UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

PARADIGMAS DE PROGRAMACIÓN

Informe N°3: “CAPITALIA”

(Implementado en Java usando POO)

Alumno:

Angel Benavides Araya

Profesor:

Gonzalo Martinez Ramirez

02 de Julio de 2025

Índice de Contenidos

Índice de tablas y figuras

1. Introducción

El presente informe se enmarca en el desarrollo de *Capitalia*, una implementación del conocido juego de mesa *Monopoly*, que representa un entorno dinámico de compra, intercambio y gestión de propiedades entre jugadores. Esta adaptación no solo plantea un desafío desde el punto de vista técnico, sino que también invita a estudiar y reflexionar acerca de la representación de sistemas complejos con múltiples entidades, reglas y piezas en movimiento.

En este contexto, se ha optado por utilizar el paradigma de Programación Orientada a Objetos (POO), debido a su capacidad para representar entidades del mundo real mediante clases, objetos y relaciones jerárquicas. La POO permite encapsular comportamientos, abstraer componentes y facilitar la interacción entre elementos del sistema de forma coherente y mantenible. Así, conceptos como jugadores, propiedades, cartas o el tablero se modelan como objetos que interactúan según las reglas del juego.

Más allá de la implementación técnica, el desafío consistió en traducir una experiencia lúdica y análoga a una estructura formal y digital, conservando su lógica interna y asegurando su funcionalidad en un entorno programático. Este ejercicio no sólo pone a prueba habilidades de diseño e implementación, sino también la capacidad de pensar en términos de objetos, responsabilidades y relaciones, elementos centrales del paradigma orientado a objetos.

1.1 Descripción del problema

El problema a abordar consiste en la implementación digital del juego Capitalia (clon de Monopoly) utilizando POO de manera que se logre como mínimo una simulación básica del juego. Esto implica: representación de jugadores, tablero, propiedades, interacciones (compra/venta), cartas, eventos, casillas especiales y transacciones.

Además, se debe considerar una jugabilidad dinámica; a diferencia del Monopoly tradicional, al inicio de la partida se definen diversos parámetros que varían en cada juego, como por ejemplo: cantidad de propiedades, número máximo de casas/hoteles, impuesto inicial, entre otros.

Existen ciertas limitaciones asociadas a la implementación total de un clon tipo *Monopoly*, lo que implica desafíos adicionales que deben ser considerados en el planteamiento de la solución.

1.2 Descripción del paradigma

La programación orientada a objetos (POO) es un paradigma que organiza el software en torno a objetos que combinan datos (atributos) y comportamientos (métodos). Cada objeto es una instancia de una clase, que actúa como molde para definir tipos de entidades con características y acciones comunes. Este enfoque nació con el lenguaje Simula en 1962 y ganó popularidad con Smalltalk en 1972. Desde entonces, se ha convertido en uno de los paradigmas más usados en la industria del software, sobre todo en el desarrollo de sistemas complejos y mantenibles. Lenguajes como Java, C#, Python y Kotlin lo aplican ampliamente, y es el estándar en muchas empresas que desarrollan aplicaciones web, móviles, de escritorio o sistemas financieros.

Entre sus conceptos clave está la clase, que encapsula datos y operaciones relacionadas, promoviendo una estructura coherente y cohesiva. Al instanciar una clase se crean objetos con estado propio que interactúan entre sí según reglas bien definidas.

La herencia permite establecer relaciones jerárquicas entre clases, facilitando la reutilización de código. Una clase hija puede heredar atributos y métodos de una clase padre, y redefinir comportamientos mediante la sobreescritura. El polimorfismo, por su parte, permite que distintos objetos respondan de manera específica a un mismo método, ya sea mediante sobrecarga (en tiempo de compilación) o sobreescritura (en tiempo de ejecución).

También destacan conceptos como las interfaces y clases abstractas, que definen contratos de comportamiento sin imponer una implementación concreta, fomentando flexibilidad y bajo acoplamiento. Además, las relaciones entre clases (dependencia, asociación, agregación y composición) permiten modelar diferentes niveles de conexión y control sobre el ciclo de vida de los objetos.

Para representar estas estructuras de forma visual, se utiliza el lenguaje de modelado UML, lo que facilita el diseño y la comunicación de arquitecturas orientadas a objetos.

2. Desarrollo

2.1 Análisis del problema

Como ya se menciona, el problema consiste en implementar una versión digital del juego Monopoly, cumpliendo con 26 requerimientos funcionales obligatorios y esenciales Estos RF's abarcan desde la creación de clases básicas hasta el manejo completo del flujo del juego y la interacción entre sus componentes. También se incluyen 3 RF's opcionales.

Una primera dificultad está en la correcta estructuración de los TDAs principales: Juego, Tablero, Jugador, Propiedad, Hotel, Carta, CartaComunidad, CartaSuerte (RF01, RF03–RF10) y otras posibles clases adicionales. Si bien la implementación técnica de constructores y atributos es directa, el problema está en modelar bien las relaciones entre clases (herencia, composición, etc.) para que el resto de las funcionalidades se pueda apoyar sobre esa base sin generar acoplamientos innecesarios. Además, esto obliga a pensar detenidamente en el 'que' (estructura y relaciones) antes de lanzarse al 'como' (código como tal).

Por otro lado, el RF02 exige construir un menú interactivo que permita ejecutar todas las funcionalidades del juego desde consola. Este punto, aunque sencillo desde lo técnico (uso de scanner para recibir inputs, ciclos y condicionales), implica una integración limpia del sistema. No basta con que las funciones existan: deben estar correctamente conectadas para mantener la lógica del turno y reflejar los cambios de estado del juego de manera clara y continua. Además, se debe implementar un apartado para poder verificar el correcto funcionamiento de cada RF por separado, no solo el flujo principal del juego.

En cuanto al flujo del juego, hay RFs como cargar datos iniciales (RF11), agregar propiedades o jugadores (RF12, RF13), o identificar el jugador actual (RF14), que parecen simples pero implican mantener el estado del juego sincronizado. Desde ahí se habilitan acciones como lanzar dados (RF15), mover al jugador (RF16), y realizar acciones como comprar propiedades, calcular y pagar renta, construir hoteles o hipotecar (RF17–RF22). Todas estas funciones se cruzan entre sí y modifican atributos compartidos por varias clases, por lo que definir en qué clase ubicar cada método y cómo hacer que se comuniquen correctamente no es trivial.

El RF25, que corresponde a ejecutar un turno completo, representa el mayor punto de integración. No solo implica combinar varias acciones anteriores, sino también mantener la secuencia lógica del juego, asegurando que el estado de cada objeto se actualice correctamente. Si el diseño inicial no es claro o modular, esta función se vuelve difícil y podría volverse engorrosa de implementar.

El principal desafío de este laboratorio no está en la dificultad de las funciones por separado, sino en coordinar el comportamiento de múltiples objetos dentro de un flujo controlado, manteniendo la coherencia del sistema completo. El paradigma orientado a objetos ayuda a estructurar esa lógica, pero también exige tomar decisiones claras sobre diseño y responsabilidad de cada clase.

2.2 Diseño de la solución

Inicialmente se hizo un UML draw.io (ver figura). Allí solo aparecían las entidades básicas y sus relaciones de herencia o composición, sin entrar en mucho detalle de los métodos, solo los "esenciales" ese esqueleto sirvió de guía durante toda la implementación. Casi todas las clases: Juego, Tablero, Jugador, Casilla y sus derivadas, Carta y los mazos se implementaron en el código tal cual estan, con la única variación de desglosar la clase CasillaEspecial en clases concretas cuando cada tipo necesitaba un comportamiento propio en su método acción.

En el código la aplicación quedó dividida, de forma casi espontánea, en dos partes Primero la parte de la lógica detrás de los RF's y de la lógica general del juego (en particular la manera en la que interaccionan los atributos y los métodos). Por otra parte la parte de interacción con el usuario se redujo a la clase Main en vez de crear una clase aparte, ahí se implementó un menú de consola que maneja todas las entradas por medio de un único Scanner, haciendo uso de librerías propias de Java.

Cada requisito funcional terminó convertido en una pieza concreta del modelo. El tablero, por ejemplo, se implementó como un objeto Tablero que contiene una lista de Casilla; el id de cada casilla coincide con su posición en esa lista, así que ya no es necesario un atributo posicion explícito como en laboratorios anteriores. Hotel hereda de Propiedad, el enunciado exige literalmente que sea "un tipo de propiedad" y esa elección evita repetir campos como precio o hipotecas. Además se implementó un método para que las propiedades se transformen en Hotel si el jugador construye uno. Las cartas siguen la misma idea: Carta es abstracta y sus dos subclases únicamente redefinen aplicarEfecto(). Para gestionarlas se creó en la clase de Mazo unas listas de cartas nuevas y cartas usadas y un método barajar que recicla las cartas cuando se agotan.

Hubo momentos en que resultó difícil mantener un diseño limpio, a veces un método necesitaba datos alojados en clases que en el diagrama original no se relacionaban, y hubo que exponer algún getter extra o aceptar un vínculo adicional para no detener el avance ni cambiar sustancialmente el modelo. También la presión del tiempo llevó a generar algo de código Spaghetti.

Para los dados se usó java.util.Random, produciendo valores entre 1 y 6. La renta se calcula según la fórmula del enunciado: precio base más un 20 % por cada casa; si existe hotel, el resultado se duplica. El turno completo quedó encapsulado en un solo método: lanzar los dados, mover al jugador, ejecutar la acción de la casilla y verificar si la bancarrota global —todos menos uno sin dinero— obliga a cerrar la partida. La interfaz de consola se construyó con un switch/case y evita ciclos ocultos: cada opción llama a un submétodo visible, lo que facilita las pruebas de los RF de forma independiente.

En resumen, la arquitectura final conserva la claridad del diseño inicial, aunque algunos compromisos prácticos sobre todo en Juego y en el intercambio de datos entre clases alejaron el código de la idea original. Las repercusiones de esas elecciones, junto con las mejoras pendientes, se discuten en las conclusiones.

2.3 Aspectos de implementación

Todo el código fue desarrollado en Java 11 (OpenJDK 11.0.27), sin emplear bibliotecas externas. Se utilizó Gradle 8.1.4 mediante Gradle Wrapper, con la configuración por defecto generada por IntelliJ IDEA Community Edition. El desarrollo se realizó en dicha plataforma, estructurando el proyecto como una aplicación Gradle desde el inicio. El entorno de ejecución fue un notebook Lenovo ThinkPad X380 Yoga con sistema operativo Linux Mint 22.1 x86_64 y entorno de escritorio Cinnamon 6.4.8.

2.4 Instrucciones de uso

Para ejecutar el código, todos los archivos deben mantenerse dentro del mismo directorio, ya que forman parte de un mismo paquete. En particular, la clase Main requiere acceso a funciones y clases definidas en los demás archivos, debido a la existencia de relaciones de herencia, clases abstractas y composición entre objetos.

El programa puede ejecutarse de dos formas. La primera, y más directa durante el desarrollo, es a través del software IntelliJ IDEA. Para ello, se debe abrir el proyecto como una carpeta Gradle, navegar hasta el archivo Main.java , ubicado en ‘CodigoLab3/src/main/java/org/benavidesangel/lab3/Main’, y ejecutar con la flecha verde ubicada en la esquina superior. Esta opción es independiente del sistema operativo utilizado.

La segunda forma, y la solicitada en el enunciado, es mediante el uso de Gradle desde consola. Para ello, se debe abrir una terminal en el directorio raíz del proyecto y ejecutar los siguientes comandos:

1. ‘./gradlew build’ para compilar el proyecto en sistemas Linux, o ‘gradlew.bat build’ en Windows

2. ’./gradlew run’ para ejecutar el programa en Linux, o ’gradlew.bat run’ en Windows.

Al ejecutar el programa, aparecerá un menú interactivo en la consola con opciones para iniciar el juego o probar los distintos requerimientos funcionales de forma individual.
Cabe destacar que ambas formas de ejecución (por IntelliJ y por consola usando Gradle) fueron probadas exitosamente en un equipo con Linux.

Con respecto a los archivos html con la documentación en JavaDoc, estos se encuentran en:

‘doc/org/benavidesangel/lab3’

2.5 Resultados y autoevaluación

Durante la etapa de pruebas se ejecutaron los veintiséis requerimientos funcionales previstos (RF 01–RF 25) mediante el menú “Probar RF”. La mayor parte respondió como se esperaba: las clases se instancian sin error, el tablero se genera con quince propiedades y dos mazos de diez cartas cada uno, los dados arrojan valores válidos y la renta incorpora correctamente el 20 % por casa y el factor 2 cuando existe hotel. Funciones críticas para la representación del juego como: comprar propiedad, pagar renta, hipotecar, extraer carta, etc. actualizan el estado global y los saldos de los jugadores de forma consistente, de modo que es posible iniciar una partida con parámetros personalizados y avanzar turnos hasta que solo quede un jugador con dinero

Quedaron, sin embargo, aspectos a medio camino. El método jugarTurno (RF25) cumple la secuencia básica, pero todavía no contempla el efecto de dobles consecutivos ni la regla de ir a la cárcel tras tres dobles y la interacción con las cartas no tiene el funcionamiento esperado. Se trato de que el submenú para probar los RF fuera sencillo de usar y se trató de avisar mediante validaciones si falta algo para ejecutar ciertos RF’s, aún así, podrían quedar casos sin considerar y podrían arrojar errores.

Los aspectos a medias fueron: Pulir el turno completo con todas las reglas generales y por el lado de los RNF también quedó sin hacerse, por falta de tiempo, el diagrama UML “post-implementación” que debía reflejar el diseño final del código.

3. Conclusiones

Lo logrado de la implementación se considera medianamente logrado para una correcta representación de un clon de Monopoly en Java. Lamentablemente, el mal cálculo de tiempo que tomaría el laboratorio causó que faltara el último diagrama UML y el funcionamiento al 100% de RF25. Con respecto al paradigma, se recalca la capacidad de abstracción y representación de relaciones entre clases que representan elementos del mundo real, logrando simpleza en la manera de relacionar distintos módulos de código. Sale a flote el fácil uso de elementos propios de los TDA (Constructores, getters, setters) y la reutilización de código mediante la herencia, la capacidad de escribir métodos que cumplen funciones similares a través de técnicas como el polimorfismo, tanto de métodos como de clases.

Sin embargo, aunque el POO favorece la modularización y organización de código mediante diversas herramientas, esto puede llevar a una mayor complejidad del sistema dependiendo de la representación escogida por el programador. La gran cantidad de relaciones entre clases puede desorientar tras unas cuantas capas de abstracción, y es fácil que aparezca sobre-ingeniería: clases creadas sólo para sostener una jerarquía demasiado ambiciosa o atributos que terminan viviendo en la clase equivocada. En el proyecto ese riesgo se materializó en el objeto Juego, que terminó

albergando más responsabilidad de la prevista, además de no saber para dónde avanzar en el proceso de codificación tras cometer errores de diseño.

El contraste con los laboratorios anteriores es evidente tanto en el aspecto técnico como en la experiencia personal. Con Prolog la lógica consistía en hechos y reglas: resultando en un programa elegante pero complicado de pensar. La sintaxis declarativa de Prolog y la manera de ejecutar el código resultó frustrante a comparación con el paradigma orientado a objetos y Java. Con Racket la diferencia es clara debido a la inmutabilidad de las variables y la necesidad de utilizar recursiones de cola y natural constantemente. Algo tan simple en un lenguaje imperativo como actualizar la posición de un jugador implicaba encadenar varias funciones de primer orden para realizar algo básico. Frente a estos enfoques, la orientación a objetos (y Java) se sintió más natural para una implementación de este estilo. Si bien tiene su complejidad, la curva de aprendizaje fue menor que con otros paradigmas. Finalmente, el resultado es un clon de Monopoly que cumple con la mayoría de RF, aunque quedaron pendientes ciertas reglas para que sea completamenet funcional y el diagrama UML final. La POO ofrece una base más intuitiva y modular que las obtenidas en los paradigmas lógico y funcional.

4. Bibliografía

[1] Martínez, G. (2025b). Técnicas OOP, parte 1: Tipos de relaciones [Apunte de clase no publicado]. Departamento de Ingeniería Informática, Universidad de Santiago de Chile.

[2] Martínez, G. (2025a). Técnicas OOP, parte 2: Técnicas de polimorfismo [Apunte de clase no publicado]. Departamento de Ingeniería Informática, Universidad de Santiago de Chile.

5. Anexos

Tabla 1. Descripción de requisitos funcionales

RF	Descripción
1	Clases y estructuras que forman el programa.
2	**Menú interactivo por terminal**
3	**TDA Jugador - Constructor**
4	**TDA Propiedad - Constructor**
5	**TDA Hotel - Constructor**
6	**TDA Carta - Constructor**
7	**TDA CartaComunidad - Constructor**
8	**TDA CartaSuerte - Constructor**
9	**TDA Tablero - Constructor**
10	**TDA Juego - Constructor**
10b	**TDA Juego - Constructor**
11	**Cargar datos iniciales**
12	**Agregar Propiedad**
13	**Agregar Jugador**

14	**Obtener Jugador Actual**
15	**Lanzar dados**
16	**Mover Jugador**
17	**Comprar Propiedad**
18	**Calcular Renta Propiedad**
19	**Calcular Renta Jugador**
20	**Construir Hotel**
21	**Pagar renta**
22	**Hipotecar Propiedad**
23	**Extraer Carta**
24	**Verificar bancarrota**
25	**Jugar Turno**

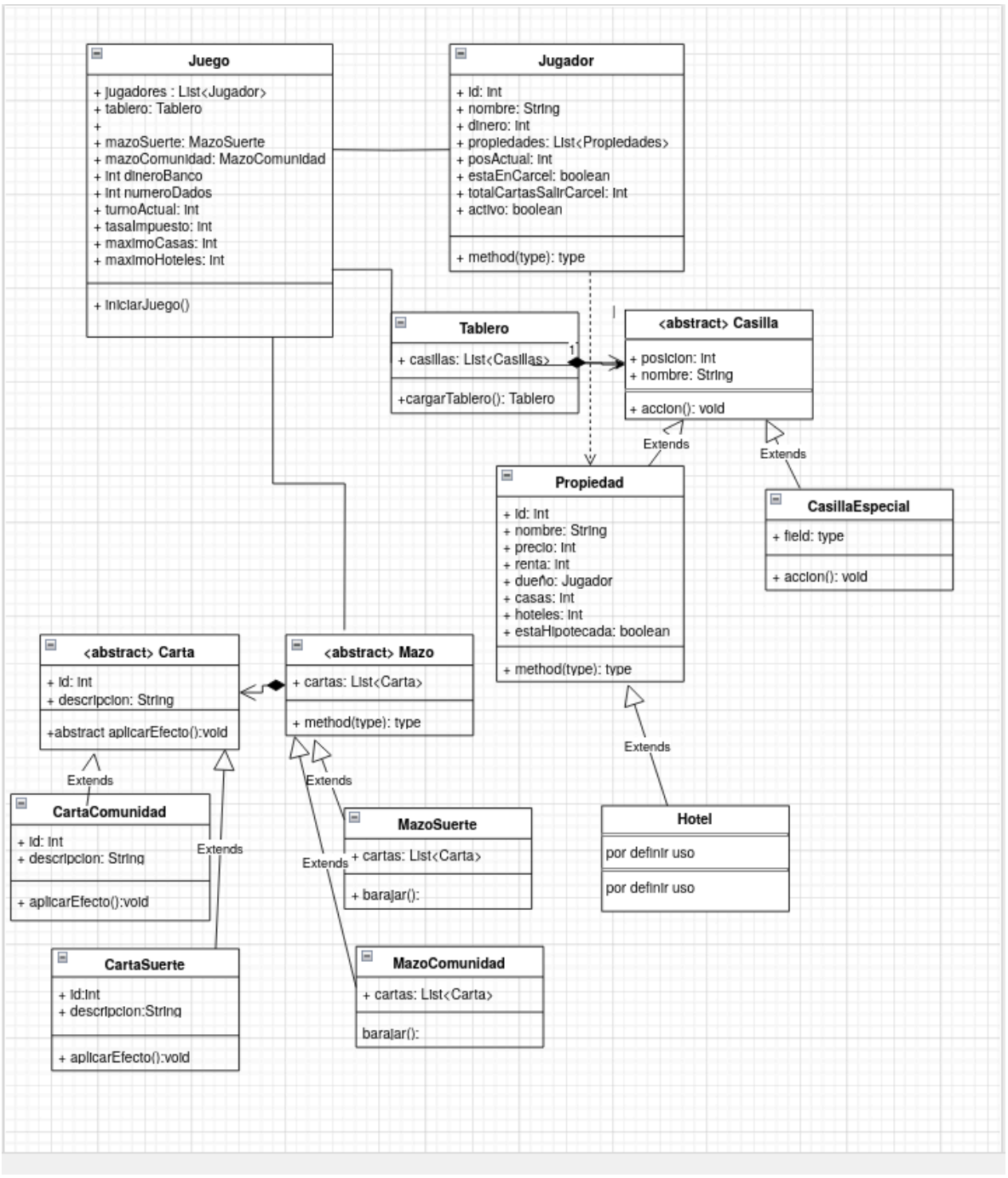

Figura 1. UML Inicial