	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Dra. Rocío Alejandra Aldeco Pérez

Asignatura: Programación orientada a objetos -1323

Grupo: 6

No de Práctica(s): 13

Integrante(s): Mendoza Hernández Carlos Emiliano

No. de Equipo de cómputo empleado:

No. de Lista o Brigada:

Semestre: 2023-1

Fecha de entrega: 2 de diciembre del 2022

Observaciones:

CALIFICACIÓN: _____

Práctica 13.

Patrones de diseño.

OBJETIVOS

- Implementar una aplicación en un lenguaje orientado a objetos utilizando algún patrón de diseño.

ACTIVIDADES

- Conocer diferentes patrones de diseño.
- Implementar una aplicación utilizando algún patrón de diseño.

INTRODUCCIÓN

En la ingeniería de software, un patrón de diseño es una solución repetible y general para problemas de ocurrencia cotidianos en el diseño de software. Es importante aclarar que un patrón de diseño no es un diseño de software terminado y listo para codificarse, más bien es una descripción o modelo (template) de cómo resolver un problema que puede utilizarse en diferentes situaciones.

Por lo tanto, los patrones de diseño permiten agilizar el proceso de desarrollo de solución debido a que proveen un paradigma desarrollado y probado.

Un diseño de software efectivo debe considerar detalles que tal vez no sean visibles hasta que se implemente la solución, es decir, debe anticiparse a los problemas y tratar de cubrir todos los resquicios. La reutilización de patrones de diseño ayuda a prevenir los detalles sutiles que provocarían problemas más grandes, además de ayudar a la legibilidad de código para programadores o analistas que estén familiarizados con patrones.

INSTRUCCIONES

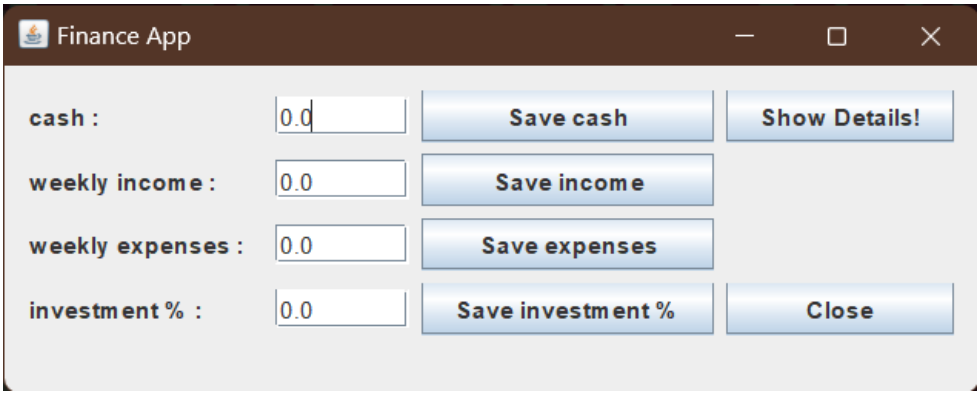
Realiza las siguientes actividades después de leer y revisar en clase la *Práctica de Estudio 13: Patrones de diseño.*

Tomando como referencia la aplicación que está anexa a esta práctica (*java-swing-gui-master.zip*) realiza los siguientes pasos:

1. Descarga el *.zip* e importa el proyecto en Eclipse, Netbeans o cualquier IDE que utilices. Si lo deseas puedes continuar el proceso en consola.
2. Descarga la última versión del controlador de *jdbc* del espacio de práctica llamado *sqlite-jdbc-3.30.1.jar*. Copia este archivo *.jar* en la ruta */java-swing-gui-master/bin/.* Si estás usando un IDE debes importar el archivo.
3. Abre el archivo *RunFinanceProgram.java* ahí encontrarás una variable llamada *DATABASE_URL*, cambia el *String* que tiene asignado por la ruta de tu computadora donde se encuentra el archivo *test.db*. Este archivo está dentro de la carpeta del proyecto (*…/java-swing-gui-master/src/test.db*), pero debes incluir la ruta completa.
4. Ahora deberás compilar el proyecto completo.
5. Ahora ejecuta el programa.
6. Contesta las preguntas dadas.

DESARROLLO

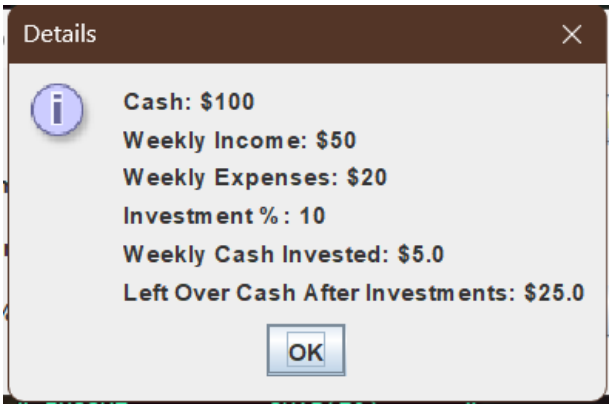
Ejecución del programa



1. **Describe la funcionalidad de la aplicación (esto es, ¿qué hace?)**

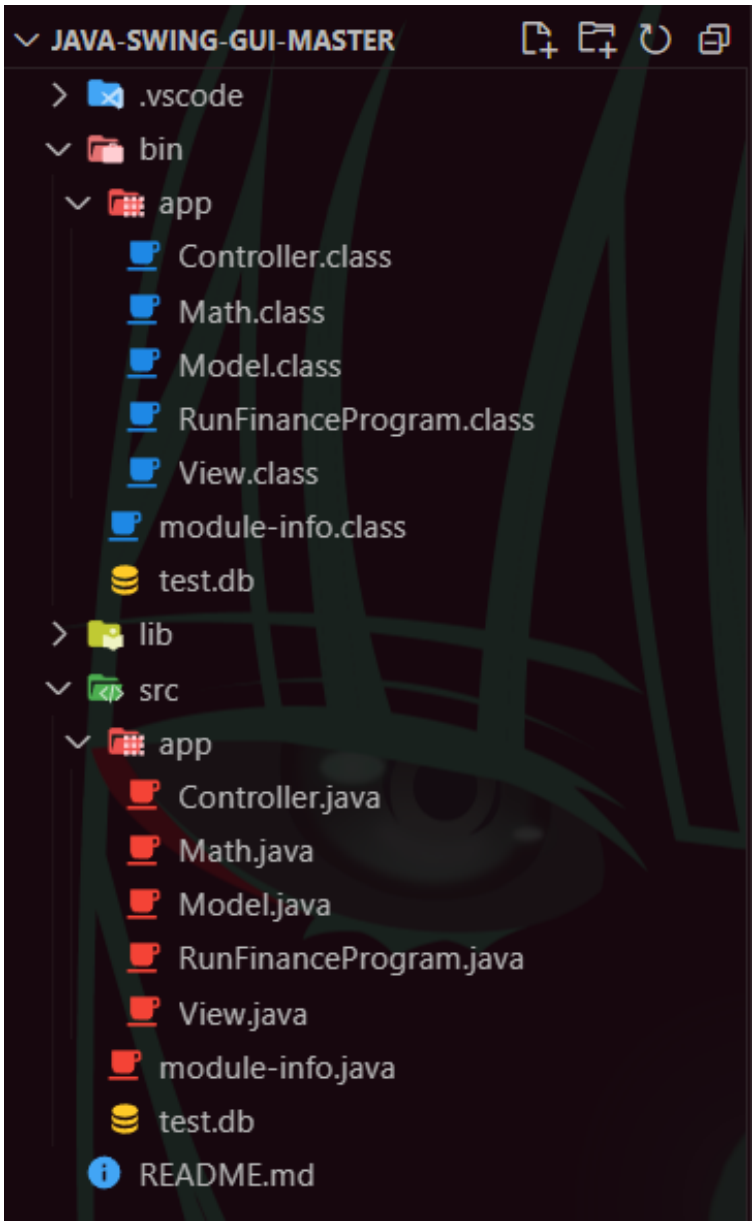
El programa tiene las siguientes opciones:

- **Save cash:** Permite ingresar una cantidad de dinero que se supone tenemos en efectivo.
- **Save income:** Permite ingresar una cantidad de dinero que se supone corresponde a nuestros ingresos semanales.
- **Save expenses:** Permite ingresar una cantidad de dinero que se supone corresponde a nuestros egresos semanales.
- **Save investment %:** Permite ingresar un porcentaje que se supone ahorraremos de nuestros ingresos semanales.
- **Show details:** Nos da un informe de nuestras finanzas. Especifica cuanto dinero tenemos en efectivo (*cash*), nuestros ingresos semanales (*weekly income*), nuestros gastos semanales (*weekly expenses*), el porcentaje de ahorro (*investment %*), la cantidad semanal destinada a ahorros (*weekly cash invested*) y la cantidad sobrante después de quitar los gastos y el ahorro semanal (*left over cash after investments*).



- **Close:** Termina la ejecución del programa, sin embargo, al volver a ejecutarlo tenemos el último estado en el que lo dejamos ya que guarda y lee la información de una base de datos.

2. **Revisa la estructura de archivos de la aplicación**
   a. **¿Dónde se encuentran los archivos *.java*?**
      En la carpeta del proyecto, se encuentran dentro de la carpeta *src*.
   b. **¿Dónde se encuentran los archivos *.class*?**
      En la carpeta del proyecto, se encuentran dentro de la carpeta *bin*.
   c. **¿Qué carpeta se usa para mantener la estructura del paquete?**
      La estructura del paquete se contiene en la carpeta *app*.



3. **Ve a la carpeta donde se encuentran todos los archivos *.java* y abrelos.**
   d. **¿En qué clase se encuentra lo relacionado con el *GUI* de la aplicación? (En este caso *Swing*)**
   En la clase *View*

View.java

src > app > View.java > ...

```
1   package app;
2
3   import java.awt.BorderLayout;
4   import javax.swing.GroupLayout;
5   import javax.swing.JButton;
6   import javax.swing.JFrame;
7   import javax.swing.JLabel;
8   import javax.swing.JTextField;
9   import javax.swing.SwingConstants;
10
```

e. ¿En qué clase se encuentra lo relacionado con el manejo de la base de datos? (En este caso *SQLite*)

En la clase *Model*

Model.java

src > app > Model.java > {} app

```
1   package app;
2
3   import java.sql.*;
4
```

f. ¿Qué clase conecta la funcionalidad de las dos clases anteriores?

La clase *Controller*

Controller.java

src > app > Controller.java > {} app

```
1   package app;
2
3   import javax.swing.JOptionPane;
4
5   public class Controller {
6
7       private Model model;
8       private View view;
9
```

g. ¿Qué clase es la clase principal? ¿Qué método del resto de las clases es llamado primero? ¿Por qué?

La clase *RunFinanceProgram*. El primer método que se invoca (excluyendo el manejo de la base de datos) es *initController*. Se debe invocar este método porque el controlador es el encargado de hacer funcionar la vista y comunicarla bidireccionalmente con el modelo. Por medio del controlador es que los objetos de la vista se enteran de los cambios de los objetos del modelo y viceversa. En general, el controlador interpreta las acciones realizadas por el usuario en los objetos de la vista y comunica estas acciones hacia la capa del modelo.

4. ¿Qué patrones de diseño son usados en esta aplicación? Explica por qué y cómo son usados.

Se utiliza el patrón de diseño Modelo-Vista-Controlador (MVC). Porque define tres clases con tres roles diferentes (*Model*, *View* y *Controller*). Se comunican entre ellos de la siguiente manera:

En la clase *View* se define toda la parte referente a la GUI, como se puede observar:

```
public View(String title) {
    frame = new JFrame(title);
    frame.getContentPane().setLayout(new BorderLayout());
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(width: 500, height: 200);
    frame.setLocationRelativeTo(c: null);
    frame.setVisible(b: true);
    // Create UI elements
    cashLabel = new JLabel(text: "cash :");
    incomeLabel = new JLabel(text: "weekly income :");
    expensesLabel = new JLabel(text: "weekly expenses : ");
    investmentLabel = new JLabel(text: "investment % : ");
    cashTextfield = new JTextField();
    incomeTextfield = new JTextField();
    expensesTextField = new JTextField();
    investmentTextField = new JTextField();
    cashSaveButton = new JButton(text: "Save cash");
    incomeSaveButton = new JButton(text: "Save income");
    expensesSaveButton = new JButton(text: "Save expenses");
    investmentSaveButton = new JButton(text: "Save investment %");
    hello = new JButton(text: "Show Details!");
    bye = new JButton(text: "Close");
    // Add UI element to frame
    GroupLayout layout = new GroupLayout(frame.getContentPane());
    layout.setAutoCreateGaps(autoCreatePadding: true);
    layout.setAutoCreateContainerGaps(autoCreateContainerPadding: true);
    layout.setHorizontalGroup(layout.createSequentialGroup()
```

En la clase *Model* se lee y actualiza la base de datos de la aplicación, aquí se definen los atributos con valores numéricos necesarios para manipular las cantidades de dinero. También hace uso de la clase *Math* para apoyarse a realizar los cálculos. En general, se encarga de la parte lógica y matemática del programa.

```java
public class Model {

    private double cash;
    private double income;
    private double expenses;
    private double investmentPerc;
    private double cashInvested;
    private double leftOverCash;
    private Math math = new Math();

    public Model() {

    }

    public double getCash() {
        return cash;
    }

    public String getCashString() {
        Connection c = null;
        Statement stmt = null;
        String cashFromDB = "0.0";

        try {
            Class.forName(className: "org.sqlite.JDBC");
            c = DriverManager.getConnection("jdbc:sqlite:" + RunFinanceProgram.DATABASE_URL);
            c.setAutoCommit(autoCommit: false);
            System.out.println(x: "Opened database successfully");

            stmt = c.createStatement();
            ResultSet rs = stmt.executeQuery(sql: "SELECT * FROM FINANCES;");

            while (rs.next()) {
                int id = rs.getInt(columnLabel: "id");
                cashFromDB = rs.getString(columnLabel: "cash");

                System.out.println("ID = " + id);
                System.out.println("CASH = " + cashFromDB);

                System.out.println();
```

En la clase *Controller* se inicializa la vista y además está a la escucha de los eventos generados por el usuario para comunicarlos tanto a la base de datos (*Model*) como a la GUI (*View*). Como se puede observar, es el intermediario de la vista y el modelo.

```java
public class Controller {

    private Model model;
    private View view;

    public Controller(Model m, View v) {
        model = m;
        view = v;
        initView();
    }

    public void initView() {
        view.getCashTextfield().setText(model.getCashString());
        view.getIncomeTextfield().setText(model.getIncomeString());
        view.getExpensesTextField().setText(model.getExpensesString());
        view.getInvestmentTextField().setText(model.getInvestmentPercString());
    }

    public void initController() {
        view.getCashSaveButton().addActionListener(e -> saveCash());
        view.getIncomeSaveButton().addActionListener(e -> saveIncome());
        view.getExpensesSaveButton().addActionListener(e -> saveExpenses());
        view.getInvestmentSaveButton().addActionListener(e -> saveInvestmentPerc());
        view.getHello().addActionListener(e -> showDetails());
        view.getBye().addActionListener(e -> sayBye());
    }

    private void saveCash() {
        model.setCashFromString(view.getCashTextfield().getText());
        JOptionPane.showMessageDialog(parentComponent: null, "Cash saved : $" + model.getCashString(), title: "Info",
                JOptionPane.INFORMATION_MESSAGE);
    }

    private void saveIncome() {
        model.setIncomeFromString(view.getIncomeTextfield().getText());
        JOptionPane.showMessageDialog(parentComponent: null, "Income saved : $" + model.getIncomeString(), title: "Info",
                JOptionPane.INFORMATION_MESSAGE);
    }
```

Por ejemplo:

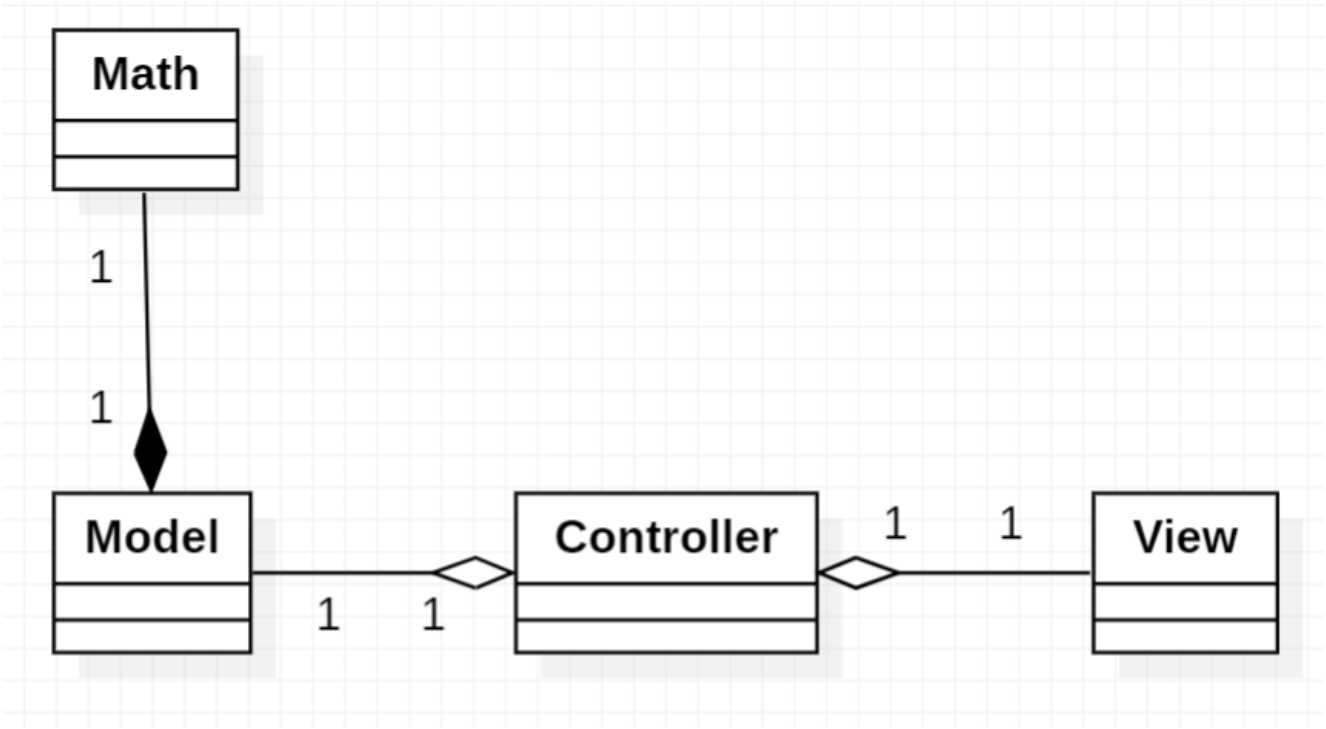
```java
view.getCashSaveButton().addActionListener(e -> saveCash());
```

En esta linea se observa que el botón *Save Cash* está a la espera de un evento: al detonarse el evento, el modelo hace su tarea correspondiente (guardar la cantidad de dinero en la base de datos). A su vez, el método *saveCash* relaciona a ambas clases (vista y controlador) para llevar a cabo la tarea.

```
private void saveCash() {
    model.setCashFromString(view.getCashTextfield().getText());
    JOptionPane.showMessageDialog(parentComponent: null, "Cash saved : $" + model.getCashString(), title: "Info",
            JOptionPane.INFORMATION_MESSAGE);
}
```

Es así como el modelo encapsula la información de la aplicación y define la lógica con la que se van a manipular los datos; la vista es el objeto que el usuario ve (GUI) y muestra la información del modelo, permitiendo editar la información; y el controlador comunica ambas capas.

5. **Realiza el diagrama de clases de esta aplicación.**



CONCLUSIONES

En esta práctica se revisó una aplicación que implementa el patrón de diseño Modelo-Vista-Controlador. En él se identificaron las funciones de las capas que componen a este patrón de diseño: mostrar la información y responder a las acciones del usuario (Vista), manejar la información y parte lógica del programa (Modelo) y comunicar la vista con el modelo (Controlador).

Con este patrón de diseño se desacopla el modelo de datos de la interfaz gráfica, estableciendo un protocolo de suscriptor/notificador entre ellos, de tal manera que se debe asegurar que la vista refleje el estado del modelo. Así, si la información del modelo cambia, el modelo debe notificar a la vista que depende de él. Una ventaja de esto es que la aplicación podría ser escalada, reutilizando el modelo para ser usado con diferentes vistas.

REFERENCIAS

- *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison Wesley Professional, 1994.

Yo, Carlos Emiliano Mendoza Hernández, hago mención que esta práctica fue de mi autoría.