineChart und BarChart). Die beiden implementieren das Interface ObserverChart und werden upgedatet, sobald Änderungen im csvManager Objekt in de Main-Klasse stattfinden.



Quelle:

Prof. Uwe Zdun. Foliensatz Design Patterns 1.
Prof. Uwe Zdun. Foliensatz Key Design Principles.