# Trabajo Práctico: Sistema de alquileres

## Mlembros:


- Contardo Juan Pablo : contardo.juanpablo@gmail.com
- Aduco Matias : matias.aduco@gmail.com
- Villalba Joaquin : joaquinvillalba8@gmail.com


## Publicación y alquiler

Al comienzo del modelado partimos de cosas que eran similares para nosotros, no tanto como programadores, sino como usuarios consumidores de distintas plataformas web, ya sean mercadolibre, trivago, etc. Los requerimientos para este ítem especificaban la necesidad de poder hacer que en un sitio web se pueda  dar de alta distintos inmuebles puestos en alquiler. Para poder hacer esto el usuario debe especificar una serie de datos.

Propusimos entonces la siguiente división:

- Tipo de Inmueble (habitación, departamento, casa, quincho, etc).
- Superficie: medida en metros cuadrados
- País
- Ciudad
- Dirección
- Servicios con los que cuenta (agua, gas, electricidad, baño privado/ compartido, calefacción, aire acondicionado, wi-fi).
- Capacidad: cantidad de personas que pueden alojarse allí.
- Fotos: hasta 5 fotos del inmueble.
- Horario de check-in y check-out: El horario de check-in indica a partir de qué momento del día es posible hacer uso de la reserva. Por su parte, el horario de check-out indica el tiempo límite para dejar una habitación el último día.
- Formas de pago aceptadas. Éstas pueden ser efectivo, tarjeta de débito y tarjeta de crédito.
- Precio:el precio del inmueble por día. Notar que el precio puede variar

La sección marcada con color **verde** hacía referencia a datos que la clase Inmueble podría tener, mientras que, la parte roja hacía referencia a datos que una Publicación podía tener.
Fue así que esta división surgió pensando en las futuras implementaciones que tendría y justificamos su división gracias al capítulo 3 - **Bad Smells** in Code del libro de **Refactoring**. Existe un apartado llamado **Large Class**, en el cual se sostiene que *"Cuando una clase intenta hacer demasiado, a menudo aparece como demasiadas variables de instancia. Cuando una clase tiene demasiadas variables de instancia, el código duplicado no puede quedarse atrás"***.** Sumado además  la delegación de responsabilidades, sin perder el foco en que el inmueble no es más que un objeto parte de un sitio web y no su mera representación física.

Patrón implementado para la Publicación:
Durante el modelado de la clase publicación, notamos que esta tendría distintos comportamientos dependiendo de si la misma estaba (Disponible, Reservada y Finalizada). Por esto decidimos implementar el patrón State, ya que de esta forma dependiendo el estado actual, la misma desarrolla distintas acciones llevadas a cabo por el estado.

# Usuario

Esta clase engloba al Inquilino y al Propietario, como el Usuario puede ser tanto Inquilino como Propietario, creímos que lo mejor es tener todo en una misma clase.

# Búsqueda de Inmuebles

Para la búsqueda de publicaciones con distintas condiciones se implementa dos patrones de diseño:

1. El patrón composite.
2. El patrón Template method.

Uso del patrón composite:

El método que efectúa la búsqueda, recibe como parámetro una **Condición**, la cual puede ser una base (Hoja) o una compuesta. De esta forma es posible saber si aplica una Condición Compuesta o Base para las publicaciones.

Ejemplo:

Una CondicionAND con dos condiciones: **Ciudad** : “MarDelPlata” y **CantidadDeHuespedes** : 4
Primero filtra todas las publicaciones del sitio obteniendo aquellas cuya ciudad sea “MarDelPlata” y sobre esas publicaciones obtenidas, toma aquellas que su capacidad de huéspedes sea mayor o igual a 4.

Uso del patrón Template method:

Para el caso de la Condición Base se detectó que todas hacen el filtrado de la misma manera, lo único que cambiaba era la condición. Por lo que el filtar(List<Publicacion>) seria el template method y cumpleConCondicion(Publicacion) seria el hook method.

**Aclaración:**

El profesor Diego Cano nos dijo que no era necesario verificar que estuviesen las obligatorias internamente, ya que esa restricción sería a través de la interfaz.

# Ranking de inmuebles y propietarios

Para este punto simplemente generamos una clase nueva dedicada a contener los datos de la reseña.
Cada clase (Usuario e Inmueble) tiene la posibilidad de agregar a una lista privada la reseña correspondiente. Esto solo sucede si se cumple con la condición solicitada en el punto.
Lo hicimos de esta forma ya que nos proporciona una mayor facilidad a la hora de consultar los promedios de puntuación, acceder a los mejores inquilinos, inmuebles o propietarios, etc.

# Visualización y Reservas

Se crearon los métodos necesarios para que la capa de servicio pueda mostrar la información por pantalla al usuario cuando lo seleccione.

En caso de que el usuario escoja ver los datos del inmueble ubicado en la publicacion podra:

1. Hacer uso del getInmueblePublicado() para obtener el Inmueble y así mostrar los datos que dicha capa crea conveniente. Además desde el inmueble se podrá obtener las reseñas, a través de getReseñas(), para su visualización (Mostrar comentarios y puntajes que dejaron los usuarios en distintas categorías).
2. Hacer uso de getPropietario() para obtener el propietario y así toda la información personal requerida. Mediante cantidadDePublicacionesConInmueble(Inmueble) podrá mostrar cuantas publicaciones hizo para el inmueble publicado,con inmueblesQueAlquilo() podrá mostrar los inmuebles que alquiló alguna vez en el sitio y cantidadDeAlquileresRealizados() el número de publicaciones que hizo ese propietario.

Permitiendo así que la capa de servicio tenga de donde obtener los datos para la visualización requerida.

# Administración de reservas para inquilinos

- Todas las reservas.
- Las reservas futuras: muestra aquellas reservas con fecha de ingreso posterior a la fecha actual.
- Reservas de una ciudad en particular.
- Ver las ciudades en las que tiene reservas.

Para poder cubrir todos los puntos solicitados en la búsqueda de la reserva hicimos 2 métodos, el primero se encarga de filtrar todas las reservas realizadas por el inquilino, este método recibe como parámetro una clase “ModoDeFiltrado”.
Esta última clase está creada en base al Template Method y funciona para filtrar la lista de reservas del usuario, esto lo hace con su respectivo Hook Method.

```
public abstract class ModoDeFiltrado {

    public List<Publicacion> filtrar(List<Publicacion> historialDeReservas) {
        List<Publicacion> listaFiltrada = new ArrayList<>();
        listaFiltrada = historialDeReservas.stream()
                .filter(reserva -> cumpleConCondicion(reserva)).collect(Collectors.toList());

        return listaFiltrada;
    }

    protected abstract Boolean cumpleConCondicion(Publicacion reserva);

}
```

Aunque también nos encontramos con que una de las solicitudes de filtrado pide ver las ciudades en las que hay reservas, por ende tendría que devolver un tipo de dato diferente, así que en este caso decidimos crear un segundo método para filtrar por ciudades y luego devolverlas.
Como última aclaración nosotros podríamos haber creado un método extra que se ocupara de devolver todas las reservas, así como lo pide en el primer punto, pero esto también lo puede hacer el “ModoDeFiltrado” y esto nos ahorraría un mensaje.

# Administración Del Sitio

La implementación de este punto fue desarrollada en la clase **ModuloDeAdministracion, es la** encargada de dar de alta las categorías y gestionarlas mediante el uso de la clase entidad. Decidimos hacer uso de la clase enumerativa **Entidad** debido a que no se apreciaba un comportamiento por parte de las entidades, ni tampoco podía llegar a surgir otro nuevo tipo de entidad que justificara otra implementación. El **ModuloDeAdministracion** se inicializa con las entidades ya cargadas, evitando posibles errores de gestión. Mientras que el **ModuloDePuntaje** se encarga de la lógica referida a cómo se calculan los puntos en el sitio, delegando la responsabilidad a esta clase.

# Politicas de cancelacion

Se implementó usando conceptos de herencia y polimorfismo, los tipos de cancelación que puedan surgir en un futuro, simplemente deben implementar a su manera montoACobrar(Publicación). Permitiendo así tratar a todos los tipo es de manera polimórfica

# Notificaciones

Para completar el punto de las notificaciones hicimos uso del patrón observer, basado en el **ComplexSensor**, dado por el profe Diego Torres en la clase de **Observer**. En el cual muestra como una clase pese a no extender de Observable, tiene los métodos necesarios para llevar a cabo dichas acciones. Esto se pensó, sabiendo que en nuestro modelo las publicaciones solo viven mientras esté en el periodo de tiempo asignado. De esta manera diferentes sitios, apps, etc podrían suscribirse a las publicaciones que más les parezcan relevantes y llevar a cabo notificaciones las notificaciones. Desde el lado de **observer** se diagramó la futura idea de que con la misma estructura puedan surgir nuevos intereses, es así que cada observer concreto saber responder si ese es el interés que tiene asociado. Se utilizó un enumerativo para identificar a qué interés está asociado dicho observer, debido a que solo tendría un método get.

# Cobertura de los test

El TP será entregado con una cobertura del 98,2%.

**Aclaración:**

La clase **UsuarioSATCopia** testea un método, el cual desde la clase original no era posible. Es por eso que la cobertura del usuario debería ser mayor a la dada por el plugin ECL EMMA.