UT5 TA5 – Grupo 1

Ejercicio 0.

En el patrón chain of responsability se aplica SRP, ya que cada una de las clases de la cadena tiene una sola responsabilidad, y en caso de no poder procesar la solicitud, la pasa al siguiente handler.

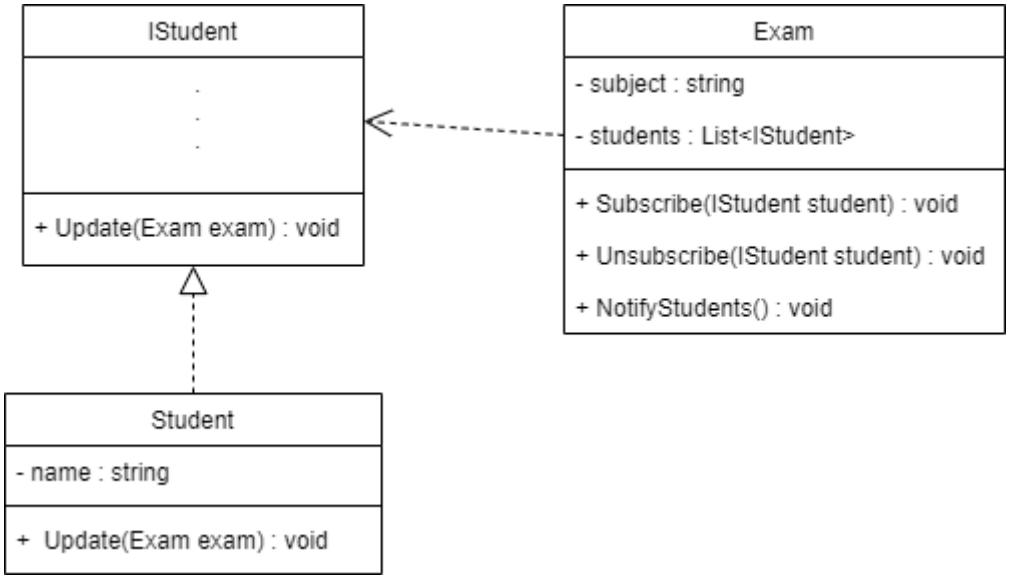
También se respeta OCP ya que se pueden agregar más handlers a la cadena, sin tener que modificar los ya existentes, y se aplica LSP ya que los handlers son fácilmente intercambiables entre sí, sin provocar cambios en el comportamiento del programa.

Finalmente, se respeta DIP ya que normalmente la estructura de los handlers es definida en interfaces, lo que significa que las implementaciones de cada manejador son independientes de la estructura de la cadena.

Por otra parte, puede que en algunos casos no se respete ISP, a veces tener una única interfaz para todos los handlers es demasiado general y quedan métodos en algunos handlers sin implementar. Esto es resuelto haciendo interfaces más específicas cuando sea necesario.

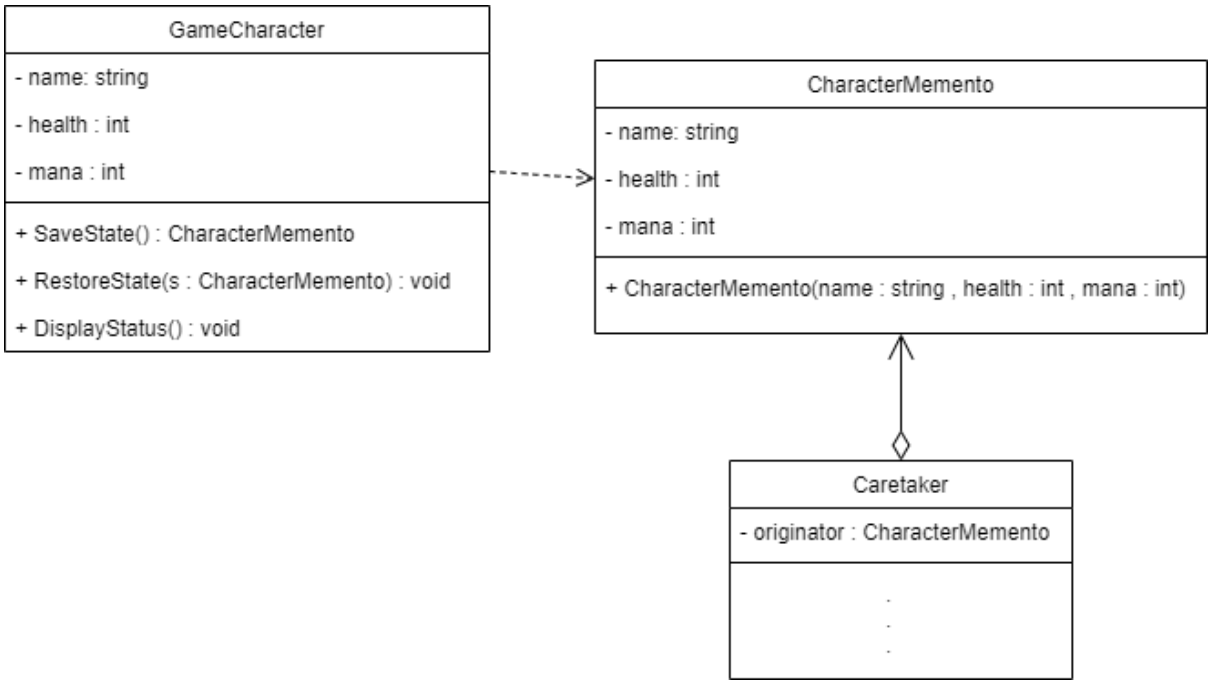
Ejercicio 1.

Seleccionamos el patrón de diseño *Observer* ya que, este patrón nos permite gestionar una mayor cantidad de estudiantes y suscripciones a diferentes exámenes, lo que lo hace escalable. Este patrón de diseño facilita que un objeto del tipo "*Exam*" notifique a objetos del tipo "*Student*" sobre cambios en su estado sin crear un acoplamiento fuerte entre ellos.



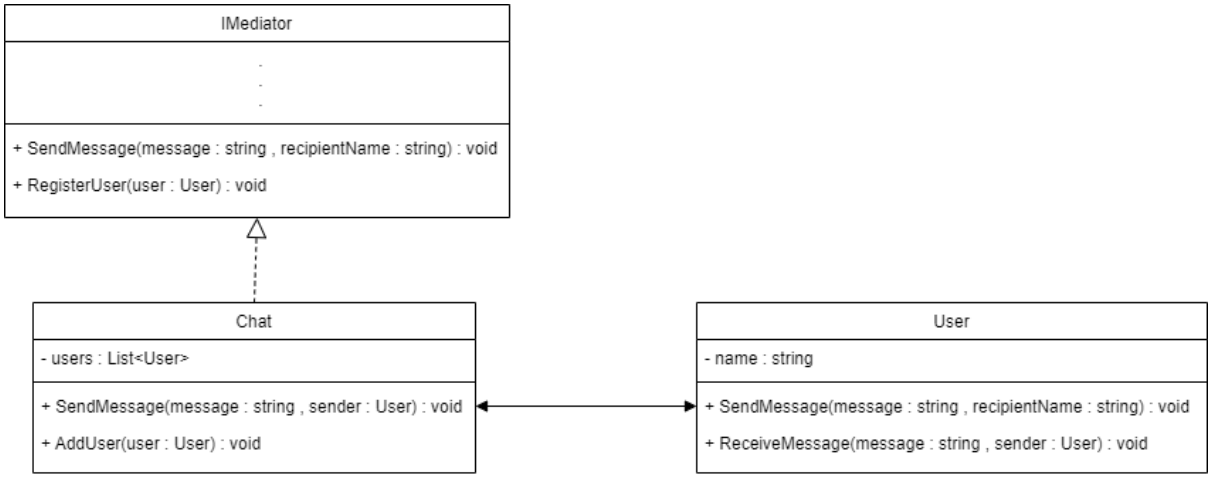
Ejercicio 2.

Implementamos el patrón de diseño *Memento* ya que, a través de este patrón nos permite guardar el estado actual del personaje (health y mana) en un momento específico y más tarde, tenemos la posibilidad de restaurar dicho estado.



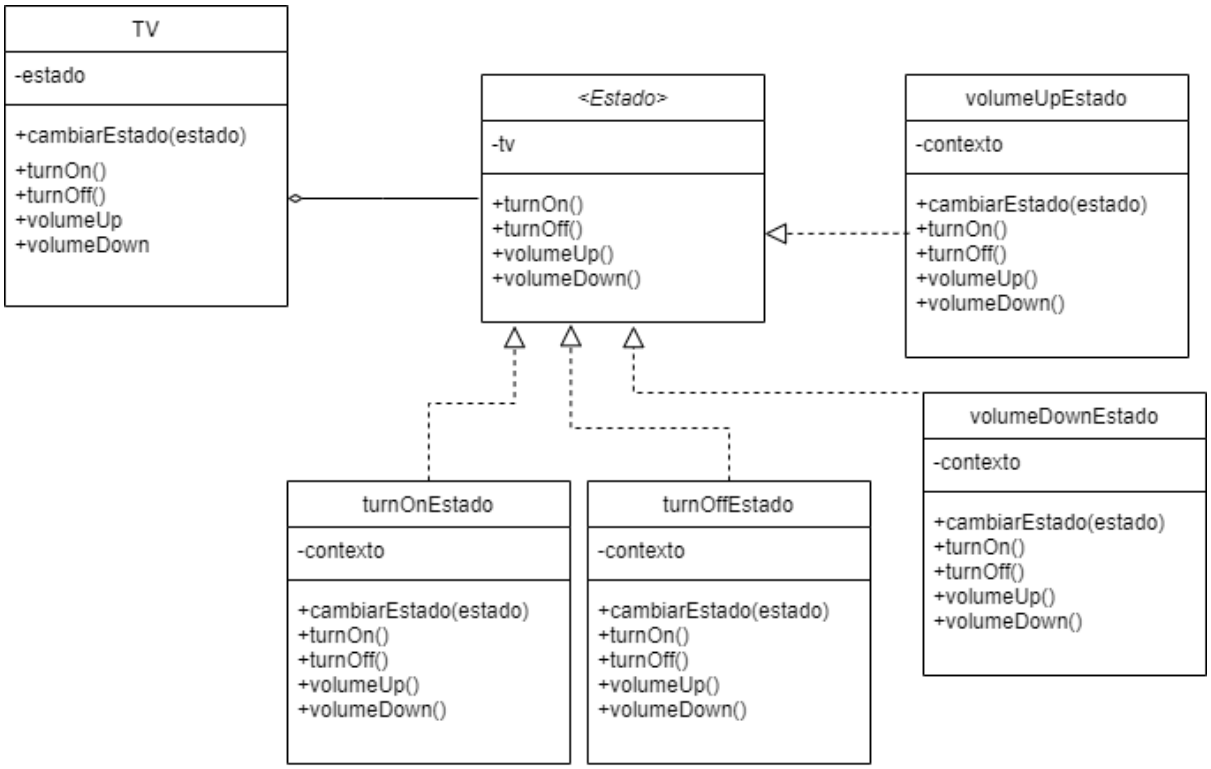
Ejercicio 3.

Al observar que existe una comunicación directa y bidireccional entre dos usuarios a través de simples mensajes, seleccionamos el patrón de diseño *Mediator* con el fin de proponer una solución escalable para casos en que existan más de dos usuarios intercambiando mensajes haciendo uso de una lista de tipo "*User*" donde la misma incorporará todos los usuarios a los que se desea enviar un mensaje determinado a excepción del remitente.



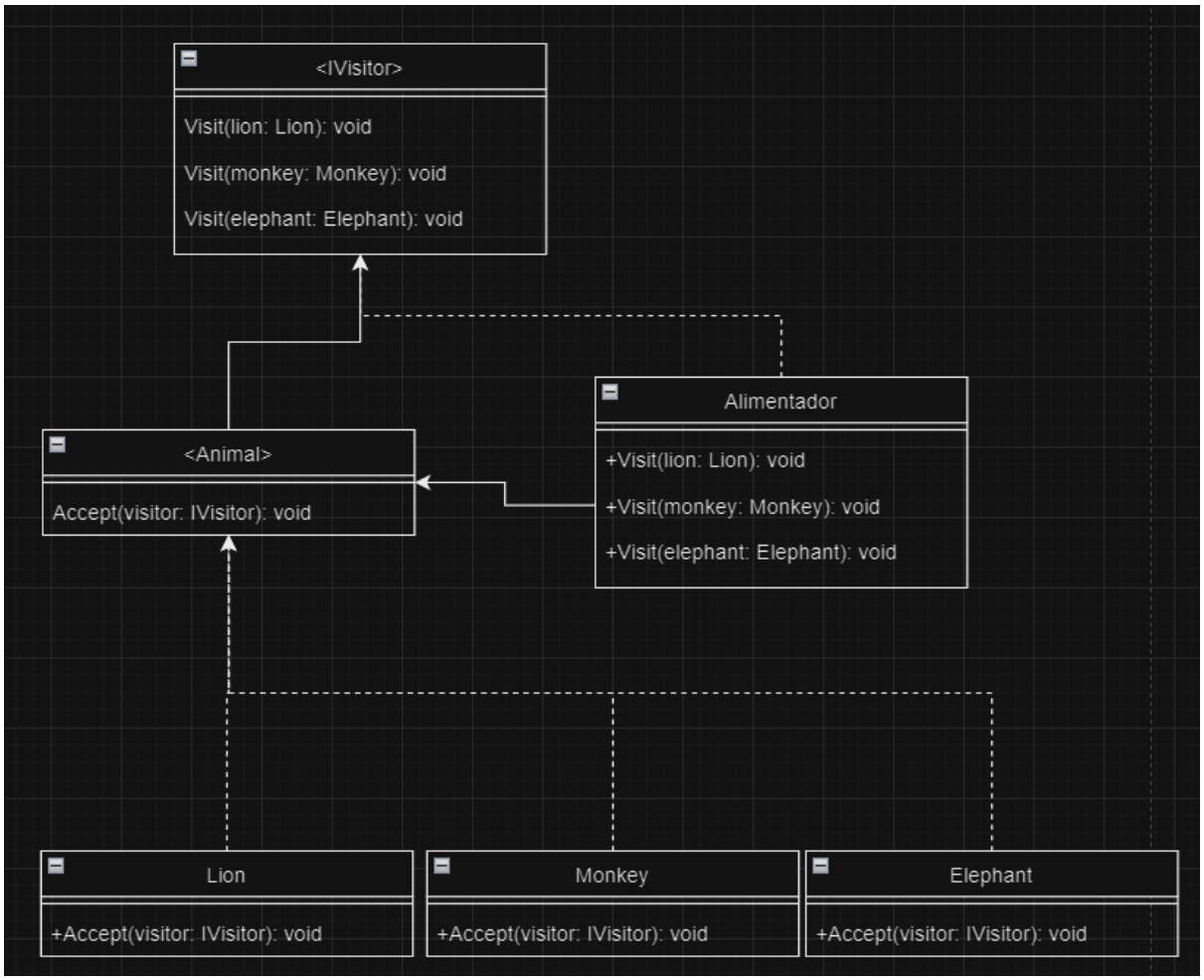
Ejercicio 4.

El patrón que utilizamos para solucionar este caso es el State patrón. Como había varios Switch Case pensamos que lo mejor era separar esa logica en diferentes clases que representan cada uno de los estados, los cuales utilizan la misma interfaz.



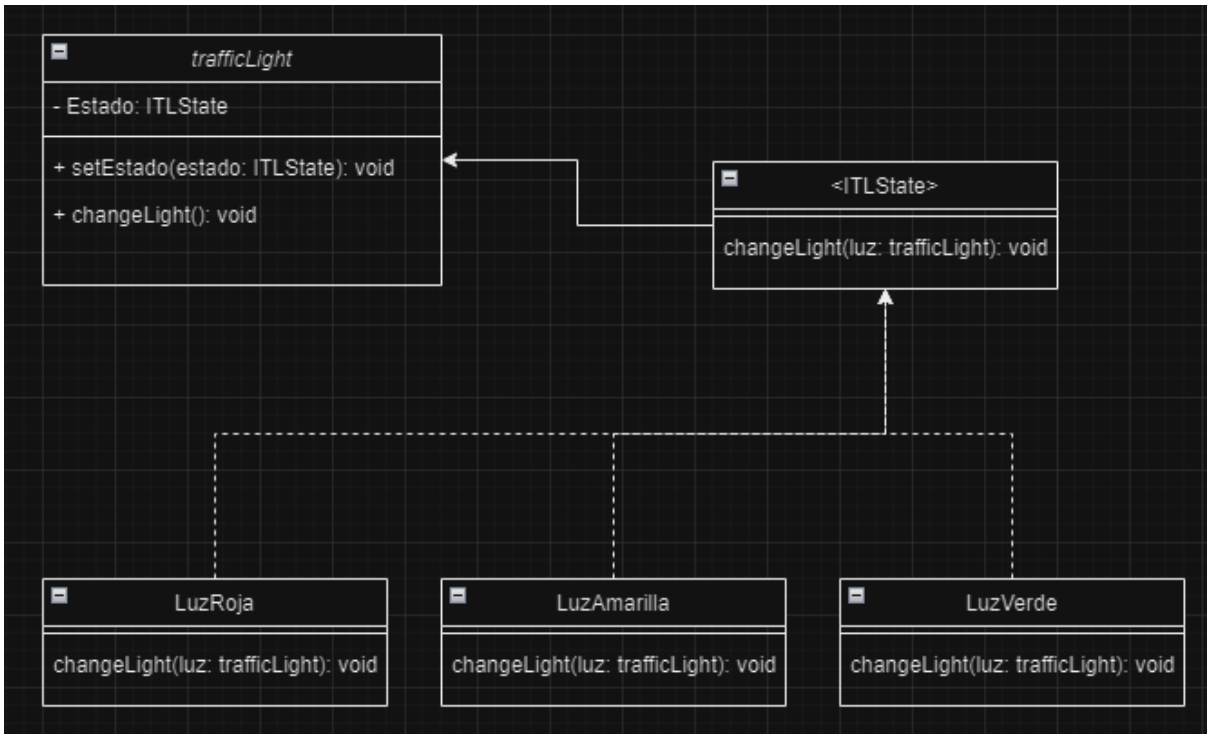
Ejercicio 5.

Se puede resolver utilizando el patrón Visitor, cada Animal tiene un método Accept que acepta un IVisitor, y en vez de contener la lógica de alimentar al animal, llama al método visit del IVisitor que recibió como parámetro. En el caso de que se quiera agregar más operaciones, se haría otro visitor con la operación que se quiera hacer.



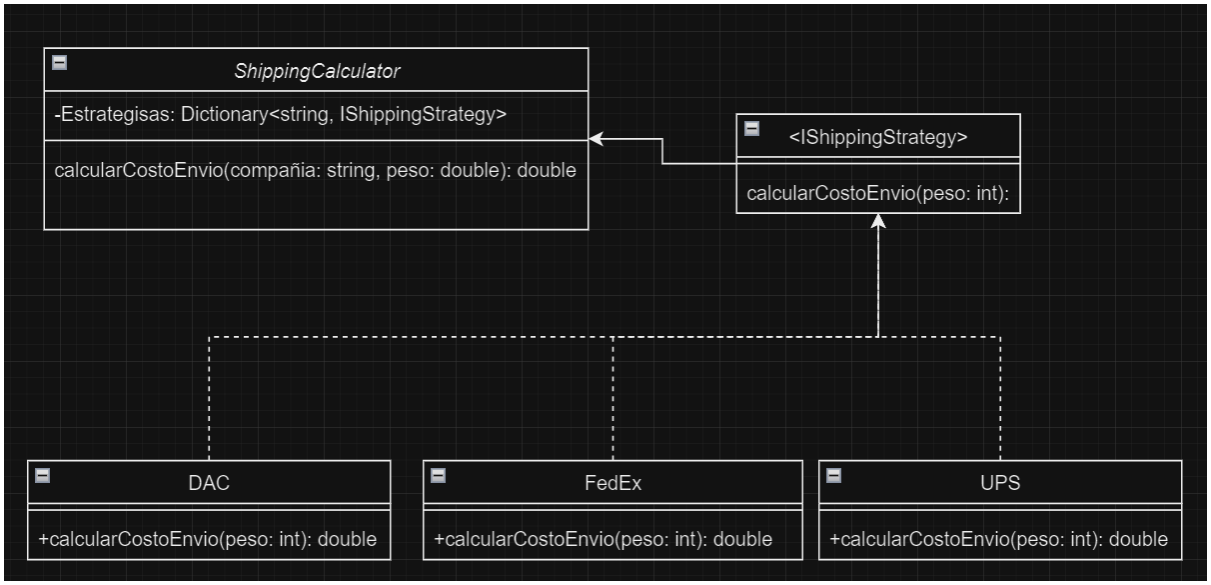
Ejercicio 6.

Este ejercicio puede ser resuelto con el patrón State. En vez de tener un switch case con un caso para cada color de luz, se crea una interfaz TLState y cada estado del semáforo implementa esta interfaz



Ejercicio 7.

Para este ejercicio, se puede usar el patrón Strategy, tendríamos una interfaz IShippingStrategy que es implementada por cada una de las compañías de envíos, y un diccionario que tenga como clave el nombre de cada compañía, y de valor un objeto de tipo IShippingStrategy. Cuando se quiere calcular el costo de envío se usa el método calcularCostoEnvio del objeto que está en el índice del diccionario que contiene el nombre de la compañía.



Ejercicio 8.

Para este caso el patrón que más se ajusta es el patrón Factory, ya que se necesitan instanciar objetos relacionados sin tener que especificar la clase exacta, por ejemplo, `EmailService` realiza dos tareas diferentes, envía un email regular y envía un newsletter.

En vez de tener métodos separados en `EmailService` para cada tipo de los envíos mencionados anteriormente, podemos usar un factory para crear instancias de diferentes tipos de servicios de correo.

Ejercicio 9.

En este caso el patrón factory puede ser utilizado para crear instancias de diferentes niveles de soporte (`SupportHandler`) basadas en el nivel de la solicitud.

De esta forma se proporciona una forma flexible de agregar nuevos niveles de soporte sin modificar el sistema central de soporte, mejorando la mantenibilidad y extensibilidad.

Ejercicio 10.

Para este caso, el patrón Prototype pude ser útil al momento de crear nuevas instancias de objetos configurados previamente mediante la clonación de una instancia, en este caso es adecuado cuando se manejan configuraciones complejas o para optimizar instancias, también se pueden crear prototipos de saludos para cada nacionalidad y clonarlos cuando sea necesario usarlos, eliminando la necesidad de condicionales en `GreetingSystem`.