PATRÓN DE DISEÑO: MEMENTO

El patrón Memento es un patrón de diseño de comportamiento que permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación. Es decir, el Memento captura el estado interno de un objeto y lo externaliza, de modo que el objeto pueda ser restaurado a ese estado más tarde. Este patrón promueve el encapsulamiento, ya que el objeto cuyo estado se guarda (el Originator) no expone su estructura interna directamente.

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

Participantes del Patrón Memento

El patrón Memento tiene tres participantes clave:

1. **Originator** (Originador): Es el objeto cuyo estado necesita ser guardado y restaurado. El Originator crea un objeto Memento que contiene una instantánea de su estado actual.

También puede usar un objeto Memento para restaurar su estado anterior. En nuestro caso, Clientesclass es el Originator.

2. **Memento**: Es el objeto que almacena el estado del Originator. El Memento es inmutable: una vez creado, su estado no puede ser modificado. Debe tener dos interfaces: una interfaz estrecha para el Caretaker (que no permite modificar el estado) y una interfaz amplia para el Originator (que permite acceder al estado para restaurarlo). En nuestro código, ClienteMemento es el Memento.
3. **Caretaker** (Cuidador): Es el responsable de guardar y gestionar los Mementos. El Caretaker no examina ni modifica el contenido del Memento; simplemente lo almacena y lo pasa al Originator cuando es necesario restaurar el estado. El Caretaker no conoce la estructura interna del Originator ni del Memento. En nuestra implementación, Clientes (el JPanel) actúa como el Caretaker.

Módulo clientes

El módulo de clientes en el sistema CopyMax tiene como objetivo principal gestionar la información de los clientes (nombre, apellidos, celular, RFC, correo) 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.
- **Actualización Diferida** (Memento, después de implementarlo): Los cambios realizados a un cliente no se aplican inmediatamente a la base de datos. Se crea un Memento con el estado original del cliente, se modifican los datos en la interfaz, y un temporizador se encarga de aplicar los cambios a la base de datos después de un minuto. Dentro de ese

minuto, el usuario puede deshacer los cambios, restaurando el estado original a partir del Memento.

Implementación del patrón Memento

Clase Clientesclass

Se añadieron los siguientes métodos para la implementación del patrón memento:

- createMemento() que crea y devuelve un nuevo objeto ClienteMemento, encapsulando el estado actual del cliente.
- restoreFromMemento() que recibe un ClienteMemento y restaura el estado del cliente a partir de los datos contenidos en el Memento.

Código:

```
        // Método para crear un Memento (guardar el estado)
    public ClienteMemento createMemento() {
        return new ClienteMemento(id, Correo, Celular, Rfc, Nombre,
Apellidos);
    }

    // Método para restaurar el estado desde un Memento
    public void restoreFromMemento(ClienteMemento memento) {
        this.id = memento.getId();
        this.Correo = memento.getCorreo();
        this.Celular = memento.getCelular();
        this.Rfc = memento.getRfc();
        this.Nombre = memento.getNombre();
        this.Apellidos = memento.getApellidos();
    }
```

Clase ClienteMemento

Se creó la clase para el memento en la cual el constructor es package-private, lo que restringe su creación a clases del mismo paquete, como Clientesclass. Además, se cuenta con inmutabilidad por lo que solo tiene getters, no setters. Una vez creado, el estado del Memento no puede cambiar.

Código:

```
public class ClienteMemento {

    private final int id;
    private final String Correo;
    private final String Celular;
    private final String Rfc;
    private final String Nombre;
    private final String Apellidos;

    // Constructor (nota el modificador 'package-private' o
'default', para que solo sea accesible desde el mismo paquete)
    ClienteMemento(int id, String correo, String celular, String
rfc, String nombre, String apellidos) {
        this.id = id;
        this.Correo = correo;
        this.Celular = celular;
        this.Rfc = rfc;
        this.Nombre = nombre;
        this.Apellidos = apellidos;
    }

    // Getters (solo getters, no setters, para que el Memento sea
inmutable)
    public int getId() {
        return id;
    }

    public String getCorreo() {
        return Correo;
    }

    public String getCelular() {
        return Celular;
    }

    public String getRfc() {
        return Rfc;
    }

    public String getNombre() {
        return Nombre;
    }

    public String getApellidos() {
```

```
        return Apellidos;
    }
}
```

Clase Clientes

Se creó la pila mementos, la cual almacena los objetos ClienteMemento. Se usa una pila para que la última modificación sea la primera en deshacerse (LIFO - Last In, First Out). La instancia clienteActual es una referencia al objeto Clientesclass que se está editando. Esto es importante para poder restaurar el estado correcto, ya que la tabla se actualiza inmediatamente. Y se añadió un temporizador timerActualizacion para la actualización del cliente en la base de datos.

Se modificó el listener del botón Modificar, BtnModificarActionPerformed para que se obtengan los datos del cliente a modificar, posteriorme se crea el objeto clienteActual y así guardarlo en el memente para antes de cualquier cambio, luego de forma visual se muestran los datos del cliente modificados en la tabla para que después de que el memento se haya guardado y el cleinteActual se haya creado, se inicialice el temporizador pasando como parámetro el objeto clienteActual.

Una vez recibido el clienteActial, el método iniciarActualizacionTemporizada cancela cualquier temporizador que esté antes de su llamado, de esta forma inicia uno nuevo y edita el hilo SwingUtilities.invokeLater para llamar el método actualizarClienteBD y así ejecutarlo pasado 1 minuto si no se cancela antes, permitiendo la actualización del cliente en la base de datos.

Finalmente, el listener del botón deshacer, BtnDeshacerActionPerformed, cancela el timer y recupera el memento del stack, restaurando así los datos del cliente.

Código:

```
    private Stack<ClienteMemento> mementos = new Stack<>();
    private Clientesclass clienteActual;   // Para mantener una
referencia al cliente que se está editando
    private Timer timerActualizacion; // Temporizador para la
actualización
…
```

```
    private                                                    void
BtnModificarActionPerformed(java.awt.event.ActionEvent evt) {
        // Obtiene la fila seleccionada en la tabla Tablaclientes
        int selectedRow = Tablaclientes.getSelectedRow();
        if (selectedRow != -1) {  // Extrae los datos del cliente de
la fila seleccionada
            // Obtener datos ANTES de modificar
            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);

            // Crear clienteActual y guardar el Memento ANTES de
cualquier cambio.
            clienteActual = new Clientesclass();
            clienteActual.setNombre(nombre);
            clienteActual.setApellidos(apellidos);
            clienteActual.setCelular(celular);
            clienteActual.setRfc(rfc);
            clienteActual.setCorreo(correo);
            mementos.push(clienteActual.createMemento()); // Guarda
el estado

            //Después de modificar, actualiza los datos del objeto
desde la tabla
            clienteActual.setNombre((String)
Tablaclientes.getValueAt(selectedRow, 0));
            clienteActual.setApellidos((String)
Tablaclientes.getValueAt(selectedRow, 1));
            clienteActual.setCelular((String)
Tablaclientes.getValueAt(selectedRow, 2));
            clienteActual.setRfc((String)
Tablaclientes.getValueAt(selectedRow, 3));
            clienteActual.setCorreo((String)
Tablaclientes.getValueAt(selectedRow, 4));

            //Iniciar el temporizador *después* de guardar el
memento y actualizar clienteActual
            iniciarActualizacionTemporizada(clienteActual); // Pasa
clienteActual
        } else {
```

```
            // Muestra un mensaje si no hay una fila seleccionada
            JOptionPane.showMessageDialog(this,   "Seleccione   una
fila para modificar.");
        }


    }
…
    private       void       iniciarActualizacionTemporizada(final
Clientesclass cliente) {
        // Cancela cualquier temporizador anterior (importante!)
        if (timerActualizacion != null) {
            timerActualizacion.cancel();
            timerActualizacion.purge(); // Liberar recursos

        }
        timerActualizacion  =  new  Timer();  //  Crea  un  nuevo
temporizador
        TimerTask tarea = new TimerTask() {
            @Override
            public void run() {
                // Tarea que se ejecutará después del retraso.
                SwingUtilities.invokeLater(new Runnable() { // Usar
invokeLater
                    @Override
                    public void run() {
                        actualizarClienteBD(cliente);  //  Pasa  el
cliente
                        System.out.println("Cliente  actualizado  en
la base de datos.");
                    }
                });
            }
        };

        //  Programa  la  tarea  para  ejecutarse  después  de  60000
milisegundos (1 minuto)
        timerActualizacion.schedule(tarea, 60000);
        System.out.println("Temporizador iniciado. Actualización en
1 minuto.");
    }
…
    private                                                    void
BtnDeshacerActionPerformed(java.awt.event.ActionEvent evt) {
        if (!mementos.isEmpty() && clienteActual != null) {
            // Cancelar la actualización temporizada, si existe
```

```
            if (timerActualizacion != null) {
                timerActualizacion.cancel();
                timerActualizacion.purge();
                timerActualizacion = null; // Establece a null
después de cancelar
                System.out.println("Temporizador cancelado.");
            }
            ClienteMemento memento = mementos.pop(); // Recupera el
último estado
            clienteActual.restoreFromMemento(memento); // Restaura
el estado

            // Actualiza la fila en la tabla
            int selectedRow = Tablaclientes.getSelectedRow();
            if (selectedRow != -1) {
                modelo.setValueAt(clienteActual.getNombre(),
selectedRow, 0);
                modelo.setValueAt(clienteActual.getApellidos(),
selectedRow, 1);
                modelo.setValueAt(clienteActual.getCelular(),
selectedRow, 2);
                modelo.setValueAt(clienteActual.getRfc(),
selectedRow, 3);
                modelo.setValueAt(clienteActual.getCorreo(),
selectedRow, 4);
                JOptionPane.showMessageDialog(this,        "Cambio
deshecho. Actualización cancelada.");
                actualizarTabla();
            }

        } else {
            JOptionPane.showMessageDialog(this, "No hay acciones
para deshacer.");
        }
    }
```

Justificación

Permitir el modificar los datos del cliente y tener un período de tiempo (1 minuto) para deshacer los cambios antes de que se apliquen a la base de datos, es un ejemplo de aplicación del patrón Memento el cual proporciona una forma simple de lograr esto sin tener que implementar una lógica compleja de seguimiento de cambios o copias temporales de la base de datos. Además, encapsula

el estado interno del objeto Clientesclass. El panel Clientes (Caretaker) no necesita conocer los detalles internos de Clientesclass para guardar y restaurar su estado. Esto mejora la modularidad y reduce el acoplamiento entre las clases. Esto permite la escalabilidad ya que en caso de añadir nuevos campos basta con modificar Clientesclass y ClienteMemento. Por otro lado la implementación de una pila permite almacenar los mementos y poder deshacer las acciones en orden inverso a como se realizaron.

UML

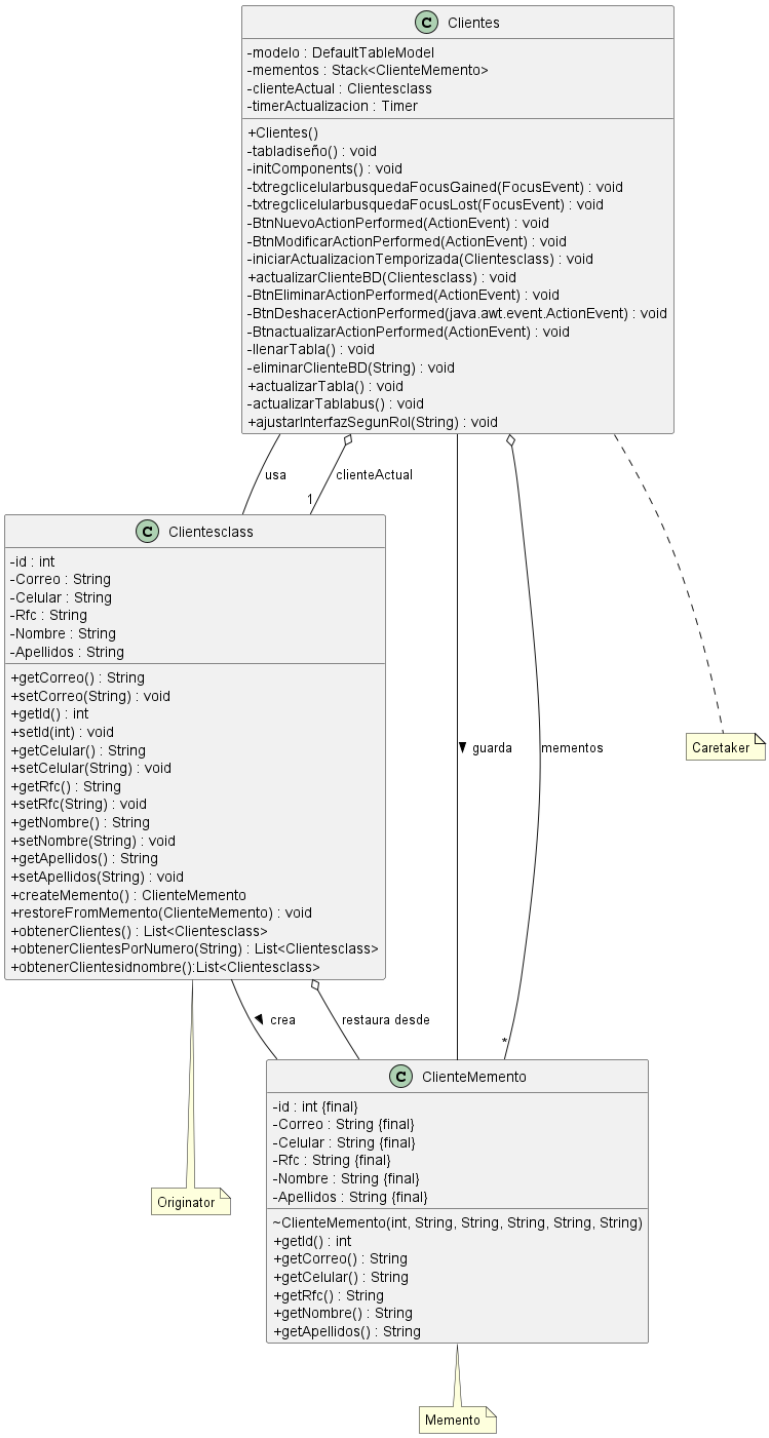

Figura 2. *Diagrama de clases de implementación de patrón memento en el módulo de clientes del sistema CopyMax.*

Ventajas y Desventajas

Tabla 1. Ventajas y desventajas del patrón memento

Ventajas	Desventajas
Encapsulamiento: Oculta la estructura interna del Originator. El Caretaker no necesita conocer los detalles del estado del Originator.	Consumo de memoria: Si el estado del Originator es grande o si se guardan muchos Mementos, el consumo de memoria puede ser significativo.
Simplifica el Originator: El Originator no necesita gestionar el historial de sus estados; esa responsabilidad se delega al Caretaker.	Sobrecarga de creación: Crear Mementos puede ser costoso en tiempo y recursos si el estado del Originator es complejo o si se crean Mementos con mucha frecuencia.
Desacoplamiento: El Originator y el Caretaker están desacoplados. El Caretaker no depende de la implementación concreta del Originator.	Complejidad (en algunos casos): Si el Originator tiene una estructura de estado muy compleja, la lógica de createMemento() y restoreFromMemento() puede volverse compleja.
Fácil de deshacer/rehacer: Proporciona una forma sencilla de implementar la funcionalidad de deshacer/rehacer.	Gestión del ciclo de vida de los Mementos: El Caretaker debe ser cuidadoso al gestionar los Mementos para evitar fugas de memoria (memory leaks) si los Mementos ya no son necesarios.
Restauración del estado: Permite restaurar el Originator a un estado previo de forma segura.	Acceso limitado al Memento: El Caretaker no tiene acceso al estado interno del Memento, lo cual es bueno para el encapsulamiento, pero podría ser una limitación en algunos casos muy específicos.
Delegación de la responsabilidad: Se delega la responsabilidad del guardado y restauración del estado.	-

Nota: *En el contexto del módulo de clientes, las ventajas del patrón Memento superan con creces las desventajas. El consumo de memoria y la sobrecarga de creación no son un problema significativo, dado que el estado del cliente (nombre, apellidos, etc.) es relativamente pequeño y las modificaciones no son extremadamente frecuentes.*

Pruebas de escritorio

Figura 3. *Estado inicial del cliente de prueba (Tacho)*

Figura 4. *Modificación de los datos del cliente (Tacho por Pancho) e inicialización del temporizador al pulsar en el botón modificar.*

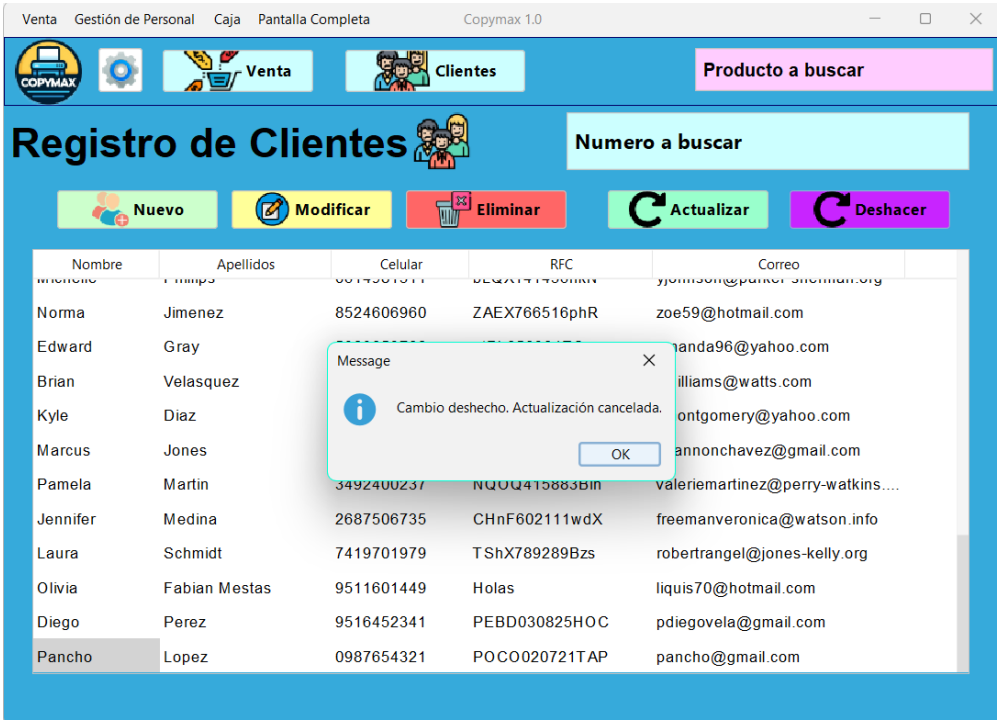

Figura 5. *Deshaciendo los cambios de los datos del cliente haciendo clic en el botón deshacer.*

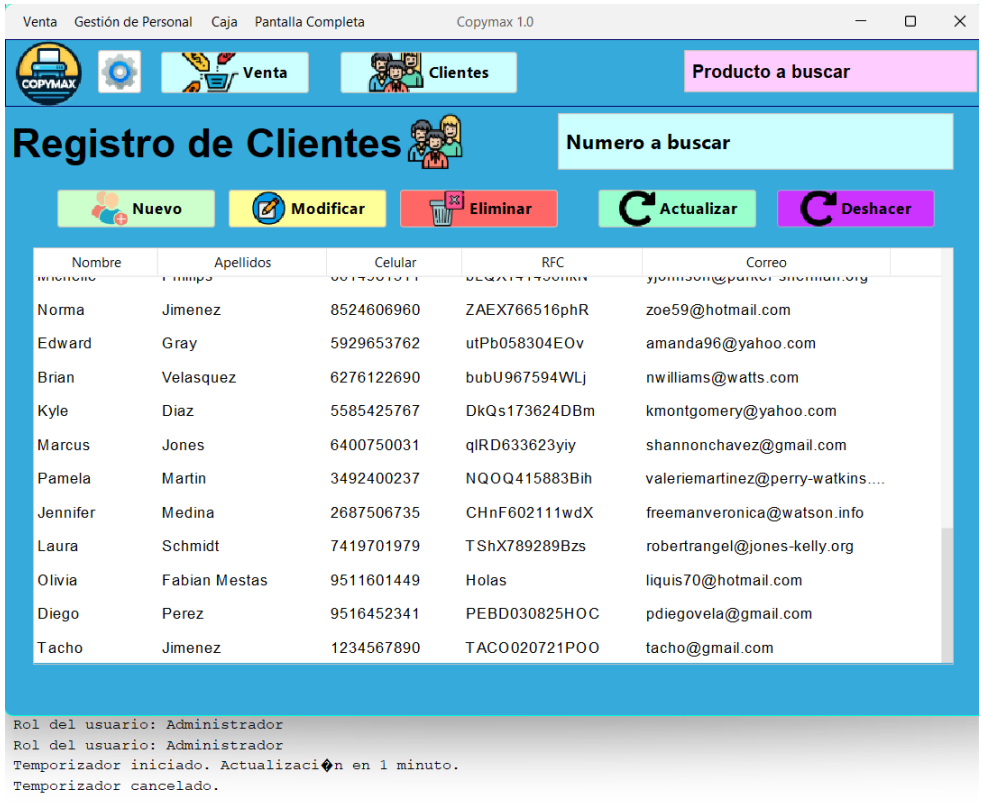

Figura 6. *Restauración de los datos del cliente Tacho (estado anterior aplicando el patrón Memento) y cancelación del temporizador.*

Figura 7. *Permitiendo que se actualice los datos del cliente.*

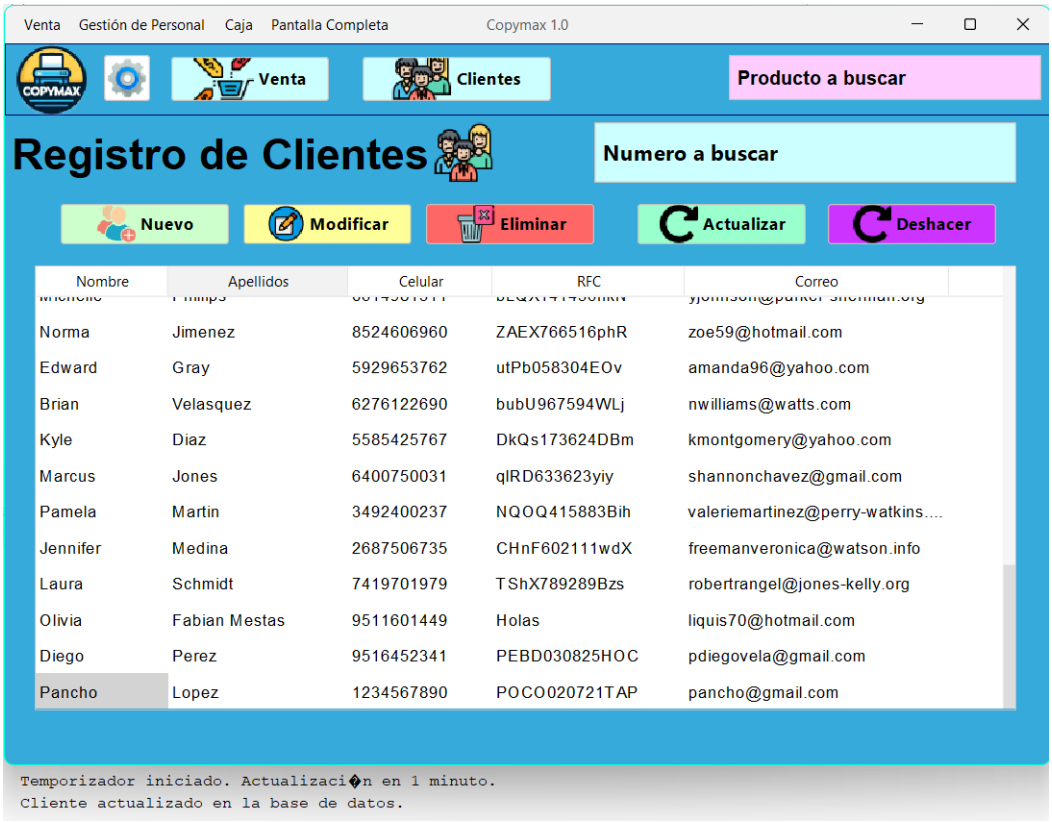

Figura 8. *Confirmación de modificación de los datos del cliente.*