

Universidad ORT Uruguay
Facultad De Ingeniería

Obligatorio Diseño de Aplicaciones 1

**Entregado como requisito para la obtención del Obligatorio 1 Diseño
de Aplicaciones 1**

Federico Gutiérrez - 262684

Franco Ramos - 230508

Docente - Tutor: Gastón Mosques

Docentes Practico: Ignacio Azaretto – Martin Radovitzky

Grupo: M5D

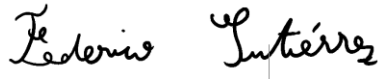
2024

https://github.com/IngSoft-DA1-2023-2/262684_230508

Declaración de autoría

Nosotros, Federico Gutiérrez y Franco Ramos, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos el Obligatorio 1 de Diseño de Aplicaciones 1;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas como por ejemplo la inteligencia artificial “ChatGPT” y el asistente virtual de inteligencia artificial “copilot” entre otros;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes;



Federico Gutiérrez

09/05/2024



Franco Ramos

09/05/2024

Abstract:

El obligatorio a realizar está compuesto por 5 grandes secciones.

Estas secciones serian:

- User Interface (Interfaz de Usuario)
- Business Logic (Lógica de Negocio)
- Repositories (Repositorios Del Dominio)
- Model (Dominio de la aplicación)
- Tests de cada sección

Palabras Clave

- Blazor
- C#
- Unit Tests
- Code Average Tests
- Clase abstracta
- Polimorfismo
- Gitflow
- Bootstrap

Índice

Declaración de autoría.....	2
Abstract:	3
Palabras Clave	3
Descripción general del trabajo a realizar	5
Git Flow	5
Test Driven Development (TDD).....	5
Clean Code	5
Bugs y funcionalidades no Implementadas	6
Descripción y justificación del Diseño.....	6
Dependencias entre Paquetes.....	9
Diagramas de clases (UML)	11
Mecanismos Generales y decisiones de Diseño	13
Criterios seguidos para asignar Responsabilidades	13
Mantenibilidad y Extensibilidad	13
Análisis de Dependencias.....	14
Pruebas Unitarias y Cobertura de Código	14

Descripción general del trabajo a realizar

En la siguiente propuesta de trabajo a nuestro equipo de programadores se nos solicitó desarrollar una aplicación para la iniciativa emergente denominada “DepoQuick” que busca revolucionar el mercado de alquiler de depósitos. La misma tiene como nombre designado Producto Mínimo Viable (MVP).

DepoQuick es un sistema que permite a un usuario cliente poder registrarse y alquilar un depósito de forma sencilla y eficaz. Al mismo tiempo pudiendo mantener control del mismo y de toda su información.

DepoQuick también es un sistema que permite el registro de **un solo** administrador para la gestión de todos los depósitos y sus estados (Alta, Baja, Promociones, Reservas, etc.).

En la siguiente documentación y aplicación se cumplen todos los requerimientos (funcionales y no funcionales) solicitados por el cliente en su mayor propósito.

Git Flow

Git Flow es un modelo de flujo de trabajo que organiza el desarrollo de software. Esta metodología se utiliza para crear distintas ramas de características (features) que trabajan en paralelo en diferentes funcionalidades del sistema que posteriormente se integran al proyecto principal una vez finalizadas.

En este proyecto trabajamos con la rama principal main y la develop. En la main se almacena una versión completa final y estable de la aplicación. Por otro lado, en la rama develop se fueron creando y fusionando las conocidas “features”.

Test Driven Development (TDD)

El Desarrollo Guiado por Pruebas (más conocido como TDD) es un método de desarrollo de software que se utiliza bastante en la actualidad. Consiste en realizar pruebas de una funcionalidad previo a su codificación. En este proyecto realizamos gran uso del mismo para la parte lógica del sistema. Su ciclo de desarrollo consiste en tres simples fases:

El proceso se divide en tres fases:

1. Fase “RED”: Consiste en escribir una prueba para verificar un comportamiento esperado de una funcionalidad. Esta misma no funciona e incluso puede que ni siquiera compile.
2. Fase “GREEN”: Se basa en implementar la funcionalidad para que la prueba pase, sin tener en cuenta si cumple con el “clean code”, si es eficiente, etc.
3. Fase “REFACTOR”: Finalmente luego de que la prueba pase se comienza a refactorizar (optimizar, reestructurar, etc.) el código. Esto deriva en mejor mantenibilidad, eliminando los code smells como código repetitivo, ineficiencias, etc.

Del empleo del TDD logramos destacar ciertas ventajas:

- Mejora la calidad de código
- Detección temprana de errores
- Acelera el proceso de desarrollo a largo plazo

Clean Code

En el desarrollo de nuestra aplicación dedicamos mucho tiempo para asegurarnos que nuestro código sea fácil de comprender, mantener y hasta mejorar. Eso resulta en aplicar una programación de tipo “clean code”. La misma se centra en la escritura de código de alta calidad para que sea legible, eficiente y bien estructurada. Con este “método” quiera llamarse así logramos evitar los conocidos “code smells”. Estos se refieren a estructuras o prácticas presentes en el código fuente que pueden indicar la presencia de problemas o deficiencias en el diseño del mismo. Por lo dicho anteriormente evitamos la duplicación de código, métodos o funciones largas, clases o módulos con demasiadas responsabilidades, entre otros.

Bugs y funcionalidades no Implementadas

En este punto queremos notificar algunos errores conocidos en nuestro proyecto ya que debido a falta de tiempo o errores personales no pudimos concretar sus funcionalidades.

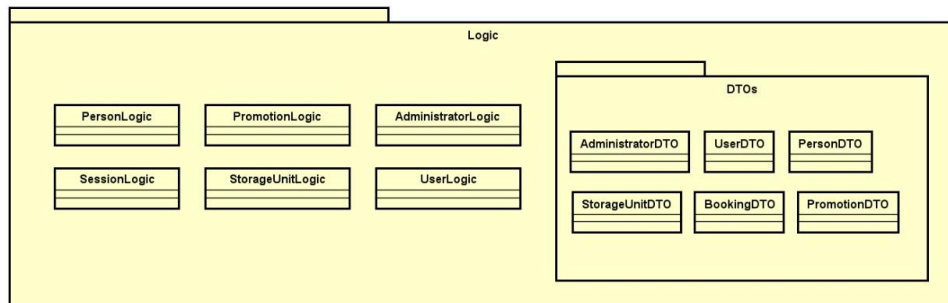
Creando la lógica de administrador en la UserInterface nos percatamos que habíamos realizado la lógica solamente del usuario. Para poder arreglar dicho problema, necesariamente tendríamos que crear una lógica para la persona, la cual es heredada por el usuario y el administrador. Así mismo, también debemos cambiar el repositorio de usuario para que sea un repositorio de persona. Realizando este cambio también podemos guardar el **único** administrador de la página en el repositorio con los usuarios. Estos cambios dichos anteriormente los realizamos en una rama de nombre **hotfix/Repositories&Logic**.

Un bug el cual arreglamos de forma no tan agradable fue en la página **CreateStorageUnit**. Este bug sucede cuando se aprieta el botón para incluir una promoción. La misma si se des-selecciona no se elimina correctamente de la lista de promociones que se utiliza para crear el objeto StorageUnit. Por lo tanto, la función de lista **.Remove()** la cambiamos por una función **.RemoveAll()**.

Descripción y justificación del Diseño

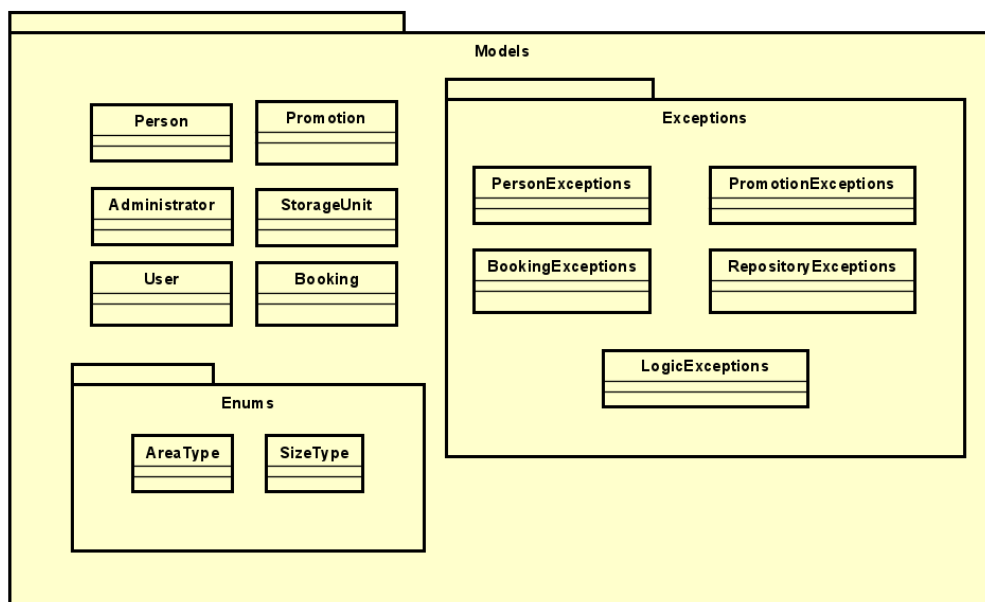
Diagrama de Paquetes

Paquete Logic



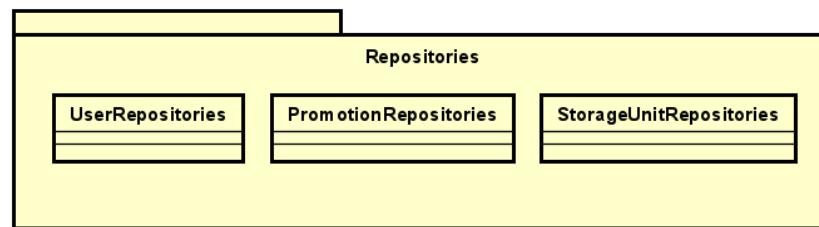
- La principal responsabilidad de este paquete no se define de otra forma que, con su nombre Lógica. Es el encargado de controlar la lógica de la página y mantener la sesión activa de una persona en la misma guiándola por todas sus interfaces y funciones que ofrece la misma. Tiene dependencia con el paquete de repositorios ya que obtiene las listas de los objetos para poder interactuar lógicamente con ellos. Con la pagina interactúa a través de los DTOs para no hablar directamente con el backEnd.

Paquete Model



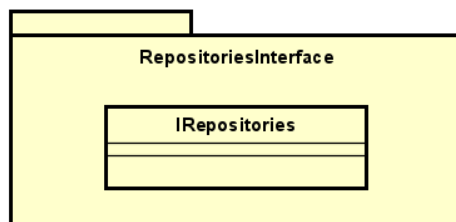
- La principal responsabilidad de este paquete es definir todas las clases que se utilizarán en la aplicación, tanto como sub-paquetes para el utilizo de excepciones y de enums. Principalmente en el mismo se representa la estructura de nuestros objetos para ser posteriormente ser utilizados en la aplicación, como, por ejemplo, atributos, métodos, herencias de objetos, entre otros. Cabe destacar que este paquete al ser el Dominio de la aplicación no depende de ningún otro.

Paquete Repositories



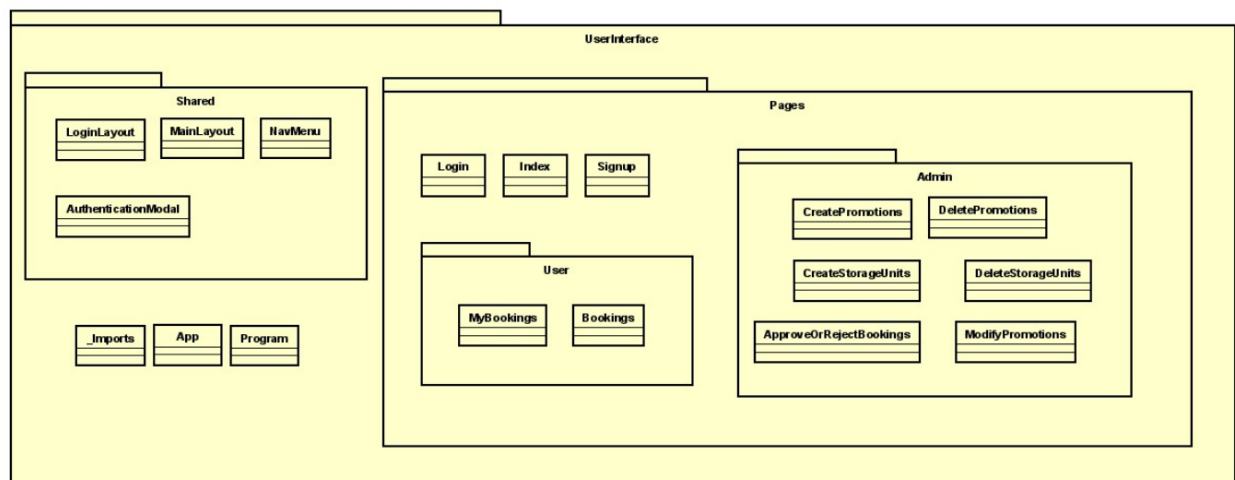
- La principal responsabilidad de este paquete es similar al de una Base de Datos, es decir, es encargado de guardar toda la información en memoria de lo que necesite la aplicación. Por ejemplo, los usuarios, promociones, depósitos y reservas (que se guardan dentro del usuario) creados, entre otros. Tiene dependencia con el paquete Model ya que almacena objetos del mismo y con el paquete RepositoriesInterface ya que heredan la interfaz del repositorio.

Paquete RepositoriesInterface



La principal responsabilidad de este paquete es almacenar el modelo de la interfaz de un repositorio cualquiera con sus operaciones básicas (agregar, eliminar, existe, obtener, obtener todos) para la aplicación que se hereda por los repositorios PersonRepositories, PromotionRepositories y StorageUnitRepositories en los cuales se implementan posteriormente.

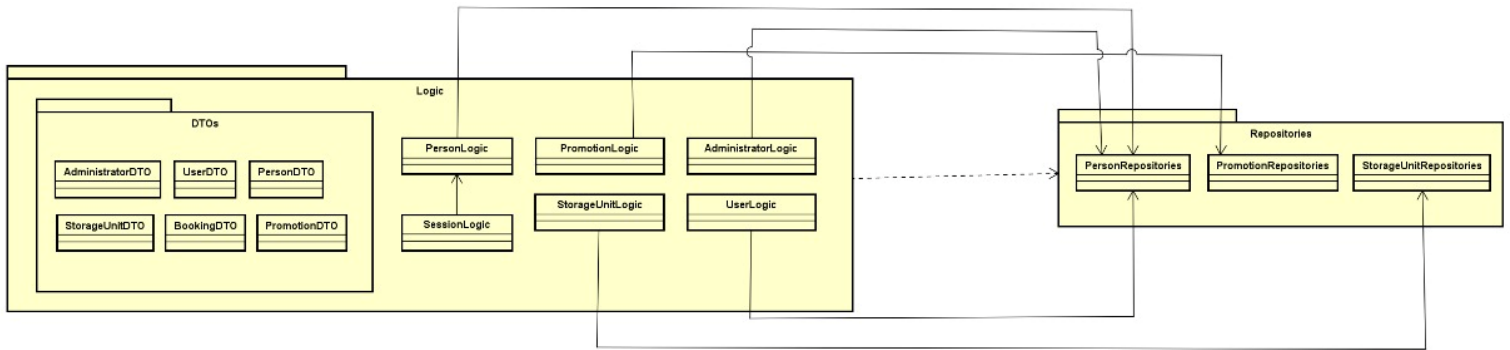
Paquete UserInterface



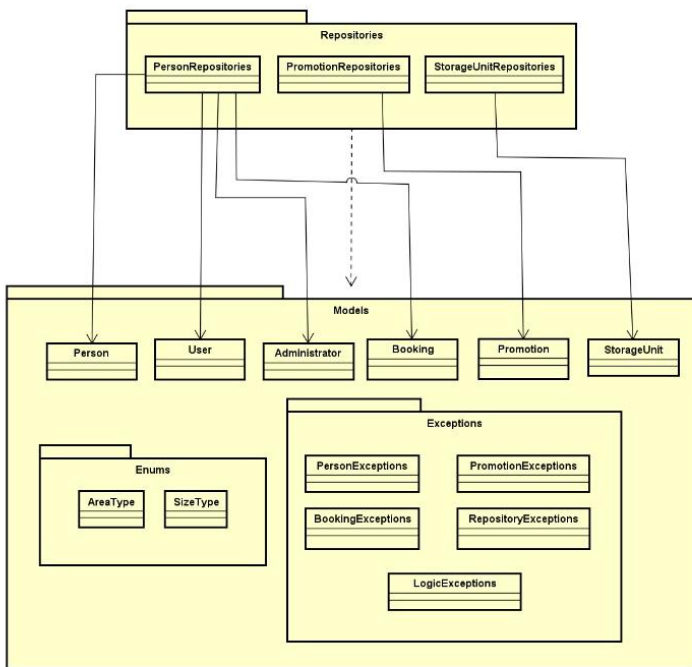
- La principal responsabilidad de este paquete es almacenar toda la información de la página web. Se hace cargo que se muestren las interfaces y sus funcionalidades (implementadas en los otros paquetes) al usuario de forma adecuada, guiándolo de la mejor forma posible para que no tenga inconvenientes con su utilizzo. En este paquete como se puede apreciar tiene muchos sub-paquetes como Pages (encargado de las páginas de la aplicación) y Shared (que se ocupa de la navegabilidad y estilos de las interfaces), entre otros.

Dependencias entre Paquetes

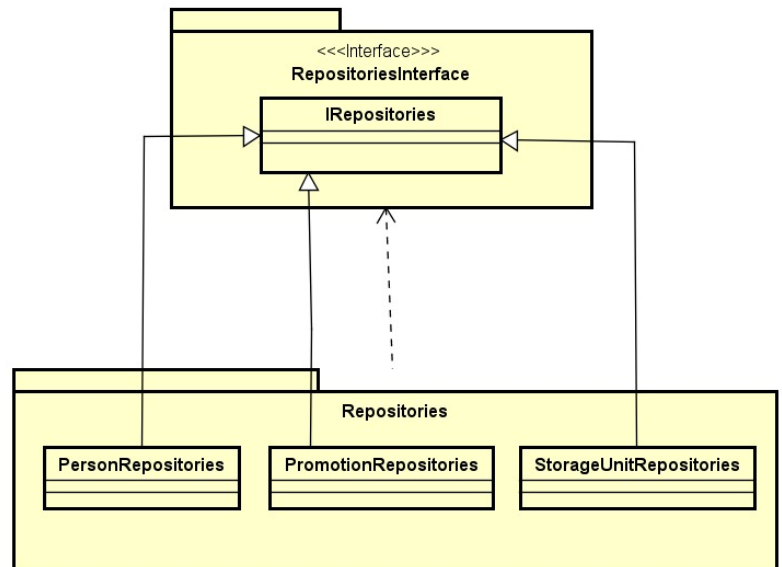
Dependencia Logic-Repositories



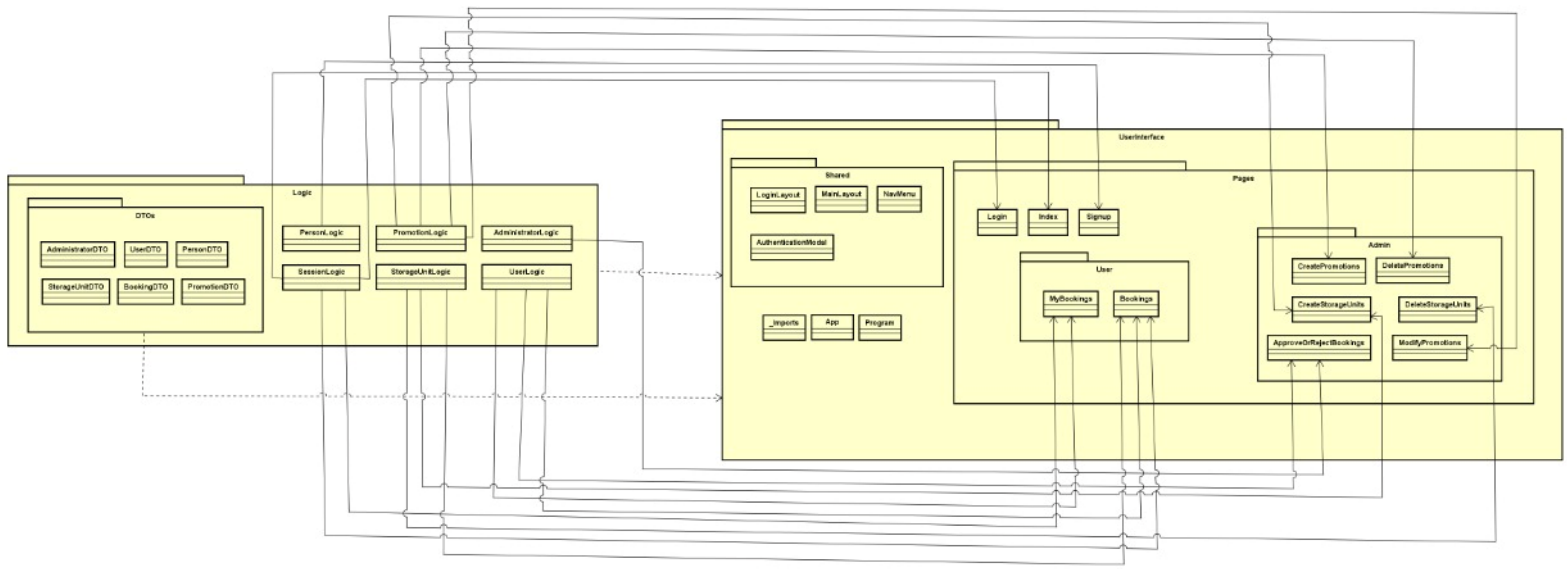
Dependencia Repositories-Model



Dependencia Repositories-RepositoriesInterface

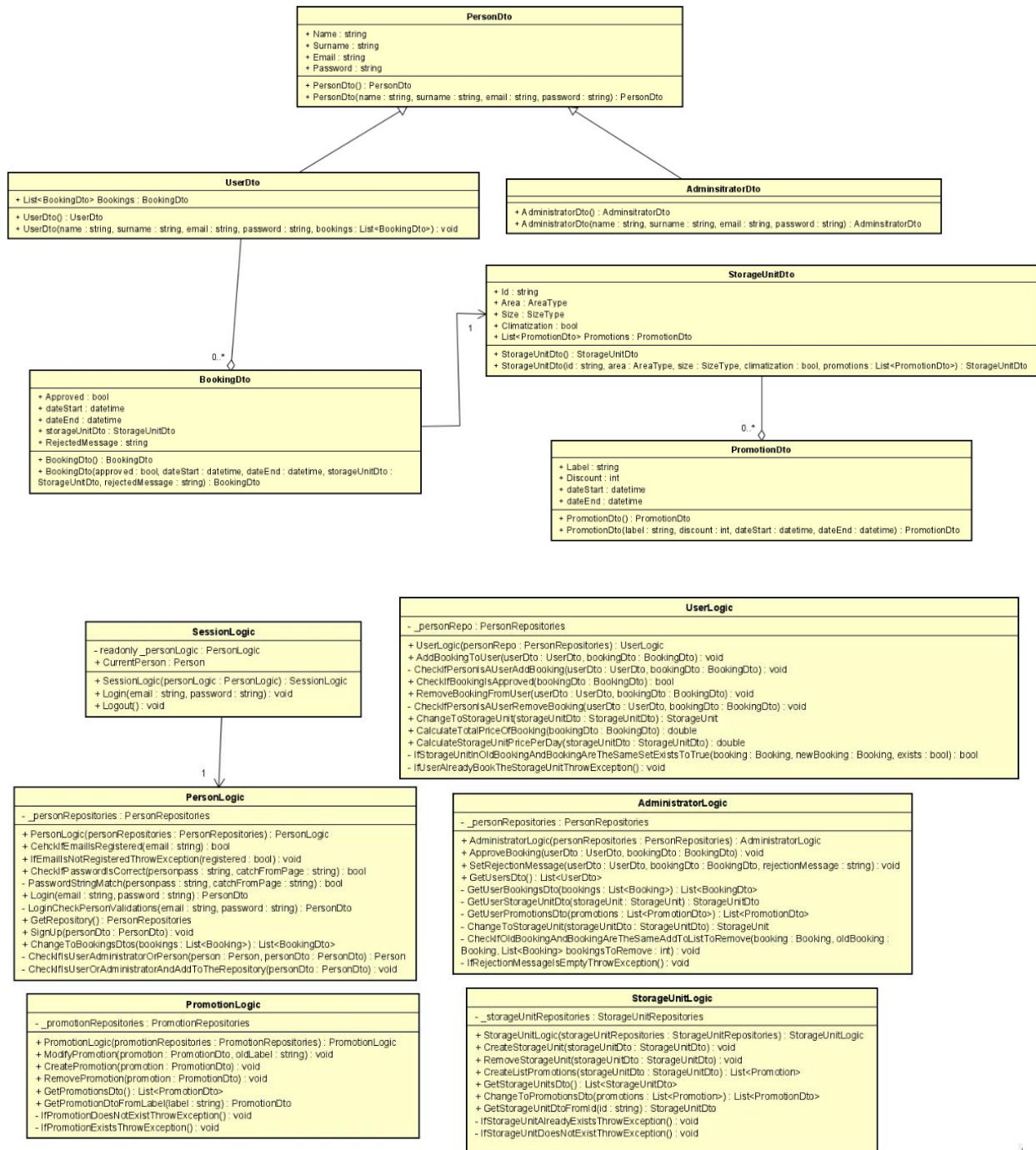


Dependencia UserInterface-Logic

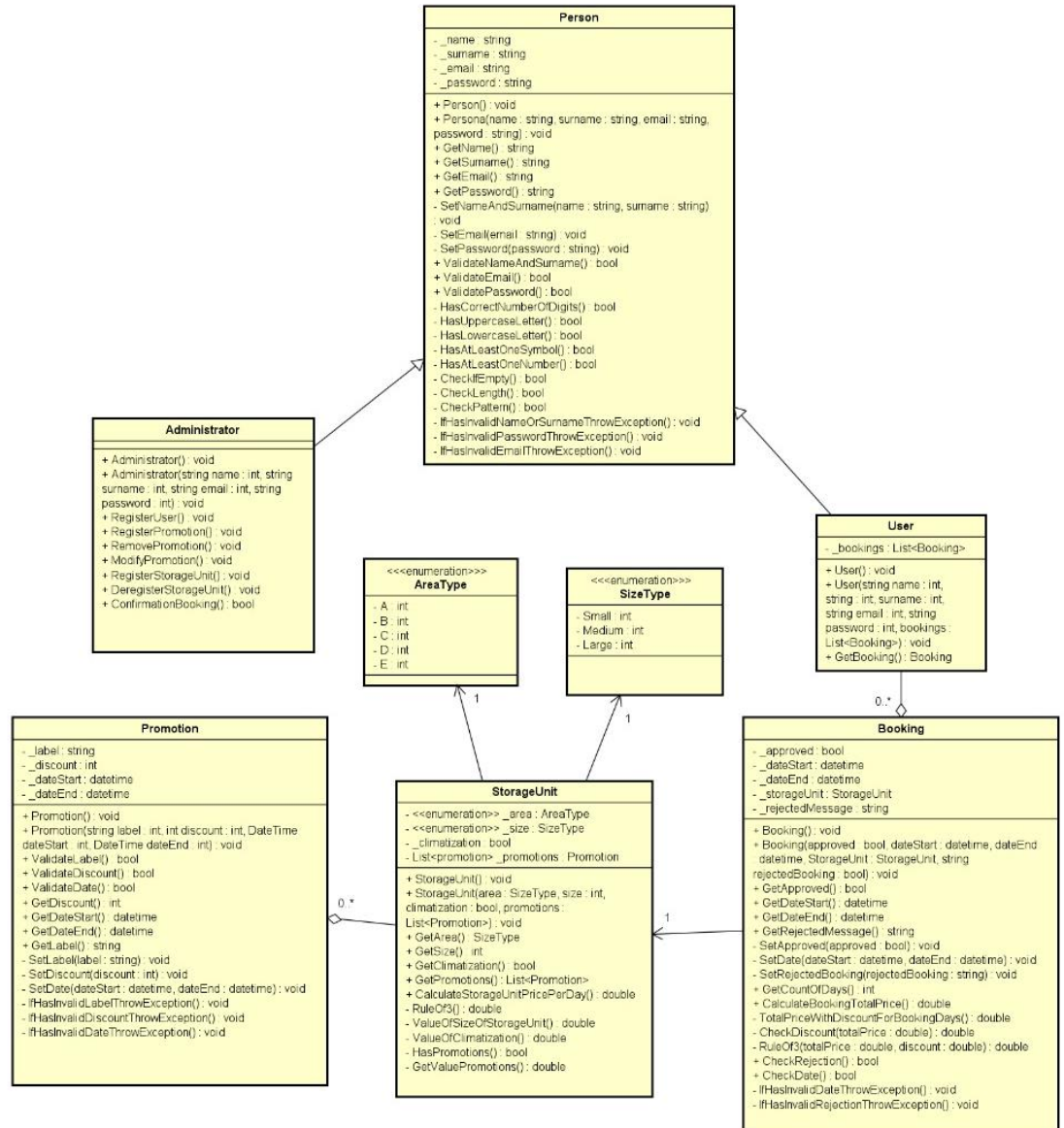


Diagramas de clases (UML)

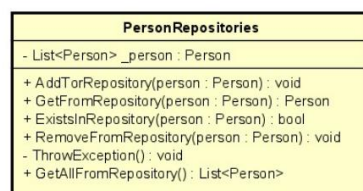
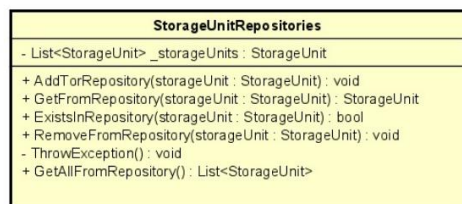
Paquete Logic



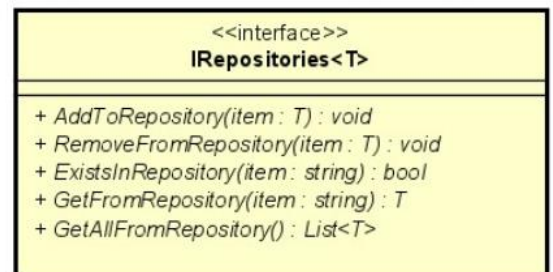
Paquete Model



Paquete Repositories



Paquete RepositoriesInterface



Mecanismos Generales y decisiones de Diseño

Interacción Interfaz-Dominio

La forma en la cual la interfaz interactúa con el Dominio es a través del paquete Logic, el cual tiene conocimiento de todos los objetos que posteriormente al ser creados se guardan en los repositorios en memoria.

Almacenamiento de datos

En este caso el almacenamiento de datos se realiza en el paquete Repositories. No fue requerido el almacenamiento de información en base de datos, por lo tanto, se almacena en memoria. Próximamente se implementará la persistencia de los datos para un diseño más íntegro.

Manejo de Errores y Excepciones

El manejo de errores y excepciones fue considerado en su totalidad. Se crearon las necesarias excepciones para los errores más frecuentes del ingreso de datos de un usuario promedio. Esto teniéndolo en cuenta para todos los objetos. Por otro lado, en la Interfaz de usuario utilizamos sentencias de try-catch para capturar algunas excepciones ya creadas anteriormente.

Utilización de Polimorfismo

La utilización del polimorfismo es una práctica muy utilizada a día de hoy por parte del mundo de la programación. Se hizo priorizar en su uso y se adaptó de la mejor forma en el proyecto. En nuestro caso, presentamos una herencia de Persona por parte de Usuario y Administrador, tanto como para la creación de los objetos que comparten atributos, tanto como para la Lógica de los mismos en la página.

Criterios seguidos para asignar Responsabilidades

En nuestro caso nos planteamos realizar en su mejor forma la asignación de responsabilidades. Estamos abiertos a críticas constructivas porque puede ser que no haya sido la más óptima. Priorizamos que cada objeto tenga en su totalidad solamente tareas que sean específicamente solo de sí mismo, respetando la cohesión, el encapsulamiento, la abstracción, entre otros.

Mantenibilidad y Extensibilidad

Para mantenibilidad utilizamos la favorable práctica de herencia. En el caso de PersonRepositories que almacena personas ya sean usuarios o administradores. Un objeto Person el cual comparte todos sus atributos con un User y un Administrator. También con todos los Repositorios que heredan la interfaz de IRepositories. Esta práctica evita la duplicación de código y ayuda a mantener coherencia en la aplicación.

Para extensibilidad, por ejemplo, en nuestro caso podría ser una notificación al usuario que su depósito está por vencer. Cuando este ingresa con sus datos le saltaría una notificación que la reserva de cierto depósito está por terminar, y este debería enviar una petición para prolongarla si así lo desea.

Análisis de Dependencias

Para la función: Cálculo de precio de depósito

En este caso las dependencias serían:

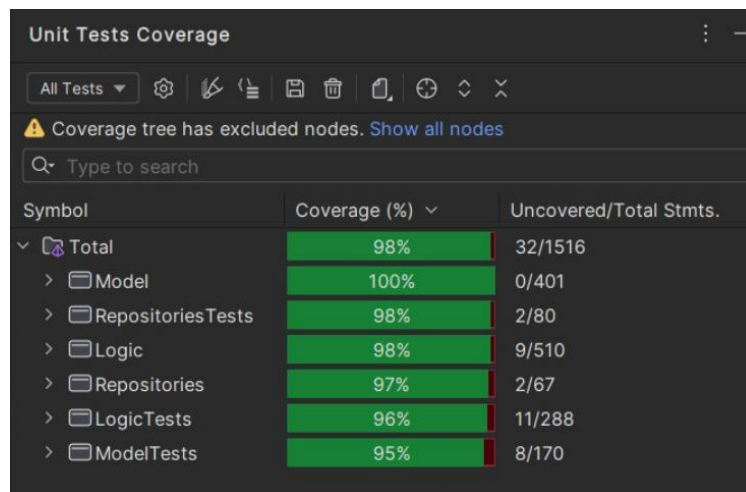
- StorageUnit depende de Promotion
- Booking depende de StorageUnit

Nosotros para implementar esta función comenzamos calculando el precio del depósito por día en el mismo objeto StorageUnit. Verificando con métodos privados el valor asignado a su tamaño, el valor asignado si tiene climatización y por último el valor asignado si tiene promociones incluidas. Finalmente sumando dichos valores y aplicando una regla de tres si tiene promociones vigentes.

Por otro lado, en la clase Booking para obtener el precio del depósito **final** faltaba verificar si se aplicaba un descuento al precio total por la duración de los días. Se chequea esa última condición y se multiplica los días del Booking por el precio por día del depósito calculado en StorageUnit.

Pruebas Unitarias y Cobertura de Código

En el recorrido de este largo proyecto, se incentivó a utilizar la metodología de TDD (Test-Driven Development) para realizar tests previo a la codificación del código final. En nuestro caso mantuvimos control de la cobertura de nuestro código en todo momento, re factorizando si algunos casos no se cubrían con nuestras líneas de código. En términos generales llegamos a un muy buen porcentaje de cobertura por nuestra dedicación y mantenimiento de nuestro código. Se puede apreciar en la imagen que se pierde un mínimo porcentaje de cobertura, pero esto es debido a los casos de excepciones que se realizaron al largo del proyecto para reafirmar sus validaciones al momento de ingreso de datos.



Caso: Calculo de precio de deposito

TDD en StorageUnit – CalculatingStorageUnitPricePerDayWithValidations_ShouldReturnPrice

[RED]: Como en cualquier otro test planteamos datos de un StorageUnit aleatorio con sus respectivos datos y chequeamos por ejemplo con un "Assert.AreEqual(100, storageUnit.CalculateStorageUnitPricePerDay())"

[GREEN]: Siguiendo la metodología de TDD hacemos un return con el valor esperado solo para que la prueba pase

[REFACTOR]: Finalmente vamos agregando los métodos ValueOfSizeOfStorageUnit(), ValueOfClimatization(), RuleOf3(), HasPromotions() y GetValuePromotions() que calculan distintos datos para el valor final del método original que es de tipo **double** ya que puede tener decimales luego de la coma.

TDD en Booking - CalculatingBookingTotalPriceWithValidations_ShouldReturnTotalPrice

[RED]: Planteamos un test con datos de una Promotion, un StorageUnit y una Booking aleatorios creados en un TestInitialize. Guardando la promoción en una lista dentro del depósito y este último dentro de la reserva.

[GREEN]: Hacemos un return con el valor esperado por el Assert para que la prueba pase con el mínimo código.

[REFACTOR]: Para finalizar creamos los métodos GetCountOfDays(), CheckDiscount() y RuleOf3(). Utilizando el resultado del método en StorageUnit y aplicándole estos últimos tres dichos anteriores obtenemos el precio total de un StorageUnit para ese Booking.

Casos de prueba

Test - CalculatingStorageUnitPricePerDayWithValidations_ShouldReturnPrice

```
List<Promotion> promotions = new List<Promotion>();

Promotion mypromotion = new Promotion("Descuento Invierno", 25, new DateTime(2024,7,15), new
DateTime(2024,10,15));

promotions.Add(mypromotion);

StorageUnit mystorageunit = new StorageUnit();

Assert.AreEqual(0, mystorageunit.CalculateStorageUnitPricePerDay());

mystorageunit= new StorageUnit("",AreaType.A, SizeType.Small, true,_promotions);

Assert.AreEqual(52.5, mystorageunit.CalculateStorageUnitPricePerDay());

promotions = new List<Promotion>();

mystorageunit = new StorageUnit("",AreaType.B, SizeType.Medium, false, promotions);

Assert.AreEqual(75, mystorageunit.CalculateStorageUnitPricePerDay());

mystorageunit = new StorageUnit("",AreaType.C, SizeType.Large, true, promotions);

Assert.AreEqual(120, mystorageunit.CalculateStorageUnitPricePerDay());
```

Test - CalculatingBookingTotalPriceWithValidations_ShouldReturnTotalPrice

```
List<Promotion> promotions = new List<Promotion>();

Promotion mypromotion= new Promotion("Descuento Invierno", 25, new DateTime(2024,7,15), new
DateTime(2024,10,15));

promotions.Add(mypromotion);
```

```
StorageUnit mystorageunit= new StorageUnit("",AreaType.A, SizeType.Small, true, promotions);  
Booking mybooking = new Booking(false, new DateTime(2024, 7, 1), new DateTime(2024, 8, 15),  
mystorageunit, "");  
Assert.AreEqual(2126.25, mybooking.CalculateBookingTotalPrice());  
mystorageunit= new StorageUnit("",AreaType.A, SizeType.Small, true, promotions);  
mybooking = new Booking(false, new DateTime(2024, 7, 1), new DateTime(2024, 7, 4), mystorageunit,  
"");  
Assert.AreEqual(157.5, mybooking.CalculateBookingTotalPrice());  
mybooking = new Booking(false, new DateTime(2024, 7, 1), new DateTime(2024, 7, 9), mystorageunit,  
"");  
Assert.AreEqual(399, mybooking.CalculateBookingTotalPrice());
```

Funcionalidades de Administración de Depósitos y Alta de Depósito

Video en youtube funcionalidades Depósitos:

<https://www.youtube.com/watch?v=UFLkqDX-iJs>

Video en youtube funcionalidades COMPLETO:

<https://www.youtube.com/watch?v=y4WHXnaXZqg>