# TECNOLOGICO NACIONAL DE MEXICO

# INSTITUTO TECNOLOGICO DE OAXACA

Carrera:

Ingeniería en Sistemas Computacionales

Materia:

Diseño e Implementación de software con patrones

Documentación de patrón Chain of Responsibility

Docente:

Espinoza Pérez Jacob

Equipo:

Ordaz Pacheco Ruudvan

Santos Manuel Julia Marlenny

Vera Acevedo Héctor Aramís

Grupo:

7SB

Fecha de entrega:

07/04/2025

# Documentación del Patrón Chain of Responsibility en el Sistema de Ventas

## Introducción

El patrón **Chain of Responsibility** es un patrón de diseño de comportamiento que **permite desacoplar el emisor de una solicitud de sus receptores**, dando a múltiples objetos la oportunidad de manejar la solicitud. En el sistema de gestión de stock, este patrón se implementó para coordinar las validaciones al actualizar el inventario, mejorando la modularidad y mantenibilidad del código.

**Interfaz** StockHandler

La interfaz StockHandler define el contrato que deben implementar los manejadores para procesar validaciones:

**Responsabilidades:**

- handle(): Ejecuta la validación específica y decide si pasa la solicitud al siguiente eslabón.
- setNext(): Establece el siguiente manejador en la cadena.

```
public interface StockHandler {
    void setNext(StockHandler next);
    boolean handle(InterActualizarStock frame);
}
```

### Clases Concretas de Manejadores

Cada manejador implementa una validación específica:

1. ProductoSeleccionadoHandler
    - Verifica que se haya seleccionado un producto en el JComboBox.

```
import javax.swing.JOptionPane;
import vista.InterActualizarStock;

public class ProductoSeleccionadoHandler implements StockHandler {
    private StockHandler next;

    @Override
    public void setNext(StockHandler next) {
        this.next = next;
    }

    @Override
    public boolean handle(InterActualizarStock frame) {
        if (frame.jComboBox_producto.getSelectedItem().equals("Seleccione producto:")) {
            JOptionPane.showMessageDialog(null, "Seleccione un producto");
            return false;
        }
        return next == null || next.handle(frame);
    }
}
```

2. CamposVaciosHandler
    - Valida que el campo de cantidad no esté vacío.

```
public class CamposVaciosHandler implements StockHandler {
    private StockHandler next;

    @Override
    public void setNext(StockHandler next) {
        this.next = next;
    }

    @Override
    public boolean handle(InterActualizarStock frame) {
        if (frame.txt_cantidad_nueva.getText().isEmpty()) {
            JOptionPane.showMessageDialog(null, "Ingrese una nueva cantidad para sumar el stock del producto");
            return false;
        }
        return next == null || next.handle(frame);
    }
}
```

3. NumerosValidosHandler
    - Comprueba que el valor ingresado sea numérico.

```java
public class NumerosValidosHandler implements StockHandler {
    private StockHandler next;

    @Override
    public void setNext(StockHandler next) {
        this.next = next;
    }

    @Override
    public boolean handle(InterActualizarStock frame) {
        try {
            Integer.parseInt(frame.txt_cantidad_nueva.getText().trim());
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "En la cantidad no se admiten caracteres no numéricos");
            return false;
        }
        return next == null || next.handle(frame);
    }
}
```

4. CantidadPositivaHandler
    - Asegura que la cantidad sea mayor a cero.

```java
public class CantidadPositivaHandler implements StockHandler {
    private StockHandler next;

    @Override
    public void setNext(StockHandler next) {
        this.next = next;
    }

    @Override
    public boolean handle(InterActualizarStock frame) {
        int cantidad = Integer.parseInt(frame.txt_cantidad_nueva.getText().trim());
        if (cantidad <= 0) {
            JOptionPane.showMessageDialog(null, "La cantidad no puede ser cero ni negativa");
            return false;
        }
        return next == null || next.handle(frame);
    }
}
```

**Flujo de trabajo:**

1. La interfaz InterActualizarStock envía la solicitud al primer manejador.
2. Cada manejador procesa su validación y decide si continúa la cadena.
3. Si alguna validación falla, se muestra un mensaje de error y se detiene el proceso.

**Modificaciones en** InterActualizarStock

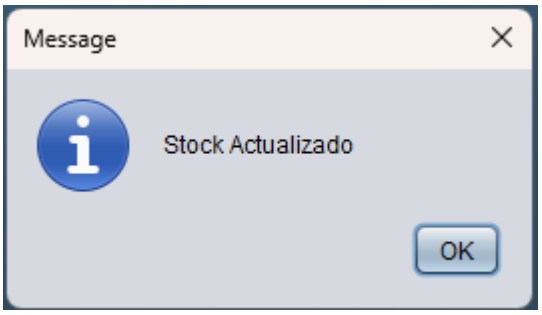
La interfaz gráfica fue adaptada para:

1. **Configurar la cadena** de manejadores en el método jButton1ActionPerformed.
2. **Delegar las validaciones** a la cadena antes de actualizar el stock.

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        //validamos seleccion del producto
     StockHandler productoSeleccionado = new ProductoSeleccionadoHandler();
    StockHandler camposVacios = new CamposVaciosHandler();
    StockHandler numerosValidos = new NumerosValidosHandler();
    StockHandler cantidadPositiva = new CantidadPositivaHandler();

    productoSeleccionado.setNext(camposVacios);
    camposVacios.setNext(numerosValidos);
    numerosValidos.setNext(cantidadPositiva);

    // Ejecutar la cadena de validaciones
    if (!productoSeleccionado.handle(this)) {
        return; // Si alguna validación falla, salir del método
    }

    // Si todas las validaciones pasan, proceder con la actualización
    Producto producto = new Producto();
    Ctrl_Producto controlProducto = new Ctrl_Producto();
    int stockActual = Integer.parseInt(txt_cantidad_actual.getText().trim());
    int stockNuevo = Integer.parseInt(txt_cantidad_nueva.getText().trim());

    stockNuevo = stockActual + stockNuevo;
    producto.setCantidad(stockNuevo);

    if (controlProducto.actualizarStock(producto, idProducto)) {
        JOptionPane.showMessageDialog(null, "Stock Actualizado");
        jComboBox_producto.setSelectedItem("Seleccione producto:");
        txt_cantidad_actual.setText("");
        txt_cantidad_nueva.setText("");
        this.CargarComboProductos();
    } else {
        JOptionPane.showMessageDialog(null, "Error al Actualizar Stock");
    }

    }
```

**Resultados**


Actualizar Stock de los Productos
Actualizar Stock de Productos
Producto:
Cerveza
Stock Actual:
80
Stock Nuevo:
120
Actualizar


Message
Stock Actualizado
OK


Actualizar Stock de los Productos
Actualizar Stock de Productos
Producto:
Cerveza
Stock Actual:
200
Stock Nuevo:
Actualizar

## Conclusión

La implementación del patrón Mediator en el sistema de gestión de clientes ha demostrado ser una solución efectiva para:

1. **Organizar** las interacciones entre componentes
2. **Reducir** las dependencias directas
3. **Facilitar** el mantenimiento y extensión del sistema
4. **Centralizar** la lógica de coordinación

Este enfoque permite que el sistema evolucione de manera más controlada, haciendo más sencilla la incorporación de nuevas funcionalidades o la modificación de las existentes sin afectar múltiples componentes.