Universidad ORT Uruguay

Obligatorio 2

Diseño de Aplicaciones 2

Documentación

Martin Edelman - 263630

Tatiana Poznanski - 221056

Tomas Bañales - 239825

Profesores: Nicolás Fierro, Alexander Wieler, Marco Fiorito

2023

Link al repositorio:

https://github.com/IngSoft-DA2-2023-2/263630_221056_239825.git

1. Link al repositorio

https://github.com/IngSoft-DA2-2023-2/263630_221056_239825.git

2. Declaración de autoría

Nosotros, Tatiana Poznanski, Martin Edelman y Tomas Bañales declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

1. La obra fue producida en su totalidad mientras cursamos Marketing de Tecnologías.
2. Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad.
3. Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra.
4. En la obra, hemos acusado recibo de las ayudas recibidas.
5. Cuando la obra se basa en el trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y que fue construido por nosotros.
6. Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

3. Abstract

El proyecto fue desarrollado en Visual Studio 2022, más precisamente en el lenguaje C#. Durante toda la implementación, se aplicaron las prácticas de TDD (Test Driven Development) y Clean Code. Asimismo, utilizamos Microsoft SQL Server Express 2019 para almacenar los datos de la aplicación.

Con el fin de agregar nuevas funcionalidades al código, se emplearon las herramientas de GitHub y se siguió un flujo de trabajo de git. A medida que se deseaba trabajar en nuevas características, se fueron creando ramas (branches), una vez completadas, las mismas se fusionaban con la rama principal llamada "develop" para mantener el código actualizado.

Asimismo, identificamos tres componentes clave en el código y aplicamos la metodología GitFlow para dividir el trabajo de manera efectiva. La asignación de tareas se llevó a cabo de la siguiente manera:

- Martin Edelman, junto con Tatiana, se encargó de la UI, principalmente del desarrollo de los usuarios, centrándose en el perfil y los administradores. A su vez, llevó a cabo ciertos arreglos necesarios en los endpoints del backend.
- Tatiana Poznanski, junto con Martin, se encargó de la UI, principalmente de todas las funcionalidades del carrito y productos, concluyendo en compras exitosas, así como el filtrado de productos en base a diversas características.
- Tomás Bañales se encargó de llevar a cabo el reflection, gestionó y supervisó el manejo del stock de productos y trabajó en habilitar las promociones.

4. Descripción general del sistema

Desarrollamos un ecommerce para tiendas de moda en Uruguay que aborda la implementación de promociones, el cual le permite a los usuarios comprar productos a voluntad. Este sistema incluye una funcionalidad dinámica de promociones, la cual permite a la persona que tenga acceso al mantenimiento del sistema el activar o descartar promociones a ser usadas en las compras en tiempo real. Cada usuario puede realizar compras, y en estas seleccionar el método de pago a utilizar, dependiendo de cuál se use, como se verá afectado el precio (la pagina tiene convenio con Paganza, por un 10% extra de descuento; más allá de las promociones automáticas, aplicadas siempre y seleccionada la que el menor precio proporcione para la compra en curso). En la versión actual del sistema se incluye un módulo de front-end, creado con Angular. Este completa la conexión entre la Api, de la primera entrega, y el usuario final del proyecto.

Para el desarrollo de nuestro sistema hicimos uso de Entity Framework Core junto con la interacción de SQL Server para la base de datos y ASP .Net para la aplicación de las APIs.

Con el fin de reducir el acoplamiento y aumentar la cohesión, implementamos 4 módulos principales (Api, Servicios, DataAccess, Dominio), los cuales contribuyen significativamente a la aplicación, y otros 3, encargados de las pruebas unitarias, la inyección de dependencias y la migración de datos a la base de datos.

4.1 Instalación

La instalación del sistema se logra instalando una serie de paquetes de NuGet y configurando el connection string dentro del appsettings.json.

Los paquetes a instalar son los siguientes:

- Microsoft.AspNetCore.Authentication.JwtBearer

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Design
- Microsoft.EntityFrameworkCore.InMemory
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools
- Microsoft.Extensions.DependencyInjection
- MSTest.TestFramework
- Swashbuckle.AspNetCore
- Swashbuckle.AspNetCore.Swagger
- Newtonsoft.Json

Después de instalar los paquetes, procedemos a configurar el appsettings.json en el proyecto de API y DataAccess, en estos archivos se agrega el connection string en el apartado de EcommerceDB.

De esta manera, estamos realizando el connection string en un archivo de configuración, lo que nos permite cambiar la conexión en tiempo de ejecución.

Además, se debe actualizar la migration. Esto se logra en la terminal del proyecto, entrando al proyecto de migrations y ejecutando el comando:

$ dotnet ef update -p ../DataAccess

4.2 Justificaciones de diseño

A continuación detallaremos, en cada proyecto, las justificaciones de diseño.

En primer lugar, en Dominio existen dos subpaquetes: el de Producto y el de Usuario; siendo el subpaquete Compra la unión. El de Usuario, naturalmente, es en el que se encuentra la clase Usuario.cs, donde el id y el correo electrónico son valores únicos.

Además, dicha clase contiene un valor rol de entidad, “CategoriaRol”, el cual indica si el usuario es Administrador, Cliente o ambas. Dentro del mismo paquete, se encuentra el Enum de roles en CategoriaRol.cs.

En el subpaquete de Compra, se decidió establecer la entidad Compra con vínculos a Producto, Usuario y MetodoDePago, que, análogamente al paquete anterior, es un enum usado en la clase compra.

En el subpaquete de Producto se encuentran la clase Producto y sus clases asociadas; Color, Categoría, y Marca. Este paquete, directo como es, engloba lo que es un producto completo en el sistema, con sus vínculos y valores.

Tanto en los paquetes Servicio como Repositorio se diseñaron interfaces para que cada una de sus clases implemente. Esto permite seguir los principios DIP e ISP de SOLID, ya que en cada inyección de dependencias se apunta a depender de la interfaz, en vez de centrarse en la instancia (DIP); y se segregaron las interfaces por función, en vez de crear una interfaz general que abarca muchas “áreas”, por así decirlo (ISP).

Con respecto a la base de datos, se modificó el relacionamiento de la clase Producto con Color, pasando este a ser 1 a N, en ese orden, junto con las ya establecidas para Compras, Marca y Categoría.

En DataAccess se tienen dos subpaquetes: Interfaces; para la creación de la interfaz del data access de Usuario, Producto y Compra, por si en un futuro se desea modificar la base de datos o guardar datos localmente; y Migraciones.

Este último, es en el que se encuentran todas las migraciones de la base de datos alguna vez realizadas, cada vez que se cambia una entidad, se agrega una nueva migración. Asimismo, en el proyecto se plantea la implementación del contexto en ECommerceContext.cs para la creación del modelo de la base de datos en Entity Framework 6, utilizando la estrategia de Code First. El acceso a la base de datos se hace en los archivos RepositorioUsuario, RepositorioCompra y RepositorioProducto, accediendo estos ultimos dos a la base de datos sin ninguna lógica de negocio, simplemente retornando listas o elementos específicos. A su vez, se conectan las

distintas entidades que se relacionan, como por ejemplo, Productos y Colores o Compras y Productos, que mantienen una relación 1 a N entre ellos.

Como intermedio entre el Data Access y la Api, creamos Servicios que maneja toda la lógica de negocio, como la validación de email y de la contraseña, para que el usuario sea válido.

Previo a la explicación de Api, cabe explicar ServicioFactory. Este servicio utiliza el patrón de diseño Factory y, tiene como objetivo, tanto la creación de una clase que inyecta dependencias al inicio de la aplicación y se mantiene a lo largo de la misma, así como la indicación de qué Contexto utilizar de la base de datos, qué servicios de usuario y de producto, y qué data access emplear.

Api es instrumental en el sistema, es el que se ejecuta cuando se da inicio al programa. Este contiene DTOs, los cuales contribuyen a un uso adecuado de la API al momento de ingresar los datos y validarlos, pudiendo no mostrar todo lo que tiene la entidad, sino únicamente lo que interesa. Además, contiene filtros encargados de mantener el código más limpio, evitando las expresiones como try y catch. Estos filtros captan excepciones y las manejan devolviendo estatus de error como: 400, 401, 404 y 500.

Por último, se encuentran los controladores, cada uno con su propio paquete. Optamos por dividir los controladores en cuatro, uno por cada uri principal; siendo tres de estos usuarios, producto y compra. Por otra parte, decidimos crear un controlador solamente para el inicio de sesión, separado del de usuario, con el fin de bajar el acoplamiento del proyecto y cumplir con los principios de Single Responsibility Principle y Open/Closed Principle. Como esta clase utiliza aspectos de Json Web Token, utilizamos JWT dado que uno de los integrantes del proyecto tenía experiencia con el mismo, pero en otro lenguaje de programación. Hacer uso del JWT implicó evitar tablas extra que guarden los tokens, la información de login del usuario y permitió ahorrar espacio en la base de datos. Asimismo, compras también tiene un controlador dedicado, por más que los endpoints de esta no sean tan numerosos como los de las entidades “principales” Producto y Usuario. Esto se debe a que, a pesar de tener las funcionalidades de “Mostrar compras del usuario” en el controlador

de Usuario; justamente; la función de retornar todas las compras del sistema, reservada únicamente para los usuarios con permiso de administrador, no pertenece con las ya mencionadas.

Tanto el proyecto Pruebas como Migrations son ajenos a la ejecución del proyecto. Migrations, por su parte, se implementó ya que el proyecto Api está encargado de procesar las request, por lo que su responsabilidad es correr las migraciones, y asignarle otra responsabilidad relacionada a la tecnología EF Core rompería con SRP. Parecería lógico realizar esta tarea en DataAccess, pero surge la preocupación de que si esta no está vinculada al framework Net6, realizar migraciones podría volverse complicado en el futuro. Cambiar el framework de netstandard2.0 a netcoreapp3.1 en el archivo DataAccess.csproj permitiría ejecutar las migraciones ubicadas en ese directorio, pero esto presenta desafíos debido a las diferencias de estructura y tecnología.

4.3 Mejoras implementadas

4.3.1 Funcionalidades nuevas

Siguiendo los requisitos de la segunda entrega, se implementaron las siguientes funcionalidades nuevas:

> Se agregó una cuenta de la cantidad de productos restantes en el sistema al momento de realizar una compra. De no haber ninguno del modelo requerido, este no se agrega a la compra.

> Se agrega la posibilidad de que un Producto pueda permanecer exento de la lógica de promociones, cuando ésta sea aplicada, sumando su precio base al total de la compra sin modificación alguna.

> Se expandieron las opciones de los filtros de búsqueda a la hora de consultar productos en el sistema. En esta versión se puede optar por buscar productos dentro de un rango de precios.

> Se agregó la posibilidad de especificar el método de pago utilizado para la compra que se está creando. Habiendo un descuento del diez por ciento, aplicado

post-promociones, si se paga con Paganza. Esto es realizado mediante un enum por un tema de simpleza.

4.3.2 Mejoras al código existente

El área de Promociones se vio modificada extensivamente, ya que tanto para su arreglo como mejora se requirió prácticamente la re-creación de las clases enteras de cada una.

> Se optimizó la interfaz IPromocionStrategy; la cual aplica el patrón Strategy; y se modificaron tanto los métodos como su visibilidad. Esto principalmente se hizo para mejorar la legibilidad y mantenibilidad del código, ya que había métodos con nombres sumamente similares que tenían funciones distintas.

> Se organizaron las clases de promociones de manera que el método Aplicar Promoción es el único con visibilidad pública; aplicando así el patrón de Diseño Facade, ya que este lo que hace es coordinar los otros métodos privados para facilitar la interacción y simplificar el proceso. A esto se accede automáticamente desde ManejadorUsuario, al accionar el metodo “AgregarCompraAlUsuario”, mediante el método de ServicioCompra “DefinirMejorPrecio”.

Respecto a las API, modificamos algunos endpoints y creamos otros desde cero.

Todos los endpoints documentados a continuación están explicados más detalladamente con el output en el anexo.

Los endpoints que creamos desde 0 son:

- GET /api/v1/productos/colores
- GET /api/v1/productos/marcas
- GET /api/v1/productos/marcas

Modificamos los endpoints producto y autenticación, pero más específico en los parámetros que reciben en el body y lo que devuelve, ya que al producto se le agregó el stock y si acepta promociones, en autenticación, al body de respuesta se le agregó la información del usuario, anteriormente devolvía solamente el token de autenticación, pero nos dimos cuenta que era ineficiente porque luego en la API

íbamos a tener que realizar un GET Usuario/id y eso requería otro request. Nos pareció más eficiente centralizar todo en una sola request a la hora de iniciar sesión. Por último modificamos el modelo de la compra de respuesta ya que tiene información extra y cuando creamos la compra le agregamos el valor "metodoDePago"

Los siguientes endpoint fueron los modificados

- POST /api/v1/productos/{id}
- GET /api/v1/productos/{id}
- GET /api/v1/productos
- PUT /api/v1/productos/{id}
- GET /api/v1/compras
- GET /api/v1/usuarios/{id}/compras
- POST /api/v1/usuarios/{id}/compras

<+> Visualización gráfica de la aplicación del patrón Strategy en las promociones conocidas en la sección **Diagrama 5.2.3 - Servicios** del anexo

<+> Diagrama de secuencia del funcionamiento estándar de una clase Promocion cualquiera tras la aplicación del patrón facade dentro de una promoción cualquiera de entre las conocidas a continuación. Ver en la seccion **Diagrama 4.3.2 - Patrón Facade** del anexo.

4.4 REST aplicado

Se adoptó una arquitectura basada en REST, específicamente el modelo cliente-servidor. Esto proporciona flexibilidad y facilita la adaptabilidad del sistema a cambios futuros. La interacción con el servidor se realiza a través de un punto de entrada (endpoint), donde se envía información y se espera una respuesta, todo gestionado mediante comunicación HTTP y aprovechando operaciones estándar como Get, Put, Post y Delete.

Dentro del Api, compuesto por Filters, Controllers y DTOs, destaca su función como el canal de comunicación clave entre el frontend y el backend del proyecto.

Los controllers utilizan modelos para gestionar las entidades, donde cada modelo cuenta con un método ToEntity que facilita la conversión del Modelo a la entidad correspondiente. Estos controllers heredan de la clase ControllerBase de .NET, permitiendo el uso de servicios de la API mediante peticiones HTTP.

En el proceso de las peticiones, los controllers ejecutan la lógica de negocio, implementando las funcionalidades requeridas. En caso de éxito, se devuelve una respuesta "Ok" o "Created" con la información obtenida, o se manejan errores mediante el bloque catch.

Asimismo, en los controllers se configuran y definen los distintos endpoints para permitir la interacción con la API y el sistema en general.

Cuando se recibe una solicitud con un JSON en el cuerpo, se gestiona mediante un DTO. Esta práctica permite distinguir los modelos expuestos en la API de aquellos manejados internamente por la lógica de negocio. Por ejemplo, en la lógica interna, una carpeta puede tener una lista de componentes, y un componente, si es una carpeta, puede tener otra lista de componentes.

Los filtros desempeñan un papel crucial al ejecutar código relevante antes de cada solicitud. Por ejemplo, se utilizan para verificar si el usuario está autenticado mediante un token de Json Web Token, y también para gestionar los permisos en acciones específicas.

Interfaz Uniforme:

Evitamos tener varios endpoints con nombres similares que realicen acciones parecidas. En nuestro enfoque, preferimos elegir un único endpoint y gestionar todas las operaciones relacionadas con ese recurso desde ese punto. Estos endpoints son Usuarios, Compras, Productos

Sin Estado (stateless):

En cada interacción, es esencial autenticarse con la API. La falta de autenticación significa que la API no reconoce al usuario y no retiene información entre solicitudes. En el protocolo HTTP, al no establecer un canal de comunicación persistente, siempre se debe enviar toda la información necesaria en cada request.

Utilizamos versiones, y no utilizamos verbos ni más de 3 niveles de la URI:
En nuestra URI, incorporamos versiones para mantener funcionalidades consistentes con diferentes iteraciones de código, en la URI del obligatorio podrán ver que todas son “v1” debido a que están en su primera versión. Cuando utilizamos IDs en la URI, siempre hacen referencia al recurso de la izquierda, por ejemplo, utilizamos “usuarios/2/compras” cuando deseamos que el usuario 2 haga algo con cierta compra. Optamos por utilizar sustantivos en lugar de verbos en nuestros recursos, como bien se demuestra en el ejemplo, en lugar de "usuarios/2/realizar-compras", preferimos "usuarios/2/**compras**".

Con el objetivo de mejorar la legibilidad y mantenimiento, limitamos nuestra URI a no más de 3 niveles. Por ejemplo, v1/usuarios/{id}/compras es el máximo que permitimos, donde la versión no cuenta como un nivel y los tres recursos son el límite.

Asimismo, cuando diseñamos nuestras URI, seguimos la convención de que el recurso más a la derecha indica qué se va a modificar.

Estas prácticas nos permiten seguir los principios fundamentales de REST y facilitan la comprensión y el mantenimiento de nuestra API.

4.5 Lógica de autenticación

Utilizamos tokens para el acceso y manejo de sesiones con el fin de garantizar la identidad del usuario, asegurando que las solicitudes estén correctamente autenticadas y autorizadas, lo que contribuye a un sistema robusto y protegido.

AuthService (Servicio de Angular en la UI)
Inicialización y Comprobación de Sesión: el estado “isLoggedIn” se inicializa según la presencia de un token en la sesión del usuario (sessionStorage), indicando si está autenticado.

Inicio de Sesión (login): se realiza una solicitud de autenticación al endpoint correspondiente. Si la autenticación tiene éxito, se almacena el token en la sesión del usuario.

Cierre de Sesión (logout): elimina el token y cualquier información relacionada con la sesión del usuario, marcando como no autenticado.

Verificación de Sesión (UserIsLoggedIn): comprueba el estado de isLoggedIn para determinar si el usuario está autenticado.

TokenUserService (Servicio de Angular en la UI)

Uso del Token en Solicitudes: cada solicitud HTTP incluye el token de autorización para asegurar la autenticación y autorización en las operaciones con usuarios y compras.

Métodos para Operaciones con Usuarios y Compras: los métodos operan en recursos de usuarios y compras, empleando el token para garantizar la autenticación y autorización.

4.6 Clean Code - aplicación y ventajas

A lo largo del proceso de creación del proyecto, se hizo un esfuerzo consciente de mantener el código lo más apegado a los estándares de producción del Clean Code posible. Se trató de cumplir con los siguientes puntos, a grandes rasgos:

1) Los nombres de las variables siempre deben ser representativos de sus propósitos y/o la naturaleza de los valores que guardan
2) Las funciones deben ser lo más concisas posibles, y a su vez deben poseer un solo propósito por unidad.
3) Las funciones deben recibir la menor cantidad de parámetros posible.
4) Los números mágicos son mala práctica.
5) Las variables privadas deben iniciar con un _ seguido de camelcase.

Bajo este conjunto de reglas, el equipo desarrolló la completitud del código que se está entregando.

La aplicación de las mismas garantiza un estándar general de mantenibilidad a través del proyecto que no solo facilita el mantenimiento del mismo, sino que también promovería la actualización y expansión del mismo, de desearse realizar las mismas.

4.8 Resources

4.8.1 Códigos de error específicos

Se mantuvo el filtro existente. Hasta el momento de la entrega, las excepciones manejadas consisten de:

> KeyNotFoundException = 404 -> Activada cuando se referencia la id de un elemento inexistente en la base.

> ArgumentException = 400 -> Activado por errores de sintaxis en campos del body de una request (mail sin @, por ejemplo).

> UnauthorizedAccessException = 401 -> Activada cuando se intenta realizar una acción habilitada únicamente para usuarios con ciertos permisos que no se tienen.

> Exception (generica) = 500 -> Activada por casos borde no contenidos por las otras tres.

Se mantuvo la política aplicada en la entrega anterior, la cual plantea que los errores específicos detectados son manejados con las primeras 3 excepciones, y cualquier error que no resulte contenido por éstas será contenido por la 500, la cual se usa de comodín para casos de error no encontrados hasta el momento.

5. Vistas de Diseño e Implementación

5.1. Diagrama de paquetes

<+> Ver en anexo

El diagrama anterior se ve dividido en secciones de 5 colores, cada una representando la naturaleza de la vista correspondiente a sí misma en el modelo 4+1.

Verde: Representa la vista de procesos, la cual se encuentra compuesta por los procesos del sistema y la manera en que se interconectan.

Azul: Representa la vista lógica, la cual engloba las funcionalidades que el sistema proporciona a los usuarios finales

Rojo: Representa la vista de despliegue, la cual muestra la división del programa en términos de “áreas generales” a los ojos del programador.

Violeta: Representa la vista física, la cual contiene las conexiones entre todos los componentes del sistema.

Naranja(+1): Representa la vista de escenarios, la cual contiene los casos de uso del software; y con ellos la unión de las otras cuatro vistas bajo sí misma.

5.2. Descripción de cada paquete

5.2.1. DataAccess

<+> Ver en anexo

En este paquete se define “ECommerceContext”, la cual hereda de DbContext, y sirve como intermediario para la interacción de la lógica del negocio con la base de datos. Además, está la carpeta “Migrations”, la cual almacena todas las versiones de cambios de la base de datos, y diversas clases de repositorios, encargadas de llevar a cabo la creación, lectura, actualización y eliminación de entidades particulares.

5.2.2. Dominio

<+> Ver en anexo

En este se definen las clases que representan las entidades del dominio del proyecto, se dividen en dos paquetes, estos siendo Usuario y Producto donde cada paquete se encarga de complementar al Usuario y Producto. Se agrega MetodoDePago en esta entrega, siendo esta clase similar a CategoriaRol.

5.2.3. Servicios

El proyecto servicios se encarga de ser la capa intermedia entre lo que es DataAccess y la API, además, contiene el patrón strategy de promociones para aplicarlas en la lógica de negocio de la compra en el usuario. Ya que las clases de promociones se

encuentran abstraídas del diseño a nivel de clases, sino más bien que se detectan en formato dll, el primer diagrama es representativo, más que factual.

<+> Ver en anexo

5.2.4. ServicioFactory

El Servicio Factory tiene un solo programa que inyecta dependencias a lo largo de toda la solución.

5.2.5. Migrations

Creado para cumplir el principio de Single Responsibility Principle y crear las migraciones mediante líneas de comando.

5.2.6. Pruebas

Las pruebas fueron divididas en 4 paquetes: aquellas relacionadas con los productos, promociones, compras y usuarios. Con el fin de trabajar de manera más prolija, dentro de cada proyecto se crearon archivos de prueba dependiendo de qué clase se iba a probar, por ejemplo, uno para probar controladores y otro para probar el servicio, nuevamente, para poder mantener el SRP. Necesariamente, para las pruebas de promociones se crearon clases de prueba particulares con los contenidos de las promociones conocidas; ya que el paquete de pruebas no tiene Reflection implementado, y lo que es necesario probar es la lógica, no la instancia de las clases en sí.

5.2.7. Api

Es el paquete con los controladores que exponen los endpoints de la API mediante la cual el usuario puede interactuar con el sistema y modificar o consultar el estado de este. Además, se comunica con los servicios a través de interfaces para responder a las peticiones que se realizan.

5.3. Modelo de tablas

<+> Ver en anexo

La tabla ComprarProducto es una tabla autogenerada por la base para representar la relación N a N entre las clases Compra y Producto. Estas clases ambas una lista de la otra para significar este vínculo; Producto tiene una propiedad Compras de tipo List<Compra> y Compra tiene una de producto de tipo List<Producto>, inversamente.

Se incluye en el diagrama la tabla _EFMigrationsHistory.

5.4. Diagrama de componentes

<+> Ver en anexo

Justificación y explicación del diseño en base al uso de principios de diseño, patrones de diseño y métricas.

Extensibilidad solicitada en la funcionalidad (Reflection)

Se modificó el código para albergar una nueva estructura del proyecto, en la cual las promociones existentes no se encuentran estáticas dentro de un assembly, sino que estas se pueden modificar dinámicamente. Esto se realiza mediante el uso de lo que se conoce como **Reflection**, aplicado a la lógica de instanciación de promociones en la clase ServicioCompra. La implementación específica se puede dividir en dos fases, la de descubrimiento y la de uso.

La **fase de descubrimiento** consiste en el análisis del directorio proveído (Se creó una carpeta promociones a nivel de proyectos donde se deben guardar los archivos .dll de cada una de estas), y el posterior filtrado de los resultados descubiertos. Este filtrado se basa en identificar los archivos de tipo .dll, que categoricen como clases, no abstractas ni interfaces; pero que implementen la interfaz IPromocionStrategy. De este modo, se agregan a una lista del tipo de la interfaz todas las promociones identificadas que sigan el formato dado por el patrón utilizado.

La **fase de uso**, siendo relativamente lineal, consiste en recorrer la lista creada en el paso anterior mediante un foreach, y, para cada elemento encontrado, utilizar el método CompararPrecio de ServicioCompra; el cual utiliza el método AplicarPromocion de cada una de estas para ver el precio posible que resulta en caso de ser posible aplicarlas y retornarlo; de ser este menor al precio guardado hasta el momento como óptimo, este se actualizará.

5.5 Dependencias

Como ya fue mencionado en secciones anteriores, se crearon interfaces de las cuales las clases de lógica dependen. Tanto los repositorios como los servicios; o manejadores, dependiendo de cada clase; implementan una interfaz con sus métodos públicos.

Esto se creó para fomentar y facilitar la aplicación de los principios DIP e ISP de SOLID; ya que al inyectar dependencias de la capa DataAccess en la capa de servicios; y subsecuentemente al realizar esto con Servicios en la API, se establecen vínculos de dependencia con las interfaces apropiadas directamente. La gran ventaja presentada por esta práctica consiste principalmente en que, en caso de requerirse la implementación de distintas variantes del mismo concepto (repositorio para productos, por ejemplo), no habría necesidad de modificar el código existente para permitir la integración del nuevo, mas allá de las funcionalidades y la conexión de los métodos exclusivos a través de las capas; si es que estos fueron agregados. Solamente con crear la clase que implemente la interfaz ya podemos utilizarla en todo lugar donde la otra se encuentra. Por el lado de ISP, como ya fue mencionado en la sección 5.2, la ventaja de definir interfaces concisas y “dedicadas”; por asi llamar al hecho de que cada una de estas, y por extensión de las clases que las implementan, se centra alrededor de una cierta función, o orientación de sus tareas; por ejemplo, la interfaz IPromocionStrategy, únicamente se dedica a lo que es lógica de promociones, no comparte métodos con la interfaz de ManejadorUsuario. Esto, progresivamente, fomenta SRP en las clases afectadas.

5.6 Diagramas de Secuencia

<+> Ver ambos en anexo

Estos diagramas representan procesos que el equipo considero importantes dentro del sistema; comenzando desde la request en los controladores, y mostrando el proceso transversalmente.

El registro de compra muestra la detección de dlls de promociones con Reflection (CargarPromociones), junto con la evaluación de los precios posibles y la modificación del precio de la compra dada.

Se omitieron detalles como la autenticación del lado del Postman, ya que se consideró que no entraban dentro del alcance del diseño y su código, sino la comunicación de herramientas externas con el mismo.

Analizar la calidad del diseño discutiendo su calidad en base a las métricas de diseño y contrastándolas respecto a la aplicación de principios.

Ensamblado	Cohesión relacional (H)	Inestabilidad (I)	Abstracción (A)	Distancia (D)
Dominio	2.33	0.34	0	0.47
DataAccess	1.86	0.96	0.14	0.07
Servicios	1.8	0.88	0.4	0.2
ServiciosFactory	1	0.97	0	0.02
Api	2.63	1	0	0
Migrations	1	1	0	0

Ensamblado: Dominio

- Cohesión Relacional (H): El valor de cohesión relacional es 2.33, lo que indica que hay una fuerte relación entre los tipos dentro del ensamblado. Esto sugiere

que las clases y componentes en este ensamblado están estrechamente relacionados entre sí, lo que puede facilitar la colaboración y el mantenimiento.

- Inestabilidad (I): La inestabilidad es baja (0.34), lo que significa que el ensamblado es menos propenso a cambios. Puede ser considerado como más estable, ya que hay una menor probabilidad de que se produzcan modificaciones en sus dependencias.
- Abstracción (A): La abstracción es 0, lo que sugiere que las clases en el ensamblado son concretas y no abstractas. Esto puede indicar que el ensamblado contiene implementaciones específicas en lugar de solo interfaces o clases base.
- Distancia (D): La distancia es 0.47, indicando que hay una distancia moderada en las dependencias entre el ensamblado y otros componentes. No es ni muy cercano ni muy lejano en términos de dependencias.

Ensamblado: DataAccess

- Cohesión Relacional (H): El valor de cohesión relacional es 1.86, sugiriendo una relación moderada entre los tipos en el ensamblado. Puede haber cierta cohesión, pero no tan fuerte como en el ensamblado de Dominio.
- Inestabilidad (I): La inestabilidad es moderada (0.96), lo que indica que el ensamblado puede ser propenso a cambios en sus dependencias, pero no de manera extrema.
- Abstracción (A): La abstracción es 0.14, lo que indica que hay un nivel bajo de abstracción. Puede haber algunas clases abstractas o interfaces, pero en su mayoría, las clases son concretas.
- Distancia (D): La distancia es 0.07, lo que sugiere que el ensamblado tiene dependencias cercanas con otros componentes.

Ensamblado: Servicios

- Cohesión Relacional (H): El valor de cohesión relacional es 1.8, indicando una relación moderada entre los tipos en el ensamblado. Hay cierta cohesión, pero no tan fuerte como en el ensamblado de Dominio.

- Inestabilidad (I): La inestabilidad es moderada (0.88), lo que sugiere que el ensamblado puede experimentar cambios en sus dependencias, pero no de manera extrema.
- Abstracción (A): La abstracción es 0.4, indicando un nivel moderado de abstracción. Puede haber algunas clases abstractas o interfaces en el ensamblado.
- Distancia (D): La distancia es 0.2, lo que sugiere que el ensamblado tiene dependencias moderadamente cercanas con otros componentes.

Ensamblado: ServiciosFactory

- Cohesión Relacional (H): El valor de cohesión relacional es 1, lo que sugiere una relación más débil entre los tipos en el ensamblado. Puede haber una menor cohesión, lo que puede hacer que la colaboración y el mantenimiento sean más desafiantes.
- Inestabilidad (I): La inestabilidad es alta (0.97), lo que indica que el ensamblado es propenso a cambios en sus dependencias.
- Abstracción (A): La abstracción es 0, indicando que las clases en el ensamblado son concretas y no abstractas.
- Distancia (D): La distancia es 0.02, indicando que el ensamblado tiene dependencias muy cercanas con otros componentes.

Ensamblado: Api

- Cohesión Relacional (H): El valor de cohesión relacional es 2.63, sugiriendo una fuerte relación entre los tipos en el ensamblado. Este ensamblado parece tener una cohesión significativa entre sus clases y componentes.
- Inestabilidad (I): La inestabilidad es baja (1), indicando que el ensamblado es menos propenso a cambios en sus dependencias.
- Abstracción (A): La abstracción es 0, indicando que las clases en el ensamblado son concretas y no abstractas.
- Distancia (D): La distancia es 0, lo que sugiere que el ensamblado tiene dependencias muy cercanas con otros componentes.

Ensamblado: Migrations

- Cohesión Relacional (H): El valor de cohesión relacional es 1, lo que indica una relación más débil entre los tipos en el ensamblado. Puede haber una menor cohesión, lo que puede hacer que la colaboración y el mantenimiento sean más desafiantes.
- Inestabilidad (I): La inestabilidad es 1, lo que indica que el ensamblado es propenso a cambios en sus dependencias.
- Abstracción (A): La abstracción es 0, indicando que las clases en el ensamblado son concretas y no abstractas.
- Distancia (D): La distancia es 0, lo que sugiere que el ensamblado tiene dependencias muy cercanas con otros componentes.

Clausura Común:

- Dominio: Tiene una cohesión relacional bastante alta (2.33), lo que sugiere que las clases dentro del ensamblado están fuertemente relacionadas. Esto es coherente con el principio de Clausura Común, ya que las clases relacionadas están agrupadas en el mismo ensamblado.
- Api: Al tener una cohesión relacional fuerte (2.63), también cumple con el principio de Clausura Común. Las clases que forman parte del API están estrechamente relacionadas y agrupadas en el mismo ensamblado.

Reuso Común:

- Api: Dado que el API tiene una cohesión relacional alta y una inestabilidad baja, puede ser un buen candidato para el reuso. La baja inestabilidad sugiere que es menos propenso a cambios, lo que es favorable para el reuso sin introducir efectos colaterales no deseados.

Dependencias Ascíclicas

- Al ser un proyecto en .NET, no hay dependencias ascíclicas debido a que no vamos a poder compilarlo y ejecutarlo.

Abstracciones Estables:

- ServiciosFactory: Aunque tiene una inestabilidad alta, la abstracción es baja (0), lo que indica que las clases son concretas y no abstractas. Esto podría sugerir que las implementaciones concretas están más sujetas a cambios, lo que podría afectar la estabilidad. Sin embargo, este ensamblado tiene una distancia muy baja (0.02), lo que indica que las dependencias son muy cercanas. Esto puede favorecer la estabilidad en términos de impacto de cambios.

Dependencias Estables:

- Api: Tiene una inestabilidad baja (1) y una cohesión relacional alta (2.63), lo que sugiere que las dependencias en el API son estables. Es menos propenso a cambios y las clases están fuertemente relacionadas, lo que favorece la estabilidad en las dependencias.

En el Anexo se puede ver la gráfica de Inestabilidad vs Abstracción teniendo los ensamblados Servicios y DataAccess con una distancia buena pero no perfecta a la recta debido a que son los ensamblados que se encuentran entre el Dominio y la Api. En cambio el Dominio se encuentra en la zona de dolor debido a que Api, DataAccess y Servicios dependen de este.

6 Anexo

Endpoints Nuevos

- GET `/api/v1/productos/colores`
  - No recibe parámetros, header, body
  - Respuesta:

```
[
  {
    "id": number,
    "nombre": string
  }
]
```

- GET `/api/v1/productos/marcas`
  - No recibe parámetros, header, body
  - Respuesta:

```
[
  {
    "id": number,
    "nombre": string
  }
]
```

- GET /api/v1/productos/marcas
    - No recibe parámetros, header, body
    - Respuesta:

```
[
  {
    "id": number,
    "nombre": string
  }
]
```

Endpoints Modificados

- POST /api/v1/productos/{id}
    - No recibe parámetros
    - Recibe un header {‘Authorization’ : bearer + token}
    - Body:

```
{
  "nombre": string,
  "precio": number,
  "descripcion": string,
  "marcaId": number,
  "categoriaId": number,
  "stock": number,
  "aplicaParaPromociones": boolean,
  "colorId": number
}
```

    - Respuesta:

```
{
    "id": number,
    "nombre": string,
    "precio": number,
    "descripcion": string,
    "stock": number,
    "marca": {
        "id": number,
```

```
            "nombre": string
        },
        "categoria": {
            "id": number,
            "nombre": string
        },
        "color": {
            "id": number,
            "nombre": string
        },
        "aplicaParaPromociones": boolean
    }
```

- GET /api/v1/productos/{id}
    - No recibe parámetros, header ni body
    - Respuesta:

```
[
  {
    "id": number,
    "nombre": string,
    "precio": number,
    "descripcion": string,
    "stock": number,
    "marca": {
      "id": number,
      "nombre": string
    },
    "categoria": {
      "id": number,
      "nombre": string
    },
    "color": {
      "id": number,
      "nombre": string
    },
    "aplicaParaPromociones": boolean
  }
]
```

- GET /api/v1/productos
    - Tiene parámetros: Nombre, PrecioEspecifico, MarcaId, CategoriaId, TienePromociones, RangoPrecio

- No recibe header ni body
- Respuesta:

```
{
  "id": number,
  "nombre": string,
  "precio": number,
  "descripcion": string,
  "stock": number,
  "marca": {
    "id": number,
    "nombre": string
  },
  "categoria": {
    "id": number,
    "nombre": string
  },
  "color": {
    "id": number,
    "nombre": string
  },
  "aplicaParaPromociones": boolean
}
```

- PUT /api/v1/productos/{id}
  - No recibe parámetros
  - Recibe un header {'Authorization' : bearer + token}
  - Body:

```
{
  "nombre": string,
  "precio": number,
  "descripcion": string,
  "marcaId": number,
  "categoriaId": number,
  "stock": number,
  "aplicaParaPromociones": boolean,
  "colorId": number
}
```

  - Respuesta:

```
{
    "id": number,
    "nombre": string,
    "precio": number,
    "descripcion": string,
```

```
    "stock": number,
    "marca": {
        "id": number,
        "nombre": string
    },
    "categoria": {
        "id": number,
        "nombre": string
    },
    "color": {
        "id": number,
        "nombre": string
    },
    "aplicaParaPromociones": boolean
}
```

- GET /api/v1/compras
  - No recibe parámetros ni body
  - Recibe un header {'Authorization' : bearer + token}
  - Respuesta:

```
[
    {
        "id": number,
        "productos": [
             number
        ],
        "precio": number,
        "nombrePromo": string,
        "fechaCompra": string,
        "usuarioId": number
    }
]
```

- GET /api/v1/usuarios/{id}/compras
  - No recibe parámetros ni body
  - Recibe un header {'Authorization' : bearer + token}
  - Respuesta:

```
[
    {
        "id": number,
        "productos": [
             number
        ],
        "precio": number,
```

```
            "nombrePromo": string,
            "fechaCompra": string,
            "usuarioId": number
        }
    ]
```

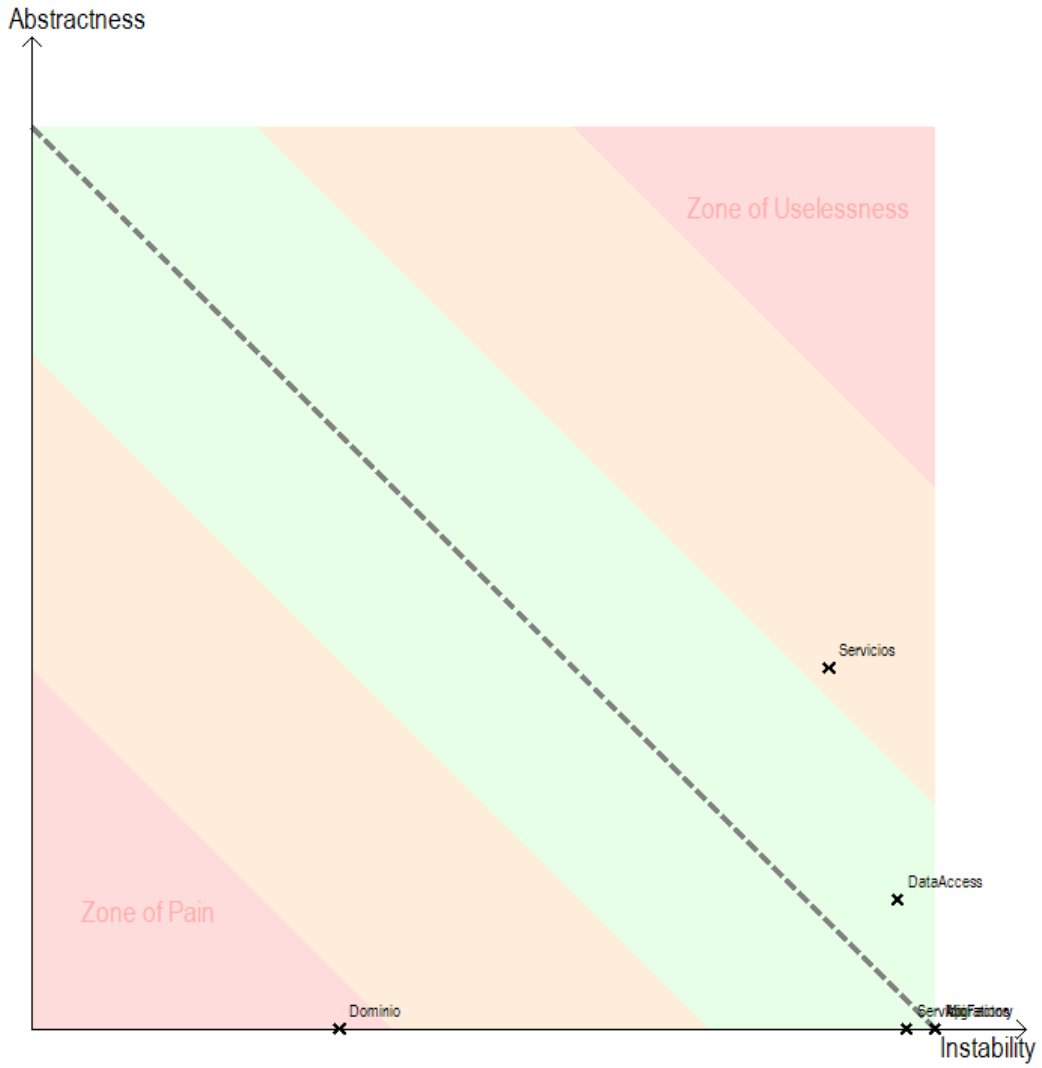
- POST /api/v1/usuarios/{id}/compras
    - No recibe parámetros
    - Recibe un header {‘Authorization’ : bearer + token}
    - Body:

```
{
  "idProductos": [
    number
  ],
  "metodoDePago": string
}
```

    - Respuesta:

```
{
  "idProductos": [
    number
  ],
  "metodoDePago": string
}
```

Inestabilidad vs Abstracción de la solución



TDD

6.2 Diagramas grandes

Diagrama 4.3.2 - Patrón Facade

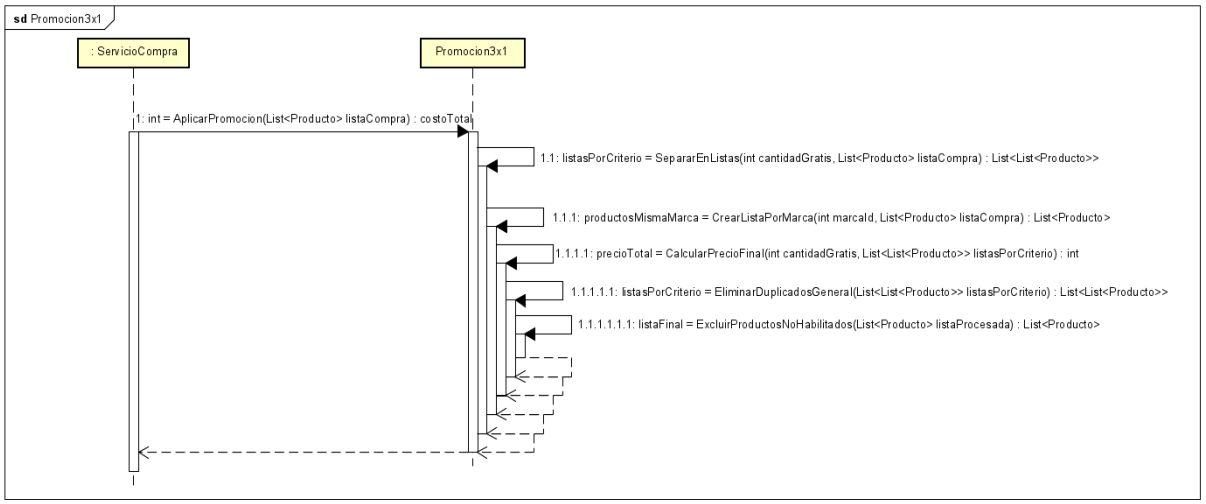


Diagrama 5.1 - Paquetes

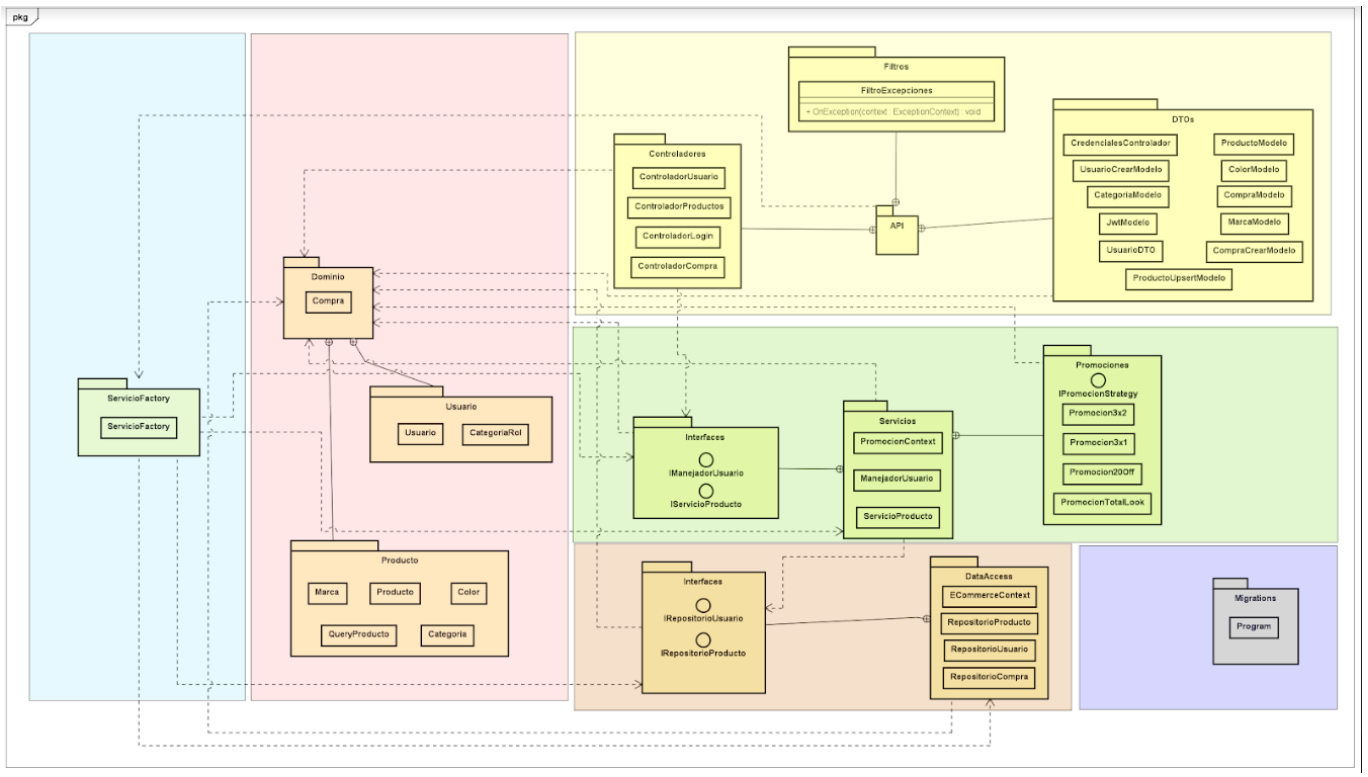


Diagrama 5.2.1 -DataAccess

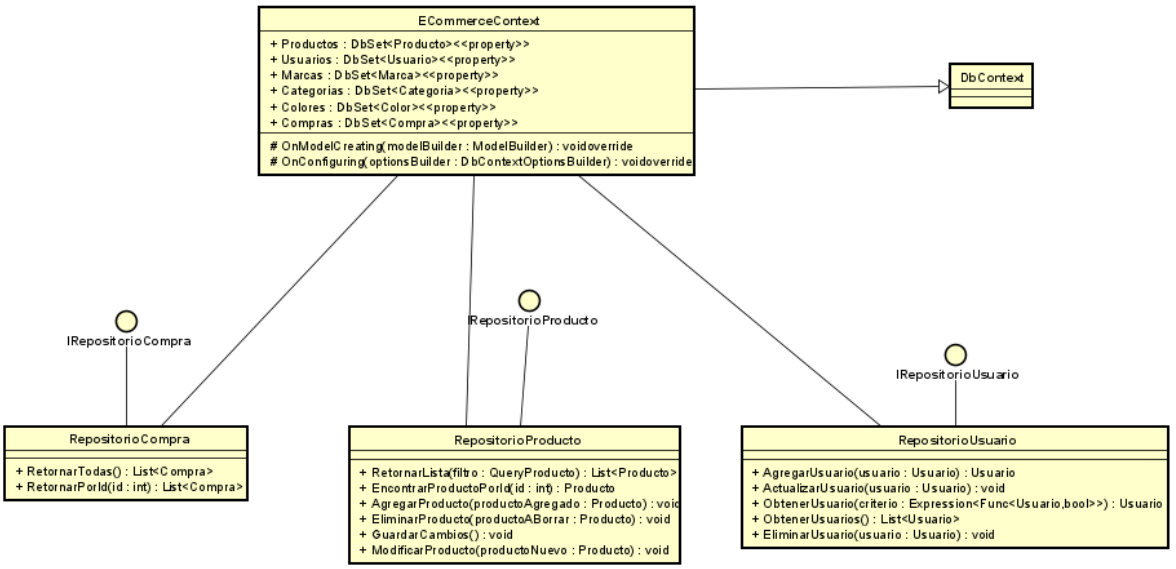
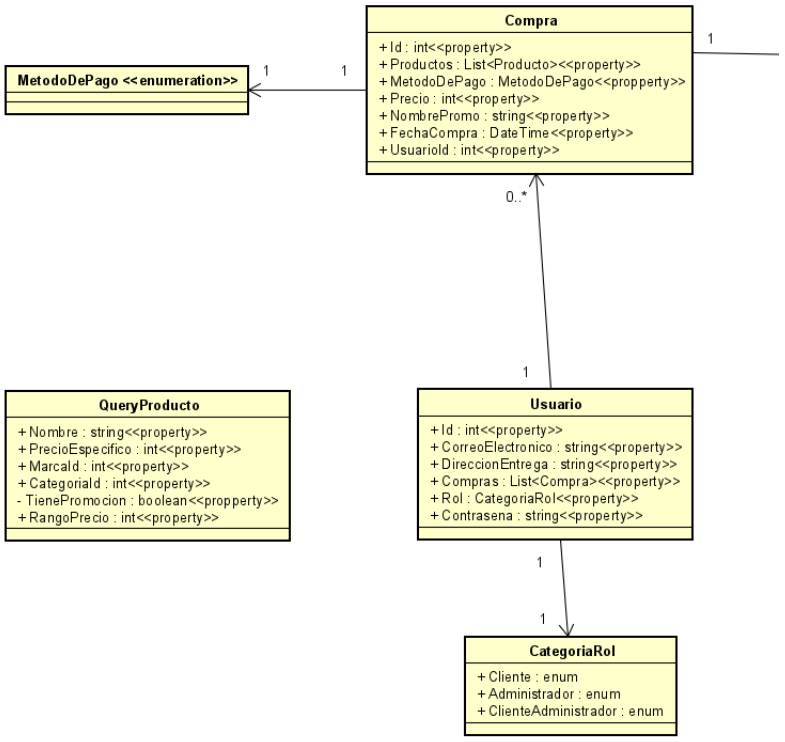


Diagrama 5.2.2 - Dominio



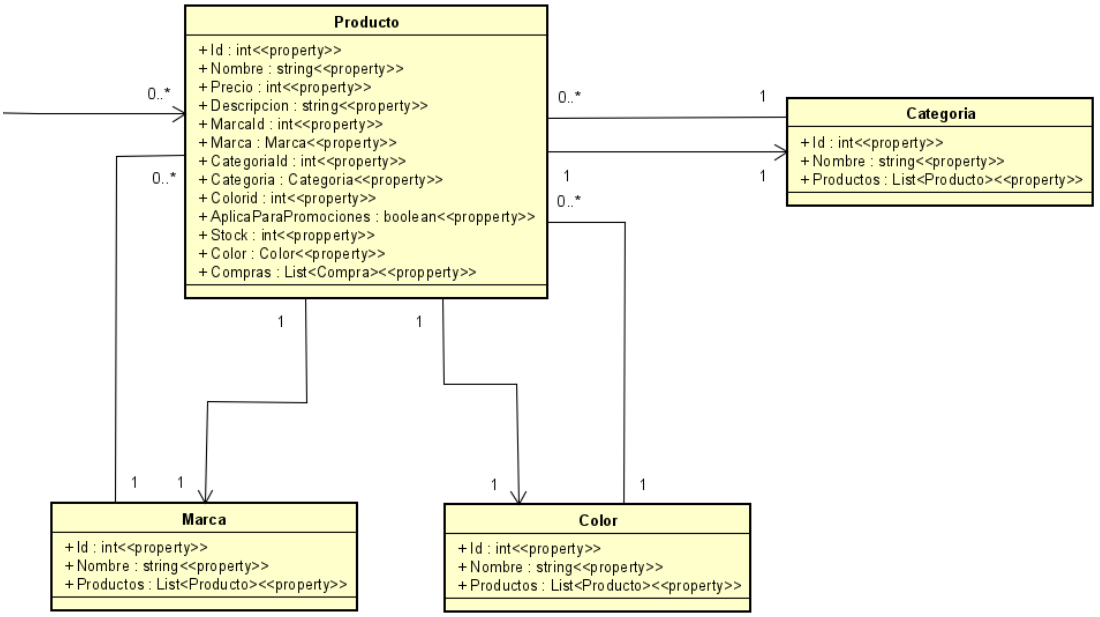
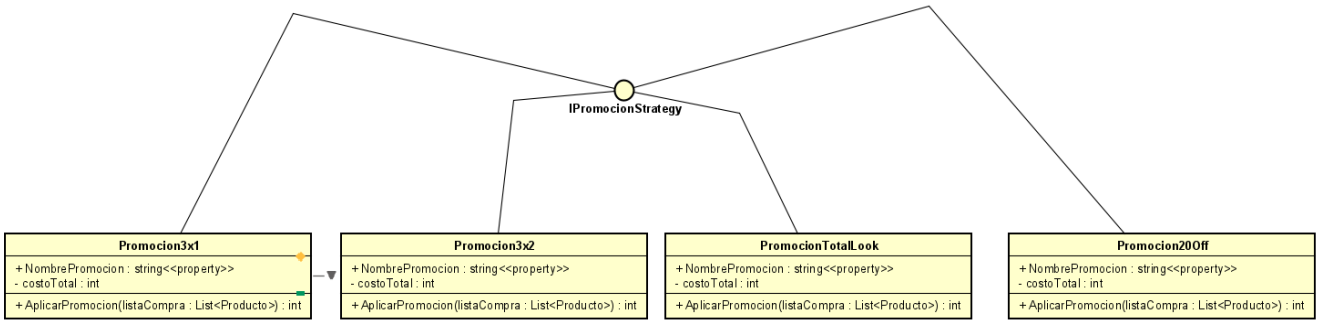


Diagrama 5.2.3 - Servicios



(Patrón strategy aplicado a promociones)

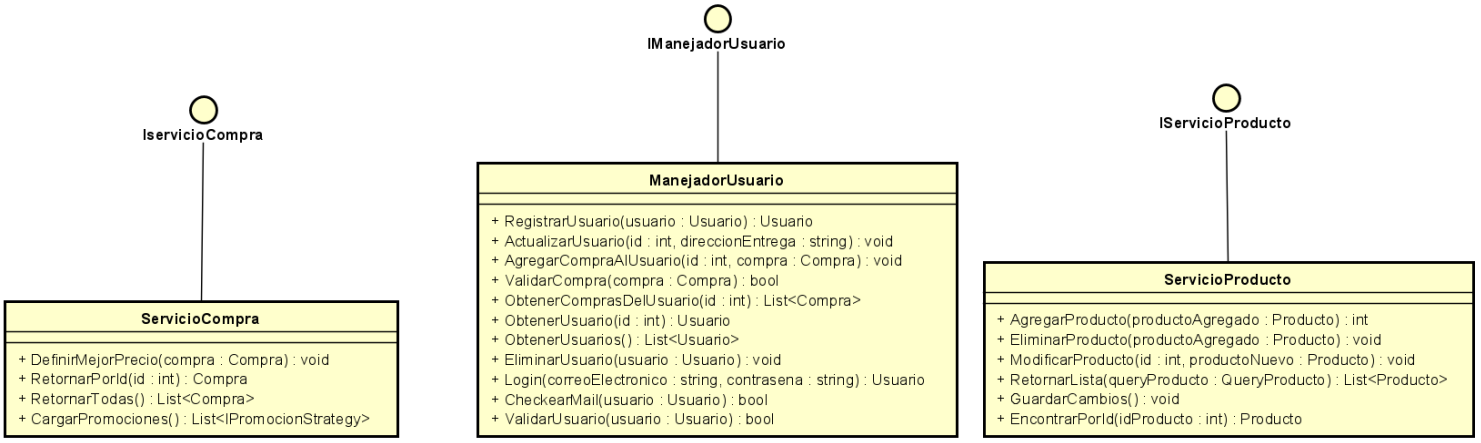


Diagrama 5.4 - Componentes

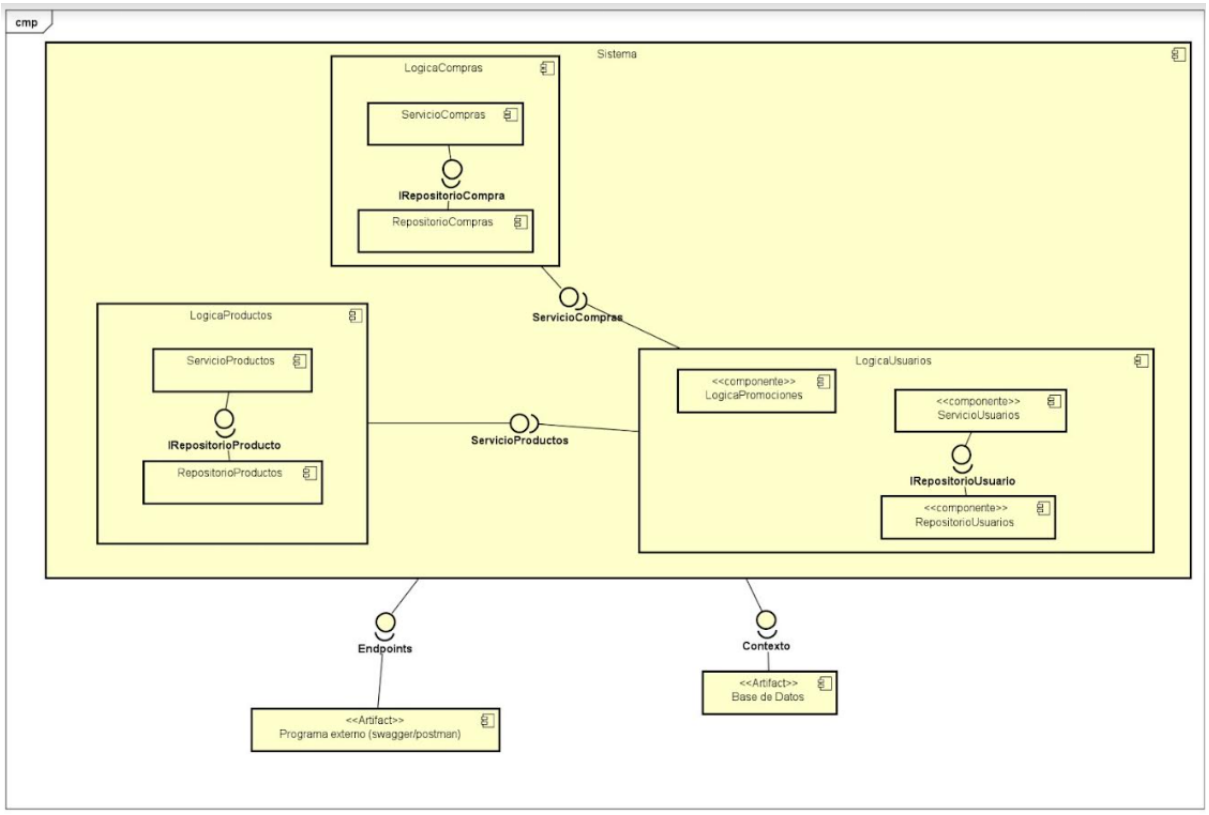


Diagrama 5.2.7 - API

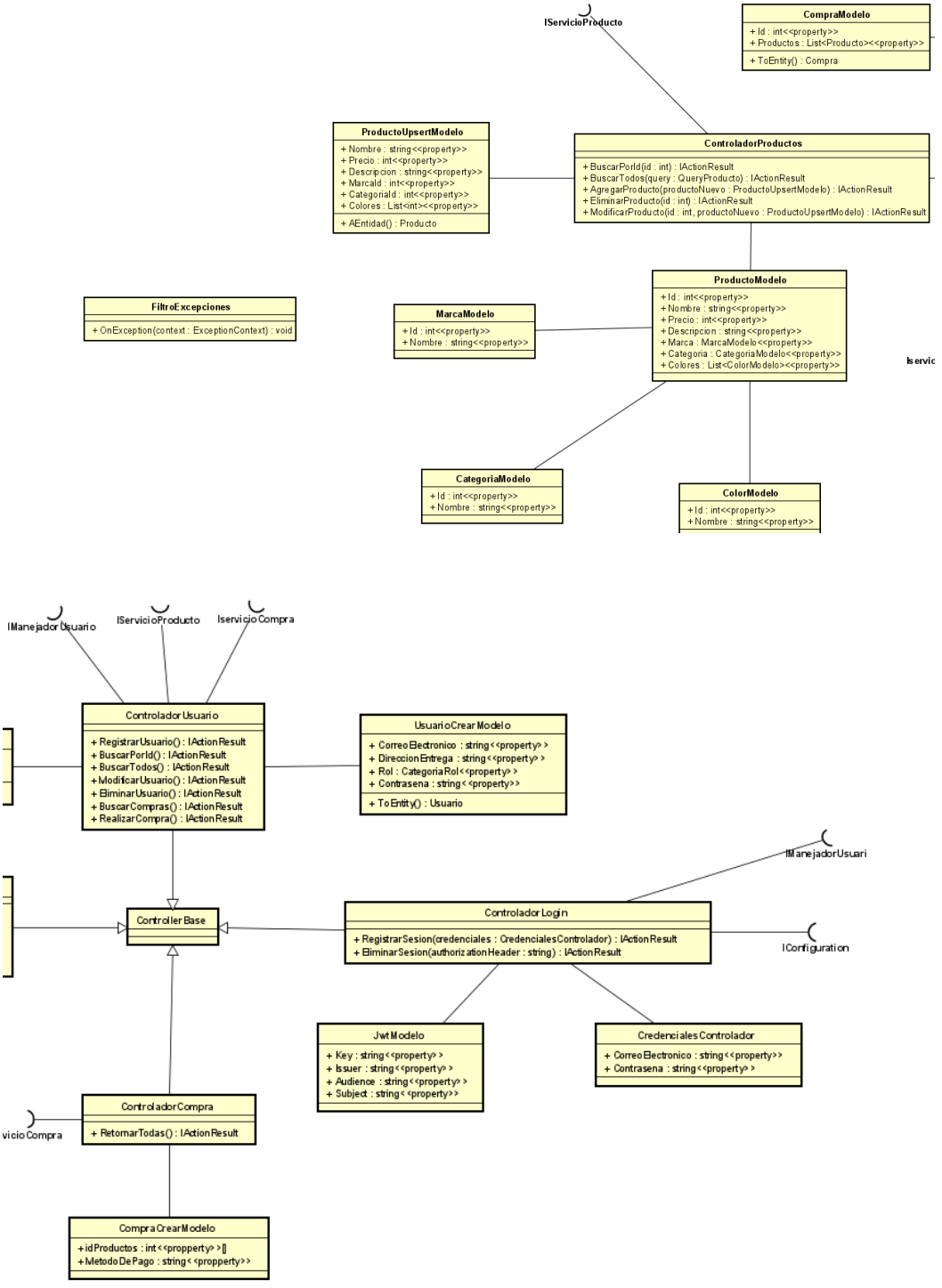


Diagrama 5.3 - Modelo de tablas

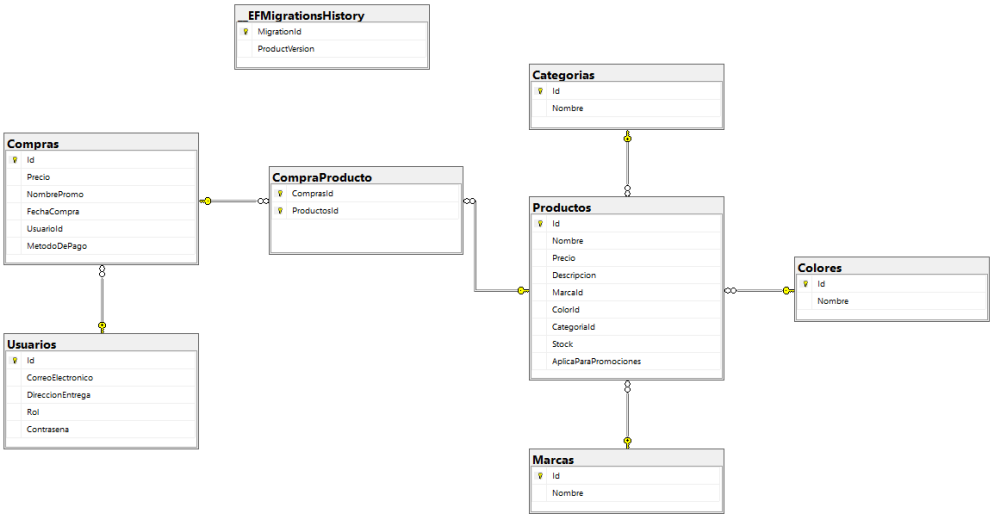
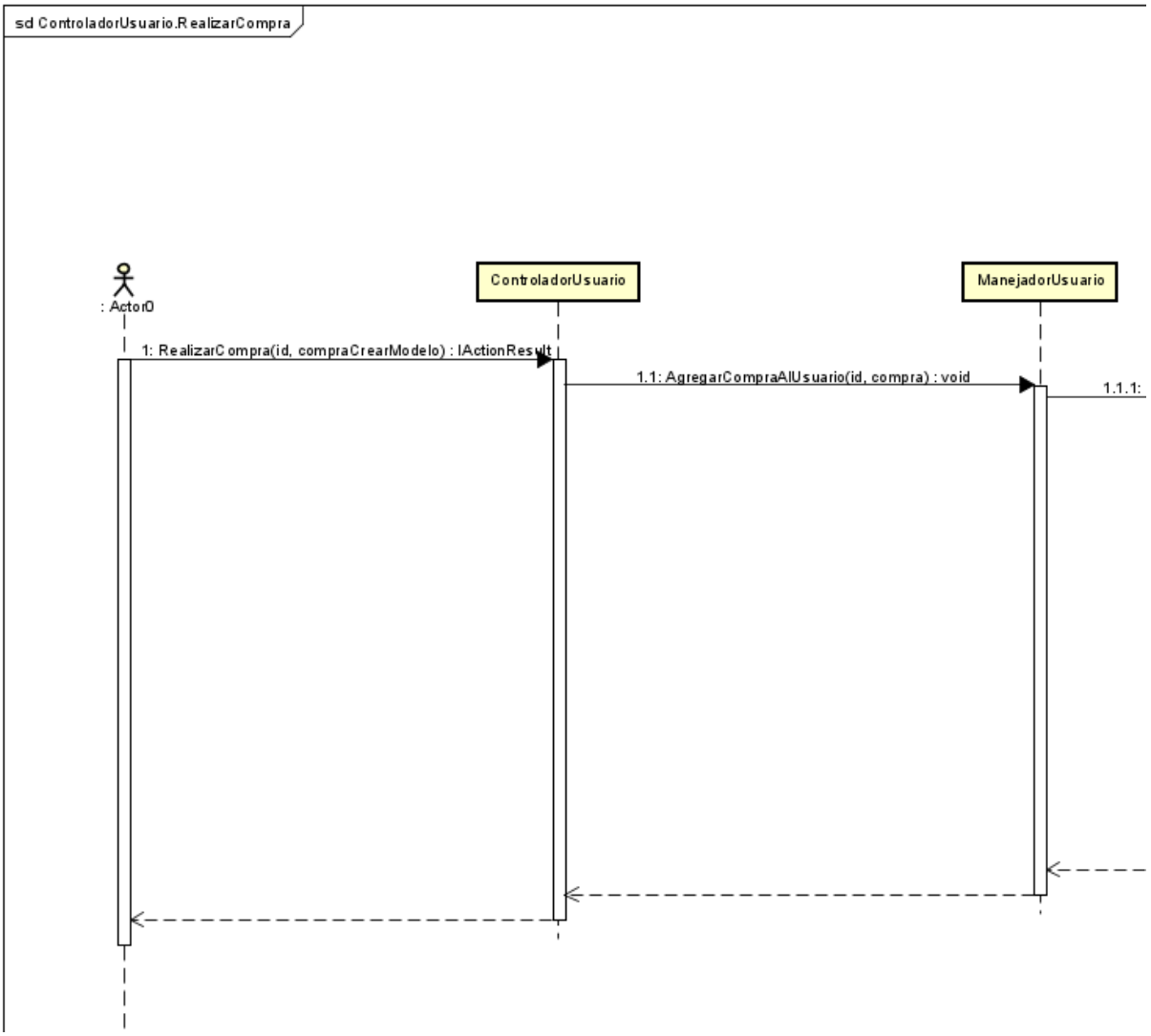


Diagrama 5.6 A - Secuencia de la funcionalidad “Agregar compra al usuario”



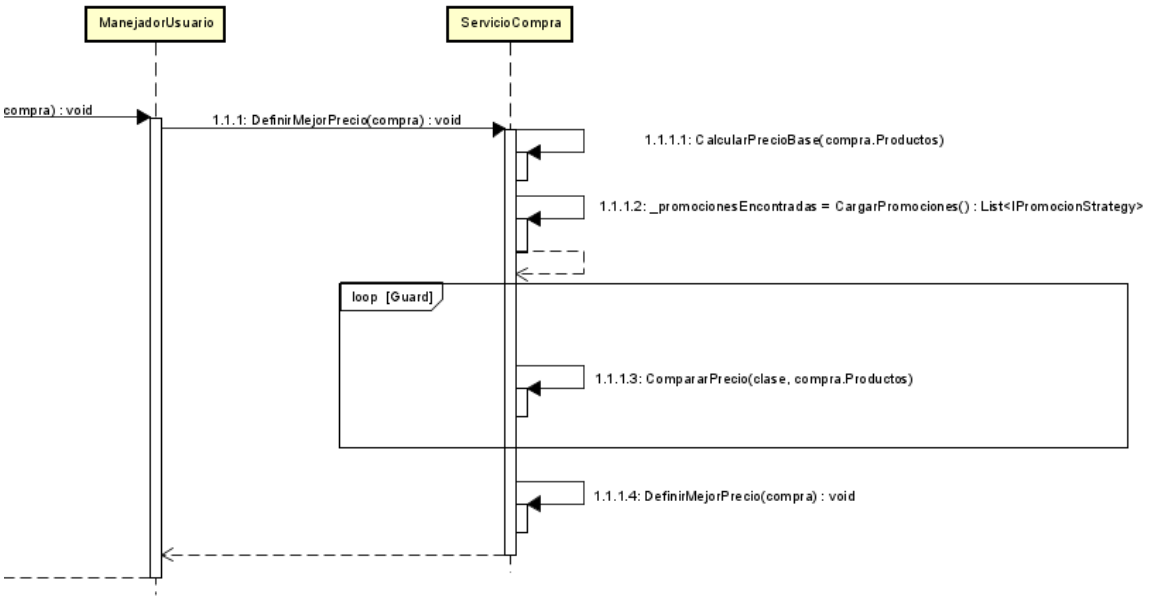
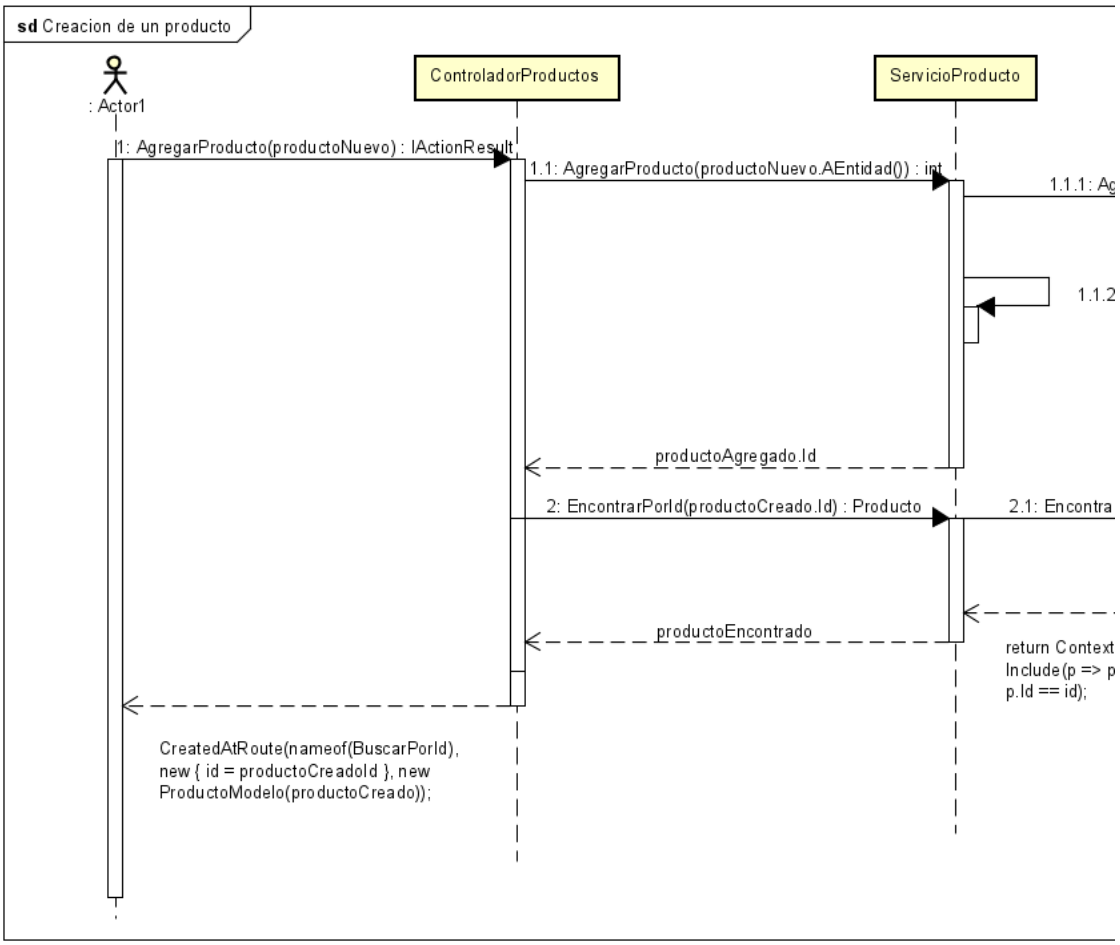
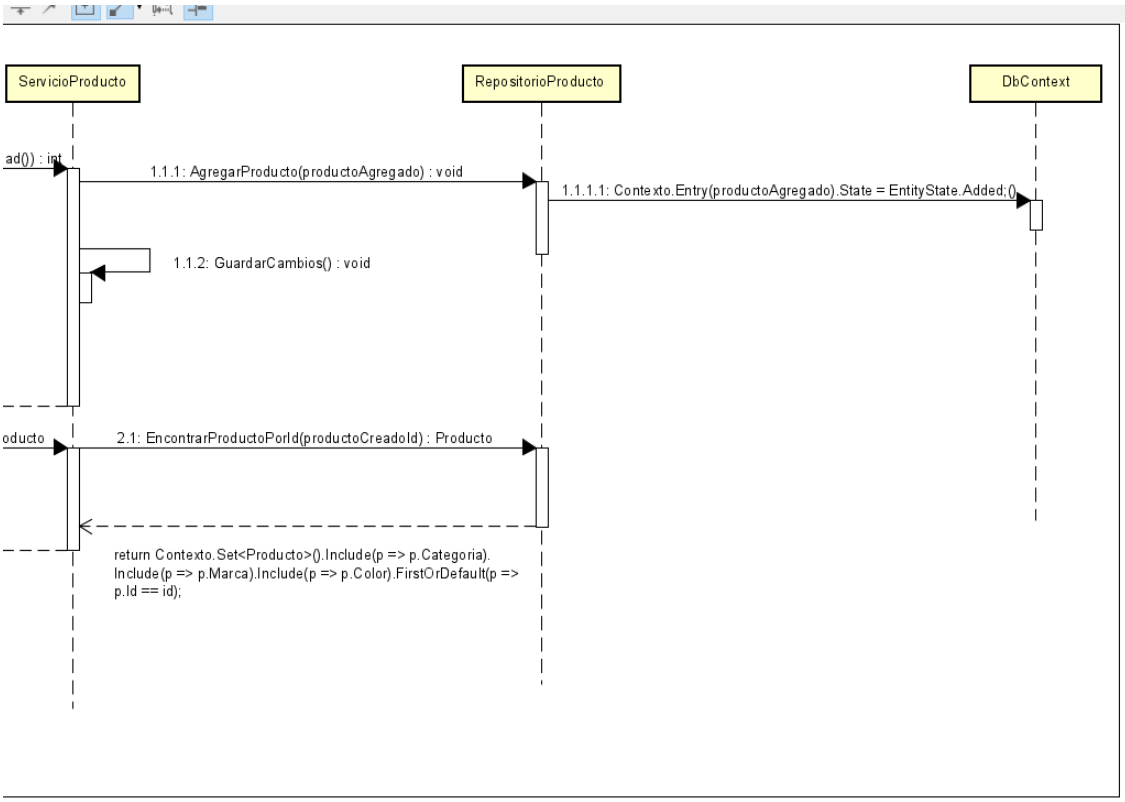

ManejadorUsuario
ServicioCompra
compra) : void
1.1.1: DefinirMejorPrecio(compra) : void
1.1.1.1: CalcularPrecioBase(compra.Productos)
1.1.1.2: _promocionesEncontradas = CargarPromociones() : List<IPromocionStrategy>
loop [Guard]
1.1.1.3: CompararPrecio(clase, compra.Productos)
1.1.1.4: DefinirMejorPrecio(compra) : void

Diagrama 5.6 B - Secuencia de la funcionalidad “registrar producto”




ServicioProducto
RepositorioProducto
DbContext
ad() : int
1.1.1: AgregarProducto(productoAgregado) : void
1.1.1.1: Contexto.Entry(productoAgregado).State = EntityState.Added;()
1.1.2: GuardarCambios() : void
oducto
2.1: EncontrarProductoPorId(productoCreadoId) : Producto
return Contexto.Set<Producto>().Include(p => p.Categoria).
Include(p => p.Marca).Include(p => p.Color).FirstOrDefault(p =>
p.Id == id);