Informe Trabajo Integrador

Programación con Objetos II
2º Cuatrimestre de 2024

Alumnos:	Emails:
Fernandez Requejo, Cristian	cristian.adrian.f.r@gmail.com
Rodriguez Fontana, Abril	Arodriguezfontana@gmail.com
Salvanescki, Nicolás	salvanescki.unq@gmail.com

Índice

1. Introducción

El presente trabajo tiene por objetivo diseñar e implementar un modelo de sitio web que conecta de forma directa a propietarios particulares que desean ofrecer sus inmuebles para alquiler temporal y a potenciales inquilinos interesados en alquilar. Dicho modelo fue pensado para ser mantenible y escalable, utilizando las mejores prácticas del paradigma orientado a objetos, enseñadas en la materia Programación con Objetos II.

2. Diseño

En lo que respecta al diseño decidimos, basándonos en la consigna, que los usuarios sean capaces de alquilar y también de poner un inmueble en alquiler, o lo que nosotros llamamos publicar. Esto se traduce inmediatamente en una clase Usuario que puede ser tratada como Inquilino y/o Propietario, siendo estas dos últimas, interfaces.

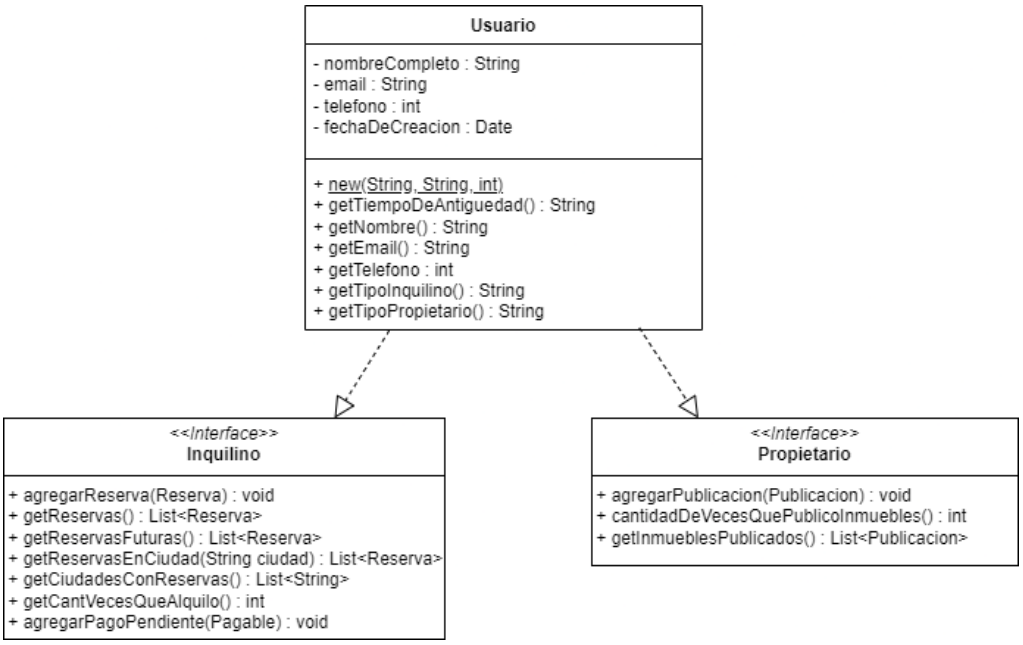

Figura 1: Relación Usuario con Inquilino y Propietario.

Cuando un Usuario desea dar de alta un inmueble en el Sitio, debe introducir una gran cantidad de datos sobre dicho inmueble. Estos van estar alojados en las variables de instancia de la clase Publicacion.

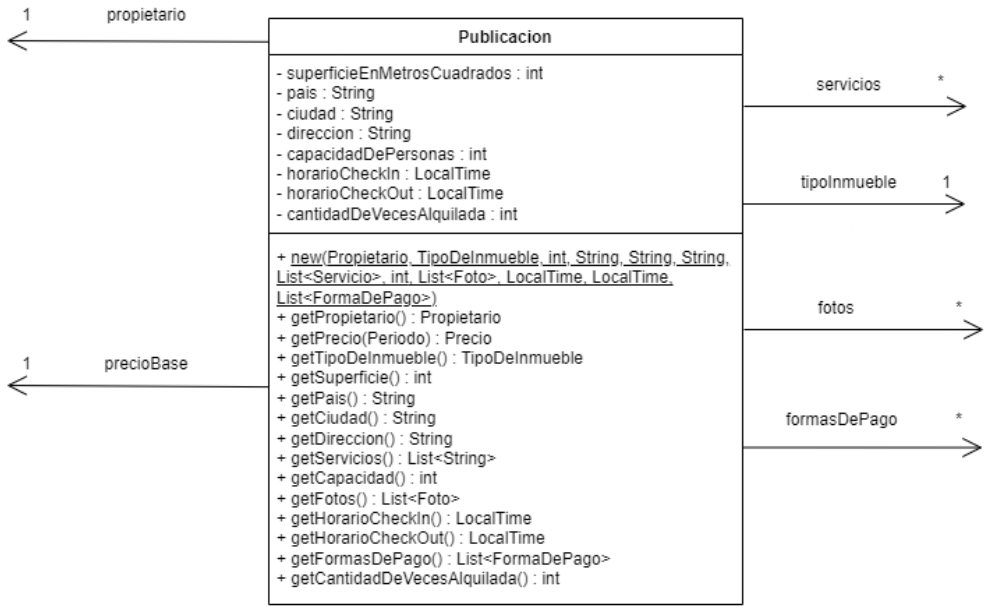

Figura 2: Versión básica de Publicacion (solo constructor y getters).

En el caso de las fotos, al no estar ni implementando el manejo de archivos del File System, ni consumiendo una API que nos traiga un BLOB o algo similar, preferimos dejarla como una clase vacía que solo existe para que se pueda crear la Lista de Fotos en Publicacion.

Las formas de pago, por otro lado, en un principio las habíamos pensado como un tipo enumerativo (enum), pero eso nos restringía si queríamos, en un futuro, extender la cantidad de formas de pago, adoptando nuevas tecnologías que antes no estaban (mercado pago, cripto, etc). Por lo que, al final, nos terminamos inclinando por crear una clase que tenga un nombre que identifique la forma de pago y que pueda saber si es igual, o no, a otra forma de pago redefiniendo el método de Object **equals**.

Respecto a los servicios y tipos de inmueble, la idea fue exactamente la misma que con las formas de pago. Esto se debe a que el administrador tiene la capacidad de agregar servicios, categorías y tipos de inmueble al Sitio web. Si bien las Categorías de Ranking aún no las presentamos, tienen la misma estructura que las clases anteriormente mencionadas. A continuación se muestra dicha estructura:

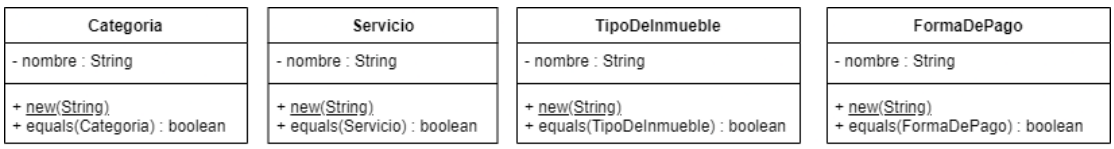

Figura 3: Tipos “enumerativos extensibles”

Para evitar problemas de igualdad, al tratarse nombre de un String, en el constructor de cada una de estas clases “formateamos” el String, para que quede en minúsculas, sin espacios, sin símbolos, con solo letras (y tal vez, números).

2.1. Búsqueda

Las búsquedas las pensamos inicialmente como un patrón Composite, ya que alguien, por ejemplo, podría querer buscar un alquiler en "Quilmes" con fecha desde el 1/12 hasta el 15/12 **o** en "Berazategui" con fecha desde el 10/12 al 15/12. Es decir, que para una misma Búsqueda, los filtros podían tener casos disyuntivos.

Por lo que, si hubiesemos implementado un Composite tendríamos que haber hecho un filtro compuesto por OR y AND (como en una query).

Sin embargo, al no tener los casos donde se tenga que implementar la disyunción lógica (OR), sino solo casos de conjunción (AND), preferimos optar por el patrón Builder. La decisión se basa principalmente en que hay 3 filtros obligatorios y 3 filtros opcionales. Los obligatorios serían modelados como parámetros del constructor de la Búsqueda, mientras que los opcionales pueden incluirse aprovechando las ventajas del patrón de diseño.

El patrón Builder permite, mediante el envío de mensajes (uso de métodos), ir estableciendo valores que pueden, o no, estar presentes al finalizar la configuración de la Búsqueda. Por razones de comfort y sugar syntax, en los métodos devolvemos la instancia luego de setear los valores opcionales. Por ejemplo:

```
public Busqueda conCantidadDeHuespedes(int cantHuespedes){
    // Establecemos el valor (this.cantHuespedes = cantHuespedes;)
    return this;
}
```

Esto permite, al construir una búsqueda, hacerlo de la siguiente manera:

```
new Busqueda("Quilmes", "2024-05-16", "2024-05-23")
            .conCantidadDeHuespedes(4)
            .conPrecioMinimo(1000000.00)
            .conPrecioMaximo(99999999.99);
/*
Quitando el hecho que el tipo de las fechas aca es String cuando lo hicimos como
    LocalDate, y lo mismo con los double de los precios (Precio).
*/
```

Esto es muy similar a la forma en la que se usan los streams en Java y también lo aplicamos en otra clase, esta vez de utilidad, no necesaria para el trabajo: **Precio**.

Entrando en detalles, la busqueda guarda los filtros opcionales en una lista de **Predicate<Publicacion>**. A la hora de aplicar dichos filtros, para cada uno que esté en la lista, se va a aplicar un filter sobre un stream de la lista ya filtrada por los obligatorios. Esto devuelve, al finalizar el loop, la lista filtrada tanto por los filtros obligatorios como por los opcionales.

2.2. Precio (Utilidad)

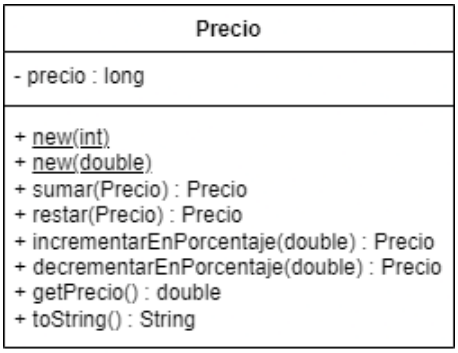

Figura 4: Clase Precio

Como puede notarse en la Figura 4, los métodos que operan con el Precio, devuelven un Precio. Esto brinda la posibilidad de encadenar las operaciones para, al final, devolver el resultado en un número con getPrecio. Por ejemplo:

```
double costo = new Precio(20000.25)
                        .sumar(new Precio(30))
                        .incrementarEnPorcentaje(21)
                        .decrementarEnPorcentaje(10)
                        .restar(new Precio(50.5)
                        .getPrecio();
```

Esto es extremadamente útil para calcular intereses y descuentos, dado un monto. Además de tener la ventaja de no perder precisión en las operaciones porque se usan números enteros (long) en vez de punto flotante.

Por último, aunque excede los conocimientos que se enseñan en Programación con Objetos II, nos aseguramos que las operaciones sean funciones puras. Esto es que el comportamiento de estos métodos tiene dos cualidades: primero que no cambian el estado de lo que están operando, sino que crean una copia que devuelven; y segundo que no dependen de valores aleatorios, por lo que ante la misma entrada de datos, devuelven la misma salida siempre.

Creemos que valió la pena resaltar la forma en la que pensamos la clase Precio, ya que se acerca a las prácticas de programación funcional que están teniendo lugar en la industria del software actualmente.

2.3. Ranking

El sistema de calificaciones que modelamos tiene tres casos a tomar en cuenta:

- Inquilino califica un Inmueble (Publicacion) que alquiló previamente.
- Inquilino califica al Propietario de un Inmueble que alquiló previamente.
- Propietario califica a un Inquilino que le alquiló un inmueble.

Todas estas calificaciones se hacen, luego del check-out, sobre una lista de Categorias que se encuentran disponibles en el Sitio web. Dichas Categorías se dividen en tres, una por cada tipo de entidad Rankeable (Inquilino, Propietario, Publicacion).

Los ranking (o calificaciones) los modelamos de la siguiente forma:

Figura 5: Clase Ranking

Como puede notarse en la figura 5, hay dos métodos estáticos a la clase Ranking. El primero es un setter para el sitio web, ya que Ranking debe tener una referencia al sitio para poder validar si una categoría de ranking que le están pasando es válida. El segundo es el validador que, dado un Ranking y un String que representa el tipo de Rankeable, chequea si el ranking tiene un puntaje y categoría válidos.

El check-out, por otro lado, se valida dentro de la clase que está recibiendo el ranking. Esto fue una decisión de implementación que se justifica en que: en dicha clase (sea un Inquilino, Propietario o una Publicacion) existe siempre una manera de chequear si se realizó, o no, el check-out. Si tuvieramos que validar el check-out estáticamente en la clase Ranking podría violarse el encapsulamiento, acoplarse demasiado las clase Ranking con las Rankeables o podríamos tener que sobrediseñar el código para lograr no romper las anteriores (agregando complejidad a Ranking y al diseño en general).

Las clases Rankeables, a su vez, implementan la siguiente interfaz:

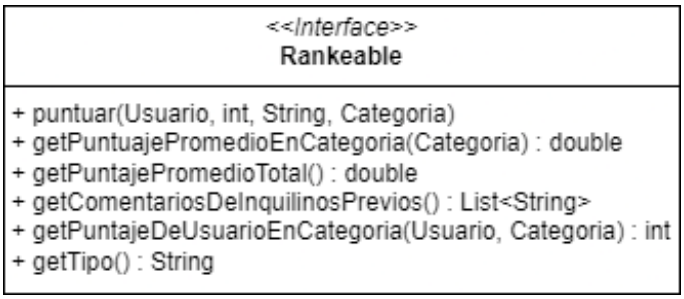

Figura 6: Interfaz Rankeable

Es decir que nuestro programa, por cada clase rankeable, nos permite calcular el puntaje promedio por categoría y el promedio total, obtener comentarios de inquilinos previos (en caso que sea una Publicacion o un Propietario al que se esté preguntando) y obtener el puntaje que puso un Usuario específico en una categoría.

2.4. Sitio Web

Como se mencionó brevemente en la sección anterior (2.3), las categorías de ranking varían para cada tipo de Rankeable. Para esto, en primer lugar decidimos aplicar un patrón Strategy para cada tipo de categoría distinta como se ve a continuación:

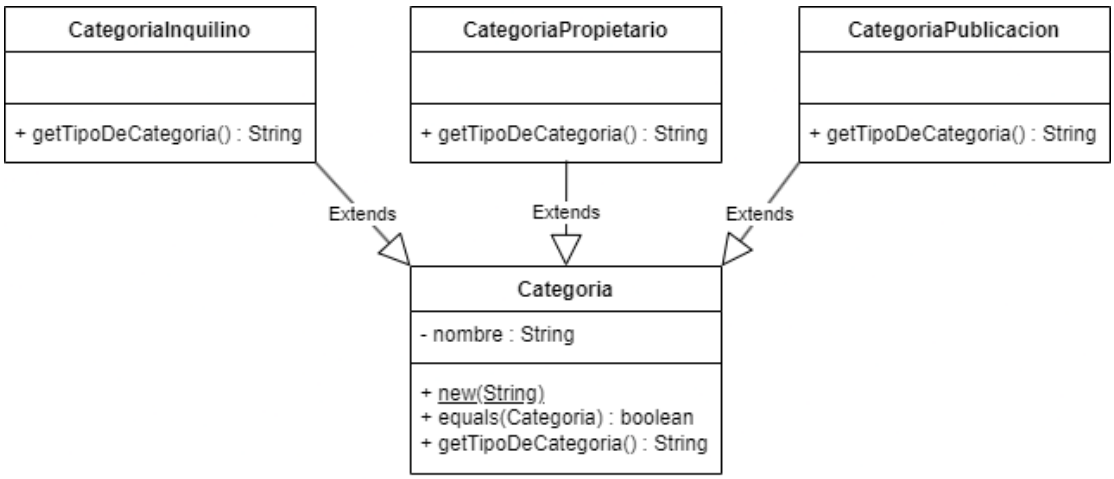

Figura 7: Clases Categoria

La razón de implementar estas subclases es que puedan responder polimórficamente al mensaje **getTipoDeCategoria**, el cual devuelve con un String que tipo de categoría son (Ej. Ïnquilino", "Propietario.º "Publicacion").

Esto permite al SitioWeb preguntar por el tipo de la categoría y compararlo con el tipo del rankeable. Si estos coinciden, entonces la categoría es válida para ese Rankeable; sino, no lo es.

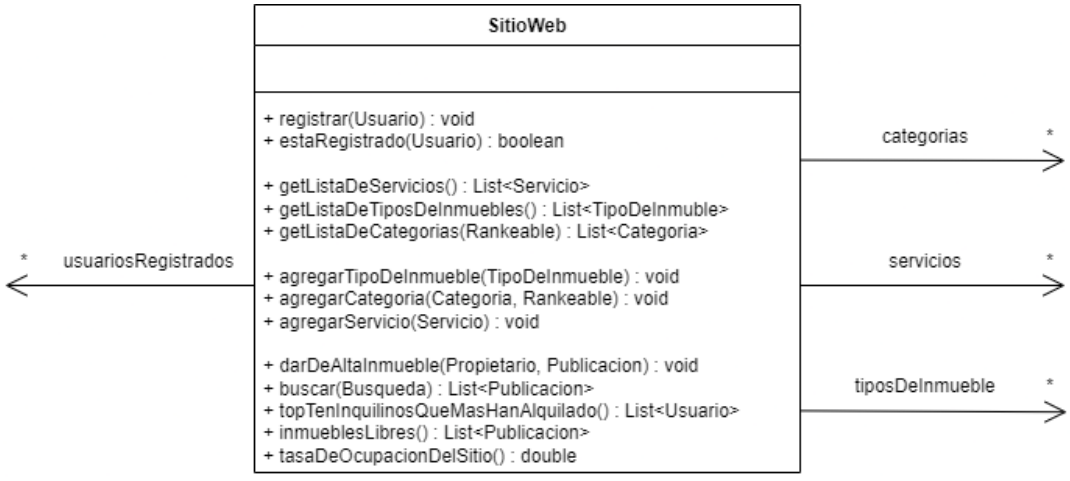

Figura 8: Clase SitioWeb

Según la figura 8, el SitioWeb permite:

- Registrar usuarios
- Preguntar si un Usuario ya está registrado
- Obtener las listas de servicios, tipos de inmueble y categorías, válidos en el sitio.
- Agregar servicios, tipos de inmueble y categorías nuevas (Administrador)
- Dar de alta un inmueble (Propietario)
- Buscar publicaciones indicando ciertos filtros (Inquilino)
- Obtener listados de gestión como los diez inquilinos que más alquilaron, los inmuebles libres, la tasa de ocupación del sitio, etc.

2.5. Periodo

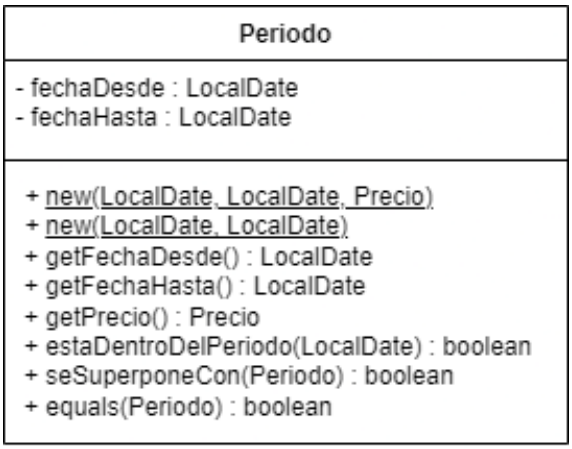

Figura 9: Clase Periodo

El Periodo es, en términos simples, un rango de fechas que puede, o no, tener asociado un Precio. Esto permite a las demás clases, establecer ciertos periodos que tengan un precio especial, distinto al precio base, como es el caso de Publicación. En esta última, el Propietario es capaz de definir Periodos de tiempo con un precio distinto y mediante el mensaje **getPrecio(Periodo)** de Publicacion, se puede obtener el precio para cada fecha de un Periodo pasado por parámetro (Si pertenece a un período especial tiene ese precio, sino el precioBase de la Publicacion).

Entre la lógica más interesante que tiene para ofrecer está: puede detectar si se superpone con otro Periodo pasado por parámetro, esto es si coinciden en al menos un día; puede indicar si una fecha pertenece al Periodo y puede, mediante la redefinición del equals de Object, saber si es igual a otro Periodo (si coinciden sus fechas inicio y fin, el precio no nos importa).

2.6. Reserva

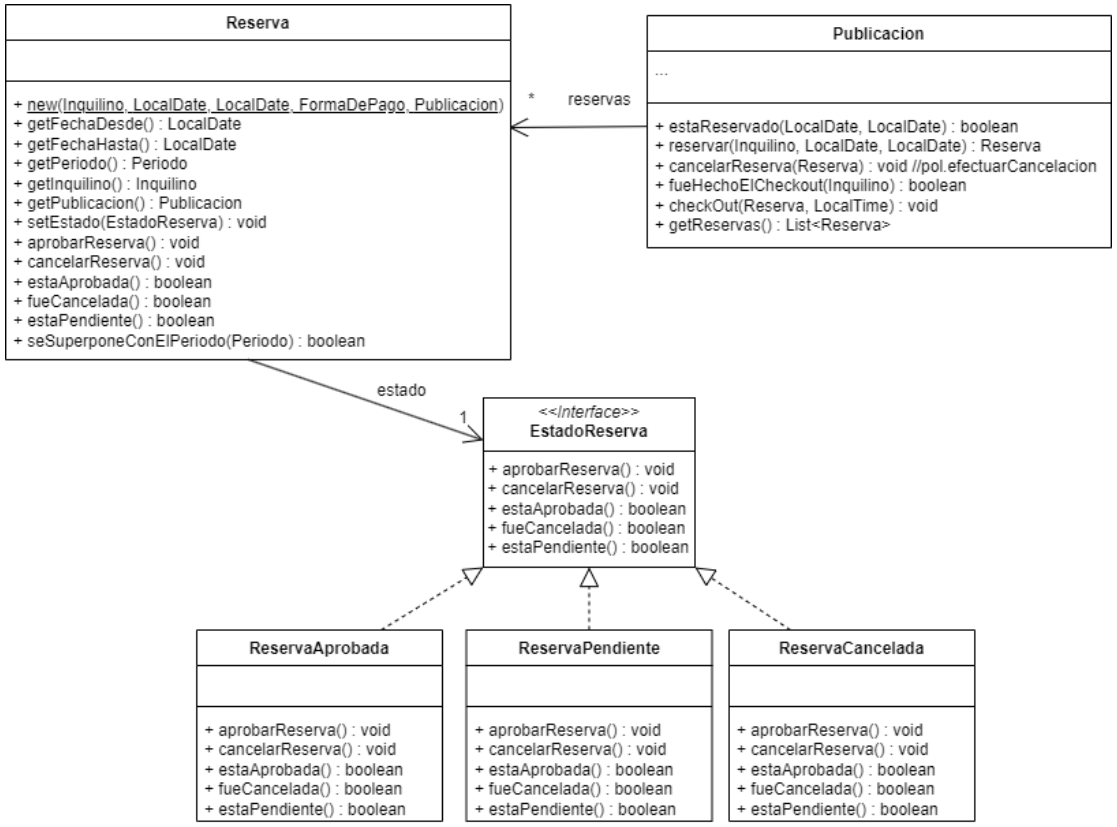

Figura 10: Clase Reserva

Las Reservas tienen un Periodo asociado, lo cual permite, entre otras cosas, saber si una Publicacion está reservada y/o alquilada en un Periodo determinado. En la figura 10 se muestra que Publicacion tiene una lista de reservas y varios mensajes que interactúan de una forma u otra con la clase.

En lo que a diseño respecta, utilizamos un patrón State para manejar los estados de una Reserva. Como puede verse a simple vista en la figura 10, EstadoReserva es una interfaz implementada por 3 clases concretas: ReservaAprobada, ReservaPendiente y ReservaCancelada. Estas clases son los posibles estados en los que puede encontrarse una reserva.

Al inicializar una reserva, su estado predeterminado es ReservaPendiente, ya que aun debe ser aprobada por el Propietario o cancelada por el Inquilino. Cuando ocurre alguno de estos eventos, el estado llama a **Reserva.setEstado()** y define el nuevo estado en el que estará la reserva (aprobada si se aprueba, cancelada si se cancela).

Los casos en donde se aprueba una ReservaAprobada o se cancela/aprueba una ReservaCancelada, lanzan una excepción de tipo OperacionInvalidaConEstadoReservaException indicando que hizo mal el usuario de la Reserva por medio del mensaje de excepción.

En caso de que el Propietario apruebe la reserva, se manda un mail al Inquilino

avisando de dicho evento. Si el Inquilino cancela la reserva, el Propietario también recibe un mail que advierte sobre dicha cancelación.

2.7. Politicas de Cancelación

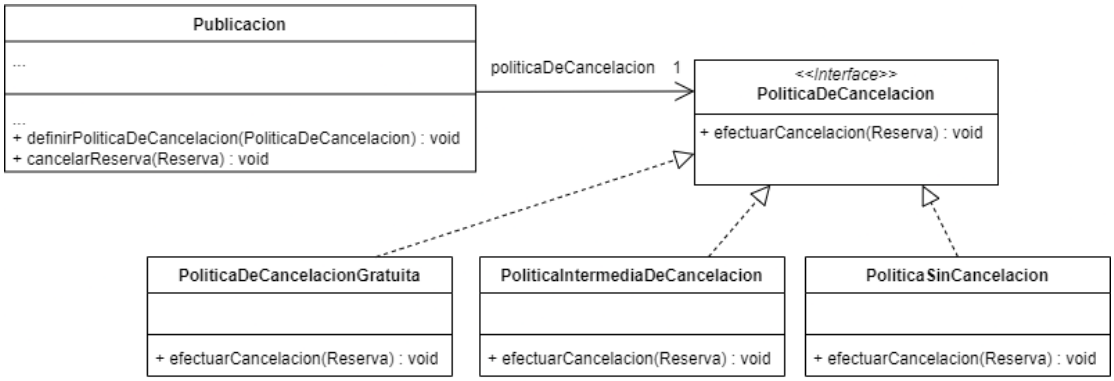

Figura 11: Clases PoliticaDeCancelacion y su relación con Publicacion

Para las políticas de cancelación decidimos implementar otro patrón Strategy. Todas las políticas responden al mensaje **efectuarCancelacion(Reserva)** que es enviado desde **cancelarReserva(Reserva)** en Publicacion.

A su vez, la estrategia, o política, a utilizar se define con el mensaje **definirPoliticaDeCancelacion(PoliticaDeCancelacion)** de Publicacion, el cual puede utilizar el Propietario de la misma para poder elegir la adecuada a sus gustos.

Aunque ya está explicado en el enunciado, vale la pena recordar que implica cada tipo distinto de politica y, de paso, detallaremos un poco la forma en que lo implementamos.

Para facilitarnos un poco el trabajo, creamos una clase Deuda que implementa una interfaz Pagable. También, a Usuario le creamos una List<Pagable>que tenga las deudas pendientes. Entonces, cuando se efectua la política de cancelación, se agrega a dicha lista lo que el usuario deba pagar a futuro.

2.7.1. Cancelacion gratuita

La política más beneficiosa para el Inquilino, ya que le permite cancelar de manera completamente gratuita siempre que lo haga hasta 10 días antes de iniciada la fecha de ocupación. Si se pasa de dicha fecha, tendrá que pagar el equivalente a 2 días de la reserva.

2.7.2. Sin Cancelación

Esta política beneficia completamente al Propietario, perjudicando en el proceso al Inquilino. Simple y llanamente, el Inquilino debe pagar la totalidad de los días reservados como si se hubiese hospedado en el lugar.

2.7.3. Intermedia

La política intermedia, como su nombre lo indica, es la más justa para ambas partes. Si el Inquilino cancela la reserva hasta 20 días antes de la fecha de ocupación, es gratis. Si la cancela entre el día 19 y el día 10 anterior, debe abonar un 50 % del precio total. Posterior a eso, paga la totalidad de los días reservados.

2.8. Notificaciones

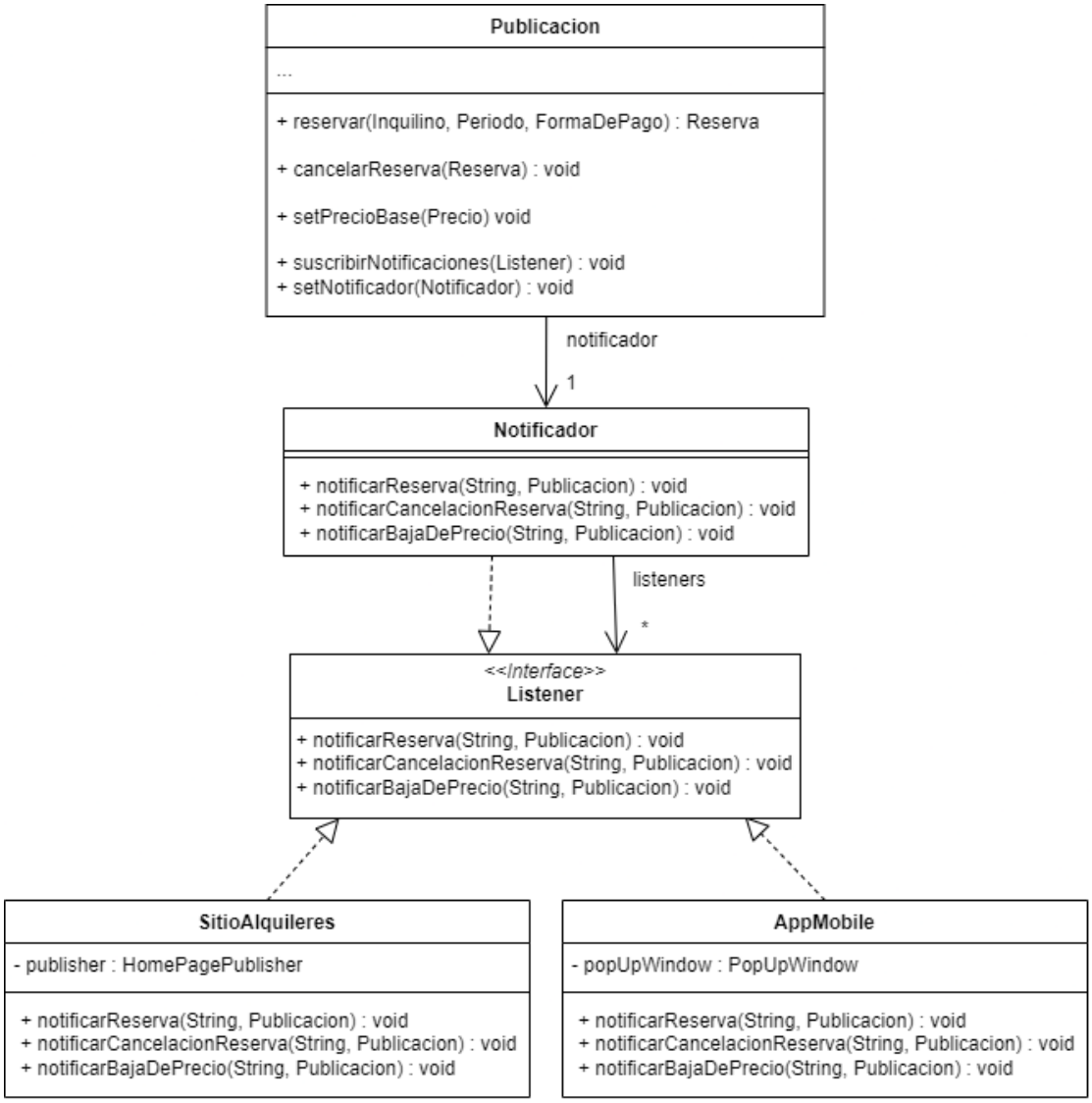

Figura 12: Clases Listener y su relación con Publicacion

Para notificar los eventos que ocurren en el sistema elegimos implementar un patrón Observer con Listeners. La razón detrás de nuestra elección fue que era muy posible que en el futuro, dada la naturaleza del sistema, se requiera de notificar de eventos distintos a los actuales. Los Listeners nos permiten dejar el diseño lo

suficientemente flexible para adaptarse a estos nuevos cambios.

Actualmente se notifica de cancelación de reservas, reservar y bajas de precio, y se llama al notificador para avisar a los listeners de estos eventos en los métodos de Publicación **cancelarReserva**, **reservar** y **setPrecio** (cuando el precio baja) respectivamente.

Se mandan los mensajes descritos en la consigna, de la manera que se indica allí también.

2.9. Reserva condicional

En dicho asunto decidimos no diferenciar los tipos de Reserva en Reserva y ReservaCondicional, sino que preferimos tratarlas igual y que lo que cambie es el comportamiento de Publicacion al estar en dicho contexto. Si un Usuario reserva en un Periodo que la Publicacion ya está reservada, su reserva quedará pendiente y en la lista de reservas de la Publicacion.

Ahora la cuestión es la siguiente: ¿Qué diferencia una Reserva Condicional de una normal?, la respuesta son dos cosas:

- Ambas están pendientes en su creación pero la condicional no puede ser aprobada hasta que la reserva actual sea cancelada, mientras que la otra si.
- La Reserva Condicional coincide o se superpone en Periodo con la reserva actual, mientras que una reserva normal, salvo que sea la "actual" no se superpone con otra de la lista.

Por lo tanto, con que haya un método que busque la primera ocurrencia de Reserva (Pendiente) que coincida con la "actual" (recién cancelada), devolvería la reserva condicional. Esto se puede asegurar porque la ArrayList<Publicacion>se van agregando en orden de llegada las reservas, por lo que funciona como una cola si la recorres desde el principio.

El único problema que capaz vimos con esta solución, pero que no llegamos a probarlo es qué pasa con las reservas condicionales de la lista cuando se aprueba la reserva actual en vez de cancelarse. Como no se eliminan de la cola quedan ahi, eso podría generar un problema.

NOTA: Si creen que debemos cambiar eso, no tenemos ningún problema en pensarlo y hacerlo para una re-entrega.

3. Roles (Gamma et. al.)

En esta sección se va a hacer un resumen de los patrones que utilizamos, definiendo los roles de cada participante de los mismos según el libro Gamma et. al. (1994) *Design Patterns, Elements of Reusable Object Oriented Software*

3.1. Builder (Busqueda)

- Director: Hay dos posiciones, una es que la misma clase Busqueda es el director, ya que opera con ella misma para construir la Busqueda. La otra es que el Director es el Cliente que construya la Busqueda instanciándola y concatenando sus métodos que agregan filtros.
- ConcreteBuilder: la clase Busqueda
- Product: la List<Publicacion>que devuelve efectuarBusqueda, ya que es lo que se fue filtrando con los filtros construidos.

3.2. Strategy (Categoria y PoliticaDeCancelacion

En el caso de Categoria los roles son:

- Strategy: Categoria, aunque sea clase concreta igual define el mensaje al que van a responder polimórficamente las subclases.
- ConcreteStrategy: CategoriaInquilino, CategoriaPropietario, CategoriaPublicacion
- Context: SitioWeb, ya que es la clase que referencia a la Strategy.

Por otro lado, en PoliticaDeCancelacion los roles se describen a continuación:

- Strategy: PoliticaDeCancelacion
- ConcreteStrategy: PoliticaDeCancelacionGratuita, PoliticaIntermediaDeCancelacion, PoliticaSinCancelacion
- Context: Publicacion, ya que puede desde ahi se cambia la ConcreteStrategy que se usa, efectuarCancelacion se llama también desde ahí y tiene siempre la referencia a la Strategy.

3.3. State (Reserva)

- Context: Reserva
- State: EstadoReserva
- ConcreteState: ReservaAprobada, ReservaPendiente ReservaCancelada

No hay mucho que explicar, Reserva tiene una instancia de ConcreteState que responde polimórficamente a los mensajes de EstadoReserva y cada ConcreteState tiene formas de cambiar el estado de Reserva.

3.4. Observer (Notificador)

- Subject: Notificador, ya que conoce a sus observers y tiene métodos para agregar o eliminarlos.
- Observer: Listener, ya que es quien ofrece la interfaz de updating, es decir los mensajes por los cuales se va a notificar a los Observers.
- ConcreteSubject: Notificador/Publicacion, ya que Notificador solo avisa a la lista de Listeners del cambio de estado de Publicacion pero, a su vez, es quien guarda la lista, mientras que publicacion no lo hace.
- ConcreteObserver: Todas las clases que implementen Listener y se suscriban a las notificaciones de Publicacion.

NOTA: Publicacion también tiene un método para suscribir listeners, pero lo único que hace es notificador.suscribe(Listener), por lo que realmente no estaría cumpliendo la función de Subject, mientras que Notificador si.

4. Conclusión

El presente trabajo tuvo por objetivo diseñar e implementar un modelo de sitio de alquileres de manera mantenible, flexible y escalable, utilizando las mejores prácticas que se enseñaron en la materia Programación con Objetos II.

A lo largo de este informe, estuvimos repasando los patrones que utilizamos, las decisiones de diseño que tomamos y su justificación, algunos detalles de implementación que valían la pena explicarse, aunque sea en lenguaje coloquial. También, definimos los roles según el libro de Gamma et. al. y dimos un vistazo general a lo que fue el trabajo.

En resumen, aplicamos muchos conocimientos de la materia en este trabajo y esperamos sea lo que se esperaba de nosotros, ya que pusimos mucho esfuerzo de nuestra parte para que quede un, si se quiere, software de calidad según el paradigma orientado a objetos.

Agradecemos enormemente el trabajo de corregir de los profesores y deseamos buena suerte a los demás grupos. Muchas gracias.