Patrones de diseño

Patrón Singleton

Se utilizará el patrón Singleton para la parte del administrador y asistente.

Este patrón asegura que una clase tenga solo una instancia y proporciona un punto de acceso global a ella. Puede utilizarse para controlar el acceso de los administradores o asistentes, garantizando que solo haya una única instancia del administrador activo en un momento dado.

Aplicación

- Gestiona las sesiones de usuario de manera centralizada, asegurando que los datos de sesión sean consistentes y accesibles de forma controlada.
- Manejar tokens de autenticación y mantener el estado de inicio de sesión.
- Al crear una clase SessionManager que controle la creación, mantenimiento y terminación de sesiones de usuario, garantiza una única instancia activa en todo el sistema.

Ventajas

El patrón Singleton asegura que solo exista una única instancia de una clase en toda la aplicación. Esto es útil cuando necesitas un control global sobre algún recurso o servicio, como un registro de configuración, una conexión a la base de datos o un logger centralizado. Al garantizar una única instancia, puedes evitar inconsistencias y asegurar que todos los componentes de la aplicación utilicen la misma configuración o recurso.

Al tener una única instancia accesible desde cualquier parte del código, el patrón Singleton simplifica el acceso a servicios o recursos compartidos. Esto facilita el mantenimiento del código, ya que no es necesario pasar instancias de objetos a lo largo de diferentes módulos o funciones. Además, centralizar la instancia en un único lugar hace que las modificaciones sean más fáciles de manejar.

En escenarios donde crear múltiples instancias de una clase puede ser costoso en términos de tiempo o memoria, el patrón Singleton ayuda a conservar recursos al asegurar que solo se cree una única instancia. Esto puede mejorar la eficiencia del sistema y reducir la carga sobre los recursos compartidos.

Patrón Strategy

Se utilizara el patrón Strategy para la parte de métodos de pago.

Este patrón define una familia de algoritmos o métodos y los hace intercambiables. Puede utilizarse para implementar diferentes métodos de pago (tarjeta, efectivo) sin cambiar el código cliente.

Aplicación

- Manejar diferentes métodos de pago como tarjeta de crédito y efectivo, permitiendo agregar fácilmente nuevos métodos en el futuro.

- Al definir una interfaz PaymentStrategy con implementaciones concretas como CreditCardPayment y CashPayment.
- El sistema selecciona la estrategia adecuada en función de la elección del usuario al momento del pago.

Ventajas

Con el patrón Strategy, es fácil añadir nuevas estrategias o modificar las existentes sin cambiar el código que las utiliza. Esto permite que el sistema sea extensible y que nuevas funcionalidades se incorporen de forma más rápida y sin riesgo de afectar otras partes del sistema.

Este patrón sigue el principio de diseño SOLID de "abierto para extensión, cerrado para modificación". Esto significa que puedes añadir nuevas estrategias sin modificar las clases existentes, lo que reduce la probabilidad de introducir errores en un código ya probado.

El patron Strategy permite cambiar el comportamiento de una clase en tiempo de ejecución seleccionando diferentes estrategias según el contexto. Además, las estrategias pueden ser reutilizadas en diferentes partes del sistema o en otros proyectos, lo que promueve la reutilización del código.

Patrón Observer

Se utilizará el patrón Observer para la parte de notificaciones, cancelaciones y cambios de estado de los usuarios.

Este patrón permite a los objetos suscribirse o escuchar eventos. Es útil en la aplicación cuando, por ejemplo, los conductores deben ser notificados cuando un pasajero solicita un viaje o cancela uno.

Aplicación

- Gestionar notificaciones de eventos importantes, como confirmación de registro, asignación de viaje, cancelaciones, y bloqueos de cuenta.
- Permite que múltiples componentes reaccionen a eventos sin estar fuertemente acoplados entre sí.
- Un objeto NotificationSubject al que se suscriben observadores como EmailNotifier, SMSNotifier, y PushNotifier, que envían notificaciones según el tipo de evento.

Ventajas

El patrón Observer promueve un diseño más flexible y desacoplado. Los observadores y el sujeto pueden variar de forma independiente, ya que el sujeto no necesita saber qué observadores lo están siguiendo. Esto reduce la dependencia entre componentes, facilitando la modificación y la evolución del sistema.

Es fácil añadir nuevos observadores sin modificar el código existente del sujeto o de otros observadores. Esto sigue el principio de abierto/cerrado, permitiendo extender el sistema con nuevas funcionalidades sin alterar el código que ya funciona.

Cuando el estado del sujeto cambia, todos los observadores suscritos se actualizan automáticamente y de manera consistente. Esto garantiza que todas las partes interesadas

estén sincronizadas con el último estado del sujeto, lo que reduce la posibilidad de inconsistencias y errores en el flujo de datos.