Proyecto 02: Butchery Manager

Equipo: SQLazo

- César Becerra Valencia (322064287)
- Victor Abraham Sánchez Morgado (322606003)
- José Luis Cortes Nava (322115437)

Breve descripción:

Consiste en una consola para registro de ventas, control de inventario, manejo de impuestos y cálculo de precios de una carnicería. Los patrones de diseño trabajados en esta práctica son: **State, Strategy, Observer, MVC, Singleton y Factory**.

Problemática a resolver:

El dueño de una carnicería considera que sus empleados necesitan llevar un registro de las siguientes cosas:

- Tipo de carne comprada por el consumidor (por kilo o por pieza)
- Descuentos aplicados en ventas realizadas
- Las órdenes de productos que un cliente compraría y el estado en que se encuentra esta orden (cancelada, pendiente o pagada)
- Llevar un control del inventario disponible

Sin embargo, sin un sistema que automatice estas acciones podrían haber errores de cálculo y de existencia en el inventario, por lo que nos solicita hacer un sistema que lleve el control de los puntos anteriores. Además, nos solicita que el sistema tenga una estructura MVC y al menos cuatro patrones de diseño para que el código sea mantenible y extensible.

El sistema debe permitir las siguientes acciones:

- Registrar productos en el inventario: tipo de producto, unidad (kilo o pieza), cantidad
- Actualizar el inventario automáticamente
- Proporcionar el costo de un producto a partir de su costo base y su estrategia de descuento
- Vender productos en base a las estrategias de descuento configurables
- Notificar de cambios en el inventario y ventas a observadores

La implementación de esta problemática debe estar orientada a los trabajadores de la carnicería, no a los clientes.

La interfaz del sistema debe proveer al trabajador de la carnicería un menú con opciones como las que se muestran a continuación:

Menú principal

1. Registrar productos en el inventario
2. Ver inventario

3. Calcular precio
4. Registrar venta (elegir estrategia de descuento)
5. Ver total de ventas
6. Salir

También, el dueño de la carnicería solicita que el sistema utilice persistencia de datos para que no se tenga que rellenar el inventario de nuevo cada vez que se vuelva a utilizar el programa, por lo que pide que se utilicen bases de datos para guardar los productos en el inventario. Por último, se pide que todos los cambios realizados en el inventario queden descritos en un archivo txt.

Patrones de diseño a utilizar:

MVC

Utilizamos MVC para separar la lógica del negocio y la presentación, lo cual nos permitiría cambiar de interfaz sin tener que cambiar el modelo (aumenta la extensibilidad de nuestro código). Los menús de consola y las validaciones de entradas no afectan la lógica interna de nuestro negocio gracias a este patrón. Los casos de uso como consultar el inventario o la venta de un producto se ven manejados por el apartado de Model. ConsoleView no conoce Inventory ni Sale, solo llama al Controller. ButcheryController solo usa el modelo y nunca habla directo con la vista.

Las clases relacionadas con este patrón son: ConsoleView, ButcheryController y ModelFacade.

Strategy

Lo utilizamos para intercambiar modelos de descuento en tiempo de ejecución para no tener la necesidad de escribir una cadena de if-else gigante y poder agregar más estrategias de descuento si es necesario. Se utiliza la interfaz IDiscountStrategy y las clases concretas NoDiscount, FrequentCustomerDiscount y PercentageDiscount.

- NoDiscount: se utiliza si el producto a vender no tiene ningún tipo de descuento aplicable
- FrequentCustomerDiscount: a aquellos clientes frecuentes se les hace un tipo de descuento especial
- PercentageDiscount: se aplica si el producto a vender tiene un descuento fijo

Este patrón nos permite modificar el precio de un producto en proceso de ser vendido dependiendo de si el cliente gusta utilizar un descuento.

Las clases relacionadas con este patrón son: NoDiscount, FrequentCustomerDiscount y PercentageDiscount.

Factory

Su propósito es encapsular la lógica de construcción de productos por peso o por unidad. Así el Controller no necesita conocer detalles de cada subtipo de producto. Gracias a este patrón podemos construir cualquier cantidad de productos que deseemos con una forma de medida dinámica, es decir, se pueden medir con respecto a su peso, por unidad, o se puede extender el código a que se utilice otro tipo de medida.

Las clases relacionadas con este patrón son: ProductFactory, ProductByWeight y ProductByUnit.

Singleton

La razón de utilizar el patrón Singleton es que solamente se pueda tener una única instancia del inventario durante la ejecución. Evita inventarios paralelos y estados inconsistentes cuando varias partes del sistema lo consultan. Tiene un constructor privado y un método getInstance() público que nos garantiza una única instancia del inventario.

Las clases relacionadas con este patrón son: Inventory.

Observer

El objetivo de usar Observer es notificar a aquellos interesados cuando cambie el inventario, sin tener que acoplar la clase Inventory a salidas concretas. Gracias a este patrón de diseño podemos hacer que se guarden en un archivo los cambios realizados en el inventario y que a la vez se impriman en consola.

Las clases relacionadas con este patrón son: FileNotifier, ListNotifier y ConsoleNotifier.

State

Utilizamos el patrón State en especial para evitar usos incorrectos del sistema. Con esto evitamos condicionales y acciones como intentar cancelar una venta después de que haya sido pagada, lo cual no debería suceder. Cada estado tiene una cantidad fija de acciones que se pueden cumplir en ese instante, el resto regresan mensajes que indican que no se puede realizar esa acción concreta en ese estado. La importancia de este patrón está en que únicamente se puede cambiar internamente el estado del objeto en tiempo de ejecución (no se puede escoger en qué estado se encuentra); a diferencia del patrón Strategy, con el que buscamos que se pueda escoger el tipo de descuento a aplicar.

Las clases relacionadas con este patrón son: PendingState, PaidState y CanceledState.