Sección 1: Single Responsability Principle

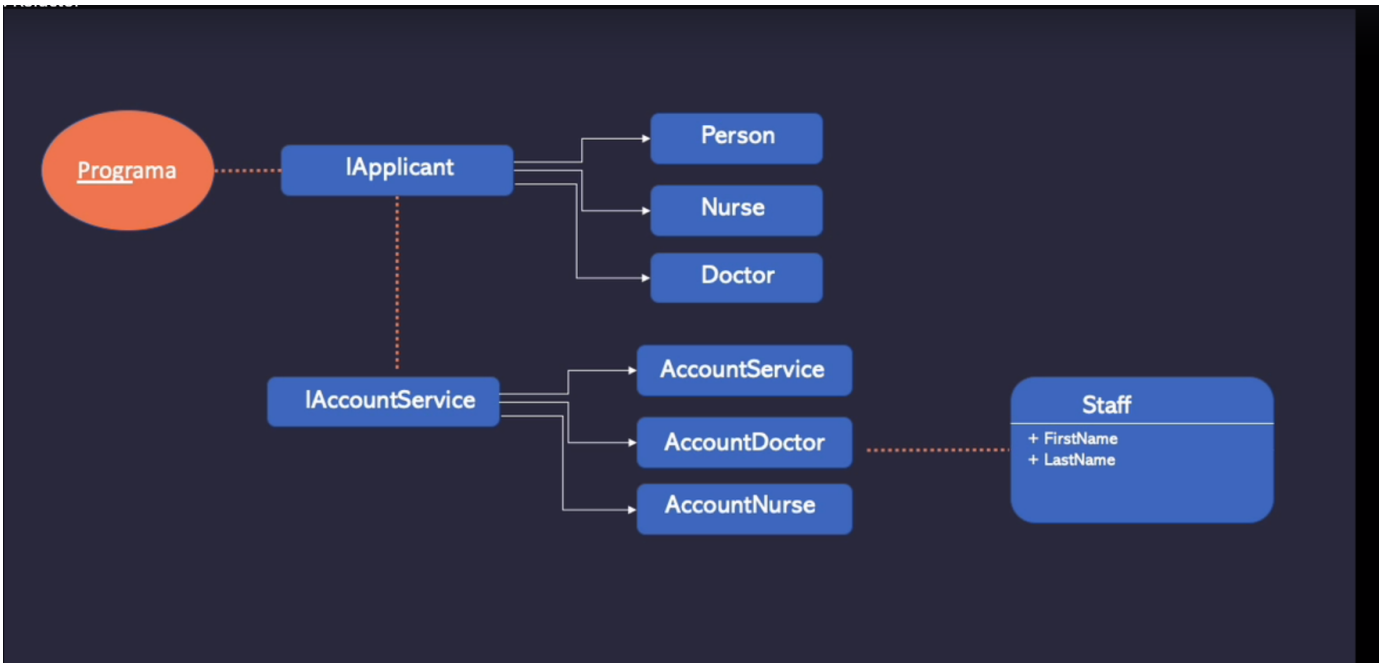
1. Principio de una sola Responsabilidad

- Cada clase debe tener una sola responsabilidad (se debe encargar sólo de una tarea): Cada tarea debe tener sus propias validaciones, aquí es importante acompañar las validaciones con sus respectivos tests.
- Ejercicio git -> Mi solución propuesta con TDD está en la carpeta MySolution.

Sección 2: Open Closed Principle

- Cada clase debe estar abierta a extensiones pero cerrada para modificaciones: La idea es que cuando nos llegue un requerimiento nuevo en vez de realizar el cambio directamente en la clase o en el método, se tenga un buen diseño (éste sea flexible) de forma que se pueda crear la implementación de una interfaz y poder extender ese cambio, de tal forma que no rompamos código ya existente y en funcionamiento.
- Sólo se modifica código existente en el caso de bugs, si no es por causa de un bug, debemos extender nuestro código.
- En las implementaciones de código hay que tender a usar interfaces para poder extender el código de una manera más sencilla en un futuro.

-Por ejemplo:

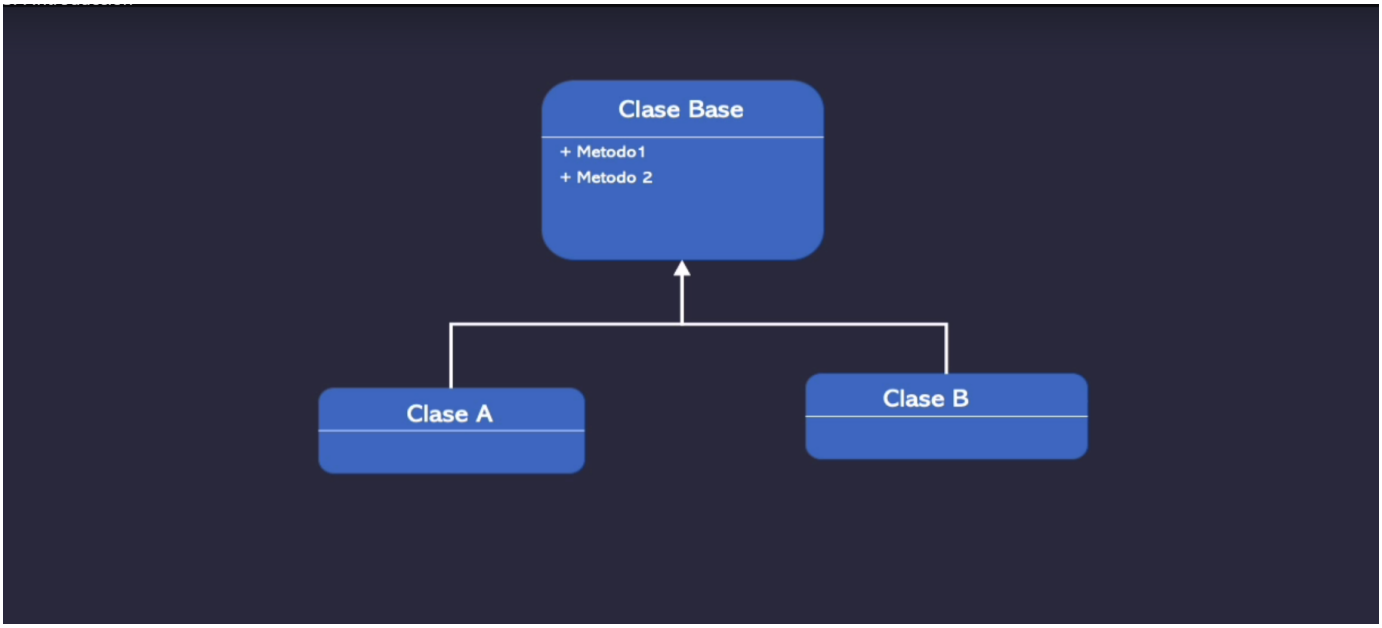


Usamos interfaces para modelar comportamientos determinados según un tipo u otro de implementación del objeto.

- Ejercicio git
- Ejercicio git

Sección 3: Liskov Substitution Principle

- Las clases que hereden desde otras clases pueden ser utilizadas con el comportamiento mínimo de la clase.



Si las clases A y B heredan de la Clase Base, entonces los métodos 1 y 2 de la clase Base deben funcionar en ambas clases A y B.

Por ejemplo, si Tengo la clase Animal y, en este caso, Pájaro y Perro heredan de Animal, Pájaro podría tener el Método Volar pero Perro no lo tendría, con lo cual habría que utilizar de manera alternativa otra interfaz IAnimalFly que albergue este método Fly(), de esta manera podríamos evitar tener Métodos en una clase base que no deben ser usados por una clase hija.

Ejercicio git

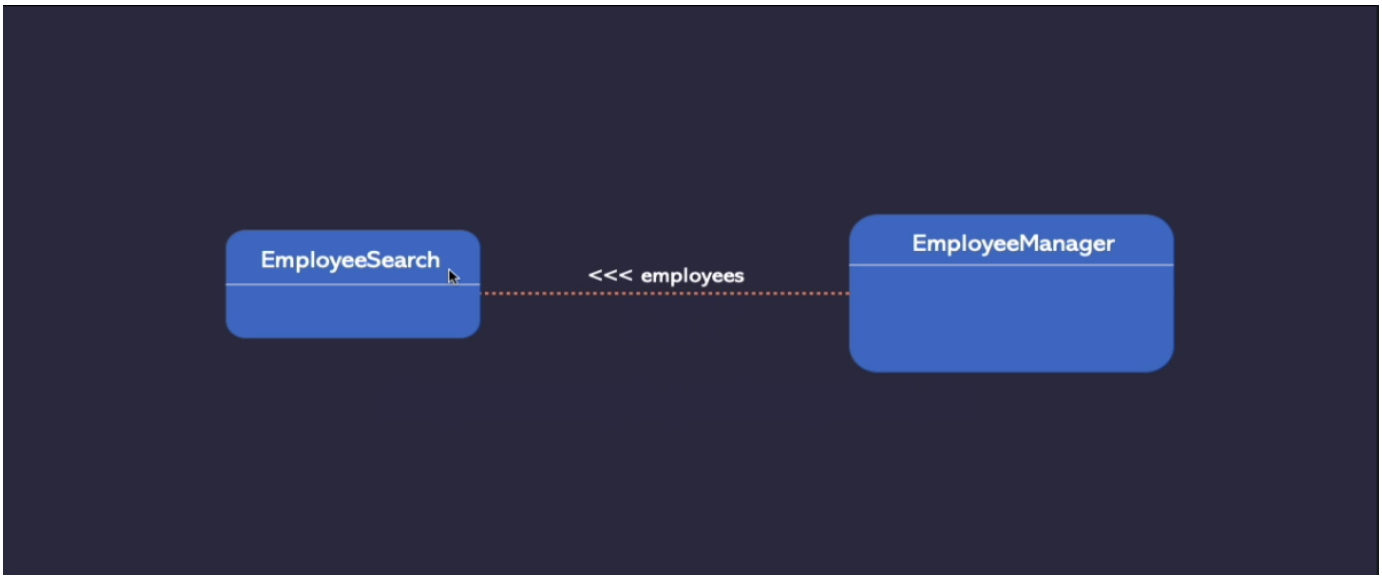
Sección 4: Interface Segregation Principle

- No deben existir interfaces grandes en nuestro programa, de manera que no se obligue a los objetos que la implementan añadir métodos que no necesitan. (análogo al principio Open Closed Principle pero con Interfaces en vez de con clases, el principio OCP desemboca en interfaces pequeñas y está íntimamente relacionado con ISP)
- En la medida de lo posible evitar interfaces grandes ya que esto implica que las clases que lo implementan serán a su vez grandes también, y es muy probable que en alguna haya que añadir un throw not implemented method, añadiendo código no útil al proyecto.

Ejercicio git

Sección 5: Dependency Inversion Principle

- Los módulos de alto nivel no deben depender de los módulos de bajo nivel a menos que sea mediante una abstracción.
- Siguiendo esta filosofía tendremos proyectos más modulares y con un nivel de acoplamiento bajo. Esto nos permite poder inyectar objectos según las necesidades que tengamos.
- Si se elimina la dependencia directa entre 2 clases acopladas y se quiere modificar alguna funcinoalidad, sólo habría que modificar código en una de ellas en vez de ambas, es decir, la abstracción permanece inmutable en el tiempo.



Ejercicio git