Sistema de alquileres temporales

Informe

Programación Orientada a Objetos II

Profesores: Diego Cano - Diego Torres - Matias Butti

Integrantes: Lucas Coronel-Nicolás Ploza-Alexander Ferragut

Fecha: 14/11/2024

Detalles de implementacion y patrones de diseño utilizados

PUBLICACIÓN Y ALQUILER

El sistema conoce a muchos usuarios y a muchos inmuebles. Los Usuarios pueden ser inquilinos o propietarios. Los usuarios conocen su nombre completo, email y teléfono.
Un propietario puede dar de alta inmuebles en el sistema.
Para modelar el estado de un inmueble se utilizó el patrón Object Type para los atributos, Servicios, fotos, forma de pago.
Para ciudad, superficie, país, dirección, check-in, check-out y capacidad se utilizaron tipos primitivos.
Respecto al precio, el inmueble cuenta con un precio default por dia , el cual utiliza para calcular el precio en un periodo dado.
En caso de las fechas que pertenecen a un periodo específico declarado por el propietario, inmueble conoce a una clase Período la cual tiene una fecha de inicio y de fin y un costo por día.

BUSQUEDAS DE INMUEBLES

Para implementar la búsqueda de inmuebles se utilizó el patrón de diseño composite, donde el component es una clase abstracta llamada Search, la cual es superclase de las clases And y Or que serían dos composite, los cuales se construyen a partir de dos objetos tipo Search. En cuanto a los leaf se tratan de filtros simples que tambien heredan de la clase Search, donde cada uno recibe una lista de inmuebles y devuelve una lista de inmuebles que cumplan su condición de filtrado. Para el método filtrar se utilizo el patron Template method donde el método primitivo es la condición de filtrado de cada Search.

RANKING DE INMUEBLES Y PROPETARIOS

Tanto la clase Usuario como Inmueble implementan la interfaz Rankeable con lo cual saben recibir calificaciones, calcular los promedios de los puntajes por categoría y promedio de puntajes en total.
La clase calificación tiene como atributo la "Categoría", que es un Object Type, y el puntaje para dicha categoría.

VISUALIZACION Y RESERVA

En el caso de esta funcionalidad el sistema tiene toda la lógica para brindarla mediante el uso de interfaz de usuario, el inmueble tiene todos los getters y tiene acceso a sus propietarios. En cuanto a los datos de propietarios, se accede mediante los getters de la clase Propietario, incluyendo su antiguedad en el sitio. Para la visualización de la cantidad de alquileres de propietarios, inquilinos y las veces que fue alquilado un inmueble, la clase sistema sabe responder a esos pedidos.

CONCRECION DE UNA RESERVA

La Reserva tiene como atributo el inmueble, el propietario, al potencial inquilino, y el metodo de pago.
Cuando se instancia una reserva el estado del mismo es "Pendiente". El propietario tiene la responsabilidad de aceptar dicha reserva, cuando este lo acepta el estado de la reserva pasa a "Aceptada".
* Los datos correspondientes en el inciso A, se refiere a los datos mínimos y fundamentales que tiene que tener una reserva: inquilino, propietario, inmueble (ASUMIMOS ESO).
Cuando se acepta la reserva se le envía un correo al email del Inquilino mediante la Reserva usando la clase GestionadorDeNotificaciones.

ADMINISTRACION DE RESERVAS PARA INQUILINOS

El Sistema brinda acceso a la siguiente información de las reservas para los distintos usuarios: reservas de una ciudad particular, reservas futuras, las ciudades en donde tiene reservas.
Cuando el inquilino desee cancelar la reserva entonces reserva dispara un evento hacia su GestionadorDeNotificaciones el cual le hace llegar los eventos correspondientes a EmailSender, el cual esta interesado en el evento de la cancelación de una reserva.

ADMINISTRACION DEL SITIO

El sistema tiene las logicas para realizar estas funciones, entran por interfaz de usuario. No optamos por crear una clase administrador porque solo delegaria al sistema.

POLITICAS DE CANCELACION

Para modelar las políticas de cancelación se optó por un strategy donde el rol del strategy lo tiene la interfaz PoliticaDeCancelacion donde las

concreteStrategy son: Cancelacion, SinCancelacion, Intermedia. El cálculo del costo para cada política de cancelación lo realizan estas.
Por otro lado, el inmueble contiene una de estas estrategias concretas y el calculo del costoDeCancelacion(): consiste en una delegación a la estrategia que tenga el inmueble para calcularlo.

NOTIFICACIONES

Registramos en esta parte a dos clases que notifican diferentes eventos, lo administramos con una clase manager llamada GestionadorDeNotificaciones, la cual dispara los eventos:

- Cancelación de un inmueble
- Reserva de un inmueble
- Baja de precio de un inmueble

Las clases que usan al gestionador para disparar estos eventos son Reserva e Inmueble, y las clases interesadas en estos eventos son AppMobile y SitioWeb, ambas a su vez implementan una interfaz para realizar las acciones correspondientes.

RESERVA CONDICIONAL

Una vez que la reserva esté aceptada y registrada en el sistema el usuario no podrá reservar el inmueble directamente, sino que podrá añadirse a la cola con el mensaje añadirALaCola.

PATRONES DE DISEÑO UTILIZADOS

ROLES PARA STRATEGY:
Context: Inmueble
Strategy: PoliticaDeCancelacion <<interface>>
ConcreteStrategyA: Cancelacion
ConcreteStrategyB: SinCancelacion
ConcreteStrategyC: Intermedia

ROLES PARA COMPOSITE:
Component: Search
Composite: And - Or

Leaf: FiltroPorFecha - FiltroCantHuespedes - FiltroPrecioMinimo - FiltroPrecioMaximo - FiltroCiudad

ROLES PARA TEMPLETE METHOD:
AbstractClass: Search
ConcreteClass: FiltroPorFecha - FiltroCantHuespedes - FiltroPrecioMinimo - FiltroPrecioMaximo - FiltroCiudad

ROLES PARA STATE:
Context: Reserva
State: EstadoReserva
ConcreteStateA: Pendiente
ConcreteStateB: Aceptada
ConcreteStateC: Cancelada

ROLES PARA SINGLETON:
Singleton: Pendiente - Aceptada - Cancelada

ROLES PARA OBSERVER:
ConcreteSubject: Reserva - Inmueble
Subject(ChangeManager): GestionadorDeNotificaciones
Observer: Interesado
ConcreteObserver: SitioWeb - AppMobile - EmailSender

ALCANCE

Se implementaron todas las partes del enunciado y se agregaron algunas características extra detalladas a continuación:

Cuentas e inicio de sesión:

- Al momento de dar de alta una cuenta con sus credenciales, se agregó la validación para que no haya una cuenta con el mismo nombre en uso. Ya que esto complicaría la distinción entre las cuentas registradas.

Contenido:

- Los contenidos del mismo tipo dentro de un mismo directorio no pueden tener el mismo nombre, para una búsqueda de contenido más precisa.

Privilegios:

- Un usuario puede tener más de un privilegio asignado, por ejemplo puede tener privilegios de escritura y borrado, porque así pueda editar y borrar un contenido como si fuera propio.
- Un usuario creador puede dejar de compartir un contenido con cierto usuario, porque así el usuario creador puede mantener su contenido en privado.

Búsqueda:

- El filtrado de archivos se puede hacer por usuario y también se puede hacer una búsqueda general en todo el sistema, esto es, de todos los usuarios del sistema obtener una colección con los archivos buscados. Esto se debe a que inicialmente se planteó la búsqueda general y luego se implementó que cada usuario pueda hacerlo en sus carpetas.

- El filtrado combinado se implementa de tal manera que el usuario pueda usar la cantidad de filtros disponibles que quiera y combinarlos en cualquier orden. Solo ingresando los parámetros de búsqueda se realizará el filtrado combinando los filtros correspondientes. En caso de que se agreguen más filtros en un futuro el filtro combinado se adaptará sin problema.

MODELO

#MiniMiUNQ:

Representa al sistema del tipo UNIX y claramente es uno de los objetos de mayor importancia sus colaboraciones de mayor relevancia están la parte de registro e inicio de sesión

- **registrar cuentas y quitarlas:** esto lo hace añadiendo o quitando una cuenta de su colección de cuentas registradas.

- **iniciar y cerrar sesión de las cuentas registradas**: esto lo hace añadiendo o quitando una cuenta registrada de su colección de sesiones activas.

- **y validar las credenciales**: comparando las credenciales ingresadas con las que tiene en su colección de cuentas registradas.

#Root y #Administrador:

Siendo subclase de la clase **#Cuenta** saben cómo iniciar sesión con sus credenciales y cerrar sesión. Ambas tienen la responsabilidad de dar de alta nuevas cuentas en el sistema y en el caso de Root también modificarlas y darlas de baja.

#Usuario:

También es subclase de la clase **#Cuenta** por lo que sabe cómo iniciar sesión con sus credenciales y cerrar, pero al iniciar sesión se le asigna un directorio ya que sabe como manipular su contenido. Esto es, acceder a sus carpetas y navegar en las mismas; modificar y compartir su contenido con diferentes privilegios; y buscar archivos entre su contenido a través de los diferentes filtros.

#Contenido:

Se divide en dos subclases (Directorio y Archivo). Es uno de los objetos con más importancia ya que el sistema Mini-MiUNQ está basado en un gestor de archivos, en los cuales los usuarios tienen control total si son autores, como crear, editar y borrar el contenido. También pueden compartir estos objetos con otros usuarios del sistema con diferentes privilegios.

PROGRESO

Comenzamos la realización del TP utilizando TDD para una construcción iterativa e incremental. Lo primero en testar fue que exista un sistema MiMiniUNQ con una cuenta de Root registrada. Siguiendo el enunciado se probó que las cuentas de administrador y root puedan dar de alta otras cuentas o modificar cuentas de usuario, y solo root pueda modificar y dar de baja cualquier tipo de cuenta. También se probó que solo haya una sola cuenta de root en el sistema, en caso contrario se lanza un error.

Luego se probó el inicio de sesión de las cuentas asumiendo que no se deben registrar nombres de cuenta que ya estén registrados y asumiendo que las credenciales ingresadas para iniciar sean correctas. En caso que no se cumpla lo antes mencionado, se lanza un error.

Para la parte de directorios, fuimos diseñándolos como objetos que tienen un nombre y un contenido, un bag que contiene todos los archivos de texto y directorios

en esa carpeta. Cada vez que los usuarios se registran en el sistema se les asigna un directorio con el nombre “Directorio Raíz” en el que podrán crear nuevos directorios o archivos de texto. Los archivos de texto fuimos diseñándolos como objetos que también tienen un nombre y un contenido, en este caso el contenido se basa en una cadena de caracteres.

Tanto el nombre como el contenido de los directorios y los archivos de texto pueden ser modificados por el usuario. Un directorio no puede tener el mismo nombre que otro directorio que se encuentre en la misma carpeta, lo mismo sucede con los archivos de texto.

A los usuarios le dimos la posibilidad de poder acceder a los directorios que se encuentren dentro del directorio actual en el que se ubica el usuario, al momento de registrarse al sistema y al otorgarle el primer directorio este se encuentra en la raíz. Un usuario además de poder ver el contenido de sus directorios también puede ver el texto de los archivos de texto que tenga creados.

Los usuarios también tienen la posibilidad de borrar tanto directorios como archivos de texto que se ubiquen en la carpeta actual en la que se encuentre el usuario. Para poder eliminar los directorios estos deben estar vacíos sino no se podrán borrar.

Le asignamos a los usuarios la posibilidad de poder ir a la carpeta anterior siempre y cuando no se encuentre en la carpeta raíz. Para hacerlo implementamos una Ordered Collection donde se van guardando las carpetas vistas en orden.

Cada usuario dentro del sistema puede compartir sus directorios o archivos de texto con otros usuarios del sistema dependiendo de los privilegios dados por el usuario creador. Lo fuimos implementando de la siguiente manera:

- Si se comparte con privilegio de lectura; al usuario que le llega el contenido compartido sólo podrá ver el contenido sin posibilidad de poder editarlo o borrarlo.
- Si se comparte con privilegio de escritura; al usuario que le llega el contenido compartido podrá ver el contenido y además podrá editar su nombre y contenido. El usuario creador podrá ver estos cambios realizados como también a todos aquellos que tengan compartido el contenido.
- Si se comparte con privilegio de borrado; al usuario que le llega el contenido compartido podrá ver el contenido y además podrá borrarlo. Cuando se borra, todo aquel que tiene ese contenido, sea el autor como otros usuarios, se le borrara de su cuenta sin posibilidad de recuperarlo.

Cada contenido, sea archivo o carpeta, tiene un colaborador interno llamado “privilegios” que es una lista con los 3 privilegios mencionados antes. Cuando un archivo se comparte con algún privilegio a un usuario, este usuario se agrega a la colección “usuariosAsignados” que pertenece al privilegio dado. Cada privilegio tiene esta colección y así se sabe a qué usuarios fue compartido. Un usuario puede

recibir varios permisos sobre un mismo contenido sin problemas; por ejemplo, puede tener permisos de escritura y borrado entonces el usuario podrá editar tanto el nombre como el contenido y también podrá borrarlo. Además, un usuario creador de un contenido podrá dejar de compartir su contenido a cierto usuario que el desee, entonces ese usuario ya no podrá modificar ni borrar el contenido.

Para el final, probamos las funcionalidades de los diferentes filtrados. Inicialmente se planteó una búsqueda general llevada a cabo por el sistema MiMiniUNQ, esto se logra pidiendo a cada usuario registrado sus archivos totales y luego aplicando los diferentes filtros a la totalidad de archivos. Pero luego decidimos que cada usuario pueda realizar una búsqueda con filtros en sus propios directorios. Para esto, fuimos testeando cada filtro que pidió el enunciado y finalmente testeamos el filtro combinado. El filtro combinado toma los parámetros de búsqueda ingresados y aplica los filtros correspondientes automáticamente. En caso de que los parámetros de búsqueda no sean válidos se lanza una excepción.

DIFICULTADES

Al principio nos topamos con la dificultad de comprender bien el dominio, esto nos llevó a averiguar más acerca de lo que nos pedía el enunciado. Un ejemplo de esto es que no sabíamos que era un sistema del tipo UNIX. Esto nos llevó a averiguar entre otras cosas que es ese sistema y cómo funciona la gestión de usuarios.

Otra cosa fueron algunas ambigüedades del enunciado que pudimos resolverlo consultando con los profesores y ayudantes

Una de las partes más difíciles fue el implementar el privilegio de borrado, ya que de la manera en la que fuimos modelando los objetos nos resultaba rebuscado que cada usuario que tenga ese contenido compartido se le eliminará. Tuvimos que crear un mensaje para usuario para que pudiera borrar el contenido de forma “especial”, sin necesidad de tener algún privilegio, obviamente el usuario que decida borrarlo en primera instancia debe ser el autor o tener dicho permiso de borrado.

Primero habíamos pensado hacerlo usando el sistema Mini-MiUNQ pero decidimos que la mejor forma era que cada usuario que tenga ese contenido pueda borrarlo de una forma diferente a como se borran naturalmente los contenidos en nuestro sistema.

Finalmente, el filtro de búsqueda combinado nos presentó bastantes problemas. primeramente no comprendiamos lo a que se referia el enunciado con combinar los filtros. Primeramente lo planteamos como un mensaje que sabía responder el objeto Usuario colaborando con los 3 filtros antes implementados y devolviendo una

colección de archivos resultado de los 3 filtros realizados. El problema con esta implementación es que no permitia al usuario combinar los filtros como quiera y que siempre debían ser los 3 filtros en el orden establecido por el mensaje. Luego de varios intentos pudimos llegar a la implementación final que puede realizar la búsqueda con la cantidad de filtros que el usuario quiera y en el orden que quiera ingresarlo. Además en caso de que se agreguen más filtros el programa se adapta sin problema solamente hay que agregar la condición de filtrado y la selección del parámetro de búsqueda.

CONCLUSIONES

Luego de la realización de este TP y en vista del proceso y las dificultades mencionadas anteriormente podemos concluir en dos ideas principales:

-importancia de TDD

En vista de un dominio más complejo se vio clara la necesidad de usar TDD para la realización de este TP. No solamente a la hora de realizar la implementaciones sino al momento de revisar y refactorizar hubiese sido muy complicado si no se hubiese utilizado esta metodología.

-importancia de entender el dominio.

Comprender el dominio antes de empezar a programar. Saber qué es lo que queremos hacer y no codear sin tener una idea en mente.