

Facultad de Ingeniería

Universidad ORT

Descripción del diseño

Diseño de aplicaciones 2 Obligatorio 2

Martin Aristov (214634)

Gabriel Guerra (276712)

Felipe Heredia (297297)

https://github.com/IngSoft-DA2-2023-2/297297_276712_214634

16 de noviembre de 2023

Índice

Índice.....	2
Descripción general.....	4
Diagrama general de paquetes.....	5
Responsabilidades de cada paquete.....	6
Web API.....	6
BusinessLogic.....	7
BusinessLogic.Interface.....	8
DataAccess.....	8
DataAccess.Interface.....	9
Factory.....	9
Entities.....	9
Exceptions.....	10
Reflection.....	10
Jerarquías de herencia.....	12
Estructura de la base de datos.....	12
Diagramas de interacción.....	13
Login.....	13
Create Brand Discount.....	14
Diagrama de componentes.....	14
Justificación del diseño.....	15
Arquitectura de capas.....	15
Uso de interfaces.....	16
Web API.....	16
Entity Framework.....	17
GRASP.....	17
SOLID.....	18
Mecanismos utilizados para mejorar la extensibilidad.....	18
Reflection.....	18
Inyección de dependencias.....	19
Acceso a datos.....	19
Login.....	19
Manejo de Excepciones.....	20
Manejo de errores en WebAPI.....	20
Informe de métricas.....	21
Cohesión Relacional.....	21
Inestabilidad.....	22
Abstracción.....	23
Gráfico Inestabilidad - Abstracción.....	24
Desvío de los paquetes respecto a la secuencia principal.....	25
Grafo de dependencias.....	27

Principales cambios respecto a la versión anterior.....	28
Problemas conocidos.....	28
Anexo.....	28
Informe de cobertura.....	33
Referencias Bibliográficas.....	34
Declaración de Autoría.....	35

Descripción general

El proyecto consiste en un sistema cuyo propósito es permitir a las tiendas de ropa migrar al comercio electrónico, implementando también las promociones que se utilizan en las tiendas físicas. Debido a los cambios frecuentes en las estrategias promocionales de las tiendas, se espera que el sistema sea fácil de adaptar a dichos cambios dinámicos que se puedan querer hacer. Por este motivo se intenta hacer un sistema fácil de mantener, entender y escalar.

El sistema ha sido desarrollado en C#, utilizando el framework ASP.NET Core 6.0, siguiendo una arquitectura API REST, y utilizando Entity Framework Core para la persistencia de datos.

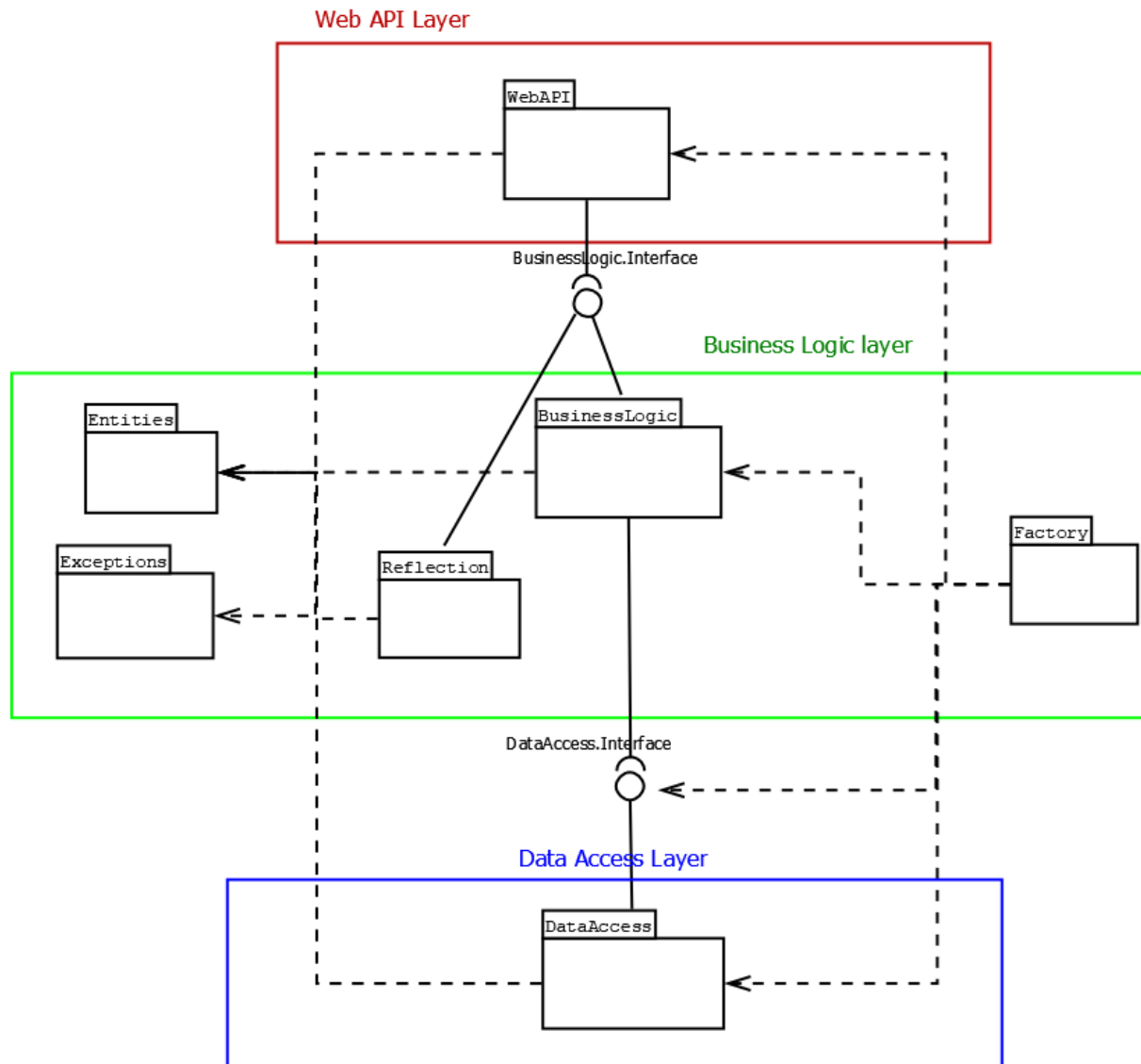
Esta versión cuenta con toda la lógica de negocio de la aplicación, las operaciones de data access y la API mediante la cual se pueden hacer solicitudes al sistema. Además esta versión del sistema cuenta con un Frontend hecho en el framework Angular, a partir de éste los usuarios pueden usar las funcionalidades del sistema.

Ejemplos de estas funcionalidades son: Loguearse, comprar productos, a partir de una lista de productos calcular los descuentos aplicables automáticamente, aplicar promociones según el método de pago, crear descuentos, modificar descuentos, eliminar descuentos, crear productos.

En este proyecto se ha utilizado la metodología TDD en todo el sistema, se han seguido las prácticas de código presentadas por Clean Code y se ha utilizado Git para el control de versiones del proyecto.

Diagrama general de paquetes

El diagrama general de paquetes del proyecto es el siguiente. No se incluyen proyectos de test.

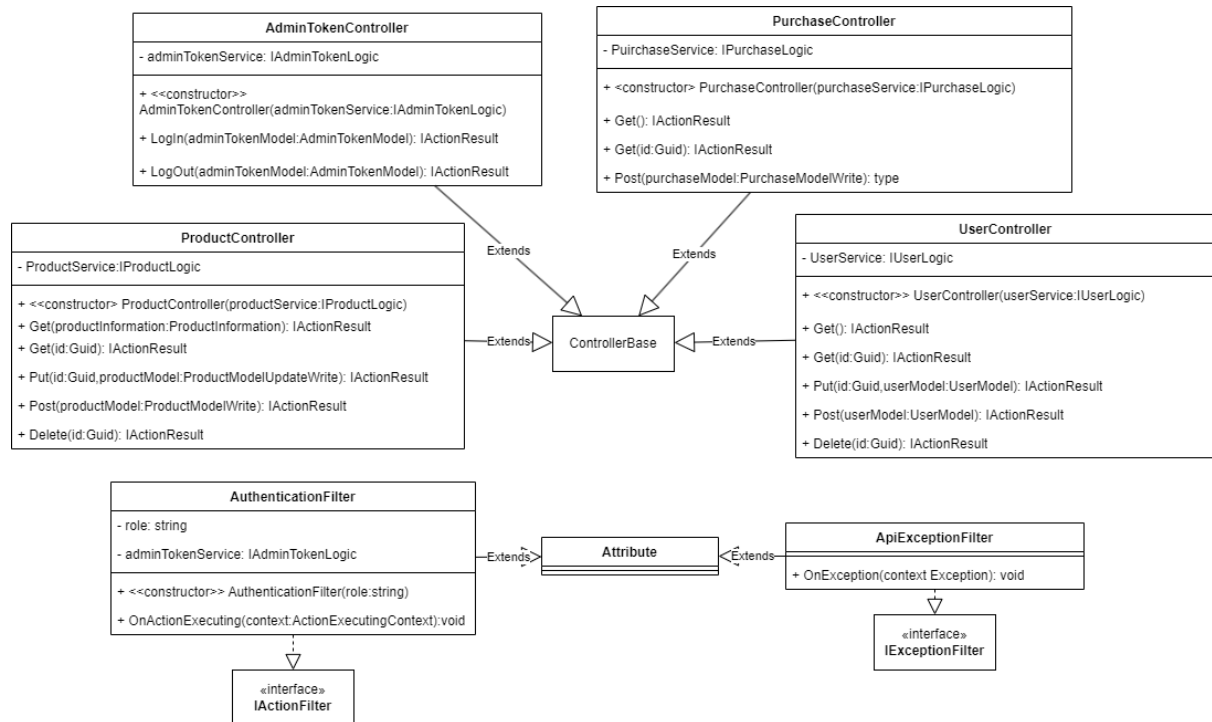


Responsabilidades de cada paquete

Web API

Aquí están los controladores que se encargan de recibir y responder a las solicitudes que se hacen al sistema, llamando a las funciones de lógica de negocio que correspondan para procesar la información de entrada y emitir una salida.

También se hace cargo de los códigos de estado y códigos de error que se envían al usuario cuando se responde a una solicitud.

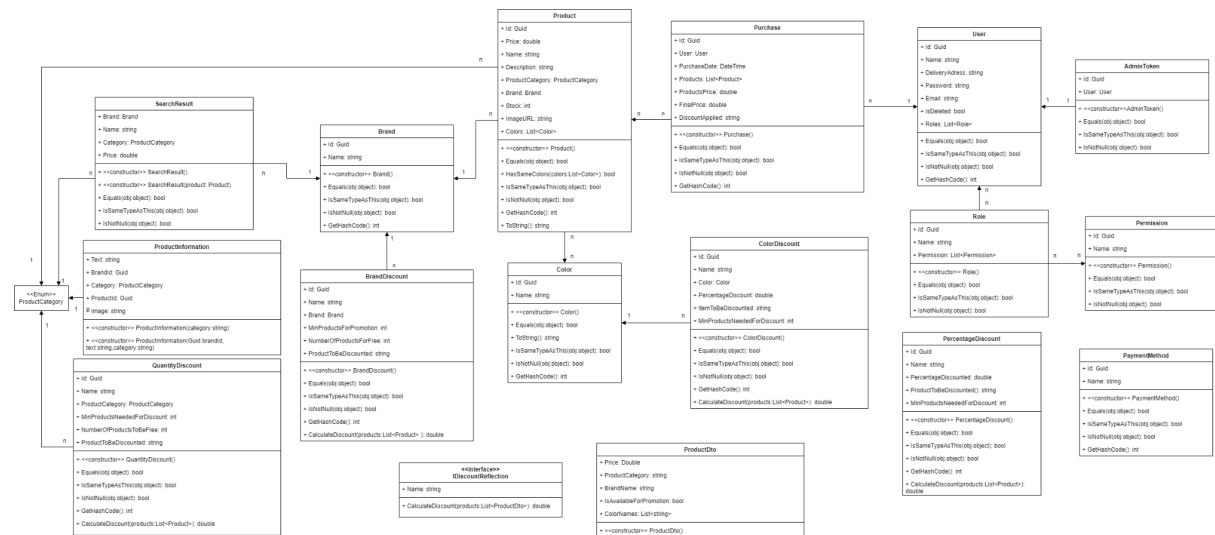


BusinessLogic

Las clases de este paquete se encargan de todo el procesamiento de la información y todas las operaciones relacionadas a la lógica de negocio. Esta capa se encarga de llamar a DataAccess cuando necesita almacenar, recuperar, modificar o eliminar información relacionada con el sistema.

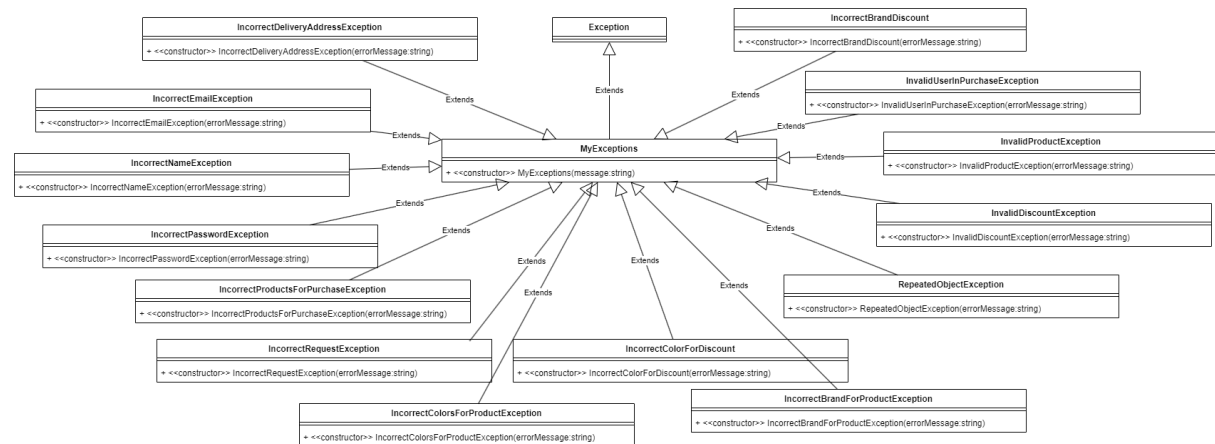
Entities

Este paquete cuenta con las diferentes clases del dominio del sistema.



Exceptions

Este paquete contiene todas las excepciones utilizadas por el sistemas.



Cada paquete debe tener una breve descripción de responsabilidades y un diagrama de clases.

Reflection

Este paquete es el que nos permite hacer Reflection en el proyecto. Esto significa poder agregar archivos dll en tiempo de ejecución, permitiendo añadir nuevos tipos de descuento al sistema sin necesidad de cerrar el servidor para recompilar toda la solución, lo cual sería muy costoso al implicar que el servidor estaría fuera de servicio durante dicho cambio. Para esto, se expuso la siguiente interfaz que es la que debe implementar cualquier desarrollador que quiera utilizar esta feature:

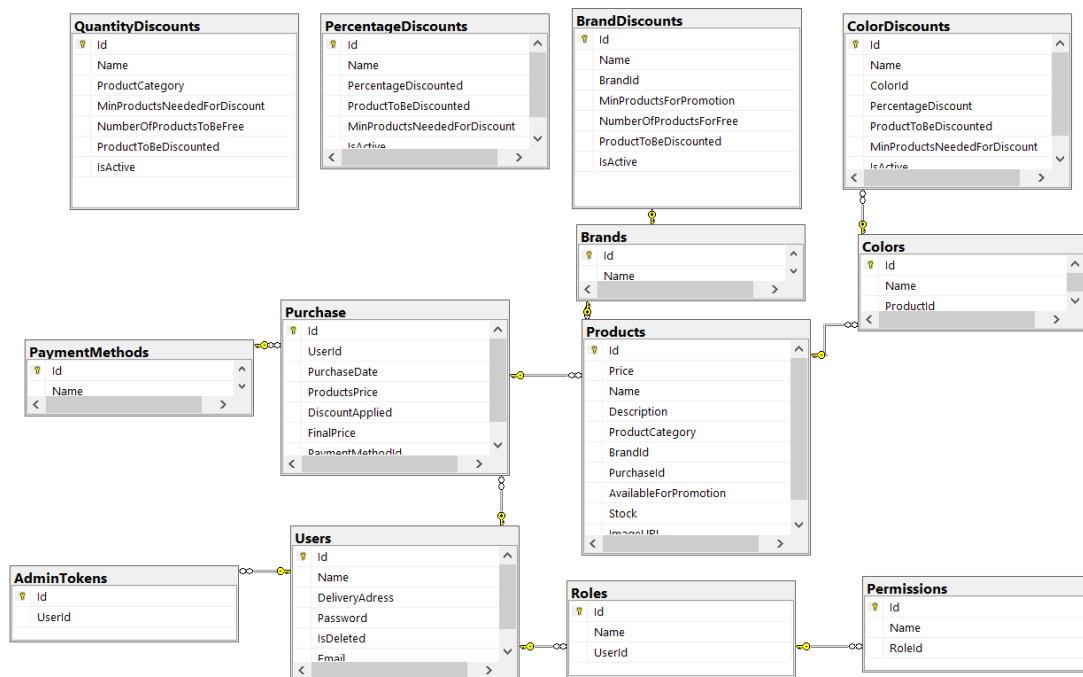
```
namespace Entities
{
    // references | Martin Aristov, 30 days ago | 1 author, 1 change
    public interface IDiscountReflection
    {
        // 1 reference | Martin Aristov, 30 days ago | 1 author, 1 change
        public string Name { get; set; }
        // 1 reference | Martin Aristov, 30 days ago | 1 author, 1 change
        public double CalculateDiscount(List<ProductDto> products);
    }
    // 6 references | Martin Aristov, 30 days ago | 1 author, 1 change
    public class ProductDto
    {
        // 1 reference | Martin Aristov, 30 days ago | 1 author, 1 change
        public double Price { get; set; }
        // 1 reference | Martin Aristov, 30 days ago | 1 author, 1 change
        public string ProductCategory { get; set; }
        // 1 reference | Martin Aristov, 30 days ago | 1 author, 1 change
        public string BrandName { get; set; }
        // 1 reference | Martin Aristov, 30 days ago | 1 author, 1 change
        public bool IsAvailableForPromotion { get; set; }
        // 1 reference | Martin Aristov, 30 days ago | 1 author, 1 change
        public List<string> ColorNames { get; set; }
        // 1 reference | Martin Aristov, 30 days ago | 1 author, 1 change
        public ProductDto() { }
    }
}
```

Como se ve, se debe proporcionar una clase que implemente el **Name** para el descuento, como así el método **CalculateDiscount**, en base a una clase **ProductDto** que se encuentra allí mismo. Se decidió exponer también una versión reducida de la clase producto, ya que no todos los atributos de producto son relevantes y/o útiles para calcular el descuento. La clase **ProductDto** busca exponer solamente los atributos mínimamente necesarios para poder calcular un descuento en base a los productos existentes.

Jerarquías de herencia

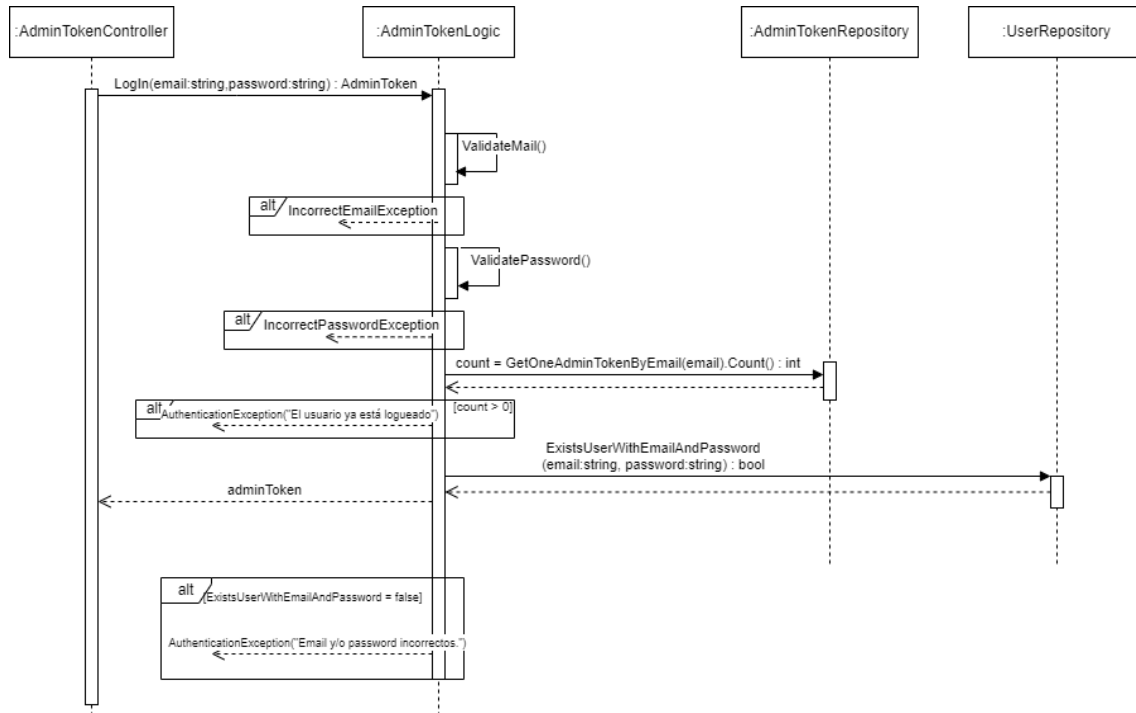
Se han utilizado algunas jerarquías de herencia para reutilizar código en el sistema. Por ejemplo todas las clases de tests heredan de TestSetUps, todas las excepciones heredan de la clase MyExceptions y las clases de lógica de los diferentes tipos de descuentos heredan de una clase llamada DiscountLogic.

Estructura de la base de datos



Diagramas de interacción

Login



Create Brand Discount

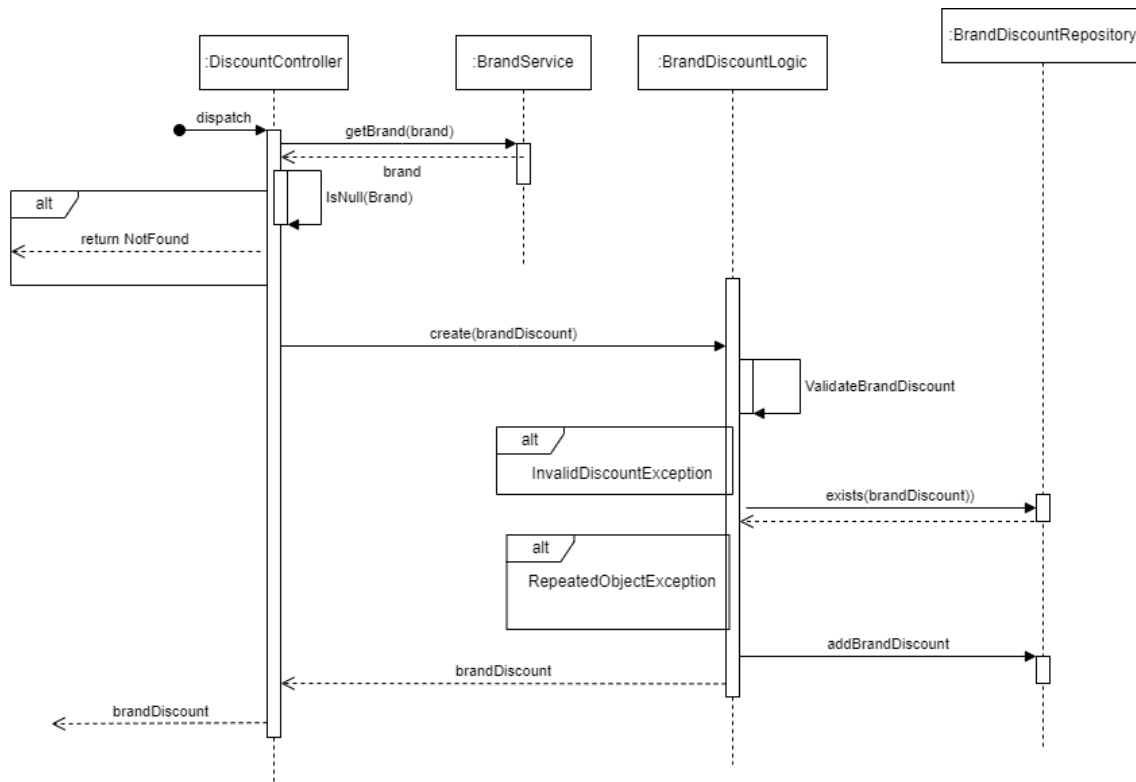


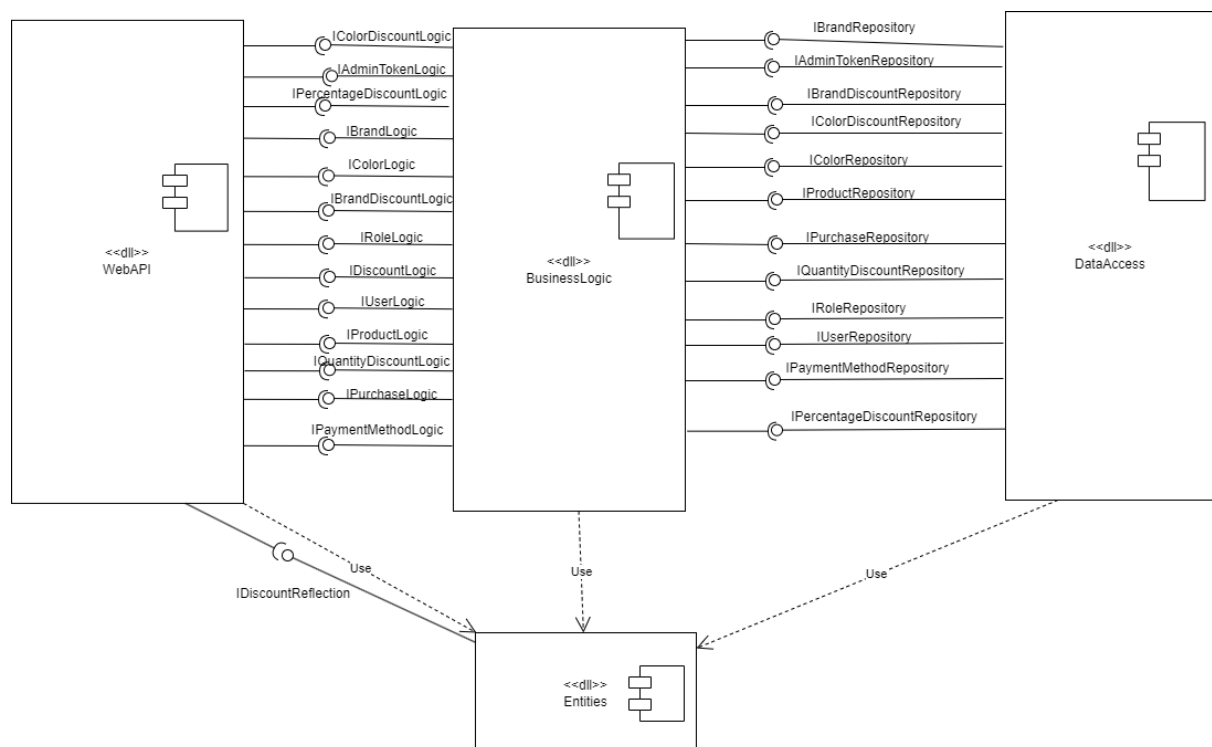
Diagrama de componentes

La solución usa una arquitectura en base a capas, en consecuencia de eso es posible ver cómo los componentes se ajustan a ese tipo de estructura. El componente de DataAccess se encarga de hacer las operaciones de comunicación con la base de datos.

DataAccess provee interfaces a BusinessLogic que contiene la lógica de las operaciones del negocio como crear, modificar, calcular, verificar, etc. BusinessLogic provee a la WebAPI de interfaces con lógica de negocio, al mismo tiempo Entities expone una interfaz con la implementación de Reflection, éstas interfaces son proveídas a la WebAPI mediante inyección de dependencias.

WebAPI hace uso de dichas interfaces para atender las solicitudes entrantes al servidor.

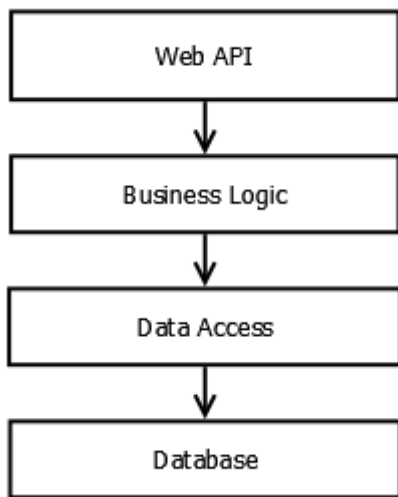
Entities contiene las entidades de la solución y a la interfaz de Reflection.



Justificación del diseño

Arquitectura de capas

Se ha creado el sistema utilizando una arquitectura por capas, esto aumenta la mantenibilidad de la aplicación, permite aislar diferentes partes del sistemas aumentando la cohesión del sistema y que se comporte de forma controlada. Separando el sistema en capas, se busca que cada capa cumpla el principio de responsabilidad única, encapsulando las funciones y permitiendo aislar los errores para que no afecten a toda la aplicación.



Uso de interfaces

Se utilizan interfaces de negocio entre capas para ocultar el comportamiento de cada una a las demás, esto nos permite un mejor aislamiento de las capas, lo cual a su vez conlleva a una mejor mantenibilidad en el proyecto. Esto es debido a que los errores se propagan con menos facilidad entre las capas, haciéndolos más localizables, y además permite a los programadores ver el programa como un conjunto de piezas diferentes y bien delimitadas con la complejidad dividida entre ellas, en lugar de una única sola pieza con enorme complejidad.

Se utilizan interfaces entre capas para desacoplar el sistema, permitiendo mayor flexibilidad y mantenibilidad en el proyecto.

Web API

En el paquete de Web API se usan controladores, con estos se crean los endpoints para procesar las solicitudes HTTP que los clientes enviarán al programa. Las solicitudes son primero procesadas por los filters que han sido establecido en el sistema con el propósito de quitarle carga a las capas inferiores, ejemplos de estos filtros son AuthenticationFilter para validar que una autenticación sea válida y ApiExceptionHandler para atrapar excepciones que no hayan sido procesadas en otras funciones.

Para procesar las solicitudes, la WebAPI llama a las funciones de BusinessLogic mediante las interfaces que corresponden según cada caso.

Entity Framework

Se utiliza Entity Framework para la persistencia de datos, el cual está diseñado para trabajar en conjunto con tecnologías como C# y .NET Framework, nos ofrece una abstracción de la base de datos que nos facilita la tarea de implementarla y mantenerla. Entity Framework realiza el mapeo automático de las tablas, automáticamente asociando los datos de la base de datos con objetos en C#. Para las consultas utilizamos LINQ, lo cual nos permite hacer consultas a la base de datos sin tener que hacer consultas SQL directamente.

El nivel de abstracción que EF y LINQ ofrecen hace que el manejo de la base de datos sea mucho más sencillo, más predecible y los problemas son más fáciles de manejar y detectar, mejorando significativamente la mantenibilidad.

GRASP

Varios de patrones GRASP son aplicados en el proyecto.

El patrón “Experto” se aplica constantemente en la solución, todas las capas del sistema tienen un propósito específico y todas las clases tanto de BusinessLogic

como de DataAccess y Web API. Por ejemplo, DiscountLogic es la clase experta en calcular el descuento óptimo y cada vez que se tiene que calcular dicho descuento se llama a dicha clase.

Se apunta a tener una alta cohesión y un bajo acoplamiento. Dividir el programa en múltiples clases con propósitos específicos hace que aumente la cohesión del sistema y reduce las dudas a la hora de entender la solución o introducir nuevas funcionalidades. También se implementan interfaces entre las capas del proyecto, reduciendo significativamente el acoplamiento de la solución, cambiar la implementación de las funcionalidades no debería afectar a los componentes que las usan debido a que dicha implementación está oculta detrás de la interfaz.

El patrón de controlador es usado en la capa de Web API, encargándose de procesar las solicitudes que entran al sistema.

SOLID

Se han implementado las clases procurando respetar el principio de SRP (Single Responsibility Principle), cada clase se dedica a una responsabilidad específica y cuando surge otra responsabilidad, se crea una clase nueva para esa responsabilidad.

Se cumple el principio Open/Closed porque es posible extender las funcionalidades del programa mediante herencia y polimorfismo.

Cuando se hace una herencia, las clases hijas se comportan como la clase padre, por lo tanto se cumple Liskov.

En la solución se utilizan interfaces para segregar clases entre sí, especialmente para separar aquellas clases que están en diferentes capas, por lo tanto cumple ISP (Interface Segregation Principle)

Mecanismos utilizados para mejorar la extensibilidad

Reflection

Se implementó Reflection, lo que permite a la solución analizar su estructura en tiempo de ejecución, permitiendo agregar archivos dll sin necesidad de recompilar. Usamos Reflection para permitir agregar nuevos descuentos sin necesidad de recompilar y volver a hacer el deploy, lo cual significa un costo muy grande al implicar dejar fuera de servicio la API temporalmente.

Inyección de dependencias

Otro mecanismo utilizado es la inyección de dependencias en la capa de WebAPI. Permite separar la API de la implementación de las funciones que se realizan en la lógica de negocio, mejorando la cohesión y la mantenibilidad.

Acceso a datos

Para el acceso a datos hicimos una capa llamada DataAccess, consiste en un paquete que se encarga de hacer todas las operaciones correspondientes a la base de datos, la capa de lógica de negocios se encarga de llamar a esta capa cuando necesita hacer una operación de ese tipo. Cuando el cliente quiera hacer una operación que involucre la base de datos, lo que va a hacer es enviar una solicitud a la Web API, la cual va a llamar a la lógica de negocio correspondiente, la cual va a llamar a Data Access si dicha operación es válida, la cual va a hacer dicha operación en la base de datos, cuando todo eso termina la Web API avisa al usuario del resultado o le da un código de error si corresponde.

Este diseño permite tener mejor control de lo que sucede, ayuda a reducir la complejidad, desacopla el sistema y el usuario no es capaz de interactuar directamente con la base de datos.

Login

El login se realizó Utilizando una Clase llamada “**AdminToken**”, la cual está relacionada con la clase User. A nivel de controladores, se requiere que un usuario

ingrese una combinación de email y contraseña correctos (esto es, que exista en la base de datos un usuario que tenga ese email y esa contraseña), para poder realizar el login. A su vez, cada usuario tiene asociado sus roles y sus permisos, los cuales le permiten o no realizar diversas acciones en el sistema, pero esto es validado por la clase Authentication al momento de realizar una request a algún endpoint, validando los permisos que este usuario tenga asociado y los requeridos por el recurso al que quiere acceder.

Manejo de Excepciones

Se han creado excepciones personalizadas para los casos donde se considera adecuado y se ha creado un paquete cuyo propósito es almacenarlas, todos los paquetes de la solución que las usan dependen de este paquete.

Las excepciones del programa son disparadas con throw en los paquetes de BusinessLogic y de DataAccess, y son atrapadas en el paquete de WebAPI con el objetivo de responder a las solicitudes con el código de error correspondiente.

Manejo de errores en WebAPI

Para responder a las solicitudes se utilizan códigos de estado de HTTP, cuando un error se dispara, se responde a la solicitud con un código de estado que se considera apropiado para el error en cuestión. Por ejemplo, cuando el cliente no está autenticado e intenta una solicitud se tira el código 401, si los datos ingresados tienen un formato incorrecto se envía el código 400, si se intenta utilizar un token inválido se envía el código 403, si se intenta acceder a un producto o recurso que no existe se tira un 404, cuando no hay errores se tiran códigos como 200 para situaciones generales y 201 para posts, y si ocurre una excepción no controlada se tira el código 500.

Informe de métricas

Para realizar el informe de las métricas, utilizamos la herramienta conocida como NDepend. Esta es una herramienta de análisis estático para el código administrado. Esta herramienta nos permite visualizar las dependencias mediante gráficos dirigidos y una matriz de dependencia.

Estas métricas son indicadores que permiten relevar medidas relacionadas a la calidad del software.

Es importante destacar que si las métricas dan dentro del rango esperado no significa que el diseño sea excelente. Lo mismo en el caso que las métricas no estén dentro de lo esperado, no significa que el diseño es malo.

	Afferent Coupling(Ca)	Efferent Coupling(Ce)	Abstractness	Instability	Dist. From main seq	Relational Cohesion
BusinessLogic.Interface	28	30	0.81	0.52	0.23	1.5
DataAccess.Interface	26	22	0.8	0.46	0.18	1
Reflection	7	31	0.2	0.82	0.011	1.6
WebApi	0	144	0.056	1	0.039	3.65
Entities	92	25	0.048	0.21	0.52	2.67
Exceptions	18	8	0	0.31	0.49	1.8
DataAccess	1	124	0	0.99	0.0057	1.83
BusinessLogic	1	104	0	0.99	0.0067	1.82
Factory	1	67	0	0.99	0.01	1.25
Migrations	0	10	0	1	0	1

Cohesión Relacional

La cohesión mide la relación entre las clases de un paquete indicando que tan cohesivo es. Lo ideal, es que las clases dentro de un mismo paquete estén fuertemente relacionadas entre sí, sin embargo, tampoco es bueno que estén excesivamente relacionadas ya que caeríamos en el caso donde tenemos sobre acoplamiento.

El indicador para determinar que su valor es adecuado se calcula de la siguiente forma: $H = (R + 1) / N$. Siendo R el número de relaciones internas del paquete y N el número de clases del paquete. Si H se encuentra entre 1.5 y 4.0, entonces el indicador nos dice que la cohesión relacional de nuestro paquete es adecuada, fuera de ese rango, no lo es.

En nuestro caso, los paquetes que pertenecen al rango adecuado son todos exceptuando los paquetes Migrations, Factory y DataAccess.Interface con valores de 1, 1.25 y 1 respectivamente. Respecto a estos paquetes que no entran dentro del llamado rango correcto, podemos decir que son paquetes que presentan muchas relaciones internas entre sí, por la cantidad de clases que hay dentro del paquete. En ambos casos de Migrations y DataAccess.Interface, podemos decir que en estos paquetes hay tantas relaciones internas como clases ($R+1 < N$). En líneas generales, obtenemos la conclusión que nuestros paquetes son en general cohesivos, exceptuando los 3 casos mencionados anteriormente.

Inestabilidad

La inestabilidad impacta en el esfuerzo requerido para cambiar un paquete, y se calcula de la siguiente forma: $I = \text{Acoplamiento eferente} / (\text{Acoplamiento eferente} + \text{Acoplamiento aferente})$. Siendo el acoplamiento eferente las dependencias “salientes” y el acoplamiento aferente las dependencias “entrantes”. El indicador de inestabilidad se encuentra entre 0 y 1.

Si nos acercamos a 1 ($I = 1$), entonces el paquete es muy inestable (ya que hay más dependencias salientes que entrantes), y si nos acercamos a 0 ($I = 0$), entonces el paquete es muy estable (ya que hay más dependencias entrantes que salientes).

Según la tabla anterior, WebApi y Migrations tienen una inestabilidad de 1, por lo cual tiene más relaciones salientes que entrantes, lo cual tiene sentido, ya que de estos dos paquetes no depende nadie, sino que ellos son los que dependen de otros paquetes. Por ejemplo, en el caso de webApi las relaciones Aferentes son 0, esto es, nadie depende de este paquete, lo cual es correcto. En cambio, WebApi si que depende de muchas clases, en específico, de todas las clases dentro de BusinessLogic para poder funcionar. Algo similar sucede con Migrations. Luego está el caso de DataAccess, BusinessLogic y Factory, que tienen una inestabilidad de 0.99 (básicamente 1 como en el caso de WebAp y Migrations mencionado anteriormente). A estos tres paquetes les sucede algo similar que a

WebApi y Migrations, y es que dependen de más paquetes que los que dependen de ellos.

Es interesante destacar el caso de Entities y Exceptions, que tienen una inestabilidad de 0.21 y 0.31 respectivamente, lo cual también tiene sentido ya que todos o casi todos dependen de ellos, pero ellos no dependen de nadie, lo cual los convierte en paquetes muy estables.

Abstracción

La abstracción es la relación entre el número de clases y las clases abstractas de un paquete. Nos indica qué tan abstracto es un paquete. A mayor estabilidad en un paquete, mayor abstracción, y, a mayor inestabilidad, más concreto es el paquete.

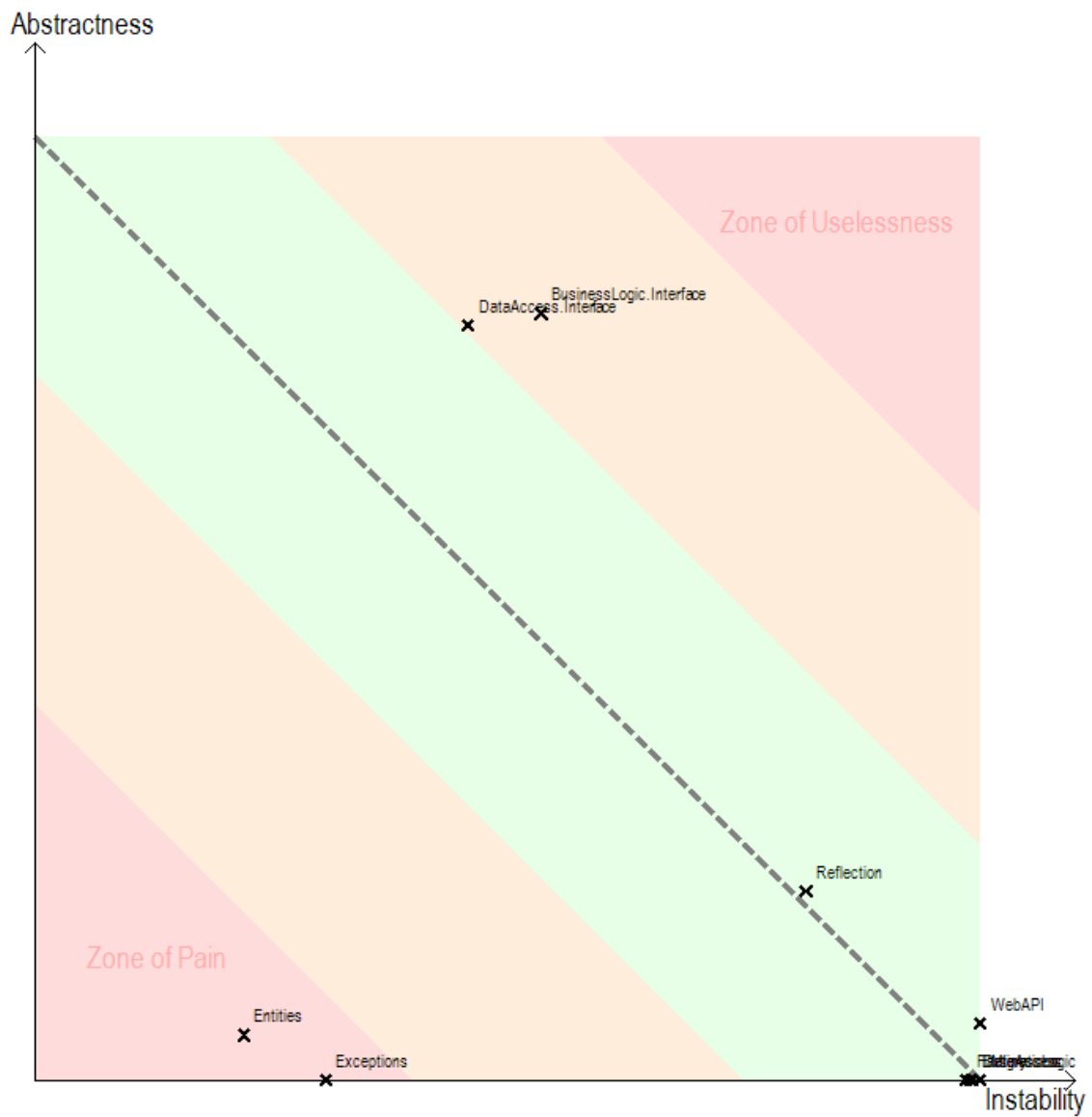
La abstracción se calcula de la siguiente manera: $A = N_a/N_c$, siendo N_a el número de clases abstractas e interfaces, y N_c el número de interfaces, clases abstractas y clases concretas. El valor de A se encuentra entre 0 y 1. Si nos acercamos a 0, entonces hay muchas clases concretas, y por ende, el paquete es más concreto. Si por otro lado, nos acercamos a 1, significa que existen pocas clases concretas, por lo que el paquete es más abstracto.

En nuestro caso, todos los paquetes cuentan con abstracción de 0 o un valor muy aproximado (como es el caso de Entities y WebApi que cuentan con una abstracción de 0.048 y 0.056, al igual que Reflection que cuenta con una abstracción de 0.2, pero ignoraremos estos paquetes para este análisis ya que sus valores son muy pequeños).

En el caso donde tenemos valores 0 o muy cercanos a ellos, se debe a que no existen abstracciones ni interfaces en el paquete. Sin embargo, en el caso de DataAccess.Interface e BusinessLogic.Interface, hacemos uso de interfaces, por lo cual es normal y esperable que los valores se aproximen mucho a 1.

Gráfico Inestabilidad - Abstracción

Para poder realizar un mejor análisis, se muestra una gráfica generada en NDepend donde en el eje x se muestra la inestabilidad en el eje y se muestra la abstracción.



El gráfico anterior muestra la relación que existe entre la métrica de abstracción y la de inestabilidad de los paquetes de nuestro sistema. Lo que se busca es que nuestros paquetes se encuentren lo más cerca posible de la línea punteada, llamada secuencia principal, ya que representa el equilibrio entre las métricas mencionadas.

Como se puede apreciar, existen dos sectores rojos. Por un lado, la zona denominada **“Zone of pain”** representa aquellos paquetes que si bien son muy estables, no son para nada abstractos, si no que por el contrario, son muy concretos. Por otro lado, si nos encontramos en la zona **“Zone of uselessness”**, quiere decir que si bien los paquetes son muy abstractos, no son para nada estables. Hay que evitar tener paquetes en estas zonas.

En el caso de los paquetes Exceptions y Entities se encuentran en la Zona del Dolor, ya que son paquetes muy estables, como se mencionó en secciones anteriores, pero para nada abstractos

Por otro lado en el caso de los paquetes DataAccess.Interface y BusinessLogic.Interface se encuentran muy próximos a la Zona de Inutilidad, dado que son paquetes que contienen muchas interfaces pero muy pocas clases abstractas(ninguna, en realidad). El resto de los paquetes se encuentran próximos a la secuencia principal, con un desvío específico que será mostrado a continuación, para cada uno de los paquetes.

Desvío de los paquetes respecto a la secuencia principal

El valor de D' nos indica que tan cerca de la secuencia principal se encuentra un paquete. Cuando el valor de D' es más cercano a 0 más cerca de la secuencia principal se encuentra el paquete. El valor de D' se calcula de la siguiente manera: $D' = |A + I - 1|$.

Valores cercanos a 0 (zona de dolor) indican que el paquete es concreto y estable (responsable). Estos paquetes no son buenos porque no son extensibles y si cambian impactan en otros. Valores cercanos a 1 (zona de poca utilidad) indican que el paquete es abstracto e inestable. El paquete es extensible pero tiene pocos paquetes que dependan de él. Recordemos la tabla al comienzo de esta sección y los valores de D (Dist. from main seq) para cada paquete.

	Afferent Coupling(Ca)	Efferent Coupling(Ce)	Abstractness	Instability	Dist. From main seq
BusinessLogic.Interface	28	30	0.81	0.52	0.23
DataAccess.Interface	26	22	0.8	0.46	0.18
Reflection	7	31	0.2	0.82	0.011
WebApi	0	144	0.056	1	0.039
Entities	92	25	0.048	0.21	0.52
Exceptions	18	8	0	0.31	0.49
DataAccess	1	124	0	0.99	0.0057
BusinessLogic	1	104	0	0.99	0.0067
Factory	1	67	0	0.99	0.01
Migrations	0	10	0	1	0

Como se puede ver, los paquetes que presentan una mayor diferencia son aquellos que se encuentran en la zona de dolor como Entities y Exceptions(0.52 y 0.49 respectivamente).

En el caso de los paquetes BusinessLogic.Interface y DataAccess.Interface, que como se mencionó anteriormente, se encuentran cerca de la zona de inutilidad, presentan una distancia respecto a la secuencia principal de 0.23 y 0.18, por lo cual están bastante próximos a la misma. Es más, si complementamos este número con la observación de la gráfica anterior, podemos asegurar que se encuentran a medio camino(aproximadamente) entre la secuencia principal y la zona de inutilidad, algo que no es malo del todo.

Respecto al resto de paquetes que no se mencionaron anteriormente, todos se encuentran realmente próximos a la secuencia principal, lo cual es bueno. Es interesante destacar el caso de Migrations que con una distancia de 0 se encuentra sobre la secuencia principal, al igual de Factory que tiene una distancia de 0.01, BusinessLogic con 0.0067 o DataAccess con 0.0057.

Principales cambios respecto a la versión anterior

- Se implementó Frontend de la aplicación. Permitiendo enviar solicitudes a la API del sistema desde el mismo.
- Se implementó Reflection para que sea posible agregar nuevos descuentos durante el tiempo de ejecución.
- Se agregaron endpoints a la API, los cuales están especificados en el anexo.
- Se agregó un atributo de stock a la entidad producto.
- Se agregó la lógica de los métodos de pago.

Problemas conocidos

- La cobertura de pruebas ha bajado desde la entrega anterior. Esto se debe a la adición de nuevos endpoints.
- Por otra parte, en la sección 'home', donde se muestran los productos que se encuentran registrados en la base de datos, el filtrado de ellos no funciona correctamente debido a un problema con la lógica implementada en el backend, pero por problemas de tiempo no fue posible solucionarlo.

Anexo

Además de los resources con los que contábamos anteriormente, se agregaron nuevos que se detallan a continuación.

Brand

GET /api/brands

Parameters

No parameters

Try it out

Responses

Code	Description	Links
200	Success	No links

Media type: text/plain

Controls Accept header

Example Value Schema

```
{  "id": "string",  "name": "string"}
```

GET /api/brands/{id}

Parameters

Try it out

Name	Description
id * required	string(\$uuid) (path)

Responses

Code	Description	Links
200	Success	No links
Media type: text/plain		
Controls Accept header.		
Example Value Schema		
<pre>{ "id": "string", "name": "string" }</pre>		
401	Unauthorized	No links
Media type: text/plain		
Example Value Schema		
<pre>{ "type": "string", "title": "string", "status": 0, "detail": "string", "instance": "string", "additionalprop1": "string", "additionalprop2": "string", "additionalprop3": "string" }</pre>		

Con la implementación del front nos dimos cuenta de que necesitábamos un endpoint para poder traer las Brand de la base de datos y poder mostrarlas, para eso son los dos endpoint anteriores

Color

GET /api/colors

Parameters Try it out

No parameters

Responses

Code	Description	Links
200	Success	No links

Media type:

Controls: Accept header

Example Value | Schema

```
{
  "id": "string",
  "name": "string"
}
```

GET /api/colors/{Id}

Parameters Try it out

Name	Description
id * required string(\$uuid) (path)	<input type="text" value="id"/>

Responses

Code	Description	Links
200	Success	No links
401	Unauthorized	No links

Media type:

Controls: Accept header

Example Value | Schema

```
{
  "id": "string",
  "name": "string"
}
```

Media type:

Controls: Accept header

Example Value | Schema

```
{
  "type": "string",
  "title": "string",
  "status": 0,
  "detail": "string",
  "instance": "string",
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

El mismo problema anterior nos surgió con Color, ya que antes no teníamos una forma de “traer”, o pedir, los colores que estaban registrados en la base de datos.

Discount

GET /api/discounts

Parameters

Try it out

Name	Description
productsIds	

string (query)

Responses

Code	Description	Links
200	Success	No links

Lo mismo sucede para este endpoint de Discounts, ya que fue implementado para poder así calcular y enviar información sobre el descuento que se debe aplicar desde la base de datos.

PaymentMethod

GET /api/payments

Parameters

Try it out

No parameters

Responses

Code	Description	Links
200	Success	No links

Media type

text/plain

Controls Accept Header:

Example Value | Schema

```
{
  "id": "string",
  "name": "string"
}
```

Diferente es el caso de PaymentMethod, ya que este fue un nuevo requerimiento para esta segunda entrega, y tuvimos la necesidad de implementar un Get para poder traer todos los PaymentMethod que se encuentren en el sistema.

Role

GET

/api/roles

Try it out

Parameters

No parameters

Responses

Code	Description	Links
200	Success	No links

GET

/api/roles/{Id}

Try it out

Parameters

Name	Description
id * required string(\$uuid) (path)	<div>id</div>

Responses

Code	Description	Links
200	Success	No links
401	Unauthorized	No links

Media type

text/plain

Example Value

Schema

```
{
  "type": "string",
  "title": "string",
  "status": 0,
  "detail": "string",
  "instance": "string",
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

También se implementaron 2 endpoint para role, y así poder validar en el frontend los roles y permisos que posee un usuario y poder decidir qué elementos mostrar y cuales no.

Informe de cobertura

arist_MARTINPC_2023-11-16.16_06_38.coverage	10082	3257	5016	24	4332	75.58%
▶ entities.test.dll	525	0	476	0	0	100.00%
▶ webapi.test.dll	1176	0	516	0	0	100.00%
▶ dataaccess.test.dll	784	8	604	2	7	98.99%
▶ businesslogic.test.dll	3262	60	1034	0	50	98.19%
▶ testsetup.dll	585	18	575	0	19	97.01%
▶ entities.dll	764	64	380	8	45	92.27%
▶ exceptions.dll	28	6	14	0	3	82.35%
▶ businesslogic.dll	795	269	524	2	138	74.72%
▶ webapi.dll	908	662	565	12	382	57.83%
▶ dataaccess.dll	1255	2112	328	0	3645	37.27%
▶ reflection.dll	0	58	0	0	43	0.00%

Como se puede ver en la imagen anterior, y como se comentó en problemas conocidos, el porcentaje de cobertura de las pruebas descendió respecto a la primera entrega, esto se debe básicamente a un problema con el tiempo, y que se priorizo implementar las funcionalidades del frontend al completo, descuidando por desgracia la cobertura de pruebas.

Test Explorer

Build succeeded

Test	Erro...
▶ ✓ BusinessLogic.Test (1...	
▶ ✓ DataAccess.Test (78)	
▶ ✓ Entities.Test (97)	
▶ ✓ WebApi.Test (36)	

Group Summary

BusinessLogic.Test

Tests in group: 100

⌚ Total Duration: 558 ms

Outcomes

✓ 100 Passed

De todos modos, se tienen 311 test funcionales.

Llegamos a desarrollar una cantidad de pruebas considerables las cuales pasan satisfactoriamente. Es preciso aclarar, que esto no es independiente de la sección anterior, donde se mencionaba el porcentaje de cobertura de las pruebas en el proyecto. Esto es un factor a mejorar a futuro, dado que en caso de realizarse más pruebas, relevantes para el proyecto, podríamos alcanzar el 100% de cobertura que es el objetivo.

Referencias Bibliográficas

- Repositorio Github del docente
<https://github.com/TOPOFGR>
- Material de aulas y el foro de DA2 de aulas
<https://aulas.ort.edu.uy/>
- Documentación oficial de Microsoft sobre .NET

<https://learn.microsoft.com/en-us/dotnet/framework/>

- Documentación oficial de Microsoft sobre Entity Framework

<https://learn.microsoft.com/en-us/ef/>

- Stack overflow:

<https://stackoverflow.com/>

- Wikipedia

<https://es.wikipedia.org/>

- HTTP Cats

<https://http.cat/>

- Chat GPT

<https://chat.openai.com/>

Declaración de Autoría

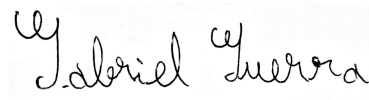
Nosotros, Martin Aristov, Felipe Heredia y Gabriel Guerra, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos el curso de Diseño de Aplicaciones 2;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;


- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.



Martín Arístu
5/10/23



Gabriel Guerra
Gabriel Guerra
5/10/23



Felipe Merediz
5/10/23