UNIVERSIDAD AUTÓNOMA DE YUCATÁN

Diseño del Software

Proyecto Final

Integrantes:

- Marín Chacón Alberto Ezequiel
- Moo Uc Itzel Andrea
- Morales Che Alan Jesús

Profesor: Juan Francisco Garcilazo Ortiz

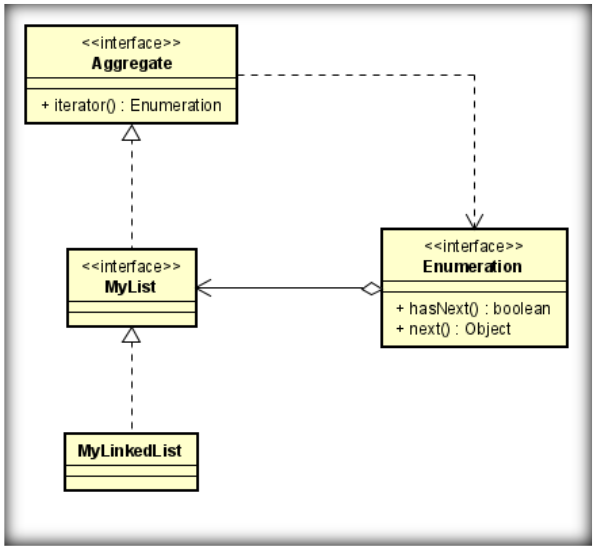
7 de junio de 2020

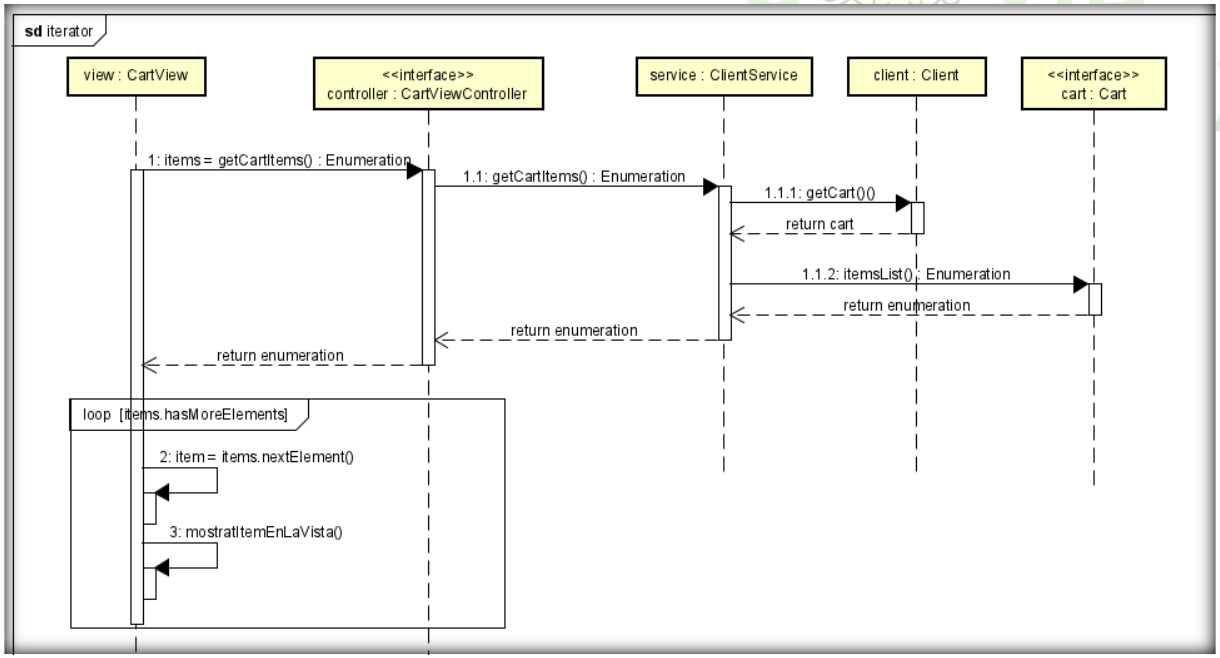
Patrón Iterator

Para este proyecto utilizamos iterator y nuestra propia implementación de lista ligada. Utilizamos la interface Aggregate para declarar la función iterator(), la cual devuelve una objeto Enumeration que nos permitirá proteger nuestras listas de ser modificadas por otras clases.

Como se ve en el diagrama, MyList extiende Aggregate, y a su vez, la clase concreta MyLinkedList implementa MyList, por lo tanto, está obligada a sobreescribir la función iterator().

Decidimos plantearlo así, pues esto nos permite tener un diseño open-closed, en lo que a SOLID se refiere. Pues gracias a la interface MyList, nosotros podemos modificar MyLinkedList sin que afecte a otras clases, al igual que podemos agregar otras implementaciones de listas.



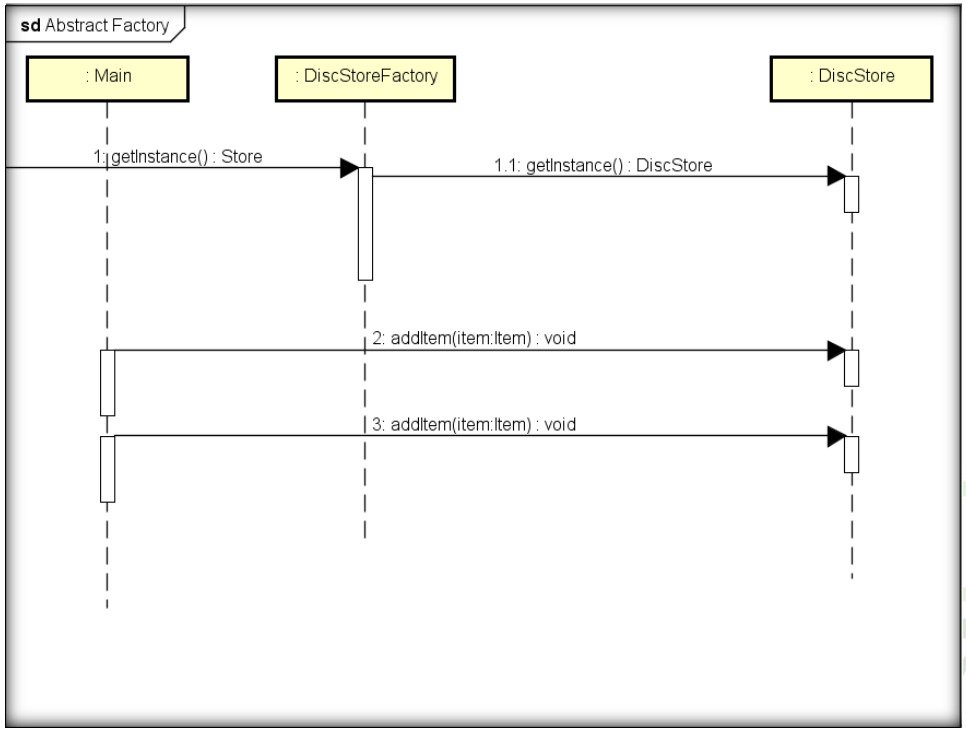


En el diagrama de secuencias podemos observar un ejemplo de aplicación de los Enumeration.

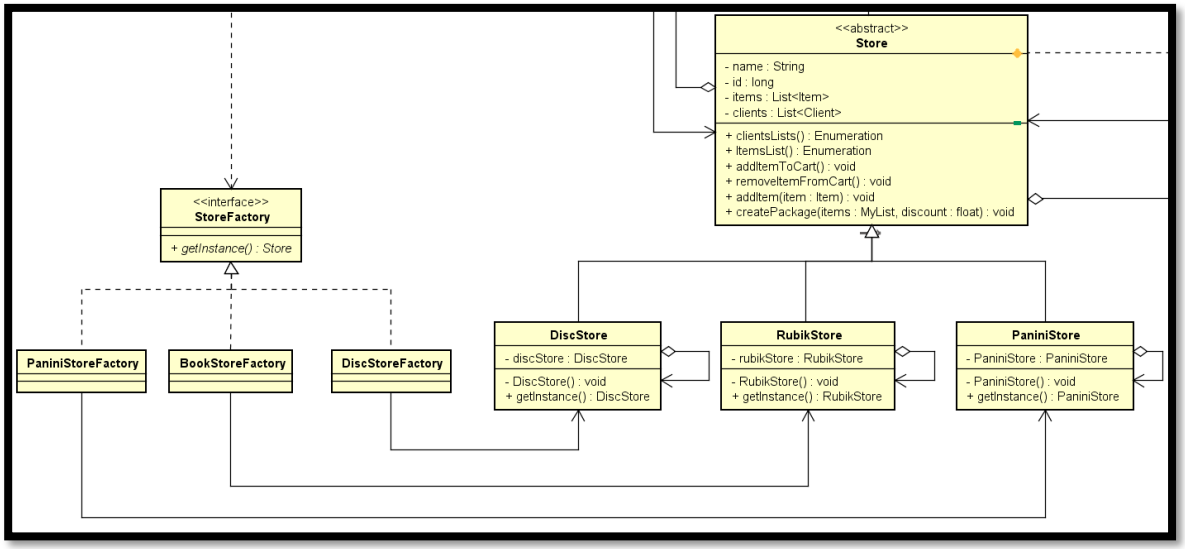
En este caso, la vista solicita la lista de ítems del carrito del cliente, para poder mostrarlo.

Cuando la obtiene, utiliza un ciclo para ir mostrando uno por uno. Podemos observar que la vista únicamente hace uso de las dos funciones de la clase Enumeration, hasMoreElements() y nextElement().

Abstract Factory

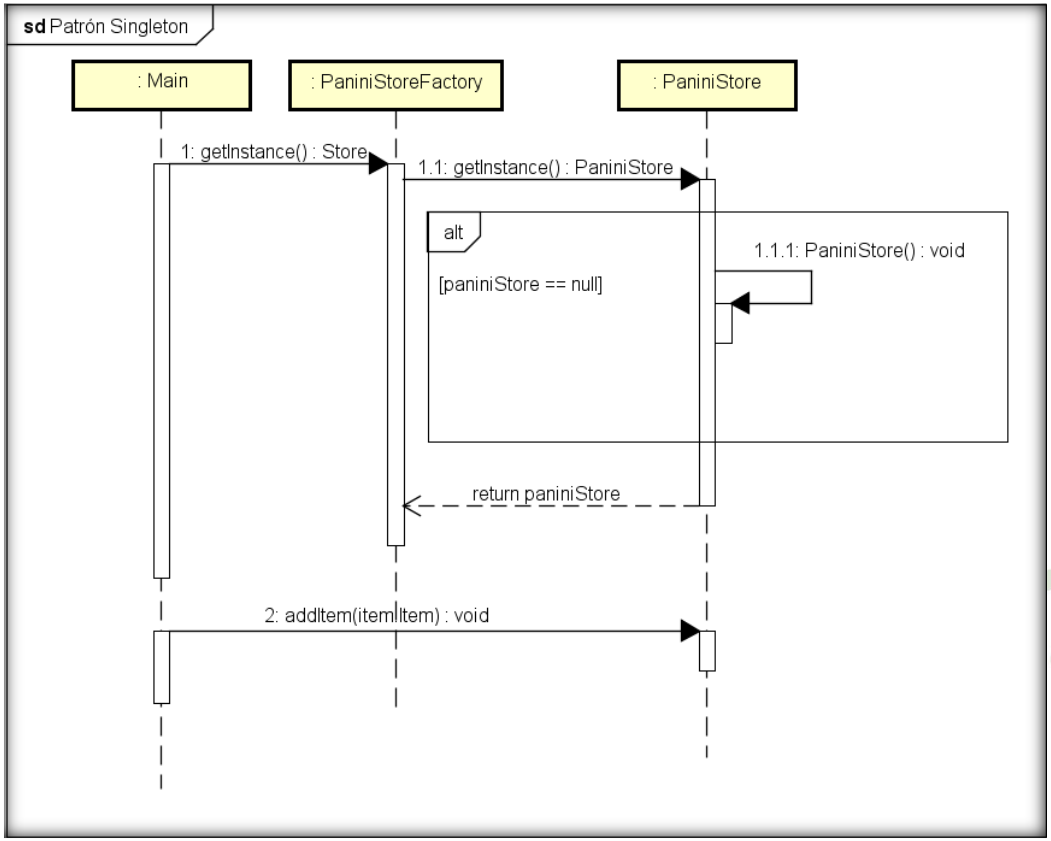


En el diagrama de secuencia se muestra el ejemplo de la creación de una tienda de discos, una de las 3 tiendas del programa. Desde el Main se crea la clase *DiscFactory* que implementa la interfaz *StoreFactory*, se retorna un objeto de tipo *DiscStore* por medio de la función de la interfaz, es utilizado desde la clase Main para agregar los ítems que venderá, además permite crear los paquetes de esa tienda para el patrón *decorator*.



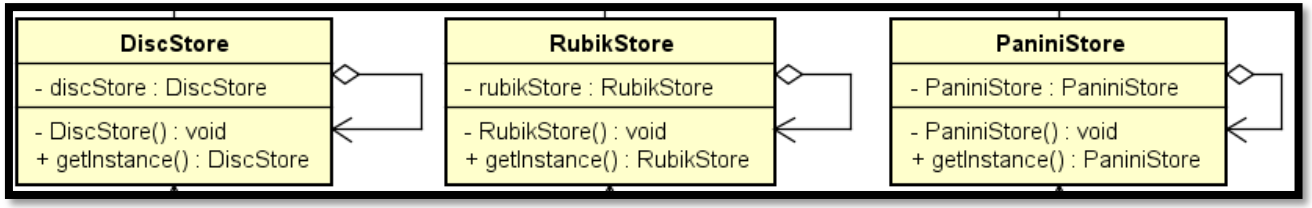
Utilizamos la interfaz StoreFactory que es implementada por las fábricas de cada una de las 3 tiendas del programa. Cada fabrica retorna la instancia correspondiente a la tienda. Los atributos y métodos están definidos por la clase abstracta *Store*, de la que heredan cada una de las tiendas.

Singleton

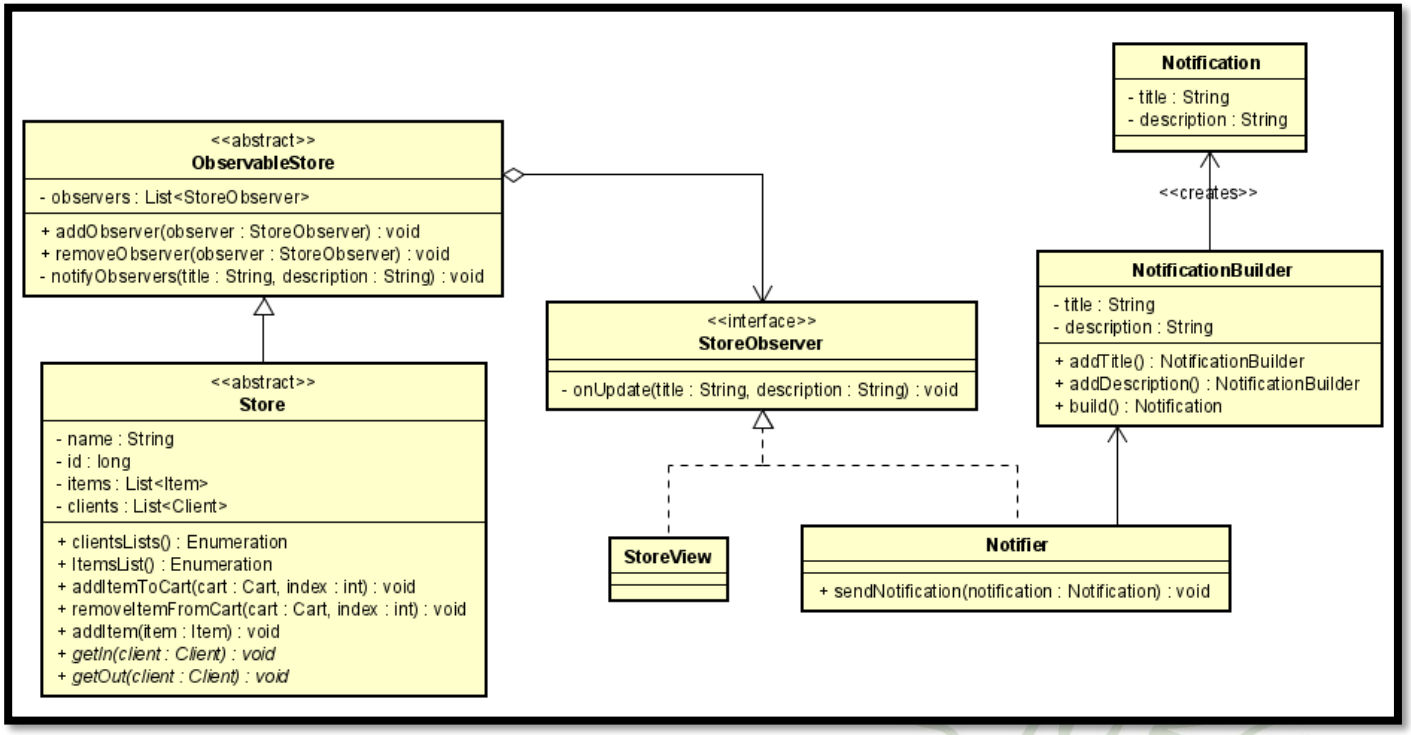


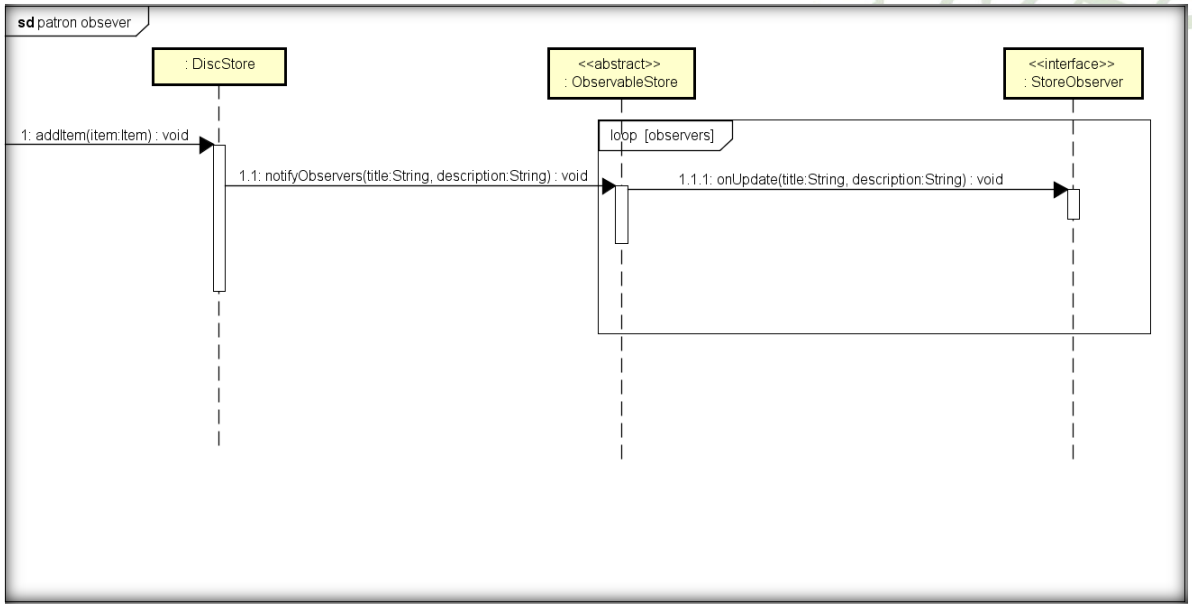
Tomando como ejemplo la clase *PaniniStore*, el *Main* utiliza la clase *PaniniStoreFactory* para generar la instancia de la tienda. La fábrica implementa el método getInstance() que viene de la interfaz StoreFactory, de este modo se llama al método *getInstance()* propio de la clase *PaniniStore*. Si la instancia no ha sido generada, se llama al constructor de la clase y posteriormente se retorna el objeto. Si ya fue creada simplemente se retorna la instancia.

El patrón se encuentra contenido dentro del *Abstract Factory*, cada una de las 3 tiendas devuelve la misma instancia de su clase al utilizar las fábricas. La interfaz *StoreFactory* permite que las fábricas implementen un método que llama a las funciones propias del patrón *singleton* contenidas en las tiendas. De esta forma no es necesario definir el tipo de tienda a crear, siempre que se usen las fábricas se devolverá la misma instancia de la tienda.



Observer





Para este patrón, hicimos que Store extienda de la clase abstracta ObservableStore.

Todas las tiendas extienden de la clase Store, lo cual nos permite poder añadir tiendas fácilmente.

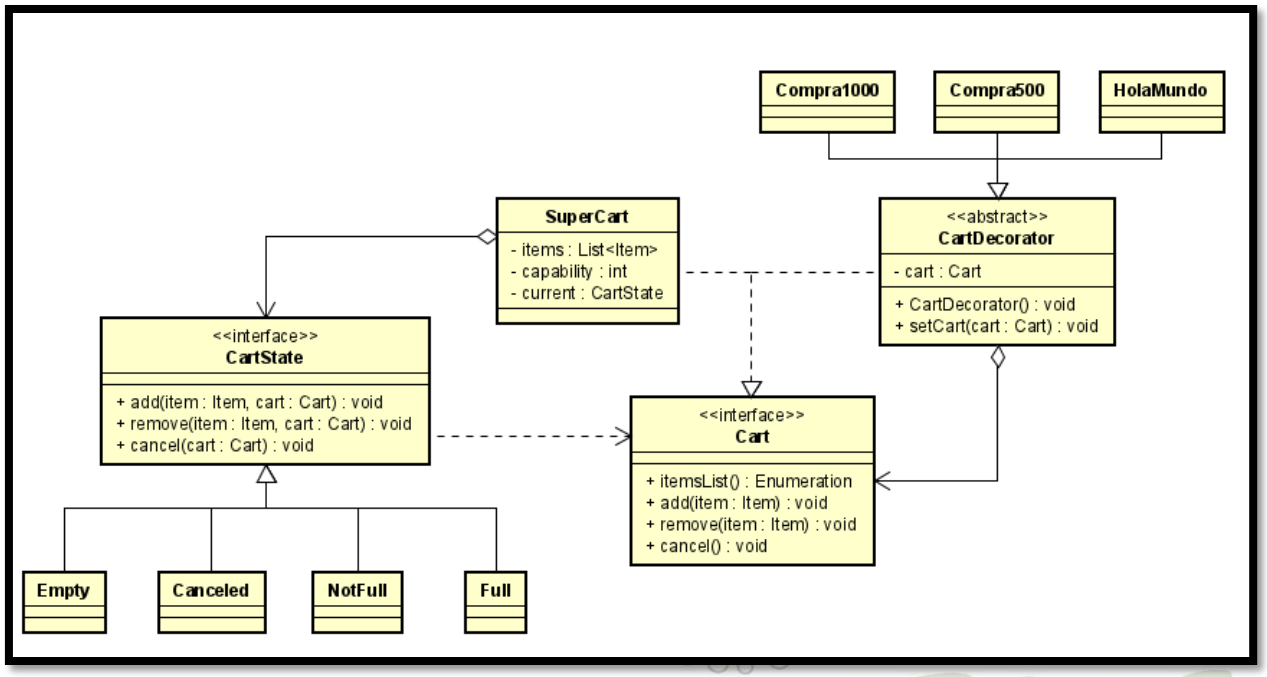
Para los observers, tenemos la interface StoreObserver. Las clases StoreView y Notifier implementan esta interface, lo que significa que pueden ser observadores.

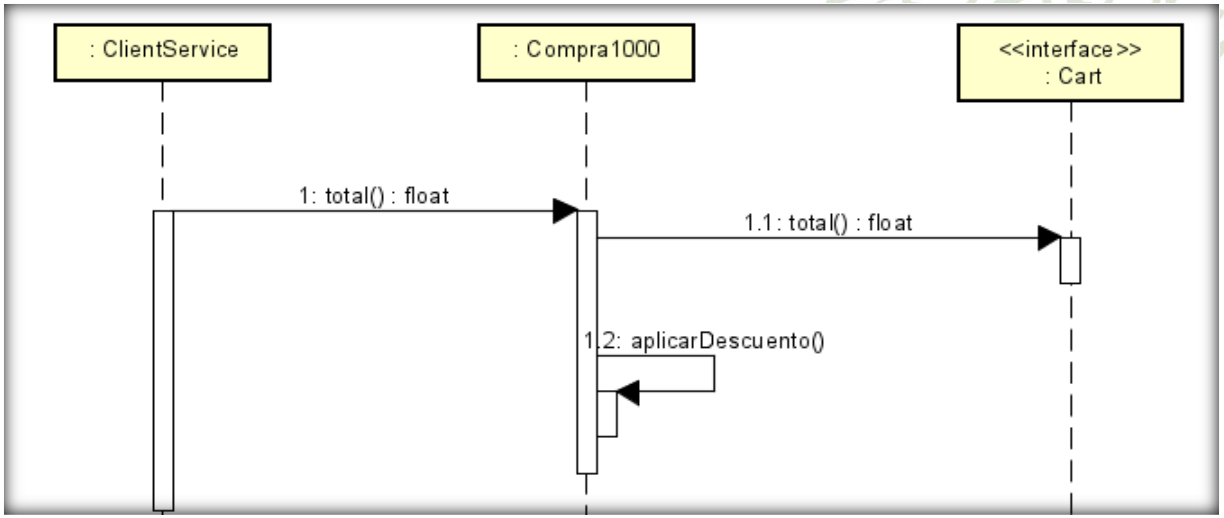
En el caso de StoreView se vuelve observador porque cuando se agrega un producto a una tienda, StoreView es capaz de saberlo y actualizar la lista de productos en la vista.

En el caso de Notifier, su trabajo es enviar notificaciones a la vista cada vez que se agrega un producto. En el diagrama también podemos observar el patrón builder, el cual implementamos para construir notificaciones.

En el diagrama de secuencias se observa como se hace el llamado a todos los observers cada vez que llega un producto nuevo.

Decorator





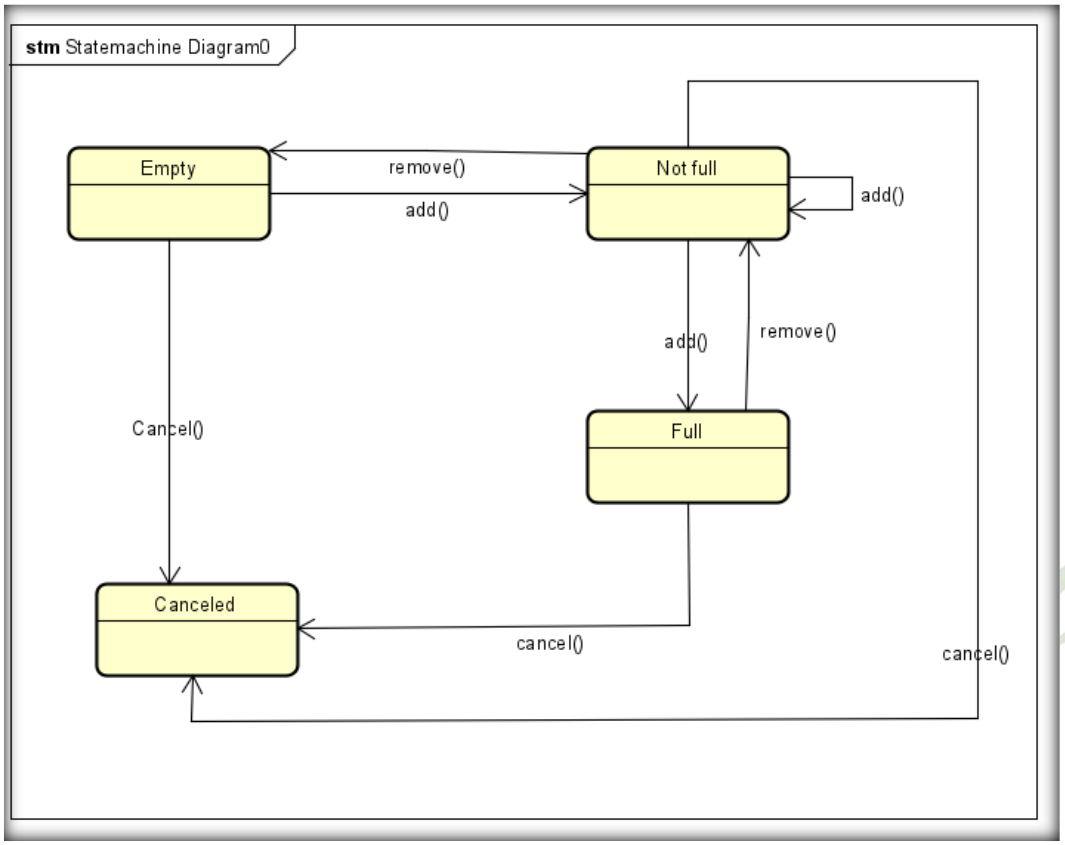
En el diagrama de clases podemos observar tanto el patrón decorator, como el de estados, del cual hablaremos más adelante.

Para nuestra implementación, los decoradores representan cupones, los cuales aplican un descuento al total del carrito, solo si se cumplen ciertos requisitos.

Cuando el cliente ingresa un cupón, este llega a una clase encargada de aplicar copones, desde la cual se intercambia el carrito del cliente por un CartDecorator. Esto funciona pues CartDecorator es también un Cart, como se puede observar en el diagrama. Y también porque el cliente solo conoce la interface Car y no la implementación concreta.

En el diagrama de secuencias podemos observar como el ClientService solicita el total al carrito del cliente, el cual en este caso es una implementación de CartDecorator y, por lo tanto, solicita el total al carrito del cual está compuesto (al que decora) y luego aplica un descuento antes de devolverlo al ClientService.

Estado



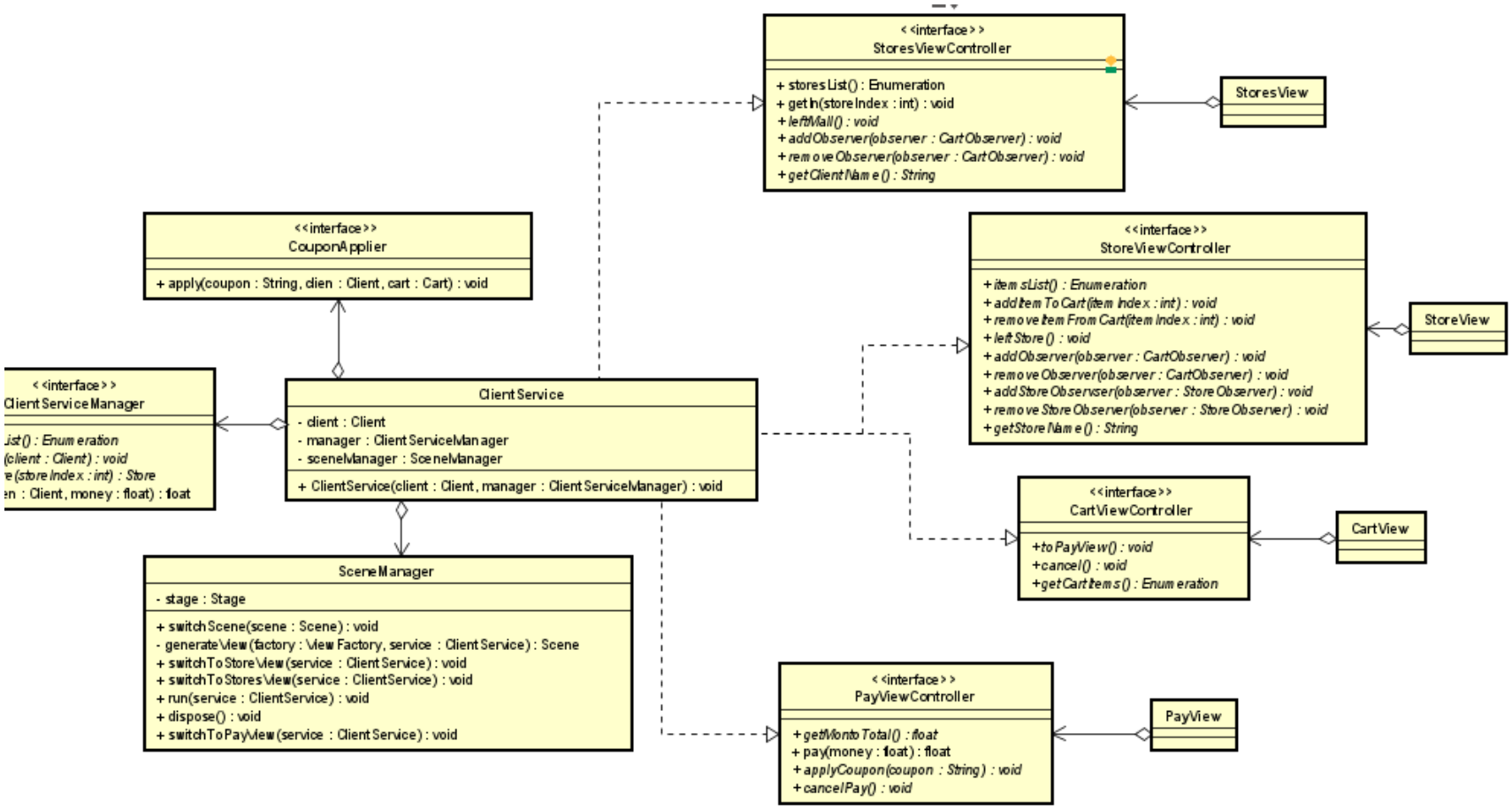
El diagrama de la izquierda nos muestra los cuatro estados del carrito de la tienda “CovidStore”

- Empty: Que es el estado inicial de carrito, y es cuando no tiene ningún ítem dentro, la única manera de acceder a él es al inicio, y cuando el carro se encuentra en estado “Not full” teniendo únicamente un ítem, y se usa la fusión remove(). Otro estado que se puede acceder estando en estado empty es el cancelado.
- Not Full: El estado está parcialmente lleno, y se puede añadir ítems una y otra vez hasta que se alcance el límite, y el estado se convierta a full.
- Full: El carro está completamente lleno y la única función que se puede realizar desde ese estado es cancel, y pasará a estado canceled y utilizando remove, pasará en Not full.
- Canceled: Es un estado accesible por todos los estados usando cancel.

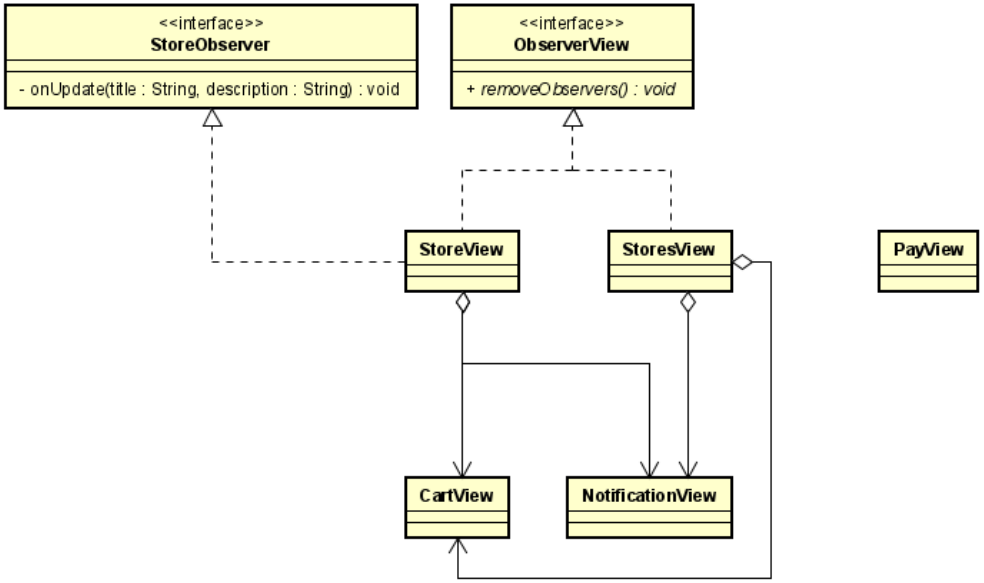
El diagrama de clases de este patrón se puede observar junto con el Decorator en la página anterior.

Diagramas de Clases

Como un diagrama de clases completo no se podría apreciar bien en el documento, dejamos aquí las partes más relevantes. En la carpeta del proyecto se puede encontrar un archivo de Astah, donde pueden verse los estos diagramas, y el diagrama de clases unificado.

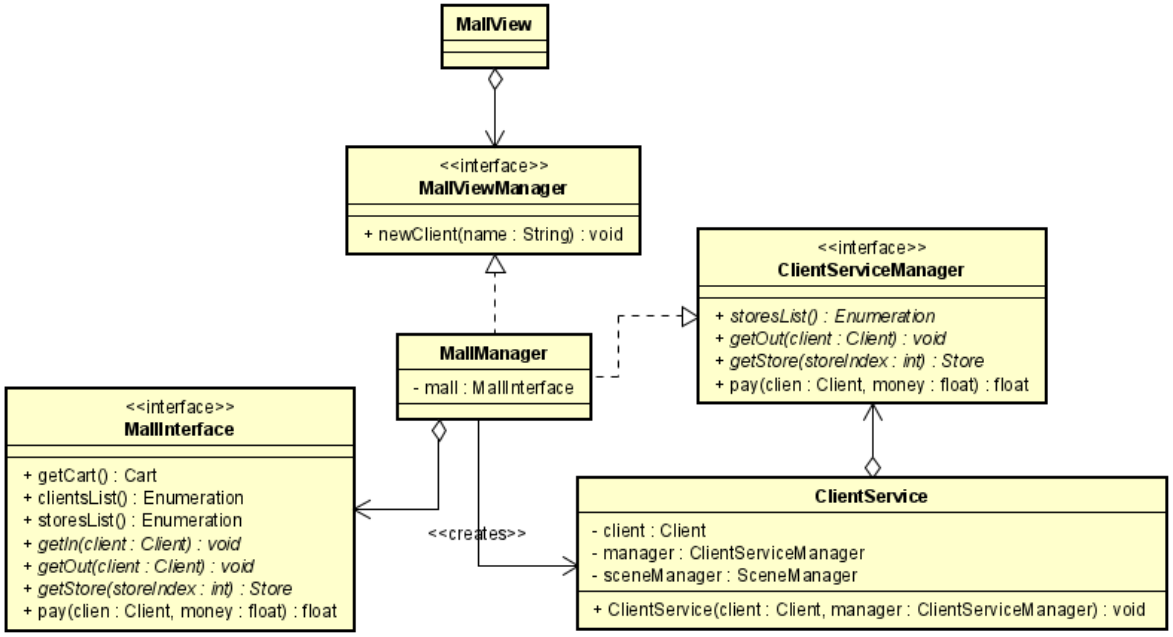


En este diagrama de puede observar la clase ClientService, la cual es la encarga de proveer todos los servicios necesarios al cliente durante su estancia en el centro comercial. Para lograrlo, se apoya de dos interfaces y de la clase SceneManager, que se encarga de hacer cambios de vistas para el cliente. Aquí se puede ver perfectamente dos principios SOLID, como lo son la segregación de interfaces y la inversión de dependencias. Segregación, porque las vistas conviven con ClientService, peor lo hacen a través de interfaces, las cuales solo contienen los métodos que se van a utilizar en cada vista. Inversión, porque el ClientService se tiene que adaptar a las vistas, que como vimos en clases, son las que tiene mayor nivel de abstracción.



En este diagrama podemos ver un poco la composición de las vistas. En este caso, tanto StoreView y StoresView se componen de CartView y NotificationView. Este es porque son el cliente debe ser capaz de observar su carrito y leer las notificaciones siempre que se mantenga dentro del centro comercial.

Igualmente, CartView y NotificationView son componentes independientes de las otras vistas, estas clases se puede reutilizar en alguna otra vista. Esto permite que, por ejemplo, si queremos rediseñar CartView, el cambio se vería reflejado en todas las vistas sin necesidad de modificarlas.



Por último, tenemos al MallManager, este se segrega en dos interfaces, una para servir al ClientService y otra para MallView. El trabajo de MallManager es justamente manejar una Mall, gracias a que utiliza una interfaz, podría controlar servir para cualquier otro Mall que implemente MallInterface, esto lo hace abierto al cambio.

Cuando un cliente quiere acceder a centro, desde MallView, MallManager genera le otorga un ClientService y, desde ahí, ClientService se encarga de él. A su vez, ClientService puede usar algunos métodos MallManager a través de la interface ClientServiceManager, como pay() o getOut(), para el momento que el cliente quiera pagar o salir, respectivamente.