

Ingeniería de Software II

Practica 3. Modelo Vista Controlador (MVC)

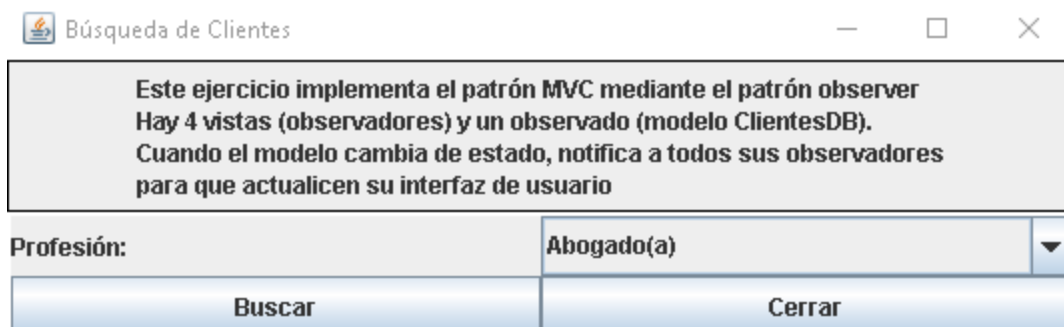
Nombres:

Brayan Stiven Garcia Navia
Daniel Alejandro Mejía Ascuntar

1 ¿Por qué y cómo funciona la aplicación utilizando el modelo vista controlador (MVC)?

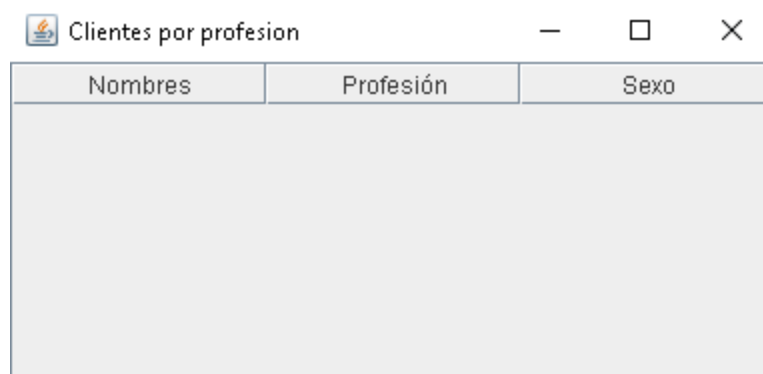
Como primera instancia, es preciso explicar algunas de las modificaciones (mejoras) que realizamos al código.

- **Botón Estadística:** Agregamos un botón a la vista GUIClientesProfesion, que tiene como funcionalidad desplegar las vistas GUIEstadisticaPorSexo y GUIEstadisticaPorSexoGrafica; En la versión original estas 4 vistas se mostraban al principio, es decir, antes de tener datos almacenados.
- Versión original:(Se muestran las 4 vistas al principio)



Búsqueda de Clientes	
Este ejercicio implementa el patrón MVC mediante el patrón observer Hay 4 vistas (observadores) y un observado (modelo ClientesDB). Cuando el modelo cambia de estado, notifica a todos sus observadores para que actualicen su interfaz de usuario	
Profesión:	Abogado(a) ▼
Buscar	Cerrar

GUIBusquedaClientes



Clientes por profesion		
Nombres	Profesión	Sexo

GUIClientesProfesion

Estadística por sexo

Hombres: 0

Mujeres: 0

GUIEstadisticaPorSexo

Profesión

GUIEstadisticaPorSexoGrafica

- Versión modificada:(Sólo se muestra una vista al principio)

Búsqueda de Clientes

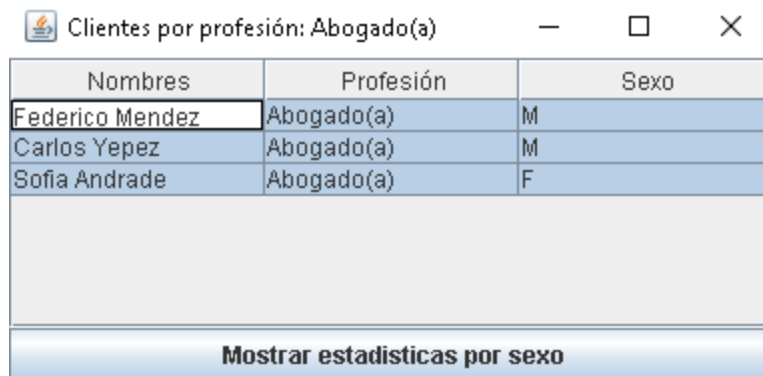
Este ejercicio implementa el patrón MVC mediante el patrón observer
Hay 4 vistas (observadores) y un observado (modelo ClientesDB).
Cuando el modelo cambia de estado, notifica a todos sus observadores
para que actualicen su interfaz de usuario

Profesión: Abogado(a)

Buscar Cerrar

GUIBusquedaClientes

Cuando se presiona Buscar, se despliega la segunda vista con el nuevo botón:



A Java Swing window titled "Clientes por profesión: Abogado(a)" with standard window controls. It contains a table with three columns: "Nombres", "Profesión", and "Sexo". The table lists three clients: Federico Mendez (Male), Carlos Yepez (Male), and Sofia Andrade (Female). Below the table is a button labeled "Mostrar estadísticas por sexo".

Nombres	Profesión	Sexo
Federico Mendez	Abogado(a)	M
Carlos Yepez	Abogado(a)	M
Sofia Andrade	Abogado(a)	F

Mostrar estadísticas por sexo

GUIClientesProfesion

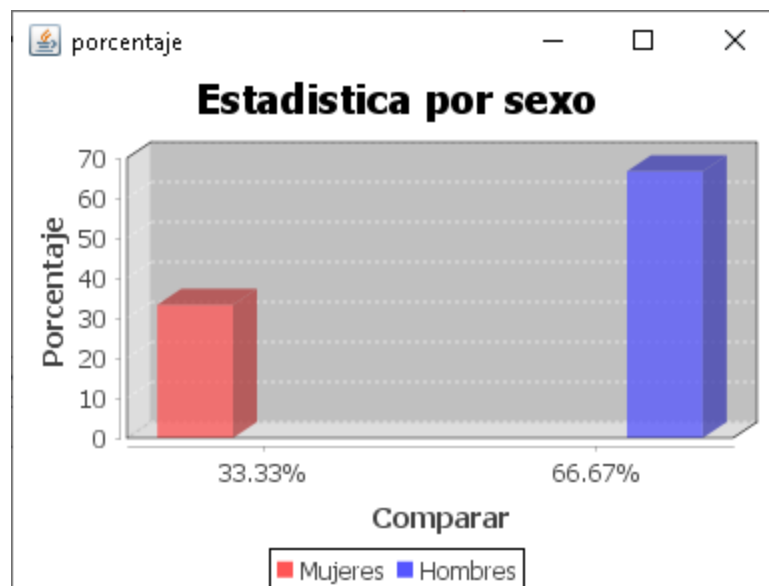
Cuando se presiona el botón Mostrar estadísticas por sexo, se despliega lo siguiente:



A Java Swing window displaying the results of the statistics button. It shows two lines of text: "Hombres: 2" and "Mujeres: 1".

Hombres:	2
Mujeres:	1

GUIEstadisticaPorSexo



GUIEstadisticaPorSexoGrafica

Al añadir esta funcionalidad, se creó un Controlador llamado **GUIEstadisticaPorSexoController** para “escuchar” las acciones del usuario sobre el nuevo botón.

Observaciones: Las modificaciones realizadas favorecen a la abstracción denominada “Separación de preocupaciones” y evita el despliegue de información que no ha sido pedida.

Teniendo esto en claro, procedemos a explicar cómo es que funciona el MVC en nuestra aplicación:

- La aplicación tiene 4 vistas, 1 modelo y 2 controladores. En la vista GUIEstadisticaPorSexoGrafica utilizamos JFreeChart para el gráfico de barras.
- ¿Cómo funcionan las notificaciones y los Observables y los Observadores?

Cuando se define el modelo que es ClientesDB implementa Observable de Java

```
*/  
public class ClientesDB extends java.util.Observable {  
  
    private ArrayList<Cliente> clientes;  
  
    private String profesion;  
    private ArrayList<Cliente> clientesPorProfesion;  
    private int totalHombres;  
    private int totalMujeres;  
    private double porcentajeHombres;  
    private double porcentajeMujeres;  
  
    public ClientesDB() {  
        inicializarDatos();  
    }  
}
```

Esto quiere decir que se declara a ClientesDB como un objeto el cual puede ser observado por un Observador el cual es una vista.

Como ClientesDB hereda Observable, puede acceder a los métodos públicos y atributos protegidos, entre ellos SetChanged(), y NotifyObservers().

SetChanged(): Lo que hace esta función es poner el atributo de la clase Observer Change a True, de manera que es una función bandera que notifica el cambio del modelo.

NotifyObservers(): Invoca al método update de todos los observadores de el modelo actual.

```
}  
  
for (int i = arrLocal.length-1; i>=0; i--)  
    ((Observer)arrLocal[i]).update(this, arg);
```

Entonces cada vez que el modelo cambia tiene que invocar las 2 anteriores funciones para notificar a los observadores de cualquier cambio realizado en el modelo. Por ejemplo en estas 2 funciones:

```
public void buscarClientesPorProfesion(String profesion) {  
    this.profesion = profesion;  
    clientesPorProfesion = new ArrayList();  
    this.totalHombres = 0;  
    this.totalMujeres = 0;  
  
    for (int i = 0; i < clientes.size(); i++) {  
        Cliente cli = clientes.get(i);  
        if (cli.getProfesion().equalsIgnoreCase(profesion)) {  
            clientesPorProfesion.add(cli);  
        }  
    }  
    setChanged();  
    notifyObservers();  
}
```

```
public void BuscarClientesPorSexo(){  
    //activarVistaEstadistica=true;  
    totalHombres=0;  
    totalMujeres=0;  
    for(int i=0;i<clientesPorProfesion.size();i++){  
        if(clientesPorProfesion.get(i).getSexo().equalsIgnoreCase("M")){  
            totalHombres++;  
        }else{  
            totalMujeres++;  
        }  
    }  
    setChanged();  
    notifyObservers();  
}
```

Ya cubrimos la parte del modelo, ahora abordaremos la de las vistas los cuales son los observadores.

Observadores:

Cuando se hace una vista no se hereda de ninguna clase sino que se implementa una interface llamada Observer.

Ahora si, se sobrescribe el método update del cual hablamos anteriormente, y dentro del update se programa la lógica a ejecutar cada vez que el modelo cambie, por ejemplo en la GUIEstadisticaPorSexoGrafica:

```
@Override
public void update(Observable o, Object arg) {

    ClientesDB cliDB = (ClientesDB) o;
    //f.setVisible(cliDB.isActivarVistaEstadistica());
    dataset.removeValue("Mujeres", String.valueOf(anteriorM)+"%");
    dataset.removeValue("Hombres", String.valueOf(anteriorH)+"%");
    double x=cliDB.getPorcentajeMujeres();
    anteriorM=x;
    dataset.addValue(x, "Mujeres", String.valueOf(x)+"%");
    x=cliDB.getPorcentajeHombres();
    anteriorH=x;
    dataset.addValue(x, "Hombres", String.valueOf(x)+"%");
    chart=ChartFactory.createBarChart3D("Estadística por sexo", "Comparar", "Porcentaje", dataset,PlotOrientation.VERTICAL,true,true,true);
}
```

Ahora, cómo entonces se agregan los observadores al modelo? Qué observador observa a qué?

Bueno eso se realiza con un método de la clase Observable llamado addObserver de esa manera puedes agregar uno o más observadores al observable, por ejemplo:

```
ClientesDB myModel = new ClientesDB();
GUIBusquedaClientes myView = new GUIBusquedaClientes(400, 50);
GUIClientesProfesion myView2 = new GUIClientesProfesion(400,250);
GUIEstadisticaPorSexo myView3= new GUIEstadisticaPorSexo(600,450);
GUIEstadisticaPorSexoGrafica myView4= new GUIEstadisticaPorSexoGrafica();

myModel.addObserver(myView);
myModel.addObserver(myView2);
myModel.addObserver(myView3);
myModel.addObserver(myView4);
```

Controladores:

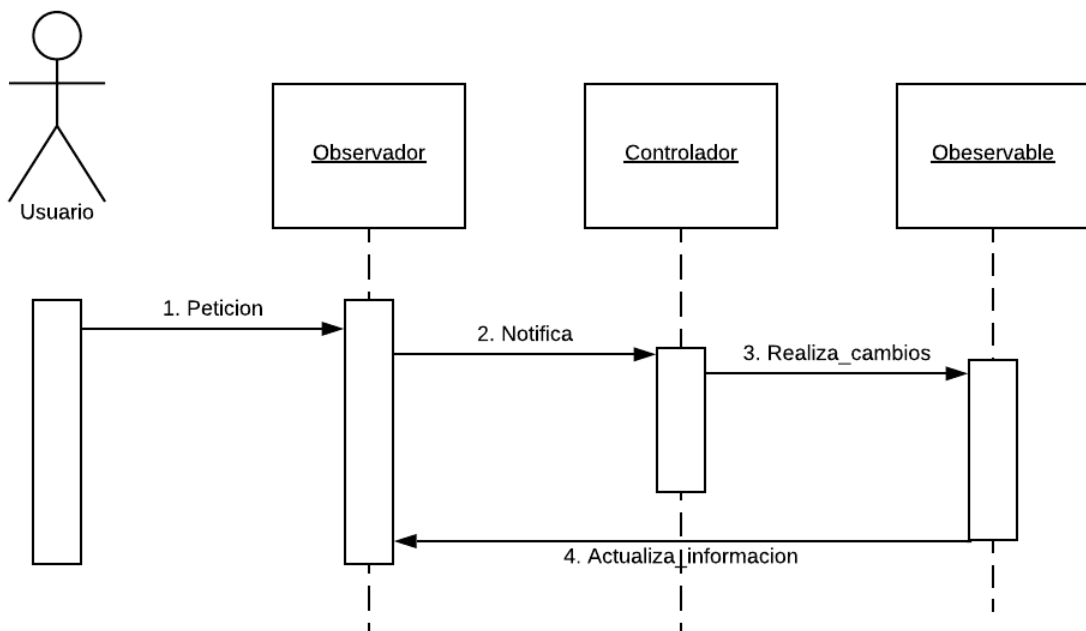
El controlador puede tener 1 o más vistas y un modelo, cuando se hace un controlador se implementa la interface ActionListener el cual tiene una función actionPerformed, la cual se sobrescribe. más tarde hablaremos de esta función.

Para agregar un modelo al controlador se usa una función simple llamada addModel, para añadir vistas addView. también debemos añadir el controlador a la vista en cuestión, con un simple addController, pero esta función no es de java, debemos

implementarla nosotros porque en esta función lo que hacemos es que a un botón en particular de la vista le añadimos un `addListener` además del original o primer `actionPerformed`, en el caso de la vista `GUIBusquedaClientes`, en el código `addController` se usó al botón buscar de la vista, esto quiere decir que después de haberse ejecutado el evento normal del botón buscar, el siguiente código que se ejecuta es el del `Controller` asociado en este caso `GUIBusquedaClientesController`, que sería este código:

```
@Override
public void actionPerformed(ActionEvent e) {
    vista2.setVisible(true);
    vista4.setVisible(false);
    vista3.setVisible(false);
    vista2.SetButtonVisible(true);
    modelo.buscarClientesPorProfesion(vista.getProfesion());
}
```

- Diagrama de secuencia del modelo vista controlador:



Síntesis:

Cada vez que se interactúa con el botón buscar se ejecuta el evento normal de la vista, luego el `actionPerformed` del controller, luego el `Controller` envía una petición al modelo en este caso una función llamada `buscarClientesPorProfesión` se modifica el modelo, el modelo notifica de su cambio y las vistas realizan su código `update`, de esta manera es muy simple la actualización de las vistas ya que se hace automáticamente cada vez que se notifique un cambio en el modelo.

Pros:

- La implementación se realiza de forma modular.

- Sus vistas muestran información actualizada siempre. El programador no debe preocuparse de solicitar que las vistas se actualicen, ya que este proceso es realizado automáticamente por el modelo de la aplicación.
- Cualquier modificación que afecte al dominio, como aumentar métodos o datos contenidos, implica una modificación sólo en el modelo y las interfaces del mismo con las vistas, no todo el mecanismo de comunicación y de actualización entre modelos.
- Las modificaciones a las vistas no afectan al modelo de dominio, simplemente se modifica la representación de la información, no su tratamiento.

Contras:

- MVC requiere la existencia de una arquitectura inicial sobre la que se deben construir clases e interfaces para modificar y comunicar los módulos de una aplicación. Esta arquitectura inicial debe incluir, por lo menos, un mecanismo de eventos para poder proporcionar las notificaciones que genera el modelo de aplicación; una clase Modelo, otra clase Vista y una clase Controlador genéricas que realicen todas las tareas de comunicación, notificación y actualización que serán luego transparentes para el desarrollo de la aplicación.
- MVC es un patrón de diseño orientado a objetos por lo que su implementación es sumamente costosa y difícil en lenguajes que no siguen este paradigma.