Trabajo Práctico Final
Sitio Web de Alquileres

Integrantes:

Nehuen Gallitelli - nehuengallijans@gmail.com
Cristian Sanabria - crissanabria1995@gmail.com
Facundo Mosquera - facundomosquera@gmail.com

Materia: Programación con Objetos 2

Contenido

Decisiones de Desarrollo

Introducción

Antes de comenzar a programar se interpretó el enunciado en su totalidad y se debatió cuales iban a ser los principales objetos, como así también, las tareas que debían realizar cada uno de ellos. En ese momento se decidió que los objetos *`SitioWeb`*, *`Usuario`* y *`Reserva`* sean los que resolvieran la mayoría de las funcionalidades, pero a medida que se avanzó en el desarrollo del trabajo fueron surgiendo más objetos que cumplen un rol para poder llevar a cabo todas las partes del enunciado en su totalidad, por ejemplo, para dar de alta se crearon tres clases, *`TipoDeServicio`*, *`TipoDeInmueble`* y *`Categoria`*. Por otro lado, para poder realizar el ranking de los usuarios e inmuebles se creó la clase *`Rankeo`*. Para las reservas se creó una clase intermedia llamada *`SolicitudDeReserva`* que permite un mejor manejo de los estados de las reservas. Otra de las funcionalidades es el uso de una interfaz *`Filtro`* que se relaciona con otras clases para poder buscar uno o varios inmuebles que cumplan con los filtros pasados por parámetro.

Manejo de los Usuarios

En un primer momento se barajó crear una clase *`Usuario`* del tipo Abstracta y que tuviera dos hijos, un *UsuarioPropietario* y un *UsuarioInquilino*, pero luego de releer el enunciado y consultar en clase se llegó a la conclusión de que esto podría generar un problema de datos duplicados si un mismo usuario quiere ser Inquilino y Propietario. Por lo que, se tomó la decisión de crear una sola clase llamada Usuario, la cual permite ser Propietario e Inquilino sin la necesidad de tener dos clases diferentes. Por otro lado, el enunciado pedía que las funcionalidades de Ranking y Comentarios sepan diferenciar a un Inquilino de un Propietario, por lo que se tomó la decisión de crear listas específicas para Inquilino y Propietario dentro de la clase Usuario y listas específicas para las categorías de cada uno dentro del *`SitioWeb`*, lo que nos permitió diferenciar entre ambos y mantener la consistencia de sus datos.

Administrador

En el caso de esta funcionalidad en un primer momento se decidió crear una clase Administrador, pero luego al ver que la misma no poseía un comportamiento en particular más que los métodos de dar de alta, se tomó la decisión que los métodos solicitados formen parte de la clase *SitioWeb* donde va a poder dar de alta los diferentes tipos de servicios, tipos de inmueble y categorías. Para que esto sea posible se crearon tres clases diferentes correspondientes a cada tipo, *TipoDeServicio*, *TipoDeInmueble* y *Categoria*. El *SitioWeb* además de darlas de alta permite que el Usuario pueda verificar que la *Categoria* que esta queriendo utilizar para rankear sea una existente, que el tipo de inmueble seleccionado sea uno existente y que los tipos de servicios también estén correctamente dados de alta. Por otro lado, posee una lista de todos los usuarios del sitio, una lista de inmuebles dados de alta para que el Usuario pueda buscar los inmuebles que cumplen con los filtros pasados por parámetro.

Búsqueda de inmuebles

En este apartado decidimos implementar un método *buscarInmuebles* dentro de la clase *SitioWeb*, el cual recibe por parámetro los diferentes tipos de filtrado que el Usuario quiere aplicar a la búsqueda generando una lista de filtros, la cual es recorrida para obtener el o los inmuebles que cumplan con los filtros. Para esto creamos cuatro clases de filtros, *FiltroCiudad*, *FiltroPrecio*, *FiltroFechas*, *FiltroCantHuespedes*, que heredan de una interfaz llamada *Filtro*. Cuando el usuario decide buscar un inmueble, el sitio web interactúa con esta interfaz, pasándole como argumento su lista de inmuebles y recibe como resultado otra lista con los inmuebles filtrados, la cual es retornada en el método buscarInmuebles del *Usuario* para que pueda elegir un inmueble a visualizar y su posible *Reserva*.

Solicitudes de reserva y Reservas

Para manejar las reservas que un *Usuario* puede realizar en el sistema, decidimos crear una clase intermedia llamada *SolicitudDeReserva* la cual tiene dos estados, *EstadoDeSolicitudPendiente* y *stadoDeSolicitudAprobada*. Esta clase se encarga de informarle al Propietario del inmueble que hay una solicitud de reserva pendiente, permitiéndole al mismo visualizar información del Usuario que solicita la misma y en caso de estar conforme aceptar la misma.

Una vez aprobada la solicitud se da de alta la reserva mediante la clase *Reserva*, la cual tiene cuatro estados diferentes, *EstadoCofirmada*, *EstadoCacelada*, *EstadoFinalizada* y *EstadoCondicional*. Al momento de darse de alta se evalúa y si el inmueble no tiene una reserva en el rango de fechas de la nueva reserva, la misma pasa a *EstadoConfirmada*, pero en caso de que el inmueble ya tenga una reserva registrada en el rango de fechas, la reserva pasa a *EstadoCondicional*.

Para poder controlar si un inmueble está reservado cuando se recibe una nueva reserva, decimos guardar una lista de reservas en la clase Inmueble que cuando el Inmueble pasa a estar Reservado guarda esta referencia en la lista, lo que nos permite ver si el rango de fechas de una nueva reserva coincide con alguna reserva actual del inmueble sin tener que ir a buscar las reservas de cada Propietario y extraer las que son del Inmueble especifico que deseamos averiguar. Al finalizar una reserva, se elimina de la lista de reservas de la clase *Inmueble*.

Para el manejo de las reservas condicionales, se decidió utilizar el *EstadoCondicional*, lo que permite diferenciar a la reserva y guardarla en una lista de reservas condicionales dentro de la clase *Reserva*, lo que va a facilitar que cuando se cancele una reserva se verifique si hay alguna reserva condicional disponible para que pueda pasar a ser una reserva confirmada. Esta decisión nos permite un rápido acceso a las reservas condicionales, sin tener que ir a buscar las mismas a otra clase.

Ranking de inmuebles y propietarios

Al ver esta funcionalidad, surgió una incógnita. ¿Puede un usuario rankearse a sí mismo?, la respuesta a esta pregunta es: no debería. Por lo que se tomó la decisión de agregar dos listas diferentes dentro de *`Usuario`*, una que sea la lista de ranking perteneciente al Propietario y otra perteneciente al Inquilino, además para este apartado se diseñó la clase *`Rankeo`*, la cual nos permite tener una lista consistente con una categoría y un puntaje que están dentro de cada instancia de *`Rankeo`*.

. Por otro lado, se pedía que un usuario solamente pueda rankear a otro una vez finalizado el check-out de una reserva, por lo que la solución a esto fue aplicar un validador que verifique el estado de la reserva, si la misma tiene el estado finalizado se podrá ejecutar el método, caso contrario se dispara una excepción informando que la reserva aún no finalizó.

Notificaciones

Para el manejo de las notificaciones se tomó la decisión de agregar la clase *`Manager`* que se encarga de notificar a los usuarios interesados en bajas de precio y/o disponibilidad de reserva de un inmueble en particular. Para poder llevar a cabo esta lógica se tomó la decisión de utilizar el patrón *Observer* con una interfaz *`Listener`* que se encarga de conectar las clases que estén interesadas en recibir alguna notificación, por el momento siguiendo el enunciado las clases interesadas son *`OtroSitioWeb`* y *`AplicacionMovil.`* La interfaz *`Listener`* es implementada y conocida por la clase *`Manager`*, la cual se encarga de notificar a todos los suscriptores el evento correspondiente cuando el mismo sucede en la clase *`Reserva`* o Inmueble que conocen a la clase *`Manager`*.

A pesar de que la interfaz tenga dos métodos a implementar y que las clases *`AppMobile`* y *`OtroSitioWeb`* le dan comportamiento a un método diferente cada una, pero no a ambos, elegimos esta forma ya que si en el futuro una nueva clase se quiere suscribir a ambos eventos simplemente debe implementar la interfaz, lo que permite escalabilidad en el *Observer*.

Patrones de Diseños Utilizados

State

Utilizado para *SolicitudDeReserva*, con el objetivo de representar en qué estado se encuentra la solicitud de reserva.

Los roles son:

- **Context**: *SolicitudDeReserva*
- **State**: *EstadoDeSolicitud*
- **ConcreteState**: *EstadoDeSolicitudPendiente* / *EstadoDeSolicitudAprobada*

Además, utilizamos este patrón para Reserva, con el objetivo de representar en qué estado se encuentra la reserva y hacer lo correspondiente en cada caso.

Los roles son:

- **Context**: *Reserva*
- **State**: *EstadoReserva*
- **ConcreteState**: *EstadoAprobada* / *EstadoCancelada* / *EstadoFinalizada* / *EstadoCondicional*

Strategy

Utilizamos este patrón para las diferentes políticas de cancelación que puede tener un inmueble, ya que, dependiendo el tipo de cancelación el dueño del inmueble recibirá un resarcimiento diferente.

Los roles son:

- **Context**: *Inmueble*
- **Strategy**: *PoliticaDeCancelacion*

- **ConcreteStrategy**: *CancelacionGratuita* / *SinCancelacion* / *CancelacionIntermedia*

Observer

Se utilizo el patrón para que cuando se realice una baja de precio de un inmueble, la cancelación de una reserva o el alta de una reserva, los suscriptores a los eventos puedan recibir una notificación de lo sucedido.

Los roles son:

- **Subject**: *Manager*
- **Observer**: *Listener*
- **ConcreteObserver**: *AppMobile* / *OtroSitioWeb*
- **ConcreteSubject**: *Reserva* / *Inmueble*