# Universidad ORT Uruguay

## Facultad de ingeniería

# Diseño de aplicaciones 2
# Obligatorio 1

https://github.com/ORT-DA2/SobralGoni

210361 - José Pablo Goñi
192247 - Juan Pablo Sobral

# Resumen

El sistema busca brindar una aplicación que permita a un usuario almacenar archivos y carpetas en una especie de nube, se trata de asemejar a un drive de archivos de texto(por el momento).
En la siguiente documentación se explicará y justificará el diseño y arquitectura de nuestra solución relacionando las decisiones tomadas con distintos principios y patrones de diseño vistos en clase. Para esto incluimos distintos diagramas que ayudarán a entender por completo la solución propuesta. A su vez introduciremos evidencias de Clean Code que se cuidaron en el proyecto así como también evidencias de TDD y pruebas unitarias.

# ÍNDICE

# Diseño y arquitectura

Para la realización de una documentación ordenada y explicativa, decidimos basarnos en la metáfora del periódico propuesta por los lineamientos de Clean Code. La misma afirma que una clase comienza con un título descriptivo y una descripción sin detalles que explica el contenido, después vienen los detalles. Podemos entender la clase con los métodos superiores sin necesidad de ahondar en
los detalles desde el principio, es por esto que se comenzará por una descripción a más alto nivel, y se irá descendiendo en los niveles de abstracción hasta una descripción más detallada. Esto también se aplicó frente a los diagramas, sabiendo que los mismos son de utilidad solo si al visualizarlos y entenderlos es más fácil que leer el código. Por lo que a través de ellos se irán mostrando diferentes niveles de abstracción. Así mismo, se mostrarán alguna de las vistas del modelo 4 + 1.

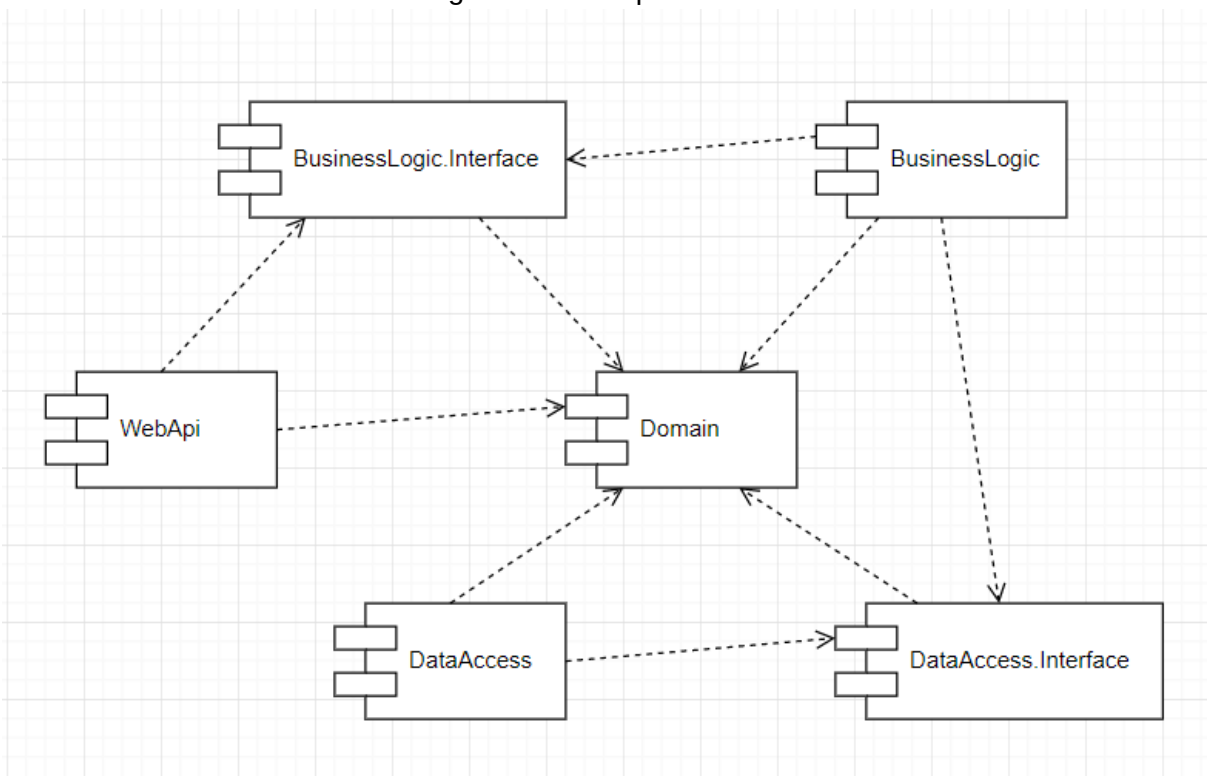
## Vista de desarrollo: Sistema

La vista de desarrollo ilustra el sistema de la perspectiva del programador y está enfocado en la administración de los artefactos de software. Esta vista también se conoce como vista de implementación. Utiliza el Diagrama de Componentes UML para describir los componentes de sistema. Otro diagrama UML que se utiliza en la vista de desarrollo es el Diagrama de Paquetes. Enfocándonos en nuestra solución, para poder definir los distintos diagramas debemos saber que el sistema que realizamos consta de 7 proyectos,1 de ellos refiere a los test del sistema. Todos ellos cumplen con los principios de diseño SOLID.

### Diagrama de componentes

A continuación mostraremos los proyectos en un diagrama de componentes, el cual lo utilizamos para modelar la vista estática y dinámica del sistema. A su vez podremos visualizar la organización y las dependencias entre un conjunto de componentes.

- Diagrama de componentes -




A continuación explicaremos los patrones y/o principios aplicados que se desprenden del diagrama:

+ Inversión de dependencias(SOLID) , ya que apunta siempre a la lógica de negocios. Esto es porque ninguno de los módulos de alto nivel dependen de módulos de bajo nivel. Para ello se crearon abstracciones de cada nivel que generan esta separación.

+ Principio de segregación de interfaz(SOLID) , pudimos apreciar luego de armar todo que la lógica de archivos, la de carpetas y la de usuario tenian muchas cosas en comun ya que en todas teníamos ABM aunque por el lado de archivos y carpetas teníamos algunas otras funciones como mover de lugar. Dado esto no podíamos tener la misma interfaz para todo ya que usuarios no iba a implementar cosas de la interfaz declarada y ademas teniamos el problema de que la logica de archivo y carpeta tenían muchas cosas que se hacen igual, lo único que variaba era el tipo de dato que se manejaba, por eso implementamos una solución basada en estos hechos.

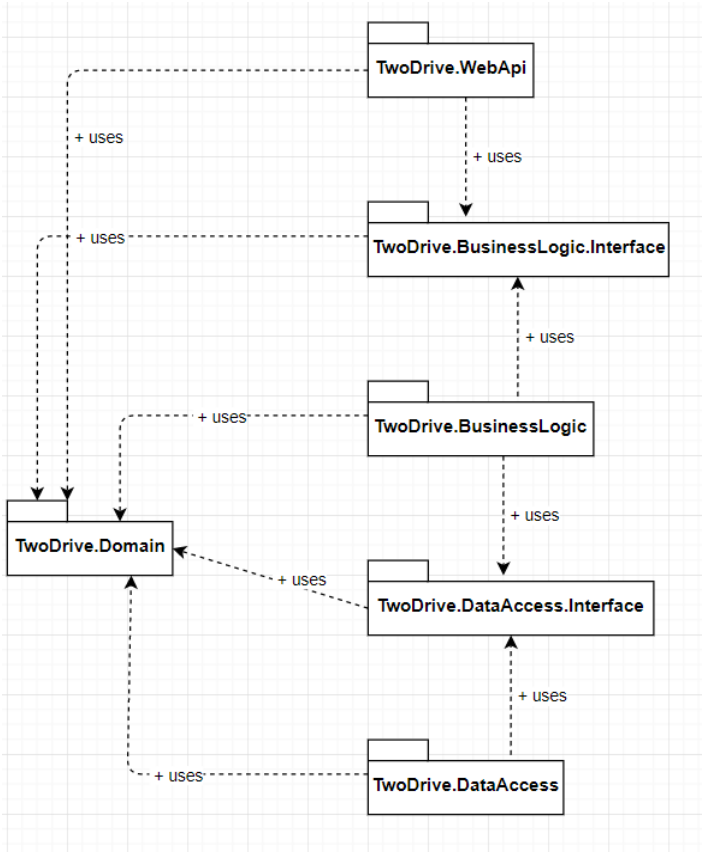
+ Indirection(GRASP) , creamos clases intermedias para desacoplar clientes de servicios. De esta forma creamos estabilidad en el sistema.

+ Creator(GRASP) , derivamos la creación de todo a la webApi, inyectando dependencias.

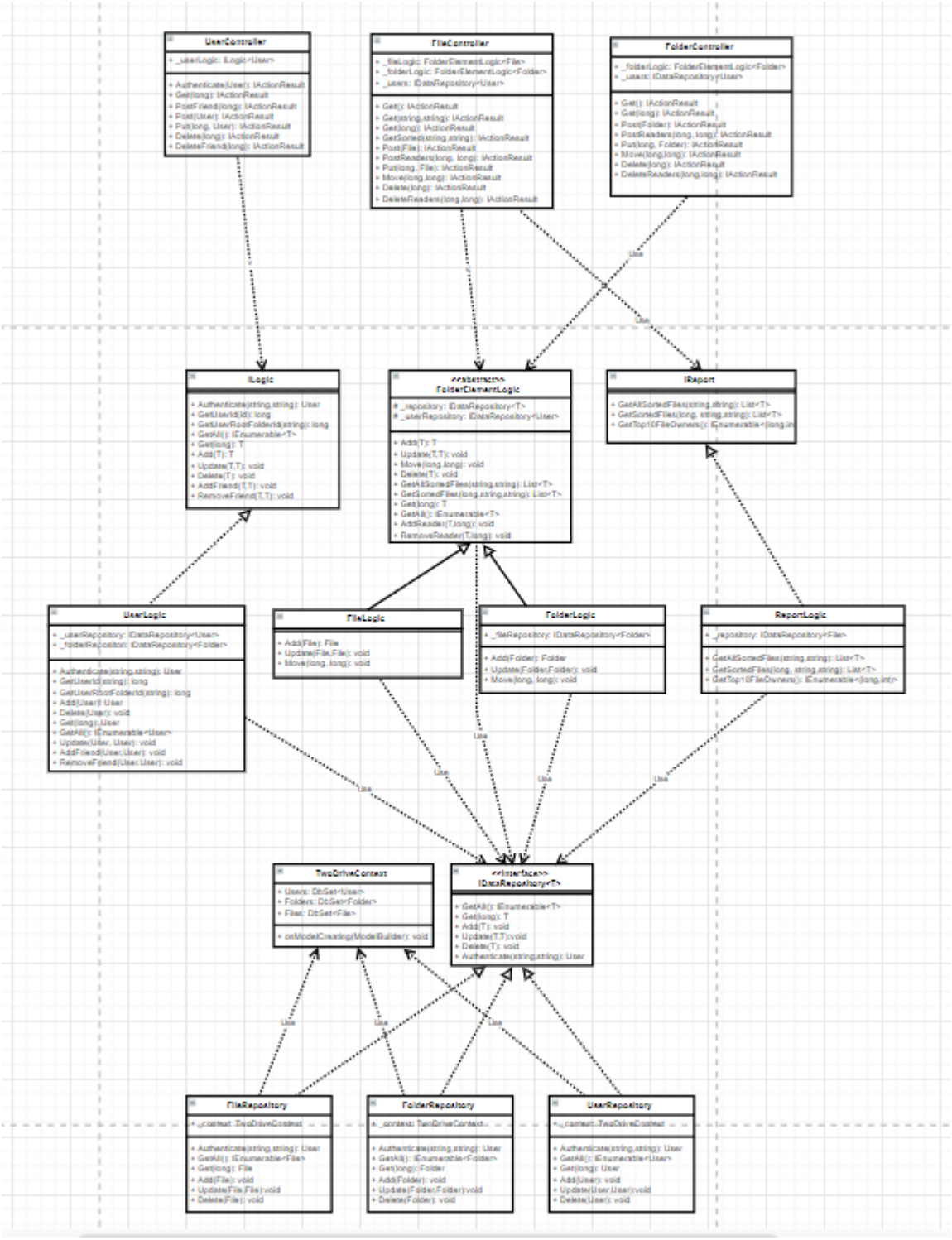
+ Protección de variaciones(GRASP) , se decidió que las dependencias fueran a nivel de abstracciones, como se verá en los siguientes diagramas. De esta manera se buscó que el sistema fuera mantenible y extensible siendo este robusto frente a cualquier posible modificación y resolviendo el menor impacto posible sobre el código fuente ya existente, tal como es definido en el principio de abierto cerrado. Al hacer que el alto nivel no dependa del bajo nivel, se logra que el sistema sea más resistente a cambios. Esto también se logra haciendo que las dependencias se encuentren a nivel de abstracciones ya que se crea un efecto de ocultamiento hacia los detalles cambiantes.

## Diagrama de paquetes

Por otro lado, el diagrama de paquetes muestra como un sistema está dividido en agrupaciones lógicas y las dependencias entre esas agrupaciones. Se puede visualizar que se cumple con el principio SOLID de responsabilidad ùnica ya que a la hora de crear los paquetes se tuvo en cuenta que los mismos tengan solo una razón por la cual cambiar y por ende tenga una ùnica responsabilidad. Se trata de llevar al límite este principio, pero siempre respetando el bajo acoplamiento, la alta cohesión, y el reuso de código. El paquete domain tiene una arquitectura paralela a todo el sistema para el uso de todos los paquetes.




## Diagrama de clase(sistema completo)



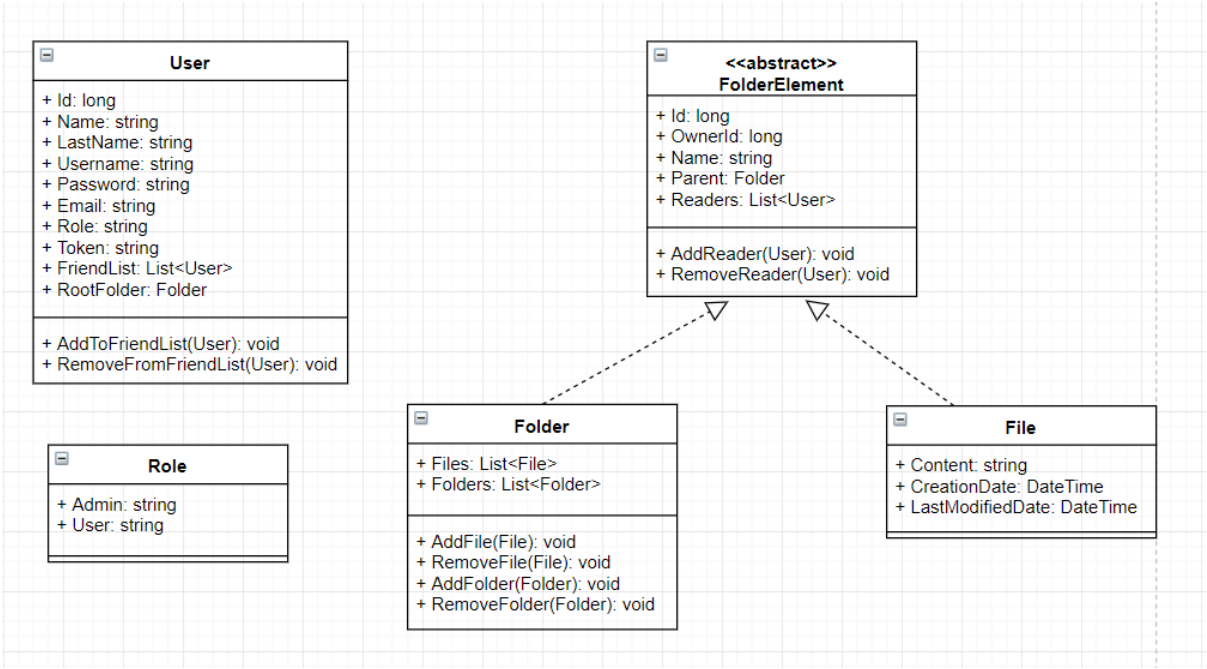

(Las clases usan también el dominio que no se incluyo ya que quedaba mal la captura abarcando todo.)

# Vista Lógica

## Domain

En este paquete se encuentran las entidades cuyos nombres son sustantivos propios del lenguaje del dominio del problema. En el dominio se encuentra las entidades principales que modelan el sistema. Se decidió crear una clase abstracta llamada “FolderElement” ya

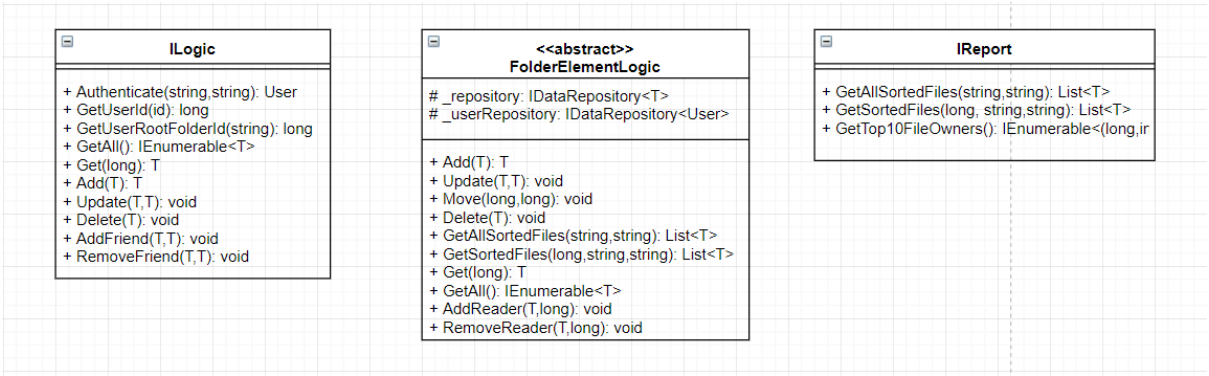
que las clases file y folder tenian muchos atributos en común, de esta forma no repetimos código innecesario. Creamos una clase para los usuario y una clase para los roles de los mismo para poder autenticarse ya que los usuarios pueden cumplir diferentes funciones dentro del sistema. No se creo una jerarquía de usuarios ya que en la letra nos dicen que no se le de mucha importancia a futuros cambios de tipos de usuarios.
A continuación mostramos el diagrama de clases del mismo:



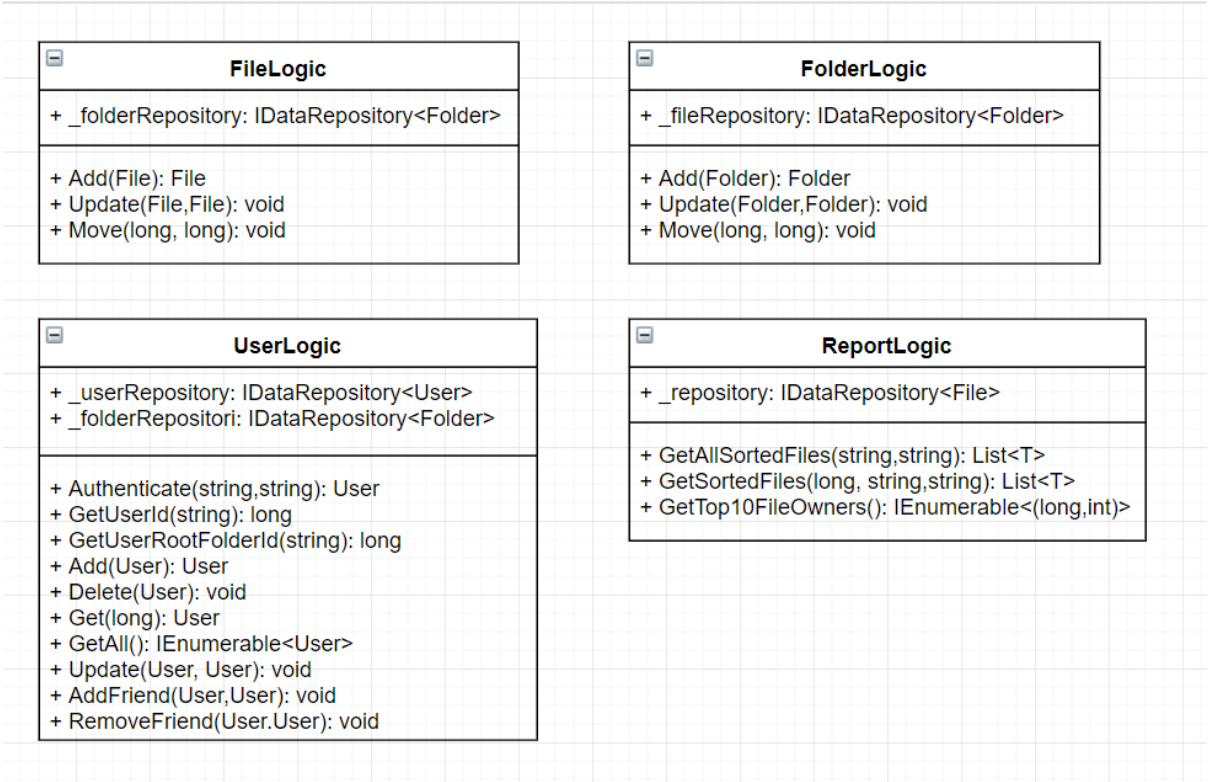

## BusinessLogic y BusinessLogic.Interface

La lógica de negocio empezó como se había planificado antes de empezar el proyecto pero durante el mismo nos dimos cuenta que estábamos repitiendo mucho código con la lógica de archivos y carpetas, ya que las mismas tenían muchas cosas que se comportan de la misma forma pero lo único que cambiaba era el tipo de dato que se manejaba. Por ese motivo se decidió crear una clase abstracta que implementara la lógica en común a los dos y derivar el comportamiento individual a cada uno por su lado. Nos basamos en el patrón de diseño **Template Method** para realizar esto. Luego para la parte de lógica de usuario creamos una interfaz estable y chica para que la web api solo conozca el comportamiento y no como se implementó, también creamos una interfaz de reportes para los requerimientos de ordenar los archivos que se pedía. A continuación mostramos un diagrama detallando lo antes descripto:

-BusinessLogic.Interface-



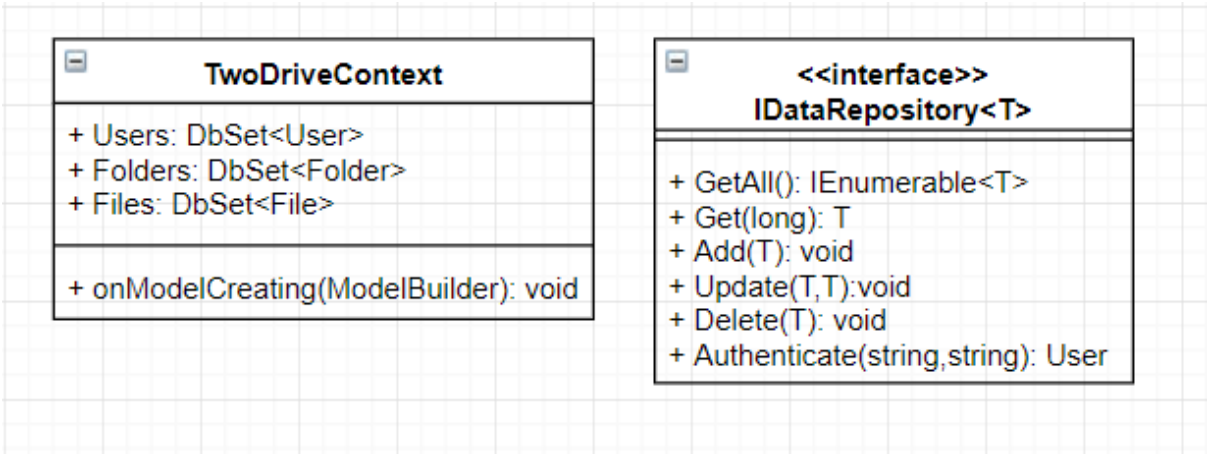

-BusinessLogic-



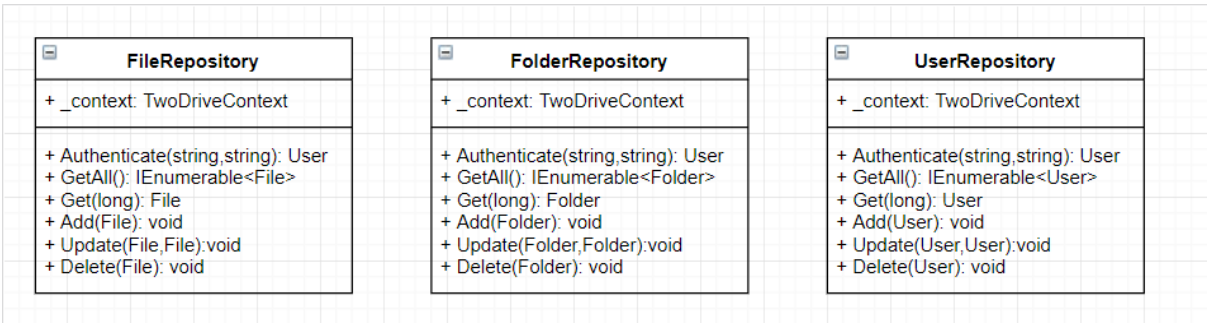
## DataAccess y DataAccess.Interface

En esta parte apreciamos que todas las entidades del sistema realizaban el mismo comportamiento hacia la base de datos, por eso se decidió crear una interfaz IRepositorio la cual tiene las firmas que cada entidad que quiera guardarse debe respetar. Gracias a esto no dejamos que la lógica de negocio vea los detalles de la base de datos, si la misma se guarda en memoria, en una base sql o en otra base de datos que un futuro nos puedan brindar. A continuación mostramos un diagrama con lo visto anteriormente:

-DataAccess.Interface-




-DataAccess-

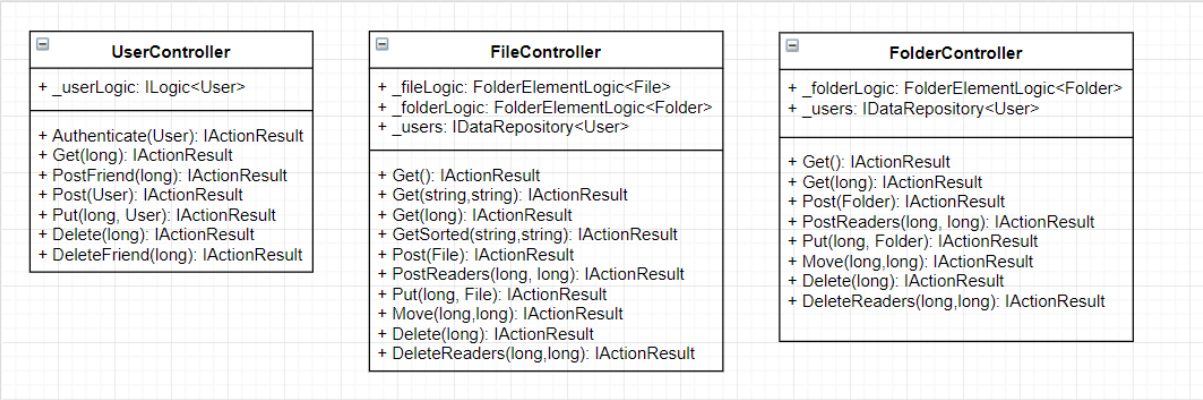



## WebApi

En primer lugar es importante aclarar que se cumple con la arquitectura basada en REST( Representational State Transfer) que vendría a ser un estilo de arquitectura cliente-servidor . Habitualmente cuando se realiza una comunicación cliente servidor accedemos al servidor en un punto de acceso , se envía una información y se recibe un resultado. Para poder acceder desde cualquier sitio se debe seguir una comunicación HTTP mediante distintas operaciones que nos permiten realizar dicha comunicación. Alguna de estas operaciones utilizadas en nuestra solución son: GET, POST, PUT, DELETE. El haber creado una arquitectura Rest nos asegura mantenibilidad, extensibilidad y robustez frente a cualquier tipo de cambio posible.

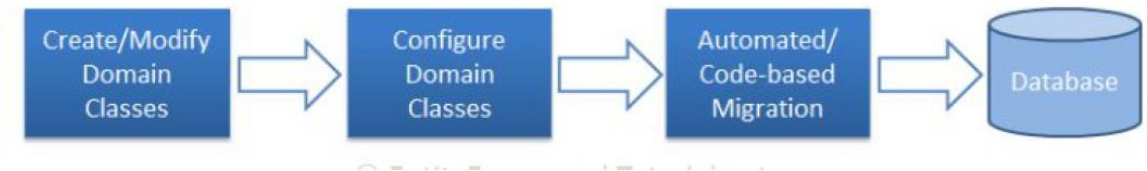
La WebApi de nuestra solución está dividida en controllers los cuales definen los distintos end points que van a permitir la interacción con el sistema. Los detalles de los mismo se pueden encontrar en el siguiente link: [https://app.swaggerhub.com/apis-docs/juanpaSobral/TwoDrive-APIGoniSobral/1](https://app.swaggerhub.com/apis-docs/juanpaSobral/TwoDrive-APIGoniSobral/1)
O ejecutando el código del sistema y se le abrirá toda la documentación.
A continuación detallamos el diagrama de clase de la web api:

-WebApi-




# Base de datos y persistencia

Para la persistencia de datos se utilizó la tecnología Entity Framework que actúa como un mappeador de objetos no relacionales. A su vez como metodología se siguieron los conceptos de CODE FIRST ya que primero creamos el modelo con código y después se genera automáticamente la base de datos. Se actúa de la siguiente manera:



# Modelo de la base de datos

Diagrama de tablas y sus relaciones: