

## **Descripción del Diseño**

Diseño de Aplicaciones 2 - 17/11/2022

<https://github.com/ORT-DA2/DA2-163471-223139-201250>

Federico Czarniewicz - 201250 - M6C-1

Ignacio Olivera - 223139 - M6C

Cristhian Maciel - 163471 - N6A

<b>1. Descripción general del trabajo</b>	<b>4</b>
1.1. Introducción	4
1.2. Oportunidades de mejora	5
1.2.1. Bugs corregidos y mejoras implementadas	5
1.2.2. Bugs conocidos y deuda técnica	6
<b>2. Decisiones de diseño</b>	<b>7</b>
2.1. Diagrama general de paquetes	7
2.2. Capa de dominio	8
2.2.1. Paquetes Domain y AuthDomain	8
2.2.2. Paquete ExporterDomain	9
2.3. Capa de presentación	9
2.3.1. Paquete WebApi	9
2.3.2. Paquete WebApi.Models	9
2.3.3. Paquete WebApi.Filters	10
2.4. Capa transversal	10
2.4.1. Inyección de dependencias	10
2.4.2. Manejo de excepciones	10
2.5. Capa de lógica de negocio	11
2.6. Capa de autenticación y autorización	12
2.7. Capa de acceso a datos	15
2.7.1. Mecanismo de acceso a datos	15
2.7.2. Modelo entidad relación	17
<b>3. Vista de implementación</b>	<b>18</b>
<b>4. Impactos de nuevos requerimientos</b>	<b>19</b>
4.1. Lógica de negocio	19
4.2. Reflection	19
4.3. Cobertura de código de test unitarios	21
<b>5. Análisis de métricas</b>	<b>21</b>
5.1. Herramienta	21
5.2. Definiciones de métricas utilizadas	21
5.3. Resultados obtenidos	22
5.4. Principios REP, CCP, CRP	23
5.5. Principio de dependencias estables (SDP)	23
5.6. Principio de Abstracciones estables (SAP)	24
<b>6. Conclusión</b>	<b>25</b>
<b>7. Glosario</b>	<b>26</b>
<b>8. Anexo</b>	<b>27</b>
<b>8.1. Comparación de inestabilidad de paquetes</b>	<b>27</b>
<b>8.2. Interfaz de usuario</b>	<b>28</b>
8.2.1. Proyecto desarrollado	28

8.2.1.1. Módulos	29
8.2.1.2. Modelos	29
8.2.1.3. Componentes	31
8.2.1.4. Servicios	31
8.2.2. Usabilidad	32
8.2.2.1. Catálogo de medicamentos	32
8.2.2.2. Seguimiento de compra	32
8.2.2.3. Logueo	32
8.2.2.4. Pantalla de administrador	33
8.2.2.5. Pantalla de empleado	33
8.2.2.5.1. Compras	33
8.2.2.5.2. Medicamentos	33
8.2.2.5.3. Solicitudes	33
8.2.2.5.4. Exportación de medicamentos	33
8.2.2.6. Pantalla de dueño	34
8.3. Reporte de cobertura de código	34

# 1. Descripción general del trabajo

## 1.1. Introducción

Se nos solicitó realizar un sistema que cumpla con la capacidad de administrar diferentes tipos de usuarios para el manejo del negocio de una red de farmacias. La aplicación permite a los usuarios realizar compras bajo un perfil anónimo, como también registrarse siguiendo los requisitos correspondientes. Existen tres roles de usuario diferentes que modelan las reglas del negocio e interactúan con la plataforma de farmacias, la cual denominamos **PharmacyManager**.

En la primera entrega desarrollamos el backend de nuestra aplicación. Para esta segunda versión se agregan nuevos requerimientos y cambios por parte del cliente; así como también el objetivo principal de desarrollar la interfaz web de usuario e integrar las diferentes partes del sistema (backend y frontend) para completar el desarrollo del **PharmacyManager**.

Los principales servicios ofrecidos son:

- Registrar un usuario
- Dar de alta/baja a un medicamento
- Solicitar stock de un medicamento
- Aprobar/Rechazar solicitudes de stock
- Aprobar/Rechazar compras de un medicamento
- Ver solicitudes de reposición de stock para aprobar o rechazar, pudiendo filtrar la búsqueda
- Ver reportes de las compras

Los nuevos requerimientos para esta segunda versión son:

- Construir una Single Page Application con Angular que implemente todas las funcionalidades
- Editar invitaciones a usuarios nuevos
- Seguimiento de compra y el estado correspondiente de cada uno de sus medicamentos que pueden ser de variadas farmacias
- Exportación de medicamentos con la posibilidad de extender nuestro sistema agregando en tiempo de ejecución nuevos exportadores desarrollados por terceros

## 1.2. Oportunidades de mejora

### 1.2.1. Bugs corregidos y mejoras implementadas

- En la primera versión había quedado pendiente que el nombre de los roles sea case insensitive. Es decir, al asignarle un nombre a cada uno de los roles existentes, el mismo pueda empezar tanto con minúscula como con mayúscula. Esto quedó controlado con el simple hecho de integrar la interfaz de usuario, debido que ahora el usuario tiene las opciones a la vista para ingresar el rol en su debido momento.
- El borrado de un medicamento quedó implementado mediante un borrado lógico. Para poder mejorar este bug, optamos por agregar una propiedad al medicamento con su estado: activo/inactivo. De esta manera, nunca se eliminan realmente de nuestro sistema los medicamentos que fueron creados, permitiendo mantener íntegra la base de datos. Por ende, no repercute en las operaciones que previamente fueron realizadas con dicho medicamento como crear una solicitud o efectuar una compra, sino que inhabilita las futuras operaciones y el medicamento puede seguir “viviendo” en dichos objetos.
- Se desacoplan los paquetes AuthLogic y BusinessLogic separando responsabilidades de autenticación de la lógica de negocio, favoreciendo principios de paquetes que se detallan en la sección [Capa de autenticación y autorización](#).
- Se agregan endpoints necesarios para cumplir las demandas del frontend y se hace un rediseño especial en el endpoint de reportes de compras para un owner. Estos cambios se pueden ver reflejados en el documento adjunto Evidencia de Diseño y especificación de la API.pdf.
- A nivel de mantenimiento de código se trata de estandarizar en todo el proyecto el previo guardado en variables de los retornos de los métodos logrando un código más limpio.
- Se mejora el handleo de excepciones de autenticación ya que en la primera versión los intentos de autenticación inválidos que retornaban código de error 500 ahora están retornando código 401.

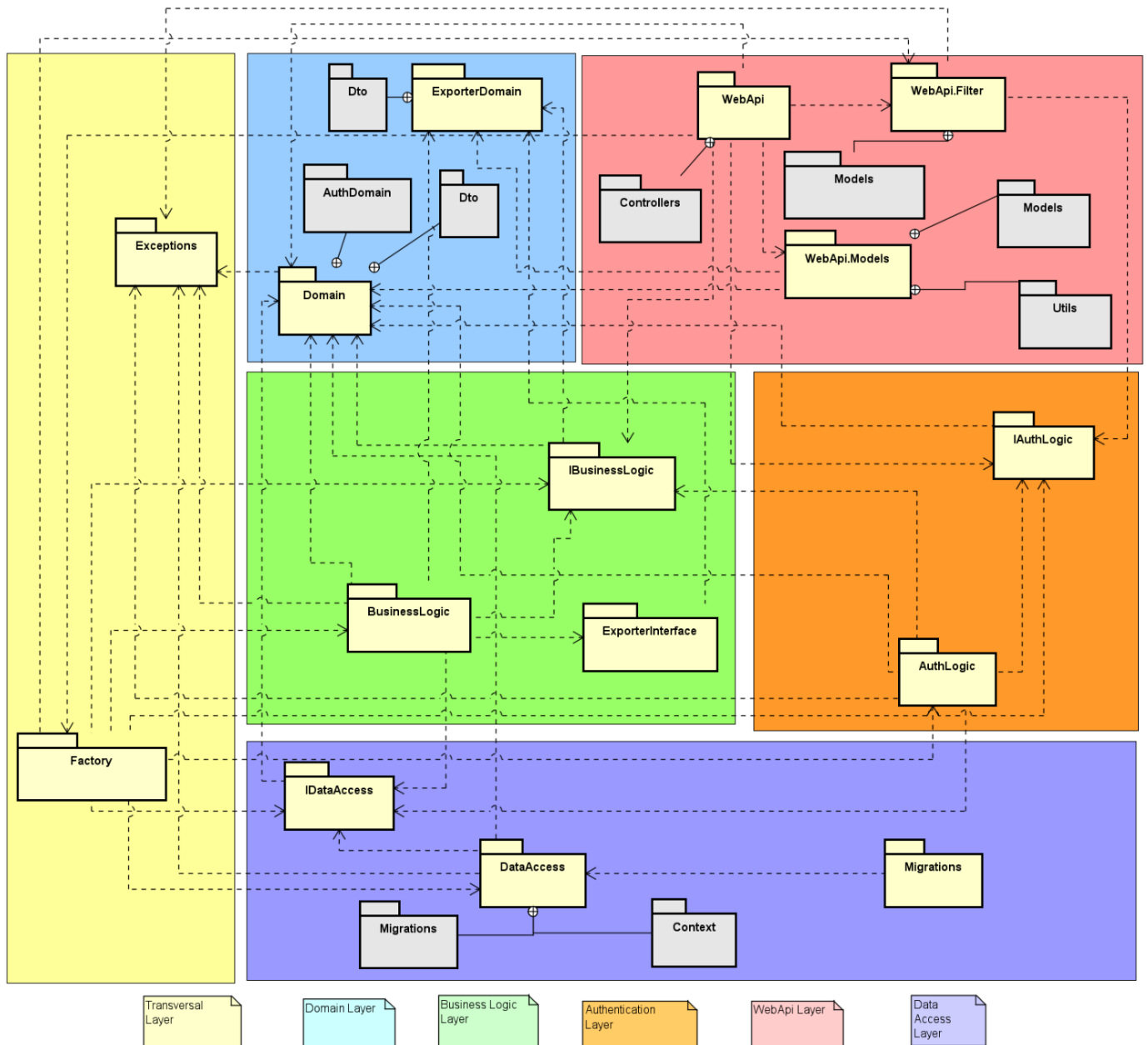
### 1.2.2. Bugs conocidos y deuda técnica

- El manejo de excepciones para el requerimiento de exportación de medicamentos es muy pobre ya que cualquier excepción lanzada por la implementación será mapeada a un error genérico convirtiéndose en una respuesta de código 500. Lo mejor sería definir las excepciones que cada implementación debería arrojar y que la API pueda retornar mensajes más explicativos.
- En cuanto al mapeo de entidades de dominio a objetos de la capa de presentación se trató de separar en clases diferentes por entidad para favorecer principios de responsabilidad única y en consecuencia tener clases más limpias. Aquí nos quedó pendiente separar algunas entidades.
- Faltó implementar el mostrar el mensaje de error cuando se ingresan datos incorrectos en el registro de un usuario nuevo en el frontend.

## 2. Decisiones de diseño

### 2.1. Diagrama general de paquetes

El siguiente diagrama de paquetes contempla todos los paquetes de nuestro sistema y las relaciones que existen entre ellos. También se agrupan lógicamente en capas para dar legibilidad y explicar decisiones de diseño de cada una en las secciones siguientes.



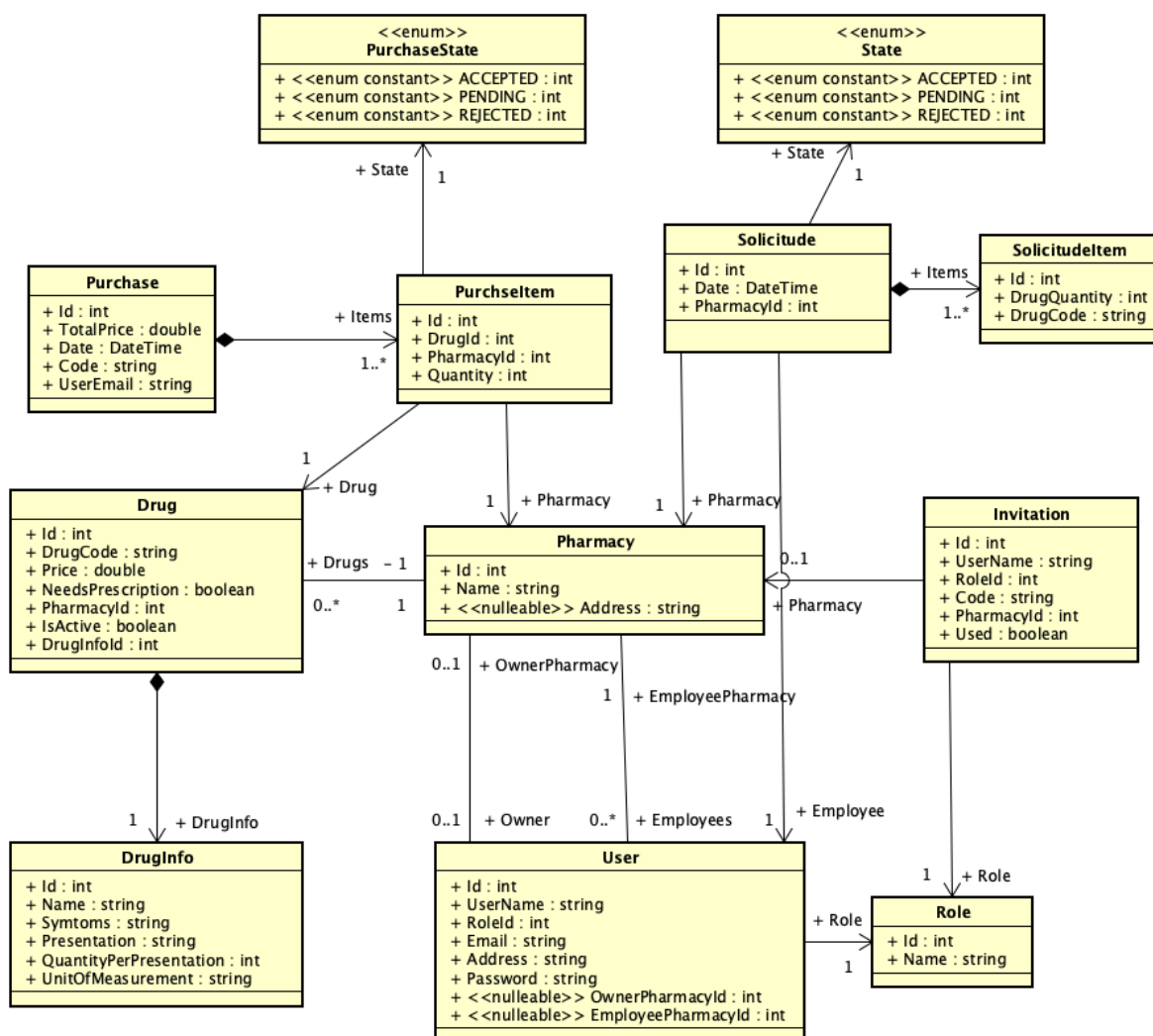
## 2.2. Capa de dominio

### 2.2.1. Paquetes Domain y AuthDomain

La capa de dominio es donde se encuentran todos los paquetes necesarios para modelar las entidades del negocio. Está compuesto por tres paquetes principales **Domain**, **AuthDomain**, **ExporterDomain**

El paquete **Domain** contiene todas las entidades que modelan el negocio de nuestro sistema PharmacyManger. El subpaquete **AuthDomain** contiene las clases relacionadas al mecanismo de autenticación y autorización. A pesar de que vimos una potencial segregación de responsabilidades entre estos dos dominios elegimos ubicar AuthDomain dentro de Domain porque están fuertemente acopladas mediante el uso de las clases User y Role. Sin embargo el desacoplarlas favorecería el reuso y modularización de la lógica del mecanismo de autenticación cumpliendo los principios de paquetes que apuntan a la cohesión y reuso, a saber **REP**, **CCP** y **CRP**. Al ser un cambio complejo decidimos no incluirlo en nuestro scope priorizando los requerimientos solicitados.

El siguiente diagrama de clases representa el paquete **Domain**:





El subpaquete **Dto** de **Domain** contiene los objetos que son utilizados como parámetros de entrada o retorno de los métodos de los servicios de lógica de negocio. Desde el punto de vista de principios de paquetes **REP**, **CCP** y **CRP** consideramos que podemos beneficiarnos si separamos este paquete de nuestro dominio de negocio ya que los objetos Dto tienen razones de cambio y de uso diferentes. Una posible mejor ubicación puede ser dentro del paquete **IBusinessLogic** ya que definen un contrato para interactuar con los servicios de negocio.

El subpaquete **Utils** de **Domain** surgió de la necesidad de reutilizar algunos algoritmos de validación de atributos en las clases de dominio. Cabe destacar que consideramos el principio de experto de información para agregar validaciones simples dentro de las clases de dominio, como por ejemplo, formato, valores no vacíos, rango numérico ,y de esta forma la capa de negocio recibe objetos validados. Con esta decisión también estamos favoreciendo el principio **SRP**.

### 2.2.2. Paquete ExporterDomain

El paquete **ExporterDomain** se creó independiente del paquete **Domain** para favorecer el reuso de la lógica relacionada a la exportación de medicamentos. Contiene una réplica de la clase de dominio **Drug** y objetos DTOs que se usan en el contrato **ExporterInterface** e **IExporterManager**. Nuestra intención fue favorecer el principio **CRP**. En la sección [Capa de autenticación y autorización](#) se explican los beneficios de esta decisión.

La razón del subpaquete **Dto** de **ExporterDomain** sigue el mismo razonamiento que el subpaquete **Dto** de **Domain** que se explicó anteriormente.

## 2.3. Capa de presentación

### 2.3.1. Paquete WebApi

El subpaquete **Controller** de **WebApi** agrupa los controllers que atienden las solicitudes a la API. Aplicando el principio de responsabilidad única llevamos la responsabilidad de filtros y “traducción” de modelos a objetos de dominio hacia otros paquetes y de esta forma **WebApi** queda más descongestionado.

### 2.3.2. Paquete WebApi.Models

El subpaquete **Models** se creó para agrupar todos los objetos utilizados en la capa de presentación ya sea en una solicitud o respuesta de nuestra API. La ventaja de utilizar estos modelos y no directamente los objetos de dominio es que podemos tener control de lo que queremos exponer a los clientes sin dependencia del dominio del negocio. Un cambio en lo que queremos retornar impacta sobre la capa de presentación y no en el dominio del negocio. De esta forma podemos decir que favorece **DIP** ya que un cambio en el protocolo o tecnología no tiene impacto en el alto nivel.

Por otro lado también se creó el subpaquete **Utils** el cual define clases para reutilizar funcionalidades de “traducción” de entidades de dominio a modelos que utiliza la API y por lo tanto favorecemos el principio **DRY**. Decidimos crear un paquete con estos utilitarios y no ubicarlos dentro de cada objeto de dominio para continuar evitando la dependencia desde la capa de dominio a la capa de presentación favoreciendo **DIP**.

### 2.3.3. Paquete WebApi.Filters

El paquete **WebApi.Filters** tiene la responsabilidad de definir los filtros utilizados en **WebApi** y se creó con el objetivo de mantener responsabilidades únicas sin sobrecargar WebApi. Existen filtros para resolver la autenticación y autorización antes que se resuelva cualquier acción, y también filtros para capturar excepciones convirtiéndolas a códigos de error al momento de retornar respuestas al cliente.

## 2.4. Capa transversal

### 2.4.1. Inyección de dependencias

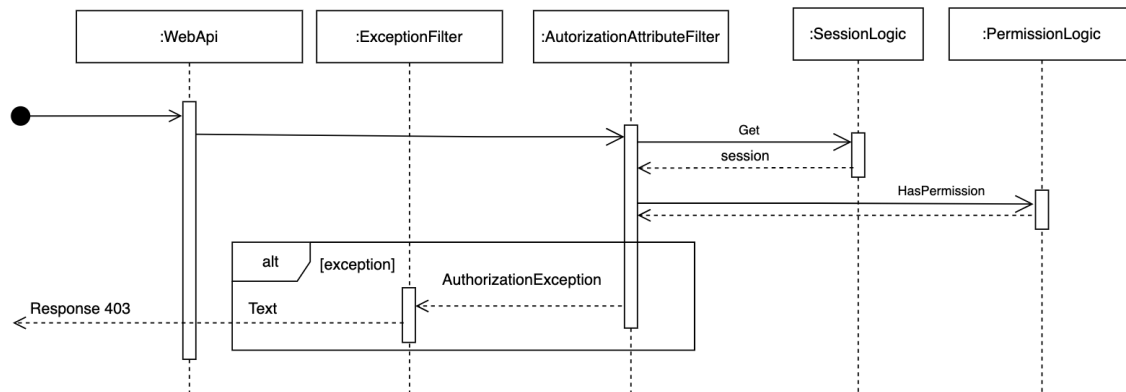
El paquete **Factory** tiene la responsabilidad de mantener la creación de instancias de los servicios del sistema de esta forma logramos un desacople entre **WebApi** y **BussinessLogic**, y entre **BussinessLogic** y **DataAccess** para favorecer **DIP** y **OCP**. Este paquete está en una capa transversal porque tiene una dependencia hacia todos los paquetes del sistema esto rompería con muchos principios pero es el precio que hay que pagar para centralizar la responsabilidad de creación de instancias y aplicar **IoC**.

Para su implementación, se creó una clase **ServiceFactory** que se encarga de instanciar y dejar disponible dicha instancia en el contenedor de dependencias para ser utilizado en un ciclo de vida AddScope (en cada solicitud HTTP).

### 2.4.2. Manejo de excepciones

El paquete **Exceptions** tiene la única responsabilidad de agrupar los tipos de excepciones definidas en el sistema. En el paquete **WebApi.Filters** se define la clase **ExceptionFilter** que captura todas las excepciones arrojadas por las capas inferiores y las mapea a respuestas de la API con su correspondiente código de error y mensaje.

En el siguiente diagrama de secuencia se aprecia como una excepción es capturada por el filtro y se retorna la respuesta correspondiente al cliente:



Se definieron los siguientes tipos de excepciones representadas con clases:

- **AuthenticationException**: Relacionada al mecanismo de autenticación
- **AuthorizationException**: Relacionada al mecanismo de autorización sobre un recurso luego de haberse autenticado
- **ResourceNotFoundException**: Cuando se intenta accionar sobre un recurso que no existe
- **ValidationException**: Errores de validación en formato ingresado o reglas del negocio

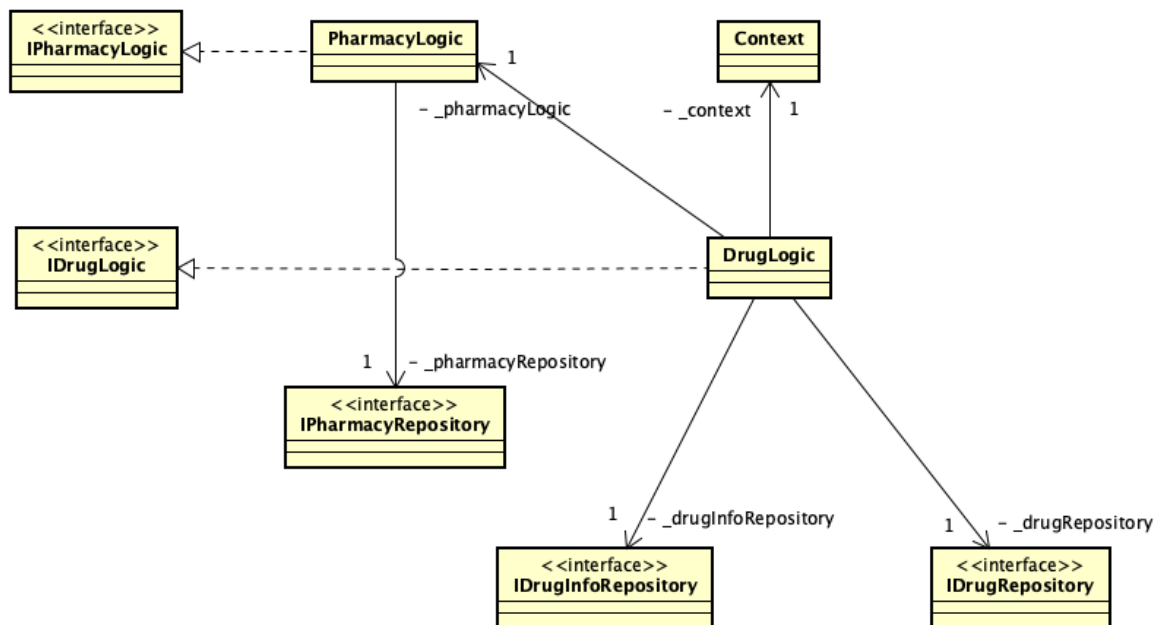
Con esta estrategia tenemos un paquete centralizado para definir excepciones del sistema pero violamos los principios de cohesión y reuso (**REP**, **CCP**, **CRP**) ya que estamos agrupando excepciones de diferentes contexto y si se pretende reutilizar un paquete nos estaríamos llevando también excepciones que no necesitamos.

## 2.5. Capa de lógica de negocio

En el paquete **BusinessLogic** se ubican todas las clases que implementan las reglas de negocio. Cada clase corresponde a acciones sobre un recurso en particular manteniendo el principio **SRP**. Se favorece el principio **ISP** y la alta cohesión proviendo interfaces definidas en **IBusinessLogic**.

También **IBusinessLogic** es un contrato entre **WebApi** y **BussinessLogic**, logrando un código fácil de testear. De todas formas podríamos no contar con esta abstracción y aún así seguir favoreciendo **DIP** ya que **WebApi** es un paquete de tecnología y puede depender del alto nivel sin problemas.

El siguiente diagrama de clases muestra la relación entre clases de **BusinessLogic**:



En el ejemplo vemos que **DrugLogic** tiene conocimiento de sus repositorios porque es el “experto” de esa información, lo mismo ocurre con **PharmacyLogic**. **DrugLogic** no accede directamente a los repositorios de base de datos sino que lo hace a través de **PharmacyLogic**. Consideramos que las clases de **BusinessLogic** se pueden ver entre sí sin violar algún principio ya que están en el mismo paquete y al ser de negocio son estables. Quisimos evitar muchas dependencias entrantes hacia un mismo repositorio aplicando el principio de dependencias estables **SDP** y lograr que un repositorio tenga un impacto menor sobre los que dependen de él. Si bien en este ejemplo consideramos solamente 2 clases este razonamiento se extrapola a todas las clases del paquete.

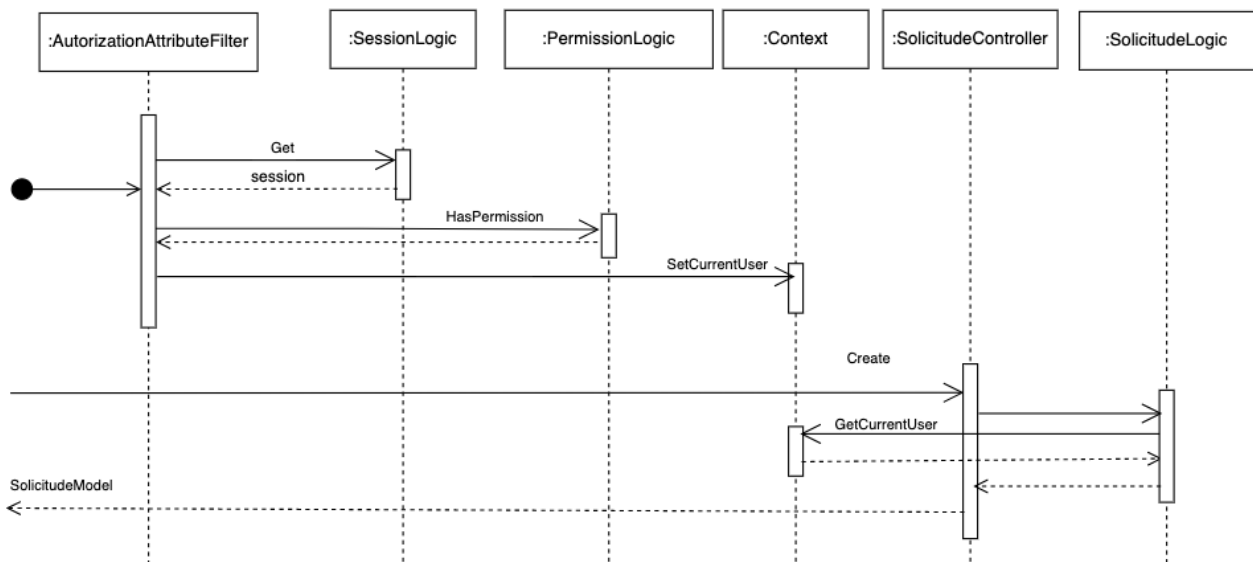
## 2.6. Capa de autenticación y autorización

El paquete **AuthLogic** implementa la lógica referida a la autenticación y autorización de usuarios en nuestro sistema. Se eligió crear un nuevo paquete para mantener una responsabilidad única sobre el paquete **BusinessLogic** y hacer modular la lógica de autorización y autenticación.

Logramos un **AuthLogic** “parcialmente” modularizado ya que se acopla del paquete **Domain** y a **IDataAccess** conservando dependencias a recursos que no necesita. Aquí

estaríamos violando los principios de paquetes **CCP** y **CRP**. Debido a la complejidad que nos llevaría separar el acceso a datos y el dominio decidimos en esta instancia continuar acoplados pero dejando la posibilidad de refactorizar esta solución a un módulo reusable.

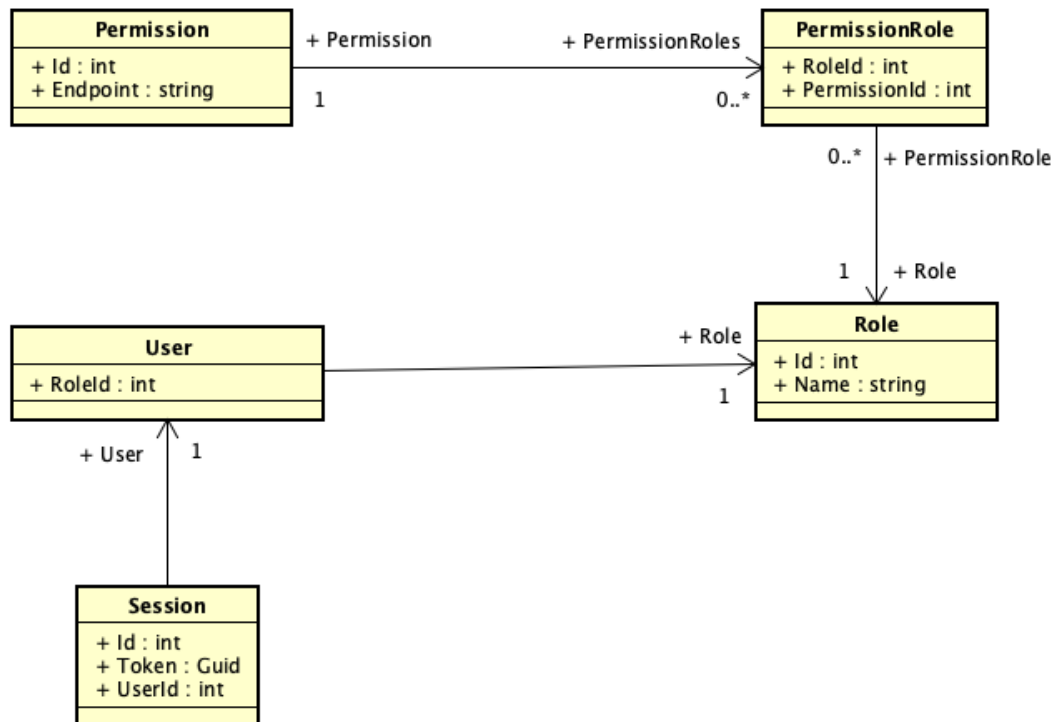
En el siguiente diagrama de interacción se muestra como se orquestan las clases de **AuthLogic** mediante el filtro de autorización en **WebApi** para poder garantizar el acceso de un usuario a un recurso del sistema:



Como se observa el usuario logueado se mantiene dentro de una instancia de la clase **Context** la cual está configurada para vivir dentro del contenedor de dependencias por el tiempo que viva la solicitud HTTP. Aprovechando este mecanismo de inyección de dependencia nos beneficiamos en desacoplar la lógica de autenticación de la lógica de negocio por lo que si a futuro se quiere migrar a un mecanismo de autenticación de terceros el proyecto sufrirá un impacto de cambio muy bajo. No podemos asegurar que el impacto es nulo porque nos quedó pendiente desacoplarse a nivel de capa de dominio como se discutió anteriormente.

El manejo de roles y permisos lo diseñamos de forma configurable para que se puedan agregar nuevos roles y permisos con un simple agregado de una nueva fila en la base de datos sin hacer cambios a nivel de código.

A continuación se muestra un diagrama de las clases en cuestión:



Como se observa el rol de un usuario lo modelamos como una instancia de una clase **Role** que tiene un nombre único. Contamos con los 3 roles (**Employee**, **Owner**, **Admin**) que se deben cargar junto con el esquema de la base de datos.

Un rol se asocia a una lista de permisos y cada permiso se asocia a una acción sobre un endpoint de nuestra API. La entidad intermedia **PermissionRole** fue necesaria para representar la relación N:N entre **Role** y **Permission** necesaria para Entity Framework Core. Aquí consideramos que sería beneficioso mantener un conjunto de entidades dentro de **DataAccess** que representen a nuestro **Dominio** para evitar este tipo de impacto debido a la tecnología y no violar de alguna manera el principio **DIP**.

## 2.7. Capa de acceso a datos

### 2.7.1. Mecanismo de acceso a datos

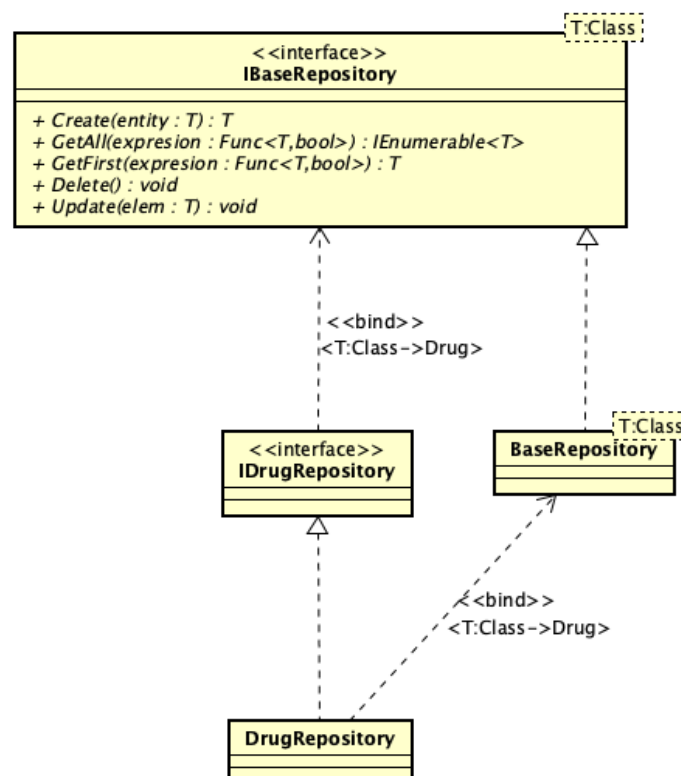
Para lograr una persistencia en una base de datos relacional decidimos utilizar la estrategia **Code First** mediante el uso del **ORM Entity Framework**. Esto nos permitió mapear entidades de dominio a tablas de base de datos de una forma automática sin la necesidad de definir un modelo de tablas. Aquí nos vimos atados al framework ya que en cada entidad de dominio teníamos que agregar un atributo **Id** correspondiente a la clave foránea de la entidad con relación de asociación. Por ello identificamos una oportunidad de mejora ya que podríamos definir dentro del paquete **DataAccess** los modelos a persistir en la base de datos y las correspondientes funciones de mapeo con las clases de dominio como se menciona en el capítulo anterior.

El subpaquete **Migrations** almacena las migraciones realizadas sobre la base de datos y en **Context** se define el contexto de la base de datos, con datos precargados como roles, permisos, usuario admin.

El paquete **IDataAccess** se ubica entre **BusinessLogic** y **DataAccess** favoreciendo el principio **DIP** y esta forma cortamos el impacto de cambio de las clases de tecnología hacia las de lógica de negocio. Además damos la posibilidad a futuro de implementar otras formas de persistencia de datos sin impactar en las capas de alto nivel.

A medida que fuimos construyendo los repositorios de cada entidad nos percatamos que los métodos CRUD eran funcionalidades en común a reusar. De ahí creamos la clase **BaseRepository** que implementa con el uso de Generics de .NET, y de esta forma logramos ahorrar código duplicado y tests unitarios.

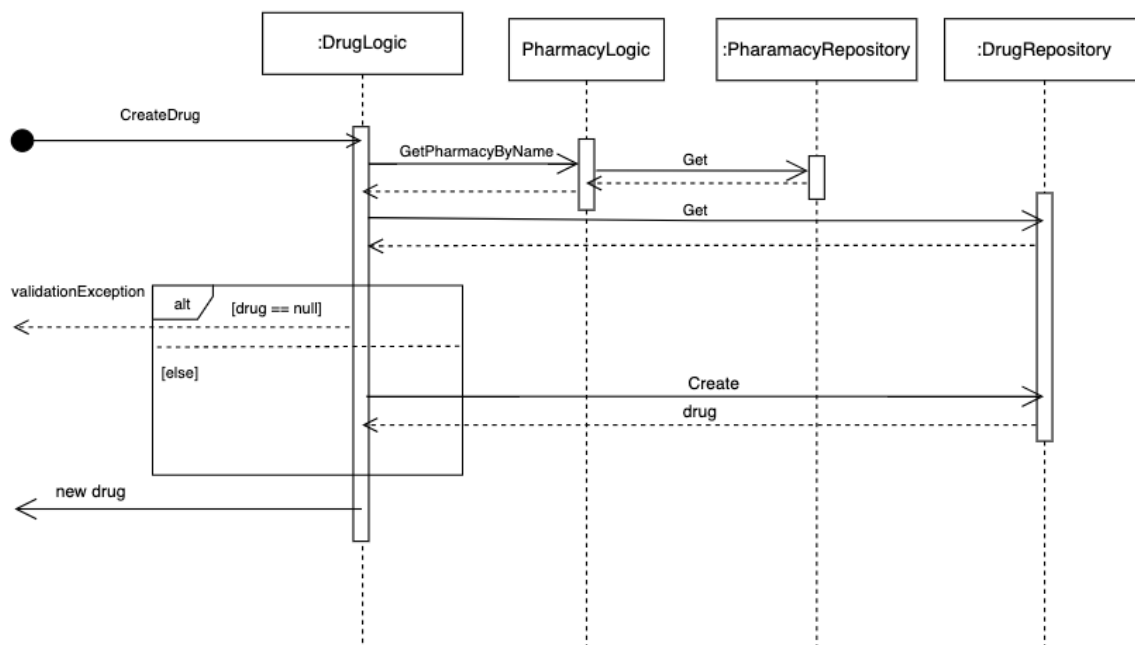
El siguiente diagrama muestra cómo logramos diseñar un repositorio genérico:



Se observa claramente como escondemos la implementación con SQL detrás de la interfaz **IDrugRepository** que es consumida por **DrugLogic** por lo que podemos decir que aplicamos el patrón **Adapter** siendo **DrugRepository** un adaptee al cliente ServerSQL y así logramos protegernos de cambios de librerías de terceros.

Para la obtención de datos en base decidimos utilizar el mecanismo de **Eager Loading** lo que nos llevó a sobrescribir algunos métodos Get y GetAll del repositorio base para poder realizar Joins con otras tablas y recuperar entidades dependientes. Aquí identificamos una oportunidad de mejora ya que podríamos implementar el patrón template method para sobrescribir en cada repositorio específico las entidades dependientes que queremos traer.

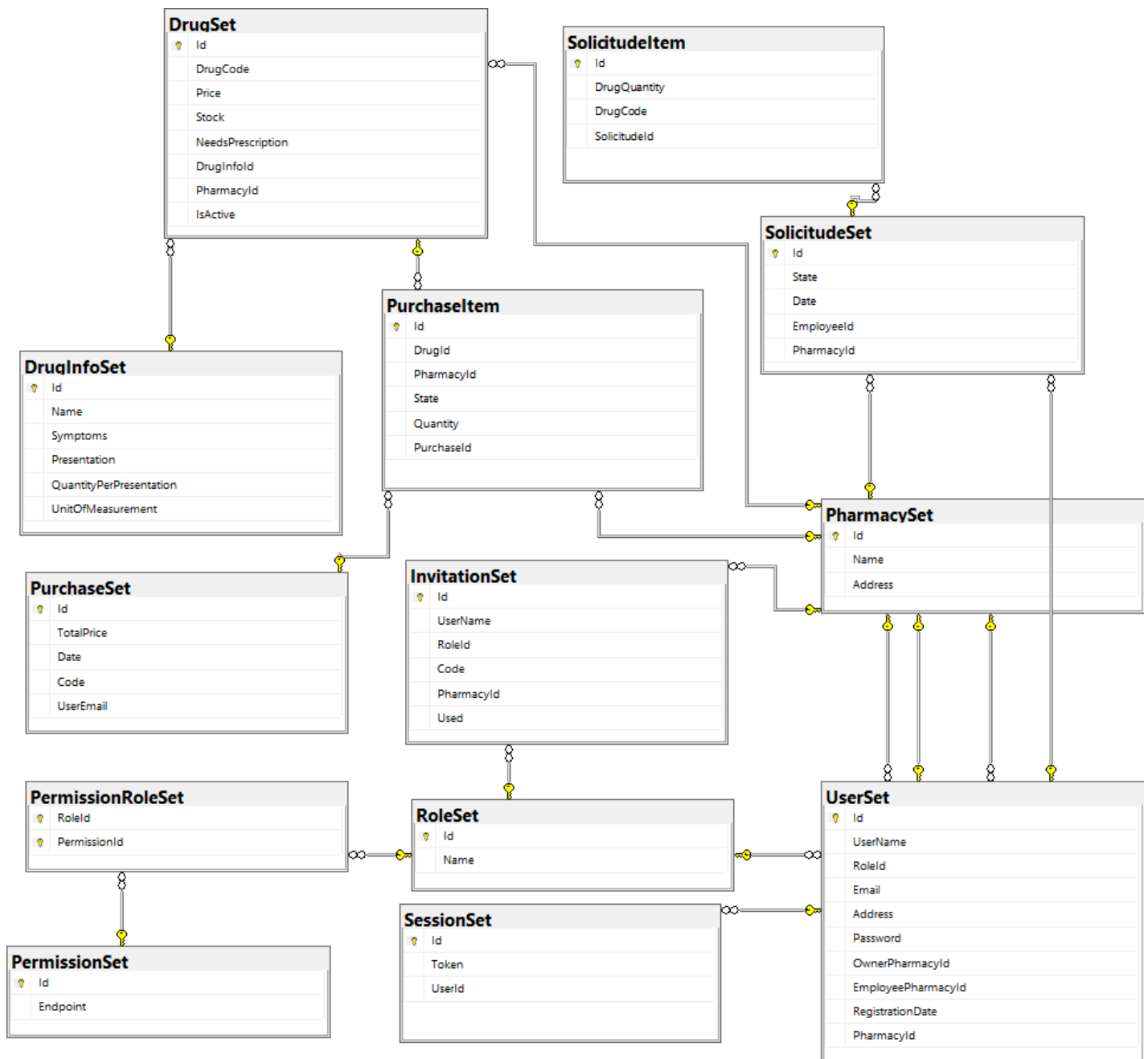
El siguiente diagrama de interacción muestra la interacción de las clases de lógica negocio entre sí y los correspondientes repositorios de cada entidad:





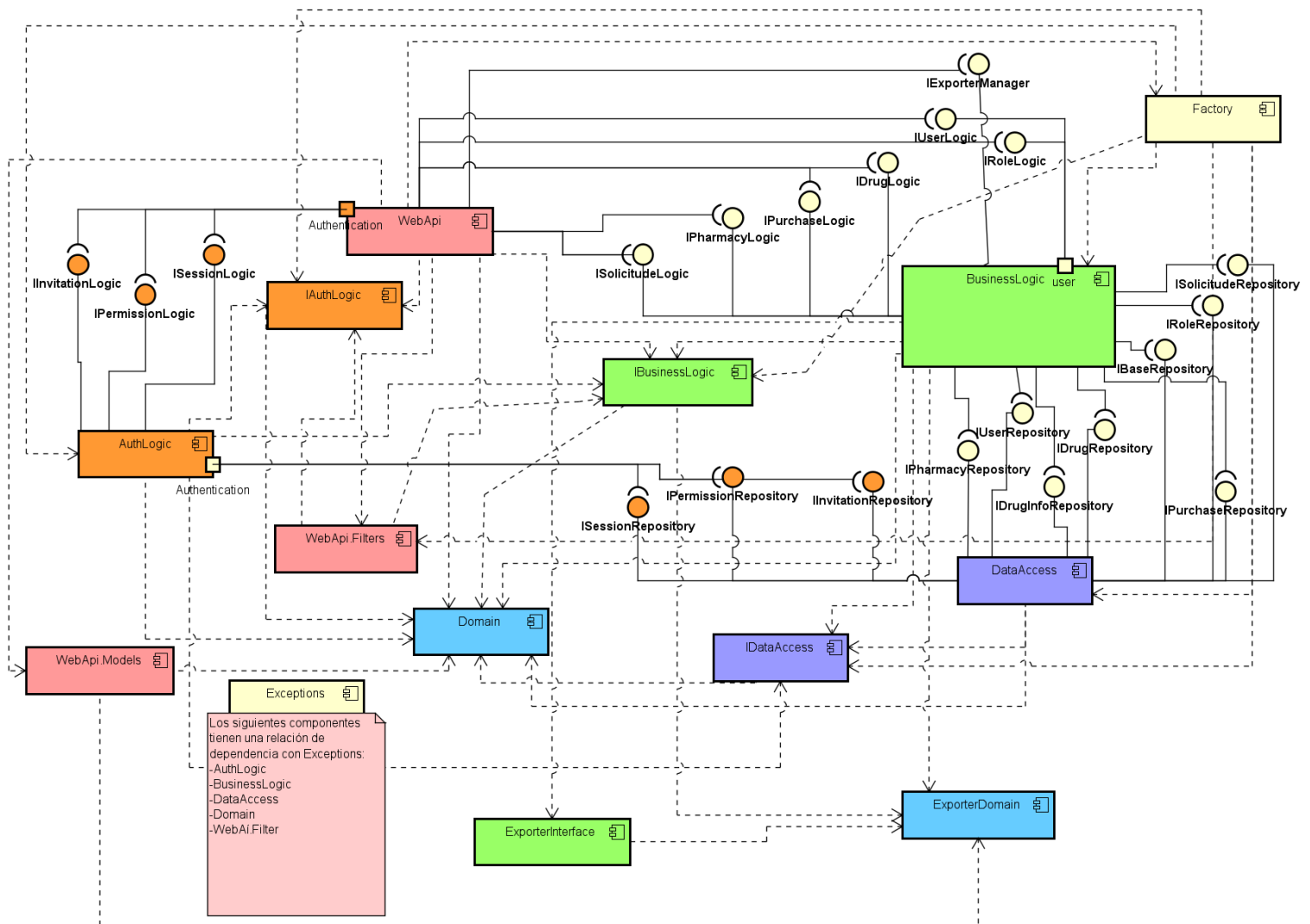
### 2.7.2. Modelo entidad relación

El siguiente diagrama entidad relación representa la estructura de tablas de nuestra base de datos, junto con sus relaciones y atributos:



### 3. Vista de implementación

Mediante el diagrama de componentes, podemos representar la vista de implementación. Acá es donde se muestra el sistema desde la perspectiva de un desarrollador y está enfocada en los diferentes artefactos y/o componentes del software. Este diagrama deja en evidencia el cumplimiento del principio **DIP** al ver como los módulos de alto nivel no dependen de los de bajo nivel, sino que ambos dependen de abstracciones. Esto se ve reflejado a través de las Interfaces que se proveen que también sirven como fachada entre ambos módulos.



## 4. Impactos de nuevos requerimientos

### 4.1. Lógica de negocio

En esta segunda entrega del obligatorio se hizo el mejor esfuerzo por mantener el mismo nivel de calidad de código siguiendo estándares de clean code que ya teníamos. Además al crear o modificar la API se pretende continuar con estilo REST.

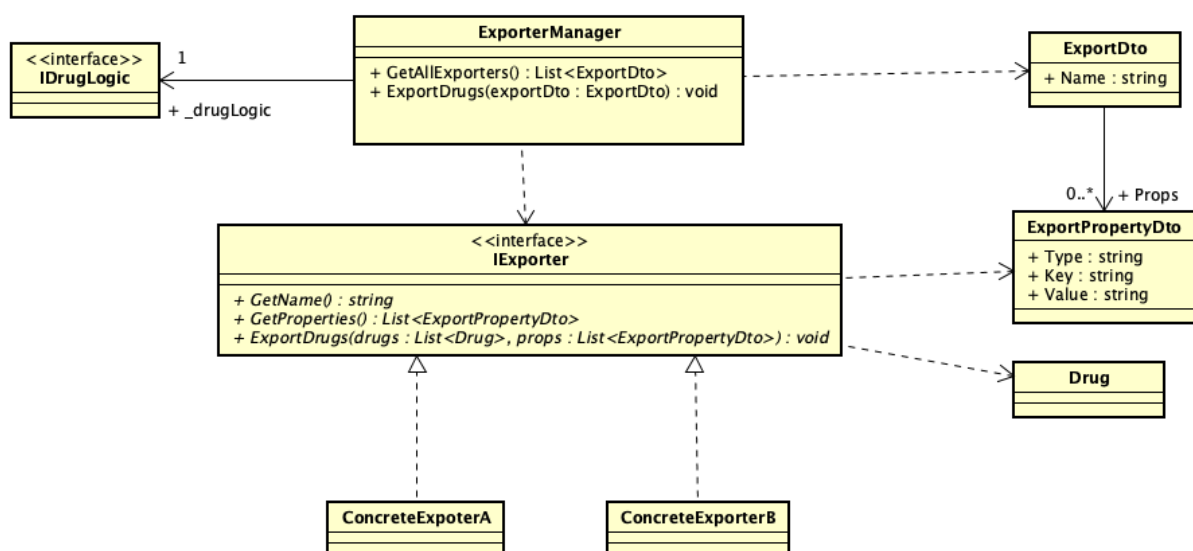
Los cambios de los nuevos requerimientos impactaron principalmente en las capas **WebApi**, **BusinessLogic** y **Domain**. Esto fue porque se cambió la forma en modelar las entidades invitación y solicitud, y también algunas reglas de negocio en relación a estas entidades.

En cuanto a los cambios que involucraban permisos para usuarios nos vimos beneficiados ya que el impacto fue sobre de la base de datos agregando o cambiando permisos y un nuevo filtro a nivel de **WebApi** que creamos para permitir un acceso anónimo y autenticado sobre el mismo endpoint. Por esto podemos decir que el mecanismo de autenticación es extensible a nuevos permisos y roles.

### 4.2. Reflection

Para cumplir con el requerimiento de exportación de medicamentos creamos un nuevo servicio que utiliza **Reflection** de .NET y de está forma leer assemblies e instanciarlos en tiempo de ejecución.

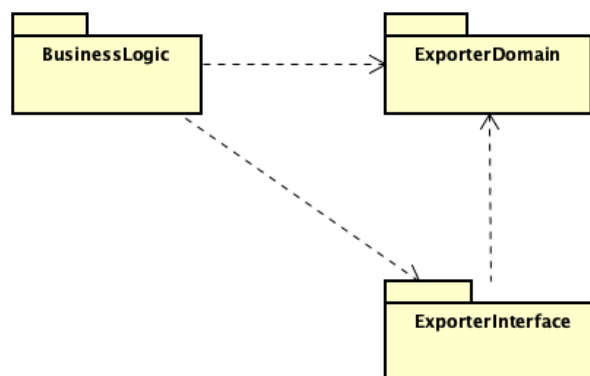
El siguiente diagrama de clases representa nuestra solución usando Reflection:



La clase **ExporterManager** tiene la responsabilidad de orquestar la exportación de medicamentos. Interactúa con el servicio **DrugLogic** para obtener la lista de medicamentos y se las envía al exportador concreto. **ExporterManager** utiliza **Reflection** para listar los assemblies que se encuentran disponibles en el directorio Exporters de donde esté corriendo el proyecto, instanciándolos en tiempo de ejecución. Estos assemblies deben implementar la interfaz **IExporter** para ser visibles por **ExporterManager**. Podemos decir que estamos haciendo uso de **polimorfismo** y **Liscov Substitution** para lograr tratar por igual a cada clase concreta.

En el caso que un tercero quiera desarrollar un nuevo exportador solamente tendría que implementar la interfaz **IExporter** y disponibilizar el compilado dll en el directorio Exporters. Cabe destacar que Drug es una representación diferente al medicamento que manejamos en la capa de dominio y podemos compartir ese paquete sin exponer el resto de nuestro dominio de negocio.

El siguiente diagrama muestra cómo favorecemos el principio **CRP** en los paquetes **ExporterDomain** y **ExporterInterface**, que contienen las clases **Drug**, **ExportPropertyDto** e **IExporter** respectivamente:



En el repositorio de github agregamos un nuevo proyecto con una implementación de exportador a un archivo json. El mismo se encuentra disponible en la carpeta **ExporterImplementation**.

### 4.3. Cobertura de código de test unitarios

En cuanto a los tests unitarios continuamos siguiendo TDD pero de una forma más descontracturada ya que se crearon commits con cambios y tests sin indicar el paso del proceso de TDD en que nos encontrábamos.

El porcentaje de cobertura actual del proyecto es 75% y se puede ver un detalle del reporte en la sección [Reporte de cobertura de código](#) del Anexo. En comparación con la primera entrega bajamos 2 puntos porcentuales debido principalmente a que no se incluyeron tests unitarios para el requerimiento de exportación de medicamentos.

Podemos decir que el porcentaje de cobertura no subió más debido a que hay constructores sin parámetros que no se testean principalmente en clases de BussinessLogic. También en DataAccess hay algunos repositorios concretos los cuales no tienen sus tests correspondientes, algunas pruebas del paquete Domain no fueron exhaustivas sobre todo en los métodos de validación y también métodos que obviamos como el Equals. Filters tampoco fue exhaustivo en los casos de uso de sus pruebas.

## 5. Análisis de métricas

### 5.1. Herramienta

Con el uso de la herramienta NDepend obtuvimos un reporte sobre las métricas de nuestro proyecto. Las métricas nos dan un panorama de evaluación sobre la calidad del diseño de nuestra aplicación específicamente analizado desde el código. Aunque no es suficiente para asegurar un buen diseño, aporta información objetiva sobre el código y ayuda a entender sus características.

Nos centraremos en las métricas orientadas a paquetes (assembly) dejando constancia del reporte más abajo y comentando sobre el cumplimiento de los principios de paquetes.

### 5.2. Definiciones de métricas utilizadas

La **Cohesion relacional (H)** nos brinda información sobre que tan cohesivo es un paquete analizando las relaciones que hay entre sus clases. Se consideran buenos las medidas entre 1,5 y 4

La **Inestabilidad (I)** está relacionada al acoplamiento aferente y eferente. Es decir, la cantidad de dependencias entrantes y salientes del componente, brindándonos información sobre qué tan fácil de cambiar es nuestro paquete. La Inestabilidad 0 significa que el paquete es estable solo tiene dependencias entrantes. Mientras que la inestabilidad 1 significa que el paquete es inestable y solo tiene dependencias entrantes.

La **Abstracción (A)** es una relación entre la cantidad de clases abstractas de un paquete y la cantidad total de clases en él. Un componente estable no es fácil de cambiar ya que tiene muchas dependencias. Una abstracción 0 indica un paquete que tiene solo clases concretas y 1 que solo hay clases abstractas.

### 5.3. Resultados obtenidos

La siguiente tabla muestra el resultado obtenido de las métricas de paquetes obtenidas con la herramienta NDepend.

Assemblies	Acoplamiento aferente	Acoplamiento eferente	cohesión relacional	Inestabilidad	Abstracción	Distancia
Exceptions	28	8	1	0.22	0	0.55
Domain	46	28	2.53	0.38	0	0.44
ExporterDomain	5	14	1.67	0.74	0	0.19
IBusinessLogic	15	25	1	0.62	0.7	0.23
IDataAccess	21	9	1	0.3	0.79	0.06
ExporterInterface	1	10	1	0.91	0.25	0.11
BusinessLogic	1	74	2.2	0.99	0	0.01
IAuthLogic	8	16	1	0.67	0.5	0.12
AuthLogic	1	46	1.5	0.98	0	0.02
DataAccess	1	120	2.24	0.99	0	0.01
WebApi.Filter	2	38	2	0.95	0	0.04
Factory	1	59	1.25	0.98	0	0.01
WebApi.Models	7	47	2.76	0.87	0	0.09
WebApi	0	115	1.77	1	0	0

## 5.4. Principios REP, CCP, CRP

Al observar los valores obtenidos de la métrica de **cohesión relacional** vemos que hay paquetes que están por debajo de los valores aceptables. Estos son **Exceptions**, **IBusinessLogic**, **IDataAccess**, **ExporterInterface**, **IAuthLogic** y **Factory**. Al tener medias que estan por debajo del umbral bueno de cohesión relacional podemos decir que las clases de estos tienen muy poco acoplamiento entre ellas. Tiene sentido que esto sea así porque son agrupaciones lógicas de contratos o clases que no son dependientes entre sí.

Por otro lado el resto de paquetes tienen una cohesión relacional buena, esto es porque en las clases de estos paquetes se mantiene la misma razón de cambio y se pueden reutilizar en conjunto y por lo tanto estamos cumpliendo **REP**, **CCP** y **CRP**.

## 5.5. Principio de dependencias estables (SDP)

*"Un paquete debe depender de un paquete más estable que él"*

Se basa en la métrica de inestabilidad y las dependencias entre paquetes.

En la sección [Comparación de inestabilidad de paquetes](#) del anexo se adjunta una tabla en donde se comparan las inestabilidades de paquetes dependientes. Se deduce que todos los paquetes cumplen con **SDP** con excepción de **IDataAccess**, **IBusinessLogic** y **Factory**. Las siguientes son las dependencias junto a su inestabilidad que hacen violar este principio:

- IBusinessLogic 0,62 -> ExporterDomain 0,74
- IDataAccess 0,3 -> Domain 0,38
- Factory 0,98 -> BusinessLogic 0,99
- Factory 0,98 -> DataAccess 0,99

En el caso de **Factory** es de esperar que sea menos inestable ya que no tiene muchas dependencias entrantes sino que son todas salientes con excepción de **WebApi**. También consideramos que la diferencia de 0.1 es muy pequeña.

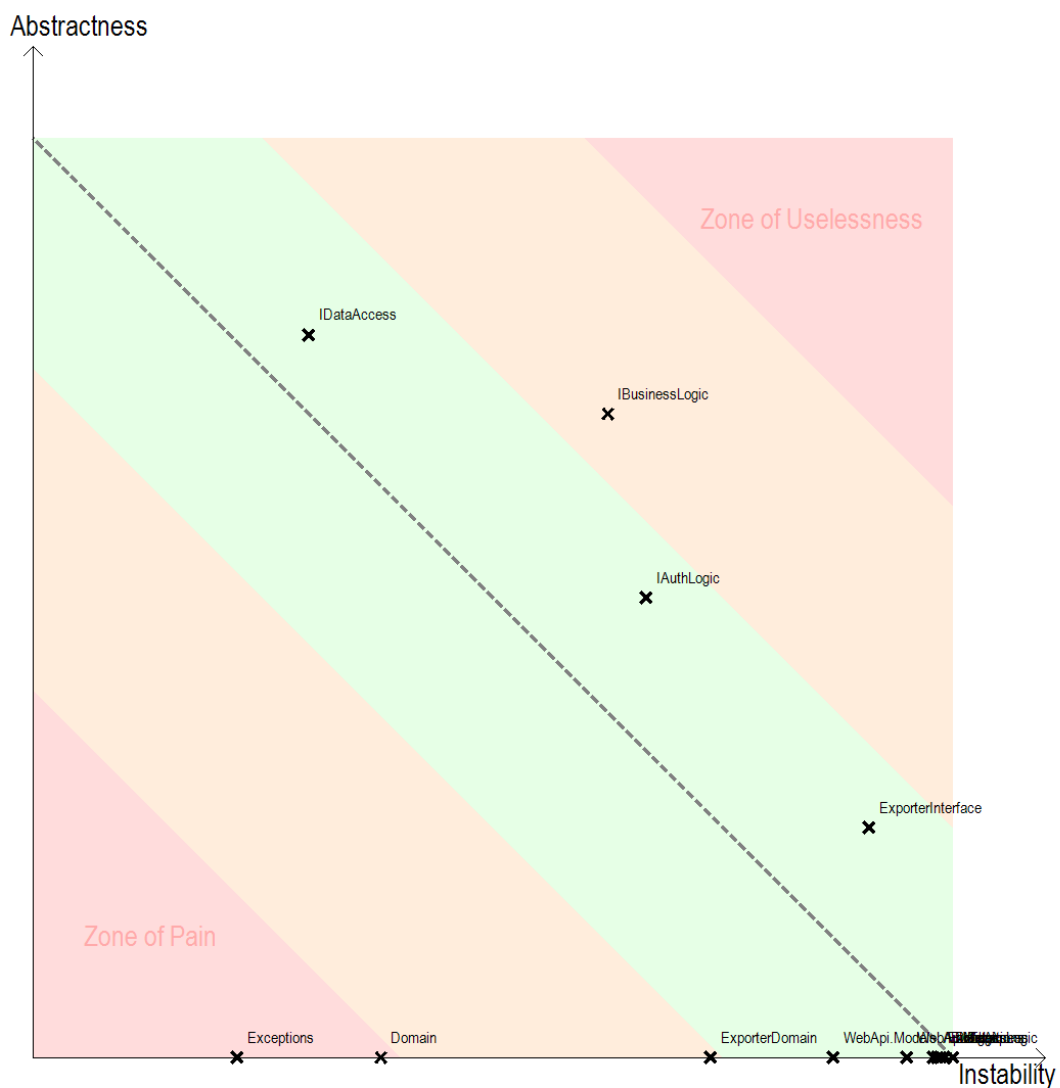
En el caso de **IBusinessLogic** e **IDataAccess** son dos clases más estables por lo que es de esperar que su grado de inestabilidad sea menor. **Domain** no tienen muchas dependencias entrantes pero si muchas salientes por lo que el grado de inestabilidad es bajo pero no lo suficiente para ser menor que **IBusinessLogic** que solo es consumido por **WebApi**. En el caso de **ExporterDomain** existen pocas relaciones aferentes lo cual aumenta el grado de inestabilidad.

Estos desvíos en la métrica de inestabilidad son despreciables desde nuestro punto de vista ya que en el reporte también se están considerando dependencias a librerías de terceros. Podemos afirmar que nuestro sistema si favorece el uso de SDP.

## 5.6. Principio de Abstracciones estables (SAP)

*“Un paquete estable, debe ser abstracto para que la estabilidad no impida la extensión. Un paquete inestable deberá ser concreto para que la inestabilidad permita cambiar fácilmente el código”*

Analizamos el cumplimiento de **SAP** mediante el uso de métricas de inestabilidad en conjunto con la abstracción y distancia. El siguiente gráfico generado de abstracciones/inestabilidades fue generado con Ndepend:





Como se observa en la imagen la recta que va desde el punto (0,1) al (1,0) es la secuencia principal y describe la zona donde podrían estar los paquetes idealmente ubicados. En base a la secuencia principal se definen categorías en el gráfico que describen diferentes comportamientos: zona verde, amarilla y roja. Las zonas rojas son las que debemos evitar y representan:

- Zona de dolor: paquete estable y concreto
- Zona de inutilidad: paquete inestable y abstracto

La distancia es la métrica que indica que tan alejado de la secuencia principal se encuentra un componente. Por ende, se busca que la distancia sea lo más cercano a 0 teniendo paquetes ni muy abstractos ni muy inestables.

Los paquetes **Exceptions**, **Domain** e **IBusinessLogic** son los que quedan a mayor distancia de la secuencia principal. **Exceptions** y **Domain** que se encuentran en la zona de dolor, son paquetes estables porque no tienen dependencias pero sí tienen muchas dependencias eferentes. Y al mismo tiempo son paquetes con clases concretas. Por lo que la razón de cambio se expande hacia muchas dependencias. El paquete **Domain** sería parte de nuestro modelado de negocio por lo tanto podemos decir que es sumamente estable. En cambio **Exceptions** al ser un paquete transversal a todos los es un motivo de cambio para los que dependen de él por lo tanto está violando el principio de SAP.

**IBusinessLogic** se encuentra también apenas alejado de la secuencia principal esto se da porque tiene dependencias hacia **Domain** y **ExporterDomain** lo que deja ligeramente más inestable. Al ser un paquete que agrupa interfaces es también abstracto. Podríamos pensar en hacer una segregación de interfaces en **IBusinessLogic** y que una de ellas corresponda a la lógica de exportación de medicamentos, de tal forma de reducir la inestabilidad de este paquete.

## 6. Conclusión

Como integrantes del equipo fue un desafío grande llevar adelante el desarrollo de este sistema a nivel de tecnología pero siempre teniendo en cuenta buenas prácticas de desarrollo, principios de clases y paquetes. Valoramos también la experiencia que tuvimos al entender un problema de negocio, estimar los requerimientos, experimentar con nuevas tecnologías y plantearnos objetivos a lo largo del semestre para poder llegar a las entregas superando obstáculos que se nos presentaban.

El paso del diseño y la construcción de un “mapa” antes de comenzar a desarrollar código es una excelente enseñanza que nos ha dejado esta experiencia. Algo que notamos durante el semestre es que la lista de cosas a mejorar pueden llegar a ser interminables, las discusiones en cuanto a pros y contras de diferentes alternativas y la vara alta que nos ponemos en el nivel de calidad de código pueden llegar a traducirse en mucho tiempo y

energía dedicada a análisis por lo que es importante poner foco también en resultados esperados con una metodología de trabajo ágil.

Nos llevamos la sensación de logro y una experiencia de desarrollo de inicio a fin en un tiempo bastante acotado, en especial con la curva de aprendizaje que tuvimos con Angular ya que ninguno del equipo manejaba esta tecnología.

## 7. Glosario

**SRP:** Principio de responsabilidad única  
**REP:** Reuse-release Equivalence Principle  
**CCP:** Common-Closure Principle  
**CRP:** Common-Reuse Principle  
**DRY:** Don't repeat yourself  
**DIP:** Dependency Inversion Principle  
**OCP:** Open-Close Principle  
**ISP:** Interface Segregation Principle  
**SDP:** Stable Dependency Principle  
**SAP:** Stable Abstract Principle  
**IoC:** Principio de Inversión de Control

## 8. Anexo

### 8.1. Comparación de inestabilidad de paquetes

Paquete	Inestabilidad de paquete	Dependencia	Inestabilidad de dependencia	SDP
Exceptions	0,22			N/A
Domain	0,38	Exceptions	0,22	TRUE
ExporterDomain	0,74			N/A
IBusinessLogic	0,62	Domain	0,38	TRUE
	0,62	ExporterDomain	0,74	FALSE
IDataAccess	0,3	Domain	0,38	FALSE
ExporterInterface	0,91	ExporterDomain	0,74	TRUE
BusinessLogic	0,99	IDataAccess	0,3	TRUE
	0,99	Domain	0,38	TRUE
	0,99	Exceptions	0,22	TRUE
	0,99	IBusinessLogic	0,62	TRUE
	0,99	ExporterDomain	0,74	TRUE
	0,99	ExporterInterface	0,91	TRUE
IAuthLogic	0,67	Domain	0,38	TRUE
AuthLogic	0,98	IDataAccess	0,3	TRUE
	0,98	IBusinessLogic	0,62	TRUE
	0,98	Domain	0,38	TRUE
	0,98	Exceptions	0,22	TRUE
	0,98	IAuthLogic	0,67	TRUE
DataAccess	0,99	Exceptions	0,22	TRUE
	0,99	IDataAccess	0,3	TRUE
	0,99	Domain	0,38	TRUE
WebApi.Filter	0,95	IAuthLogic	0,67	TRUE
	0,95	Domain	0,38	TRUE
	0,95	Exceptions	0,22	TRUE
Factory	0,98	IBusinessLogic	0,62	TRUE
	0,98	BusinessLogic	0,99	FALSE
	0,98	IAuthLogic	0,67	TRUE
	0,98	AuthLogic	0,98	TRUE
	0,98	Domain	0,38	TRUE
	0,98	IDataAccess	0,3	TRUE

	0,98	DataAccess	0,99	FALSE
	0,98	WebApi.Filter	0,95	TRUE
WebApi.Models	0,87	Domain	0,38	TRUE
	0,87	ExporterDomain	0,74	TRUE
WebApi	1	Factory	0,98	TRUE
	1	WebApi.Filter	0,95	TRUE
	1	IBusinessLogic	0,62	TRUE
	1	WebApi.Models	0,87	TRUE
	1	ExporterDomain	0,74	TRUE
	1	Domain	0,38	TRUE
	1	IAuthLogic	0,67	TRUE

## 8.2. Interfaz de usuario

### 8.2.1. Proyecto desarrollado

Para la implementación de la interfaz del usuario se utilizó el framework de Angular. En esta sección queremos destacar algunas de las implementaciones de diseño y arquitectura desarrolladas.

Se creó una Single Page Application (SPA) que permite renderizar los componentes de la página sin tener que refrescar el navegador. De esta manera el usuario puede interactuar con esta de una manera más fluida y rápida, permitiendo a los componentes estáticos permanecer en la página y los dinámicos renderizarse solo cuando se necesiten. Esto favorece de manera indirecta a la usabilidad, y creemos que una buena solución arquitectónica de la lógica del sistema debe venir acompañada de su par gráfico; de lo contrario posiblemente muchas de las funcionalidades y requerimientos previstos serán descartados si estos no son lo suficientemente amigables para el usuario.

A grandes rasgos, Angular es una plataforma y framework utilizado para escribir aplicaciones web. Cuenta con diferentes librerías; muchas son parte del core y son necesarias para el funcionamiento correcto de sus aplicaciones. En nuestro caso cabe aclarar que hemos hecho uso de la librería bootstrap para todo lo relacionado a los estilos de las distintas interfaces.

Para crear nuestras aplicaciones con Angular, generamos componentes que cuentan con un template HTML y una clase Typescript con lógica, y un archivo con estilos CSS. Así mismo, agregamos lógica en nuestros servicios para manejar la data que nuestra aplicación tendrá y finalmente “encapsulamos” nuestros componentes y servicios en módulos.

Podemos decir que se intentó que la estructura del proyecto realizado respete las convenciones que Angular y TypeScript recomiendan y como consecuencia Clean Code.

#### 8.2.1.1. Módulos

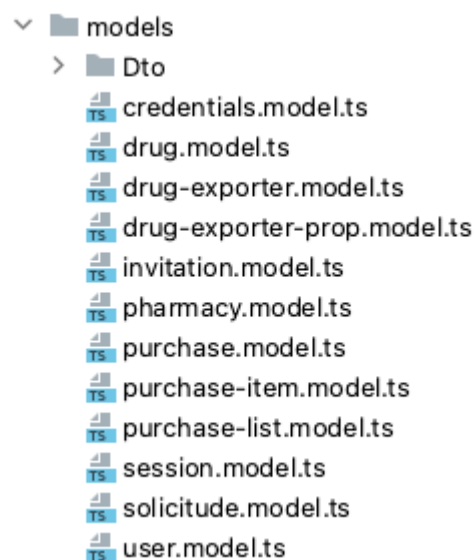
Un módulo o NgModule declara un contexto de compilación para un conjunto de componentes. Un NgModule puede asociar sus componentes con código relacionado, como servicios, para formar unidades funcionales. Cada aplicación generada con Angular cuenta con un root module llamado convencionalmente AppModule, el cual provee el mecanismo de arranque que inicia nuestra aplicación.

Nuestra aplicación cuenta también con un módulo de ruteo, el cual direcciona a los diferentes componentes según las acciones realizadas a lo largo del sistema. Cabe destacar, que según el rol del usuario que inicia sesión, las acciones permitidas, y las pantallas que se muestran son gestionadas mediante el uso de Guards. Los Guards utilizados fueron los siguientes: OwnerGuard, AdminGuard y EmployeeGuard, acordes a los roles de la aplicación.

#### 8.2.1.2. Modelos

Los modelos son las entidades del dominio al igual que manejamos en el backend. En otras palabras, son interfaces que nos permiten mapear las respuestas de la Api del backend de manera de convertirlos a entidades conocidas y encapsuladas mediante el tipado de datos. Estos modelos son utilizados de manera constante a lo largo de toda nuestra aplicación, muchas veces instanciados desde los servicios y utilizados luego en los componentes o viceversa.

Se detalla a continuación la lista de alguno de los modelos utilizados y el ejemplo de uno de ellos.

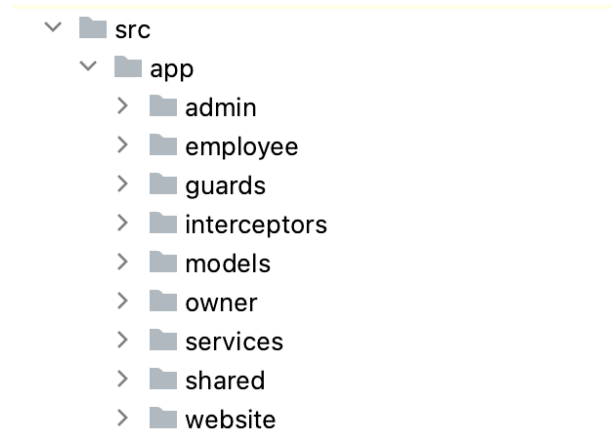


La siguiente interfaz representa el tipo de datos del modelo medicamento:

```
export interface Drug {  
  id: number,  
  drugCode: string,  
  name: string,  
  price: number,  
  symptoms: string,  
  presentation: string,  
  quantityPerPresentation: number,  
  unitOfMeasurement: string,  
  needsPrescription: boolean,  
  pharmacyId: number,  
  stock: number,  
  pharmacyName: string  
}
```

#### 8.2.1.3. Componentes

La organización de los componentes se realizó en base a los requerimientos del sistema y dependiendo de su uso y reuso global en la aplicación. Es decir, los mismos fueron encapsulados según su funcionalidad asociada o intención. Se muestra a continuación la división de los mismos seguido de una breve explicación:



Dentro del admin, owner y employee se encuentran todos los componentes relacionados a lo que pueden hacer dichos roles dentro de la aplicación, los mismos serán explicados con mayor detalle en la sección de usabilidad. Por otro lado se encuentran los guards y los modelos, así como también los interceptores, el website donde se encuentran todas las páginas públicas de la aplicación, como también el módulo shared que contiene los componentes que comparten distintos módulos y por último los servicios.

#### 8.2.1.4. Servicios

Estos servicios son los encargados de realizar llamadas al Backend, mapearlos y trasladarlos a donde sea necesario para que se procese tal información. En otras palabras, aquellas acciones que necesiten del uso de información asincrónica (por ej. al obtener la lista de solicitudes) deberán de suscribirse al evento, el cual el mismo procederá a obtener la información y en ese caso notificará a sus suscriptores para que actualicen su información. A continuación se detalla un ejemplo del servicio que trae del back todas las solicitudes:

```

getAllSolicitudes() {
  return this.http.get<SolicitudGetDto[]>(`url: `${this.apiUrl}``).pipe(
    map( project: (data: SolicitudGetDto[]) => {
      return data.map(this.solicitudDtoToModel)
    })
  )
}

```

## 8.2.2. Usabilidad

Tal como es requerido, la aplicación debe ser fácil de utilizar, intuitiva y atractiva. Si bien entendemos que no es lo fundamental de la entrega, es importante pensar en el usuario y su uso. Las heurísticas de Nielsen son un buen indicativo y guía de una buena usabilidad de la aplicación. A continuación ofrecemos una breve explicación del funcionamiento de cada una de las pantallas.

### 8.2.2.1. Catálogo de medicamentos

Por defecto el usuario anónimo ingresa al catálogo de medicamentos. Aquí se pueden visualizar todos los medicamentos registrados con sus respectivos datos, independientemente de la farmacia a la que pertenezcan. Se presentan dos tipos de filtros para poder visualizar los medicamentos por nombre o por stock.

Desde esta misma pantalla, se permite ver el detalle de los medicamentos así como también la posibilidad del usuario de agregar cierta cantidad al carrito de compras. Luego se pueden visualizar todas las compras agregadas con la posibilidad de realizar cambios, o bien finalmente crear la compra ingresando un mail. Una vez validado el mail, el usuario recibe un código para poder hacer un seguimiento de la misma.

### 8.2.2.2. Seguimiento de compra

La segunda opción proporcionada por la navbar, donde se puede buscar una compra realizada a partir del código de seguimiento, y obtener todos los datos y estado de la misma.

### 8.2.2.3. Logueo

Otra de las acciones que se puede realizar dentro de la navbar pública es el login, para poder navegar según el rol que corresponda. A continuación dividiremos la explicación según el rol con el cual el usuario se loguea.



#### 8.2.2.4. Pantalla de administrador

Por defecto se ingresa a la pantalla que muestra el listado de farmacias ya registradas en el sistema, pudiendo agregar una nueva en caso que lo desee. A su vez, el administrador podrá gestionar las invitaciones de usuarios para registrarse en el sistema. Es así que el admin le envía una invitación a cada usuario para que el mismo pueda completar su registro mediante un código que le fue enviado. A su vez, el admin puede visualizar el listado de invitaciones que fueron usadas y las que no, filtrando por nombre de farmacia, nombre de usuario y rol. Por último, podrá editar alguna de las invitaciones ya enviadas.

#### 8.2.2.5. Pantalla de empleado

Cada empleado pertenece solamente a una farmacia. Para registrarse, un empleado previamente debió haber recibido un código de registro generado por el admin tal como se explicó anteriormente. Una vez registrado y logueado, podrá navegar en las siguientes pantallas

##### 8.2.2.5.1. Compras

Aquí se visualizan todas las compras de los medicamentos de los usuarios, solamente con los medicamentos de la farmacia del empleado, para aceptarlas o rechazarlas por ítem una a una.

##### 8.2.2.5.2. Medicamentos

Los empleados son quienes crean e ingresan la información correspondiente para dar de alta un medicamento para su farmacia por defecto con stock en 0.

##### 8.2.2.5.3. Solicitudes

Aquí se visualizan las solicitudes de stock de los distintos medicamentos pudiéndose filtrar por código, nombre y fecha de solicitud. A su vez, el empleado puede en dicha pantalla crear una nueva solicitud de stock de alguna de las drogas que precise.

##### 8.2.2.5.4. Exportación de medicamentos


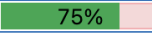













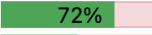



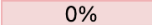
Se puede exportar el listado de drogas utilizando el exporter que se seleccione.

### 8.2.2.6. Pantalla de dueño


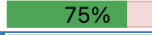

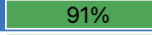

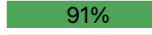



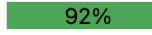


El dueño vendría a ser como el jefe de cada una de las farmacias. El mismo puede crear invitaciones para que nuevos empleados se unan al sistema, así como también ver los reportes de compra de la farmacia a la que pertenece y por último es el que tiene la potestad de aceptar o rechazar las solicitudes de stock que se realizaron.

## 8.3. Reporte de cobertura de código












### 8.3.1. Resumen

Symbol	Coverage (... ▼)	Uncovered/Total Stmts.
▼  Total	75% 	543/2136
>  AuthLogic	91% 	26/302
>  WebApi	83% 	24/141
>  Domain	79% 	90/439
>  WebApi.Filter	76% 	19/79
>  Exceptions	75% 	3/12
>  WebApi.Models	75% 	89/362
>  BusinessLogic	72% 	164/580
>  DataAccess	49% 	96/189
>  ExporterDomain	0% 	32/32





















### 8.3.2. AuthLogic

Symbol	Coverage (... ▼)	Uncovered/Total Stmts.
▼  Total	75% 	543/2136
▼  AuthLogic	91% 	26/302
▼  AuthLogic	91% 	26/302
>  SessionLogic	93% 	3/43
>  InvitationLogic	92% 	18/237
>  PermissionLogic	77% 	5/22

### 8.3.3. WebApi

Symbol	Coverage (... ▼)	Uncovered/Total Stmts.
▼  Total	75%	543/2136
>  AuthLogic	91%	26/302
▼  WebApi	83%	24/141
▼  WebApi.Controllers	83%	24/141
>  InvitationsController	100%	0/21
>  PharmaciesController	100%	0/14
>  PurchasesController	100%	0/31
>  SessionsController	100%	0/16
>  SolicitudesController	100%	0/21
>  DrugsController	58%	10/24
>  DrugExporterController	0%	14/14

### 8.3.4. Domain

Symbol	Coverage (... ▼)	Uncovered/Total Stmts.
▼  Total	75%	543/2136
>  AuthLogic	91%	26/302
>  WebApi	83%	24/141
▼  Domain	79%	90/439
▼  Domain	79%	90/439
>  DrugInfo	100%	0/12
>  Purchase	100%	0/27
>  Dto	100%	0/25
>  PurchaseItem	96%	1/24
>  User	94%	4/72
>  Utils	90%	2/20
>  Role	88%	2/16
>  Invitation	83%	4/23
>  AuthDomain	77%	5/22
>  SolicitudeItem	67%	7/21
>  Dtos	66%	30/89
>  Drug	64%	14/39
>  Solicitude	64%	9/25
>  Pharmacy	55%	10/22
>  Context	0%	2/2






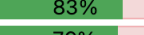

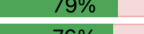

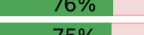

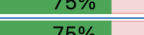

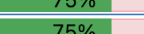

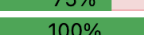

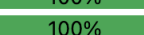

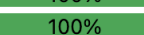

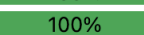

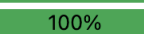

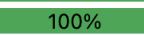

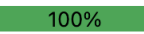

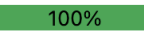

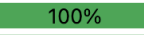



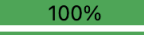

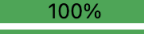

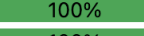

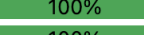



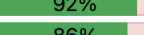

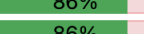

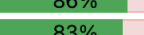

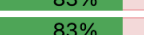

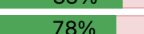

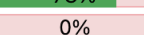

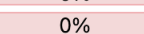

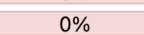

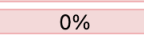

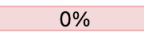




### 8.3.5. WebApi.Filter

Symbol	Coverage (... ▼)	Uncovered/Total Stmts.
▼  Total	75%	543/2136
>  AuthLogic	91%	26/302
>  WebApi	83%	24/141
>  Domain	79%	90/439
▼  WebApi.Filter	76%	19/79
▼  WebApi.Filter	76%	19/79
>  AuthorizationAttributeFilter	100%	0/38
>  ExceptionFilter	79%	5/24
>  Models	75%	1/4
>  AuthorizationAttributePublicFilter	0%	13/13

### 8.3.6. Exceptions

Symbol	Coverage (... ▼)	Uncovered/Total Stmts.
▼  Total	75%	543/2136
>  AuthLogic	91%	26/302
>  WebApi	83%	24/141
>  Domain	79%	90/439
>  WebApi.Filter	76%	19/79
▼  Exceptions	75%	3/12
▼  Exceptions	75%	3/12
>  AuthenticationException	100%	0/3
>  ResourceNotFoundException	100%	0/3
>  ValidationException	100%	0/3
>  AuthorizationException	0%	3/3

### 8.3.7. WebApi.Models

Symbol	Coverage (... ▼)	Uncovered/Total Stmts.
▼  Total	75% 	543/2136
>  AuthLogic	91% 	26/302
>  WebApi	83% 	24/141
>  Domain	79% 	90/439
>  WebApi.Filter	76% 	19/79
>  Exceptions	75% 	3/12
▼  WebApi.Models	75% 	89/362
▼  WebApi.Models	75% 	89/362
>  CredentialsModel	100% 	0/4
>  InvitationRequestModel	100% 	0/6
>  PharmacyModel	100% 	0/4
>  PurchaseItemModel	100% 	0/11
>  PurchaseItemPutModel	100% 	0/4
>  PurchaseItemReportModel	100% 	0/6
>  PurchasePutModel	100% 	0/2
>  PurchaseReportModel	100% 	0/4
>  PurchaseRequestModel	100% 	0/4
>  PurchaseResponseModel	100% 	0/15
>  SolitudeItemModel	100% 	0/10
>  SolitudePutModel	100% 	0/2
>  SolitudeRequestModel	100% 	0/2
>  SolitudeResponseModel	100% 	0/13
>  TokenModel	100% 	0/2
>  InvitationResponseModel	92% 	1/13
>  DrugRequestModel	86% 	3/21
>  InvitationConfirmedModel	86% 	2/14
>  InvitationPutModel	83% 	2/12
>  SessionProfileModel	83% 	1/6
>  Utils	78% 	35/162
>  DrugGetModel	0% 	27/27
>  ExportPropertyRequestModel	0% 	6/6
>  ExportPropertyResponseModel	0% 	4/4
>  ExportRequestModel	0% 	4/4
>  ExportResponseModel	0% 	4/4


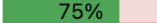



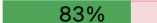



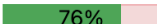



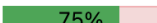

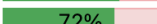



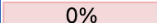


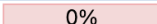


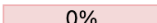


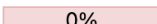
### 8.3.8. BusinessLogic

Symbol	Coverage (... ▼)	Uncovered/Total Stmts.
▼  Total	75%	543/2136
>  AuthLogic	91%	26/302
>  WebApi	83%	24/141
>  Domain	79%	90/439
>  WebApi.Filter	76%	19/79
>  Exceptions	75%	3/12
>  WebApi.Models	75%	89/362
▼  BusinessLogic	72%	164/580
▼  BusinessLogic	72%	164/580
>  UserLogic	100%	0/12
>  PurchaseLogic	91%	21/236
>  SolicitudeLogic	87%	12/92
>  RoleLogic	67%	4/12
>  DrugLogic	62%	44/117
>  PharmacyLogic	57%	21/49
>  ExporterManager	0%	62/62

### 8.3.9. DataAccess

Symbol	Coverage (... ▼)	Uncovered/Total Stmts.
▼  Total	75%	543/2136
>  AuthLogic	91%	26/302
>  WebApi	83%	24/141
>  Domain	79%	90/439
>  WebApi.Filter	76%	19/79
>  Exceptions	75%	3/12
>  WebApi.Models	75%	89/362
>  BusinessLogic	72%	164/580
▼  DataAccess	49%	96/189
▼  DataAccess	49%	96/189
>  BaseRepository<T>	100%	0/38
>  InvitationRepository	100%	0/25
>  PermissionRepository	100%	0/15
>  SessionRepository	100%	0/15
>  DrugInfoRepository	0%	3/3
>  DrugRepository	0%	22/22
>  PharmacyRepository	0%	15/15
>  PurchaseRepository	0%	25/25
>  RoleRepository	0%	3/3
>  SolicitudeRepository	0%	25/25
>  UserRepository	0%	3/3

### 8.3.10. ExporterDomain

Symbol	Coverage (... ▼)	Uncovered/Total Stmts.
▼  Total	75% 	543/2136
>  AuthLogic	91% 	26/302
>  WebApi	83% 	24/141
>  Domain	79% 	90/439
>  WebApi.Filter	76% 	19/79
>  Exceptions	75% 	3/12
>  WebApi.Models	75% 	89/362
>  BusinessLogic	72% 	164/580
>  DataAccess	49% 	96/189
▼  ExporterDomain	0% 	32/32
▼   ExporterDomain	0% 	32/32
>   Drug	0% 	22/22
>   Dto	0% 	10/10