**Documentación de tarea de patrones de diseño**

**1. Patrones Creacionales**

**Singleton**

* **Cómo**: Se implementó la clase ReservationSystem como un Singleton, asegurando que solo haya una instancia de este sistema en toda la aplicación. Esto se logra mediante un constructor privado y un método estático que controla la creación de la instancia.
* **Por qué**: El sistema de reservas debe ser único para evitar inconsistencias en las reservas y gestionar el estado global de la aplicación. Tener un punto único de acceso simplifica la gestión de recursos y configuración.

**Factory Method**

* **Cómo**: La clase ReservationFactory crea diferentes tipos de reservas (como HotelReservation y FlightReservation) mediante un método estático.
* **Por qué**: Este patrón permite la creación de objetos sin especificar la clase exacta. Facilita la extensibilidad; si más tipos de reservas se añaden en el futuro, solo se necesitaría modificar el factory, no el código cliente.

**Builder**

* **Cómo**: Dentro de las implementaciones de reserva (como HotelReservation y FlightReservation) permiten construir un objeto de reserva de manera flexible, configurando diferentes opciones (tipo de servicio, seguro, servicios adicionales) antes de construir la reserva final.
* **Por qué**: Este patrón es útil para crear objetos complejos de manera que los detalles de la construcción queden separados del objeto final. Esto mejora la legibilidad y el mantenimiento del código, especialmente cuando hay muchas configuraciones posibles.

**2. Patrones Estructurales**

**Adapter**

* **Cómo**: ExternalServiceAdapter adapta un servicio externo para que se pueda utilizar en el sistema de reservas como un tipo de Reservation.
* **Por qué**: Permite integrar servicios de terceros que pueden no seguir la misma interfaz. Esto facilita la interoperabilidad y la reutilización de código existente, permitiendo que el sistema se extienda fácilmente.

**Facade**

* **Cómo**: ReservationFacade proporciona una interfaz simplificada para las operaciones de reserva, ocultando la complejidad de la creación de diferentes tipos de reservas y su manejo.
* **Por qué**: El patrón Facade reduce la complejidad del sistema al proporcionar una interfaz clara. Mejora la legibilidad y hace que el sistema sea más fácil de usar.

**Decorator**

* **Cómo**: InsuranceDecorator añade funcionalidad adicional a las reservas (como la opción de añadir un seguro) sin modificar la clase original de Reservation.
* **Por qué**: Permite agregar nuevas funcionalidades de manera flexible y dinámica. Los decoradores son ideales para adherir comportamientos a objetos de manera incremental, sin la necesidad de crear un número excesivo de subclases.

**3. Patrones de Comportamiento**

**Strategy**

* **Cómo**: PaymentStrategy y sus implementaciones (CreditCardPayment, PayPalPayment) encapsulan diferentes métodos de pago. El cliente elige uno de estos métodos y lo utiliza para procesar el pago.
* **Por qué**: Este patrón permite cambiar el algoritmo de pago de manera dinámica en tiempo de ejecución. Facilita la adición de nuevos métodos de pago sin modificar el código existente, promoviendo así la extensibilidad.

**Observer**

* **Cómo**: Observer es una interfaz que permite que EmailNotifier y SMSNotifier respondan a cambios en el estado de la reserva. Cuando se crea una reserva, se notifica a todos los observadores.
* **Por qué**: Permite desacoplar el código de notificación del código de reserva. Los observadores pueden ser añadidos o eliminados sin afectar a la clase que genera los eventos, promoviendo un diseño más flexible y menos acoplado.

**Command**

* **Cómo**: ReservationCommand encapsula la solicitud de reserva como un objeto, permitiendo que las operaciones sean deshechas o reintentadas.
* **Por qué**: Este patrón es útil para manejar operaciones que pueden necesitar ser deshechas o ejecutadas de nuevo. Proporciona un enfoque estructurado para manejar acciones, manteniendo el código más organizado y fácil de manejar.