CICLO JAVA
PATRONES DE COMPORTAMIENTO STRATEGY

Imagina que estás desarrollando una tienda en línea moderna, donde los usuarios pueden adquirir una variedad de productos, desde electrónicos hasta ropa y accesorios. Como desarrollador, tu objetivo es garantizar que la experiencia de compra sea lo más fluida y personalizada posible. Una parte crucial de esta experiencia es el **sistema de pagos**.

En esta tienda, los clientes deben tener la libertad de elegir entre diferentes métodos de pago, como **tarjetas de crédito o débito**, **PayPal** y **efectivo contra entrega**. Cada uno de estos métodos tiene una lógica diferente:

- **Tarjeta de crédito o débito:** El sistema debe conectarse con una pasarela de pago externa para procesar la transacción. Esto incluye validar los datos de la tarjeta, confirmar fondos disponibles y registrar la transacción en la base de datos.
- **PayPal:** En este caso, el sistema debe redirigir al usuario al portal de PayPal, donde autentica su cuenta y confirma el pago. Una vez completado, el sistema recibe una notificación para validar la transacción.
- **Efectivo contra entrega:** No requiere interacción con pasarelas de pago. En lugar de eso, el sistema simplemente registra el pedido con un estado de "pendiente de pago" y notifica al cliente sobre las instrucciones para pagar al momento de recibir el producto.

Dado que cada método de pago tiene una implementación única, no sería eficiente manejar todas estas lógicas dentro de una única clase, ya que esto haría que el código sea difícil de mantener y de escalar. Por ejemplo, si en el futuro la tienda desea agregar un nuevo método de pago (como criptomonedas), el código tendría que ser modificado, rompiendo el principio de diseño abierto/cerrado, que sugiere que las clases deben estar abiertas a la extensión pero cerradas a la modificación.

Aquí es donde el **patrón de diseño Strategy** se convierte en una solución ideal. Este patrón te permite encapsular cada lógica de pago en su propia clase, delegando la responsabilidad de seleccionar y ejecutar el método adecuado a un contexto centralizado. De este modo, el sistema se vuelve más modular, fácil de mantener y preparado para futuras extensiones.

Tu tarea es diseñar este sistema usando el **patrón Strategy** para implementar los métodos de pago. Asegúrate de que el sistema sea lo suficientemente flexible como para permitir la adición de nuevos métodos de pago sin necesidad de modificar el código existente. ¡Manos a la obra!