Universidad ORT Uruguay

Facultad de Ingeniería

Obligatorio 1 Diseño de Aplicaciones

Enzo Izquierdo (283145)

Manuel Graña (285727)

Repositorio

2024

Índice

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

- **Registro y gestión de usuarios:** Permite el registro de un solo administrador y de clientes, incluyendo funcionalidades de inicio y cierre de sesión. El primer usuario es registrado como Administrador. El resto de usuarios registrados se consideran clientes.

- **Administración de depósitos:** Los administradores pueden dar de alta y baja a los depósitos. No se permite dar de baja un depósito para el que existen reservas (sin importar si están rechazadas o aprobadas). No se permite reservar de hoy para hoy y la duración de la reserva es la resta entre los días.

- **Administración de promociones:** Permite al administrador crear, modificar y eliminar promociones aplicables a los depósitos. No se permite eliminar una promoción que se encuentre aplicada a un depósito existente.

- **Reservas de depósitos:** Permite al administrador y los clientes realizar reservas de depósitos que posteriormente deben ser confirmadas por el administrador. Antes de realizar una reserva, se indica el precio del depósito con sus correspondientes promociones.

- **Confirmación de reservas:** El administrador puede aprobar o rechazar reservas según crea conveniente, pudiendo en el caso de rechazarlas, adjuntar un mensaje.

- **Cálculo de precio de depósito:** Se ha implementado un algoritmo para calcular el precio de alquiler de los depósitos basado en su tamaño, duración del alquiler, necesidad de climatización y promociones aplicables.

A su vez, hemos implementado una UI web, a la cual es posible acceder tanto desde dispositivos móviles como en un navegador de escritorio.

Arquitectura del Sistema

Arquitectura en dos capas: La aplicación utiliza una arquitectura de dos capas, con una capa de presentación que se maneja a través de Blazor y una capa de lógica de negocio en C#. Para esta entrega, los datos se guardan en memoria, por lo cual al reiniciar la aplicación se borran los datos ingresados. La comunicación entre la interfaz y la lógica de negocios se hace a través de DTOs (Data Transfer Objects).

Diseño

Organización de la aplicación

La aplicación fue diseñada dividiendo la solución en dos proyectos principales: BusinessLogic y UI. 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.

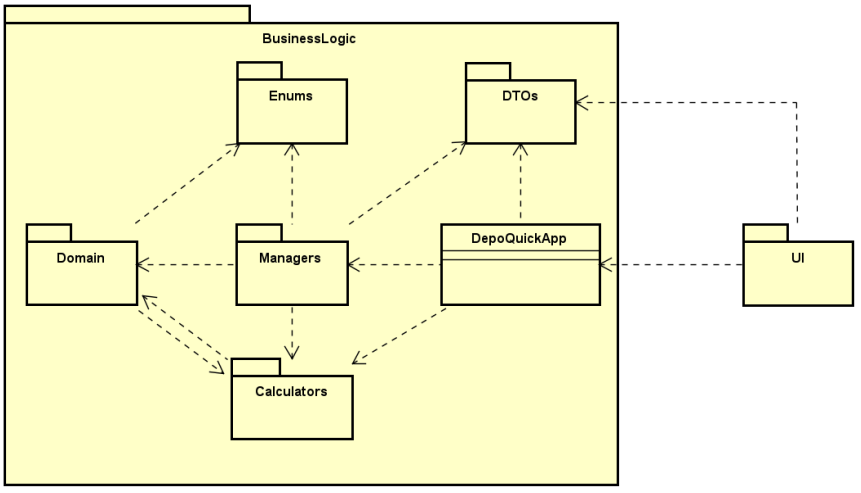
Namespace	Clase	Responsabilidad
BusinessLogic.Domain	User	Gestión y validación de datos de Usuario
BusinessLogic.Domain	Promotion	Gestión y validación de datos de Promoción
BusinessLogic.Domain	Deposit	Gestión y validación de datos de Depósito

BusinessLogic.Domain	Booking	Gestión y validación de datos de Reserva
BusinessLogic.Managers	UserManager	Gestión centralizada de operaciones de datos para usuarios, incluyendo obtener (Login) y establecer (Register) datos de usuario
BusinessLogic.Managers	PromotionManager	Gestión centralizada de operaciones de datos para promociones, incluyendo obtener, establecer, eliminar y modificar datos de promociones
BusinessLogic.Managers	DepositManager	Gestión centralizada de operaciones de datos para depositos, incluyendo obtener y eliminar depositos
BusinessLogic.Managers	BookingManager	Gestión centralizada de operaciones de datos para reserva, establecer, aceptar y rechazar reservas
BusinessLogic.Calculators	IPriceCalculator PriceCalculator	Cálculo de precio de un déposito
BusinessLogic	DepoQuickApp	Orquestación y coordinación de los managers para facilitar la interacción entre la UI y las operaciones del dominio

Además contamos con el namespace “DTOs” que tiene todos los Data Transfer Objects, “Enums” con todos los enums y “Test” con todos los tests.

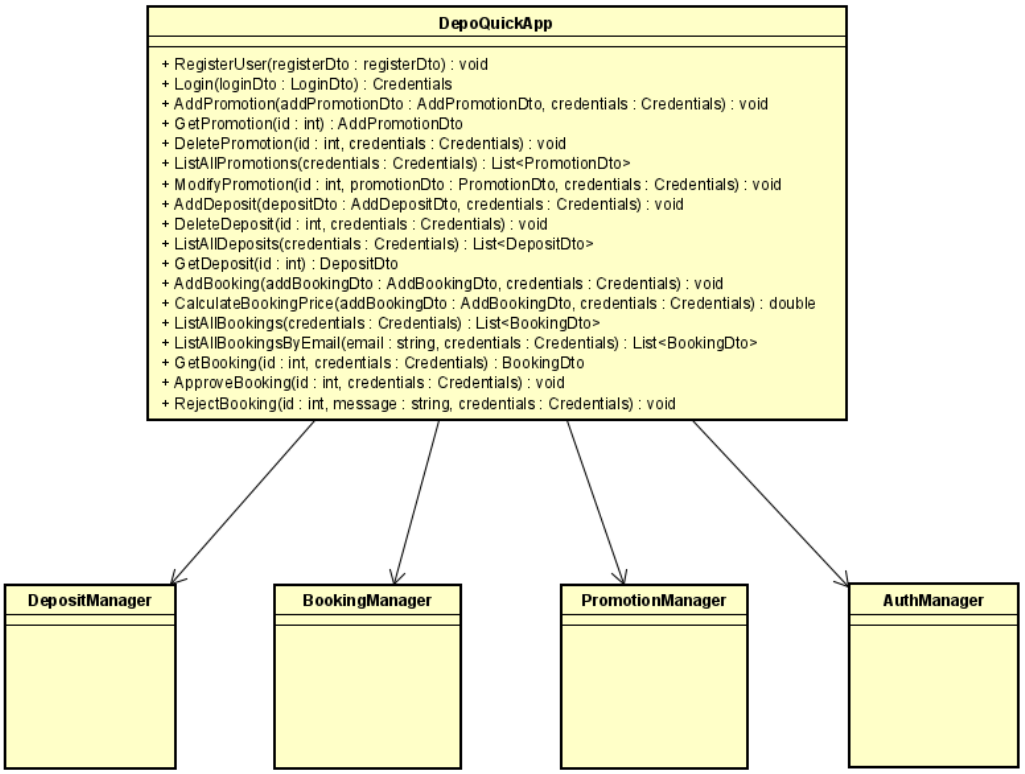
Namespaces

A continuación, el diagrama de paquetes de nuestra aplicación, donde se puede observar los namespaces que mencionamos anteriormente junto con sus dependencias. Consideramos relevante además incluir la clase **DepoQuickApp** para la explicación posterior de cómo se comunica la interfaz con la lógica de negocios.



Managers

El sistema se expone a la UI a través de la clase **DepoQuickApp**. La misma provee una forma simplificada de comunicarse con toda la lógica de negocios de la aplicación. Su función es orquestar/coordinar las acciones de los Managers y las clases del dominio para llevar a cabo las acciones clave del sistema.

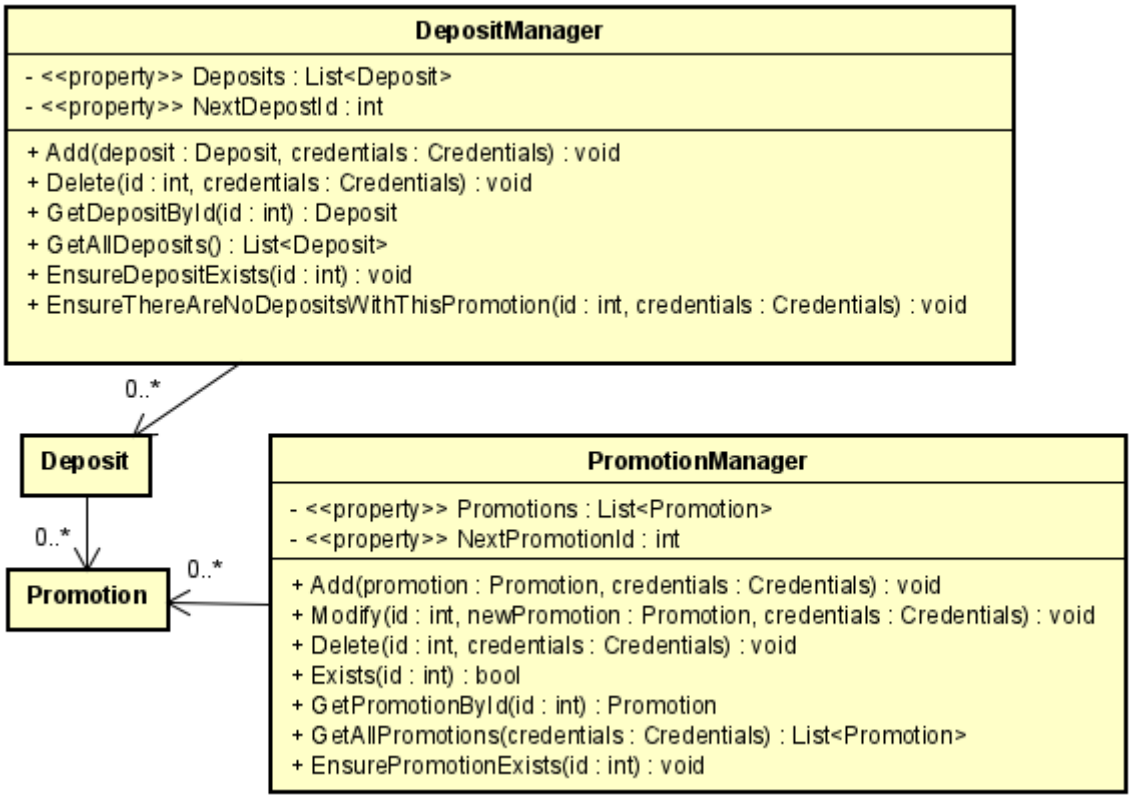


La clase conoce qué Manager es responsable de llevar a cabo cada acción y coordina las llamadas. Por ejemplo, en la llamada del Método Register(), instancia un Usuario con los valores deserializados del DTO y se lo pasa por parámetro a la clase AuthManager que es responsable de registrarlo.

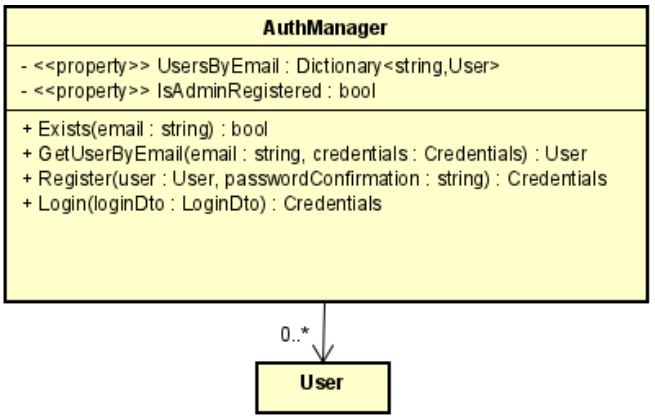
Un ejemplo donde se evidencia la función de coordinación de la clase es eliminar un depósito del sistema. Desde la interfaz se llama a DeleteDeposit(), y DepoQuickApp procede de la siguiente manera:

1. Solicita a BookingManager que se asegure que no haya una reserva para el depósito que se está eliminando.
   (_bookingManager.EnsureThereAreNoBookingsWithThisDeposit())
2. Indica a DepositManager que proceda con la eliminación del depósito.
   (_depositManager.Delete())

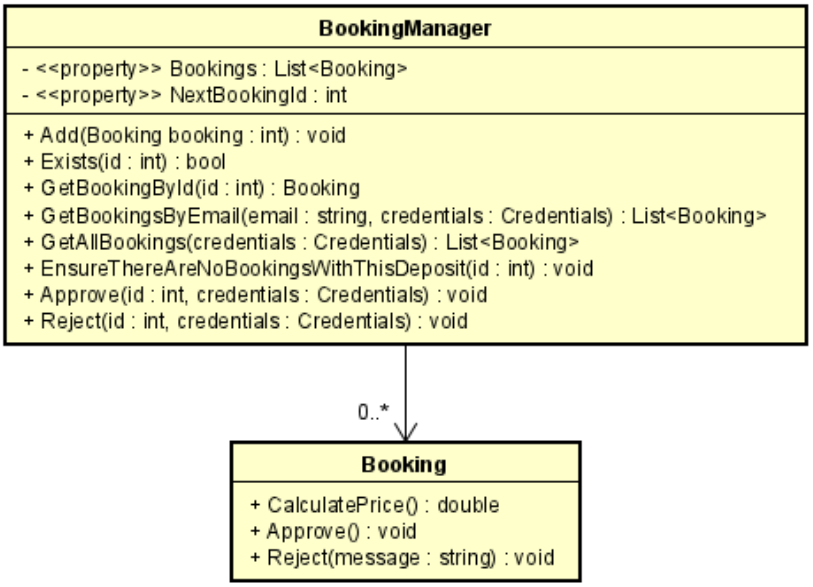
En nuestro proyecto, nos referimos con Manager a una clase que se encarga de gestionar un elemento del dominio. Contienen una lista con elementos del dominio y se encargan de realizar operaciones sobre la misma (agregar, eliminar, modificar, obtener,...), además de otras operaciones más relacionadas con la lógica de negocio (asegurarse que un depósito exista en DepositManager, asegurarse que exista una reserva con un depósito dado en BookingManager, asegurarse que una promoción exista en PromotionManager).



AuthManager sigue la misma lógica: tiene operaciones de lista (GetUserByEmail, Exists) y además tiene operaciones de lógica de negocios (Register, Login) que operan sobre la lista y retornan credenciales.

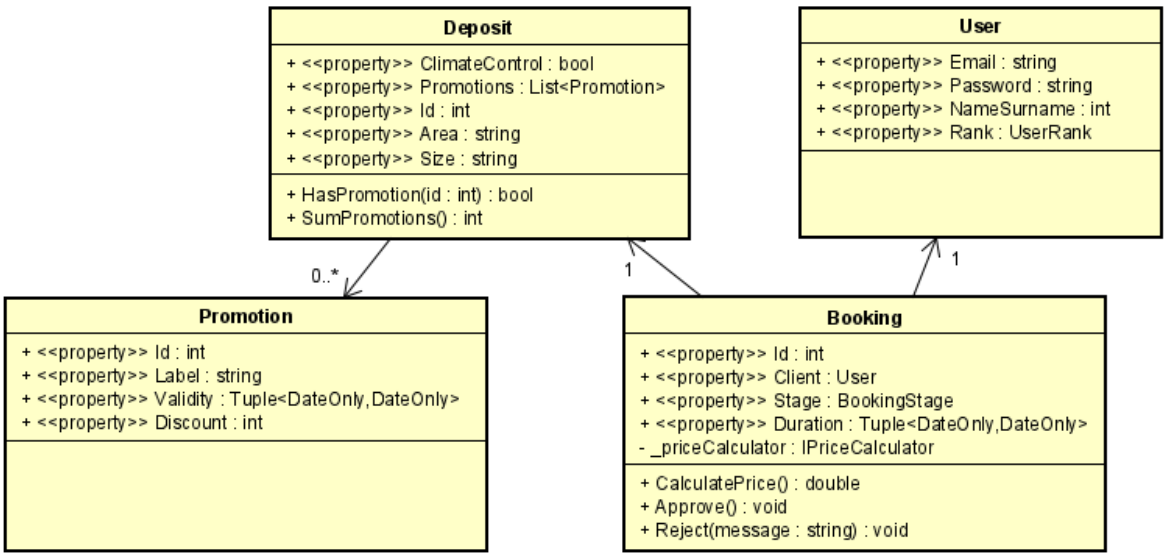


En BookingManager además se encuentran las operaciones de aprobar y rechazar reservas, que se delegan a la clase Booking. BookingManager se encarga en estos casos de asegurarse que el usuario sea administrador, buscar la reserva por ID y llamar a los métodos Approve y Reject respectivamente de la clase booking.



Dominio

En el ámbito del dominio, hemos definido cada clase relevante según los requerimientos, incluyendo sus respectivas validaciones (métodos privados que optamos por no incluir en el diagrama).



Sin métodos privados

En este caso específico, las clases se relacionan de la siguiente manera: Booking tiene un Usuario y un Depósito, y este último agrupa Promociones.

La principal ventaja de esta estructura radica en que las entidades del dominio no están acopladas a la implementación de las clases superiores. Por ejemplo, un Booking no se encarga de validar si su usuario existe (es decir, está registrado en el sistema) antes de ser creado; esa validación es responsabilidad del BookingManager, que lo añade a la lista. De esta manera, Booking se centra exclusivamente en sus propiedades, lo que mejora la modularidad y la claridad del diseño.

En Depósito hemos incluído los métodos HasPromotion() y SumPromotions(). La motivación de esto es para evitar que las clases que acceden a depósito, como DepositManager, tengan que acceder a la lista de promociones y terminen violando la ley de Demeter; “hablando con extraños” (con Promoción). En Booking, incluímos los métodos Approve y Reject ya que solo cambian el estado de la reserva (encapsulación), particularmente los atributos Stage y Message. También hay un método CalculatePrice, pero los detalles del cálculo de precio son delegados a otra clase a través de la interfaz IPriceCalculator, por lo que Booking no asume demasiada responsabilidad.

Descripción de Arquitectura y Decisiones de Diseño Clave

Interacción entre UI y Business Logic

La interacción entre la UI y Business Logic se realiza mediante Objetos de Transferencia de Datos (DTOs). La UI comunica sus requerimientos a través de la clase DepoQuickApp, cuya función fue explicada anteriormente.

Usar DTOs nos permite garantizar que la UI no necesite ni deba conocer detalles de la implementación del sistema. Estos DTOs se crean en la UI y son enviados a DepoQuickApp, 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 software.

Almacenamiento de Datos

En esta instancia hemos optado por almacenar los datos en memoria, ya que no disponemos de una base de datos, por lo tanto no hay persistencia de datos al volver a abrir la página.

La clase DepoQuickApp actúa como el núcleo de nuestro sistema y contiene cuatro administradores (managers) que son responsables del manejo de las diferentes categorías de datos.

Cada uno de estos managers mantiene en memoria listas de objetos del dominio correspondientes (User, Promotion, Deposit, y Booking), permitiendo un acceso rápido y eficiente a los datos durante la ejecución del sistema. Esta estructura no solo simplifica el desarrollo, sino que también facilita la transición a una solución de almacenamiento persistente en futuras iteraciones del proyecto.

Manejo de Errores y Excepciones

En nuestro sistema, utilizamos solamente excepciones estándar de C#.

Las excepciones son lanzadas por las clases del dominio y los managers cuando ocurren errores en las operaciones. Estas excepciones son capturadas en la UI, donde se manejan adecuadamente, no permitiendo a la operacion continuar y mostrando su mensaje asociado.

Este enfoque nos permite mantener una separación clara entre la lógica del dominio y la UI, facilitando un sistema más robusto y mantenible.

Utilización de Polimorfismo

En nuestro diseño, el polimorfismo se usó para mejorar la flexibilidad y la capacidad de extensión del sistema. Este concepto se ilustra de manera destacada en la funcionalidad de "Cálculo de precio de depósito". Aquí, empleamos la interfaz IPriceCalculator, la cual define un contrato común para calcular precios de reservaciones. Al método CalculatePrice en Booking, le pasamos un objeto que implementa esta interfaz, junto con un depósito. Este objeto determina cómo se calcula el precio de la reservación, lo que nos brinda la flexibilidad de modificar este proceso en el futuro si es necesario.

Criterio de asignación de responsabilidad

En la asignación de responsabilidades, seguimos el principio de responsabilidad única, donde las clases del dominio se encargan solo de la validación y asignación de datos, los managers gestionan las operaciones sobre las listas de elementos y DepoQuickApp interactúa exclusivamente con los managers para obtener datos solicitados por la interfaz de usuario, manteniendo así una clara separación de preocupaciones y facilitando la modularidad y mantenimiento del código.

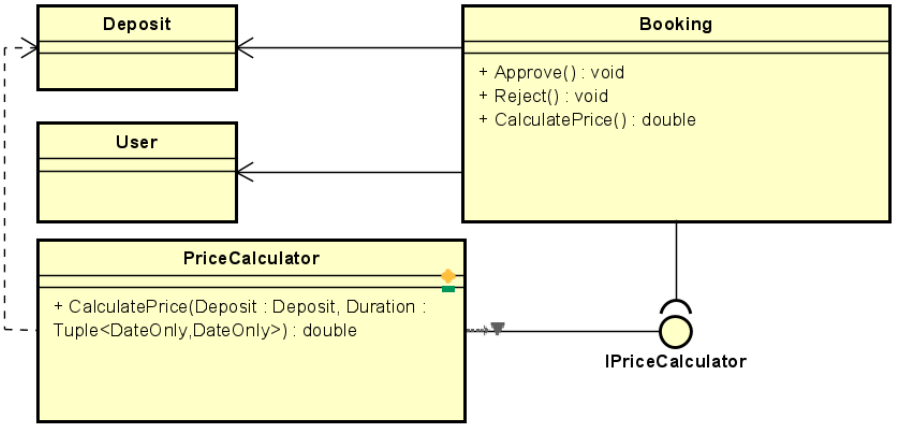
Mantenibilidad y Extensibilidad

Supongamos que en el futuro necesitamos implementar nuevas formas de calcular los precios de los depósitos por que los precios se calculan de forma distinta según su area. La funcionalidad puede ser implementada fácilmente ya que deberíamos solamente hacer uso de la interfaz IPriceCalculator e implementar la clase AreaPriceCalculator. En la clase Booking, simplemente le pasaríamos al constructor esta nueva implementación de IPriceCalculator;

para que finalmente cuando se llame al método CalculatePrice se use la implementación correcta de IPriceCalculator.

Análisis de dependencias en “Cálculo de precio de depósito”

Para la funcionalidad de Cálculo de precio de depósito, interactúan de forma directa las clases Booking, Deposit y una clase que implemente IPriceCalculator, en nuestro análisis consideraremos PriceCalculator.



Cohesión

- **Booking:** Alta cohesión. Para calcular el precio, usa sus propios atributos conocidos: el **depósito** y su **duración**. Delega la implementación del cálculo a una clase que implemente IPriceCalculator.
- **PriceCalculator:** Baja cohesión. La clase no usa sus propios atributos para calcular, sino que depende de parámetros externos.
- **Deposit:** Cohesión moderada. En el cálculo de precio de depósito, asume la responsabilidad de sumar sus promciones.

Acoplamiento

- **Booking:** Bajo acoplamiento. Booking no usa métodos ni atributos de la clase Deposit, solo necesita conocerlo. Si bien llama a un método de PriceCalculator, el acoplamiento es levemente más bajo gracias a que depende de la interfaz 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.

- **Deposit:** Acoplamiento moderado. Su responsabilidad en este escenario es sumar las promociones, para lo que necesita conocer el descuento asociado a Promotion.

Luego de realizado el análisis, concluímos que se puede reducir el acoplamiento trasladando la lógica de cálculo de precios a la clase Deposit. Esto implicaría, sin embargo, que Deposit asuma una responsabilidad adicional. Además reduciría la extensibilidad del sistema al limitar las formas de calcular el precio comparado al uso de una interfaz particular.

Cobertura de pruebas unitarias

Se alcanzó un 99% de cobertura, con 3 “statements” sin cubrir. Uno de ellos 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 deposito 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. Los otros dos statements no cubiertos están en BookingManager (en EnsureNoOverlappingDates) y DepositManager (en NextDepositId, no cubrimos crear más de un depósito).

Evidencia de TDD: “Cálculo de precio de deposito”

Inicio y Primer Ciclo [Red-Green]

Red (1): Iniciamos creando la primera prueba *TestCanCalculateBasePrice*. Para escribirla, nos imaginamos la interfaz que queremos para nuestra operación, enfocándonos ahora solo en el cálculo del precio base del depósito basado en tamaño y climatización. Para que compile, creamos la clase vacía PriceCalculator con el método CalculatePrice. El test falla.

Green (2): Desarrollamos la implementación mínima necesaria para pasar la prueba, utilizando DataRow para validar varias combinaciones simultáneamente.

Segundo Ciclo: Añadiendo Descuento por Duración [Red-Green]

Red (3): Agregamos un test para el descuento por duración de la reserva, el cual falló inicialmente.

Green (4): Actualizamos la implementación para incluir el cálculo de descuentos por duración, asegurando que los nuevos tests pasaran con la mínima modificación necesaria.

Tercer, Cuarto y Quinto Ciclo: Integración de Promociones [Red-Green]

Luego, agregamos un test para validar el cálculo con promociones, mas especificamente, una sola promoción cuyo descuento sea menor al 100%.

[red] (5) → [green] (6)

Luego probamos para varias promociones (tambien cuya suma sea menor a 100%).

[red] (7)→ [green] (8)

Luego agregamos la validación de que si hay varias promociones, se sumen con un tope del 100%

[red] (9) → [green] (10)

Refactorización y Mejoras

Con la funcionalidad principal funcionando correctamente, procedimos a refactorizar el código para mejorar su legibilidad. Extrajimos constantes, refactorizamos métodos y renombramos variables para clarificar su propósito.

Integración en la Clase de Reservas

Red (11): Desarrollamos un test para asegurarnos que, al integrar el calculador de precios en el proceso de reserva, se calculara correctamente el precio final.

Green (12): Verificamos que la integración con la clase de reserva funcionara adecuadamente, pasando todos los tests.

Preparación para Futuras Extensiones y Refactor final

Refactor (13): Anticipando futuras necesidades de modificación en el método de cálculo de precios, implementamos la interfaz IPriceCalculator. Adaptamos nuestro calculador para que

implementara esta interfaz, permitiendo que la función de cálculo de precios aceptara cualquier objeto que la implemente.

Refactor (14): Notamos la violación de la ley de Demeter al estar llevando a cabo la suma de promociones en la clase CalculatePrice, ya que la misma solo conoce a Deposit y no debería comunicarse directamente con Promotion. Por lo tanto, en este último refactor, movimos la lógica de sumar promociones a la propia clase Deposit.

Casos de prueba: "Administración de Depósitos y Alta de depósito"

Prueba	Objetivo	Clases objetivo	Datos de prueba	Criterio de aceptación
Creación con datos válidos	Verificar si puedo crear un deposito con datos válidos	Deposit	- Area: A - Size: Small - Climate Control: true - Lista de promociones	El objeto depósito es creado (no es null)
Tamaño de Depósito Inválido	Asegurar que el tamaño del depósito es válido comparando el valor ingresado (string) con los valores del enum Size	Deposit	Deposito con Size: Extra Large	El string proporcionado debe coincidir con uno de los valores predefinidos en el enum Size. Si no coincide, se debe lanzar una excepción.
Área de Depósito Inválida	Asegurar que el área del depósito es válida comparando el valor ingresado (string) con los valores del enum Size	Deposit	Area: Z	El string proporcionado debe coincidir con uno de los valores predefinidos en el enum Area. Si no coincide, se debe lanzar una excepción.
Añadir Depósito con datos válidos	Puedo añadir un deposito a la lista correspondiente	Deposit Manager	- Area: A - Size: Small - Climate Control: true - Lista de promociones	La lista de depositos contiene un elemento (el deposito añadido)
Borrar depósito existente	Puedo borrar un deposito de la lista dado su id.	Deposit Manager	Id de un depósito existente Id: 1	La lista de depósitos esta vacia luego de ejecutada la prueba
Borrado de Deposito inexistente	No puedo borrar un deposito de la lista dado su id si este no pertenece a la lista.	Deposit Manager	Id de un depósito inexistente	Se lanza una excepción al intentar borrar el depósito con id 1.
Autorización para Añadir Depósitos	Comprobar que sólo los administradores tienen permiso para añadir depósitos nuevos	Deposit Manager	User que no es admin	Si el usuario es administrador, se crea y añade a la lista. Si no lo es, tirará una excepción
Autorización para borrar Depósitos	Comprobar que sólo los administradores tienen permiso para borrar depósitos	Deposit Manager	User que no es admin	Si el usuario es administrador, se borra de la lista. Si no lo es, tirará una excepción
Obtención de la lista de Depósitos	Comprobar que se retorna la lista correctamente	Deposit Manager	User 1 deposito	Se debera devolver una lista con el deposito agregado

No realizamos validación con ClimateControl ya que es un valor booleano y ambos valores son válidos y tampoco validamos Campos vacíos ya que la firma de la función previene esta situación.

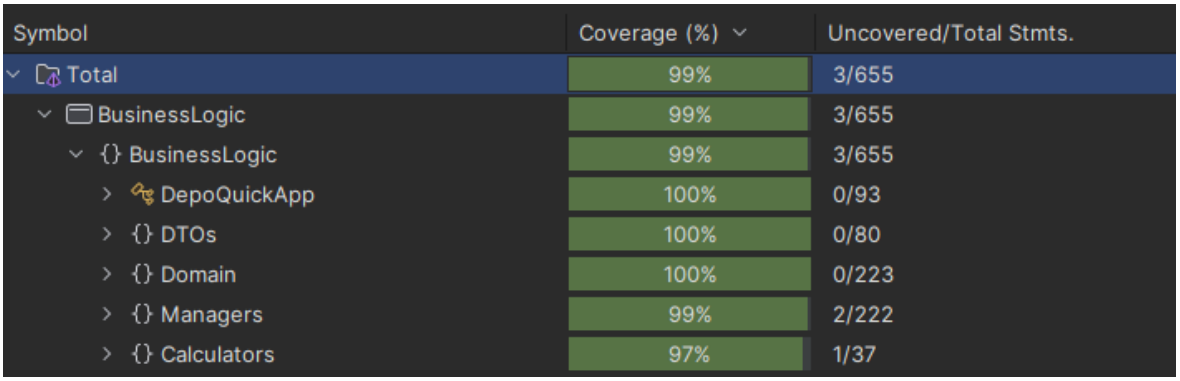
Anexo

Enlaces

Enlace al repositorio:

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

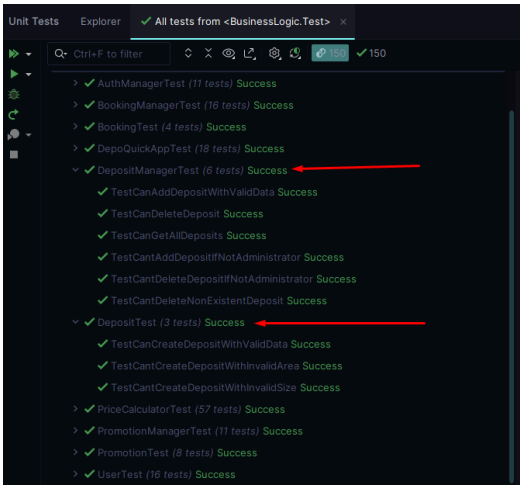
Cobertura de pruebas unitarias:

- 

Framework de CSS utilizado:

- beercss.com

Pruebas de Administracion de deposito y alta de deposito:



Enlaces a commits de Cálculo de precio de deposito:

1. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/673c00e325c58b7dddd3c00a528dab69c8e3d6ee#diff-1a8106ac790771e6f2dbfd559445210c13d5e07b56add5439fb0c563d67611ec
2. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/5b35586adc0efd88f24987a68dd960b7839a1b70
3. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/3c9dea5ce67b40a80156541f85b926fcd09397d5
4. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/ffd64a9899affc206080b8e628ba30e5f6a5309d
5. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/bde8b654be0f6fbe0de302ecba13dafba00ffce5
6. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/6e42d574cbc76b10393f4dcfa530bd7e6c954c22
7. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/63e80ee9e11b47f02bc24fbed0825d5effa0db21
8. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/fcca31e48a3e16a1d51b5907da9d761abbbd0b5d
9. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/29305b83d33ce5d1862b78126de36aa4468d54e8
10. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/05c38dc7f4452b732b0dfeec82dfb5ae567ae550
11. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/c3665de6ee0ef5238463671743aaff9f7bb3eaf4
12. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/0541ad843c649481766cfe3b974ec24a94db79b9
13. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/2aed348a436eae9bf311a4d3622b713b33ca487b
14. https://github.com/IngSoft-DA1-2023-2/283145_285727/commit/2f0fc93f0d66fd5895a8f85ae9208d552201d896