Universidad ORT Uruguay

Facultad de Ingeniería

DepoQuick
Diseño de aplicaciones 1
Obligatorio 2

Pedro Azambuja - 270218
Tomas Zelaschi - 281344

Link al repositorio: [IngSoft-DA1-2023-2/281344_270218 (github.com)]

2024

Índice:

Descripción General del Proyecto

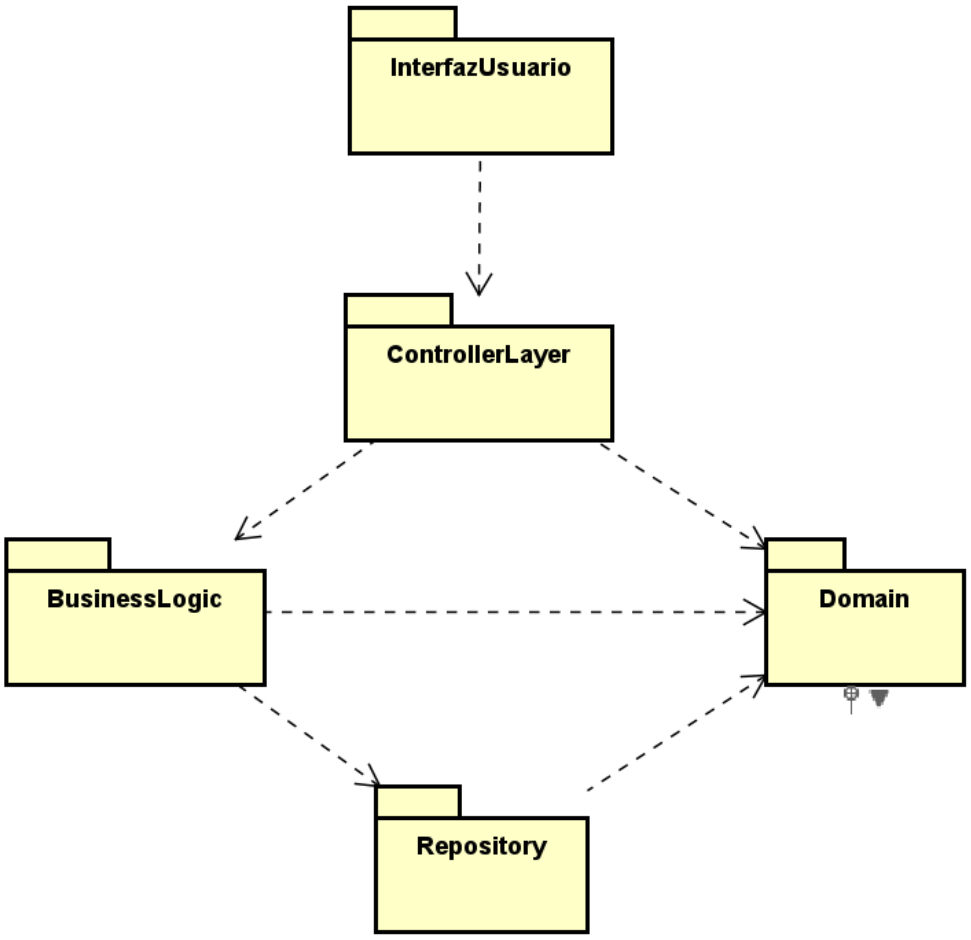
DepoQuick es una aplicación para la gestión de alquiler de depósitos. Hay dos tipos de usuarios: los clientes y el administrador. El cliente es el que reserva los depósitos. El administrador gestiona esas reservas, los depósitos, las promociones y los clientes

El sistema cuando se inicia tiene la opción de registrarse o login. En el panel de la izquierda se pueden ver las distintas opciones que son: depósitos, reservas, promociones y clientes. dependiendo de si se ingresa como administrador o como cliente qué funcionalidades y vistas va a tener. El administrador puede ver todas las opciones y puede crear clientes, depósitos, reservas y promociones. El cliente solo tiene en su vista las reservas las cuales puede ver y crear.

Justificación del diseño

Para esta segunda versión mantuvimos los mismos paquetes que para la primera entrega a diferencia de los paquetes de los test que los unificamos todos en un paquete llamados Tests. Esto lo hicimos ya que en la corrección de la primera entrega nos mencionaron que tener tantos paquete no era necesario y juntar estos no cambiaba la funcionalidad del proyecto. Los paquetes que utilizamos fueron BusinessLogic, Domain, Repository, ControllerLayer, InterfazDeUsuario y Tests.

Diagrama de paquetes



Como se puede ver en el diagrama, pudimos cumplir uno de los objetivos que nos planteamos en la primera entrega, mejoramos las dependencias entre los paquetes y pudimos desacoplar la interfaz de usuario de la lógica de negocio (BusinessLogic). Esto lo logramos aplicando uno de los patrones G.R.A.S.P. Este fue el del uso del controlador. Usamos el paquete ControllerLayer que mediante los DTOs (data transfer objects) pudimos transferir la información desde la interfaz de usuario a la lógica y viceversa sin que estas estén en contacto directo.

En el paquete Domain pusimos las clases con sus atributos y métodos de validación.

En el paquete de BusinessLogic incluimos la lógica de negocio que aplica los métodos y le da el “comportamiento” a las clases del dominio. El propósito de la creación de este paquete es buscar la separación de responsabilidades, buscando que el código quede más claro y mantenible en un futuro.

El paquete Repository lo modificamos ya que nuestro guardado de datos no iba a ser más en memoria sino en una base de datos real. Este paquete se encarga de la persistencia de los datos en la base de datos.

El paquete InterfazDeUsuario es lo que ve el usuario y se encarga de obtener los datos que este ingresa.

Por último, el paquete ControllerLayer se creó para hacer de intermediario entre la interfaz de usuario y la lógica de negocio como mencionamos anteriormente.

Tabla de responsabilidades:

Namespace	Clase	Responsabilidades
Domain	Persona	Definir atributos comunes y validaciones para las clases administrador y cliente como son nombre y apellido, mail y contraseña
Domain	Administrador	Representar las características y comportamientos específicos de un administrador, heredando atributos comunes de la clase Persona.
Domain	Cliente	Representar las características y comportamientos específicos de un cliente, heredando atributos comunes de la clase Persona.
Domain	Depósito	Almacenar y validar atributos de un depósito como son el área, tamaño y si cuenta con climatización. Gestionar la asignación y eliminación de promociones asociadas a un depósito. Proporcionar las promociones aplicables en un día específico a un depósito y determinar la mejor promoción posible.
Domain	Promoción	Almacenar y validar atributos de una promoción, como su id, etiqueta, porcentaje de descuento y periodo de validez.
Domain	Reserva	Almacenar y validar atributos de una reserva como la fecha de inicio, la fecha de fin, el depósito asociado, el cliente asociado, el precio y el estado.

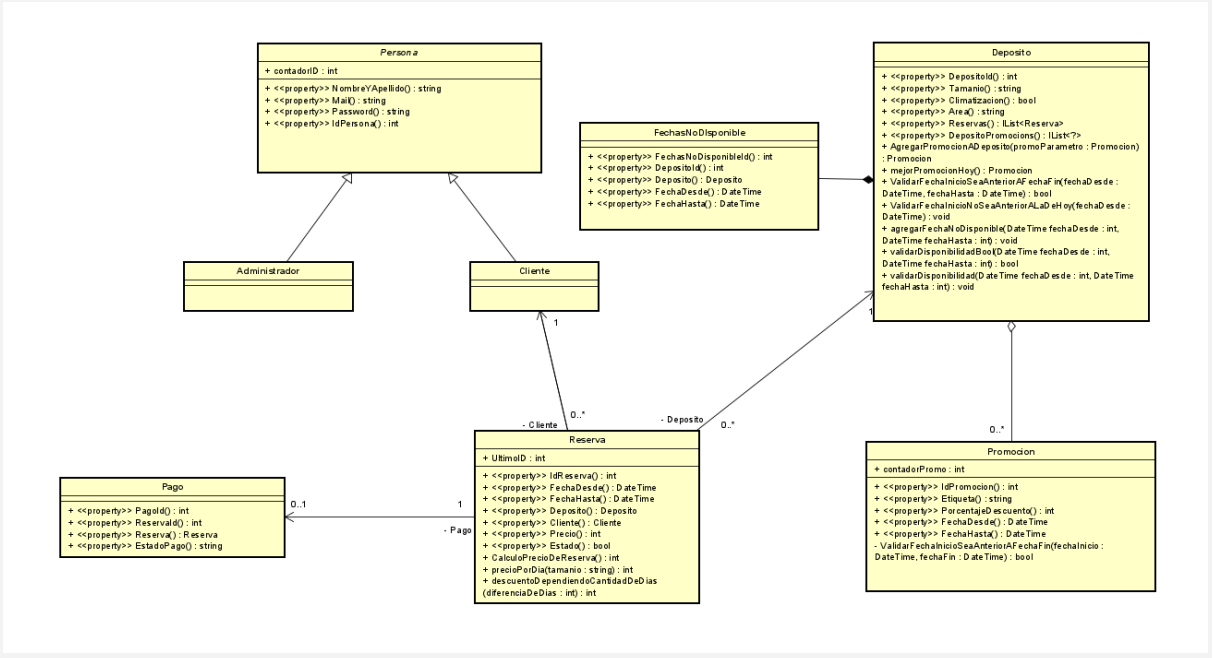
		Calcular el precio de la reserva. Almacenar y validar la justificación del rechazo de una reserva si esto es necesario.
Domain	DepositoPromocion	Se creó para poder persistir en la base de datos la relación entre promoción y deposito ya que esta relación es de N a N
Domain	FechasNoDisponibles	Representa los intervalos de tiempo en los cuales un depósito no está disponible
Domain	Pago	Llevar el estado del pago de la reserva que tenga asociada
ControllerLayer	Controller	Realizar operaciones entre la capa InterfazDeUsuario y el BusinessLogic para así independizar a la interfaz de la lógica. Convertir los dtos en entidades de dominio para que puedan ser procesados por la lógica.
ControllerLayer	DTOAdministrador	Transportar datos de administradores entre procesos entre la lógica de negocio y la interfaz de usuario
ControllerLayer	DTOCliente	Lo mismo que DTOAdministrador pero con cliente.
ControllerLayer	DTODeposito	Lo mismo que DTOAdministrador pero con depósito.
ControllerLayer	DTOPromocion	Lo mismo que DTOAdministrador pero con promoción.
ControllerLayer	DTOReserva	Lo mismo que DTOAdministrador pero con reserva.
ControllerLayer	DTOPago	Lo mismo que DTOAdministrador pero con pago.
ControllerLayer	DTOSesion	Proporcionar un contenedor para la información de sesión, ayudando a gestionar el estado

		de autenticación del usuario en la aplicación, incluyendo si es un administrador
BusinessLogic	Administrador Logic	Gestiona la creación y actualización del administrador. Asegura que solo exista una administrador en el sistema. Trae información sobre el administrador existente.
BusinessLogic	ClienteLogic	Añadir nuevos clientes asegurando que no haya duplicados de mail. Listar todos los clientes. Buscar un cliente específico por id o por mail. Actualizar información de clientes existentes. Eliminar clientes del sistema.
BusinessLogic	DepositoLogic	Añadir nuevos depósitos al sistema. Listar todos los depósitos existentes. Buscar un depósito por su id. Eliminar depósitos del sistema.
BusinessLogic	PromocionLogic	Añadir nuevas promociones al sistema. Listar todas las promociones. Buscar una promoción por su id. Eliminar una promoción del sistema. Actualizar la información de una promoción existente.
BusinessLogic	ReservaLogic	Añadir nuevas reservas al sistema. Listar todas las reservas existentes. Buscar una reserva por su id. Actualizar reservas, incluyendo cambiar el estado a “Aceptada” o “Rechazada” basado en la evaluación del administrador.
BusinessLogic	IExportador	Es una interfaz que cuenta con el método Exportar para la creación de los reportes de las reservas.
BusinessLogic	ExportadorTXT	Esta clase implementa a la clase IExprotador para generar

		reportes de tipo TXT
BusinessLogic	ExportadorCSV	Esta clase implementa a la clase IExprotador para generar reportes de tipo CSV
Repository	IRepository	Definir un “contrato” para operaciones básicas de memoria. Estas operaciones son crear, leer, actualizar y eliminar.
Repository	ClienteRespository	Implementar la interfaz IRepository para manejar operaciones específicas sobre las entidades de Cliente.
Repository	DepositoRespository	Lo mismo para Deposito.
Repository	PromocionRespository	Lo mismo para Promocion.
Repository	ReservaRespository	Lo mismo para Reserva.
Repository	AdministradorRepository	Lo mismo para Administrador.
Repository	DepositoContext	Se encarga de configurar y gestionar la conexión con la base de datos SQL Server. Define las entidades del dominio y sus relaciones, y proporciona métodos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos.
Repository	DepositoContextEnMemoria	Se encarga de configurar y crear una instancia de DepositoContext que utiliza una base de datos en memoria. Esto permite realizar pruebas sin necesidad de una base de datos real, lo cual puede acelerar el proceso de desarrollo y garantizar que las pruebas sean aisladas y reproducibles.

Diagramas de clases

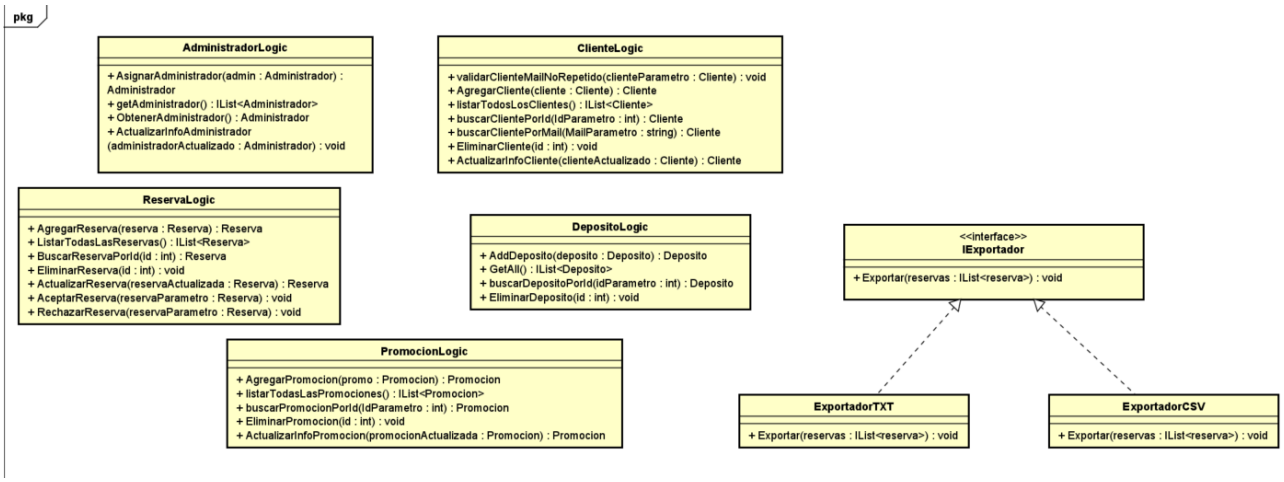
Diagrama Domain:



Como se puede ver en el diagrama, el paquete Domain cuenta con una clase abstracta Persona la cual está compuesta por dos subclases: Usuario y Administrador. Pensamos que esta abstracción nos iba a facilitar pero después nos dimos cuenta que la herencia no era necesaria y nos estaba complicando más que ayudando. Pensamos en cambiarla pero el costo de cambiar todo el otro código era mayor entonces por tema de tiempos decidimos dejarla.

La clase reserva se asocia con la clase cliente y depósito ya que dentro de una reserva está el cliente que la creó y el depósito que le corresponde. Para esta segunda entrega también le agregamos una asociación con la clase pago, ya que la reserva cuenta con un pago pero si esta es rechazada el pago se le desasocia.

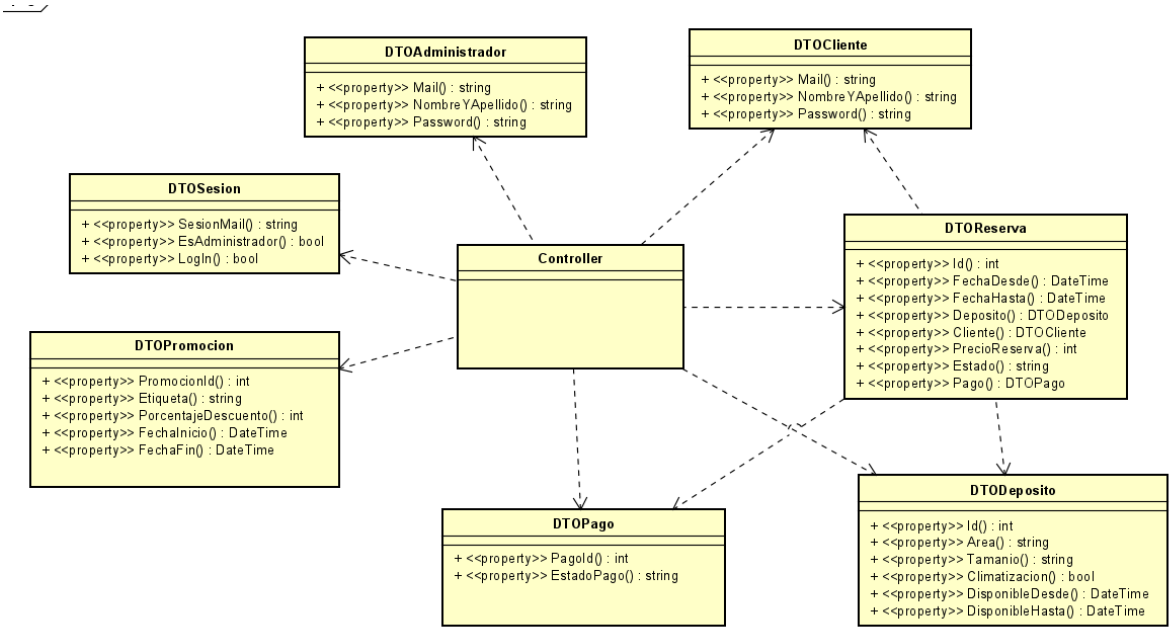
A los depósitos cuando se crean se les puede asignar una o más promociones las cuales van a poder utilizarlas los clientes cuando hagan las reservas. Esta relación es N a N, por lo tanto para guardarla en la base de datos tuvimos que crear una nueva tabla llamada DepositoPromocions. Para esta segunda entrega nos pidieron que los depósitos tengan fechas en las que están disponibles y que si una reserva está usando un depósito en ciertas fechas, que este no esté disponible en esas fechas. Para hacer esto creamos la clase fechasNoDsponibles la cual representa las fechas en la que el depósito no está disponible. Esta relación es una composición ya que si el depósito se elimina sus fechas asociadas también.
Diagrama BusinessLogic:



En el paquete, BusinessLogic, se encuentran definidas las clases que gestionan la lógica de negocio asociada a las entidades del dominio. Cada clase en este módulo encapsula los métodos necesarios para manejar las operaciones específicas relacionadas con su respectiva entidad de dominio.

En comparación a la primera entrega, en esta paquete incluimos la clase IExportador, esta es una interfaz que cuenta con el método exportar. Esta interfaz la implementan las clases ExportadorTXT y ExportadorCSV. Estas clases las decidimos ponerlas en este paquete ya que se encargan de la operación de crear los reportes de las reservas.

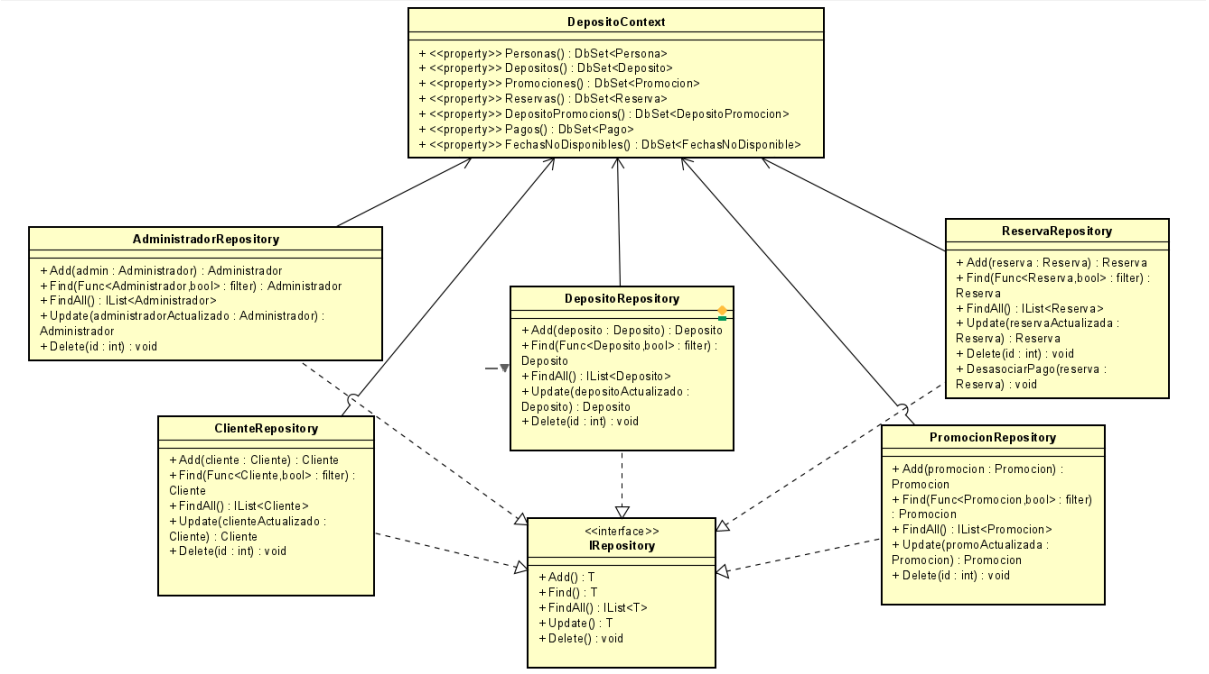
Diagrama ControllerLayer:



Los métodos de Controller están disponibles en el anexo, no los pusimos en el diagrama porque eran demasiados e iban a arruinar la vista y el entendimiento de todo el diagrama.

Acá se puede ver como el controller conoce a todas las clases y a partir de sus métodos crea las instancias de las clases en la lógica.

Diagrama Repository:



En este diagrama se puede ver como cada clase Repository implementa a la interfaz IRepository y a partir de esos métodos las clases se comunican con la base de datos.

Metodología de trabajo y herramientas

Para realizar esta tarea programamos la aplicación en Visual Studio utilizando C#. Para controlar las versiones utilizamos GitHub. Además, utilizamos la técnica de Git Flow para organizar el desarrollo del software mediante la división del trabajo en varias ramas dentro del repositorio de código. Este método nos permitió separar claramente las diferentes etapas de desarrollo de la clase en la que estábamos trabajando, asignando a cada rama una función específica centrada en la incorporación de nuevas características. Este enfoque facilitó una gestión ordenada y eficiente del proyecto, permitiéndonos colaborar y controlar versiones eficazmente.

Inicialmente, creamos dos ramas principales en nuestro repositorio: main y develop. Todo el desarrollo se realizó en la rama develop, desde donde también creamos ramas secundarias denominadas feature/clase para trabajar

específicamente en los requerimientos de cada clase. Una vez completado el trabajo en una de las ramas feature, realizamos un merge de vuelta a develop. La rama main se mantuvo sin uso durante el desarrollo y solo la utilizamos para hacer el merge final para la entrega del proyecto.

Ramas creadas:

- main
- develop
- feature/Deposito
- feature/Deposito2
- feature/Reserva
- feature/Promocion
- feature/Administrador
- feature/Controller
- feature/BaseDeDatos
- feature/InterfazDeUsuario
- feature/ReservaLogic
- feature/AdministradorLogic
- feature/CalculoDePrecioDeposito
- feature/EntityFrameworkSetup
- feature/DisponibilidadFechas
- feature/ReporteReservas
- feature/RealizarPago
- feature/arregloDependencias
- feature/Pagos
- feature/FrontEF
- fix/estructuraTests

La rama feature/Deposito2 la tuvimos que crear porque nos encontramos con inconvenientes en el merge con develop en feature/Deposito y por temas de tiempo no pudimos resolver.

Para esta segunda entrega tuvimos que crear la rama feature/EntityFrameworkSetup que la utilizamos para realizar la transicion a la base de datos. Y también creamos otras ramas para agregar las funcionalidades nuevas y adaptar todo el proyecto a estos cambios.

Metodología de desarrollo

El desarrollo lo realizamos a partir de la técnica de TDD (Test Driven Development), esta técnica consiste en escribir primero las pruebas y después la funcionalidad.

Las reglas de esta técnica son:

- No escribir código sin tener la prueba.
- Escribir el mínimo código para que la prueba de verde.
- Refactorizar el código para que sea más eficiente.

TDD tiene tres fases:

- [RED]: Escribir una prueba que falla porque aún no tiene código que implemente la funcionalidad.

- [GREEN]: Escribir el mínimo código necesario para que la prueba pase.
- [REFACTOR]: Modificar el código sin cambiar la funcionalidad siguiendo las prácticas de clean code.

Para plasmar el uso de esta técnica, adoptamos un formato específico en los mensajes de los commits. Usamos el formato [RED], [GREEN] y [REFACTOR] seguido del nombre de la funcionalidad que estábamos trabajando. Al principio no aplicamos muy bien la parte del REFATOR pero a lo largo del proyecto fuimos tratando de mejorarlo y lo aplicamos más. En el método de calcularPrecioReserva hicimos mucho énfasis en respetar las tres etapas.

Prácticas de Clean Code

Resumen:

Tratamos de aplicar las prácticas lo mejor posible para que el código pueda ser prolijo y entendido por cualquier programador

Nuestro idioma base para programar fue el español, pero por falta de comunicación y muchas veces basándonos en cosas que vimos en la clase mezclamos nombres en inglés y español. Hubiese estado bueno tener el inglés como base ya que es el idioma global pero en la práctica nos quedó más práctico el español.

Prácticas que utilizamos:

- Nombre de clases, métodos, namespaces con PascalCase
- Nombres de variables locales, parámetros con camelCase
- Nombres de interfaces que arranquen con I
- Buena indentación y mantener un orden y formato consistente.
- Funciones con no más de dos parámetros
- No usar comentarios, que el código tenga nombres descriptivos para que se entienda solo
- Nombres de métodos que sean verbos

Decisiones de diseño

Desde el comienzo del proyecto siempre hicimos énfasis en el principio de separación de responsabilidades, asignando a cada clase responsabilidades específicas y agrupandolas en paquetes según los roles que estas cumplian. Este principio además de ayudar a la utilización y codificación del programa ayuda a la legibilidad y mantenibilidad del mismo ya que viendo las clases a simple vista ya se puede identificar qué rol cumplen.

Algo que pensamos en la primera entrega que iba a beneficiarnos fue la herencia de las clases Administrador y Cliente respecto a Persona pero a medida que avanzaba el proyecto notamos que las desventajas eran aún mayores. Inicialmente consideramos que como eran dos roles distintos los cuales tenían interacciones tan distintas con el programa que convenía separar los dos roles en dos clases distintas. Nos dimos cuenta más adelante sobre todo con la implementación de Entity Framework que la diferencia en sus comportamientos no era lo suficientemente distinta como para justificar la cantidad de código extra que nos generó hacer la herencia. Optamos con seguir con el modelo de la herencia porque todo este código extra que mencione ya estaba implementado de la entrega 1 por lo que unificar las clases nos iba a llevar más tiempo todavía para cambiar algo que ya funcionaba.

Una corrección de la entrega 1 fue que la clase controller catchea excepciones específicas y las vuelve a tirar con el nuevo mensaje pero con una excepción genérica. Tomamos la decisión de implementarlo de esta manera porque queríamos facilitar la implementación y legibilidad del front-end y aislar el comportamiento del back-end respecto al front y esto lo logramos haciendo que la clase controller tire solo excepciones genéricas.

La clase Pago surgió de la necesidad de representar el estado del pago de la promoción, inicialmente pensábamos implementar esta funcionalidad con un atributo de tipo string pero leyendo el foro nos dimos cuenta que una vez rechazada la reserva el pago debía ser desasociado por lo que era más sencillo y claro representarlo con una clase.

Una decisión la cual nos llevó bastante pienso fue la de cómo implementar las fechas en las cuales un depósito no está disponible. Luego de debatirlo llegamos a la decisión de implementarlo con pares los cuales representan el comienzo y fin de un período de tiempo en el cual el depósito no está disponible. Planteamos la posibilidad de hacerlo de la manera inversa haciendo pares donde sí estaba disponible pero esta solución implicaba más código teniendo que “cortar” las fechas disponibles cuando se quería ingresar una nueva fecha donde no está disponible. En la primera implementación de la lista de fechasNoDisponibles lo habíamos representado con una lista de instancias de la clase pares pero cuando intentamos representar esta nueva estructura de datos en Entity Framework nos dimos cuenta que era mucho más sencillo y fácil de implementar representarlo con una lista de una clase fechasNoDisponibles.

En cuanto a las fechas en la entrega 1 no teníamos bien definido el formato el cual íbamos a utilizar y en el front-end teníamos campos datetime los cuales no eran ideales. Utilizamos el formato dd/mm/aaaa. Implementamos muchas restricciones en cuanto a las fechas las cuales consideramos que dio como resultado un sistema más robusto en ese aspecto. Hicimos validaciones en los campos los cuales se piden una fecha de inicio y fecha de fin para declarar la vida de un elemento; validamos que la fecha de inicio sea anterior a la fecha fin para que no se creen

campos inconsistentes que generen errores o malentendimientos. Validamos que cuando se registre un Depósito, Promoción o una Reserva que la fecha de disponibilidad inicial o inicio no sean anteriores a la fecha actual.

Llegamos a contar con un proyecto de Test para cada namespace que queríamos probar por lo que teníamos 3 proyectos MSTest. Esto no solo era innecesario sino que se tornaba molesto al momento de buscar un archivo específico en el solution explorer. Para solucionar este problema, unificamos los proyectos de Test en un solo proyecto llamado “Tests” el cual tiene tres directorios uno para cada uno de los proyectos que prueban distintos dominios de la solución.

Persistencia

Para llevar a cabo la implementación de la persistencia utilizamos Entity Framework Core el cual es un ORM (Object Relational Mapping) desarrollado y mantenido por Microsoft para .net y lo impactamos en una base de datos Azure SQL en un contenedor Docker.

Como ya teníamos el dominio de la solución ya implementada tomamos la estrategia Code-First la cual consiste en primero desarrollar el dominio y luego mapearlo a entidades para que Entity Framework las represente y almacene en la base de datos.

Aprovechamos de la interfaz IRepository que implementamos para la entrega 1 para crear nuevas clases Repository las cuales impactan en la base de datos los cambios en vez de las listas de memorias. Al utilizar la interfaz pudimos implementar manejadores con métodos los cuales tenían las mismas firmas que las clases Memory Repository haciendo el impacto de la migración de base de datos en memoria a base de datos en Azure SQL menor.

Se incluyeron las clases DepositoContext y DepositoContextEnMemoria y la interfaz DepositoContextFactory al namespace Repository.Context que es el directorio donde se encuentra todo lo relacionado a la configuración de la base de datos. El propósito de DepositoContext es que para que cuando se corran los Tests sobre la base de datos, se cree una base de datos temporal en memoria sobre la cual se realizan las pruebas y luego de cada prueba se desecha. Lo nombramos DepositoContext y a la base de datos DepositoDB porque como estamos desarrollando una aplicación de gestión de depósitos consideramos que los nombres eran acordes.

La configuración de las tablas en la base de datos se realizaron en el archivo DepositoContext donde definimos las tablas: Personas, Depositos, Promociones, Reservas, FechasNoDisponibles, Pagos, DepositoPromocions. Siendo las primeras la representación de las clases. DepositoPromocions la creamos para la relación N a N entre Promocion y Deposito siguiendo los pasos de la guía en EntityFrameworkTutorial.

Para establecer la conexión con el front-end se utilizó el método options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection") el cual busca el Connection String en el archivo appsettings.json. También especificamos que sea Singleton con ServiceLifetime.Singleton. Esto lo usamos para acceder a la base de datos por el mismo Context, a esto en la documentación se lo llama “Connected Scenario”. Inicialmente, queríamos crear el contexto cada vez que se accedía a la base de datos ya que esto es más eficiente en el uso de recursos pero como se nos pidió que el sistema no se caiga si se pierde la conexión con la base de datos recurrimos a utilizar el escenario de connected para evitar el error que nos daba el método .Migrate.

Implementamos la herencia de Cliente y Administrador de Persona con la estrategia TPH (Table Per Hirarchy) la cual se basa en representar las dos clases en una sola tabla con un atributo tipoUsuario el cual diferencia si es un Cliente o Administrador.

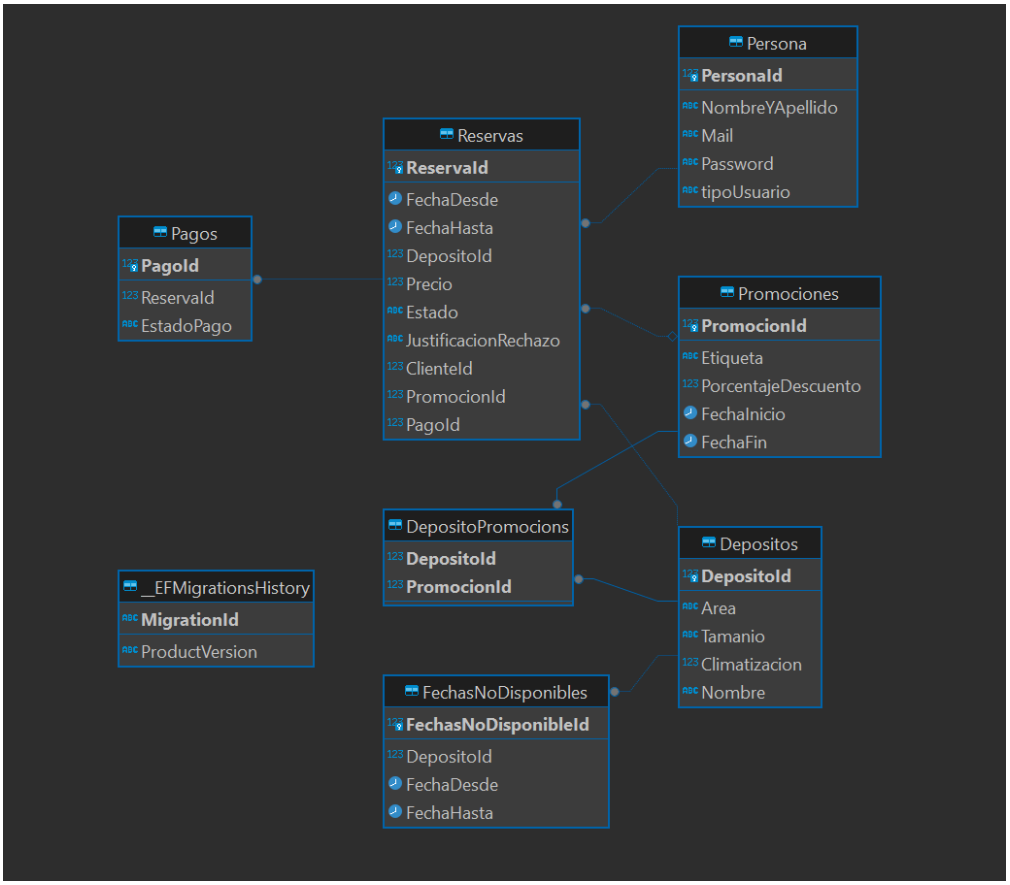
Nos encontramos con varios desafíos al integrar EF al proyecto. Tuvimos problemas para la relación N a N entre Promoción y Deposito, sobre todo con los atributos los cuales exige EF que tiene que tener cada clase para que funcione el sistema como queremos. También tuvimos dificultades con la implementación del TPH, esta complicación no hubiese existido si no hubiéramos implementado la herencia. También tuvimos problemas con Reserva ya que es una clase que tiene muchas relaciones y nos sucedía que por veces no nos recuperaba todos los datos de Reserva. Por último, nos empezaron a dar errores con campos los cuales podrían ser nulos.

Para probar la conexión con la base de datos y que los cambios que realizamos se impacten sobre la base de datos realizamos pruebas sobre Controller que es la clase que impacta y consume la base de datos utilizando las clases Repository. Utilizamos la clase DepositoContextEnMemoria con el DepositoContextFactory para crear una base de datos en memoria y realizar las pruebas sobre ella para que las pruebas no impacten la base de datos real y las condiciones de las pruebas sean todas las mismas.

Como FechasNoDisponible tiene una relación de composición respectoDeposito

Como FechasNoDisponible tiene una relación de composición respecto a Deposito, si se elimina un Deposito deberian tambien eliminarse las FechasNoDisponibles relacionadas a este deposito, por eso especificamos en DepositoContext este comportamiento con el método .OnDelete(DeleteBehavior.Cascade). Como Pagos tiene una relación de 1 a 0,1 tampoco tiene sentido que exista un pago sin una Reserva por lo que Pagos también se elimina en cascada si se elimina una reserva.

Modelo de tablas de la base de datos



Análisis de los criterios para asignar responsabilidades

Para este proyecto, asignamos las responsabilidades siguiendo varios principios clave. Primero, aplicamos el principio de responsabilidad única para asegurar que cada clase tuviera una función específica y bien definida. Por ejemplo, en el paquete Domain, las clases se encargan de definir y validar los atributos específicos de sus respectivas entidades.

También, como mencionamos anteriormente, tuvimos en cuenta la cohesión y el acoplamiento. La creación del paquete ControllerLayer sirvió para desacoplar la capa de interfaz de usuario de la lógica de negocio. Cada clase tiene alta cohesión, facilitando así la mantenibilidad y comprensión del código.

Con la creación de la interfaz IExportador, aplicamos el principio de inversión de dependencias, permitiendo la variación de las implementaciones sin afectar al controlador. La inyección de dependencias en el Controller facilita la sustitución y prueba de estas dependencias. Respetamos el principio de sustitución de Liskov

(LSP) mediante el uso de polimorfismo con IExportador, permitiendo que ExportadorTXT y ExportadorCSV se intercambien sin modificar el código del cliente.

Aplicamos patrones de diseño como el Strategy a través de IExportador y sus implementaciones, lo que permite definir y encapsular algoritmos intercambiables para exportar reportes. También utilizamos el patrón Repository para abstraer la lógica de acceso a datos y proporcionar una interfaz para operaciones CRUD, facilitando la prueba y el mantenimiento.

Exporte de reservas

Para esta segunda entrega, se solicitó que agregáramos una funcionalidad para que el administrador pueda obtener un reporte de las reservas. Este reporte se solicitó que se pueda obtener en dos tipos de formatos: txt y csv, pero se aclaró que en un futuro se podrían agregar nuevos formatos.

En base a esto, nos dimos cuenta que la manera de crear los reportes y la información iba a ser siempre la misma, lo único que iba a cambiar era la forma de presentar la información. Por lo que decidimos implementar el patrón Strategy, este patrón se utiliza para definir una familia de algoritmos, encapsular a uno de ellos y hacerlos intercambiables. Este patrón permite que el algoritmo varíe independientemente del cliente que lo utiliza.

La interfaz IExportador define el método exportar que las otras clases van a tener que implementar. ExportadorTXT y ExportadorCSV implementan esta interfaz y para cada uno de los casos el método exportar va a variar ya que tiene el algoritmo específico que necesita esta clase.

Las clases involucradas en la generación del reporte de reservas son Controller ya que tiene el método GenerarReporteReservas y depende de reservaLogic e IExportador. Despues ReservaLogic porque tiene el metodo ListarTodasLasReservas y depende de ReservaRepository para acceder a la persistencia de datos. ReservaRepository está involucrada con el método FindAll para traer las lista de reservas de la base de datos y depende de DepositoContext. Por último, IExportador también está involucrada con el método exportar y sus implementaciones ExportadorTXT y ExportadorCSV, controller depende de esta clase ya que selecciona el exportador basado en el formato que busque sin tener que conocer la implementación específica del exportador.

En cuanto a la cohesión, todas las clases involucradas tienen una alta cohesión ya que cada una tiene una función específica, la de ReservaLogic es manejar la lógica de las reservas, la de ReservaRepository manejar la persistencia y las clases ExportadorTXT y ExportadorCSV se encargan de la exportación de los reportes en los distintos formatos.

En cuanto al acoplamiento, el acoplamiento es bajo. Controller está acoplado a ReservaLogic y a IExportador, este acoplamiento es necesario y esperado para

esta función. ReservaLogic está acoplado a ReservaRepository que es necesario para separar la lógica de la persistencia. ExportadorTXT y ExportadorCSV no tienen dependencias adicionales, lo que mantiene un bajo acoplamiento.

En resumen, la funcionalidad de exportar el reporte de reservas, al contar con un acoplamiento bajo y una cohesión alta, permite que el código sea mantenible y entendible. Además, facilita la incorporación de nuevos formatos para el reporte en el futuro de manera sencilla.

Diagramas de interacción

Las imágenes de los diagramas las incluimos en el anexo. Además en el archivo que subimos en gestión están los archivos para que se puedan ver de forma más clara.

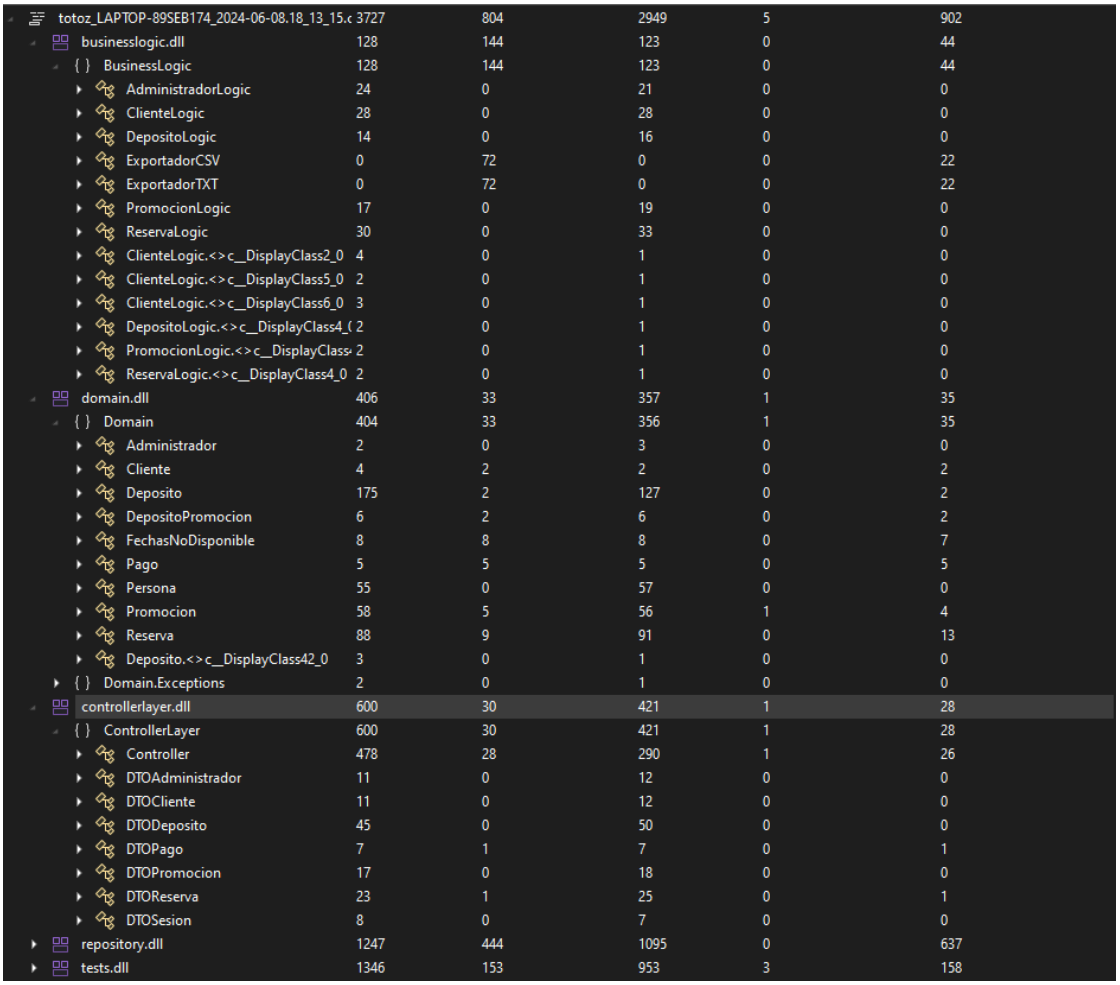
1.Modificar Promoción

2. Registro de cliente

3. Exportar Reporte de reservas en formato TXT

Cobertura de pruebas unitarias

Seguimos con el objetivo de tener por lo menos un 90% de coverage en nuestros proyectos. Si evaluamos el coverage obtenemos los siguientes resultados:



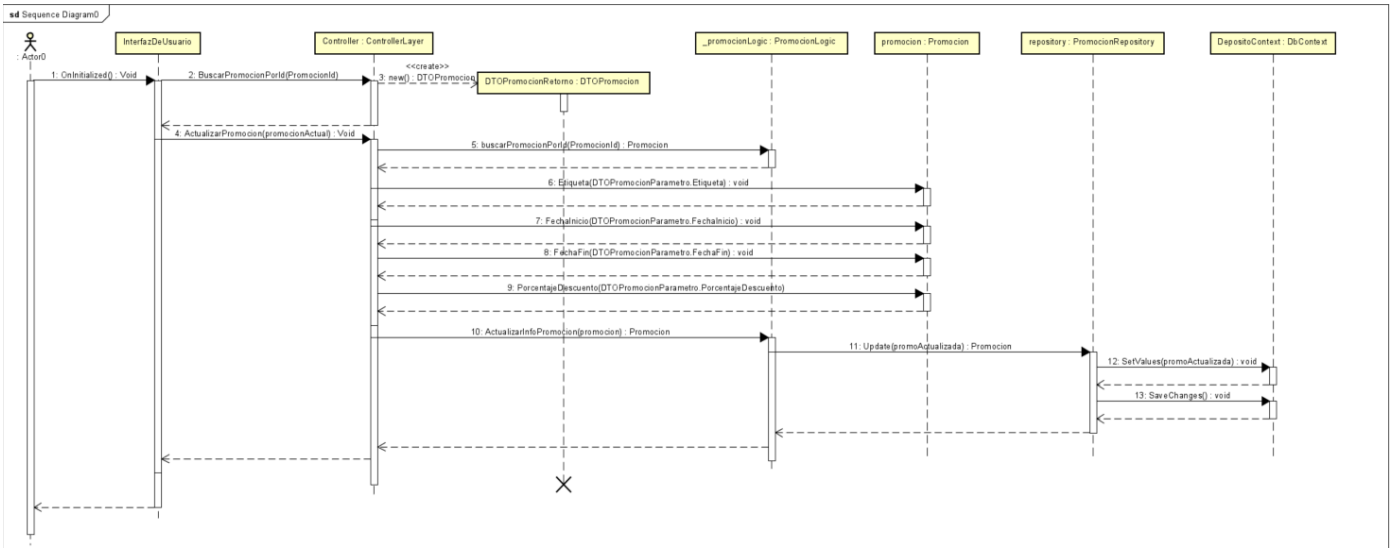
totoz_LAPTOP-89SEB174_2024-06-08.18_13_15.c	3727	804	2949	5	902
businesslogic.dll	128	144	123	0	44
{ } BusinessLogic	128	144	123	0	44
AdministradorLogic	24	0	21	0	0
ClienteLogic	28	0	28	0	0
DepositoLogic	14	0	16	0	0
ExportadorCSV	0	72	0	0	22
ExportadorTXT	0	72	0	0	22
PromocionLogic	17	0	19	0	0
ReservaLogic	30	0	33	0	0
ClienteLogic.<>c__DisplayClass2_0	4	0	1	0	0
ClienteLogic.<>c__DisplayClass5_0	2	0	1	0	0
ClienteLogic.<>c__DisplayClass6_0	3	0	1	0	0
DepositoLogic.<>c__DisplayClass4_(	2	0	1	0	0
PromocionLogic.<>c__DisplayClass	2	0	1	0	0
ReservaLogic.<>c__DisplayClass4_0	2	0	1	0	0
domain.dll	406	33	357	1	35
{ } Domain	404	33	356	1	35
Administrador	2	0	3	0	0
Cliente	4	2	2	0	2
Deposito	175	2	127	0	2
DepositoPromocion	6	2	6	0	2
FechasNoDisponible	8	8	8	0	7
Pago	5	5	5	0	5
Persona	55	0	57	0	0
Promocion	58	5	56	1	4
Reserva	88	9	91	0	13
Deposito.<>c__DisplayClass42_0	3	0	1	0	0
{ } Domain.Exceptions	2	0	1	0	0
controllerlayer.dll	600	30	421	1	28
{ } ControllerLayer	600	30	421	1	28
Controller	478	28	290	1	26
DTOAdministrador	11	0	12	0	0
DTOCliente	11	0	12	0	0
DTODeposito	45	0	50	0	0
DTOPago	7	1	7	0	1
DTOPromocion	17	0	18	0	0
DTOReserva	23	1	25	0	1
DTOSesion	8	0	7	0	0
repository.dll	1247	444	1095	0	637
tests.dll	1346	153	953	3	158

Podemos ver que en businesslogic hay 44 líneas sin cubrir. Estas provienen de dos archivos de la exportación de reportes, no sabemos cómo testear los métodos relacionados a la creación del reporte por lo que no vamos a contarlos en el estudio del coverage. Por lo que conseguimos un coverage del 100% en ese proyecto.
En Domain tenemos 35 líneas no cubiertas, estas provienen de atributos los cuales fueron necesarios para la configuración de Entity Framework pero no son relevantes para nuestra lógica de negocio. De igual manera logramos cumplir nuestro objetivo de tener por lo menos un 90% de coverage ya que estas 35 líneas no cubiertas representan el 10% del código del proyecto.
Si miramos el proyecto de controllerlayer vemos 28 lineas sin cubrir. De estas 10 vienen de un método relacionado a la creación del reporte y los otros son en su mayoría algunos catches los cuales no son probados explícitamente en pruebas de controller pero sí en pruebas de businessLogic. Se cumple el objetivo
El proyecto en su totalidad tiene un coverage del 94% si no se tienen en cuenta el código relacionado a la generación del reporte.

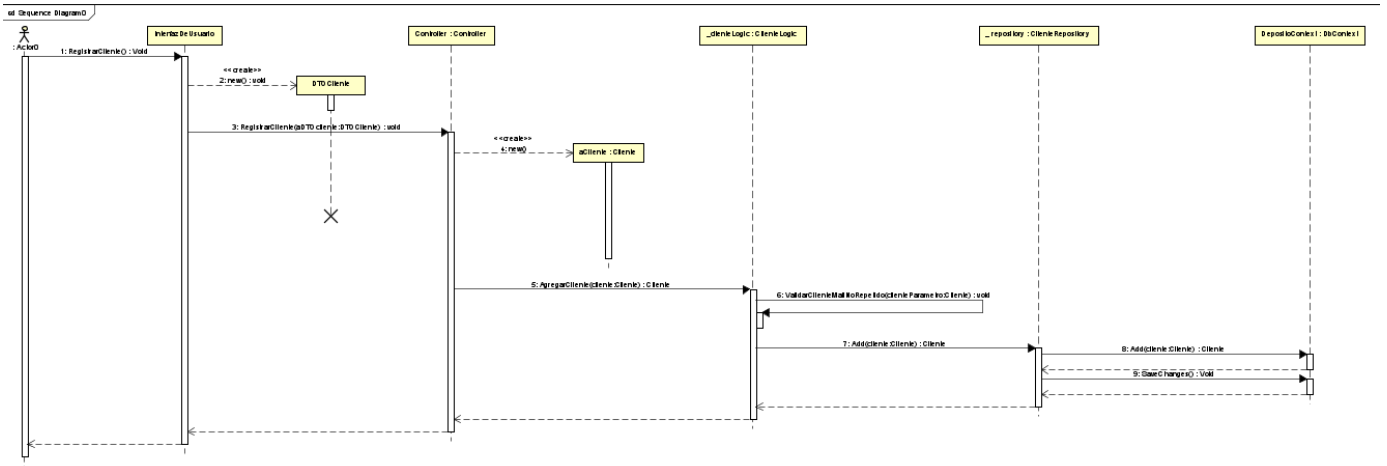
Anexo

Diagramas de interacción:

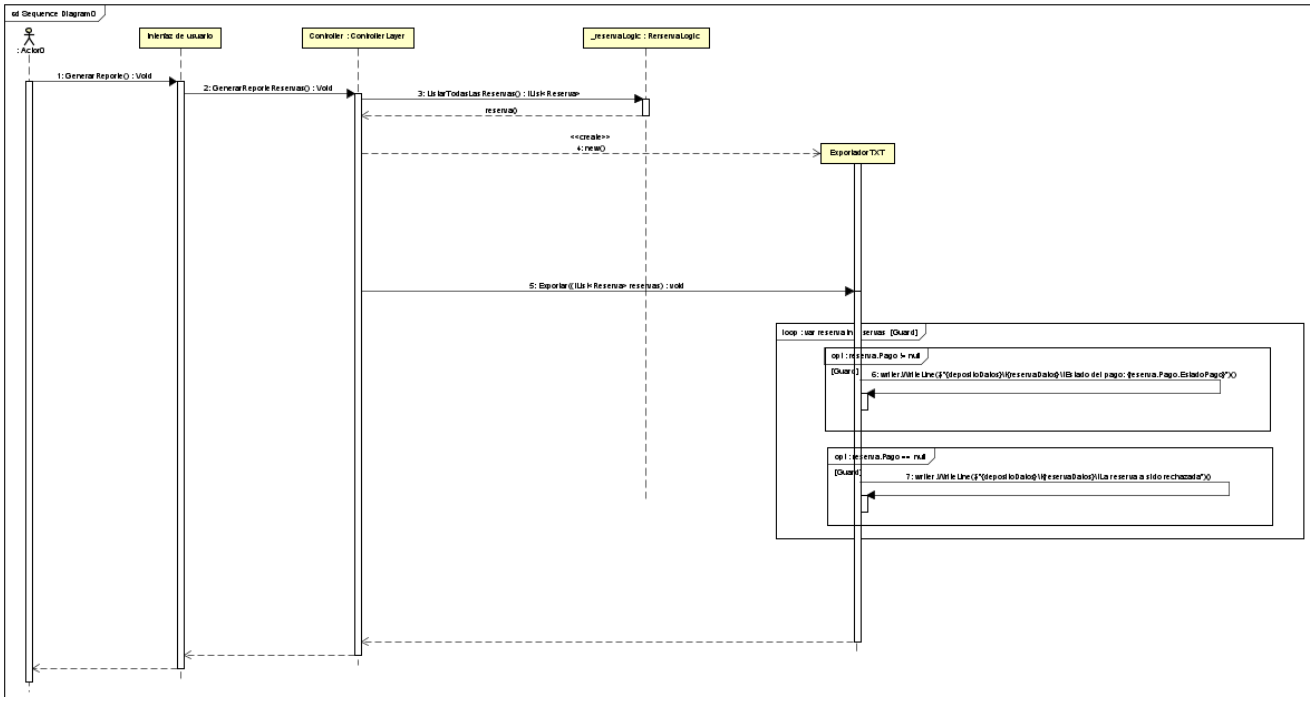
1.Modificar Promoción



2.Registro de cliente



3.Exportar Reporte de Reservas en formato TXT



Métodos de controller:

1. public void RegistrarCliente(DTOCliente aDTOCliente)
2. public IList<DTOCliente> listarTodosLosClientes()
3. public DTOCliente buscarClientePorMail(string mailParametro)
4. public void RegistrarAdministrador(DTOAdministrador aDTOAdministrador)
5. public DTOAdministrador ObtenerAdministrador()
6. public bool EstaRegistradoAdministrador()
7. public int RegistrarDeposito(DTODeposito aDTODeposito)
8. public void AgregarPromocionADeposito(DTOPromocion aDTOPromocion, DTODeposito aDTODeposito)
9. public IList<DTODeposito> listarTodosLosDepositos()
10. public DTODeposito BuscarDepositoPorId(int IdParametro)
11. public void validarQueDepositoNoEsteAsociadoAReserva(DTODeposito aDTODeposito)
12. public void EliminarDeposito(DTODeposito DTODepositoParametro)
13. public int RegistrarPromocion(DTOPromocion aDTOPromocion)
14. public IList<DTOPromocion> listarTodasLasPromociones()
15. private void validarQuePromocionNoEsteEnUso(DTOPromocion DTOPromocionParametro)
16. public void ElminarPromocion(DTOPromocion DTOPromocionParametro)
17. public DTOPromocion BuscarPromocionPorId(int IdParametro)
18. public void ActualizarPromocion(DTOPromocion DTOPromocionParametro)
19. public int RegistrarReserva(DTOReserva DTOReservaParametro)
20. public DTOReserva BuscarReservaPorId(int idParametro)
21. public IList<DTOReserva> ListarTodasLasReservas()
22. public void PagarReserva(DTOReserva DTOReservaParametro)

23. public void AceptarReserva(DTOReserva DTOReservaParametro)
24. public void RechazarReserva(DTOReserva DTOReservaParametro)
25. public IList<DTOReserva> listarReservasDeCliente(DTOCliente aDTOCliente)
26. public void justificacionRechazo(String rechazo, DTOReserva DTOReservaParametro)
27. public bool LogIn(string Mail, string Pwd)
28. public bool esAdministrador(string mail)
29. public IList<DTODeposito> DepositosDisponiblesParaReservaPorFecha(DateTime fechaDesde, DateTime fechaHasta)
30. public void GenerarReporteReservas(string formato)