

Universidad ORT Uruguay

Facultad de Ingeniería

Obligatorio 1 para Diseño de Aplicaciones 2

Ingeniería en Sistemas

Docente Ignacio Valle

Año 2018

Genaro Girardi (199394)

Matías Hernández (169236)

Descripción del Trabajo	4
1. Diseño del obligatorio	5
1.1. Arquitectura	5
1.1.1. Diagrama de paquetes	6
1.1.2. Diagrama de Componente	7
1.2. Domain	8
1.3. WebApi	9
1.3.1. Controllers	9
1.3.2. UML de los métodos del controller	9
1.3.3. Muestra de los Endpoints	10
1.3.4. Models	12
1.4. Service	14
1.4.2 ServiceImp	15
1.5. Repository	27
1.5.1. RepositoryFactory	27
1.5.2. Repository	28
1.5.3. RepositorySQLServer	28
Configurations	29
Fluent Api	29
2. Justificaciones de Decisiones de Diseño Tomadas	31
2.1. Arquitectura	31
2.2. Domain	33
2.3. WebApi	35
2.4. Service	36
2.5. Repository	40

2.5.2. Utilización de un RepositoryFactory	41
3. Manejo de excepciones	41
4. Manejo de Tablas de la Base de Datos	42
5. Justificación de Clean Code	44
6. Pruebas	46
6.1. UML de clases de prueba	46
6.1.1. DomainTest	46
6.1.2. RepositoryTest	46
6.1.3. ServiceTest	47
6.2. Reporte de pruebas	48
7. Instalación	51
8. Justificaciones de Diseño de las Nuevas Funcionalidades	53
8.1. Red Social	53
8.2. Nuevos Estilos	56
Agregar el estilo Borde y el tipo de letra Verdana al sistema	56
8.3 Sistema de Logs	56
8.4 Importador de Formatos	57
9. Informe de Métricas	59
9.1 Lista de Métricas	59
9.2 Gráfica de Abstractness vs Instability	62
9.3 Grafo de Dependencia	64
9.4 Principios de Paquetes	65
10. FrontEnd (Angular)	66

Descripción del Trabajo

Desarrollamos un programa de edición de Documentos y de visualización de los Documentos según determinadas preferencias de clase de estilos y formato. El programa permite a los usuarios con permisos de administrador mantener los datos de los usuarios y formatos y la obtención de datos estadísticos respecto a la creación de documentos por usuario y las modificaciones a documentos realizadas por usuarios por día. Los usuarios con permiso de editor, por otro lado, pueden mantener sus documentos, filtrarlos y ordenarlos, además de personalizar las clases de estilo que utilizan.

La visualización de los Documentos, en la versión actual es implementada con etiquetado HTML, por lo cuál nuestro programa posee una funcionalidad que permite visualizar el documento según la representación HTML de sus estilos. La

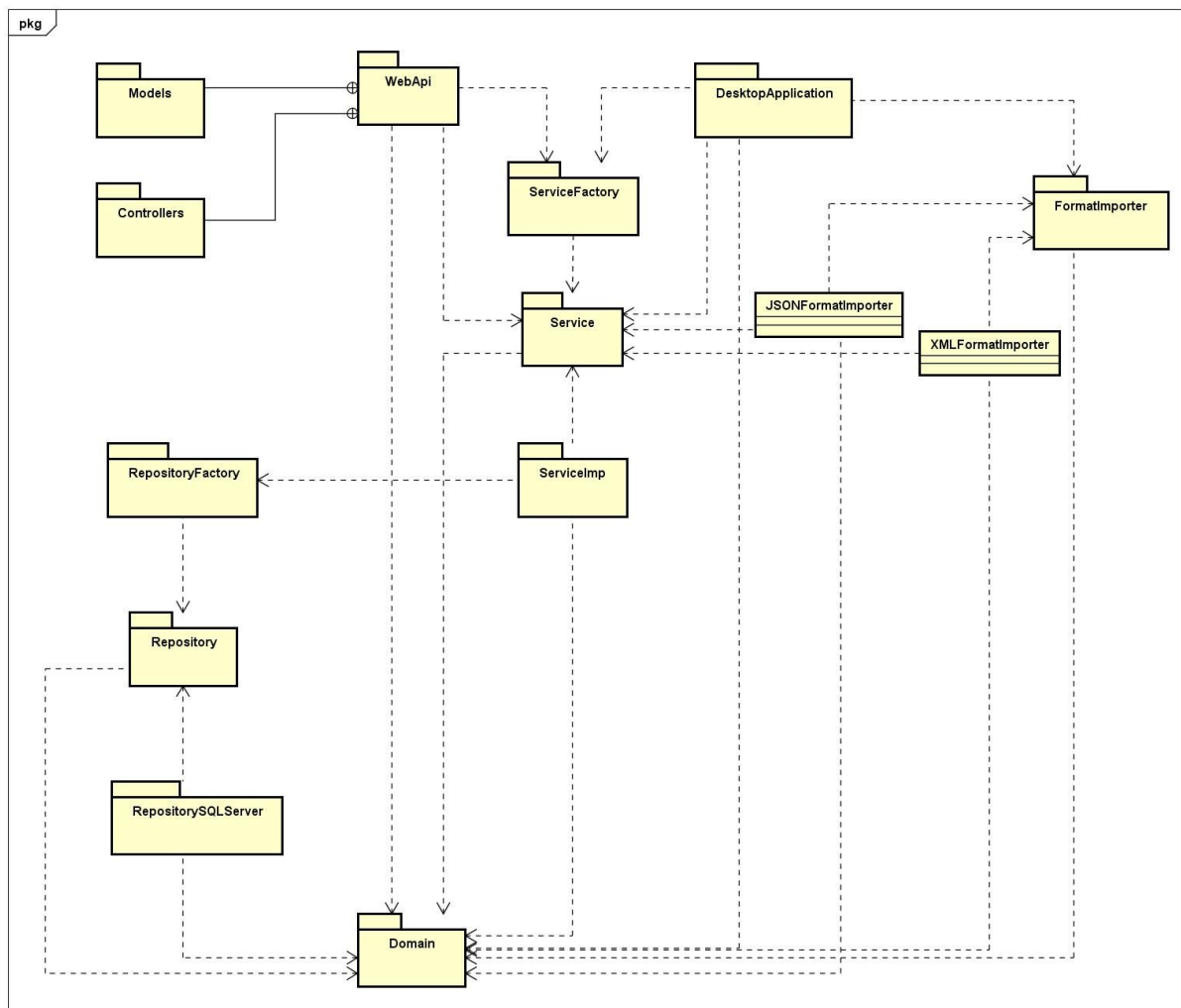
visualización respeta la herencia de la letra del obligatorio, con los estilos de los contenidos tomando precedencia sobre el de los contenedores.

1. Diseño del obligatorio

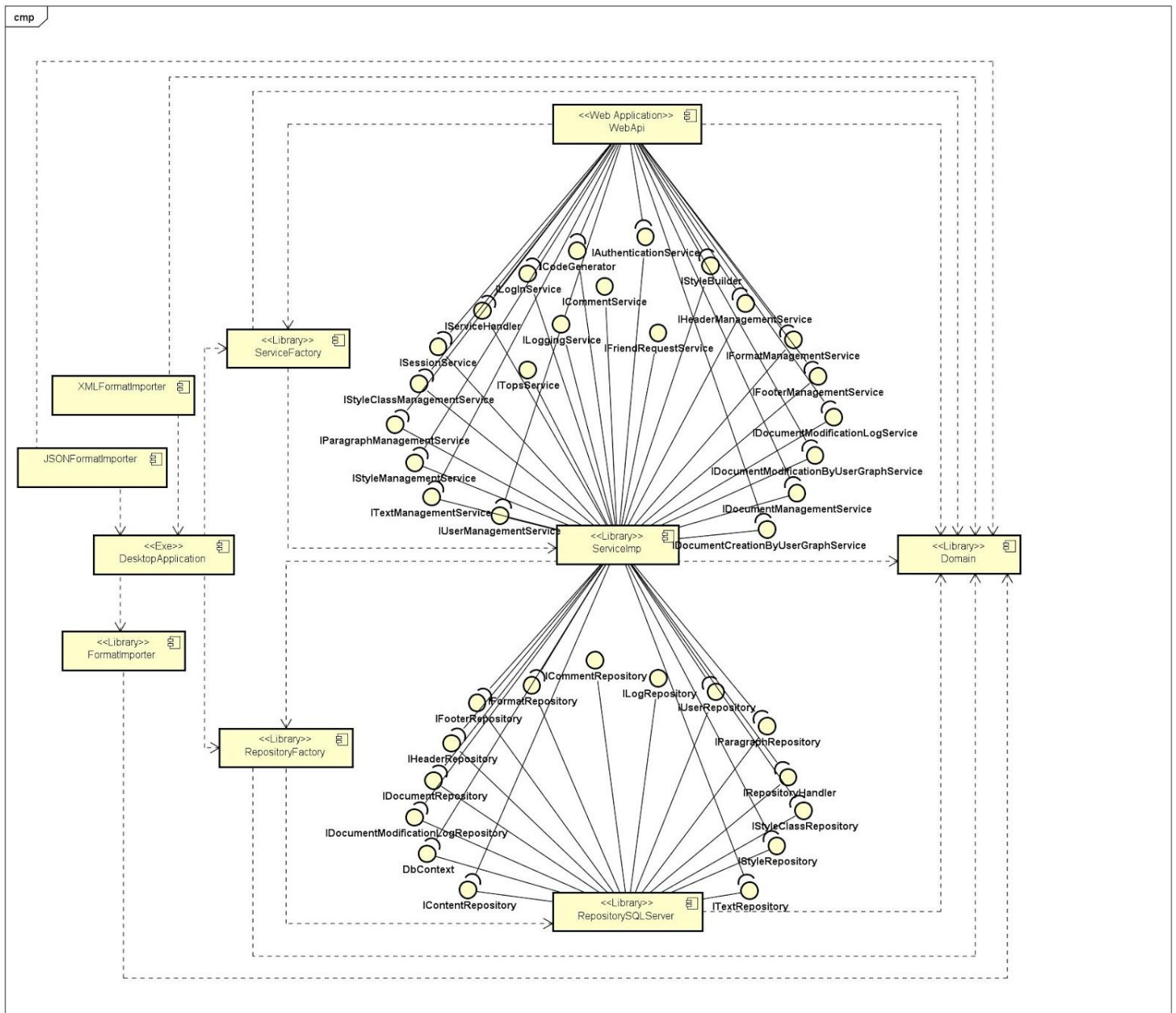
1.1. Arquitectura

El programa tiene una arquitectura de 4 capas : Capa de WebApi, capa de Services, capa de Repository y capa de Domain. También posee dos UIs implementadas, una es DesktopApplication y DocManager (Angular). A continuación, se muestra un diagrama de paquetes y un diagrama de componentes que muestran la interacción y composición de las mismas.

1.1.1. Diagrama de paquetes

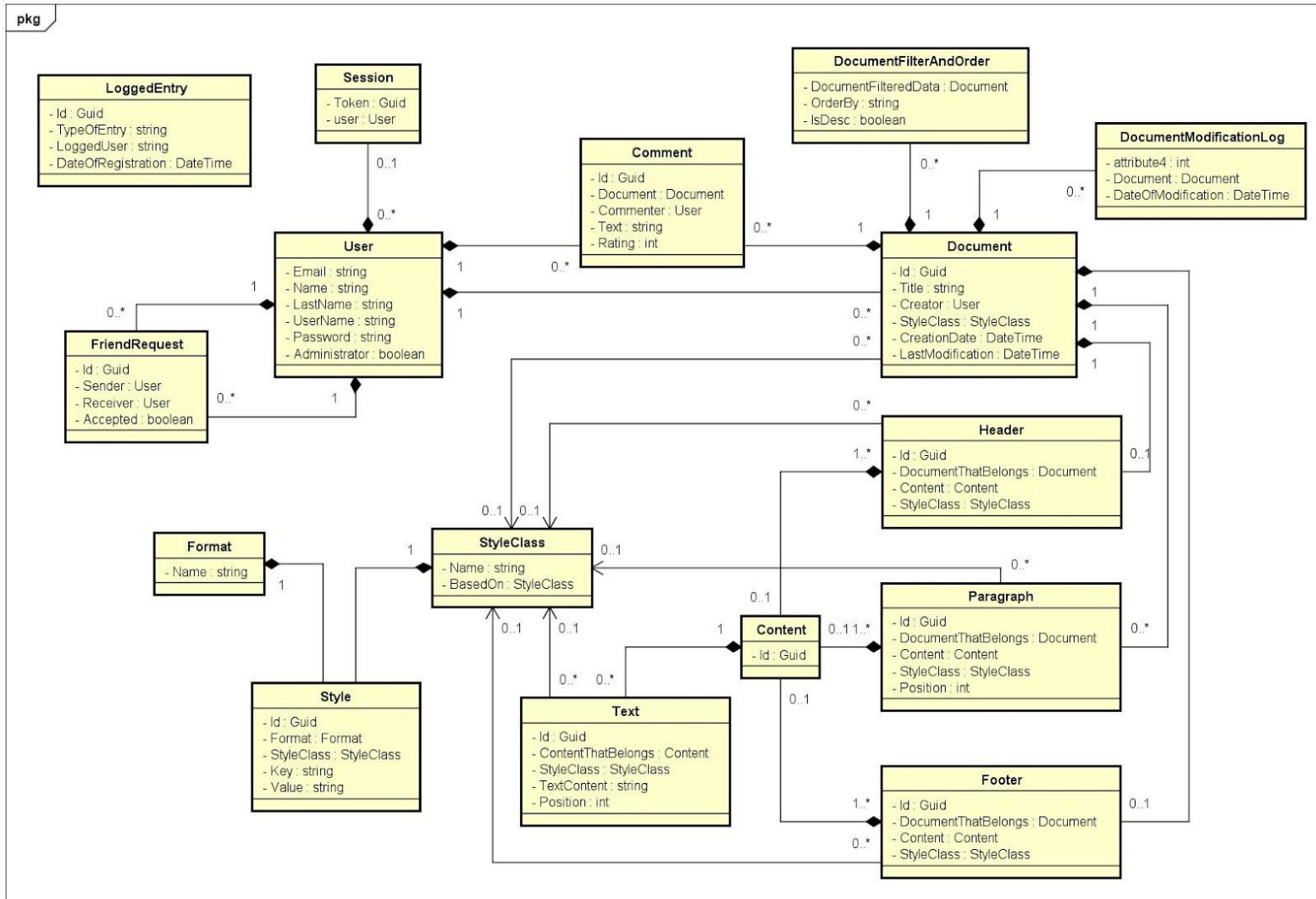


1.1.2. Diagrama de Componente



1.2. Domain

Para el dominio del programa tenemos las clases User, Document, Header, Footer, Paragraph, Content, DocumentFilterAndOrder, DocumentModificationLog, Style, StyleClass, Text, LoggedEntry, Session, FriendRequest y Comment.



Todos los atributos en el Dominio están implementados con Properties, por lo que tienen métodos públicos 'Getters' y 'Setters' que no están en el diagrama.

1.3. WebApi

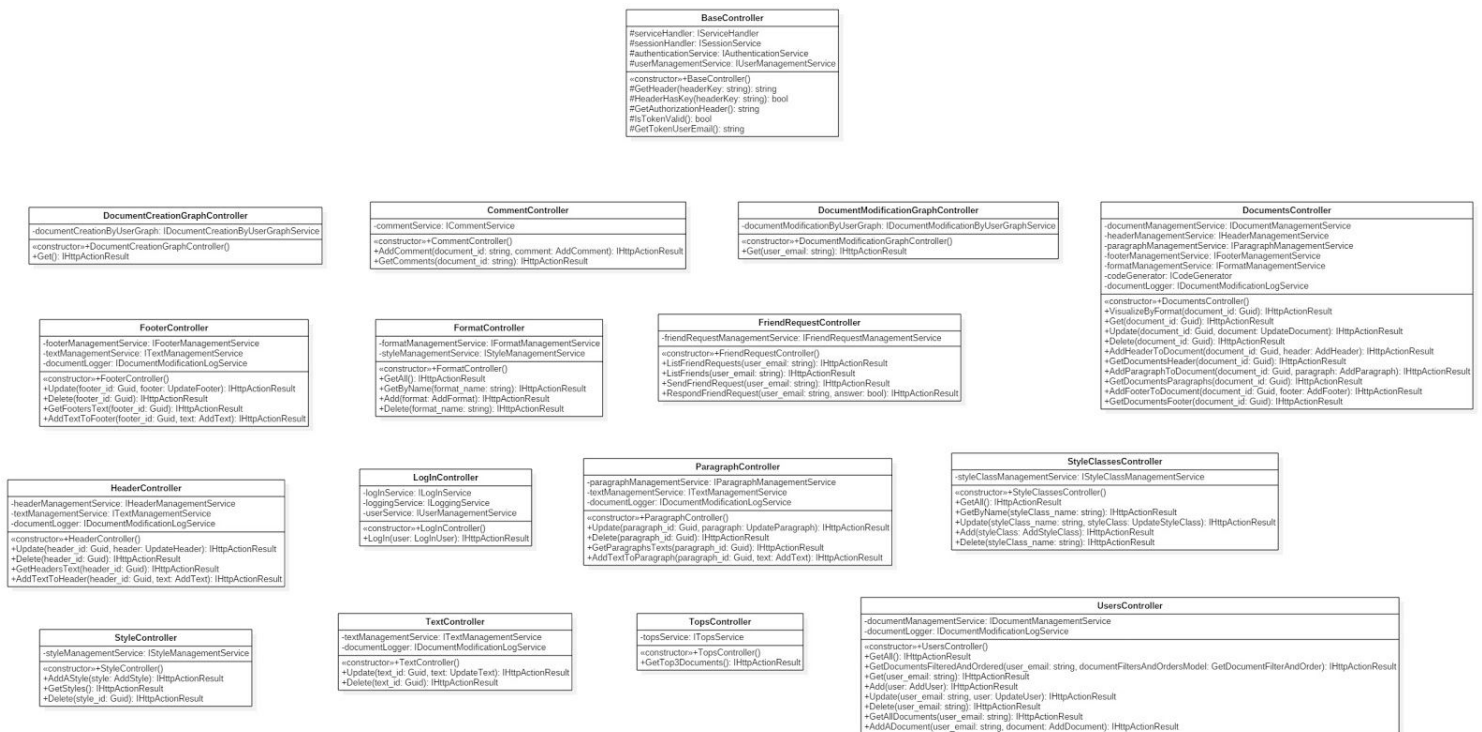
Contendrá un package de Controllers y un package de Models.

El package Controllers tendrá los siguientes controllers :

1.3.1. Controllers

Los controles a usar serán : UserController, TopsController, DocumentCreationGraphController, TextController, StyleController, ParagraphController, HeaderComponent, FormatController, FooterController, DocumentController, DocumentModificationGraphController, CommentController, StyleClassController, LoginController y FriendRequestController.

1.3.2. UML de los métodos del controller



Nota : Se omitió incluir las herencias en de cada uno de los controllers al BaseController, por cuestiones de facilitar el entendimiento del diagrama. Cada uno de los controllers debería tener una herencia hacia BaseController.

Se decidió hacer una clase base BaseController de la cual todos los demás controllers Heredaron de ella, y ella Hereda de ApiController.

1.3.3. Muestra de los Endpoints

Controller	Resource	Post	Get	Put	Delete
UserController	/users	Create a new user	Get all the users	-	-
	/users/{email}	-	Get the user	Update the user	Delete the user
		GetDocumentsFilteredAndOrdered			
	/users/{email}/documents	Create a new document	Get the user documents	-	-
FriendRequestController	/friends/{email}/listrequests		Get user friend requests		
	/friends/{email}/list		Get user friends		
	/friends/{email}/sendrequest	Add a friend request			
	/friends/{email}/respondrequest	Accept or reject a friend request			
DocumentController	/documents/{id}	-	Get the document	Update the document	Delete the document
	/documents/{id}/visualize		Visualize document		
	/documents/{id}/headers	Create a new header	Get the header of the document	-	-
	/documents/{id}/paragraphs	Create a new paragraph	Get all paragraphs of the document	-	-
	/documents/{id}/footers	Create a new footer	Get the footer of the document	-	-

HeaderController	/headers/{id}	-	-	Update the header	Delete the header
	/headers/{id}/texts	Create a text	Get the text	-	-
ParagraphController	/paragraphs/{id}	-	-	Update the paragraph	Delete the paragraph
	/paragraphs/{id}/texts	Create a text	Get the texts	-	-
FooterController	/footers/{id}	-	-	Update the footer	Delete the footer
	/footers/{id}/texts	Create a text	Get the text	-	-
TextController	/texts/{id}	-	-	Update the text	Delete the text
FormatController	/formats	Create a new format	Get all the formats	-	-
	/formats/{name}	-	Get the format	-	Delete the format
StyleClassController	/styleclasses	Create a new styleclass	Get all the styleclasses	-	-
	/styleclasses/{name}	-	Get the styleclass	-	Delete the styleclass
StyleController	/styles/{id}	Create a new style	Get all the styles for that format and that styleclass	-	Delete the style
LogInController	/login	Log in	-	-	-

1.3.4. Models

Los modelos utilizados fueron para satisfacer las peticiones y respuestas de los endpoints. De tal forma, se podrá recibir las propiedades que uno quiera y podrá filtrar otras. Un ejemplo de utilidad es el Modelo del Login, ya que éste, recibe solamente un Email y un Password del usuario, por lo tanto, será un modelo de usuario que solamente tiene esos 2 atributos.

Los models que cuenta el programa son :

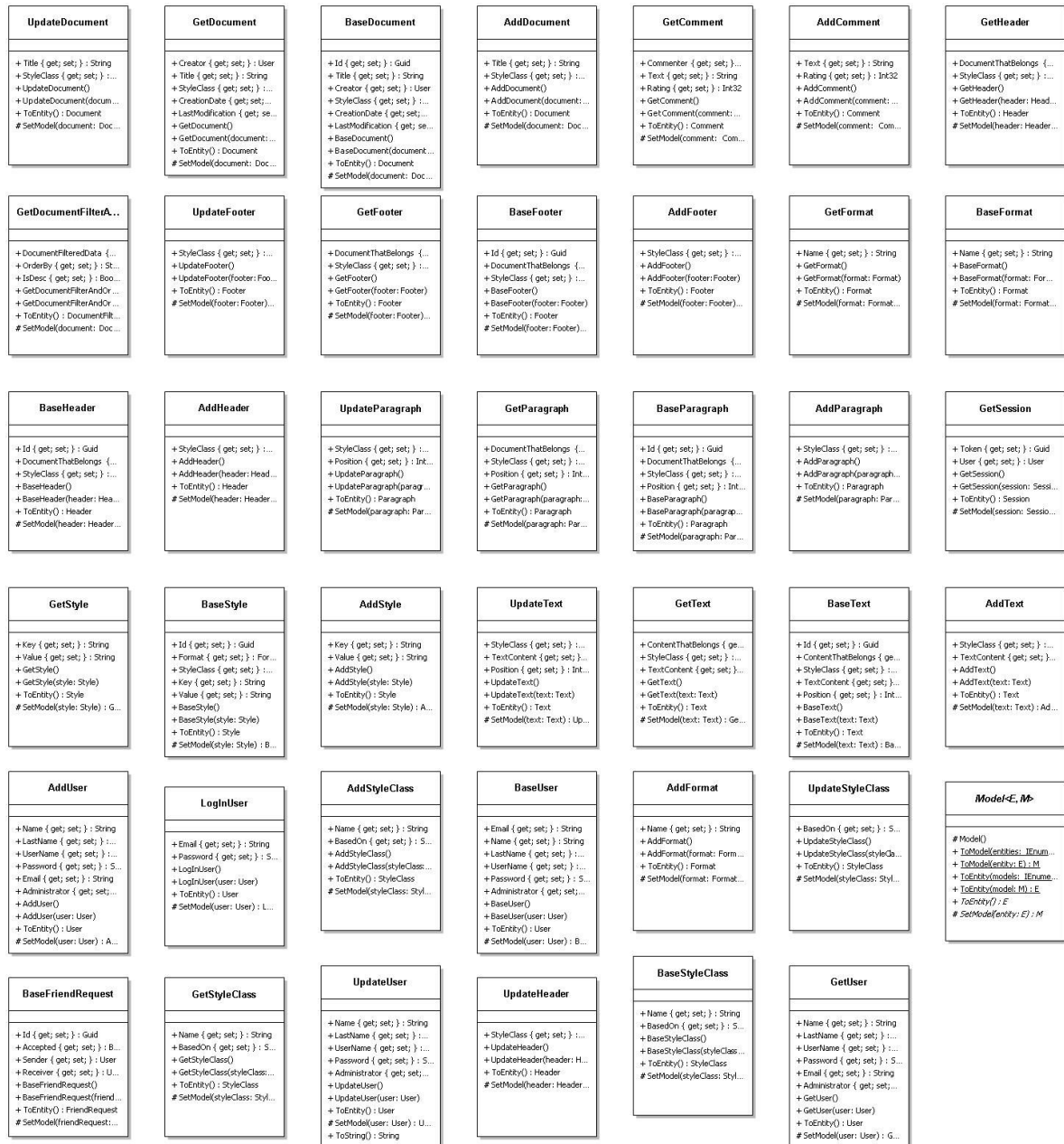
Los modelos Add, hacen referencia a que serán utilizados principalmente en los endpoints de Add del respectivo controller asociado.

Los modelos Get, hacen referencia a que serán utilizados principalmente en los endpoints de Get del respectivo controller asociado.

Los modelos Update, hacen referencia a que serán utilizados principalmente en los endpoints de Update del respectivo controller asociado.

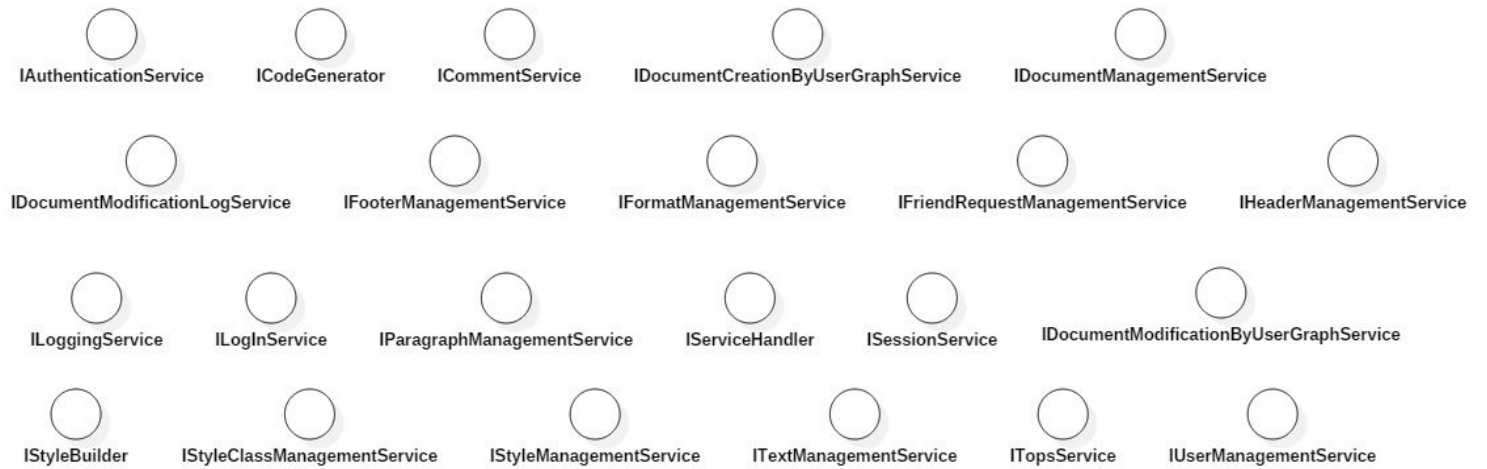
Los modelos Base, cuentan con todos los datos de la clase de Dominio asociada, por lo tanto no tienen ningún tipo de filtros.

Diagrama de los Modelos :



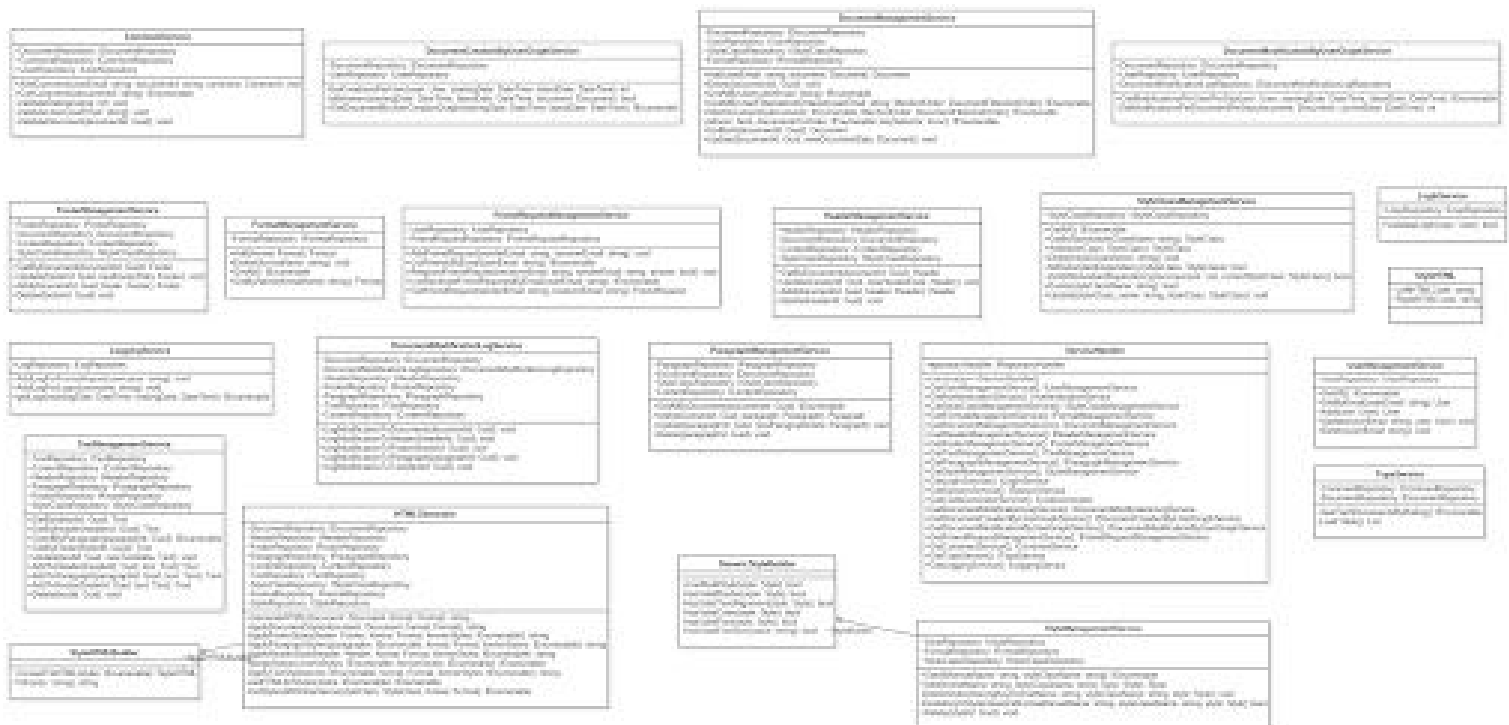
1.4. Service

UML de las interfaces de Service :



1.4.2 ServiceImp

Para ServiceImp, cada funcionalidad está implementada en una clase encapsulada de las otras. Las únicas clases que tienen dependencias dentro del paquete son StyleManagementService y HTML Generator, que requieren el uso de un Builder, y StyleHTMLBuilder que utiliza la clase StyleHTML, necesaria para su implementación. HTML Generator también utiliza a StyleHTML.

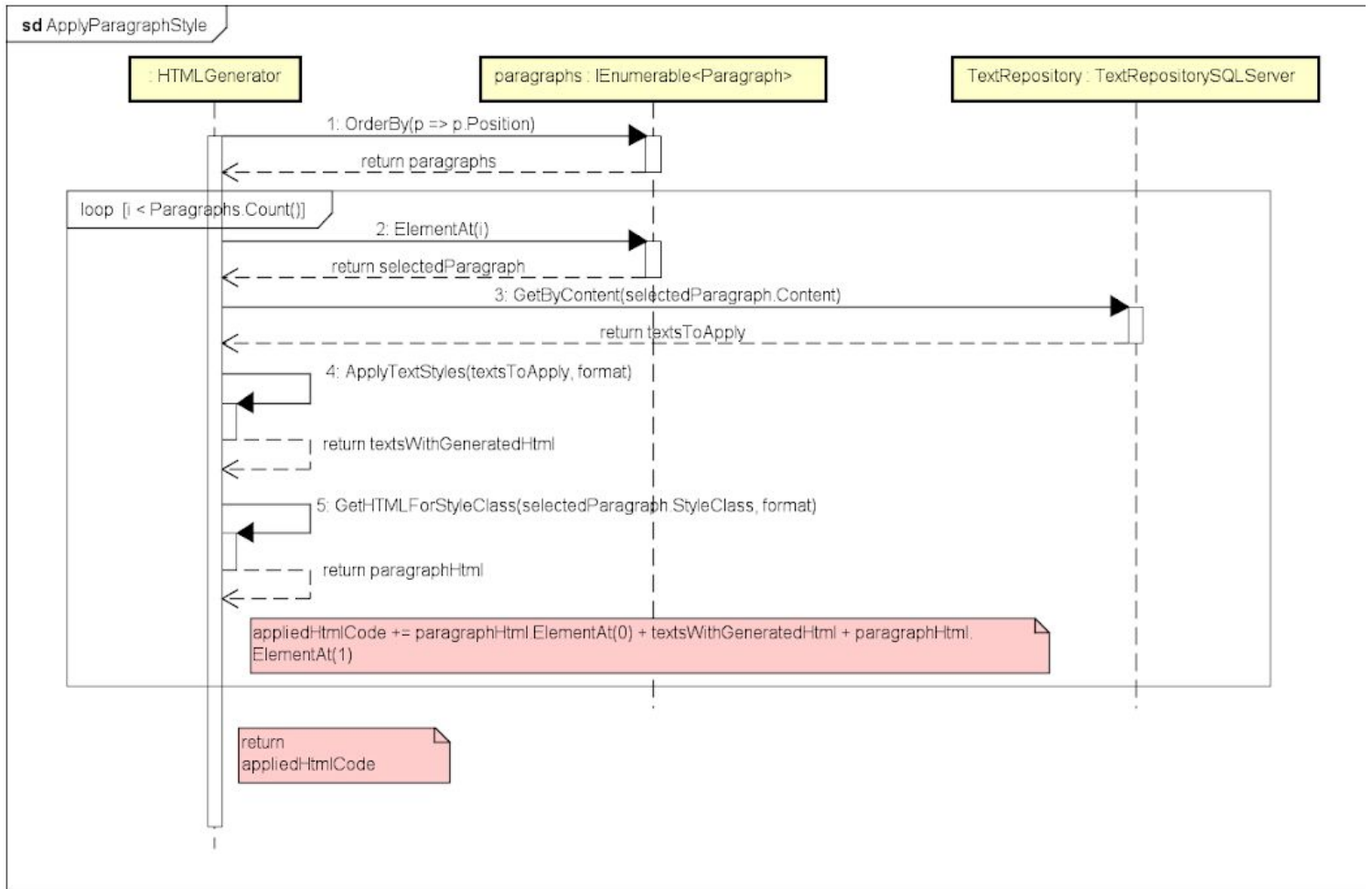


Estos son los diagramas de secuencia de la función principal de generación de código HTML y las demás funciones a las que llama.



4.4 ApplyHeaderStyle





sd ApplyFooterStyle

:HTMLGenerator

TextRepository : TextRepositorySQLServer

1: GetByContent(header.Content)

return textsToApply

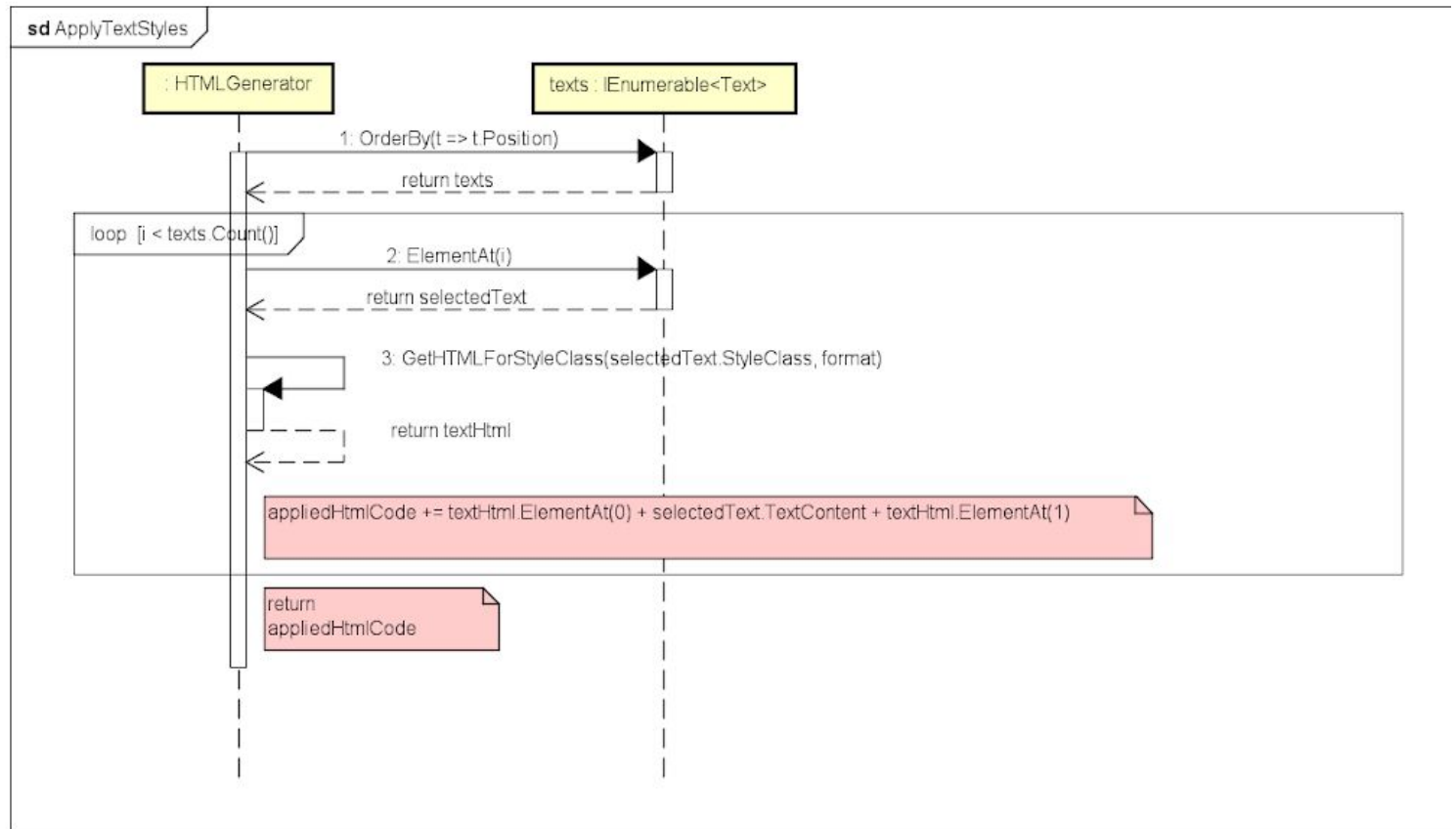
2: ApplyTextStyles(textsToApply, format)

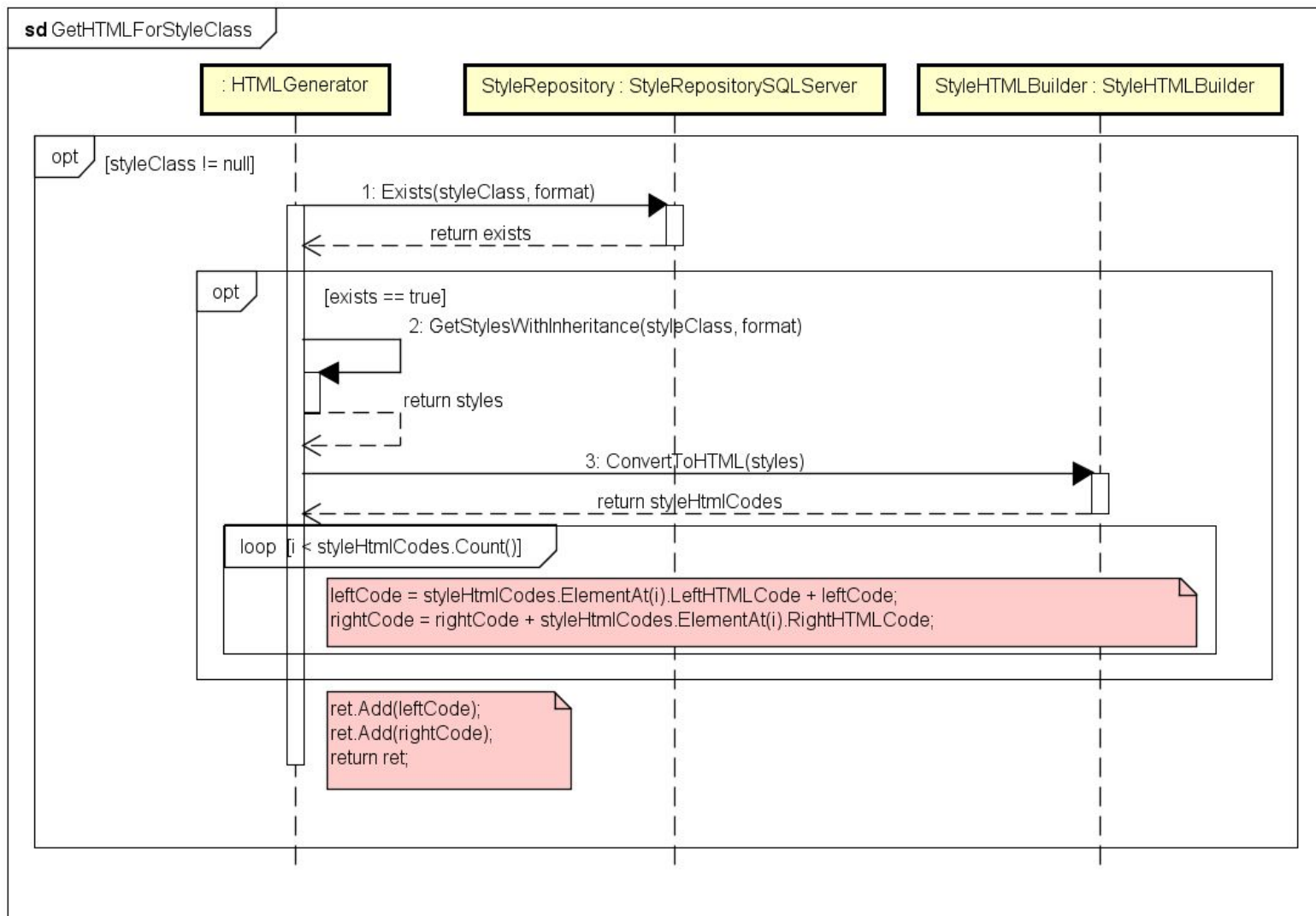
return appliedHtmlCode

3: GetHTMLForStyle(footer.StyleClass, format)

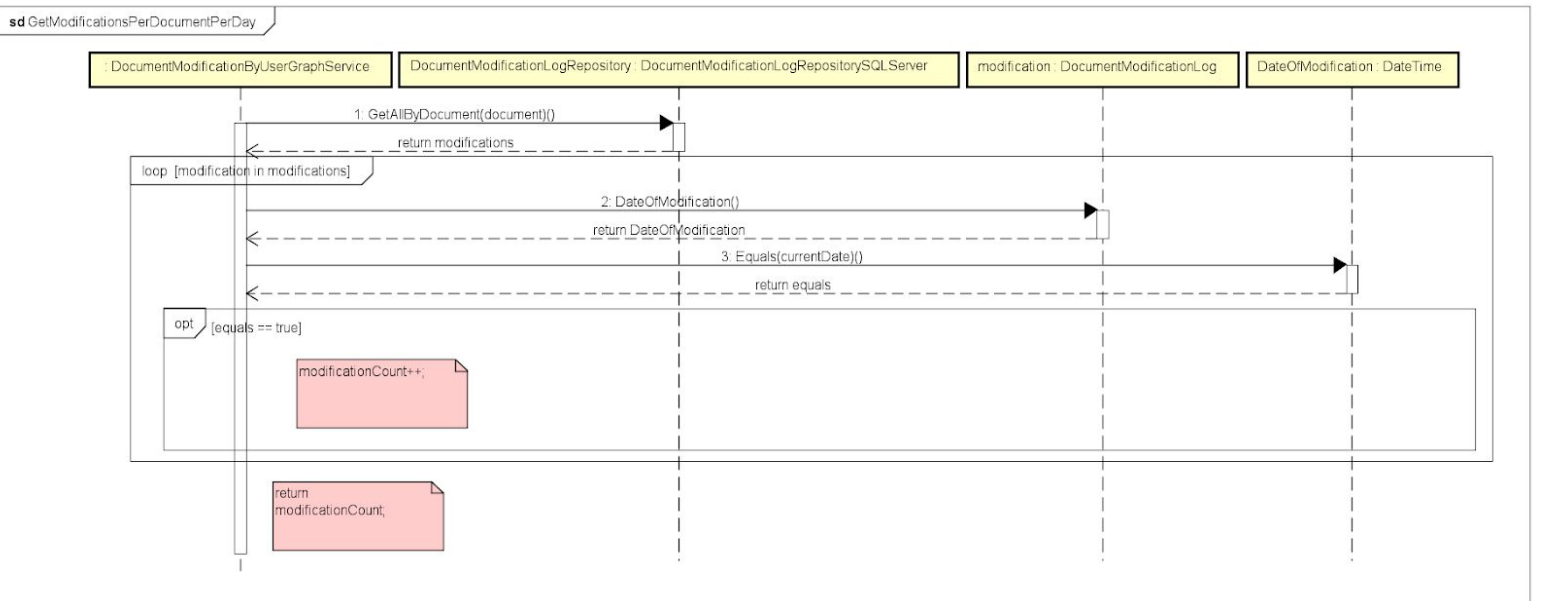
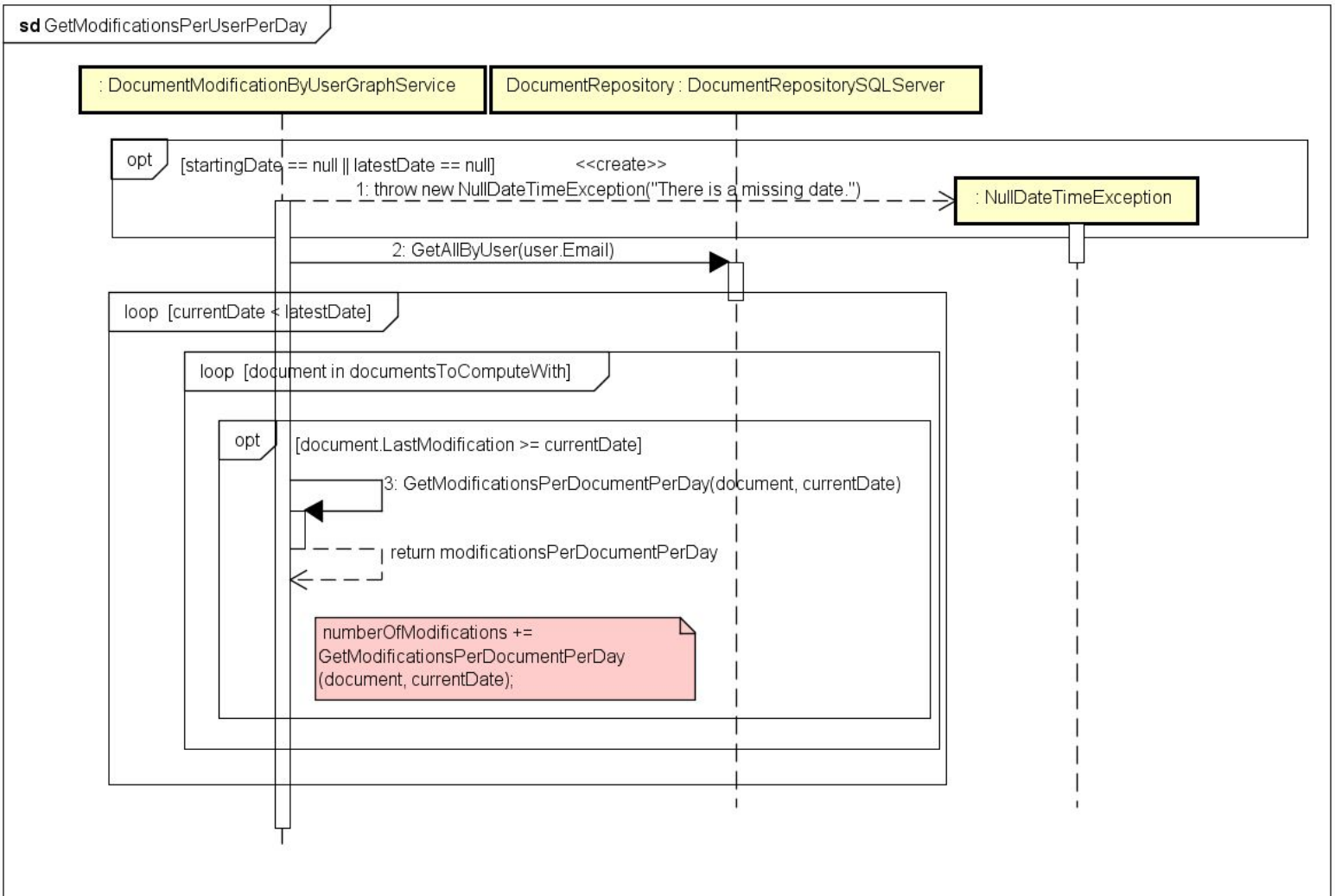
return footerHtml

return = footerHtml.ElementAt(0) +
appliedHtmlCode + footerHtml.ElementAt(1);

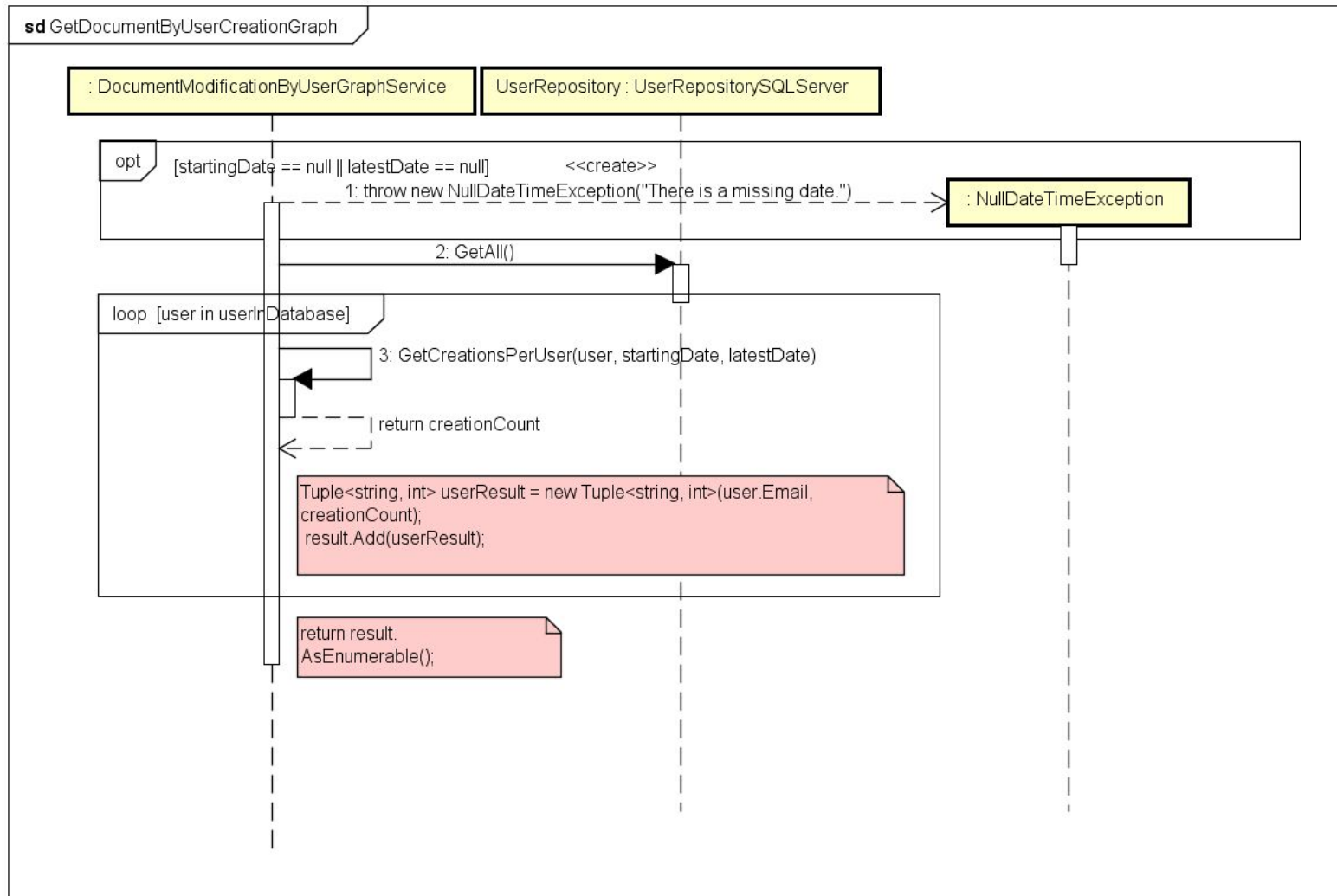


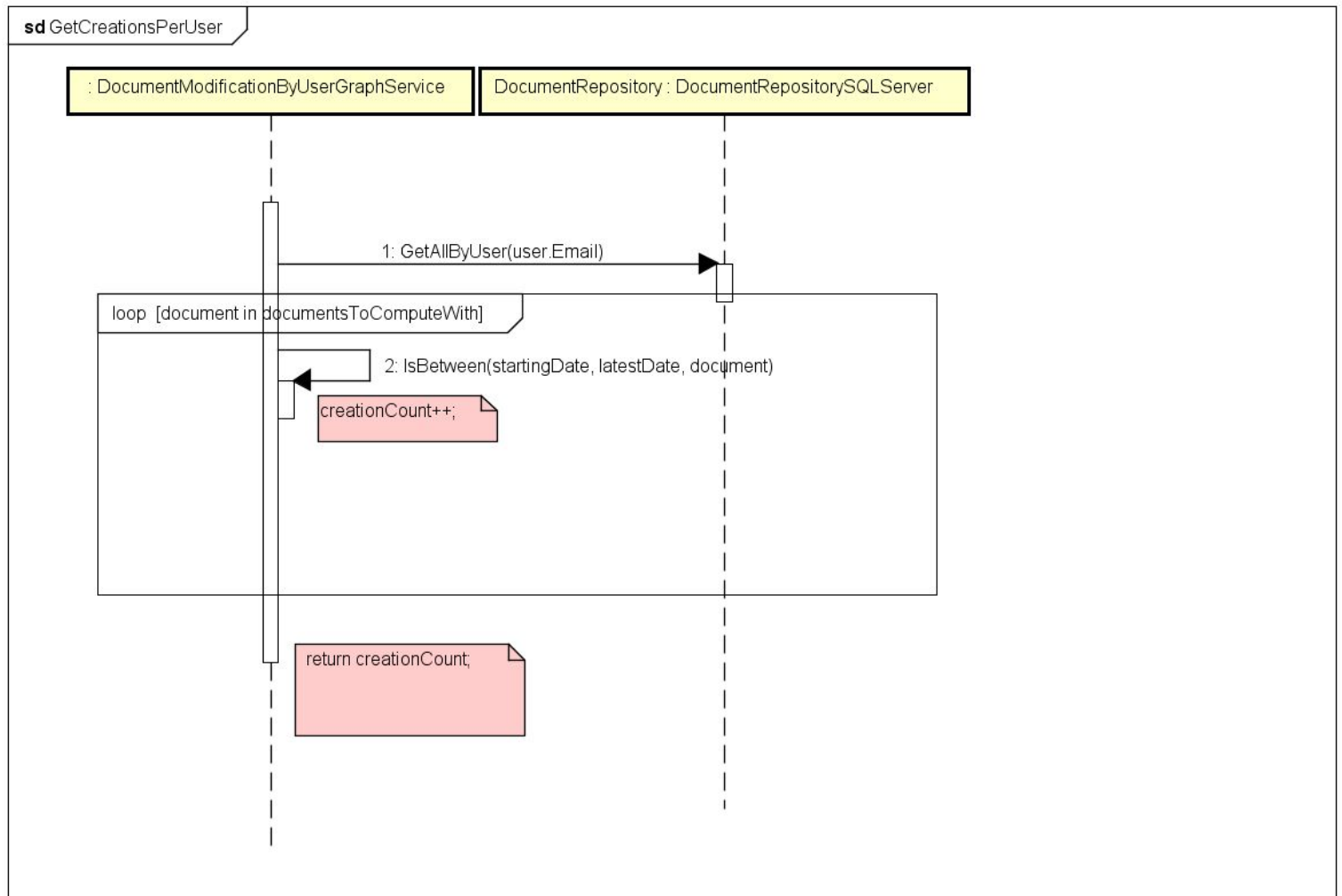


Estos son los diagramas de secuencia de las funciones del reporte que trae las modificaciones de documentos realizadas por día por un usuario dado.

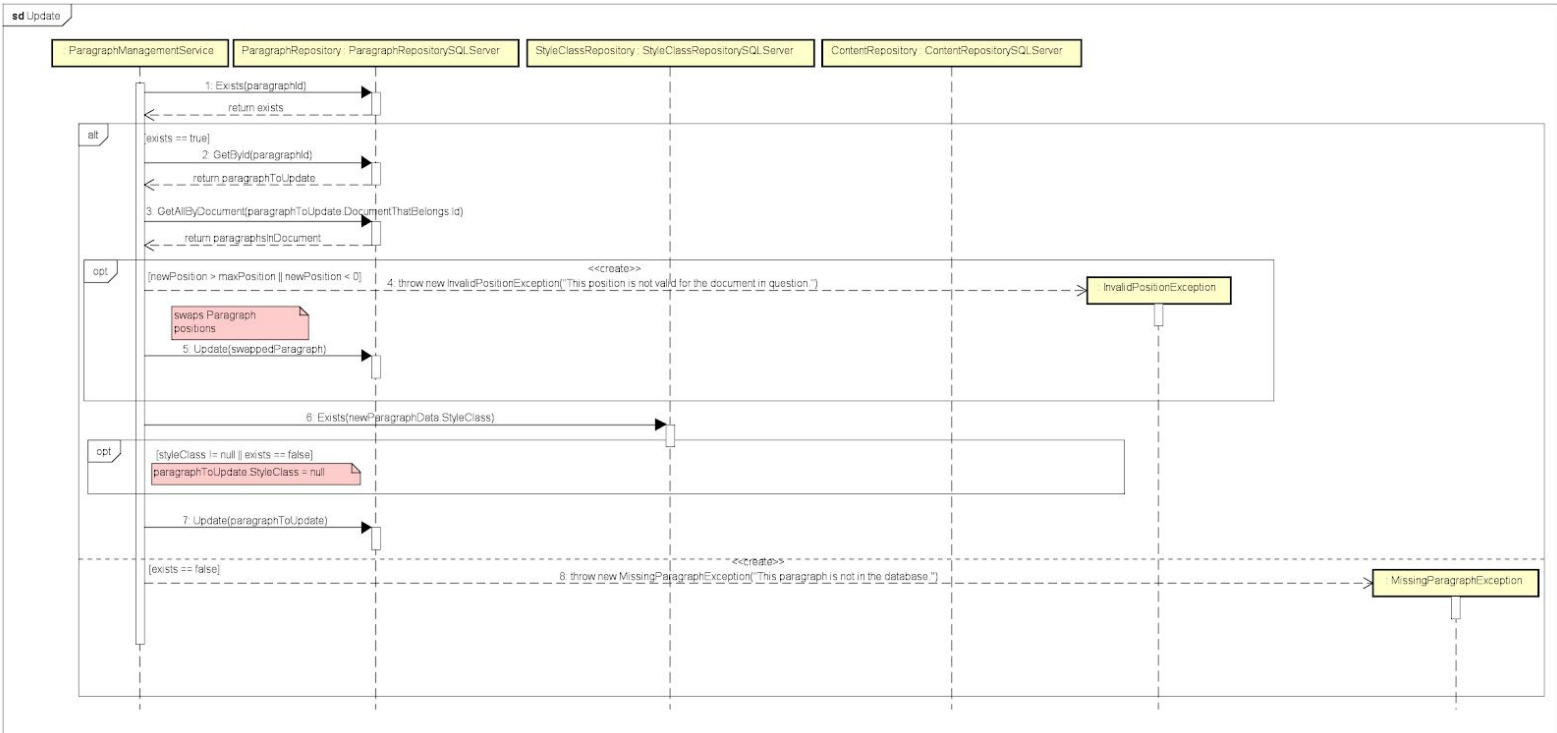
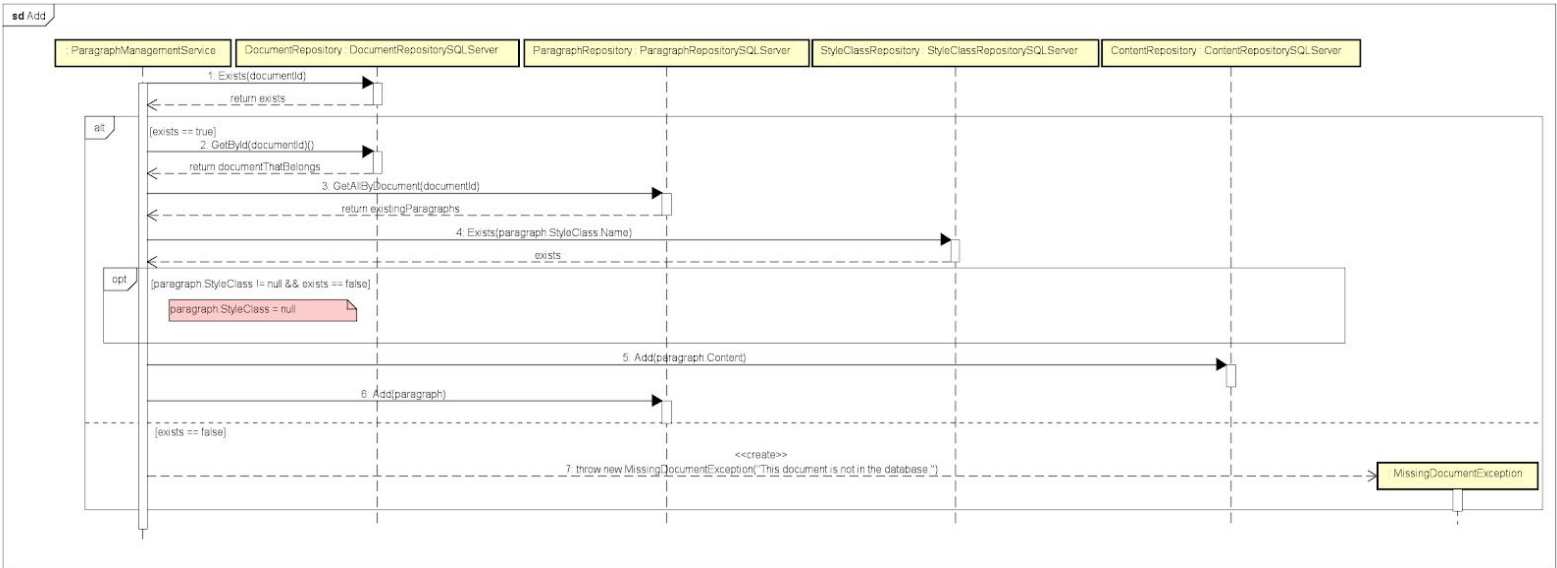


A continuación están los diagramas de secuencia que muestran las llamadas que realiza la clase de ServiceImp que implementa el reporte de la cantidad de documentos creados por usuarios entre dos fechas dadas.

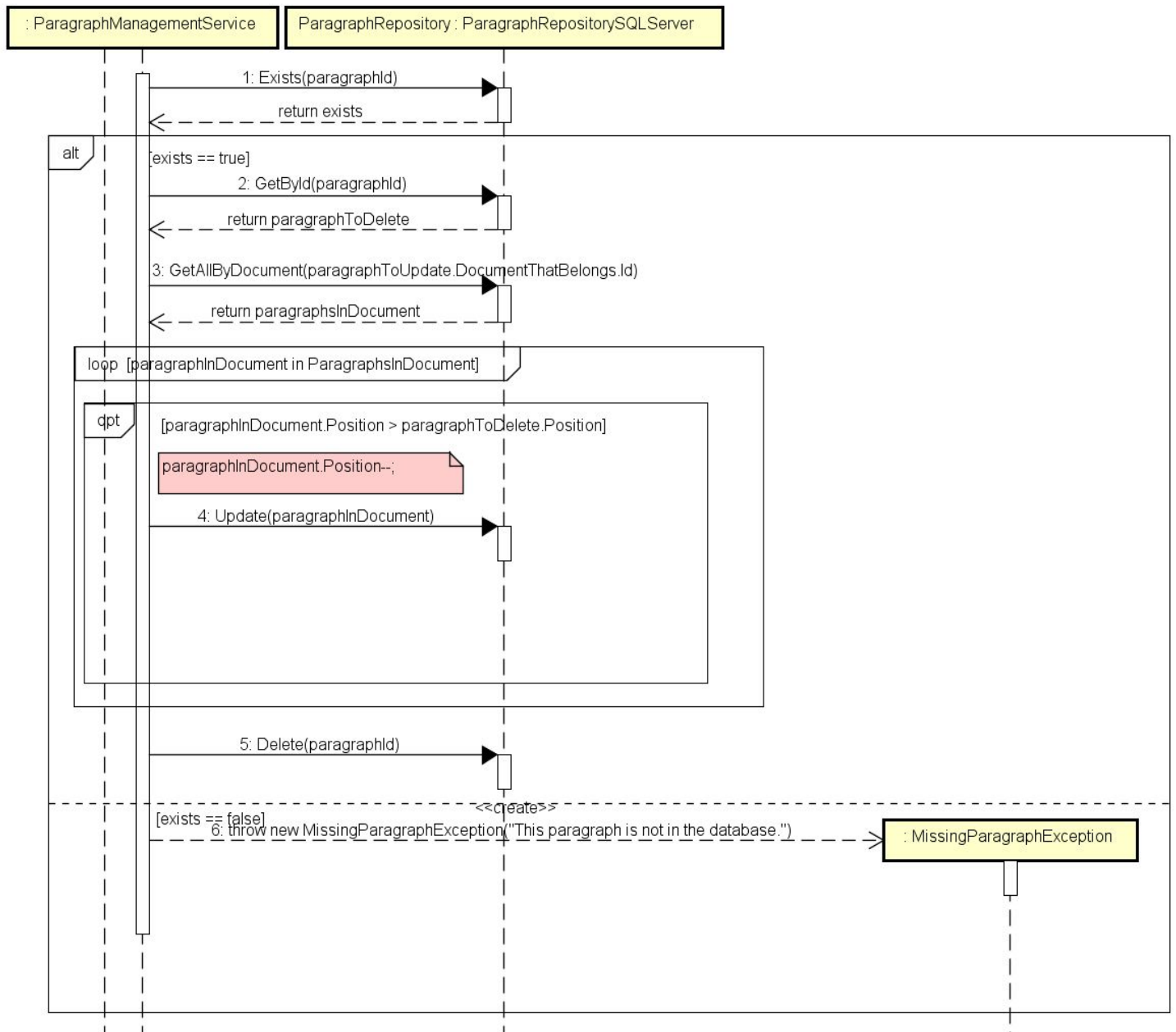




Estos son los diagramas de secuencia de los métodos Add, Update y Delete del ParagraphManagementService, que contemplan la funcionalidad de poder ordenar a los párrafos según su posición en el documento.



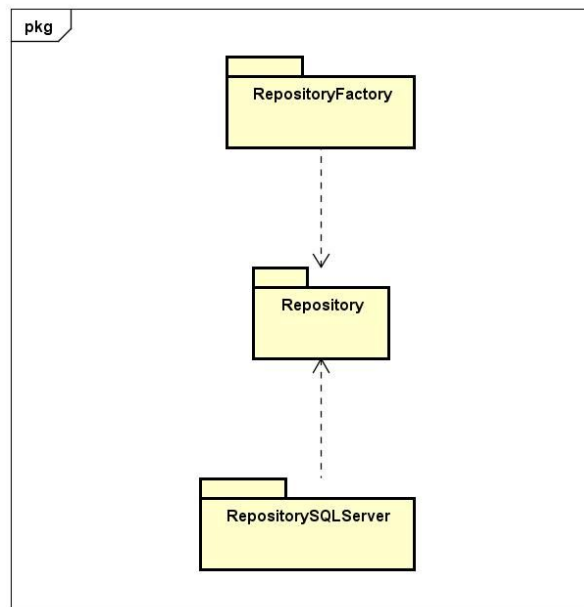
sd Delete



1.5. Repository

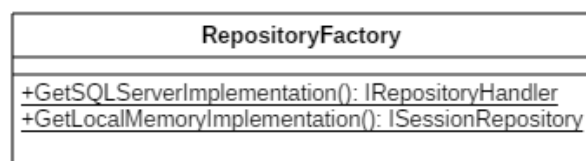
El manejo del Repositorio fue realizado por medio de 3 packages, los cuales son :
Repository, RepositoryFactory y RepositorySQLServer

Interacción entre dichos packages :



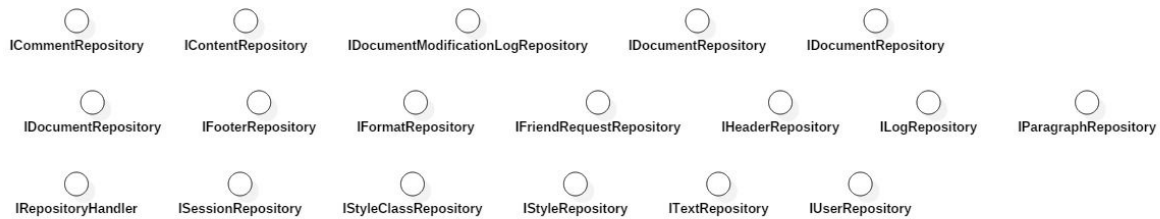
1.5.1. RepositoryFactory

Es el encargado de instanciar la implementación correcta de repositorio que se quiera utilizar, en nuestro caso, entre SQL Server o Local Memory.



1.5.2. Repository

Contiene las interfaces que se utilizarán para acceder al repositorio.



Cada una de las interfaces, contiene los métodos necesarios para hacer peticiones a la base de datos por medio de algún servicio.

La interfaz IRepositoryHandler contiene las firmas de los métodos que devuelven una instancia de cada una de las implementaciones de las interfaces de Repository. Dicha implementación se encuentra en un package diferente, el cual implementa todas esas interfaces de Repository. Por lo tanto, en caso de cambiar de base de datos, o de framework, se deberán implementar todos los repositorios que ya cuenta el programa. De tal forma, el resto del programa queda intacto a dicha modificación.

1.5.3. RepositorySQLServer

Contiene un package Configurations y otro package Migrations, a parte de todas las implementaciones de las interfaces de Repository.

Las implementaciones de las interfaces de Repository, fueron realizadas con Entity Framework, se utilizó la metodología Code First de instanciación de la base de datos.

Las tablas de la base de datos fueron creadas mediante implementaciones de EntityTypeConfiguration. Se crearon una clase de Configuración por cada clase del Dominio que se quería agregar a la base de datos con sus adecuados atributos.

Configurations

Existen configuraciones para :

Comment, Content, Document, DocumentModificationLog, Footer, Format, FriendRequest, Header, LoggedEntry, Paragraph, Session, StyleClass, Style, Text y User.

Ya que eran las clases de las cuales nos interesaba generar una tabla en la base de datos.

Estas configuraciones fueron requeridas para ser utilizadas por Entity Framework Fluent Api :

Fluent Api

Fluent Api genera una independencia total de la implementación RepositorySQLServer del Dominio, ya que el Dominio no conoce al RepositorySQLServer, en cambio, si se utilizara DataAnnotations, habría un problema de acoplamiento entre Dominio y la implementación específica de SQL Server del repositorio, lo cual es incorrecto y desprolijo.

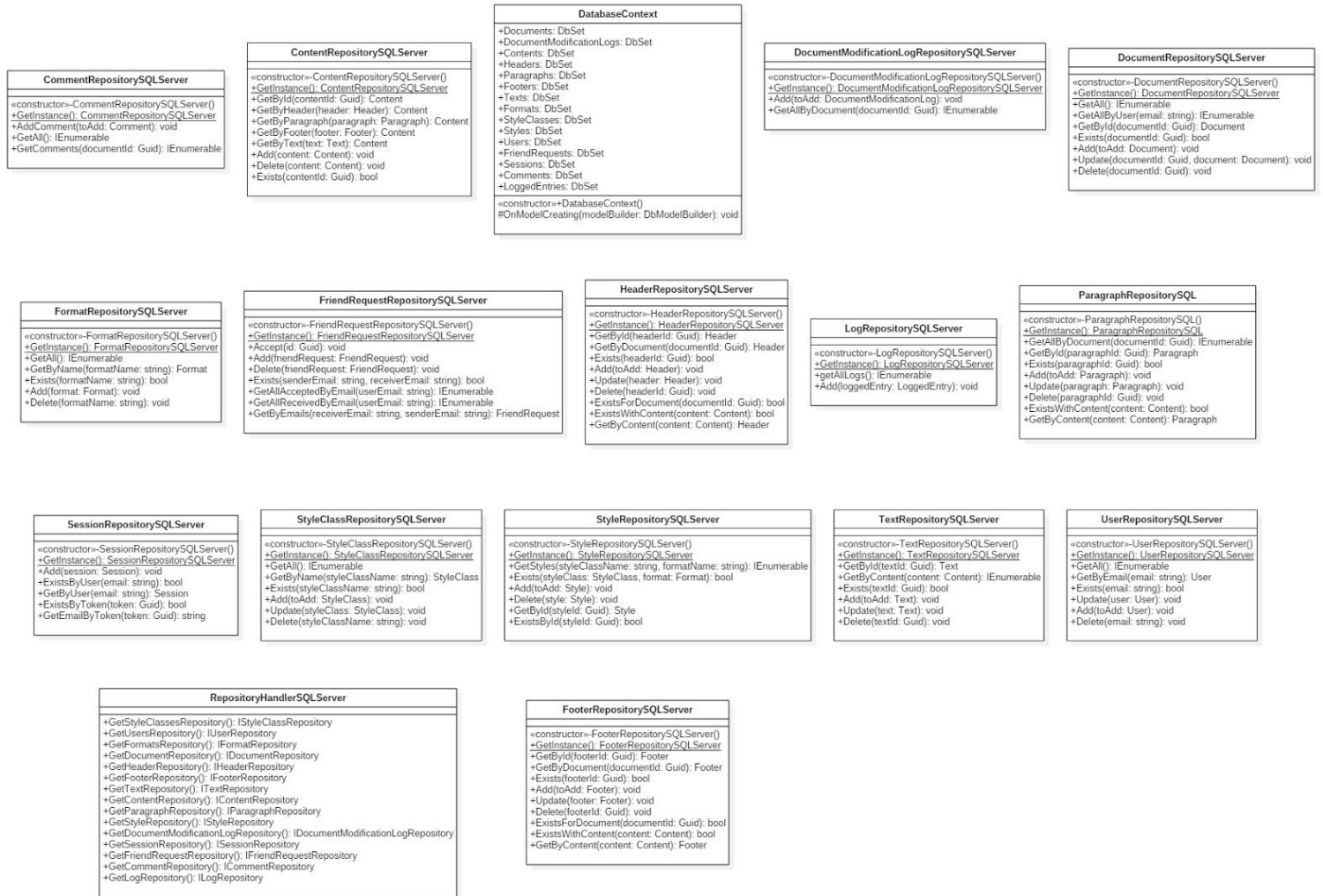
Clases de RepositorySQLServer

Cada clase de RepositorySQLServer es un Singleton, ya que se quiere mantener una única instancia de cada clase.

Los métodos utilizados son atómicos, ya que no se hacen comprobaciones de lógica. Sólomente se aplicaran métodos atómicos que haga una llamada específica en el repositorio, deslindando la responsabilidad de que llegue bien los datos, a la lógica. Donde debería estar dicha responsabilidad.

Hay una clase por Tabla en la base de datos, para poder administrar las responsabilidades del acceso a modificar cada tabla a su clase específica.

Diagrama UML de RepositorySQLServer



2. Justificaciones de Decisiones de Diseño Tomadas

2.1. Arquitectura

Se decidió hacer este modelo de capas, ya que establece un nivel de dependencia muy bajo entre capa y capa.

Cada capa se encarga de solucionar problemas a la capa de abajo, dependiendo si la responsabilidad de solucionar dicho problema tiene cohesión con la capa en cuestión.

Se intentó que cada capa estuviera lo menos acoplada posible a las capas de las que depende, por lo general, utilizando una interfaz que sea la que provea las funcionalidades.

Las 4 capas utilizadas fueron :

2.1.1. Capa WebApi

Provee las clases que se comunican con la UI, la WebApi solamente llama a la capa de servicios para que se realicen las funciones correspondientes a cada endpoint de la WebApi, devolviendo de manera correcta los datos al usuario una vez que se haya concretado la petición.

Contiene 2 paquetes fundamentales para el diseño de nuestro proyecto, un paquete de Models, que es el que tiene los modelos de las clases del dominio. Los cuales serán usados para recibir y para devolver los datos al usuario. También fueron utilizados para filtrar datos que no se quieran devolver o que no se quieran recibir.

El paquete de Controllers, que es el que contiene los endpoints del sistema, los cuales serán llamados por el usuario mediante verbos HTTP y una correcta nomenclatura y estructura.

2.1.2. Capa de Servicios

Realiza operaciones de lógica de negocio, comprobaciones, tira excepciones en caso de que la petición no se correcta y también contacta con la capa de Repositorio para delegar las operaciones que contacten directamente con la base de datos.

Contiene las funciones que hacen directamente contacto con la lógica de negocios, como por ejemplo, saber que un documento tiene 1 header, 1 footer y n paragraphs.

2.1.3. Capa de Repositorio

Es la que realiza operaciones atómicas a la base de datos, harán funciones específicas de llamadas específicas a la base de datos correspondiente al Repositorio usado.

2.1.4. Capa de Dominio

Es la encargada de contener el dominio del problema, la misma no cuenta con lógica y solamente cuenta con la estructura de los datos a manejar durante todo el programa.

De esta forma creo que se logró una alta cohesión entre los paquetes, ya que tienen responsabilidades muy concretas, como fueron especificadas recién.

Por la forma en la cual se realizaron las dependencias entre los paquetes, se logró también un bajo acoplamiento, el cual va a ser especificado a continuación.

Utilización de Factories

ServiceFactory : Se utilizó para que la WebApi no dependiera de una implementación concreta de la capa de servicios, y en cambio, dependiera de un Factory encargado de retornar la implementación que debería utilizar. De esta manera para la capa de WebApi, dichas implementaciones quedan ocultas y se genera un bajo acoplamiento entre ellas. De tal forma, un cambio en la

implementación de ServiceImp por otra implementación distinta de Service, no impactaría en lo más mínimo a las otras capas.

Se eliminó la dependencia que existía entre ServiceFactory y ServiceImp, haciendo que el código sea aún menos acoplado entre los paquetes.

RepositoryFactory : Análogamente a ServiceFactory, con la diferencia de que se utilizó para que la capa de Servicios, no dependiera directamente de la capa de Repositorio. Las razones son las mismas que se utilizaron en ServiceFactory.

Como extra, en la capa de Repositorio, podría tener aún más sentido a que actualmente se utilizan 2 implementaciones distintas de repositorio. Uno en memoria local para las sesiones y otro en SQL Server. Por lo tanto, se puede ver en acción el efecto que tiene RepositoryFactory.

Las dependencias que se perdieron por haber implementado Reflection, son las que existen entre RepositoryFactory con RepositorySQLServer y RepositoryLocalMemory.

2.2. Domain

Las clases Document, Header, Footer, Paragraph, Format, Text y User están ahí porque encapsulan entidades que están definidas en la letra (usuarios, documentos, etc.)

Las clases Header y Footer las decidimos representar por separado para reducir el acoplamiento. A pesar de que tengan el mismo comportamiento en la letra actual decidimos que no era complejidad innecesaria diseñarlas como entidades lógicas distintas, y esto es porque si en un momento posterior se cambia la estructura de Footer esto no impactaría en la estructura de Header.

En el caso de la relación entre usuario y documento, que es una donde el usuario tiene a varios documentos y el documento conoce a su creador, decidimos implementarlo con un atributo de usuario "Creador" en el documento. La razón por la que lo hicimos así y no con una lista de documentos en el usuario es para reducir el

acoplamiento. Si se fuera a cambiar la relación y el documento dejara de importarle al usuario (por ejemplo si se tuviera que borrar) no es necesario modificar al usuario en el dominio.

La relación entre los Headers, Footers y Paragraphs con el Document es similar a la relación entre User y Document, por lo que lo implementamos de la misma manera, con los Headers, Footers y Paragraphs guardándose el Document al que pertenecen. La justificación es la misma de antes, para que si se decide en algún momento borrar una de las clases no tiene impacto en la clase que 'componen'.

La clase Content existe sólo por una decisión de diseño que tomamos nosotros, y está ahí para que la clase Text dependa de una clase menos volátil que las clases Paragraph, Header y Footer. Es más probable que cambie la definición de las partes de un documento que la definición de lo que es un contenedor de texto. Esto respeta una de las sentencias del principio de Inversión de Dependencias de SOLID, que enuncia que si dos clases volátiles dependen entre sí es mejor tener una clase intermediaria estable del cuál ambas dependan. Esto favorece la extensibilidad del código, ya que si agregamos o borramos partes del documento no afectan a la lógica de como está hecho un texto o su contenedor.

La clase Style no está explícitamente definida en la letra del obligatorio, pero fue nuestra decisión de Diseño agregarla al dominio de todas maneras. Esto es porque sabemos que las clases de estilos difieren en lo que van a mostrar según el formato, y no tiene sentido que una Clase de Estilos sin formato alguno tenga un estilo para mostrar. Al revés, no tiene sentido que un formato tenga estilos sin una clase de estilos.

Por lo tanto Style es una clase que conoce la configuración de un estilo determinado para un par de Format y StyleClass. Tiene una relación de vida con StyleClass y Format, por lo que si se borra un Format o StyleClass se borran los Style relacionados a ellos también. Esto está bien pues Style es de menor nivel de abstracción que Format y StyleClass, que son más cercanos a entidades de la letra.

Las clases `DocumentFilterAndOrder` y `DocumentModificationLog` son necesarias porque la lógica de negocio requiere que conozcamos la cantidad de modificaciones a los documentos y los posibles filtros y criterios de ordenamiento que se pueden utilizar. Estas dos son necesarias para dos funciones específicas de la letra. La razón por la que decidimos almacenar estos atributos en clases nuevas es para respetar el principio del Open-Close. Al no modificar `Documento` y extender el Dominio agregando la clase con los atributos necesarios para la nueva funcionalidad, respetamos el OCP.

Para el segundo obligatorio se agregaron nuevas clases al Dominio para las funcionalidades de Logging y redes sociales. Estas fueron `LoggedEntry`, `Comment`, y `FriendRequest`. Las justificaciones para éstos se especifican en la sección 8.

2.3. WebApi

La capa de WebApi, se encargará de solucionar problemas de pasaje de datos propias de la interfaz, también es el que se encarga de presentarle al usuario los datos e interpretar las excepciones provenientes de la lógica y mostrarlas de mejor manera la usuario.

La elección de las uris para hacer las llamadas a cada endpoint, fueron siguiendo el estándar de no más de 3 niveles.

Se intentó hacer la forma que nosotros notamos como más coherente de acceder a cada endpoint, delegando la responsabilidad al controller adecuado.

Se crearon modelos extras o que estaban “de más”, por ejemplo : utilizar el `AddFooter` y el `GetFooter`, aunque por código ambos cumplen la misma función. Esto se hizo para no generar un acoplamiento de un `GetFooter`, en un endpoint de `AddFooter`, porque en un posible futuro, se podría querer agregar un Footer de manera distinta que la que se quisiera mostrar el Footer. Análogamente se llega a la misma conclusión con el resto de los modelos con este problema.

El BaseModel de cada clase, generalmente es utilizado para devolver datos, ya que contiene todos los datos de la clase del Dominio, salvo en excepciones que no se quiere devolver esto.

Se manejan los atributos de cada Service mediante un prefijo readonly, que existe porque solamente se quiere instanciar estos servicios mediante el constructor.

Cada uno de los Controllers, tiene acceso a los servicios que utilizará en dicho controller.

2.4. Service

AuthenticationService es una clase que encapsula la responsabilidad de determinar si un usuario dado tiene la autorización para utilizar un servicio específico del programa. La razón por la que decidimos encapsular esto en su propia clase es para respetar el Single Responsibility Principle y además por el hecho de que no sabemos si el manejo de permisos va a ser siempre el mismo. Por lo tanto, si cada funcionalidad tuviera su propio servicio de autenticación un cambio en el sistema de permisos las afectaría a todas. De este modo el cambio impacta en sólo una clase.

Las clases que terminan en 'ManagementServices' son las que encapsulan la responsabilidad de realizar el mantenimiento de las entidades representadas en el dominio. Por lo tanto son responsables de las operaciones de agregado, modificación y borrado (el llamado 'ABM') de cada clase del dominio. Además tienen métodos que permiten obtener las entidades del dominio, estos están ahí para responder a los requests de GET de la WebApi, los cuáles son necesarios para que el usuario pueda realizar las actividades de mantenimiento en primer lugar.

El DocumentManagementService tiene las operaciones ABM de Document, además de implementar la funcionalidad de filtrar y ordenar los documentos que un usuario quiere ver. Aquí reconocemos que hay un design smells de una clase que es demasiado grande, y que el filtrado y ordenado debería ser realizado por una clase separada, pero debido al tiempo que faltaba para la entrega decidimos dejarlo como estaba porque a fin y al cabo el filtrado y ordenado de una lista de documentos que

el usuario va a editar es una extensión de la funcionalidad de mantenimiento. Tenemos en cuenta que se podría mejorar, y para la segunda entrega sería un objetivo corregir este mal diseño.

FormatManagementService, HeaderManagementService y FooterManagementService no tienen ninguna decisión importante de diseño que haya que reportar. Realizan controles básicos de que no se puedan agregar formatos con el mismo nombre o agregar más de un header a un documento, pero todos los controles que realizan son los que están implícitos en la letra.

Para UserManagementService, decidimos identificar a los usuarios por su email, y controlar que no se puedan agregar usuarios con ese atributo con valores iguales.

LoginService tiene la responsabilidad de verificar que el usuario usando el programa sea en efecto un usuario existente. Está en la lógica de negocio porque no sería adecuado que la WebApi pudiera preguntar sobre la existencia de un usuario en la base de datos y además por razones de extensibilidad- Si se cambia la capa de interacción con el usuario la lógica del login sigue siendo la misma.

SessionService se encarga de darle a la WebApi el token correspondiente al usuario que está conectado. Puede ser que el SessionService deba estar ubicado en la WebApi y no en la capa de la lógica, porque el manejo de tokens es inherente a la implementación de WebApi. En este caso no hay tiempo de hacer el cambio pero queda pendiente para la segunda entrega cambiarlo o borrarlo de volverse obsoleto.

StyleClassManagementService tiene una funcionalidad adicional a los otros managers y es que verifica que no existan dependencias redundantes entre clases de estilo, además de que no permite borrar a clases de estilo que sean 'padres' de otra. Pensamos que esto cae dentro la responsabilidad de realizar un mantenimiento del StyleClass de acuerdo a las restricciones de la letra, por lo cual las funcionalidades están en la clase correcta.

ParagraphManagementService y TextManagementService realizan un control adicional comparado con los otros managers y es que ellos también manejan la

posición de las clases de las que son responsables dentro de sus respectivos contenedores (Document para el Paragraph, Content para el Text). Esto es porque en nuestra implementación de Text y Paragraph ellos conocen la posición en donde están, por lo tanto la gestión de su posición en el documento es responsabilidad de sus respectivos ManagementServices.

StyleManagementService mantiene los estilos según el Format y StyleClass al que pertenecen, y además sobrescribe estilos que tengan la misma clave (por ejemplo, si yo quisiera agregar un estilo de fuente Times New Roman tamaño 12 para un Format y StyleClass donde ya hay uno que es Arial 14, se eliminaría el estilo antiguo y sería reemplazado por el primero). Esta decisión la tomamos porque lógicamente no tiene sentido que existan dos estilos del mismo tipo para una misma visualización, porque uno de los dos estilos sería invisible.

El GenericStyleBuilder es una clase que le informa al StyleManagementService cuáles son los estilos que son compatibles con el programa actual (es decir, le muestra las combinaciones de clave-valor que son aceptables). Está hecho de esta manera por razones de extensibilidad, tal de que si los estilos válidos cambiaran sólo es necesario reemplazar el builder.

El HTMLGenerator lo diseñamos sólomente como parte de esta implementación, ya que es una implementación de la interfaz CodeGenerator, cuya responsabilidad es generar el código de visualización. Esto es para no adherirnos a la implementación 'HTML' de la visualización.

La clase HTMLGenerator funciona utilizando una clase llamada StyleHTMLBuilder, que tiene la responsabilidad de transformar una clase Style genérica en su código HTML respectivo, a tal modo de no adjuntar esa responsabilidad al HTMLGenerator, que lo haría demasiado grande. El HTMLGenerator aplica el StyleHTMLBuilder a todos los contenidos del documento que tengan clases de estilo válidas para el formato y arma el texto HTML en el orden correcto.

Hay una función que el HTMLGenerator posee llamada GetStylesWithInheritance que no debería estar ahí, sino en el StyleManagementService, debido a que sólo la

lógica de Style debería conocer la estructura de herencias de Style. Esto es algo que vamos a mejorar para la segunda entrega.

Además, a pesar de que intentamos usar el Builder Pattern para la creación de estilos este no está bien implementado, porque estamos utilizando un Concrete Builder directamente, y no utilizamos una interfaz. Sería una buena práctica de diseño corregir esto también.

DocumentCreationByUserGraphService se encarga de la funcionalidad de administrador que hace una gráfica con las creaciones de documentos hechas por cada usuario entre dos fechas determinadas. La única decisión de diseño que vale notar aquí es que decidimos representar la gráfica con una lista de tuplas de string y int, siendo estas el mail del usuario y la cantidad de documentos creados, respectivamente. Esto es porque decidimos que esos dos datos son los más indicados para implementar una gráfica de barras en la segunda entrega.

DocumentModificationByUserGraphService es nuestra implementación de la funcionalidad de administrador que hace una gráfica con las modificaciones a documentos para tal día para todos los días entre dos fechas determinadas, para un usuario dado. Esta implementación implicó tomar varias decisiones de diseño que eran necesarias para su funcionamiento. Una de ellas fue la adición de DocumentModificationLog, que registra la fecha de una modificación de un documento específico.

Utilizando DocumentModificationLog la implementación de DocumentModificationByUserGraphService se volvió simple ya que sólo era necesario consultar con el repositorio las modificaciones totales de cada usuario por día. La razón por la que implementamos esto para documento y no para usuario es por si en algún momento fuera necesario realizar otra gráfica basada en las modificaciones realizadas a documentos, el código que no está relacionado con la formación específica de la gráfica ya está encapsulada en otra clase.

Debido a los cambios al dominio y la base de datos generados por DocumentModificationLog, decidimos hacer un DocumentModificationLogService

que se encargará de registrar la modificación al documento para cualquier componente de éste que hubiera sido modificado de alguna manera. Esto lo justificamos por una necesidad de encapsular el registro de modificaciones, que fue agregado solamente para lograr cumplir una funcionalidad específica.

Un cambio del primer obligatorio es que ahora se cuenta con un `LogInService` que soluciona la necesidad de comunicarse con el acceso a datos para iniciar sesión como un usuario.

También se agregaron nuevas clases como `CommentService`, `LoggingService`, `FriendRequestManagemenService` y `TopsService`. La justificación de estas se explica en la sección 8.

2.5. Repository

Las implementaciones se encuentran en un package diferente al package de las interfaces, ya que en caso de que se deba cambiar o reemplazar el package de implementación, no se tendrá que compilar las interfaces nuevamente. Los otros packages de servicios que dependen de `Repository`, no se verán afectados en nada por un cambio de implementación.

Decidimos utilizar una clase distinta singleton, para cada una de las tablas de la base de datos, de esta forma se genera una más alta cohesión de los métodos que se van a utilizar en dicha clase.

2.5.1. Utilización de FluentApi

Se decidió utilizar `FluentApi` para que el dominio no tenga dependencias del `EntityFramewrok`.

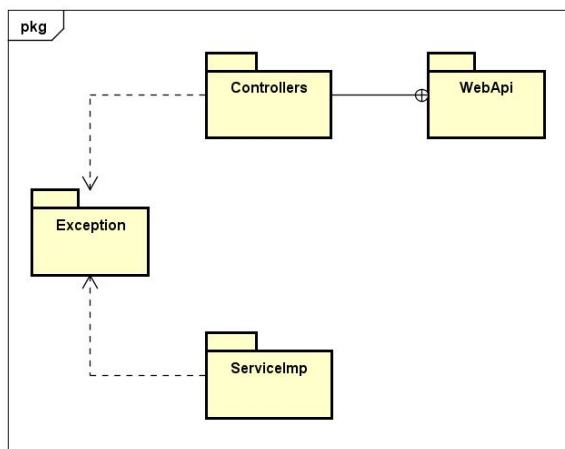
En caso de usar `Data Annotations`, existirían esas dependencias, por lo tanto, decidimos utilizar `FluentApi`.

La utilización de `FluentApi` fue detallada en las Decisiones de Diseño.

2.5.2. Utilización de un RepositoryFactory

Se utilizó un repository factory para que la implementación del repositorio a utilizar por el servicio, sea responsabilidad de él, por lo tanto, el servicio le pide al repository factory que le de la implementación a utilizar.

3. Manejo de excepciones



Los package ServiceImp y Controllers.WebApi son los encargados de usar las excepciones.

ServiceImp tira las excepciones en caso de que algo que recibe no cumple con la lógica establecida y Controllers.WebApi, atrapa las excepciones y se las muestra al usuario en forma de IHttpActionResult con el mensaje adecuado.

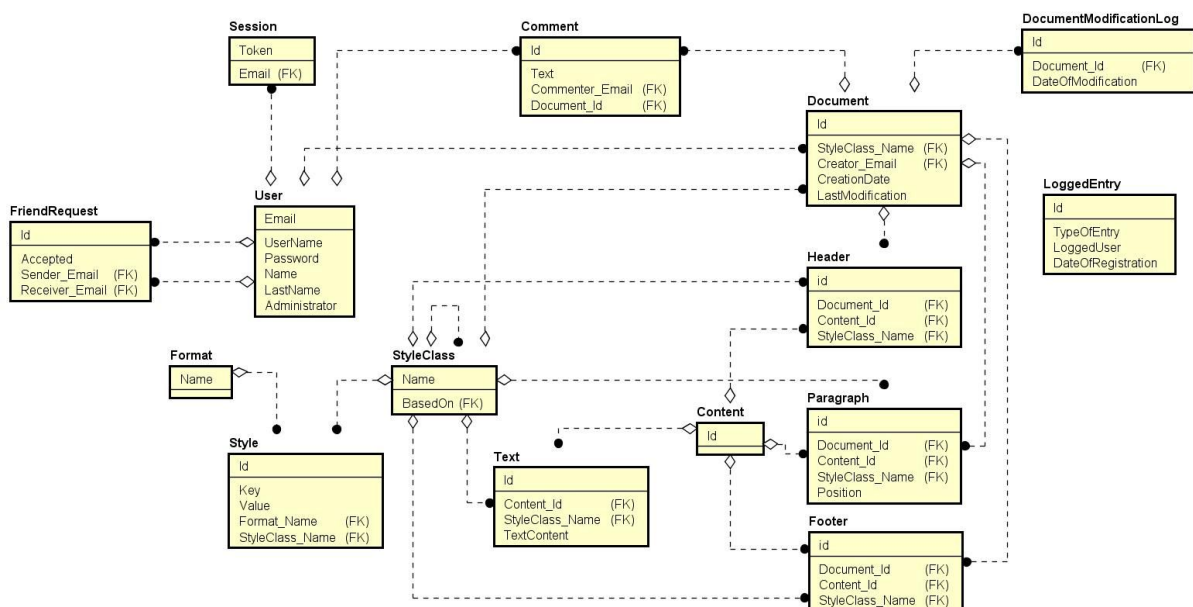
Se crearon clases particulares de Excepciones para cada una de las excepciones a tratar en el sistema, las cuales reciben por parámetro un mensaje apropiado y heredan de Exception las funciones.

Las excepciones tratadas fueron :



4. Manejo de Tablas de la Base de Datos

4.1. Diagrama de Entidad Relación



Para cada combinación de StyleClass y Format, existen n Styles que pertenecen a dicha combinación, por lo tanto, Style tiene FK de Format y StyleClass.

La utilización de Content fue explicada en el dominio, ver dicha sección para aclaraciones.

La relación que poseen en la base de datos, es un espejo de la relación que poseen en el Dominio del problema ya que fue utilizada la técnica Code First para crearla.

5. Justificación de Clean Code

Una de las prácticas más comunes de Clean Code son los nombres mnemotécnicos para las variables, clases y métodos, algo que utilizamos extensivamente en el desarrollo de nuestro programa.

El programa está escrito en inglés, por lo que todos los métodos y variables también lo están. Además, cada variable se escribe como lo que se supone que es en la lógica del algoritmo, tal y como cada método describe su función.

Por ejemplo, en el dominio, el Documento se guarda un User, pero este no se llama User. Se llama Creator, porque lógicamente es el creador del documento, y no sólo un usuario cualquiera.

Otra práctica de Clean Code que procuramos respetar fue la de evitar las clases demasiado grandes. Al crear una clase determinamos cuál sería la responsabilidad de esa clase y nos mantenemos enfocada en esa responsabilidad. Advertimos que hay unas clases anteriormente mencionadas en la justificación de diseño que tienen responsabilidad en exceso, pero al saber a dónde se tiene que transferir esa responsabilidad es posible arreglarlo en una siguiente entrega.

Como un ejemplo, tenemos las clases de DocumentCreationByUserGraphService y DocumentModificationByUserGraphService. La responsabilidad de estas clases es generar los datos necesarios para mostrar la gráfica de las funciones de administrador dictadas en la letra, y esta es su única responsabilidad. No alteran los estados del dominio o la base de datos porque no deben hacer mantenimiento y únicamente sirven para la función de la que son responsables.

Otra práctica de Clean Code es pasar pocos parámetros en un método. En la mayoría de los casos respetamos esto, pasando a lo sumo 2 parámetros por método. Las únicas excepciones son métodos en DocumentCreationByUserGraphService y DocumentModificationByUserGraphService que requieren 3 parámetros por tener dos DateTimes y un usuario, y uno de los métodos para filtrar y ordenar los documentos,

que requiere un boolean, un criterio de ordenamiento y la lista que hay que ordenar. Estos métodos se podrían evitar si alguno de los parámetros fuera un atributo, pero al estar encapsulados en la clase estos métodos grandes sólo impactan en esa área del programa.

Un code smells que reconocemos en el código es que hay ocasiones donde pasamos los styleClass como nulls, lo cuál nos generó dificultades a la hora de escribir la lógica. Una posible mejora de clean code sería borrar estos argumentos nulos creando un new StyleClass vacío siempre que se reciba algún nulo.

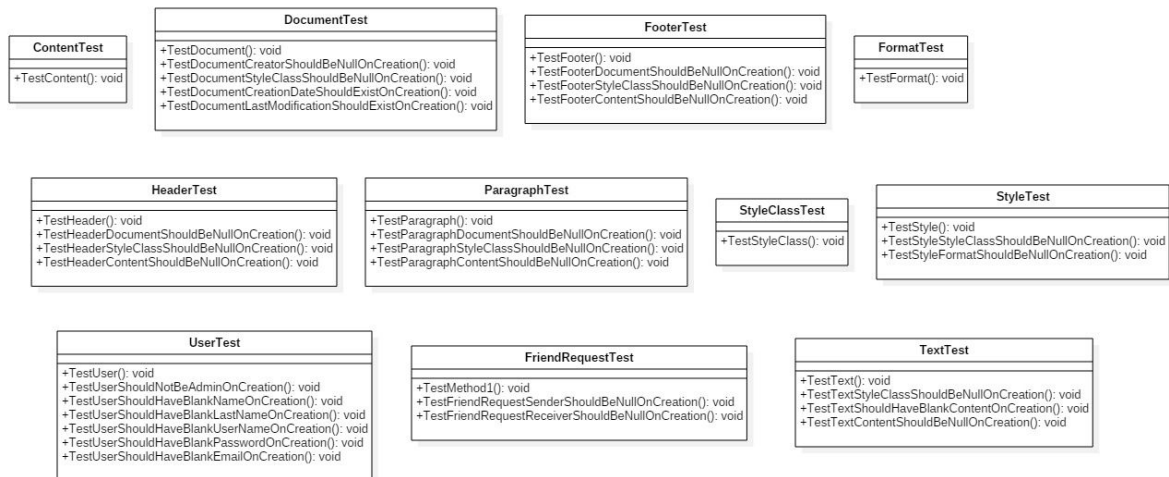
Otro problema es que algunos métodos en los ManagementServices sufren de 'choque de trenes', cuando es necesario operar con atributos de clases que son parte de otra clase de la que el ManagementService pone su principal enfoque (por ejemplo, cuando un DocumentManagementService pregunta por el nombre del StyleClass del Documento). La razón por la que hacemos esto y no crear una función del StyleManagementService que se ocupe de hacer la verificación lógica necesaria es porque queremos no acoplar los Managers entre sí, encapsulando las funciones de las que son responsables en sus clases. Así, si se cambia alguna parte de la lógica, no afecta como funciona el resto de ella.

Aparte de esos casos, evitamos el choque de trenes al asignar los objetos devueltos por un método a una variable siempre que estos eran utilizados, cuidando no dejar líneas demasiado largas en el código donde se pudieran evitar.

6. Pruebas

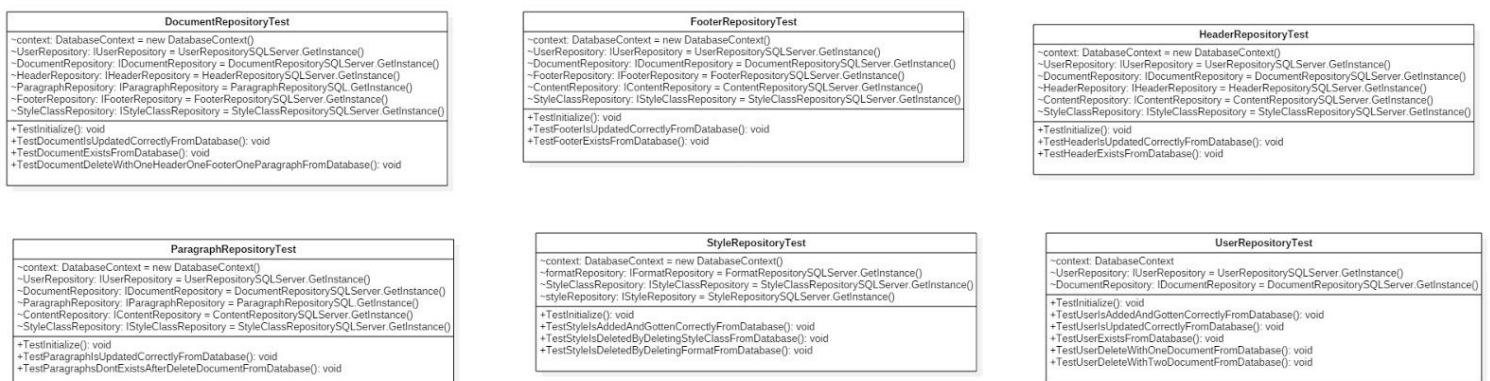
6.1. UML de clases de prueba

6.1.1. DomainTest



Se realizaron pruebas bastante básicas en el proceso de TDD, ya que nuestras clases del Dominio no contiene lógica, solamente contienen un constructor sin parámetros y atributos.

6.1.2. RepositoryTest



Se realizaron pruebas bastante pesadas para los métodos del repositorio para asegurar la funcionalidad de funciones complicadas. Dichas funciones ayudaron a encontrar errores en el repositorio y a comprobar que el funcionamiento estaba correctamente implementado.

6.1.3. ServiceTest

CommentServiceTest

```
+TestGetCommentsCallsRepositoryGetAll(): void
+TestAddCommentCallsRepositoryAddComment(): void
-GetFakeUser(): User
-GetFakeDocument(): Document
-GetFakeComment(): IEnumerable
```

DocumentCreationByUserGraphServiceTest

```
+TestGetDocumentCreationByUserGraphReturnsValidData(): void
-GetFakeUsers(): IEnumerable
-GetFakeDocuments(users: IEnumerable): IEnumerable
```

DocumentManagementTest

```
+TestGetDocumentsByUserWorksOnExistingUser(): void
+TestGetDocumentsByUserThrowsExceptionOnMissingUser(): void
+TestGetDocumentByIdWorksOnExistingDocument(): void
+TestGetDocumentByIdFailsOnMissingDocument(): void
+TestAddDocumentWorksOnExistingUser(): void
+TestAddDocumentFailsOnMissingUser(): void
+TestUpdateDocumentWorksOnExistingDocument(): void
+TestUpdateDocumentFailsOnMissingDocument(): void
+TestDeleteDocumentWorksOnExistingDocument(): void
+TestDeleteDocumentFailsOnMissingDocument(): void
-GetFakeDocuments(): IEnumerable
-GetFakeUser(): User
```

DocumentModificationByUserGraphServiceTest

```
-mockUserRepository: Mock
-mockDocumentRepository: Mock
-mockLoggingRepository: Mock
~modificationGraphLogic: IDocumentModificationByUserGraphService

«constructor»+DocumentModificationByUserGraphServiceTest()
+TestGetModificationsPerUserPerDayReturnsValidGraph(): void
-GetFakeUser(): User
-GetFakeDocument(fakeUser: User): Document
-GetFakeModifications(fakeDocument: Document): IEnumerable
```

StyleManagementServiceTest

```
-mockStyleRepository: Mock
-mockFormatRepository: Mock
-mockStyleClassRepository: Mock
-styleLogic: IStyleManagementService

«constructor»+StyleManagementServiceTest()
+TestGetStylesReturnsSomeStyles(): void
+TestAddStyleWorksOnValidStyle(): void
+TestAddStyleFailsOnInvalidStyle(): void
+TestDeleteStyleWorksOnExistingStyle(): void
+TestDeleteStyleFailsOnMissingStyle(): void
-GetFakeStyleClass(): StyleClass
-GetFakeFormat(): Format
-GetFakeStyles(): IEnumerable
```

HeaderManagementTest

```
+TestGetByDocumentReturnsHeaderOnExistingHeaderAndDocument(): void
+TestGetByDocumentFailsOnMissingDocument(): void
+TestGetByDocumentFailsOnMissingHeader(): void
+TestAddHeaderWorksOnMissingHeaderAndExistingDocument(): void
+TestAddHeaderFailsOnMissingDocumentAndHeader(): void
+TestAddHeaderFailsOnMissingDocument(): void
+TestUpdateHeaderWorksOnExistingHeader(): void
+TestUpdateHeaderFailsOnMissingHeader(): void
+TestDeleteHeaderWorksOnExistingHeader(): void
-GetFakeDocument(): Document
-GetFakeHeaderForDocument(document: Document): Header
```

FooterManagementTest

```
+TestGetByDocumentReturnsFooterOnExistingFooterAndDocument(): void
+TestGetByDocumentFailsOnMissingDocument(): void
+TestGetByDocumentFailsOnMissingFooter(): void
+TestAddFooterWorksOnMissingFooterAndExistingDocument(): void
+TestAddFooterFailsOnExistingDocumentAndFooter(): void
+TestAddFooterFailsOnMissingDocument(): void
+TestUpdateFooterWorksOnExistingFooter(): void
+TestUpdateFooterFailsOnMissingFooter(): void
+TestDeleteFooterWorksOnExistingFooter(): void
-GetFakeDocument(): Document
-GetFakeFooterForDocument(document: Document): Footer
```

FormatManagementTest

```
+TestGetAllFormatsCallsRepositoryGetAll(): void
+TestGetFormatWorksOnExistingFormat(): void
+TestGetFormatFailsOnMissingFormat(): void
+TestAddFormatWorksOnMissingFormat(): void
+TestAddFormatFailsOnExistingFormat(): void
+TestDeleteFormatWorksOnExistingFormat(): void
+TestDeleteFormatFailsOnMissingFormat(): void
-GetFakeFormats(): IEnumerable
```

HTMLGenerationTest

```
+TestHTMLGeneratorMakesDesiredString(): void
-GetFakeDocument(): Document
-AssignFakeHeader(document: Document): Header
-AssignFakeFooter(document: Document): Footer
-AssignFakeParagraphs(document: Document): IEnumerable
-AssignTextForHeader(header: Header): IEnumerable
-AssignTextForFooter(footer: Footer): IEnumerable
-AssignTwoTextsForParagraph(paragraph: Paragraph): IEnumerable
-AssignThreeTextsForParagraph(paragraph: Paragraph): IEnumerable
-GetStyleClassNormal(): StyleClass
-GetStyleClassTitle(): StyleClass
-GetStyleClassForm(): StyleClass
-GetFakeFormat(): Format
-GetFakeNormalStyle(): IEnumerable
-GetFakeFormStyle(): IEnumerable
-GetFakeTitleStyle(): IEnumerable
-GetFakeResultForFirstTest(): string
```

StyleClassManagementTest

```
+TestGetAllStyleClassesCallsRepositoryGetAll(): void
+TestAddStyleClassWorksOnMissingStyleClass(): void
+TestAddStyleClassFailsOnExistingStyleClass(): void
+TestAddStyleClassFailsOnRedundantDependency(): void
+GetStyleClassWorksOnExistingStyleClass(): void
+GetStyleClassFailsOnMissingStyleClass(): void
+DeleteStyleClassWorksOnExistingStyleClass(): void
+DeleteStyleClassFailsOnMissingStyleClass(): void
-GetFakeStyleClasses(): IEnumerable
```

TopsServiceTest

```
+TestGetTop3DocumentsByRatingWithCeroDocuments(): void
+TestGetTop3DocumentsByRatingWithOneDocuments(): void
+TestGetTop3DocumentsByRatingWithThreeDocuments(): void
-GetFakeUser(): User
-GetFakeDocument0(): Document
-GetFakeDocument1(): Document
-GetFakeComment(): IEnumerable
```

ParagraphManagementTest

```
-mockParagraphRepository: Mock
-mockDocumentRepository: Mock
-mockStyleClassRepository: Mock
-mockContentRepository: Mock
-paragraphLogic: IParagraphManagementService
































«constructor»+ParagraphManagementTest()
+TestGetAllParagraphsByDocumentWorksOnExistingDocument(): void
+TestGetAllParagraphsByDocumentFailsOnMissingDocument(): void
+TestAddParagraphWorksOnExistingDocument(): void
+TestAddParagraphFailsOnMissingDocument(): void
+TestUpdateParagraphWorksOnExistingParagraphAndValidPosition(): void
+TestUpdateParagraphFailsOnMissingParagraph(): void
+TestUpdateParagraphFailsOnInvalidPosition(): void
+TestDeleteParagraphWorksOnExistingParagraph(): void
+TestDeleteParagraphFailsOnMissingParagraph(): void
-GetFakeDocument(): Document
-GetFakeParagraph(): Paragraph
-GetFakeParagraphsForDocument(document: Document): IEnumerable
```

UserManagementTest

```
+TestGetAllShouldCallTheRepositoryAndReturnAListOfUsers(): void
+TestAddUserShouldAddAnUnexistingUserToTheList(): void
+TestAddUserShouldNotAddExistingUserToTheList(): void
+TestUpdateShouldModifyAnExistingUser(): void
+TestUpdateShouldNotModifyUnexistingUser(): void
+TestGetByEmailShouldGetAnExistingUserByEmail(): void
+TestGetByEmailShouldFailIfUserDoesNotExist(): void
+TestDeleteShouldDeleteExistingUser(): void
+TestDeleteShouldFailOnMissingUser(): void
-GetFakeUsers(): IEnumerable
```

6.2. Reporte de pruebas

repositorysqlserver.dll	2455	61,01 %	1569	38,99 %
{ } RepositorySQLServer	1805	60,84 %	1162	39,16 %
CommentRepository...	80	100,00 %	0	0,00 %
ContentRepositoryS...	217	85,43 %	37	14,57 %
DatabaseContext	5	6,25 %	75	93,75 %
DocumentModificati...	64	100,00 %	0	0,00 %
DocumentRepositor...	53	23,87 %	169	76,13 %
FooterRepositorySQL...	187	59,74 %	126	40,26 %
FormatRepositorySQ...	43	55,84 %	34	44,16 %
FriendRequestReposi...	276	100,00 %	0	0,00 %
HeaderRepositorySQ...	181	57,83 %	132	42,17 %
LogRepositorySQLSe...	22	100,00 %	0	0,00 %
ParagraphRepository...	143	51,62 %	134	48,38 %
RepositoryHandlerS...	45	100,00 %	0	0,00 %
SessionRepositorySQ...	125	100,00 %	0	0,00 %
StyleClassRepository...	105	33,76 %	206	66,24 %
StyleRepositorySQLS...	49	24,38 %	152	75,62 %
TextRepositorySQLSe...	202	100,00 %	0	0,00 %
UserRepositorySQLS...	8	7,62 %	97	92,38 %
serviceimp.dll	1521	59,48 %	1036	40,52 %
{ } ServiceImp	1521	59,48 %	1036	40,52 %
AuthenticationService	443	100,00 %	0	0,00 %
AuthenticationServic...	10	100,00 %	0	0,00 %
CommentService	7	14,89 %	40	85,11 %
DocumentCreationB...	0	0,00 %	48	100,00 %
DocumentManagem...	109	56,48 %	84	43,52 %
DocumentManagem...	8	100,00 %	0	0,00 %
DocumentModificati...	1	1,75 %	56	98,25 %
DocumentModificati...	102	100,00 %	0	0,00 %
FooterManagementS...	17	18,28 %	76	81,72 %
FormatManagement...	0	0,00 %	34	100,00 %
FriendRequestMana...	134	100,00 %	0	0,00 %
GenericStyleBuilder	53	76,81 %	16	23,19 %
HTMLGenerator	27	13,50 %	173	86,50 %
HTMLGenerator.<>c	0	0,00 %	4	100,00 %
HTMLGenerator.<>c...	0	0,00 %	4	100,00 %
HTMLGenerator.<>c...	0	0,00 %	8	100,00 %
HeaderManagement...	17	18,28 %	76	81,72 %
LogInService	13	100,00 %	0	0,00 %
LoggingService	44	100,00 %	0	0,00 %
ParagraphManagem...	15	11,45 %	116	88,55 %

▸  ParagraphManagem...	0	0,00 %	2	100,00 %
▸  ServiceHandler	195	100,00 %	0	0,00 %
▸  SessionService	45	100,00 %	0	0,00 %
▸  StyleClassManagem...	40	37,04 %	68	62,96 %
▸  StyleHTML	0	0,00 %	4	100,00 %
▸  StyleHTMLBuilder	16	26,23 %	45	73,77 %
▸  StyleManagementSe...	11	13,75 %	69	86,25 %
▸  TextManagementSer...	210	100,00 %	0	0,00 %
▸  TextManagementSer...	2	100,00 %	0	0,00 %
▸  TopsService	2	3,13 %	62	96,88 %
▸  TopsService.<>c	0	0,00 %	2	100,00 %
▸  TopsService.<>c_Di...	0	0,00 %	5	100,00 %
▸  UserManagementSer...	0	0,00 %	44	100,00 %
▾  domain.dll	66	28,33 %	167	71,67 %
▾  { } Domain	66	28,33 %	167	71,67 %
▸  Comment	3	18,75 %	13	81,25 %
▸  Content	0	0,00 %	4	100,00 %
▸  Document	0	0,00 %	19	100,00 %
▸  DocumentFilterAnd...	6	100,00 %	0	0,00 %
▸  DocumentModificati...	2	33,33 %	4	66,67 %
▸  Footer	0	0,00 %	13	100,00 %
▸  Format	3	37,50 %	5	62,50 %
▸  FriendRequest	3	23,08 %	10	76,92 %
▸  Header	0	0,00 %	13	100,00 %
▸  LoggedEntry	15	100,00 %	0	0,00 %
▸  Paragraph	0	0,00 %	16	100,00 %
▸  Session	8	100,00 %	0	0,00 %
▸  Style	10	38,46 %	16	61,54 %
▸  StyleClass	15	44,12 %	19	55,88 %
▸  Text	1	6,25 %	15	93,75 %
▸  User	0	0,00 %	20	100,00 %

Se realizaron también 80 pruebas de postman, las cuales sirvieron como pruebas de integración y pruebas funcionales del software. Las cuales fueron corridas a mano, verificando en la base de datos y comprobando que estaban los resultados correctamente de las 80 pruebas.

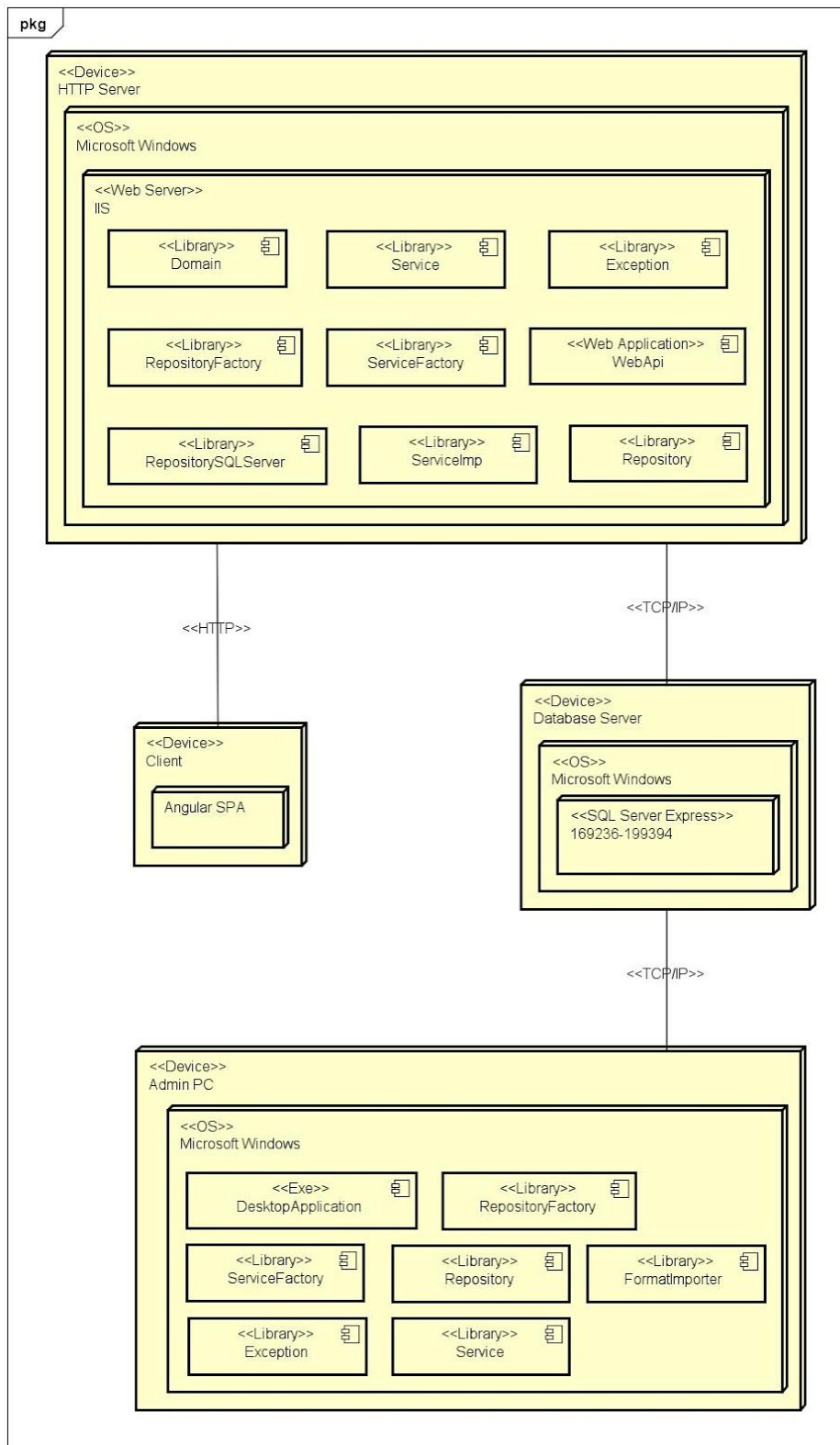
A pesar de tener poca cobertura de la lógica, reforzamos esto con pruebas de integración en el Postman, que verifican la interacción de la lógica con el repositorio y la WebApi.

La principal razón por la ausencia de pruebas unitarias es haber terminado de desarrollar los controladores y el repositorio cerca de la fecha de entrega, lo cual dejó poco tiempo para realizar una cobertura completa con pruebas unitarias.

Una buena práctica de las pruebas de ServiceImp es que todas utilizan de la tecnología Mock para llamar a las interfaces del repositorio, por lo que las pruebas que sí están ahí solamente prueban el comportamiento de la lógica de negocio como tendría que ser.

A lo largo del segundo obligatorio se realizaron varias pruebas más, pero la cobertura no mejoró. Podemos atribuir esto a la presencia de varias clases con bloques lógicos muy extensas, como AuthenticationService que posee más de 400 bloques.

7. Instalación



1. Iniciar el servicio de Windows de Microsoft SQL Server (MSSQLSERVER)
2. Activar Internet Information Services si no está activado
3. Copiar la carpeta de los releases en C:\inetpub\wwwroot.
4. Actualizar el archivo web.config en la carpeta del release, en la línea ConnectionStrings con el nombre del servidor que se utilizará.
5. Abrir el Administrador de IIS con el comando de ejecutar "inetmgr".
6. Agregar un nuevo sitio con un nombre cualquiera y la ruta física de la carpeta de la WebApi en wwwroot, y un puerto sin utilizar (borrar el sitio por defecto para usar 80).
7. Hacer lo mismo con la carpeta de Angular.
8. Abrir SQL Server Management Studio 2014 y conectarse a la base de datos.
9. Abrir la carpeta Seguridad en el SQL Server 2014 con el botón derecho y crear un nuevo inicio de sesión bajo el dominio IIS APPPOOL\{Nombre del Sitio} y ponerle todos los roles.
10. Hacer lo mismo para el sitio de Angular.
11. Convertir la carpeta dentro del sitio de Angular en una aplicación.
12. Si tira un error, hay que utilizar IIS APPPOOL/DefaultAppPool en vez del sitio.
13. Actualizar los sitios con el botón Examinar.

8. Justificaciones de Diseño de las Nuevas Funcionalidades

8.1. Red Social

Domain :

Se creó una `ICollection<User>` para que el usuario pudiese tener una lista de amigos dentro de la clase `User` del Dominio.

Repository :

Se creó una nueva tabla en la base de datos llamada `Friendship` donde se muestra la relación de amistad entre 2 usuarios, la cual tiene el mail de ambos usuarios relacionados.

El usuario podrá visualizar la lista de usuarios registrados en el sistema.

Controller :

Se crea un controller `GetFriends` en `UserController` que recibe por Uri el usuario que quiere visualizar la lista de usuarios.

El usuario podrá enviar solicitud de amistad a otro usuario registrado en el sistema.

Domain :

Se crea una clase FriendRequest la cual tiene un Sender que es el usuario que envía la solicitud de amistad, tiene un Receiver que es el usuario receptor de la solicitud de amistad y tiene un Id que determina la identificación de la solicitud.

Controller :

Se crea un controller Post en UserController en la ruta `/users/{user_email}/sendrequest`, llamado SendFriendRequest que recibe como parámetro el mail del usuario a agregar.

El usuario una vez invitado para ser amigo, podrá aceptar o rechazar la invitación. En caso de que acepte, se comparte la información de sus documentos entre ambos.

Controller :

Se crea un controller Post en UserController en la ruta `/users/{user_email}/respondrequest`, llamado RespondFriendRequest que recibe como parámetro el mail del usuario a agregar y un bool que representa la respuesta.

El usuario podrá visualizar el perfil de un amigo y la información de sus documentos

Controller : Se utiliza el controller Get de UserController, previamente creado para el obligatorio 1, para que un usuario pueda ver un perfil de otro usuario.

Se agrega una autenticación para que el usuario solo pueda ver perfiles de un amigo o su perfil propio.

Solamente usuarios administradores son capaces de ver los perfiles de todos los usuarios.

Se agrega autenticación para que un usuario pueda ver los documentos de los usuarios de sus amigos.

Se utiliza el controller GetAll de DocumentController para recibir dichos documentos.

El usuario podrá realizar una valoración (0 a 5 estrellas) y un comentario al documento de un amigo.

Controller :

Se crea un controller CommentController que se encarga de agregar una valoración con comentario.

Service :

Se crea CommentService que se encarga de la lógica para el manejo de las valoraciones y los comentarios.

Repository :

Se crea CommentRepository que se encarga de las preguntas a la base de datos.

Domain :

Se crea Comment que posee al usuario Commenter, un Document, Text y Rating.

En la ventana principal del usuario, se muestran el top 3 de los documentos con más estrellas del sistema.

Controller :

Se crea un controller `TopsController` que se encarga los tops del sistema, entre ellos el top 3 de los documentos con más estrellas del sistema.

Service :

Se crea `TopsService` que se encarga de la lógica para el manejo de los tops del sistema.

Repository :

Se crea `TopsRepository` que se encarga de las preguntas a la base de datos.

8.2. Nuevos Estilos

Agregar el estilo Borde y el tipo de letra Verdana al sistema

Para agregar los nuevos estilos, solamente hizo falta cambiar los builders de la solución, tanto el `GenericStyleBuilder` que construye los estilos aceptados por el sistema como el `HTMLStyleBuilder`, que construye el código HTML de dichos estilos.

8.3 Sistema de Logs

Para implementar el Logger, se creó una nueva entidad llamada `LoggedEntry` en el Domain. Esta entidad tiene la arquitectura de la entrada de un log, con un tipo de registro, el nombre de usuario del usuario conectado y la fecha de registro del log.

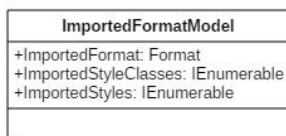
Decidimos hacerlo de esta manera y no con un builder debido a que los requerimientos sólo pedían extensibilidad a la hora de mantener el log de distintas maneras, por lo que decidimos no implementar un builder para la estructura básica pero sí para su implementación.

Para asegurar la extensibilidad del logger en tiempo de compilación utilizamos una interfaz `ILogService`, a tal modo de que simplemente reemplazando la clase de implementación `LoggingService` se cambia la manera de mantener el Log.

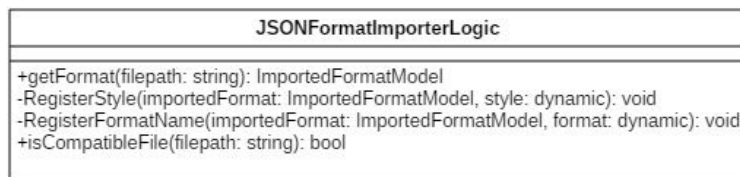
También se agregó una interfaz `ILogRepository` e `LogRepositorySQLServer`, porque la manera de mantener el log también podría pasar de una base de datos a otro modo.

8.4 Importador de Formatos

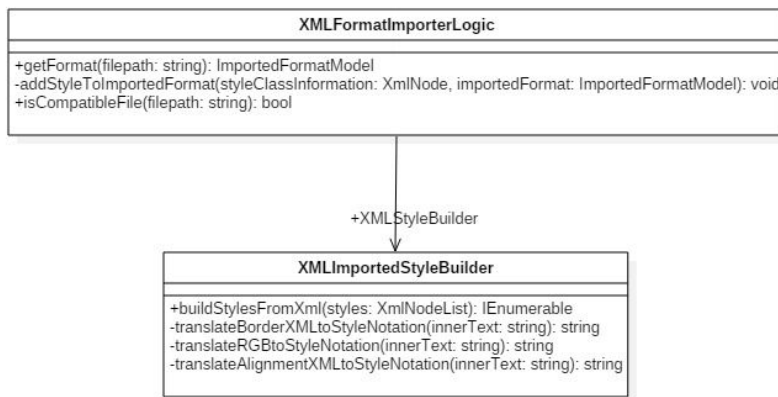
FormatImporter



JSONFormatImporter



XMLFormatImporter



Para la implementación del importador de formatos, creamos una interfaz llamada `IFormatImporterLogic`, la cuál tiene la responsabilidad de dar las operaciones necesarias para importar un formato. Para esto, se especificó una clase llamada `ImportedFormatModel` que se guarda el formato a importar, las clases de estilo y los estilos respectivos a importarse.

Tanto `JSONFormatImporter` como `XMLFormatImporter` fueron construidos en assemblies distintos, y ningún assembly los referencia directamente. En vez de eso, en la capa de interfaz con el usuario es él en tiempo de ejecución que selecciona el .dll de importer que quiere usar, y es la responsabilidad del importador concreto saber que archivos puede importar. Este problema lo solucionamos utilizando `Reflection`.

En el caso del `XMLFormatImporter`, también tuvimos que crearle un builder específico de estilos, debido a que el formato solicitado por el cliente era distinto al lenguaje convenido internamente en el programa para los estilos. Este es el `XMLImportedStyleBuilder` que se encuentra dentro de su assembly.

9. Informe de Métricas

9.1 Lista de Métricas

Assemblies	# lines of code	# IL instruction	# Types	# Abstract Types	# lines of comment	% Comment	% Coverage	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
Domain v1.0.0.0	175	678	16	0	17	8.85	-	153	9	1.38	0.06	0	0.67
Repository v1.0.0.0	0	0	16	16	-	-	-	38	20	1	0.34	1	0.24
RepositorySQLServer v1.0.0.0	774	15425	48	0	18	2.27	-	0	95	1.27	1	0	0
Service v1.0.0.0	0	0	22	22	-	-	-	51	22	0.95	0.3	1	0.21
ServiceFactory v1.0.0.0	9	47	1	0	17	65.38	-	2	14	1	0.88	0	0.09
Exception v1.0.0.0	24	140	23	0	17	41.46	-	34	4	0.04	0.11	0	0.63
RepositoryFactory v1.0.0.0	9	47	1	0	17	65.38	-	1	14	1	0.93	0	0.05
ServiceImp v1.0.0.0	1083	7063	24	0	20	1.81	-	0	106	1.12	1	0	0
WebApi v1.0.0.0	1073	7554	72	1	20	1.83	-	0	112	1.9	1	0.01	0.01
FormatImporter v1.0.0.0	6	25	2	1	17	73.91	-	4	10	1	0.71	0.5	0.15
DesktopApplication v1.0.0.0	813	4485	12	0	358	30.57	-	0	90	0.92	1	0	0
XMLFormatImporter v1.0.0.0	93	526	2	0	17	15.45	-	0	27	1	1	0	0
JSONFormatImporter v1.0.0.0	23	499	4	0	17	42.5	-	0	34	1.25	1	0	0

Estos fueron los resultados obtenidos luego de analizar la solución del obligatorio con nDepend. Observando el Afferent Coupling, Efferent Coupling, la Inestabilidad, y la Abstracción podemos utilizar estas métricas para analizar la extensibilidad del diseño de nuestro obligatorio.

Afferent Coupling (Ca)

Es el número de tipos (clases, interfaces, enums, etc.) externas al assembly que dependen de nuestro assembly. Cuánto más alto el afferent coupling, mayores los indicadores de que los assemblies relacionados tienen mucha responsabilidad.

El mayor afferent coupling lo tiene el Dominio, lo cual es adecuado al diseño del obligatorio. Se supone que el Dominio sea el assembly con la mayor importancia porque el dominio es el que define el 'idioma' que tienen que usar los módulos para aplicar al negocio.

Las otras assemblies con mucho afferent coupling son Repository, Service y Exception. Repository y Service son interfaces que definen las operaciones que hay que hacer en el acceso a datos y la lógica de negocio, por lo que los assemblies de sus implementaciones dependen de ellos. Exception, por otro lado, define varios tipos de excepciones personalizadas para el negocio, por lo que la lógica de negocio la usa mientras que assemblies de servicios también la utilizan para capturar las excepciones.

Efferent Coupling (Ce)

El Efferent Coupling indica la cantidad de tipos externos al assembly utilizados por los tipos hijos del assembly examinado. Cuanto mayor el Ce, más dependiente es la assembly.

Las assemblies con mayor efferent coupling son ServiceImp, RepositorySQLServer, WebApi y DesktopApplication. Esto es adecuado al diseño, puesto que ServiceImp y RepositorySQLServer necesariamente dependen del Dominio e implementan las interfaces de Service y Repository. En cuanto a la WebApi y DesktopApplication, ambas dependen de la lógica de negocio (Domain y Service), por lo que son assemblies dependientes.

Inestability (I)

La Inestability es la proporción de Efferent Coupling respecto al Coupling total (es decir: $Ce / (Ca + Ce)$). Es un indicador de 0 a 1, con un 0 representando un assembly estable que no es susceptible a cambiar y un 1 representando un assembly altamente inestable y propenso al cambio.

Las únicas assemblies con baja inestabilidad son las del Domain, Repository, Service y Exception. Esto tiene sentido para las primeras tres, ya que Domain es una abstracción de las entidades involucradas en el modelo de negocio mientras que Service y Repository son interfaces, lo cuál hace que tengan que ser estables porque cambiar la interfaz repercute en todas sus implementaciones.

En el caso de Exception, la estabilidad ocurre debido a que la responsabilidad del assembly es proveer excepciones al sistema, por lo que el que se agreguen errores nuevos para controlar no impacta en la solución, pero borrar errores afectaría el manejo de ellos en otras clases.

En cuanto a FormatImporter, que es un assembly que provee la interfaz para los importadores de formatos, su importante inestabilidad viene de la decisión de crear un modelo de formatos importados para reducir los parámetros enviados por los métodos, lo cual generó una dependencia al dominio.

Los otros assemblies son inestables porque sus tipos dependen de tipos de assemblies que cumplen roles más abstractos.

Abstractness (A)

La Abstractness, o Abstracción, mide la proporción de tipos abstractos respecto a la cantidad de tipos totales dentro de un assembly. Efectivamente indica que tan abstracta es la clase, con un 0 representando una clase perfectamente concreta y un 1 una clase completamente abstracta.

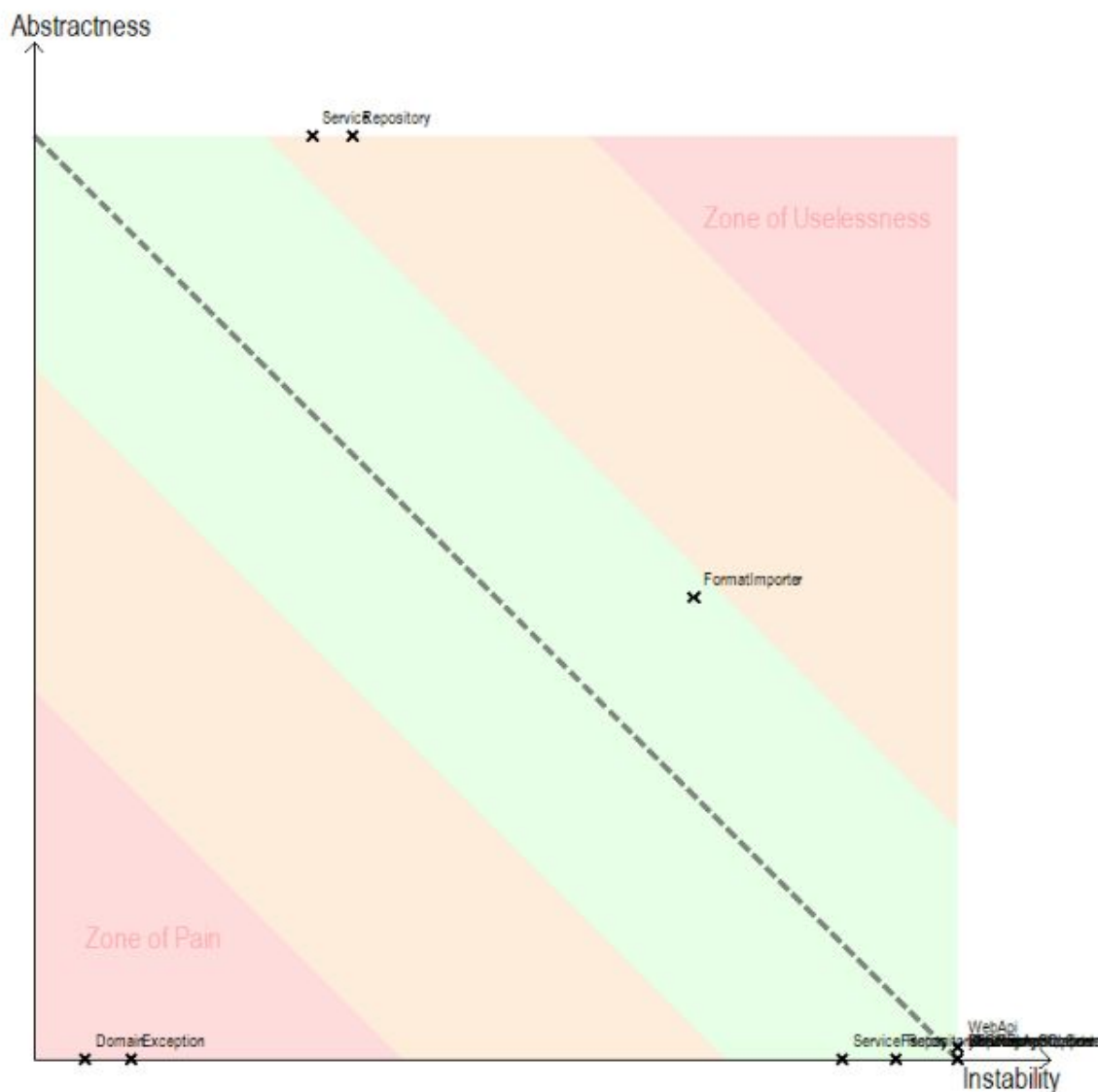
Las únicas clases abstractas son FormatImporter, Service y Repository, que son assemblies que se guardan las interfaces que son necesarias para implementar el diseño de la lógica de negocio y el acceso a datos.

Relational Cohesion

La cohesión relacional mide que tan fuertemente relacionados están los tipos dentro de un assembly particular. Este número idealmente debería ser alto, con coeficientes entre 1.5 y 4.0 dentro del rango de lo aceptable.

Generalmente, la cohesión relacional de la solución es baja. Esto es una consecuencia de la decisión de diseño que tomamos de encapsular cada funcionalidad separada de la lógica de negocio, con el fin de que si fuera necesario cambiar alguna, el resto no sería impactada por el cambio.

9.2 Gráfica de Abstractness vs Instability



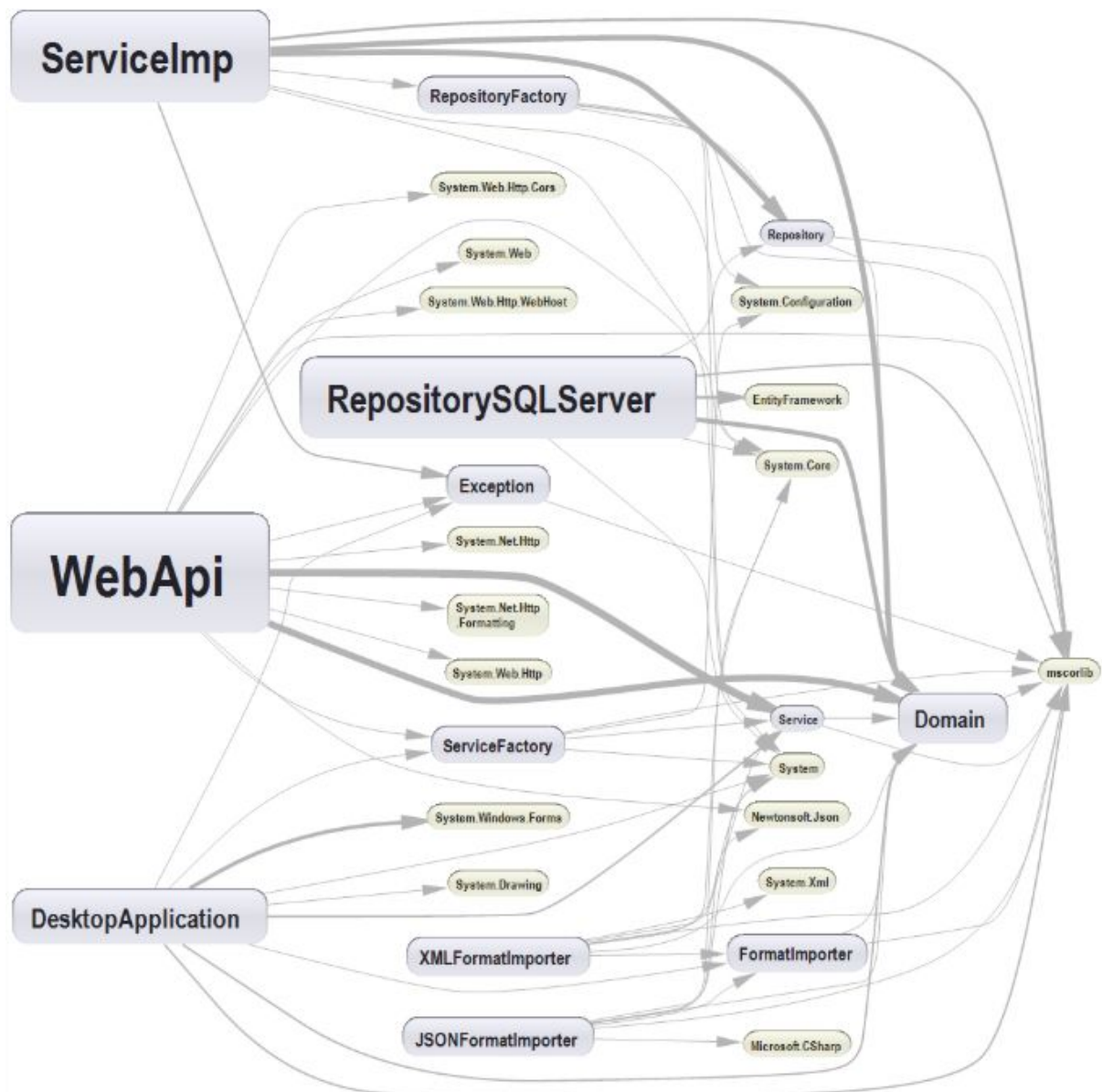
Como podemos ver la gran mayoría de los assemblies se encuentran dentro de la zona deseable, con las únicas dos excepciones siendo Domain y Exception.

Service y Repository se encuentran en la zona “amarilla” debido a que su utilidad es limitada al sólo ser interfaces, pero están suficientemente cerca de la zona verde como para que esto no sea tan problemático.

Domain y Exception son casos especiales. Domain es un assembly que encierra toda la abstracción de las entidades que queremos manipular en el negocio. Debido a esto sabemos que su impacto de cambio en los otros assemblies es enorme, pero por otro lado es muy raro que se desee realizar una alteración a tipos del dominio ya existentes.

Exception comparte estas características, aunque es más esencial para el manejo de errores que para definir las entidades del negocio. Aun así, no está previsto el diseño que sea necesario modificar los tipos existentes en el assembly.

9.3 Grafo de Dependencia



Cómo podemos ver en el grafo de dependencias, hay un claro respeto por el principio SOLID de inversión de dependencias. Las clases concretas se encuentran más a la izquierda, y solamente dependen de los assemblies de mayor nivel de abstracción.

La gran mayoría de los assemblies tienen dependencias con el assembly del Domain, pero esto es de esperarse dado el diseño de la solución.

9.4 Principios de Paquetes

REP: Release/Reuse Principle

Nuestro obligatorio cuenta con media reusabilidad de paquetes. Nos aseguramos de encapsular las áreas claves del obligatorio en las 'cajas negras' como son los assemblies .dll.

Sin embargo, los paquetes no recurren a un sistema de Tracking para llevar en cuenta su versionado, lo cual limita su reusabilidad. A pesar de esto respetamos la documentación, mantenibilidad y confiabilidad del paquete registrando todos los cambios hechos y minimizando el impacto de ellos.

CCP: Common closure principle

Podemos argumentar que en general se respeta el CCP en nuestro obligatorio, debido a nuestra decisión de encapsular cada aspecto del obligatorio en un paquete distinto. Los paquetes de servicios se encuentran dentro de un mismo paquete, igual que los del dominio o los de la capa de servicios. Esto se aplica siempre que se encuentre alto acoplamiento entre clases que deberían estar en la misma capa.

CRP: The common reuse principle

Por la misma razón que respetamos el CCP, respetamos el CRP. Las clases que deberían utilizarse juntas, como son las entidades del dominio, los servicios o los accesos a base de datos, están todos encapsulados dentro de sus respectivos paquetes.

ADP: The acyclic dependencies principle

Nuestros paquetes respetan el ADP. Visual studio no permite utilizar paquetes con dependencias circulares.

SDP: The stable dependency principle

Podemos observar que el SDP es bien respetado gracias al informe de métricas mostrado anteriormente. Las clases más estables son Service, Repository, Domain y Exception. De éstas, sólo Service y Repository dependen de otro paquete de los mencionados y este paquete es Domain, que es aún más estable que ellas.

SPA: The stable abstraction principle

Nuestro paquete viola el SPA debido a los paquetes Domain y Exception, que a pesar de ser clases concretas son altamente estables. Esto es porque Exception solo se usa en manejo de errores y las clases internas a su paquete no deberían ser modificadas, lo mismo se aplica al Dominio, que por naturaleza debería ser estático.

10. FrontEnd (Angular)

Para el FrontEnd de la aplicación, se requería hacerlo con el Framework Angular, entonces fue implementado con Angular.

La composición del proyecto Angular fue realizada mediante :

Modules :

app.module.ts : Module principal de la aplicación. Es el module que contiene todos los servicios, componentes y importa modules a utilizar del proyecto.

app-config.ts : Contiene el header base para los request a la web api (Authorization con el token del usuario) y también contiene el puerto y el dominio principal del url. El cual se completará en cada uno de los servicios, brindando ellos, su identificación de ruta correspondiente.

app-routing.module.ts : Contiene las rutas dentro del proyecto Angular, las cuales redirigen a cada uno de los componentes de la aplicación.

app-material.module : Contiene los módulos principales para poder hacer el display de la interfaz de usuario, el cual ayuda a que la aplicación luzca de mejor manera. Es una librería de google.

Services :

Se realizaron servicios donde se encapsulan los requests a la WebApi de cada uno de los controllers. Por lo tanto hay un service, por cada controller de la WebApi, y dicho service realiza requests a su contraparte de la WebApi.

También contiene un service de catch, el cual se encarga de recibir los errores que pueda lanzar la WebApi y replicarlos de manera más prolija con un error.

auth.guard.ts : se encarga de controlar si el usuario se encuentra logueado a la hora de acceder a una de las rutas del sistema.

Entities :

Se creó una entidad por cada uno de los posibles retornos de JSONs de la WebApi, para poder mapear dicho JSON a una clase de TypeScript respectiva.

Como caso particular, se creó una clase coordinate, la cual se encarga de recibir la tupla de datos enviada por la WebApi y parsearla en la clase coordinate. La cual no se encuentra en la lógica del sistema, solamente en Angular.

Components :

Se crearon diversos components descriptivos por nombre, el cual muestra el funcionamiento de dicho componente.

Por ejemplo, comment-add, es un component con la funcionalidad de agregar un comment al sistema.

Se utilizaron dialogs para hacer pop-ups interactivos con el usuario y facilitar el add y el update de la mayoría de los componentes del sistema.

Se intentó realizar la mayor cantidad de componentes posibles para que sea más fácil el desarrollo y el entendimiento del código. A pesar de no haber contado con experiencia previa en Angular, se intentó hacerlo lo más entendible y “Clean Code” posible.