Principios SOLID

Caso 1 - Email: Violaciones detectadas y soluciones propuestas

- No se estaba respetando el principio de **Single Responsibility** porque la clase ClienteEMail tenía demasiadas responsabilidades, por eso se generaron dos clases distintas, una para gestionar los correos y otra para gestionar las acciones del servidor, y desde el Cliente simplemente se delega el mensaje a las clases particulares para que cada una cumpla con una única función, que es la que le compete.
- Para mejorar y asegurarnos de que se aplique el principio de **Open Closed**, se modificaron las clases para que cada una tenga solo los atributos que le pertenecen y los mismos sean privados, exponiendo solamente los métodos que son necesarios, pero no sus atributos. En resumen, se pensó una solución en la que todas las clases quedaron cerradas a la modificación pero abiertas a la extensión.
- Con respecto al principio de **Liskov Substitution**, se agregó una interfaz más detallada que extiende de una interfaz más general, si reemplazamos en la clase concreta ServidorPop la implementación de la interfaz padre que es IServidor, por la interfaz hija más específica, que es la IServidorAvanzado, el programa sigue funcionando igual, solo que adquiere la posibilidad de llamar a nuevos mensajes, pero no se rompe lo existente.
- En cuanto al principio de **Interface Segregation** en la solución se dividió la interfaz de IServidor en dos, ya que detecté que había métodos que estaban demás y no tenía sentido que la clase concreta implemente una interfaz con mensajes que no utilizaba, tenía más sentido que tenga una interfaz acotada, y en caso de requerirse a futuro por otra clase, se implemente una interfaz de IServidorAvanzada más completa.
- Por último, en cuanto al principio de **Dependency Inversion**, considero que era necesario la clase ClienteEMail no esté dependiendo de implementaciones concretas, por lo tanto, en ambos casos, tanto para gestionar correos, como para comunicarse con el servidor, se crearon abstracciones con interfaces, de modo que, si es necesario utilizar otra implementación a futuro, la clase principal del cliente no debería modificarse, ya que depende de la abstracción solamente.

Caso 2 - Banco y Préstamos:

- La solución propuesta no viola los principios SOLID porque cada clase cumple con una única responsabilidad, además están cerradas a la modificación porque cumplen con los principios de encapsulamiento, pero están abiertas a la extensión, en caso de ser requerido. Por otro lado, para los casos de clases donde hay extensión y tenemos clases hijas, se cumple el principio de Liskov Substitution. Y por último, si bien no se crearon interfaces porque era un sistema muy pequeño y no lo consideré necesario, si se creó una clase abstracta para la Solicitud de Crédito, con las características generales y compartidas de todas las solicitudes, y luego se crearon implementaciones específicas para cada uno de los tipos de las mismas. De esta manera se da la posibilidad a que se creen nuevas implementaciones específicas en caso de ser necesario en el futuro, pero asegurándonos de que el Banco siempre las trate como lo mismo a todas, Solicitudes de Crédito genéricas, sin importar el tipo de implementación puntual de cada tipo de solicitud, lo que implica que que el Banco depende de una abstracción y no de una implementación puntual, en otras palabras, cumple con el principio de Dependency Inversion.