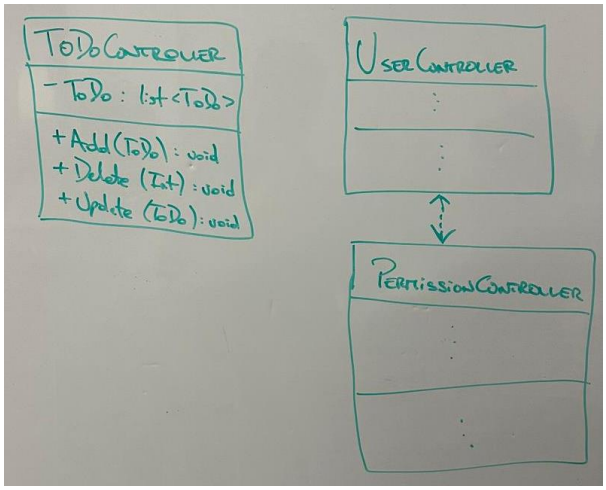
UT5 TA2 – GRUPO 1

Ejercicio 1

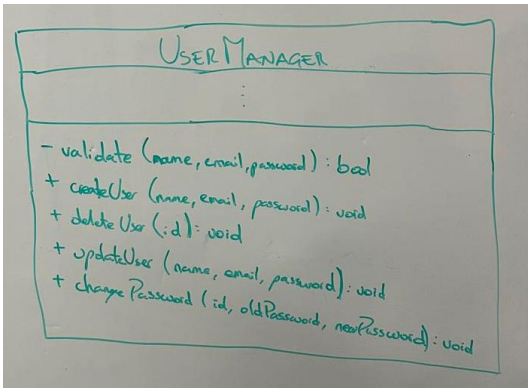
Identificamos el antipatrón **“The Blob”** ya que, una única clase denominada “*TODOController*” carga con una gran variedad de responsabilidades. Es recomendable dividir la clase “*TODOController*” en clases más pequeñas y especificas con el fin de redistribuir responsabilidades.



Ejercicio 2

Identificamos el antipatrón de **Cut and Paste**, ya que se está reutilizando el código de validación en cada una de las funciones que contiene la clase denominada “*UserManager*”. Si es necesario realizar una modificación en el código que realiza la validación, será necesario modificar el código en cada sitio que se utilice ese bloque de código.

Una solución posible es crear e implementar una función que valide lo que sea necesario validar y luego, hacer una llamada a esa función cada vez que sea necesario desde cada función.

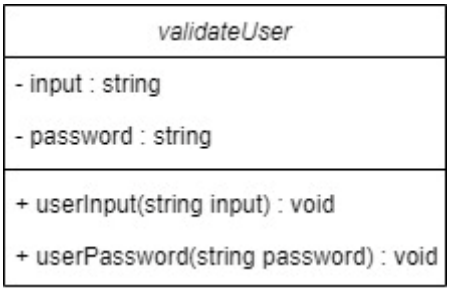


Ejercicio 3

El antipatrón que identificamos en el ejercicio 3 es **Spaghetti Code**, ya que es un bloque de código que carece de una estructura clara, y se ejercen malas prácticas como nombres de variables poco descriptivos.

Ejercicio 4

En el ejercicio 4 identificamos el antipatrón de **Cut and Paste**, ya que se está reutilizando el código de validación de input para la validación de contraseña, esto causa que, para hacer cualquier cambio en la validación, se tenga que cambiar en todos los lugares donde se utiliza este bloque de código



Ejercicio 5

Identificamos el antipatrón “**Lava Flow**” debido a que se encuentra implementada una clase denominada “*OldUnusedClass*” que contiene código obsoleto y sin uso.

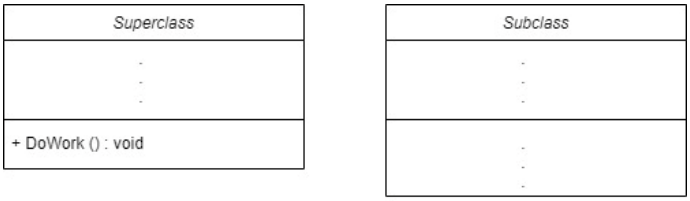
A su vez, podríamos identificar el antipatrón “**The Blob**” ya que, bajo una única clase existe una gran cantidad de código que realiza una única función. Perfectamente se podría dividir las responsabilidades en distintas clases cumpliendo con la modularidad.

De ser así, existe la posibilidad de eliminar dicha clase en conjunto con el código implementado dentro de la misma con el fin de depurar el código del proyecto.

Ejercicio 6

Existe una clase denominada “*Superclass*” y luego encontramos una nueva clase denominada “*Subclass*” que hereda de la clase padre “*Superclass*”, donde la clase “*Subclass*” tiene responsabilidades totalmente distintas a las que realiza la clase “*Superclass*” y realiza un trabajo completamente distinto al trabajo que realiza la clase “*Superclass*”.

Identificamos el antipatrón “**The Blob**” ya que detectamos falta cohesión a nivel de las operaciones de ambas clases, por lo que una posible solución es que ambas clases no tengan relación entre sí, si no comparten ninguna responsabilidad en común.



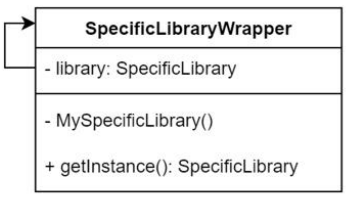
Ejercicio 7

```
public class DataProcessor
{
    // This is a specific library or tool used everywhere
    SpecificLibrary library = new SpecificLibrary();

    public void ProcessData(List<int> data)
    {
        library.Method1(data);
        library.Method2(data);
        library.Method3(data);
    }
}
```

Antipatrón: Cut-and-Paste Programming

La librería/herramienta se instancia en varias partes del código, en vez de usar una sola instancia con un solo punto de acceso global. Se puede corregir creando un wrapper (porque se asume que SpecificLibrary es externa y no modificable) que use el patrón Singleton, aunque esto requiere el compromiso de obtener la instancia de SpecificLibrary a través del wrapper y solo funciona si los métodos son puros y no afectan el estado interno de la instancia.

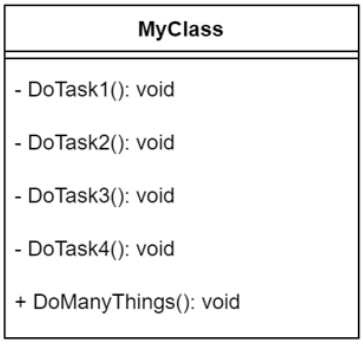


Ejercicio 8

```
public class MyClass
{
    public void DoManyThings()
    {
        // Lot of code for task 1
        // ...
        // Lot of code for task 2
        // ...
        // Lot of code for task 3
        // ...
        // Lot of code for task 4
        // ...
    }
}
```

Antipatrón: The Blob

DoManyThings() realiza tareas distintas en el mismo método. Lo más razonable (sin conocer exactamente dependencias entre una tarea y otra) es extraer el código de cada tarea en un método privado separado.



Ejercicio 9

En el ejercicio 9 identificamos que el antipatron que más de adapta a lo planteado en el código es el Goden Hammer ya que en este caso, se aplica una metodología para hacer los cálculos solicitados pero por ejemplo, aplica un enfoque único, ya que para el cálculo de impuestos esta implementado dentro de una única función con calculo específicos hardcodeados directamente en el método, esto genera una falta de flexibilidad como por ejemplo para añadir nuevos rangos de impuestos y/o modificar los existentes lo cual no sería posible sin tener que modificar el código, este código no está diseñado para adaptarse a cambios de manera fácil, cada cambio implica modificación de código.

Ejercicio 10

En el ejercicio 10 identificamos el anti-patron Cut–and–Paste Programming, ya que por ejemplo el código contiene lógica repetida para manejar el agregado de productos al carrito, así como también el control de cantidad de productos, los cuales se podrían armar en lógica separadas y reutilizarse en lugar de cortarlo y pegarlo varias veces en el mismo código