Universidad ORT Uruguay

Facultad de Ingeniería

Bernard Wand Polak

Licenciatura en Sistemas - 2023

Diseño de aplicaciones 2

Obligatorio I (Descripción del diseño)

Agustín Fioritto	**Nro est. 238927**
Joaquín Calvo	**Nro est. 203832**
Chiara Sosa	**Nro est. 241699**

ÍNDICE

Descripción general del trabajo

El proyecto actual tiene como objetivo principal la implementación del Backend y la API REST, que serán la base para la creación del Frontend de una aplicación web de comercio electrónico completa.

Las principales funcionalidades de esta entrega incluyen:

- Gestión de usuarios: registro de nuevos usuarios, inicio de sesión y cierre de sesión de usuarios ya creados, actualización de datos de usuarios, historial de compras.

- Manejo de productos del comercio: creación, eliminación y actualización de productos, búsqueda de productos según sus características.

- Gestión de carritos de compras: agregar y eliminar productos a los carritos, cálculo y aplicación de la mejor promoción aplicable al carrito disponible.

- Ejecución de pedidos: procesos de creación y confirmación de pedidos.

- Persistencia de datos: la información sobre clientes, productos, carritos y compras es almacenada en una base de datos, con la cual se interacciona en todo momento para actualizarla según las operaciones realizadas.

En esta fase inicial, nos enfocamos en desarrollar todos los endpoints esenciales que permitirán la futura interacción entre los usuarios y la base de datos. Estos componentes formarán la base sobre la cual se construirá la interfaz de usuario en el Frontend durante la segunda fase del proyecto.

En este documento se detallan distintos diagramas respectivos al modelo 4 + 1, los cuales nos permitieron tanto a nosotros como equipo de trabajo comprender y asociar ciertos aspectos del sistema, como también comunicarlos a terceros interesados en el proyecto. Se incluyen diagramas de clases de los distintos paquetes para modelar la estructura interna del sistema desde una perspectiva *lógica*. Respecto a la *vista de procesos,* se detallan diagramas de secuencia de dos casos de uso que se consideran relevantes, ayudando a visualizar cómo se ejecutan los mismos y cómo se coordinan las actividades internas del sistema. Por último, para ilustrar la vista de implementación, se adjuntan los diagramas de componentes y paquetes.

Bugs

- Agregar productos al carrito no logueado:
  Para no manejar sesiones anónimas, en esta primera entrega no se contempló el caso de uso de agregar productos al carrito sin estar logueado. Esa funcionalidad queda pendiente para la segunda parte del obligatorio quedando a cargo del frontend. Los productos en esos casos se van a guardar en un local storage.

- Las validaciones propias de los atributos de cada clase del dominio consideramos que sería mejor ubicarlas dentro de las clases de entidades en Domain y no en BusinessLogic, ya que teniendo en cuenta que se desea respetar el patrón de diseño “Experto”, tendria mas sentido ubicarlas ahí y no en el paquete que se encarga de la lógica de las operaciones. Es una responsabilidad propia de las entidades.

- Por cuestiones de tiempo para la sección de productos, no se implementó la búsqueda por un texto, una marca y una categoría. Esta será implementada para la segunda entrega.

- Por tema de tiempos no se pudo implementar la persistencia de un usuario logueado mediante sesiones. Para poder hacer esto se settearon usuarios dummy como logueados. Por lo que a la hora de hacer pruebas y trabajar con la API hay que tener en cuenta que el usuario Dummy de esta sección debe estar registrado en la base antes.

```
5 referencias | fiioro25, Hace 9 horas | 2 autores, 28 cambios
public class UserService : IUserService
{
    private readonly IUserManagment _userManagment;
    private User? loggedInUser;



    21 referencias | fiioro25, Hace 14 horas | 1 autor, 4 cambios
    public User? GetLoggedInUser()
    {
        var result = new User
        {

            UserID = 3,
            UserName = "UsuarioAdministrador",
            Email = "usuarioAdministrador@ejemplo.com",
            Password = "ContraseñaAdministrador",
            Role = "Administrador",
            Address = "DireccionAdministrador",
            Cart = null,
            Purchases = null,

            /*
            UserID = 28, // ID del usuario logueado
            UserName = "UsuarioComprador",
            Email = "usuarioComprador@ejemplo.com",
            Password = "ContraseñaComprador",
            Role = "Comprador",
            Address = "DireccionComprador",
            Cart= null,
            Purchases = null,
            */
        };

        return result;
    }
```

- A pesar de haber implementado los endpoint necesarios para las resources Cart y Purchase, estas quedaron funcionando con objetos dummy ya que no se concretó la conexión con la base de datos para esos casos. Para la segunda entrega se buscará solucionar esos detalles.
- Quedó pendiente la implementación de las tablas de categorías, marcas y colores en la base de datos.

Diagrama general de paquetes

La solución se diseñó de manera tal que la organización y la estructura de la misma fomenten la eficiencia, la escalabilidad y la mantenibilidad del sistema, evitando dependencias innecesarias entre paquetes para lograr un bajo acoplamiento entre ellos. Esta estructura en capas (layers) facilita la comprensión y el mantenimiento del código, organizándolo de manera coherente y lógica. Cada capa tiene responsabilidades claras y se

comunica con las capas adyacentes de manera que no se intercambie información innecesaria. Esto ayuda a la modularidad y la facilidad de mantenimiento del sistema a medida que se desarrolla y escala.

A continuación se proporciona una breve descripción de cada una de las capas:

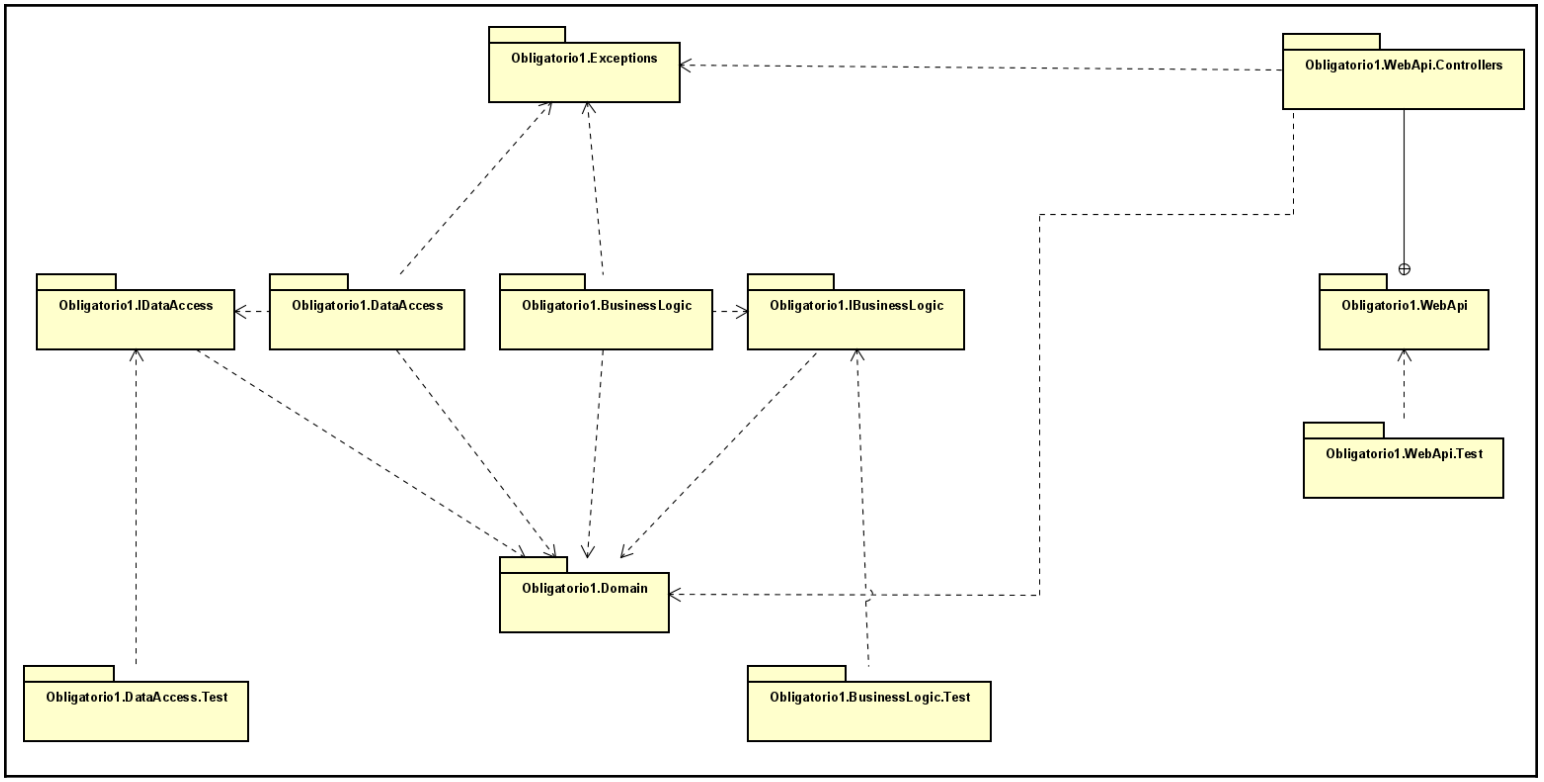
Capa del Dominio: El paquete "Obligatorio1.Domain" contiene las entidades del dominio, como "User", "Cart", “Product” y "Purchase". Estas entidades representan los objetos utilizados. La capa del dominio se encarga de definir la estructura de datos fundamental de la aplicación.

Capa de Lógica de Negocio: El paquete "Obligatorio1.BusinessLogic" alberga la lógica de las operaciones del sistema. Cada clase en este paquete contiene métodos para realizar operaciones relacionadas con las entidades, habiendo una clase de servicio para cada entidad (sumadas las clases con las lógicas de las 4 promociones). Esto ayuda a mantener la lógica del negocio separada de otras capas. El paquete "Obligatorio1.IBusinessLogic" contiene interfaces implementadas por BusinessLogic, y será el punto de acceso a estos servicios sin tener que depender de las implementaciones. Esta lógica también incluye un paquete con tests.

Capa de Acceso a Datos: El paquete "Obligatorio1.DataAccess" se encarga de gestionar el acceso a la base de datos. Las clases de este paquete interactúan directamente con la base de datos y mediante las interfaces contenidas en “Obligatorio1.IDataAcces” ayudan a mantener la lógica de acceso a datos separada de su utilización e interacción con la lógica de negocio.

Capa de Web Api: En el paquete "Obligatorio1.WebApi," se encuentran los controladores necesarios para permitir que los usuarios interactúen con la aplicación a través de una interfaz RESTful. Estos controladores manejan las solicitudes HTTP y comunican las acciones del usuario con la lógica de negocio. También se incluye un paquete de tests para estas operaciones.

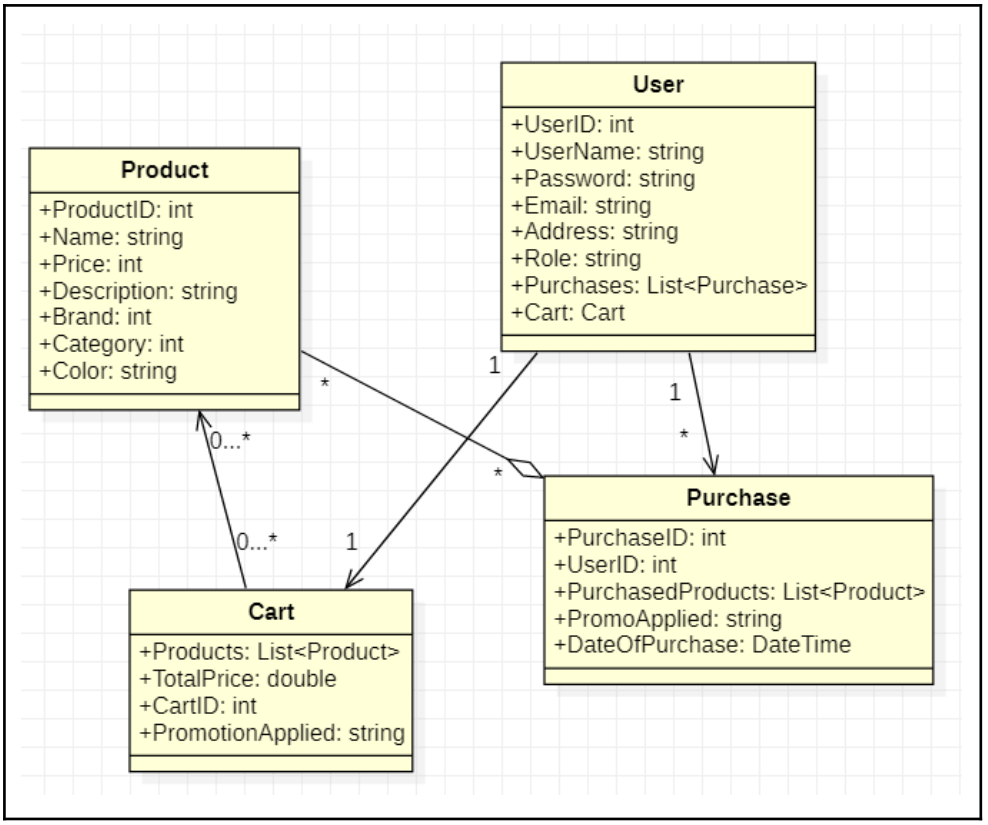
A continuación se detalla un diagrama general de los paquetes que componen el proyecto conformando cada una de estas capas.



Paquete Obligatorio1.Domain

Dentro de Domain, implementamos todas las entidades del dominio siendo estas las representaciones de los objetos del mundo real en nuestra aplicación. Estas clases únicamente contienen sus atributos y su constructor. El siguiente diagrama de clases representa los atributos de cada entidad y las relaciones entre las clases (incluyendo su respectiva cardinalidad). Se evitó incluir para cada una de ellas los constructores sin y con parámetros para descomplejizar el diagrama.

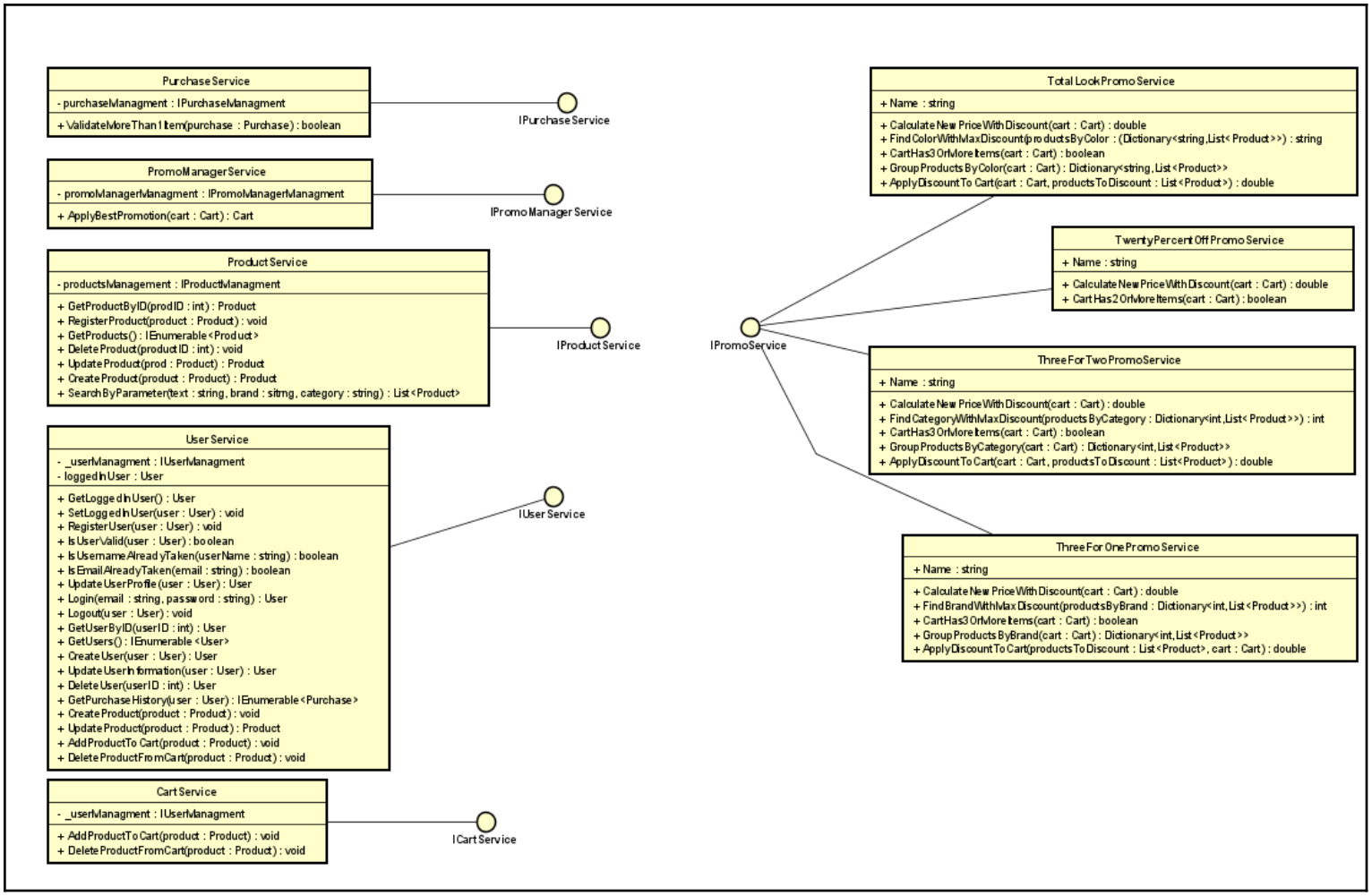
La clase User tiene una relación de asociación con las clases Cart y Purchase, haciendo uso de las mismas para instanciar algunos de sus atributos. La clase Cart está asociada a Product, y la clase Purchase tiene una dependencia de agregación con la clase Product, dado que está formada por productos y no pueden crearse instancias de Purchase si no existen productos en nuestro proyecto.



Paquete Obligatorio1.BusinessLogic

En el apartado de BusinessLogic, se pueden encontrar todas las operaciones disponibles en el proyecto, para cada una de las clases. La sintaxis de los nombres de las clases que contiene BusinessLogic, es el nombre de la clase seguido de la palabra service, a modo de ejemplo, UserService que contendrá todas las operaciones correspondientes a User y el manejo de usuarios: registro, logueo, actualización de datos, etc.

En el diagrama se representan, además de las clases incluidas en este paquete, qué interfaces de IBusinessLogic son implementadas por cada una de las clases.



Paquete Obligatorio1.IBusinessLogic

IBusinessLogic es un conjunto de interfaces, que contienen las definiciones de las funcionalidades, implementadas luego dentro de cada una de sus clases dentro de BusinessLogic. Esto se realizó no solo para abstraer la implementación de la lógica y poder hacer uso de las funcionalidades en otras partes de la solución sin depender de la misma, sino que también nos permitió obtener distintas implementaciones en simultáneo, en el caso de las promociones. Por ejemplo para la clase userService, su interfaz será IUserService.

Paquete Obligatorio1.BusinessLogic.Test

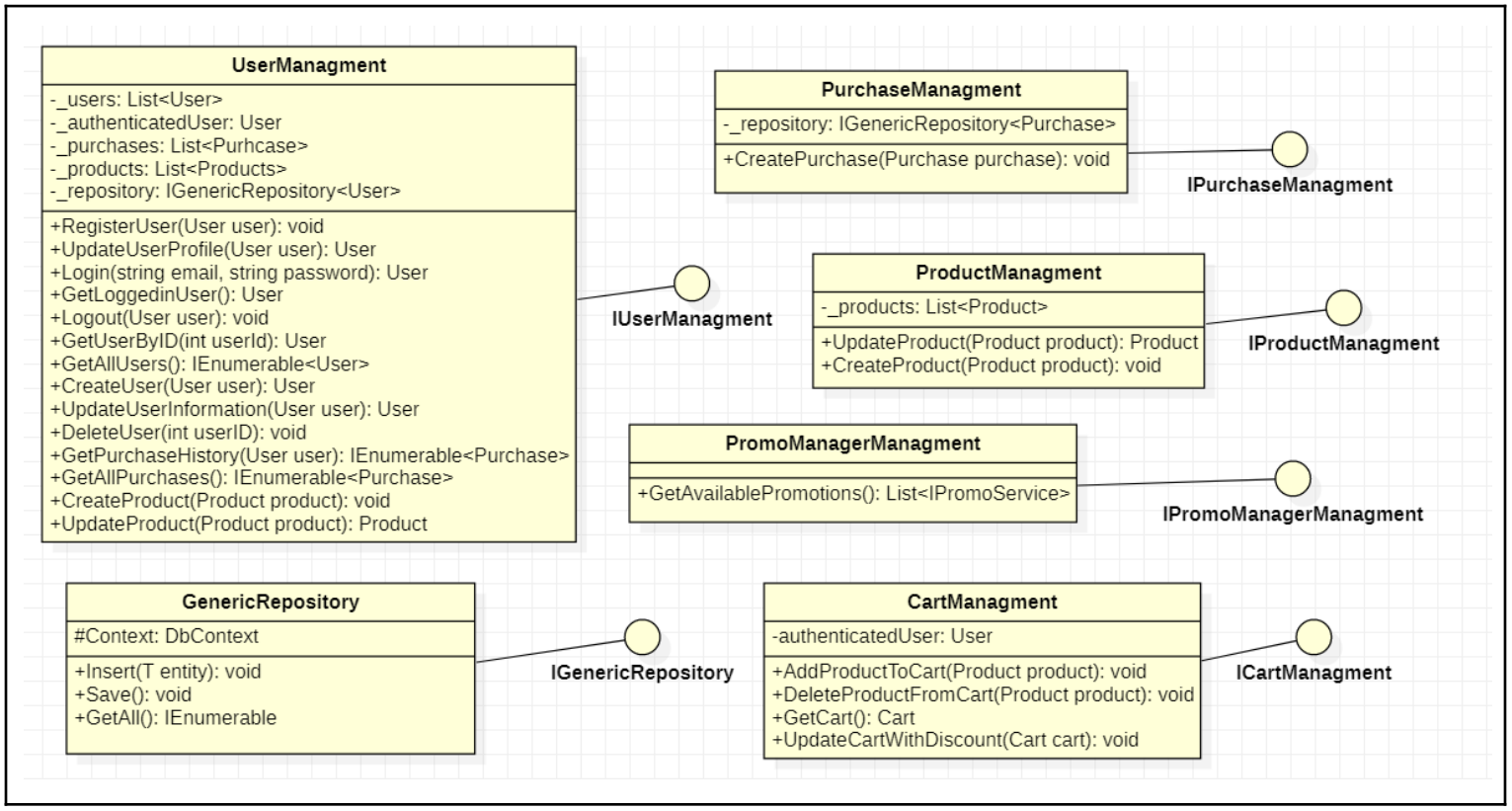
En BusinesssLogic.Test, realizaremos todos los tests asociados a Business Logic.

Paquete Obligatorio1.DataAccess

Una de las principales responsabilidades de DataAccess, es la gestión del contexto de la base de datos con la que vamos a trabajar. Contiene las implementaciones para las operaciones descritas en las interfaces de IDataAccess.

El siguiente diagrama de clases muestra las clases del proyecto Obligatorio1.DataAcces, más específico la carpeta Repositories. Además de sus atributos y

métodos, se muestra para cada una de ellas la interfaz (en todos los casos ubicadas en el proyecto Obligatorio1.IDataAccess) que implementa.



Paquete Obligatorio1.IDataAccess

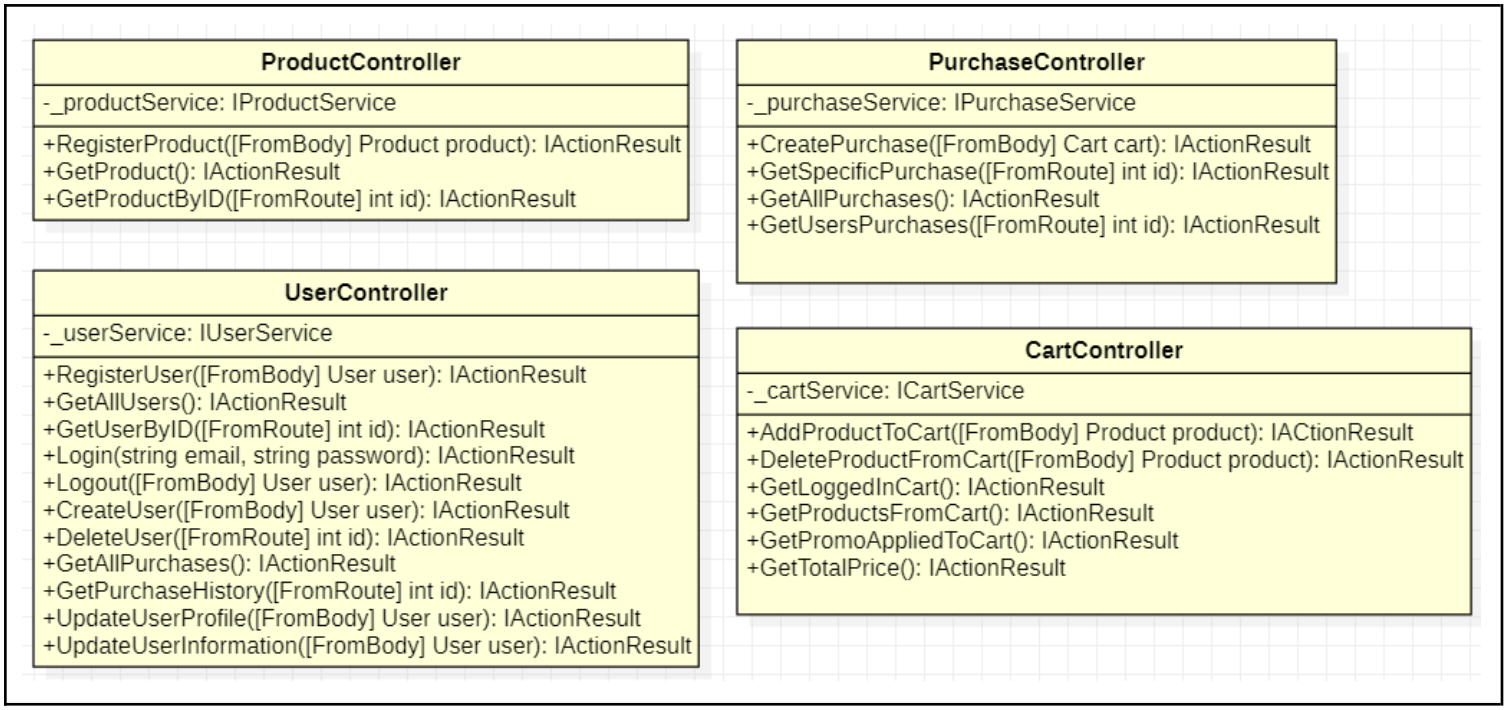
IDataAccess contiene las interfaces implementadas en DataAccess. Al hacer uso de interfaces, nos permitió generar una abstracción de la lógica de ciertas funciones que no son necesarias para la manipulación de datos en la base de datos.

Paquete Obligatorio1.DataAccess

En DataAccesTest, realizaremos todos los tests asociados a DataAccess.

Paquete Obligatorio1.WebApi

Dentro del proyecto Obligatorio1.WebApi encontraremos los controladores. Los dividimos en cuatro clases, cada una haciendo referencia a los endpoints relacionados a una clase del dominio.



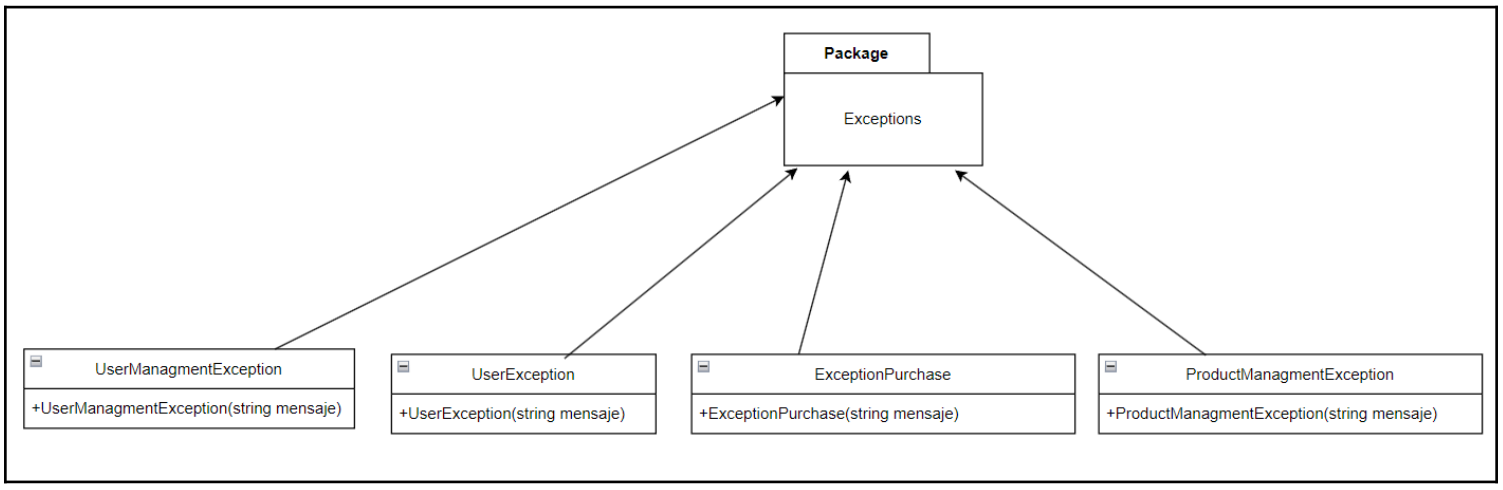
Los 4 controllers de la solución heredan la clase abstracta ControllerBase (perteneciente a Microsoft.AspNetCore.Mvc) para poder definir acciones, enrutamiento y gestionar solicitudes y respuestas HTTP.

Paquete Obligatorio1.WebApi.Test

En Obligatorio1.WebApi.Test, realizamos todos los tests asociados a los endpoints de los controladores en la WebApi.

Paquete Obligatorio1.Exceptions

Este paquete contendrá clases definidas para distintas excepciones. Cada una de estas clases implementa una excepción para cada objeto definido en el obligatorio. A modo de ejemplo, para los usuarios se definió la excepción UserExceptions.



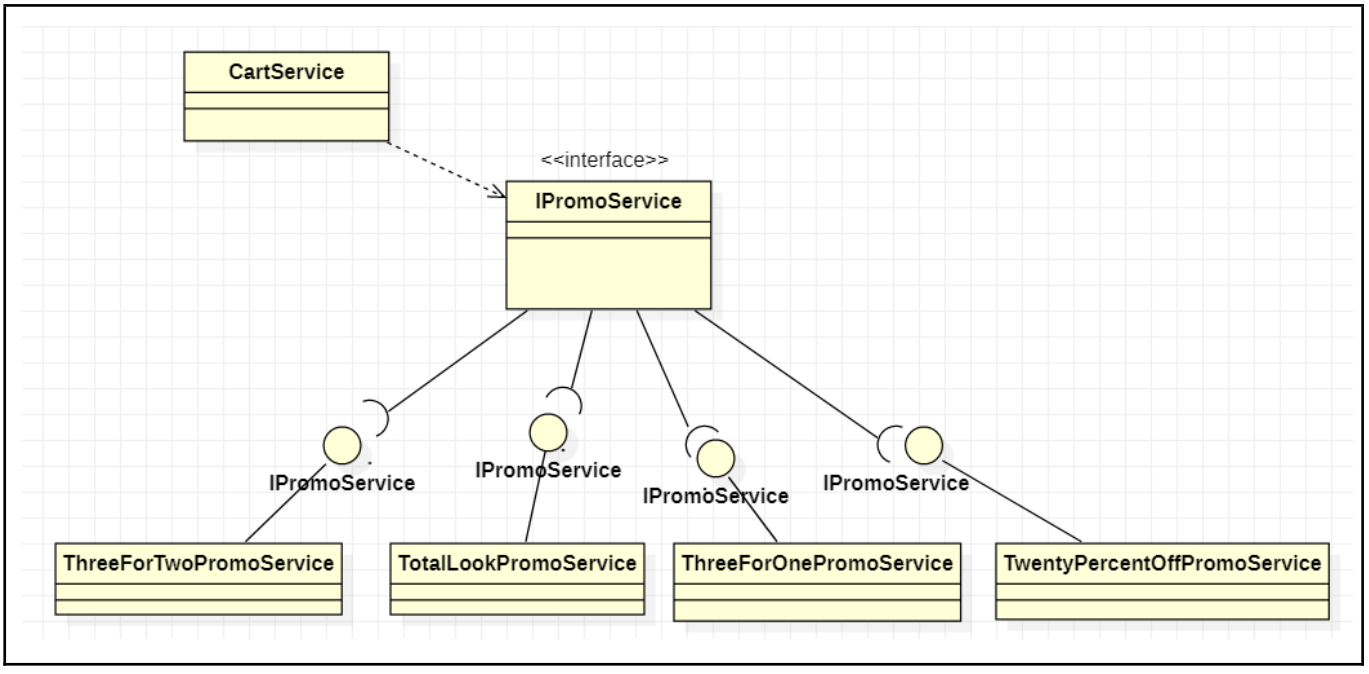
Jerarquías de herencia

La estructura de herencia en nuestro proyecto se basa principalmente en el uso de interfaces. A lo largo del trabajo, nos enfocamos en utilizar interfaces para separar la lógica de las operaciones de su implementación. Esto nos permite tener un código más flexible y fácil de mantener. Esta técnica se aplicó tanto en la parte de la lógica de negocio como en el acceso a datos, donde cada clase tiene una interfaz asociada. De esta manera, podemos realizar ciertas operaciones en todo el proyecto sin necesidad de conocer los detalles de cómo se ejecutan.

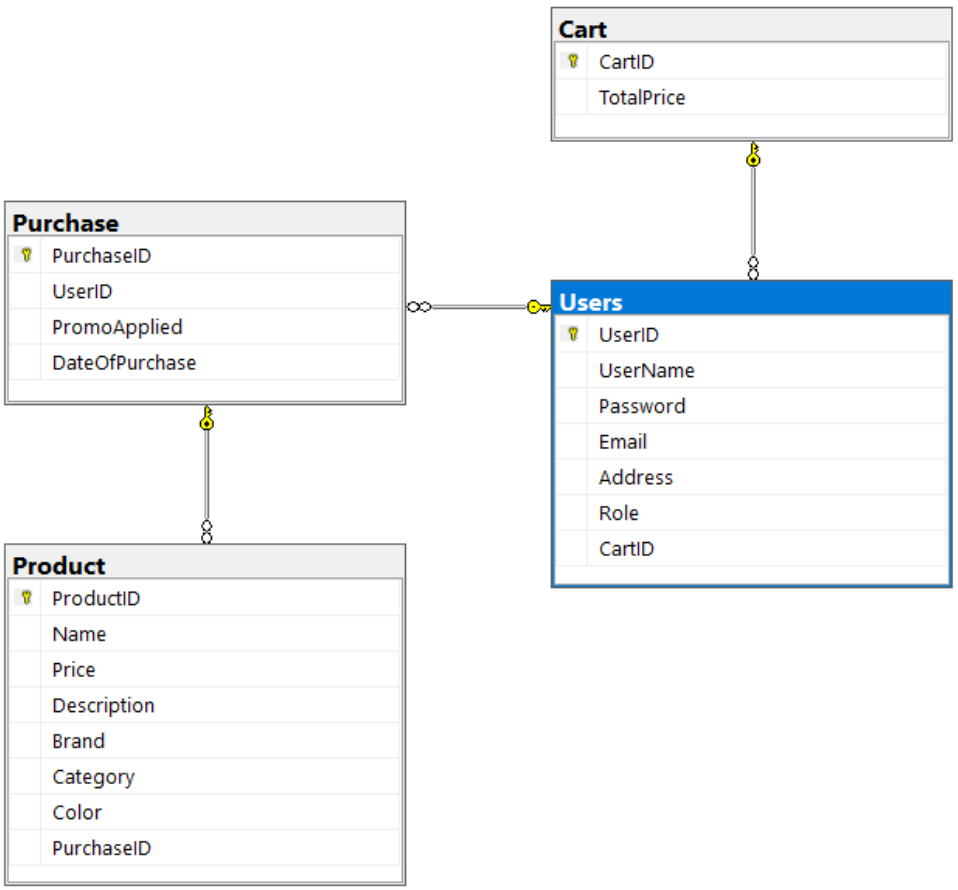
Un ejemplo interesante en el cual se aplicó esta técnica es en el manejo de las promociones. Implementamos una única interfaz llamada IPromoService, que contiene un método para calcular el precio del carrito aplicando una promoción. Esta interfaz es implementada por cuatro clases de lógica de negocio, cada una de ellas encargada de calcular el precio final del carrito según el tipo de promoción que representa. Al momento de aplicar la mejor promoción al carrito de compras después de haber agregado o eliminado un producto, se comparan los resultados que devuelven cada una de las promociones vigentes de la base de datos, todas del tipo IPromoService. Estos resultados se obtienen mediante la llamada al método en común instanciado en la interfaz, sin necesidad de conocer cómo están implementados los algoritmos internamente.

En este caso, seguimos el patrón de diseño "strategy", que sugiere crear una interfaz única con las funcionalidades que necesitamos y permitir que cada clase que la implementa desarrolle su propia implementación según sus necesidades específicas. Esto nos ha ayudado a mantener un código más organizado y fácil de extender en nuestro proyecto.

En el siguiente diagrama de clases se puede visualizar claramente el uso del patrón. Se representan las clases que participan en esta jerarquía, al igual que la clase CartService que es la que hace uso de la misma y se ve beneficiada por el uso de interfaces.



Modelo de tablas



Diagramas de interacción

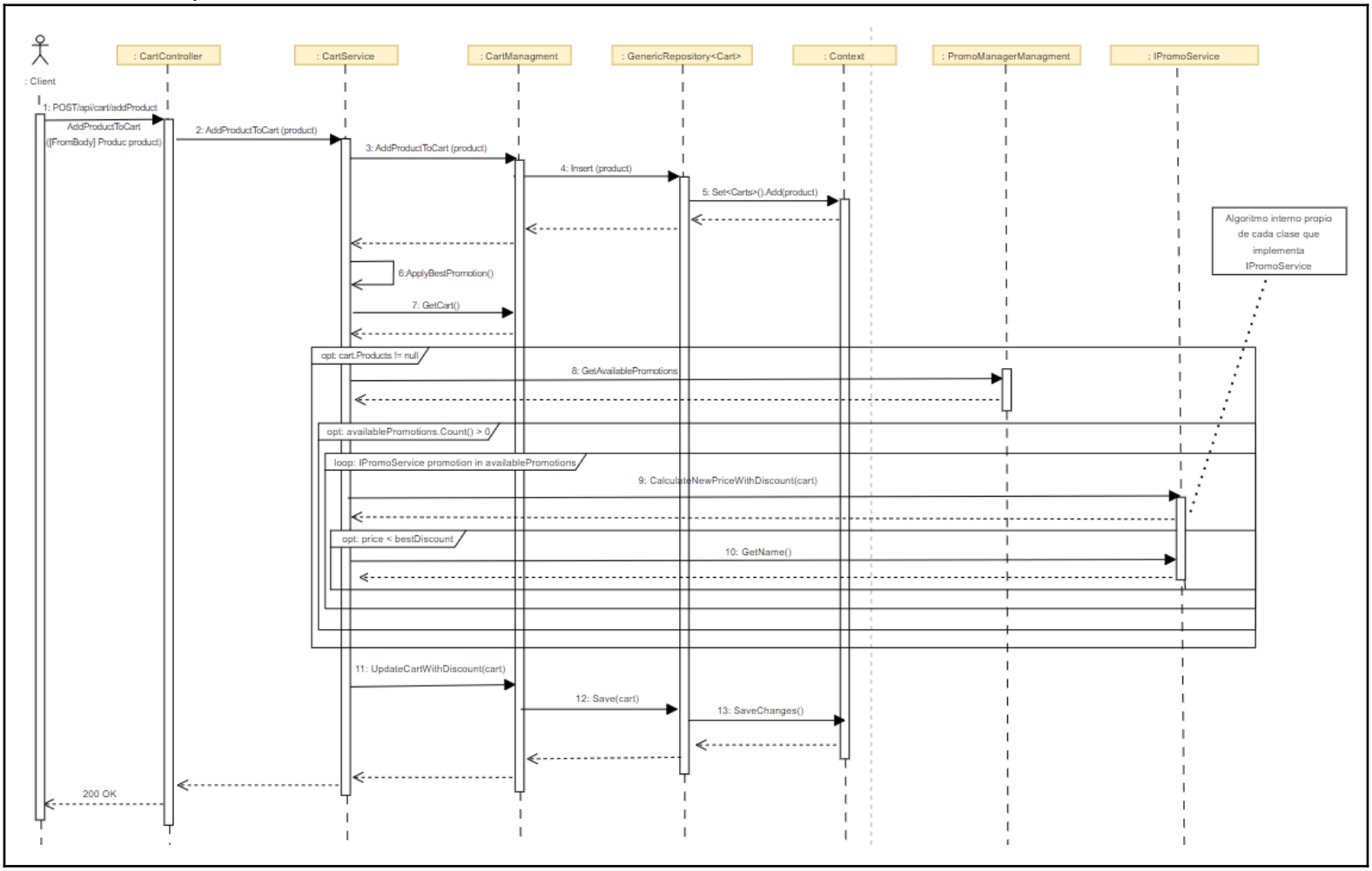
Añadir un producto al carrito

Se decidió analizar y diagramar el caso de uso de agregar un producto al carrito y cómo esta acción del usuario da comienzo a la comunicación entre las distintas capas del sistema y sus clases. Mediante el diagrama de interacción se puede visualizar todo el proceso completo desde que el cliente desea agregar un producto al carrito mediante la operación POST y como a partir de eso, el controlador del carrito en la capa de Web Api se comunica con la capa de lógica de negocio (más específicamente la clase CartService) y luego con la capa de acceso a datos. Se decidió también incluir el algoritmo para calcular la mejor promoción como proceso que sucede cada vez que se inserta o elimina un producto del carrito.

La capa de lógica de negocio se comunica con la de acceso a datos y esta con la base de datos dos veces durante este proceso: una vez cuando se agrega el producto y otra vez cuando se actualiza el carrito del usuario.

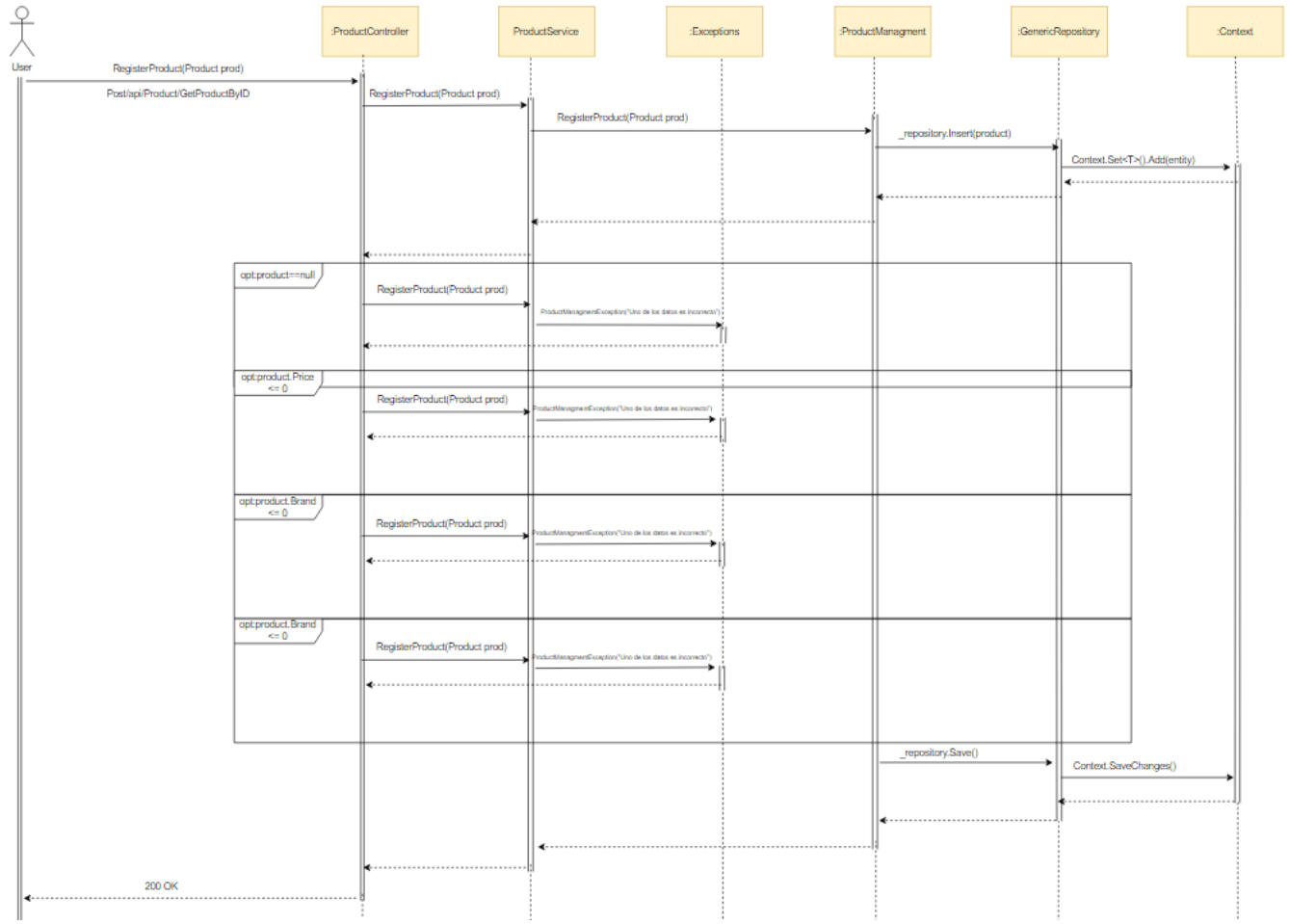
Los algoritmos específicos de cómo cada promoción calcula el nuevo precio del carrito no se incluyeron en este diagrama debido a la complejidad de los mismos y a que no resulta interesante representarlos en un diagrama de interacción ya que se ubican en clases que no interaccionan con las demás de la solución.

En el diagrama se muestra la interacción entre las clases para el caso en que la request es exitosa.



Registrar un producto

El segundo diagrama, se optó por realizar el caso de uso registro de un producto en el sistema, pasando por la base de datos e interactuando con las distintas capas del proyecto. Cuando el usuario realiza POST para realizar la solicitud de registro, interactuando con Web Api y las capas de BusinessLogic con DataAccess. En este caso, se muestra un ejemplo de registro válido, el cual queda persistido en la base de datos (GenericRepository e interactuando con el context). Si todos los datos del producto son correctos, se envía una respuesta 200 OK al usuario indicando que se logró registrar al producto.
Adicionalmente se muestra cómo se interactúa con casos de datos erróneos, los cuales se lanzarán excepciones indicando cuál fue la equivocación.



Justificación del diseño

Para esta entrega, fue utilizado uno de los patrones vistos en clase, el cual es strategy. Strategy nos permite determinar y controlar el intercambio de mensajes entre objetos. Específicamente con strategy buscamos que el objeto "cliente" tenga a disposición prestaciones o algoritmos (funciones) de las cuales pueda optar por implementar.

Ahora llevando este patrón a nuestro proyecto, fue utilizado para la creación y utilización de las distintas promociones (threeForTwo, TrheeForOne, ToltalLook y 20%Off), las cuales según el comportamiento que tenga la compra (Purchase) se debe implementar un tipo específico de promoción, en particular, la más conveniente.
Dentro de la sección jerarquías de herencias, fue puesto a disposición un diagrama de su implementación con una breve descripción del comportamiento.

Por otro lado, se siguieron los principios SOLID. Se respetó el Single Responsibility Principle asignando una única responsabilidad a cada clase y manteniendo así una única

razón por la cual cada una debería cambiar. El Open Close Principle, fue aplicado también, visible en el tema de las promociones, creando un código que es totalmente abierto a la extensión permitiendo que se agreguen futuras promociones y cerrado a la modificación ya que no es necesario cambiar el código existente para extender la cantidad de promociones. En el mismo contexto de las promociones, se respeta Liskov Substitution Principle, permitiendo que las clases hijas (las 4 clases que implementan los servicios de las promociones) cumplan con las especificaciones del padre (interfaz). Todas implementan codigo sin violar el contrato que establece la clase padre. Respecto a Interface Segregation Principle, en este trabajo se tuvo especial atención a ese principio, generando una gran colección de interfaces, cada una personalizada para cada entidad/ necesidad del cliente. Por último, el Dependency Inversion Principle también se cumple generando una abstracción entre los módulos de bajo nivel y los de alto nivel. La lógica de negocio se comunica con la lógica de acceso a datos a partir de interfaces, de manera que no dependa directamente de los detalles de implementación.

Consideraciones sobre el obligatorio:

- Se interpretó que los productos podían tener un color en string.
- Los atributos brand y category, son registrados como ints, ya que hacen referencia al ID en la base de datos. A modo de ejemplo una brand de valor 1 tendrá en la base de datos ID con valor 1 y nombre de la marca nike. Lo mismo para category, que en la base de datos tendrá el ID y un nombre como artículos de limpieza.
- Al insertar objetos en la base de datos mediante Postman, no se debería mandar como atributo el ID de esos objetos ya que la base lo autogenera.

Se optó por la utilización de excepciones, para continuar con las buenas prácticas de Clean Code. Para ello, se creó el paquete Excepciones, el cual contiene un conjunto de ellas, pensadas para implementar en distintos sectores del proyecto. Fueron creadas excepciones para los usuarios, los productos y las compras, las cuales son utilizadas en varios sectores del código, como puede ser BusinessLogic, DataAccess y WebApi. La variación que se tiene en cada excepción, es el mensaje que este devuelve, dependiendo para que se desee implementar (capturar una acción específica para devolver un mensaje específico).

Diagrama de componentes

A continuación se muestra el diagrama de componentes de la solución, proporcionando una vista de alto nivel de la arquitectura del sistema. El mismo ayuda a comprender la organización de los módulos que componen el proyecto y cómo unos utilizan a los otros.

Se muestran las relaciones de dependencia entre los componentes, indicando qué componentes se basan en otros para su funcionamiento:

- La Web Api utiliza las interfaces instanciadas en IBusinessLogic para desarrollar sus operaciones de interacción con el usuario.

- IBusinessLogic hace uso de las interfaces proporcionadas por IDataAccess para establecer la conexión entre la lógica del negocio y el acceso a la base de datos.

- Tanto DataAccess como BusinessLogic dependen de los módulos que proveen las interfaces respectivas para cada uno de ellos.

- BusinessLogic también depende del dominio y las excepciones.

- Ambos módulos de Test dependen de los módulos que están testeando, en este caso, WebApi.Test de WebApi y BusinessLogic.Test de BusinessLogic.

La solución se dividió en dichos componentes para mejorar la modularidad, la mantenibilidad y la escalabilidad del sistema. Esta organización nos permitió un desarrollo más eficiente al separar las preocupaciones, ya que cada componente tiene una responsabilidad clara y definida en el proyecto.

El diagrama de componentes sirvió como herramienta de visualización de nuestro sistema para comprender bien la comunicación entre todo los módulos que por momentos se hacía medio confuso. Esto permitió un mayor comprendimiento del mismo y contribuyo al desarrollo más eficiente.