TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE OAXACA

INGENIERÍA EN SISTEMAS COMPUTACIONALES.

MATERIA.

DISEÑO E IMPLEMENTACION DE SOFTWARE CON PATRONES.

TRABAJO:

DOCUMENTACIÓN.

PRESENTA:

Carlos Alberto Sosa Perera
Ríos Cruz Roberto Javier
Martínez Miguel Alexis
Hernández Ruiz Kevin Eduardo
Damián González Braulio Antonio
Hernández Velasco Luis Andrik

GRUPO:
7SB

PROFESOR:
Espinosa Pérez Jacob.

FECHA:

08 de mayo del 2025

Implementación del Patrón State en la Clase Cobro

¿Qué es el Patrón State?

El patrón State es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parecerá como si el objeto cambiara de clase. Este patrón:

1. Encapsula los diferentes comportamientos en clases de estado separadas
2. Delega las solicitudes al objeto de estado actual
3. Permite transiciones entre estados de manera limpia y organizada

Mejoras aportadas por la implementación del State

1. Organización y claridad del código

Antes: La lógica de los diferentes estados (selección de método de pago, validación, procesamiento) estaba mezclada en los métodos de la clase Cobro, haciendo difícil mantener y extender el código.

Después: Cada estado tiene su propia clase con responsabilidades claramente definidas, lo que hace el código más legible y mantenible.

2. Manejo consistente de transiciones de estado

Antes: Las transiciones entre estados se manejaban con flags y condicionales complejos dentro de los métodos.

Después: Cada estado sabe a qué otro estado puede transicionar y bajo qué condiciones, siguiendo el principio de responsabilidad única.

3. Validaciones más robustas

Antes: Las validaciones estaban dispersas en diferentes métodos y a veces se repetían.

Después: Cada estado maneja sus propias validaciones de manera consistente:

```
// Ejemplo en EstadoEfectivoSeleccionado
@Override
public void ingresarMonto(Venta.Cobro cobro, double monto) {
    if (monto >= cobro.getPre()) {
        double cambio = monto - cobro.getPre();
        cobro.setCambio(cambio);
        cobro.notifyCambioCalculado(cambio);
        cobro.setEstadoActual(new EstadoMontoValidado());
    } else {
        JOptionPane.showMessageDialog(cobro, "El monto ingresado es insuficiente.");
    }
}
```

4. Comportamiento específico por estado

Antes: El comportamiento variaba según flags booleanos (efectivoSeleccionado, tarjetaSeleccionada).

Después: Cada estado implementa su propia versión de los métodos:

```
// Comportamiento diferente en EstadoEfectivoSeleccionado vs EstadoTarjetaSeleccionada
@Override
public boolean procesarPago(Venta.Cobro cobro) {
    // Implementación específica para tarjeta
    if (cobro.getCorreo().trim().isEmpty()) {
        JOptionPane.showMessageDialog(cobro,
            "Por favor, ingrese un correo electrónico para enviar el comprobante.");
        return false;
    }
    // ...
}
```

5. Mejor manejo de errores

Antes: Los mensajes de error eran genéricos y no siempre contextuales.

Después: Cada estado provee mensajes de error específicos:

```
// En EstadoProcesandoPago
@Override
public void seleccionarMetodoPago(Venta.Cobro cobro, String metodoPago) {
    JOptionPane.showMessageDialog(cobro,
        "No se puede cambiar el método de pago mientras se procesa la transacción.");
}
```

6. Extensibilidad

Antes: Añadir nuevos estados o comportamientos requería modificar la clase Cobro directamente.

Después: Se pueden añadir nuevos estados simplemente implementando la interfaz CobroState, sin modificar la clase Cobro:

```
public class NuevoEstado implements CobroState {
    // Implementación de los métodos requeridos
}
```

7. Mejor integración con otros patrones

El State trabaja bien con:

- **Observer**: Notificando cambios de estado
- **Memento**: Guardando y restaurando estados
- **Facade**: Simplificando interacciones complejas

Impacto en la Clase Cobro

La clase Cobro ahora:

1. Delega el comportamiento a los objetos de estado
2. Es más simple y enfocada en coordinar las transiciones
3. Tiene una estructura más limpia para añadir nuevos estados
4. Maneja mejor las precondiciones y postcondiciones para cada operación

Ejemplo de Flujo con State

1. **EstadoInicial**: Esperando selección de método de pago
2. **EstadoEfectivoSeleccionado**: Validando monto recibido
3. **EstadoMontoValidado**: Listo para procesar pago
4. **EstadoProcesandoPago**: Generando ticket y enviando correo
5. **EstadoVentaCompletada**: Operación finalizada

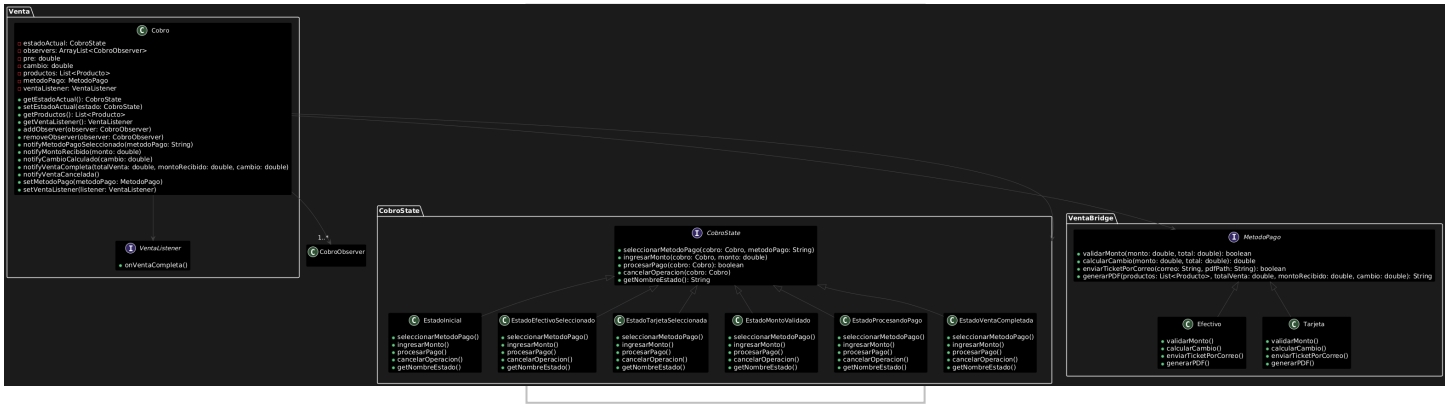
Cada transición es manejada por el estado actual, manteniendo la coherencia en todo el proceso.

Conclusión

La implementación del patrón State en la clase Cobro ha mejorado significativamente su diseño al:

- Separar claramente las responsabilidades
- Hacer el código más mantenible y extensible
- Proporcionar un flujo de trabajo más robusto y consistente
- Facilitar la adición de nuevos estados o comportamientos en el futuro

Diagrama UML



Evidencia de Funcionamiento