Universidad ORT Uruguay Facultad de Ingeniería

**Primer Obligatorio Diseño de Aplicaciones 2**

**Mayo 2018**

**Profesor: Gabriel Piffaretti**

**Federico Cetraro(193221)  
Fabian Grobert (194738)**

Índice

[Descripción general 3](#_Toc517332539)

[Justificación de Diseño 3](#_Toc517332540)

[Diagrama de Clases 4](#_Toc517332541)

[Capas Lógicas 5](#_Toc517332542)

[Diagrama de Secuencia 6](#_Toc517332543)

[Diagrama de entrega: 8](#_Toc517332544)

[Diagrama de Componentes 8](#_Toc517332545)

[Modelo de Tablas 9](#_Toc517332546)

[Decisiones de diseño 10](#_Toc517332547)

[Informe de Clean Code y pruebas. 15](#_Toc517332548)

[Clean Code 15](#_Toc517332549)

[TDD 18](#_Toc517332550)

[Evidencia de TDD 18](#_Toc517332551)

[Porcentaje de cobertura: 20](#_Toc517332552)

[Mocking 21](#_Toc517332553)

# Descripción general

# Descripción de Diseño

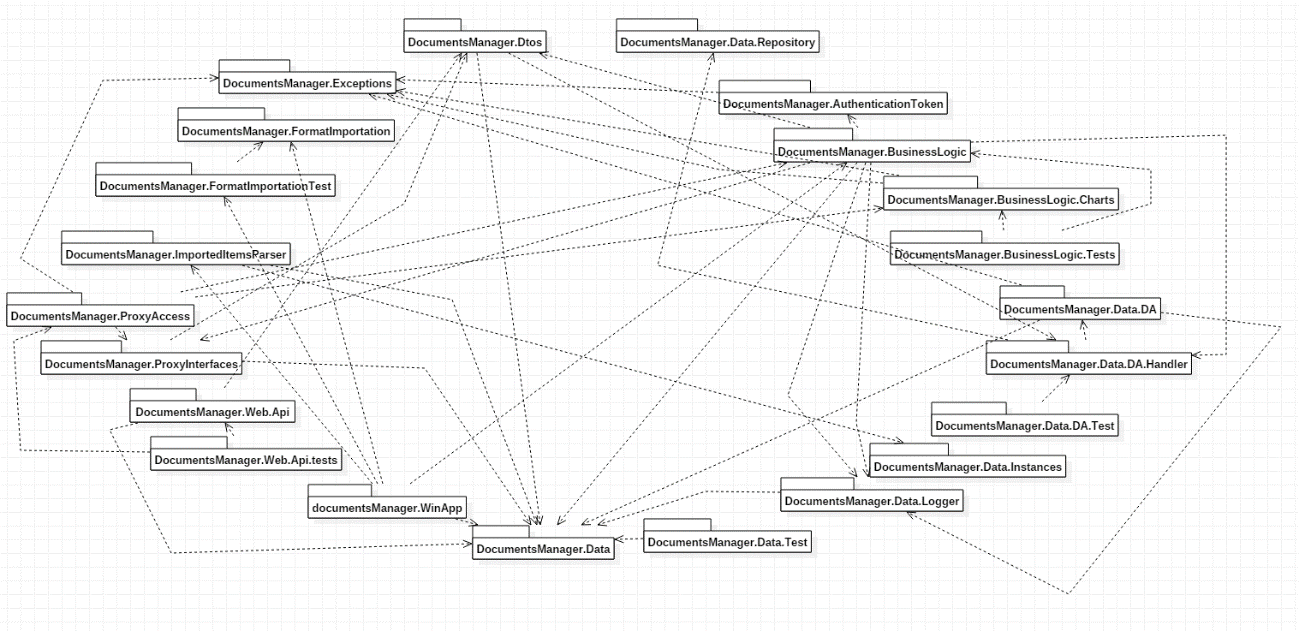
Para la mejor realización de este obligatorio, la primera decisión de diseño que

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 poder realizar las funcionalidades pedidas y por cada

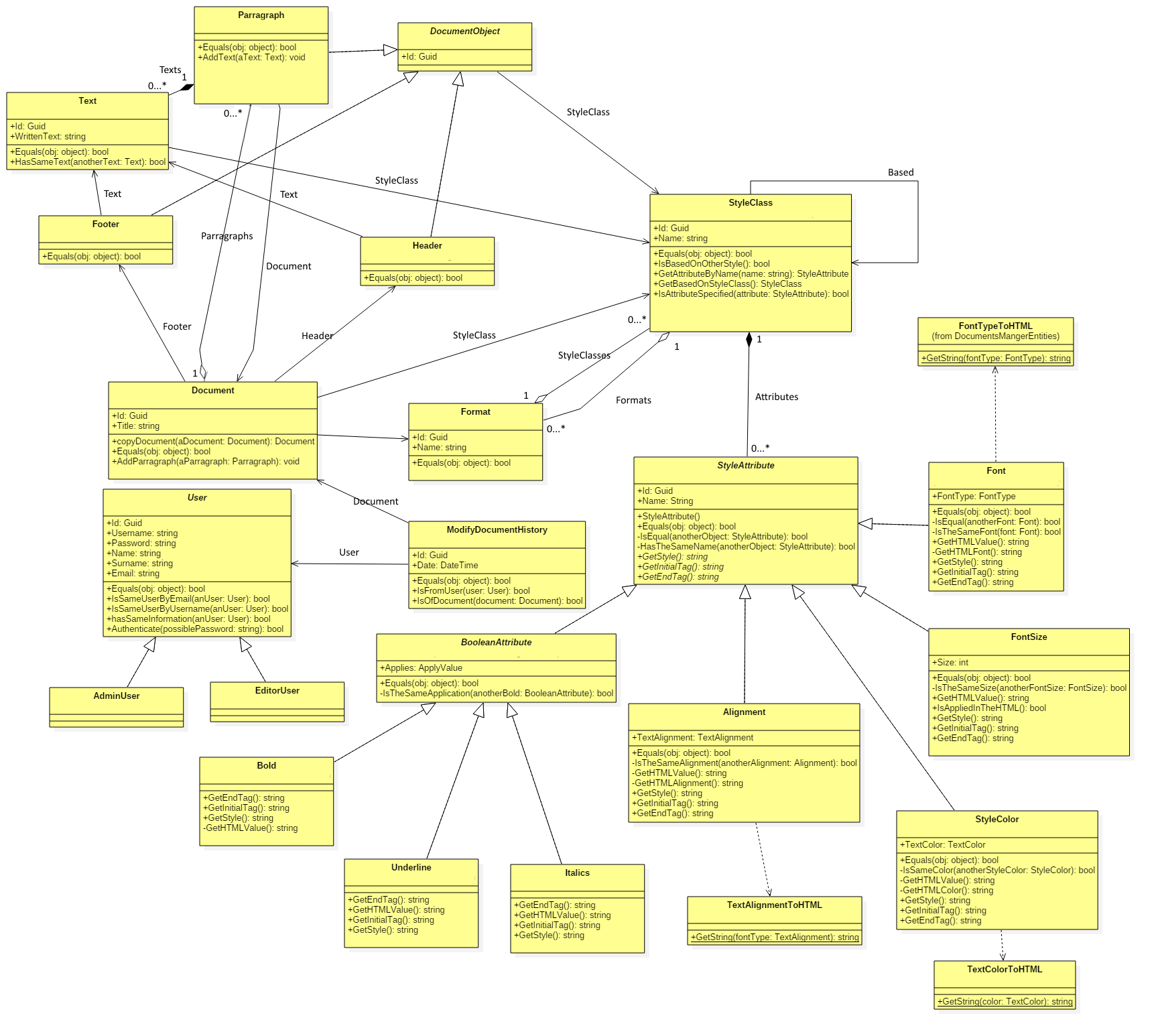
uno de estos se creó un paquete de prueba.

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 funcionalidades 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.

## 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 responsable 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.

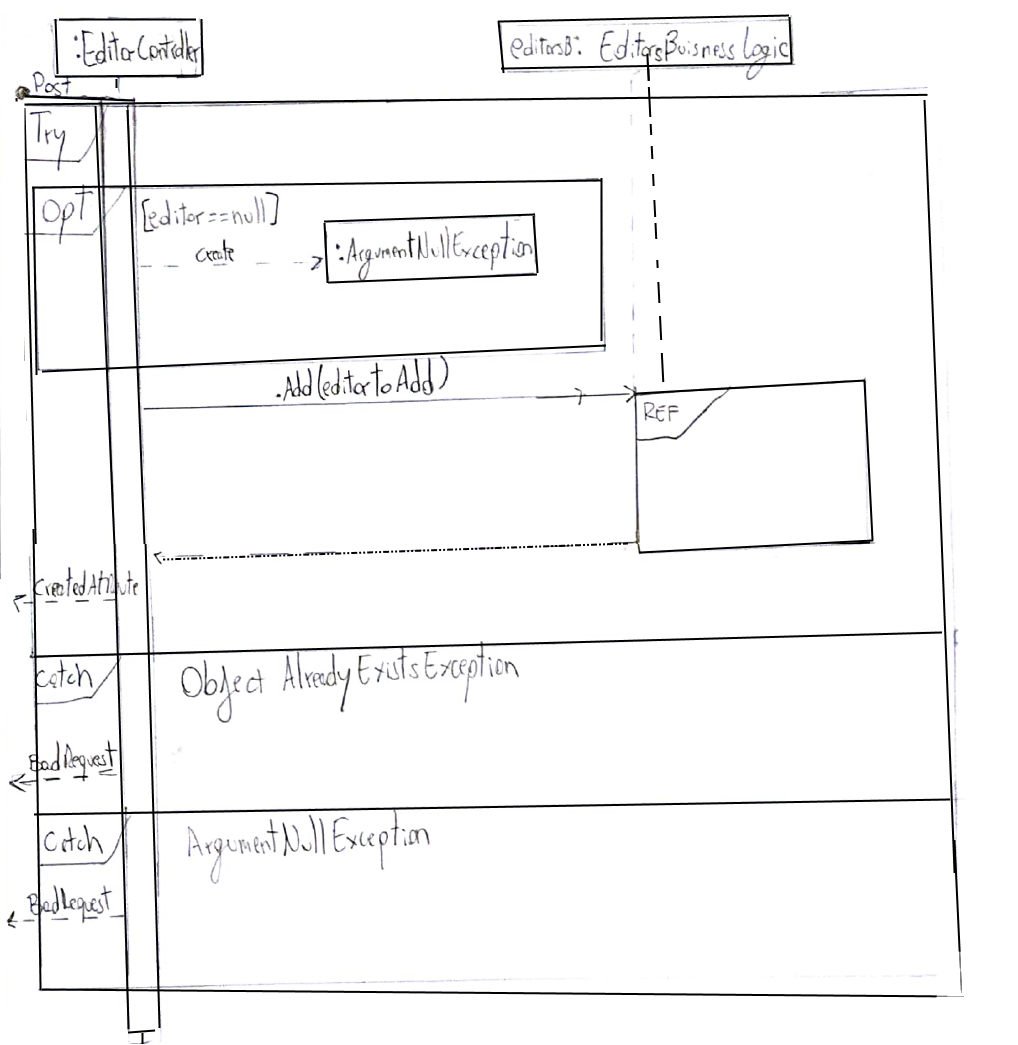
La clase Unit of Work es fundamental, ya que maneja el contexto, así de esta forma nos aseguramos de que todos los repositorios usen el mismo contexto.

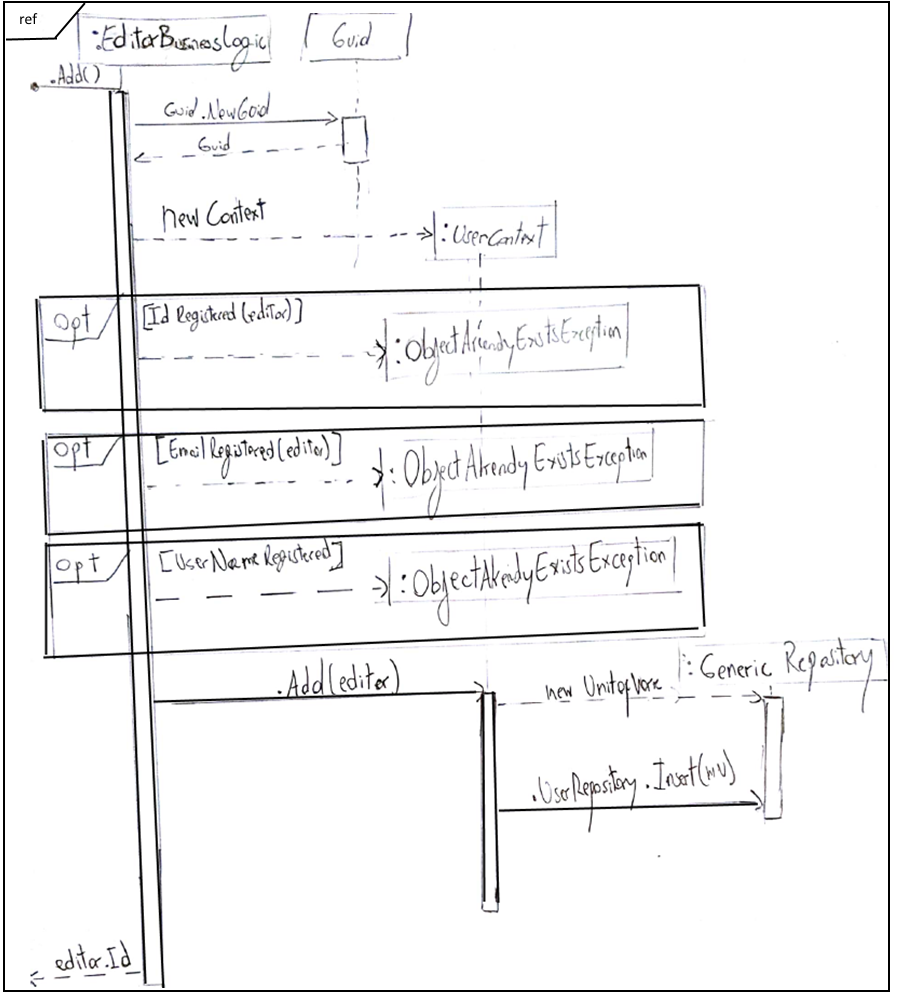
A su vez, también existe un repositorio genérico, debido a que existen consultas básicas que se repiten e incluir estas consultas en cada repositorio específico significaría repetir una gran cantidad de código. Se puede apreciar las operaciones básicas como el get, insert, delete, exists, entre otros.

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

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

## Diagrama de Secuencia

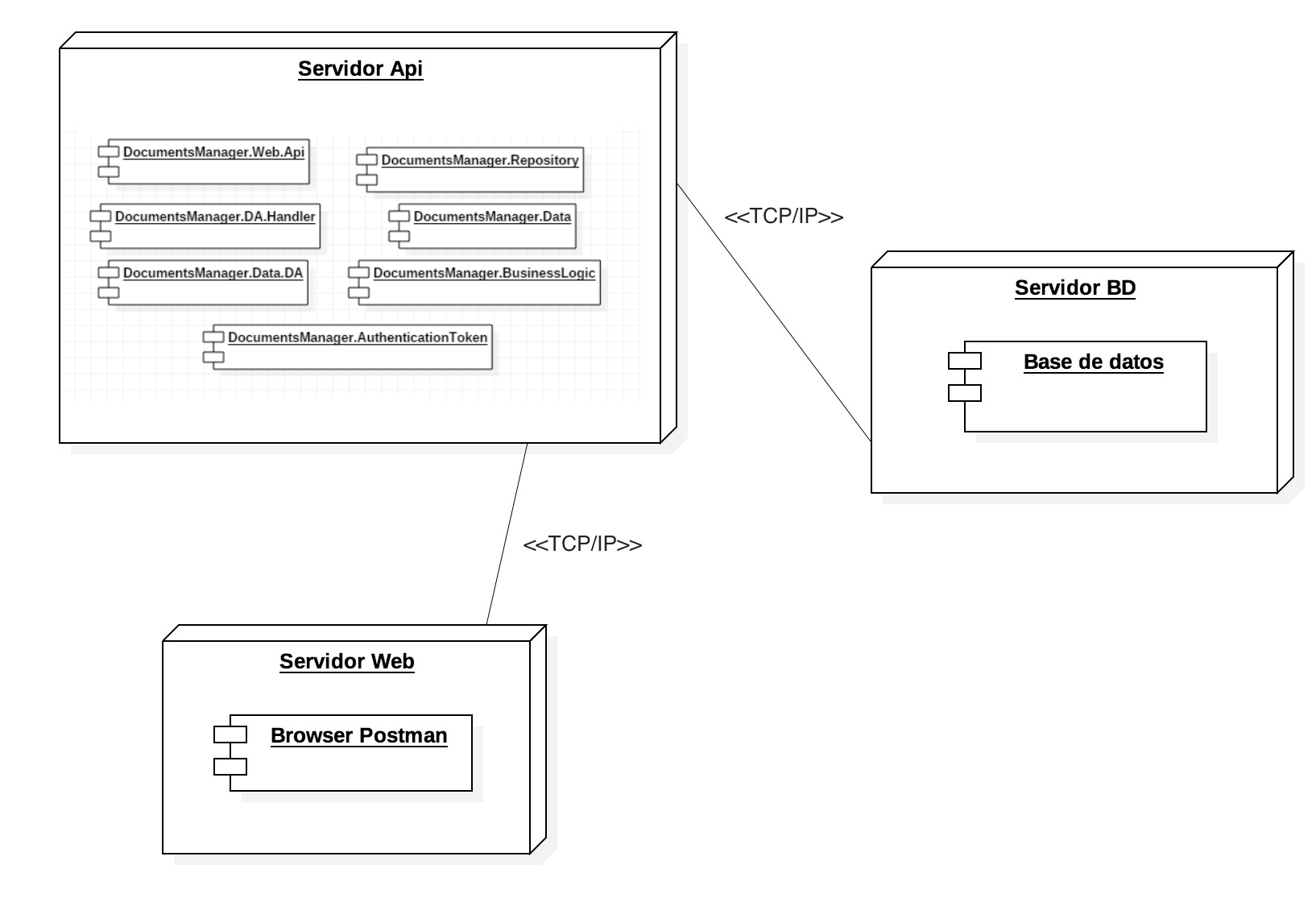




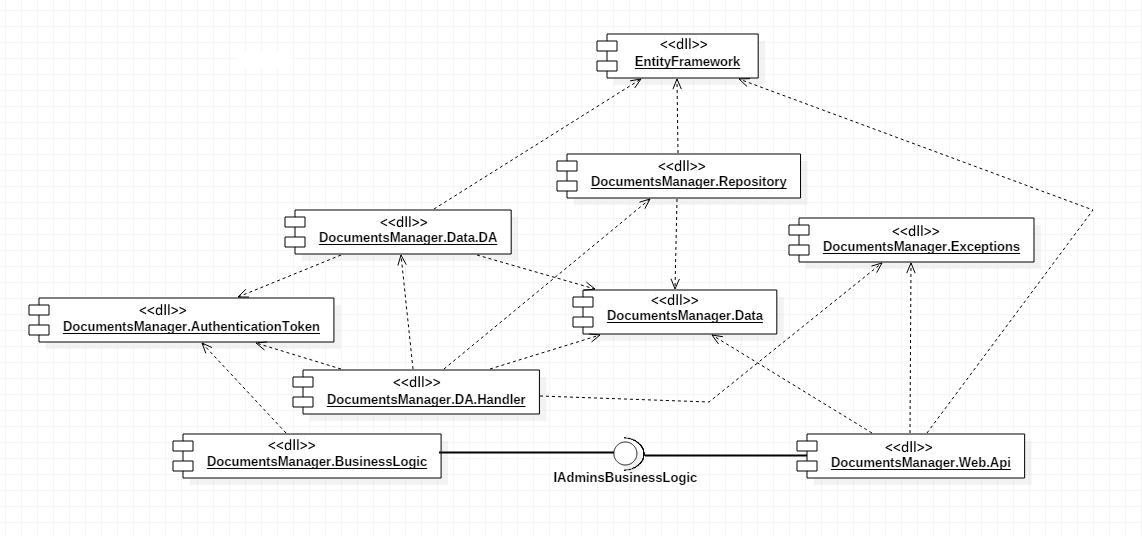
Para culminar, añadimos los siguientes componentes de entrega. No se incluyen los paquetes de prueba ya que creemos que estos son importantes para quien desarrolla la solución y no para el cliente que la consume.

DocumentsManager.Web.Api

## Diagrama de entrega:



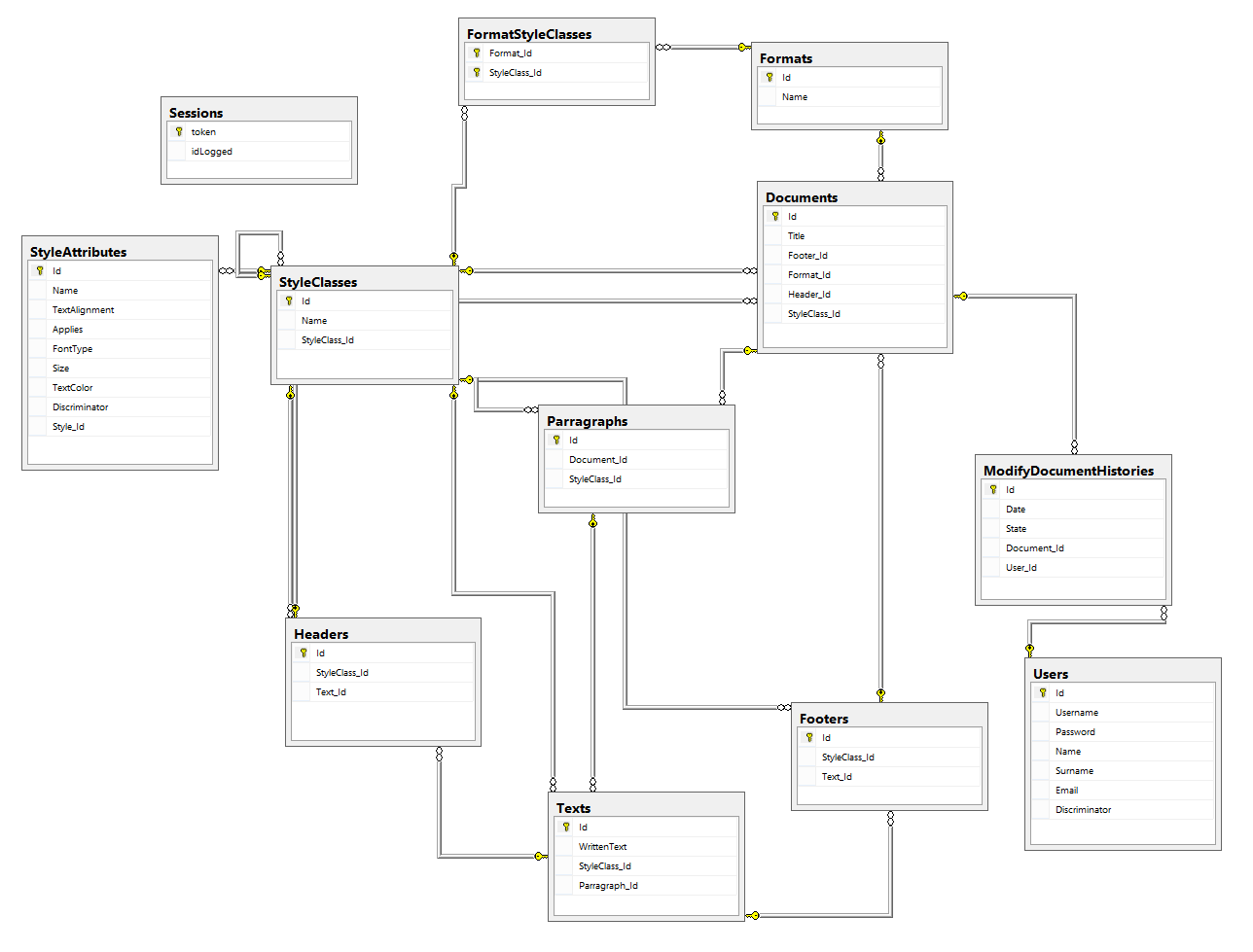
## Diagrama de Componentes



## 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

## Decisiones de diseño

**Patrón Proxy.**

El patrón Proxy es un patrón perteneciente a la familia de patrones estructurales. Su propósito es proporcionar un intermediario de un objeto para controlar el acceso al mismo.

Implementamos un tipo específico de proxy llamado proxy de protección. Este subtipo del patrón tiene la particularidad de que se está controlando el acceso sobre el objeto original y no una copia local (proxy remoto) o un objeto de costos a demanda costo (proxy virtual).

Lo utilizamos para que funcione de intermediario entre la WebApi y la lógica de negocios. Los controllers usan la clase proxy para utilizar las funcionalidades de la lógica de negocios, pasando por los controles que proporciona proxy.

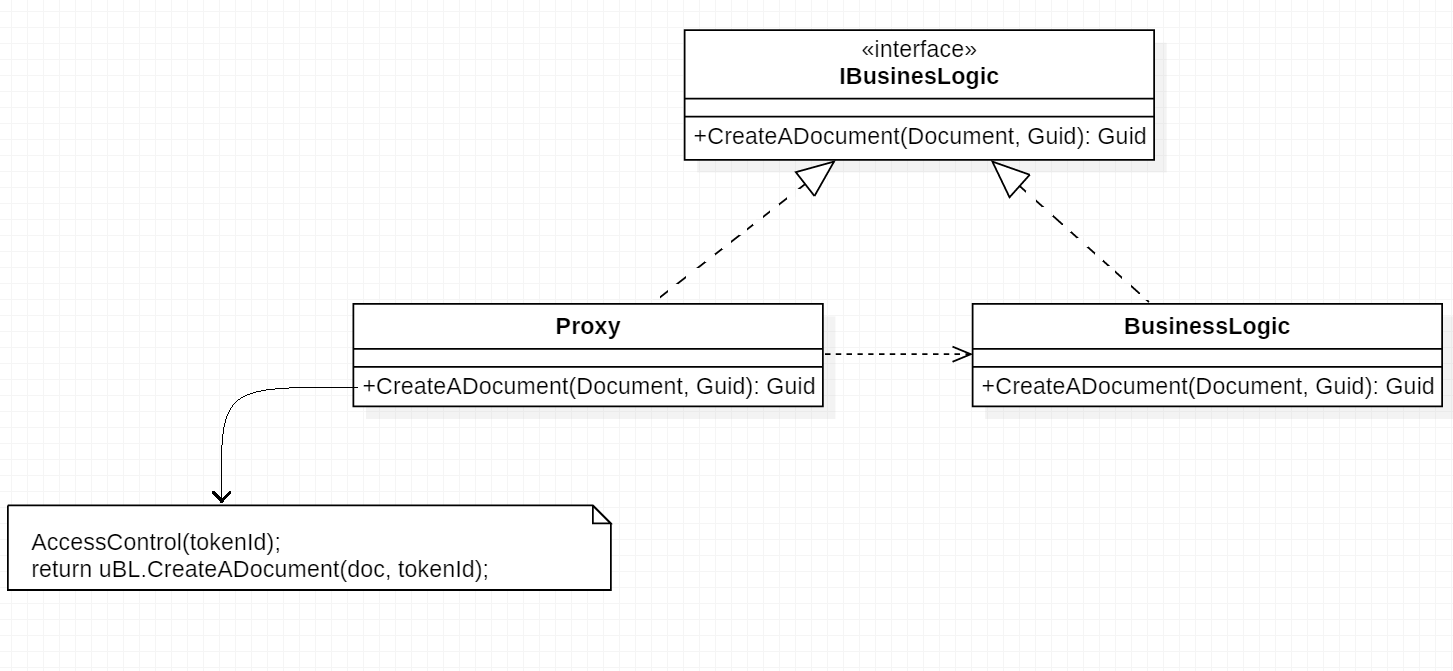


Figura que representa a nivel general nuestra implementacion de proxy en el Sistema (no es una representación exacta del codigo, es reducida para facilitar la comprensión siendo más intuitiva).

En la figura descripta anteriormente, se intenta representar nuestra implementación del patrón proxy pero muy general.

La clase BusinessLogic, representa un conjunto de muchas clases(DocumentsBusinessLogic, UserBusinessLogic, AdminBusinessLogic, StyleClassBusinesLogic, FormatBusinessLogic, entre otras) y lo mismo sucede con el interfaz IBusinessLogic(IDocumentsBusinessLogic, IUserBusinessLogic, IAdminBusinessLogic, IStyleClassBusinesLogic, IFormatBusinessLogic, entre otros). Se separaron en distintas clases con el objetivo de reducir la carga y complejidad de cada una y no concentrar la lógica de negocios en una sola clase.

Proxy al igual que las BusinessLogic implementan los interfaces IBusinessLogic. La diferencia entre sus implementaciones es que por ejemplo para el caso representado de CreateADocument, la implementación en proxy, se llama a una función que retorna void. Esta controla que el usuario llevando a cabo esta acción siga registrado en la Base de datos y su sesión no expiró (es decir que siga existiendo su instancia de sesión).

Luego de realizar esos controles, si no se tiró una excepción personalizada (por ejemplo, SessionExpiredException, con un mensaje interno: "La sesión actual expiró o no existe.") se realiza la llamada al método CreateADocument de la clase UserBusinessLogic. Esta implementación se encarga de crear el documento con sus partes y agregarlo a la base de datos mediante llamadas a métodos del contexto y de otras clases de la lógica de negocios. Finalmente se retorna el Guid con el que el Document fue agregado a la base de datos.

De una forma muy similar funciona el proxy implementado con las otras funcionalidades. En muchos casos estableciendo otros controles con diversas excepciones personalizadas para distintos casos. Para luego llamar a la clase que contenga la lógica de negocios correspondiente a la funcionalidad que se ejecutó.

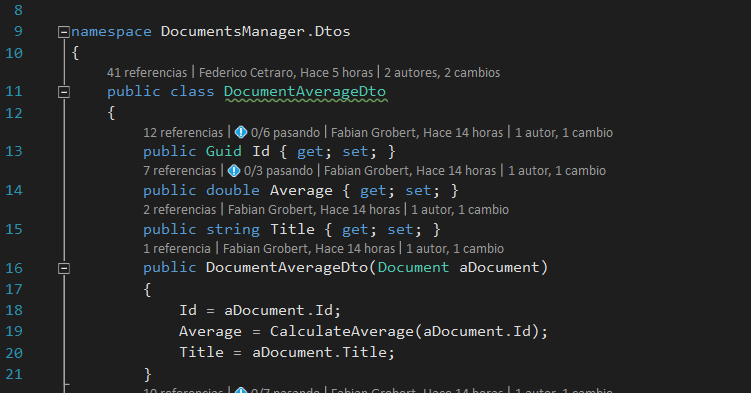
No es poco destacar que al utilizar este patrón también se logró desacoplar las relaciones entre la WebApi y la BusinesLogic, ya que anteriormente, en la WebApi cada controller tenia diversas instancias de las diferentes clases de la lógica de negocios. Ahora cada controller tiene una referencia a un proxy y es el proxy el que referencia a la lógica, creando así un único punto de entrada a la lógica de negocios.

**Creación de Dtos**

Se decidió crear Dtos para facilitar la manipulación de ciertos objetos. Estos Dtos son variaciones de las entidades del sistema. Los Dtos se crean a partir de objetos del sistema (tienen un constructor con parámetros que recibe un objeto no Dto y lo “convierte” en uno). Estos Dtos muestran menos o diferente información a la que contiene el objeto real.

En el caso en que muestran menos información, se debe a que creemos que cierta información no incluida en el Dto no debería llegar al FrontEnd, de manera que le llega una versión “reducida” del objeto real con la información que si creemos relevante.

En el caso en que muestran diferente información, lo relacionamos con SRP (Single Responsibility Principle), ya que consideramos que la clase del domino no debería contener cierta información al no es su responsabilidad. Por ejemplo, Document tiene un Dto llamado DocumentAverageDto.



Porción de Código extraída del Proyecto DocumentsManager.Dtos que muestra una definición de un Dto.

Este Dto se crea a partir de un objeto Document y tiene como información el id del documento original, su título, y el promedio de sus valoraciones (Cómo se muestra en la imagen). Esta última se calcula dentro del constructor, recorriendo las valoraciones y realizando el promedio. Luego es enviado al FrontEnd con dicha información.

De esta forma podemos calcular el porcentaje sin almacenarlo en la clase y romper con el principio mencionado de responsabilidad única.

**Patrón Strategy.**

El patrón strategy pertenece a la familia de patrones de comportamiento. Este permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo en tiempo de ejecución según sus necesidades.

Al 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.

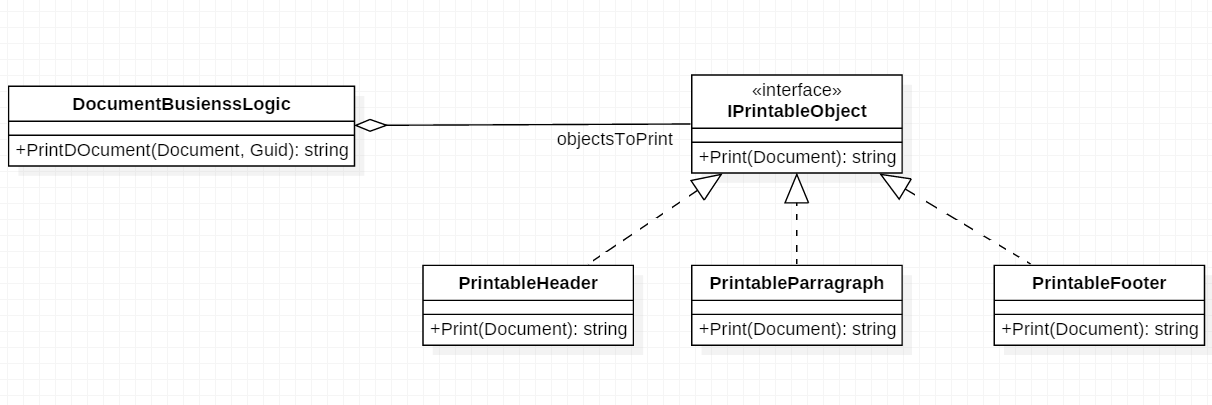


Figura que representa a nivel general nuestra implementacion de proxy en el Sistema (no es una representación exacta del codigo, es reducida para facilitar la comprensión siendo más intuitiva).

Al utilizar el interfaz mencionado, se puede tratar a los objetos de manera indiferente sin importar su tipo, ya que todo implementan a su forma los métodos que se llaman, cumpliendo de esta manera con el principio de Liskov.

También de esta manera, si el día de mañana se desea añadir a documento un nuevo tipo de parte, este para imprimirse debe cumplir con el contrato de IPrintableObject, de esta forma cumpliendo con OCP (Open Close Principle), siendo abierto para la extensión y cerrado para la modificación.

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 por su tipo.

Pero estos no son los únicos puntos en los que intentamos aplicar principios SOLID entre otros. A lo largo de todo el desarrollo tuvimos en cuenta dichos principios. Por ejemplo, a la hora de implementar la funcionalidad de agregar amigos, creamos una clase friendship, la cual entre otros atributos tiene uno que particularmente describe que tipo de relación es.



Porción de Código extraída del Proyecto DocumentsMangerEntities de la clase Friendship

Como indica la imagen, el atributo State, es un enum FriendshipState, el cual actualmente puede ser Request o Friend. Pero si se desea implementar por ejemplo una funcionalidad de Bloquear a un usuario o de Follow (Seguir), es solamente agregar el enum y la lógica deseada Minimizando de esta forma el impacto de cambio.

**Importación de formatos**

En esta entrega se implementó un lector de importadores que permiten agregar nuevos formatos cargándolos a partir de diferentes fuentes. Actualmente el sistema soporta agregar nuevos formatos desde 2 tipos de fuentes, pero, se diseñó de tal manera de que sea sencillo añadir nuevos tipos de fuentes.

Para modelar esta solución utilizamos también utilizamos el patrón de comportamiento “Strategy”.

Este patrón nos permite encapsular las distintas soluciones en una misma jerarquía y el cliente puede elegir cual utilizar en tiempo de ejecución.

En primer lugar, creamos la interfaz “IFormatImportation”. En esta interfaz se define los métodos que cada importador particular debe implementar para realizar la importación de productos.

Para añadir un nuevo tipo de fuente de importación se definieron 2 métodos que deben ser implementados por cualquier importador que se quiera leer en nuestra solución.

En primer lugar, tenemos el método “GetParameters”. Gracias a este método podemos saber que necesita cada importador para poder trabajar.

Para poder cumplir con el requerimiento solicitado se buscó que nuestra solución sea amplia para cualquier tipo de fuente, por lo que permitimos a los importadores pedirnos diferentes tipos de parámetros.

Actualmente el sistema cuenta con 2 tipos de parámetros permitidos (Strings y Paths Locales), pero la solución diseñada es abierta a la extensión permitiendo añadir nuevos parámetros de una manera sencilla que se detallara a continuación:

Para añadir un nuevo tipo de fuente hay que añadir en primer lugar añadir un nuevo "ParamerType" (enum encargado de definir todos los tipos de importación soportados en el sistema).

Una vez añadido solo resta añadir la lógica necesaria para que a partir de ese tipo de fuente se autogenere la Interfaz gráfica para que el usuario final nos pueda facilitar los datos necesarios para importar los formatos de dicha fuente.

Una vez que ya contamos con la información necesaria para poder importar los formatos solo basta llamar al importador concreto por medio del método “ImportFormats”

Como resultado de este método se espera obtener los formatos de una manera que nuestro sistema pueda interpretarlos y añadirlos en el sistema.

Como se dijo anteriormente se implementaron 2 lectores de Formatos concretos distintos para esta entrega.

En primer lugar se implementó el importador desde archivos .xml que únicamente necesita la ruta del archivo del tipo parámetro “Path”.

JSON

Para añadir un nuevo lector de Formatos se debe implementar la interfaz “IFormatImportation” descripta anteriormente y para utilizarla en el sistema bastara con tener el .dll sin la necesidad de recompilar toda la solución.

Esto es posible gracias al uso de Reflection.

Reflection es la habilidad de un programa de auto examinarse para buscar ensamblados en tiempo de ejecución. De esta forma, se pueden utilizar los métodos de los importadores específicos en tiempo de ejecución, sin haber tenido que hacer referencia al ensamblado de la implementación del importador.

Para poder utilizar estas herramientas, utilizamos el namespace

System.Reflection. Este contiene las herramientas necesarias para manejar los ensamblados. Reflection permite crear dinámicamente instancias de un tipo, obtener el tipo de un objeto existente e invocarle métodos de forma dinámica.

Una vez que tenemos cargados en el sistema todos los tipos de lectores cargados de sus .dll previamente gracias a Reflection el cliente puede escoger en tiempo de ejecución cual utilizar.

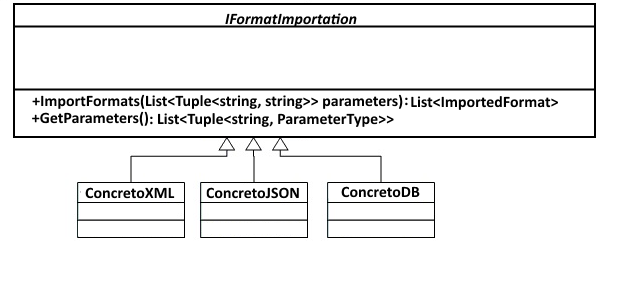
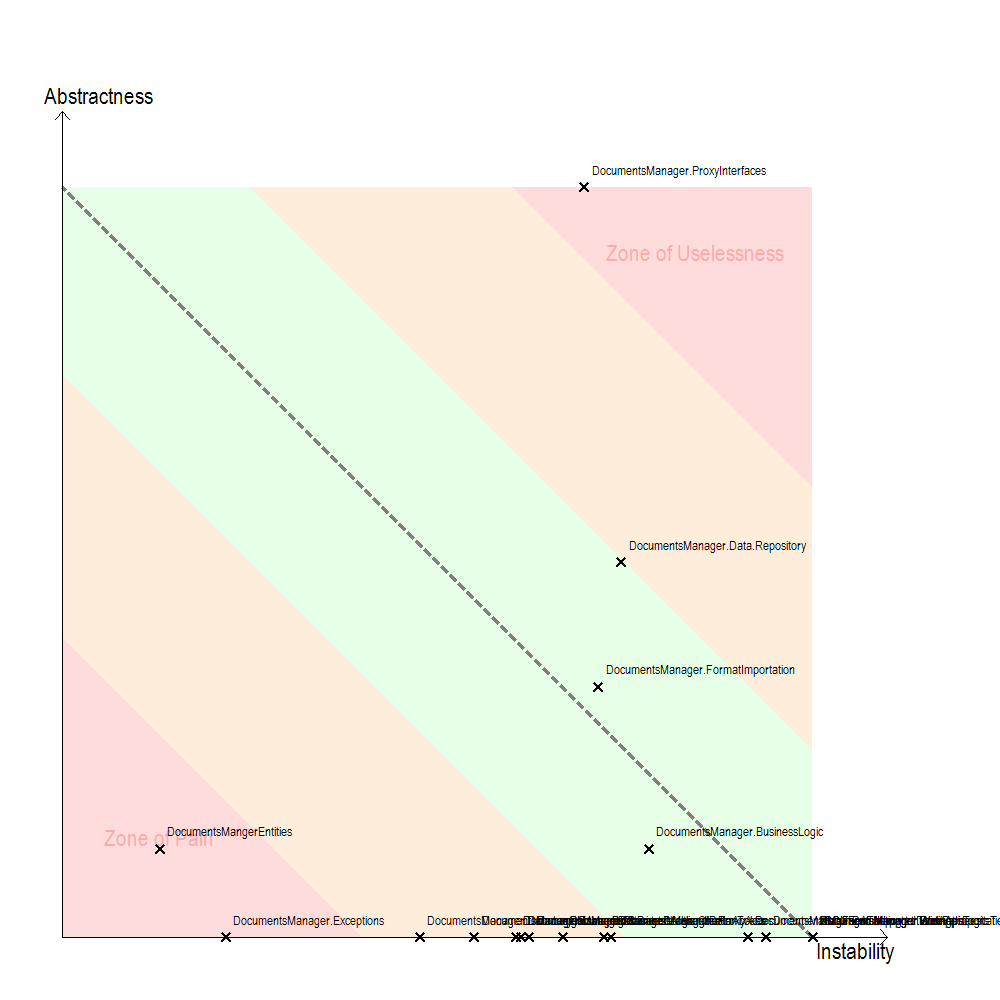


Diagrama de la implementación de la solución. En ese caso se implementaron 3 importadores distintos y todos deben cumplir con la interfaz establecida por “IFormatImportation”. Luego el cliente podrá escoger que importación realizar en tiempo de ejecución. La imagen es una representación de lo realizado en el código. En el código se implementó el xml y json, escritos en idioma inglés.

Metricas

Las métricas a nivel de diseño son importantes porque nos dan un análisis cuantitativo de la solución. Ayudan a evaluar de alguna manera el diseño que desarrollamos.

Para este caso en particular decidimos basarnos en la gráfica que muestra la inestabilidad y abstracción de paquetes. El resultado sin un análisis carece completamente de sentido. Por más que los valores estén abarcados dentro del rango aceptable, no significa un diseño aceptable, de la misma forma que si se encuentran en rojo indica que sea completamente incorrecto. Lo que, si logra, es alertarnos y avisarnos en que casos debemos detenernos a tiempo y analizar el porqué de los valores.

Se puede visualizar la gráfica generada por NDepend a continuación.

# Informe basado en métricas

Se puede observar que el paquete DocumentsManager.Exceptions no es para nada abstracto al presentar únicamente clases concretas que heredan de Exception. A su vez, es una clase muy estable ya que no depende de nadie. Si este proyecto cambiase en el futuro tendría un enorme impacto de cambio sobre la solución ya que la mayoría de los paquetes dependen de él.

Se puede ver como una desventaja, pero si se estudia, uno se da cuenta que no existen razones justificadas correctamente para que este paquete cambie, por eso no lo consideramos como un problema grave, más allá de que esté marcado como en peligro.

También se puede ver que el paquete que contiene a las clases la lógica de negocio, DocumentsManager.BusinessLogic está bastante estable.

En lo que refiere al DocumentsManager.FormatImportation, se encuentra en zona correcta. Esto se debe a que es algo abstracto por contener a la interfaz que implementan los distintos importadores particulares.

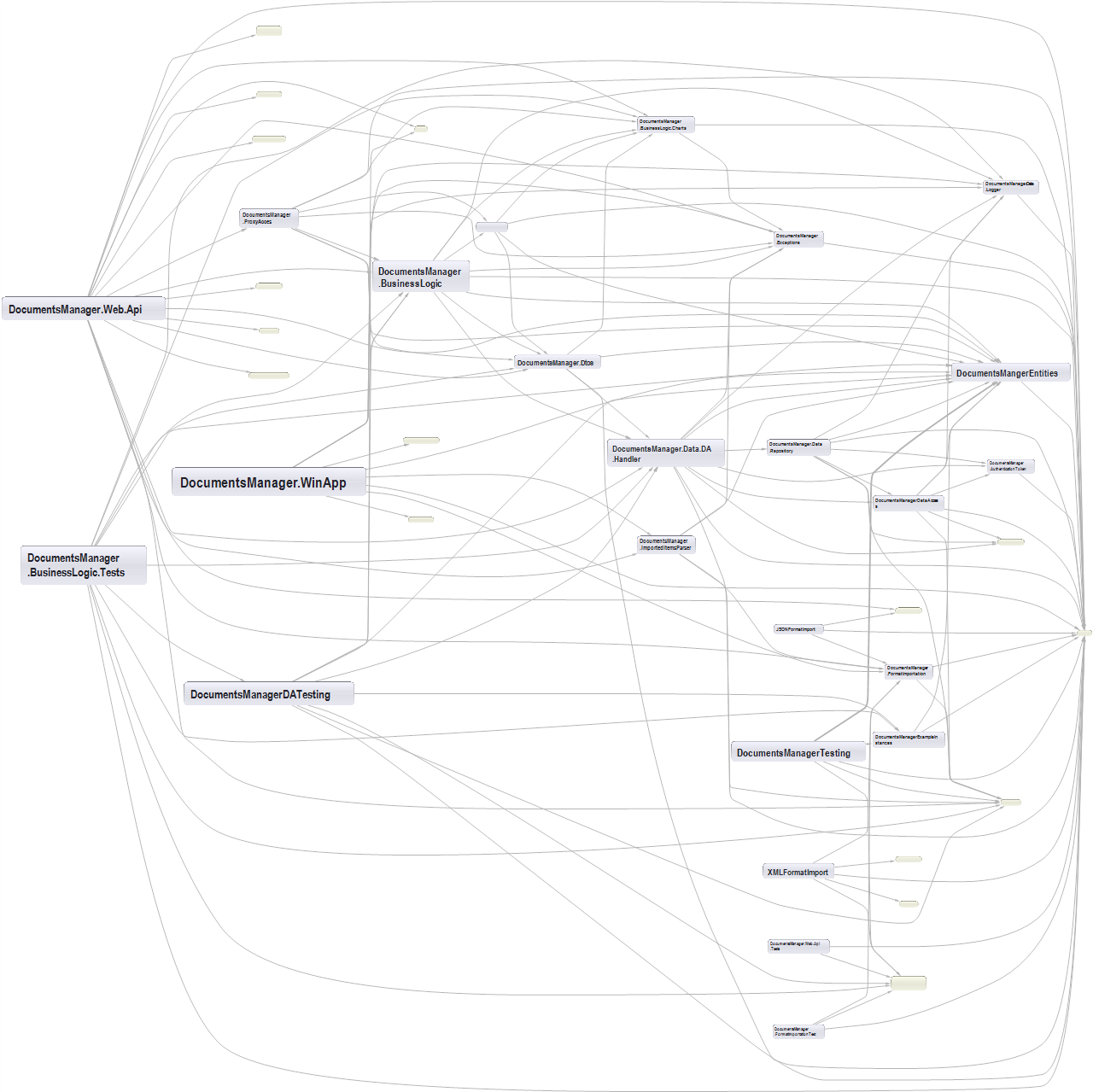
El paquete de DocumentsManagerEntities se ve en zona roja, lo cual probablemente sea porque es usado a lo largo de todo el proyecto y todos lo referencian. Además, es poco abstracto.

Otro de los paquetes en zona critica es el de ProxyInterfaces. Este paquete es sumamente abstracto ya que solo posee interfaces, con la definición de los contratos que deben cumplir las diferentes clases de la lógica para poder pertenecer a la solución.

Además se puede ver el paquete DocumentsManager.Data.Repository el cual está en el borde de la zona aceptable. Este es un 50% abstracto aproximadamente, ya que es el que contiene al UnitOfWork y repository. Este posee el interfaz de UnitOfwork. Está en el medio con cierta tendencia a cercarse a la zona de poco uso ya que este paquete sol es referenciado desde el paquete de DocumentsManager.DA.Handlers (donde están los contextos), para agregar los objetos a la base de datos.

Por último, podemos ver que el paquete de DocumentsManager.Web.Api es uno de los más inestables ya que depende de la mayoría de los otros paquetes. Por otro lado, el mismo no es nada abstracto, debido a que no tiene ninguna abstracción. Este paquete es el más fácil de cambiar ya que nadie depende de él. Lo mencionado hace que la única razón de cambio sea una razón propia del paquete.

Por dicho motivo, en cuanto a inestabilidad y abstracción es uno de los mejores posicionados.



Grafo de dependencias generado por NDepend

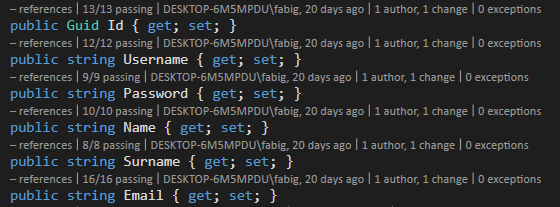
# Informe de Clean Code y pruebas.

## **Clean Code**

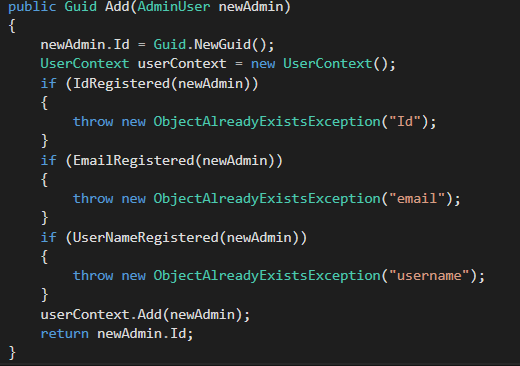
La implementación de este proyecto fue realizada siguiendo las prácticas sugeridas en el libro de Clean Code “Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*.”.

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.



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

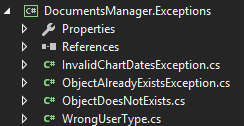
Para las funciones seguimos la misma práctica de los nombres descriptivos y claros. Buscamos reducir al máximo el número de argumentos de cada función, es por eso que la función que recibe más parámetros es 3.



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.

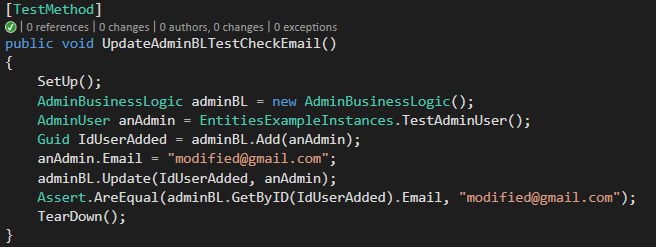


*Según el error manejado genera una nueva excepción.*

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.



*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.

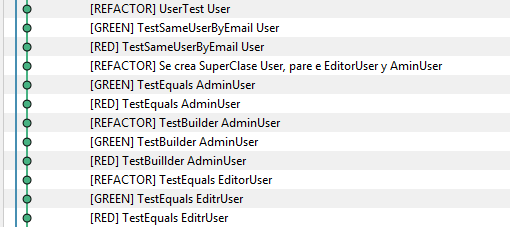
El desarrollo del software a partir de las pruebas unitarias nos ayudó a reducir el número de errores.

## Evidencia de TDD

En el repositorio se puede ver la evidencia del uso de TDD. Se ven los pasos que

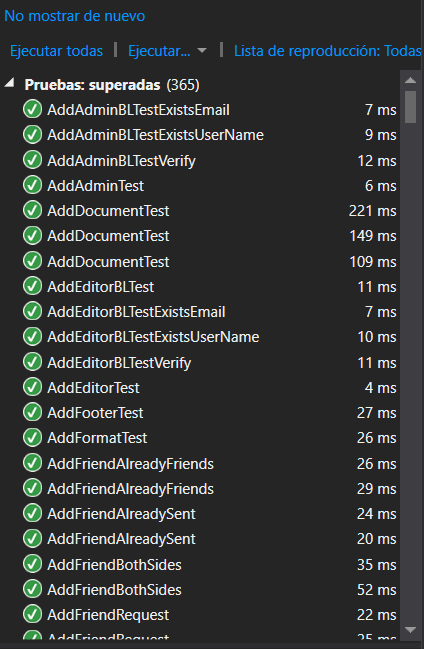
Utilizamos mencionados anteriormente agrupados en 3 etapas distintas:

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.

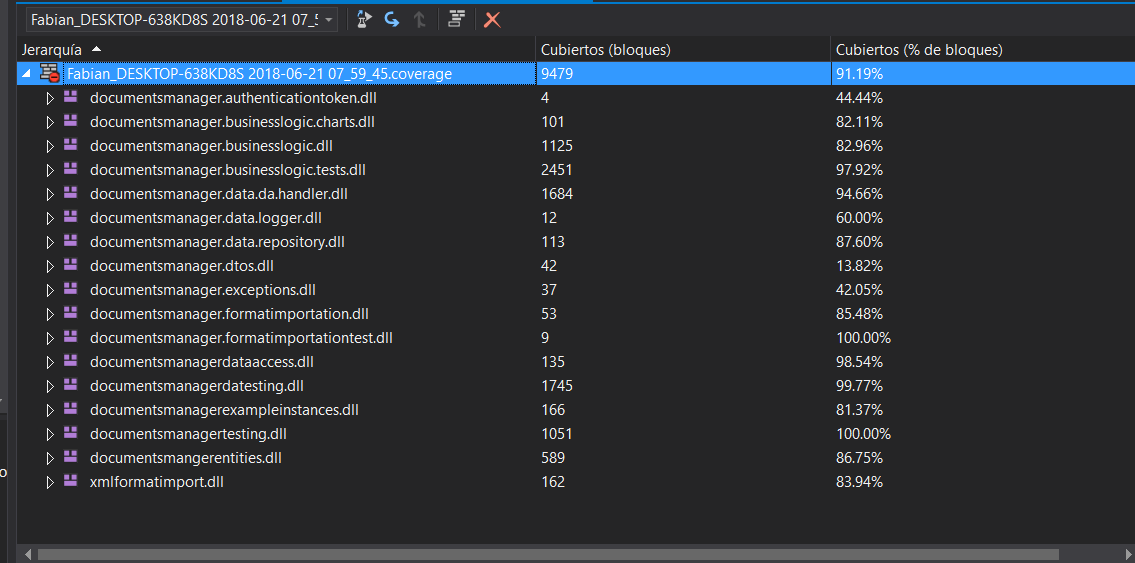


Se realizaron 365 pruebas a lo largo de del desarrollo de la Solución.

## 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.



## Mocking

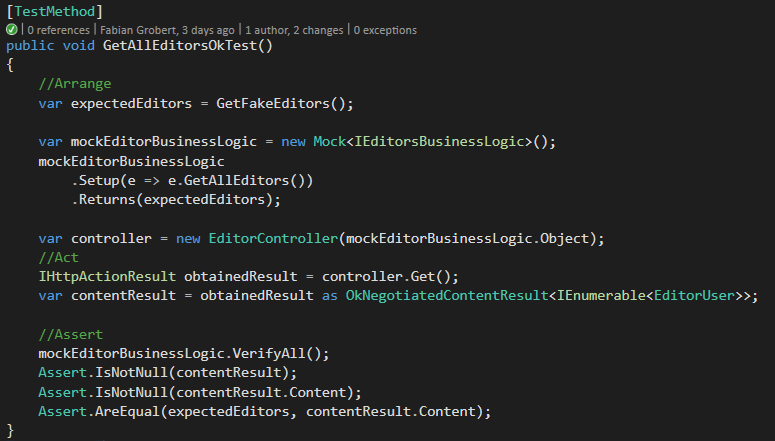
Los mocks son pruebas conocidas como "test doubles" (objetos que no son reales respecto a nuestro dominio, y que se usan con finalidades de testing) que existen para probar nuestros sistemas. Los más conocidos son los Mocks y los Stubs, siendo la principal diferencia en ellos, el foco de lo que se está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



# Manual de Instalación

Debido a que el desarrollo de este obligatorio es la realización de una WebApi, donde hay una capa de servicios y clientes que la consumen, es que el de la instalación es un poco más elevado a lo que se venía trabajando anteriormente. A continuación, se explicará de qué manera se debe instalar el sistema para su utilización.

Para empezar, se debe configurar el connection string para el nombre de instancia que tenga el sql server en la pc donde se desee instalar. Luego, se deben habilitar los servicios de IIS en la pc donde se vaya a instalar. Una vez habilitados se debe copiar la carpeta del Release donde se encuentra el obligatorio a la ruta: C:\inetpub\wwwroot. Para la creación del nuevo sitio que se desea utilizar, se debe abrir el administrador de IIS y en la lista de los sitios agregar un nuevo sitio web. Los datos que se deben completar son la ruta física, que se debe seleccionar la carpeta que se pegó en wwwroot; ingresar un puerto (en las computadoras de la facultad se aconseja un número mayor a 6000). Por último, se agrega el nombre del sitio, a elección.

Una vez realizada esta primera etapa se debe habilitar los servicios de sql server, posterior a importar la base de datos con los datos generados. Esto se hace seleccionando en el nombre de la instancia “Seguridad” y luego “Inicios de sesión”, “Nuevo inicio de sesión”. En el nombre del sitio de debe ingresar “IIS APPPOOL \” seguido del nombre que le asignamos al sitio agregado anteriormente. Por último, debemos chequear en los roles los siguientes: dbcreator, public y sysadmin.

En el caso de que se quiera ingresar el sitio publicado desde otra computadora, se debe deshabilitar el firewall desde el panel de control. Una vez así, ingresando la IP con el nombre del sitio publicado se podrá acceder desde otra pc.

Esta guía fue detallada para acceder al cliente web que está consumiendo nuestra capa de servicios. Sin embargo, para ejecutar la aplicación de escritorio el procedimiento es más sencillo. Debemos ejecutar el archivo de nombre “DocumentsManager”