

Universidad ORT Uruguay

Facultad de Ingeniería



Obligatorio 2 Diseño de Aplicaciones

Enzo Izquierdo (283145)

Manuel Graña (285727)

[Repositorio](#)

2024

Índice

Índice	2
Introducción	3
Descripción General	3
Funcionalidades Implementadas	3
Instalación	4
Descripción de Arquitectura y Decisiones de Diseño Clave	4
Arquitectura del Sistema	4
Organización de la aplicación	5
Namespaces	7
Service - Repository	8
Dominio	11
Gestión de pago	11
Gestión de disponibilidad de reservas	12
Generación de reportes de reservas	12
Interacción entre UI y BusinessLogic	14
Manejo de excepciones	16
Análisis de dependencias en “Exportar reporte de reservas”	16
Cohesión	16
Acoplamiento	17
Persistencia y base de datos	18
Cobertura de pruebas unitarias	19
Anexo	20
Enlaces	20



DepoQuickApp

Introducción

DepoQuickApp es una plataforma digital diseñada para facilitar y optimizar el proceso de alquiler de depósitos. Este sistema permite a los usuarios, tanto administradores como clientes, gestionar reservas de depósitos de forma eficiente y efectiva.

Descripción General

Funcionalidades Implementadas

Se implementaron todas las funcionalidades previstas para esta entrega.

- **Registro y gestión de usuarios**
- **Administración de depósitos**
- **Administración de promociones**
- **Reserva de depósitos**
- **Confirmación de reservas**
- **Cálculo de precio de depósitos**
- **Gestión de disponibilidad de reservas**
- **Pagos**
- **Exporte de reservas**

Instalación

La base de datos del proyecto corre en un contenedor de Docker, por lo que lo primero es configurarlo. Iremos al archivo “/docker-compose.yml” en el directorio raíz y lo podremos configurar. Hay que indicar la contraseña en **SA_PASSWORD** y el puerto en **ports**. Por defecto son “*Passw1rd*” y “*1433*”, respectivamente.

Luego debemos configurar los connection strings para poder realizar la conexión a este. En la carpeta publish, en el archivo appsettings.json configuraremos el string para la versión de producción. Buscaremos la línea “DefaultConnection”. Aquí podremos configurar la IP (Server), nombre de la base de datos (Database), User Id y Password, esta última debe ser la misma que utilizamos en el archivo “/docker-compose.yml”.

Por último, debemos ejecutar UI.exe dentro de publish.zip, en el release, y abrir localhost en el puerto que se indica en la consola en el navegador.

Ubicación de los scripts SQL con datos de prueba: En la release, dentro de publish.zip en la carpeta /Test Data. Aquí podremos encontrar tanto el DDL como inserts con los datos, ambos en sql.

Usuarios creados en los datos de prueba:

- **Administrador:** *admin@test.com* - contraseña: *1234567@Ad*
- **Cliente 1:** *client@test.com* - contraseña: *1234567@Cl*
- **Cliente 2:** *user2@test.com* - contraseña: *1234567@Cl*

Librerías requeridas para release: Microsoft.EntityFrameworkCore 7.0.20,
Microsoft.EntityFrameworkCore.Design, Microsoft.EntityFrameworkCore.SqlServer

Para desarrollo: MSTest, Microsoft.NET.Test.Sdk, Microsoft.EntityFrameworkCore.Sqlite

Descripción de Arquitectura y Decisiones de Diseño Clave

Arquitectura del Sistema

Diseño en tres capas: La aplicación fue desarrollada siguiendo una arquitectura de tres niveles, con una capa de presentación hecha en Blazor, una capa de lógica de negocio en C# y una capa de acceso a datos basada en Entity Framework y conectada a una base de datos Azure SQL Edge corriendo en un contenedor de Docker. La comunicación entre la interfaz y la lógica de negocios se hace a través de DTOs (Data Transfer Objects).

Organización de la aplicación

La aplicación fue diseñada dividiendo la solución en cuatro proyectos principales: BusinessLogic, Domain, DataAccess y UI. Tenemos proyectos adicionales, que se implementaron para respetar principios de inversión de dependencias (se detalla más adelante, fue hecho de esta forma en el cálculo de precio y generación de reportes) y también para tener mejor organizado el proyecto. Cada uno de estos proyectos se organizó en varios namespaces, asignados de tal manera que agrupan responsabilidades similares. A continuación, se muestra una tabla donde se indican las clases junto con su namespace y su responsabilidad.

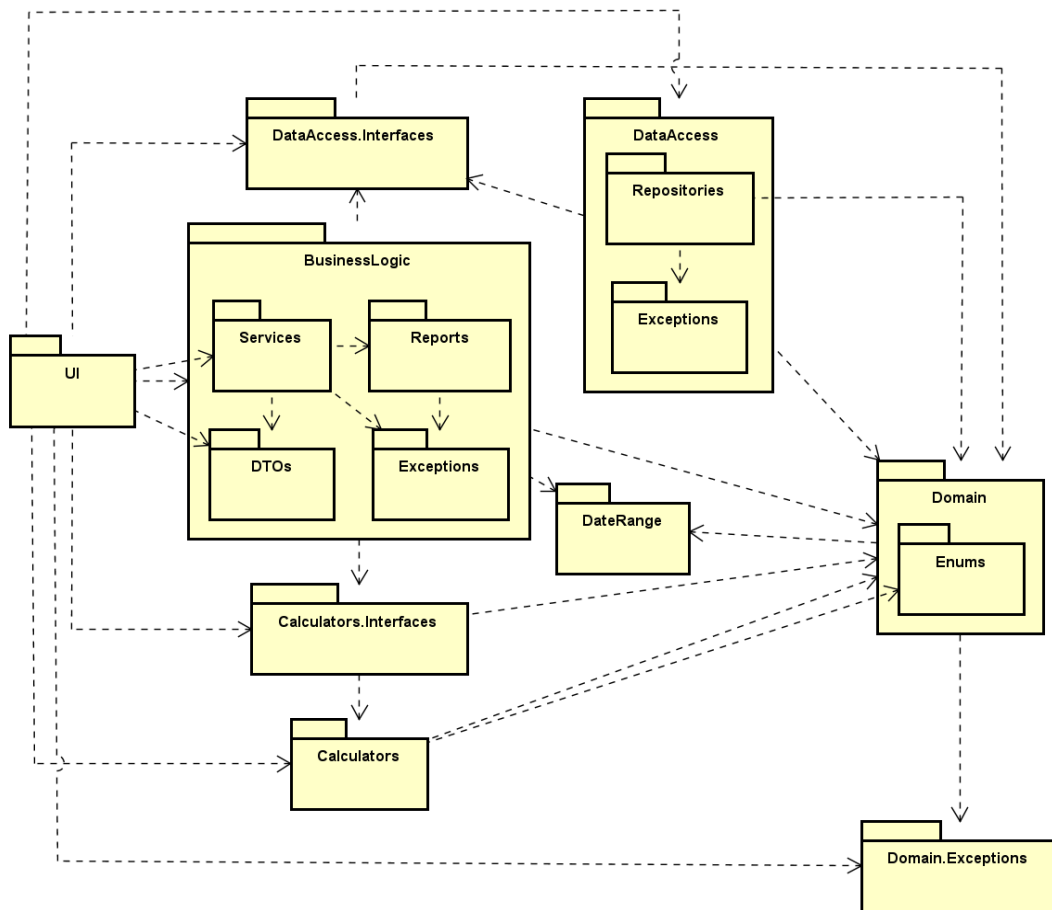
Namespace	Clase	Responsabilidad
Domain	User	Gestión y validación de datos de Usuario
Domain	Promotion	Gestión y validación de datos de Promoción
Domain	Deposit	Gestión y validación de datos de Depósito
Domain	Booking	Gestión y validación de datos de Reserva
Domain	Payment IPayment	Gestión y validación de datos de Pago asociado a una reserva
Domain	AvailabilityPeriods	Gestión y validación de datos de la lista de períodos

		de disponibilidad de un depósito
BusinessLogic.Services	UserService	Controlador de las operaciones sobre usuario. Responsable de pasaje de DTO a objeto y viceversa
BusinessLogic.Services	PromotionService	Controlador de las operaciones sobre promoción. Responsable de pasaje de DTO a objeto y viceversa
BusinessLogic.Services	DepositService	Controlador de las operaciones sobre depósito. Responsable de pasaje de DTO a objeto y viceversa
BusinessLogic.Services	BookingServices	Controlador de las operaciones sobre reserva. Responsable de pasaje de DTO a objeto y viceversa
Calculators Calculators.Interfaces	IPriceCalculator PriceCalculator	Cálculo de precio de un depósito. Fabricación pura.
BusinessLogic.Reports	IBookingReport TxtBookingReport CsvBookingReport	Generación de reportes en cierto formato de archivo. La responsabilidad de generar un Txt o un Csv depende del tipo de la clase, aplicamos Polimorfismo.
BusinessLogic.Reports	BookingReportFactory	Fábrica para la generación de reportes. Creador de IBookingReport.
DataAccess.Interfaces	IUserRepository IPromotionRepository IDepositRepository IBookingRepository	Interfaces que establecen las operaciones CRUD (solo las necesarias para cada clase) de cada repositorio.
DataAccess.Repositories	UserRepository	Operaciones de Creación y Lectura de usuarios
DataAccess.Repositories	PromotionRepository	Operaciones CRUD sobre promociones
DataAccess.Repositories	DepositRepository	Operaciones CRUD sobre depósitos
DataAccess.Repositories	BookingRepository	Operaciones de Creación, Lectura y Modificación sobre reservas
BusinessLogic.Exceptions Domain.Exceptions DateRange.Exceptions DataAccess.Exceptions	BusinessLogicException DomainException DateRangeException DataAccessException	Excepciones de su respectivo paquete

Además contamos con los namespaces “DTOs” en BusinessLogic, que tiene todos los Data Transfer Objects y “Enums” en Domain con todos los enums. También tenemos un namespace .Test para los namespaces “Domain”, “BusinessLogic” y “DateRange”. Esta última contiene una clase auxiliar que utilizamos para guardar y manejar operaciones sobre rangos de fechas.

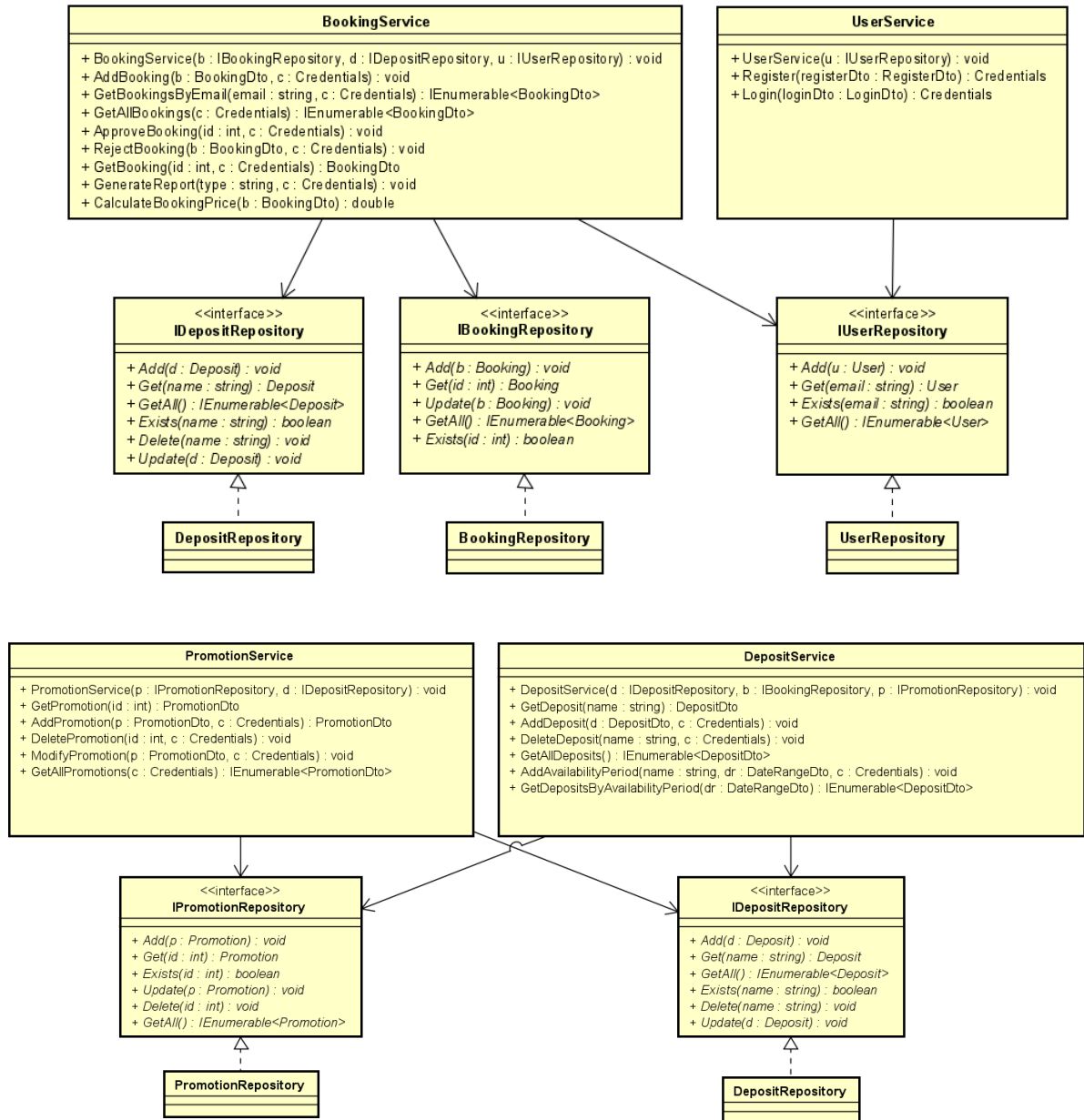
Namespaces

A continuación, se muestra el diagrama de paquetes de la solución. La solución está dividida en namespaces de manera que agrupen responsabilidades similares. **UI** contiene todo el proyecto de la interfaz Blazor, **Domain** agrupa las clases del dominio (lo separamos de BusinessLogic en esta entrega para evitar dependencia circular con DataAccess), **DataAccess** gestiona el contexto de la base de datos y los repositorios (en **DataAccess.Interfaces** se encuentran las abstracciones de los repositorios como detallamos más adelante), **BusinessLogic** donde se encuentran los controladores (Services) como se detalla más adelante y las clases relacionadas a la generación de reportes. Así mismo, se cuenta con **Calculators** (junto con **Calculators.Interfaces** donde ubicamos la abstracción IPriceCalculator para respetar el principio SOLID de inversión de dependencias) y **DateRange**, la primera se encarga del cálculo de precio de depósito y el segundo contiene una clase auxiliar que usamos para manejar los rangos de fechas.



Service - Repository

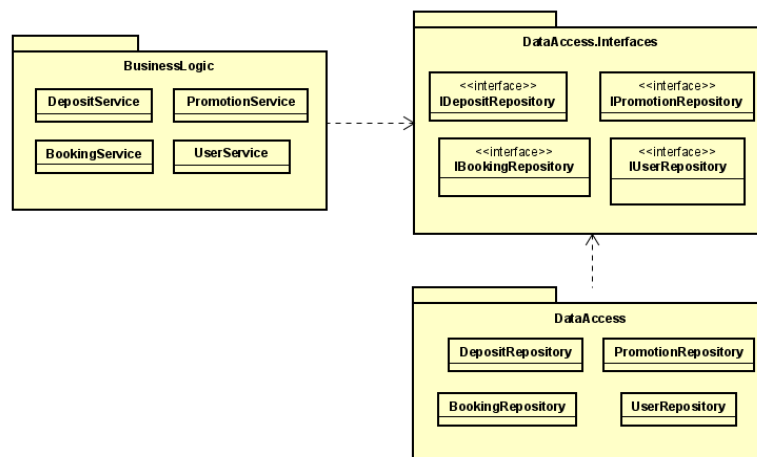
En la entrega anterior, teníamos una clase “Manager” para cada entidad del dominio. Violabamos el principio de responsabilidad única ya que dicha clase se encargaba tanto del acceso a los datos como la lógica de negocios para un caso de uso. En esta entrega, reemplazamos dicha clase por una clase Service y Repository para cada entidad relevante del dominio, en particular Booking, Deposit, Promotion y User. Esta división nos permite separar la responsabilidad de acceder a los datos de las operaciones de la lógica de negocios.



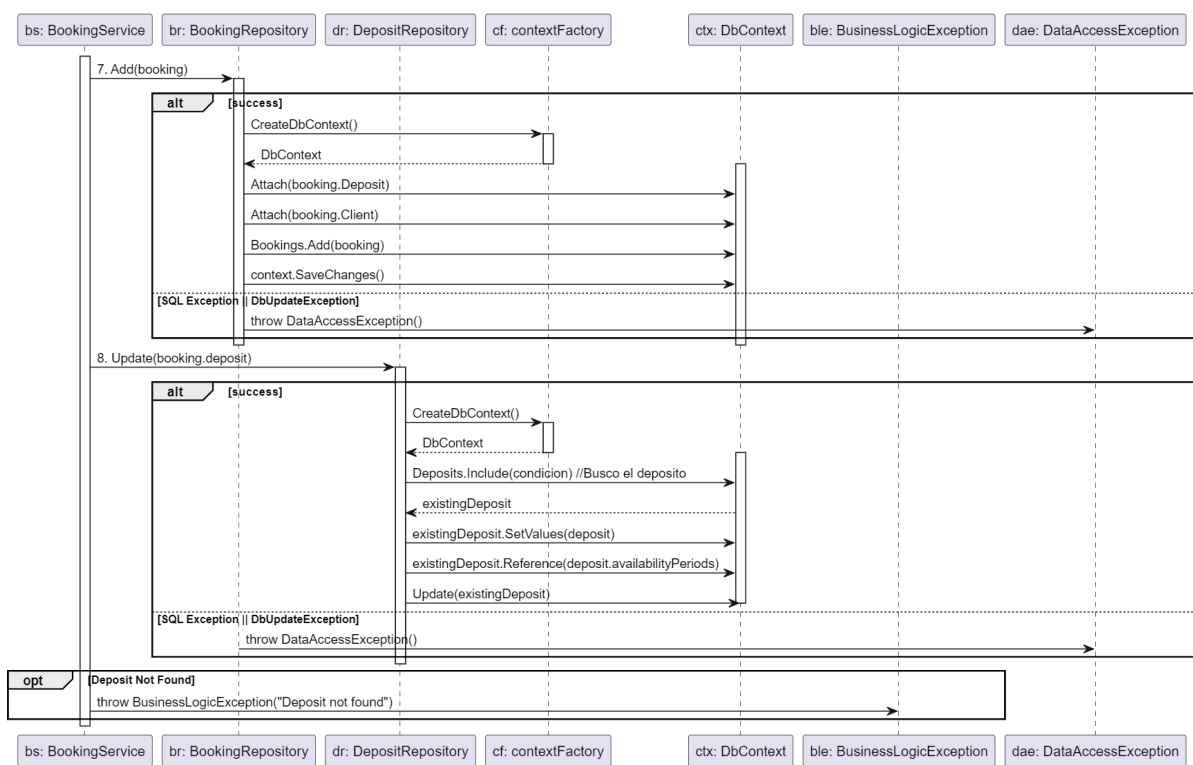
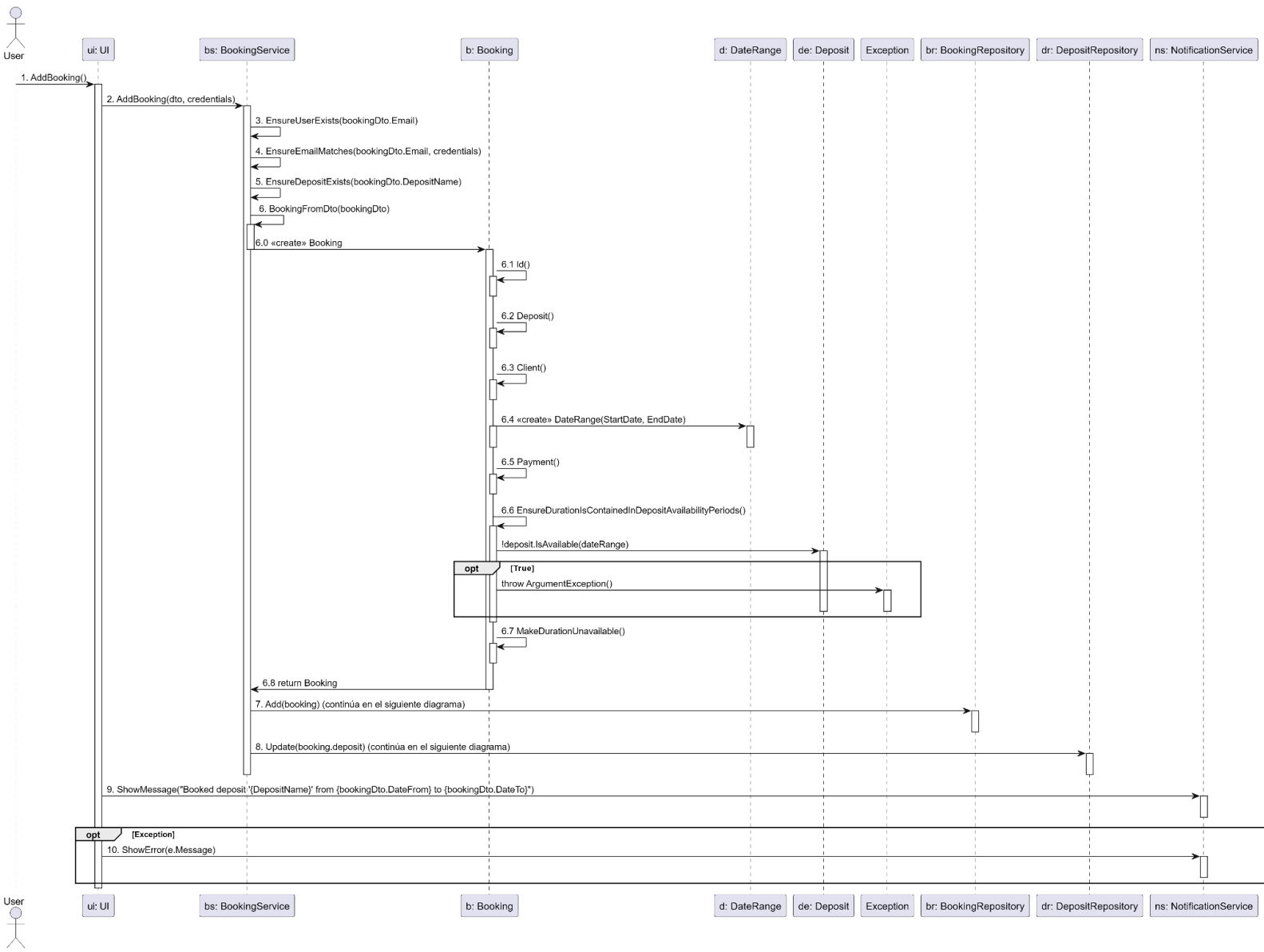
La clase Repository es responsable de abstraer el acceso a los datos. Encapsula las operaciones básicas que se realizan sobre la base de datos (Add, Get, Delete, Modify, ...). Se encuentra en el paquete DataAccess. La clase Service es responsable de las operaciones de la lógica de negocios. Delega las operaciones de agregar, eliminar, modificar al repositorio pero realiza las validaciones necesarias propias de la lógica de negocio antes de ejecutar cada operación.

Para cada entidad del dominio, tenemos una interfaz I<Entidad>Repository y a cada service le inyectamos él o los repositorios que necesite. De esta manera estamos respetando el principio SOLID Open-Closed, ya que si se agregara una nueva forma de acceder a los datos simplemente hay que crear nuevas clases que implementen las interfaces de los repositorios e inyectarlas a los servicios, sin modificar el código existente. Esto ya ocurrió en el proyecto cuando pasamos de almacenar los datos en memoria en la primera entrega a almacenar los datos en una base de datos en esta entrega.

Además, respetamos el principio de inversión de dependencias (DIP), ya que las clases service dependen de abstracciones de los repositorios; que se encuentran en paquetes separados como se muestra a continuación. Si se agrega un nuevo repositorio no hay que compilar todo el código de los servicios de nuevo, ni el paquete de interfaces, sino que simplemente debemos compilar el paquete donde colocamos los repositorios.

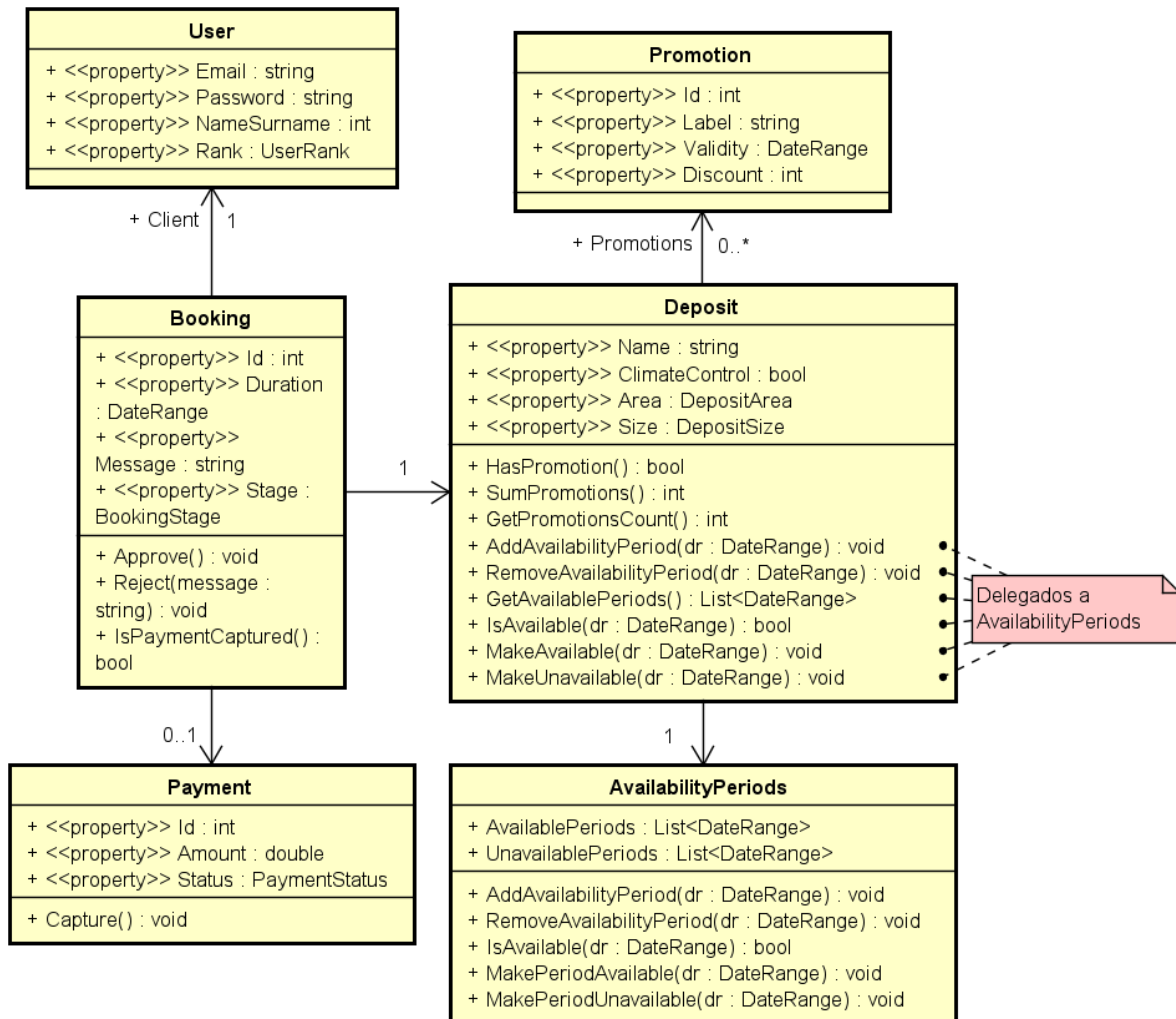


A continuación el diagrama de secuencia de la funcionalidad “Reserva de depósito” para ejemplificar el uso de estos:



Dominio

Se agregaron dos nuevas clases al dominio. El resto se mantuvo sin cambios a excepción de Depósito, donde agregamos métodos para gestionar el período de disponibilidad (delegados a la nueva clase **AvailabilityPeriods**) con la finalidad de respetar la Ley de Demeter en las clases que usan Deposit (evitar que hablen con AvailabilityPeriods), y Booking, donde por el mismo motivo agregamos un método que informe si el pago (clase **Payment**) está capturado.



Gestión de pago

La clase **Payment** es responsable de gestionar el estado del pago, conteniendo así la información referente al mismo. De este modo, al no tener el estado del pago en Booking,

respetamos el principio de responsabilidad única (SRP). Booking sólo tiene que cambiar si cambian las reglas de negocio referente a las reservas, mientras que Payment cambia si cambian las reglas de negocio referente a los pagos, por ejemplo si cambia la forma en que se capturan los pagos o si se agregan nuevos estados.

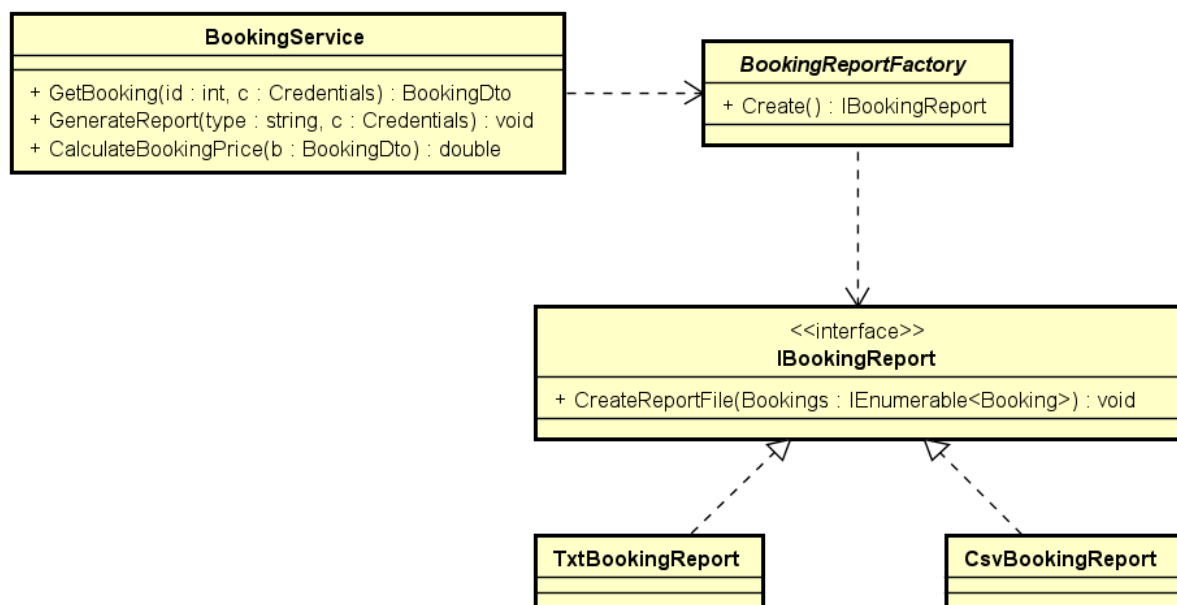
Gestión de disponibilidad de reservas

La clase **AvailabilityPeriods** se utiliza para gestionar los períodos de disponibilidad y no disponibilidad de los depósitos. Incorpora toda la lógica de negocios referente a los períodos de disponibilidad, por ejemplo, se encarga de fusionar las fechas de disponibilidad si estas se encuentran con períodos superpuestos. La clase Deposit delega entonces todo lo referido a la disponibilidad del mismo a esta clase.

A su vez, creamos una clase auxiliar **DateRange** que representa un rango de fechas y agrupa los métodos que nos permiten saber si dos fechas son iguales, si dos fechas son adyacentes, están contenidas entre sí, entre otras funciones.

Generación de reportes de reservas

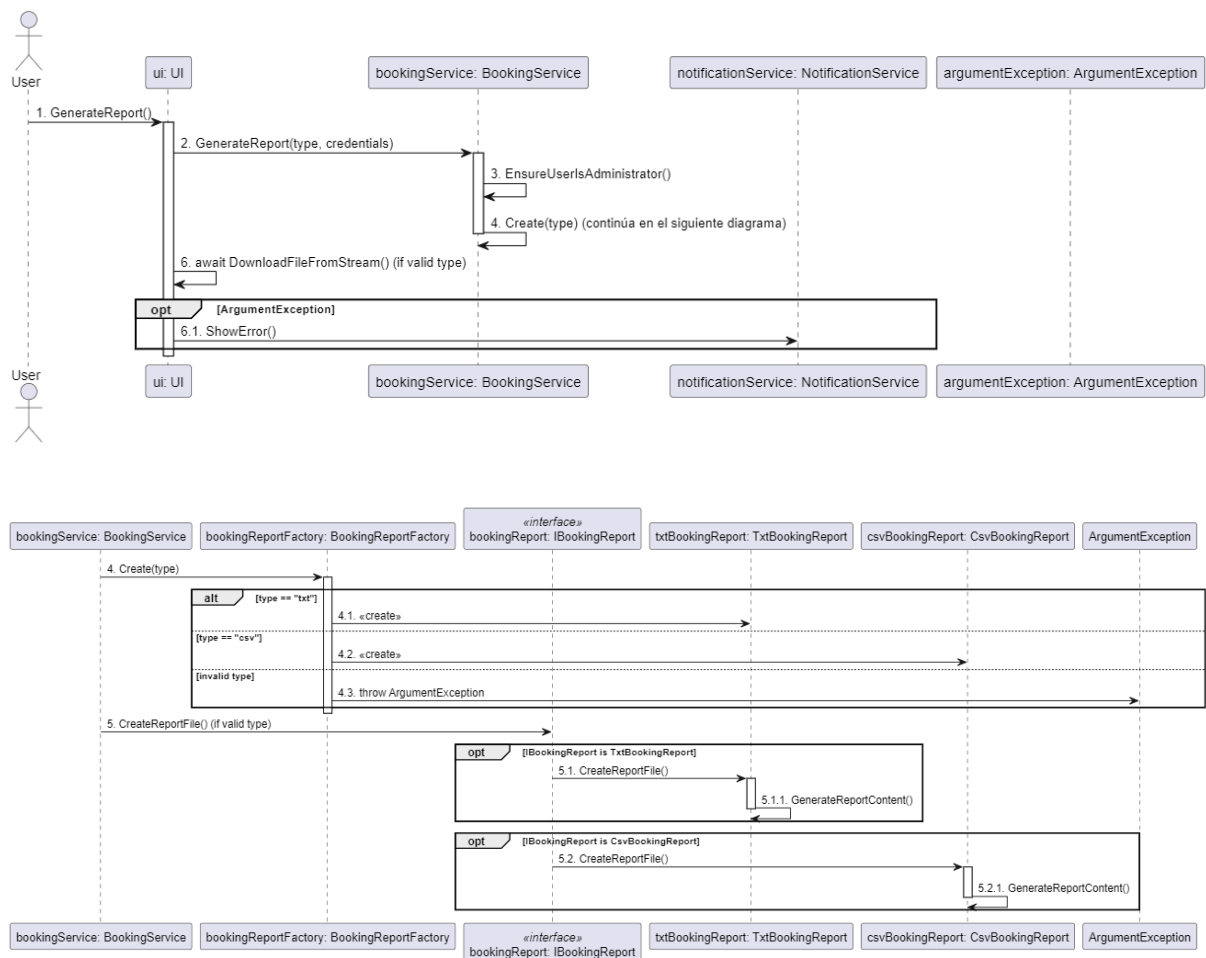
A la hora de diseñar la funcionalidad de reportes, seguimos varios principios SOLID y GRASP como se detalla a continuación.



Hacemos uso de una Factory (**BookingReportFactory**) que se utiliza para crear instancias de los reportes. Se observa así el patrón GRASP Creador en esta clase, puesto que la responsabilidad de crear instancias de IBookingReport fue asignada a esta clase (es su única responsabilidad; por lo que también respetamos el principio SOLID de responsabilidad única).

Aplicamos también Polimorfismo en la interfaz IBookingReport y sus dos implementaciones concretas TxtBookingReport y CsvBookingReport: la responsabilidad depende del tipo de objeto. Se respeta también el principio SOLID Open/Closed, puesto que si apareciera un formato nuevo de reporte, basta con agregar una clase que implemente IBookingReport. Debemos actualizar también la Factory.

A continuación se muestra el diagrama de secuencia del requerimiento:



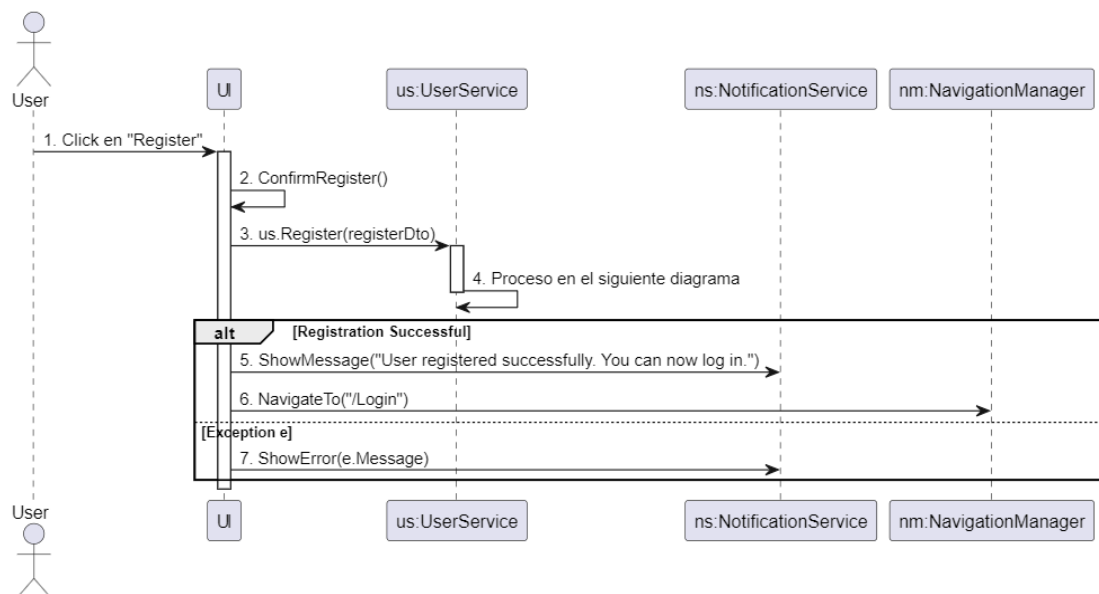
Interacción entre UI y BusinessLogic

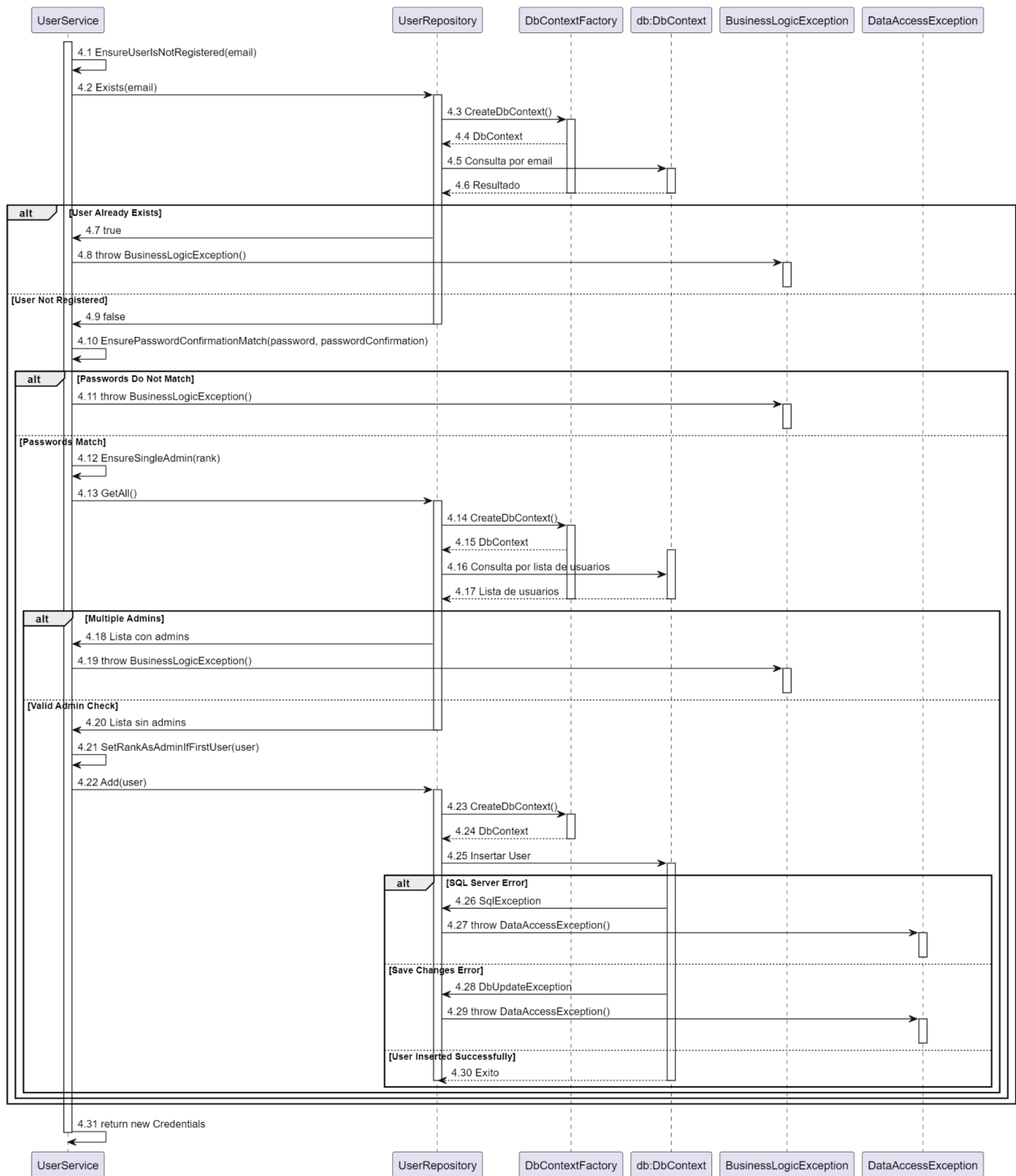
La interacción entre la UI y Business Logic se realiza, como en la anterior entrega, mediante Objetos de Transferencia de Datos (DTOs). La UI comunica sus requerimientos a los controladores de caso de uso Service y estos realizan acciones sobre el sistema.

Usar DTOs nos permite garantizar que la UI no necesite ni deba conocer detalles de la implementación del sistema y mucho menos conocer a las clases de nuestro dominio. Estos DTOs se crean en la UI y son enviados a los Service, entonces, en vez de instanciar clases del dominio en la interfaz; nos limitamos a crear estos objetos.

Los datos que se envían desde la BusinessLogic también se encapsulan en DTOs. Esto no solo simplifica la transferencia de datos entre la interfaz y el dominio, sino que también mantiene la separación y la independencia de las capas del sistema, lo cual es crucial para la mantenibilidad y escalabilidad del proyecto.

A continuación se muestra el diagrama de secuencia de **registro de usuario**, mostrando lo mencionado anteriormente (manejo de excepciones simplificado para ser más comprensible):





Manejo de excepciones

En nuestro sistema, utilizamos una excepción por paquete, es decir tenemos una excepción en BusinessLogic, Domain, DataAccess y DateRange. Particularmente, hicimos un paquete separado de Dominio llamado Domain.Exceptions para no acoplar la interfaz al dominio.

Las excepciones son lanzadas por las clases de cada paquete cuando ocurren errores en las operaciones. Estas excepciones son capturadas en la UI, donde se manejan adecuadamente, no permitiendo a la operación continuar y mostrando su mensaje asociado, incluso enviando al usuario a otras pantallas o hacia atrás, llegando a cerrar la sesión en caso de una caída de la base de datos. En caso de que se produzca una excepción que no contemplamos, se informa en la UI que ocurrió una excepción inesperada, evitando que el usuario vea todo el código de error sin manejar.

Contemplar las excepciones de esta forma nos permite saber que las excepciones que llegan a la interfaz de usuario son excepciones que esperábamos y provienen de nuestra implementación. En la anterior entrega usábamos las excepciones provistas por .NET, pero no podíamos diferenciar si por ejemplo, un error de ArgumentException era lanzado por nosotros o por algo no contemplado en nuestro sistema.

Análisis de dependencias en “Exportar reporte de reservas”

Para esta funcionalidad, interactúan de forma directa las clases BookingService, IBookingRepository, BookingReportFactory y una clase que implemente IBookingReport. También interactúan IPriceCalculator para el cálculo de precio y Booking (indirectamente las otras clases del dominio que interactúan con Booking).

Cohesión

- **Booking:** Cohesión media. La validación se hace mediante sus propios atributos. Las responsabilidades de gestión de fechas se delegan a DateRange, la validación de los pagos se delega a Payment, la gestión de períodos de disponibilidad se delega a Deposit.

- **BookingService:** Alta cohesión. Actúa como controlador de caso de uso para los métodos relacionados a reservas, en esta caso generar un reporte de las mismas, solo llama a otras funciones pero delega todo el trabajo. No conoce que tipo de reporte va a generar ya que utiliza la interfaz.
- **BookingRepository:** Alta cohesión. Solo es usada para operaciones CRUD, en este caso se llama para pasarle todas las reservas al Booking Report que corresponda.
- **BookingReportFactory:** Alta cohesión. Implementación del patrón factory para la creación de IBookingReport a partir de un string, en este caso la extensión del archivo.
- **IBookingReport - TxtBookingReport, CsvBookingReport:** Alta cohesión. Todos sus métodos cumplen la función de generar un archivo con el reporte de reservas. Las clases que la implementan tienen solo sus métodos y auxiliares para estos.
- **IPriceCalculator - PriceCalculator:** Baja cohesión. La clase no usa sus propios atributos para calcular, sino que depende de parámetros externos.

Acoplamiento

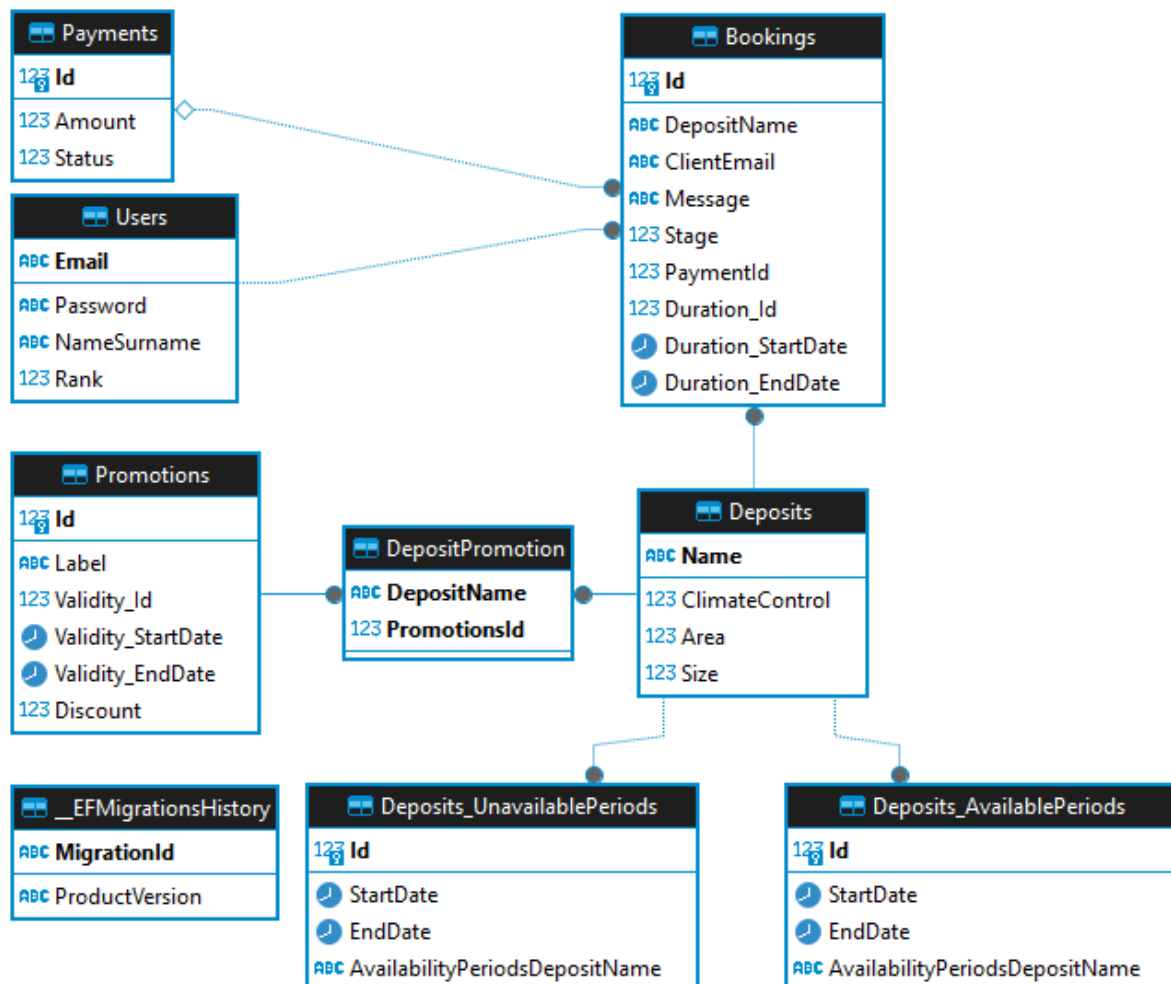
- **Booking:** Acoplamiento medio. Depende directamente de Deposit, User y Payment. Utiliza métodos de Deposit y Payment para delegarles responsabilidad.
- **BookingService:** Acoplamiento medio. El esperado de un controlador, llama a métodos de las otras clases en cuestión, por lo que está relacionado con todos. Sin embargo, depende de abstracciones IBookingRepository y IBookingReport lo que reduce el acoplamiento.
- **BookingRepository:** Bajo acoplamiento. No conoce de la existencia de las otras clases, solo conoce las reservas y es llamada por sus métodos.
- **BookingReportFactory:** Bajo acoplamiento. Esta clase al ser una factory solo cambia si se agrega una nueva implementación de la interfaz o se quita una ya existente. Depende solo de la interfaz IBookingReport.
- **IBookingReport - TxtBookingReport, CsvBookingReport:** Bajo acoplamiento, en nuestro caso usamos TxtBookingReport y CsvBookingReport, hablaremos de ellas. Al estar generando un reporte con sus datos, si la clase cambia tendremos que

modificar los reportes. Además también utilizan IPriceCalculator para calcular el precio de la reserva. Si bien es verdad que esto es así en estas implementaciones de IBookingReport, si en el futuro no se requiriera del precio de la reserva, ya no estaría esta última dependencia.

- **IPriceCalculator - PriceCalculator**: Alto acoplamiento. Esta clase está fuertemente acoplada a Deposit ya que necesita acceder a sus atributos y además necesita conocer la suma de sus promociones. No usa sus propios atributos para el cálculo.

Luego de realizado el análisis, concluimos que la funcionalidad tiene una cohesión alta y un acoplamiento medio.

Persistencia y base de datos



Decidimos utilizar un esquema desconectado, ya que nos pareció más adecuado al no mantener una conexión constante a la base de datos.

En el archivo `DataAccess/Context.cs`, realizamos varias configuraciones en el método `OnModelCreating()` para adaptar las clases del dominio al modelo de la base de datos.

Establecimos una relación one-to-many entre `Deposit` y `Promotion`, para ello se creó la tabla `DepositPromotion`.

Establecimos que la propiedad `Duration` (de la clase `DateRange`) en `Booking` sea un "owned type". EF Core trata a `Duration` como una entidad separada; pero se almacena en la misma tabla que `Booking`. Lo mismo ocurre con la propiedad `Validity` en `Promotion`.

En `Deposit`, para los períodos de disponibilidad, establecimos la propiedad `AvailabilityPeriods` como un owned type. Para ello tuvimos que definir las listas `AvailablePeriods` y `UnavailablePeriods` (ambas listas de `DateRange`) como owned types de `AvailabilityPeriods`, lo que resultó en la creación de dos tablas: `Deposits_AvailablePeriods` y `Deposits_UnavailablePeriods`.

Anecdóticamente, la versión de EF Core utilizada no soportaba el tipo `DateOnly`, para lo que hicimos un `DateOnlyConverter` en el mismo archivo.

Cobertura de pruebas unitarias

Se alcanzó un 99% de cobertura. Una de las líneas sin cubrir surge en cálculo de precio de depósito, en la clase `PriceCalculator` contamos con un switch en el método `GetPricePerDay`. De forma defensiva, tiramos una excepción si el tamaño del depósito no está entre los 3 disponibles. Sin embargo, nunca llegamos a esta línea ya que los datos se obtienen de un depósito; donde esto ya se encuentra validado. Luego en `DateRange` y `Payment` agregamos un `Id` para que funcione `EntityFramework`, al solo usarse en este no lo testamos. Se adjuntan capturas en el anexo. `DataAccess`, el cual tiene los repositorios, queda con todo los catches sin testear, ya que manejan si se cae la base de datos.

Anexo







Enlaces

Enlace al repositorio:

- https://github.com/IngSoft-DA1-2023-2/283145_285727

Cobertura de pruebas unitarias:

- Captura excluyendo DataAccess y Tests

▼  Total	99%	5/879
>  Domain.Exceptions	100%	0/3
>  BusinessLogic	100%	0/386
>  Calculators	98%	1/42
>  DateRange	97%	2/60
>  Domain	99%	2/388

- Captura sin exclusiones (Sólo UI)

Symbol	Coverage (%)	Uncovered/Tota... ^
▼  Total	82%	458/2523
>  Domain.Exceptions	100%	0/3
>  DateRange.Test	100%	0/48
>  Domain.Test	100%	0/300
>  BusinessLogic	100%	0/386
>  Calculators	98%	1/42
>  DateRange	97%	2/60
>  Domain	99%	2/388
>  BusinessLogic.Test	98%	14/627
>  DataAccess	34%	439/669

Framework de CSS utilizado:

- beercss.com