Trabajo Grupal

Integrantes:

- Osores Bianca; biancaosores91@gmail.com
- Justiniano Sofía; sofia.agustina.justiniano@gmail.com
- Jaramillo Caceres Rodrigo ; rodicom10@gmail.com (AUSENTE)

Repositorio de trabajo:

- https://github.com/rodrigo-caceres-jaramillo/unqui-po2-tpfinal

Introducción
Tareas a Realizar
Se deberán completar y entregar los siguientes puntos:

1. Un diseño de la solución completa utilizando diagrama de clases UML.
2. Documentación en un archivo PDF que incluya los integrantes del grupo y sus direcciones de email, las decisiones de diseño, detalles de implementación que merezcan ser explicados, patrones de diseño utilizados y los roles según la definición de Gamma et. al.
3. Implementación completa en lenguaje JAVA que incluya test de unidad con un 95 % de cobertura.
4. Todo lo anterior debe estar alojado en un repositorio de acceso por parte de los docentes donde se pueda realizar un seguimiento del trabajo.

Nota: no se requiere la implementación de interfaces gráficas de ningún tipo.

Decisiones de diseño:

- Realizamos entre los 3 integrantes diferentes UML de los casos que deberíamos implementar en el proyecto (se encuentran disponibles las ideas ORIGINALES en https://github.com/rodrigo-caceres-jaramillo/unqui-po2-tpfinal/tree/main/Documentacion) de esta forma pudimos realizar la división de tareas para cada integrante y encontrar diferentes diseños que debríamos implementar.
  Diseñamos primeramente al Sitio Web como clase principal, que se encargaría de básicamente todas las responsabilidades primordiales de administración , también la clase abstracta Usuario, donde al comienzo desplegamos las ramas de los distintos tipos de usuarios que eran Propietario e Inquilino(1). (Se encuentra disponible UML /UML V2.drawio);

- Testeamos los casos con TDD, de esta forma desarrollamos comportamientos a las clases (por ejemplo cuando el sitio registra usuarios) y casos bordes.

- Implementamos Mocks para que los test unitarios de clases (SUT) no dependan de las otras clases (DOT) que estaban siendo desarrolladas en el mismo momento. De esta forma:
    - Logramos que la clase testeada no se vea afectada por la implementación de otra/s clases
    - Logramos seguridad en la programación y refactorización en equipo ya que “los errores no afectan el desarrollo que realiza mi compañero ni viceversa”
    - Logramos aislar las implementaciones de cada clase.

- Añadimos Administrador.class y las clases que debe administrar, estas son CategoriaDePuntaje.Class ; Servicios.Class ; TipoDeInmueble.Class;

- A medida que se iban testeando los casos en Sitio Web, nos dimos cuenta que al sitio lo sobrecargamos de muchos métodos, como por ejemplo el mismo sitio sabía acceder a su listado de publicaciones y hacer cosas con ella, como filtrar, agregar, remover, etc.
  Las responsabilidades que tenía el sitio web se extendía hacia objetos que no requería conocer; Por lo que delegamos en otros objetos adicionales:
    - AdministradorUsuario.class
    - AdministradorPublicacion.class
    - Administrador.class
    - AdministradorOcupaciones.class
    - AdministradorReservas.class

  De esta forma logramos que se pueda colaborar entre sitio y administrador para lograr diferentes tareas que debía hacer sitio.

- No tuvimos mayor inconveniente a la hora de desarrollar las clases Inmuebles y publicaciones, aclaramos que queda pendiente el refactor de FormasDePagoEnum ya que no llegamos con los tiempos por las razones previamente comentadas.

- (1)Realizamos un refactor importante en donde eliminamos las subclases Inquilino y Propietarios, esto nos permite abstraernos también del tipo de usuario que estemos testeando y los tipos de Usuarios que podía registrar un sitioWeb. Básicamente las dos subclases no difieren en responsabilidades.

- Una parte importante de nuestro trabajo la concentramos en búsquedas de inmuebles ya que no conocíamos cómo parametrizar los datos opcionales “nulos”. Pensamos en una posible solución, que fue crear un objeto que que represente a los parámetros <<ParametrosBusqueda>>. Ésta tiene 2 constructores, en uno sólo puede tener los 3 datos obligatorios, en el otro los

6 y su responsabilidad es realizar la verificación en los primeros 3 campos obligatorios que no deben ser nulos.

- Implementamos el patrón observer para el caso en el que se genera una notificación cuando se actualiza un precio de alguna publicación:
    - En este sentido los elementos que entran en juego son
        - Observer : SitioDeOfertasObserver.class
        - Observable: SitioWeb.class
    - En el envío de la notificación de cambios para los observers decidimos establecer un parámetro (Publicación) el cual permite al observer verificar que ese cambio es de su interés.

☐ Realizamos la regla de negocio cancelaciones de reserva, en el cual identificamos el Strategy Design Pattern, a continuación identificamos los elementos resultantes:

- Compositor: <<Abstract PoliticaDeCancelacion>>
    - *+ elUsuarioPuedeCancelarReserva():boolean*
- ConcretStateA: PoliticaDeCancelacionGratuita
- ConcretStateB: PoliticaDeCancelacionSimple
- ConcretStateC: PoliticaDeCancelacionIntermedia

De esta forma abarcamos las diferentes situaciones que propone el enunciado, logramos:

- ☐ Evitar el anidamiento de condicionales por cada “tipo” de cancelación.
- ☐ Representamos los posibles casos determinados en el enunciado en objetos que se responsabilizan a partir de un estado interno.
    - ☐ Encapsulamos la implementación en el método +*elUsuarioPuedeCancelarReserva():boolean,*
- ☐ Este encapsulamiento permite un nivel mayor de polimorfismo (todos los hijos pueden responder y sobreescribir el método)
- ☐ Permite la apertura y flexibilidad en caso de que el enunciado agregue o elimine algún caso de cancelación.
- ☐ En este sentido
    - ☐ La implementación nos permite cumplir con Open-Closed principle.
    - ☐ La implementación nos permite cumplir con Liskov principle.

- [ ] Para finalizar, encontramos algunos detalles que continuamos refactorizando:
- [ ] Implementamos interfaces que permiten establecer un protocolo en común entre las entidades registrables en el sistema<Registrables>:
    - (administrador - tipoDeInmueble - Usuario - publicacion - reserva - categoriaDePuntaje) son aquellas entidades que el usuario requerirá que se registren.
    - De esta forma logramos ampliar la capacidad del sitio web para su extensión a futuro sin modificar el comportamiento que poseen estos; podremos cumplir con el protocolo de Open-Closed principle.
    - Al solo tener en cuenta que se puedan registrar, no requerimos que los mismos tengan que implementar métodos que no le corresponden, podremos cumplir con el protocolo de Interface Segregation Principle.
    - En este sentido, podríamos decir que incluír esta interfaz permite establecer un state para que cada objeto registrable deba implementar registrarseEn(SitioWeb sitio);

- [x] ~~Implementamos interfaces respecto a <IEntidadesPuntuables>~~
    - Estas nos va a permitir que sin importar cuál sea el objeto receptor del mensaje (Usuario o Inmueble) pueda registrar un Puntaje y manipular los mismos para ofrecernos datos;
    - Permitirá que en un futuro esté abierto a modificaciones sin afectar a las entidades ya generadas, por ejemplo si se quisiera añadir un nuevo objeto que pueda recibir o dar puntajes.
- [ ] Implementamos interfaces respecto a <Administrable>:
    - Esto nos va a permitir que el Sitio web pueda añadir una flexibilidad en cuanto a la administración de otras entidades.
    - Pueden añadirse nuevas funcionalidades al sitio sin tener que modificar la funcionalidad desarrollada
- [ ]