**Academia Kodigo**



**Tema:** Desarrollo de Aplicación Usando POO y Patrones de Diseño

**Integrantes del Grupo:**

* Edwin Romeo Rivas Diaz
* Luis Ernesto Figueroa Vásquez
* Kevin Alexander Aquino Vásquez
* Luis Alexis Velázquez Godoy
* Guillermo Alberto Asensio Jiménez
* Moisés Roberto Hernández Hernández
* Jonathan Alexander Ramírez Vázquez

**[17/11/2024]**

**Patrones de diseño aplicados**

**Singleton**:

El patrón Singleton se aplica a la clase ReservationSystem. La idea es que el sistema de reservas debe tener una única instancia en toda la aplicación. Al usar este patrón, se asegura de que no haya múltiples instancias del sistema, para facilitar el control de las reservas y evitar inconsistencias en el estado del sistema.

ReservationSystem system = ReservationSystem.getInstancia();

**Explicación**: Se obtiene la instancia única del sistema mediante el método estático getInstancia(). Esto garantiza que todas las operaciones sobre el sistema de reservas se realicen sobre la misma instancia.

**Observer**:

El patrón Observer se utiliza en la clase Notificable, donde los clientes se registran como "observadores" que serán notificados cuando se realicen ciertas acciones (como la creación de una reserva). Este patrón permite una comunicación eficiente entre el sistema de reservas y los clientes, sin necesidad de que el sistema tenga que conocer la implementación de los clientes.

notificable.addObserver(customer);

notificable.notify("Se crea una reservación a nombre de " + customerById.getName() + "...");

**Explicación**: En el código, cuando se crea una reserva, los clientes registrados como observadores reciben una notificación. Esto es útil para mantener a los clientes informados sobre eventos relevantes, como la creación de una nueva reserva.

**Factory Method**:

Se aplica el patrón Factory Method para la creación de habitaciones. Las diferentes clases de habitaciones (StandardRoom, DoubleRoom, PresidentialRoom) se crean mediante fábricas específicas (StandardRoomFactory, DoubleRoomFactory, PresidentialRoomFactory). Este patrón permite encapsular la lógica de creación de objetos, lo que facilita la extensión y mantenimiento del sistema.

Room room = new StandardRoomFactory().create();

**Explicación**: El patrón Factory Method facilita la creación de habitaciones sin necesidad de exponer los detalles internos de cómo se crean. Si en el futuro se agregan nuevos tipos de habitaciones, solo será necesario añadir una nueva fábrica sin modificar el código que utiliza las fábricas existentes.

**Decorator**:  
El patrón Decorator se aplica en el proceso de creación de reservas con servicios adicionales. En lugar de crear subclases para cada tipo de reserva (por ejemplo, ReservaConDesayuno, ReservaConTransporte), el sistema usa decoradores (BreakFastService, TransportService, SpaService) para agregar funcionalidades adicionales a las reservas existentes.

Reservation reservation = new BreakFastService(reservaBase);

**Explicación**: El uso de decoradores permite agregar servicios adicionales a una reserva sin alterar la clase BaseReservation. Esto facilita la extensibilidad del sistema, permitiendo agregar nuevos servicios sin modificar el código existente.