



Obligatorio 1

Diseño de aplicaciones 1

Mathias Huque – 284934

Docentes

Facundo Arancet

Martin Radovitzky

Gastón Mousques

2024

[Repositorio GitHub](#)

Índice

Descripción general.....	3
Gitflow.....	3
Test Driven Development (TDD).....	3
Descripción y justificación de diseño.....	4
Diagramas de paquetes.....	4
Diagramas de clases.....	6
Diagrama de clases de paquete Dominio.....	6
Diagrama de clases de paquete Excepcion.....	7
Diagrama de clases de paquete Repositorio.....	8
Diagrama de clases de paquete BusinessLogic.....	9
Implementación de la interfaz.....	11
Detalle de cobertura.....	12
Cobertura de Excepcion:.....	12
Cobertura del Repositorio.....	13
Cobertura del BusinessLogic.....	13
Cobertura del Dominio.....	13
Cálculo de precio de depósito.....	14
Escribir una prueba (Fase "Red").....	14
Escribir el código (Fase "Green").....	14
Refactorizar el código (Fase "Refactor").....	14
Evidencia de pruebas funcionales.....	15

Descripción general

El objetivo de este proyecto fue desarrollar una aplicación llamada "DepoQuick" para simplificar el proceso de alquiler de depósitos, haciéndolo tan sencillo como realizar un pedido en línea. La aplicación se diseñó y construyó cumpliendo con todos los requisitos establecidos en el enunciado del obligatorio.

Gitflow

Durante todas las fases del proyecto, se ha seguido la metodología GitFlow. Esta técnica permite el desarrollo simultáneo e independiente de diferentes funcionalidades, facilitando la creación de características por separado que luego se integran en el proyecto principal una vez completadas.

El proceso se inició en la rama principal, desde donde se creó una rama "develop" para agregar nuevas funcionalidades y mejoras. Posteriormente, se mantuvo una rama principal llamada "main" para conservar una versión estable de la aplicación. Solo al finalizar un ciclo con todos los cambios se realizaron fusiones de cambios en esta rama.

Test Driven Development (TDD)

El Desarrollo Guiado por Pruebas (TDD, por sus siglas en inglés) fue implementado de forma integral en todo el proyecto, con excepción de la capa de interfaz de usuario. Este enfoque metodológico consiste en escribir pruebas automatizadas antes de desarrollar el código real, siguiendo un proceso dividido en tres fases distintas:

1. Escribir una prueba (Fase "Red"): Se crearon pruebas para verificar el comportamiento esperado de una característica o componente específico.
2. Escribir el código (Fase "Green"): Una vez definidas las pruebas, se procedió a escribir el código necesario para que estas sean superadas y cumplan con los requisitos establecidos.
3. Refactorizar el código (Fase "Refactor"): Finalmente, se realizó una revisión y optimización del código escrito, garantizando su calidad, eficiencia y legibilidad.

Esta metodología permitió un desarrollo incremental de la aplicación, asegurando su correcto funcionamiento mediante pruebas exhaustivas en cada etapa del proceso de desarrollo.

Descripción y justificación de diseño

Diagramas de paquetes

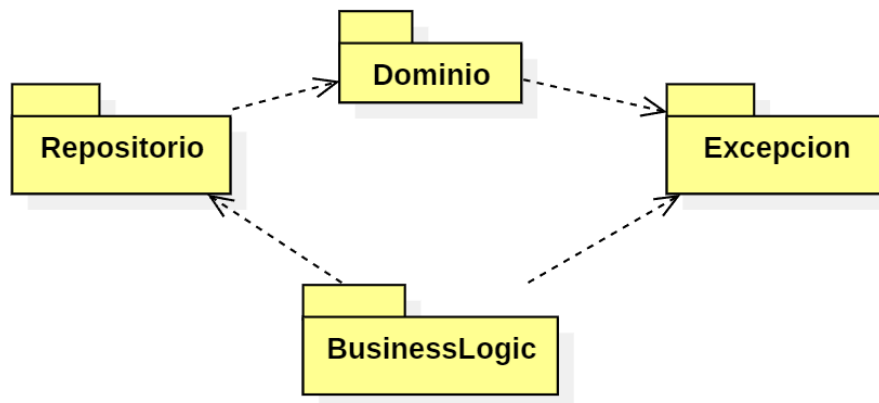


Imagen Diagrama 1.0

En el diagrama 1.0 se observa la estructura básica de la aplicación, organizada en varios paquetes que representan distintas capas de la arquitectura.

El paquete "Dominio" contiene las clases de modelo o entidades que representan los datos manipulados por la aplicación. Estas entidades, que reflejan objetos del mundo real, se convertirán en tablas de una base de datos en futuras implementaciones.

Siguiendo el patrón de diseño Repository, se ha creado un paquete del mismo nombre que contiene la interfaz IRepository. Esta interfaz establece un contrato para la lógica de acceso a la base de datos. Las clases DepositoRepository, PromocionRepository, ReservaRepository y UsuarioRepository implementarán esta interfaz y proporcionarán la lógica específica para interactuar con la base de datos de cada entidad.

El paquete "BusinessLogic" representa el núcleo funcional de la aplicación. Aquí se definen y ejecutan las reglas de negocio, como validaciones, gestión de usuarios, obtención de depósitos, y operaciones de alta, baja y modificación de promociones, entre otras.

El paquete "Excepcion" se encarga de manejar todas las excepciones del sistema. Proporciona una forma estructurada de identificar y gestionar los errores que puedan surgir durante la ejecución de la aplicación, permitiendo un mejor control y manejo de situaciones inesperadas.

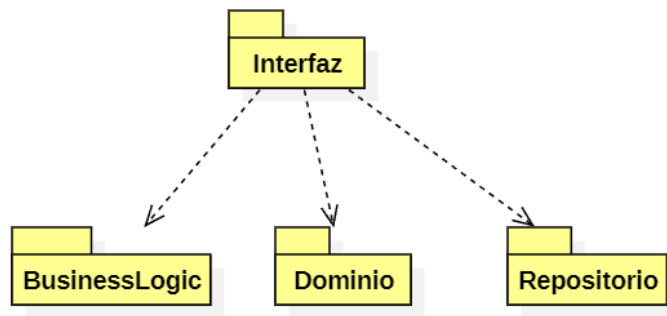


Imagen Diagrama 1.1

En el Diagrama 1.1 se presenta la interacción de dependencias del paquete "Interfaz" con los demás paquetes del sistema.

En el paquete "Interfaz" se ubica el servidor de Blazor Server, junto con el conjunto de páginas que conforman el frontend de la aplicación.

Diagramas de clases

Diagrama de clases de paquete Dominio

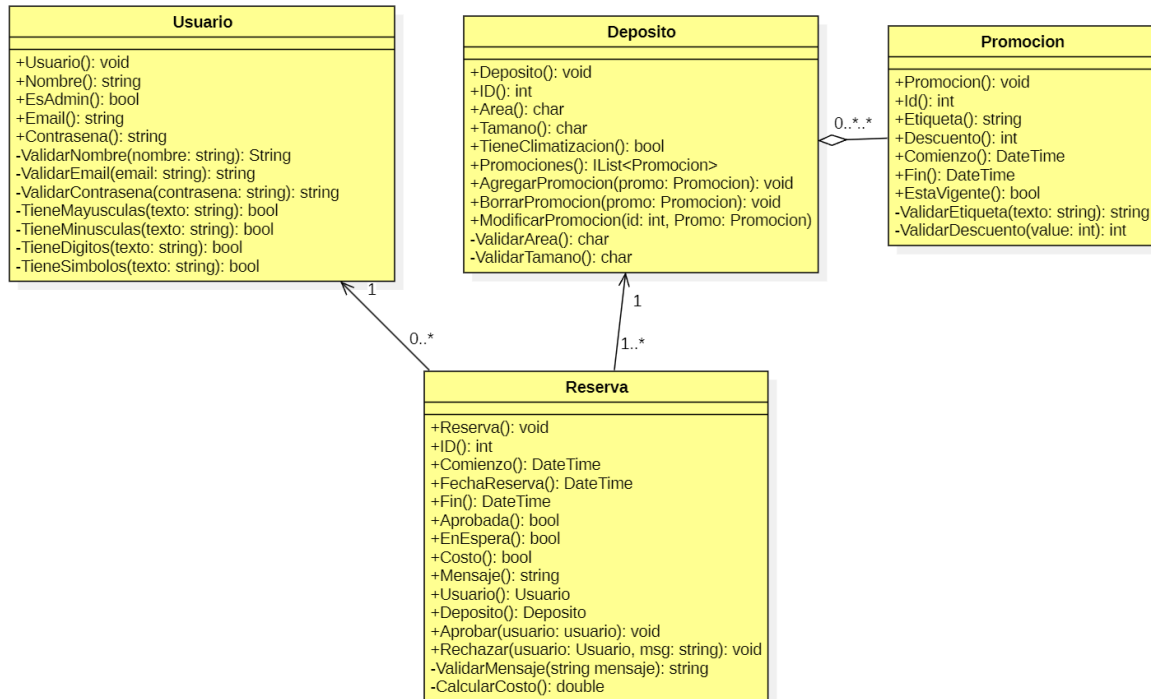


Imagen Diagrama 2.0

El diagrama UML en la imagen 2.0 muestra el modelado del paquete Dominio, que describe las clases principales de nuestro sistema.

La clase Usuario se encarga de validar los datos del usuario, incluyendo su contraseña y correo electrónico, para asegurar que tengan el formato correcto. También proporciona una función para determinar si el usuario es administrador o un usuario común, En lugar de utilizar herencia o polimorfismo para distinguir entre usuarios y administradores, esta función cumple con el objetivo de diferenciar los roles dentro del sistema.

La clase Promocion almacena información sobre las promociones, como su identificador, etiqueta, descuento, fecha de inicio y fecha de fin. Además, contiene métodos para verificar si la promoción está vigente y para validar que la etiqueta y el descuento cumplan con los requisitos establecidos.

La clase Depósito guarda los datos relacionados con un depósito, incluyendo validaciones para garantizar la integridad de los datos. También mantiene una lista de promociones asociadas al depósito, y proporciona métodos para agregar y eliminar promociones.

La clase Reserva encapsula la información relacionada con la reserva de un depósito, como el cliente, el depósito reservado, las fechas de inicio y fin de la reserva, y el estado de la reserva (aprobada, en espera de aprobación, rechazada). Incluye métodos para aprobar y

rechazar la reserva, asegurando que sólo un administrador pueda realizar estas acciones. También calcula el costo de la reserva en función de los detalles del depósito (tamaño, climatización, promociones vigentes) y la duración de la reserva.

Esta estructura de clases proporciona una base sólida para el manejo de usuarios, promociones, depósitos y reservas dentro del sistema, manteniendo un diseño claro y fácil de mantener.

Diagrama de clases de paquete Excepcion

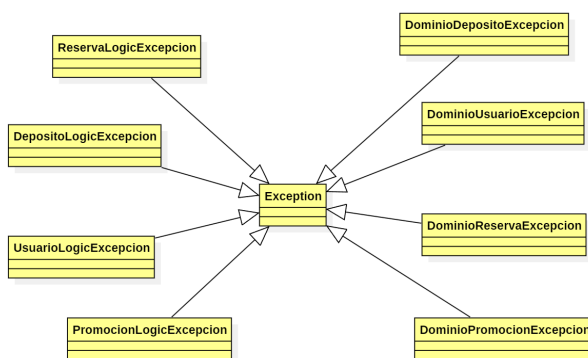


Imagen Diagrama 2.1

En el diagrama 2.1 se han definido ocho excepciones que heredan de la clase `Exception`. Estas excepciones tienen como propósito capturar situaciones en las que no se cumple una regla de negocio en diferentes áreas del sistema.

Las excepciones `ReservaLogicExcepcion`, `UsuarioLogicExcepcion`, `DepositoLogicExcepcion` y `PromocionLogicExcepcion` están diseñadas para ser lanzadas desde la capa de lógica de negocio. Su función es proporcionar información detallada sobre el error y ayudar a identificar dónde se encuentra el problema en caso de una excepción.

Por otro lado, las excepciones `DominioDepositoExcepcion`, `DominioUsuarioExcepcion`, `DominioReservaExcepcion` y `DominioPromocionExcepcion` están destinadas a ser lanzadas a nivel de dominio. Se utilizan principalmente cuando se intenta crear instancias de elementos con un formato incorrecto, como ingresar una contraseña que no cumple con los requisitos (símbolos, números, mayúsculas, etc.) al crear un usuario.

En todas estas excepciones se hace uso de los métodos `Message()` y el constructor del padre `Exception` para proporcionar información útil sobre el error que ocurrió. Esto ayuda a los desarrolladores a identificar y resolver rápidamente los problemas que puedan surgir en el sistema. Por lo que ayuda a la mantenibilidad del sistema.

Diagrama de clases de paquete Repositorio

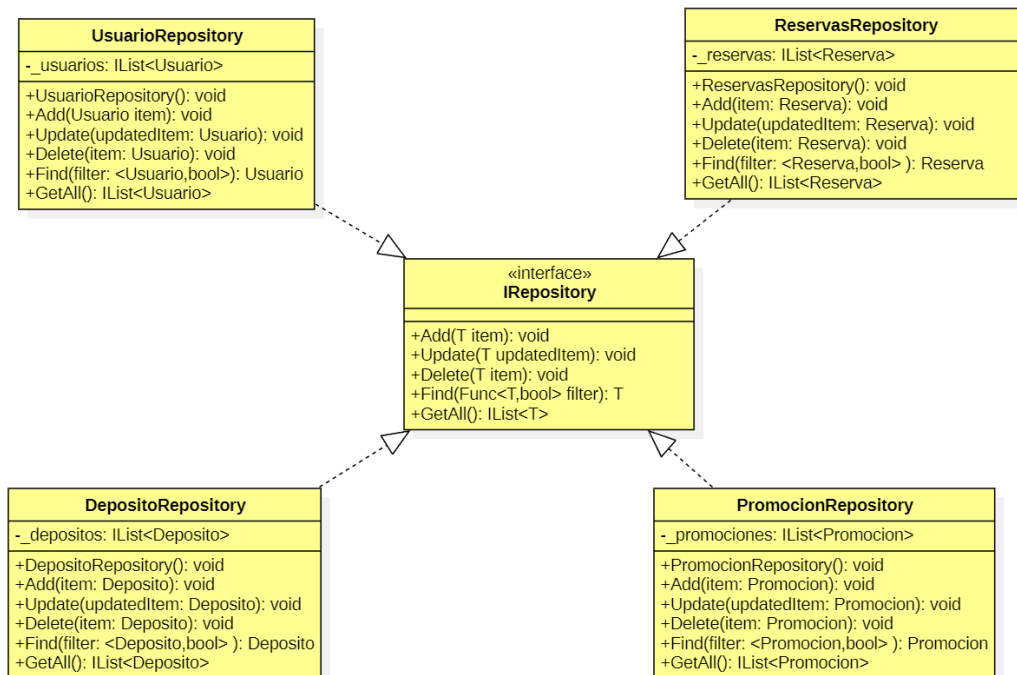


Imagen Diagrama 2.2

Siguiendo el patrón de diseño Repository del diagrama 2.2, se ha establecido una interfaz que define cinco operaciones estándar para el manejo de datos (CRUD) en la futura base de datos. Esta interfaz proporciona un contrato común que será implementado por cada uno de los repositorios con el fin de garantizar la consistencia en la manipulación de datos.

Los repositorios definidos, como **UsuarioRepository**, **DepositoRepository**, **PromocionRepository** y **ReservasRepository**, actúan como capas intermedias entre la lógica de la aplicación y los datos almacenados en memoria. Cada repositorio utiliza una lista en memoria para almacenar los objetos correspondientes y ejecuta las operaciones CRUD sobre esta estructura de datos.

La ventaja de utilizar este enfoque es que se establece una clara separación entre la lógica de negocio y la lógica de almacenamiento, lo que facilita la gestión y mantenimiento del código. Además, al emplear una interfaz para definir las operaciones comunes, se facilita la extensión del programa para incluir nuevas funcionalidades, como la modificación de un usuario en este caso, sin necesidad de modificar el código existente de manera significativa.

En resumen, el patrón Repository junto con el uso de interfaces proporciona una arquitectura flexible y escalable, que permite adaptar el sistema a los cambios y requerimientos futuros de manera eficiente y sin afectar la estabilidad del código existente.

Diagrama de clases de paquete BusinessLogic

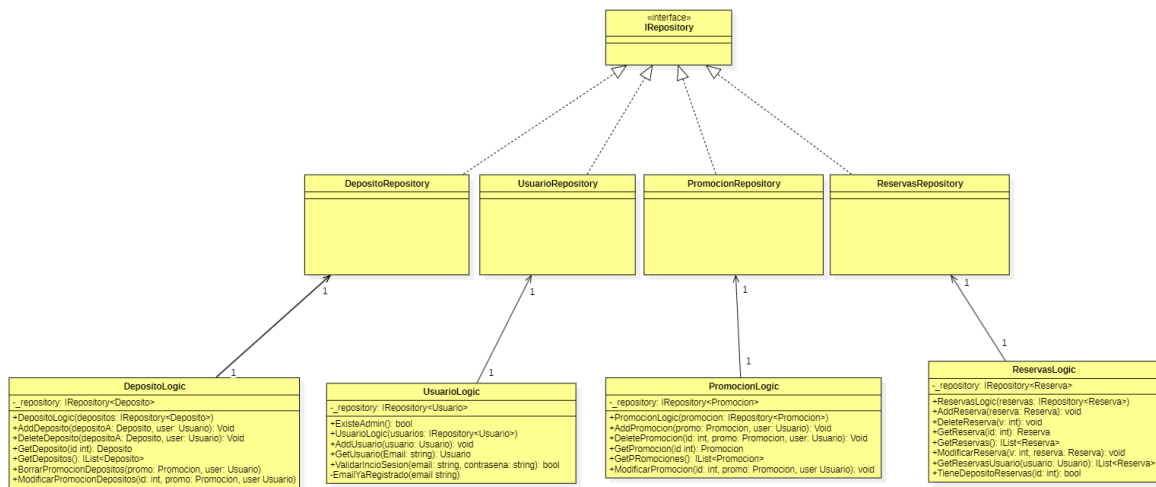


Imagen Diagrama 2.3

En el namespace BusinessLogic se ha implementado una estructura de clases separadas para cada elemento del dominio, con el propósito de servir como intermediario entre la interfaz de usuario y la capa de acceso a datos.

La clase DepositoLogic contiene una instancia de DepositoRepository y métodos para gestionar altas, bajas y selección de depósitos. Para estas operaciones se verifica que el usuario sea Administrador. También tiene para modificación y baja de promociones de depósitos, supongamos que desde la interfaz se modifica una promoción, los depósitos que tienen dicha promoción también deberán modificar la promoción que tienen guardado, lo mismo para baja de promociones.

UsuarioLogic contiene una instancia de UsuarioRepository, con métodos para verificar la existencia de un administrador en memoria, gestionar altas y selección de usuarios, así como validaciones para el inicio de sesión y la verificación de la no duplicidad de correos electrónicos en el registro.

En cuanto a PromocionLogic, esta clase contiene una instancia de PromocionRepository y métodos para gestionar altas, bajas, selección y modificaciones de promociones, con validación de permisos de administrador.

La clase ReservasLogic guarda una instancia de ReservasRepository y ofrece métodos para gestionar altas, bajas, selección y modificaciones de reservas. También proporciona una función para buscar reservas asociadas a un depósito específico.

Esta arquitectura facilita la organización y mantenimiento del código, manteniendo una clara separación de responsabilidades entre las distintas capas del sistema.

Diagrama de clases de Interfaz

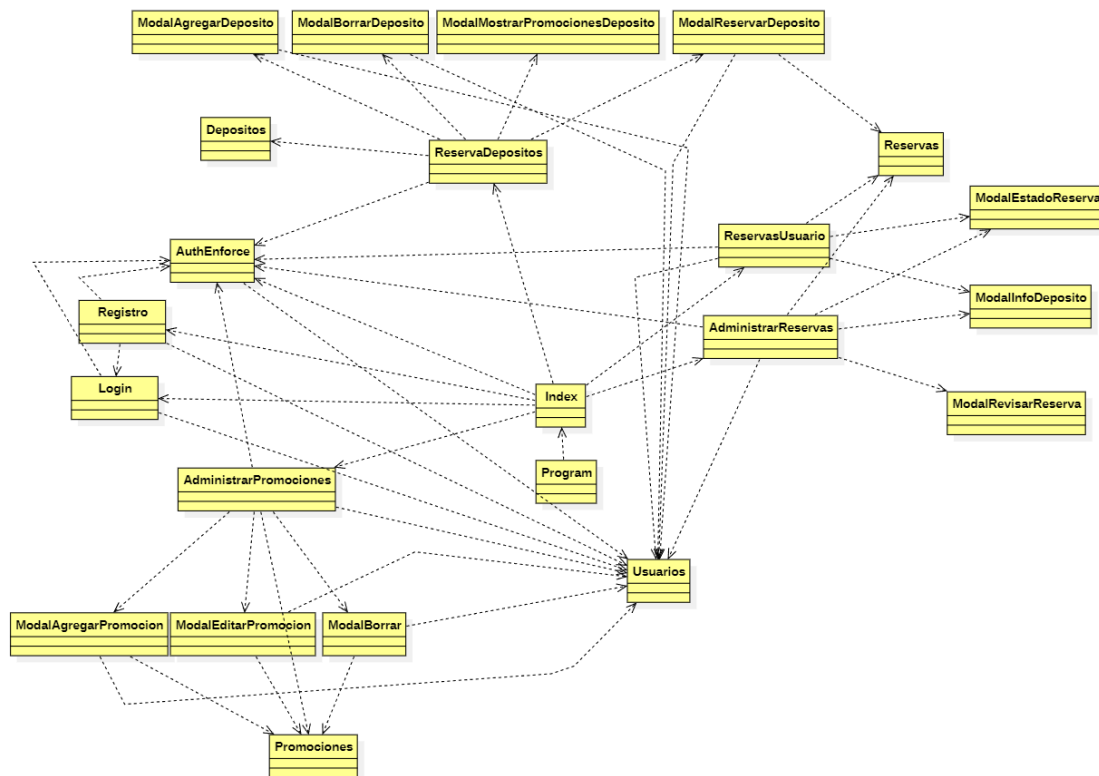


Imagen Diagrama 2.4

El diagrama muestra la interacción entre las clases Program, Depositos, Promociones, Reservas y Usuarios con las páginas de la interfaz.

La clase Program inicia la página Index, y si el usuario no está registrado, se redirige a la página de Registro a través del authEnforce, el cual garantiza que el usuario haya iniciado sesión antes de acceder a la página. Una vez autenticado, se permite el acceso a la página Index y se guarda la información del usuario en la clase Usuario. Una vez iniciada la sesión, se puede acceder a las demás páginas.

Las clases Depositos, Promociones y Reservas son utilizadas por las diferentes páginas para acceder a las funciones de obtener, borrar, agregar y modificar los objetos relacionados con depósitos, promociones y reservas. Estas clases actúan como intermediarios entre las páginas de la interfaz y la lógica del negocio, facilitando la interacción entre ambos componentes.

En resumen, estas clases juegan un papel crucial en la conexión entre la interfaz de usuario y la lógica del negocio, permitiendo que las páginas accedan y manipulen los datos de manera eficiente y segura.

Implementación de la interfaz

En la implementación de todas las capas de nuestra aplicación en una interfaz web utilizando Blazor Server, se configuraron los siguientes servicios Singleton en la clase Program.cs:

```
// Add services to the container.

builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddSingleton<Promociones>();
builder.Services.AddSingleton<Depositos>();
builder.Services.AddSingleton<Usuarios>();
builder.Services.AddSingleton<Reservas>();
builder.Services.AddBlazoredModal();
builder.Services.AddAuthenticationCore();
```






Estas llamadas a `Services.AddSingleton()` registran tipos concretos en el contenedor de inyección de dependencias, lo que permite que estén disponibles en toda la aplicación. Se crea una única instancia del servicio durante todo el tiempo de ejecución en memoria, dado que no hay persistencia de datos en una base de datos en este caso.

Se inyectaron en las diferentes páginas los servicios mencionados, como `NavigationManager` para poder utilizar el redireccionamiento de páginas en los eventos, y las instancias por tiempo de ejecución de las clases `Usuarios`, `Depositos`, `Promociones` y `Reservas`. Estas clases contienen instancias de sus respectivas capas lógicas y de datos, y las funciones que poseen son simplemente llamadas a funciones de dichas capas, evitando así un desarrollo de lógica dentro de la capa de interfaz.

En cuanto al diseño, se utilizan modales para mejorar la experiencia del usuario. Para facilitar la implementación de los mismos, se hizo uso de [Blazored Modals](#).











Detalle de cobertura

Como se aprecian en la imagen, todos los paquetes, Excepcion, Repositorio, Dominio y BusinessLogic fueron testeados aplicando TDD y logrando una cobertura de 100%. Para el análisis de cobertura no se tomó en cuenta la capa de interfaz así como la cobertura de los proyectos de tests.







Symbol	Coverage (%) ▲	Uncovered/Total Stmts.
▲  Total	100%	0/642
▷  Excepcion	100%	0/24
▷  Repositorio	100%	0/114
▷  Dominio	100%	0/250
▷  BusinessLogic	100%	0/254

Este porcentaje se tuvo muy en cuenta a lo largo de todo el proyecto, siempre asegurándose de que con los test se cubrieran todos los posibles escenarios de cada funcionalidad.







Cobertura de Excepcion:

▲  Excepcion	100%	0/24
▲  Excepcion	100%	0/24
▷  DepositoLogicExcepcion	100%	0/3
▷  DominioDepositoExcepcion	100%	0/3
▷  DominioPromocionExcepcion	100%	0/3
▷  DominioReservaExcepcion	100%	0/3
▷  DominioUsuarioExcepcion	100%	0/3
▷  PromocionLogicExcepcion	100%	0/3
▷  ReservaLogicExcepcion	100%	0/3
▷  UsuarioLogicExcepcion	100%	0/3







Cobertura del Repositorio

▲  Repositorio	100%	0/114
▲  Repositorio	100%	0/114
▷  ReservasRepository	100%	0/34
▷  DepositoRepository	100%	0/27
▷  PromocionRepository	100%	0/27
▷  UsuarioRepository	100%	0/26

Cobertura del BusinessLogic

▲  BusinessLogic	100%	0/254
▲  BusinessLogic	100%	0/254
▷  ReservasLogic	100%	0/82
▷  DepositoLogic	100%	0/72
▷  PromocionLogic	100%	0/53
▷  UsuarioLogic	100%	0/47

Cobertura del Dominio

▲  Dominio	100%	0/250
▲  Dominio	100%	0/250
▷  Reserva	100%	0/101
▷  Usuario	100%	0/67
▷  Deposito	100%	0/48
▷  Promocion	100%	0/34

Cálculo de precio de depósito

Escribir una prueba (Fase "Red")

Durante la fase "Red" de la implementación de pruebas, se identificó que el cálculo del precio del depósito puede variar según varios factores, como el tamaño del depósito, la climatización y la duración de la reserva. Se crearon múltiples métodos de prueba para cubrir los diferentes casos, lo que resultó en una repetición de código. Sin embargo, para cumplir con el enfoque de Desarrollo Guiado por Pruebas (TDD), se decidió mantener este código repetitivo y avanzar hacia la fase "Green".

Escribir el código (Fase "Green")

En la fase "Green", se establecieron los diferentes costos esperados según los detalles de la reserva. El objetivo era desarrollar una función que pudiera calcular estos costos esperados considerando los diferentes parámetros. Una vez escrita y probada la función, se procedió a la fase "Refactor".

Refactorizar el código (Fase "Refactor")

Durante la fase "Refactor", se realizó una revisión exhaustiva del código escrito para optimizarlo en términos de calidad, eficiencia y legibilidad. Para abordar la repetición de código en los tests, se implementó el uso de Datarows para reducir la cantidad de pruebas necesarias y mejorar la mantenibilidad del código y se comentó el código para tener para mayor claridad.

```
[TestMethod]
[DataRow('S', false, 6, 50.0)] // Size = S, Climatization = false, Days <7, price = 50
[DataRow('M', false, 6, 75.0)] // Size = M, Climatization = false, Days <7, price = 75
[DataRow('L', false, 6, 100.0)] // Size = L, Climatization = false, Days <7, price = 100
[DataRow('S', true, 6, 70.0)] // Size = S, Climatization = true, Days <7, price = 70
[DataRow('M', true, 6, 95.0)] // Size = M, Climatization = true, Days <7, price = 95
[DataRow('L', true, 6, 120.0)] // Size = L, Climatization = true, Days <7, price = 120
[DataRow('S', false, 7, 47.5)] // Size = S, Climatization = false, Days 7, price = 47.5
[DataRow('M', false, 7, 71.25)] // Size = M, Climatization = false, Days 7, price = 71.25
[DataRow('L', false, 7, 95.0)] // Size = L, Climatization = false, Days 7, price = 95.0
[DataRow('S', true, 7, 66.5)] // Size = S, Climatization = true, Days 7, price = 66.5
[DataRow('M', true, 7, 90.25)] // Size = M, Climatization = true, Days 7, price = 90.25
[DataRow('L', true, 7, 114.0)] // Size = L, Climatization = true, Days 7, price = 114.0
[DataRow('S', false, 14, 47.5)] // Size = S, Climatization = false, Days 14, price = 47.5
[DataRow('M', false, 14, 71.25)] // Size = M, Climatization = false, Days 14, price = 71.25
[DataRow('L', false, 14, 95.0)] // Size = L, Climatization = false, Days 14, price = 95.0
[DataRow('S', true, 14, 66.5)] // Size = S, Climatization = true, Days 14, price = 66.5
[DataRow('M', true, 14, 90.25)] // Size = M, Climatization = true, Days 14, price = 90.25
[DataRow('L', true, 14, 114.0)] // Size = L, Climatization = true, Days 14, price = 114.0
[DataRow('S', false, 15, 45.0)] // Size = S, Climatization = false, Days >14, price = 45
[DataRow('M', false, 15, 67.5)] // Size = M, Climatization = false, Days >14, price = 67.5
[DataRow('L', false, 15, 90.0)] // Size = L, Climatization = false, Days >14, price = 90.0
[DataRow('S', true, 15, 63.0)] // Size = S, Climatization = true, Days >14, price = 63.0
[DataRow('M', true, 15, 85.5)] // Size = M, Climatization = true, Days >14, price = 85.5
[DataRow('L', true, 15, 108.0)] // Size = L, Climatization = true, Days >14, price = 108.0
// 0 references
public void Calculo_Precio_Sin_Descuento(char Tamano, bool climatization, int days, double expectedPrice)
{
```

```
private double CalcularCosto()
{
    double costoBase = 0;

    // Calcular el costo base según el tamaño del depósito
    if (Deposito.Tamano == 'S') { costoBase = 50.0; }
    else if (Deposito.Tamano == 'M') { costoBase = 75.0; }
    else if (Deposito.Tamano == 'L') { costoBase = 100.0; }

    // Aplicar suplemento por climatización si es necesario
    double costoClimatizacion = Deposito.TieneClimatizacion ? 20.0 : 0.0;

    // Calcular el costo total sin descuentos ni promociones
    double costoTotal = costoBase + costoClimatizacion;

    // Calcular la duración del alquiler en días
    TimeSpan diferencia = Fin - Comienzo;
    int cantDias = (int)diferencia.TotalDays + 1;

    // Aplicar descuentos por duración del alquiler
    if (cantDias >= 7 && cantDias <= 14) { costoTotal *= 0.95; }
    else if (cantDias > 14) { costoTotal *= 0.90; }

    // Aplicar promociones vigentes
    double descuentoPromociones = 0;
    for (int i = 0; i < Deposito.Promociones.Count && descuentoPromociones < 1.0; i++)
    {
        Promocion promo = Deposito.Promociones[i];
        if (promo.EstaVigente())
        {
            descuentoPromociones += 0.01 * promo.Descuento;
            if (descuentoPromociones > 1.0)
            {
                descuentoPromociones = 1.0;
            }
        }
    }

    costoTotal *= 1.0 - descuentoPromociones;

    return costoTotal;
}
```

Evidencia de pruebas funcionales

Video: <https://youtu.be/HkTLTN9yh7I>