



EDUCACIÓN
Tecnológico Nacional De México



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 Observer”

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

INTRODUCCIÓN	3
PATRÓN DE DISEÑO OBSERVADOR: IMPLEMENTADO EN COPY MAX.....	4
CÓDIGO IMPLEMENTADO.....	5
ESTRUCTURA UML.....	9
VENTAJAS Y DESVENTAJAS	9
CONCLUSIÓN.....	10

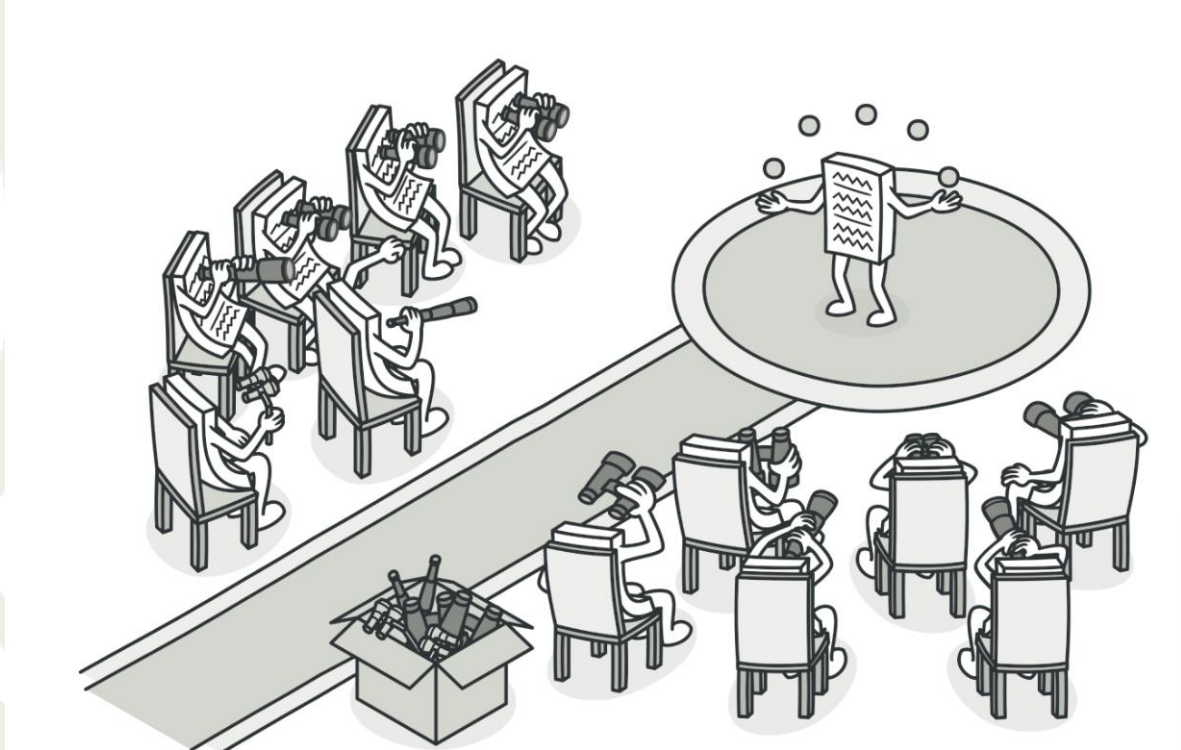
INTRODUCCIÓN

En un sistema de papelería donde múltiples módulos interactúan entre sí, es fundamental mantener actualizada la información de manera automática y en tiempo real. Para lograrlo, se implementó el patrón de diseño Observer, el cual permite establecer una relación de dependencia entre objetos de forma que, cuando uno de ellos cambia su estado, todos los objetos dependientes son notificados y actualizados automáticamente.

En este caso en el sistema la clase Ventas actúa como el sujeto, y Productos como observador, para así lograr permitir que la vista de productos actualice su tabla de inventario cada vez que se realiza una venta, sin necesidad de intervención manual por parte del usuario.

PATRÓN DE DISEÑO OBSERVADOR: IMPLEMENTADO EN COPY MAX

Observer es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando .



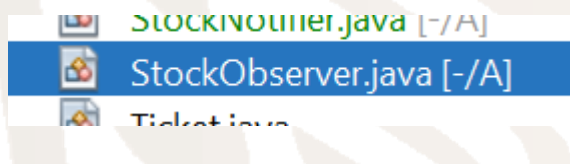
Se implementó el Patrón de Diseño Observer para que la vista de productos (Productos.java) se registre como "observador" de las ventas (Ventas.java), y así pueda recibir una notificación automática para refrescar los datos tras una venta exitosa.

CÓDIGO IMPLEMENTADO

Interfaz StockObserver

Se creó una interfaz llamada StockObserver que define un único método:

```
public interface StockObserver {  
  
    void actualizarStock();  
  
}
```



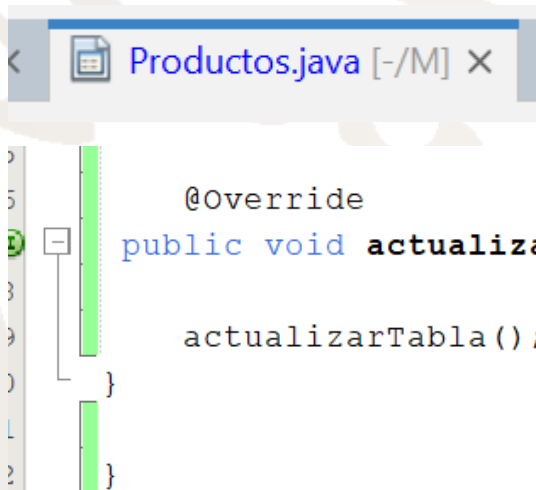
```
/**  
 *  
 * @author Admin  
 */  
  
public interface StockObserver {  
    void actualizarStock();  
}
```

Clase Productos como Observador

La clase Productos implementa la interfaz StockObserver, sobrescribiendo el método actualizarStock:

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

Esto permite que Productos actualice automáticamente su tabla cuando sea notificado.



```
< Productos.java [-/M] X
@Override
public void actualizarStock() {
    actualizarTabla();
}
```

Clase Ventas como Sujeto (Notifier)

La clase Ventas funciona como el sujeto que notifica a sus observadores. Para ello:

Se declaró una lista de observadores en ventas.java :

```
private List<StockObserver> observadores = new ArrayList<>();
```

Se implementaron los siguientes métodos:

```
public void agregarObservador(StockObserver observer) {
    observadores.add(observer);
}
```

```
public void notificarVenta() {
    for (StockObserver observer : observadores) {
        observer.actualizarStock();
    }
}
```



```

private List<StockObserver> observadores = new ArrayList<>();

public void agregarObservador(StockObserver observer) {
    observadores.add(observer);
}

public void notificarVenta() {
    for (StockObserver o : observadores) {
        o.actualizarStock();
    }
}

```

Registro del Observador

En el método inicializarCardLayout(), la clase Productos se registra como observador del panel de ventas:

ventaspanel.agregarObservador(productos);

```

private void inicializarCardLayout() {
    cardLayout = new CardLayout();
    Panelacambiar.setLayout(cardLayout);

    Productos productos = Productos.getInstance(); // singleton
    clientesPanel = new Clientes();
    ventaspanel = Ventas.getInstance();
    ventaspanel.agregarObservador(productos); // observador
}

```

Esto asegura que Productos será notificado cada vez que ocurra una venta.

Notificación tras una venta

Dentro del método ActualizarinventarioBd() de la clase Ventas, se llamó a notificarVenta() después del commit() exitoso, garantizando que los datos estén guardados antes de notificar:

notificarVenta();

```

7 public void ActualizarInventarioBd() {
8     int filas = modelo.getRowCount();
9     Conexion conex = new Conexion();
10    try {
11        // Iniciar una transacción
12        conex.getConnection().setAutoCommit(false);
13
14        for (int i = 0; i < filas; i++) {
15            String nombreProducto = (String) modelo.getValueAt(i, 1);
16            int cantidadVendida = Integer.parseInt(modelo.getValueAt(i, 0).toString());
17
18            // Actualizar la cantidad del producto en la base de datos
19            String consulta = "UPDATE Productos SET cantidad = Cantidad - ? WHERE Nombre_producto = ?"
20            try (PreparedStatement pst = conex.getConnection().prepareStatement(consulta)) {
21                pst.setInt(1, cantidadVendida);

```

después del commit() exitoso, para asegurar que los datos ya fueron guardados.

```

3 String consulta = "UPDATE Productos SET cantidad = Cantidad - ? WHERE Nombre_producto = ?"
4 try (PreparedStatement pst = conex.getConnection().prepareStatement(consulta)) {
5     pst.setInt(1, cantidadVendida);
6     pst.setString(2, nombreProducto);
7     pst.executeUpdate();
8 }
9
10 // Confirmar la transacción
11 conex.getConnection().commit();
12
13 // Notifica a observadores después de guardar exitosamente
14 notificarVenta();

```

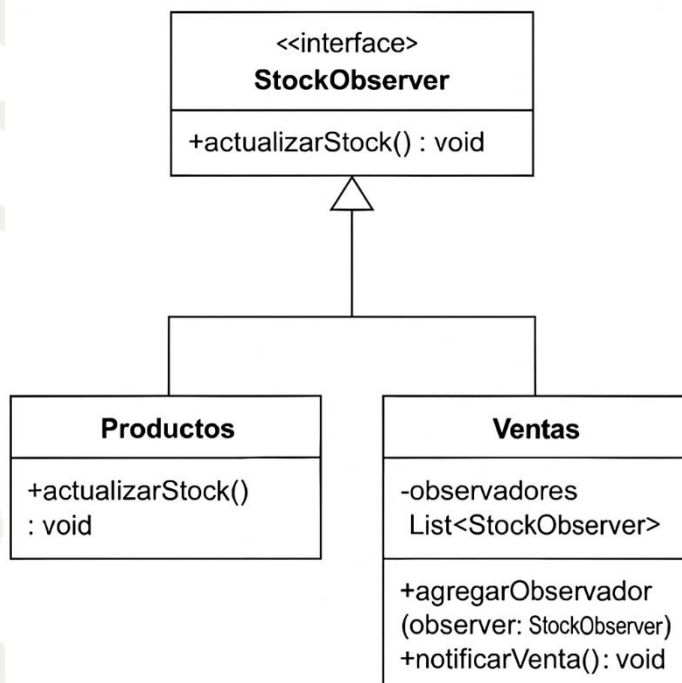
```

3 String consulta = "UPDATE Productos SET cantidad = Cantidad - ? WHERE Nombre_producto = ?"
4 try (PreparedStatement pst = conex.getConnection().prepareStatement(consulta)) {
5     pst.setInt(1, cantidadVendida);
6     pst.setString(2, nombreProducto);
7     pst.executeUpdate();
8 }
9
10 // Confirmar la transacción
11 conex.getConnection().commit();
12
13 // Notifica a observadores después de guardar exitosamente
14 notificarVenta();

```

Esto permite que la vista de productos se mantenga sincronizada automáticamente con las ventas realizadas.

ESTRUCTURA UML



VENTAJAS Y DESVENTAJAS

Ventajas	Desventajas
Actualización automática: Los cambios en Ventas se reflejan al instante en Productos.	Complejidad adicional: La implementación del patrón puede complicar la estructura del código.
Desacoplamiento: Ventas no necesita saber los detalles de Productos.	Difícil de depurar: Puede ser difícil rastrear qué observadores están registrados y cómo reaccionan.
Escalabilidad: Es fácil agregar más observadores en el futuro, como reportes o logs.	Riesgo de notificaciones innecesarias si no se gestionan bien las condiciones de cambio.

CONCLUSIÓN

La aplicación del patrón Observer en el sistema de Copy Max ha demostrado ser una solución eficaz para mantener la sincronización entre las ventas y el inventario. Gracias a esta implementación, se mejora la experiencia del usuario al automatizar la actualización de la información, se reduce la posibilidad de errores por omisión y se incrementa la eficiencia operativa del sistema.