UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

Departamento de Ingeniería Informática

LABORATORIO 3 – PARADIGMA ORIENTADO A OBJETOS

Paradigmas de programación

Ignacio Andrés Tapia Donaire 21.142.512-1

Sección:

13204-0-A-1

Profesor:

Edmundo Leiva Lobos

Fecha Entrega:

02 de julio del 2025

Introducción:

El DIINF de la Usach ha solicitado la construcción de un juego llamado **CAPITALIA**, juego de similares características a **Monopoly** donde su particularidad radica en el uso de impuestos y el cambio de reglas que pueden modificar el cómo se juega a este título para hacer de este mismo una experiencia más interesante y única. El encargado de llevar a cabo esta idea es el presente en el informe, donde se debe de utilizar el paradigma especificado para construir el juego en su totalidad.

El presente documento consta de una Introducción, descripción del problema, explicación en detalle del paradigma, análisis del problema, diseño de solución, aspectos de implementación, instrucciones de uso, resultados y autoevaluación y conclusiones, adicionalmente el paradigma abordado va a ser el **paradigma orientado a objetos** en el lenguaje de **Java**.

Descripción del problema:

El problema principal radica en la creación de una forma de gestión general entre jugadores y juego, donde se debe tener especial cuidado de que el jugador pueda interactuar correctamente con los elementos establecidos por el programador y que las interacciones entre clases sean coherentes entre sí para tener un juego funcional. La dificultad principal dentro de esto radica en la correcta gestión de herencias e interfaces ya que existen relaciones entre clases (como lo pueden ser Propiedad y Hotel) que requieren de ser precisas para no caer en inconsistencias, además, el sistema en general debe de ser lo suficientemente flexible como para permitir adaptabilidad, ya sea cambiando la cantidad de jugadores, propiedades, impuestos, dados, casas, hoteles, etc. Con todo lo anterior es razonable decir que la problemática necesita e implora de un diseño lo suficientemente robusto como para atacarlo, enfrentarlo y superarlo.

Paradigma:

Para el presente laboratorio se encuentra en utilización el paradigma **orientado a objetos**, el cual tiene varios conceptos que son **la clave** para entenderlo:

- **Clases:** La base principal del paradigma, son el molde o plantilla que define que atributos y comportamientos tendrán los objetos que se crean a partir de ella. Por ejemplo, una clase **Persona** puede tener atributos como el nombre y la edad, y métodos de como caminar y hablar.
- **Objeto:** Son instancias concretas de una clase, es decir, cuando se crea un objeto se esta generando un ejemplar especifico con valores propios definidos por la **clase**, por ejemplo, **persona1** es un **objeto** de la **clase Persona**, donde se le atribuye un nombre “Juan” y una edad “25”.

- **Atributos:** características o propiedades que definen el estado de un **objeto**, por ejemplo, en una clase **Auto** sus atributos pueden ser **color, marca y modelo.**
- **Métodos:** Acciones o comportamientos que pueden realizar un **objeto**, por ejemplo, el **Auto** puede **acelerar() o frenar()**.
- **Herencia:** Existe una **subclase** que **hereda** los **métodos y atributos** de su **superclase**¸ ósea, existe una clase (llamada **superclase**) que le "entrega" sus funcionalidades a sus clases heredadas (llamadas **subclases**), esto habilita la reutilización de código y establecer jerarquías, por ejemplo, un **Auto** y una **Moto** son **subclases** de los **Vehículo** ya que ambos pertenecen a esa misma categoría.
- **Encapsulamiento:** Dejar escondido (o **encapsulado**) ciertos métodos y atributos y solo permitir mediante **ciertos** métodos entregarlos, usualmente se usa para mantener le integridad de los datos, siempre y cuando no se quieran compartir.
- **Polimorfismo:** Permite que varias clases respondan a un mismo método de **distintas** maneras, por ejemplo, un **Auto** y una **Moto** tienen el **método avanzar()** pero la implementación es especifica de cada clase.

Estos detalles son **fundamentales** para el correcto funcionamiento del **POO (P**aradigma **O**rientado a **O**bjetos) ya que la comunicación entre clases es la metodología principal para la resolución de todo problema dentro del paradigma, es gracias a esto se puede mantener una buena organización y lectura del código escrito.

Análisis del problema:

Para el correcto análisis debemos ir punto por punto:

- **Creación de una partida e inicio del juego:**
  - Se requiere de una forma de poder iniciar la partida dadas las condiciones iniciales, por ejemplo, una cantidad mínima de jugadores, una cantidad mínima de dados, cantidad de impuestos, cantidad máxima de casas y hoteles, etc.
- **Turnos y movimiento:**
  - Analizar el orden en el que parten los jugadores, si es aleatorio o basado en un evento activo, además del manejo de dados dobles, triples y cartas que puedan afectar directamente en ello.
- **Compra de propiedades:**
  - Ver si se le permite al jugador comprar directamente una propiedad o si tiene requerimientos especiales (como dar una vuelta al tablero), también el cómo certificar que X propiedad pertenece a Y jugador y sus asociados como la construcción de casas y hoteles junto con el cobro de renta.

- **Pago de Renta:**
    - El jugador que caiga en X propiedad que ya pertenezca a Y jugador debe de pagarle una suma determinada, basada en la cantidad de casas y hoteles que este tenga, si es que tiene.
- **Pago de impuestos:**
    - Cuando el jugador pase por la salida se le deben cobrar impuestos sobre sus propiedades y ver si este impuesto será sobre la renta o el precio de las propiedades.
- **Construcción de casas y hoteles:**
    - Básicamente ver cuándo y en qué momento un jugador puede colocar casas y hoteles y el límite de estos mismos, su efecto en la renta, etc.
- **Cartas de suerte y comunidad:**
    - Cuando caiga en la casilla correspondiente debe de ser capaz de sacar la carta correspondiente y esta debe ejercer su acción, eliminando la carta de la lista en el proceso.
- **Ir a la cárcel:**
    - Dependerá del caso, ósea si cae en la casilla, si fue una carta de suerte o comunidad o el caso de los dados dobles seguidos, también el cómo puede salir, si es mediante fianza, sacar dobles u ocupar cartas especiales.
- **Hipotecas y prestamos:**
    - El jugador debería ser capaz de hipotecar sus propiedades en el momento que quiera o solo si está en bancarrota, esta forma de hipoteca se puede también considerar un préstamo.
- **Bancarrota y eliminación:**
    - Si un jugador está en bancarrota pierde automáticamente, siempre y cuando ya haya explorado las posibilidades del préstamo e/o hipoteca, en la teoría el jugador puede elegir irse a bancarrota por sí solo.

Adicional existen requerimientos propios del enunciado como que el tablero sea dinámico, una gestión de jugadores a la hora de ingresar y salir del juego, compra, venta e hipoteca de propiedades, sistema de rentas, fluctuación de impuestos y mercado, eventos económicos y políticos, banca y gestión de victoria y condiciones de victoria establecidos(Oracle,2025).

Diseño de solución:

Para poder afrontar el problema general, es necesario tener una base lo suficientemente robusta como para poder confrontarse a ello, dado que nos encontramos en el **POO** es factible ir clase por clase, denotando que roles deben cumplir cada uno de estos:

TDA Player:

El jugador es una pieza fundamental en el juego, este debe ser capaz de **comprar propiedades, casas, hoteles e hipotecar** estos mismos, posee atributos tales como su **nombre, posición, dinero, propiedades en posesión, las cartas para salir de la cárcel que tiene y si se encuentra en una cárcel(Anexo A)**, con los getters y setters correspondientes, es posible cambiar atributos como **posición, cantidad de cartas y si está en la cárcel.**

TDA Property:

Las propiedades son el principal mecanismo del juego, en el sentido que permiten que los jugadores puedan reducir la cantidad de dinero que tienen sus oponentes, siempre y cuando estos posean al menos una de ellas, estas tiene por definición un **nombre, posición, precio, renta, dueño, la cantidad de casas (y hoteles de ser el caso) y si se encuentra o no hipotecada(Anexo B)**, las propiedades deben de ser capaces de manejar sus **rentas**, ya sean las del **jugador, de la propia propiedad y el cuándo se pagan.**

TDA Card:

Las cartas pueden y deben ser capaces de crear situaciones tanto favorables como desfavorables, estas se construyen con **un id, una descripción y la acción especifica**, en general su tarea principal es simplemente **ejecutar las acciones de las cartas que puedan ser herencia de la misma(Anexo C).**

TDA Board:

El tablero contiene las **cartas, propiedades y casillas especiales** que se encuentran dentro del juego(**Anexo D**), dado que tiene la lista de cartas y las propiedades tienen su propio manejo, el tablero se encarga de **extraer las cartas de la lista**.

TDA Game:

El corazón del juego se encarga de la coordinación general del flujo de juego, tiene listas para los **jugadores, propiedades, cartas y casillas especiales** para poder utilizarlas de ser necesario, maneja la lógica de **la cárcel y las hipotecas** y también el **turno jugado**, también

tiene a su cargo el **cargar los datos iniciales** y una **parte** de la interacción del jugador con el juego en sí, pero es importante notar que no lo hace en su totalidad.

Se debe señalar que el **TDA Game** pareciera cargar la gran mayoría de la responsabilidad del juego, pero la cantidad de llamados a otras clases hace de esta simplemente el encargado de **jugar**, por lo cual es natural ver a la absoluta mayoría de clases ir en dirección a **Game**.

Otro detalle importante es que existen subclases para **el menú del juego** y el **manejo de acciones de las cartas** mediante el uso de **interfaces(Anexo D),** esto para no cargar a **Game** y así mantener una **alta cohesión y un acoplamiento lo más bajo posible.**

Aspectos de implementación:

Estructura del Proyecto:

Se utilizo el **paradigma orientado a objetos** para la realización del presente proyecto, donde fueron utilizadas características **clave** de este para el correcto desarrollo de este, especificando:

- **Herencia:** Se utiliza herencia de clases para **Property** y **Card**, donde en **Property** es el **Hotel** la clase que hereda de este, agregando sus propios métodos y atributos para poder hacer el correcto manejo de hoteles, aprovechando la estructura ya creada por Property.
- **Encapsulación:** Todas las clases principales poseen getters y setters correspondientes para el manejo de sus variables dependiendo del problema donde se requieren, manteniendo así la privacidad de los datos y entregándolos sola y exclusivamente cuando son necesarios.
- **Polimorfismo:** Existen formas de polimorfismo dentro del proyecto, principalmente el pago de las rentas, donde dependiendo si es un hotel o una propiedad lo hace de A o B manera.

Adicional, existe un **menú interactivo** que permite al jugador interactuar con el sistema de juego que fue creado.

Interprete utilizado:

- **Versión de Java: Java (Version 11).**
- **Entorno de desarrollo (IDE): IntelliJ IDEA 2025.1.3 (Community Edition).**
- **Compilador: OpenJDK version “11.0.27”/Temurin-11.0.27+6 (build 11.0.27+6).**

- **Sistema de construcción: Gradle 8.8 or 8.13 (ambos presentes en la carpeta .gradle) (Gradle,2025).**

Razones de uso:

El IDE de IntelliJ es ideal para trabajar con java ya que indica no solo errores de escritura, sino que además sugiere que cosas podrían ser las que provocan dicho error, a la hora de escribir setters y getters este te autocompleta el código en cuestión y en general es extremadamente cómodo y útil a la hora de programar en **Java.**

Instrucciones de uso:

Las instrucciones de uso son para OpenJDK 11.0.27 en el sistema operativo Windows 10:

1.- Ir a la carpeta contenedora de archivos de nombre "Lab3_21142512_IgnacioTapiaDonaire" (**Anexo E**).

2.- Abrir una terminal en la dirección de la carpeta contenedora o abrir la consola (**cmd**) y ocupar el comando "cd [ruta de la carpeta]". Si esto no es posible, se puede dirigir a **la esquina superior izquierda** donde dice **Archivo**, hacer click y luego hacer click en **Abrir Windows Powershell(Anexo F).**

3.- Dentro de la terminal (sea Powershell o cmd) ejecute el comando ".\gradlew.bat build" **sin las comillas**.

4.- Una vez listo el paso anterior ejecute el comando " .\gradlew.bat run --console=plain"

Con el paso 4 completado, el menú del juego debería de estar en consola presente, de no ser así, por favor, repetir los pasos anteriores **uno por uno** para asegurar la correcta ejecución del programa.

Resultados y autoevaluación:

La autoevaluación se encuentra dentro de la misma carpeta de "Lab3_21142512_IgnacioTapiaDonaire" , se puede decir que el programa funciona de manera satisfactoria, donde a la hora de jugar varias veces no se presentaron errores **mayores** como el fallo de la build en gradle o acciones imposibles.

Sin embargo, es importante notar que hay veces que los turnos se confunden y se adelantan el uno al otro, también es importante notar que existe una carta que, por razones desconocidas, reinicia la cantidad de casas a la por "default" establecida en el main del programa, aunque no interrumpe realmente a la ejecución del juego. Con los detalles anteriores denotados se puede decir que existe una simulación satisfactoria del juego Capitalia.

Conclusiones:

El presente proyecto presento dificultades en la correcta organización de las clases y sus distintos propósitos, teniendo problemas hasta casi el final del desarrollo donde la absoluta mayoría de la lógica se encontraba dentro del TDA Game, esto no solo destruye el principio de alta cohesión y bajo acoplamiento, sino que vuelve mucho más complicada la lectura del propio programa, sin embargo, se pudo superar después de separar una gran parte de la lógica en sus clases respectivas.

El POO es un paradigma extremadamente similar a los vistos en principios de la carrera -dígase C y Python- con la característica de que es bastante simple el poder reutilizar el código y con ello el agilizar la creación de métodos dentro de las clases, los conceptos de encapsulación y polimorfismo ayudan de gran manera a manejar los atributos creados para cada clase y las herencias junto con las interfaces son un aire fresco bastante grande, permitiendo "delegar" esas funcionalidades a otra clase en lugar de saturar una en general.

En conclusión y retrospectiva, el lenguaje y proyecto enfatizan en la importancia de la organización y el pensamiento de ingeniero que uno debe tener a la hora de enfrentar tales desafíos, donde un buen análisis y correcta asignación hacen la diferencia a la hora de crear y plasmar la idea en el lenguaje/proyecto.

Referencias:

Gradle. (2025). *Version Catalogs.* En Gradle User Manual. https://docs.gradle.org/current/userguide/version_catalogs.html#header

Oracle. (2025). *Object-Oriented Programming Concepts*. En Java Documentation. https://docs.oracle.com/javase/tutorial/java/concepts/

Anexos:

```
*/
public class Player_21142512_IgnacioTapia {  Damegoll *
    private int id;  2 usages
    private String nombre;  3 usages
    private int dinero;  5 usages
    private List<Property_21142512_IgnacioTapia> propiedades;  3 usages
    private int posicion;  3 usages
    private boolean enCarcel;  3 usages
    private int contadorRepetidos;  2 usages
    private int totalCartasSalirCarcel;  3 usages
```

Anexo A

```java
public class Property_21142512_IgnacioTapia {  58 usages  1 inheritor  Damegoll *
    private int idProp;  2 usages
    private int posProp;  2 usages
    private String nombreProp;  2 usages
    private int precio;  2 usages
    private int renta;  3 usages
    private String duenio;  3 usages
    private int casas;  3 usages
    private boolean estaHipotecada;  3 usages
public void construirHotel(Player_21142512_IgnacioTapia jugadorHotel,  2 usages  Damegoll
                           Property_21142512_IgnacioTapia propiedadCambiar,
                           int maxCasas){
    if (propiedadCambiar.getCasas() == maxCasas){
        Hotel propiedadHotel = new Hotel(
                propiedadCambiar.getIdProp(),
                propiedadCambiar.getNombreProp(),
                propiedadCambiar.getPosProp(),
                propiedadCambiar.getPrecio(),
                propiedadCambiar.getRenta(),
                propiedadCambiar.getDuenio(),
                propiedadCambiar.getCasas(),
                propiedadCambiar.isEstaHipotecada()
        );
        int dineroTemp = jugadorHotel.getDinero();
        propiedadHotel.setCasas(0);
        propiedadHotel.aumentarHoteles();
        jugadorHotel.setDinero(dineroTemp - propiedadHotel.getRenta());
        jugadorHotel.getPropiedades().remove(propiedadCambiar);
        jugadorHotel.getPropiedades().add(propiedadHotel);
```

Anexo B

```java
public class Card_21142512_IgnacioTapia {  36 usages
    private int idCarta;  2 usages
    private String descCarta;  2 usages
    private AccionCarta accion;  4 usages
```

```java
public interface AccionCarta {
    void ejecutar(Player_21142512_IgnacioTapia jugador, Game_21142512_IgnacioTapia juego);  10 implementations
}
```

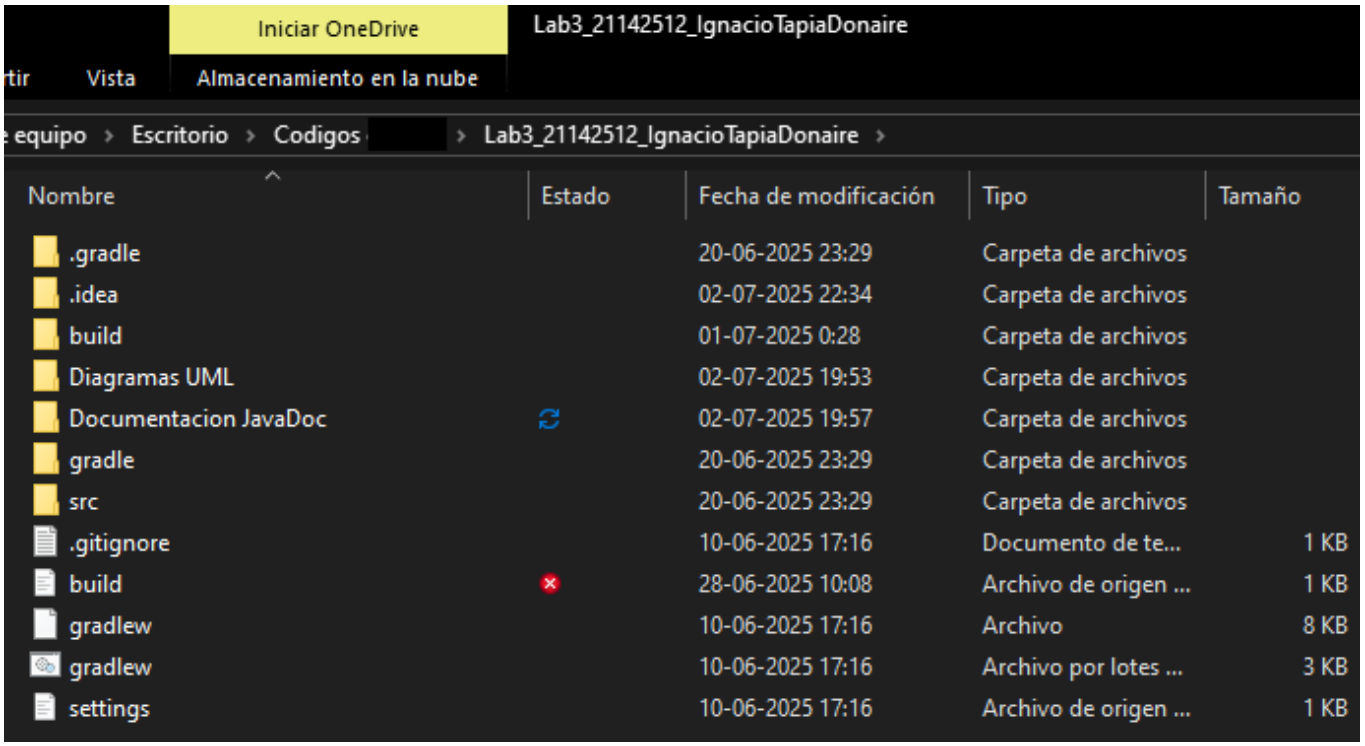
```java
public class AccionCarcelDirecto implements AccionCarta{  1 usage
    @Override
    public void ejecutar(Player_21142512_IgnacioTapia jugador,Game_21142512_IgnacioTapia juego){
        for (CasillasEspeciales casilla : juego.tablero.getCasillasEspeciales()){
            if (casilla.getNombreCasilla().equalsIgnoreCase( anotherString: "Carcel")){
                jugador.setPosicion(casilla.getPosCasillaEspecial());
            }
        }
        jugador.setDinero(1);
    }
}
```

```java
public class AccionMoverCasillaEsp implements AccionCarta{  2 usages
    private String nombreCasilla;  2 usages
     public AccionMoverCasillaEsp(String nombreCasilla){ this.nombreCasilla = nombreCasilla; }  2 usages
    @Override
    public void ejecutar(Player_21142512_IgnacioTapia jugadorMover, Game_21142512_IgnacioTapia juego) {
        for (CasillasEspeciales casilla : juego.tablero.getCasillasEspeciales()) {
            if (casilla.getNombreCasilla().equalsIgnoreCase(nombreCasilla)) {
                jugadorMover.setPosicion(casilla.getPosCasillaEspecial());
                casilla.accionar(jugadorMover, juego);
                break;
            }
        }
    }
}
```

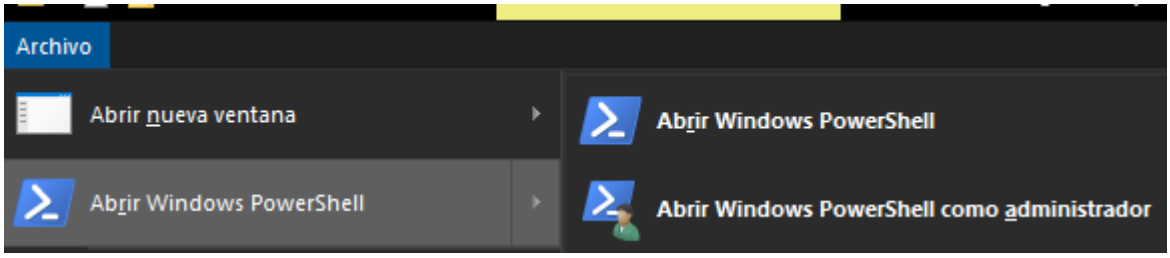
Anexo C

```java
public class Board_21142512_IgnacioTapia {  2 usages  Damegoll *
    private List<Property_21142512_IgnacioTapia> listaProp;  2 usages
    private List<Card_21142512_IgnacioTapia> listaCartas;  4 usages
    private List<CasillasEspeciales> casillasEspeciales;  2 usages
```

Anexo D



Anexo E



Anexo F