
DOCUMENTACIÓN DE PROYECTO

Obligatorio 2 - Diseño de Aplicaciones 1

Link de repositorio:

https://github.com/IngSoft-DA1-2023-2/281535_288944

JUNIO 2024

M5C

Matías Corvetto (281535)

Daniel Andrade (288944)

Índice

Índice.....	2
2. Descripción general.....	3
3. Justificación de diseño.....	4
3.1 Diagrama de paquete (link a drive).....	4
3.2 Diagramas de clases por paquete.....	7
3.3 Diagramas de interacción.....	7
3.4 Modelo de tablas.....	7
3.5 Mecanismos generales.....	11
3.5.1 Interacción entre la interfaz de usuario y el dominio.....	11
3.5.2 Almacenamiento de datos.....	12
3.5.3 Manejo de errores.....	13
3.5.4 Polimorfismo y patrones.....	14
3.5.5 Análisis de criterios.....	15
4 Análisis de dependencias.....	16
5 Cobertura de pruebas unitarias.....	17
6 Comparación con la entrega anterior.....	18
ANEXO.....	19

2. Descripción general

Para esta segunda entrega se han implementado todos los requerimientos funcionales solicitados. No se han observado errores recurrentes ni problemas de diseño. Las principales funcionalidades son:

- Gestión de disponibilidad de reservas

Para implementar este requerimiento, se extendió la clase Deposito, para almacenar más datos. Ahora también almacenando su nombre y una lista con la cantidad de rangos de disponibilidad. Se optó por abstraer los rangos de fecha como una clase independiente, para poder ser usadas también por otras clases como lo son Promoción o Reserva. Desde la interfaz, el usuario puede elegir el rango de disponibilidad del depósito, donde la propia lógica implementada irá acoplando y concatenando los rangos de fecha superpuestos o consiguientes, para así optimizar tanto la visibilidad como el funcionamiento del programa.

Posteriormente, a la hora de realizar una reserva, se validará que exista un depósito con disponibilidad para la fecha seleccionada, solo si es así, el usuario podrá realizar el procedimiento de reserva. Una vez que un depósito queda capturado para algún rango de fechas, ningún otro usuario podrá realizar reservas para el mismo período, esto se maneja en el backend.

- Pagos

Los pagos se interpretaron como una propiedad distinta al estado de confirmación, por esto mismo se creó un nuevo atributo para las reservas con un Enum que puede indicar 2 estados de pago: reservado y capturado, haciendo uso también del estado nulo, como cuando no se ha realizado el pago.

La secuencia de reserva es la siguiente: el usuario elige el rango de fechas deseado para su reserva, se le muestran los depósitos disponibles para esa fecha y puede elegir sobre cual de los ofrecidos realizar la reserva. Una vez que ha realizado una reserva, en la página “Mis Reservas”, se le muestra la información de estas. Existe una columna que indica el estado de la solicitud, que funciona de manera muy similar a la entrega pasada, y una nueva columna estado, que hace referencia al estado de pago. Inicialmente el usuario siempre después de realizada la reserva, tendrá la misma como pendiente. Ahí podrá pagarla, para así reservar efectivamente el depósito y esperar la confirmación. El administrador, quien confirma las reservas, sólo podrá visualizar aquellos usuarios que ya hayan pagado sus reservas, y una vez aceptada o rechazada la solicitud, los datos se actualizan. Si la solicitud fue aceptada, el estado queda como capturado, si fue rechazada, queda como cancelado, y se puede ver el motivo de rechazo. Si se acepta una solicitud en un rango, automáticamente todas las demás solicitudes en el mismo son rechazadas.

Los estados de pago como los de confirmación, se validan según los atributos de reserva, y a partir de condicionales se muestra el estado correspondiente.

- Exportar reporte de reservas

La exportación se presenta al usuario en la página “Generar Reporte”, allí se puede elegir formato Csv o Txt. En el backend, esto se creó como una interfaz IGeneradorReporte, donde se puede realizar como un generador en Csv o en Txt. Esto es creado para ser usado por la clase FabricaGeneradorReporte, que es la clase a la cual se le pasa por parámetro el path para guardar el reporte y el formato que se quiere, y según sea el formato de reporte elegido, hará lo correspondiente.

3. Justificación de diseño

3.1 Diagrama de paquete ([link](#) a drive)

Namespace	Clase	Responsabilidad
Dominio	Usuario	Contiene los datos del usuario y sus comportamientos
Dominio	Reserva	Contiene los datos necesarios para hacer una reserva y sus comportamientos
Dominio	Deposito	Contiene los datos necesarios para dar de alta un depósito y sus comportamientos
Dominio	Promocion	Contiene los datos necesarios para dar de alta una promoción y sus comportamientos
Dominio	CalculadoraPrecio	Dada una reserva puede calcular el precio final de la misma
Dominio	RangoFechas	Sirve para validar y guardar rangos de fechas de formato fecha inicial - fecha final
Dominio.Excepciones Dominio	UsuarioExcepcion	Se encarga de crear las excepciones que surgen al hacer operaciones inválidas de usuario
Dominio.Excepciones Dominio	PromocionExcepcion	Se encarga de crear las excepciones que surgen al hacer operaciones inválidas de promoción

Dominio.ExcepcionesDominio	ReservaExcepcion	Se encarga de crear las excepciones que surgen al hacer operaciones inválidas de reserva
Dominio.ExcepcionesDominio	DepositoExcepcion	Se encarga de crear las excepciones que surgen al hacer operaciones inválidas de depósito
Dominio.ExcepcionesDominio	RangoFechasExcepcion	Se encarga de crear las excepciones que surgen al hacer operaciones inválidas de rangos de fechas
LogicaDeNegocios	UsuarioLogica	Intermediario entre Dominio y Repositorio, responsable de administrar el almacenamiento, eliminación, modificación y obtención de los usuarios creados en la aplicación
LogicaDeNegocios	ReservaLogica	Intermediario entre Dominio y Repositorio, responsable de administrar el almacenamiento, eliminación, modificación y obtención de las reservas creadas en la aplicación
LogicaDeNegocios	DepositoLogica	Intermediario entre Dominio y Repositorio, responsable de administrar el almacenamiento, eliminación, modificación y obtención de los depósitos creados en la aplicación
LogicaDeNegocios	PromocionLogica	Intermediario entre Dominio y Repositorio, responsable de administrar el almacenamiento, eliminación, modificación y obtención de promociones creadas en la aplicación.
LogicaDeNegocios.GeneratorReporte	FabricaGeneradorReporte	Encargado de crear un generador de reporte según el tipo que se solicita
LogicaDeNegocios.GeneratorReporte	IGeneradorReporte	Interfaz con la firma del método GenerarReporte
LogicaDeNegocios.GeneratorReporte	GemeradorReporteTxt	Genera un reporte de todas las reservas en formato txt
LogicaDeNegocios.GeneratorReporte	GeneradorReporteCsv	Genera un reporte de todas las reservas en formato csv
LogicaDeNegocio.Ex	UsuarioLogicaExcepcion	Se encarga de crear las

cepccionesLogica		excepciones que surgen al hacer operaciones inválidas de UsuarioLogica
LogicaDeNegocio.ExcepccionesLogica	ReservaLogicaExpcion	Se encarga de crear las excepciones que surgen al hacer operaciones inválidas de ReservaLogica
LogicaDeNegocio.ExcepccionesLogica	DepositoLogicaExpcion	Se encarga de crear las excepciones que surgen al hacer operaciones inválidas de DepositoLogica
LogicaDeNegocio.ExcepccionesLogica	PromocionLogicaExpcion	Se encarga de crear las excepciones que surgen al hacer operaciones inválidas de PromocionLogica
Repositories	IRepository	Interfaz genérica que tiene las operaciones Agregar, Encontrar, Actualizar, Eliminar y ObtenerTodos
Repositories	UsuarioRepository	Es el responsable de manejar la persistencia y gestión de datos de entidades Usuario, además implementa la interfaz IRepository para aplicar los métodos sobre las entidades en la base de datos
Repositories	ReservaRepository	Es el responsable de manejar la persistencia y gestión de datos de entidades Reserva, además implementa la interfaz IRepository para aplicar los métodos sobre las entidades en la base de datos
Repositories	DepositoRepository	Es el responsable de manejar la persistencia y gestión de datos de entidades Deposito, además implementa la interfaz IRepository para aplicar los métodos sobre las entidades en la base de datos
Repositories	PromocionRepository	Es el responsable de manejar la persistencia y gestión de datos de entidades Promocion, además implementa la interfaz IRepository para aplicar los métodos sobre las entidades en la base de datos

Controladores	SesionLogica	Encargado de guardar el email del usuario de la sesión actual
Controladores	Fachada	Es el intermediario entre el frontend y el backend, contiene todas las operaciones con las llamadas correspondientes para que el usuario pueda interactuar con la aplicación

3.2 Diagramas de clases por paquete

Dominio ([link](#) a anexo)

LogicaDeNegocio ([link](#) a anexo)

Repositories ([link](#) a anexo)

Controlador([link](#) a anexo)

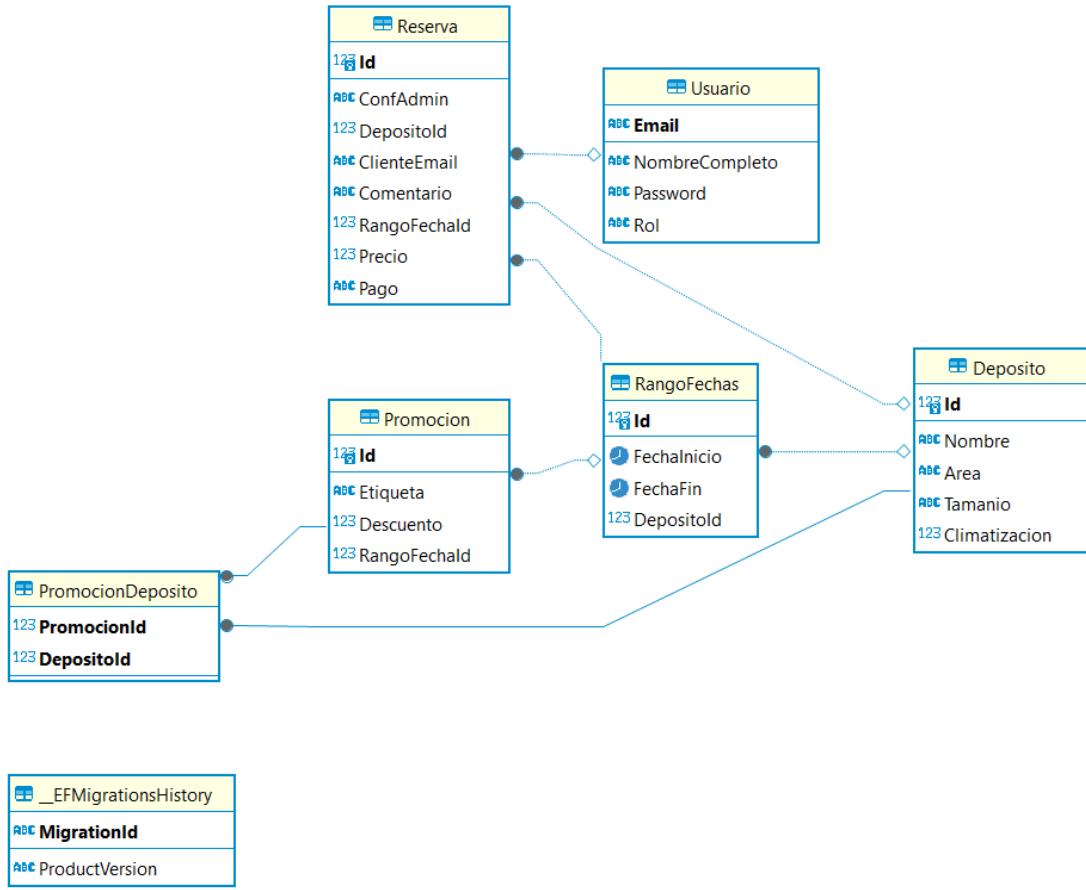
3.3 Diagramas de interacción

Diagrama generar reporte reserva ([link](#) al anexo)

Diagrama pagar reserva ([link](#) a anexo)

Diagrama Agregar Reserva([link](#) a drive)

3.4 Modelo de tablas



A excepción de casos puntuales no se tuvieron que hacer grandes cambios en las entidades del código con respecto a las tablas de la base de datos.

Primer caso: Para pasar correctamente la relación N a N entre Promoción y Depósito tuvimos que modificar el código para que cumplan las convenciones (agregar lista de depósitos en promoción) y escribir nuestra propia configuración para crear la tabla “PromocionDeposito” en la base de datos.

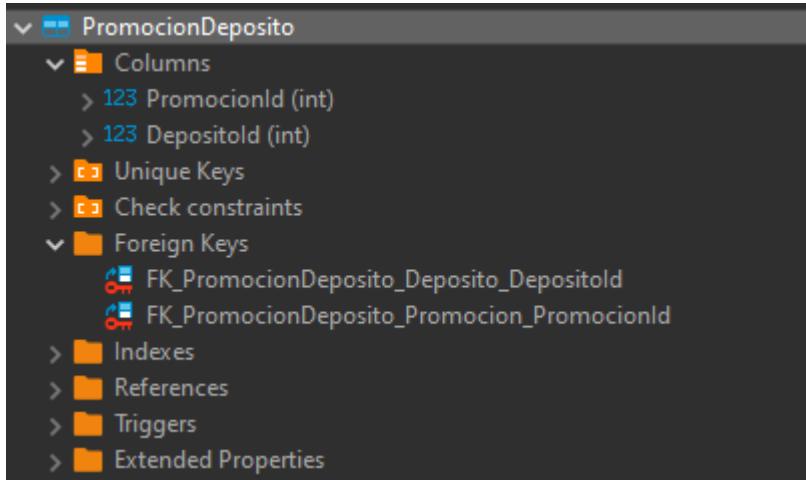
Configuración en clase EFContext

```

modelBuilder.Entity<Promocion>() //EntityTypeBuilder<Promocion>
    .HasMany(navigationExpression: p:Promocion => p.Depositos) //CollectionNavigationBuilder<Promocion, Deposito>
    .WithMany(navigationExpression: d:Deposito => d.Promociones) //CollectionCollectionBuilder<Deposito, Promocion>
    .UsingEntity<Dictionary<string, object>>(
        joinEntityName: "PromocionDeposito",
        configureRight: j:EntityTypeBuilder<Dictionary<...>> => j.HasOne<Deposito?>().WithMany().HasForeignKey("DepositoId"),
        configureLeft: j:EntityTypeBuilder<Dictionary<...>> => j.HasOne<Promocion?>().WithMany().HasForeignKey("PromocionId"),
        configureJoinEntityType: j:EntityTypeBuilder<Dictionary<...>> =>
    {
        j.HasKey(params propertyName: "PromocionId", "DepositoId");
    });

```

Base de datos

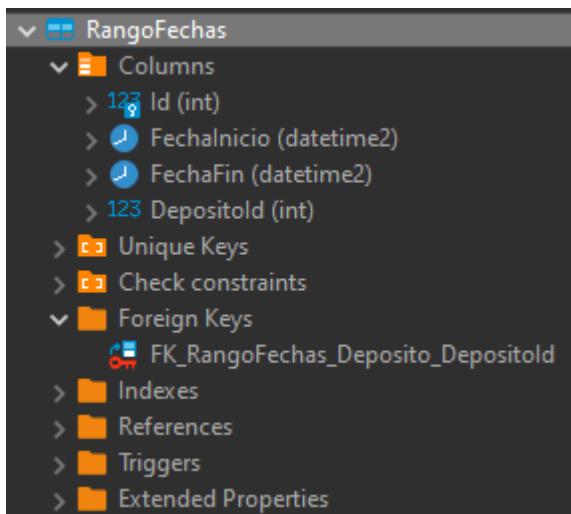


Segundo caso: En el uso de rangos de fechas para las clases Promoción y Reserva se decidió usar la convección 1 a 1 que aplica Entity Framework Core, por lo que no hubo que aplicar ninguna configuración extra, solo que al momento de borrar una promoción o reserva se debe borrar “a mano” el rango de fechas asociado.

PromocionRepository.cs

```
public void Eliminar(int id)
{
    var entidad :Promocion = Encontrar(funcion: p :Promocion => p.Id == id);
    var rangoFechasDeEntidad = entidad.RangoFecha;
    _context.Promocion.Remove(entidad);
    _context.RangoFechas.Remove(rangoFechasDeEntidad);
    _context.SaveChanges();
}
```

Tercer caso: Para poder tener más de un rango de fechas para un mismo depósito usando la misma clase RangoFechas usada también para Promoción y Reserva, no fue necesario escribir una configuración ya que al crear la migración teniendo una referencia a múltiples instancias de RangoFechas en Deposito se crea una columna nulleable en la tabla de RangoFechas llamada “Depositoid”.



Es conveniente que esta columna sea nulleable ya que al agregar una reserva o promoción con rango de fecha, no debería hacerse referencia a ningun deposito.

	129 Id	FechaInicio	FechaFin	123 Depositoid
1	3.002	2024-06-08 00:00:00.000	2024-06-09 00:00:00.000	[NULL]
2	3.004	2024-06-08 00:00:00.000	2024-06-09 00:00:00.000	3.002 ↗
3	4.002	2024-06-09 00:00:00.000	2024-06-29 00:00:00.000	4.002 ↗
4	4.003	2024-06-09 00:00:00.000	2024-06-10 00:00:00.000	[NULL]

Al igual que en las reservas y promociones debemos eliminar de forma “manual” los rangos de fechas asignados a un depósito cuando queremos eliminar ese depósito.

DepositoRepository.cs

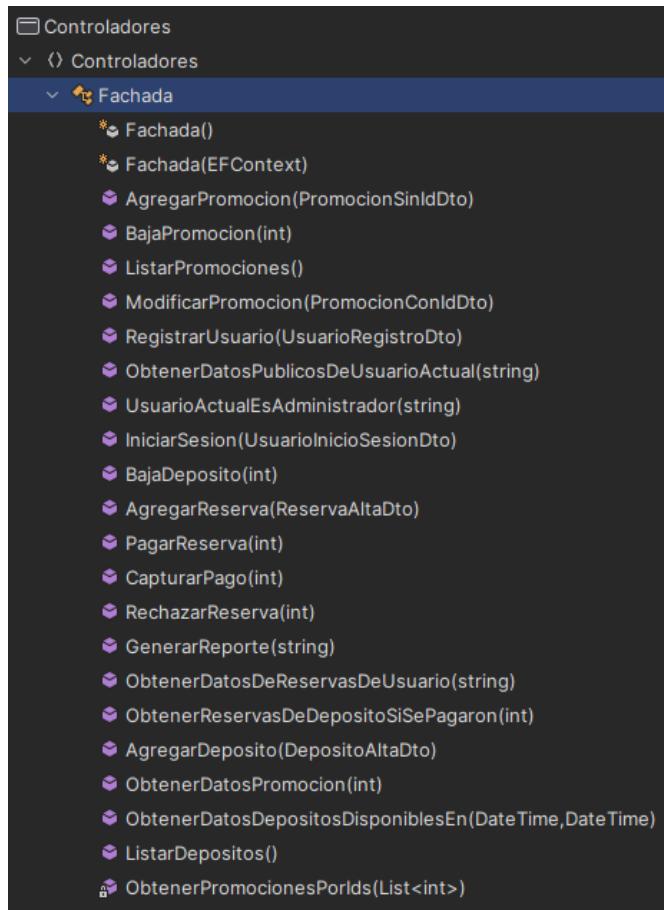
```
public void Eliminar(int depositoId)
{
    var entidad :Deposito? = Encontrar(funcion: d :Deposito => d.Id == depositoId);
    foreach (var entidadRangoFecha in entidad.Disponibilidad)
    {
        _context.RangoFechas.Remove(entidadRangoFecha);
    }
    _context.Deposito.Remove(entidad);
    _context.SaveChanges();
}
```

3.5 Mecanismos generales

3.5.1 Interacción entre la interfaz de usuario y el dominio

Para esta aplicación decidimos que todas las acciones que haga el usuario para interactuar con la página pasen por un solo controlador llamado Fachada, este implementa el patrón estructural Facade. Este nos proporciona el acceso a métodos que pueden estar en constante cambio y evolución. En este caso tener una clase que centralice las llamadas a la lógica de negocio y el dominio permite abstraer la implementación subyacente, facilitando su uso y mantenimiento.

La Fachada además ofrece la ventaja de desacoplar la interfaz de usuario de la implementación de las clases de Dominio y LogicaDeNegocio, a cambio de un alto nivel de acoplamiento y un bajo nivel de cohesión. Sin embargo, este sacrificio es solo en la Fachada, ya que el resto de clases se ven beneficiadas por una mayor cohesión y menor acoplamiento.



En la aplicación agregamos el servicio Fachada con un tiempo de vida scoped porque depende de “EFContext”, que también se registra como scoped al registrarlo con “AddDbContext”. Esto asegura que ambos comparten el mismo ciclo de vida en cada solicitud.

```
builder.Services.AddDbContext<EFContext>( optionsAction: options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString(name: "DefaultConnection")));
builder.Services.AddScoped<Fachada>();
```

3.5.2 Almacenamiento de datos

Se aplicó el enfoque code first para la creación de tablas de nuestra base de datos en función de las clases de nuestro dominio y sus configuraciones. Para la creación de las migraciones se creó la clase EFContext la cual hereda de la clase DbContext de “EntityFrameworkCore”.

Program.cs

```
builder.Services.AddDbContext<EFContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString(name: "DefaultConnection")));

builder.Services.AddScoped<Fachada>();
```

El EFContext se incluye en la aplicación mediante el método AddDbContext lo cual permite que el contexto esté disponible para la inyección de dependencias, esto es beneficioso porque permite injectar el contexto en el controlador como una dependencia, para esto ser posible se debe tener definido un constructor con el contexto como parámetro. (referencias en anexo, [link al anexo](#))

Implementamos el patrón Repository para el manejo de este contexto desde la aplicación, para controlar cada tipo de entidad se creó una clase Repository por entidad, estas clases heredan de la interfaz IRepository que tiene las operaciones básicas CRUD (Create, Read, Update y Delete).

Las ventajas principal que nos brinda aplicar el patrón Repository es poder separar la lógica de acceso de datos de la lógica de negocio de la aplicación, lo cual beneficia la mantenibilidad del código ya que si se desea cambiar la fuente de datos no afectará a la lógica del negocio. (referencias en anexo, link al anexo)

3.5.3 Manejo de errores

Decidimos manejarnos con nuestras propias clases de excepciones en el backend para mientras desarrollamos saber exactamente en donde se encuentran los errores. Mientras que en el frontend en caso de haber un error al agregar, crear o eliminar un elemento se muestre por pantalla el mensaje de la excepción capturada en un bloque catch. Se buscó este enfoque para poder brindarle una retroalimentación precisa al usuario de lo que está fallando en caso de haber errores.

En casos especiales se muestran mensajes de error sin esperar excepciones, esto se debe a que algunas pocas operaciones son exclusivas del frontend.

Para reducir aún más el conocimiento del frontend con el dominio y la LogicaDeNegocio capturamos las excepciones al mandar a hacer

operaciones al backend como excepciones genéricas, esto con el fin de desacoplarse lo más posible a la implementación del backend

Ejemplo: Alta deposito Interface/Depositos/Alta.razor ([link](#) al anexo)

3.5.4 Polimorfismo y patrones

Debido a la implementación del patrón repository en el proyecto, tuvimos que implementar una interfaz IRepository genérica para lograr abstraer el manejo de datos de la lógica de negocio. Es útil para sí en un futuro se quiere cambiar la fuente de datos, tenga el mínimo impacto en el resto de la aplicación.

Otro uso de interfaces es en la generación de reportes de reserva ya que se buscó que la implementación pueda ser abierta a agregar otros formatos además de txt y csv. Con el uso de interfaces y del patrón creacional Factory Method implementamos las siguientes clases: ([link](#) al anexo)

La clase “FabricaGeneradorReporte” es quien crea el generador de reporte y se decide por cual según el formato que recibe como parámetro, el tipo que retorna es “IGeneradorReporte” ya que es una interfaz usada para que no haya diferencias a la hora de crear y devolver el generador de reporte.

El aplicar este patrón también implica aplicar principios como abierto/cerrado ya que se pueden incorporar nuevos tipos de reportes en el programa sin romper el código existente. También cumple el principio de responsabilidad única, ya que cada generador se ocupa de una sola cosa; generar el reporte. Y la fabrica solo crea el generador de reporte solicitado. Además se evita un alto acoplamiento ya que se separa la lógica de crear el generador con la de su lógica.

Para la funcionalidad de cálculo del precio de reserva, se aplicó el principio de fabricación pura mediante la creación de una clase llamada CalculadoraPrecio. Esta clase asume la responsabilidad exclusiva de calcular el precio de la reserva. La introducción de esta clase ha mejorado la cohesión de la clase Reserva, ya que la mayoría de sus métodos estaban relacionados con el cálculo del precio. Esto ha simplificado su estructura y ha permitido un mejor mantenimiento y comprensión del código. ([link](#) al anexo)

3.5.5 Análisis de criterios

Dominio: En las entidades del dominio se buscó aplicar el patrón Experto para asignar las responsabilidades, se buscó que cada clase cumpla con conocer los datos que la componen y validarlos. Un caso especial es Reserva, que antes se calculaba el precio dentro de reserva ya que conocía todo lo necesario para hacerlo, ahora decidimos implementar la clase “CalculadoraPrecio”, que dados un depósito y un rango de fechas calcula el precio correspondiente a una reserva con esas características. “Reserva” al crearse crea también una instancia de “CalculadoraPrecio” para delegar la responsabilidad de calcular el precio, con esto logramos que “Reserva” tenga mayor cohesión afectando un poco al acoplamiento de “CalculadoraPrecio” y “Reserva”. ([link](#) al anexo)

LogicaDeNegocio: Las clases del namespace LogicaDeNegocio están pensadas para solo manejar la lógica de las operaciones que se realizan sobre las entidades del dominio, además de métodos para aplicar filtros a la hora de mostrar datos solicitados desde la interfaz de usuario.

LogicaDeNegocio.GeneradorReporte: Las clases de esta carpeta cumplen solo con el propósito de poder entregar al usuario final el reporte de reservas solicitado, sea cual sea el formato.

Repositories: Las clases del namespace Repositories son las encargadas de mandar a aplicar las operaciones básicas sobre los datos de la base de datos. Además se encuentra la clase EFContext que es quien se encarga de gestionar la conexión con la base de datos y proveer un punto de acceso a través del cual las demás clases del Namespace pueden interactuar con los datos.

Cada clase Repository aplica las operaciones sobre un solo tipo de entidad, aunque en casos como “PromocionRepository” también se deben realizar operaciones sobre “RangoFechas” ([link](#) al anexo)

Controladores: La idea de las clases de este namespace actúan como intermediario entre las solicitudes del cliente y nuestro dominio y lógica de negocios.

4 Análisis de dependencias

Decidimos ubicar la lógica para generar reportes en clases del Namespace “LogicaDeNegocio”, Estas clases son únicamente invocadas por la fachada que es quien le solicita a la fábrica de generadores de reportes que cree uno para su uso. En cuanto a las clases del dominio solo se interactúa directamente con Reserva, ya que tanto la fábrica como los generadores reciben la lista de reservas por parámetro al crearse, para consultar sobre los datos de una reserva para cada línea del reporte.

Analizando las clases que interactúan en la generación de los reportes podemos ver que hay una mayor cohesión gracias al patrón Factory ya que la clase fabrica solo se encarga de la creación mientras que las clases que se encargan de generar un reporte solo cumplen con esa única función.

En cuanto al acoplamiento creemos que cumplimos con un valor bajo para el correcto funcionamiento de las clases. Las clases “GeneradorReporteTxt” y “GeneradorReporteCsv” implementan la interfaz IGeneradorReporte lo que permite a la fábrica crearlas y devolverlas sin conocer en detalle sus implementaciones. Estas clases además se acoplan a Reserva ya que es necesario para obtener los datos de cada reserva y así cumplir su función. En la clase FabricaGeneradorReportes depende de la interfaz IGeneradorReporte, lo cual es ideal para no acoplarse a implementaciones concretas de generadores de reportes. Esto facilita la extensión de la funcionalidad pudiendo agregar nuevos generadores de distintos formatos de forma sencilla y sin modificaciones a esta clase.

Un problema que podemos ver es el incumplimiento de la ley de Demeter, ya que para cada línea del reporte tenemos que preguntar por los datos de reserva, y en más de una ocasión consultamos directamente por datos de instancias de otras clases del dominio que maneja la clase reserva.([link](#) al anexo)

5 Cobertura de pruebas unitarias

Excluyendo el namespace Interface y la carpeta Migrations de Repoistories

Symbol	... ▾	Uncovered/Total Stmt.
↳ Total		98% 58/3098
> ↳ DominioTest		100% 0/721
> ↳ LogicaDeNegocioTest		100% 0/532
> ↳ Controladores		100% 0/266
> ↳ ControladoresTest		100% 0/466
> ↳ LogicaDeNegocio		99% 3/369
> ↳ Repositories		98% 3/163
> ↳ Dominio		98% 6/392
> ↳ Dtos		76% 46/189

Tenemos un alto nivel de cobertura del código. Algunos Dtos no fueron probados debido a su poco uso en la aplicación, ya que consideramos que no era necesario.

Las clases del dominio fueron cuidadosamente testeadas, ya que hubo cambios importantes, como la implementación de la clase “RangoFechas” para el manejo de fechas y “CalculadoraPrecio” para el cálculo del precio de una reserva. Debido a estas nuevas implementaciones, fue necesario modificar las clases del dominio y los tests ya entregados para la primera entrega.

En LogicaDeNegocio se implementan nuevas funciones relacionadas a las nuevas funcionalidades las cuales fueron testeadas para cubrir todos los casos posibles de uso.

Los Controladores fueron testeados pensando en la experiencia final del usuario, los test reflejan cómo podría interactuar el usuario con la aplicación. Creemos haber logrado una buena cobertura de los casos posibles que se pueden presentar durante el uso de la aplicación.

Debido a la implementación de la base de datos en nuestra aplicación optamos por manejar los test en una base de datos en memoria, esto lo logramos gracias a la siguiente configuración

EFContext y el uso de la siguiente clase (referencias en anexo, [link](#) al anexo)

La clase es invocada para crear una instancia de EFContext para los test, es utilizada en todos las clases test de la aplicación, acá un ejemplo de uso ([link](#) al anexo)

Las ventajas de haber aplicado esta base de datos en memoria para la ejecución de tests es que no se va a llenar una base de datos real con datos innecesarios que son solo de test. Además, gracias al método CleanUp, después de cada método de test se borra la base de datos en memoria, asegurando que cada test se ejecute en un entorno limpio y sin datos previos. Esto permite evitar errores debidos a estados persistentes no deseados y facilita el reuso de datos y consistencia en los resultados de los tests. Así, los tests pueden correr de manera más rápida y eficiente, ya que no se requiere interacción con un sistema de base de datos externo.

6 Comparación con la entrega anterior

Diagramas de paquetes ([link](#) al anexo)

El cambio principal es que interfaz solo depende de Controlador y del paquete contenido en Repositories llamado Context, ya que hicimos que desconozca toda la lógica del backend y la implementación de las entidades del dominio. Por temas de conocimiento no se buscó implementar que la interfaz no conozca EFContext que es la clase que maneja la base de datos.

Diagramas de Dominio([link](#) a anexo)

En la comparación de este paquete con respecto a antes se puede ver cosas anteriormente mencionadas, se quiso incluir la clase RangoFechas para el reuso de código para manejar fechas además de quitarle la responsabilidad a las clases que representan entidades en nuestro dominio, y lo mismo con la clase CalculoPrecio. Además podemos ver que tenemos mas atributos para poder implementar las nuevas funcionalidades solicitadas.

Diagramas de LogicaDeNegocio([link](#) a anexo)

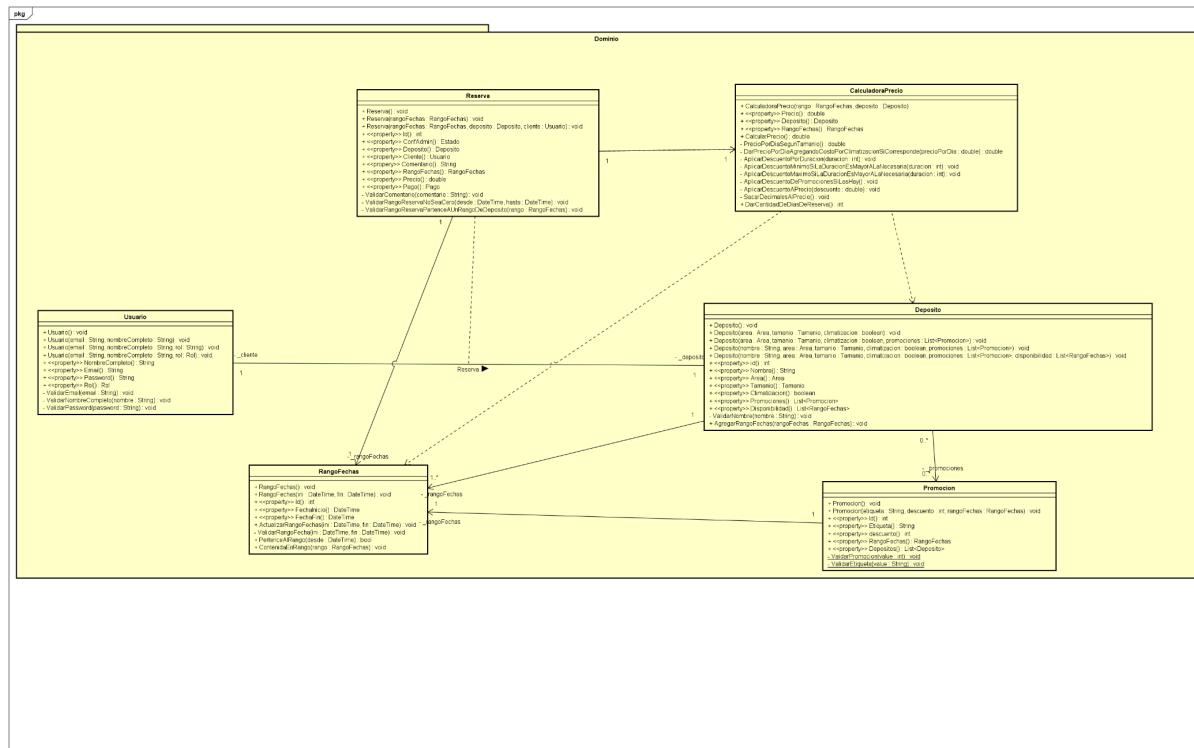
En la comparación de este paquete con respecto a antes se puede ver las implementación de nuevas funciones para el correcto funcionamiento de las nuevas funcionalidades de la aplicación.

Diagramas de Repositories([link](#) a anexo)

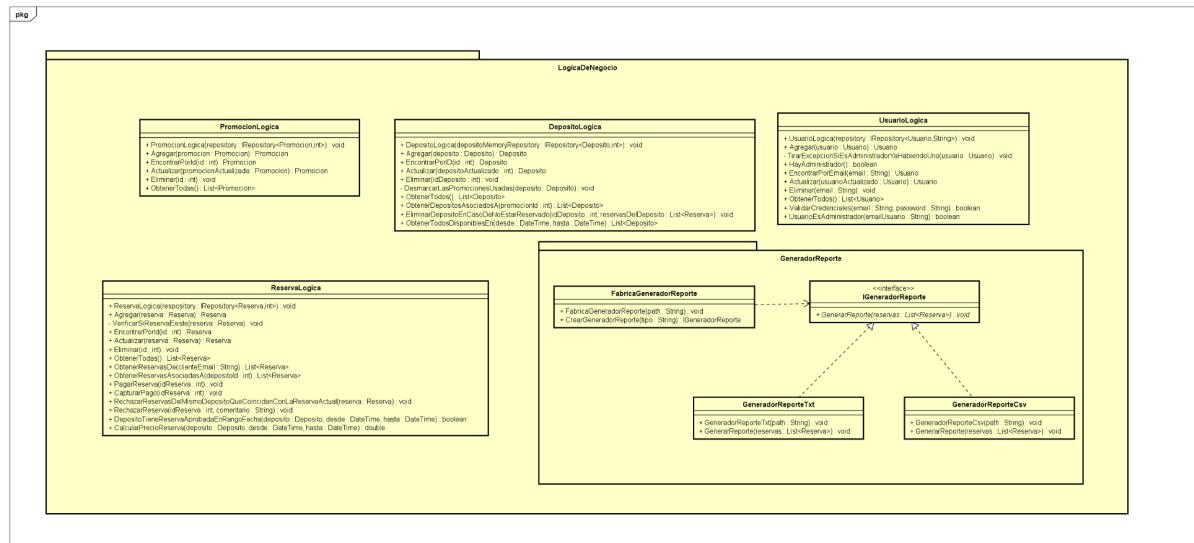
El cambio principal es el cambio de manejar los datos en listas en memoria a usar el EFContext como única puerta para los datos de la aplicación ubicados en una base de datos externa.

ANEXO

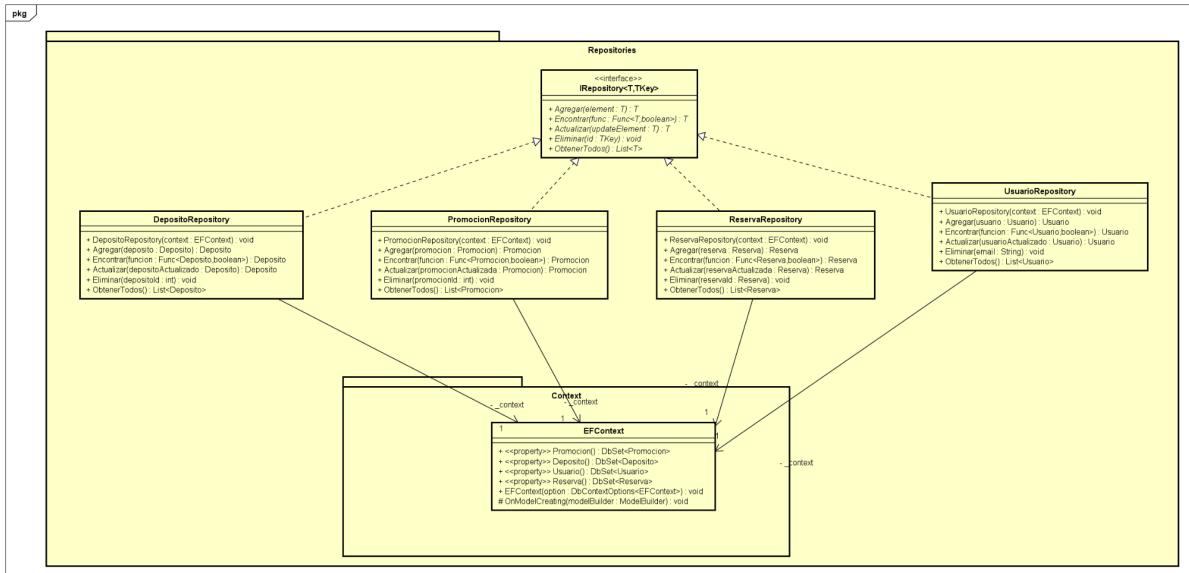
1



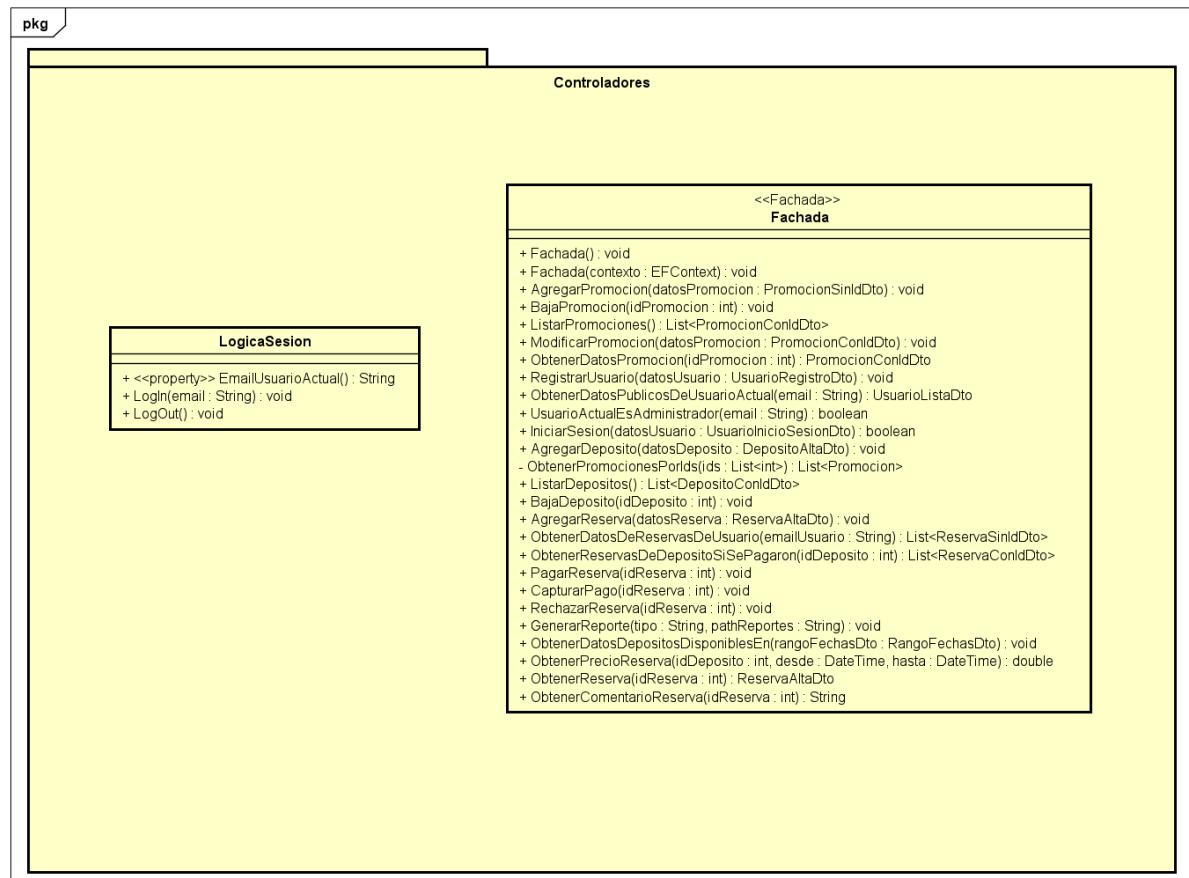
2



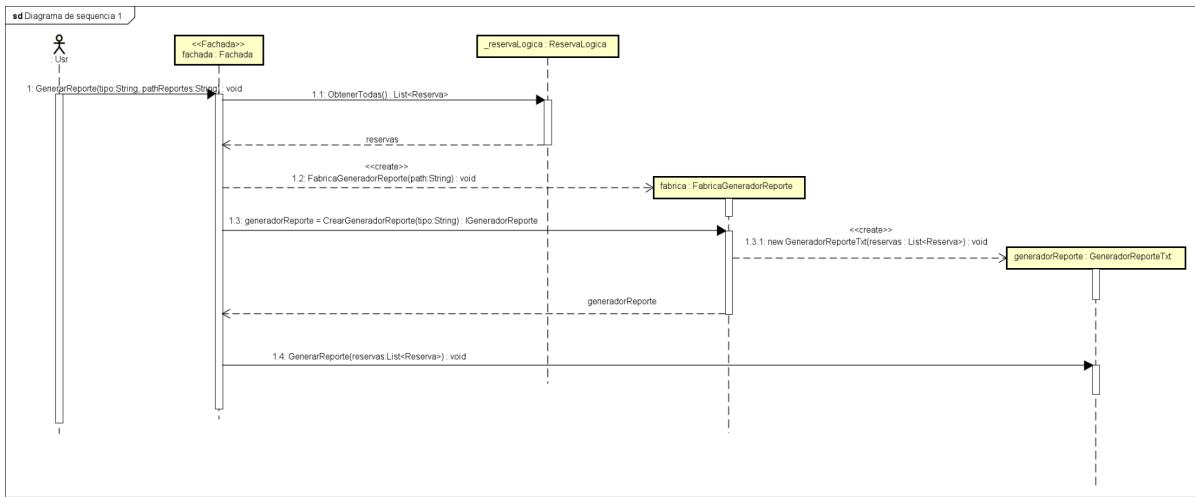
3



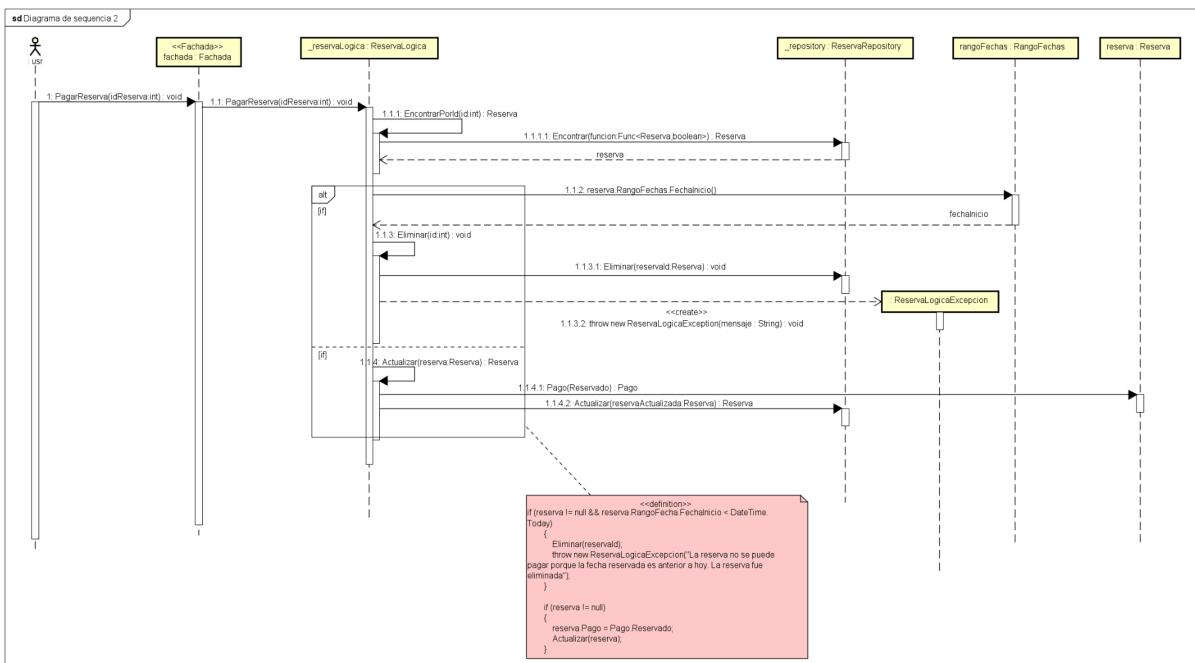
4



5



6



Constructor Clase Fachada

```
public Fachada(EFContext context)
{
    _promocionRepository = new PromocionRepository(context);
    _promocionLogica = new PromocionLogica(_promocionRepository);

    _depositoRepository = new DepositoRepository(context);
    _depositoLogica = new DepositoLogica(_depositoRepository);

    _reservaRepository = new ReservaRepository(context);
    _reservaLogica = new ReservaLogica(_reservaRepository);

    _usuarioRepository = new UsuarioRepository(context);
    _usuarioLogica = new UsuarioLogica(_usuarioRepository);
}
```

Configuracion EFContext

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<Reserva>() // EntityTypeBuilder<Reserva>
        .Property(r :Reserva => r.ConfAdmin)
        .HasConversion(new EnumToStringConverter<Estado>());

    modelBuilder.Entity<Reserva>() // EntityTypeBuilder<Reserva>
        .Property(r :Reserva => r.Pago)
        .HasConversion(new EnumToStringConverter<Pago>());

    modelBuilder.Entity<Deposito>() // EntityTypeBuilder<Deposito>
        .Property(d :Deposito => d.Tamanio)
        .HasConversion(new EnumToStringConverter<Tamanio>());

    modelBuilder.Entity<Deposito>() // EntityTypeBuilder<Deposito>
        .Property(d :Deposito => d.Area)
        .HasConversion(new EnumToStringConverter<Area>());

    modelBuilder.Entity<Deposito>() // EntityTypeBuilder<Deposito>
        .Property(d :Deposito => d.Climatizacion)
        .IsRequired();

    modelBuilder.Entity<Deposito>() // EntityTypeBuilder<Deposito>
        .Property(d :Deposito => d.Nombre)
        .IsRequired();

    modelBuilder.Entity<Reserva>() // EntityTypeBuilder<Reserva>
        .Property(r :Reserva => r.Comentario)
        .IsRequired(false);
    modelBuilder.Entity<Reserva>() // EntityTypeBuilder<Reserva>
        .Property(r :Reserva => r.ConfAdmin)
        .IsRequired();
    modelBuilder.Entity<Reserva>() // EntityTypeBuilder<Reserva>
        .Property(r :Reserva => r.Precio)
        .IsRequired();
```

```
modelBuilder.Entity<Usuario>() // EntityTypeBuilder<Usuario>
    .Property(u:Usuario => u.NombreCompleto)
    .IsRequired();
modelBuilder.Entity<Usuario>() // EntityTypeBuilder<Usuario>
    .Property(u:Usuario => u.Rol)
    .HasConversion(new EnumToStringConverter<Rol>())
    .IsRequired();

modelBuilder.Entity<Promocion>() // EntityTypeBuilder<Promocion>
    .Property(p:Promocion => p.Descuento)
    .IsRequired();
modelBuilder.Entity<Promocion>() // EntityTypeBuilder<Promocion>
    .Property(p:Promocion => p.Etiqueta)
    .IsRequired();

modelBuilder.Entity<Promocion>() // EntityTypeBuilder<Promocion>
    .HasMany(navigationExpression: p:Promocion => p.Depositos) // CollectionNavigationBuilder<Promocion, Deposito>
    .WithMany(navigationExpression: d:Deposito => d.Promociones) // CollectionCollectionBuilder<Deposito, Promocion>
    .UsingEntity<Dictionary<string, object>>(
        joinEntityName: "PromocionDeposito",
        configureRight: j:EntityTypeBuilder<Dictionary<...>> => j.HasOne<Deposito?>().WithMany().HasForeignKey("DepositoId"),
        configureLeft: j:EntityTypeBuilder<Dictionary<...>> => j.HasOne<Promocion?>().WithMany().HasForeignKey("PromocionId"),
        configureJoinEntityType: j:EntityTypeBuilder<Dictionary<...>> =>
    {
        j.HasKey(params propertyName: "PromocionId", "DepositoId");
    });
}
```

```
namespace Repositories;

    ↗ 19 usages ↗ 8 inheritors ↗ danielTorre
public interface IRepository <T, TKey>
{
        ↗ 4 usages ↗ 8 implementations ↗ danielTorre
    T Agregar(T element);

        ↗ 5 usages ↗ 8 implementations ↗ danielTorre
    T? Encontrar(Func<T, bool> func);

        ↗ 4 usages ↗ 8 implementations ↗ danielTorre
    T? Actualizar(T updateElement);

        ↗ 4 usages ↗ 8 implementations ↗ danielTorre
    void Eliminar(TKey id);

        ↗ 4 usages ↗ 8 implementations ↗ danielTorre
    List<T> ObtenerTodos();
}
```

```
using Dominio;

namespace Repositories;

[11 usages] danielTorre
public class UsuarioRepository : IRepository<Usuario, String>
{
    private EFContext _context;

    [6 usages] 101 tests OK, 1 failed * danielTorre
    public UsuarioRepository(EFContext context)
    {
        _context = context;
    }

    [2+4 usages] 62 tests OK * danielTorre
    public Usuario Agregar(Usuario usuario)
    {
        _context.Usuario.Add(usuario);
        _context.SaveChanges();
        return usuario;
    }

    [2+5 usages] 64 tests OK * danielTorre
    public Usuario Encontrar(Func<Usuario, bool> funcion)
    {
        return _context.Usuario.Where(funcion).FirstOrDefault();
    }

    [0+4 usages] 2 tests OK danielTorre
    public Usuario Actualizar(Usuario usuarioActualizado)
    {
        var findUsuario = Encontrar(funcion: u:Usuario => u.Email == usuarioActualizado.Email);
        findUsuario = usuarioActualizado;
        return findUsuario;
    }

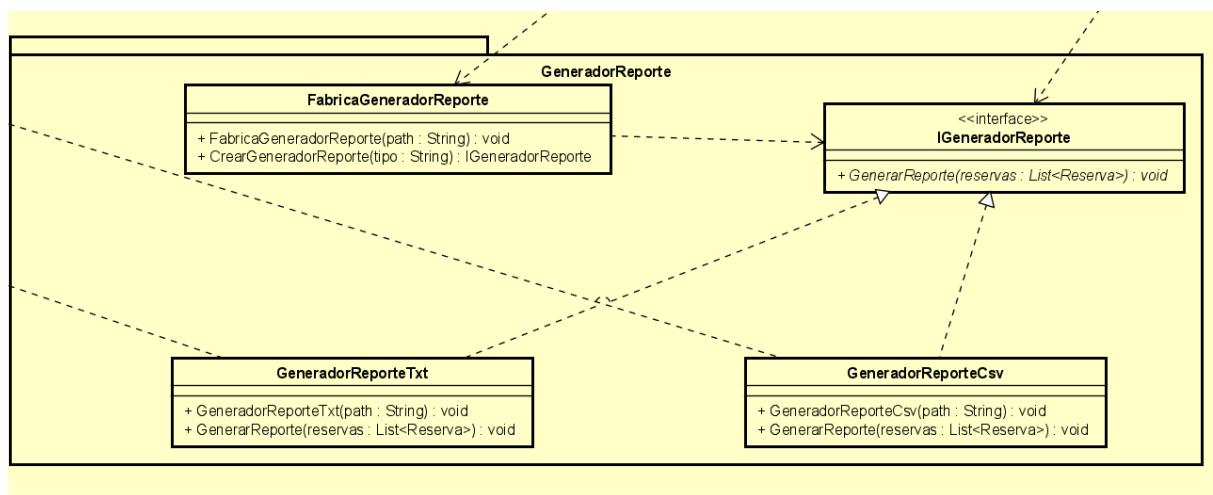
    [0+4 usages] 2 tests OK danielTorre
    public void Eliminar(String email)
    {
        var entidad:Usuario = Encontrar(funcion: u:Usuario => u.Email == email);
        _context.Usuario.Remove(entidad);
        _context.SaveChanges();
    }

    [0+4 usages] 60 tests OK * danielTorre
    public List<Usuario> ObtenerTodos()
    {
        return _context.Usuario.ToList();
    }
}
```

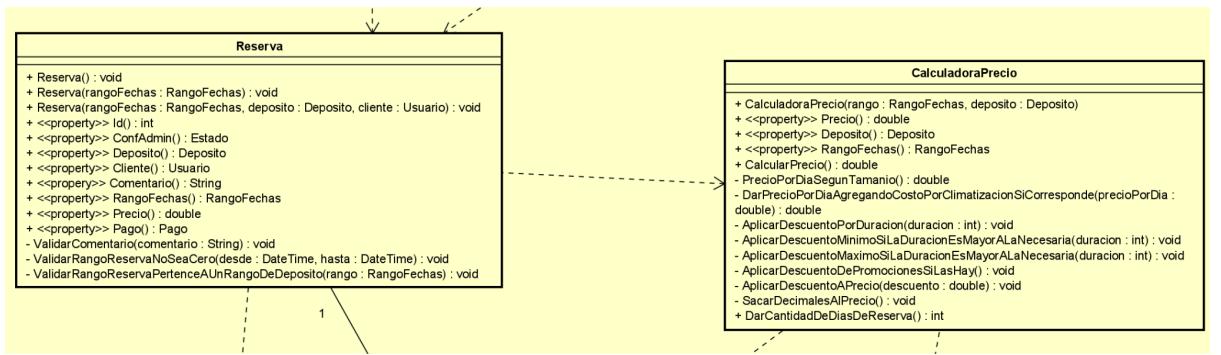
9

```
private void AltaDeposito()
{
    _mensajeError = "";
    try
    {
        if (_rangosFechas.Count() == 0)
        {
            _mensajeError = "El deposito al menos debe tener un rango de fechas de disponibilidad";
            return;
        }
        var datosDeposito = new DepositoAltaDto(_nombre, _area, _tamaño, _climatizacion,_idsPromociones, _rangosFechas);
        Fachada.AgregarDeposito(datosDeposito);
        navManager.NavigateTo(url: "/depositos/lista");
    }
    catch (Exception e)
    {
        Console.WriteLine($"Se a producido una excepción: {e.Message}");
        _mensajeError = e.Message;
    }
}
```

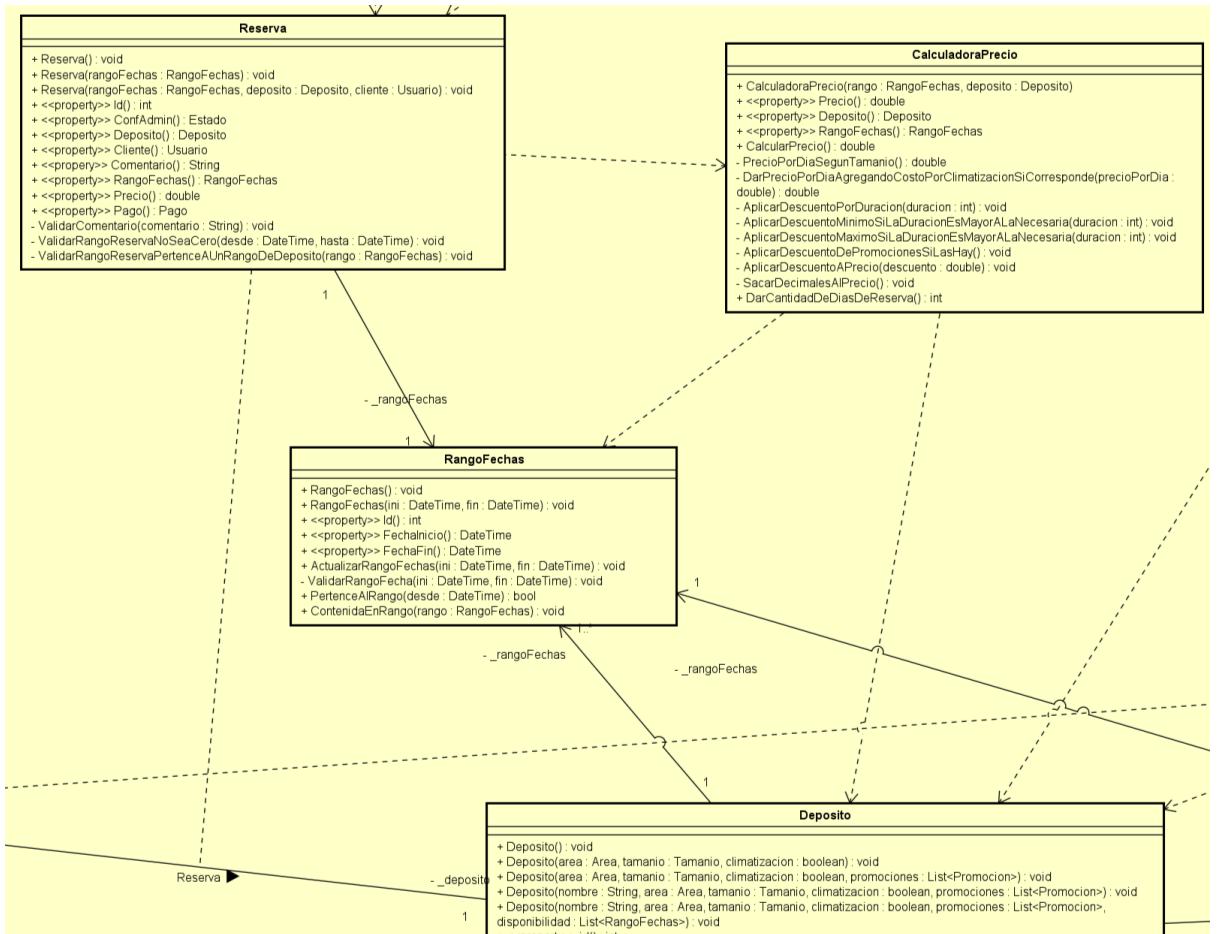
10



11



12



13

```
0+4 usages 4 tests OK * danielTorre +1
public void Eliminar(int id)
{
    var entidad:Promocion = Encontrar(funcion:p:Promocion => p.Id == id);
    var rangoFechasDeEntidad = entidad.RangoFecha;
    _context.Promocion.Remove(entidad);
    _context.RangoFechas.Remove(rangoFechasDeEntidad);
    _context.SaveChanges();
}
```

14

```
public void GenerarReporte(List<Reserva> reservas)
{
    List<String> lineas = new List<string>();
    lineas.Add(item:"Nombre deposito\tFecha reserva\tUsuario\tPrecio\tEstado del pago");
    foreach (var reserva in reservas)
    {
        String pagoEstado = reserva.Pago != null ? reserva.Pago.ToString() : "Sin_pago";
        var fechasString = $"{reserva.RangoFecha.FechaInicio.ToString("dd/MM/yyyy")}-{reserva.RangoFecha.FechaFin.ToString("dd/MM/yyyy")}";
        var depositoString = "Deposito eliminado";
        if (reserva.Deposito != null) depositoString = reserva.Deposito.Nombre;
        var linea = $"{depositoString}\t{fechasString}\t{reserva.Cliente.Email}\t${reserva.Precio}\t{pagoEstado}";
        lineas.Add(linea);
    }
    File.WriteAllLines(_path,lineas);
}
```

15

```
1 usage 172 tests OK ~ danieltorre
public EFContext(DbContextOptions<EFContext> options) : base(options)
{
    if (!Database.IsInMemory())
    {
        Database.Migrate();
    }
}
```

```
using Microsoft.EntityFrameworkCore;

namespace Repositories;

[12 usages] [danielTorre]
public class InMemoryEFContextFactory
{
    [6 usages] [172 tests OK] [danielTorre]
    public EFContext CreateDbContext()
    {
        DbContextOptionsBuilder<EFContext> optionsBuilder = new DbContextOptionsBuilder<EFContext>();
        optionsBuilder.UseInMemoryDatabase(databaseName: "TestDB");

        return new EFContext(optionsBuilder.Options);
    }
}
```

16

```
public class UsuarioLogicaTest
{
    private UsuarioRepository _usuarioRepository;
    private EFContext _context;
    private UsuarioLogica _usuarioLogica;
    private readonly InMemoryEFContextFactory _contextFactory = new InMemoryEFContextFactory();

    [TestInitialize]
    [20 tests OK] [danielTorre]
    public void SetUp()
    {
        _context = _contextFactory.CreateDbContext();
        _usuarioRepository = new UsuarioRepository(_context);
        _usuarioLogica = new UsuarioLogica(_usuarioRepository);
    }

    [TestCleanup]
    [20 tests OK] [danielTorre]
    public void CleanUp()
    {
        _context.Database.EnsureDeleted();
    }
}
```

Diagrama de paquetes entrega pasada

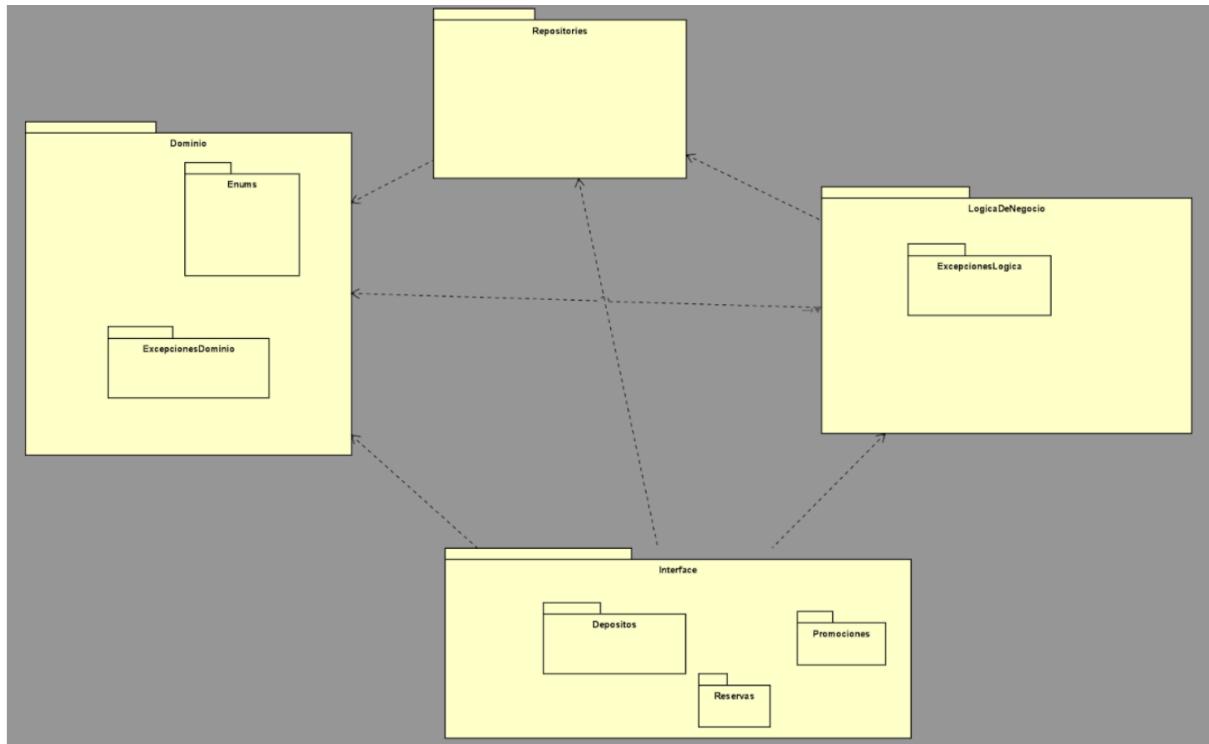


Diagrama de paquetes de esta entrega

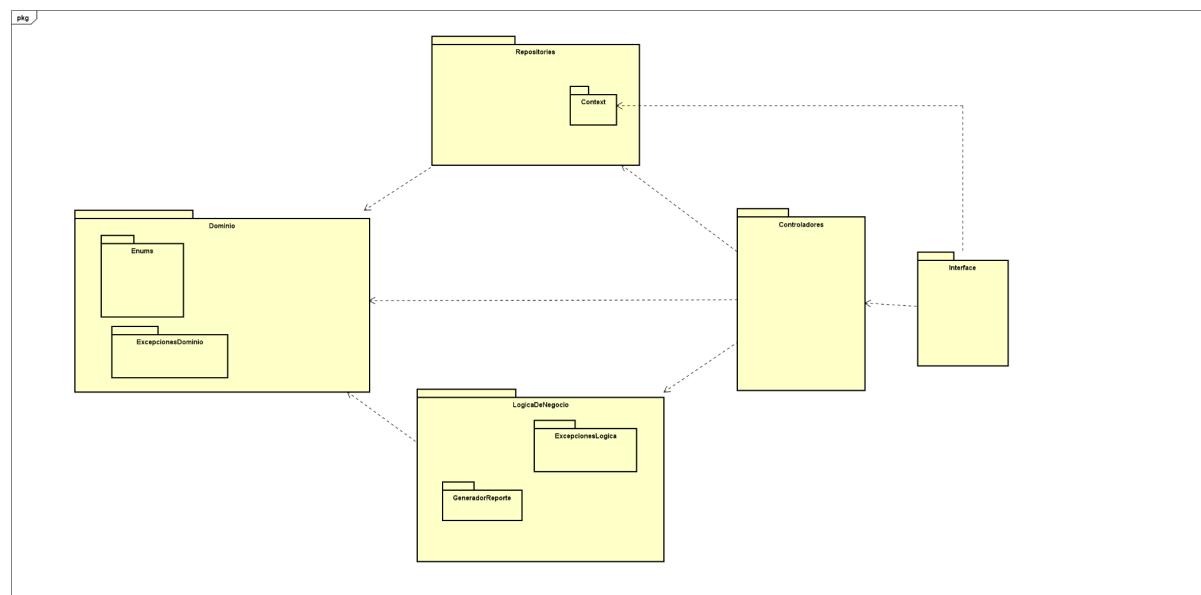


Diagrama Dominio de entrega 1

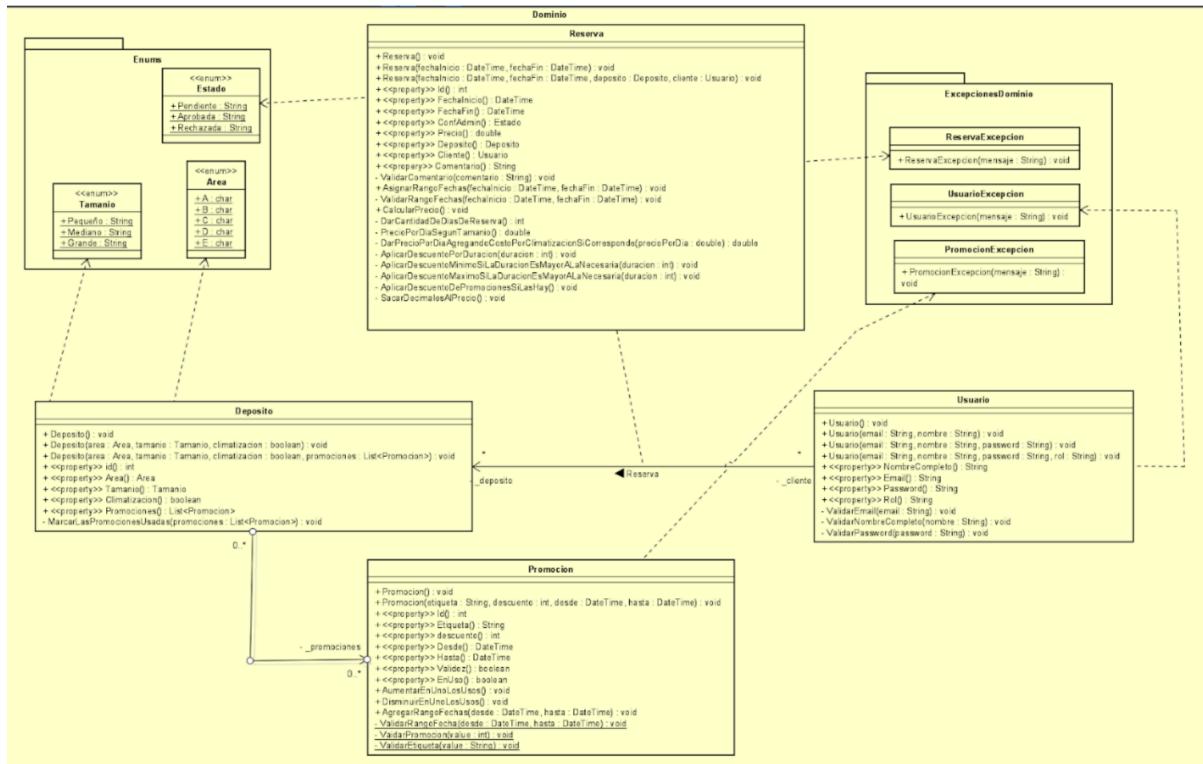
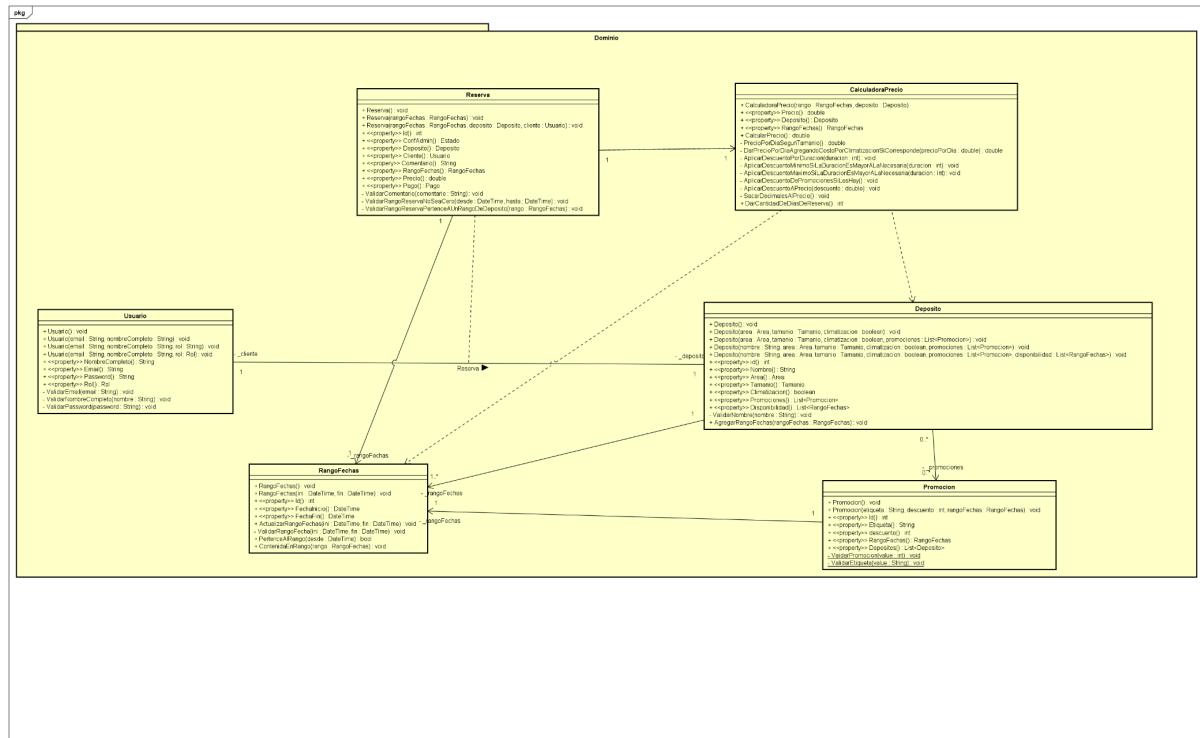
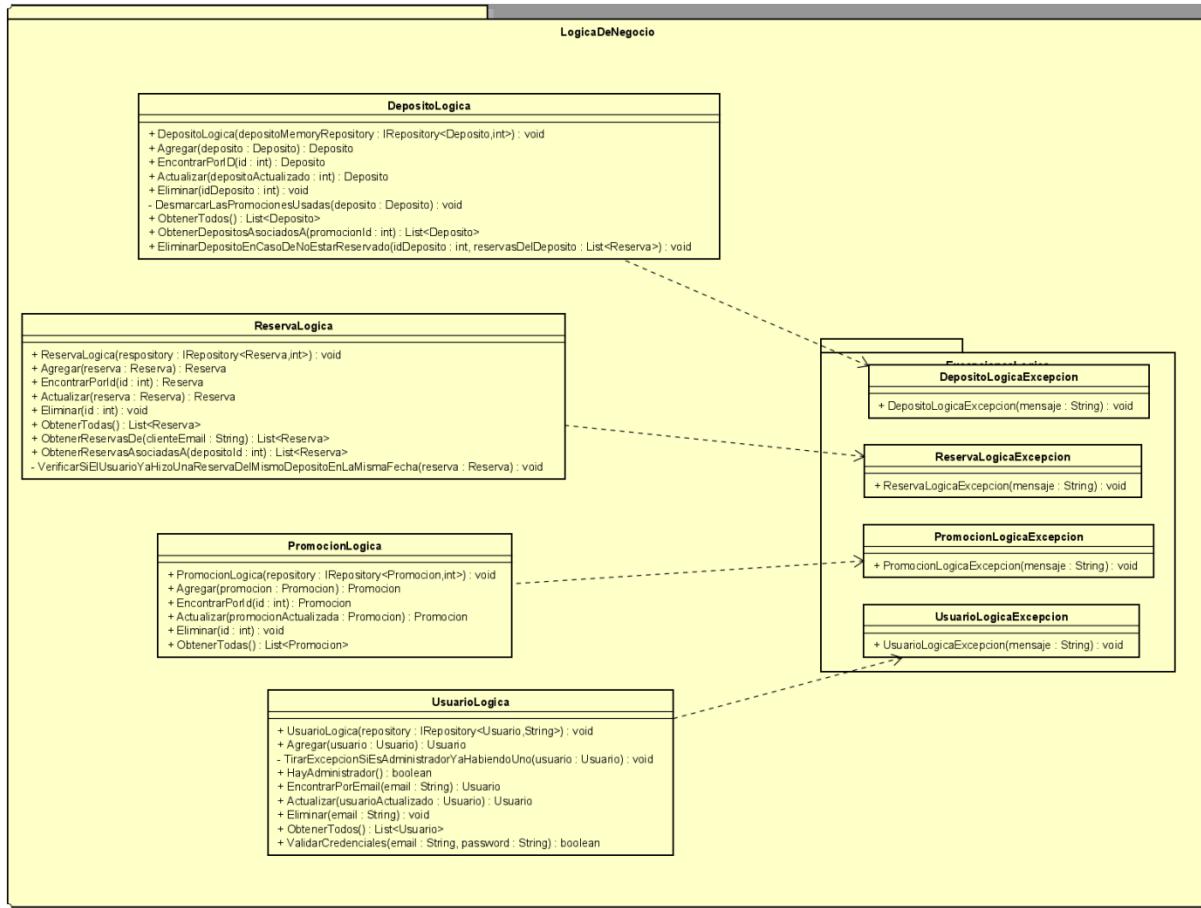


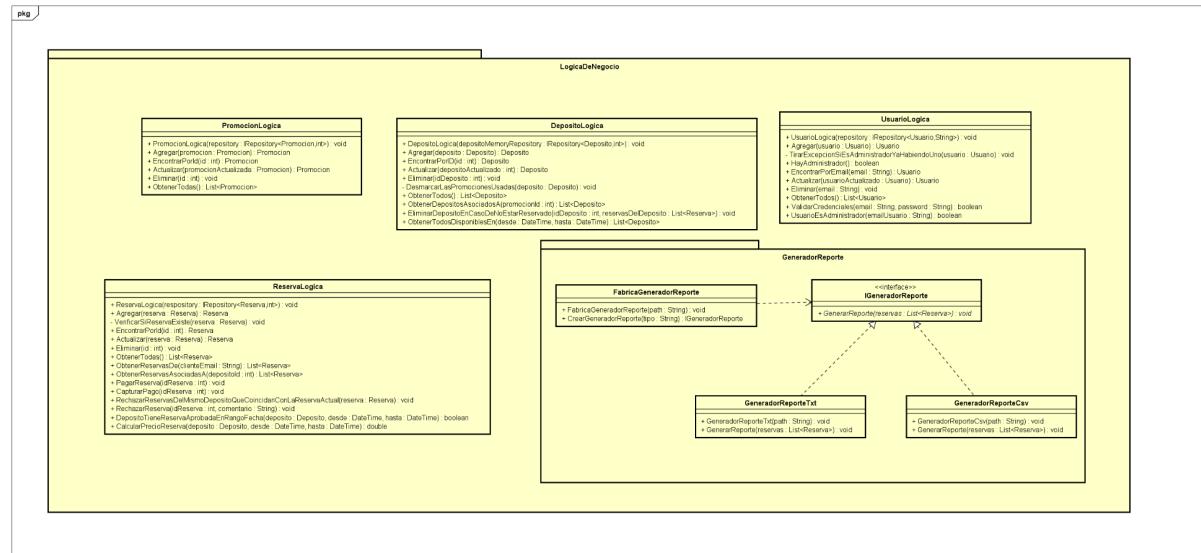
Diagrama Dominio de entrega 2



Paquete LogicaDeNegocio entrega 1

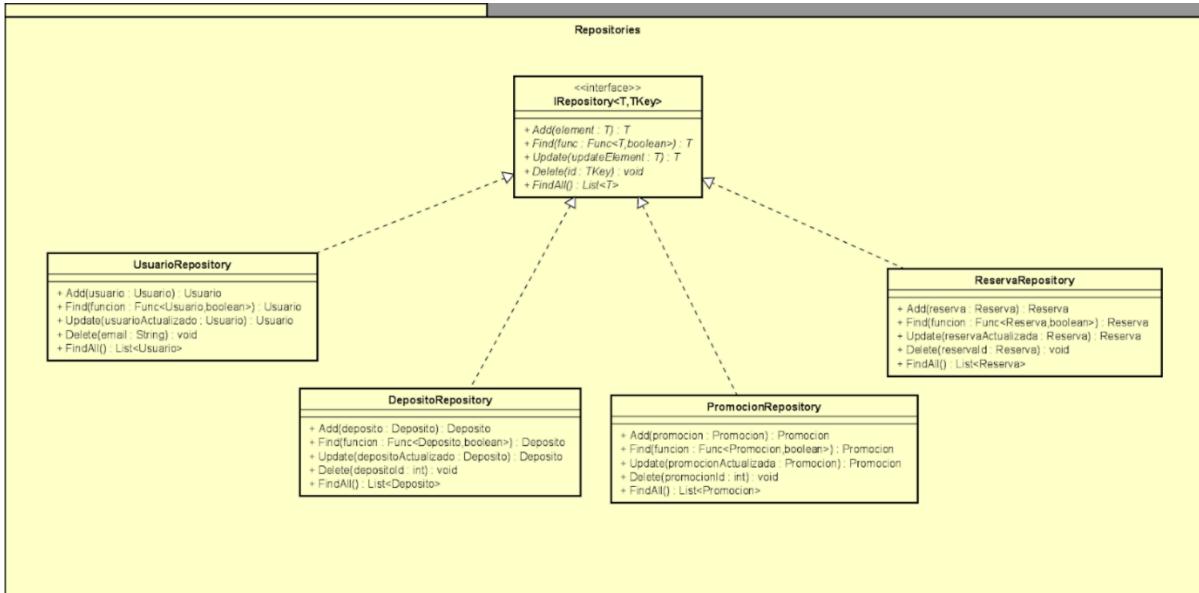


Paquete LogicaDeNegocio entrega 2



20

Paquete Repositories entrega 1



Paquete Repositories entrega 2

