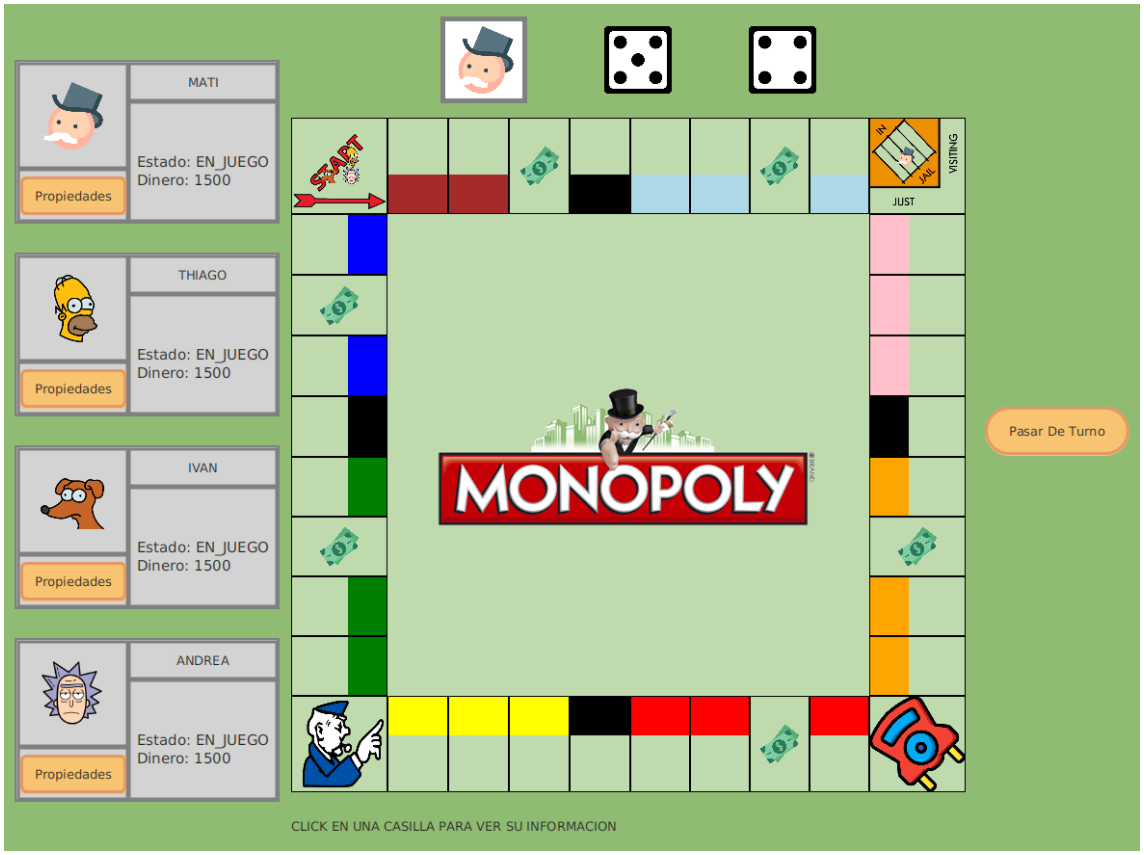
.UBAfiuba
FACULTAD DE INGENIERÍA

Paradigmas de Programación
(TB025) Curso Cano

Trabajo Práctico 1
Programacion Orientada a Objetos: Monopoly

26 de abril de 2024

Thiago Pacheco	Matias Bartellone	Ivan Maximoff	Andrea Figueroa
111003	110484	110868	110450



1. Objetivo del trabajo

La idea del trabajo práctico es crear un juego Monopoly con variantes propias en la jugabilidad y reglas, diseñando un modelo de solución utilizando programación orientada a objetos y por ende aprovechando los diversos patrones y principios de diseño donde resulte conveniente, procurando mantener las buenas prácticas de programación impartidas por la cátedra.

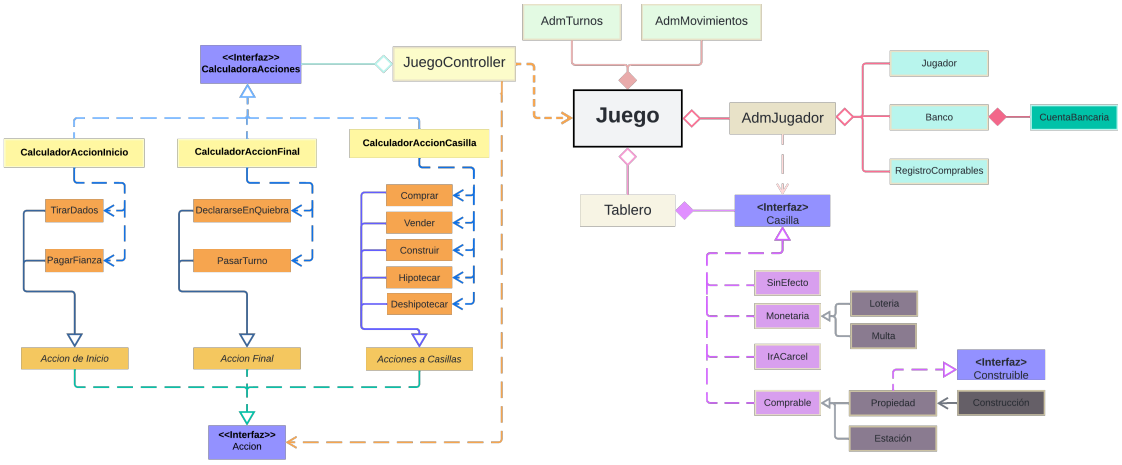
2. Supuestos

Dada la libertad para tomar criterios respecto a especificaciones del juego no provistas por la cátedra, en la siguiente sección expondremos las hipótesis tomadas durante la realizacion del mismo.

- Para salir de prisión sin pagar la fianza un jugador debe obtener en ambos dados iguales.
- El valor que un jugador recibe por la venta de construcciones es la mitad del que se pagó por esta.
- El valor de las hipotecas y deshipotecas de cada propiedad es la mitad correspondiente al pago que se realizó por la compra de las mismas.
- Las estaciones tambien son sujetas a ser hipotecadas y deshipotecadas como lo son las propiedades.
- Se agregó un estado *CRISIS*, el cual describe que un jugador ha caido en una casilla que forzó al mismo a realizar un pago para el cual no tenia fondos. De ser el caso, el jugador en *crisis* solo podrá salir de este estado hipotecando sus propiedades o vendiendo construcciones hasta poder completar el pago, de no poder salir del estado de crisis el jugador solo tendrá la opción de aceptar entrar en quiebra y por lo tanto ya no ser partícipe del juego.

3. Diseño

Se presenta a continuación una primera visualización del diagrama de clases, destacando únicamente los nombres de las clases involucradas en el diseño del sistema. Esta representación simplificada proporciona una visión general del diseño del trabajo abstraida de los detalles de implementación. *(Para visualisar el diagrama de clases completo véase la sección UML)*



Entidades:

Clases Principales:

- Juego
- Tablero
- Jugador
- AdmJugador
- JuegoController
- Realizaciones de Casilla

Clases de Apoyo:

- AdmTurnos
- AdmMovimientos
- RegistroComprable
- Banco
- CuentaBancaria

Interfaces:

- Casilla
- Construible
- CalculadoraAcciones
- Accion

4. Funcionamiento

Nuestro programa que replica el juego Monopoly se estructuró orientándose al patrón de arquitectura de módulos: Model-View-Controller (MVC). Por lo que lo percibido por la vista es procesado e interpretado como acciones por el Controller, quien se comunica con el modelo para gatillar las implicancias de estas mismas sobre el juego, analiza el estado del modelo e interpreta qué posibles acciones puede tomar el usuario en turno para comunicarle a la vista que muestre al usuario cómo se encuentra el juego y qué opciones tiene este en su turno.

Esta estructura nos permite separar las incumbencias de la lógica del modelo y la elección de interfaz para la vista al usuario (trivial en el presente trabajo práctico) , brindando felexibilidad a nuestro código, en el sentido en que, de querer modificarse algún requerimiento incumbente a la lógica del modelo se puedan implementar estos directamente en el módulo del modelo.

4.1. Modelo del juego

Compendiamos las entidades partícipes en el modelo del juego para, posteriormente, explicar las responsabilidades correspondientes a cada una: (para ver una simplificación de sus relaciones véase la *Figura*1 : *Entidades del modelo*)

- Juego
- AdmTurnos
- AdmMovimientos
- AdmJugador
- Tablero
- Jugador
- Banco
- RegistroComprable
- CuentaBancaria
- Casillas

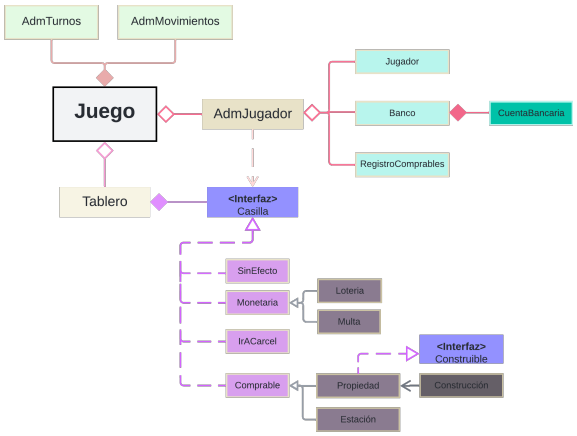


Figura 1: Entidades del modelo

Juego

El objeto Juego opera principalmente como un pasamanos de funcionalidad dentro del sistema, siendo responsable de gestionar todas las acciones posibles del juego. Su función principal consiste en invocar al administrador correspondiente para cada acción específica y solicitarle que la lleve a cabo. Aunque puede analizar varias situaciones de juego y algunos condicionantes para realizar ciertas tareas, no contiene mucha lógica adicional por sí mismo.

En resumen, el objeto *Juego* actúa como una interfaz central que coordina las diferentes acciones del juego al dirigir las solicitudes a los administradores especializados correspondientes, sin agregar una cantidad significativa de lógica adicional.

Tablero

Tablero es una representacon de un tablero circular de juego real del monopoly, compuesto por casillas. Carga con la responsabilidad del orden del tablero y sus implicancias sobre la partida, esta se traduce en métodos que permiten calcular la casilla donde cae un jugador dado un resultado de los dados y de reconocer cuando un jugador da una vuelta entera al mapa.

Casilla

Casilla define el comportamiento básico de una casilla en un juego de mesa, proporcionando un tipo y una acción pasiva que afecta al jugador que cae sobre ella. Estas acciones pasivas pueden incluir efectos como modificar el dinero del jugador, enviarlo a la cárcel u otras acciones relacionadas con el juego que no son producto de una elección del jugador.

- **Casillas Comprables.-** Vale la pena diferenciar las casillas comprables que desempeñan un rol de alta relevancia en el juego, estas mismas pueden ser compradas, hipotecadas y deshipotecadas (elecciones explícitas del jugador). La acción pasiva de las casillas comprables consiste en cobrar un alquiler a cualquier jugador que caiga en ella y que no sea propietario. Además, el valor del alquiler puede variar a lo largo del juego según se decida.

Construible

Construible define el comportamiento para poder crear y destruir construcciones, permite acceder a su información. Este es implementado por Proiedad que es el u´nico comprable que permite contrucciones, pero otras clases podrían implementarla de ser necesario.

Administradores

Los tres administradores de juego son objetos a los que se les delega tareas consisas, con la finalidad de diluir las resposabilidades del *juego* en estos y facilitar la interpretación del juego.

- El Administrador de Turnos se encarga exclusivamente de la gestión de los turnos.
- El Administrador de Movimientos se dedica únicamente a mover efectivamente a los jugadores en el tablero.
- El Administrador de Jugadores juega un rol más importante en el desarrollo de este, pues administra los objetos involucrados en las responsabilidades asociadas al manejo integral de los jugadores.

Administrador de Jugadores: Es responsable de direccionar los efectos de las acciones activas de los jugadores, tales como como comprar, construir, vender, hipotecar, deshipotecar y pagar fianzas. Además, también se encarga de realizar las acciones pasivas de las casillas y el juego,

ya que dispone de la funcionalidad y el conocimiento necesarios para llevar a cabo estas acciones. Por lo tanto, tiene acceso a todos los jugadores para realizar estas tareas. Para delegar ciertas tareas, implementamos el **Banco**, que se encarga de gestionar el dinero de los jugadores, realizando transacciones y accediendo a sus cuentas bancarias. También implementamos el **Registro de Comprables**, que almacena y gestiona toda la información relacionada con los bienes adquiribles, como los propietarios, los barrios y la cantidad de construcciones.

4.2. Controller

Entidades:

- JuegoController
- Accion
- CalculadoraDeAcciones

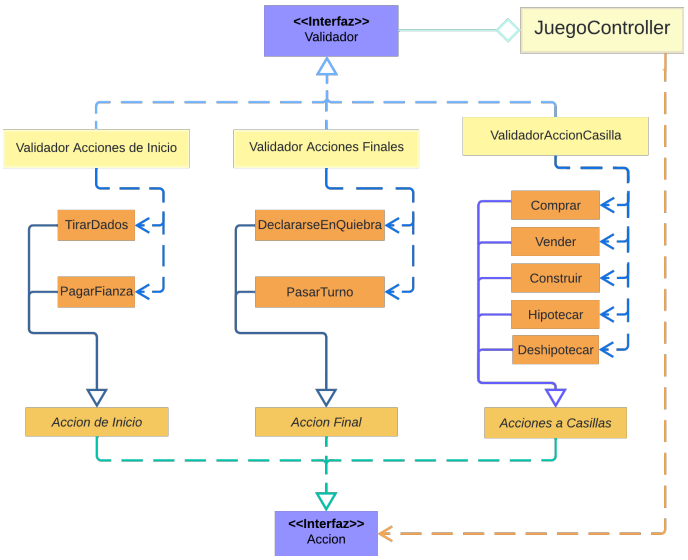


Figura 2: Entidades del controller

JuegoController

Obtiene las posibles acciones que puede realizar el jugador en turno pidiéndoselas a las *calculadoras de acciones* y le indica a la vista que muestre las mismas. Así mismo procesa lo percibido por la vista y se encarga de gatillar estas acciones para que se reflejen en el modelo y se consideren los efectos de dicha acción sobre las opciones de los jugadores. En síntesis funciona como intermediario entre la parte visual del juego y la lógica del mismo.

Calculadora de Acciones

La función principal de la *calculadoradeacciones* es compilar las opciones de acciones disponibles para el jugador en turno leyendo el estado del juego.

Cada implementación concreta de *calculadora* carga con la responsabilidad de analizar qué acciones son posibles para cada jugador considerando solo las de un determinado tipo de *Accion* (vease la subseccion de Accion)

Accion

Las Acciones posibles para un jugador en turno se dividen según la etapa del turno en la que esté el mismo.

- **Acciones de apertura** Aquellas que se deben de realizar al inicio del turno de un jugador. Como tirar los dados o en caso de que el jugador se encuentre en la cárcel se tiene también la opción de pagar la fianza.
- Una vez realizadas las acciones de apertura del turno se pueden llevar a cabo las acciones involucradas con las casillas del juego (hipotecas, dehipotecas, construcciones, etc) de forma ilimitada, en tanto se cumplan con las condiciones, además de las acciones de cierre de turno.

4.3. Vista

Dada la circunstancia de que la implementacion de la visualización del juego resulta trivial para la evaluación del presente trabajo práctico nos tomamos la libertad de omitir el análisis y presentación de los detalles asociados a esta, sin embargo, algo que sí mencionaremos es que fue diseñada de modo en que tenga conocimiento de lo que hay en el modelo mas no el recíproco,es decir, el modelo se mantiene totalmente ajeno a cuál es la implementación de la vista.

5. UML

Previa a la exposición del diagrama de clases desarrollado, vemos relevante mencionar:

- La clase *Juego* fue implementada de modo que sirve como un **pasamanos**, lo que es, que si bien expone varios comportamientos en su interfaz pública, la implementacion de estos en la clase juego consiste principalmente en delegar dichas tareas a las diferentes entidades del modelo con las que *juego* tiene comunicación directa (principalmente otros administradores, esto se implementó así con la finalidad de que sirva como el **principal punto de acceso al modelo**.
- De forma relativamente análoga se diseñó la clase *AdmJugador*, pues, si bien esta expone una cantidad considerable de métodos, la misma fue diseñada para desempeñar un papel de *central* para la administración y delegación de las diferentes situaciones relacionadas exclusivamente a a roporcione un fácil acceso a las distintas

A continuación exponemos el diagrama de clases respectivo a nuestro programa.

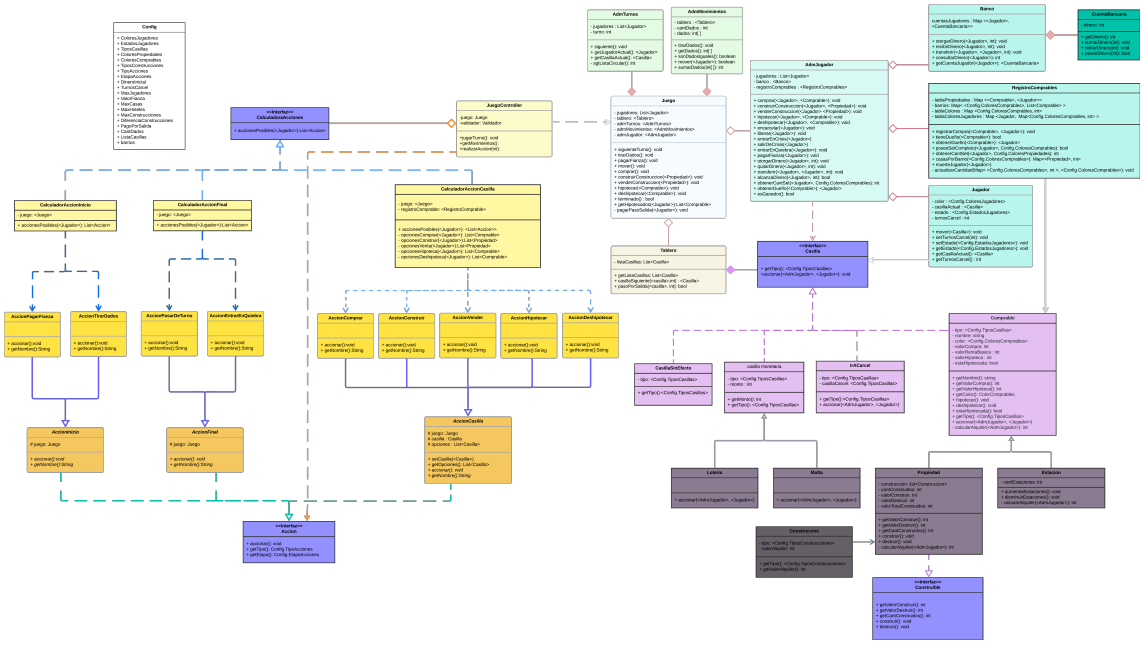


Figura 3: Diagrama de clases global

Anexo para una mejor visualización . Link

6. Detalles de Implementación

Dado que la finalidad del presente trabajo práctico consiste en implementar el juego Monopoly respentando los principios de programación brindados por la cátedra, a continuación **expondremos las decisiones de implementación que consideramos convenientes** para la realizacion de dicho propósito y una compilación de los motivos considerados por los cuales optamos por estas.

6.1. Implementacion clase Registro Comprable

Una discusión recurrente durante el diseño del modelo del juego fue respecto a *a qué entidad le concierne llevar el registro de qué jugador es dueño de qué propiedades*. Finalmente se llegó al acuerdo de emular lo que sucede en la vida real modelando la institución *registro público* en la que se inscriben los actos relativos bienes, propiedades y derechos sobre las mismas. De esta manera reunimos la información necesaria en el juego relativa a las propiedades, estaciones y la cantidad de construcciones sobre las mismas, y modelamos comportamientos relevantes para las distintas reglas del juego que son relativas al estado de esta clase. Por ejemplo, el *RegistroComprable* proporciona el comportamiento para saber cuántas propiedades relativas a un barrio son de propiedad de un jugador. De esta manera que nos ahorramos cargar a cada jugador con información que realmente no es útil para su comportamiento a la vez demás podemos acceder a un comportamiento más trabajado que el que hubiese resultado de tener que preguntarle a cada jugador qué propiedades tiene y luego tener que contar manualmente cuales son de un determinado color.

6.2. Implementación de la interfaz Casilla

Un factor que resultó decisivo en el modelado de las diferentes casillas y tipos de casillas en el tablero fue el considerar que debería ser relativamente **fácil ampliar** la variedad de casillas que hay, en el sentido de ampliar las funcionalidades o efectos que podría tener una casilla del juego, pues, dadas las directrices del principio Open-closed, las entidades deben estar abiertas para su extensión pero cerradas para su modificación.

En el caso de la interfaz *casilla*, su diseño permite que nuevas implementaciones puedan ser agregadas fácilmente en el futuro para satisfacer requisitos adicionales o cambiar el comportamiento existente, permitiendo que el resto del juego se abstraiga de en qué tipo de casilla cayó un jugador al moverse, pues lo que se sabe es que cayó en una *Casilla* que expone en su interfaz la funcionalidad *accionar*, de manera que cada realización de la interfaz *casilla* se queda con la responsabilidad de implementar qué hacer cuando un jugador cae en esta misma.

6.3. Implementación de la herencia de casillas monetarias

Dentro de la variedad de efectos y utilidades de las casillas especificadas en los requerimientos generales de la lógica del juego identificamos que las *casillas de loteria* y las de *multa* **son una** variedad de una misma entidad que representaría una casilla directamente asociada a montos de dinero. En este contexto, vimos por conveniente implementar las mismas mediante una clase abstracta ***casilla monetaria*** que considere en sus atributos el monto al cual está asociada dicha casilla. Siendo que cada una de las concretizaciones se responsabiliza de sobrescribir cómo es el accionar cuando un jugador cae en la misma, decidiendo qué se hace con dicho monto (descontar o incrementar dinero). De esta forma resulta parametrizable y flexible el monto base respecto al cual trabaja cada casilla monetaria concreta. Además resulta un modelo extensible en el sentido de funcionalidades, pues de quererse agregar un nuevo tipo de casilla asociada a algún monto monetario esta misma se puede agregar como otra concretización de ***casilla monetaria***.

6.4. Implementación de la herencia de casillas comprables

Considerando las funcionalidades especificadas de *Propiedades* y *Estaciones* nos percatamos de la presencia de posibles atributos en común, como lo son: el valor que se cobra a un jugador que caiga en estas, el color de estas que indica la pertenencia a un grupo de las mismas, entre otros. Además encontramos similitudes en los comportamientos, el más destacable sería el cómo afecta a un Jugador que caiga en estas (cobro de alquiler).

6.5. Implementación de la interfaz Calculadora

Para la comunicación entre lo que se muestra y el estado del modelo del juego, el controller requería de algún objeto que se encargase de la lógica de leer qué opciones puede tomar un jugador en turno. La intención al plantear una interfaz *CalculadoraAcciones* fue de poder abstraer al Controller mismo de cómo se consiguen las Acciones posibles, mas aún así poder acceder a la funcionalidad de obtener estas. De esta manera, cada realización de las calculadoras se ocupan de leer y considerar distintos ámbitos del juego que generarían opciones para el jugador y a la vez desempeñar un papel análogo a un *abstract factory*, ya que instancian diferentes implementaciones concretas de *Accion*, mas el tipado de los datos que devuelven es del de la interfaz.

6.6. Implementación de la interfaz Accion

Para el manejo de las distintas acciones posibles en el turno implementamos la interfaz Accion, la cual tiene el metodo accionar, que es implementado por distintas clases concretas, como por ejemplo AccionComprar llama a juego.comprar(). A su vez las acciones estan divididas en tres: AccionInicio, AccionCasilla y AccionFinal. Estas representan las distintas etapas de acciones, por ejemplo, AccionTirarDados es inicio porque solo se puede hacer al inicio. Esto es una especie de patron command ya que manejamos Accion como una interfaz general que puede realizar disintas tareas.

6.7. Implementación de la relación entre la calculadora y Acción

Encontramos la oportunidad de instanciar las diferentes acciones que el jugador tiene a disposición, sin atar al controller a las diferentes clases específicas que representan las diferentes acciones posibles, para esto seguimos el patrón abstract factory, pues en esta situación *Calculadora* es el factory y las realizaciones de la interfaz son los distintos tipos de fábricas, siendo el producto las difrentes acciones que cumplen con la interfaz *Accion*. Destacar que gracias a esta implementación se agrega un nivel de extensibilidad y escalaibilidad al código.

6.8. Implementacion del Config

El archivo Config es lo que nos permite configurar el juego con valores predeterminados. Esto nos permite una fácil modificación del juego, e incluso tener varias versiones del mismo. Para la ListaCasillas se implemento una función que permite ir agregando propiedades de la ListaPropiedades de manera sencilla, que se encarga de agregar un número específico de propiedades y actualizar el index actual luego poder agregar otras sin repetir.

7. Conclusiones

La programación orientada a objetos resulta ser un enfoque de programación ventajoso en contextos como lo es el diseño del juego Monopoly, es decir, sistemas que son modelables con entidades del mundo real, donde cada entidad carga con responsabilidades específicas del problema. Un factor que resultó trascendental en el desarrollo del trabajo práctico fue el proceso previo a la programación propiamente, es decir**el diseño** de las entidades que serían partícipes en el programa. Este mismo conllevó una gran parte del esfuerzo y tiempo invertido en el desarrollo íntegro del sistema, para el cual resultaron relevantes el uso de diagramas UML y las discusiones grupales que permitieron tener en cuenta pros y contras de las diferentes tentativas de diseño que cada integrante del grupo percibió.

Finalmente, es importante destacar que, a pesar de que el objetivo durante el desarrollo del trabajo práctico fue la implementación y asimilación de los diferentes principios de diseño difundidos por la cátedra, resultó una tarea compleja el cumplir con todos los principios y no toparnos con situaciones en las que finalmente romperíamos alguno de los mismos, por lo que surgieron discusiones y tomas de decisiones respecto a qué se debería priorizar según el contexto.

En conclusión, un programa orientado a objetos y los diferentes principios de diseño apuntan a la construcción de un programa flexible, extensible y fácil de entender y mantener. Lo cual implica que el código esté bien diseñado sopesando qué prácticas se priorizan.