

Universidad ORT Uruguay

Facultad de Ingeniería

**DepoQuick
Diseño de aplicaciones 1
Obligatorio 1**

**Pedro Azambuja - 270218
Tomas Zelaschi - 281344**

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

2024

Índice:

Introducción.....	3
Metodología de trabajo y herramientas.....	3
Metodología de desarrollo.....	4
Prácticas de Clean Code.....	4
Descripción General de proyecto.....	5
Errores.....	5
Justificación del diseño	5
Diagrama de paquetes.....	6
Tabla de responsabilidades.....	8
Diagrama de clases	11
Mantenibilidad y extensibilidad.....	13
Método Cálculo de precio de depósitos.....	14
Cobertura de pruebas unitarias.....	15
Funcionalidades de depósito	15
Anexo.....	16

Introducción

Metodología de trabajo y herramientas

En este proyecto se nos pidió llevar a cabo una aplicación para administrar el alquiler de depósitos. 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

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.

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 REFACTOR 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

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. Como mejora nos hubiera gustado que se agreguen restricciones para entrar a ciertas vistas ya que por ejemplo entrar a la vista de crear una reserva sin que haya depósitos disponibles no tiene sentido.

Errores

Las funcionalidades que se piden en la letra están todas implementadas. Un error que tuvimos fue la vista para seleccionar las fechas, está el formato pero hay que escribirlas a mano y no queda muy práctico para el usuario.

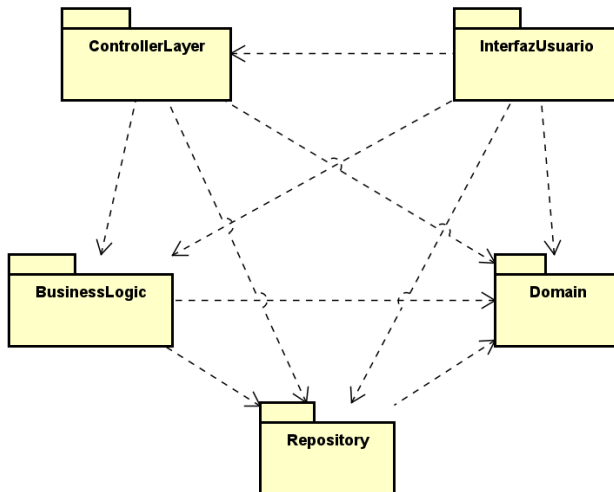
Otro error que tenemos que mejorar para la segunda entrega es que si el administrador rechaza una reserva y después quiere eliminar el depósito que tiene asociado, el sistema le marca que el depósito está siendo utilizado en una reserva.

Justificación del diseño

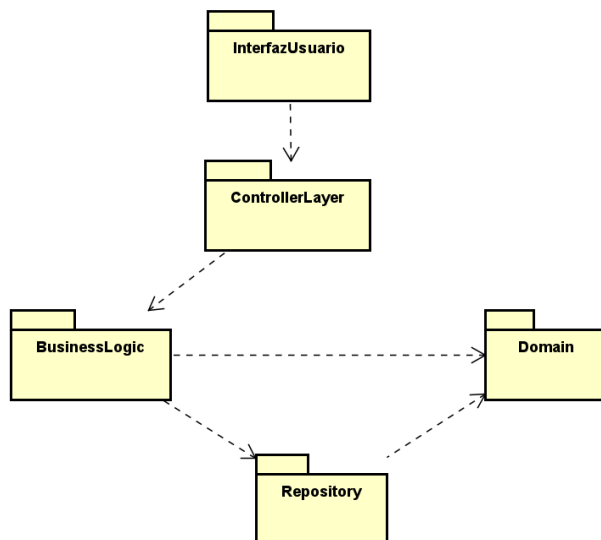
A continuación mostraremos la organización general de la aplicación utilizando solo las dependencias de los paquetes: los paquetes que utilizamos

fueron BusinessLogic, BusinessLogicTest, Domain, DomainTest, Repository, ControllerLayer, ControllerLayerTest e InterfazDeUsuario.

Diagrama de paquetes:



Este es el diagrama de dependencia de paquetes que obtuvimos como resultado de nuestra implementación. Si bien sabemos que la solución no es la óptima, no contamos con los recursos necesarios como para crearla. Repository se relaciona con Domain para guardar las instancias de cada una de las clases. BusinessLogic se relaciona con Repository y con Domain, esto es porque tiene la responsabilidad de almacenar y hacer procesamiento de las instancias de Domain en Repository- ControllerLayer se relaciona con BusinessLogic, Repository y Domain; utiliza BusinessLogic para poder acceder a las instancias de las Clases de Domain (por eso su relación con Domain) en Repository y esto es correcto pero tiene que utilizar Repository porque para crear las instancias de BusinessLogic se necesita primero crear las instancias de Repository. InterfazUsuario se relaciona con todos los paquetes. Esto es porque para crear la instancia de Controller debe crear instancias de BusinessLogic y la última necesita crear instancias de los Repositorys los cuales deben saber que tiene que almacenar (por eso su relación con Domain).



Este es el diagrama de relaciones que creemos que sería óptimo para realizar la mejor implementación del problema.

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.

Con la creación del paquete Repository buscamos resolver el tema de nuestra “base de datos” en memoria, que en esta primera entrega vendrían a ser la lista de depósitos, clientes, reservas y promociones. Dentro de este paquete creamos una clase llamada IMemoryRepository que es una interfaz y nos permitió poder forzar a las clases ClienteMemoryRepository, DepositoMemoryRepository, AdministradorMemoryRepository, PromocionMemoryRepository y ReservaMemoryRepository a implementar los metodos de la interfaz (crear, leer, actualizar y eliminar) para que sea más simple el acceso y manejo de los datos. Esto también nos va a permitir tener una transición más simple a una base de datos real ya que tiene los mismos métodos.

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. Esto asegura una mayor seguridad ya que desde el frontend no se interactúa directamente con la lógica. La obtención de los datos se hace mediante Data Transfer Objects(DTO) y el controller los transforma en entidades de Dominio y viceversa. Esto aporta un modelo de interacción limpio y desacoplado.

Para la segunda entrega vamos a cambiar el código para que funcione de la manera descrita en esta parte que nos parece la óptima.

Tabla de responsabilidades:

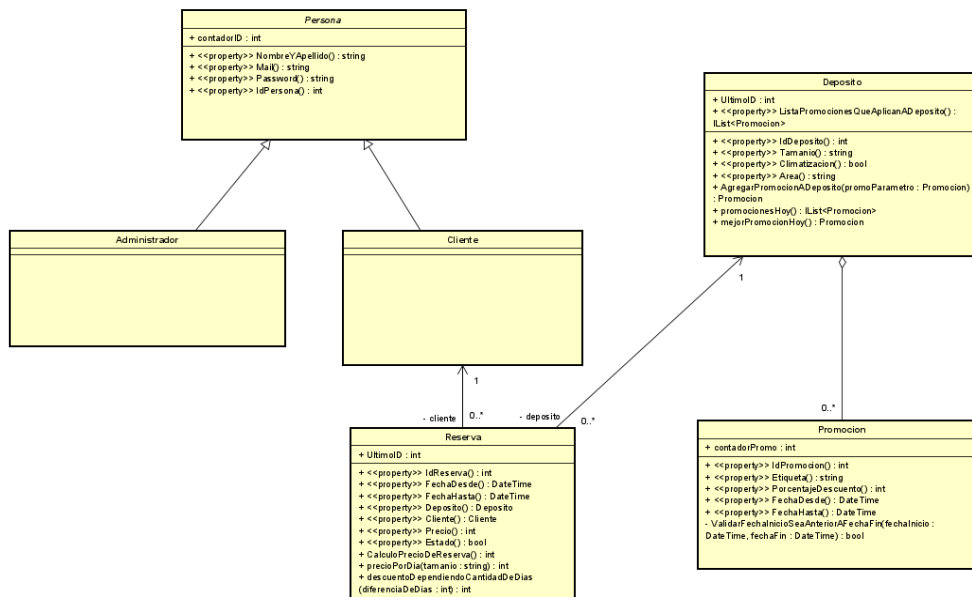
Namespace	Clase	Responsabilidades
Domain	Persona	Definir atributos comunes y validaciones para las clases administrador y cliente como son nombre y apellido, mail y contraseña
Domain	Administrador	Representar las características y comportamientos específicos de un administrador, heredando atributos comunes 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.
ControllerLayer	Controller	Realizar operaciones entre la capa InterfazDeUsuario y el BusinessLogic para así

		<p>independizar a la interfaz de la lógica.</p> <p>Convertir los dtos en entidades de dominio para que puedan ser procesados por la lógica.</p>
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	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	<p>Gestiona la creación y actualización del administrador.</p> <p>Asegura que solo exista una administrador en el sistema.</p> <p>Trae información sobre el administrador existente.</p>
BusinessLogic	ClienteLogic	<p>Añadir nuevos clientes asegurando que no haya duplicados de mail.</p> <p>Listar todos los clientes.</p> <p>Buscar un cliente específico por id o por mail.</p> <p>Actualizar información de clientes existentes.</p> <p>Eliminar clientes del sistema.</p>
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.
Repository	IMemoryRepository	Definir un "contrato" para operaciones básicas de memoria. Estas operaciones son crear, leer, actualizar y eliminar.
Repository	ClienteMemoryRespository	Implementar la interfaz IRepository para manejar operaciones específicas sobre las entidades de Cliente.
Repository	DepositoMemoryRespository	Lo mismo para Deposito.
Repository	PromocionMemoryRespository	Lo mismo para Promocion.
Repository	ReservaMemoryRespository	Lo mismo para Reserva.
Repository	AdministradorMemoryRepository	Lo mismo para Administrador.

Diagramas de clases

Diagrama Domain:



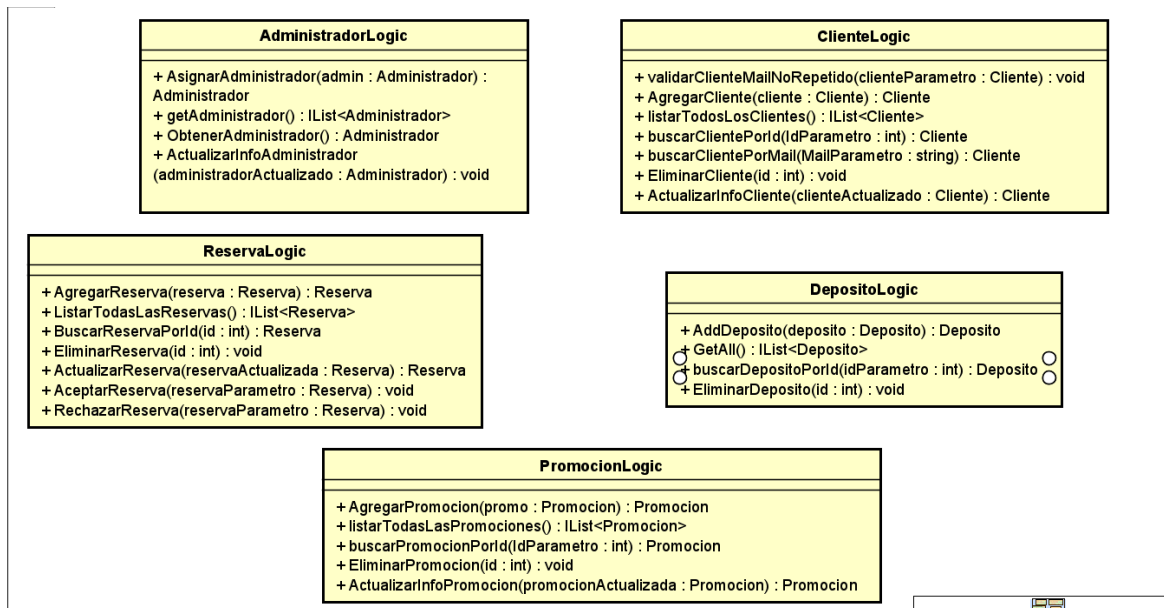
Como se puede ver en el diagrama, el paquete Domain cuenta con una clase abstracta Persona la cual está compuesta por dos subclases: Usuario y Administrador. Esto lo hicimos ya que las dos subclases tenían muchos atributos en común y así pudimos ahorrar código y no repetir métodos.

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.

A los depósitos cuando se le puede asignar una o más promociones las cuales van a poder utilizarlas los clientes cuando hagan las reservas las cuales son almacenadas en listaPromocionQueAplicanADeposito. Es por eso que deposito y promoción tienen una relación de agregación.

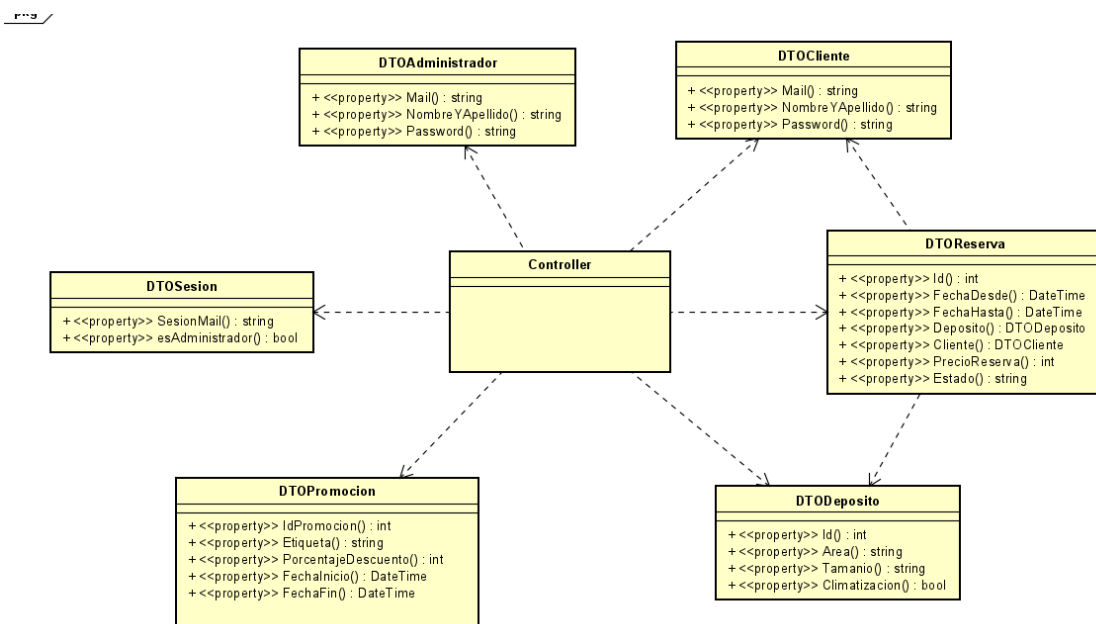
Los contadores de las clases deberían ser privados pero los tuvimos que hacer públicos porque cuando corríamos muchos tests a la misma vez la asignación de los identificadores daba errores porque no se reiniciaban a 0. Entonces tuvimos que forzarlos a que se inicialicen en cero en el TestInitialize y para conseguir esto los tuvimos que hacerlos públicos.

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.

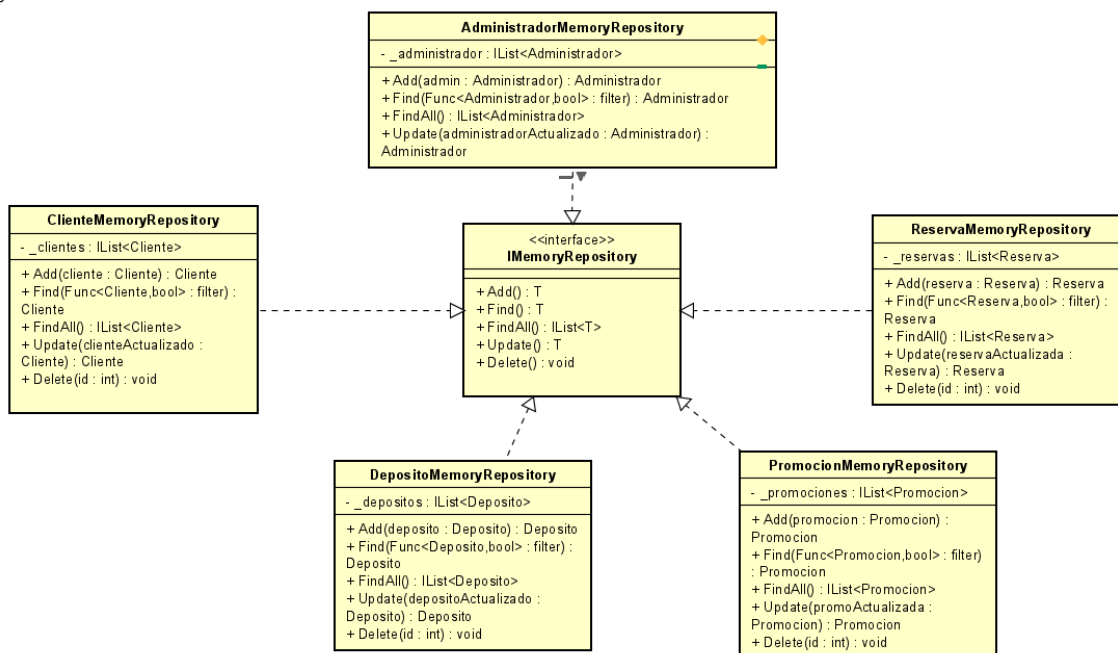
Diagrama ControllerLayer:



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 puede todas las clases y a partir de sus métodos crea las instancias de las clases en la lógica.

Diagrama Repository:



Como ya hablamos anteriormente, acá se pueden ver las listas para el guardado de los datos y los métodos para manejarlas.

Mantenibilidad y extensibilidad

Gracias a la estructura de nuestra solución logramos un alto acoplamiento pudiendo separar las clases utilizadas para representar el diseño de la solución (Domain), el comportamiento y comunicación con la base de datos (BusinessLogic) y la comunicación entre front-end y el back-end (ControllerLogic). Al modularizar responsabilidades esto genera que a largo plazo sea menos laborante extender o mantener el proyecto ya que es más fácil localizar el elemento a encontrar.

Otro aspecto de nuestra solución que aporta a una mantenibilidad y/o extensibilidad más sencilla fue la metodología empleada a la hora de crear la firma de los métodos. Intentamos que los nombres de los métodos incluyen tanto los parámetros que necesita (cuando sea necesario) como el que es lo que hace el método. Además los nombres de las variables que recibe son indicadores de que es

que requiere el método específicamente. Esto es una práctica de Clean Code que nos permitió acelerar el proceso de desarrollo y crear métodos más legibles.

Por último, al utilizar la metodología TDD con un alto coverage nos aseguramos que al momento de agregar features nuevas poder asegurarse que lo anterior funcione como se desea. Esto es porque para cada método hay al menos una prueba, sino más, que verifican que el comportamiento sea el deseado, por lo que si al agregar algo nuevo rompemos algo viejo las pruebas automatizadas nos lo harán saber aportando la información extra de que es lo que no está funcionando de manera esperada.

En resumen, gracias al gran acoplamiento, la metodología empleada para nombrar los métodos (Clean Code) y el TDD; tenemos como resultado una solución fácil tanto de mantener como de extender.

Método `CalculoPrecioDeReserva()` (Cálculo precio de depósito)

El método `CalculoPrecioDeReserva` para calcular el precio de un depósito en una reserva determinada lo implementamos en la clase `Reserva`. Las clases del dominio que interactúan en este método son `Deposito`, `Reserva` y `Promoción`. La clase `Reserva` está asociada con la clase `Deposito`, una reserva se realiza para un depósito específico. La clase `Promoción` está asociada a `Deposito` ya que una reserva tiene su lista de promociones que le aplican. Este método obtiene de la clase `Deposito` su tamaño y si tiene climatización o no.

En cuanto a la cohesión, el método tiene una baja cohesión ya que lo único que hace es devolver el precio de la reserva. Los pasos intermedios que tiene que dar como saber cuanto es el precio dependiendo del tamaño del depósito o dependiendo de la cantidad de días de la reserva lo hace mediante llamadas a otras funciones auxiliares para no tener que hacer más de una cosa.

En cuanto al acoplamiento, la función está altamente acoplada a la clase `Depósito`. Esto tiene sentido ya que se está calculando el precio del depósito pero como el precio depende de la cantidad de días lo tuvimos que implementar en `Reserva` sino no tendría sentido.

En el anexo dejamos la evidencia de la aplicación de la técnica de TDD en la creación del método.

Cobertura de Pruebas Unitarias



Hierarchy	Covered (Blocks)	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered (Lines)
totoz_LAPTOP-89SEB174_2024-05-08.23_51_18.coverage	2095	159	1651	4	190
businesslogic.dll	121	0	117	0	0
repository.dll	148	4	118	0	4
businesslogictest.dll	361	4	281	0	4
domain.dll	285	8	284	2	6
controllerlayer.dll	468	14	337	1	13
domaintest.dll	171	79	136	1	121
controllerlayertest.dll	541	50	378	0	42

La cobertura del código en general es alta. Cumpliendo en todos los casos el requisito de que la cobertura sea mayor a 90%. En repository las líneas no cubiertas son por métodos no implementados.

En total se logró una cobertura del 97% en toda la solución.

Estos resultados son producto del uso de TDD. Desde el principio del proyecto se aplicaron los pasos RED, GREEN y REFACTOR pero se le dio una mayor importancia al principio del proyecto ya que era una metodología nueva y al final se le dio más prioridad a la velocidad de producir código. Este método nos obligó a realizar las pruebas antes de escribir el código asegurándonos de que todo el código que utilicemos esté siendo contemplado por una prueba.




















Funcionalidades de la creacion y administracion de depositos

En este video se puede ver la demostración de estas funcionalidades incluyendo los casos bordes.

Link al video: <https://www.youtube.com/watch?v=6orZ0W2PmZM>

Anexo:

Aplicación de TDD en el método CalculoPrecioDeReserva

[GREEN] calculoDePrecioReservaGrandeConRefrigeracionDe3DiasConVariasPromocionesQueAplicanElMismoDiaQueApliqueLaMejorTest Zelaschi committed last week	6ccd53a		<>
[REFACTOR] Zelaschi committed last week	699db0c		<>
[GREEN] calculoDePrecioReservaGrandeSinRefrigeracionDe10DiasConPromocionDe20PorcientoTest Zelaschi committed last week	f133d3f		<>
[RED] calculoDePrecioReservaGrandeSinRefrigeracionDe10DiasConPromocionDe20PorcientoTest Zelaschi committed last week	8288625		<>
[REFACTOR] Agregar Test al final de la firma de cada test Zelaschi committed last week	3e771b9		<>
[REFACTOR] Zelaschi committed last week	cc5d5eb		<>
[GREEN] calculoDePrecioDeReservaMedianoConRefrigeracionDe15Dias Zelaschi committed last week	9efe825		<>
[RED] calculoDePrecioDeReservaMedianoConRefrigeracionDe15Dias Zelaschi committed last week	ae8fbe0		<>
[GREEN] calculoDePrecioDeReservaMedianoConRefrigeracionDe7Dias Zelaschi committed last week	49b0e8c		<>
[RED] calculoDePrecioDeReservaMedianoConRefrigeracionDe7Dias Zelaschi committed last week	c4e972e		<>
[RED] calculoDePrecioDeReservaChicoUnDiaConRefrigeracion Zelaschi committed last week	a32d6c8		<>
[REFACTOR] Zelaschi committed last week	dfc2c90		<>
[GREEN] calculoDePrecioDeDepositoMedianoUnDiaConRefrigeracion Zelaschi committed last week	4f7f11f		<>
[RED] calculoDePrecioDeDepositoMedianoUnDiaConRefrigeracion Zelaschi committed last week	50d73cf		<>
[GREEN] siNoTienePromocionHoyQueDevuelvaNull Zelaschi committed last week	39b1788		<>
[GREEN] BuscarPromocionDelDiaYQueDevuelvaLaMejorPromocionQueLeAplicaTest Zelaschi committed last week	255fce4		<>
[GREEN] AgregarPromocionADepositoYHayPromocionHoyTest Zelaschi committed last week	10c0c26		<>
[GREEN] AgregarPromocionADepositoYHayPromocionHoyTest Zelaschi committed last week	9680e42		<>
[RED] AgregarPromocionADepositoYHayPromocionHoyTest Zelaschi committed last week	7d88fec		<>

Métodos de la clase controller:

- + RegistrarCliente(aDTOCliente: DTOCliente): void
- + listarTodosLosClientes(): IList<DTOCliente>
- + buscarClientePorMail(mailParametro: string): DTOCliente
- + RegistrarAdministrador(aDTOAdministrador: DTOAdministrador): void
- + ObtenerAdministrador(): DTOAdministrador
- + EstaRegistradoAdministrador(): bool
- + RegistrarDeposito(aDTODeposito: DTODeposito): int
- + AgregarPromocionADeposito(aDTOPromocion: DTOPromocion, aDTODeposito: DTODeposito): void
- + listarTodosLosDepositos(): IList<DTODeposito>
- + BuscarDepositoPorId(IdParametro: int): DTODeposito
- + validarQueDepositoNoEsteAsociadoAReserva(aDTODeposito: DTODeposito): void
- + EliminarDeposito(DTODepositoParametro: DTODeposito): void
- + RegistrarPromocion(aDTOPromocion: DTOPromocion): void
- + listarTodasLasPromociones(): IList<DTOPromocion>
- + validarQuePromocionNoEsteEnUso(DTOPromocionParametro: DTOPromocion): void
- + ElminarPromocion(DTOPromocionParametro: DTOPromocion): void
- + BuscarPromocionPorId(IdParametro: int): DTOPromocion
- + ActualizarPromocion(DTOPromocionParametro: DTOPromocion): void
- + RegistrarReserva(DTORReservaParametro: DTORReserva): int
- + BuscarReservaPorId(idParametro: int): DTORReserva
- + ListarTodasLasReservas(): IList<DTORReserva>
- + AceptarReserva(DTORReservaParametro: DTORReserva): void
- + RechazarReserva(DTORReservaParametro: DTORReserva): void
- + listarReservasDeCliente(aDTOCliente: DTOCliente): IList<DTORReserva>
- + justificacionRechazo(rechazo: String, DTORReservaParametro: DTORReserva): void
- + LogIn(Mail: string, Pwd: string): bool
- + esAdministrador(mail: string): bool

