Universidad ORT Uruguay

Facultad de Ingeniería

Bernard Wand Polak

Diseño de Aplicaciones II

Obligatorio 2

https://github.com/ORT-DA2/164660-185720-234059

Santiago Alvarez – N° Estudiante: 185720

Juan Castro – N° Estudiante: 164660

Marcelo Tilve – N° Estudiante: 234059

Contenido

Descripción general

La solución de este obligatorio brinda la posibilidad de gestionar farmacias y sus medicamentos a partir de distintos roles de usuarios teniendo presente los permisos de cada uno.

Se creó una Web Api llamada PharmaGo la cual expone recursos para que sean consumidos por los distintos tipos de usuarios que interactúen con ella. Algunos de sus servicios son manejo de farmacias, usuarios, medicamentos, gestión de stock de medicamentos, compras de medicamentos por usuarios anónimos, generación de reportes, entre otros.

Dentro del sistema existen 3 tipos de usuarios, Administradores, Empleados y Dueños. Como se mencionó anteriormente, cada uno tiene sus permisos, restringiendo a cada uno de la información a la que no debe tener acceso.

Para persistir la información se utilizó la herramienta Entity Framework Core aplicando la metodología Code First. Primero se definió el dominio de la solución para poder a través del uso de migraciones modelar la base de datos en SQL Server.

En cuanto al control de versionado, el repositorio de GitHub fue desarrollado teniendo en cuenta el modelo GitFlow.

Para evidenciar TDD, se realizaron commits de tests reds, luego tests green y por último commits de refactor de ser necesarios. A su vez, se utilizó la librería Mock para poder mockear la interfaz que utiliza cada capa de la aplicación y de esta forma lograr tests unitarios de cada clase que se testea.

El flujo de trabajo para las funcionalidades solicitadas por la letra consistió en desarrollar requerimientos en profundidad desde afuera hacia adentro o desde adentro hacia afuera. Para cada requerimiento, se comenzó por el controlador que se encargaba del mismo, para luego hacer la lógica y por último la gestión de los datos en la base. Idem para el desarrollo desde adentro hacia afuera, pero comenzando desde la base hasta llegar al controlador.

La utilización de TDD con Mocks y la inyección de dependencias con interfaces nos permitió poder basarnos en este modelo de trabajo, ya que nuestros proyectos no dependen directamente uno del otro y por lo tanto no era necesario tener toda una funcionalidad completa para probarla.

Diagrama de clases

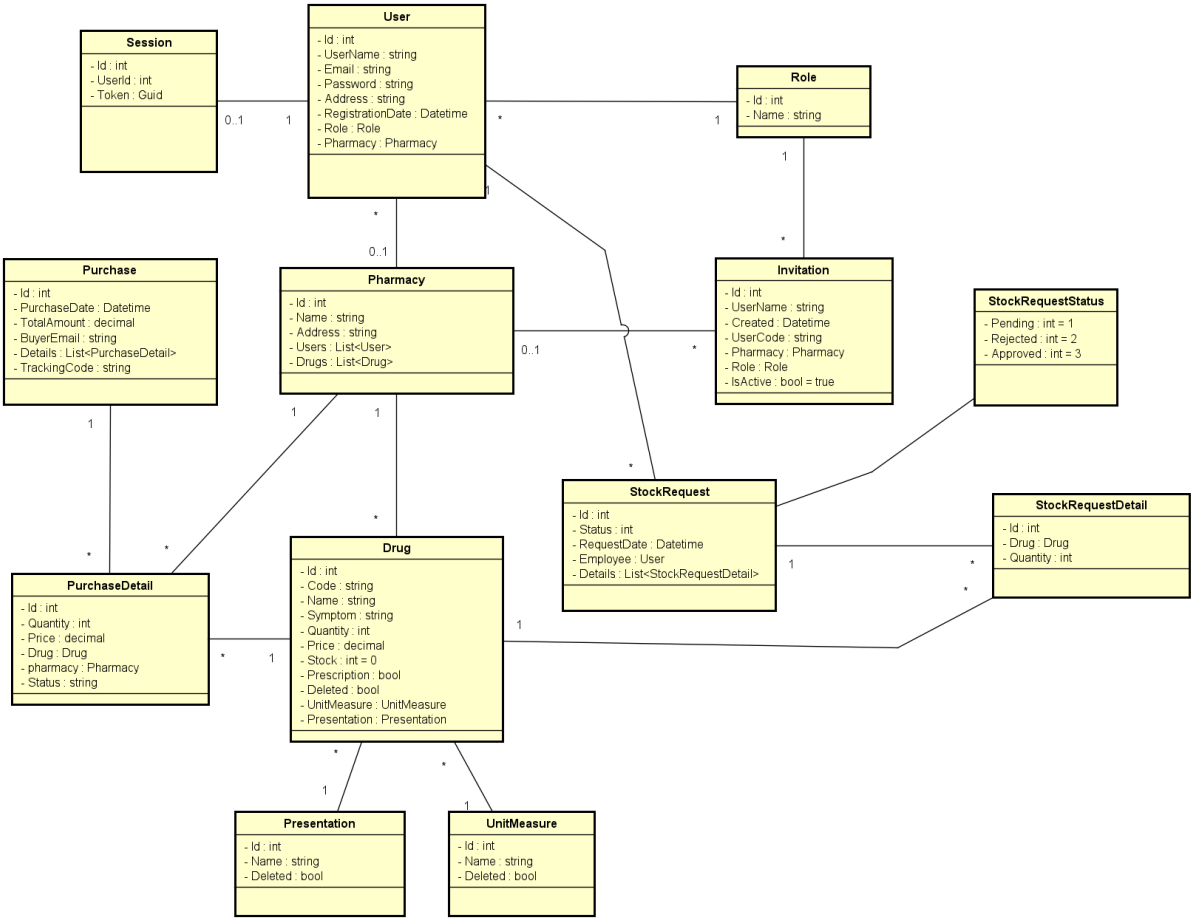


Ilustración 1: Diagrama de clases.

El diseño se basó en lograr obtener una solución que contemple la mantenibilidad y extensión del sistema, con el objetivo de que los cambios y nuevos desarrollos que puedan ser solicitados a futuro sean de bajo impacto al agregar nuevas características.

Para iniciar, en el paquete Domain, se encuentran las entidades necesarias para llevar a cabo el proyecto.

En el paquete ***IBusinessLogic***, se ubican las interfaces, que encapsulan operaciones que son específicas de las funcionalidades de cada controlador, que serán utilizadas por las clases del paquete ***BusinessLogic***, estas clases son las que proveen los servicios a la WebApi.

Lo mismo ocurre en el paquete ***IDataAccess*** y sus interfaces, que provee las operaciones que el paquete **DataAccess** implementará.

El paquete WebApi, mediante sus controladores, definen las rutas y se obtienen los datos de las request enviadas, con esa información, pide al servicio que resuelva la request, y el servicio mediante ***BusinessLogic*** accede a la base de datos a través del ***DataAccess***.

En este paquete también se encuentra el filtro de autorización, con el fin de autenticar permisos de acceso para los diferentes roles de la aplicación.

Está división de paquetes y las interfaces especifica de manera clara qué responsabilidad tiene cada paquete, volviéndonos independientes unos de otros, por lo que si en el futuro, se decide por ejemplo, dejar de utilizar Angular como Front end y pasar a un Windows Form, se debería solamente desechar la app de Angular y comenzar a implementar el Windows Form.

Modelo de tablas

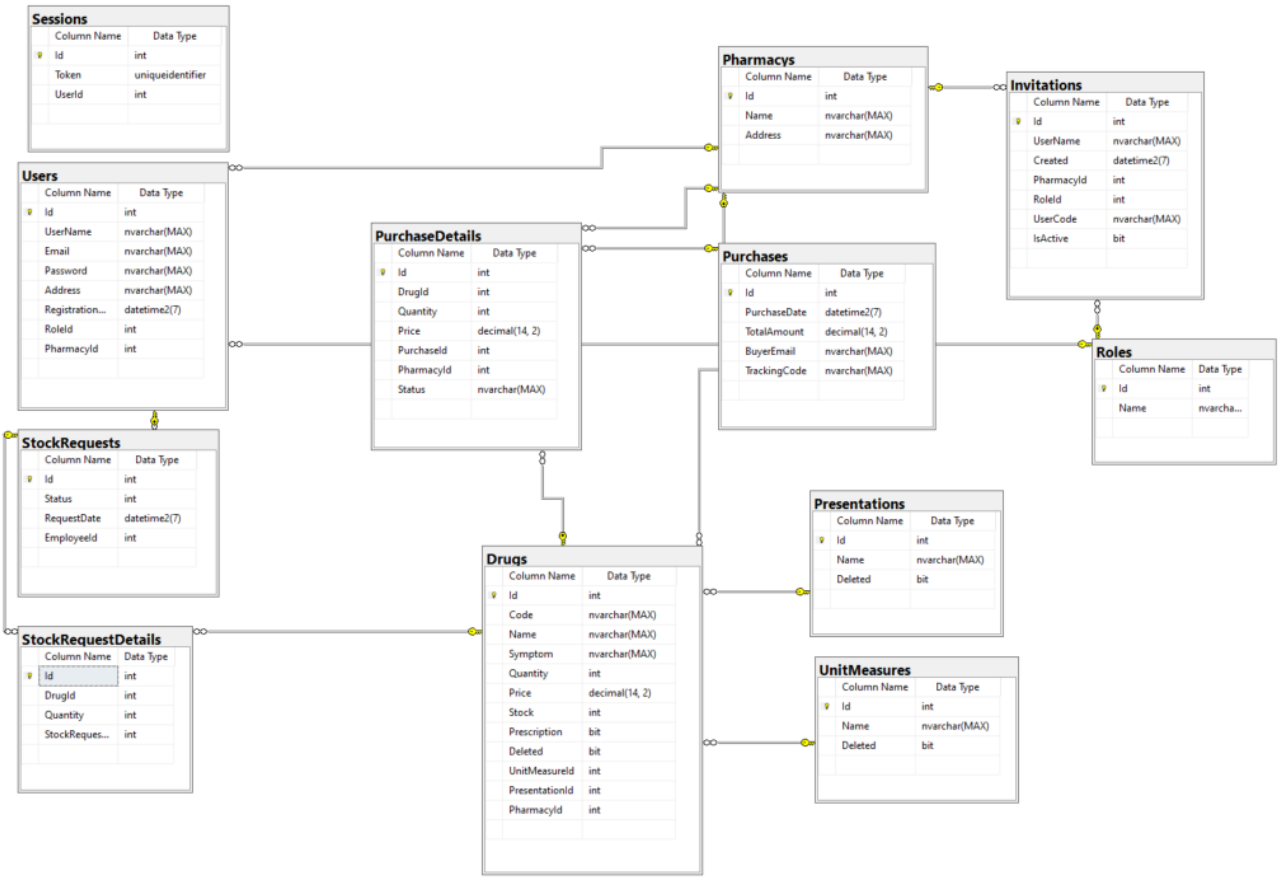


Ilustración 2: Modelo de tablas de la base de datos.

Diagrama de Paquetes

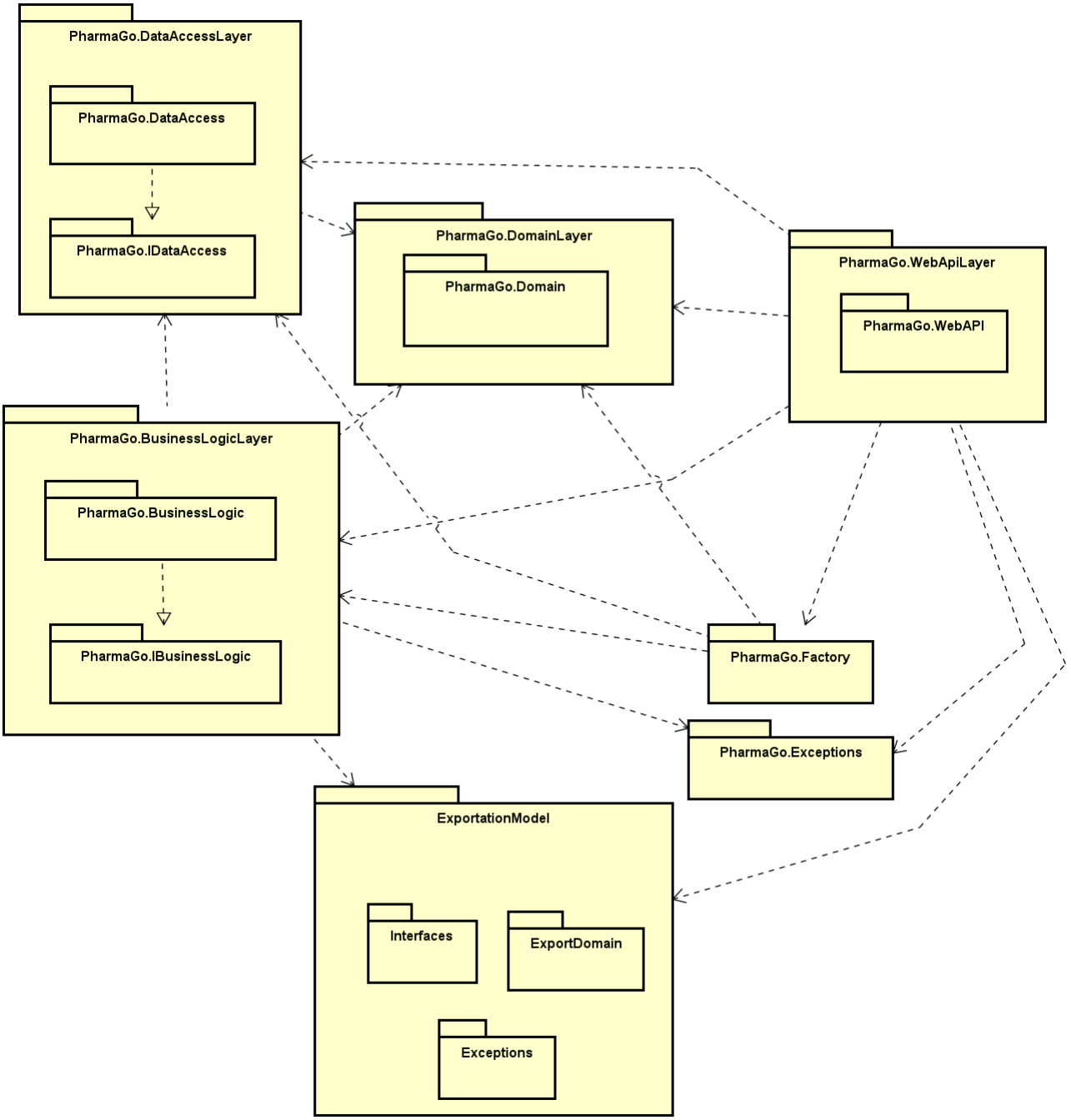


Ilustración 3: Diagrama de paquetes.

A continuación se detallan las clases de cada paquete. Tener en cuenta que para el paquete Dominio se debe observar la *Ilustración 1: Diagrama de clases*

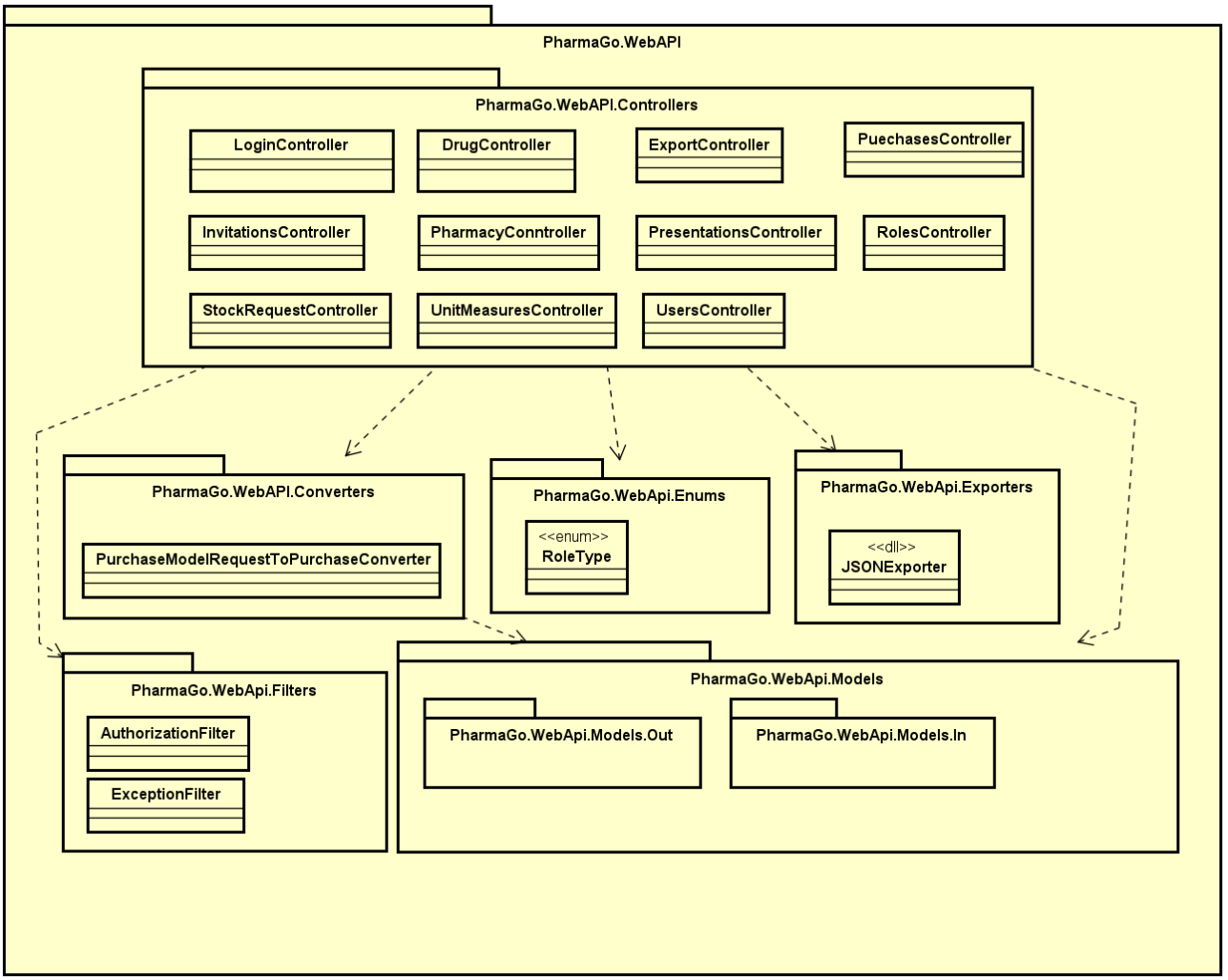


Ilustración 4: Diagrama del paquete PharmaGo.WebApi.

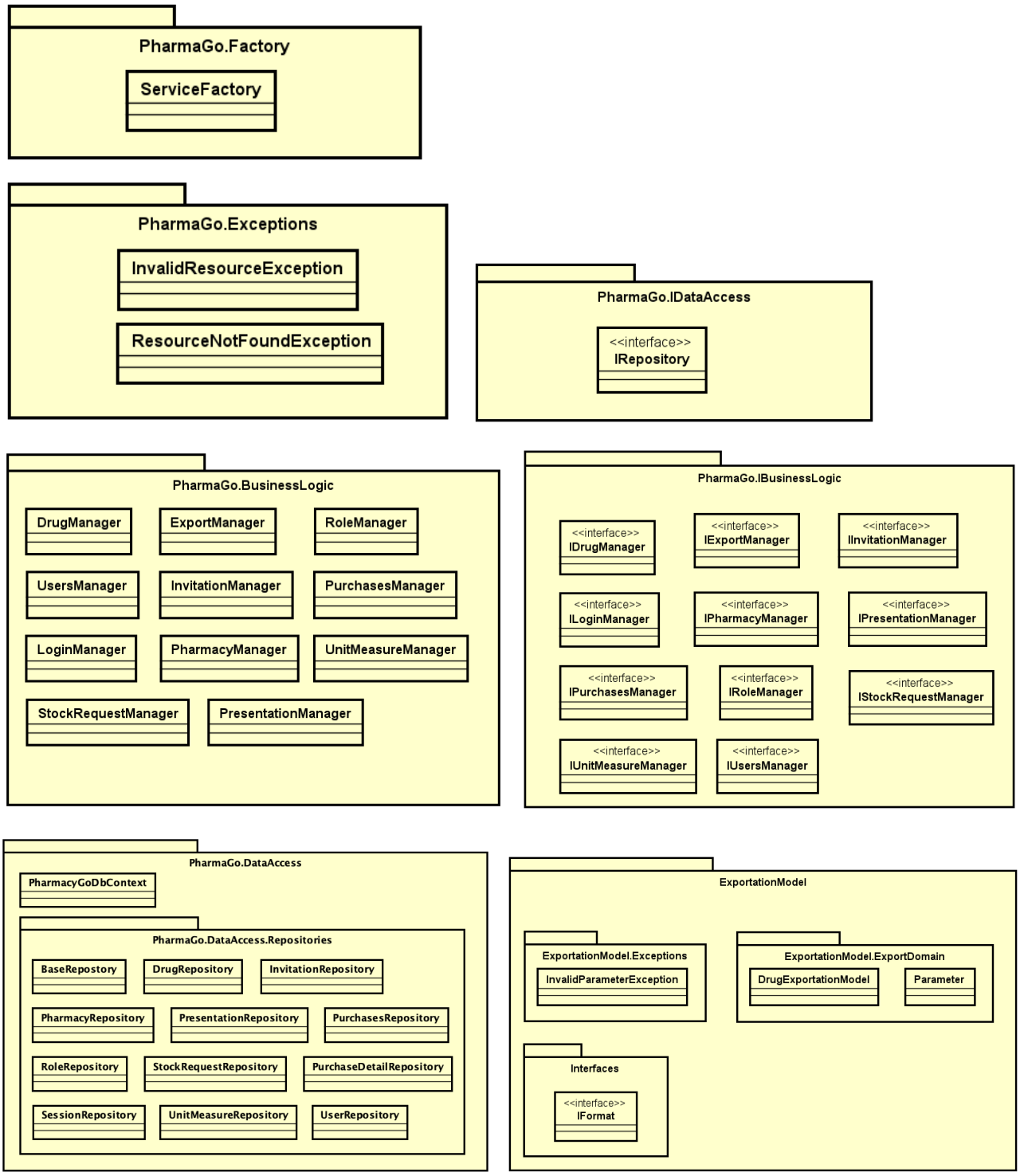


Ilustración 5: Diagramas del resto de los paquetes del sistema.

Diagrama de Paquetes Anidados.

En este diagrama de descomposición se puede ver la organización y jerarquía de paquetes (namespaces).

Ilustración 6: Diagrama de paquetes anidados.

Diagramas de secuencia

- Crear Invitación

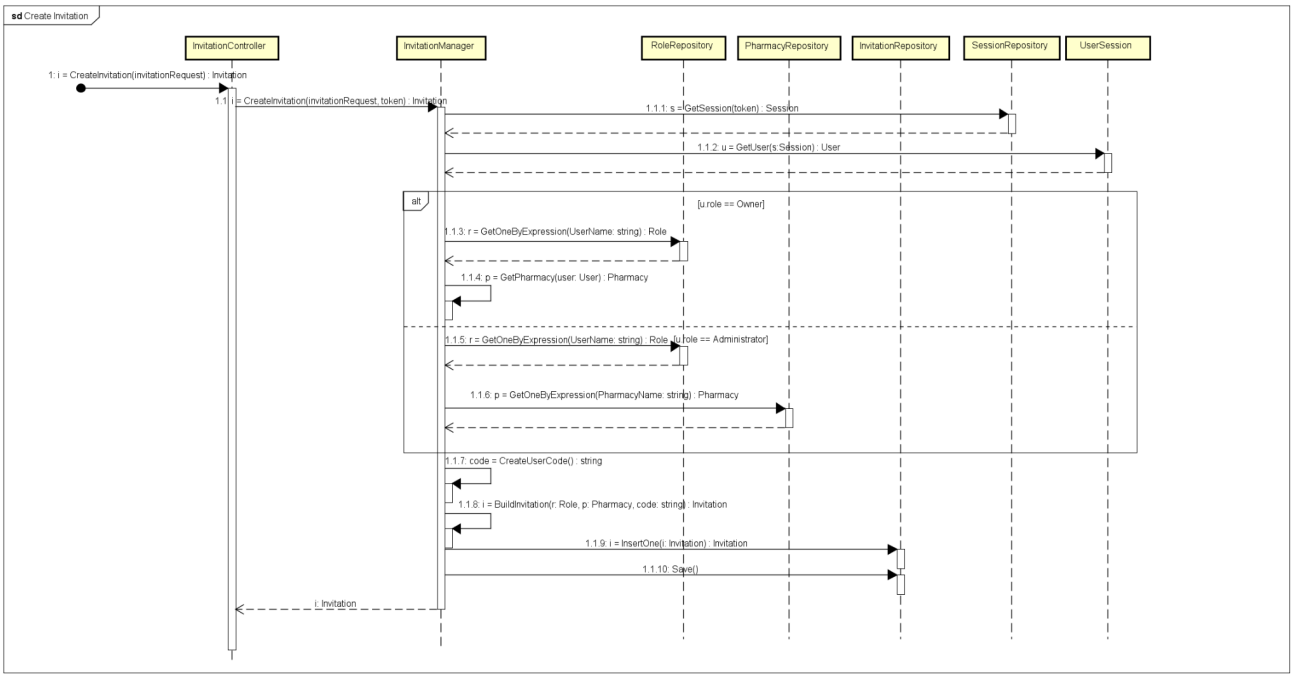


Ilustración 7: Diagrama de secuencia crear invitación.

- Crear Farmacia

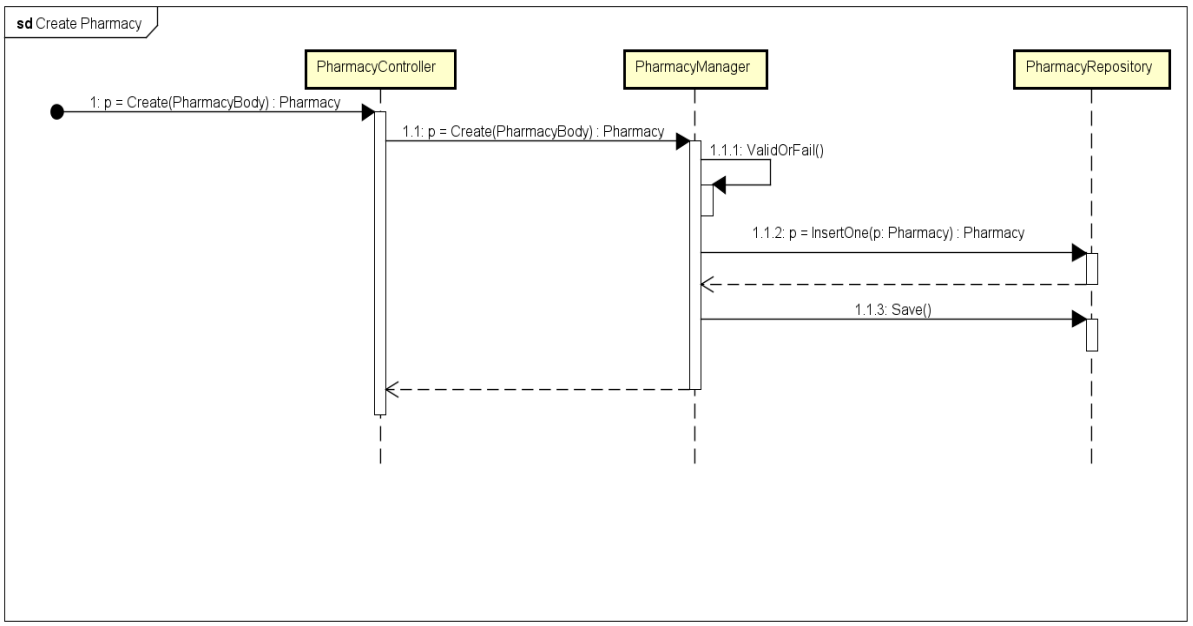


Ilustración 8: Diagrama de secuencia crear farmacia.

- Crear Compra

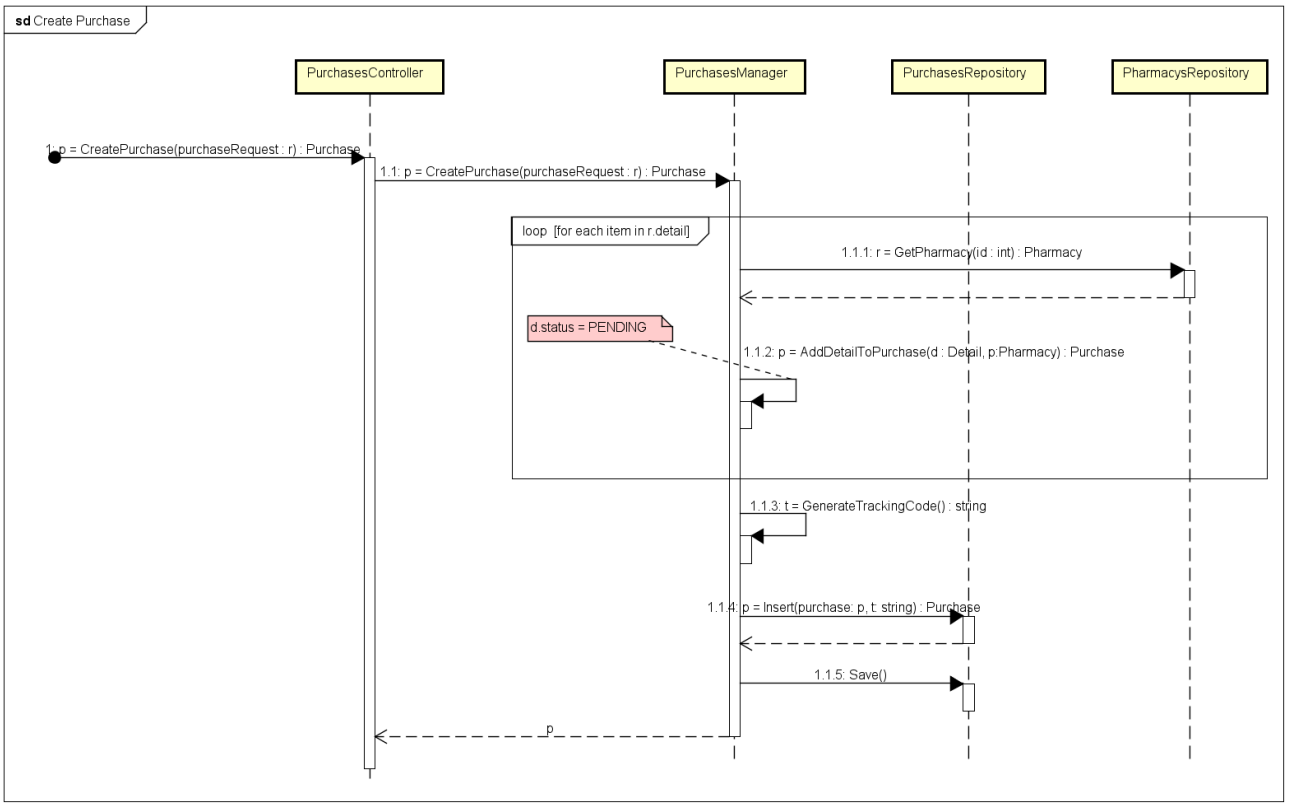


Ilustración 9: Diagrama de secuencia realizar compra.

- Crear solicitud de Stock

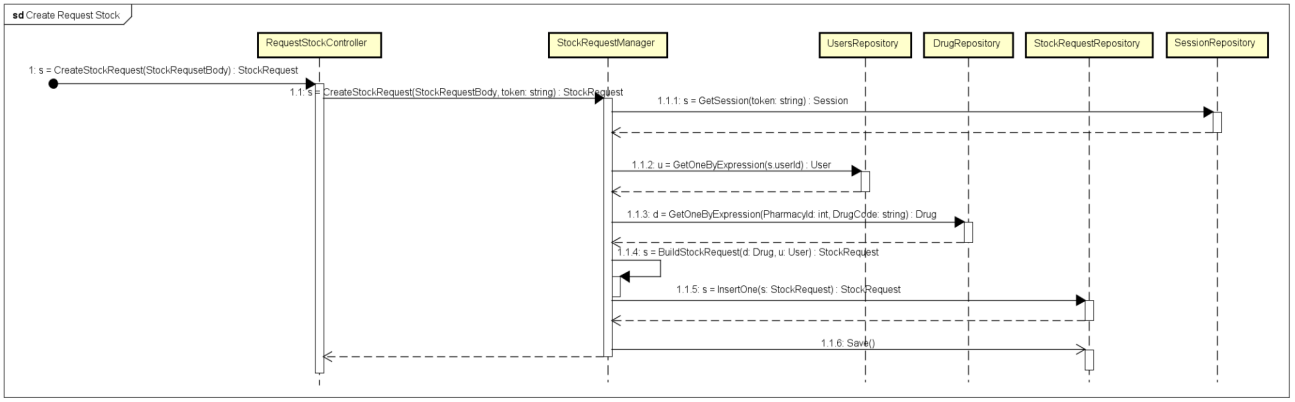


Ilustración 10: Diagrama de secuencia solicitud de reposición de stock de un medicamento.

- ## Crear Usuario

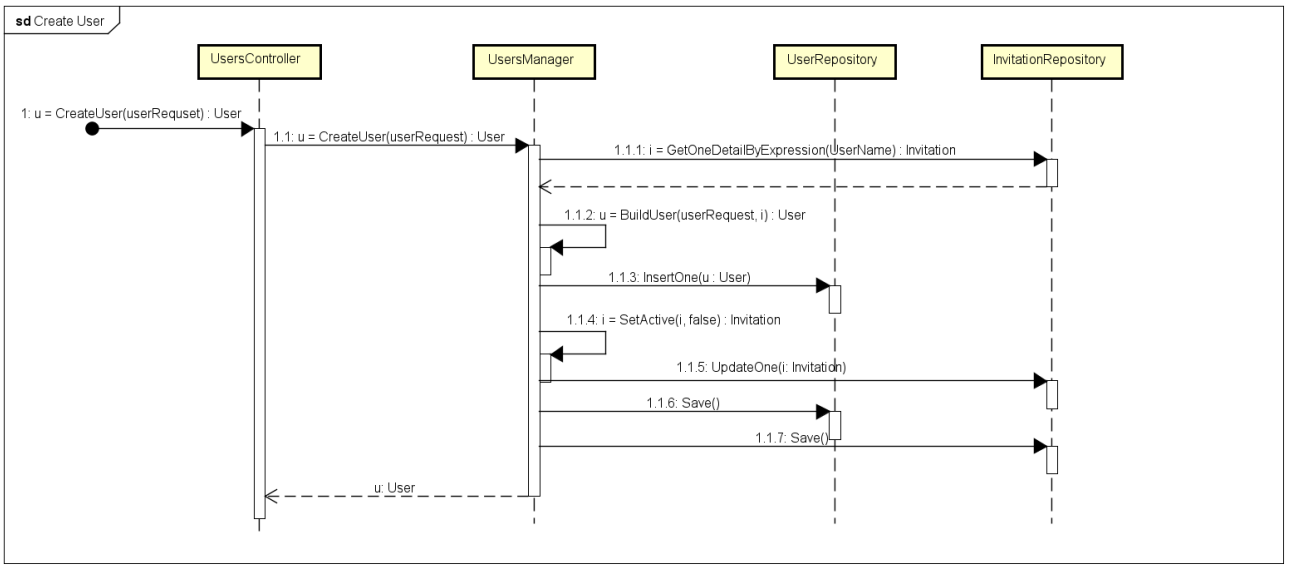


Ilustración 11: Diagrama de secuencia crear usuario.

- ## Login

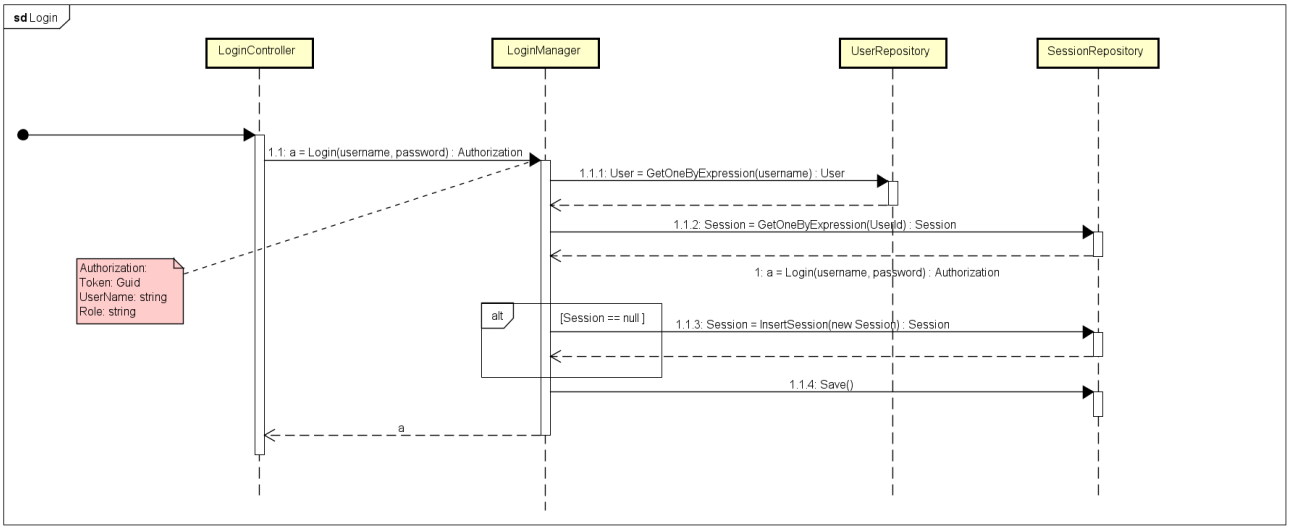


Ilustración 12: Diagrama de secuencia login.

- Detalle de compra por rango de fechas

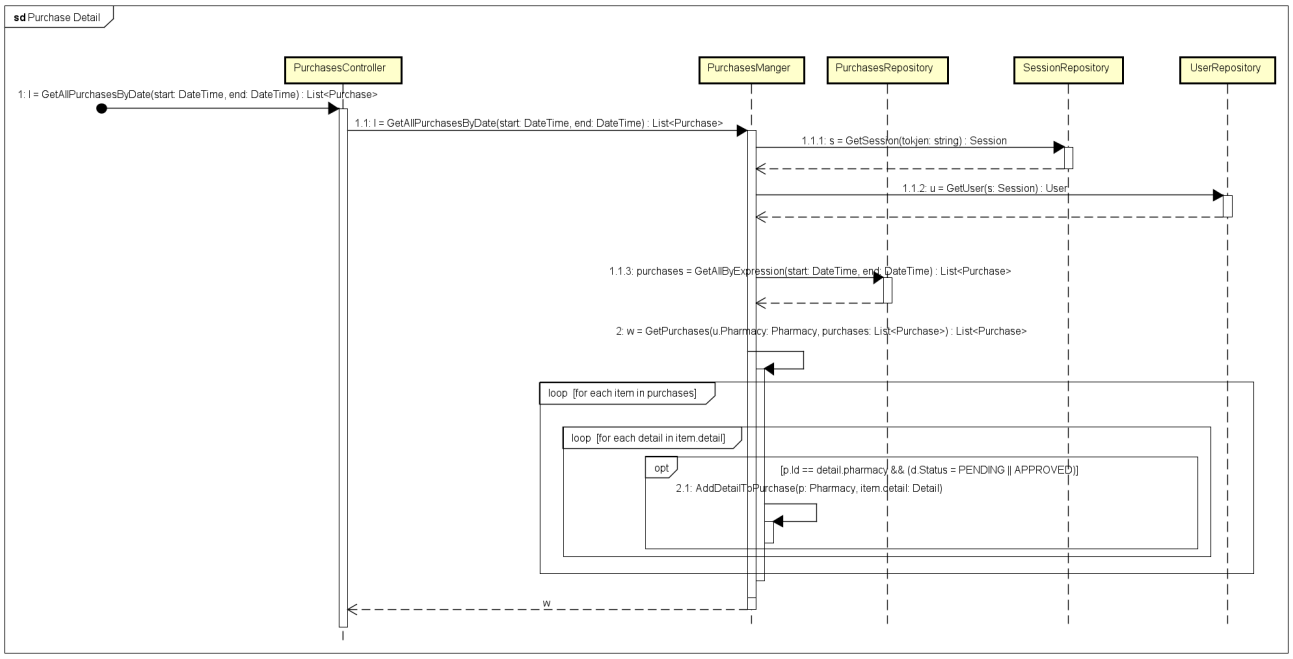


Ilustración 13: Diagrama de secuencia consulta compras realizadas en un rango de fechas dado.

Decisiones de Diseño

Para la realización de este obligatorio y el anterior, debimos tener en cuenta varios principios y patrones, ajustándose a las buenas prácticas de programación y a nuestros conocimientos para lograr un sistema flexible y abierto a la extensión.

- Encapsulamiento: agrupando dentro de una misma estructura elementos que se comportan de igual forma o que tendían a cambiar por el mismo motivo.
- Se puede entender a la api como la implementación del patrón Fachada.
- Dependencias acíclicas: se procuró que la dependencia entre paquetes formarán un grafo dirigido y acíclico.
- Information Hiding: los atributos de las clases son todos privados y solo se puede acceder a ellos mediante los métodos accesores.
- Modularización: se descompuso el sistema en varios subsistemas.
- Se decidió que el borrado de las Drugs sea lógico, de esta forma se podrá acceder a dicha información en caso de ser eliminado del sistema.

- Se decidió mantener el Precio de una Compra en PurchaseDetail, pensando en el caso de que a futuro se desee actualizar los precios de las Drugs, de esta forma el histórico de precios de la compra será inalterable.

- Se decidió que las Presentaciones como las Unidades de Medida se mantengan en su correspondiente tablas, de esta forma se puede insertar y borrar de forma lógica mediante consultas a la base de datos.

- Los diagramas indicados en el documento no cuentan con un detalle del 100% de las propiedades que pueden tener para mejorar la legibilidad en el documento.

Supuestos

StockRequest

Se supone que cuando se crea una solicitud de stock, la misma es creada con medicamentos existentes.

Aprobar una solicitud, puede demorar cierto tiempo en ser aprobada, en esa demora, es posible que un medicamento se elimine. Para tener presente este comportamiento, al momento de aprobar una solicitud, si hay algún medicamento eliminado en la misma, se muestra un mensaje de error.

PharmacyController y DrugController

Como extensión de la letra del obligatorio se decidió definir endpoints extra dentro de estos controladores para completar las operaciones CRUD sobre la tabla de la base de datos.

Por otro lado, el delete de Farmacia no es posible hacerlo en el caso de que la misma tenga medicamentos asociados. Se tomó esta decisión ya que el delete de una drug realiza un borrado lógico de la tupla, para poder seguir mostrándola en reportes futuros. Por lo tanto, el CascadeDelete no era una opción a la hora de borrar una Farmacia ya que borraría sus drugs.

Diagrama de componentes

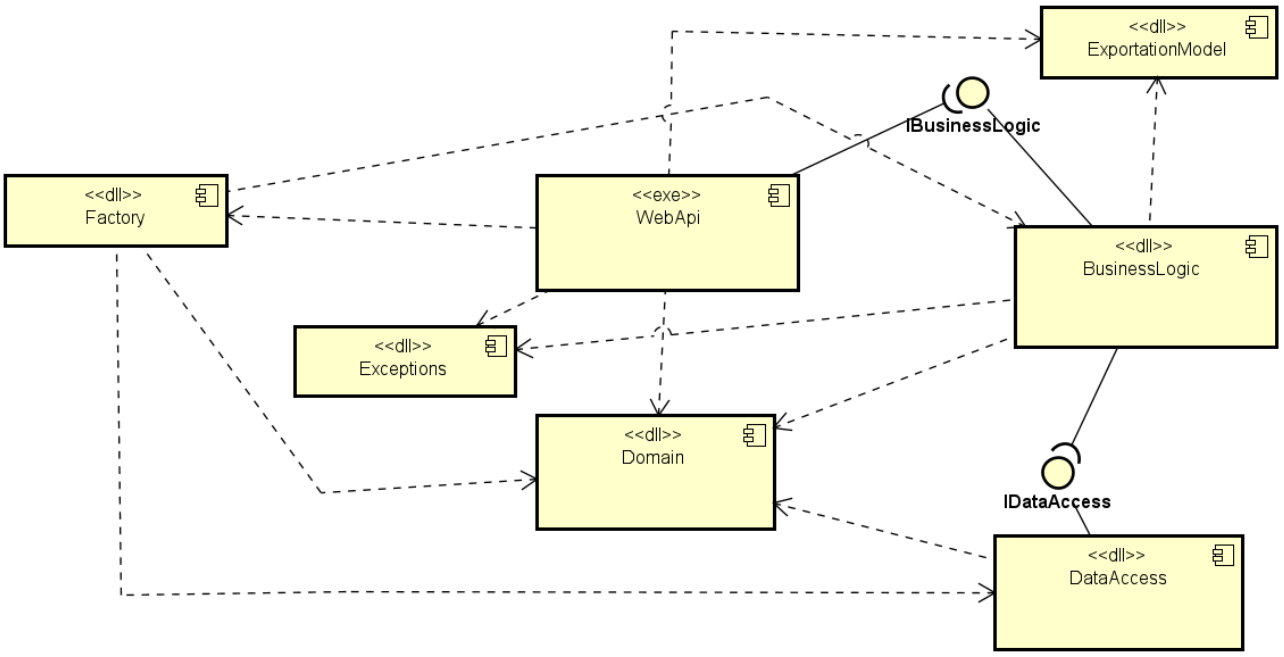


Ilustración 14: Diagrama de componentes.

Como se puede observar, a través de la aplicación desarrollada en Angular, se accede a la WebApi para traer las respuestas a las solicitudes del usuario, la misma a través de la interface IBusinessLogic provista por BusinessLogic accede a los servicios del mismo, para que estos a través de la interface IDataAccess, provista por DataAccess, extraiga los datos necesarios de la base de datos PharmaGoDb y los maneje de acuerdo a las necesidades del usuario y sus funcionalidades.

Esta división nos pareció la opción más correcta a desarrollar según los requerimientos, ya que tenemos una división de componentes clara, siendo lo más independiente posible uno de otro, dándole un enfoque importante a la mantenibilidad y teniendo fácilmente identificado las responsabilidades de cada uno, en el caso del Domain, se encargará de almacenar las entidades básicas requeridas para el funcionamiento de la aplicación.

DataAccess se encarga de almacenar, mantener y obtener los datos que se soliciten, en la BusinessLogic se encarga con dichos datos de realizar la lógica necesaria para que los requerimientos se cumplan, y a través de la capa de la WebApi proveemos a los usuarios acceso a las funciones del sistema.

Mecanismo de inyección de dependencias

La inyección de dependencias es un patrón de diseño que permite a PharmaGo romper el acoplamiento entre diferentes componentes. Para resolver la inyección de dependencias se creó un Proyecto PharmaGo.Factory que se encarga de resolver esto.

PharmaGo.Factory cuenta con una clase ServiceFactory que se encarga de registrar tanto las dependencias de la capa de negocios como la de acceso a datos:

- RegisterBusinessLogicServices: se encarga de registrar y asociar las interfaces IBussinesLogic con los Managers de BussinesLogic.
- RegisterDataAccessServices: se encarga de registrar y asociar las interfaces de cada repositorio IRepository de IDataAccess con los repositorios de DataAccess. Por otro lado se encarga de registrar el contexto de la base de datos PharmaGoDb obteniendo el connection string del archivo **appsettings.json.**

PharmaGo.WebApi se decidió que sea el proyecto ejecutable de inicio dentro de la solución, dentro del mismo se encuentra Program.cs en donde se llama a los métodos antes mencionados de la Factory para registrar las dependencias.

La inyección de dependencias nos permite un sistema extensible y que usa acoplamiento de interfaz y no de implementación, generando que este sea flexible a los cambios, que permita cambiar implementaciones en tiempo de ejecución y respetar los SOLID de diferentes formas.

Descripción del mecanismo de acceso a datos utilizado

Mediante 'CodeFirst' se definieron entidades en el proyecto **PharamaGo.Domain**, luego se definió el **PharmacyGoDbContext** en DataAccess, permitiendo así mediante el Entity Framework poder generar migrations y posteriormente generar la base de datos.

Las Migrations se encuentran en **PharmaGo.DataAccess.Migrations** y muestran la evolución del diseño a lo largo del tiempo y las decisiones que debieron tomarse de forma iterativa e incremental durante el desarrollo.

Se decidió utilizar un repositorio genérico **IRepository** con un tipo **T**, permitiendo utilizar las operaciones básicas como insertar, borrar, obtener y actualizar de forma sencilla y para todas las

entidades. Luego se definieron repositorios específicos en caso de ser necesario para consultas más complejas de las que el repositorio base podría ofrecer.

El desarrollo se realizó mediante el **Entity Framework Core 6.0.8** con **SQL Server Express Edition 14.0**.

Descripción del manejo de excepciones

Para el manejo de Excepciones se decidió crear un filtro ***ExceptionFilter*** encargado de capturar todas las excepciones que suceden en el sistema, estas se dividen en:

- **ResourceNotFoundException**: para capturar excepciones donde no se encuentra los recursos requeridos por las request del cliente, devolviendo un StatusCode de 404.
- **InvalidResourceException**: para capturar excepciones donde no se cumplen validaciones de los diferentes datos recibidos por el request del cliente, devolviendo un StatusCode de 400.
- **FormatException**: para el caso específico de que el token recibido por el header Authorization no cumple con el formato Guid, devolviendo un StatusCode de 400.
- **Exception**: para capturar el resto de las posibles excepciones, devolviendo un StatusCode de 500.

El formato de todas las respuestas de error devuelve un objeto JSON que tiene una key “message” en donde se muestra el mensaje de error de la excepción.

Las excepciones **ResourceNotFoundException** y **InvalidResourceException** son dos excepciones custom que heredan de **Exception** y cuentan con su propio proyecto PharmaGo.Exceptions dentro de la solución.

Dentro de Program.cs en el Proyecto PharmaGo.WebApi se agrega este filtro para que sea tenido en cuenta dentro del sistema.

Se decidió también para esta segunda entrega agregar una excepción específica para la funcionalidad de exportar medicamentos. La misma se llama InvalidParameterException y se encuentra dentro de la carpeta Exceptions dentro del proyecto ExportationModel.

Exportación de medicamentos

A continuación, se detallan los proyectos y clases creadas junto con su responsabilidad para exportar medicamentos del sistema.

Clases y proyectos definidos

ExportController

Este controlador fue creado con el objetivo de exponer los recursos necesarios para exportar medicamentos.

Nombre Método	**Responsabilidad**
GetAllExporters()	Exponer una lista con los nombres de cada tipo de exportador proporcionado por terceros.
GetParameters([FromQuery] string exporterName)	Exponer una lista de parámetros de un exportador específico.
ExportDrugs([FromBody] DrugsExportationModel drugExportationModel)	Exponer el servicio de exportación de medicamentos recibiendo un tipo de exportador con sus parámetros necesarios para exportar medicamentos.

IExportManager

Define la firma de los 3 métodos necesarios que se llaman dentro del ExportController y se implementan en ExportManager.

ExportManager

Es el encargado de leer las librerías de terceros y desarrollar la funcionalidad de exportar medicamentos mediante reflection.

Nombre Método	**Responsabilidad**
GetAllExporters()	Lee en tiempo de ejecución distintos tipos de exportadores provistos por terceros y devuelve una lista con los nombres de cada exportador.
GetParameters([FromQuery] string exporterName)	Lee en tiempo de ejecución distintos tipos de exportadores provistos por terceros y devuelve una lista con los parámetros que necesita cada exportador para exportar medicamentos.

ExportDrugs([FromBody] DrugsExportationModel drugExportationModel)	Lee en tiempo de ejecución distintos tipos de exportadores provistos por terceros y exporta medicamentos según el tipo de formato y parámetros indicados si los datos proporcionados son correctos.
ReadDlls()	Es el encargado de leer las .dll provistas por terceros. Dentro de este método se utiliza la librería Reflection para leer en tiempo de ejecución tipos de exportadores.

ExportationModel

Se creó este proyecto para definir el modelo que los desarrolladores de terceros deben conocer para que nuestro sistema pueda interpretarlos. De esta manera hacemos que los terceros conozcan solamente 1 proyecto de nuestra solución y no varios. A su vez, las clases mencionadas anteriormente también van a conocer este proyecto para poder modelar la funcionalidad provista.
Este modelo contiene la interfaz **IFormat**, la cual va a ser implementada por cada .dll de terceros que defina un tipo de exportación. Con esto logramos abstraernos de cada implementación específica de exportación y a través de la misma y reflection, se puede conocer todas las .dll existentes en tiempo de ejecución.
También se definieron las clases **DrugExportationModel** y **Parameter**, con el objetivo de que el back end y las implementaciones de terceros compartan un modelo y puedan comunicarse entre sí.
Por último se creó también una Excepción específica **InvalidParameterException**, la cual se lanza cuando un parámetro necesario para la exportación que fue completado por un empleado no es correcto.
Otra decisión de diseño implementada fue lograr que cada exportador devuelva una lista de Parameter y no solamente uno. En el caso del **JSONExporter** implementado, este solo requiere de un path para poder exportar, pero nuestro sistema cumple con OCP, ya que si se quisiera exportar hacia una base de datos por ejemplo, en vez de recibir una lista de Parameters con un solo Parameter path, se recibirán más datos como por ejemplo el nombre de la base de datos, conexión y puerto.
Como una mejora a futuro, se podría haber definido un enum para validar el tipo de parámetros definidos, ya que en esta entrega solamente se manejan input strings. En el caso de implementar esta mejora, se podrían utilizar checkButtons o radioButtons por ejemplo que se validan e interpretan según el tipo de datos que tenga el Parameter.
También nos gustaría aclarar que el nombrado de este proyecto debería haber sido PharmaGo.ExportationModel para continuar con la nomenclatura utilizada, pero tras inconvenientes al intentar renombrarlo decidimos documentarlo y continuar con el desarrollo del obligatorio.

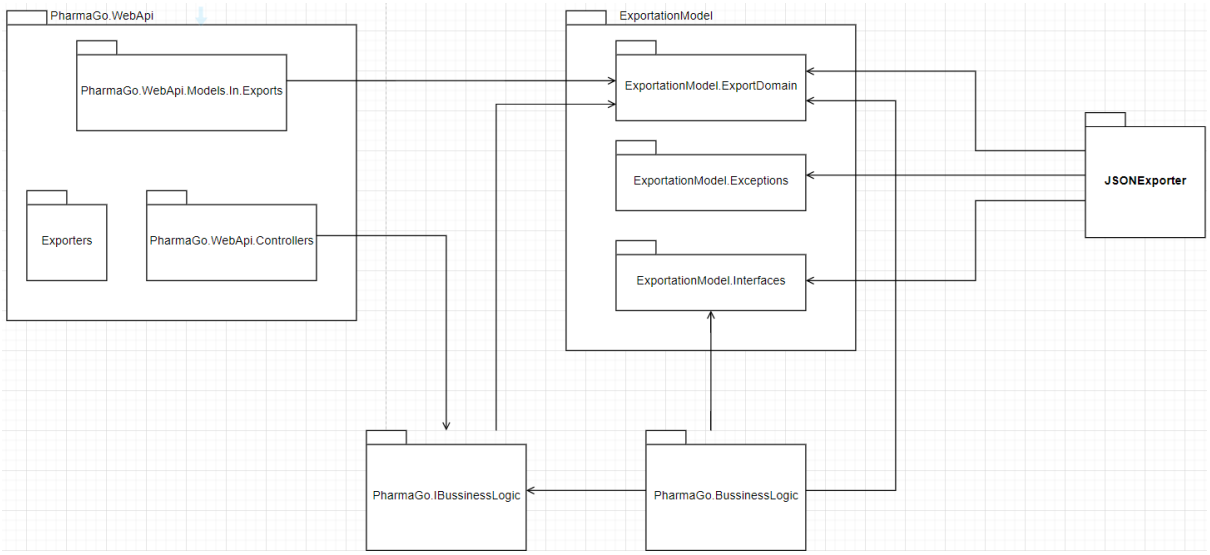
Flujo de la funcionalidad

En primer lugar, desde el front se le “pega” al endpoint **GetAllExporters()** para brindarle al empleado los nombres de los exportadores para que pueda seleccionar el tipo de exportador requerido.
En segundo lugar, con el tipo de exportador que el empleado seleccionó se consulta a través del endpoint **GetParameters([FromQuery] string exporterName)** por los parámetros necesarios que el

empleado debe completar para que se puedan exportar medicamentos en ese formato. Al obtenerlos, el front despliega una vista para que el empleado complete los mismos.
Por último, una vez que el empleado llena los datos a completar y selecciona el botón Export, se llama al método **ExportDrugs([FromBody] DrugsExportationModel drugExportationModel)**, el cual recibe la información y exporta medicamentos en el formato indicado utilizando la .dll externa indicada.
Para obtener los medicamentos, se implementó el método **GetDrugsToExport(string token)** en **DrugManager**, el cual recorre el repositorio de medicamentos y devuelve una lista solamente con los medicamentos de la farmacia del empleado logueado. A su vez, se restringe a que solamente un usuario con rol empleado pueda exportar medicamentos.

Diagramas

Diagramas de paquetes



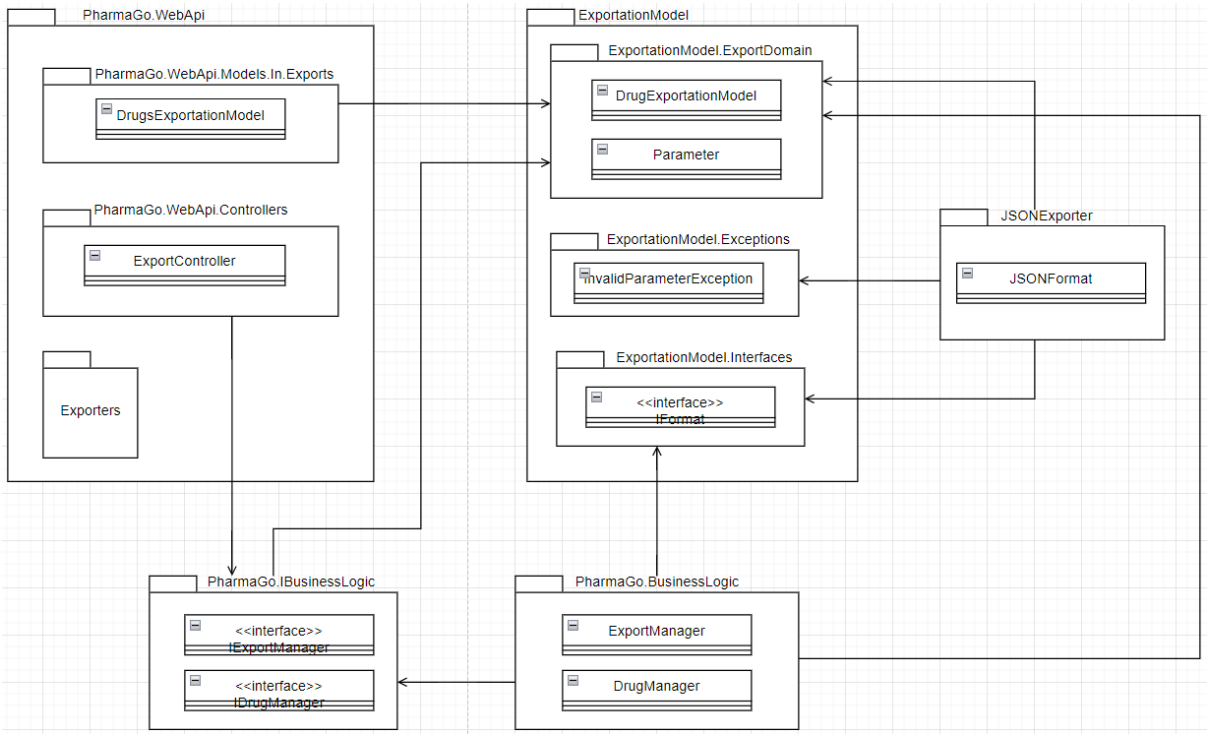
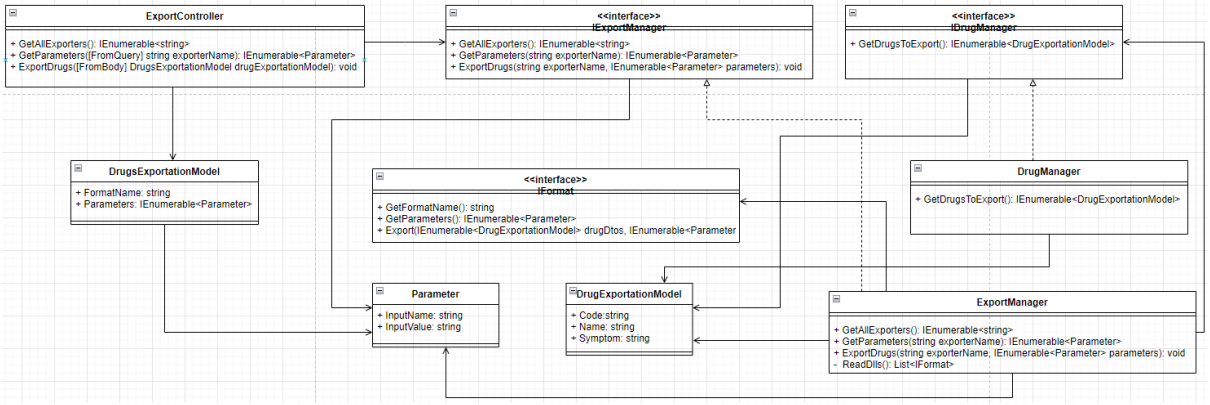
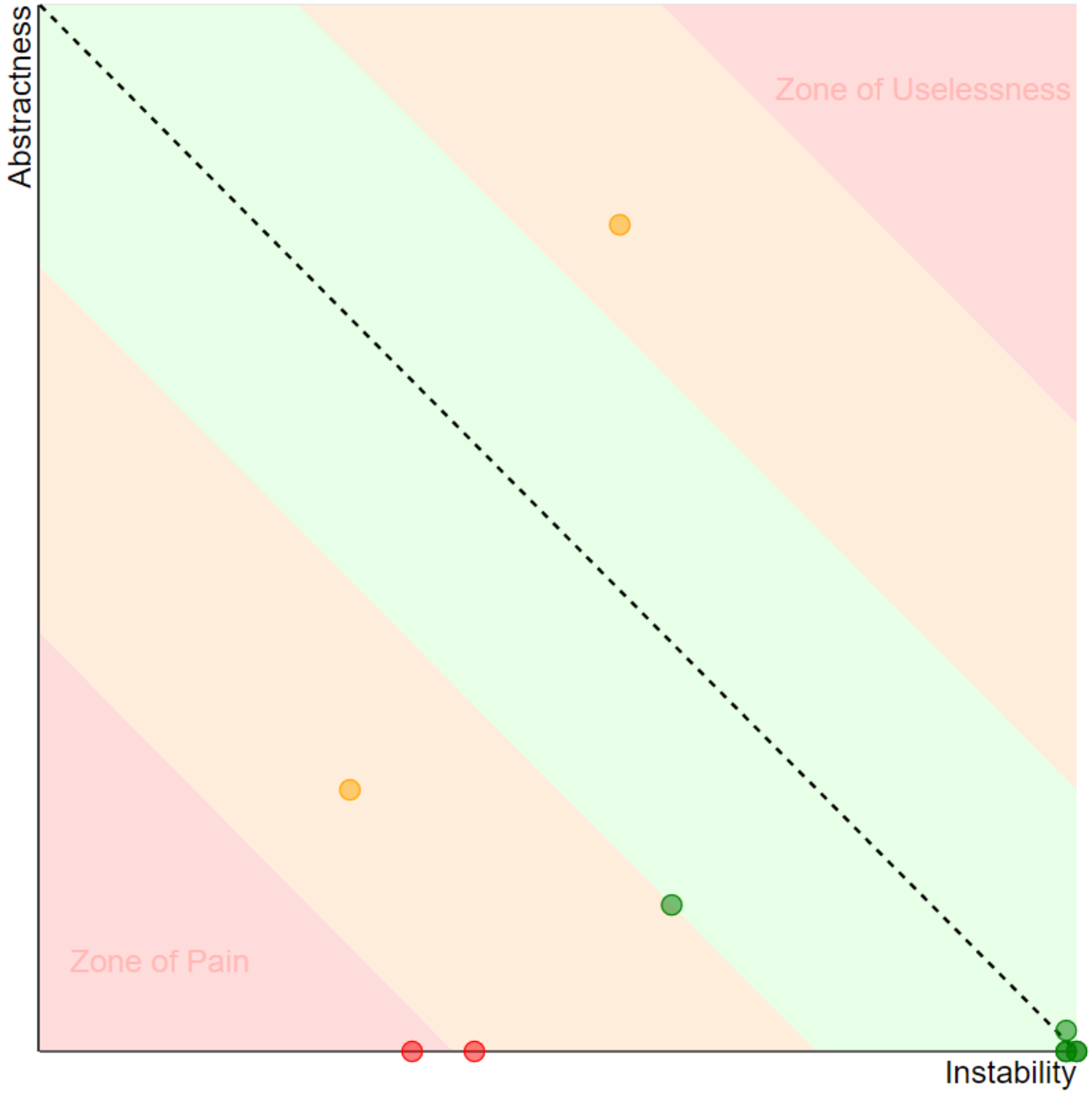


Diagrama de clases



Métricas de diseño.

Tras analizar las métricas con NDepend, se obtuvo la siguiente gráfica de abstracción vs inestabilidad:

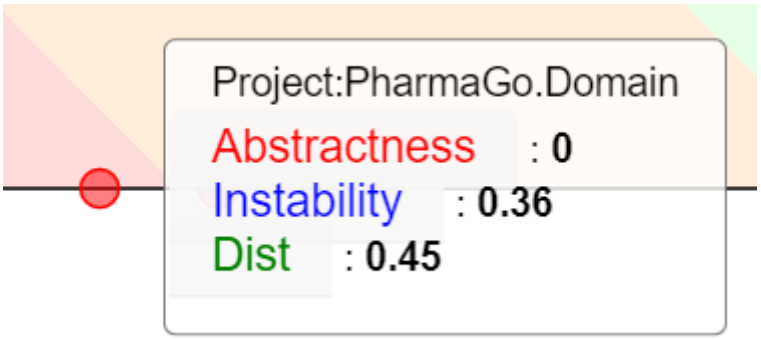


En relación al acoplamiento y dependencias, creemos que el obligatorio está en un muy buen estado, extenderlo fue sencillo y los cambios que hubo que hacer no fueron dolorosos de llevar a cabo.

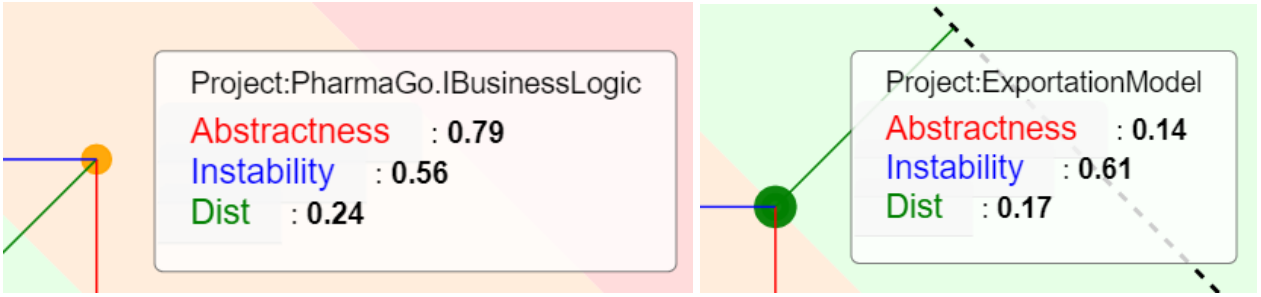
De la gráfica surge que el único punto que figura en rojo es el paquete del dominio, que se encuentra en la zona de dolor, una zona de mucha estabilidad y poca abstracción, estos paquetes no son buenos

porque no son extensibles y si cambian impactan en otros, pero si lo analizamos con detenimiento tiene sentido que se encuentre en esa ubicación de la gráfica, ya que el resto de las clases del sistema depende del Dominio y esta no depende de ninguno de ellos, por lo que debería ser uno de los paquetes menos propensos a cambiar si no se quiere impactar en todo el sistema.

También es un paquete poco abstracto, ya que en él se encuentran las clases concretas de la solución.



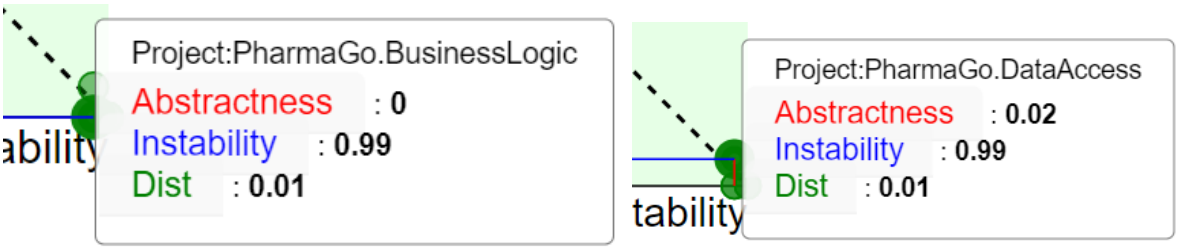
ExportationModel se encuentra en el límite entre la zona verde y la zona anaranjada inferior, si bien no se encuentra en la zona de dolor, se debe le debe prestar atención, IBusinessLogic se encuentra en la zona anaranjada superior con una inestabilidad de 0.56 la cual depende de ExportationModel con una inestabilidad de 0.61.



Si bien la diferencia es muy poca, podemos decir que en este caso no se está cumpliendo con el principio de dependencias estables (SDP), por lo cual si se quiere cumplir de forma estricta con este principio, en el futuro, quien mantenga el sistema debería modificar ExporationModel para que sea más estable, por ejemplo separando los modelos y llevándolos a otro paquete o creando uno nuevo específico para ellos.

En el caso de este trabajo obligatorio al no ser la diferencia de estabilidad muy grande se decidió no hacer modificaciones y detallar en la sección pertinente.

Por otro lado, en el cuadrante inferior derecho tenemos los paquetes que implementan las interfaces de servicios, el paquete BusinessLogic y DataAccess.



Esto se deba a que son paquetes muy concretos (implementan los contratos ofrecidos en las interfaces) e inestables (porque dependen de más paquetes de los que dependen ellos) y van a tender a cambiar por cualquier cambio que se produzca o bien en la tecnología o en los requerimientos del producto, pero por sus valores de abstracción e inestabilidad se convierten en paquete muy extensibles, ya que sus cambios no afectarían o en caso de hacerlo a muy pocos paquetes.

Considerando que la métrica de Abstracción (A) se calcula como:

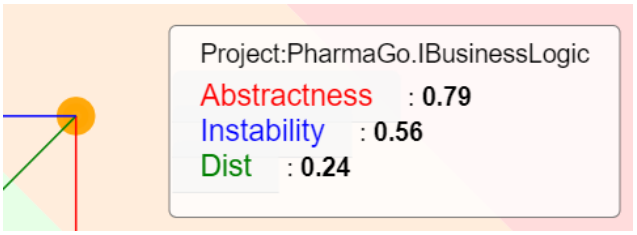
Na = Cantidad de clases abstractas e interfaces en el paquete

Nc = Cantidad de clases abstractas e interfaces más las clases concretas (todas las clases del paquete)

A = Na/Nc

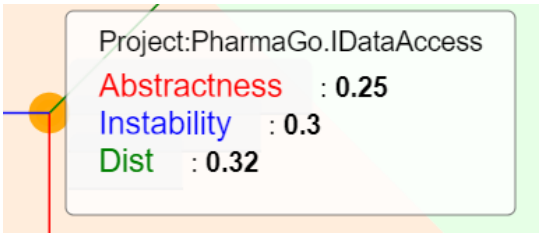
Para **PharmaGo.IBusinessLogic** tenemos: Na = 11, Nc = 11 => A = Na/Nc = **1**

Lo que contradice el resultado de NDepend donde calcula: **0,79**



Para **PharmaGo.IDataAccess** tenemos: Na = 1, Nc = 1 => A = Na/Nc = **1**

Lo que contradice el resultado de NDepend donde calcula: **0,25**



Por lo tanto, no le encontramos sentido a los resultados de NDepend para estos paquetes, ya que al ser ambos solo compuestos por interfaces la abstracción debería ser de 1.

El paquete WebApi tiene una baja abstracción y es inestable, lo que lo sitúa en la secuencia principal, lo que indica que es extensible y que pocos dependen de él.

Flujo de trabajo

Se utilizó durante todo el desarrollo el flujo de trabajo de GitFlow para gestionar las ramas de Github. De esta forma se creó un rama ‘main’, una rama ‘develop’ y múltiples ramas ‘feature/nombre-del-feature’ siguiendo rigurosamente los siguiente:

- Nunca se hace commit directamente a ‘develop’.
- Por cada nueva característica se crea una nueva rama ‘feature’.
- Las ramas ‘feature’ se mergean a ‘develop’ solo por PR.
- Al finalizar el desarrollo se mergea ‘develop’ en ‘main’ y se crea un tag de la versión.

Diseño del Front-end

Estructura del proyecto

Se creó una aplicación Angular de 0 utilizando **Angular CLI 14.2.6**, **Node 14.17.6** y **NPM 8.19.2**. Y se trabajó con **Visual Studio Code 1.73.0**. Partiendo de esto se decidió seguir la siguiente organización:

```
+---app
ª   +---custom
ª   +---guards
ª   +---interfaces
ª   +---pages
ª   ª   +---admin
ª   ª   +---employee
ª   ª   +---home
ª   ª   +---login
ª   ª   +---owner
ª   ª   +---register
ª   +---services
ª   +---utils
```

Dentro de 'app' se decidió crear una carpeta 'services' para agrupar los diferentes servicios al backend. Se optó por crear un servicio por controlador del backend, es decir, para el 'DrugController' un servicio 'drug.service', para 'LoginController' un servicio 'login.service' y así con el resto de los controladores. Cada servicio tiene un handler de errores que dispara una alerta (componente Alerta) de color rojo indicando el mensaje de error del backend. A su vez, en cada servicio se obtiene del Storage Manager el token de autorización y se setea en los headers de la request (header "Authorization"). En caso de que no sea necesario, por ejemplo para los endpoints que utilizan los usuarios anónimos, se envía vacío.

Se creo una carpeta 'guards' con la guarda de autenticación "AuthenticationGuard" que se encarga de validar el rol del usuario y en caso de no tener permisos de acceso a la ruta que se desea ingresar se re-direcciona a **"/unauthorized"** indicando un 401.

Luego una carpeta de 'utils' con la idea de mantener clases utilitarias a todo el proyecto, especialmente el "StorageManager" que se encarga de encapsular el acceso al local storage.

Se cuenta con una carpeta 'interfaces' que sirve para declarar las diferentes interfaces o clases que se utilizan como modelos para tomar las respuestas o armar los requests hacia el backend.

Una carpeta 'custom' que contiene componentes creados por nosotros que son reutilizables en diferentes partes del front-end, como lo pueden ser las headers o footers en la page **"/home"**

En la carpeta 'pages' se centralizan los componentes que definen las páginas a mostrar a los usuarios, siendo estas organizadas por:

Carpeta **home** para los Usuarios Anónimos:

```
/home
/home/cart
/home/cart/cho
/home/detail/:id
/home/tracking
```

```
/login
/register
/unauthorized
'**' redirecciona a 404
```

Carpeta **admin** para los usuarios de rol Administrador:

```
/admin
/admin/list-invitation
/admin/update-invitation
/admin/create-invitation
/admin/create-pharmacy
```

Carpeta **employee** para los usuarios de rol Empleado:

```
/employee
/employee/purchase-status
/employee/delete-drug
/employee/create-drug
/employee/stock-request
/employee/create-request
/employee/export-drugs
```

Carpeta **owner** para los usuarios de rol Dueño:

```
/owner
/owner/invitation
/owner/stock-request
/owner/purchase-by-date
```

Por otro lado, también se utilizó la carpeta de 'environments' que contienen los archivos de configuración del entorno para establecer la URL del backend y la carpeta 'assets' para manejar el contenido estático como lo puede ser la imagen de "***not_found***".

Para más detalles de las decisiones tomadas en el front end *Ver Anexo 3 - Diseño del Front-end*

Anexos

Anexo 1 - Evidencia del diseño y especificación de la API

Anexo 2 - Informe de Cobertura de las Pruebas

Anexo 3 - Diseño del Front-end