# Trabajo Práctico Final

## Programación 2

# Property Management Program

Aguilera Facundo

Sommacal Juan Pablo

Soria Franco

Sosa Yago

Cursada 2024

# Índice:

# Definición

Nuestro programa de gestión inmobiliaria está diseñado para cubrir las necesidades de agentes inmobiliarios y administradores, ofreciendo las herramientas necesarias para la gestión de distintos tipos de propiedades, clientes y las distintas operaciones que se realizan entre ellos.

El sistema permite gestionar distintos tipos de propiedades (como casas, departamentos, lotes y depósitos, entre otros) y mantener un registro de clientes interesados en alquilar o comprar. Además incluye herramientas para realizar operaciones clave, como registrar nuevas propiedades, modificar una propiedad ya registrada y asociar propietarios a propiedades.

El objetivo principal es facilitar las tareas diarias de las inmobiliarias al proporcionar una solución tecnológica que permita manejar grandes volúmenes de datos de manera rápida y confiable. Esto no solo reduce significativamente el tiempo dedicado a tareas administrativas, sino que también minimiza errores comunes, como registros duplicados, cálculos incorrectos o pérdida de información.

Adicionalmente, el programa está diseñado con una interfaz intuitiva que permite a los usuarios aprender a utilizarlo sin necesidad de capacitación técnica avanzada.

# Desarrollo

El desarrollo del programa se llevó a cabo utilizando *Intellij IDEA*, un entorno de desarrollo integrado (IDE) que ofreció un soporte sólido para el lenguaje *Java*.

Adoptamos un enfoque práctico basado en la división de tareas y propiedades. Utilizamos la plataforma Trello para organizar y asignar las tareas entre los integrantes del equipo. Este enfoque permitió desglosar el proyecto en partes manejables. Como la

implementación de clases principales, la gestión de persistencia de datos y las funcionalidades específicas del sistema.

Principales etapas del desarrollo:

- Definición de los requisitos: Identificamos las necesidades básicas del sistema.
- Diseño del sistema: Creamos un diseño inicial de clases, enfocándonos en la modularidad y claridad en la estructura del programa.
- Implementación básica: Desarrollamos las clases principales: Propiedad y cliente y establecimos relaciones entre ellas.
- Persistencia de datos: Implementamos mecanismos para que los datos se mantuvieran entre ejecuciones.
- Pruebas y refinamiento: Realizamos pruebas para identificar y solucionar errores, optimizando el código donde fue necesario.

Desafíos encontrados y soluciones:

- Manejo de persistencia de datos
  - Desafío: Lograr que los datos, como los identificadores incrementales, persistieron entre ejecuciones del programa.
  - Solución:

- Relación entre clases
  - Desafío: Diseñar operaciones claras entre propiedades, clientes y operaciones.
  - Solución: Creamos una clase genérica para el manejo de todos los objetos, reutilizando el código. Además la clase *Rent* y *Sale* se dedican al alquiler y venta de las propiedades

- Coordinación del equipo:
  - Desafío: Dividir las tareas y mantener una visión clara para el avance del proyecto.
  - Solución: Utilizamos Trello para dividir las tareas en objetivos pequeños lo que permitió visualizar el progreso y mantenernos organizados.

# Informe Técnico

Funcionamiento del sistema:

El sistema cuenta con un menú fácil de utilizar, que permite agregar nuevas propiedades con datos como dirección, precio y tipo de propiedad. También incluye opciones para ver las propiedades cargadas y poder hacer modificaciones o bajas.

También cuenta con las operaciones básicas como son la renta y el alquiler. Pudiendo llevar un control de las operaciones realizadas y las partes involucradas.

Decisiones de diseño:

Optamos por una arquitectura basada en capas para separar las responsabilidades y facilitar la comprensión del código.

- Capa de datos: Maneja la persistencia de la información en archivos JSON.
- Capa de lógica de negocios: Gestiona las operaciones principales como agregar, modificar y eliminar propiedades o clientes y realizar operaciones entre las distintas partes.
- Capa de interfaz de usuario: Utilizamos un menú con navegación con números para que el usuario no tuviera dificultad al utilizar el programa

Este diseño modular nos permite realizar cambios en una capa sin que las otras se vean afectadas.

El diseño del sistema incluye 3 clases fundamentales:

- Propiedad: Clase padre de donde se desprenden el resto de propiedades (departamento, casa, local, deposito, etc)
- Cliente: Clase padre de donde se desprenden el resto de clientes (propietario, arrendatario y comprador)
- Clase genérica: Actúa como controlador principal, administrando la relación entre clientes y propiedades, además de facilitar las búsquedas.

La clase genérica se diseñó para ser flexible y extensible. Recibe cualquier objeto que herede de Propiedad o Cliente  y nos permite realizar búsquedas, agregar, modificar o eliminar con facilidad.

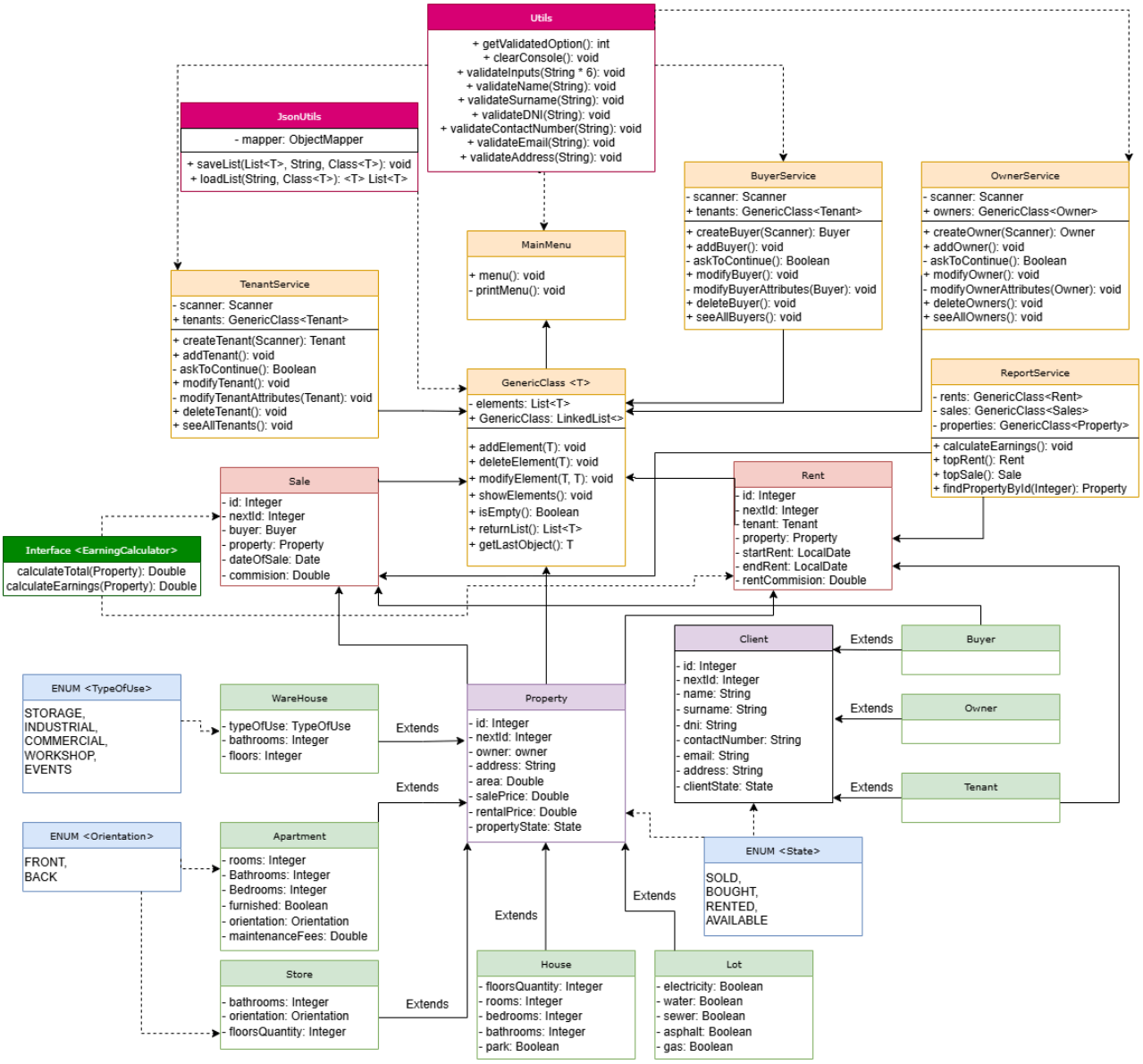
Las propiedades y clientes están relacionados mediante identificadores únicos que facilitan la trazabilidad.

Implementamos la persistencia de datos utilizando archivos JSON. Esto permite que los datos del sistema se mantengan entre ejecuciones. Decidimos utilizar JSON porque es un formato ligero y fácil de leer para máquinas y humanos. Además su uso facilita futuras integraciones con aplicaciones web o móviles.

La lógica de negocio está pensada de una forma modular, facilitando así agregar nuevas operaciones si fuera necesario. Al iniciar el programa, los datos se recuperan de los archivos JSON correspondientes y se guardan en Linked Lists. De esta forma podemos manejar grandes cantidades de datos con facilidad. Luego de terminada la sesión, se guarda todo de nuevo en el archivo JSON para su próximo uso.

El manejo de excepciones personalizadas nos permite evitar errores comunes, entre ellos intentar eliminar una propiedad que está siendo alquilada, alquilar o vender una propiedad que ya está vendida o que ya se alquiló a otra persona o ingresar la misma propiedad dos veces al sistema.

Clases utilizadas y sus relaciones:




[Click aquí para ver mejor la imagen]

Property (Propiedad): Clase padre con los siguientes atributos

- id: De tipo Integer, representa el identificador único de la propiedad. Es autoincremental.
- nextId: De tipo Integer, representa el Id que tendrá la siguiente propiedad que se agregue al sistema.
- Owner (Propietario): Objeto que hereda de la clase Cliente. Representa el dueño del inmueble

- address (dirección): De tipo String, representa la dirección del inmueble.
- área; De tipo Double, representa la cantidad de metros cuadrados que ocupa la propiedad.
- salePrice (precio de venta): De tipo Double, representa el precio de venta del inmueble. Es null si no está para la venta.
- rentalPrice (precio de alquiler): De tipo Double, representa el precio de alquiler del inmueble. Es null si no está para alquilar.
- propertyState (estado): Enum que representa el estado de la propiedad. Puede ser:
    - Sold (vendida)
    - Rented (alquilada)
    - Available (disponible)

De esta clase se desprenden las siguientes subclases

- WareHouse (Galpón)
    - typeOfUse (tipo de uso): Enum que representa el tipo de uso, puede ser:
        - Storage (depósito)
        - Industrial (industrial)
        - Commercial (comercial)
        - Workshop (taller)
        - Events (eventos)
    - bathrooms (baños): De tipo Integer, representa la cantidad de baños.
    - floorsQuantity (cantidad de pisos). De tipo Integer, representa la cantidad de pisos o plantas.

- Apartment (Departamento)
    - rooms (ambientes): De tipo Integer, representa la cantidad de ambientes.
    - bathrooms (baños): De tipo Integer, representa la cantidad de baños.
    - bedRooms (habitaciones): De tipo Integer, representa la cantidad de habitaciones.
    - furnished (amoblado): De tipo Boolean, es verdadero si el departamento está amoblado y falso si no lo está.

    - orientation (orientación): Enum que representa la orientación del departamento, puede ser:
        - Front (da a la calle)
        - Back (da a la manzana)
    - maintenanceFees (expensas): De tipo Double, representa las expensas que hay que pagar además del alquiler.

- Store (Tienda)
    - bathrooms (baños): De tipo Integer, representa la cantidad de baños.
    - orientation (orientación): Enum que representa la orientación del departamento, puede ser:
        - Front (da a la calle)
        - Back (está en una galería)
    - floorsQuantity (cantidad de pisos). De tipo Integer, representa la cantidad de pisos o plantas.

- House (Casa):
    - floorsQuantity (cantidad de pisos). De tipo Integer, representa la cantidad de pisos o plantas.
    - rooms (ambientes): De tipo Integer, representa la cantidad de ambientes.
    - bedRooms (habitaciones): De tipo Integer, representa la cantidad de habitaciones
    - bathrooms (baños): De tipo Integer, representa la cantidad de baños.
    - park (parque): De tipo Boolean, es verdadero si la casa tiene parque y falso si no lo tiene.

- Lot (Lote):
    - electricity (electricidad); De tipo Boolean, es verdadero si el lote tiene electricidad y falso si no la tiene.
    - water (agua): De tipo Boolean, es verdadero si el lote tiene agua corriente y falso si no la tiene.
    - sewer (cloacas): De tipo Boolean, es verdadero si el lote posee cloacas y falso en caso contrario.

- asphalt (asfalto); De tipo Boolean, es verdadero si la calle está asfaltada y falso si no lo está.
- gas: De tipo Boolean, es verdadero si el lote posee gas natural y falso si no posee.

Por otro lado tenemos la clase padre Client (Cliente), que posee los siguientes atributos.

- id: De tipo Integer, representa el identificador único de la propiedad. Es autoincremental.
- nextId: De tipo Integer, representa el Id que tendrá el siguiente cliente que se agregue al sistema.
- name (nombre): De tipo String, representa el nombre del cliente.
- surname (apellido): De tipo String, representa el apellido del cliente.
- dni: De tipo Integer, representa el Documento Nacional de Identidad del cliente.
- contactNumber (número de contacto): De tipo String, representa el número de contacto del cliente.
- email: De tipo String, representa el email del cliente.
- address (dirección): De tipo String, representa la dirección del cliente.
- clientState (estado del cliente) : Enum que representa el estado del cliente. Puede ser
  - Bought (participó de una compra)
  - Rented (participó de una renta)
  - Available (todavía no participo de ninguna operación)

De esta clase se desprenden las siguientes subclases:

- Owner (Propietario)
- Buyer (Comprador)
- Tenant (Arrendatario)

Las operaciones que podemos realizar están divididas en dos clases.

- Sale (Venta):
  - id: De tipo Integer, representa la ID única de la transacción. Es autoincremental.

- nextId: De tipo Integer, representa el Id que tendrá la siguiente operación que se agregue al sistema.
- Buyer (Comprador): De tipo Buyer, objeto que se desprende de la clase Cliente, representa el comprador de la propiedad.
- Property (Propiedad): De clase Property o sus subclases. Representa la propiedad en cuestión.
- dateOfSale (fecha de venta): De clase LocalDate, representa la fecha de venta de la transacción.
- saleCommision (comisión por venta): De tipo Double, representa el porcentaje de comisión por venta.

- Rent (Renta):
  - id: De tipo Integer, representa la ID única de la transacción. Es autoincremental.
  - nextId: De tipo Integer, representa el Id que tendrá la siguiente operación que se agregue al sistema.
  - Tenant (Arrendatario): De tipo Tenant, objeto que se desprende de la clase Cliente, representa el inquilino de la propiedad.
  - Property (Propiedad): De clase Property o sus subclases. Representa la propiedad en cuestión.
  - startRent (comienzo alquiler): De clase LocalDate, representa la fecha de inicio del contrato de alquiler.
  - endRent (fin alquiler): De clase LocalDate, representa la fecha de finalización del contrato de alquiler.
  - rentCommision (comisión por alquiler): De tipo Double, representa el porcentaje de comisión por alquiler.

Tenemos nuestra Generic Class (clase genérica)

- elements (elementos): De tipo LinkedList<>, representa la lista de elementos y nos da el tipo de objeto que vamos a utilizar
- GenericClass (Clase genérica): De tipo LinkedList<>, representa nuestra lista genérica.

Posee los siguientes métodos

- addElement(T) (agregar elemento) retorno void: Agrega un elemento de tipo T (objeto) a la linked list.
- deleteElement(T) (eliminar elemento) retorno void: Busca el elemento en la lista y lo elimina.
- modifyElement(T, T) (modificar elemento) retorno void: Reemplaza el elemento con uno modificado.
- showElements() (mostrar elementos) retorno void: Muestra todos los elementos en la lista.
- isEmpty() (está vacío) retorno Boolean: Retorna verdadero si está vacía la lista, falso si tiene algún contenido.
- returnList() (retornar lista) retorno List<T>: Retorna la lista de objetos.
- getLastObject() (obtener ultimo objeto) retorno T: Retorna el último objeto de la lista.

El ReportService (Service de reportes) tiene los siguientes atributos:

- rents (rentas): De tipo GenericClass<Rent>
- sales (ventas): De tipo GenericClass<Sales>
- properties (propiedades): De tipo GenericClass<Property>

Los metodos son:

- calculateEarnings() (Calcular ganancias): retorno void. Calcula las ganancias totales obtenidas de las rentas y las ventas que tomaron lugar hasta el momento.
- topRent() (mayor renta): retorno Rent. Busca la renta que mayor ganancia le da a la inmobiliaria
- topSale() (mayor venta): retorno Sale. Busca la venta que mayor ganancia le da a la inmobiliaria
- findPropertyById(Integer): retorno Property. Busca la propiedad que corresponde a la Id que se pase por parámetro.

Y por último los Service de Owner (propietario), Buyer (comprador) y Tenant (Arrendatario). Estos poseen los mismos métodos pero adaptados a los atributos de cada objeto.
A continuación detallamos los métodos y atributos del service de propietario

- scanner: De tipo Scanner, se utiliza para tomar la entrada por teclado del usuario
- owner: De tipo GenericClass<Owner> representa la lista de todos los propietarios del sistema.

Posee los métodos:

- createOwner(Scanner) (crear propietario) retorno Owner. Crea un nuevo propietario
- addOwner() (agregar propietario) retorno Void. Llama al método agregar de la clase genérica.
- askToContinue() (preguntar si continua) retorno Boolean. Pregunta al usuario si desea seguir cargando nuevos clientes, es verdadero si quiere continuar, falso si ya no carga más.
- modifyOwner() (modificar propietario) retorno Void. Reemplaza a un propietario por el mismo con sus atributos modificados.
- modifyOwnerAtributes(Owner) (modificar atributos de propietario) retorno Propietario. Modifica un atributo del propietario pasado por parámetro.
- deleteOwners() (eliminar propietario) retorno Void. Elimina un propietario de la lista.
- seeAllOwners() (ver todos los propietarios) retorno Void. Muestra por pantalla todos los propietarios.

Los service de Arrendatario y Comprador son exactamente iguales pero cambiando el objeto en cuestión.

Además contamos con una interfaz implementada por la clase Rent (renta) y Sale (venta). Los métodos que contiene esta interfaz son:

- calculateTotal(Property) (calcular total) retorno Double.
- calculateEarnings(Property) (calcular ganancias) retorno Double.

Estos son implementados por la clase Rent de la siguiente forma

- calculateTotal(Property) (calcular total) retorno Double. Calcula el precio total del contrato de alquiler.
- calculateEarnings(Property) (calcular ganancias) retorno Double. Calcula el porcentaje de ganancia de la inmobiliaria por ese contrato de alquiler.

Y por la clase Sale de la siguiente forma:

- calculateTotal(Property) (calcular total) retorno Double. Obtiene el precio de la venta.

- calculateEarnings(Property) (calcular ganancias) retorno Double. Calcula el porcentaje de ganancia de la inmobiliaria por esa venta.

Para las validaciones al ingresar datos utilizamos la clase Utils (útiles) con los siguientes métodos:

- getValidatedOption() (validar opción) retorno int. Obtiene entrada por teclado del usuario. Si es un numero lo retorna, sino tira un InvalidInputException.
- clearConsole() (limpiar consola) retorno Void. Imprime 10 saltos de línea para dar la ilusión que se limpió la consola.
- validateInputs(String * 6) (validar entradas) retorno Void. Recibe por parámetro 6 Strings que representan el nombre, apellido, dni, numero de contacto, email y dirección de un cliente. Verifica que estén todos los datos correctos utilizando los siguientes métodos:
    - validateName(String) (validar nombre) retorno Void. Verifica que el nombre no esté vacío. Si lo está, tira una InvalidInputException.
    - validateSurname(String) (validar apellido) retorno Void. Verifica que el apellido no esté vacío. Si lo está, tira una InvalidInputException.
    - validateDni(String) (validar DNI) retorno Void. Verifica que el DNI no esté vacío y que no contenga letras. Si no se cumplen estas dos condiciones, tira una InvalidInputException.
    - validateContactNumber(String) (validar número de contacto) retorno Void. Verifica que el número solo contenga dígitos. Si posee algún carácter, tira una InvalidInputException.
    - validateEmail(String) (validar Email) retorno Void. Verifica que el email contenga un “@”. Si no la contiene, tira una InvalidInputException.
    - validateAddress(String) (validar dirección) retorno Void. Verifica que la dirección no esté vacía. Si lo está, tira una InvalidInputException.

Para la persistencia de datos la clase JsonUtils con los siguientes metodos:

- saveList(List<T>, String, Class<T>) retorno void. Guarda un List en un archivo Json
- loadList(String, Class<T>) retorno <T> List<T>. Recupera un List de un archivo Json

Y finalmente tenemos nuestras Excepciones personalizadas:

- DuplicateElementException
- ElementNotFoundException
- InvalidAreaException
- InvalidInputException
- InvalidPriceException
- RentedException
- SoldException

# Conclusión

El desarrollo de este sistema de inmobiliaria nos permitió consolidar nuestros conocimientos en la programación orientada a objetos en Java. A través de este proyecto, logramos implementar una solución práctica y funcional para cubrir las principales necesidades de una inmobiliaria.

Durante el proceso enfrentamos distintos desafíos técnicos, como la implementación de identificadores autoincrementales y la utilización de una clase genérica para reutilizar los métodos, que pudimos resolver consultando con nuestro profesor. Además la utilización de Trello nos permitió organizar las tareas de forma efectiva.

Si bien el sistema cumple con los requisitos básicos, se puede modificar fácilmente para implementar nuevas operaciones o bases de datos relacionales.

En resumen, este trabajo no solo nos permitió afianzar nuestros conocimientos, sino también tener la experiencia de trabajar en equipo en un proyecto de programación.