

Universidad Autónoma de Entre Ríos

Facultad de Ciencia y Tecnología

TRABAJO PRÁCTICO N°2 “Diseño e implementación en C++ usando POO”

Licenciatura en Sistemas de Información

Programación Orientada a Objetos

Año lectivo 2025

A cargo de:

Lic. Goette, Pablo Emanuel (Titular), Brambilla, Leonardo
Miguel (JTP)

Autores:

Fornari Schwarzkopf, María Belén; Vlasic, Francisco

Fecha de entrega:

27 de julio de 2025

1. Conceptos generales	2
1.1. Introducción	2
1.2. Objetivos	2
1.3. Lista de Requerimientos del proyecto	2
1.4. Estructura de usuarios propuesta	2
1.6. Product Backlog	7
2. Modelo de sistema	8
2.1. Diagrama de flujo	8
2.3. Diagrama UML de clases	10
2.4. Funcionamiento de las clases	12

1. Conceptos generales

1.1. Introducción

El presente documento describe el diseño y la documentación de una aplicación para simular el “Juego de la Oca” en el lenguaje C++ y conceptos de Programación Orientada a Objetos. Este desarrollo se enmarca en las pautas del Trabajo Práctico N°2 de la asignatura, integrando la interfaz de usuario con el diseño de software previamente establecido en la primera entrega.

1.2. Objetivos

El objetivo de este trabajo práctico es aplicar los principios de la programación orientada a objetos (POO) en el diseño y la implementación, del juego denominado El juego de la Oca. Originalmente es un juego de mesa en el que los jugadores deben avanzar a través de un tablero de casillas, siguiendo reglas específicas como avanzar de acuerdo con el lanzamiento de un dado y saltar casillas especiales como las de "oca" o "puente".

1.3. Lista de Requerimientos del proyecto

1.3.1. Requerimientos Funcionales

- Simular partidas del Juego de la Oca entre dos o más jugadores.
- Generar y gestionar el tablero, compuesto de 63 casillas, aplicando el comportamiento definido para cada una según las reglas¹.
- Permitir el lanzamiento de un dado y mover la ficha del jugador correspondiente en función al número obtenido.
- Identificar la caída en casillas especiales y aplicar las reglas correspondientes.

1.3.2. Requerimientos no Funcionales

- Debe ser diseñado utilizando principios de Programación Orientada a Objetos.
- El diseño del sistema debe ser representado mediante diagrama de clases, UML o similares.
- La estructura y el funcionamiento del sistema deben estar debidamente documentados.
- La arquitectura del software debe ser modular para facilitar futuras modificaciones y extensiones.

1.4. Estructura de usuarios propuesta

Nivel de usuario	Tipo de usuario	Descripción de permisos
1	Jugador	Inicia una nueva partida, la juega y la finaliza
2	Administrador	Gestión y consulta de logs.

¹ Ver Bibliografía consultada.

1.5. Historias de tareas

(01) Iniciar partida guardada	
Usuario: Todos	Prioridad: Alta
Riesgo: Bajo	
El sistema permitirá recuperar una partida previamente iniciada y no finalizada mediante la carga de un archivo de texto que contenga el progreso alcanzado.	
Observaciones: Al querer salir del juego, el jugador deberá tener la opción de exportar el progreso en un archivo de texto.	

(02) Iniciar partida guardada UI	
Usuario: Todos	Prioridad: Alta
Riesgo: Bajo	
Descripción: La interfaz de usuario debe permitir la carga de un archivo de texto que contenga el progreso de una partida.	
Observaciones: En caso de errores en la carga o el formato del archivo, se deben mostrar mensajes genéricos y amigables al usuario (Ej.: "archivo inválido", "formato no aceptado").	

(03) Iniciar la partida	
Usuario: Jugador	Prioridad: Alta
Riesgo: Bajo	
Descripción: El sistema debe permitir: <ul style="list-style-type: none"> • Ingresar cantidad de jugadores. • Inicializar el tablero con 63 casillas. • Posicionar a todos los jugadores en la casilla de salida. 	
Observaciones:	

(04) Tirar el dado	
Usuario: Jugador	Prioridad: Baja
Riesgo: Bajo	
Descripción: El jugador debe poder “tirar” el dado de forma virtual y obtener un resultado aleatorio entre 1 y 6.	
Observaciones:	

(05) Movimiento de la ficha	
Usuario: Jugador	Prioridad: Baja
Riesgo: Bajo	
Descripción: En base al resultado obtenido en “Tirar el dado”, mover la ficha del jugador cuántas casillas correspondan y detectar si cae en una casilla especial.	
Observaciones:	

(06) Casillas especiales	
Usuario: Jugador	Prioridad: Alta
Riesgo: Medio	
Descripción: Identificar en qué tipo de casilla cayó: <ul style="list-style-type: none"> • Casillas de Oca (9, 18, 27, 36, 45, 54): El jugador avanza a la siguiente oca y vuelve a tirar. • Puente (casilla 6): Avanza a la casilla 12. • Posada (casilla 19): Pierde un turno. • Pozo (casilla 31): No puede moverse hasta que otro jugador caiga en la misma casilla. • Laberinto (casilla 42): Retrocede a la casilla 30. • Cárcel (casilla 56): Pierde dos turnos. • Calavera (casilla 58): Vuelve a la casilla 1. • Jardín de la Oca (casilla 63): Gana el juego. 	

Observaciones: Para ganar el juego, el jugador debe llegar de forma exacta. Si el número del dado excede la casilla 63, el jugador retrocede la cantidad de números restantes.

(07) Final del juego	
Usuario: Jugador	Prioridad: Alta
Riesgo: Bajo	
Descripción: Al llegar a la casilla 63 se debe verificar la condición de victoria y: <ul style="list-style-type: none"> • Retroceder si se excede. • Mostrar mensaje de victoria si gana. 	
Observaciones:	

(08) UI Inicial	
Usuario: Jugador	Prioridad: Alta
Riesgo: Bajo	
Descripción: Debe contar con un menú de inicio que consulte si quiere iniciar el juego, recuperar un juego o ver las reglas de juego.	
Observaciones:	

(09) UI Tablero	
Usuario: Jugador	Prioridad: Media
Riesgo: Bajo	
Descripción: <ul style="list-style-type: none"> • Generar un tablero de 63 casillas, distinguiéndose las mismas por número y destacando aquellas que sean especiales. • Mostrar ventanas emergentes al caer en una casilla especial que indique que ocurre. 	

Observaciones:

(10) Controles del juego	
Usuario: Jugador	Prioridad: Alta
Riesgo: Bajo	
Descripción: <ul style="list-style-type: none"> • Mostrar un botón “Tirar dado”. • Mostrar el número obtenido en pantalla. 	
Observaciones:	

(11) Guardado y carga	
Usuario: Jugador	Prioridad: Alta
Riesgo: Medio	
Descripción: El jugador debe poder exportar en un archivo de texto el progreso de la partida.	
Observaciones:	

(12) Registro de errores y Auditoría	
Usuario: Administrador	Prioridad: Media
Riesgo: Bajo	
Descripción: Se debe llevar un registro de los ingresos, consultas y errores que pudieran surgir. Generar un archivo de texto con la información obtenida. Debe contar con palabras claves para facilitar la búsqueda. Datos: fecha y hora, jugador, operacion, descripción	
Observaciones: Los log deben tener fecha y contexto del error (Ej: dónde ocurrió, qué pasó, con qué usuario, en qué objeto/clase)	

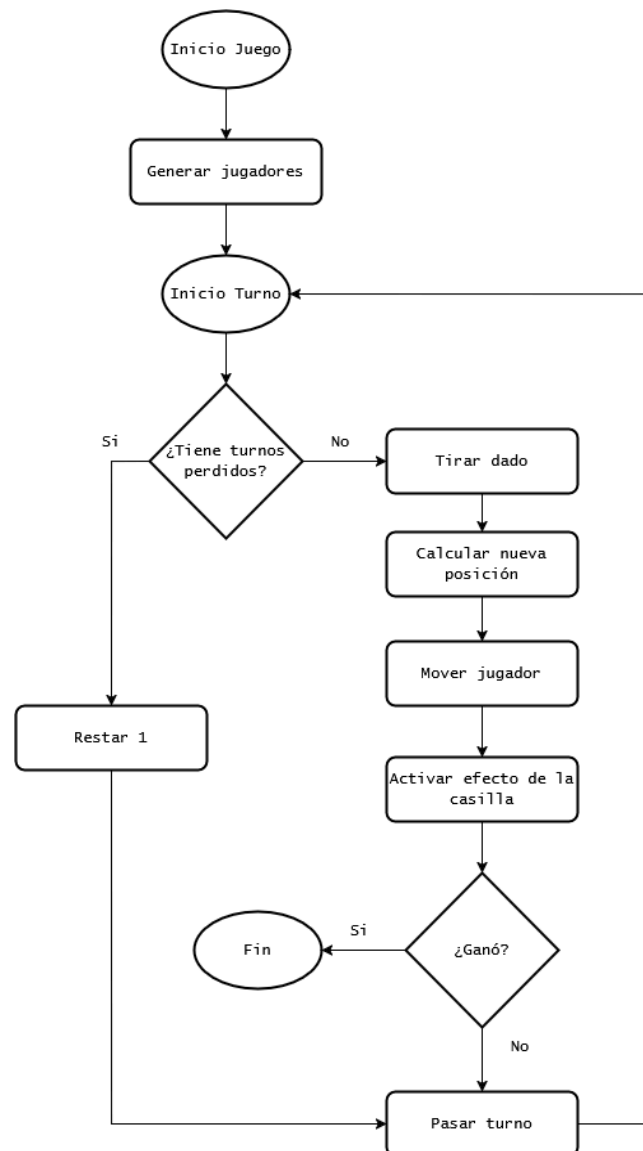
1.6. Product Backlog

Nº	Nombre	Prioridad	Sprint
01	Login	Alta	1
02	Login UI	Alta	1
03	Iniciar la partida	Alta	1
08	UI Inicial	Alta	1
04	Tirar el dado	Baja	2
05	Movimiento de la ficha	Baja	2
06	Casillas especiales	Alta	2
07	Final del juego	Alta	2
10	Controles del juego	Alta	3
09	UI Tablero	Media	3
11	Guardado y carga	Alta	3
12	Registro de errores y auditoría	Media	4

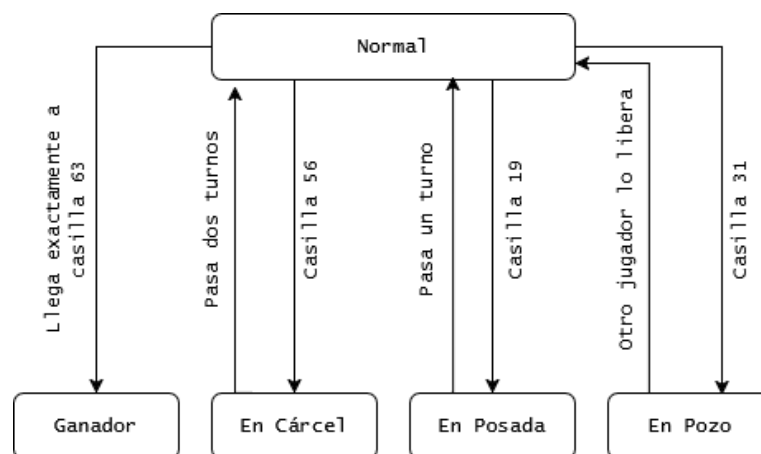
2. Modelo de sistema

2.1. Diagrama de flujo

1. Se genera un nuevo juego seleccionando la cantidad de jugadores.
2. Los jugadores ingresan sus nombres e inician la partida.
3. Inicia el turno del jugador actual:
 - a. Si tiene turnos perdidos, se resta 1 (4).
 - b. Si no tiene turnos perdidos:
4. El jugador tira el dado.
5. Avanza la cantidad de casillas correspondiente.
 - Si cae en una casilla especial, se aplica su efecto.
 - Si cae en una casilla normal, simplemente se queda.
6. Se verifica si el jugador llegó o superó la casilla final (N° 63):
 - a. Si llega exactamente, gana el juego (FIN).
 - b. Si se pasó, retrocede la diferencia y continúa en esa casilla.
4. Se pasa el turno al siguiente jugador (3).

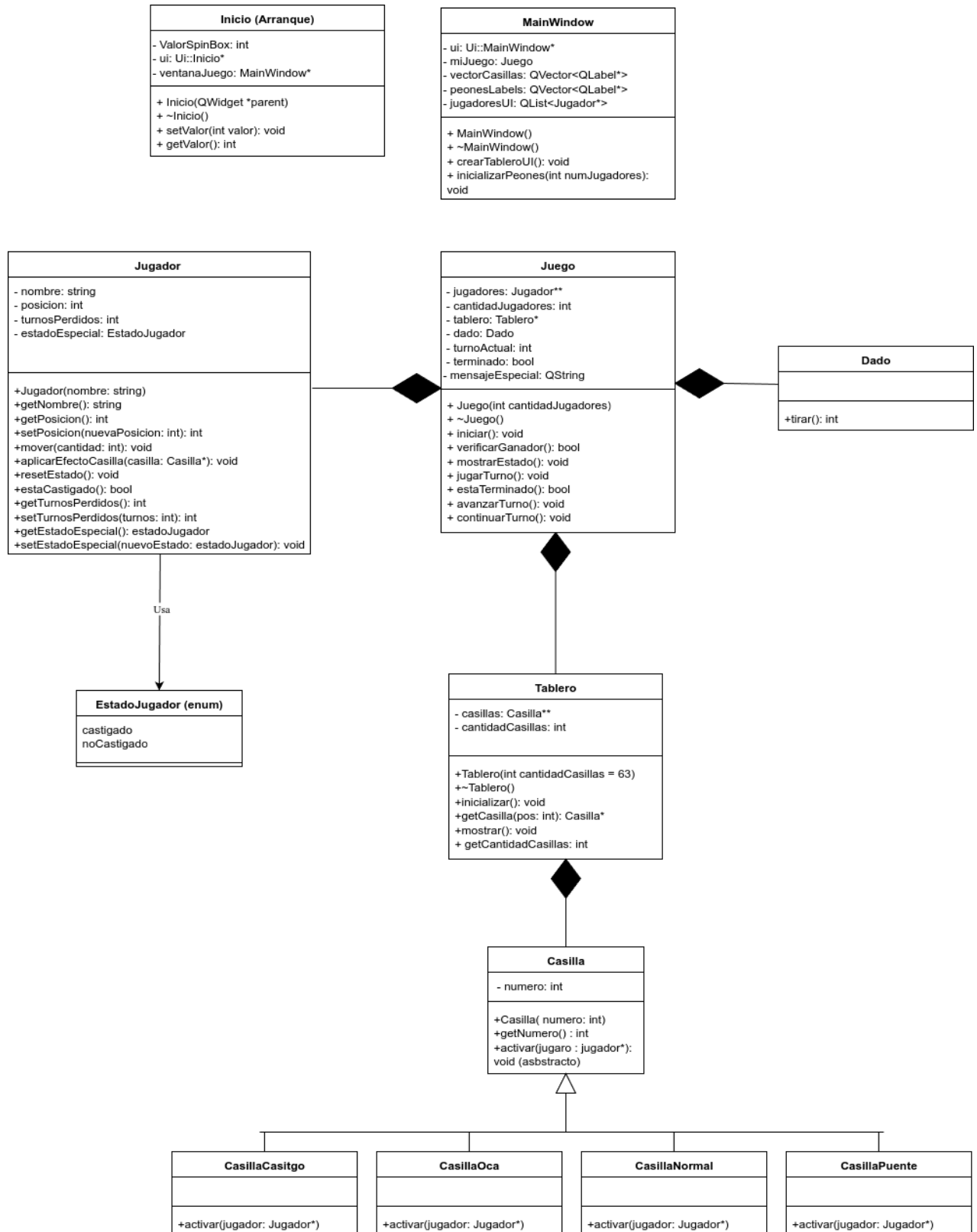


2.2. Diagrama de estados del jugador



2.3. Diagrama UML de clases

- El juego se modela mediante la clase *Juego* que contiene jugadores, el tablero y el dado.
- El *Tablero* contiene una colección de casillas, que pueden ser normales o especiales.
- La clase *Casilla*, que es abstracta, es la base para todas las variantes (castigo, oca, normal o puente).
- *Jugador* representa a cada participante e incluye nombre, posición y estado.
- *MainWindow* es la clase principal de la interfaz de usuario. Contiene el tablero, los paneles de los jugadores y el botón de “Tirar Dado”.
- *Inicio (Arranque)* es la clase del menú de ingreso al juego.



2.4. Funcionamiento de las clases

Juego

Es la clase principal, controla todo el sistema.

- Rol: Coordina el flujo de la partida, administra los turnos, el dado, el tablero y los jugadores. Es el punto de inicio del juego.
- Atributos:
 - jugadores: Jugador** : puntero a un arreglo de punteros a objetos Jugador. Se utiliza de esta manera para tener control dinámico de la cantidad de jugadores ya que este valor es ingresado por el usuario. Se habla de una relación de composición.
 - cantidadJugadores: int: total de jugadores en partida.
 - tablero: Tablero*: Juego posee y controla la existencia de Tablero. Juego necesita un tablero para funcionar y el Tablero no tiene sentido fuera de Juego, por lo que se habla de una relación de composición.
 - dado: Dado: instancia del dado para simular las tiradas.
 - turnoActual:int: índice del jugador que tiene el turno.
 - terminado: bool: indica si hay un ganador.
 - mensajeEspecial: QString: mensaje que se muestra al caer en una casilla especial.
- Señales:
 - juegoIniciado(int numJugadoresActivos): void: envía la cantidad de jugadores en juego.
 - dadoLanzado(int jugadorIndex, int valorDado): void: Envía el jugador de turno con el valor que obtuvo del dado.
 - turnoCambiado(const QString &nombreSiguienteJugador): void: envía el siguiente jugador.
 - jugadorCastigado(const QString &nombreJugador, int turnosRestantes): void: envía el nombre del jugador castigado y cuántos turnos de castigo tiene.
 - juegoGanado(const QString &nombreGanador): void: envía el ganador, si lo hay.
 - jugadorMovido(int jugadorIndex, int casillaOrigen, int casillaDestino): envía que jugador se mueve, desde que casilla y hasta que casilla.
 - mensajeParaMostrar(const QString &mensaje): envía la señal de los mensajes de la QMessageBox.
- Métodos:
 - iniciar(): pide los nombres e inicializa jugadores y tablero.
 - verificarGanador(): determina si algún jugador alcanzó exactamente la casilla 63.
 - mostrarEstado(): imprime un resumen del estado actual.
 - estaTerminado(): retorna true si el juego ha terminado.
 - avanzarTurno(): avanza al siguiente turno.
 - obtenerJugadores(): devuelve una QList de Jugador*
 - avanzarTurno() (private): avanza el turno del jugador
 - jugarTurno(): ejecuta la acción del turno.
 - continuarTurno(): continúa el turno una vez que el usuario cierre la QMessageBox correspondiente al aviso de las casillas especiales (si lo hubiera).

Tablero

- Rol: Estructura contenedora de las 63 casillas de juego. Son accesibles por número.

- Atributos:
 - casilla: Casilla**: arreglo dinámico de punteros a objetos Casilla utilizando polimorfismo. Cada posición del array apunta a una casilla del tablero y cada una de estas puede ser de un tipo distinto (CasillaNormal, CasillaPuede, etc). Es necesario el uso de polimorfismo porque todas las casillas comparten el método activar(jugador), pero cada una lo hará de manera diferente.
 - cantidadCasillas: int: total de casillas del tablero. Si quisieran agregarse más, solo debería modificarse este atributo.
- Métodos:
 - getCasilla (int pos): Casilla*: accede a la casilla según el número.
 - inicializar(): crea cada casilla e instancia el tipo que corresponda.
 - mostrar(): imprime el estado del tablero.
 - getCantidadCasillas(): Retorna la cantidad total de casillas en el tablero.

Casilla

- Rol: Define la interfaz que implementarán (por herencia) todas las casillas. Es abstracta, por lo que obliga a sus hijas a definir el método activar().
- Atributos (private):
 - numero: int: representa la posición de la casilla en el tablero.
- Métodos:
 - getNumero(): retorna el número de la casilla.
 - activar (jugador: Jugador*): método virtual que debe ser redefinido por cada subclase al momento de utilizarlo.

CasillaNormal

Hereda de Casilla.

- Rol: casilla sin efecto. El jugador cae y permanece en el lugar hasta el siguiente turno.
- Método:
 - activar(jugador): no modifica el estado ni la posición del jugador.

CasillaOca

Hereda de casilla.

- Rol: desplaza al jugador a la siguiente oca y vuelve a tirar.
- Método:
 - activar(jugador): modifica la posición y permite un turno extra.

CasillaPuede

Hereda de casilla.

- Rol: desplaza al jugador de la casilla 6 a la casilla 12.
- Método:
 - activar(jugador): modifica la posición.

CasillaCastigo

Hereda de casilla.

- Rol: Representa las penalizaciones (Posada, Cárcel y Pozo).
- Atributo:

- jugadorAtrapadoEnPozo: Jugador*: almacena si un jugador se encuentra atrapado en el Pozo.
- Método:
 - activar(jugador): aumenta turnosPerdidos del jugador según el castigo.

Dado

- Rol: Simula la tira de un dado.
- Método:
 - tirar() devuelve un número aleatorio del 1 al 6.

Jugador

- Rol: representa al jugador individual con su respectivo nombre, posición y penalizaciones.
- Atributos:
 - nombre: string
 - posición: int
 - turnosPerdidos: int
 - estadoEspecial: EstadoJugador
- Métodos:
 - mover(int cantidad): avanza la cantidad de casillas que se pasan por parámetro.
 - jugarTurno(): gestiona el turno.
 - aplicarEfectoCasilla(Casilla*): invoca al método activar de la casilla.
 - estaCastigado(): informa si debe omitir el turno del jugador.
 - resetEstado(): limpia castigos y estados especiales devolviendo al jugador a estado normal.
 - getPosicion(): devuelve la posición actual del jugador.
 - setPosicion(int nuevaPosicion): luego de mover, establece la nueva posición del jugador.
 - getTurnosPerdidos(): devuelve la cantidad de turnos perdidos pendientes del jugador.
 - setTurnosPerdidos(int turnos): establece la cantidad de turnos perdidos que tendrá el jugador según la casilla castigo en la que caiga.
 - getEstadoEspecial(): devuelve el estado en que se encuentra el jugador (castigado o noCastigado)
 - setEstadoEspecial(estadoJugador nuevoEstado): establece el estado del jugador en 'castigado' si cae en una casilla castigo.

Inicio (Arranque)

- Rol: Gestiona la ventana de inicio del juego, permitiendo al usuario interactuar para comenzar una nueva partida o realizar otras acciones iniciales.
- Atributos (private):
 - ValorSpinBox: int: Almacena un valor entero que representa la cantidad de jugadores seleccionados.
 - ui: Ui::Inicio*: Puntero a la interfaz de usuario generada por Qt Designer para la ventana de inicio.
 - ventanaJuego: MainWindow*: Puntero a la ventana principal del juego, para poder mostrarla una vez que se inicie la partida.
- Métodos:
 - setValor(int valor): Establece el valor de ValorSpinBox.

- `getValor()`: Retorna el valor de `ValorSpinBox`.
- Slots:
 - `on_BotonJugar_clicked()`: Slot privado que se ejecuta cuando se hace clic en el botón "Jugar".

MainWindow

- Rol: Gestiona la interfaz de usuario principal del juego, incluyendo el tablero, los jugadores y los controles de la partida.
- Atributos (private):
 - `*ui: MainWindow`: es la pantalla principal del juego.
 - `miJuego: Juego`: instancia del juego que se iniciará.
 - `vectorCasillas: QVector<QLabel*>`: Representa las casillas del tablero creadas.
 - `peonesLabels: QVector<QLabel*>`: Representa los peones creados.
 - `jugadoresUI: QList<Jugador*>`: Almacena punteros a los jugadores en juego.
 -
- Métodos:
 - `crearTableroUI()` (private): crea las casillas dinámicamente.
 - `inicializarPeones(int numJugadores)` (private): crea tantos peones como jugadores se hayan seleccionado en el arranque.
- Slots:
 - `mostrarMensajePopUp(const QString& mensaje)`: muestra un mensaje (`QMessageBox`) si un jugador cae en una casilla especial y emite la señal de continuar el turno al cerrarse.
 - `on_BotonTirarDado_clicked()`:
 - `configurarPanelesJugadores(int numJugadores)`:
 - `actualizarDadoJugador(int jugadorIndex, int valorDado)`:
 - `moverFichaUI(int jugadorIndex, int casillaOrigen, int casillaDestino)`:
 - `mostrarGanador(const QString &nombreGanador)`:
- Señales:
 - `continuarTurno()`: al cerrarse la ventana que avisa al jugador sobre una casilla especial, se emite esta señal para continuar con el juego.

EstadoJugador (enum)

- Rol: Define los posibles estados lógicos que puede tener un jugador a lo largo del juego.
- Atributos:
 - `castigado`: se utiliza para Cárcel, Pozo y Posada.
 - `noCastigado`: se utiliza para Oca, Puente y Normal

3. Bibliografía

Reglas del Juego de la Oca (s.f.). Recuperado el 16 de mayo de 2025 de

<https://www.juegodelaoca.com/index.htm>

Docentes de la cátedra Programación Orientada a Objetos. *Trabajo Práctico N° 1: Diseño e implementación en C++ usando POO*. Campus virtual de la Universidad Autónoma de Entre Ríos.