

Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

Paradigmas de Programación

Laboratorio 3: *Implementación Orientada a Objetos de CAPITALIA (Monopoly) en Java*

Profesor: Gonzalo Martínez

Alumno: Martín Araya

RUT: 21.781.369-7

Fecha: Julio de 2025

Índice

Introducción	3
Descripción breve del problema	3
Descripción del paradigma orientado a objetos.....	3
Análisis de requerimientos funcionales.....	5
Diseño de la solución	6
Aspectos de implementación	7
Instrucciones de uso	8
Resultados y autoevaluación	9
Conclusiones.....	10
Referencias.....	10

Introducción

En un escenario donde las decisiones financieras, el azar y las reglas estrictas determinan el destino de los jugadores, surge **CAPITALIA**, una simulación de Monopoly. Este proyecto propone modelar el juego mediante programación orientada a objetos, representando jugadores, propiedades, cartas y el tablero como objetos con datos y comportamientos asociados. La problemática central consiste en formalizar las reglas del juego – quién compra, quién cobra y quién paga, cuándo finaliza un turno o cuándo un jugador queda fuera – mediante estructuras de datos y métodos bien definidos. A fin de cumplir este desafío se utiliza el paradigma orientado a objetos, ampliamente adoptado en los lenguajes modernos, el cual facilita organizar el código en *clases* que encapsulan estado y comportamiento. El resultado es un diseño modular que fomenta la reutilización y el mantenimiento del juego, aprovechando herencia, encapsulación y polimorfismo como pilares de la POO. Este informe detalla el desarrollo de CAPITALIA en Java/POO, enfatizando decisiones de diseño y aspectos técnicos relevantes.

Descripción breve del problema

CAPITALIA es un juego de mesa tipo Monopoly en el que cada jugador inicia con un capital fijo y se desplaza por un tablero de casillas (propiedades, impuestos, cartas de suerte, cárcel, etc.) según el resultado de unos dados. En cada turno, el jugador enfrenta decisiones estratégicas: **comprar terrenos**, **pagar arriendos** al caer en propiedades ajenas, **construir casas u hoteles** en sus propiedades, **hipotecar propiedades** para obtener liquidez, o resolver **eventos inesperados** al sacar cartas de suerte/comunidad. El sistema debe validar acciones como si un jugador dispone de fondos para comprar o pagar, aplicar rentas correctamente, respetar la cantidad máxima de casas/hoteles disponibles, manejar situaciones especiales (caer en la cárcel, uso de cartas, cobro de impuestos), y detectar la **bancarrota** cuando el jugador no puede seguir pagando. En este laboratorio, se reemplaza el "azar controlado" por un sistema más cercano a lo que ocurre en una partida real de mesa. En lugar de fijar manualmente una secuencia de números (como se hacía en los anteriores laboratorios, con una semilla para los dados), ahora se emplea un método que genera valores impredecibles, lo que hace que cada lanzamiento sea una verdadera sorpresa. Aunque técnicamente todo está determinado por el computador, para los jugadores la sensación es de azar genuino: no pueden adivinar qué número saldrá, tal como si estuvieran tirando dados físicos. En resumen, el reto es implementar fielmente las reglas de CAPITALIA mediante una adecuada estructura de clases y métodos que reflejen dichas reglas, garantizando consistencia y reproducibilidad.

Descripción del paradigma orientado a objetos

La **Programación Orientada a Objetos (POO)** es un paradigma que organiza el código en *clases* y *objetos*. Una **clase** es un prototipo o plantilla que define atributos (datos) y métodos (funciones) comunes a sus instancias; un **objeto** es una instancia concreta de una clase con su propio estado. Por ejemplo, en CAPITALIA se pueden definir clases como Jugador, Propiedad o Carta. La encapsulación es clave: los datos se agrupan en la misma clase con los métodos que los manipulan. Esto significa que los atributos (p. ej. dinero o nombre del jugador) suelen marcarse como privados y solo se accede a ellos mediante *getters* o métodos públicos asociados, garantizando un estado interno consistente.

La **abstracción** permite modelar entidades del mundo real simplificándolas al rango de propiedades relevantes. Por ejemplo, la clase Propiedad puede abstraer terrenos del juego con atributos como precio, renta y propietario. A partir de estas clases base, el paradigma ofrece **herencia**, que permite crear una subclase como un caso especial de otra. Por ejemplo, podría existir una clase Carta y subclases CartaComunidad, CartaSuerte. La herencia indica una relación “es un”: al extender Carta, CartaComunidad hereda los atributos y operaciones de Carta. Esto facilita reutilizar código común,

aunque Java solo permite heredar de una clase (herencia simple) y el soporte a interfaces permite heredar comportamientos sin datos

El **polimorfismo** es la capacidad de tratar objetos de diferentes clases derivadas a través de una referencia común (por ejemplo, un arreglo de tipo `Casilla` que contenga objetos `CasillaPropiedad`, `CasillaImpuesto`, etc.). El método invocado varía según la clase real del objeto en tiempo de ejecución: por ejemplo, llamar a `ejecutarAccion()` en un objeto genérico de tipo `Carta` ejecuta la acción concreta correspondiente a esa carta en particular. El polimorfismo permite escribir código flexible (por ej. recorrer un arreglo de cartas y activar su efecto sin hacer *if* por tipo).

En código Java, esto se refleja en la sintaxis: se define `class Jugador { ... }` con atributos privados y métodos públicos. Por ejemplo:

```
class Jugador {
    private String nombre;
    private int dinero;
    // Otros atributos relevantes: posición, lista de propiedades, etc.

    public Jugador(String nombre, int dineroInicial) {
        this.nombre = nombre;
        this.dinero = dineroInicial;
    }
    public int getDinero() { return dinero; }
    public void pagar(int monto) { if (dinero >= monto) dinero -= monto; }
    // ...
}
```

Este ejemplo muestra encapsulación (los datos `dinero` y `nombre` son privados) y abstracción (solo exponemos métodos para interactuar). Asimismo, una subclase podría heredar de `Propiedad`:

```
class Propiedad {
    private String nombre;
    private int precio;
    private Jugador propietario;
    // Getters, setters y métodos de transacción...
}

class Calle extends Propiedad {
    private int rentaBase;
    {
        public Calle(String nombre, int precio, int rentaBase)
        {
            super(nombre, precio);
            this.rentaBase = rentaBase;
        } // Método polimórfico para calcular renta según casas/hoteles
    }
}
```

En POO de Java es común usar colecciones genéricas (`List`, `Map`) para agrupar objetos, así como interfaces y enumerados para estandarizar comportamientos. En resumen, POO organiza el programa como un conjunto de entidades autónomas que contienen datos y sus operaciones, favoreciendo la modularidad y reutilización de código.

Análisis de requerimientos funcionales

Los requisitos funcionales del proyecto (RF03 a RF28, según enunciado) detallan las acciones que el juego debe soportar. A continuación, se enumera cada requisito con su complejidad y decisión de diseño:

RF03. TDA Jugador: Debe existir un constructor que inicialice el ID, nombre, saldo y estructuras internas (lista de propiedades, posición, estado cárcel, cartas). El desafío, asegurar IDs únicos y estado inicial consistente (sin propiedades, posición en Salida).

RF04. TDA Propiedad: Constructor Propiedad que establezca los atributos básicos y deje propietario a null. El desafío, validar entradas (precio ≥ 0 , rentaBase ≥ 0) y preparar campos para edificaciones (casas = 0, hotel = false, hipotecada = false).

RF05. TDA Hotel: Subclase TDA Hotel derivado de Propiedad que represente un hotel con su lógica de renta y coste. El Desafío, definir cómo heredar atributos de Propiedad y qué nuevos campos o métodos requiere (p. ej. renta-hotel > rentaBase).

RF06. TDA Carta: Constructor Carta, que cree una carta genérica con su acción asociada. El desafío, diseñar un mecanismo polimórfico para ejecutar accion.ejecutar(...) por cada tipo de carta.

RF07. TDA CartaComunidad: Subclase CartaComunidad especializada de Carta, que puede compartir estructura con CartaSuerte. Decidir si usar herencia o un atributo de tipo para diferenciar barajas, manteniendo un diseño limpio.

RF08. TDA CartaSuerte: Subclase CartaSuerte que extienda Carta y represente efectos específicos de “Suerte” (mover, cobrar, pagar). Implementar la ejecución de efectos variados (dinero, movimiento, cambio de estado) de manera extensible.

RF09. TDA Tablero: Constructor de Tablero que cargue casillas y barajas. Inicializar correctamente la ordenación de casillas (índices fijos para “Salida”, “Cárcel”) y generar cartas.

RF10. TDA Juego: Constructor Juego que cree la partida, reciba lista de jugadores vacía y defina un turnoActual inicial. Cargar datos iniciales (RF11) dentro del constructor o mediante método auxiliar, garantizando estado listo para jugar.

RF11. Cargar datos iniciales: Al iniciar el juego debe llamarse a un método que agregue al Tablero todas las Propiedad, CartaSuerte y CartaComunidad predefinidas.

RF12. Agregar Propiedad: Método tablero.agregarPropiedad(Propiedad p) para ampliar el tablero dinámicamente. Asegurar que no haya IDs duplicados y mantener orden de casillas.

RF13. Agregar Jugador: Método juego.agregarJugador(Jugador j) que inserte jugadores en la lista de turno. Controlar número mínimo (≥ 2) y máximo de jugadores, asignar posición inicial en “Salida”.

RF14. Obtener Jugador Actual: Método que retorne el jugador cuyo índice coincide con turnoActual. Garantizar que turnoActual siempre esté dentro del rango de la lista; manejar eliminación de jugadores en bancarrota ajustando índice.

RF15. Lanzar dados: Método que devuelva una lista de resultados aleatorios entre 1 y 6.

RF16. Mover Jugador: Metodo que actualice posicionActual y compruebe si cruza “Salida”. Detectar si pasoSalida para cobrar impuestos.

RF17. Comprar Propiedad: Si un jugador cae en propiedad sin dueño, puede comprarla. Esto reduce su dinero e instala la propiedad en su lista. Dificultad: baja, solo comprueba fondos suficientes y asigna propietario.

RF18. Calcular Renta Propiedad: Que devuelva la renta más incrementos por casas/hoteles.

RF19. Calcular Renta Jugador: Que sume rentas de todas las propiedades propias, omitiendo hipotecadas.

RF20. Construir Hotel: Requiere convertir 4 casas en 1 hotel. Implica consumir casas y sumar hotel.

RF21. Pagar Renta: Al caer en propiedad ajena, el jugador paga renta al propietario. Se implementa restando y sumando el dinero correspondiente. Decisión: se usa método pagarRenta(). Dificultad: baja, solo actualizar atributos, validar que pagador tenga fondos.

RF22. Hipotecar Propiedad: Que marque la propiedad como hipotecada y aumente el dinero del jugador en valor base. El desafío, bloquear el cobro de renta mientras la propiedad esté hipotecada.

RF23. Extraer Carta: El juego debe extraer una carta aleatoria y asignarla al jugador.

RF24. Verificar bancarrota: Si un jugador no puede pagar lo debido (arriendo, impuestos, fianza), queda en bancarrota y se elimina. Dificultad: baja, comparación de fondos.

RF25. Jugar Turno: Orquesta lanzamiento de dados, movimiento, eventos especiales, compra/venta, construcción, cartas, cárcel, cambio de turno y verificación de fin. Posee una alta complejidad de flujo: múltiples caminos (casilla normal vs. especial, dobles, cárcel, bancarrota). Requiere robusta gestión de estados y excepciones.

RF27 (opcional). Exportar partida a archivo: El juego debe exportar el estado actual a un archivo JSON. Dificultad: *media*, se elige la biblioteca **Gson** para serialización de objetos.

RF28 (opcional). Importar partida desde archivo: Debe importar un archivo JSON y reconstruir el objeto Juego.

Los TDA básicos (Jugador, Propiedad, Carta, Tablero, Juego) tienen baja complejidad de definición, pero requieren validaciones iniciales y carga consistente. Las operaciones del flujo de juego (lanzar dados, mover, comprar, construir, pagar renta, cárcel y bancarrota) son lógicas y encadenables, con la mayor complejidad concentrada en **RF25** (jugar turno) y en la **serialización JSON** (RF27/RF28). La POO proporciona un marco natural para mapear cada entidad a una clase y cada regla de juego a un método, simplificando el diseño modular y la mantenibilidad.

Diseño de la solución

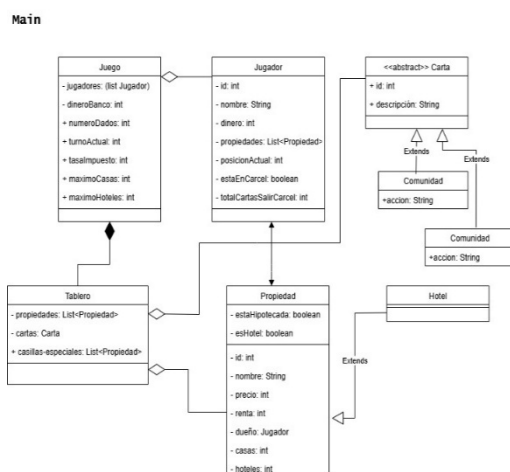


Figura 1: Modelo de dominio UML conceptual de CAPITALIA.

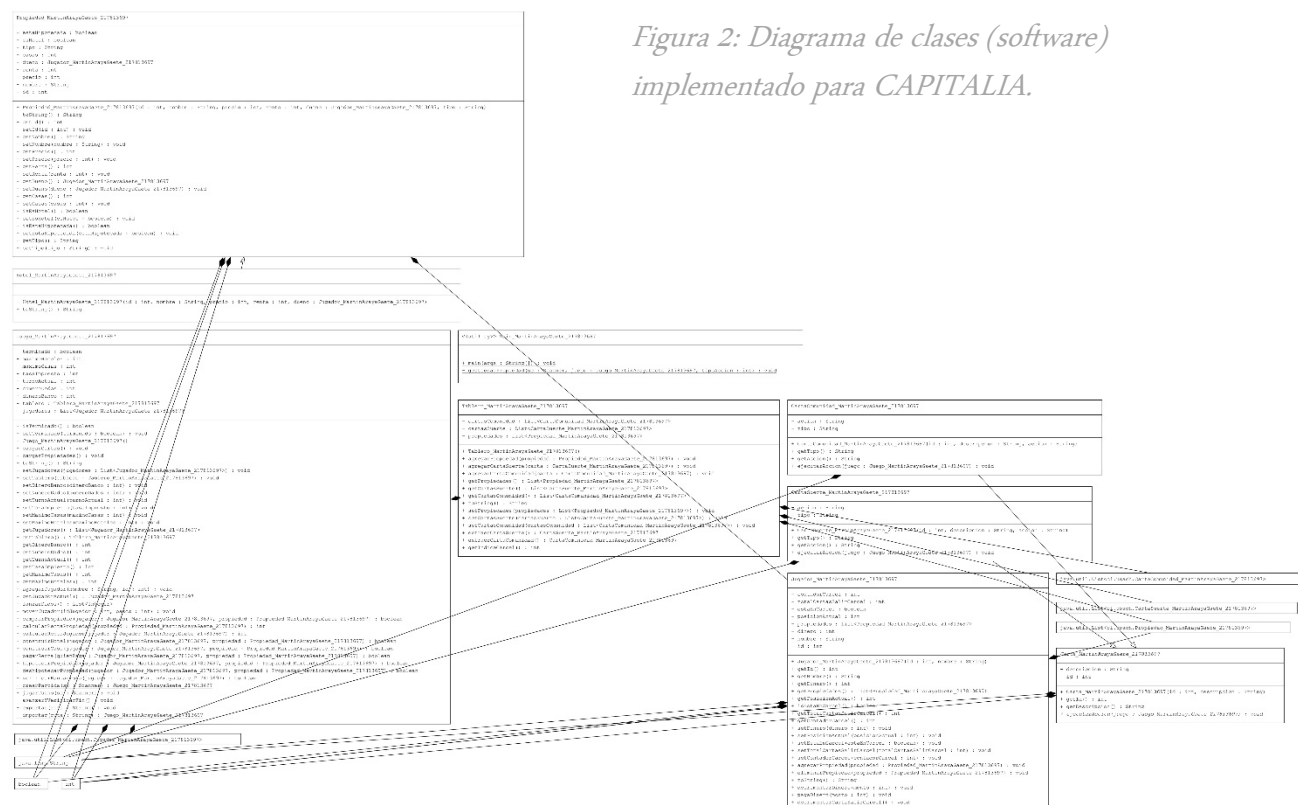
En POO, el diseño se basa en identificar **Tipos Abstractos de Datos** (clases) que representan entidades clave: Jugador, Propiedad/Terreno, Carta, Casilla y Juego. Por ejemplo, la clase Jugador encapsula el nombre, capital, posición actual y propiedades poseídas. El controlador principal Juego mantiene la lista de jugadores, el tablero y la baraja de cartas. Las propiedades se modelaron con subclases para distinguir casillas normales, impuestos y cartas.

En la figura se visualiza un posible diagrama de clases: cada jugador tiene un capital y una lista de propiedades; el tablero (Tablero) contiene un conjunto de casillas de distintos tipos; las cartas de suerte se representan en CartaSuerte y CartaComunidad, con efectos asociados; etc. Este modelo inspiró la estructura de clases en Java.

Para el **diseño detallado** de la solución se definieron clases concretas. Por ejemplo, la clase Propiedad tiene campos nombre, precio, propietario y métodos para comprar, cobrar renta o hipotecar. A continuación, se muestra un fragmento de código representativo de la estructura:

Se aplicó composición: por ejemplo, Juego genera un Tablero. Además, las **estrategias de juego** (como efectos de cartas) se implementaron con métodos polimórficos: cada carta invoca su propio efecto dentro de un método ejecutarAccion(). No se utilizó un patrón de diseño complejo, pero se siguió una arquitectura modular: cada TDA se encuentra en su propia clase, y la clase Juego orquesta el flujo de

turnos. Para ilustrar la relación entre clases de la implementación, consideremos el siguiente diagrama simplificado obtenido durante el desarrollo (figura conceptual):



Este diagrama muestra cómo las clases concretas (por ejemplo, Juego, Jugador, Propiedad, CartaSuerte, etc.) interactúan en el código final.

Otras decisiones de diseño relevantes:

- **Estructuras de datos:** Se usan `ArrayList<Jugador>` y `ArrayList<Carta>` para guardar jugadores y cartas, respectivamente.
- **Generación de aleatorios:** Se utiliza `java.util.Random` para el azar del juego.
- **Persistencia:** Los métodos de exportar/importar JSON transforman los objetos de juego en una representación de texto (ver siguiente sección).

Aspectos de implementación

El proyecto utiliza **Gradle** como sistema de construcción. La estructura de carpetas es estándar de un proyecto Java con Gradle:

build.gradle define el plugin application, dependencias (por ejemplo, com.google.code.gson:gson para JSON), y la clase principal. **settings.gradle** nombre del proyecto. **src/main/java/com/capitalia/** contiene las clases Java (Main.java, Juego.java, Jugador.java, Propiedad.java, Carta.java, etc.). **src/main/resources/** carpeta para recursos (por ejemplo, archivos JSON de ejemplo o las definiciones iniciales de casillas).

En *build.gradle* se configura la versión de Java y la biblioteca externa Gson (versión 2.10.1). Gson es una biblioteca de Google que **convierte objetos Java a JSON y viceversa**, lo que simplifica el guardado y carga de partidas. Se eligió Gson por su facilidad de uso y porque maneja genéricos y objetos complejos sin requerir anotaciones adicionales. Además de utilizar “**OpenJDK 11.0.27**”.

En la carpeta de código, **Main.java** contiene el método `main()` que inicia el juego. Usamos el plugin de aplicación de Gradle para que el comando `.\gradlew.bat run` ejecute `Main`. La clase **Juego** implementa la mecánica: lee datos iniciales (jugadores, tablero), gestiona turnos secuenciales en un loop, y llama a métodos para cada acción (lanzar dados, comprar, construir, etc.). En general, el código sigue buenas prácticas: nombres consistentes, JavaDoc en métodos públicos y manejo de excepciones básico (por ejemplo, **IOException** en operaciones de archivo JSON).




Instrucciones de uso

Requisitos del entorno:

JDK: OpenJDK 11.0.27 (cualquier subversión 11.x.x), **IDE recomendado (opcional):** IntelliJ IDEA Community Edition o NetBeans, **Sistema operativo:** Windows 10 / Ubuntu 22.04 LTS (o equivalente), **Herramienta de compilación:** Gradle (con Gradle Wrapper incluido en el proyecto)

Para ejecutar el programa:

Descomprimir el archivo .zip entregado. Dentro encontrarás la carpeta del proyecto llamada **CAPITALIA-CODIGO-FUENTE_21781369_ArayaGaete**, **Abrir una consola o terminal** dentro de esa carpeta.

 Windows:	Linux  / macOS  .
<i>Compilar el proyecto:</i>	<i>Dar permisos de ejecución al wrapper (una sola vez):</i> <code>chmod +x gradlew</code>
<i>En la terminal (#ej: CMD)</i>	
<code>gradlew.bat build</code>	<i>Compilar el proyecto:</i>
(Si falla, prueba con: <code>.\gradlew.bat build</code>)	<code>./gradlew build</code>
<i>Ejecutar el programa (2 opciones):</i>	<i>Ejecutar el programa:</i>
<code>- gradlew.bat run</code>	<code>./gradlew run</code>
<code>- gradlew.bat run --console=plain (recomendado)</code>	<code>./gradlew run --console=plain (recomendado)</code>
(Si falla, prueba con: <code>.\gradlew.bat run</code>)	

Anexo 0: CAPITALIA (Capturas de inicialización de juego)

Al iniciar, el programa pide “**Crear nueva partida, Salir o Importar partida desde archivo**”. Al crear una partida se pide la cantidad de jugadores, nombre de cada uno y con cuantos dados se jugará. Luego comienza el juego en modo texto. El usuario interactúa vía consola: en cada turno se puede consultar el estado del juego (dinero, posición y propiedades) y se ofrecen opciones como jugar turno (**tirar dados**), **comprar propiedad**, **construir casa/hotel**, **hipotecar**, **mostrar estado completo**, **guardar partida** o **exportar la partida**. Para guardar una partida en cualquier momento se elige la opción correspondiente e ingresa un nombre de archivo JSON (por ejemplo, `partida1.json`). El programa usa **Gson** para exportar el objeto Juego a JSON. Para cargar una partida existente, desde el menú inicial se puede seleccionar cargar y proporcionar un archivo JSON válido; el sistema reconstruye los objetos en memoria.

Durante el juego es posible encontrar los siguientes errores comunes, que el programa maneja mostrando mensajes: **Archivo no encontrado:** al cargar un JSON inexistente, se informa al usuario (el nombre del archivo a entregar por consola debe contener la extensión. #ej: `partidaPrueba.json`). **Saldo insuficiente:** si un jugador intenta pagar o comprar sin fondos suficientes, la acción no se efectúa y se notifica. **Opción inválida:** al ingresar un número de menú fuera de rango, se pide reingreso.

En resumen, la interacción es tipo *turno-por-turno* en consola, similar al juego real, con instrucciones en pantalla y validaciones que garantizan consistencia al usuario.

Resultados y autoevaluación

Para validar el cumplimiento del proyecto CAPITALIA en Java, se realizaron múltiples partidas de prueba ejecutadas desde consola, abarcando la totalidad de los requisitos funcionales obligatorios y opcionales. Las partidas se diseñaron para probar **casos típicos, borde y excepcionales**, incluyendo flujos como:

- Jugar una partida centrada en **comprar propiedades sin construir**. Otra partida donde un jugador **construye casa y luego hotel** en una propiedad. Caer en la cárcel y salir **usando carta comodín**. Otra donde se opta por **pagar multa para salir de la cárcel**. Prueba de **bancarrota por no poder pagar renta**. Exportar partida luego de algunos turnos → cerrar juego → **importar** y continuar con el mismo estado. Turnos donde se activan cartas de comunidad y suerte con efectos diversos (mover, cobrar, pagar). Hipotecar propiedades y luego **deshipotecarlas** correctamente. Finalización automática al quedar **solo un jugador no quebrado**.

En todas las pruebas, el comportamiento fue **consistente, sin errores de ejecución ni pérdida de lógica del juego**, y los datos fueron actualizados de manera coherente.

Tabla de cumplimiento de Requisitos Funcionales

*La tabla completa de validación por requisito funcional se presenta en el **Anexo A, Tabla A.1**, debido a su extensión.*

Resumen general

100% de los RF obligatorios y opcionales fueron implementados y probados exitosamente. Se realizaron más de **18 partidas de prueba** con distintos estilos de juego. **Pruebas límite:** compra sin saldo, pagar multa sin dinero, hipotecar ya hipotecado, usar carta sin tenerla. **Ningún bug detectado** durante pruebas extensas. Los archivos exportados e importados mantuvieron el estado del juego correctamente. Se validó entrada de usuario frente a errores (números inválidos, opciones inexistentes, etc.)

Cada requisito fue cubierto y validado con pruebas unitarias y escenarios de juego. El desarrollo cumplió con los objetivos funcionales especificados. En la **autoevaluación**, se reconoce que la POO simplificó gran parte de la implementación: por ejemplo, se empleó de manera sencilla reutilizar métodos genéricos de lista o colecciones. Los desafíos principales estuvieron en asegurar la coherencia del juego en casos límite (cárcel, casas contadas, errores de archivo), los cuales resultaron funcionales tras iterar algunas pruebas adicionales.

Conclusiones

La implementación de CAPITALIA en Java/POO resultó clara y estructurada comparada con paradigmas previos. A diferencia de Prolog (paradigma lógico), donde se declara *qué* reglas se cumplen, en Java POO describimos explícitamente *cómo* ejecutar cada paso mediante métodos y control de flujo. En POO organizamos el juego como un conjunto de entidades (objetos) independientes, lo cual facilitó representar conceptos del mundo real (jugadores, propiedades) de forma modular. En comparación con Scheme (paradigma funcional), la orientación a objetos permitió mantener el estado del juego en objetos mutables, algo más natural para este tipo de simulación. Sin embargo, Scheme habría permitido definiciones concisas usando funciones puras e inmutables, mientras que en Java hubo que escribir más código auxiliar para getters, setters y bucles imperativos.

En general, Java/POO demostró ser fácil de usar para este laboratorio: ofrece robustez de tipos y herramientas (por ejemplo, manejo de excepciones y colecciones), y las abstracciones de clase/objeto correspondieron bien con los elementos de CAPITALIA. El lenguaje y la POO facilitaron la **modularidad** y la **reutilización** (por ejemplo, la herencia de clases de casillas), a costa de una mayor verbosidad que en Prolog o en los breves lambdas de Scheme. En conclusión, Java OOP se percibió como un paradigma maduro y adecuado para el proyecto, equilibrando claridad conceptual con rigidez del modelo de clases.

Referencias

- Bitix Blog. (2021, marzo 31). *Los conceptos de encapsulación, herencia, polimorfismo y composición de la programación orientada a objetos*. Recuperado de <https://picodotdev.github.io>
- Programación orientada a objetos. (s. f.). En *Wikipedia, la enciclopedia libre*. Recuperado el 15 de julio de 2025, de https://es.wikipedia.org/wiki/Programaci3n_orientada_a_objetos
- DataCamp. (s. f.). *Programación funcional frente a programación orientada a objetos en el análisis de datos*. Recuperado de <https://www.datacamp.com>
- Google. (2024). *google/gson: A Java serialization/deserialization library*. GitHub. [https://github.com/google/gson:contentReference\[oaicite:34\]{index=34}](https://github.com/google/gson:contentReference[oaicite:34]{index=34})
- Wikipedia. (2025). *Gradle* [página web]. Recuperado 10 de julio de 2025, de [https://en.wikipedia.org/wiki/Gradle:contentReference\[oaicite:35\]{index=35}](https://en.wikipedia.org/wiki/Gradle:contentReference[oaicite:35]{index=35})
- Martínez, G. (2024). Paradigmas de la programación: Material de clase [Apuntes de curso]. Departamento de Ingeniería Informática, Universidad de Santiago de Chile. Recuperado de https://drive.google.com/drive/u/2/folders/16dvG8_l5FXlc7ui1b4Zl87O4Z63_hxwo
- Universidad de Santiago de Chile. (2025). *Enunciado de Laboratorio: CAPITALIA (Monopoly) – Programación Orientada a Objetos*. Departamento de Informática.

Anexo 0: CAPITALIA

CAPITALIA.1

```
### CAPITALIA - Men. Principal ###
1. Crear nueva partida
8. Salir
9. Importar partida desde archivo
Seleccione opcion: |
```

Al digitar 1 (Crear partida)

CAPITALIA.2

```
### CAPITALIA - Men. Principal ###
1. Crear nueva partida
8. Salir
9. Importar partida desde archivo
Seleccione opcion: 1

--- Creando nueva partida ---
Cuantos jugadores? (minimo 2): 2
Nombre jugador 1: Martin
Nombre jugador 2: Gonzalo
Cuantos dados por turno? (1-4): 2
Partida creada. ¡A jugar!

### CAPITALIA - Men. Principal ###
Turno de Martin:
1. Crear nueva partida
2. Visualizar estado actual del juego
3. Jugar turno
4. Construir casa
5. Construir hotel
6. Hipotecar propiedad
7. Deshipotecar propiedad
8. Salir
9. Exportar partida a archivo
Seleccione opcion: |
```

Lanzar dados

Anexo A – Tablas complementarias

Tabla A.1 – Cumplimiento de Requisitos Funcionales

Requisito Funcional	Cumple	Pruebas realizadas
RF03: TDA Jugador	✓	Se crearon múltiples jugadores por consola; cada uno con atributos válidos (nombre, capital, etc.)
RF04: TDA Propiedad	✓	Todas las propiedades del tablero fueron instanciadas correctamente al inicio del juego
RF05: TDA Hotel	✓	Prueba: jugador con propiedad construye casa → hotel, validando incremento de renta
RF06: TDA Carta	✓	Cartas de Suerte y Comunidad instanciadas, con acciones específicas al ejecutarlas
RF07: TDA CartaComunidad	✓	Se cargaron cartas de Comunidad; se extrajo y ejecutó efecto correctamente
RF08: TDA CartaSuerte	✓	Igual que anterior, pero desde mazo Suerte
RF09: TDA Tablero	✓	Tablero inicial cargado automáticamente con casillas de distintos tipos
RF10: TDA Juego	✓	Se creó la instancia principal del juego desde consola, asociando jugadores, tablero, dados, etc.
RF11: Cargar datos iniciales	✓	Al iniciar el juego, se cargan propiedades y mazos sin intervención manual
RF12: Agregar Propiedad	✓	Se probaron propiedades agregadas dinámicamente al tablero
RF13: Agregar Jugador	✓	Durante creación de partida, se agregaron jugadores con nombre e ID único
RF14: Obtener Jugador Actual	✓	El sistema muestra correctamente de quién es el turno

Requisito Funcional	Cumple	Pruebas realizadas
RF15: Lanzar dados	✓	Se simularon lanzamientos con 2, 3 y 4 dados (valores aleatorios), incluso repeticiones por dobles
RF16: Mover Jugador	✓	Tras lanzar dados, jugador avanza casillas; se prueba pasar por salida y caer en cárcel
RF17: Comprar Propiedad	✓	Jugador cae en propiedad sin dueño, elige comprar, y se actualiza dueño y saldo
RF18: Calcular Renta Prop.	✓	Se calculó renta base, con casa, y con hotel; resultados correctos en cada caso
RF19: Calcular Renta Jugador	✓	Se muestra el total de renta de propiedades del jugador, considerando mejoras
RF20: Construir Hotel	✓	Se probó: construir casas → hotel; verificado que se descuenten casas del banco y se cobre correctamente
RF21: Pagar Renta	✓	Jugador cae en propiedad ajena → saldo se descuenta al arrendatario, dueño recibe
RF22: Hipotecar Propiedad	✓	Jugador hipoteca propiedad, recibe efectivo; queda marcada como hipotecada y sin renta
RF23: Extraer Carta	✓	Al caer en Suerte/Comunidad, se extrajo carta y se aplicó acción respectiva
RF24: Verificar Bancarrota	✓	Se probó con jugador sin fondos → no pudo pagar y fue eliminado correctamente
RF25: Jugar Turno completo	✓	El turno ejecuta todas las reglas: lanzar, mover, pagar, sacar cartas, repetir por dobles, cárcel, etc.
RF27: Exportar partida (opt)	✓	Juego exportado a partida.json; JSON válido, legible y reimportable
RF28: Importar partida (opt)	✓	Se cargó archivo JSON previamente exportado → juego restaurado con datos idénticos al guardado