Nombre de los integrantes del equipo:

- Sebastian Alvarez
- Juan Camilo Palacio
- Juan José García

Patrones GRASP que aplica:

1. **Experto en información:** La clase “Mesa” recibe la responsabilidad de establecer el ambiente del elemento de confort, la información que requiere esta clase la puede adquirir de su misma clase.
2. **Creador:** No se implementó. Se utiliza la clase main para crear los objetos pues facilitar el proceso de creación de objetos desde una sola clase además de que permite validar pruebas.
3. **Controlador:** No se utilizó.
4. **Bajo acoplamiento:** Se asociaron clases fuertes con interfaces, en lugar de relacionar con una clase concreta. la clase “Restaurante” se asocia con a la interfaz “IReservable”
5. **Alta Cohesión:** La mayoria de las clases ejecutan sus responsabilidades sin requerir conocer el comportamiento de otras clases. Ejm: Reserva, Restaurante, Mesa.
6. **Indirección:** La clase intermedia que opera entre “Restaurante” y “Cliente” es la clase “Persona”.
7. **Polimorfismo y Variaciones Protegidas:** Cuando se verifica la disponibilidad de una reserva, el metodo “verificarDisponibilidad(List<Reserva> reservas)” se ejecuta para cualquiera de los elementos que implementen la interfaz “IReservable”.
8. **Fabricación pura:** No se implementó, pues no se considera que la distribución de responsabilidades de las clases es la correcta y no se necesitan clases adicionales.

Principios SOLID que aplica:

1. **Responsabilidad Única:** Las clases principales tienen una y solo una responsabilidad definida.
2. **Abierto/Cerrado y Sustitución de Liskov:** Se pueden crear clases para expandir la estructura del restaurante (Persona) sin que esta altere las propiedades de la clase a las que está relacionada (Cliente).
3. **Segregación de interfaces:** Las interfaces “IReserva” e “IReservable” nos ayudan a desacoplar métodos de clases que no usaran estos metodos pero que podrian depender de ellos (“Cliente” y “Mesa” respectivamente).
4. **Inversión de dependencias:** Si aplica, debido a que las clases que como parametro reciben objetos, están recibiendo es la abstracción de dichos objetos.

Atributos de Calidad que aplican:

1. **Simplicidad:** Se evidencia este atributo ya que cada clase tiene bien definida su responsabilidad.
2. **Escalable:** Las clases Cliente permite que se pueda anexar en un futuro otras clases con comportamientos ligeramente distintos. (Ejem: Que las empresas puedan realizar reservas).
3. **Usabilidad:** El sistema y su estructura con la que se planteó en si no es complejo de utilizar.

Listado de cambios del modelo V1 vs V2

Modelo inicial (V1)	Modelo mejorado (V2)
Alto acoplamiento entre clases	Se asociaron clases fuertes con interfaces, en lugar de relacionar con una clase concreta.
No se permite reservar algo diferente a una mesa	Se crea una interfaz 'IReservable' que permite que cada clase concreta que la implemente, pueda ser reservada.
Solo se permite a clientes de tipo persona hacer reservas	Se crea una interfaz 'IReserva' que permite que cada clase concreta que implemente de ella pueda realizar una reserva, en este caso, un cliente no solo podría ser una persona, sino que también, podría ser una empresa.
Todas las clases, son clases concretas	Se empiezan a utilizar clases abstractas para centralizar atributos y comportamientos comunes entre clases. Y también se empiezan a utilizar interfaces para definir un método o contrato para las diferentes clases que tengan un comportamiento común.
No había redundancia	Se agrega redundancia para tener mejor navegabilidad entre clases, como en el caso de 'Restaurante asociado con 'Clientes' y 'Restaurante asociado con 'Reservas' a pesar de que los clientes ya tienen reservas.
No se contemplaba la disponibilidad	Se tiene en cuenta la disponibilidad de cada clase que se implemente la interfaz 'IReservable'
La clase 'Reserva' tenía la responsabilidad de asignar el motivo	Crea una clase 'Motivo' con el fin de quitarle responsabilidad a la clase 'Reserva'

Patrones GoF que aplica:

Creacional

- Factoría (Parcialmente)

Estructural

- Bridge

Comportamiento

- Visitor

Listado de cambios V2 a V3

No se aplico ningun patron de diseño	Se aplicaron 2 patrones de diseño: ● Patrón de comportamiento ‘ Visitor’ para agregar comportamiento en caso de necesitarlo a nuestra clase ‘Restaurante’ y no tener que ir a modificar dicha clase. ● Patrón estructural ‘Bridge’ aplicado a lo que se puede reservar y a los ambientes que puede tener cada ítem que reservo. Con este patrón se puede crecer en ítems a reservar y ambientes para dichos ítems.
Acoplamiento entre Mesa y ElementoConfort	Para eliminar dicho acoplamiento, se creó una clase intermedia llamada ‘ItemReservable’ que es abstracta, que permite que los elementos a reservar sean independientes de los elementos de confort, pero que a su vez, cada elemento a reservar puede no tener o tener muchos elementos de confort asociados.
No se tenían clases responsables de la creación de instancias.	Se utiliza el patrón Factory para los clientes e Items reservables.

Nota: Violación del principio de responsabilidad unica.

Todas las instancias se están realizando desde el método ‘Main’ por el momento, porque designar la funcionalidad de que cada responsable cree las instancias que le corresponden, el modelo cambiaría demasiado, porque cuando se empezó a analizar como crear las instancias donde deberían ser creada, debido a fallas del modelo actual, el cambio seria grande. Por lo tanto no justifica modificarlo ya , sino para la versión final que si tendra muchos mas cambios, y adicional el tiempo para la implementación es corto.

Queda como compromiso de equipo, para la entrega final, desglosar más el problema e implementarlo de la manera correcta.