Trabajo Práctico POO2 - 2024 2° Cuatrimestre

Integrantes:

- Maximiliano Borrajo (Maximiliano Borrajo)
- Franco Orizonte (Franco Orizonte)
- Oriana Pettinelli (orianapettinelli04@gmail.com)

Resumen:

El sistema permite registrar propietarios e inquilinos en una clase única **Usuario**, diferenciando sus roles mediante un enum **RolDeUsuario**. La clase **Sistema**, apoyada en varios managers, gestiona los registros y validaciones de usuarios de distintas índole.
Los propietarios pueden publicar propiedades con datos gestionados en las clases **Inmueble** y **Alquiler**, y los precios por temporada se controlan mediante **Periodo**. El patrón ***Composite*** se usa en los filtros (**FiltroSimple**, **FiltroCompuesto**, **FiltroDeSistema**) para búsquedas avanzadas de propiedades.
Las reservas siguen un proceso en dos fases con el patrón ***State***, manejando estados como pendiente, aceptada y rechazada. Los inquilinos pueden ver, cancelar reservas, y el sistema envía notificaciones usando el patrón ***Observer*** para eventos como confirmaciones y cancelaciones.
Para rankings de usuarios y propiedades, **RankingManager** gestiona valoraciones, y los administradores pueden personalizar categorías usando **CustomEnum**. Además, se usa el patrón ***Strategy*** para aplicar políticas de cancelación, permitiendo distintos reembolsos según la elección del propietario. Las reservas en espera se gestionan en un queue, activándose si se libera una propiedad.

Requerimientos del sistema:

- Los propietarios e inquilinos deben registrarse en el sistema.
- Cada usuario debe proporcionar nombre completo, correo electrónico y teléfono al registrarse.
- Los propietarios pueden dar de alta un inmueble en el sitio web.
- Datos requeridos para el alta de un inmueble:
    - Tipo de inmueble (habitación, departamento, casa, quincho, etc.).
    - Superficie en metros cuadrados.
    - Ubicación: país, ciudad, dirección.
    - Servicios (agua, gas, electricidad, baño privado/compartido, calefacción, aire acondicionado, wi-fi).
    - Capacidad del inmueble (número de personas).
    - Fotos (hasta 5 fotos).
    - Horarios de check-in y check-out.
    - Formas de pago aceptadas (efectivo, tarjeta de débito, tarjeta de crédito).
    - Precio diario con posibilidad de variación según el período (temporada alta, fines de semana largos, etc.).
- Los inquilinos pueden buscar inmuebles por ciudad, fecha de entrada y salida, número de huéspedes, y rango de precios (mínimo y máximo).
- Los inquilinos pueden rankear al inmueble y al dueño, y los dueños pueden rankear al inquilino.
- Los rankings son permitidos solo después del check-out y calculados en promedio para cada categoría (dueño, inmueble e inquilino).
- Los usuarios pueden visualizar los detalles del inmueble seleccionado: comentarios, puntaje en distintas categorías y promedio general.
- Los usuarios pueden ver la información del propietario (incluyendo puntaje, historial de alquileres, y tiempo como usuario en el sitio).
- El sistema permite a los inquilinos realizar reservas en dos fases:
    - Fase A: El inquilino realiza la reserva indicando la forma de pago.
    - Fase B: El propietario aprueba la reserva y el sistema confirma la ocupación.

- El sistema envía un correo de confirmación al inquilino una vez consolidada la reserva.
- Los inquilinos pueden ver el estado de sus reservas (todas, futuras, por ciudad, ciudades con reservas).
- Los inquilinos pueden cancelar una reserva, notificando al propietario por correo electrónico.
- El administrador puede gestionar las categorías para puntajes de propietarios, inquilinos e inmuebles.
- El administrador puede agregar tipos de inmuebles disponibles en el sistema.
- El administrador puede definir los servicios disponibles para selección en el registro de inmuebles.
- El administrador puede obtener listados de gestión, como el top de inquilinos que más han alquilado, inmuebles disponibles y la tasa de ocupación.
- Los propietarios pueden definir políticas de cancelación para sus inmuebles:
    - Cancelación gratuita hasta 10 días antes de la fecha de ingreso.
    - Sin cancelación (pago completo a pesar de la cancelación).
    - Cancelación intermedia (gratuita hasta 20 días antes, 50% entre los días 10 y 19, y total después de los 10 días).
- Eventos con notificaciones disponibles:
    - Cancelación de una reserva de inmueble.
    - Confirmación de reserva de inmueble.
    - Baja de precio de un inmueble.
- Los inquilinos pueden realizar reservas condicionales en inmuebles ocupados.
- Las reservas condicionales se encolan y, en caso de cancelación de la reserva actual, la primera en la cola se confirma automáticamente.

Los propietarios e inquilinos deben registrarse en el sistema.

Primero que nada identificamos las diferencias principales entre propietarios e inquilinos, al ver que no había diferencias sustanciales de modo que eso nos hiciera crear dos clases diferenciadas para cada uno, se decidió crear la clase **Usuario**, con los atributos correspondientes, y diferenciar entre inquilinos y propietarios mediante un enum **RolDeUsuario**.

Luego se creó la clase **Sistema**, el cual maneja el registro de los usuarios mediante una composición con la clase **Usuario Manager**, sobre la cual se delega todo el manejo y estado de los usuarios del sistema. En la misma se creó la funcionalidad para registrar usuarios con sus validaciones correspondientes.

A lo largo del proyecto se verá repetida esta acción de delegar funcionalidades a los *managers* con el fin de reducir la responsabilidad del sistema a los requerimientos solicitados y para cumplir con el principio SOLID de *Single Responsibility Principle.*

Los propietarios pueden dar de alta un inmueble en el sitio web.

Para este caso, vimos que en el trabajo práctico se delegaba mucha funcionalidad e información sobre el inmueble. Por eso, tomamos la decisión de separar la información puntual de la clase **Inmueble** (tipo de inmueble, superficie, ubicación, etc), como también englobar en una clase **Ubicación** la información del país, ciudad y dirección ya que pertenecían al mismo tópico. Luego se separó el resto de la información en la clase **Alquiler**, la cual contiene al inmueble que será publicado al sistema e información del precio, checkin, checkout, etc.

Entonces, en el sistema se habilita la posibilidad dar de alta un inmueble, donde se provee la información necesaria y se delega el guardado del alquiler a la clase **AlquilerManager**.

En cuanto a validaciones, las mismas se aplicarán ampliamente a lo largo del proyecto, algunas de manera recurrente, para garantizar que todo se realice conforme a los requerimientos. En un proyecto real, estas validaciones se delegan a una capa distinta y no se implementarían directamente en el modelo. Sin embargo, en este caso hemos decidido incluirlas en el modelo para simular esa capa y asegurar un correcto manejo del dominio y la información.

Por otro lado, se le da la posibilidad al propietario de definir distintos precios para un alquiler en relación a un rango de fechas, con el objetivo de permitir la funcionalidad de establecer precios diferentes al habitual ya sea por temporada alta, fines de semana, feriados o alta demanda.
Para esto un alquiler puede guardar varios periodos, instancias de la clase **Período**, a la cual se le delegan las tareas de almacenar su fecha inicial y final, su precio por día y otras funcionalidades qué luego nos permitirán calcular precios por rango y fechas diferentes. Está misma clase también se usó para poder definir periodos de tiempo en los qué un alquiler no está disponible.

Los inquilinos pueden buscar inmuebles por ciudad, fecha de entrada y salida, número de huéspedes, y rango de precios (mínimo y máximo).

Para este caso, se tomaron varias decisiones, por un lado se crearon clases de **FiltroSimple** y **FiltroCompuesto**, para representar los filtros en general, ambas heredan de una clase **Filtro**. Este filtro simple toma un Predicado (una lambda, que devuelve un booleano), que permite representar todos los filtros posibles con tan solo pasar el predicado correspondiente. Ambos implementan el mensaje cumpleCondicion, donde **FiltroSimple** pregunta en el predicado y el compuesto pregunta si todos sus filtros cumple la condición.

Dando la estructura dada por el patrón de diseño ***Composite***, permitiéndole al filtro compuesto agregar cualquier filtro tipado con la abstracta **Filtro** y a su vez, como filtrarLista(lista) está en la clase abstracta **Filtro** y obliga a implementar cumpleCondicion, que utiliza un ***TemplateMethod***

En la cuestión del ***Composite*** los roles serán:
Nodo o Compuesto: **FiltroCompuesto**

Hoja: **FiltroSimple**
Componente: **Filtro**
Cliente: **FiltroDeSistema**

Aunque el ***TemplateMethod*** es simple se puede ver los roles:
Template Method: filtrarLista(lista)
Operacion Concreta (Definidas en las clases concretas): cumpleCondicion(elementoActual)

Esto se usó como base en todos los filtros, pero a su vez para el caso específico de Filtro de inmueble, se creó el objeto **FiltroDeSistema**, el cual en la creación se pasa la ciudad y las fechas, obligando asi el uso de esos filtros. Luego se crea un **FiltroCompuesto** y se agregan filtros simples correspondientes a cada uno de los anteriores (ciudad y las fechas). Para los filtros opcionales se agregan a este objeto a través de mensajes, añadiendo así el **FiltroSimple** correspondiente al colaborador interno **FiltroCompuesto** del **FiltroDeSistema**. Esta solución nos brinda que si se agregan filtro opcionales nuevos no escale en base a objetos específicos sino a mensajes en el filtroSistema, a su vez nos permite usar **FiltroSimple** en distintas situaciones dentro del sistema.

El sistema permite a los inquilinos realizar reservas en dos fases

Al sistema se le avisa que se intenta crear una reserva, para esto verifica si está el periodo disponible, y si la forma de pago es válida. Si estas dos son verdaderas, se crea una reserva con el estado pendiente, el cual espera que el propietario acepte o rechace. Al hacerlo se cambia el estado de la reserva, junto con sus implicaciones que cada estado conlleve.

Para esto se utilizó el patrón State, donde los roles son:
Estados concretos: **Pendiente**, **Aceptada**, **Rechazada**, **Cancelada** y **Finalizada**.
Estado: **EstadoReserva**
Contexto: **Reserva**

Los inquilinos pueden ver sus reservas (todas, futuras, por ciudad y todas las ciudades reservadas).

Para este requerimiento se utilizó los filtros genéricos anteriormente mencionados, para luego utilizar una herencia de **FiltroReserva**, con cada uno de los filtros como clase concretas de la misma, cada filtro concreto, donde por dentro tiene un **FiltroSimple** correspondiente a cada uno con su predicado, permitiéndole filtrar, y delegando en el mismo.

Para la parte de ver todas las ciudades de las reservas, se decidió elegir usar un mensaje aparte para esa cuestión en particular, ya que no es un filtro en sí.

Los inquilinos pueden cancelar sus reservas notificando al propietario por correo electrónico.

Los inquilinos tienen la posibilidad de cancelar la reserva avisando al sistema, cuando la misma se efectúe gracias a los estado posibles, se le mande un mail al propietario. En otros casos que la reserva no se pueda cancelar por el estado automaticamente lanzará una excepción.

Ranking y visualización.

Para esto primero se creó la clase **Ranking** la cual posee la información necesaria para poder obtener las valoraciones, promedios y comentarios. La misma posee quien es que el realiza la valoración, siendo según el dominio siempre un usuario, luego el valorado es un **Rankeable**, interfaz la cual se establece aquellas entidades que pueden ser valoradas. La misma obliga a establecer qué condiciones tiene que tener el usuario que realiza la valoración para poderse establecer en el sistema. Luego el sistema se encarga de dar la posibilidad de añadir una valoración, ver promedios por categoría, ver comentarios, etc. la cual delega al **RankingManager**, que se encarga de validar y almacenar las valoraciones.

Además para representar las categorás se crea el **CustomEnum Categoría**, el cual es uno de los otros enums modificables por un administrador en el sistema. De esto mismo se explicará más adelante en el informe.

Otros de los requerimientos, y por el cual se realizó lo explicado en los párrafos anteriores, es poder ver una **Visualizacion** de un inmueble seleccionado, para ello se creó una clase con dicho nombre. La misma requiere al inmueble y al sistema para poder realizar todos las acciones que se le soliciten. Se creó esta clase con el fin de englobar todos los requerimientos de visualización en un solo lugar.

El sistema envía un correo de confirmación al inquilino una vez consolidada la reserva.

Para poder simular el envío de un email al concretarse, o cancelarse una reserva, lo que se hizo fue crear una interfaz **MailSender** la cual permite enviar un mail. Esta interfaz es luego utilizada por el sistema y que según el caso se provee a los managers que la necesiten para poder enviar los mails pertinentes. En particular, al cancelar una reserva o aceptarse, si no ocurre ningún fallo, se envía el mail a quien corresponda al finalizar la acción.

Administración del sitio.

Como se explicó al principio del informe se crearon roles de usuario, lo cual permite diferenciar acciones posibles para distintos tipos de usuario. Para este caso, los requerimientos requieren un nuevo tipo de rol administrador.

Estos administradores tienen varias funciones puntuales como gestionar categorías, tipos de inmueble y servicios. Para estos tres se crearon clases concretas: **Categoría**, **TipoDeInmueble,** y **Servicio,** que heredan de **CustomEnum**.

La existencia de una clase como los **CustomEnum** nos habilita la posibilidad de tener objetos con comportamiento parecido a un enum, pero qué se pueden crear dinámicamente. Para qué se pueda validar a qué tipo de **CustomEnum** pertenece una clase, se incluyeron los enums **CustomEnumType**.

Una vez establecido eso, por medio de delegar al **CustomEnumManager**, se provee desde el sistema el dar de alta alguno de estos tres **CustomEnum** mencionados, solo a usuarios administradores.

Así mismo el sistema provee la posibilidad de ver el top de inquilinos que más alquilan, ver los inmuebles disponibles, ver la tasa de ocupación, etc.

Para realizar estas operaciones el sistema ya contaba con los managers necesarios para cada requerimiento, pero para el caso del top de inquilinos se creó la clase **Podio** con el fin de mostrar de forma sencilla cual es la cantidad de reservas hechas por un inquilino puntual.

Eventos con notificaciones disponibles: Cancelación de una reserva, Confirmación de reserva, Baja de precio de un alquiler.

Para solucionar este requerimiento, se creó el **NotificadorManager**, el cual tiene una lista de **Listener**. Los listener se crean cuando son necesarios, es decir cuando alguien se quiere suscribir a un evento. En el caso que el **EventoNotificador** ya existe, se agrega el **Suscriptor** al **Listener** correspondiente. Cuando el **NotificadorManager** le llega un notify con su **EventoNotificador**, éste buscará al primer listener que sea de ese evento y notificará a todos los suscriptores ahí.

Para hacer esto se creó la clase abstracta **EventoNotificador**, con subclases de cada evento donde se delega el mensaje específico se quiere que le llegue al suscriptor en las mismas, los observables implementan la interfaz **Observable**, para así pasarlos al evento en su creación y saber a que reserva o alquiler se desea suscribir.

La interfaz **Suscriptor** es la cual debe implementar quien desea suscribirse. Esto obliga a implementar los tres mensajes, los dos específicos pedidos por el enunciado y el update, esta decisión tiene la ventaja de que puede tratar a todo suscriptor por igual y luego el mensajes que le llega delegarlo a otro objeto, además potencialmente los suscriptores se puede suscribir a distintos eventos sin cambio algunos.

Se utiliza el patrón ***Observer***, el cual los suscriptores van a ser los que implementan la interfaz. Los roles son:

Observador: Interfaz Suscriptor.

Observador Concreto: en ese caso trivago y app mobile, dejando la posibilidad a cualquier otro que implemente la interfaz.

Sujetos: NotificadorManager.

Gestor de cambios: Listener.

Observables: quienes implemente la interfaz Observable; Alquiler y Reserva.

Eventos: EventosNotificador.

Reservas condicionales

En particular para este caso lo que se hizo fue cambiar la lógica que tenía el **Sistema** y **ReservaManager.** Ahora se indicará si la reserva que se está creando es condicional o no. Eso se hizo así porque se dice que el *inquilino podrá encolar su reserva*, lo que significa que no ocurre siempre y es opcional. Luego lo que se hizo fue agregar en la clase **Alquiler** un *queue* que provee Java de reservas. Con esto lo que se hace es que en el caso que el usuario quiera un reserva condicional, si la reserva que está realizando sucede sobre un periodo donde el alquiler ya está ocupado, la reserva nueva se encola en el queue de reservas, sino, se crea y registra en el sistema de forma normal.

Luego, si se cancela una reserva, lo que se hace es verificar si el alquiler de dicha reserva posee reservas encoladas, si las hay, se toma el primero en la cola y se registra en el sistema para que luego el propietario pueda aceptarla o rechazarla, es decir, que siga el flujo normal. En caso de no haber, no ocurre nada.

Politicas de cancelacion

Cada propietario debe poder definir una política de cancelación para su alquiler dado de alta, las cuales pueden ser: gratuita, intermedia o sin cancelación.

Para permitir está funcionalidad, se utilizó un patrón ***Strategy*** con lo siguientes roles:

Strategy: interfaz **PoliticaDeCancelacion**

Clases concretas: **CancelacionGratuita**, **CancelacionIntermedia** y **SinCancelacion**.

Contexto: **Alquiler**

De está manera un alquiler puede almacenar una **PoliticaDeCancelacion** y compartir está información con sus reservas, permitiendo al usuario conocer el monto del reembolso qué recibiría si este fuera a cancelar una reserva.