

Arquitectura de software

Patolli - Java



EQUIPO

Yahir Francisco Lopez Benavidez

Iliana Flores Mendívil

Luz Adela Lopez Oromea

Jesús Francisco Lara Quintero

Martin Alfredo flores Menchaca

□ INDICE

Introducción, Resumen y Objetivos

□ *Diagrama de casos de uso*

- a. *Crear Partida*
- b. *Unirse a Partida*
- c. *Dentro de Juego*
- d. *Lanzar Dado*
- e. *Movimiento a Distancia*
- f. *Final de Partida*

□ *Diagramas de robustez*

- a. *Ingreso de Jugador*
- b. *Crear Partida*
- c. *Movimiento a Distancia*
- d. *Final de Partida*

□ *Código fuente*

- a. *CÓDIGO DADOS*
- b. *CÓDIGO CLIENTE*
- c. *CÓDIGO FrmTablero, FrmTablero3jugadores y FrmTablero4jugadores*
- d. *CÓDIGO Frmmenu*
- e. *CÓDIGO JUGADOR*

□ *Diagramas de clases*

- a. *Dados*
- b. *FrmTablero4jugadores*
- c. *FrmTablero 3 Jugadores*
- d. *FrmTablero*
- e. *Frmmenu*
- f. *Jugador*

□ *Patrones de Diseño Aplicados*

- a. *Observer*
- b. *Controller*
- c. *Facade*
- d. *Proxy*
- e. *Singleton*

□ *diagrama de despliegue*

□ *diagrama de despliegue*

□ *Diagrama de clases de los componentes*

Introducción

El juego de Patolli es un antiguo juego de mesa mexica que se asemeja a una combinación de parchís y dados. Su historia se remonta a la época prehispánica, donde era jugado por nobles y emperadores como una forma de entretenimiento y ritual. En la versión digital del juego, hemos adaptado las reglas tradicionales para mantener la esencia del juego mientras se incorporan funcionalidades modernas que mejoran la experiencia del usuario. El juego permite que hasta cuatro jugadores se unan a una partida a través de una interfaz intuitiva, donde pueden lanzar los dados, mover piezas a lo largo del tablero y competir para llegar primero al final. Además, el sistema está diseñado para manejar de manera eficiente las interacciones y las transiciones de turnos entre jugadores, asegurando un flujo de juego sin interrupciones.

Resumen

Patolli es un juego tradicional que ahora se ha digitalizado para ofrecer una experiencia interactiva y competitiva. El objetivo del juego es mover todas las piezas a lo largo del tablero y ser el primero en llegar al final. Cada jugador tiene un turno en el que lanza los dados, mueve sus piezas y enfrenta desafíos que pueden incluir penalizaciones. El tablero digital está decorado con elementos culturales que recuerdan sus raíces mexicas, proporcionando no solo entretenimiento sino también una conexión cultural con el pasado.

Para unirse a una partida, los jugadores pueden crear una cuenta o ingresar como invitados, después de lo cual se les asigna un turno. El juego cuenta con un sistema de chat integrado que permite a los jugadores comunicarse entre sí, fomentando la interacción social y el espíritu competitivo. A medida que las piezas se mueven, las animaciones y efectos visuales hacen que la experiencia de juego sea dinámica y atractiva.

La lógica del juego está implementada para asegurar que todas las reglas tradicionales se respeten y se apliquen correctamente en el entorno digital. Esto incluye el manejo de los dados, la detección de victorias y derrotas, y la aplicación de penalizaciones cuando sea necesario. La arquitectura del software ha sido diseñada para ser escalable y robusta, permitiendo la incorporación de nuevas funcionalidades y mejoras con facilidad.

Objetivos del Juego

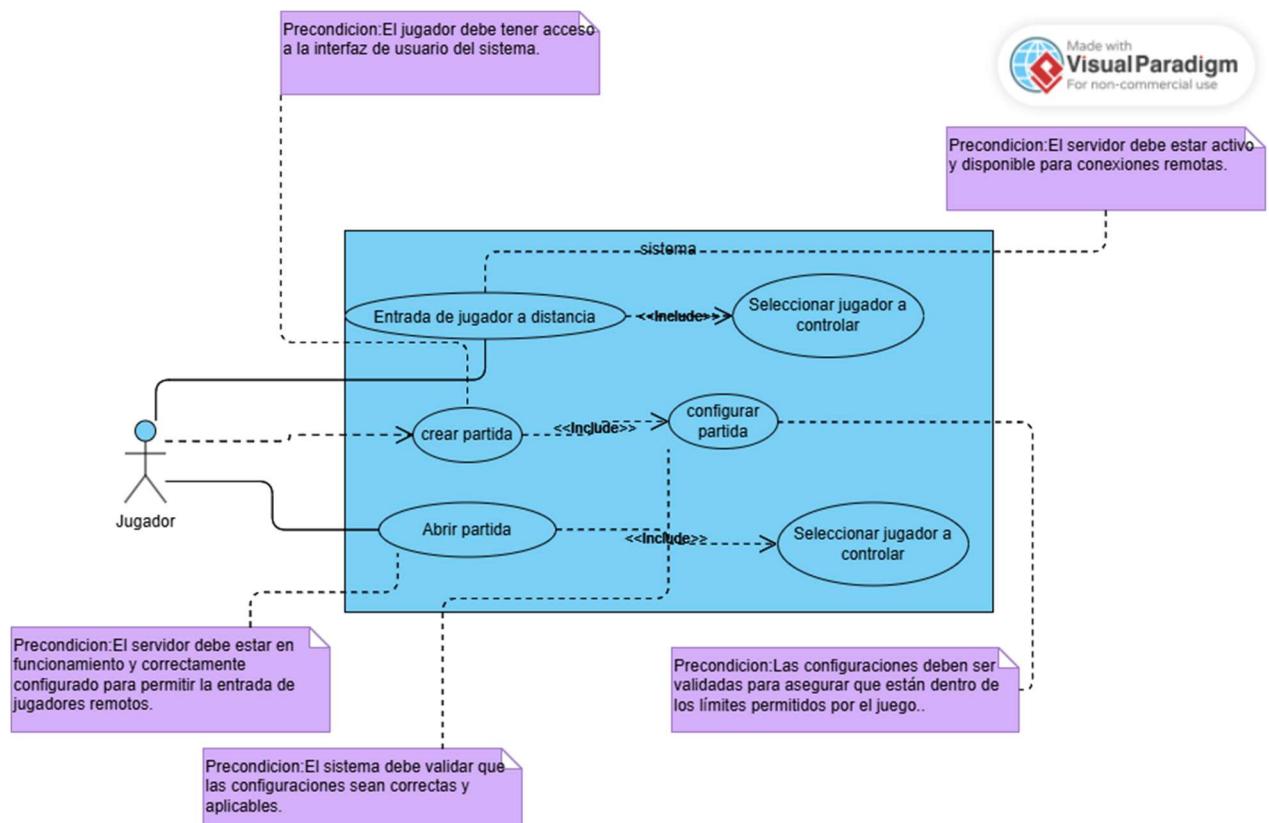
El principal objetivo del juego es mover todas las piezas a lo largo del tablero para llegar al final antes que los demás jugadores. Cada turno implica:

- *Lanzar los Dados: Los jugadores lanzan los dados para determinar cuántas casillas pueden mover sus piezas.*
- *Mover las Piezas: Basado en el resultado de los dados, los jugadores mueven sus piezas estratégicamente para avanzar en el tablero y evitar penalizaciones.*
- *Enfrentar Desafíos: A lo largo del tablero, hay casillas que pueden tener efectos especiales, como retroceder posiciones o perder un turno, agregando una capa de estrategia y azar al juego.*
- *Final del Juego: El juego termina cuando un jugador logra mover todas sus piezas al final del tablero, siendo declarado el ganador.*

DIAGRAMA DE CASOS

PREVIO AL JUEGO

PREPARACION DE PARTIDA



Actores Principales

- **Jugador:** Representa a la persona que interactúa con el sistema para crear y jugar partidas.

Casos de Uso

1. Crear partida

- **Descripción:** El jugador elige la cantidad de jugadores que participarán en la partida.

- **Asociaciones:** Este caso de uso está directamente relacionado con el actor "Jugador".
- **Relaciones:**
 - **Include:** Este caso de uso incluye "Configurar partida", lo que implica que la creación de una partida siempre conlleva la configuración inicial de la misma.
 - **Extiende:** Este caso de uso puede extenderse para incluir "Seleccionar jugador a controlar" si se requiere seleccionar el jugador durante la creación.

2. Entrada de jugador a distancia:

- **Descripción:** Permite la entrada de jugadores que se conectan desde una ubicación remota.
- **Relaciones:**
 - **Include:** Este caso de uso incluye "Seleccionar jugador a controlar", lo que significa que la entrada de jugadores remotos siempre involucra la selección del jugador que van a controlar.

3. Crear partida:

- **Descripción:** Proceso mediante el cual el jugador crea una nueva partida.
- **Relaciones:**
 - **Include:** Este caso de uso incluye "Configurar partida", lo que implica que la creación de una partida siempre conlleva la configuración inicial de la misma.
 - **Extiende:** Este caso de uso puede extenderse para incluir "Seleccionar jugador a controlar" si se requiere seleccionar el jugador durante la creación.

4. Configurar partida:

- **Descripción:** Permite al jugador establecer las configuraciones iniciales de la partida, como la cantidad de dinero y vidas que tendrán los jugadores.

5. Abrir partida:

- **Descripción:** El proceso de abrir una partida ya creada para que los jugadores puedan unirse y empezar a jugar, Habilita el servidor del juego para aceptar conexiones de jugadores remotos.
- **Relaciones:**
 - **Include:** Este caso de uso incluye "Seleccionar jugador a controlar", asegurando que los jugadores seleccionen su control antes de iniciar la partida.
 - **entrada de jugadores a distancia,** facilitando su conexión al sistema.

Precondiciones:

1. **Crear partida**

Precondicion:*jugador debe tener acceso a la interfaz de usuario del sistema.*

2. **Abrir partida**

Precondicion:*El servidor debe estar en funcionamiento y correctamente configurado para permitir la entrada de jugadores remotos.*

3. **Configurar partida**

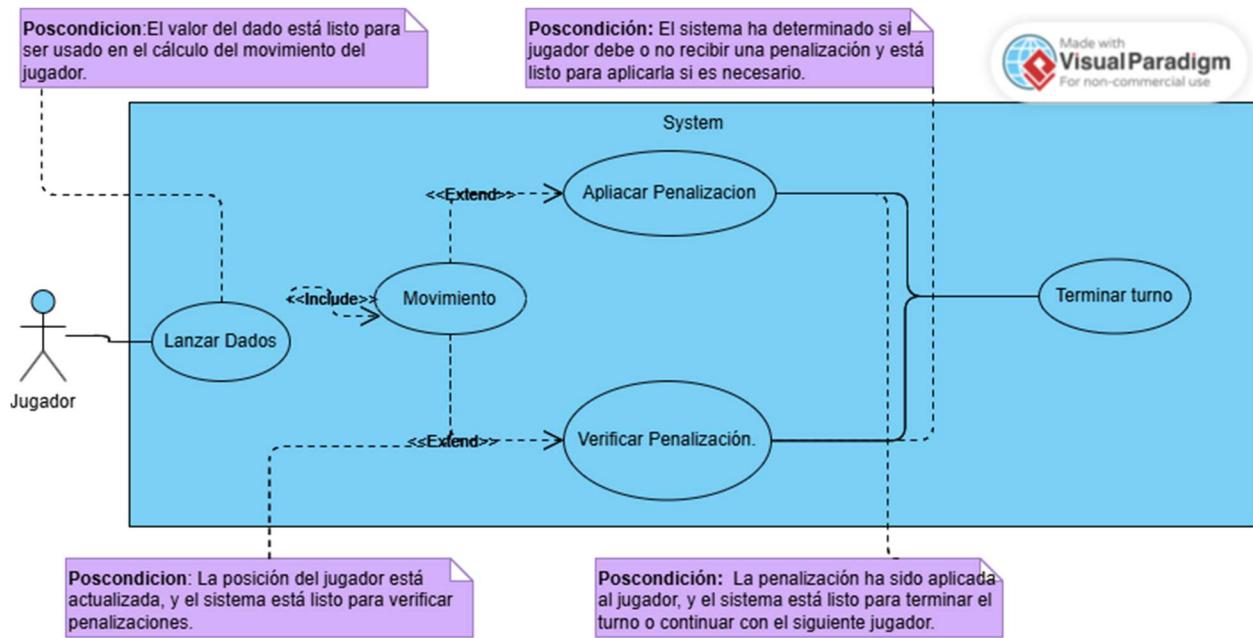
Precondicion:*El sistema debe validar que las configuraciones sean correctas y aplicables.*

Precondicion:*Las configuraciones deben ser validadas para asegurar que están dentro de los límites permitidos por el juego..*

4. **Entrada de jugador a distancia;**

Precondicion:*El servidor debe estar activo y disponible para conexiones remotas.*

Lanzar dados



Caso de Uso: Lanzar Dados

Objetivo: Permitir que el jugador lance el dado para determinar el número de casillas que avanzará.

- **Actores Primarios:** Jugador
- **Precondiciones:**
 - Es el turno del jugador.
- **Postcondiciones:**
 - Se obtiene el número de casillas que el jugador debe avanzar.
 - Se activa el caso de uso **Movimiento**.

Caso de Uso: Movimiento

Objetivo: Mover al jugador a una nueva posición en el tablero con base en el resultado del dado.

- **Actores Primarios:** Jugador
- **Precondiciones:**
 - Se ha lanzado el dado y se tiene un valor de movimiento.
- **Postcondiciones:**
 - La posición del jugador se actualiza en el tablero.
 - Se activa el caso de uso **Verificar Penalización** para verificar si el jugador ha caído en una casilla penalizada.

Caso de Uso: Verificar Penalización

Objetivo: Revisar si la nueva posición del jugador corresponde a una casilla con penalización.

- **Actores Primarios:** Sistema del Juego
- **Precondiciones:**
 - El jugador ha completado su movimiento.
- **Postcondiciones:**
 - Si la casilla es penalizada, se activa **Aplicar Penalización**.
 - Si la casilla no es penalizada, el flujo continúa normalmente hacia el **Final de Turno**.

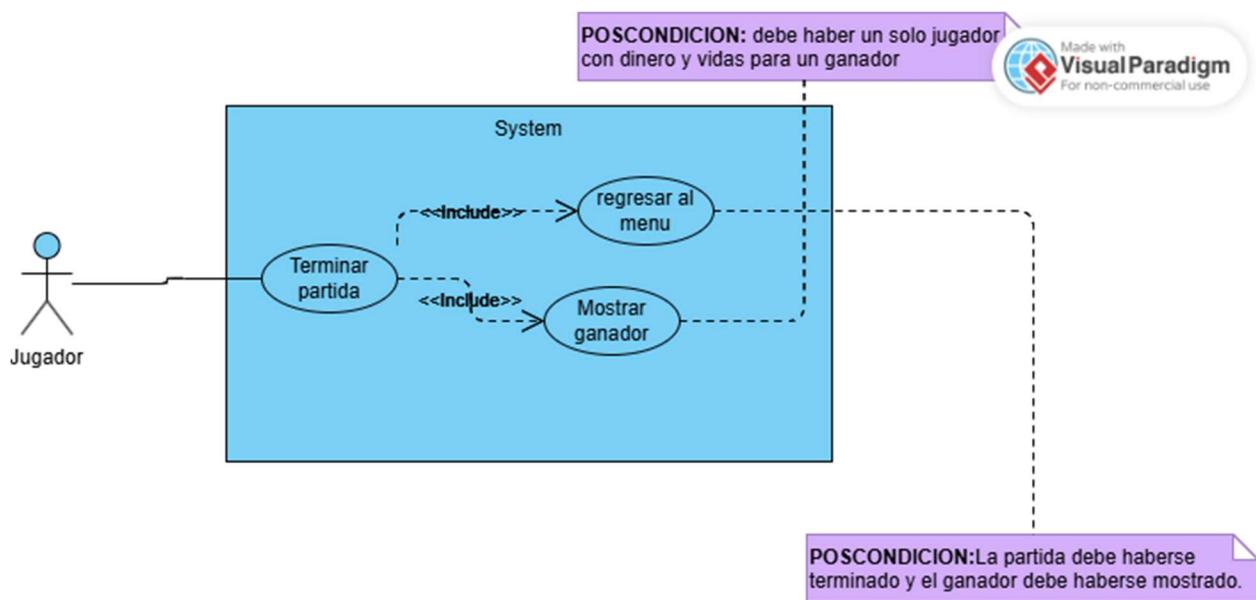
Caso de Uso: Aplicar Penalización

Objetivo: Imponer una penalización al jugador que ha caído en una casilla penalizada.

- **Actores Primarios:** Sistema del Juego
- **Precondiciones:**
 - El jugador ha caído en una casilla penalizada.
- **Postcondiciones:**
 - El jugador sufre la penalización, como perder puntos, retroceder posiciones, etc.

- El flujo continúa normalmente hacia el **Final de Turno**.

Terminar Partida



Caso de Uso con Precondiciones y Postcondiciones

Actores Principales

- **Jugador:** Representa a la persona que interactúa con el sistema para finalizar una partida y ver el resultado.

Casos de Uso

1. Terminar partida:

- **Descripción:** Proceso mediante el cual el jugador finaliza la partida actual.
- **Relaciones:**
 - **Include:** Este caso de uso incluye "Mostrar ganador", lo que significa que al terminar la partida se debe mostrar quién es el ganador.

- **Include:** También incluye "Regresar al menú", lo que implica que una vez que se muestra el ganador, el sistema debe regresar al menú principal.

- **Precondiciones:**

- La partida debe estar en curso.

2. Mostrar ganador:

- **Descripción:** Este caso de uso muestra al jugador el ganador de la partida actual una vez que la partida ha terminado.

- **Relaciones:**

- Es incluido por "Terminar partida", asegurando que se muestre el ganador al finalizar la partida.

- **Postcondiciones:**

- El ganador de la partida debe haberse mostrado correctamente al jugador.

3. Regresar al menú:

- **Descripción:** Permite al jugador volver al menú principal del juego después de terminar una partida y mostrar el ganador.

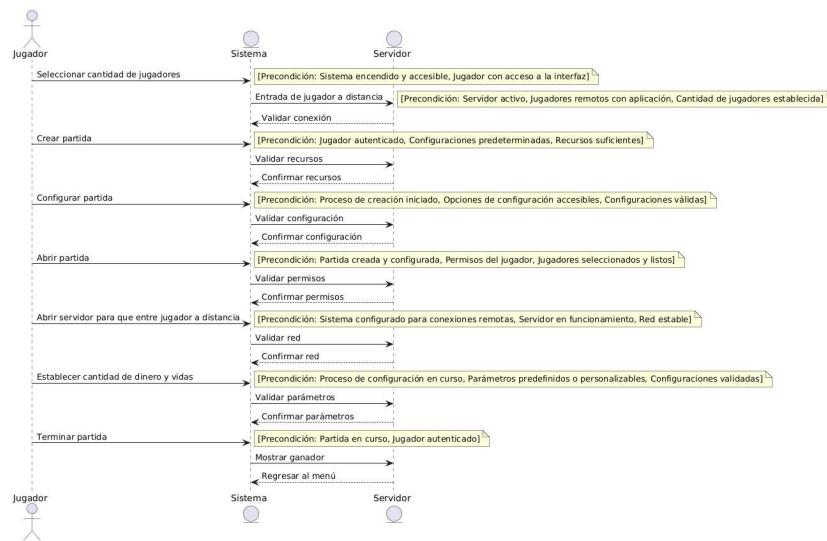
- **Relaciones:**

- Es incluido por "Terminar partida", lo que asegura que el sistema vuelva al menú principal una vez que se haya mostrado el ganador.

- **Precondiciones:**

- La partida debe haberse terminado y el ganador debe haberse mostrado.

Diagrama de Secuencia- Preparación de Partida y Terminar partida

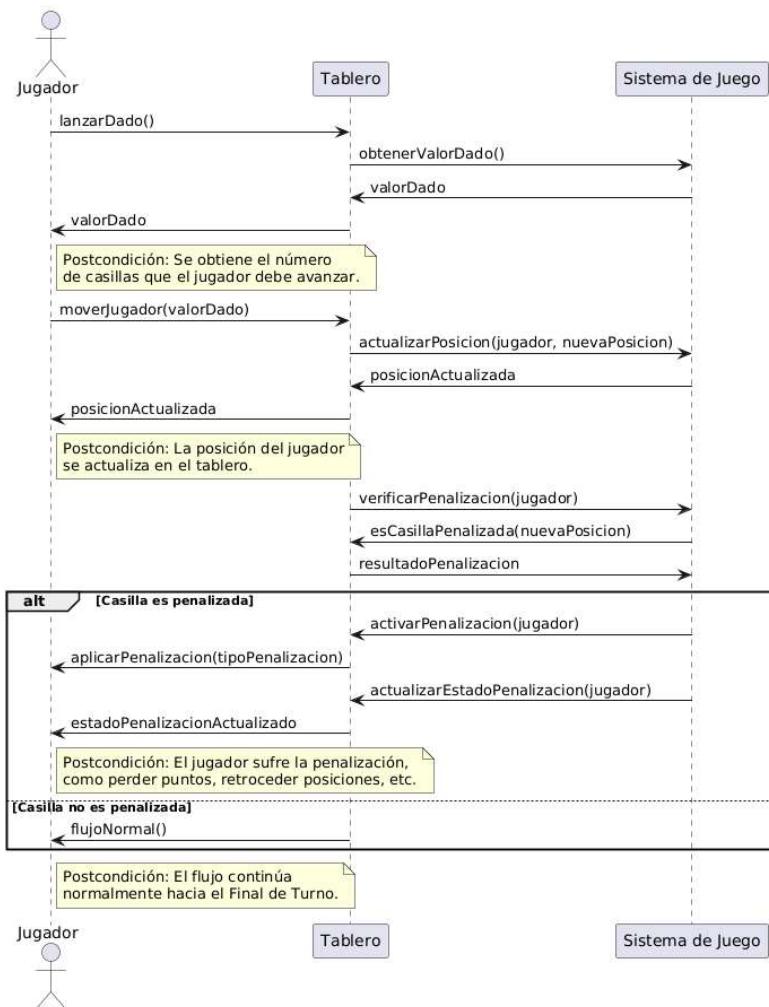


Descripción: representa las interacciones entre tres entidades: Jugador, Sistema y Servidor, en el contexto del juego digital de Patolli. Las principales interacciones son las siguientes:

1. *Seleccionar cantidad de jugadores: El jugador elige cuántos participantes habrá en la partida, lo que requiere que el sistema esté encendido y accesible.*
2. *Entrada de jugador a distancia: El sistema gestiona la entrada de jugadores remotos, validando la conexión con el servidor.*
3. *Crear partida: El jugador solicita crear una partida, lo que implica que debe estar autenticado y que haya recursos suficientes, que son validados por el servidor.*
4. *Configurar partida: El jugador configura opciones del juego, y el sistema verifica que las configuraciones sean válidas mediante el servidor.*
5. *Abrir partida: Se inicia la partida, y el sistema valida que el jugador tenga los permisos necesarios.*

6. Abrir servidor para conexiones remotas: El sistema abre el servidor para que los jugadores remotos se unan, verificando que la red esté estable.
7. Establecer cantidad de dinero y vidas: El jugador configura la cantidad de dinero y vidas iniciales, y el sistema valida estos parámetros. Terminar partida: El jugador finaliza la partida, y el sistema muestra al ganador y regresa al menú principal.

Diagrama de Secuencia- Lanzar Dados



□ Lanzar Dados

- Actor Primario: Jugador
- Precondición: Es el turno del jugador.

- *Postcondición:*
 - *Se obtiene el número de casillas que el jugador debe avanzar.*

□ *Movimiento*

- *Actor Primario: Jugador*
- *Precondición: Se ha lanzado el dado y se tiene un valor de movimiento.*
- *Postcondición:*
 - *La posición del jugador se actualiza en el tablero.*

□ *Verificar Penalización*

- *Actor Primario: Sistema del Juego*
- *Precondición: El jugador ha completado su movimiento.*
- *Postcondición:*
 - *Si la casilla es penalizada, se activa Aplicar Penalización.*
 - *Si la casilla no es penalizada, el flujo continúa normalmente hacia el Final de Turno.*

□ *Aplicar Penalización*

- *Actor Primario: Sistema del Juego*
- *Precondición: El jugador ha caído en una casilla penalizada.*
- *Postcondición:*
 - *El jugador sufre la penalización, como perder puntos, retroceder posiciones, etc.*
 - *El flujo continúa normalmente hacia el Final de Turno.*

5. Código fuente

A. CODIGO JUGADOR

Métodos de Configuración y Funciones

- **Jugador(String nombre)**

Descripción:

Constructor que inicializa el objeto Jugador con el nombre proporcionado. Establece valores iniciales para apuesta, vidas, y posición.

- **void restarVida()**

Descripción:

Disminuye el contador de vidas en 1 si el jugador tiene vidas disponibles. No retorna ningún valor.

- **String getNombre()**

Descripción:

Devuelve el nombre del jugador.

Retorno:

String: El nombre del jugador.

- **int getApuesta()**

Descripción:

Devuelve el monto actual de la apuesta del jugador.

Retorno:

int: El monto de la apuesta actual.

- **void setApuesta(int apuesta)**

Descripción:

Establece un nuevo valor para la apuesta del jugador.



```
public class Jugador {  
    private String nombre;  
    private int apuesta;  
    private int vidas;  
    private int posicion;  
  
    public Jugador(String nombre) {  
        this.nombre = nombre;  
        this.apuesta = 100; // Valor inicial de apuesta  
        this.vidas = 3; // Valor inicial de vidas  
        this.posicion = 0; // Posición inicial en el tablero  
    }  
  
    public void restarVida() {  
        if (vidas > 0) {  
            vidas--;  
        }  
    }  
  
    // Getters y Setters  
    public String getNombre() {  
        return nombre;  
    }  
  
    public int getApuesta() {  
        return apuesta;  
    }  
  
    public void setApuesta(int apuesta) {  
        this.apuesta = apuesta;  
    }  
  
    public int getVidas() {  
        return vidas;  
    }  
  
    public void perderVida() {  
        if (vidas > 0) {  
            vidas--;  
        }  
    }  
}
```

Parámetros:

int apuesta: El nuevo valor de la apuesta.

- ***int getVidas()***

Descripción:

Devuelve el número de vidas restantes del jugador.

Retorno:

int: El número de vidas restantes.

- ***void perderVida()***

Descripción:

Disminuye el contador de vidas en 1 si el jugador tiene vidas disponibles. No retorna ningún valor.

- ***int getPosition()***

Descripción:

Devuelve la posición actual del jugador en el tablero.

Retorno:

int: La posición actual del jugador.

- ***void setPosition(int posicion)***

Descripción:

Establece una nueva posición para el jugador.

Parámetros:

int posicion: La nueva posición del jugador en el tablero.

```
public void setApuesta(int apuesta) {  
    this.apuesta = apuesta;  
}  
  
public int getVidas() {  
    return vidas;  
}  
  
public void perderVida() {  
    if (vidas > 0) {  
        vidas--;  
    }  
}  
  
public int getPosition() {  
    return posicion;  
}  
  
public void setPosition(int posicion) {  
    this.posicion = posicion;  
}
```

B.CODIGO FRMMENU

Métodos de Configuración y Funciones

- **FrmMenu()**

Descripción:

Constructor que inicializa los componentes de la interfaz gráfica del menú principal al crear una nueva instancia de FrmMenu.

- **void llamadaescena1()**

Descripción:

Abre la ventana del tablero para un juego con 2 jugadores. Crea una nueva instancia de FRMTABLERO y la hace visible.

- **void llamadaescena2()**

Descripción:

Abre la ventana del tablero para un juego con 3 jugadores. Crea una nueva instancia de FRMTABLERO3Players y la hace visible.

- **void llamadaescena3()**

Descripción:

Abre la ventana del tablero para un juego con 4 jugadores. Crea una nueva instancia de FRMTABLERO4Players y la hace visible.

- **void salir()**

Descripción:

Cierra la ventana del menú actual y libera los recursos asociados.

```
public FrmMenu() {  
    initComponents();  
}  
private void llamadaescena1(){  
  
    FRMTABLERO vista = new FRMTABLERO();  
    vista.setVisible(true);  
  
}  
private void llamadaescena2(){  
  
    FRMTABLERO3Players vista = new FRMTABLERO3Players();  
    vista.setVisible(true);  
  
}  
private void llamadaescena3(){  
  
    FRMTABLERO4Players vista = new FRMTABLERO4Players();  
    vista.setVisible(true);  
  
}  
private void salir(){  
  
    this.dispose();  
}
```

interfaz



C. CÓDIGO FrmTablero, FrmTablero3jugadoresy FrmTablero4jugadores

Métodos de Configuración y Funciones

mostrarEstadoJugadores()

Muestra el estado actual de todos los jugadores, incluyendo sus vidas y dinero, en un cuadro de diálogo.

Descripción:

- Crea una cadena HTML con la información de cada jugador.
- Muestra un JOptionPane con el estado formateado.

```
public void mostrarEstadoJugadores() {  
    String estado = "<html>Estado de los Jugadores:<br>" +  
        "Jugador 1: Vidas = " + vidaJugador1 + ", Dinero = $" +  
        dineroJugador1 + "<br>" +  
        "Jugador 2: Vidas = " + vidaJugador2 + ", Dinero = $" +  
        dineroJugador2 + "<br>" +  
        "</html>";  
    JLabel label = new JLabel(estado);  
    JOptionPane.showMessageDialog(null, label, "Estado de los Jugadores",  
    JOptionPane.INFORMATION_MESSAGE);  
}
```

Interfaz:



ActualizarInterfaz(int jugador, int resultadoDado)

Actualiza la interfaz gráfica para mostrar el resultado del dado lanzado y el turno del jugador actual.

Parámetros:

- *int jugador: El número del jugador cuyo turno es.*
- *int resultadoDado: El resultado del dado lanzado.*

```
public void actualizarInterfaz(int jugador, int resultadoDado) {  
    int error = 1;  
    int VALORREAL = resultadoDado - error; // Ajustar el resultado según  
    sea necesario  
    jLabel1.setText("Resultado del dado: " + VALORREAL + " - Turno player  
" + (jugador % 4 + 1));  
    JOptionPane.showMessageDialog(null, "Resultado del dado: " + VALORREAL  
+ "\nTurno player " + (jugador % 4 + 1));  
} bel = new JLabel(estado);  
JOptionPane.showMessageDialog(null, label, "Estado de los Jugadores",  
JOptionPane.INFORMATION_MESSAGE);  
}
```

configurarJuego()

Configura el juego solicitando la cantidad inicial de dinero y vidas para cada jugador.

Descripción:

- *Pide al usuario que ingrese valores numéricos para el dinero y vidas iniciales de cada jugador.*
- *Asigna los valores ingresados a cada jugador y muestra*

```
public void configurarJuego() {  
    try {  
        // Solicitar cantidad de dinero inicial  
        String dineroInicialStr = JOptionPane.showInputDialog(null,  
"Ingrese la cantidad de dinero inicial para cada jugador:", "Configuración  
del Juego", JOptionPane.QUESTION_MESSAGE);  
        int dineroInicial = Integer.parseInt(dineroInicialStr);  
        // Solicitar cantidad de vidas inicial  
        String vidasInicialStr = JOptionPane.showInputDialog(null,  
"Ingrese la cantidad de vidas para cada jugador:", "Configuración del  
Juego", JOptionPane.QUESTION_MESSAGE);  
        int vidasInicial = Integer.parseInt(vidasInicialStr);  
        // Asignar el dinero y las vidas a cada jugador  
        dineroJugador1 = dineroInicial;  
        dineroJugador2 = dineroInicial;  
        vidaJugador1 = vidasInicial;  
        vidaJugador2 = vidasInicial;  
        JOptionPane.showMessageDialog(null, "Configuración  
completada:\nDinero inicial: $" + dineroInicial + "\nVidas: " +  
vidasInicial, "Configuración del Juego", JOptionPane.INFORMATION_MESSAGE);  
  
        mostrarEstadoJugadores();  
    } catch (NumberFormatException e) {  
        JOptionPane.showMessageDialog(null, "Por favor, ingrese valores  
numéricos válidos.", "Error de Configuración", JOptionPane.ERROR_MESSAGE);  
        configurarJuego(); // Reiniciar la configuración si hay un error  
    }  
}
```

un mensaje de confirmación.

- En caso de error en la entrada, se reinicia la configuración.

Interfaces:

Configuración del Juego



Ingrese la cantidad de vidas para cada jugador:

2

Aceptar

Cancelar

Configuración del Juego



Ingrese la cantidad de dinero inicial para cada jugador:

20

X

Aceptar

Cancelar



Configuración completada:

Dinero inicial: \$20

Vidas: 2

StartServer(ActionEvent event)

Inicia un servidor que escucha
conexiones entrantes de los clientes.

Descripción:

- **[enviarCapturaDePantalla()]:**

Captura una imagen de la ventana
de la clase FRMTABLERO utilizando
Robot.

- Convierte la imagen a formato base64 y la envía a todos los clientes conectados al servidor.

- Maneja excepciones en caso de errores en la captura o transmisión.

```
// Método para capturar la pantalla de FRMTABLERO y enviarla
private void enviarCapturaDePantalla() {
    try {
        // Obtener las coordenadas y tamaño de la ventana (FRMTABLERO)
        Rectangle frameBounds = this.getBounds();

        // Crear un robot para capturar la pantalla
        Robot robot = new Robot();
        BufferedImage screenCapture = robot.createScreenCapture(frameBounds); // Captura solo el área
        de la ventana

        // Convertir la imagen capturada a base64
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ImageIO.write(screenCapture, "PNG", baos);
        byte[] imageBytes = baos.toByteArray();
        String base64Image = Base64.getEncoder().encodeToString(imageBytes);

        // Enviar la imagen a todos los clientes
        synchronized (clientes) {
            for (PrintWriter out : clientes) {
                out.println("IMAGE:" + base64Image);
            }
        }
    } catch (AWTException | IOException e) {
        System.out.println("Error al capturar la pantalla: " + e.getMessage());
    }
}

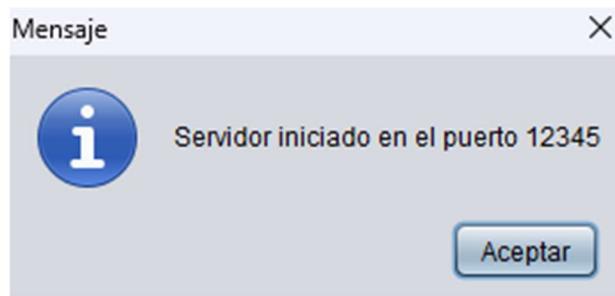
// Método para enviar la captura de pantalla periódicamente (cada 5 segundos)
public void enviarCapturasPeriodicas() {
    Timer timer = new Timer();
    timer.scheduleAtFixedRate(new TimerTask() {
        @Override
        public void run() {
            enviarCapturaDePantalla();
        }
    }, 0, 1000); // Enviar cada 5 segundos
}

// Método para iniciar el servidor
public void startServer() {
    try {
        serverSocket = new ServerSocket(PORT);
        System.out.println("Servidor iniciado en el puerto " + PORT);
        JOptionPane.showMessageDialog(null, "Servidor iniciado en el puerto " + PORT);
        // Escuchar en un hilo separado
        new Thread(() -> {
            while (true) {
                try {
                    Socket clientSocket = serverSocket.accept();
                    System.out.println("Cliente conectado: " + clientSocket.getInetAddress());

                    // Manejar la comunicación con el cliente
                    handleClient(clientSocket);
                } catch (IOException e) {
                    System.out.println("Error al aceptar la conexión: " + e.getMessage());
                }
            }
        }).start();
    } catch (IOException e) {
        System.out.println("Error al iniciar el servidor: " + e.getMessage());
    }
}
return go(f, seed, []);
}
```

- `enviarCapturasPeriodicas():`
 - Envía una captura de pantalla cada 5 segundos.
 - Utiliza un Timer para ejecutar periódicamente el método `enviarCapturaDePantalla`.
 - Llama a `handleClient(Socket clientSocket)` para manejar la comunicación con el cliente.

Interfaz



handleClient(Socket clientSocket)

Maneja la comunicación con un cliente conectado.

Parámetros:

- *Socket clientSocket: El socket del cliente conectado.*

Descripción:

- *Lee mensajes del cliente y actúa en función del contenido.*
- *Si el mensaje indica "AVANZAR", mueve al jugador correspondiente.*

inicializarPosiciones()

Inicializa las posiciones de los jugadores en el tablero.

Descripción:

- *Establece el texto y color de cada JTextField(jugadores) correspondiente a la posición inicial de cada jugador en el tablero.*

```
public void handleClient(Socket clientSocket) {
    try (BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()))) {
        String message;
        while ((message = in.readLine()) != null) {
            // Suponemos que el mensaje tiene el formato
            "PLAYER:X:AVANZAR"
            String[] partes = message.split(":");
            if (partes.length == 3 && "AVANZAR".equals(partes[2])) {
                int jugador = Integer.parseInt(partes[1]); // Obtener el
                número del jugador
                switch (jugador) {
                    case 1:
                        random1(); // Mover al jugador 1
                        break;
                    case 2:
                        random2(); // Mover al jugador 2
                        break;

                    default:
                        System.out.println("Jugador no válido: " +
jugador);
                        break;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                clientSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public void inicializarPosiciones() {
    // Posición inicial del jugador 1 en jTextField1
    JTextField field1 = getFieldByIndex(1); // Primero lo ubicamos en la
    casilla 1
    if (field1 != null) {
        field1.setText("Jugador 1");
        field1.setForeground(Color.RED);
    }

    // Posición inicial del jugador 2 en jTextField13
    JTextField field2 = getFieldByIndex(13); // Posición inicial en la
    casilla 13
    if (field2 != null) {
        field2.setText("Jugador 2");
        field2.setForeground(Color.BLUE);
    }

    // Actualizamos los índices iniciales
    currentFieldIndex1 = 1; // Jugador 1
    currentFieldIndex2 = 13; // Jugador 2
}
```

Métodos Random

*random1(), random2(),random3()y
random4(),*

*Estos métodos manejan el avance de
cada jugador en el tablero.*

Descripción Común:

- *Crea una instancia de Dados, lanza el dado y calcula la nueva posición del jugador.*
- *Actualiza la interfaz gráfica y verifica penalizaciones y colisiones.*

```
/METODOS RANDOM
public void random1() {
    Dados dados = new Dados(); // Crear instancia de Dados
    int resultadoDado = dados.lanzarDado(); // Lanzar el dado
    int nuevaPosicion = currentFieldIndex1;

    for (int i = 0; i < resultadoDado; i++) {
        cambiarEstadoJugadorPlayer1();
        nuevaPosicion++;
    }

    // Actualizar la posición del jugador 1
    currentFieldIndex1 = nuevaPosicion;

    // Actualizar la interfaz de usuario
    actualizarInterfaz(1, resultadoDado);
    verificarPenalizacion(1, nuevaPosicion);
    verificarColision(1, nuevaPosicion);
}

private void random2() {
    Dados dados = new Dados(); // Crear instancia de Dados
    int resultadoDado = dados.lanzarDado(); // Lanzar el dado
    int nuevaPosicion = currentFieldIndex2;

    for (int i = 0; i < resultadoDado; i++) {
        cambiarEstadoJugadorPlayer2();
        nuevaPosicion++;
    }

    // Actualizar la posición del jugador 2
    currentFieldIndex2 = nuevaPosicion;

    // Actualizar la interfaz de usuario
    actualizarInterfaz(2, resultadoDado);
    verificarPenalizacion(2, nuevaPosicion);
    verificarColision(2, nuevaPosicion);
}

private void random3() {
    Dados dados = new Dados(); // Crear instancia de Dados
    int resultadoDado = dados.lanzarDado(); // Lanzar el dado
    int nuevaPosicion = currentFieldIndex3;

    for (int i = 0; i < resultadoDado; i++) {
        cambiarEstadoJugadorPlayer3();
        nuevaPosicion++;
    }

    // Actualizar la posición del jugador 3
    currentFieldIndex3 = nuevaPosicion;

    // Actualizar la interfaz de usuario
    actualizarInterfaz(3, resultadoDado);
    verificarPenalizacion(3, nuevaPosicion);
    verificarColision(3, nuevaPosicion);
}

private void random4() {
    Dados dados = new Dados(); // Crear instancia de Dados
    int resultadoDado = dados.lanzarDado(); // Lanzar el dado
    int nuevaPosicion = currentFieldIndex4;

    for (int i = 0; i < resultadoDado; i++) {
        cambiarEstadoJugadorPlayer4();
        nuevaPosicion++;
    }

    // Actualizar la posición del jugador 4
    currentFieldIndex4 = nuevaPosicion;

    // Actualizar la interfaz de usuario
    actualizarInterfaz(4, resultadoDado);
    verificarPenalizacion(4, nuevaPosicion);
    verificarColision(4, nuevaPosicion);
}
```

Métodos de Cambio de Estado

`cambiarEstadoJugadorPlayer1()`,
`cambiarEstadoJugadorPlayer2()`,

Estos métodos cambian el estado de cada jugador en el tablero.

Descripción Común:

- *Actualizan el texto y color del JTextField correspondiente al jugador que está avanzando.*
- *Manejan el cambio de la casilla anterior a "casilla X" y establecen el texto del campo actual a "jugador X".*

```
public void cambiarEstadoJugadorPlayer2() {  
    // Cambia el texto de los JTextFields según el índice  
    int previousFieldIndex = currentFieldIndex2 - 1;  
    if (previousFieldIndex < 1) {  
        previousFieldIndex = 53; // Si es menor que 1, vuelve al último  
        JTextField  
    }  
  
    // Cambia el texto del JTextField anterior a "casilla X"  
    JTextField previousField = getFieldByIndex(previousFieldIndex);  
    if (previousField != null) {  
        previousField.setText("casilla " + previousFieldIndex);  
        previousField.setForeground(Color.BLACK); // Cambia el color del  
        texto a negro  
    }  
  
    // Cambia el texto del JTextField actual a "jugador 2" y cambia el  
    color  
    JTextField currentField = getFieldByIndex(currentFieldIndex2);  
    if (currentField != null) {  
        currentField.setText("jugador 2");  
        currentField.setForeground(Color.BLUE); // Cambia el color del  
        texto a azul  
    }  
  
    // Incrementa el índice para la próxima llamada  
    currentFieldIndex2++;  
  
    // Reinicia el índice si supera el número de JTextFields  
    if (currentFieldIndex2 > 53) {  
        currentFieldIndex2 = 1;  
    }  
}  
  
public void cambiarEstadoJugadorPlayer1() {  
    // Cambia el texto de los JTextFields según el índice  
    int previousFieldIndex = currentFieldIndex1 - 1;  
    if (previousFieldIndex < 1) {  
        previousFieldIndex = 53; // Si es menor que 1, vuelve al último  
        JTextField (en este caso, 52)  
    }  
  
    // Cambia el texto del JTextField anterior a "casilla X"  
    JTextField previousField = getFieldByIndex(previousFieldIndex);  
    if (previousField != null) {  
        previousField.setText("casilla " + previousFieldIndex);  
        previousField.setForeground(Color.BLACK); // Cambia el color del  
        texto a negro  
    }  
  
    // Cambia el texto del JTextField actual a "jugador 1" y cambia el  
    color  
    JTextField currentField = getFieldByIndex(currentFieldIndex1);  
    if (currentField != null) {  
        currentField.setText("jugador 1");  
        currentField.setForeground(Color.red); // Cambia el color del  
        texto a rojo  
    }  
  
    // Incrementa el índice para la próxima llamada  
    currentFieldIndex1++;  
  
    // Reinicia el índice si supera el número de JTextFields  
    if (currentFieldIndex1 > 53) {  
        currentFieldIndex1 = 1;  
    }  
}
```

getFieldByIndex(int index)

Devuelve el JTextField correspondiente a un índice dado.

Parámetros:

- *int index: El índice del campo a obtener.*

Descripción:

- *Busca y devuelve el JTextField en función del índice proporcionado.*

```
public JTextField getFieldByIndex(int index) {  
    switch (index) {  
        case 1: return jTextField1;  
        case 2: return jTextField2;  
        case 3: return jTextField3;  
        case 4: return jTextField4;  
        case 5: return jTextField5;  
        case 6: return jTextField6;  
        case 7: return jTextField7;  
        case 8: return jTextField8;  
        case 9: return jTextField9;  
        case 10: return jTextField10;  
        case 11: return jTextField11;  
        case 12: return jTextField12;  
        case 13: return jTextField13;  
        case 14: return jTextField14;  
        case 15: return jTextField15;  
        case 16: return jTextField16;  
        case 17: return jTextField17;  
        case 18: return jTextField18;  
        case 19: return jTextField19;  
        case 20: return jTextField20;  
        case 21: return jTextField21;  
        case 22: return jTextField22;  
        case 23: return jTextField23;  
        case 24: return jTextField24;  
        case 25: return jTextField25;  
  
        case 27: return jTextField27;  
        case 28: return jTextField28;  
        case 29: return jTextField29;  
        case 30: return jTextField30;  
        case 31: return jTextField31;  
        case 32: return jTextField32;  
        case 33: return jTextField33;  
        case 34: return jTextField34;  
        case 35: return jTextField35;  
        case 36: return jTextField36;  
        case 37: return jTextField37;  
        case 38: return jTextField38;  
        case 39: return jTextField39;  
        case 40: return jTextField40;  
        case 41: return jTextField41;  
        case 42: return jTextField42;  
        case 43: return jTextField43;  
        case 44: return jTextField44;  
        case 45: return jTextField45;  
        case 46: return jTextField46;  
        case 47: return jTextField47;  
        case 48: return jTextField48;  
        case 49: return jTextField49;  
        case 50: return jTextField50;  
        case 51: return jTextField51;  
        case 52: return jTextField52;  
        case 53: return jTextField53;  
        default: return null; // Retorna null si el índice no es válido  
    }  
}
```

Código cliente

Código cliente

Client()

Descripción:

Constructor que configura la interfaz gráfica del cliente del juego y establece la conexión con el servidor.

Crea una ventana con un JComboBox para seleccionar jugadores, un botón "AVANZAR", y un área de texto para mostrar mensajes.

Implementa la acción del botón para enviar el comando correspondiente al jugador seleccionado.

Inicia un hilo para manejar los mensajes recibidos del servidor y los muestra en el área de texto.

□ *void conectarAlServidor()*

Descripción:

Establece una conexión con el servidor utilizando la dirección y el puerto definidos.

Inicializa el socket y el flujo de salida. Si la conexión es exitosa, imprime un mensaje en la consola.

□ *void enviarComando(int jugador)*

Descripción:

Envía un comando al servidor indicando que el jugador seleccionado desea avanzar.

Construye un comando en el formato "PLAYER:<jugador>

" y lo envía a través del flujo de salida.

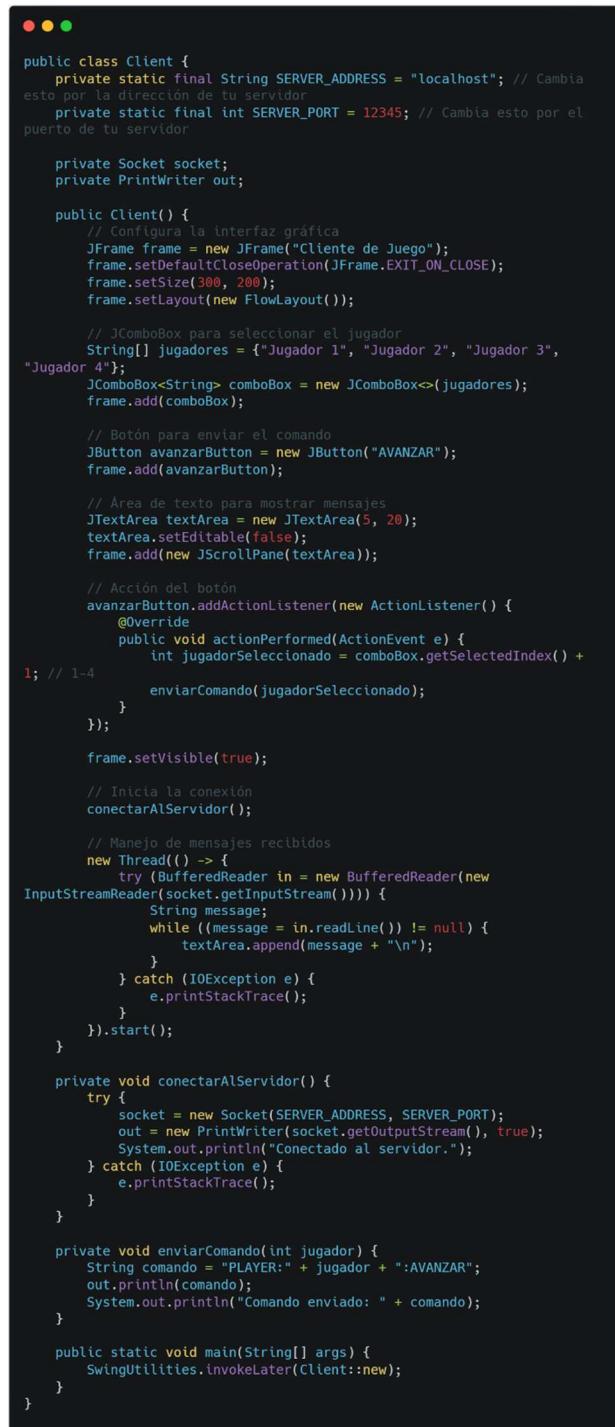
Imprime el comando enviado en la consola.

□ *static void main(String[] args)*

Descripción:

Método principal que inicia la aplicación cliente en la interfaz gráfica de Swing.

Invoca el constructor de la clase Client en el hilo de eventos de Swing.



```
public class Client {
    private static final String SERVER_ADDRESS = "localhost"; // Cambia esto por la dirección de tu servidor
    private static final int SERVER_PORT = 12345; // Cambia esto por el puerto de tu servidor

    private Socket socket;
    private PrintWriter out;

    public Client() {
        // Configura la interfaz gráfica
        JFrame frame = new JFrame("Cliente de Juego");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());

        // JComboBox para seleccionar el jugador
        String[] jugadores = {"Jugador 1", "Jugador 2", "Jugador 3", "Jugador 4"};
        JComboBox<String> comboBox = new JComboBox<>(jugadores);
        frame.add(comboBox);

        // Botón para enviar el comando
        JButton avanzarButton = new JButton("AVANZAR");
        frame.add(avanzarButton);

        // Área de texto para mostrar mensajes
        JTextArea textArea = new JTextArea(5, 20);
        textArea.setEditable(false);
        frame.add(new JScrollPane(textArea));

        // Acción del botón
        avanzarButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                int jugadorSeleccionado = comboBox.getSelectedIndex() + 1; // 1-4
                enviarComando(jugadorSeleccionado);
            }
        });
        frame.setVisible(true);

        // Inicia la conexión
        conectarAlServidor();

        // Manejo de mensajes recibidos
        new Thread(() -> {
            try (BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {
                String message;
                while ((message = in.readLine()) != null) {
                    textArea.append(message + "\n");
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }).start();
    }

    private void conectarAlServidor() {
        try {
            socket = new Socket(SERVER_ADDRESS, SERVER_PORT);
            out = new PrintWriter(socket.getOutputStream(), true);
            System.out.println("Conectado al servidor.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void enviarComando(int jugador) {
        String comando = "PLAYER:" + jugador + ":AVANZAR";
        out.println(comando);
        System.out.println("Comando enviado: " + comando);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(Client::new);
    }
}
```

Interfaz



Código Dados

Métodos

- *int lanzarDado()*

Descripción:

Lanza un dado y devuelve un número aleatorio entre 1 y 6.

- *Detalles:*

- *Crea una instancia de la clase Random para generar el número aleatorio.*
- *Utiliza el método nextInt(6) para obtener un valor entre 0 y 5, y luego le suma 1 para obtener un rango de 1 a 6.*
- *Imprime el resultado del lanzamiento en la consola para propósitos de depuración.*

- *Retorno:*

int resultado - Un número aleatorio entre 1 y 6, que representa el resultado del lanzamiento del dado.

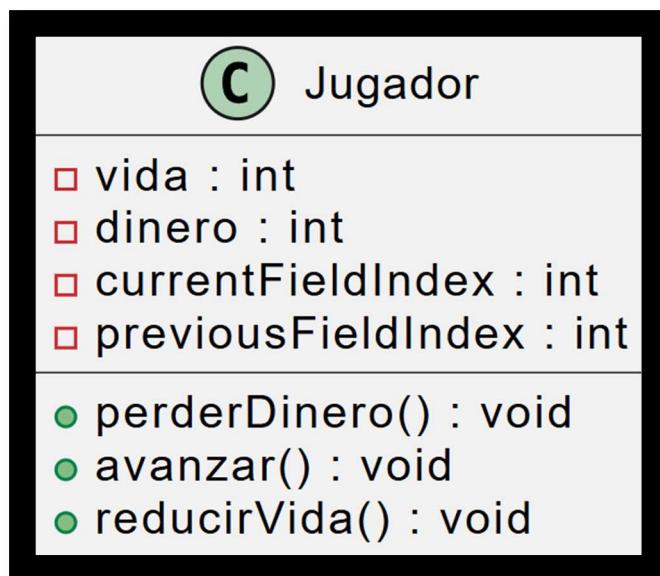
```
public class Dados {  
    public int lanzarDado() {  
        Random random = new Random();  
        int resultado = random.nextInt(6) + 1;  
        System.out.println("Resultado del dado: " + resultado); // Depuración  
        return resultado;  
    }  
}
```

4. Diagramas de clases

a. Jugador

Clase Jugador

- **Descripción:** Representa a un jugador en el juego, gestionando su estado y acciones.
- **Métodos:**
 - + perderDinero(): void - Método público que reduce la cantidad de dinero del jugador.
 - + avanzar(): void - Método público que permite al jugador avanzar en el tablero.
 - + reducirVida(): void - Método público que disminuye la vida del jugador.
- **Atributos:**
 - - vida: int - La vida actual del jugador.
 - - dinero: int - La cantidad de dinero del jugador.
 - - currentFieldIndex: int - El índice del campo actual del jugador en el tablero.
 - - previousFieldIndex: int - El índice del campo anterior del jugador.



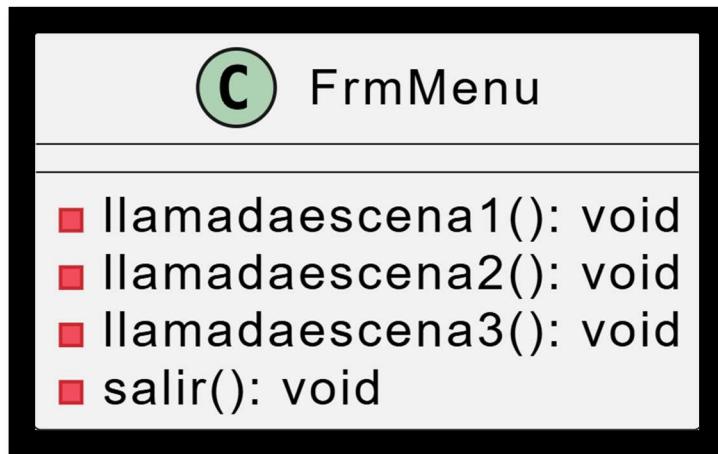
Patrones:

- **Patrón Strategy:** Podrías usar este patrón para manejar acciones específicas de los jugadores, como perderDinero() o avanzar(). Esto permite que diferentes tipos de jugadores tengan variaciones en sus estrategias de movimiento, vida, o finanzas.
- **Observer:** Si quieres que el jugador notifique a otros componentes (como la interfaz de usuario o el sistema de red) cada vez que sus atributos cambien, puedes hacer que Jugador sea un observable. Así, cuando el jugador pierde dinero o vida, se notificaría automáticamente a otros objetos suscriptores (como FRMTABLERO o FrmMenu).

b. Frmmenu

clase FrmMenu

- **Descripción:** Representa la interfaz de menú principal del juego, donde los usuarios pueden seleccionar el tipo de juego que desean iniciar.
- **Métodos:**
 - - *llamadaescena1(): void* - Método privado que probablemente abre la escena correspondiente a un juego de 2 jugadores.
 - - *llamadaescena2(): void* - Método privado que abre la escena correspondiente a un juego de 3 jugadores.
 - - *llamadaescena3(): void* - Método privado que abre la escena correspondiente a un juego de 4 jugadores.
 - - *salir(): void* - Método privado que cierra la aplicación.



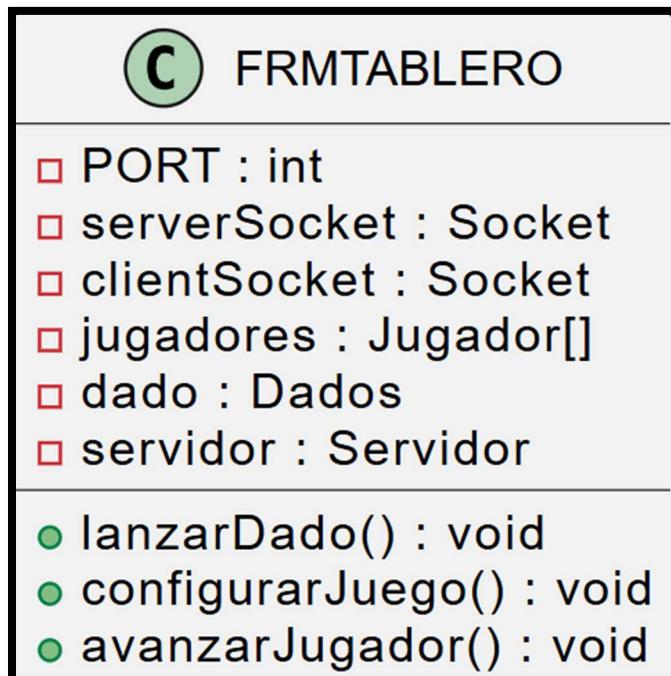
Patrones:

- **Patrón Singleton:** Esta clase *FrmMenu* podría beneficiarse del patrón *Singleton*, ya que es común que solo exista una instancia del menú en la aplicación. El patrón *Singleton* asegura que solo existe un único menú principal durante la ejecución.
- **Facade:** Podrías utilizar un patrón *Facade* para simplificar el acceso a varias configuraciones o elementos del menú, especialmente si el menú permite configurar opciones complejas como el número de jugadores, ajustes de red, o selección de niveles. La clase *FrmMenu* actuaría como un "fachada" simplificada para acceder a estas configuraciones.

C.FRMTABLERO

Clase FRMTABLERO

- **Descripción:** Clase base que representa el tablero del juego, donde se llevan a cabo las interacciones del juego.
- **Métodos:**
 - + lanzarDado(): void - Método público que simula el lanzamiento de un dado.
 - + configurarJuego(): void - Método público que configura las reglas y parámetros iniciales del juego.
 - + avanzarJugador(): void - Método público que permite que un jugador avance en el tablero.
- **Atributos:**
 - - PORT: int - El puerto utilizado para la comunicación de red.
 - - serverSocket: Socket - Socket del servidor para aceptar conexiones.
 - - clientSocket: Socket - Socket del cliente para comunicarse con el servidor.
 - - jugadores: Jugador[] - Array de objetos Jugador que representan a los jugadores en el juego.
 - - dado: Dados - Objeto de la clase Dados para gestionar los lanzamientos de dados.
 - - servidor: Servidor - Objeto de la clase Servidor que maneja la lógica de conexión en red.



D.FrmTablero3Jugadores

Clase *FrmTablero3Jugadores*

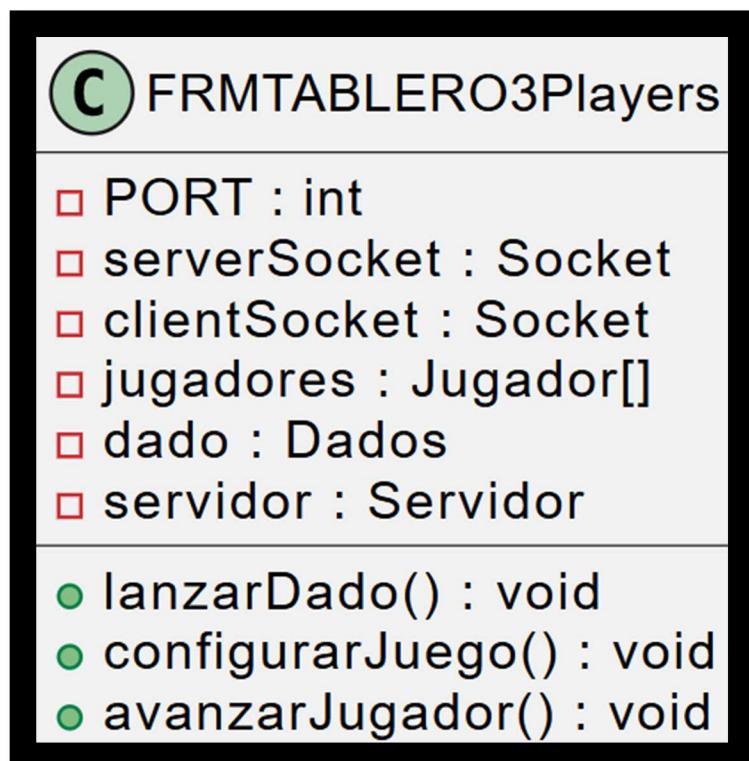
□ **Descripción:** Clase base que representa el tablero del juego, donde se llevan a cabo las interacciones del juego.

□ **Métodos:**

- *+ lanzarDado(): void* - Método público que simula el lanzamiento de un dado.
- *+ configurarJuego(): void* - Método público que configura las reglas y parámetros iniciales del juego.
- *+ avanzarJugador(): void* - Método público que permite que un jugador avance en el tablero.

□ **Atributos:**

- *- PORT: int* - El puerto utilizado para la comunicación de red.
- *- serverSocket: Socket* - Socket del servidor para aceptar conexiones.
- *- clientSocket: Socket* - Socket del cliente para comunicarse con el servidor.
- *- jugadores: Jugador[]* - Array de objetos Jugador que representan a los jugadores en el juego.
- *- dado: Dados* - Objeto de la clase Dados para gestionar los lanzamientos de dados.
- *- servidor: Servidor* - Objeto de la clase Servidor que maneja la lógica de conexión en red.



FRMTABLERO4PLAYERS

Clase: FRMTABLERO4PLAYERS

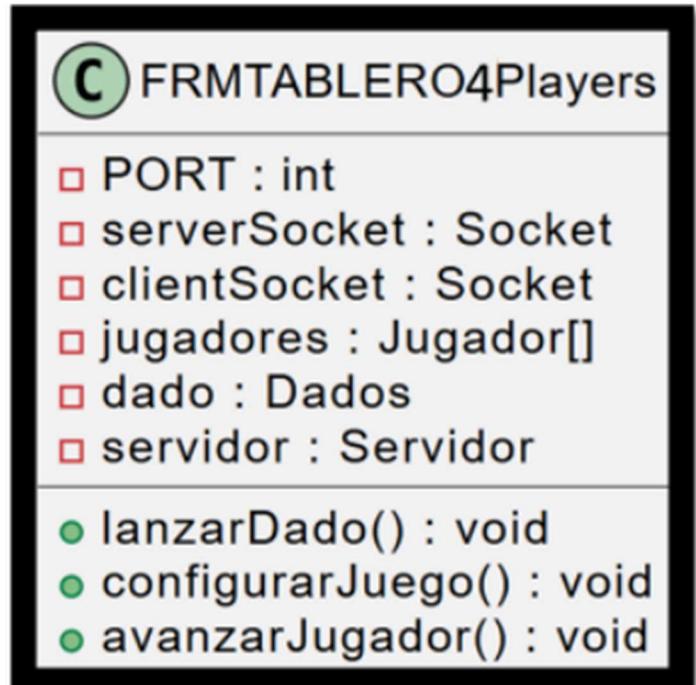
□ **Descripción:** Clase base que representa el tablero del juego, donde se llevan a cabo las interacciones del juego.

□ **Métodos:**

- + lanzarDado(): void - Método público que simula el lanzamiento de un dado.
- + configurarJuego(): void - Método público que configura las reglas y parámetros iniciales del juego.
- + avanzarJugador(): void - Método público que permite que un jugador avance en el tablero.

□ **Atributos:**

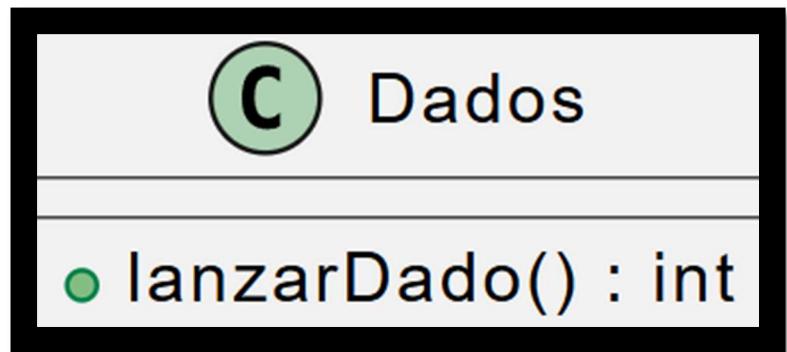
- - PORT: int - El puerto utilizado para la comunicación de red.
- - serverSocket: Socket - Socket del servidor para aceptar conexiones.
- - clientSocket: Socket - Socket del cliente para comunicarse con el servidor.
- - jugadores: Jugador[] - Array de objetos Jugador que representan a los jugadores en el juego.
- - dado: Dados - Objeto de la clase Dados para gestionar los lanzamientos de dados.
- - servidor: Servidor - Objeto de la clase Servidor que maneja la lógica de conexión en red.



Datos

Clase Dados

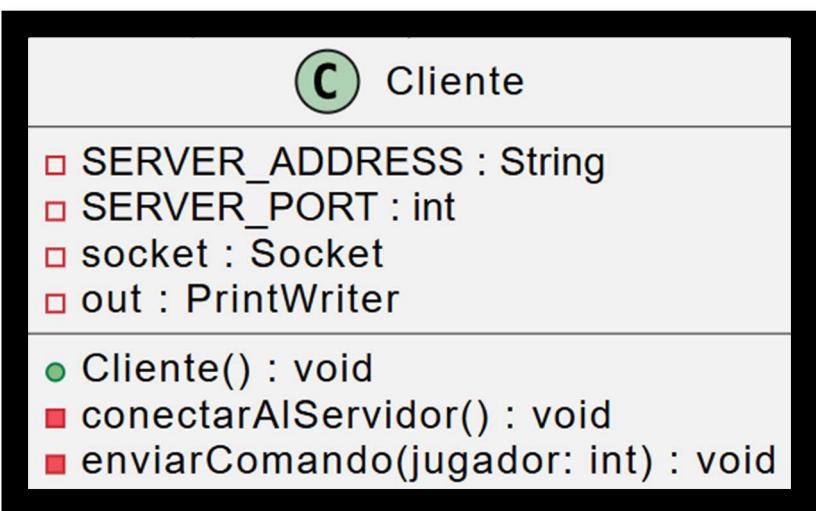
- **Descripción:** Maneja la lógica del lanzamiento de un dado en el juego.
- **Métodos:**
 - + lanzarDado(): int - Método público que simula el lanzamiento de un dado y devuelve el resultado.



CLIENTE:

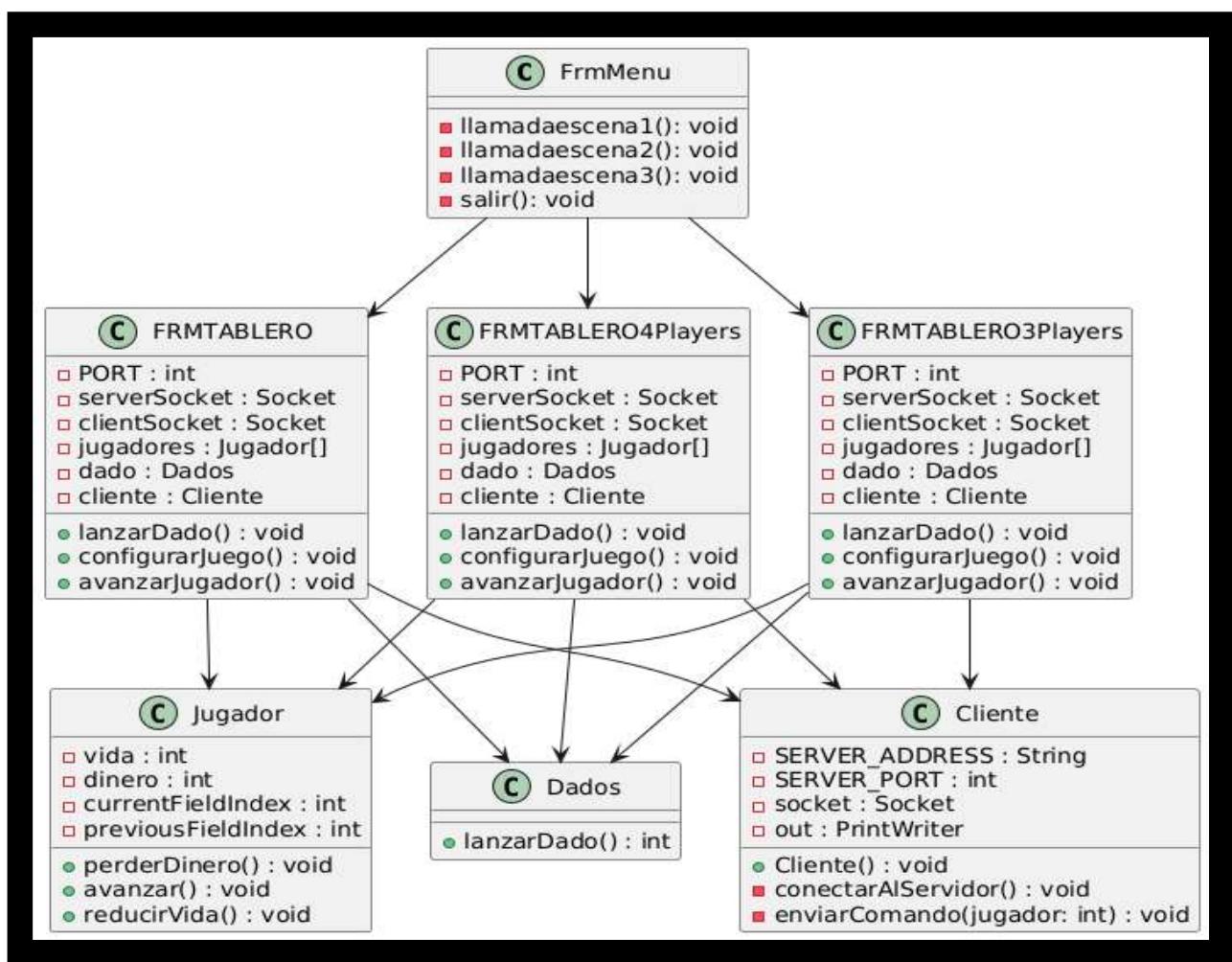
Clase Cliente

- **Descripción:** Representa la interfaz del cliente del juego, que permite a los jugadores interactuar con el servidor del juego. Se encarga de enviar comandos y recibir mensajes desde el servidor.
- **Atributos:**
 - - SERVER_ADDRESS : String - Dirección del servidor al que se conecta (por defecto, "localhost").
 - - SERVER_PORT : int - Puerto del servidor al que se conecta (por defecto, 12345).
 - - socket : Socket - Socket del cliente para la comunicación con el servidor.
 - - out : PrintWriter - Flujo de salida para enviar datos al servidor.
- **Métodos:**
 - ● Cliente() : void
 - ■ conectarAlServidor() : void
 - ■ enviarComando(jugador: int) : void



- *+ Cliente() : void* - Constructor que inicializa la interfaz gráfica del cliente y establece la conexión con el servidor.
- *- conectarAlServidor() : void* - Método privado que establece la conexión con el servidor, utilizando la dirección y el puerto definidos.
- *- enviarComando(jugador: int) : void* - Método privado que envía un comando al servidor indicando que el jugador seleccionado debe avanzar en el juego.

TRABAJO CONJUNTO DE LAS CLASES



Patrones de Diseño Aplicados

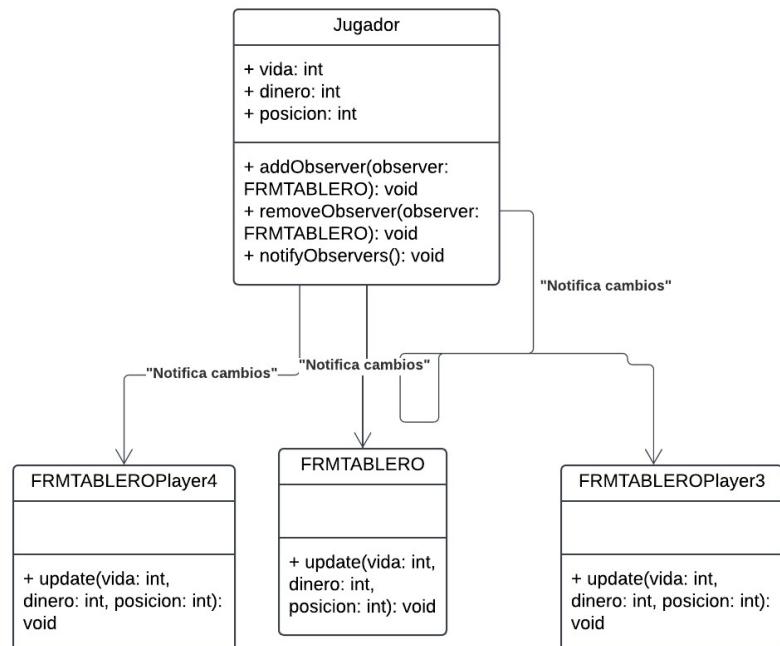
A. Observer

- **Aplicado a:** Clases Jugador (Observable) y FRMTABLERO, FRMTABLEROPlayer3, FRMTABLEROPlayer4 (Observadores).
- **Propósito:** Permitir que los objetos que dependen del estado de Jugador se actualicen automáticamente cada vez que el jugador cambia de estado.

Descripción:

- Jugador puede implementar una interfaz de Observable, notificando a otras clases (como las distintas versiones de FRMTABLERO) cada vez que se modifique un atributo importante (como vida, dinero o posición).
- FRMTABLERO, FRMTABLEROPlayer3, y FRMTABLEROPlayer4 actúan como observadores que reciben estas notificaciones y actualizan sus interfaces o estados del juego según sea necesario, manteniendo la coherencia visual y funcional en el juego.

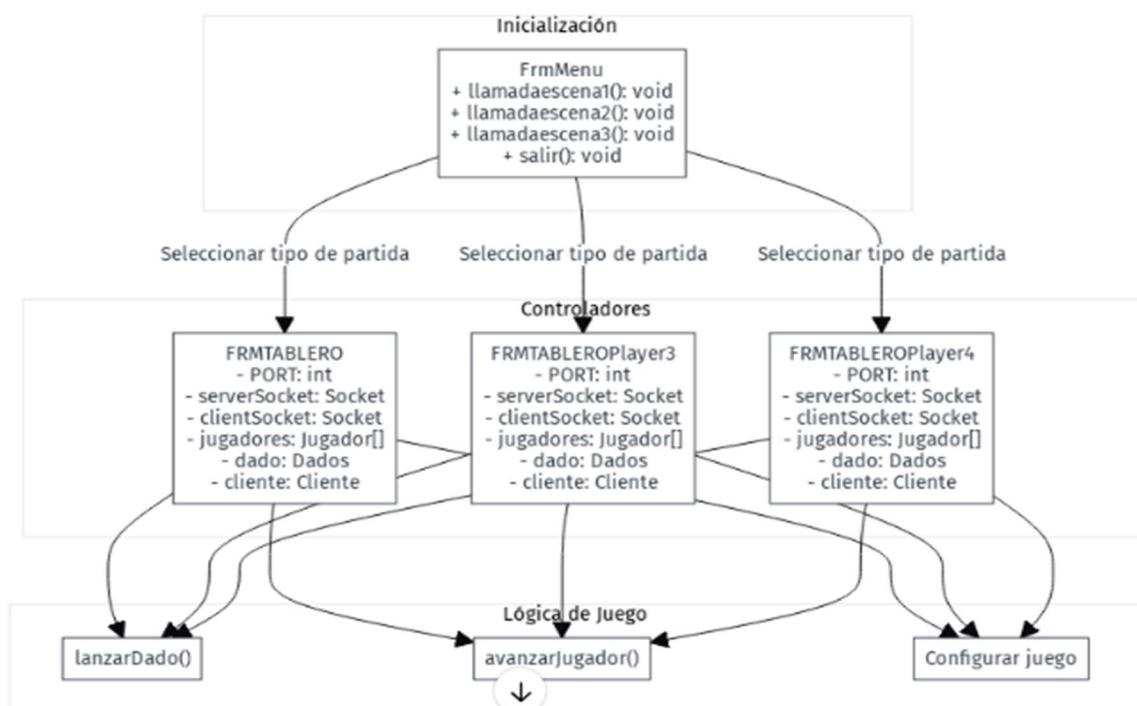
Diagramas De Patrón:



B.Controller

- **Aplicado a:** Clases *FRMTABLERO*, *FRMTABLEROPlayer3*, *FRMTABLEROPlayer4*, y *FrmMenu*.
- **Propósito:** Centralizar la lógica de cada versión del tablero, separando la lógica del juego de la interfaz gráfica.
- **Descripción:**
 - *FRMTABLERO*, *FRMTABLEROPlayer3*, y *FRMTABLEROPlayer4* actúan como controladores en sus respectivas configuraciones de partida, gestionando la interacción de los jugadores con el tablero y controlando acciones como *lanzarDado()*, *avanzarJugador()*, y la configuración inicial del juego.
 - *FrmMenu* funciona como el controlador inicial, permitiendo que el usuario seleccione el tipo de partida (2, 3 o 4 jugadores), y luego cede el control al tablero correspondiente.
 - Este patrón organiza la lógica de flujo del juego, manteniéndola independiente de la interfaz gráfica.

Diagramas De clases:



C. Facade

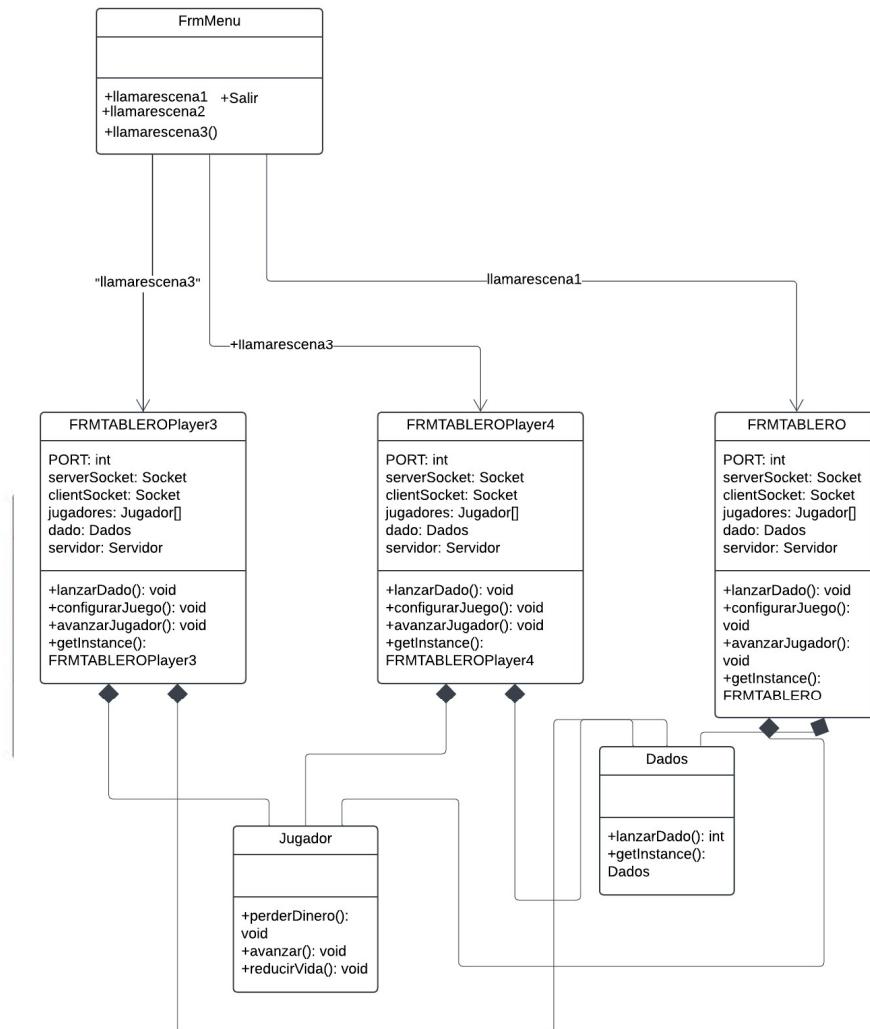
- **Aplicado a:** Clase FrmMenu.
- **Propósito:** Simplificar el acceso a la configuración inicial del juego para el usuario.

Descripción:

El patrón **Facade** simplifica la interacción del usuario con las opciones de configuración del juego, centralizando la lógica de configuración en un único punto de acceso: la clase *FrmMenu*.

- La clase *FrmMenu* ofrece una interfaz clara para que el jugador seleccione el tipo de partida y otras configuraciones, sin necesidad de preocuparse por los detalles internos del sistema.
- Este patrón reduce la complejidad de la interacción del usuario con el sistema y oculta las complejidades detrás de una interfaz sencilla y comprensible.

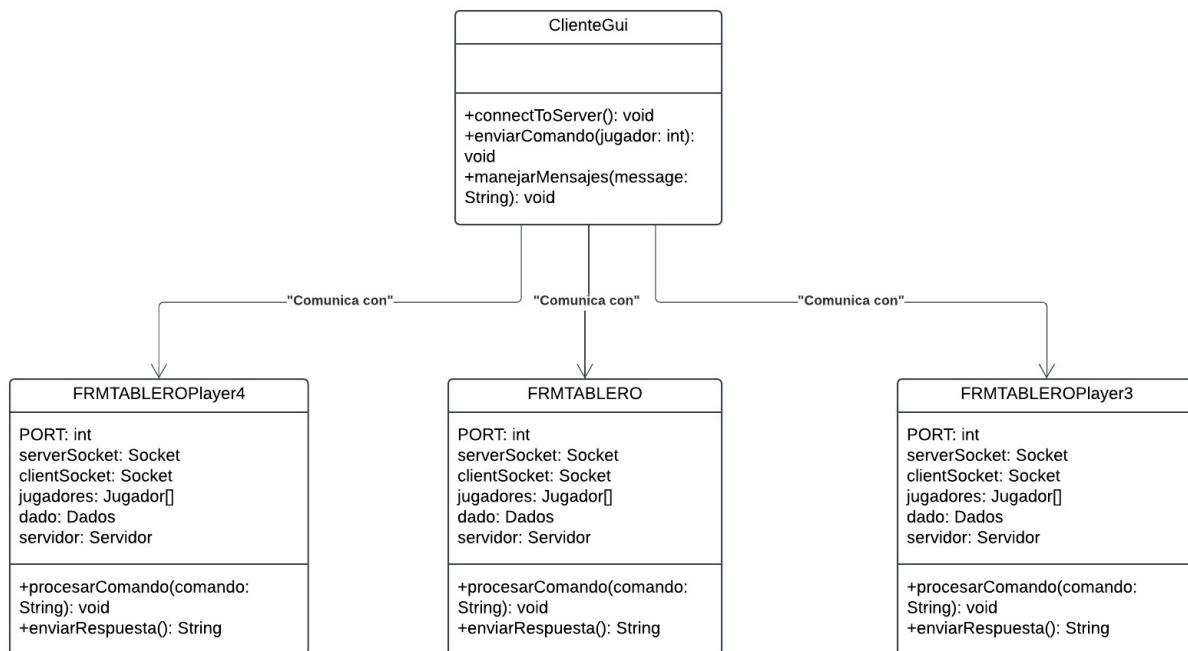
Diagramas De clases:



D. Proxy

- **eAplicado a:** Clase Cliente.
- **Propósito:** Controlar el acceso y comunicación con el servidor en el contexto de un juego en red.
- **Descripción:**
 - Cliente actúa como un Proxy, facilitando la comunicación entre el jugador y el servidor mediante el uso de sockets. Maneja la lógica de conexión (conectar al servidor y enviar comandos) y mantiene la integridad de la conexión con el servidor.
 - Este patrón permite que Cliente controle y filtre las solicitudes que envía cada jugador, asegurando que solo comandos válidos lleguen al servidor.

Diagramas De clase:



E.Singleton

- **Aplicado a:** FRMTABLERO, FRMTABLEROPLAYER3, FRMTABLEROPLAYER4 y Dados
- **Propósito:** Asegurar que solo existe una instancia de estos objetos en la ejecución del juego.
- **Descripción:**
 - FRMTABLERO (o las distintas versiones de tablero) Se implementa como Singleton para que solo exista un tablero por partida, centralizando la lógica y manteniendo la consistencia del juego..
 - Dados es un Singleton se desea que todos los jugadores comparten el mismo dado, evitando la creación de múltiples instancias y permitiendo una simulación centralizada del lanzamiento de dados.

DIAGRAMAS DE CLASES

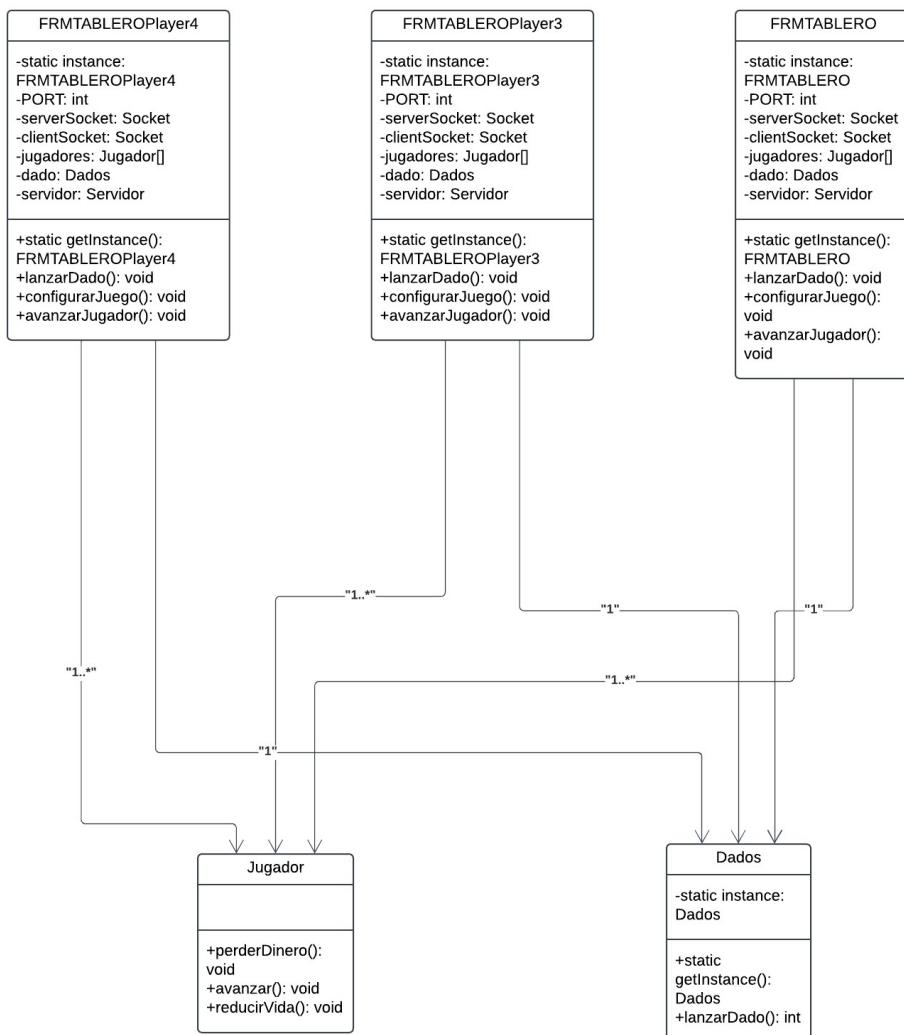
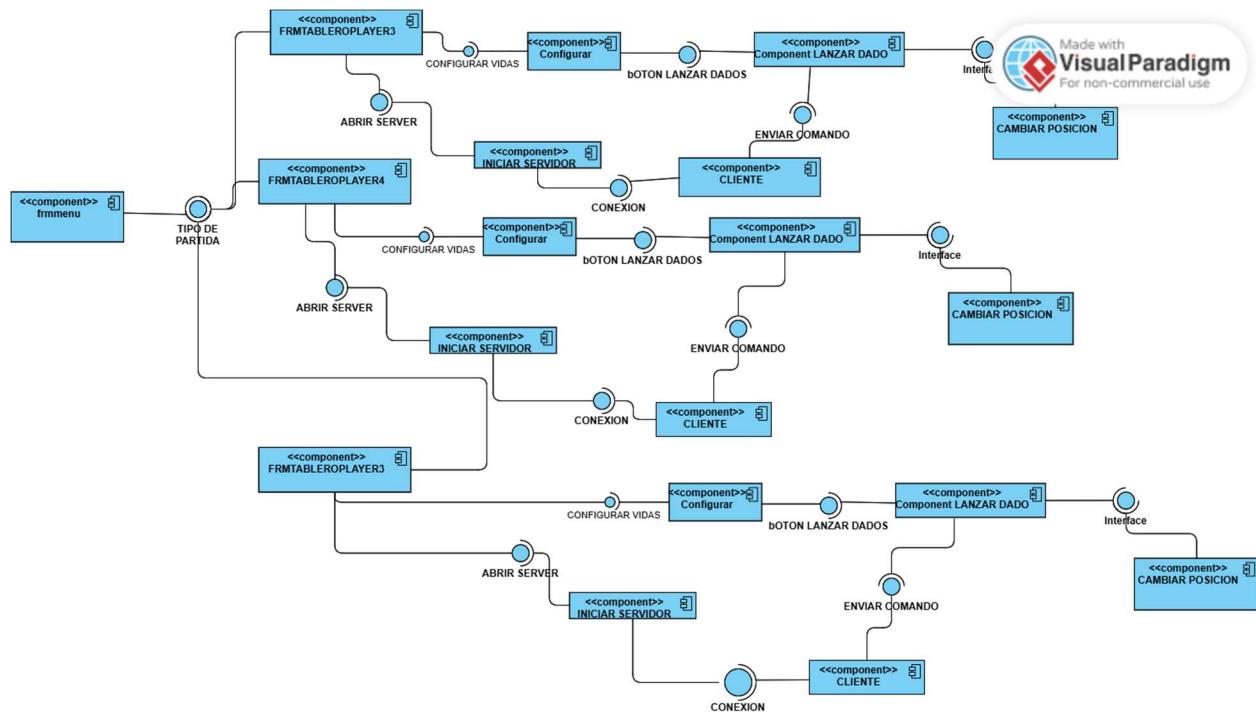


Diagrama de componentes



Componentes y Sus Conexiones:

1. **FRMTABLEROPLAYER3 y FRMTABLEROPLAYER4:**

- **FRMTABLEROPLAYER3 y FRMTABLEROPLAYER4** son componentes que probablemente representan la interfaz gráfica de los jugadores 3 y 4, respectivamente.
- Ambos están conectados al componente **CLIENTE**, lo que sugiere que forman parte de la interfaz del usuario, permitiendo a los jugadores interactuar con el sistema.

2. **Configurar:**

- Este componente se ocupa de la configuración inicial del sistema.
- Está vinculado con **FRMTABLEROPLAYER3 y FRMTABLEROPLAYER4** mediante **CONFIGURAR VIDAS**, lo que indica que el ajuste de vidas de los jugadores se realiza a través de este componente.

- También está conectado al componente **INICIAR SERVIDOR** a través de **ABRIR SERVER**, sugiriendo que puede iniciar configuraciones importantes necesarias para arrancar el servidor.

3. **INICIAR SERVIDOR:**

- Este componente es fundamental para arrancar el servidor que gestionará las conexiones de los clientes.
- Tiene una conexión directa con el componente **CLIENTE** a través de **CONEXION**, lo que permite la interacción del cliente con el servidor para sincronizar y gestionar el juego en red.

4. **CLIENTE:**

- El componente **CLIENTE** es central y actúa como el punto de contacto principal para los usuarios.
- Está conectado a **FRMTABLEROPLAYER3** y **FRMTABLEROPLAYER4**, lo que significa que la interfaz de usuario interactúa directamente con este componente.
- También está vinculado con **Component LANZAR DADO** a través de **BOTON LANZAR DATOS**, permitiendo que los jugadores lancen dados dentro del juego.
- Adicionalmente, tiene una conexión con **CAMBIAR POSICION** a través de **ENVIAR COMANDO**, que facilita el movimiento de los jugadores u otros elementos en el tablero del juego.

5. **Component LANZAR DADO:**

- Este componente se encarga de la mecánica del lanzamiento de dados.
- Está conectado al **CLIENTE**, lo que sugiere que los jugadores pueden activar esta función directamente desde su interfaz.

6. **CAMBIAR POSICION:**

- Maneja la lógica de cambio de posición de jugadores o piezas en el juego.
- La conexión **ENVIAR COMANDO** entre **CLIENTE** y **CAMBIAR POSICION** indica que las órdenes para mover piezas se envían desde la interfaz del usuario.

Detalle de Conexiones e Interacciones:

- **ABRIR SERVER:** Esta conexión sugiere que el componente de configuración puede activar el servidor, preparándolo para aceptar conexiones entrantes.
- **CONFIGURAR VIDAS:** Permite que los jugadores configuren sus vidas mediante el componente de configuración.
- **BOTON LANZAR DADOS:** Una acción que activa el componente de lanzamiento de dados, permitiendo a los jugadores ejecutar esta acción específica.
- **ENVIAR COMANDO:** Esta conexión es crucial para el movimiento dentro del juego, permitiendo que el cliente envíe comandos de movimiento al componente encargado de cambiar posiciones.
- **CONEXION:** Representa la comunicación entre el cliente y el servidor, esencial para la sincronización en un entorno de juego multijugador.