Tecnológico Nacional De México

Instituto Tecnológico de Oaxaca

Ingeniería En Sistemas Computacionales

Diseño e Implementación de Software con Patrones.

7 am – 8 am

Unidad 3

“Patrón de Diseño Mediator”

Presenta:

Implementación de los Patrones en el Sistema Copy Max

Nombres	Numero de Control
Bautista Fabian Max	**C19160532**
Celis Delgado Jorge Eduardo	**21160599**
Flores Guzmán Alan Ismael	**20161193**
García Osorio Bolívar	**20161819**
Pérez Barrios Diego	**21160750**
Pérez Martínez Edith Esmeralda	**21160752**
Sixto Morales Ángel	**21160797**

Periodo Escolar:

Febrero – Julio

Grupo:

7SB

Maestro:

Espinoza Pérez Jacob

Oaxaca de Juárez, Oaxaca **Abril 2025**

INDICE

PATRÓN DE DISEÑO: MEDIATOR

El patrón Mediator es un patrón de diseño de comportamiento que permite reducir las dependencias caóticas entre objetos. Lo hace al introducir un objeto mediador que se encarga de controlar la comunicación entre otros objetos. De esta manera, los objetos ya no se comunican directamente entre sí, sino a través del mediador.

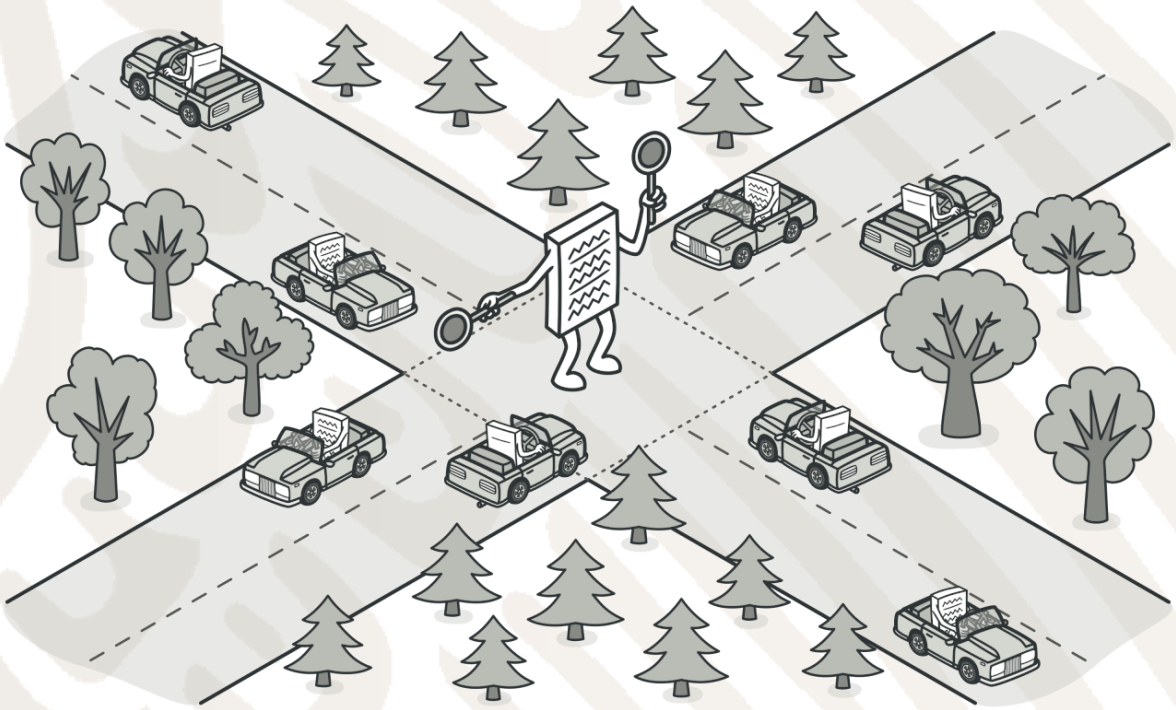

Figura 1. *Representación gráfica del patrón de diseño Mediator*. **Fuente:** https://refactoring.guru/es/design-patterns/mediator

Participantes del Patrón Memento

El patrón Mediator tiene los siguientes participantes clave:

1. **Mediator** (Mediador): Define la interfaz para la comunicación entre los objetos Colleague.
2. **ConcreteMediator** (Mediador Concreto): Implementa el Mediator y coordina la comunicación entre los Colleague. Conoce a todos los Colleague y decide cómo deben interactuar.

Colleague (Colega): Define la interfaz común para todos los colegas.

3. **ConcreteColleague** (Colega Concreto): Implementa la interfaz Colleague y se comunica con otros colegas a través del Mediator.

Módulo clientes

El módulo de clientes en el sistema CopyMax tiene como objetivo principal gestionar la información de los clientes en la base de datos.

La funcionalidad incluye:

- Visualización: Mostrar la lista de clientes en una tabla.
- Búsqueda: Filtrar clientes por número de celular.
- Nuevo: Agregar nuevos clientes (esto se delega a otra ventana, RegistroClientes).
- Modificación: Editar la información de un cliente existente.
- Eliminación: Borrar un cliente de la base de datos.

Implementación del patrón Memento

A continuación, se describen las clases creadas y los fragmentos de código modificados o añadidos en las clases existentes para la implementación del patrón Mediator.

Clase ClientesMediator (Interfaz)

Se creó una interfaz ClientesMediator que define los métodos para la comunicación entre el ClientesPanel y la clase Clientesclass.

Código:

```
package Modelo;

import Vista.Clientes;
import java.util.List;

/**
 *
 * @author Alan
```

```
 */
public interface ClientesMediator {

    void registrarClientePanel(Clientes clientesPanel);

    void registrarClientesClass(Clientesclass clientesClass);

    void obtenerClientes();

    void buscarClientesPorNumero(String numero);

    void guardarNuevoCliente(Clientesclass cliente);

    void agregarNuevoCliente(); // Para el botón Nuevo

    void modificarCliente(int selectedRow); // Para el botón
Modificar

    void eliminarCliente(int selectedRow); // Para el botón Eliminar

    void actualizarTabla(); // Para el botón Actualizar
}
```

Clase ConcreteClientesMediator (Mediador Concretor)

Se creó la clase ConcreteClientesMediator que implementa la interfaz ClientesMediator. Esta clase coordina la comunicación entre el ClientesPanel y la Clientesclass..

Código:

```
package Modelo;

import Vista.Clientes;
import Vista.RegistroClientes;
import java.util.List;
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;

/**
 *
 * @author Alan
 */
public class ConcreteClientesMediator implements ClientesMediator {

    private Clientes clientesPanel;
```

```
    private Clientesclass clientesClass;

    @Override
    public void registrarClientePanel(Clientes clientesPanel) {
        this.clientesPanel = clientesPanel;
    }

    @Override
    public void registrarClientesClass(Clientesclass clientesClass)
{
        this.clientesClass = clientesClass;
    }

    @Override
    public void obtenerClientes() {
        if (clientesClass != null && clientesPanel != null) {
            List<Clientesclass>             clientes             =
clientesClass.obtenerClientes();
            clientesPanel.actualizarTablaUI(clientes);
        }
    }

    @Override
    public void buscarClientesPorNumero(String numero) {
        if (clientesClass != null && clientesPanel != null) {
            List<Clientesclass>             clientes             =
clientesClass.obtenerClientesPorNumero(numero);
            clientesPanel.actualizarTablaUI(clientes);
        }
    }

    @Override
    public void agregarNuevoCliente() {
        if (clientesPanel != null) {
            RegistroClientes registro = new RegistroClientes(this);
// Pass the mediator
            registro.setVisible(true);
        }
    }

    // Method to handle saving a new client
    public void guardarNuevoCliente(Clientesclass cliente) {
        if (clientesClass != null) {
            clientesClass.agregarClienteBD(cliente);  //  Call  the
method in Clientesclass
            obtenerClientes(); // Refresh the table
```

```
        }
    }

    @Override
    public void modificarCliente(int selectedRow) {
        if (clientesPanel != null && selectedRow != -1) {
            DefaultTableModel modelo =
clientesPanel.getModeloTabla();
            String nombre = (String) modelo.getValueAt(selectedRow,
0);
            String apellidos = (String)
modelo.getValueAt(selectedRow, 1);
            String celular = (String)
modelo.getValueAt(selectedRow, 2);
            String rfc = (String) modelo.getValueAt(selectedRow, 3);
            String correo = (String) modelo.getValueAt(selectedRow,
4);

            Clientesclass cliente = new Clientesclass();
            cliente.setNombre(nombre);
            cliente.setApellidos(apellidos);
            cliente.setCelular(celular);
            cliente.setRfc(rfc);
            cliente.setCorreo(correo);

            if (clientesClass != null) {
                clientesClass.actualizarClienteBD(cliente);
                obtenerClientes(); // Refrescar la tabla después de
la modificación
            }
        } else if (clientesPanel != null) {
            JOptionPane.showMessageDialog(clientesPanel,
"Seleccione una fila para modificar.");
        }
    }

    @Override
    public void eliminarCliente(int selectedRow) {
        if (clientesPanel != null && selectedRow != -1) {
            DefaultTableModel modelo =
clientesPanel.getModeloTabla();
            String celular = (String)
modelo.getValueAt(selectedRow, 2);

            if (clientesClass != null) {
                clientesClass.eliminarClienteBD(celular);
```

```
                obtenerClientes(); // Refrescar la tabla después de
la eliminación
            }
        } else if (clientesPanel != null) {
            JOptionPane.showMessageDialog(clientesPanel,
"Seleccione una fila para eliminar.");
        }
    }

    @Override
    public void actualizarTabla() {
        obtenerClientes();
    }
}
```

Clase Clientes (Colega Concretor)

Se modificó la clase Clientes para que se comunique con la clase Clientesclass a través del ConcreteClientesMediator. Se eliminaron las llamadas directas a los métodos de Clientesclass..

Código:

```
public class Clientes extends javax.swing.JPanel {
    private ClientesMediator mediator;

    public void setMediator(ClientesMediator mediator) {
        this.mediator = mediator;
    }

    // ... (modificaciones en los métodos que interactúan con los
datos)

    private                                                    void
BtnActualizarActionPerformed(java.awt.event.ActionEvent evt) {
        mediator.obtenerClientes();  //    Llamada  a  través  del
Mediator
    }

    private                                                    void
BtnBuscarActionPerformed(java.awt.event.ActionEvent evt) {
        String numero = txtBuscar.getText();
        mediator.buscarClientesPorNumero(numero);   //   Llamada   a
través del Mediator
```

```
    }

    // ... (otros métodos que antes interactuaban directamente con
Clientesclass)
}
```

Clase Clientesclass (Colega Concreto)

La clase Clientesclass se mantiene con la lógica de acceso a datos, pero ahora se llama desde el ConcreteClientesMediator en lugar de directamente desde Clientes.

Código:

```
public class Clientesclass {
//… otros métodos …
    public void actualizarClienteBD(Clientesclass cliente) {
        Conexion conex = new Conexion();
        String consulta = "UPDATE Cliente SET Nombre = ?, Apellidos
= ?, RFC = ?, Correo = ? WHERE Celular = ?";

        try           (PreparedStatement           pst           =
conex.getConnection().prepareCall(consulta)) {
            pst.setString(1, cliente.getNombre());
            pst.setString(2, cliente.getApellidos());
            pst.setString(3, cliente.getRfc());
            pst.setString(4, cliente.getCorreo());
            pst.setString(5, cliente.getCelular());
            pst.executeUpdate();
            JOptionPane.showMessageDialog(null, "Cliente Modificado
Con Exito !!!");
        } catch (SQLException e) {
            JOptionPane.showMessageDialog(null, "Error al modificar
cliente: " + e.toString());
        }
    }

    public void eliminarClienteBD(String celular) {
        Conexion conex = new Conexion();
        String consulta = "DELETE FROM Cliente WHERE Celular = ?";
        try           (PreparedStatement           pst           =
conex.getConnection().prepareCall(consulta)) {
            pst.setString(1, celular);
            pst.executeUpdate();
            JOptionPane.showMessageDialog(null,  "Cliente  Eliminado
Con Exito !!!");
```

```
        } catch (SQLException e) {
            JOptionPane.showMessageDialog(null, "Error al eliminar
cliente: " + e.toString());
        }
    }

    public void agregarClienteBD(Clientesclass cliente) {
        Conexion conex = new Conexion();
        String consulta = "INSERT INTO Cliente (Nombre, Apellidos,
Celular, RFC,Correo) VALUES (?, ?, ?, ?,?)";
        try (
                PreparedStatement pst =
conex.getConnection().prepareCall(consulta)) {
            pst.setString(1, cliente.getNombre());
            pst.setString(2, cliente.getApellidos());
            pst.setString(3, cliente.getCelular());
            pst.setString(4, cliente.getRfc());
            pst.setString(5, cliente.getCorreo());
            pst.execute();
            JOptionPane.showMessageDialog(null, "Cliente Agregado
Con Exito !!!");
        } catch (SQLException e) {
            JOptionPane.showMessageDialog(null, "Error al agregar
cliente: " + e.toString());
        }
    }
}
```

Justificación

El patrón Mediator simplifica la comunicación entre objetos, permitiendo un código más legible y evitando la sobre carga entre cada una de las clases que participan, en este caso, su implementación en el módulo de clientes permitió liberar la sobrecarga entre las clases Clientesclass y Clientes en donde métodos que interactúan con la base de datos se encuentran dentro de ellas, con Mediator la comunicación fluye visualmente igual pero entre participantes se logra liberar la sobrecarga de métodos.

UML

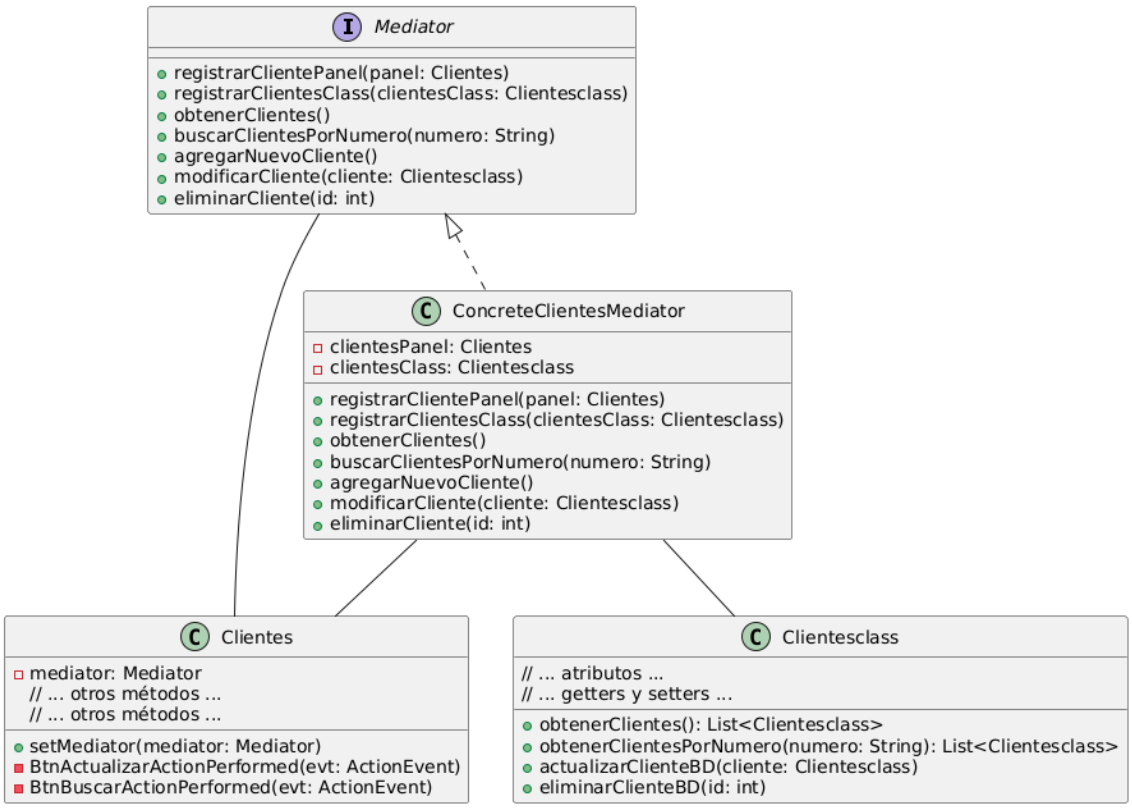

Figura 2. *Diagrama de clases de implementación de patrón Mediator en el módulo de clientes del sistema CopyMax.*

Ventajas y Desventajas

Tabla 1. Ventajas y desventajas del patrón Mediator

Ventajas	Desventajas
Reduce el acoplamiento: Al centralizar la comunicación, el Mediator desacopla los objetos Colleague. Esto facilita la modificación y extensión del sistema, ya que los cambios en un objeto no afectan directamente a otros.	Puede aumentar la complejidad: El Mediator en sí mismo puede volverse complejo si hay mucha lógica de comunicación.
Simplifica el comportamiento de los objetos: Los objetos Colleague se centran en su funcionalidad principal, delegando la comunicación al Mediator.	Puede generar una única fuente de errores: Si el Mediator falla, la comunicación entre los Colleague se interrumpe.

Nota: *En el contexto del módulo de clientes, las ventajas del patrón Mediator se vió favorecido para la comunicación entre clases*.

Pruebas de escritorio

Figura 3. *Carga de clientes, el método setMediator en la clase Clientes, solicita los clientes al Mediator, el cual a su vez los obtiene de Clientesclass y actualiza la tabla en el Clientes.*

Figura 4. *La búsqueda de clientes implica que el DocumentListener del campo de texto notifica al Mediator, quien luego pide a Clientesclass los clientes filtrados y actualiza la tabla.*

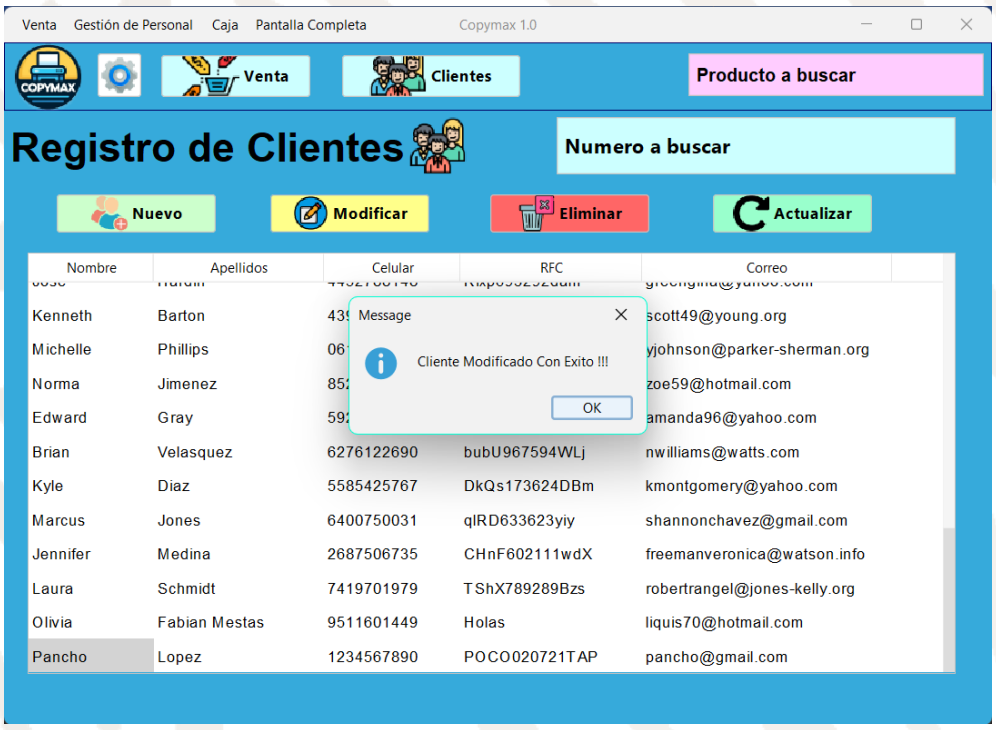

Figura 5. *Que permita la modificación de clientes es gracias a que* la acción del botón "Modificar" se delega al Mediator para actualizar los datos a través de Clientesclass.

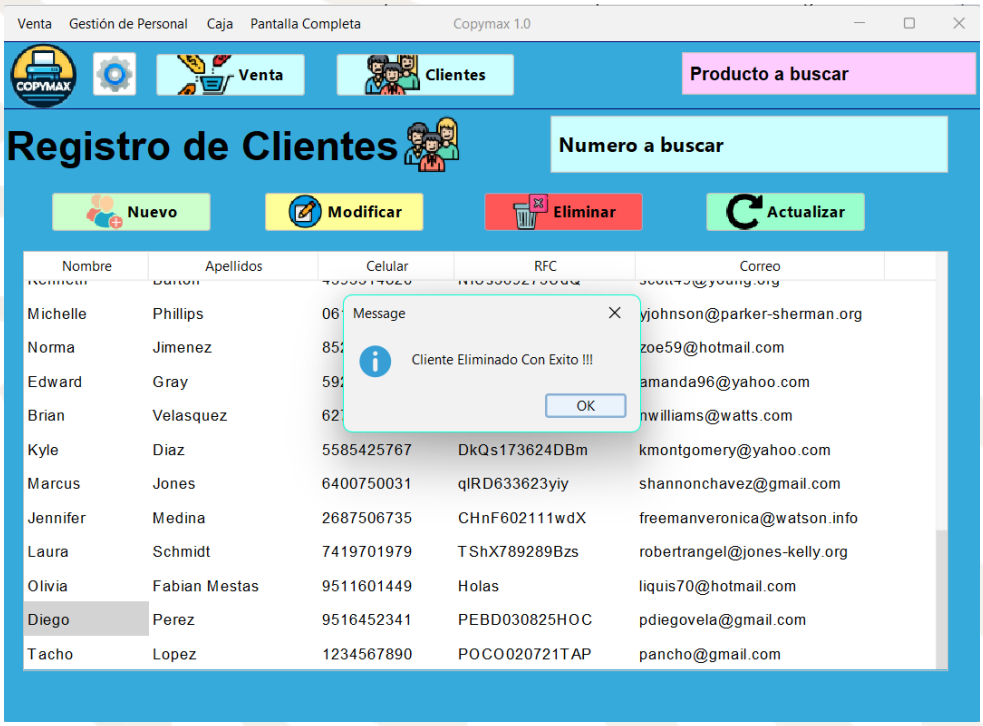

Figura 6. *La eliminación de clientes está coordinada por el Mediator*.

Conclusión

El patrón Mediator es una solución efectiva para gestionar la comunicación entre múltiples componentes dentro de un sistema de papelería, especialmente cuando la interfaz gráfica se vuelve más compleja. Centralizar las interacciones permite que el sistema sea más organizado, modular y fácil de mantener. Aunque puede generar una clase mediadora con muchas responsabilidades, su correcta implementación reduce el acoplamiento y mejora la escalabilidad a largo plazo, haciendo que la arquitectura del sistema sea más limpia y controlada.