Universidad ORT Uruguay
Facultad de Ingeniería

Primer Obligatorio Diseño de Aplicaciones 2

Mayo 2018

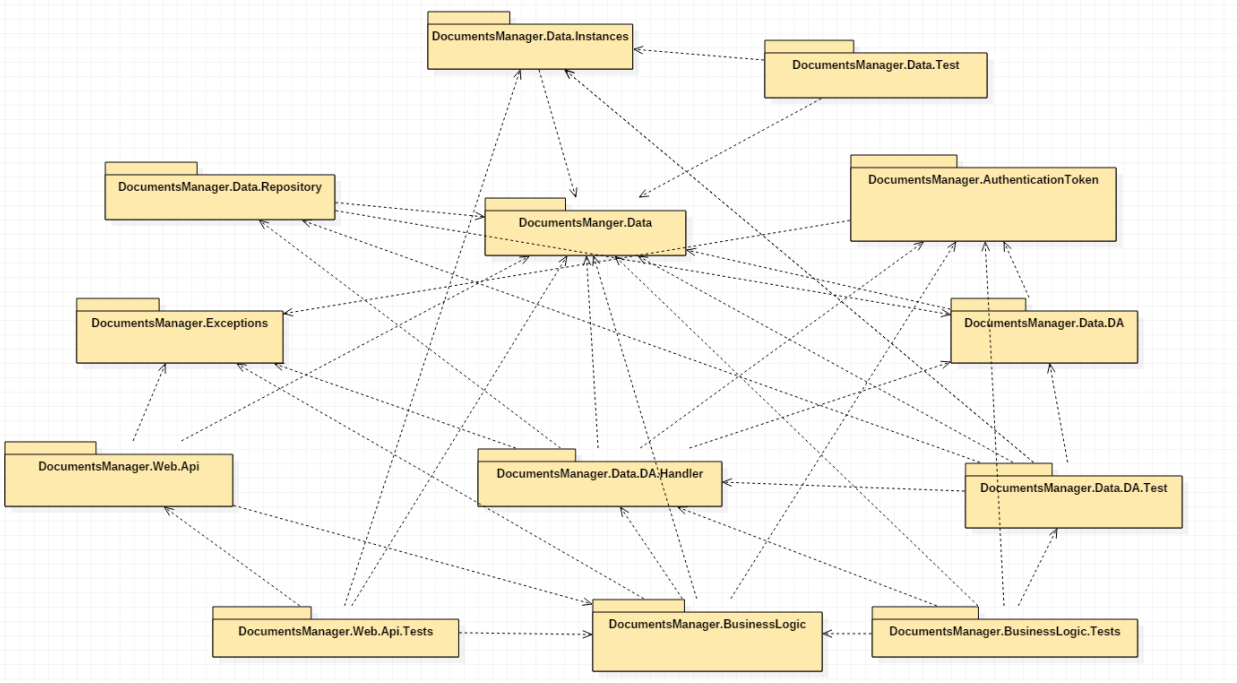
Profesor: Gabriel Piffaretti

Federico Cetraro(193221)
Fabian Grobert (194738)

Indice

Justificación de Diseño

Para la mejor realización de este obligatorio, la primera decisión de diseño que tomamos fue dividir la solución en 13 paquetes distintos. Esto se hizo principalmente para agrupar cierto orden lógico en cada uno de ellos.
Se crearon paquetes para podes realizar las funcionalidades pedidas y por cada
uno de estos se creo un paquete de prueba.
A continuación se representa mejor esta decisión:



Para nuestra solución, quisimos realizar el diseño para que sea lo más extensible

posible. Es decir, que sea fácil de agregar nuevas funcionalidad en un futuro tales como nuevos Atributos de estilo. De esta manera, para cada decisión de diseño que tomamos, tratamos de cumplir el principio abierto/cerrado.

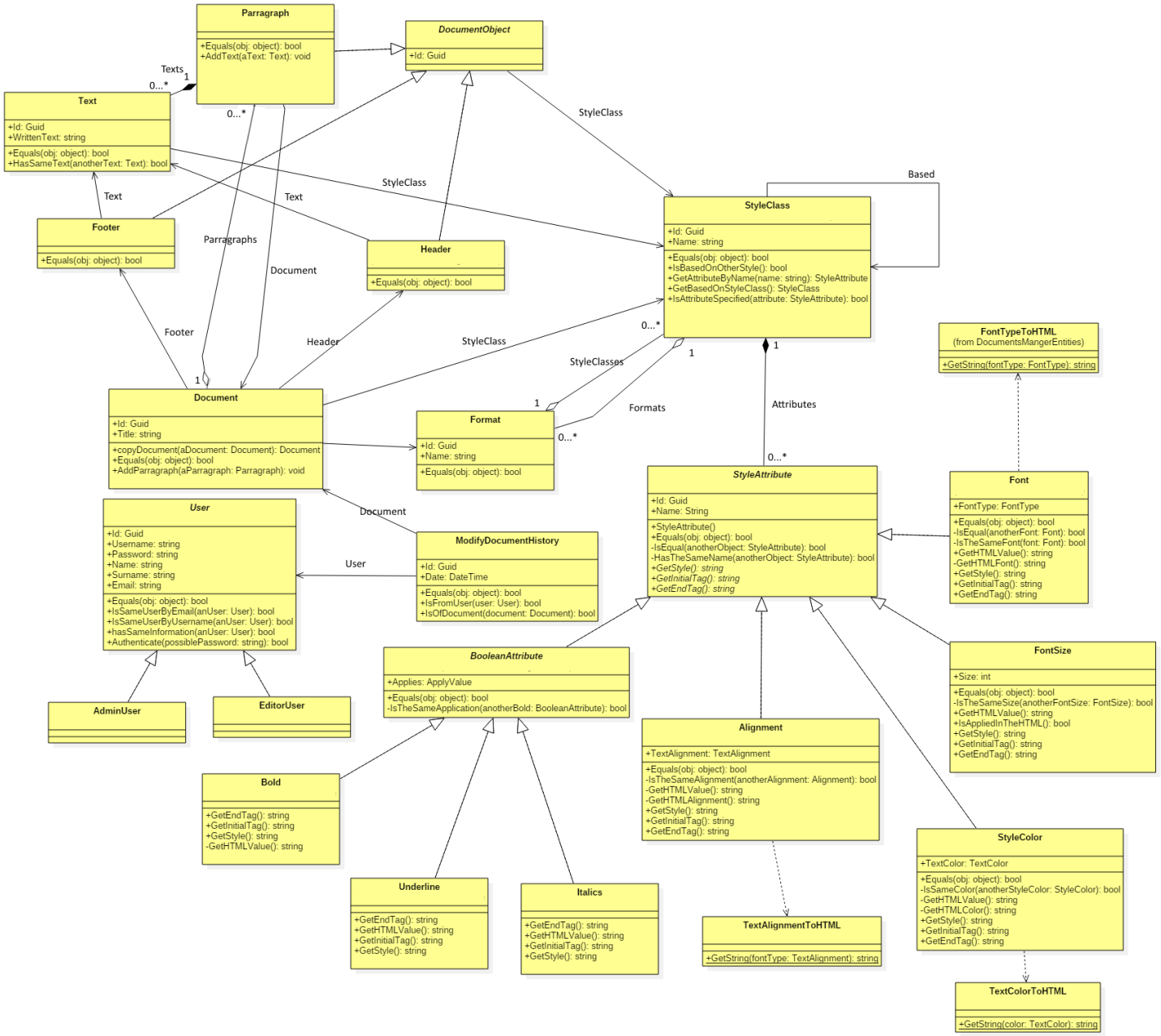
Para poder realizar una solución extensible, hicimos mucho uso del Polimorfismo. Esto se puede ver en los casos que se detallaremos a continuación:

- Para obtener el código HTML dado un StyleClass y un texto se trabaja con Polimorfismo, donde cada StyleAttribute imprime lo que le corresponde sin tener que investigar en RTTI.
- A su vez, para implementar la impresión de un documento en html, diseñamos una solución que intenta adoptar el patrón Strategy. La clase de la lógica de negocio de documento carga una lista con objetos de Tipo IPrintableObject, que estos, al implementar a la interfaz, saben cómo imprimirse de diversas formas, dependiendo si deben imprimir un encabezado,

pie o líneas del párrafo. De esta manera, si el día de mañana se desea añadir a documento una nueva parte, esta para imprimirse debe cumplir con el contrato de IPrintableObject.

Diagrama de Clases

Se buscó realizar una buena extracción de del problema de tal manera que pueda ser fácil de Extender, principalmente en el caso que se quisiera añadir más Atributos al Estilo que bastaría con solo realizar una clase que herede de StyleAttribute y luego adapte su comportamiento de acuerdo a lo que sea necesario.



Capas Lógicas

Nuestra solución está divida en capas lógicas con el fin de separar diferentes responsabilidades del sistema. A su vez, estas capas tienen un orden establecido de cómo se comunican y como se transfieren las peticiones del sistema.
Contamos con tres capas. En primer lugar la capa donde se encuentran los controladores y donde se encuentran los “end points”. Esta capa está representada por nuestro paquete DocumentsManager.Web.Api.
En este paquete los métodos no tienen mucha lógica. Simplemente validan que las peticiones sean las correctas para acceder a cada uno de los recursos y llaman a la capa de servicios. Si se realiza correctamente se devuelve el resultado y si no se devuelven los mensajes de error encontrados. Esta capa siempre devuelve los mensajes de error con “BadRequest” y el mensaje que se capturo.

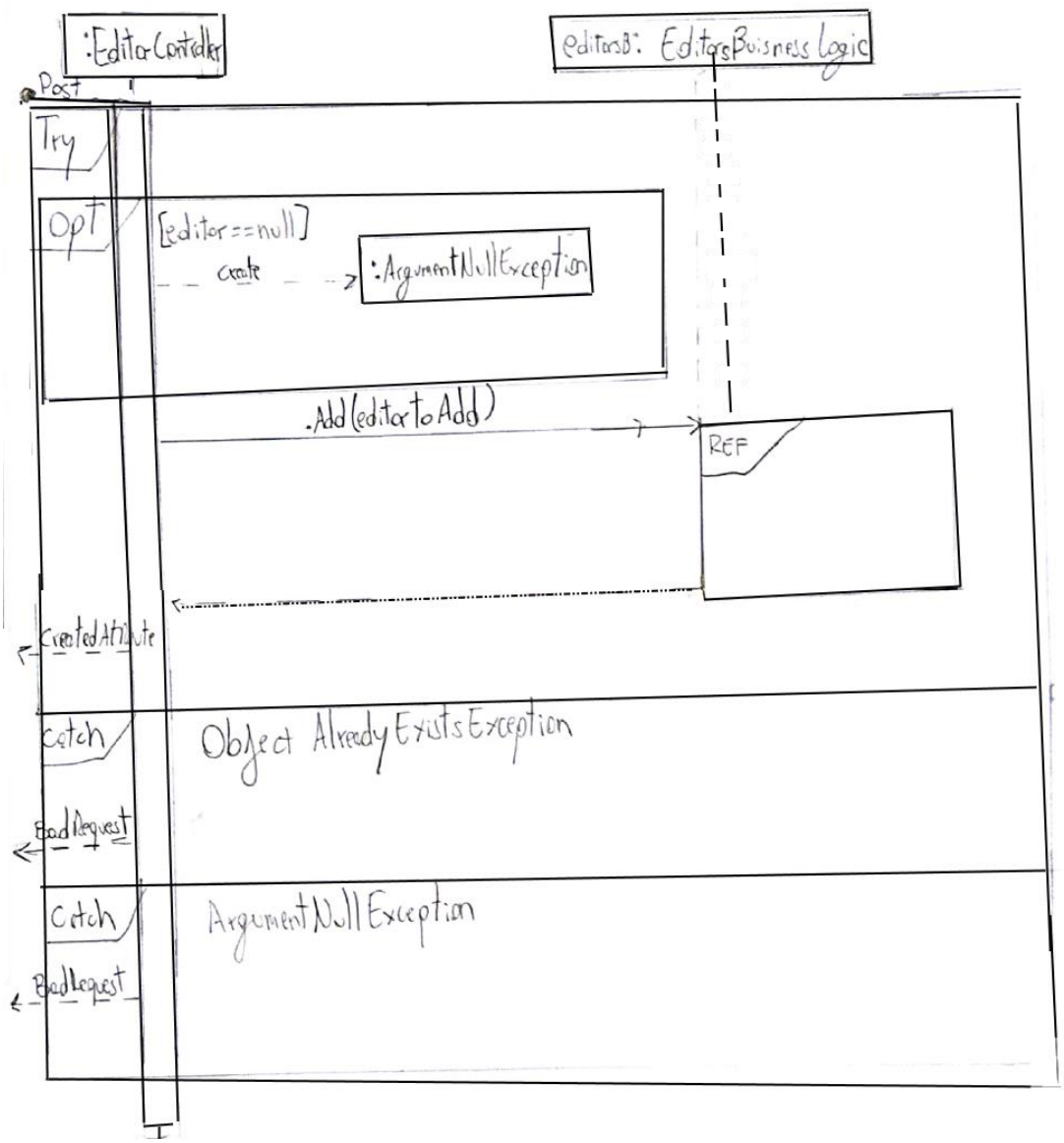
La segunda capa, la que se comunica con la Api es la capa de los servicios. Se implementa en el paquete DocumentsManager.BusinessLogic Este paquete es el que tiene la responsabilidad de hacer todas las validaciones necesarias. Por ejemplo, hacer validar que el usuario este autorizado para realizar cierto recurso, validar que el usuario y la contraseña sean correctas al iniciar sesión, validar que los atributos de los estilos, documentos y textos sean correctos al agregar uno nuevo, etc.
Esta capa no es la responsables de persistir los datos, ni comunicarse con la base de datos. Simplemente procesa y valida la información, para luego decirle a la capa siguiente que los maneje en la base de datos.

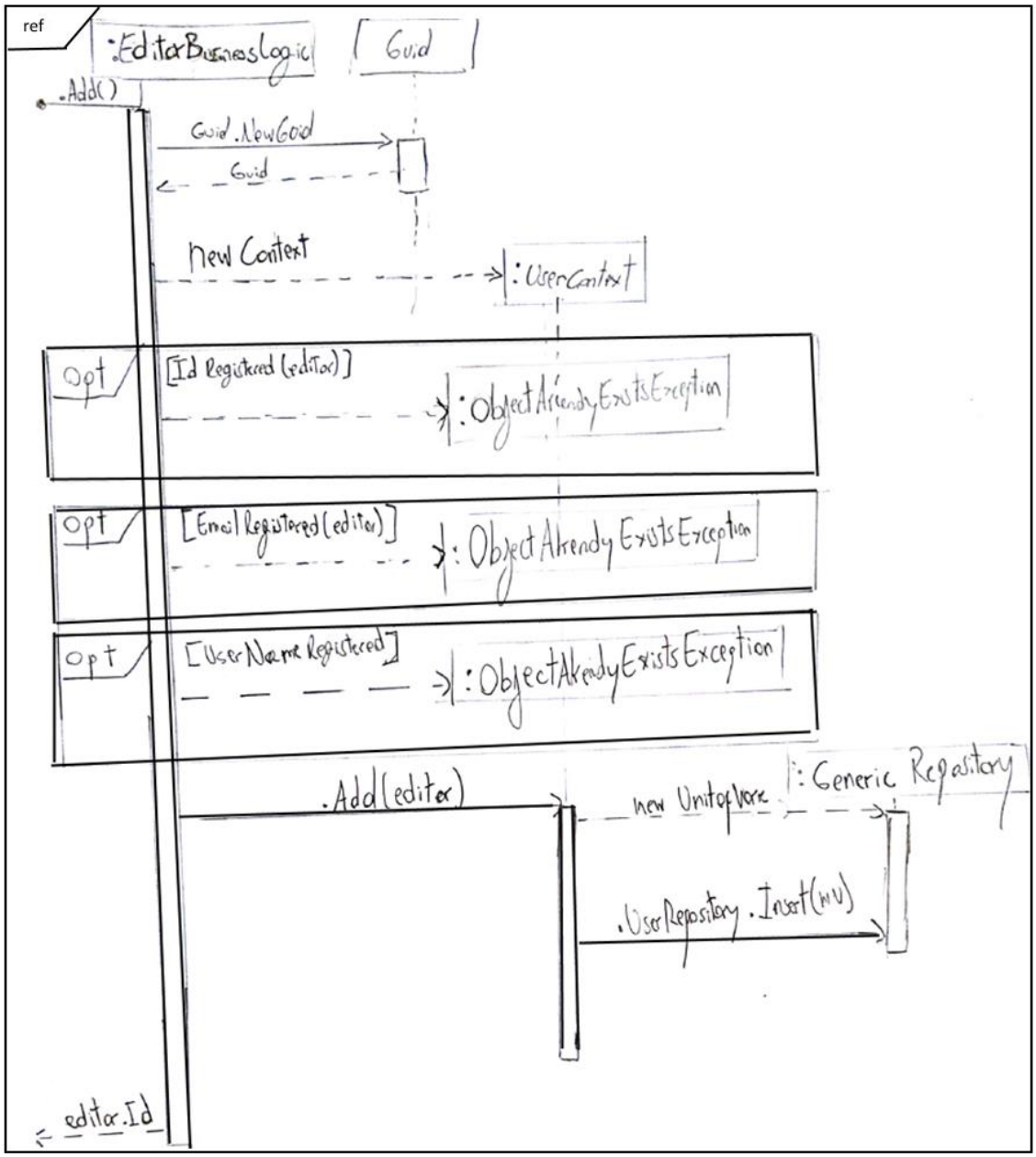
En último lugar, para persistir los datos, se utilizó el patrón Repositorio. Este patrón permite separar lo que es la persistencia y el acceso a los datos, de la capa de servicios, la cual contiene la lógica de negocio del sistema. De esta forma, se permite que la capa de servicios ignore la forma en que los datos son persistidos y accedidos.
La clase Unit of Work es fundamental, ya que maneja el contexto, así de esta forma nos aseguramos de que todos los repositorios usen el mismo contexto.
A su vez, también existe un repositorio genérico, debido a que existen consultas básicas que se repiten e incluir estas consultas en cada repositorio específico significaría repetir una gran cantidad de código. Se puede apreciar las operaciones básicas como el get, insert, delete, exists, entre otros.

Los diagramas que se encuentran a continuación hacen una muestra más clara de cómo se relacionan estas capas ante una petición.

Un Diagrama de interaccion del método del ingreso de un Editor al sistema:

Diagrama de Secuencia




ref
:EditorBusinessLogic
Guid
.Add()
Guid.NewGuid
Guid
New Context
:UserContext
Opt
[Id Registered (editor)]
:ObjectAlreadyExistsException
Opt
[Email Registered (editor)]
:ObjectAlreadyExistsException
Opt
[UserName Registered]
:ObjectAlreadyExistsException
.Add(editor)
new UnitofWork
:Generic Repository
.UserRepository.Insert(nU)
editor.Id

Para culminar, añadimos los siguientes componentes de entrega. No se incluyen los paquetes de prueba ya que creemos que estos son importantes para quien desarrolla la solución y no para el cliente que la consume.
DocumentsManager.Web.Api

Diagrama de entrega:

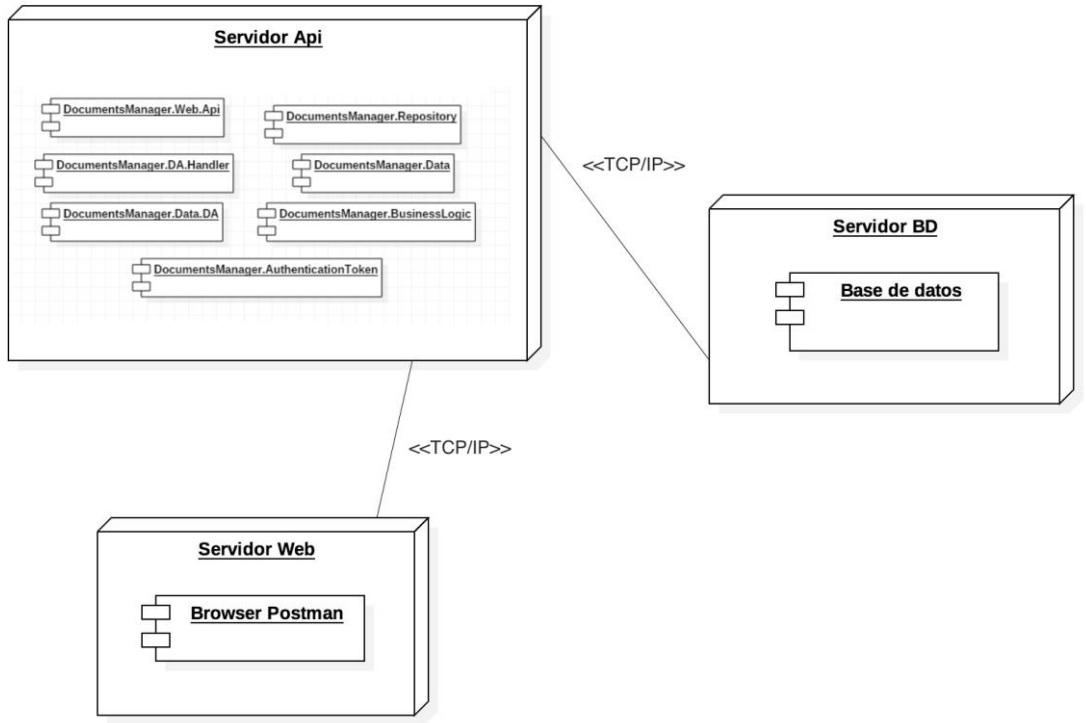
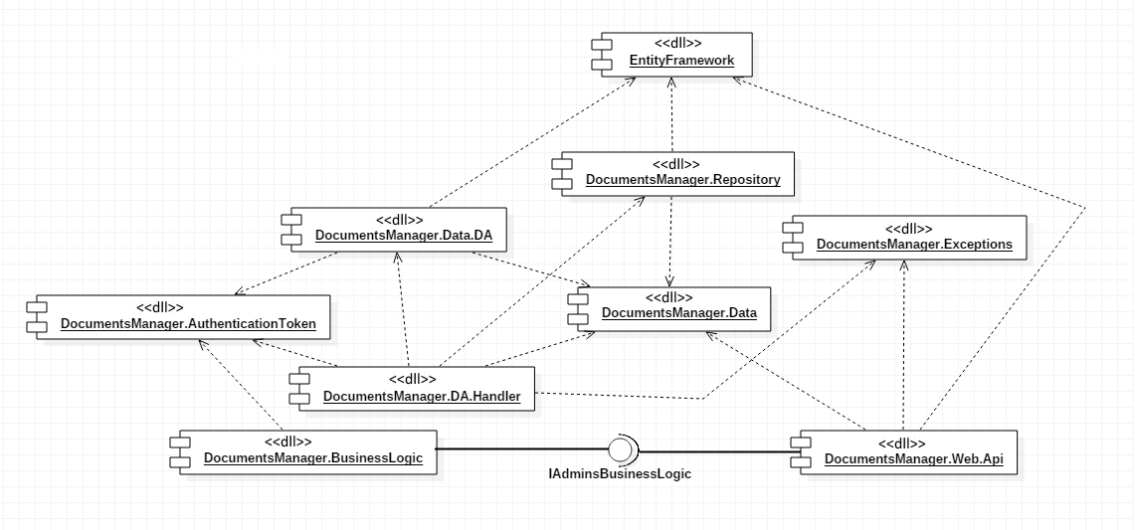
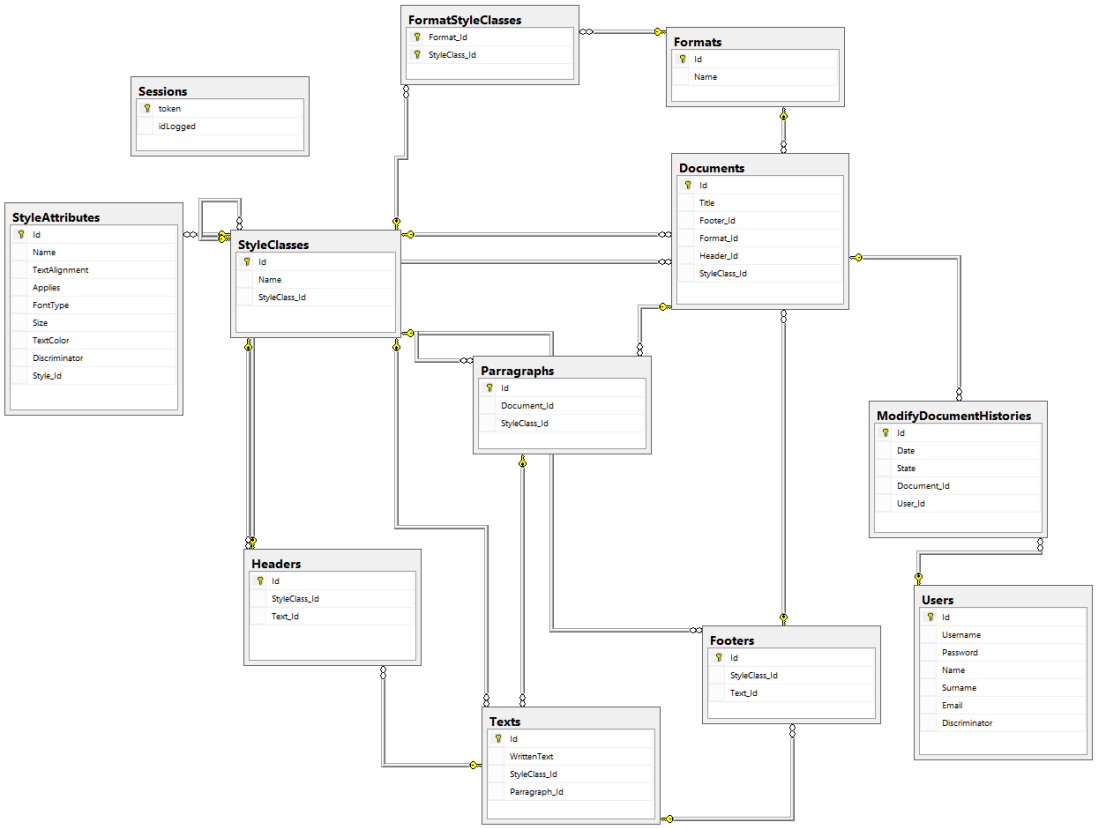


Diagrama de Componentes



Modelo de Tablas

Para persistir los datos en la base de datos decidimos mapear las tablas de la siguiente manera.
Como



Este mapeo fue realizado Mediante el uso del enfoque “Code-First” de EntityFramework

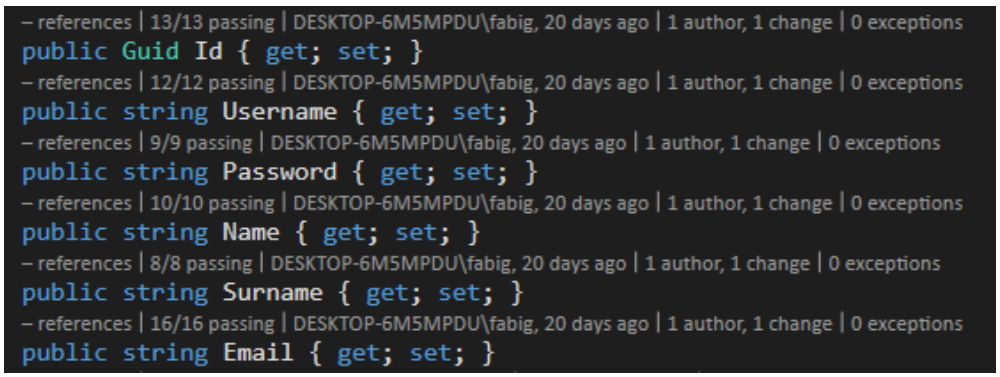
Informe de Clean Code y pruebas.

Clean Code

La implementación de este proyecto fue realizada siguiendo las prácticas sugeridas en el libro de Clean Code “Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*.”.

El capítulo 2 del libro habla de los nombres a utilizar, estos tienen que ser claros y con sólo leerlos nos deben describir que es lo que hacen. Es por eso que para declarar los atributos, nombres de las clases, nombres de los métodos, pruebas, buscamos ser los más claros y descriptivos posible para que la lectura del código sea mucho más sencilla.

En el siguiente ejemplo vemos la nomenclatura de los atributos de los usuarios que son claros y queda claro a simple vista a que se refiere:



En el capítulo 3 del libro se habla de las funciones, Una buena función es aquella de la que se puede inferir su comportamiento rápidamente. Para ello deben ser cortas, hacer una única cosa y mantenerse dentro del mismo nivel de abstracción.

```
public Guid Add(AdminUser newAdmin)
{
    newAdmin.Id = Guid.NewGuid();
    UserContext userContext = new UserContext();
    if (IdRegistered(newAdmin))
    {
        throw new ObjectAlreadyExistsException("Id");
    }
    if (EmailRegistered(newAdmin))
    {
        throw new ObjectAlreadyExistsException("email");
    }
    if (UserNameRegistered(newAdmin))
    {
        throw new ObjectAlreadyExistsException("username");
    }
    userContext.Add(newAdmin);
    return newAdmin.Id;
}
```

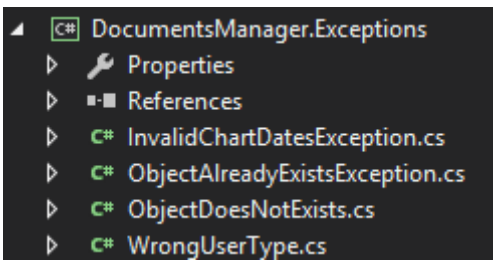
Función concreta, donde los validadores y agregar son funciones a parte.

Para las funciones seguimos la misma práctica de los nombres descriptivos y claros. Buscamos reducir al máximo el número de argumentos de cada función, es por eso que la función que recibe más parámetros es 3.

```
private void AddModifyHistory(User user, Document doc, ModifyState state)
```

El capítulo 4 del libro habla de los comentarios en el código. Es por eso que se evitó totalmente el uso de comentarios, la necesidad de comentarios para aclarar algo es síntoma de que hay código mal escrito que debería ser rediseñado. Es preferible expresarse mediante el propio código.
El único código que se utilizaron comentarios fue en las pruebas de Mock para separar cada una de las diferentes secciones.

El capítulo 7 del libro habla del manejo de errores. El manejo de errores fue algo que tuvimos presente a lo largo del desarrollo del obligatorio. Estos errores lo hicimos a través del manejo de excepciones y para ello definimos una clase de excepción para cada entidad del dominio, así como para el motor y para la conexión a la base de datos.



Según el error manejado genera una nueva excepción.

Luego para cada acción que realiza un usuario desde la WebApi se hace un *try/catch* y en caso de entrar al catch se le muestra el mensaje de error generado en la excepción.

En cuanto a las pruebas seguimos las prácticas sugeridas en el capítulo 9 del libro acerca de las pruebas unitarias. Buscamos acoplarnos lo más posible a las reglas FIRST sobre el código de test son:

- **Fast**: Se deben ejecutar rápido y muy a menudo.
- **Independent**: Las condiciones de un test no deben depender de un test anterior.
- **Repeteable**: Se deben poder ejecutar en cualquier entorno.
- **Self-Validating**: El propio test debe decir si se cumple o no, no debe hacer falta realizar comprobaciones posteriores al test.
- **Timely**: Los tests se deben escribir en el momento adecuado, que es justo ante de escribir el código de producción, lo que permite escribir código fácilmente testeable.

```
[TestMethod]
| 0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
public void UpdateAdminBLTestCheckEmail()
{
    SetUp();
    AdminBusinessLogic adminBL = new AdminBusinessLogic();
    AdminUser anAdmin = EntitiesExampleInstances.TestAdminUser();
    Guid IdUserAdded = adminBL.Add(anAdmin);
    anAdmin.Email = "modified@gmail.com";
    adminBL.Update(IdUserAdded, anAdmin);
    Assert.AreEqual(adminBL.GetByID(IdUserAdded).Email, "modified@gmail.com");
    TearDown();
}
```

Ejemplo de pruebas: Cada una prueba una única cosa. Utilizamos métodos de SetUp y TearDown para facilitar la iniciación y la finalización de cada prueba.

TDD

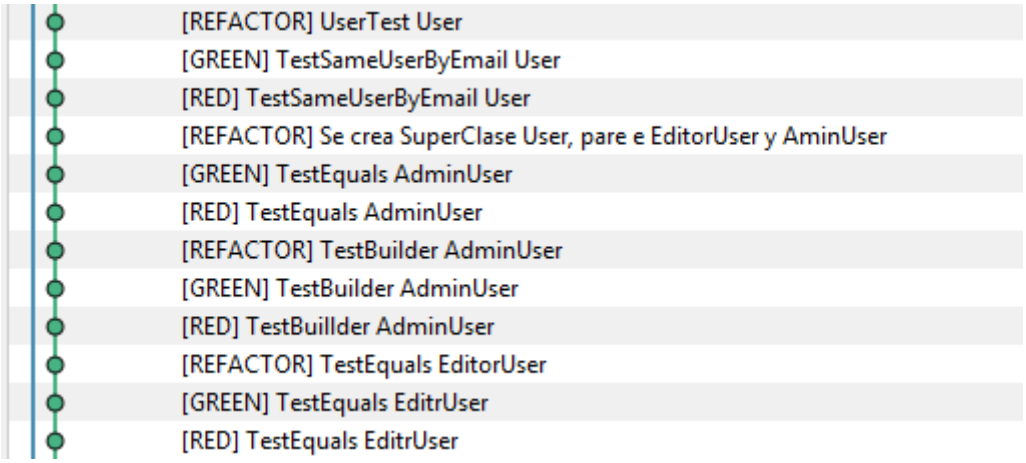
El desarrollo del software fue a partir de TDD utilizando el framework xUnit.
La implementación del obligatorio se fue realizando primero haciendo las pruebas unitarias y a medida que se iban desarrollando las mismas se hacían los métodos, clases, etc.
Primero se realizaban las pruebas para que las mismas fallen (no necesariamente compilaban) y luego se implementaban los métodos, clases, etc. mínimo que fuera necesario para que las pruebas sean correctas.
Una vez que las pruebas tenían el resultado esperado, en caso de ser necesario, se realizaba un Refactor con el fin de mejorar la calidad del código.
Las pruebas fueron implementadas tanto como para que fallen como para que no falle el sistema y se buscaron generar todos los casos bordes del sistema.
Dentro de las pruebas se probaron también las excepciones del sistema de la BusinessLogic.
Cada prueba es independiente una de la otra.

El desarrollo del software a partir de las pruebas unitarias nos ayudó a reducir el número de errores.

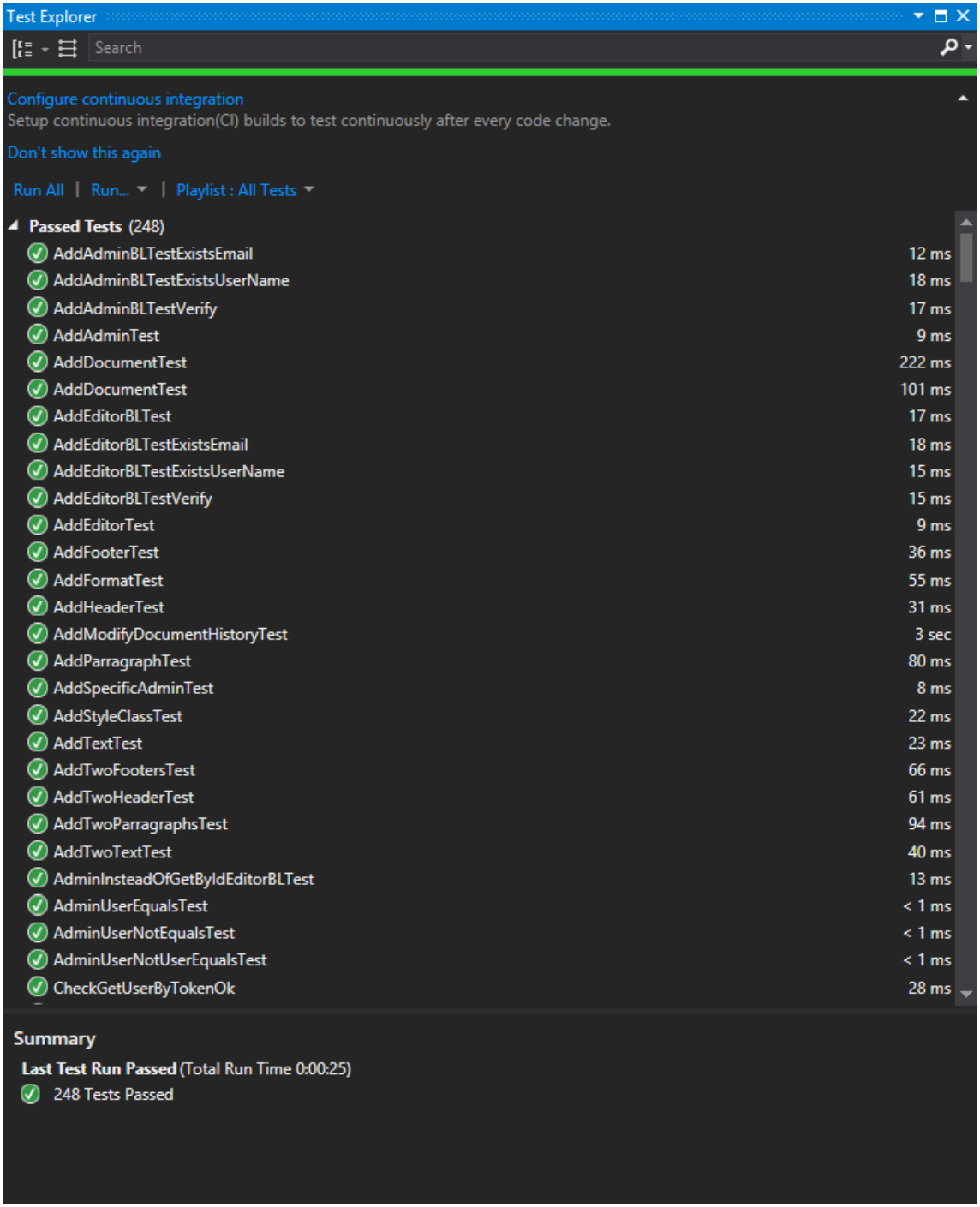
Evidencia de TDD

En el repositorio se puede ver la evidencia del uso de TDD. Se ven los pasos que
Utilizamos mencionados anteriormente agrupados en 3 etapas distintas:

1. Paso Red: Creamos la prueba aunque no compile.
2. Paso Green: Implementamos lo mínimo necesario para que la prueba compile (en caso de que no lo haga) y luego para que pase.
3. Refactor: Modificamos el código implementado en los pasos anteriores sin modificar el comportamiento del mismo aplicando las técnicas mencionadas en el Sector de “Clean Code”.



Copiamos una imagen del repositorio en la cual se puede observar claramente los pasos
“Red”, “Green” y “Refactor” mencionados anteriormente.

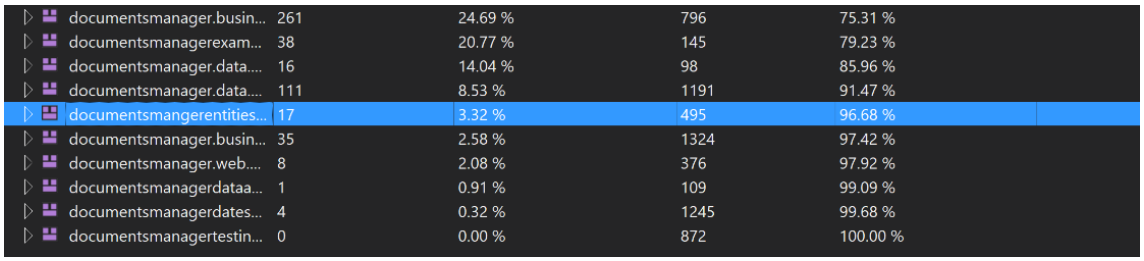


Se realizaron 248 pruebas a lo largo de del desarrollo de la Solución.

Porcentaje de cobertura:

El alto porcentaje es consecuencia del uso de la metodología de TDD.

En paquetes como el de Entidades y Data Access se tienen cifras muy cercanas al 100% de la cobertura del código.



documentsmanager.busin...	261	24.69 %	796	75.31 %
documentsmanagerexam...	38	20.77 %	145	79.23 %
documentsmanager.data....	16	14.04 %	98	85.96 %
documentsmanager.data....	111	8.53 %	1191	91.47 %
documentsmangerentities...	17	3.32 %	495	96.68 %
documentsmanager.busin...	35	2.58 %	1324	97.42 %
documentsmanager.web....	8	2.08 %	376	97.92 %
documentsmanagerdataa...	1	0.91 %	109	99.09 %
documentsmanagerdates...	4	0.32 %	1245	99.68 %
documentsmanagertestin...	0	0.00 %	872	100.00 %

Mocking

Los mocks son pruebas conocidas como "test doubles" (objetos que no son reales respecto a nuestro dominio, y que se usan con finalidades de testing) que existen para probar nuestros sistemas. Los más conocidos son los Mocks y los Stubs, siendo la principal diferencia en ellos, el foco de lo que se está testeando.

Los Mocks, nos permiten verificar la interacción del SUT (System under test) con sus dependencias. Los Stubs, nos permiten verificar el estado de los objetos que se pasan. Como queremos testear el comportamiento de nuestro código, utilizaremos los Mocks.

Utilizamos estas pruebas ya que buscamos probar objetos y la forma en que estos interactúan con otros objetos. Para ello crearemos instancias de Mocks, es decir, objetos que simulen el comportamiento externo, de un cierto objeto.

Realizamos pruebas “Mock” para probar los Controllers sin la necesidad de tener implementada la BusinessLogic que en definitiva va a utilizar. Generando un bajo acoplamiento entre una clase y sus dependencias.

Para realizar estas pruebas seguimos la metodología AAA: Arrange, Act, Assert. En la sección de Arrange, construiremos los el objeto mock y se lo pasaremos al sistema a probar. En la sección de Act, ejecutaremos el sistema a probar. Por último, en la sección de Assert, verificaremos la interacción del SUT con el objeto mock.

```
[TestMethod]
0 references | Fabian Grobert, 3 days ago | 1 author, 2 changes | 0 exceptions
public void GetAllEditorsOkTest()
{
    //Arrange
    var expectedEditors = GetFakeEditors();

    var mockEditorBusinessLogic = new Mock<IEditorsBusinessLogic>();
    mockEditorBusinessLogic
        .Setup(e => e.GetAllEditors())
        .Returns(expectedEditors);

    var controller = new EditorController(mockEditorBusinessLogic.Object);
    //Act
    IHttpActionResult obtainedResult = controller.Get();
    var contentResult = obtainedResult as OkNegotiatedContentResult<IEnumerable<EditorUser>>;

    //Assert
    mockEditorBusinessLogic.VerifyAll();
    Assert.IsNotNull(contentResult);
    Assert.IsNotNull(contentResult.Content);
    Assert.AreEqual(expectedEditors, contentResult.Content);
}
```