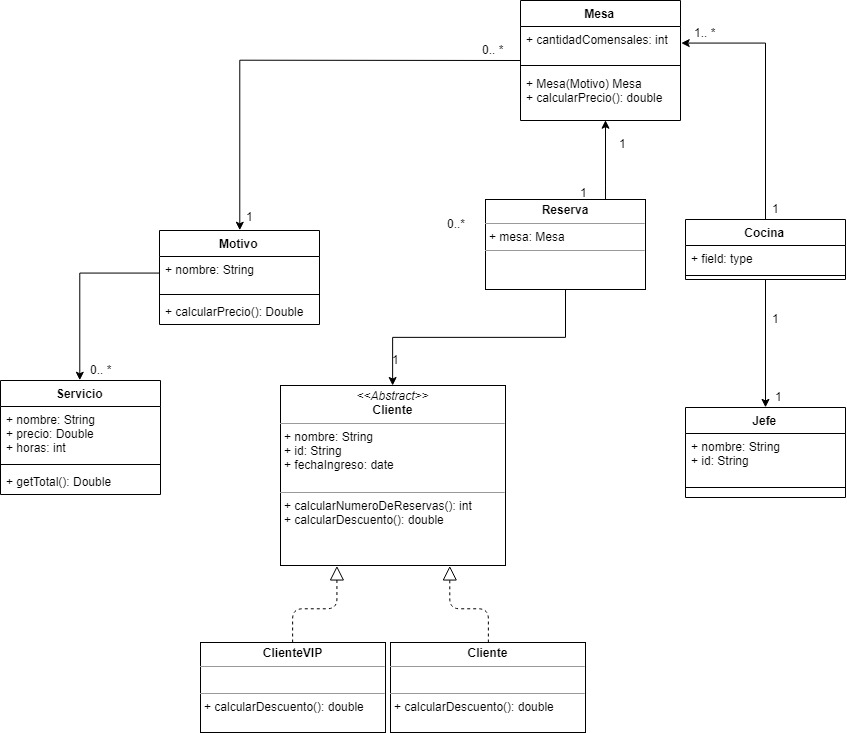
**Reporte de cambios en el diagrama UML “El gran chef”**

Dando un rápido vistazo a nuestro caso de estudio “El gran chef” se produjo en primeramente el siguiente diagrama de clases, el cual será sujeto de modificaciones a medida que se sigan principios y patrones de diseño del software:



<Imagen 1, diagrama de clases “El gran chef” sin correcciones>

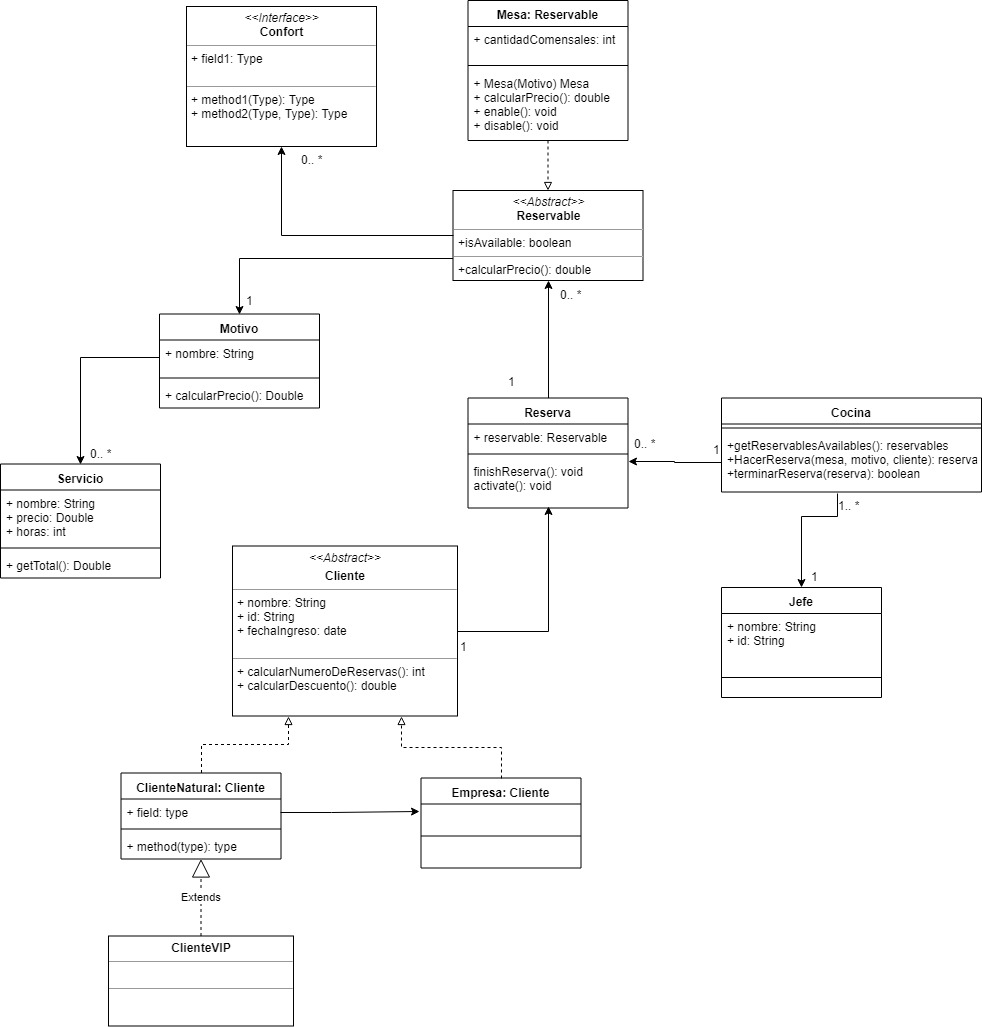
Luego, se hizo un repaso de los principios de diseño. Se procede a hacer un análisis más profundo con la finalidad de contemplar aspectos de funcionalidad, escalabilidad y robustez propios de los principios GRASP, estos cambios se enumeran:

1. Con finalidad de futuras integraciones y escalabilidad, la relación que tiene el chef y la cocina pasa de ser “uno a uno” a ser una relación “uno a muchos”
2. Mesa es una implementación concreta, ahora se creó una interfaz “Reservable” que permite que si en un futuro existen más tipos de lugares a reservar las clases con las que colabora dependan de una abstracción y no una implementación concreta.
   1. Protected variations
   2. Low
3. Se creará una interfaz “Cliente” que en un futuro nos va a permitir agregar más tipos de cliente, como “Empresa”, “Afiliado” entre otros.
   1. Liskov Substitution
4. Las clases cliente y cliente VIP serán cambiadas a una clase “ClienteNatural” que luego tendrá la facilidad de extender para agregar más tipos de cliente.
   1. Open Close
5. Ser creó una nueva interfaz llamada “Confort” la cual abstrae los comportamientos que dispositivos enunciados como televisores y DVD’s pueden pertenecer a una reserva.

Se descarta el uso de el principio “Creador” de GRASP, en nuestro diseño modelo UML debido a que:

1. No tenemos relaciones del tipo agregación en el diseño
2. A las clases que son contenidas o usadas cercanamente por otras (Como Reserva y Cocina) se le delega la función de ser creadas por sí mismas.

Después de contemplar estos cambios e implementarlos debidamente en una aplicación que nos permite crear nuevas reservas en la cocina, refinamos el diagrama, dando como resultado final el siguiente diseño:



<Imagen 2, diagrama de clases “El gran chef” refinado>

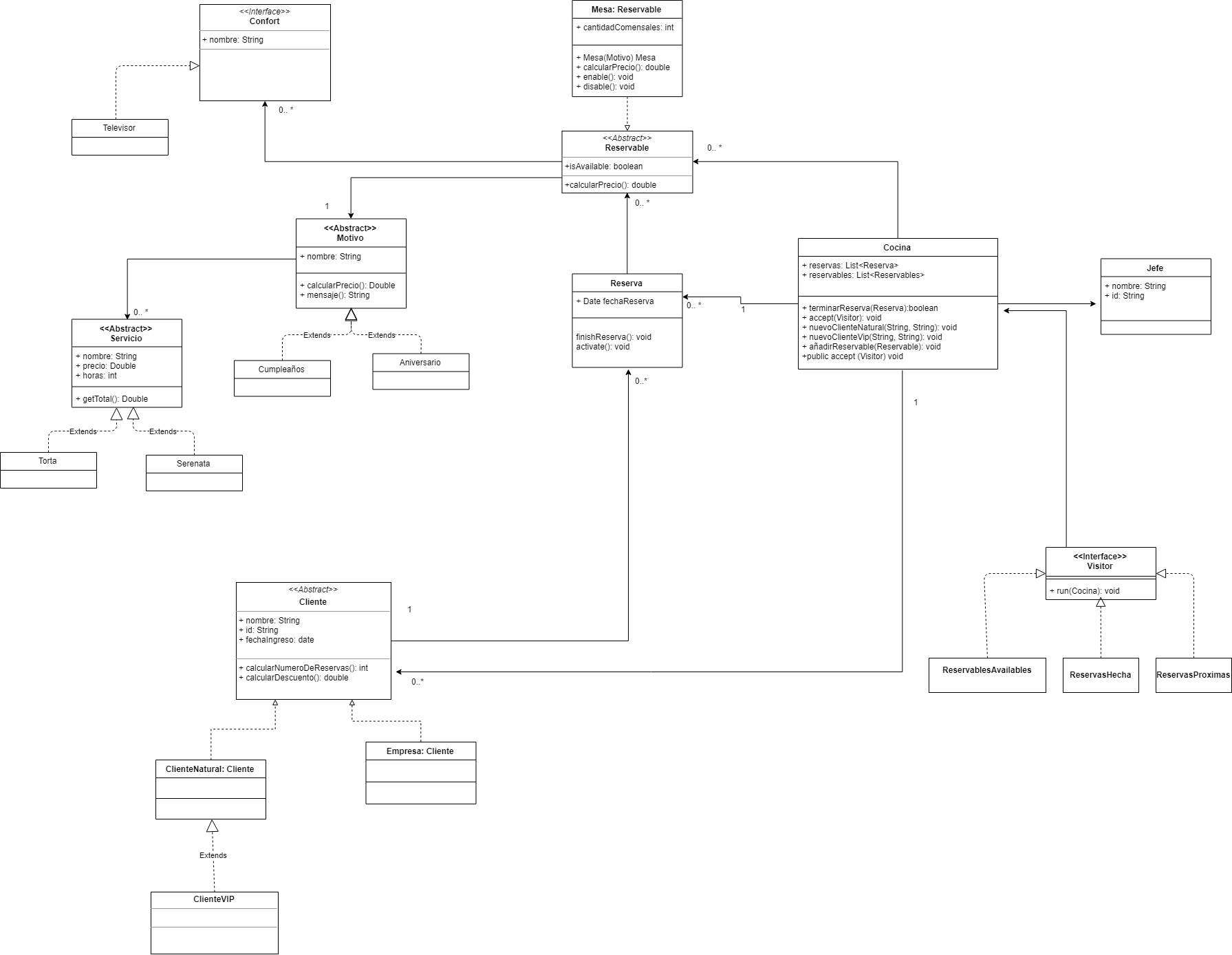
Contemplando estos cambios, se obtiene una aplicación más robusta, flexible y resistente a los futuros cambios y nuevas reglas de negocio.

**Patrones de diseño estructurales**

Complementando el diseño del diagrama de clases con los patrones de diseño estructural, decidimos complementar el prototipo con los siguientes cambios:

* Se crea una nueva relación entre cocina(Restaurante) y cliente, es responsabilidad del restaurante saber cuántos clientes posee, a su vez éstos son responsables de crear las reservas.
* Se adicionan las listas de reservas y reservables a la clase Cocina, éstas son necesarias para saber el estado de dichos elementos por parte del restaurante
* Adicionamos un confort (Televisor) para hacer más visible la relación de éste con su súper clase (Confort) a la cual le agregamos un atributo nombre, para identificarla respecto a otro confort
* En función de buenas prácticas, cambiamos la clase motivo, para que sea abstracta, con esto logramos, conjugar características comunes entre diferentes motivos. Esto mismo sucede con la clase servicio, la cual compila características similares entre sus miembros, razón por la cual también pasa a ser una clase abstracta.
* Creamos una interfaz visitor, con tres “visitantes” como ejemplo, estos van a “visitar” la clase cocina. Es implementado de esta manera por escalabilidad. Cocina es un clase que tiende a cambiar en sus operaciones, con el visitor lo podemos hacer sin modificar la clase cocina **(Visitor)**
* Aplicamos el patrón Bridge sobre la clase reservable que es una interfaz y confort que es abstracta, nos ayuda a eliminar código innecesario y repetitivo al tiempo que hacemos de la aplicación más flexible. **(Bridge)**

Lo que repercute en un diseño versión 3, con mayor detalle y robustez con respecto a la versión 2:



<Imagen 3, diagrama de clases refinado con patrones de diseño>

**Conclusiones:**

* En el diseño de aplicaciones, es preferible crecer en instancias que en clases, de otro modo, la aplicación de las clases se ve afectada por la complejidad del diagrama
* El patrón bridge, es preferible usarlo cuando la abstracción resulta poco flexible y las características comunes tienden a crecer, así se evita escribir códigos repetitivos en las clases.
* EL patrón factory nos ayuda a obtener instancias de las clases sin tener que recurrir a ellas, simplemente se la pedimos a “la fábrica”
* El patrón visitor nos permite agregar nuevas operaciones fácilmente. Estas abstraen características dentro del “visitante” con el visitor evitamos modificar las clases para agregar nuevas operaciones.