

# Universidad ORT Uruguay

## Facultad de Ingeniería

### Diseño de aplicaciones II

#### Obligatorio 2

#### Descripción del diseño

Nicolas Varela 259432

Guillermo Odabachian 231181

Grupo N6B

Docentes: - Gabriel Piffaretti

- Joselen Cecilia

-Marco Fiorito

Link al repositorio: [https://github.com/ORT-DA2/259432\\_231181](https://github.com/ORT-DA2/259432_231181)

# Índice

<b>Descripción del diseño</b>	<b>2</b>
a) Descripción general del trabajo	2
b) Errores conocidos	3
c) Diagrama general de paquetes	5
Descomposición de paquetes	6
PharmaCity.IBusinessLogic	7
PharmaCity.BusinessLogic	7
PharmaCity.IDataAccess	8
PharmaCity.DataAccess	9
PharmaCity.DataAccess.Migrations	10
PharmaCity.DataAccess.Context	11
PharmaCity.Domain	11
PharmaCity.Domain.DTO	12
PharmaCity.Domain.IN	13
PharmaCity.BusinessLogic.Tools	13
PharmaCity.WebApi.Controllers	14
PharmaCity.WebApi.Filters	14
PharmaCity.Factory	15
d) Jerarquías de herencia utilizadas	16
e) Modelo de tablas de la base de datos	16
f) Diagramas de secuencia	17
g) Justificación de diseño	19
h) Métricas	23
i) Diagrama de componentes	29
j) Cambios realizados	30
k) Mecanismos de extensibilidad	31
l) Descripción del manejo de excepciones	32
<b>ANEXO:</b>	<b>33</b>
Evidencia del diseño y especificación de la API	33
a) Discusión de los criterios seguidos para asegurar que la API cumple con los criterios REST	33
b) Descripción del mecanismo de autenticación de requests.	34
c) Descripción general de códigos de error.	34
d) Descripción de los resources de la API.	35
Evidencia de Clean Code y de la aplicación de TDD	44
a. Descripción de la estrategia de TDD seguida	44
c. Es importante mantener una clara separación de los proyectos de prueba de los de la solución.	47
d. Funcionalidades especificadas como prioritarias (*)	48

# Descripción del diseño

## a) Descripción general del trabajo

Para la materia de Diseño de aplicaciones 2 se nos encomendó diseñar e implementar tanto Backend como Frontend de un sistema para gestión de farmacias, en nuestro caso llamado “PharmaCity”, basado el backend en una correcta implementación de una API REST. Esta api es consumida por el frontend desarrollada en el framework Angular.

El sistema está desarrollado en .NET Core 5.0, dicha versión fue elegida en base a la experiencia de los desarrolladores.

Este sistema permite gestionar farmacias, de las cuales se manejan usuarios donde cada uno posee un rol específico para gestionar su accionar, en los cuales se encuentran administrador, dueño, empleado y anónimo. Donde los administradores podrán dar de alta farmacias y crear invitaciones a nuevos usuarios para poder integrarlos al sistema, y si estas no son utilizadas podrán ser editadas. Dueños los cuales pueden visualizar solicitudes de reposición de stock de medicamentos en su respectiva farmacia, de los cuales puede aceptar o rechazar la solicitud. Empleados los cuales pueden dar de alta un medicamento que comienza sin stock disponible, además pueden solicitar reposición de stock de medicamentos y estos deben ser aceptados o rechazados por el dueño de la farmacia, además de esto pueden aceptar o denegar las compras solicitadas por los clientes. Por último los usuarios anónimos los cuales no tienen necesidad de ser dados de alta en el sistema, estos pueden listar todos los medicamentos de todas las farmacias, pudiendo filtrarlos por nombre de medicamento y/o farmacias con stock, una vez decidido, puede comprar uno o varios medicamentos junto a la cantidad de cada uno, deben ser de la misma farmacia y le va a dejar concretar la compra si de todos ellos hay stock disponible. Posteriormente si la compra es exitosa esta se registra junto a un código de compra y los medicamentos comprados en la respectiva farmacia.

En el transcurso del desarrollo del sistema tratamos de cubrir todos los casos posibles tanto en backend como en nuestro frontend y que estos respondan de la manera que deben, atrapando en nuestro entendimiento la totalidad de los casos, además validando esto al utilizar casos bordes para cumplir la correctitud de este punto.

Todas las funcionalidades requeridas en la solicitud de este proyecto, fueron implementadas en el sistema y consumidas por el frontend en Angular.

## b) Errores conocidos

En este punto tratamos de que nuestro sistema sea lo más completo posible, probando con casos bordes las funcionalidades y pensando en cada una de las situaciones que se podría llegar a dar en el sistema, los bugs que serán nombrados no fueron arreglados por falta de tiempo, pero se pueden arreglar fácilmente en actualizaciones futuras. En base a esto tenemos como bugs, que un empleado logueado de una determinada farmacia, si conoce el código de una medicina específica puede borrarla aunque sea de otra farmacia. Este error no es difícil de arreglar ya que enviando el token por la consulta y chequeando que coincidan la farmacia del usuario y medicina, pero debido a los tiempos y que conlleva hacer tdd decidimos dejarlo y documentarlo.

Cuando el empleado registrado entra a ver las compras, para allí aceptarlas o rechazarlas, visualiza todas las peticiones de medicinas dentro de cada compra, no siendo todas de su propia farmacia.

Al momento de aceptar las compras por parte de un empleado, se aceptan todas las peticiones de la compra de la farmacia de dicho empleado, no cada una por separado. En un principio con este punto asumimos que si de una compra realizada por un cliente, se iban a querer aceptar las peticiones de cada medicina, se irían a aceptar todas las peticiones de la correspondiente farmacia, ya que si no se está de acuerdo con una petición de medicina, es irrelevante aceptar algunas y dejar otras sin estar aceptadas, siendo que todas se deben aceptar, por lo cual nunca se realizaría esa compra. Al releer todo nuestro proyecto, cercanos a la fecha de entrega nos dimos cuenta de esta variante de nuestra implementación, decidiendo debido al poco tiempo dejarlo de esta forma.

En la implementación de la compra a la hora de confirmar la misma, si, en el periodo entre que se ingresó una compra y se acepta la misma cambia el stock de alguna de las medicinas involucradas en esa compra debemos tener en cuenta la posibilidad de que la compra ya no sea posible de efectuar, por lo que decidimos que el check de stock de las medicinas tiene que ser cuando la compra se esté por efectuar, es decir cuando todas las peticiones se activen, esto genera el problema de que podemos tener una petición activa pero que en realidad todavía no se haya efectuado realmente la compra de los medicamentos. Es un error que tiene poco

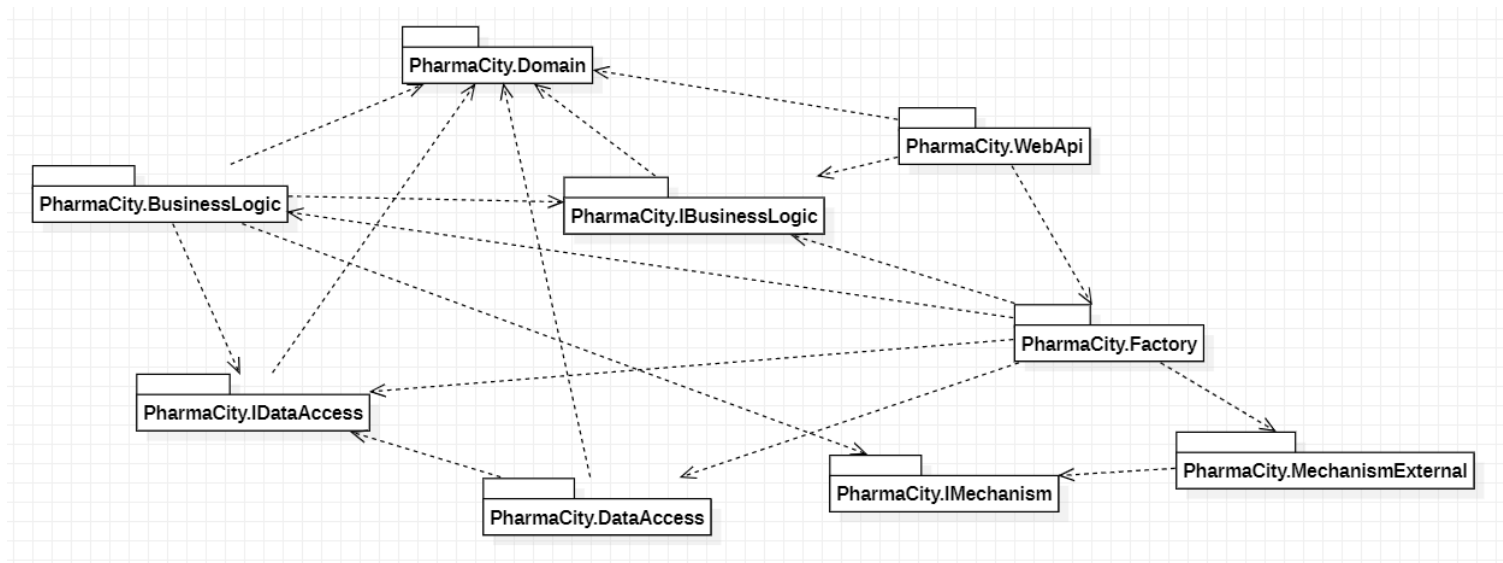
impacto funcional pero que sí se verá cuando se solicite el estado de una compra, donde podremos apreciar el estado activo de las peticiones que quizás ya no tengan stock o no se hayan efectuado.

Al ingresar un código de compra para visualizar el estado de la misma, visualiza el estado completo de la compra, no visualiza cada una de las peticiones dentro con su estado, este punto fue visto en el foro de forma tardía por lo cual por falta de tiempo queda de esta manera anteriormente mencionada, dando el estado de la compra.

En el caso de nuestro front-end, nos basamos como especificaremos más adelante en la reutilización de componentes, pero de la mano de esto somos conscientes que aunque aplicamos esto en muchos casos, en otros sabemos que podríamos haberlos aplicado pero por falta de tiempo en ocasiones no return utilizamos algún componente que ya tengamos creado. Siendo que este reuso es importante y es uno de los puntos fuertes de angular. También no conseguimos esconder de la barra de navegación las funcionalidades que no corresponden con el usuario iniciado, ya que para que estos desaparecieran o aparecieran dependiendo del rol del usuario se debía refrescar la página, por lo cual decidimos que sean visibles pero al momento de ingresar a estos se activen los guards respectivos.

Por último comentar un error de seguridad que tenemos en nuestro frontend, al utilizar localStorage para almacenar el token y el rol una vez que un usuario inicia sesión. Estas variables pueden ser editadas por quien esté navegando utilizando las “Herramientas web para desarrolladores” y así poder acceder a visualizar páginas que no tiene permiso (burlar los guards).

### c) Diagrama general de paquetes

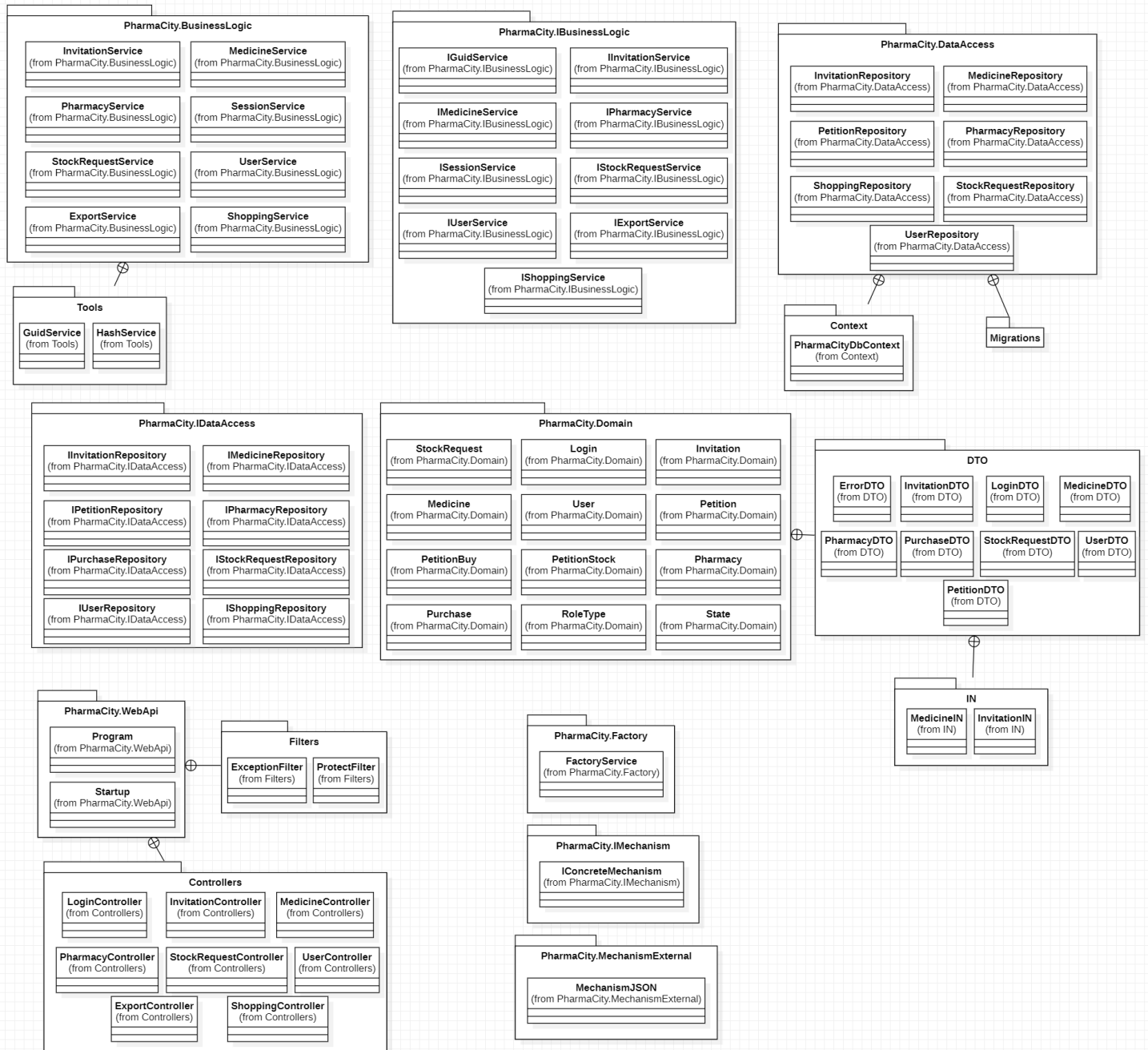


Gracias a este diagrama podemos ver como PharmaCity.WebApi se acopla a PharmaCity.IBusinessLogic y desde este se comunica a PharmaCity.IDataAccess haciendo que los módulos de alto nivel no dependan de módulos de bajo nivel sino que ambos dependen de interfaces que los comunican, gracias a esto se puede apreciar que cumple el principio de inversión de dependencias.

En este diagrama se aprecia cómo gracias al paquete PharmaCity.Factory la web api baja el acoplamiento, al no tener que depender la lógica de negocio y el acceso a datos para la inyección de dependencias (esto será detallado más adelante)

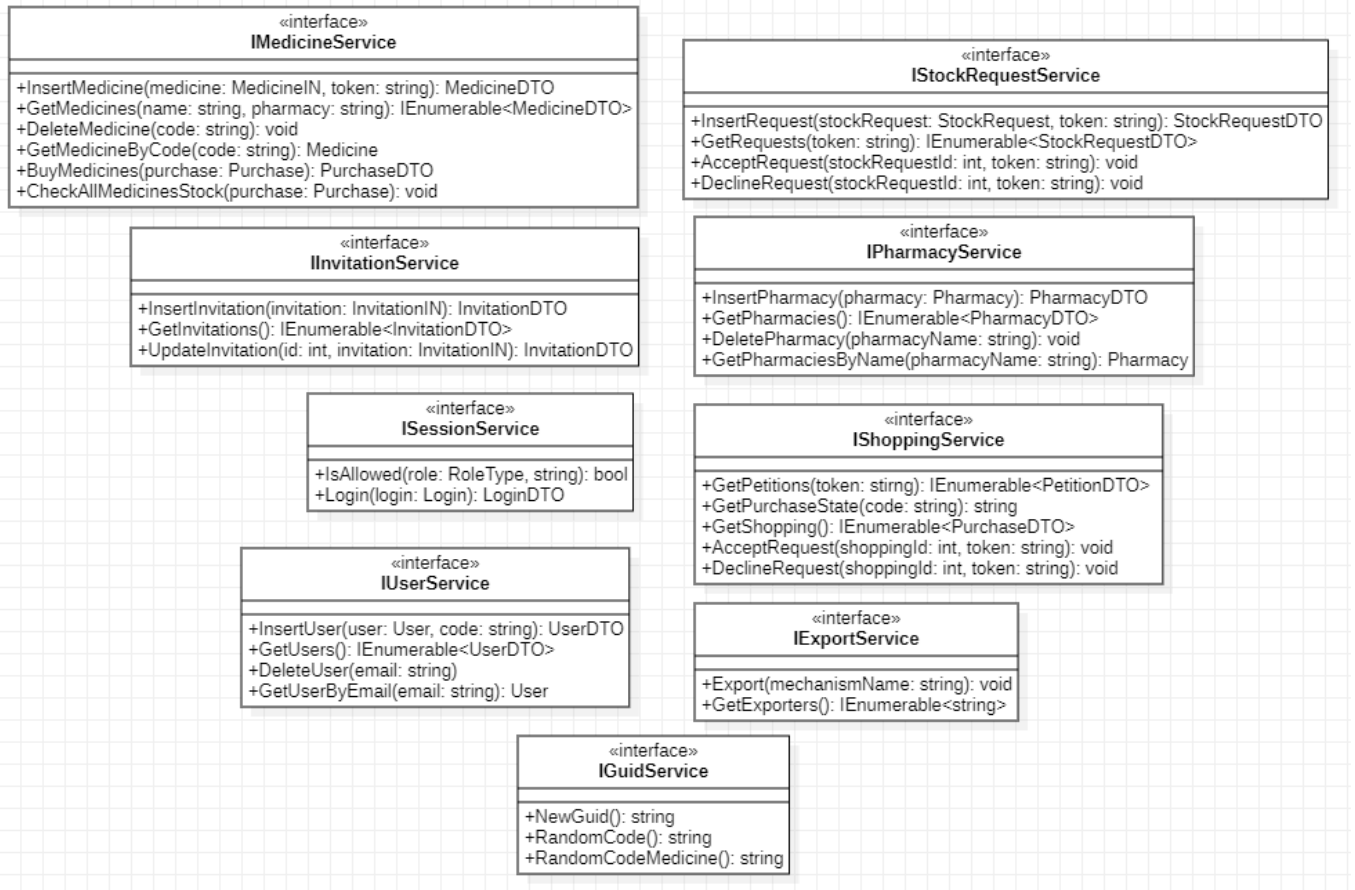
Además se puede visualizar como el proyecto de mecanismos externos ("PharmaCity.MechanismExternal"), debe conocer únicamente a IMechanism, la cual es la interfaz que debe implementar, luego no depende de ningún otro paquete, ni siquiera de Domain, ya que desacoplamos estos, manejando tipo de objetos genéricos en el mecanismo externo agregado, así no debe conocer a Domain para ser aplicados, haciendo que sea más fácil agregar nuevos mecanismos de exportación.

## Descomposición de paquetes



## PharmaCity.IBusinessLogic

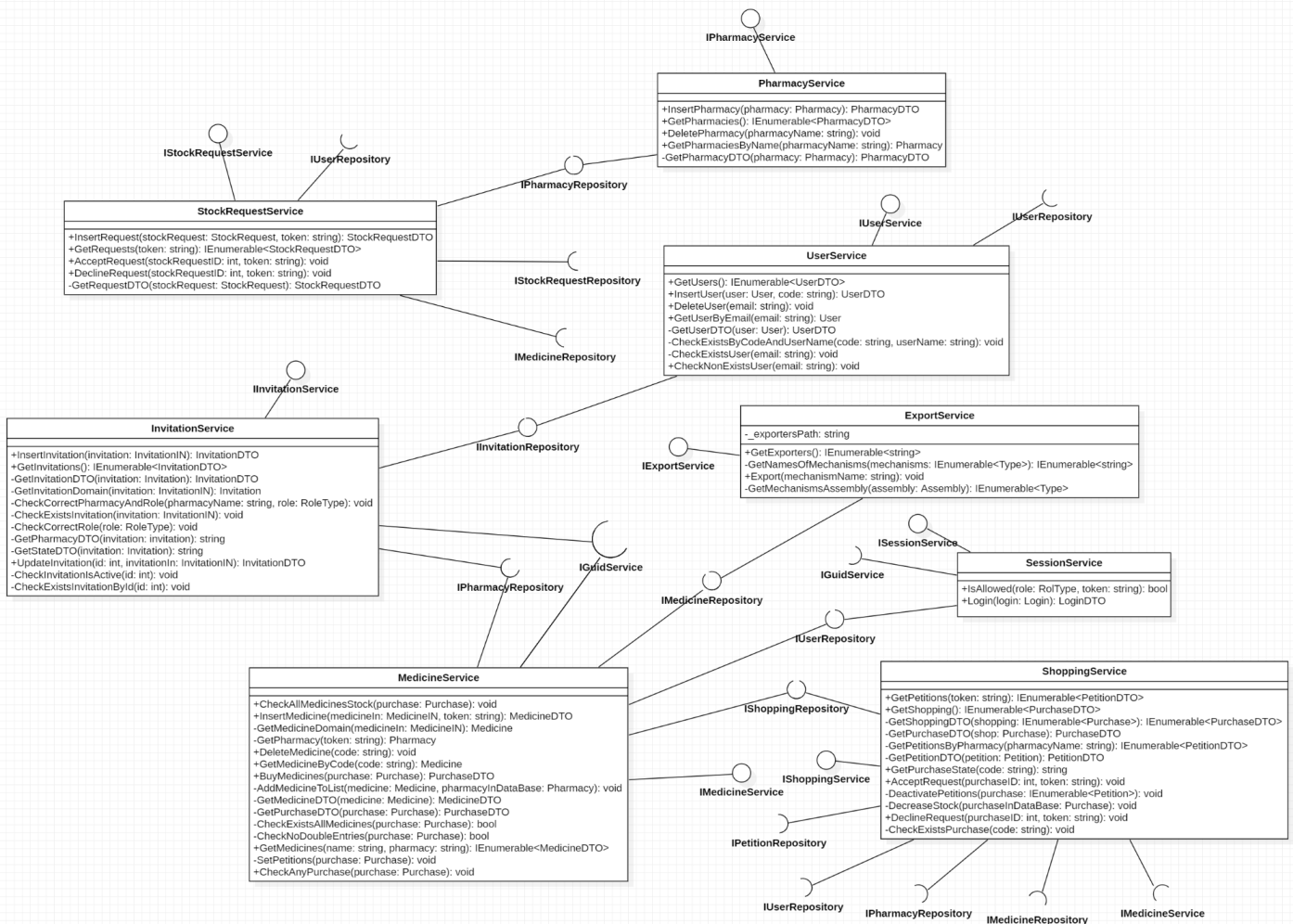
Este paquete contiene las interfaces para los métodos a ser llamados desde la Web Api a métodos de las clases de la lógica de nuestro negocio.



## PharmaCity.BusinessLogic

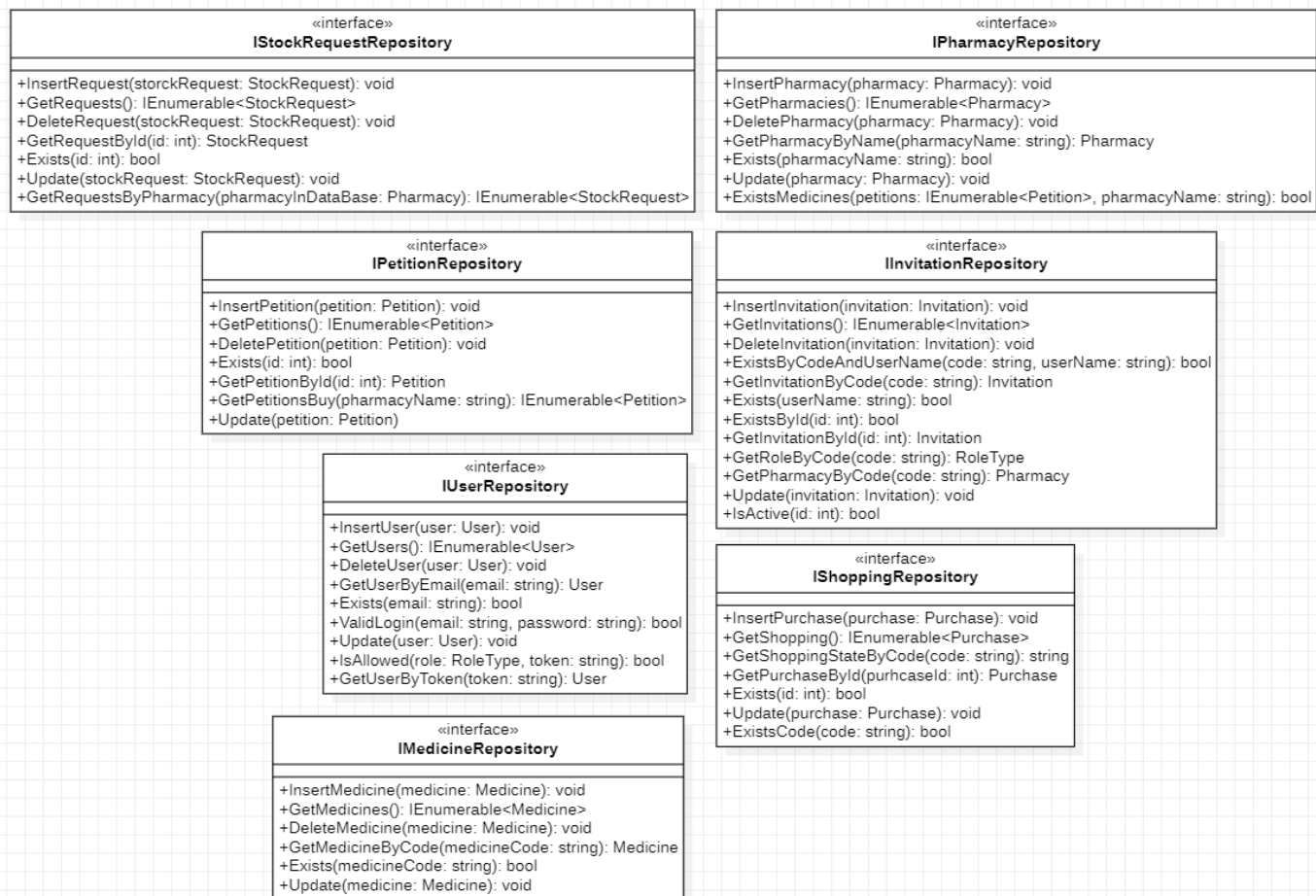
En este paquete se contienen todos los métodos que necesita el sistema para su correcto funcionamiento. Además podemos ver las interfaces requeridas y provistas de cada clase, las implementaciones de las interfaces de la IBusinessLogic.





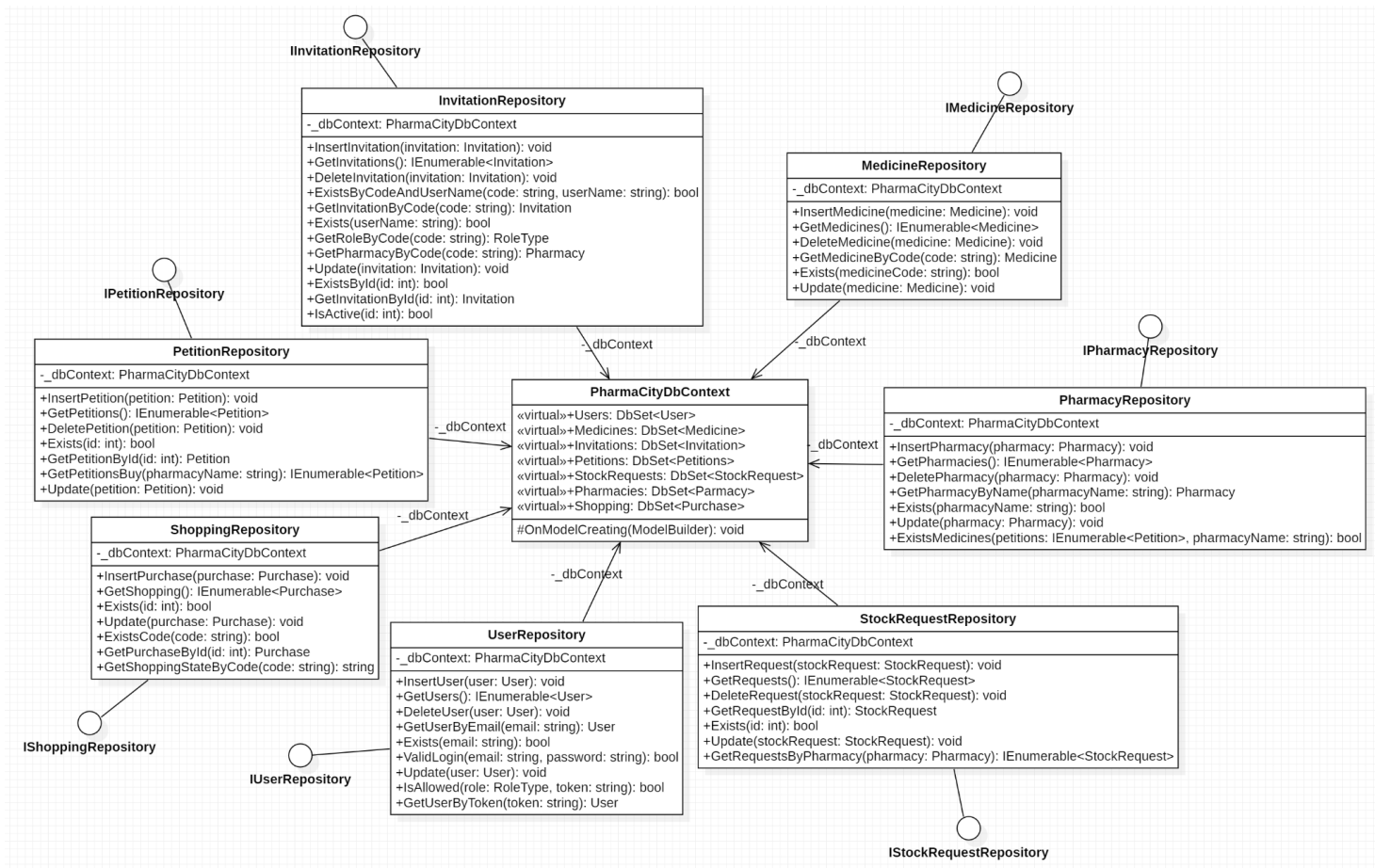
## PharmaCity.IDataAccess

Gracias a este paquete se desacopla BusinessLogic con DataAccess, por lo cual la lógica de negocios no va a estar acoplada con el paquete que se encarga de interactuar con la base de datos, haciendo que se pueda interactuar con los repositorios sin tener que conectarse directamente con el repositorio.



## PharmaCity.DataAccess

Como se ve en la imagen cada clase en este paquete implementa las interfaces provistas por cada una, las cuales podemos ver en el IDataAccess anteriormente documentado. En estas clases se hace la conexión y los cambios a la base de datos de todos los repositorios.

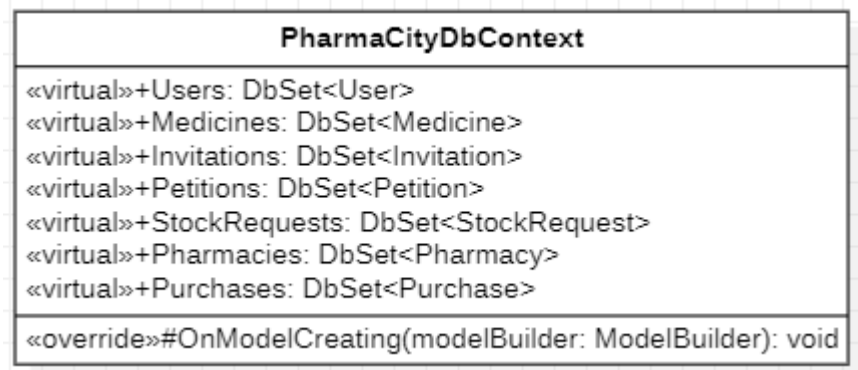


## PharmaCity.DataAccess.Migrations

Las migraciones se contienen en este paquete, siendo estas las responsables de mantener al día la base de datos. En nuestro caso mantuvimos únicamente migraciones relevantes, donde migraciones con errores fueron eliminadas.

## PharmaCity.DataAccess.Context

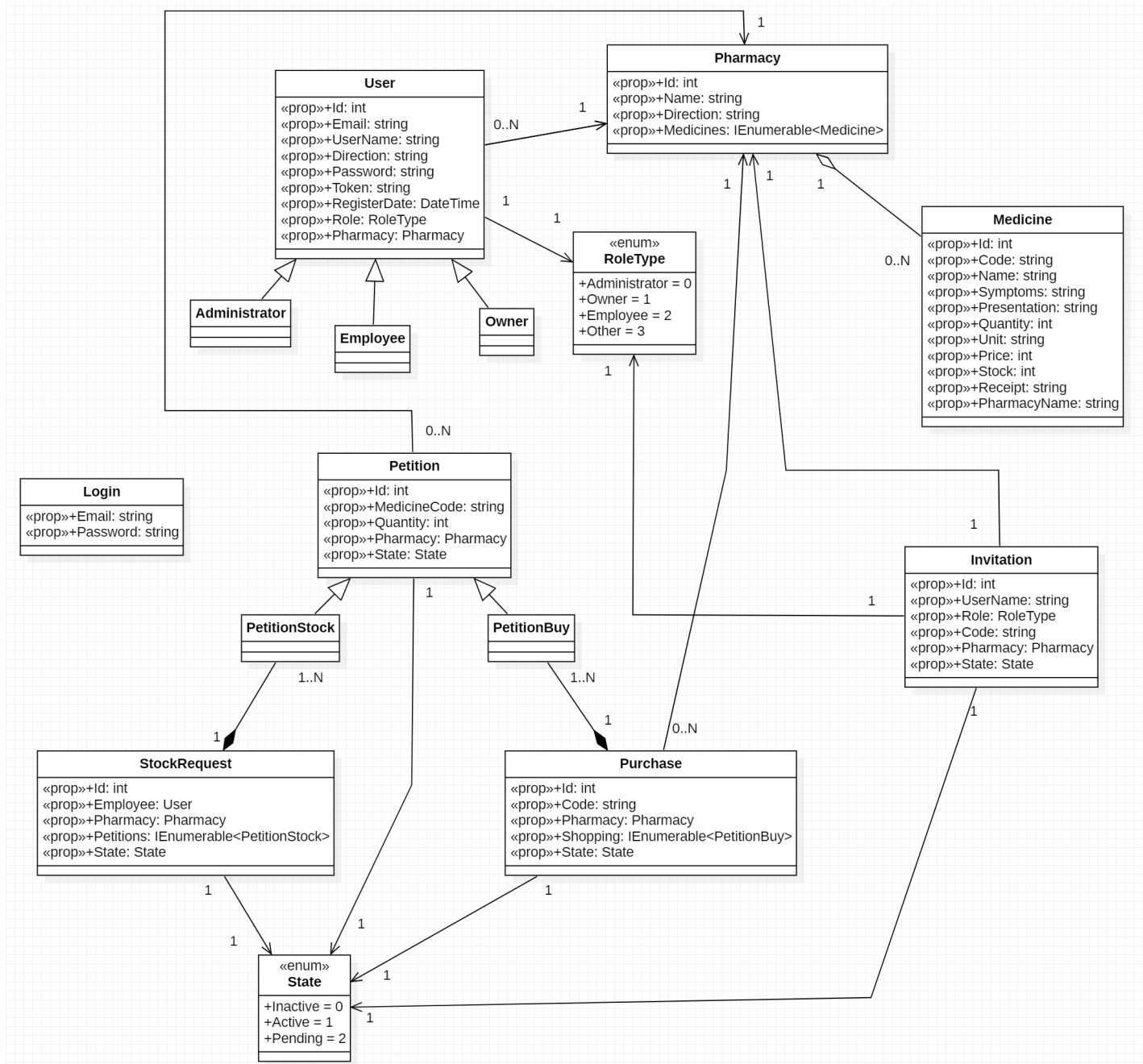
Decidimos implementar el contexto dentro de un paquete que lo separara de las demás clases, haciendo que sea más fácilmente accesible y visible en cuestiones de desarrollo.



## PharmaCity.Domain

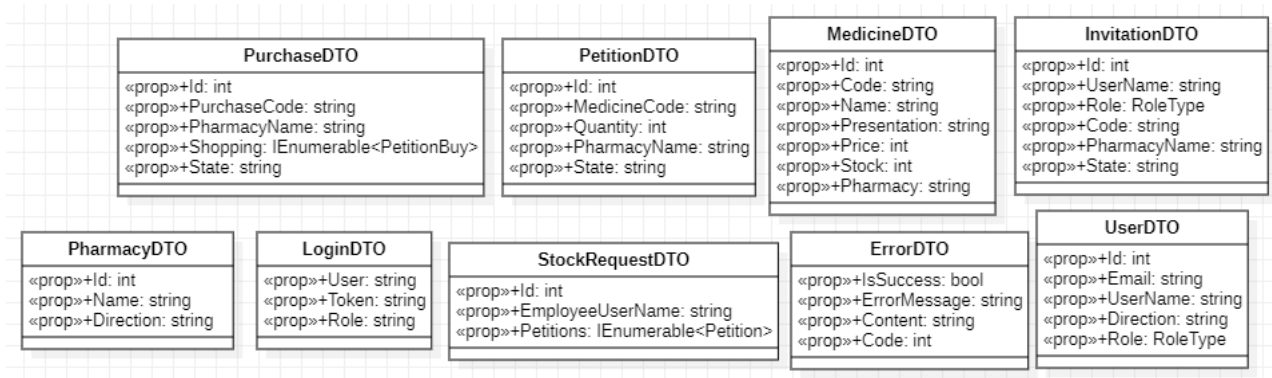
En este paquete se aprecian todas las entidades de nuestro negocio PharmaCity, quedando evidenciado cómo se relacionan entre sí, ayudando a entender de gran forma nuestro sistema ya que esta es la “base” del todo. Como vemos tenemos a los principales actores los cuales decidimos mediante un análisis extenso del sistema pedido.

Comenzamos creando la clase usuario, para la cual decidimos implementar herencia, para una posible expansión futura del sistema, con nuevos tipos de usuarios que tengan más atributos concretos de cada clase. Para estos usuarios decidimos crear una clase `RoleType`, la cual es una clase “enum” para evidenciar qué tipo de rol ocupa en el sistema y darle permisos asociados a este. Además de esto tomamos la decisión de aplicar herencia en las peticiones, ya que se necesitaban los mismos datos tanto para una petición de compra como para reposición de stock, haciendo también que se comparta una misma tabla de base de datos siendo una tabla para toda la jerarquía (TPH - Table Per Hierarchy). Implementamos una clase `Status` para controlar el estado de las reposiciones de stock y las invitaciones, así poder ver si ya se aceptaron o rechazaron tanto las reposiciones como si ya se utilizó la invitación para un usuario.



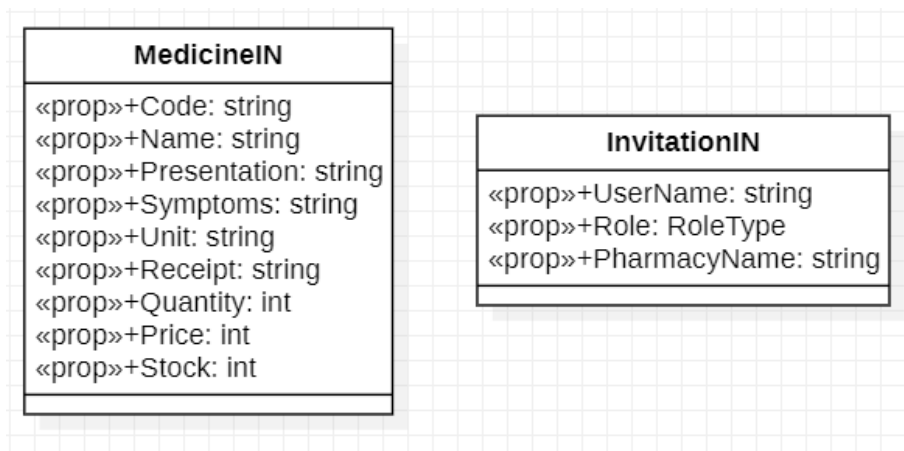
## PharmaCity.Domain.DTO

Este paquete representa los DTOs (Data Transfer Object), los cuales son objetos que actúan de intermediarios para transferir información, en nuestro caso los utilizamos como objetos de salida, ocultando información delicada que no debe ser vista por el cliente.



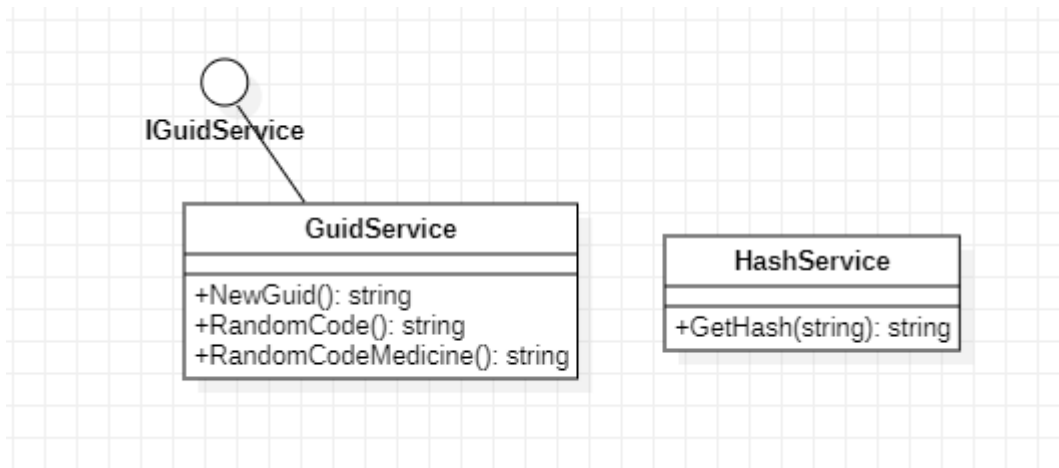
## PharmaCity.Domain.IN

Utilizamos un modelo de objeto que llamamos “IN”, para usarlo como objeto DTO (Data Transfer Object) como modelo de entrada a la lógica del negocio, ya que era necesario en este caso dado que la property “Code” era requerida si se trataba de un objeto Medicine, y este atributo es seteado en la lógica de negocio, para posteriormente crear un objeto final y completo, que sea guardado en la base de datos.



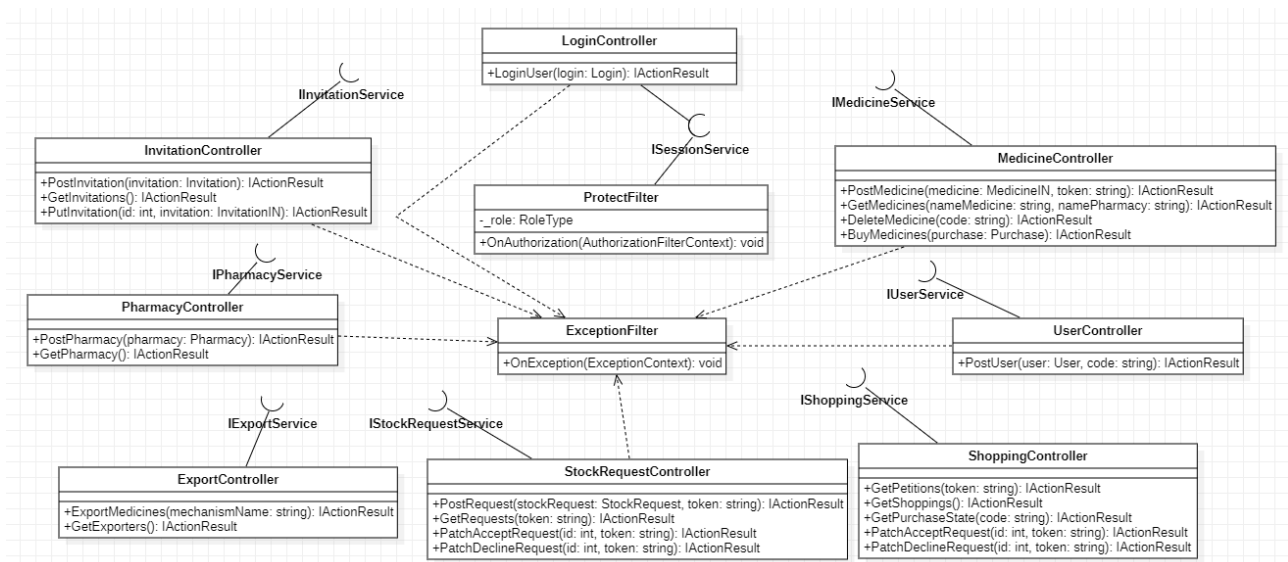
## PharmaCity.BusinessLogic.Tools

En este caso, decidimos implementar herramientas que necesitamos en el sistema, en un paquete llamado “Tools”, para separar de las demás clases de lógica de negocio. Estas clases nos ofrecen herramientas para obtener datos aleatorios como códigos o tokens y encriptar la contraseña de un usuario.



## PharmaCity.WebApi.Controllers

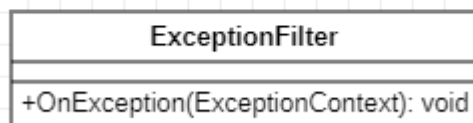
Este paquete es el encargado de obtener las consultas HTTP para llevar el llamado a BusinessLogic por medio de **IBusinessLogic** para que esta se encargue de la parte lógica de la operación, le lleva los datos que ingreso el usuario y la lógica devuelve una respuesta acorde a lo solicitado, además de esto cada una de estas clases corresponden conjuntamente a las necesidades con nuestras clases del dominio. Para llevar a cabo todo esto aquí se definen los endpoints para cada necesidad de nuestro sistema PharmaCity.



## PharmaCity.WebApi.Filters

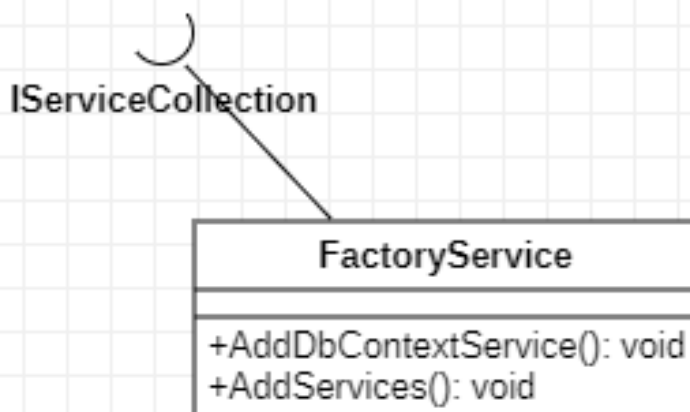
Este paquete almacena filtros que son necesarios para los Controllers, al interactuar con estos dan la posibilidad de filtrar y responder si nuestro usuario

cumple o no con el rol requerido, haciendo que no pueda acceder a funciones que no tiene acceso, esto es base al token enviado desde el header de la request, además de esto tenemos el filtro para excepciones “ExceptionHandler”, el cual nos ofrece controlar las excepciones dadas en el sistema con los códigos de error y mensajes correspondientes



## PharmaCity.Factory

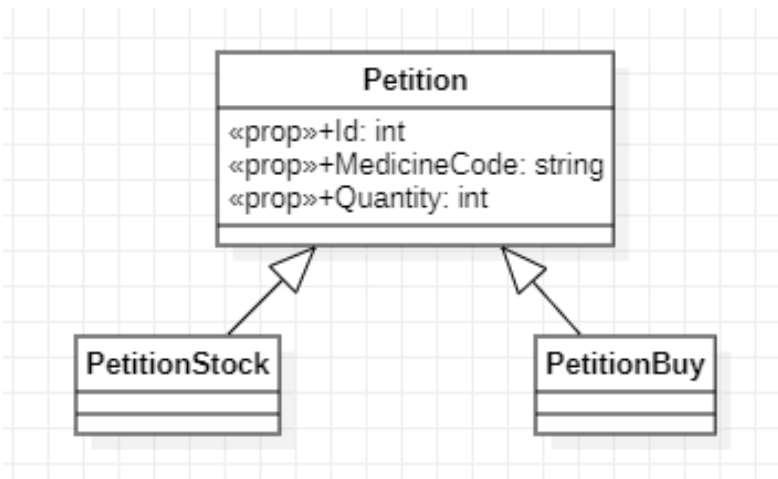
Este proyecto fue creado para bajar el acoplamiento de la WebApi y subir la cohesión haciendo que haga únicamente lo que tiene que hacer y así no romper con SRP (Single Responsibility). Se encarga de aplicar el patrón “Factory” para inyectar las dependencias de nuestro sistema, de esta forma el proyecto WebApi no estará acoplado a BusinessLogic, DataAccess y IBusinessLogic.





#### d) Jerarquías de herencia utilizadas

Tomamos la decisión de implementar una herencia, la cual pensamos que era relevante para nuestro sistema, o daban una facilidad a la hora de expandir el mismo a posibles mejoras que desee en un tiempo próximo nuestro cliente.



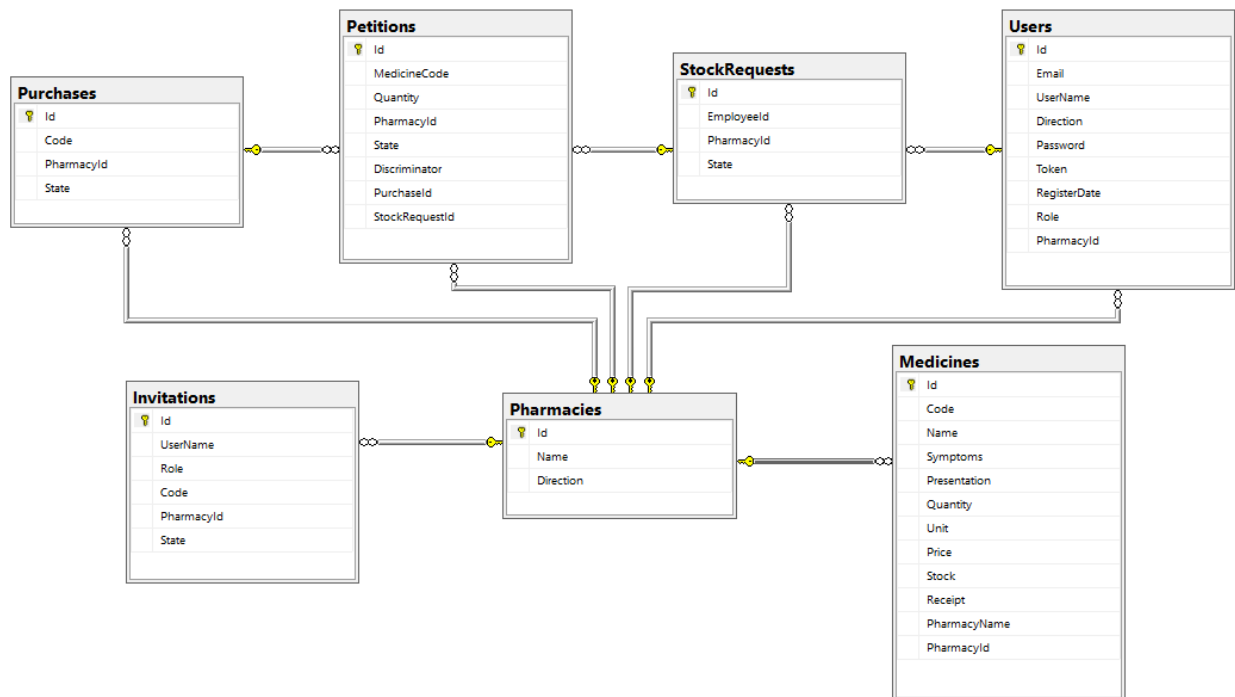
Decidimos aplicar herencia en las peticiones de nuestro sistema, ya que compartían datos con las peticiones de reposición de stock y las peticiones de compra. Esto principalmente nos ayudó para distinguir en base de datos a qué petición se refería al visualizar su estado ya que se implementó una tabla para toda la jerarquía (TPH - Table per Hierarchy). No llegamos a

aplicar polimorfismo en esta entrega como más adelante detallaremos.

Decidimos usar TPH la cual utiliza una sola tabla para toda la jerarquía, ya que nos reduce el número de tablas en la base de datos entre otras ventajas como pueden ser un mejor rendimiento en operaciones CRUD al tener todos los datos almacenados en una misma tabla.

#### e) Modelo de tablas de la base de datos

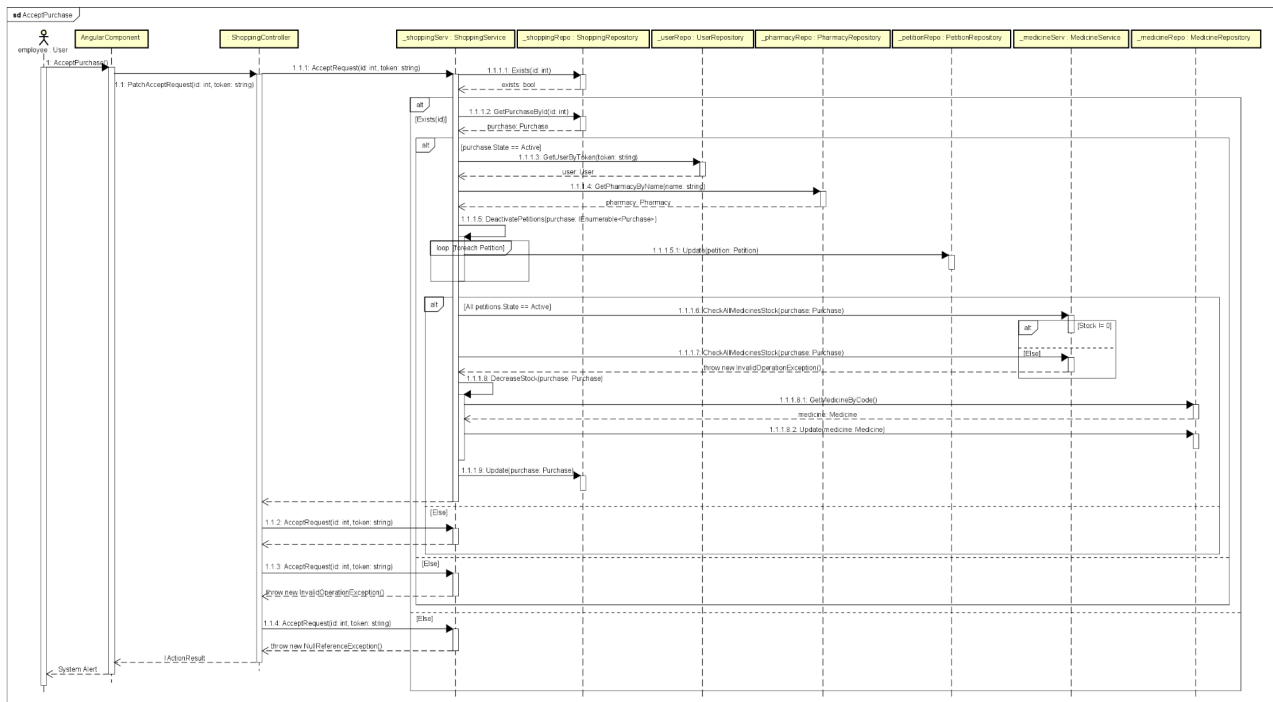
En este caso utilizamos Microsoft SQL Server Management Studio para nuestra base de datos. Además esta herramienta nos provee el modelo de tablas con las relaciones entre ellas, como se ve en la siguiente imagen:



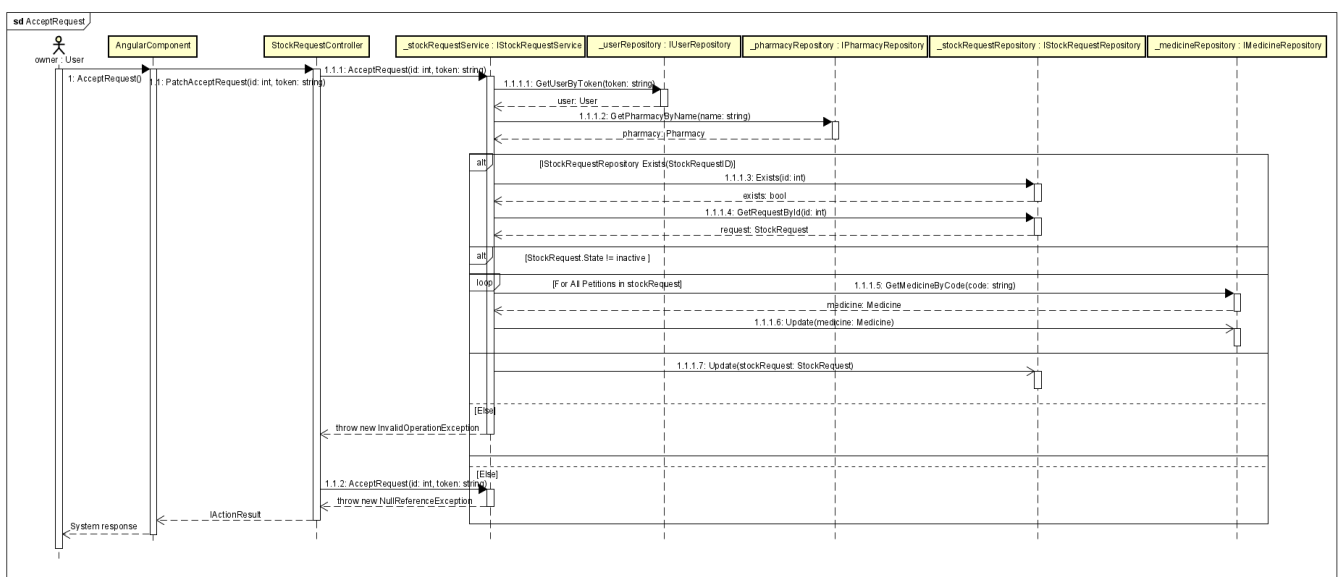
## f) Diagramas de secuencia

Decidimos realizar diagramas de secuencia con la finalidad de mostrar distintos casos de uso de nuestro sistema, con funcionalidades relevantes en este. En estos diagramas se aprecia cómo el actor comienza con una interacción con el componente web que luego pasa por el controller y termina en la lógica de negocio mediante interfaces y luego al acceso a datos, para finalmente regresar y dar una respuesta al usuario que realizó la request.

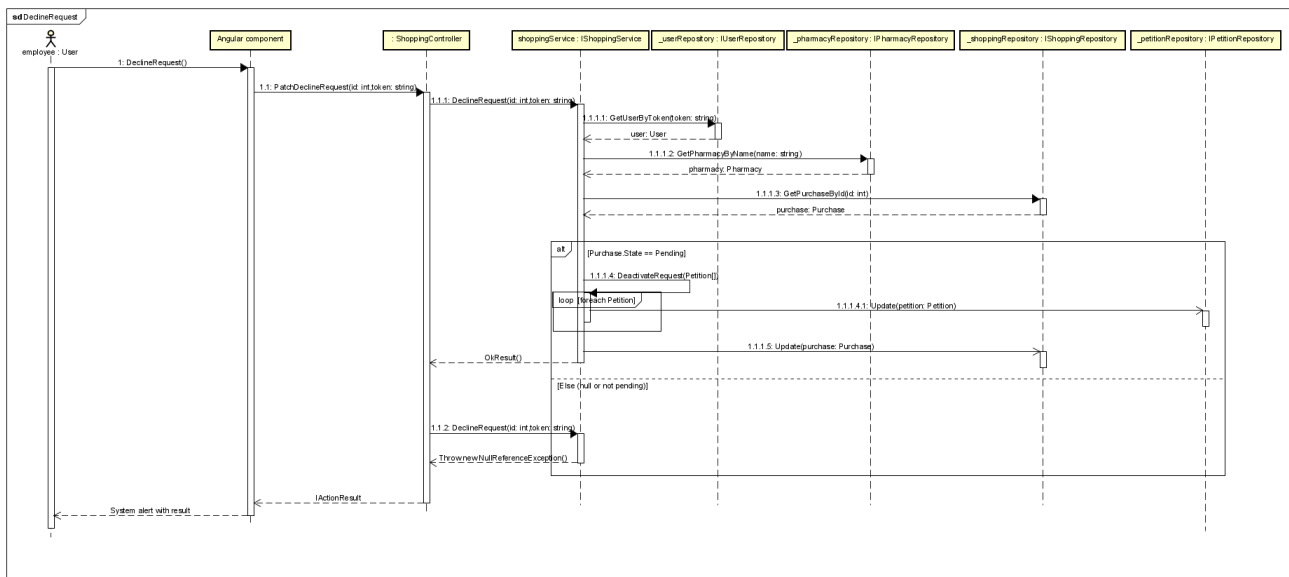
El siguiente diagrama de secuencia representa la secuencia de acciones que la aplicación en conjunto realiza para aceptar peticiones de compra. Podemos apreciar que el actor es un usuario employee, haciendo uso de la app, esté inicia la cadena de acciones donde podemos ver que el componente de angular recibe la orden y envía una llamada al controller respectivo del programa, este dispara una llamada hacia business logic y ahí es donde se da lo grueso del método por así decirlo. Para representar las pasadas malas del método, es decir cuando cae en un else y devuelve una excepción, lo representamos con llamadas desde el controller y devoluciones de excepciones, esto no es algo que suceda después del método, sino que son las posibles salidas representadas de esa manera.



La siguiente es la secuencia de request válida para aceptar una solicitud de stock para medicamentos. Esta consulta es realizada por un usuario con rol “Owner”, comienza con la llamada a la lógica de StockRequest luego solicita el token del usuario que realiza la request para así traer su respectiva farmacia del repositorio, luego hacer comprobaciones si existe la request que se desea aceptar si el estado es activo entre otras validaciones, y una vez realizadas se procede o bien a actualizar el stock de todas las medicinas solicitadas o bien responder con el respectivo error hacia el usuario.



El siguiente es un diagrama de secuencia para el declive de una solicitud de compra. En este caso comienza con una consulta Patch de html, donde se le envían los datos del usuario logueado en el sistema junto a un int con el id de la compra a rechazar, luego pasa a llamar a la interfaz de la respectiva lógica llamada en este caso IShoppingService, en donde se encomiendan llamadas al repositorio comenzando con el usuario a traer, luego otros datos que se necesitan para la operación de sus respectivos repositorios y una vez comprobado que la compra se encuentra en estado pendiente, se rechaza tanto la compra como las peticiones dentro de la misma para luego marcarla como inactiva. Termina la llamada con un IActionResult que se traduce al usuario en una alerta del sistema con la información pertinente al éxito de la operación.



## g) Justificación de diseño

Comenzando con nuestro diseño de la solución, optamos por un modelo de cuatro capas las cuales nos favorecen en el reuso de nuestro sistema, teniendo la capa de WebApi, lógica de negocios, acceso a datos y el dominio. Con este modelo especificado nos otorga muchas ventajas de las cuales entraremos en detalle.

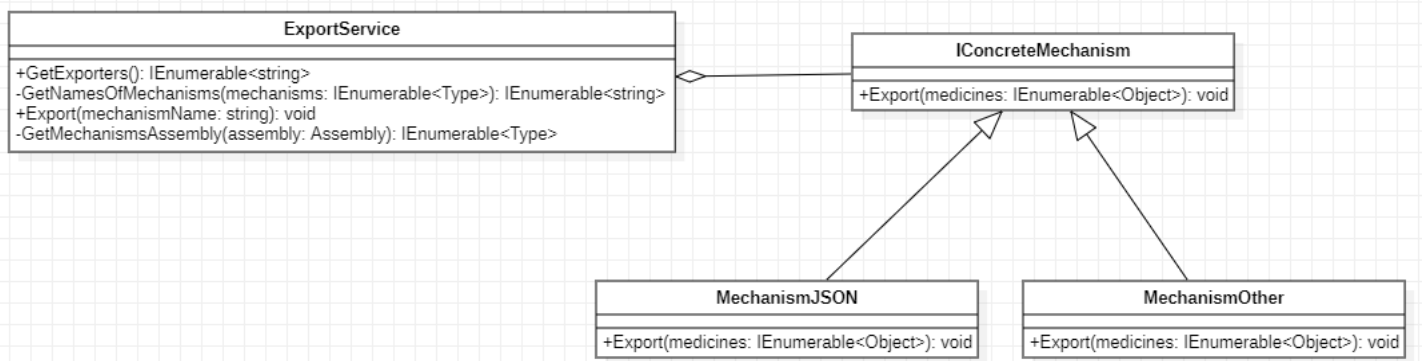
Comenzando desde la primera capa nombrada, la WebApi, en este caso uno de los principales patrones de diseño es el patrón Facade, el cual es una fachada de todo el sistema, ya que mediante estos, se proveen endpoints por los cuales se accede al sistema. Nuestro proyecto PharmaCity.WebApi implementamos el patrón Facade al implementar una fachada unificada del el conjunto de interfaces del

subsistema. Desde este paquete se interactúa con la lógica por medio de interfaces para bajar el acoplamiento.

En la implementación del requerimiento de exportar medicinas decidimos aplicar el patrón Strategy, ya que este patrón de comportamiento es el que mejor se adapta al caso, ya que define muchas maneras diferentes de aplicar un comportamiento, encapsulando cada uno y haciendo que estos sean intercambiables, así aportando a la extensibilidad de nuestro sistema pudiendo agregar nuevos mecanismos en tiempo de ejecución, una prueba más de que nuestro software está abierto a ser fácilmente extendido, agregando nuevas implementaciones sin afectar nuestro código.

Para lograr esto tenemos una interfaz “IConcreteMechanism” con los métodos que se deben aplicar por los futuros exportadores sean json, xml, entre muchos otros que podemos no conocer. Esta interfaz aplica al objeto de tipo Object, así los futuros exportadores no tienen porqué conocer nuestro dominio, concretamente no deben conocer siquiera la clase Medicine ya que trabajamos con el objeto genérico.

Para el uso del mismo en front-end se le pasa una lista de los exportadores disponibles, selecciona uno de ellos y se genera la exportación con esa estrategia.



En nuestro sistema PharmaCity se pensaron y utilizaron varios patrones, de los cuales uno de los patrones GRASP (Object-oriented design General Responsibility Assignment Software Patterns) fue el patrón experto, el cual define a quien deberíamos asignarle qué responsabilidad, siguiendo este patrón les asignamos a las que tuvieran la información necesaria para esta responsabilidad el experto en esa información, haciendo que suba la cohesión de nuestro sistema y baje el acoplamiento entre clases en conjunto otros patrones que ya detallaremos.

Notamos que Angular utiliza un patrón de diseño Observer el cual te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

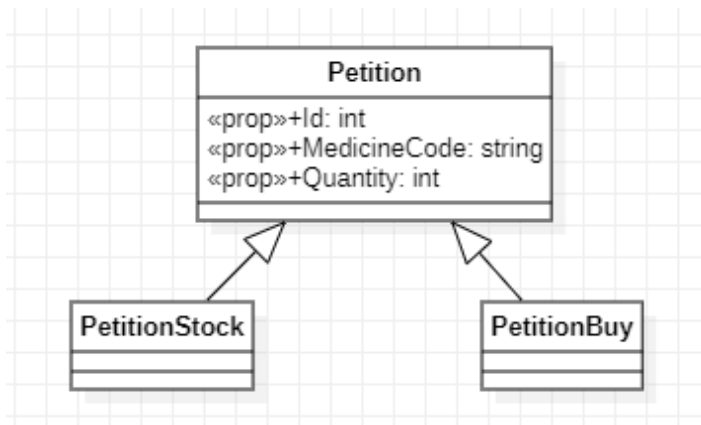
El patrón de diseño Singleton fue utilizado en nuestro front-end, concretamente en nuestros servicios, ya que cuando un componente necesita un servicio se crea una única instancia de esa clase, la cual es inyectada con inyección de dependencias y siempre se va a utilizar la misma instancia cumpliendo este patrón.

Para la reutilización de código utilizamos servicios inyectados a los componentes para así dejar todo lo que pueda llegar a variar en un lugar.

Aplicamos el patrón de variaciones protegidas al crear interfaces alrededor de módulos que puedan cambiar con el tiempo, para que posean una interfaz estable

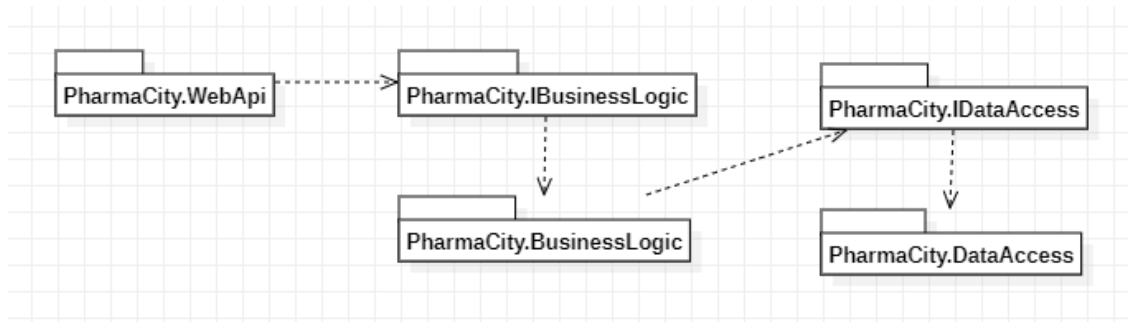
Aplicamos el patrón de indirección asignando las responsabilidades a objetos intermedios los cuales llamamos DTOs (Data Transfer Object), llevan información como intermediarios para evitar el acoplamiento.

No utilizamos polimorfismo donde utilizamos herencia, como se puede apreciar en el caso de las peticiones tanto de stock como de compra, en este punto consideramos que pudimos llegar a hacer sobre ingeniería al dejar implementada esta herencia para futuras actualizaciones que lo requieran, así se podrá extender el código sin modificarlo favoreciendo a OCP (Open–closed principle).



Además de estos utilizamos principios de diseño en nuestro código, como el principio de SRP (Single Responsibility) ya que todas nuestras clases poseen una única responsabilidad, un único motivo de cambio. Esto ayuda a una alta cohesión y un bajo acoplamiento haciendo que responsabilidades distintas no se tengan que acoplar entre sí.

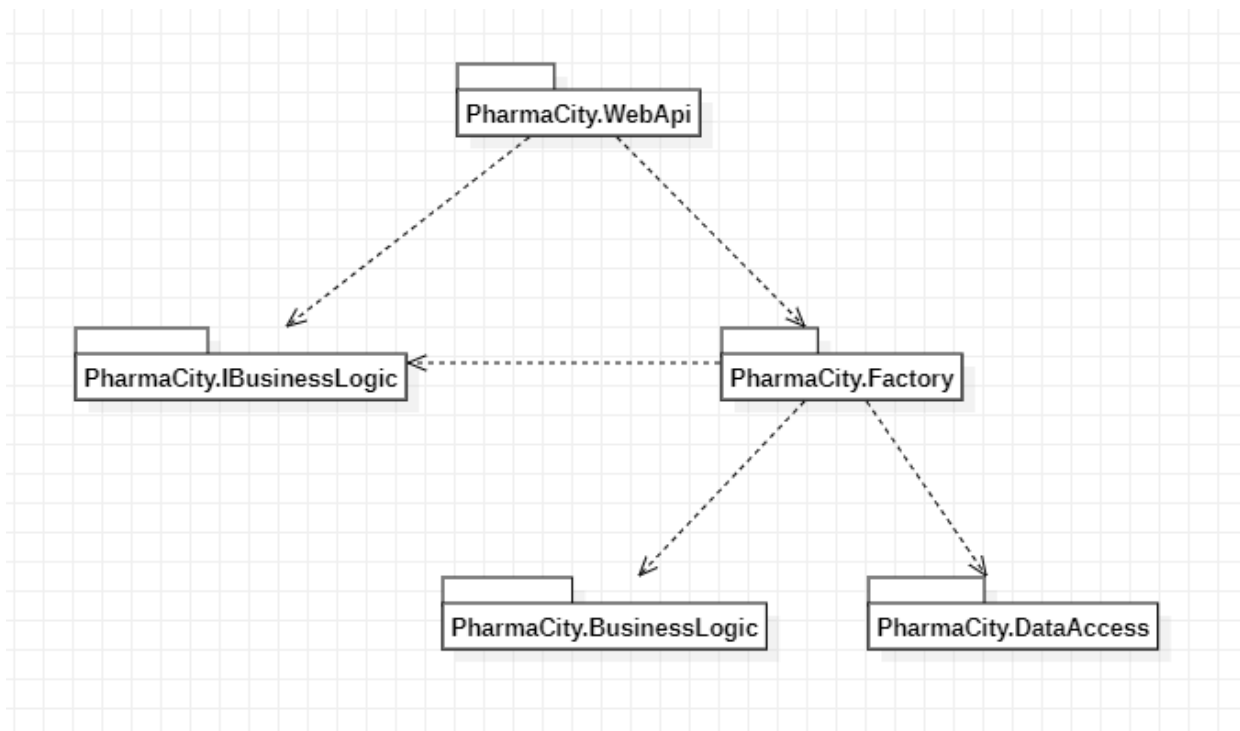
Nuestra solución también cumple con el principio SOLID de ISP (Interface segregation principle) al tener muchas interfaces específicas en vez de una genérica. Interfaces específicas que solo tienen los métodos utilizados, ya que si se tienen muchos métodos puede generar un mayor acoplamiento y dificultar la mantenibilidad del código, por lo cual decidimos crear interfaces específicas aportando positivamente a nuestro código y su mantenibilidad.



Utilizamos inyección de dependencias en nuestro sistema para evitar el acoplamiento a la lógica de negocio y el dominio. Gracias a la inyección de dependencias obtuvimos ventajas en nuestro código al dejarlo más limpio, desacoplando las capas y siendo más fácil de modificar, además de testear mucho más fácil nuestra solución.

En este caso la inyección de dependencias se utiliza con el método AddScoped brindado por .NET Core, en nuestro caso, estos métodos los implementamos en un proyecto separado llamado “PharmaCity.Factory” la cual tiene la única responsabilidad de agregar estas inyecciones de dependencia por lo que favorece al cumplir una única responsabilidad (SRP), y desacoplando la capa WebApi con BusinessLogic, IBusinessLogic y DataAccess. Además implementando como su nombre lo indica el patrón de diseño creacional Factory.

Al utilizar inyecciones de dependencia también aportamos al principio de abierto cerrado (OCP) ya que el sistema va a estar abierto a la extensión y cerrado a la modificación al tener que cada módulo solo usa sus dependencias y no las inicializa, favoreciendo a la extensión. Además implementamos con esto el patrón GRASP de fabricación pura, al tener una clase inventada “FactoryService” para bajar el acoplamiento, aumentar la cohesión y potenciar la reutilización del código.



## h) Métricas

Para el correcto análisis de este punto utilizamos la herramienta NDepend, la cual provee una extensión para .NET, aportando los datos necesarios para analizar de forma correcta. Esta herramienta nos brinda diversos resultados los cuales vamos a conocer a profundidad a continuación.

Analizaremos la abstracción e inestabilidad de nuestros proyectos, comenzando por **BusinessLogic**, este proyecto no posee interfaces ni clases abstractas dentro, por lo cual podremos decir que es un paquete concreto. Es un 99% inestable ya que no hay clases que dependen de este proyecto. Gracias a todo esto se posiciona en la secuencia principal por ser muy inestable y tener muy baja abstracción, con esto decimos también que tiene una distancia baja. Estos resultados indican que el paquete va a ser fácil de extender.

```

Project:PharmaCity.BusinessLogic
Abstractness : 0
Instability  : 0.98
Dist        : 0.01
  
```

Lo mismo que detallamos arriba ocurre con **DataAccess** el cual es un proyecto inestable y concreto, se posiciona sobre la

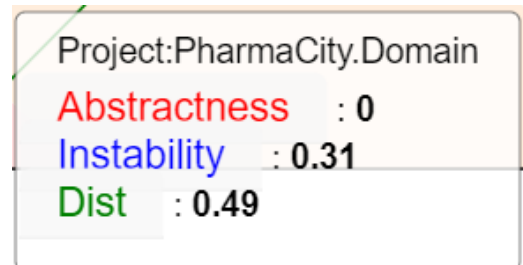
```

Project:PharmaCity.DataAccess
Abstractness : 0
Instability  : 0.96
Dist        : 0.03
  
```

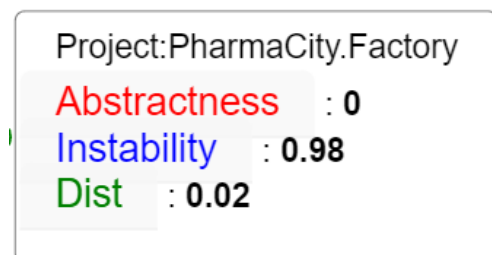


secuencia principal teniendo una distancia baja a esta, este proyecto no tiene clases dependiendo de él por lo cual es un paquete fácil de extender, ni tampoco tiene interfaces.

En el caso de **Domain** también es un proyecto concreto al no contar con clases abstractas ni interfaces. Al tener muchas clases dependiendo de este paquete es muy estable, teniendo una distancia a la secuencia principal de 0.5 y al encontrarse en la “Zona de Dolor” lo hace un paquete difícil de cambiar, ya que afectará a muchos paquetes al hacerlo. Analizando esto no lo tomamos realmente como un gran problema, ya que este proyecto se define al comenzar nuestra solución global por lo cual si el mismo es consistente no habría problemas, y no es tan común que requiera cambios en general. Nuestra lógica de negocios está basada en este paquete.



Project:PharmaCity.Domain	
Abstractness	: 0
Instability	: 0.31
Dist	: 0.49



Project:PharmaCity.Factory	
Abstractness	: 0
Instability	: 0.98
Dist	: 0.02

El proyecto **Factory** es un paquete concreto al no tener abstracciones, este paquete depende de muchas clases y pocas dependen de este lo que lo hace inestable, de la mano con esto es más fácil de extender al no afectar mucho a otros paquetes. Como vemos en los resultados está sobre la secuencia principal por lo cual tiene una distancia muy baja.

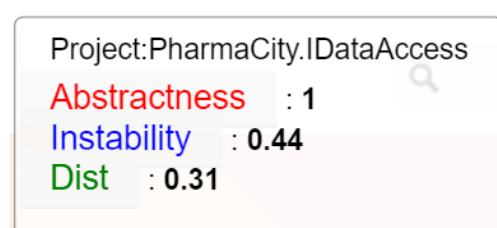
En el caso de **IBusinessLogic** nos encontramos con un proyecto 100% abstracto al tener en su interior únicamente interfaces.

Es 0.54% estable lo cual lo hace bastante estable dado que varios otros paquetes dependen de él, al darse esta situación es un paquete que no será fácil de cambiar. Se encuentra a una distancia de 0.38 de la secuencia principal, no estando tan lejos de esta lo favorece para no caer en la “Zona de Inutilidad”.



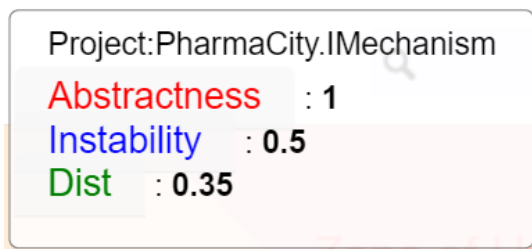
Project:PharmaCity.IBusinessLogic	
Abstractness	: 1
Instability	: 0.46
Dist	: 0.32

Al igual que en el paquete anterior, en este **IDataAccess** también es un proyecto 100% abstracto, teniendo una inestabilidad baja lo cual lo acerca más a la secuencia principal



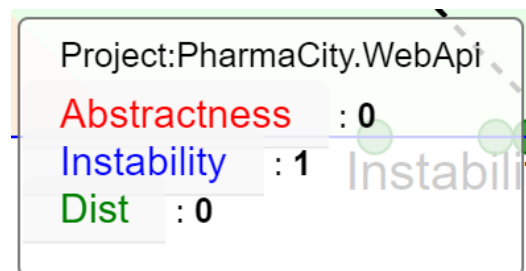
Project:PharmaCity.IDataAccess	
Abstractness	: 1
Instability	: 0.44
Dist	: 0.31

estando a 0.32. Es un proyecto abstracto ya que todas sus clases son interfaces, es algo estable ya que varios paquetes dependen de él por lo cual lo hace un paquete difícil de cambiar afectando a varios que dependen de él, y este depende de muy pocos (únicamente Domain). Este paquete aunque sea totalmente abstracto no está cerca de la “Zona de Inutilidad”.

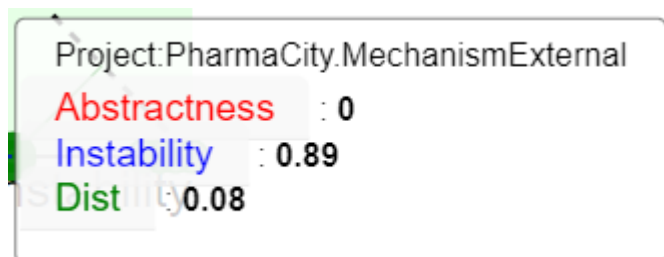


Al igual que los dos anteriores **IMechanism** es un proyecto totalmente abstracto, teniendo en su interior únicamente interfaces, tiene una inestabilidad baja al no depender de nadie, y dependen de él BusinessLogic y MechanismExternal. Es un proyecto medianamente estable y abstracto por lo que es bueno.

A diferencia del anterior el proyecto **WebApi** es totalmente inestable, y concreto ya que no posee ninguna abstracción (interfaz o clase abstracta) en su interior, gracias a esto lo convierte en un paquete muy fácil de extender ya que nadie depende de él. Se ubica sobre la secuencia principal con una distancia 0, esto reafirma que es bueno tener un paquete inestable que sea concreto.



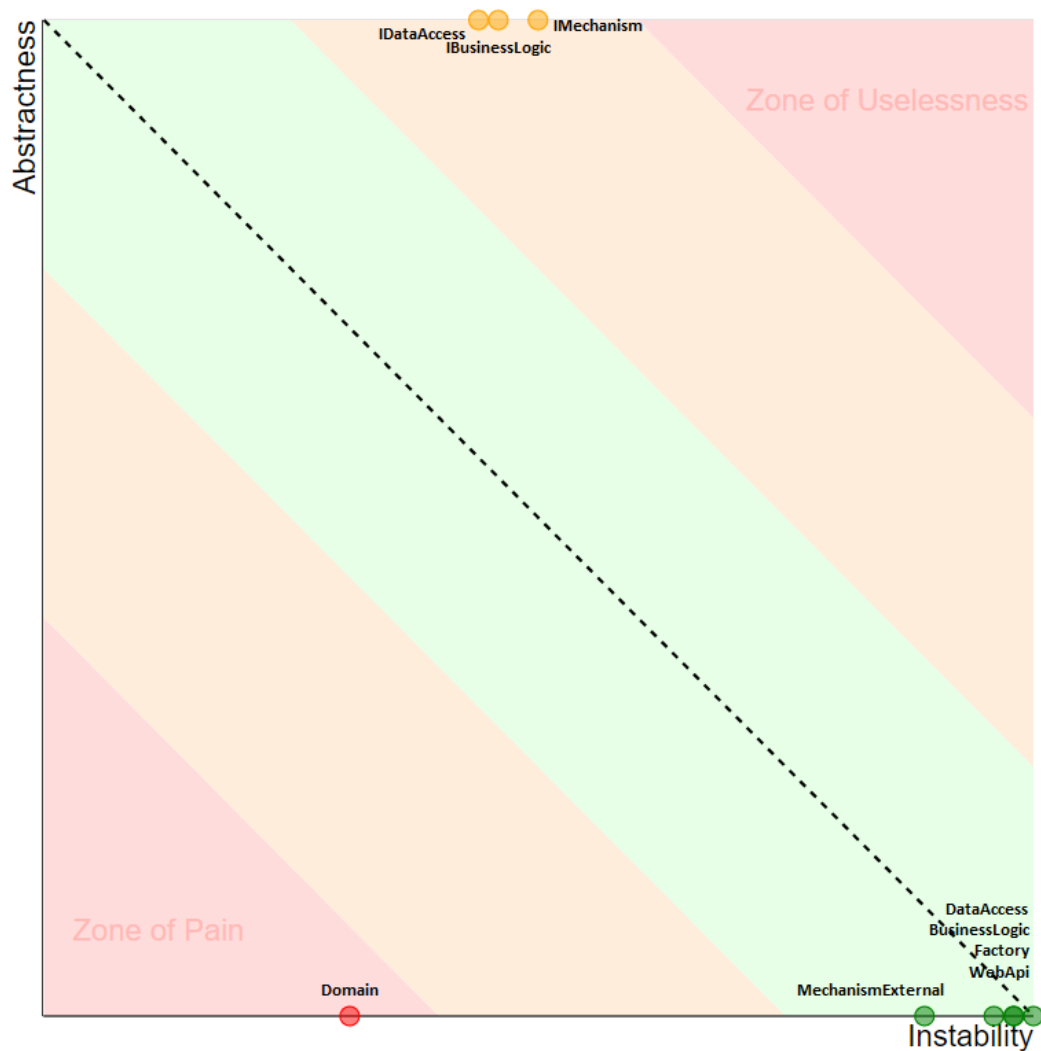
En el caso de **MechanismExternal** el cual es el proyecto que contiene los mecanismos de exportación, es un proyecto que no contiene abstracciones y es altamente inestable ya que depende de muchos y pocos dependen de él, esto hace que este proyecto sea inestable y concreto, estando a una distancia muy corta de la secuencia principal, dado todo esto es fácil de extender.



En resumen de todo estos datos de nuestro sistema se entiende que aquellos paquetes estables que son difíciles de modificar o extender siempre son paquetes que no difícilmente cambien como lo es el dominio o las interfaces implementadas para realizar inversión de dependencia, y en casos como WebApi, BusinessLogic y DataAccess que pocos paquetes dependen de estos, van a ser fáciles de extender

y flexibles al cambio por lo cual consideramos una ventaja en el diseño de nuestro sistema, ya que lo hace más mantenible y extensible en donde esto sea necesario.

Estos datos fueron brindados por la gráfica de abstracción vs inestabilidad la cual es autogenerada por NDepend:



### Valores de los Assemblies del sistema

Assemblies	<u>Relational Cohesion</u>	<u>Instability</u>	<u>Abstractness</u>	<u>Distance</u>
PharmaCity.Domain v1.0.0.0	1.06	0.31	0	0.49
PharmaCity.IDataAccess v1.0.0.0	0.17	0.44	1	0.31
PharmaCity.DataAccess v1.0.0.0	1	0.96	0	0.03
PharmaCity.IBusinessLogic v1.0.0.0	0.11	0.46	1	0.32
PharmaCity.IMechanism v1.0.0.0	1	0.5	1	0.35
PharmaCity.BusinessLogic v1.0.0.0	0.3	0.98	0	0.01
PharmaCity.MechanismExternal v1.0.0.0	1	0.89	0	0.08
PharmaCity.Factory v1.0.0.0	1	0.98	0	0.02
PharmaCity.WebApi v1.0.0.0	0.83	1	0	0

La cohesión relacional la obtuvimos de NDepend como se puede apreciar en la imagen anterior. En este punto de análisis, las clases dentro de un paquete deberían estar fuertemente relacionadas, dando una cohesión del paquete alto. Se estima que un buen valor de cohesión relacional está entre 1.5 y 4. En nuestro caso podemos observar que la gran mayoría tienen un valor cercano a 1 y otros con mucho menos de este valor. Estos valores se deben al modelo de 4 capas que manejamos en nuestro sistema, teniendo WebApi, BusinessLogic, DataAcces y Domain las clases dentro de estos proyectos no van a estar tan fuerte relacionadas entre ellas, sino que están relacionadas fuertemente con las otras capas. Este modelo empleado por nuestro sistema si bien favorece la mantenibilidad está mucho más enfocado en el reuso. Para que estos resultados se acerquen más al resultado esperado, debemos cambiar el modelo de 4 capas para así tener más relaciones entre las clases de los paquetes.

En los casos que se dan valores más cercanos al 0, se puede argumentar que son mayoritariamente proyectos que únicamente poseen interfaces por lo cual la

cohesión dentro del paquete es baja. Por parte del assembly BusinessLogic es un proyecto el cual sus clases no se relacionan entre ellas ya que tienen una alta cohesión en las clases en específico, lo cual no tienen porque acoplarse entre ellas para generar los servicios.

Una vez detallados todos los resultados obtenidos podemos pasar a evaluar los principios de diseño a nivel de paquetes, comenzando por los principios a nivel de cohesión:

El principio de Clausura Común (**CCP**) el cual establece que las clases de un mismo paquete deben cambiar por el mismo tipo de cambio, dado a que si un cambio afecta a un paquete este afecta a todas las clases del mismo y a ningún otro paquete. En base a esto pensamos que si argumentamos cual es el concepto del cambio que se hará en el sistema, lo podemos estar cumpliendo. Por ejemplo al tener un modelo de 4 capas que se comunican entre paquetes y no está todo lo que cambia junto no se puede estar cumpliendo, pero si se da un cambio a nivel de base de datos, esto afectará únicamente a las clases dentro del paquete que se centran en base de datos, cumpliendo el principio. Si bien no creemos que es la solución que más favorece a la mantenibilidad en un sistema que está evolucionando, en nuestro sistema que ya está maduro creemos que tenemos que favorecer al reuso en referencia a la tensión entre principios.

En el principio de Reuso Común (**CRP**), donde se establece que las clases que pertenecen a un paquete se rehúsan juntas, si se rehúsan clases del paquete, entonces se rehúsan todas. Dicho principio lo cumplimos fuertemente en nuestro sistema, ya que al aplicar un modelo de 4 capas, introducimos en cada paquete las clases que se rehúsan juntas, ya que se comunica en su totalidad con otro paquete como por ejemplo la lógica de negocio utiliza todas las clases del acceso a datos, sucediendo lo mismo para todo el sistema, esto favorece fuertemente al reuso, que es lo buscado debido a la madurez de nuestra solución.

Ahora pasamos a los principios a nivel de acoplamiento:

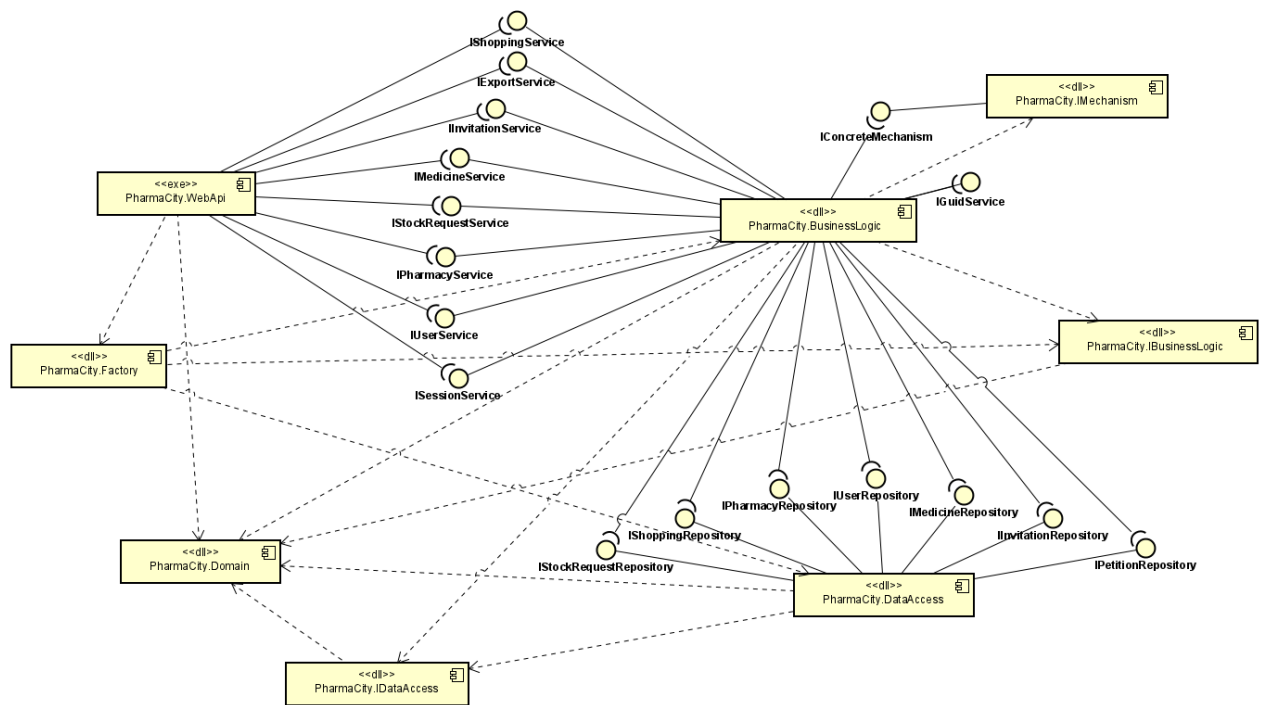
En el punto del principio de Dependencias Estables (**SDP**), el cual establece que las dependencias entre los paquetes se debe dar del menos estable al más estable y no depender hacia uno más inestable. En base a esta definición, nuestro sistema cumple totalmente este principio, ya que todas las dependencias van de un paquete más estable a otro menos estable y nunca se da lo contrario. En nuestro caso esto se cumple por el uso constante de interfaces, apoyados en el principio de inversión de dependencias ya detallado anteriormente, para desacoplar de

forma correcta los paquetes. Como ejemplos del correcto cumplimiento tenemos a WebApi con una inestabilidad de 1 acoplándose a Domain, Factory y IBusinessLogic, con inestabilidades del 0.31, 0.98, 0.46 respectivamente. Como también BusinessLogic con 0.98 acoplándose a Domain, IDataAccess y IMechanism, con 0.31, 0.44 y 0.5 respectivamente. Gracias a este correcto cumplimiento hace que nuestro sistema sea muy mantenible.

Por último también cumplimos con el principio de Abstracciones Estables (**SAP**), la que establece que los paquetes deben ser tan estables como abstractos. Gracias a los valores anteriormente brindados podemos decir que la forma de implementar nuestra solución siempre llevó a que aquellos paquetes que tienen un nivel de inestabilidad más alta sean concretos, y los que son más estables son abstractos, todo esto gracias a la implementación de interfaces para que no se acoplen entre paquetes directamente y el uso como anteriormente detallamos de inversión de dependencias. Para el caso de Domain como vimos en los valores es una excepción de esto, estando en la “Zona de Dolor”, a pesar de esto apelamos a que lo estamos cumpliendo el principio, ya que en el nivel de madurez que se encuentra nuestra solución, no se darán cambios normalmente en este paquete, como anteriormente lo discutimos, no suele cambiar si se posee un diseño acertado.

## i) Diagrama de componentes

Como se ve en la imagen, WebApi requiere las interfaces de la BusinessLogic y esta lógica requiere las interfaces de DataAccess la cual implementa las interfaces del repositorio. Factory se utiliza para desacoplar a la WebApi gracias a que se encarga de la inyección de dependencias como vimos anteriormente, requiere las interfaces de BusinessLogic y las de DataAccess, desacoplando a la WebApi de ellas.



## j) Cambios realizados

En base a los nuevos requerimientos, se tuvieron que realizar cambios a nuestro sistema. En estos se agregó la funcionalidad de poder editar los datos de una invitación creada y no utilizada, para ello se tuvo que agregar nuevas uris que contemplen esta actualización de las invitaciones.

Ahora, al momento de hacer la compra, se podrá comprar medicamentos de varias farmacias en una sola compra, y esta no descontará el stock en el momento, sino que debe hacerse la solicitud de esta y posteriormente aceptar las compras por parte de un empleado de cada farmacia involucrada. Para ello y otras nuevas funcionalidades se tuvo que crear un nuevo estado en la clase enum State, en la cual le agregamos el estado "Pending" para controlar este estado.

Agregamos la interfaz IShoppingService para manejar las compras, pudiendo aceptarlas o denegarlas, esto en base a los nuevos requerimientos brindados por el cliente.

Además implementamos también la interfaz IExportService para la exportación y obtener los diferentes tipos de exportadores disponibles que son capaces de ser

cargados en tiempo de ejecución. Para esto agregamos el proyecto de “PharmaCity.IMechanism” que contiene la interfaz de los futuros exportadores y también el proyecto “PharmaCity.MechanismExternal” el cual contiene el o los mecanismos de exportación, por ejemplo de JSON, pudiendo agregar más en tiempo de ejecución. Para que todo esto sea consumible en nuestra api agregamos los controladores para la exportación de medicamentos, con sus respectivas uris.

Cambiamos uris corrigiendo errores al tener datos pasados por los parámetros que debían ser enviados por un body, este es el caso del post de una invitación, el post de iniciar sesión, entre otros que ahora se encuentran corregidos.

## k) Mecanismos de extensibilidad

En base al requerimiento de nuestro sistema de tener la posibilidad de exportar medicamentos del sistema en diferentes formatos como JSON,XML, CSV, entre otros que podemos no conocer. Aplicamos el patrón de diseño Strategy pudiendo variar de estrategia como anteriormente lo mencionamos en detalle. Todo esto nos permite otorgarle al cliente la posibilidad de implementar en el futuro diferentes mecanismos de explotación para generarlas en el formato que se desee, haciendo altamente extensible nuestro sistema.

Con este fin decidimos implementar este requerimiento de la siguiente forma: Desde nuestro frontend se tiene una página específica para la exportación, donde en un input select se despliegan los diferentes mecanismos existentes en el sistema y con un botón para actualizar si en tiempo de ejecución se agrego alguno nuevo, luego cuando elegimos el mecanismo y le damos a exportar se generará un archivo en la carpeta “PharmaCity.WebApi” con las medicinas existentes en nuestra base de datos y en el formato del exportador seleccionado.

Una vez que se genera el archivo queremos aclarar los atributos que contendrán estos mismos y sus tipos. Cada “Medicine” posee un Id:string, Code:string, Name:string, Symptoms:string, Presentation:string, Quantity:int, Unit:string, Price:int, Stock:int, Receipt:string, PharmacyName:string.

Para hacer posible agregar nuevos mecanismos en tiempo de ejecución utilizamos Reflection que por definición “... es la habilidad de un programa de autoexaminarse con el objetivo de encontrar ensamblados (.dll), módulos, o información de tipos en



tiempo de ejecución. En otras palabras, a nivel de código vamos a tener clases y objetos, que nos van a permitir referenciar a ensamblados, y a los tipos que se encuentran contenidos”, lo utilizamos para encontrar en nuestro sistema el ensamblado donde se encuentran contenidos los tipos de exportación que son aplicables a la interfaz “IConcreteMechanism”, extrayendo de estos el que tenga el nombre solicitado por la llamada del frontend. De todo esto se encarga el servicio “ExportService” el cual una vez que define los exportadores disponibles dispara la llamada Export mediante la interfaz “IConcreteMechanism” al mecanismo especificado.

Cuando se desee extender a un nuevo mecanismo, únicamente se debe agregar el mismo al assembly .dll “PharmaCity.MechanismExternal” implementando la interfaz “IConcreteMechanism” con el método “Export”.

En el caso de los sistemas de exportación y la interfaz, hicimos que no dependan de Domain, en el caso de especificarle qué tipo de objeto será el que se exporta, la implementación es con un objeto global de tipo “Object” para así los nuevos mecanismos no tengan que conocer el dominio para poder ser implementados.

## I) Descripción del manejo de excepciones

Decidimos manejar las excepciones con las que nos provee .NET Core, ya que son específicas para situaciones concretas, haciendo que sean descriptivas para las situaciones de excepciones de la solución. Además estas excepciones fueron más que suficientemente variadas como para controlarlas con el filtro de excepciones implementado en los controllers el cual devuelve en un DTO el mensaje de error junto a un status code para la respuesta de la WebApi.

**NOTA:** Nos pasamos de las 25 páginas ya que incluimos imágenes que creemos son importantes que se visualicen en el momento con la información y no estar en busca de ellas en el anexo.

## ANEXO:

### Evidencia del diseño y especificación de la API

#### a) Discusión de los criterios seguidos para asegurar que la API cumple con los criterios REST

En nuestra WebApi utilizamos los verbos Get, Post, Put, Delete y Patch para todo el sistema implementado, asegurándonos que cumpla con la idempotencia correspondiente a cada verbo. Pensamos en buenos endpoints que no den lugar a confusiones y permitan hacer las request correspondientes sin mayores complicaciones, de la mano de esto utilizamos las query para implementar los filtros y no pasamos los 3 niveles en los endpoint.

Ejemplo donde se ve el filtrado por nombre de medicamento y farmacia desde la Query.

```
[HttpGet]
0 referencias
public IActionResult GetMedicines([FromQuery] string nameMedicine, [FromQuery] string namePharmacy)
{
    return Ok(_medicineService.GetMedicines(nameMedicine, namePharmacy));
}
```

Manejamos códigos de error correspondientes a la situación y mensajes que sean entendibles para el usuario, utilizando códigos de la familia del 200 para las request que den un resultado correcto, 400 para las request que tengan un error del lado del cliente y 500 para los errores del sistema/servicios

Nos aseguramos que nuestra Api sea Stateless y que cada request sea independiente de la siguiente, ya que toda la información necesaria para una acción se encuentra en una única request. Esto se puede ver claramente en la autenticación del usuario, ya que para que las request que requieran ingresar al sistema, estas se deben efectuar enviando un token en el header para poder comprobar que se inició sesión y tiene permisos para esta acción.

Separamos correctamente nuestra Api en capas, donde llegan las request HTTP a la WebApi esta pasa la información a la lógica de negocio y desde la lógica al acceso a los datos de la base de datos y allí la correcta obtención de los datos o escritura de ellos.

Se utilizaron objetos DTOs (Data Transfer Object) para mostrar información específica al usuario y no mostrar información sensible a este, como puede ser la contraseña de un usuario, en cambio se crea un objeto simplificado de usuario con

los datos que sí se desean mostrar. Además creamos objetos DTOs pero de entrada para las ocasiones que así los requieran.

#### b) Descripción del mecanismo de autenticación de requests.

Para el caso de la autenticación de los usuarios se optó por la implementación de tokens a las requests. Tokens que son otorgados al usuario una vez inicie sesión, brindando en la respectiva request de login su correo y contraseña, si estos datos son correctos y correspondientes a un usuario registrado en el sistema, se genera un token de tipo string, se le asigna ese token a un atributo del usuario y se le muestra en pantalla para que pueda introducirlo en sus siguientes requests en el header. Este token es creado desde la lógica de negocios con la ayuda de Guid.

Nuestro mecanismo de autenticación tiene diferentes tipos de roles los cuales nos permiten saber si ese usuario tiene permisos para hacer la acción solicitada, para esto implementamos un filtro llamado "ProtectFilter" el cual valida con la obtención del token enviado por header, si el token es nulo o no tiene permisos para la acción y si tiene permisos a esta acción le permite dar paso al envío de la request.

#### c) Descripción general de códigos de error.

Utilizamos códigos de error de la familia de los 200, 400 y 500, en concreto implementamos:

- **Ok (200):** Para las request que den como resultado correcto.
- **Bad Request (400):** Cuando se percibe un error del cliente.
- **Unauthorized (401):** Para aquellas request que requieran logeo y al momento de realizarlas no lo estés.
- **Forbidden (403):** Se utiliza para cuando no tienes permisos para realizar la acción de la request.
- **Not Found (404):** Cuando capturamos la excepción de un error por parte del cliente.
- **Unprocessable Entity (422):** Cuando capturamos un error de la propiedad enviada por el cliente.
- **Internal Server Error (500):** Cuando llega a un error del servidor o error desconocido por la lógica.

#### d) Descripción de los resources de la API.

##### **/api/users**

- POST
- Registrarse como usuario
- POST /api/users?Code=int
- Parámetros: Code = int de 6 dígitos
- Body en formato JSON:

```
{  
  "Email" : string,  
  "UserName" : string,  
  "Direction" : string,  
  "Password" : string  
}
```
- Responses:
  - Ok 200: Devuelve el usuario desde la base de datos, en un objeto UserDTO
  - Bad Request 400: "El email no es correcto"
  - Bad Request 400: "El email debe tener como mínimo 8 caracteres"
  - Bad Request 400: "El nombre de usuario no debe contener espacios"
  - Bad Request 400: "El nombre de usuario debe contener como mínimo 1 caracteres y máximo de 20 caracteres"
  - Bad Request 400: "La contraseña debe contener al menos un carácter especial"
  - Bad Request 400: "La contraseña debe tener como mínimo 8 caracteres"
  - Not Found 404: "El nombre de usuario no debe ser nulo"
  - Not Found 404: "La contraseña no debe ser nula"
  - Not Found 404: "El email no debe ser nulo"
  - Not Found 404: "La invitación con dicho código/usuario no existe"

##### **/api/login**

- POST
- Iniciar sesión
- POST /api/login?email=string&password=string

- Body en formato JSON:
 

```
{
  "Email" : string,
  "Password" : string
}
```
- Responses:
  - Ok 200: Devuelve el token del usuario que inicio sesión con un LoginDTO en formato JSON.
  - Bad Request 400: "El correo o la contraseña son incorrectos"

## **/api/invitations**

- POST
- Crear una invitacion para un usuario
- POST /api/invitations?PharmacyName=string
- Headers: "Token": string
- Body en formato JSON:
 

```
{
  "UserName" : string,
  "PharmacyName" : string,
  "Role" : int
}
```
- Responses:
  - Ok 200: Nos devuelve un InvitationDTO en formato JSON con el código de invitación.
 

```
{
    "Id" : int,
    "UserName" : string,
    "Role" : string
    "Code" : string,
    "PharmacyName" : string,
    "State" : string
  }
```
  - Bad Request 400: "El nombre de usuario no debe contener espacios"
  - Bad Request 400: "El nombre de usuario debe contener como mínimo 1 caracteres y máximo de 20 caracteres"
  - Unauthorized 401: "No tienes permisos para esta acción"
  - Forbidden 403: "No tienes permisos para esta acción"
  - Not Found 404: "El nombre de usuario no debe ser nulo"

- Not Found 404: "No existe el rol ingresado"
- Not Found 404: "La farmacia no existe"
- Not Found 404: "La invitación ya existe"
  
- GET
- Nos devuelve todas las invitaciones a usuarios
- GET/api/invitations
- Headers: "Token": string
- Responses:
  - Ok 200: Nos devuelve una lista de InvitationDTO
 

```
[{
  "Id" : int,
  "UserName" : string,
  "Role" : string
  "Code" : string,
  "PharmacyName" : string,
  "State" : string
}]
```
  
- PUT
- Actualiza la invitación a los nuevos valores enviados en el body.
- GET/api/invitations/{id}
- Headers: "Token": string
- Responses:
  - Ok 200: Nos devuelve un InvitationDTO en formato JSON con el código de invitación.
 

```
{
  "Id" : int,
  "UserName" : string,
  "Role" : string
  "Code" : string,
  "PharmacyName" : string,
  "State" : string
}
```
  - Bad Request 400: "No existe el rol ingresado"
  - Not Found 404: "La invitación ya fue utilizada"
  - Not Found 404: "La farmacia no existe"

## **/api/pharmacy**

- POST
- Crear una farmacia
- POST /api/invitations
- Headers: "Token": string
- Responses:
  - Ok 200: Nos devuelve una PharmacyDTO con los datos en formato JSON

```
{  
  "Name" : string,  
  "Direction" : string  
}
```
  - Bad Request 400: "La farmacia ya existe"
  - Bad Request 400: "El nombre de la farmacia debe contener mínimo 1 caracteres y máximo de 50 caracteres"
  - Unauthorized 401: "No tienes permisos para esta acción"
  - Forbidden 403: "No tienes permisos para esta acción"
  - Not Found 404: "El nombre de la farmacia no puede ser nulo"
- GET
- Obtener todas las farmacias
- GET /api/invitations
- Parámetros: Ninguno
- Responses:
  - Ok 200: Nos devuelve una lista de PharmacyDTO con los datos en formato JSON:

```
[{  
  "Name" : string,  
  "Direction" : string  
}]
```

## **/api/medicine**

- POST
- Crear una medicina
- POST /api/medicines

- Headers: "Token": string
- Body en formato JSON:
 

```
{
    "Name" :string,
    "Symptoms": string,
    "Presentation" : string,
    "Unit" : string,
    "Receipt" : string,
    "Quantity": int,
    "Price": int
}
```
- Responses:
  - Ok 200: Nos devuelve una MedicineDTO con los datos en formato JSON:
 

```
{
    "Name" : string,
    "Symptoms": string,
    "Presentation" : string,
    "Unit" : string,
    "Receipt" : string,
    "Quantity": int,
    "Price": int,
    "Pharmacy": string
}
```
  - Bad Request 400: "Esta medicina ya está en el sistema"
  - Bad Request 400: "El nombre del medicamento debe tener un largo de entre 1 y 30 caracteres"
  - Unauthorized 401: "No tienes permisos para esta acción"
  - Forbidden 403: "No tienes permisos para esta acción"
  - Not Found 404: "El nombre de medicamento no puede ser nulo"
- GET
- Listar las medicinas existentes y poder filtrarlas
- GET /api/medicines?nameMedicine=string&namePharmacy=string
- Parámetros: nameMedicine, namePharmacy
- Responses:
  - Ok 200: Nos devuelve una lista de MedicineDTO con los datos en formato JSON:
 

```
[{
```



```

        "Name" : string,
        "Symptoms": string,
        "Presentation" : string,
        "Unit" : string,
        "Receipt" : string,
        "Quantity": int,
        "Price": int,
        "Pharmacy": string
    }]

```

- POST
- Comprar una o varias medicinas
- POST /api/medicines/buy
- Parámetros: Ninguno, Headers: Ninguno
- Responses:
  - Ok 200: Nos devuelve una PurchaseDTO con los datos en formato JSON de la compra:
 

```

{
    "Id" : int,
    "PurchaseCode": string,
    "PharmacyName" : string,
    "Shopping" : IEnumerable<PetitionBuy>,
    "State" : string
}
                    
```
  - Bad Request 400: "No compraste ningún medicamento"
  - Bad Request 400: "No existe alguno de los medicamentos en la farmacia"
  - Bad Request 400: "No se permite ingresar la misma medicina más de una vez"
  - Bad Request 400: "La cantidad de la petición debe ser mayor a 0"
- DELETE
- Eliminar una medicina por su código
- POST /api/medicines/{code}
- Headers: "Token": string
- Responses:
  - Ok 200: "Se ha eliminado correctamente la medicina"
  - Bad Request 400: "La medicina no existe"

## /api/stockrequest

- POST
- Crear un pedido de reposición de stock
- POST /api/stockrequest
- Headers: "Token": string
- Body en formato JSON:

```
{  "Petitions" :    [{      "MedicineCode" : string,      "Quantity" : int    }]  }
```
- Responses:
  - Ok 200: Nos devuelve una StockRequestDTO con los datos en formato JSON:

```
{  "Id" : int,  "EmployeeUserName" : string,  "Petitions" : IEnumerable<Petition>}
```
  - Bad Request 400: "Alguna/s medicinas en la petición no existen en la farmacia. Revise los códigos de la/s medicina/s"
  - Bad Request 400: "La cantidad de la petición debe ser mayor a 0"
  - Unauthorized 401: "No tienes permisos para esta acción"
  - Forbidden 403: "No tienes permisos para esta acción"
- GET
- Obtener las solicitudes de reposición de stock
- GET /api/stockrequest
- Headers: "Token": string
- Responses:
  - Ok 200: Nos devuelve una lista de StockRequestDTO con los datos en formato JSON:

```
[{  "Id" : int,  "EmployeeUserName" : string,
```

"Petitions" : IEnumerable<Petition>

}}

- Unauthorized 401: "No tienes permisos para esta acción"
  - Forbidden 403: "No tienes permisos para esta acción"
- 
- PATCH
  - Aceptar un pedido de reposición de stock
  - PATCH /api/stockrequest/accept/{id}
  - Headers: "Token": string
  - Responses:
    - Ok 200: "Se aceptó exitosamente la reposición de stock."
    - Bad Request 400: "La solicitud de stock no es válida"
    - Bad Request 400: "No existe la solicitud de stock ingresada en tu farmacia"
    - Unauthorized 401: "No tienes permisos para esta acción"
    - Forbidden 403: "No tienes permisos para esta acción"
- 
- PATCH
  - Denegar un pedido de reposición de stock
  - PATCH /api/stockrequest/decline/{id}
  - Headers: "Token": string
  - Responses:
    - Ok 200: "Se denegó exitosamente la reposición de stock."
    - Bad Request 400: "La solicitud de stock no es válida"
    - Bad Request 400: "No existe la solicitud de stock ingresada en tu farmacia"
    - Unauthorized 401: "No tienes permisos para esta acción"
    - Forbidden 403: "No tienes permisos para esta acción"

## /api/shopping

- GET
- Obtener las compras pendientes y activas
- GET/api/shopping
- Headers: "Token": string

- Responses:
  - Ok 200: Nos devuelve una lista de PurchaseDTO con los datos en formato JSON:
 

```
[{
    "Id" : int,
    "PurchaseCode" : string,
    "PharmacyName" : string
    "Shopping" : IEnumerable<PetitionBuy>
    "State" : string
  }]
```
  - Unauthorized 401: "No tienes permisos para esta acción"
  - Forbidden 403: "No tienes permisos para esta acción"

## /api

- GET
  - Obtener la exportación de los datos
  - GET/api/**export**?mechanismName=string
  - Headers: "Token": string
  - Responses:
    - Ok 200: "Se ha exportado correctamente."
    - Unauthorized 401: "No tienes permisos para esta acción"
    - Forbidden 403: "No tienes permisos para esta acción"
- GET
  - Obtener una lista de los exportadores disponibles
  - GET/api/**exporters**
  - Parámetros: ninguno
  - Responses:
    - Ok 200: Nos devuelve en formato JSON una lista de string con los nombres de los exportadores

# Evidencia de Clean Code y de la aplicación de TDD

## a. Descripción de la estrategia de TDD seguida

Para el desarrollo de nuestro sistema utilizamos el desarrollo en TDD siendo esta una estrategia muy buena para tener confianza en el código y comprobar que este está funcionando de manera correcta y en su totalidad, siendo una herramienta útil en ese aspecto, desde otro punto de vista nos resultó más lenta al desarrollar, pero más eficaz al mantener y más confianza al trabajar, siendo que cuando se implementan nuevos cambios, se pueden correr los test y corroborar que todo funcione según lo esperado para seguir. Desarrollamos con el ciclo de TDD el cual se basa en tres fases:

- **RED:** Cuando creamos la prueba, la corremos y esta falla.
- **GREEN:** Cuando implementamos el código del que hicimos la prueba y este pasa.
- **REFACTOR:** Mejoramos la estructura del código implementado.

Además creemos que utilizar TDD es importante ya que es una forma de documentar nuestro código, del cual podemos ver como evoluciona con el tiempo, estando continuamente actualizada esta documentación al cumplir con el ciclo. En nuestro caso utilizamos Outside-In (Escuela de Londres), empezando por la funcionalidad final que se necesita, implementando poco a poco dicha funcionalidad, desarrollamos desde el “exterior” hacia “adentro”.

Usamos distintos proyectos de prueba siendo estos de tipo MSTest, creando proyectos de prueba para la BusinessLogic, otro para DataAccess, otro para Domain y otro para la WebApi, así mantener separados los test que deseamos hacer en diferentes proyectos y encontrarlos más fácil a la hora de desarrollar.

Utilizamos test doubles en nuestro caso mocks para testear objetos reales a partir de objetos artificiales simulando el comportamiento de uno real, los cuales implementan interfaces de los módulos que queremos probar. Esto lo utilizamos para testear BusinessLogic simulando el acceso a base de datos y la WebApi

simulando los métodos de la BusinessLoigc. Fueron especialmente útiles en la estrategia de desarrollo con TDD.

Gracias a todo esto obtuvimos una cobertura del código del 100%, aplicando de forma continua el desarrollo en TDD. Para lograr visualizar esta cobertura utilizamos la extensión Fine Code Coverage

## b. Informe de cobertura para todas las pruebas desarrolladas

### PharmaCity.BusinessLogic

PharmaCity.BusinessLogic.InvitationService	105	2	107	176	98.1%	<div><div></div></div>
PharmaCity.BusinessLogic.MedicineService	171	0	171	263	100%	<div><div></div></div>
PharmaCity.BusinessLogic.PharmacyService	41	0	41	75	100%	<div><div></div></div>
PharmaCity.BusinessLogic.SessionService	26	0	26	53	100%	<div><div></div></div>
PharmaCity.BusinessLogic.ShoppingService	120	0	120	184	100%	<div><div></div></div>
PharmaCity.BusinessLogic.StockRequestService	78	0	78	129	100%	<div><div></div></div>
PharmaCity.BusinessLogic.Tools.GuidService	21	0	21	45	100%	<div><div></div></div>
PharmaCity.BusinessLogic.Tools.HashService	7	0	7	26	100%	<div><div></div></div>
PharmaCity.BusinessLogic.UserService	63	0	63	109	100%	<div><div></div></div>

### PharmaCity.DataAccess

— PharmaCity.DataAccess	228	0	228	464	100%	<div><div></div></div>
PharmaCity.DataAccess.Context.PharmaCityDbContext	9	0	9	22	100%	<div><div></div></div>
PharmaCity.DataAccess.InvitationRepository	43	0	43	82	100%	<div><div></div></div>
PharmaCity.DataAccess.MedicineRepository	25	0	25	51	100%	<div><div></div></div>
PharmaCity.DataAccess.PetitionRepository	28	0	28	56	100%	<div><div></div></div>
PharmaCity.DataAccess.PharmacyRepository	34	0	34	66	100%	<div><div></div></div>
PharmaCity.DataAccess.ShoppingRepository	27	0	27	60	100%	<div><div></div></div>
PharmaCity.DataAccess.StockRequestRepository	28	0	28	57	100%	<div><div></div></div>
PharmaCity.DataAccess.UserRepository	34	0	34	70	100%	<div><div></div></div>

## PharmaCity.Domain

PharmaCity.Domain	215	0	215	548	100%	<div></div>
PharmaCity.Domain.DTO.ErrorDTO	4	0	4	10	100%	<div></div>
PharmaCity.Domain.DTO.IN.InvitationIN	3	0	3	15	100%	<div></div>
PharmaCity.Domain.DTO.IN.MedicineIN	9	0	9	21	100%	<div></div>
PharmaCity.Domain.DTO.InvitationDTO	6	0	6	19	100%	<div></div>
PharmaCity.Domain.DTO.LoginDTO	3	0	3	15	100%	<div></div>
PharmaCity.Domain.DTO.MedicineDTO	7	0	7	19	100%	<div></div>
PharmaCity.Domain.DTO.PetitionDTO	5	0	5	17	100%	<div></div>
PharmaCity.Domain.DTO.PharmacyDTO	3	0	3	10	100%	<div></div>
PharmaCity.Domain.DTO.PurchaseDTO	5	0	5	17	100%	<div></div>
PharmaCity.Domain.DTO.StockRequestDTO	3	0	3	11	100%	<div></div>
PharmaCity.Domain.DTO.UserDTO	5	0	5	17	100%	<div></div>
PharmaCity.Domain.Invitation	14	0	14	37	100%	<div></div>
PharmaCity.Domain.Login	2	0	2	14	100%	<div></div>
PharmaCity.Domain.Medicine	64	0	64	138	100%	<div></div>
PharmaCity.Domain.Petition	13	0	13	31	100%	<div></div>
PharmaCity.Domain.Pharmacy	12	0	12	31	100%	<div></div>
PharmaCity.Domain.Purchase	5	0	5	17	100%	<div></div>
PharmaCity.Domain.StockRequest	5	0	5	13	100%	<div></div>
PharmaCity.Domain.User	47	0	47	96	100%	<div></div>

## PharmaCity.WebApi

(Las clases con 0% de cobertura no son probadas. Ej. Filtros)

PharmaCity.WebApi	104	133	237	614	43.8%	<div></div>
PharmaCity.WebApi.Controllers.ExportController	11	0	11	36	100%	<div></div>
PharmaCity.WebApi.Controllers.InvitationController	13	0	13	45	100%	<div></div>
PharmaCity.WebApi.Controllers.LoginController	7	0	7	28	100%	<div></div>
PharmaCity.WebApi.Controllers.MedicineController	17	0	17	54	100%	<div></div>
PharmaCity.WebApi.Controllers.PharmacyController	10	0	10	32	100%	<div></div>
PharmaCity.WebApi.Controllers.ShoppingController	21	0	21	61	100%	<div></div>
PharmaCity.WebApi.Controllers.StockRequestController	18	0	18	54	100%	<div></div>
PharmaCity.WebApi.Controllers.UserController	7	0	7	27	100%	<div></div>
PharmaCity.WebApi.Filters.ExceptionFilter	0	44	44	61	0%	<div></div>
PharmaCity.WebApi.Filters.ProtectFilter	0	27	27	48	0%	<div></div>
PharmaCity.WebApi.Pages.ErrorModel	0	9	9	32	0%	<div></div>
PharmaCity.WebApi.Pages.IndexModel	0	6	6	25	0%	<div></div>
PharmaCity.WebApi.Pages.PrivacyModel	0	6	6	24	0%	<div></div>
PharmaCity.WebApi.Program	0	8	8	26	0%	<div></div>
PharmaCity.WebApi.Startup	0	33	33	61	0%	<div></div>

Como vemos en los resultados y lo mencionado anteriormente, pudimos alcanzar el 100% en todas las clases excepto en una que está en 98,1%, estos datos son brindados gracias a la extensión Fine Code Coverage. Como anteriormente mencionamos obtuvimos una cobertura completamente por encima del 98%, esto lo conseguimos asegurándonos de que cada método a implementar

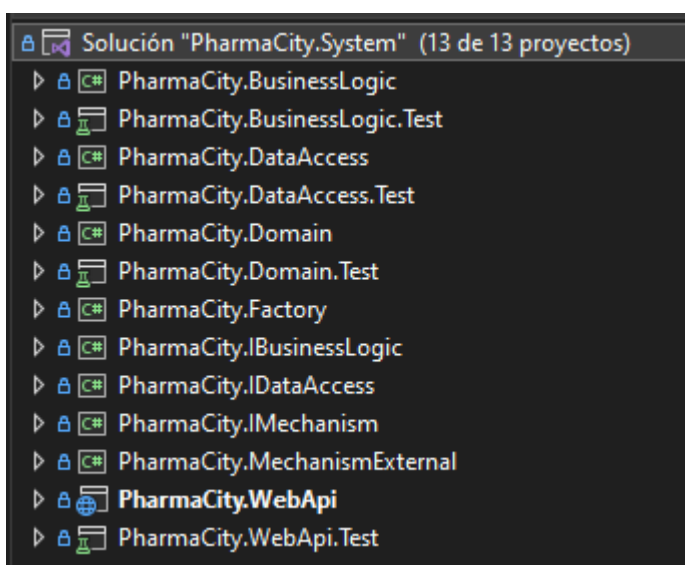
este testado y al implementarlo que esta prueba pase correctamente siguiendo los pasos de TDD.

Además queremos incorporar evidencia de la cantidad de líneas probadas, y las no probadas, para comprobar que efectivamente se realizó un amplio testeo de nuestro sistema. (Líneas no probadas son líneas que no tienen que ser probadas)

<b>Covered lines:</b>	3311
<b>Uncovered lines:</b>	302

- c. Es importante mantener una clara separación de los proyectos de prueba de los de la solución.

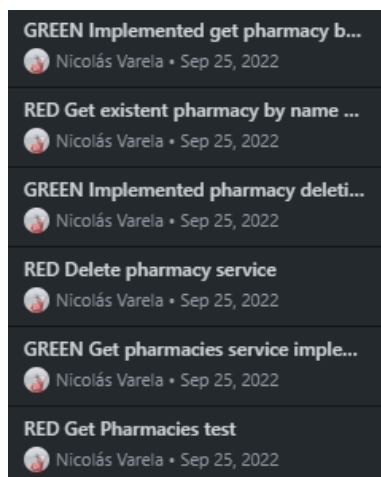
En la siguiente imagen vemos cómo separamos los proyectos de prueba de los proyectos a probar. En nuestro caso estos proyectos de pruebas fueron para cada proyecto a probar respectivamente y con sus nombres descriptivos, siendo `BusienssLogic.Test`, `DataAccess.Test`, `Domain.Test` y `WebApi.Test`





#### d. Funcionalidades especificadas como prioritarias (\*)

Evidencia de cómo se aplicó TDD correctamente para las funcionalidades especificadas como prioritarias (\*) con un ejemplo de ello:



Como vemos en las imágenes ambos desarrolladores del equipo de trabajo, realizamos una correcta implementación del ciclo TDD en las funcionalidades tanto

requeridas específicamente con esta estrategia como las que no, siguiendo en todo el flujo de trabajo esta modalidad de desarrollo y quedando documentado y accesible a ello en la respectiva plataforma de repositorio utilizada GitHub.