UT5 TA4 – Grupo 1

Ejercicio 0.

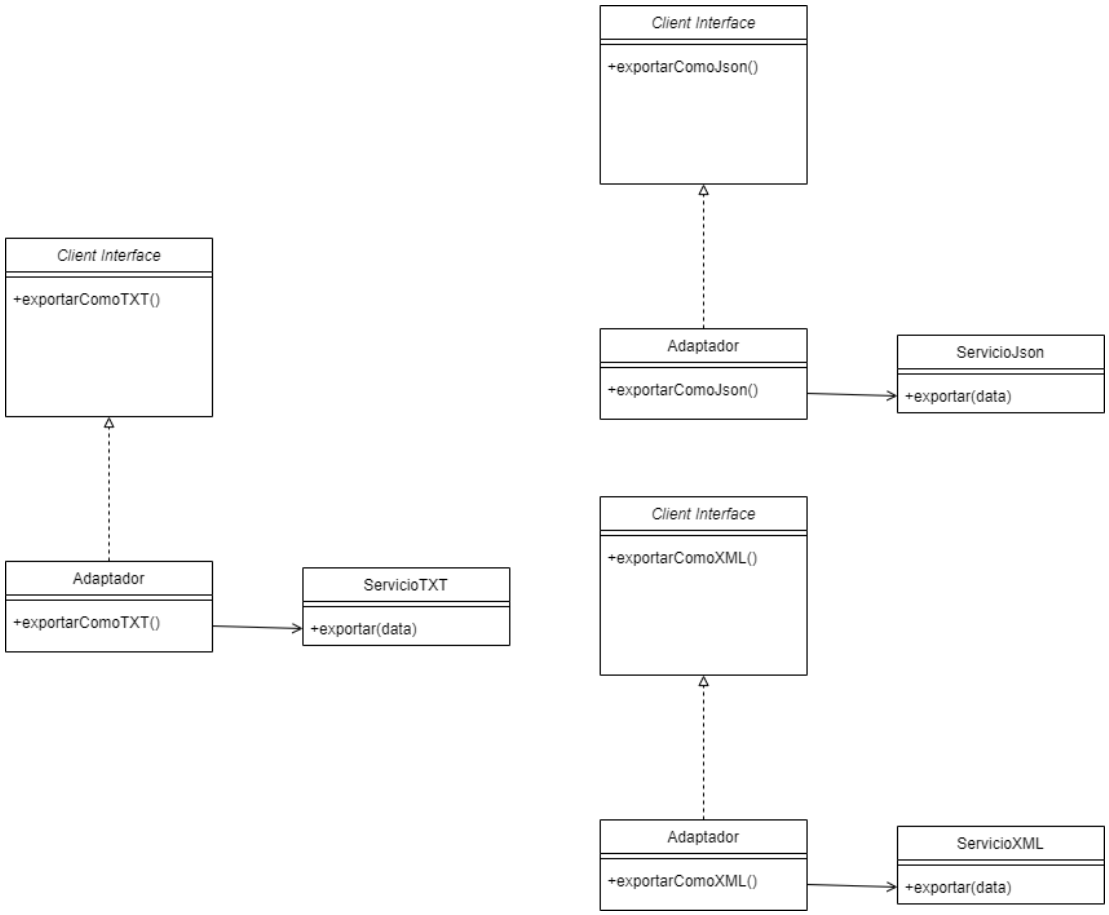
Decorator aplica los siguientes principios SOLID:

- OCP: Cumple con OCP ya que está abierto a la extensión, agregando más funcionalidades.
- SRP: Ya que cada subclase de Decorator adiciona una única responsabilidad, y esta no afecta la implementación interna del elemento decorado ni de otros decoradores.

No incumple ningún principio.

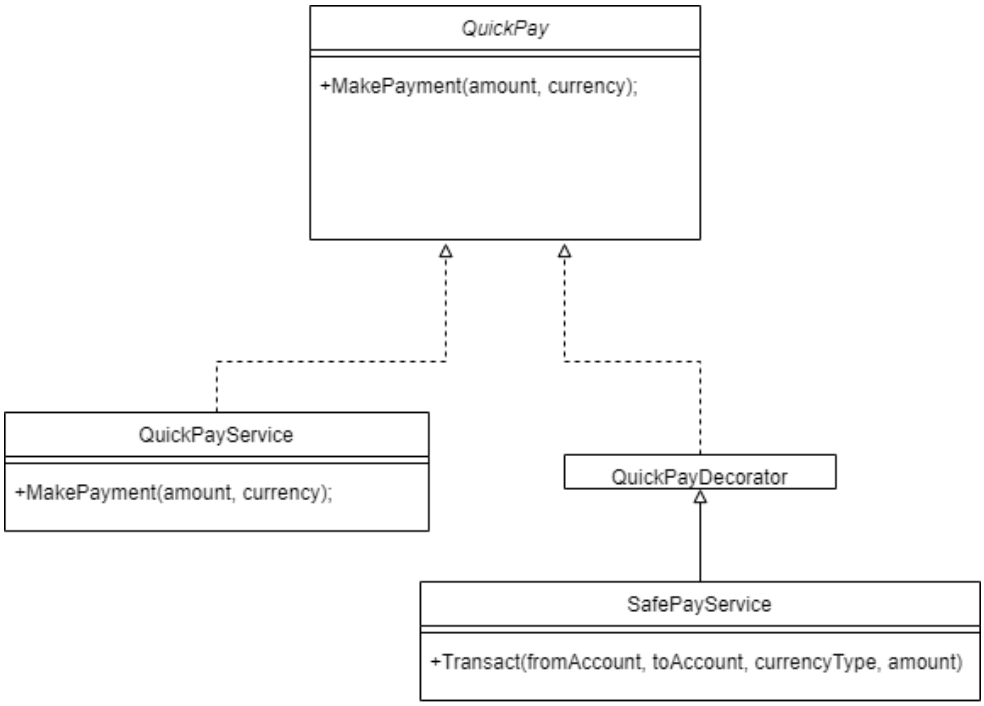
Ejercicio 1.

El patrón que se debe utilizar es el de “Adapter”. Se realiza un adaptador para cada tipo de archivo. Cada interfaz de cliente puede llegar a recibir dos tipos de archivos, y lo termina exportando en el tipo del Servicio. Esto lo logra hacer porque lo entiende gracias al adaptador.



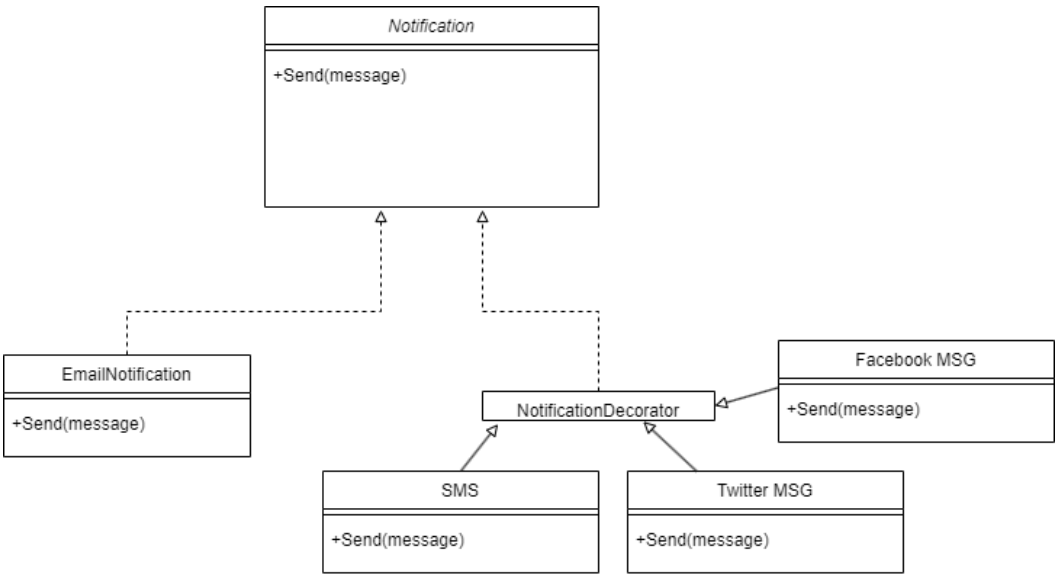
Ejercicio 2.

Para agregar la funcionalidad de SafePay sin tocar el código ya existente, se debe utilizar el patrón Decorator.



Ejercicio 3.

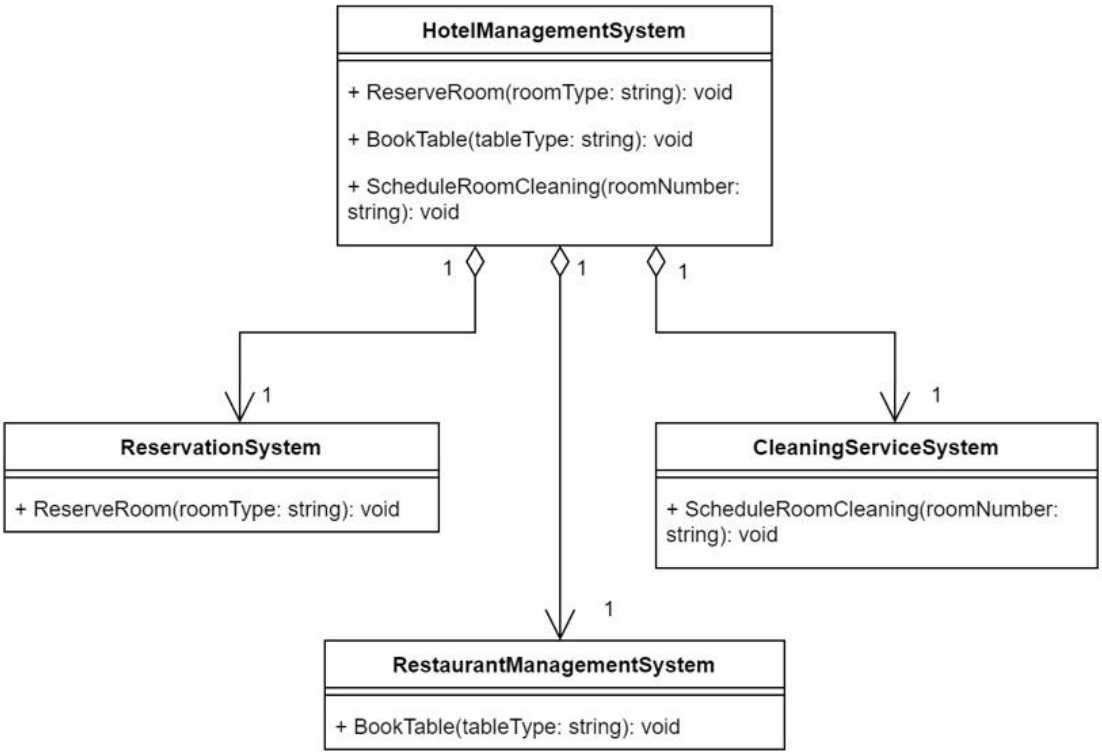
Al igual que el caso anterior, se puede utilizar el Decorator. Por lo que da a entender la letra, la notificación por Email ya existe, y se busca que el programa pueda soportar algunos tipos más de envío de notificación, por lo tanto, optamos por este patrón.



Ejercicio 4.

Patrón a aplicar: Facade

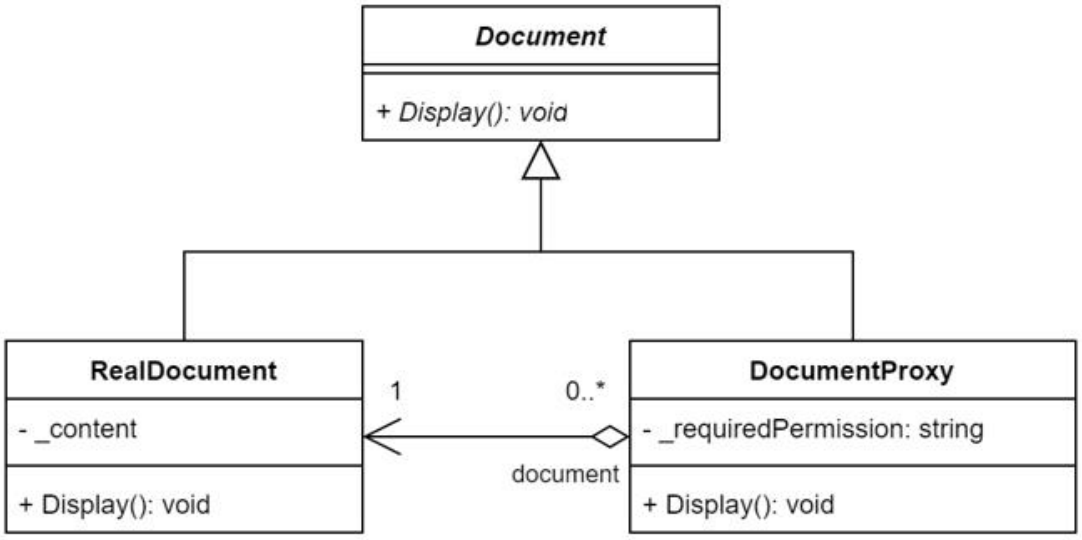
A partir del enunciado, se entiende que la interfaz de cada subsistema es compleja, por lo que puede resultar deseable aplicar el patrón Facade. Además, una clase como HotelManagementSystem puede simplificar el manejo de los subsistemas como recursos, evitando su instanciación indiscriminada en el código cliente.



Ejercicio 5.

Patrón a aplicar: Proxy

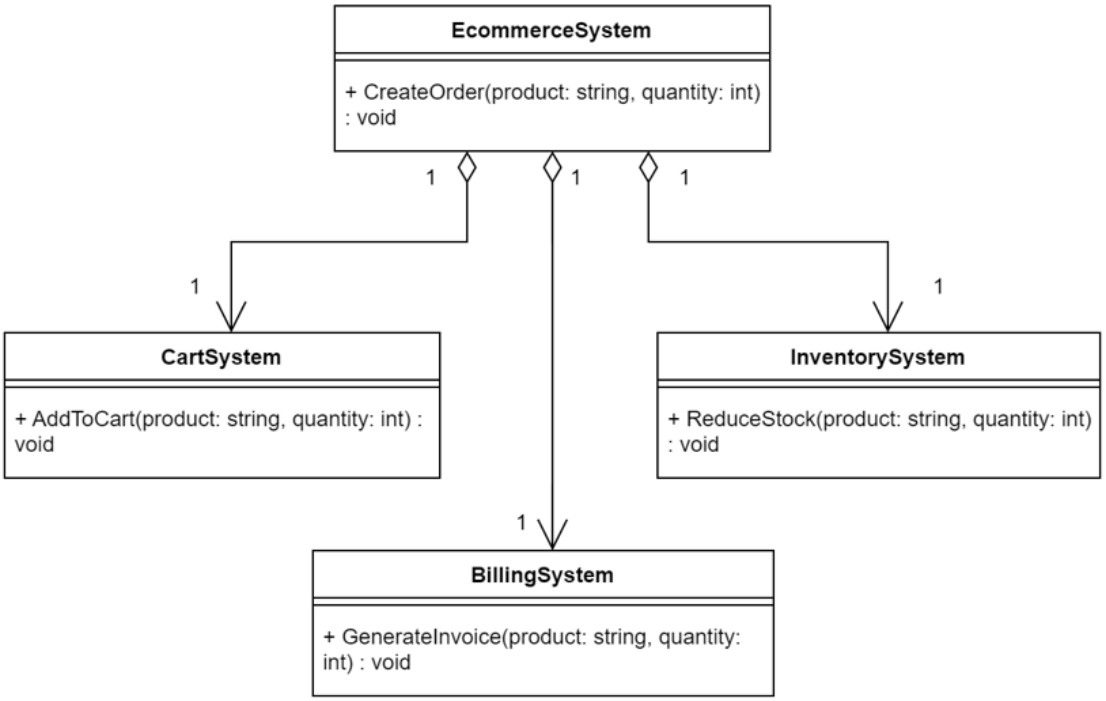
Una forma óptima de agregar capacidades de control de acceso a la clase RealDocument sin modificar su implementación es utilizando un Proxy del mismo. Ambos implementan la interfaz Document, pero el uso de un Proxy permite interceptar los mensajes enviados a RealDocument (ej. invocaciones de Display) y aplicar la lógica necesaria para el control de acceso al documento antes de reenviar el mensaje a RealDocument.



Ejercicio 6.

Patrón a aplicar: Facade

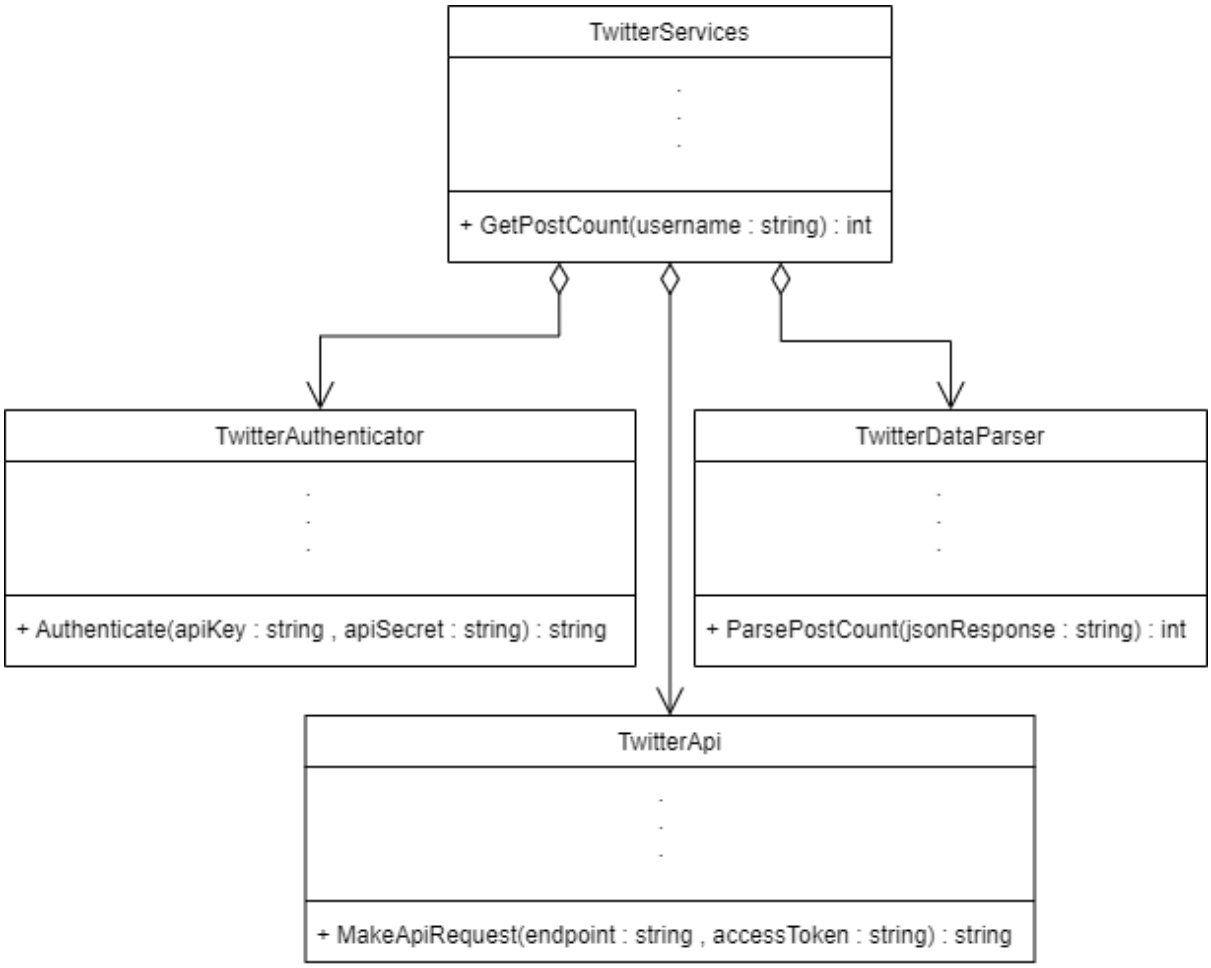
A partir de la interfaz de cada subsistema, es evidente que algunos de los métodos invocables están relacionados en cuanto a casos de uso de negocio. El uso de una clase Facade permite encapsular la lógica de ese caso de uso en un único método, en vez de forzar al código cliente a enviar mensajes a distintos subsistemas en un orden específico para respetar reglas de negocio. Como consecuencia, también reduce el acoplamiento del código cliente a los diferentes subsistemas, ya que (dependiendo del caso) la misma clase Facade puede mitigar los potenciales cambios en las interfaces de los subsistemas.



Ejercicio 7.

Consideramos implementar el patrón de diseño *Facade* donde utilizando la clase denominada "*TwitterServices*" como clase "fachada", nos permite reducir favorablemente el acoplamiento del código cliente a los distintos subsistemas ya que, el antes mencionado no necesita conocer los detalles de las clases "*TwitterAuthenticator*", "*TwitterDataParser*" y "*TwitterApi*" que son utilizadas para retornar el resultado final del servicio que consume el cliente.

A su vez, implementado este patrón nos ayuda a ocultar la complejidad de la implementación de los distintos métodos que son utilizados para retornar el resultado final al cliente sin afectar la lógica de negocio.



Ejercicio 8.

Consideramos implementar el patrón de diseño *Decorator* debido a que, interpretamos que contamos con un texto (objeto) al que debemos tener la posibilidad de decorarlo, es decir, realizar distintas modificaciones sobre el mismo.

Utilizando el patrón de diseño *Decorator* nos permite adicionar funcionalidades de manera dinámica ya que, este patrón promueve la responsabilidad única (SRP) al separar la lógica de decoración en clases diferentes.