CREANDO UN AJEDREZ EN LÍNEA

Proyecto de Fin de Ciclo

Javier Ausejo Fernández-Andes

Curso 2018-19/2019-20
IES Comercio
Desarrollo de Aplicaciones Multiplataforma

Tutor: David Pérez

ÍNDICE

1. INTRODUCCIÓN.

Desde un principio, en este proyecto se tiene la pretensión de crear un **ajedrez en red desde cero desarrollado en *Java*** que sea, desde luego, óptimo y agradable para los clientes que hagan uso de esta aplicación. A lo largo de este documento, iremos viendo cómo hemos encarado el proyecto en sus diferentes fases, desde su nacimiento hasta la creación de una primera versión en la que se incluirán diferentes elementos en virtud de las tres premisas u objetivos de trabajo que a continuación voy a explicar.

En primer lugar, como podemos intuir, se tratará de una **aplicación cliente-servidor** y, por tanto, debemos definir la relación que existirá entre estos: el servidor puede comprender muchas partidas simultáneamente, actuando como fuente de energía de estas, y encargándose de emparejar a los diferentes jugadores estableciéndose, además, como nexo en un canal de comunicaciones que permitirá a los clientes de una misma partida estar en contacto entre ellos. En resumen, como premisa básica, tenemos que **el servidor hará de intermediario de todas las partidas que comprenda, encargándose de que haya comunicación entre los clientes de una misma partida**.

En segundo lugar, vamos a definir el concepto de **comunicación entre y para clientes**. Para ello, previamente, hay que contextualizar y hablar de la **interfaz gráfica de usuario**. La GUI contará, principalmente, con cuatro elementos:

1. Un panel que comprenderá los elementos oportunos para que el usuario indique su *nickname* en la partida, así como una etiqueta indicando su estado de conexión respecto al servidor y una serie de botones para conectarse y desconectarse.
2. Un panel que hará de tablero de ajedrez.
3. Un panel de registro de los movimientos de la partida que informa sobre el transcurso de esta.
4. Un panel de chat en la que los jugadores en cuestión podrán entablar comunicación entre ellos.

Una vez definidos los elementos básicos de la interfaz, decimos que la comunicación entre y para clientes, **es toda aquella interacción del usuario con la GUI en la que se precise intervención del servidor**, como lo puede ser la conexión/desconexión del usuario, el envío/recepción de mensajes del chat o el movimiento de figuras del tablero de ajedrez etc.

Por último, **en tercer lugar**, como consecuencia de lo explicado en estos dos puntos, se define que trabajaremos con ***sockets* TCP** enviando y recibiendo objetos de tipo ***ObjectOutputStream* y *ObjectInputStream***. Se elige este tipo de datos porque, a mi juicio y como ya veremos más adelante, **a través de la serialización de objetos se nos permitirá crear clases a medida y hacer que la comunicación entre y para clientes sea más limpia y ordenada**.

2. OBJETIVOS.

Por mediación del tutor, decidimos trabajar este proyecto aplicando la **metodología ágil SCRUM**, cuya lógica responde a una gestión del trabajo en base a una división racional del proyecto en varios sprints. En cada uno de estos sprints, se deben establecer una serie de objetivos lógicos a cumplir en un determinado periodo de tiempo.

En este caso, **dividimos el proyecto en cinco sprints**, dando un margen de dos semanas de trabajo a cada uno de ellos. Una vez finalizado este periodo de tiempo, se concreta con el tutor una reunión en la que se revisa el trabajo llevado a cabo durante estas dos semanas, proponiendo posibles mejoras o cambios de planteamiento con el objetivo final de que el proyecto sea lo más eficiente posible.

Una vez definida la mecánica de trabajo a seguir, podemos explicar los diferentes sprints o fases que atraviesa el desarrollo del proyecto:

- **Sprint 1: Organización del proyecto y documentación de las herramientas de trabajo.**

  Se trata de una fase de estudio en la que se introducen las herramientas de trabajo que vamos a utilizar. La idea de este sprint es **determinar las tecnologías que terminaremos utilizando**, con una respuesta clara que justifique por qué las elegimos, procurando **articular una organización del proyecto lo más acertada posible**.

- **Sprint 2: Diseño de la interfaz de usuario.**

  Si queremos que la aplicación sea funcional, una parte importante es que esta se deje ver y sea atractiva visualmente. Durante esta etapa, se repiensa la GUI en torno a los elementos básicos explicados en la introducción y, finalmente, **tratamos de crear una interfaz intuitiva que haga la aplicación lo más sencilla posible**.

- **Sprint 3: Diseño del sistema de emparejamiento y del chat.**

  Este sprint supone la primera toma de contacto con la lógica interna del programa, más allá de la propuesta estética que se lleva a cabo en el anterior sprint. La premisa básica es **definir el sistema de emparejamiento del servidor**, con la vista puesta en que comprenda **varias partidas simultáneamente**. Para comprobar su correcto funcionamiento, deberemos programar también el **chat**, lo que implica concretar la **estructura de los mensajes** y el envío y recepción de estos por parte del servidor y de los clientes.

- **Sprint 4: Programación de movimientos, del jaque y del jaque mate.**

  Si este proyecto fuera una moneda, una cara de esta tendría que ver con la **comunicación en red que se introduce y desarrolla en el anterior bloque, y la otra sería, como indica el título de este sprint, la programación de movimientos, del jaque y del jaque mate. Se trata del corazón de la aplicación en clave de jugabilidad,** lo que convierte un ajedrez en ajedrez.

- **Sprint 5: Programación de diferentes reglas y recursos del ajedrez.**

  Durante este sprint, se **perfilan diferentes técnicas o recursos recurrentes en el ajedrez** (cronómetro, la posibilidad de pedir empate…), y **se depuran algunos de los métodos ya desarrollados para que cubran más posibilidades de la partida** (como la posibilidad del “rey ahogado”).

3. FASES DEL PROYECTO.

a. Sprint 1: Organización del proyecto y documentación de las herramientas de trabajo.

Tareas a cumplir:

- Estudio de los *sockets* TCP y sus posibilidades.
- Esbozo preliminar de *Mensaje.java*, útil en clave de comunicación entre y para clientes.

Desarrollo:

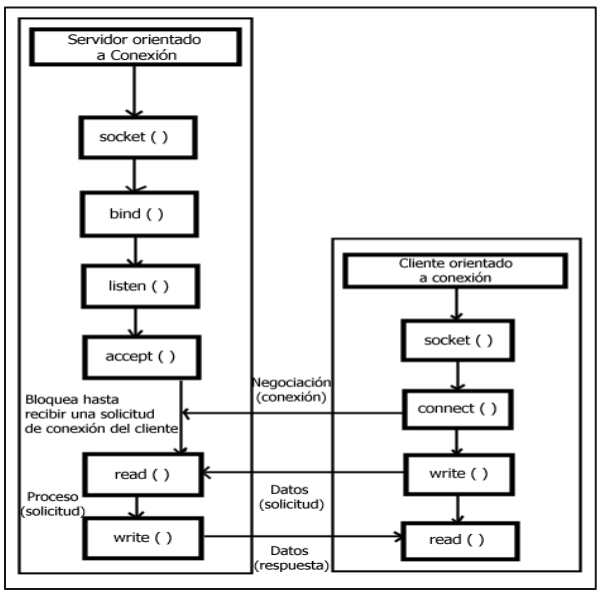
Tal y como ya adelantamos en la introducción, este proyecto se basa en el uso de ***sockets***, que, sabemos, son elementos que actúan como extremo en un proceso de comunicación por red entre diferentes aplicaciones o servicios, permitiendo así que un mensaje pueda circular de un punto a otro de la red.

Como vemos, se trata de un **proceso de comunicación por red**, y, por tanto, concluimos por entender que **todo *socket* debe tener asociado una IP** (dirección en la que se está ejecutando el servidor), **y un puerto** a través del cual la aplicación que se está ejecutando en el servidor puede escuchar e interpretar los mensajes que lleguen, y, en definitiva, comunicarse con los clientes.

Una vez comprendidos los elementos básicos de un socket, determinamos que **trabajaremos sobre un *socket* de tipo TCP**, lo que nos proporciona una **comunicación cliente-servidor bidireccional orientada a la conexión** que garantiza la total entrega de los datos en el orden en que fueron enviados hasta el momento en el que alguno de los dos se desconecte.

Los *sockets* TCP funcionan, resumidamente, de la siguiente manera:

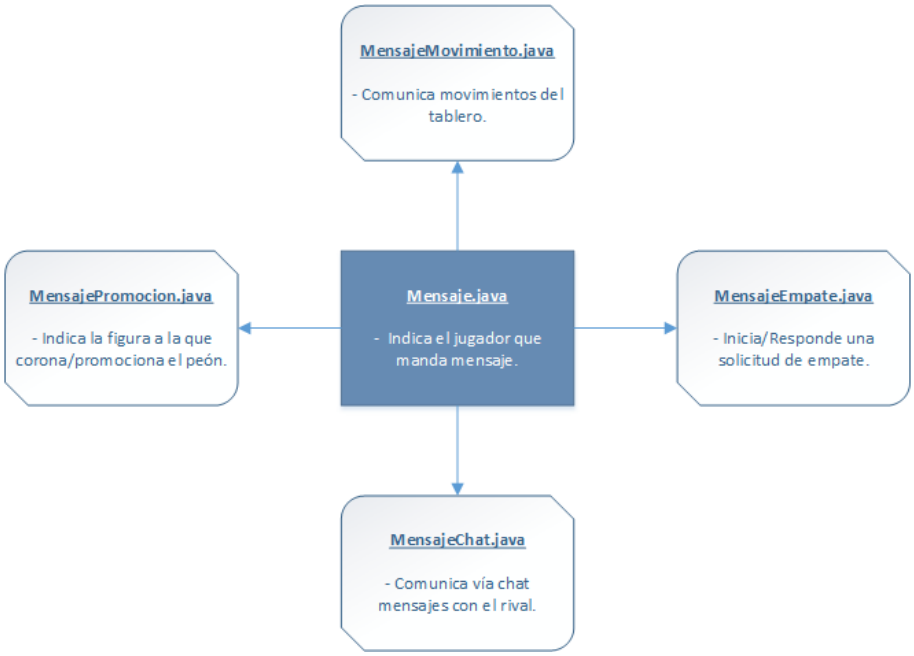
- **Inicio del servidor**, que se encuentra permanentemente escuchando peticiones de clientes.
- **Petición de conexión del cliente** a la dirección IP del servidor a través de su puerto.
- Respuesta de **petición aceptada** por parte del servidor.
- Apertura de un ***stream* de datos** entre cliente y servidor, que permite la comunicación de un flujo secuencial de datos en el que ambos tienen permisos de lectura y escritura hasta que se cierre la conexión.



Contextualizado ya lo que es un *socket* TCP, podemos proceder a introducir **de qué manera se enviarán los datos entre cliente y servidor**. Dado que este *stream* que nace de la conexión del cliente al servidor soporta diferentes tipos de datos (bytes, tipos de datos primitivos etc.), **tenemos la posibilidad de crear una clase Java que instancie objetos *Mensaje.java* hechos a nuestra medida y enviarlos por red gracias a la serialización**, lo que nos facilitaría la interpretación de dichos objetos haciéndola más limpia y ordenada.

Este proceso de serialización de objetos no es más que la conversión de un objeto a bytes, que luego se podrán recuperar y volver a convertir en un objeto de dicha clase. Para ello, debemos hacer uso de las clases ***ObjectOutputStream y ObjectInputStream***, que son las únicas que nos permiten enviar y recibir objetos en sockets TCP a través de los métodos ***readObject()*** y ***writeObject().***

A estas alturas de la práctica, y mirando al horizonte pensando en los diferentes tipos de interacciones que pueden existir en una partida de ajedrez, definimos que la clase ***Mensaje.java*** será una interfaz de la que colgarán las siguientes clases: ***MensajeChat.java***, ***MensajeMovimiento.java***, ***MensajePromocion.java*** y ***MensajeEmpate.java***. Todas implementarán un método heredado para distinguir qué jugador es el que ha enviado el mensaje y, además de eso, la **serialización de objetos** (***Serializable***).



Este *Mensaje.java* se trata de un boceto preliminar y muy primario que nos ayuda a ubicarnos y a entender cómo funcionará la comunicación entre el servidor y el cliente. La idea final es que *Mensaje.java* instancie objetos para, como hemos dicho antes, poder interpretar los mensajes como lo que son de manera adecuada y óptima. Por tanto, dado que utilizaremos la serialización de objetos, en sus correspondientes sprints veremos cómo evoluciona *Mensaje.java* y cómo a cada mensaje le podremos asociar un objeto de tipo ***Jugador.java*** **o** ***Casilla.java*****,** con el **objetivo de trabajar en un sistema de emparejamiento más eficiente que luego permita a los jugadores de una misma partida comunicarse con el rival y mover las figuras**.

b. Sprint 2: Diseño de la interfaz de usuario.

Tareas a cumplir:

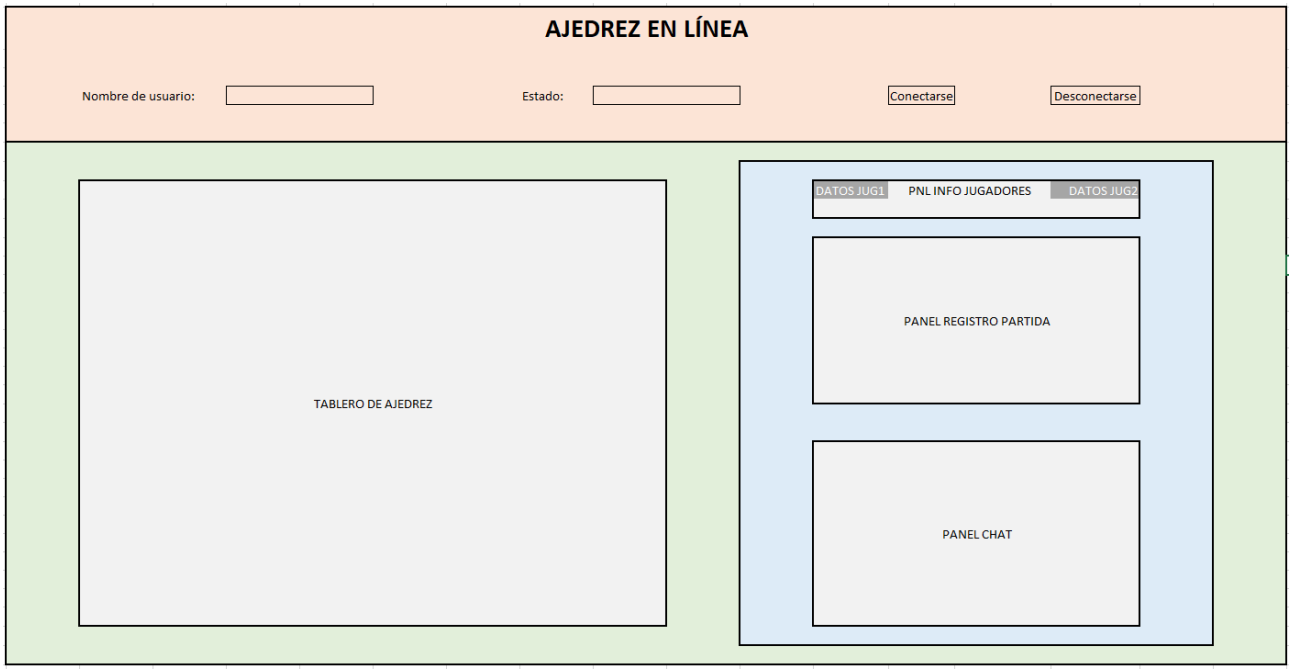
- Diseño prototipo de la GUI de la aplicación.
- Programación preliminar de *Tablero.java* y *Casilla.java.*
- Introducción a la clase *Figura.java*.

Desarrollo del sprint:

La interfaz de usuario es la cara visible de la aplicación y lo que puede determinar, en virtud de lo intuitiva y accesible que sea, si el usuario hace uso de este programa o se busca otro que se adapte mejor a sus necesidades. Por tanto, además de la importancia que supone que la aplicación haga lo que se le presupone que tiene que hacer, no es menos notable que cuente con una GUI **atractiva** y, en medida de lo posible, **autoexplicativa** para que el usuario final no tenga que comerse la cabeza ni perder tiempo pensando en cómo empezar a usar el programa.

Dado que en el plano estético se trata de una aplicación sencilla (solo contará con una "pantalla"), debemos cuidar que la disposición de los elementos quede ordenada y compensada.

¿Con qué elementos contará nuestra aplicación? Como las imágenes tienen más fuerza que las palabras, se presenta una imagen/croquis a modo de resumen con el diseño inicial:



La imagen muestra, principalmente, **dos paneles**: uno color naranja y uno de color verde, que representarían las **zonas de parámetros conexión y de área de juego**, respectivamente, que estarán disponibles (o no) en función de nuestro estado de conexión respecto al servidor.

En el **panel de parámetros de conexión** identificamos:

- Un *JLabel* con "**Nombre de usuario**" y un *JTextField* en el que se indicaría.
- Un *JLabel* con "**Estado**" y, a su derecha, otro que indicaría nuestro **estado de conexión** respecto al servidor (conectado, desconectado, a la espera de partida) con un color de fondo que nos ayudara a identificarlo.
- **Dos *JButtons* para conectarse y conectarse del servidor**.

Por otro lado, en el panel verde, que representa el **área de juego**, encontramos más miga que en el anterior, pues presenta:

- Un **tablero de ajedrez** que nace de *Tablero.java*, del que ahora hablaremos en profundidad.
- Un panel azul, que ocupa la parte derecha del panel del área de juego. Este panel azul, a su vez, se subdivide en tres paneles:
    - Un *JPanel* con **información de los jugadores en la partida**, en el que se mostrarían sus *nicknames*, información acerca de los turnos y el tiempo restante para la ejecución de un movimiento. Incluye, además, un *JButton* para pedir al rival el empate.
    - Un *JPanel* que incluirá un *JList* con las **incidencias de la partida**: movimientos, si el movimiento se trata de jaque o de jaque mate, tiempo de ejecución, si se ha comido una figura etc.
    - Un *JPanel* que funcionará a modo de **chat,** a través del cual los jugadores podrán entablar conversación durante el transcurso de la partida, es decir, una vez hayan sido emparejados por el servidor.

En definitiva, **en el panel de parámetros de conexión encontraríamos un título para la aplicación, así como diferentes elementos para identificarnos y hacer peticiones de conexión/desconexión al servidor**, mientras que **en el panel de área de juego encontraríamos los elementos de juego** necesarios que serán accesibles y que cobran sentido una vez hayamos encontrado rival contra el que jugar.

Una vez explicados superficialmente los conceptos presentes en nuestra interfaz gráfica, trataremos de explicar la **lógica que hay detrás de *Tablero.java***.

Desde un primer momento, pensé en abstraer el tablero del resto del diseño y programación de la GUI porque sabía que iba a ser el corazón de la aplicación y, por tanto, un elemento lo demasiado grande e importante como para no contar con una clase propia.

El tablero es un nuevo *JPanel* que contará con diferentes elementos y, mirando al futuro inmediato de este proyecto, deberá contar con las variables de instancia necesarias para poder enviar mensajes al servidor, identificar si el turno de la partida es el de su jugador o el del rival, conocer si ese jugador juega con figuras negras o blancas etc. Además de esto, naturalmente, el tablero es un conjunto de casillas ordenadas que pueden estar ocupadas por una serie de figuras.

En definitiva, extraigo la conclusión de que es necesario crear dos nuevas clases: ***Casilla.java*** y ***Figura.java*****,** y que el tablero de ajedrez, en este punto de la práctica, se tratará de un *JPanel* que deberá implementar ***ActionListener*** y que estará formado por un array bidimensional de casillas que estarán ocupadas por instancias de la clase *Figura.java*. Ambas clases pueden formar parte de los mensajes a intercambiar con el servidor, por lo que deben implementar la **serialización de objetos**.

Sabemos que un tablero de ajedrez cuenta con diferentes tipos de figuras que no funcionan de la misma manera, por lo que haremos de *Figura.java* una clase abstracta de la que beberán las clases particulares del peón, del alfil, del caballo, del rey, de la reina y de la torre. También sabemos que cada una de estas figuras deberá comprender un método para obtener sus posibles movimientos en virtud del estado del tablero y de la posición que ocupa, es por ello que incluiremos el método abstracto ***getPosiblesMovimientos()*** colgando de *Figura.java*. Este método recibe la posición de la figura a comprobar, el estado del tablero en un determinado momento de la partida y un *booleano* que cambiará levemente el método dependiendo de si se busca el jaque mate.

Las clases *Casilla.java* y *Figura.java* se corresponderán, por el momento, con los siguientes esquemas (incluyendo sus correspondientes *getters* y *setters*):

```java
public class Casilla implements Serializable {

    // VARIABLES DE INSTANCIA
    private JButton btnCasilla;
    private Posicion posicion;
    private Figura figura;

    public Casilla(JButton btnCasilla, Posicion posicion) {
        this.btnCasilla = btnCasilla;
        this.posicion = posicion;
        figura = null;
    }
```

```java
public abstract class Figura implements Serializable {

    // VARIABLES DE INSTANCIA
    private String nom;
    private boolean mia;

    public Figura(String nom, boolean mia) {
        this.nom = nom;
        this.mia = mia;
    }

    /**
     * Método abstracto que devuelve un hashset de instancias Posicion.java con los
     * posibles movimientos que puede hacer una figura en una posición determinada
     * del tablero.
     *
     * @param posicion      la posición actual de la figura
     * @param arrayTablero el estado del tablero
     * @param detectarJaqueMate si hay que detectar jaque mate o no
     * @return hashset de instancias Posicion.java
     */
    public abstract HashSet<Posicion> getPosiblesMovimientos(Posicion posicion,
                                                             Casilla[][] arrayTablero,
                                                             boolean detectarJaqueMate);
```

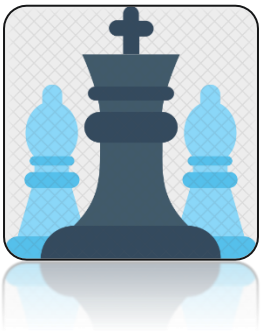
Explicado esto, podemos volver a *Tablero.java* y comprender el método que crea el array bidimensional que lo conforma.

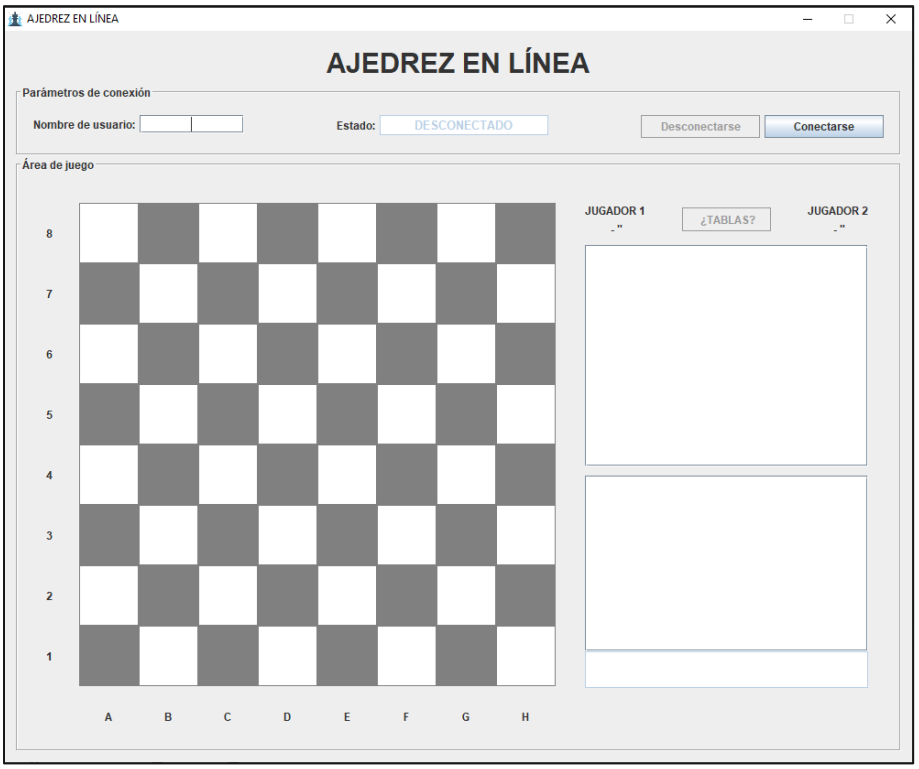
```java
/**
 * Método que crea el tablero y cambia de color las casillas.
 */
private void crearTablero() {
    arrayTablero = new Casilla[DIM_TABLERO][DIM_TABLERO]; // inicializamos el array de casillas del tablero
    String valoresColumna = " ABCDEFGH";
    Color color = Color.WHITE;
    JButton btn;
    Posicion pos;
    for (int fila = DIM_TABLERO; fila > 0; fila--) { // rellenamos tablero de botones y coordenadas numéricas de las filas
        add(new JLabel(Integer.toString(fila), SwingConstants.CENTER));
        for (int col = 0; col < DIM_TABLERO; col++) {
            pos = new Posicion( fila: fila - 1, col);
            // creamos el botón y le damos unos parámetros
            btn = new JButton();
            btn.setPreferredSize(DIM_BOTON_TABLERO);
            btn.setBackground(color);
            // implementamos el listener y le establecemos como actionComand sus propias coordenadas
            // para que luego desde actionPerformed, se pueda detectar la casilla pulsada, y por tanto,
            // determinar si hay figura o no en dicho botón, actuando en consecuencia
            btn.addActionListener( l: this);
            btn.setActionCommand(pos.getFila() + "_" + pos.getColumna());
            // añadimos el btn al tablero y al array
            add(btn);
            arrayTablero[fila - 1][col] = new Casilla(btn, pos); // empezamos a dar forma a las casillas del tablero
            // cambiamos color
            if (color.equals(Color.GRAY) && col != 7) {
                color = Color.WHITE;
            } else if (color.equals(Color.WHITE) && col != 7) {
                color = Color.GRAY;
            }
        }
    }
    for (int i = 0; i < DIM_TABLERO + 1; i++) { // última fila que indica las coordenadas de las columnas del tablero
        add(new JLabel(Character.toString(valoresColumna.charAt(i)), SwingConstants.CENTER));
    }
}
```

Como leemos en el código, creamos un tablero 8x8 en el que los botones tienen un tamaño predeterminado y van alternando el color, dándole la forma de un tablero de ajedrez. Cada vez que creamos una casilla, le añadimos un botón, una posición y una figura ***null***. ¿Por qué una figura *null*? Dado que un jugador al momento de iniciar la aplicación no sabe si va a controlar las figuras blancas o negras, creo que es más adecuado delegar esto al sistema de emparejamiento y añadir las figuras en el momento en el que se encuentre rival.

También es importante señalar que, gracias al ***setActionCommand()***, podremos conocer las coordenadas de la casilla pulsada y actuar en consecuencia.

En la siguiente captura podemos ver el aspecto final que tomará nuestra aplicación basándonos en el boceto preliminar que hemos visto al principio de este sprint. Como vemos, algunos de los elementos de interacción están bloqueados porque, por código, programamos diferentes modos de la aplicación dependiendo del estado de conexión del cliente respecto al servidor. Por último, cabe resaltar también que hemos añadido un icono representativo del proyecto.





c. Sprint 3: Diseño del sistema de emparejamiento y del chat.

Tareas a cumplir:

- Justificar la existencia de *Jugador.java* y *Partida.java*.
- Programación del sistema de emparejamiento.
- Programación del servicio chat de la aplicación.

Desarrollo:

Entendemos que **nuestro ajedrez se tratará de una aplicación multijugador que podrá ejecutar varias partidas de manera simultánea**. Es por ello que, para facilitar la posterior programación del sistema de emparejamiento, creamos dos clases nuevas acordes a esta lógica: ***Jugador.java*** y ***Partida.java***.

Un jugador se conectará al servidor y se definirá como tal cuando realice una petición de conexión/entrada a este y esta sea aceptada. En este punto, en el que la petición se ha convertido en un nuevo jugador, el servidor deberá actuar en consecuencia buscándole una partida en la que haya hueco disponible, o creando una y dejándole a la espera de la conexión de un rival.

Una vez explicado esto, comprendemos que **nuestro sistema de emparejamiento integrará un *array* de instancias de *Partida.java* en la que cada una de ellas deberá tener asignada un par de jugadores/clientes, que se agruparán en torno a un *array* de objetos *Jugador.java***.

(**Partida.java* y *Jugador.java*, cada uno con sus correspondientes *getters* y *setters*, aunque no se vean.)

```java
public class Partida {

    // VARIABLES DE INSTANCIA
    private ArrayList<Jugador> arrayJugadores;

    public Partida(String idPartida) {
        arrayJugadores = new ArrayList<>();
    }

    public void añadirJugador(Jugador jugador) {
        arrayJugadores.add(jugador);
    }

    public void eliminarJugador(Jugador jugador) throws IOException {
        arrayJugadores.remove(jugador);
        jugador.getSocket().close();
    }

    public ArrayList<Jugador> getArrayJugadores() { return arrayJugadores; }

}
```

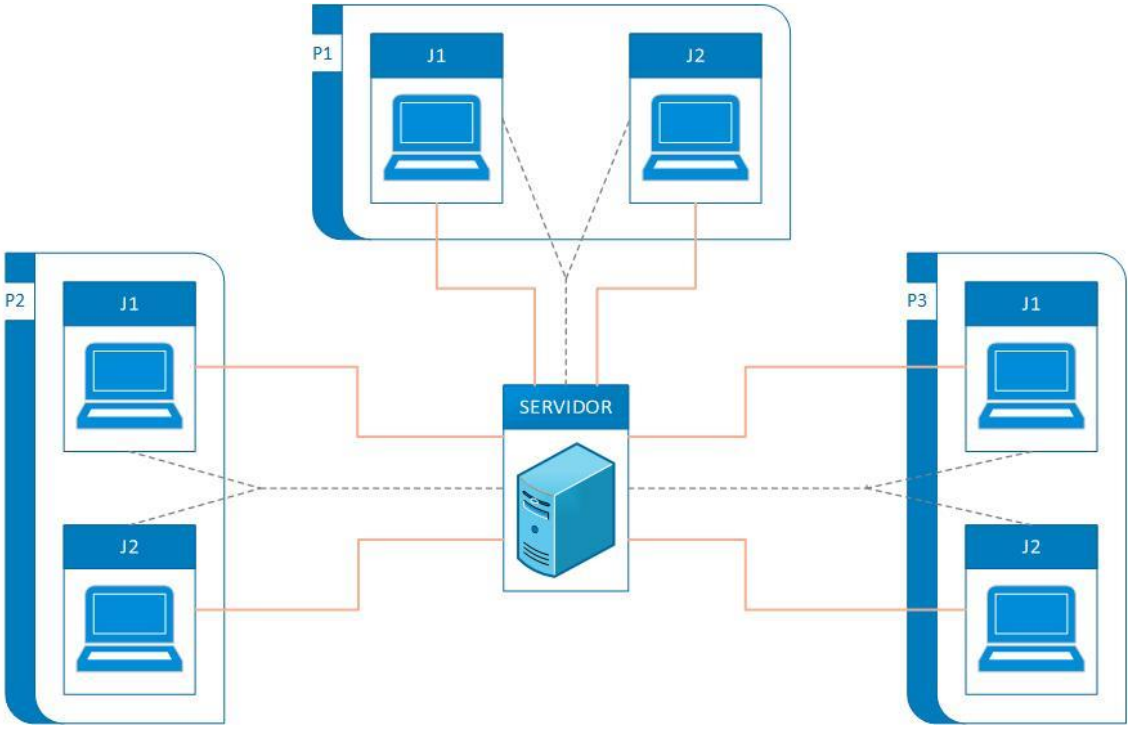
```java
public class Jugador implements Serializable {

    // VARIABLES DE INSTANCIA
    private String idJugador, nickName;
    private boolean local; // si true, fichas blancas; si false, fichas negras
    private transient Socket socket;
    private transient ObjectOutputStream flujoEscritura;
    private transient ObjectInputStream flujoLectura;

    public Jugador(String idJugador, String nickName, Socket socket,
                   ObjectOutputStream flujoEscritura, ObjectInputStream flujoLectura) {
        this.idJugador = idJugador;
        this.nickName = nickName;
        this.socket = socket;
        this.flujoEscritura = flujoEscritura;
        this.flujoLectura = flujoLectura;
    }

    public void cerrarConexion() {
        try {
            flujoEscritura.close();
            flujoLectura.close();
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Como vemos en las imágenes de los códigos, **cada una de las partidas y de los jugadores tendrá un identificador**. El identificador de la partida será único y secuencial a medida que se vayan creando nuevas, mientras que el de los jugadores será único dentro de la partida vinculada que les toque jugar.

Además, vemos que cada instancia de ***Jugador.java* comprende una serie de elementos de red** (*socket, flujoEscritura* y *flujoLectura*), que son necesarios para cerrar la conexión entre el cliente y el servidor, si se dieran las circunstancias. También vemos que estas variables están **precedidas del argumento *transient*.** Esto es porque la serialización de objetos solo nos permite la transmisión de

datos de tipo primitivo; lo que hace *transient* es poner estos valores a *null* en el momento que estos se transmiten por red, evitando así que salte algún tipo de excepción.

Por tanto, a golpe de imagen, tenemos que el sistema de emparejamiento agrupará los jugadores en diferentes partidas de la siguiente manera:



Entrando en la parte más densa del sprint, trataremos de explicar el código que rodea la puesta en marcha del servidor y la conexión de los clientes. Por parte del servidor, señalar que **utilizaremos el puerto 3000 para iniciar la escucha de los clientes/conexiones**.

Procedemos a simular el proceso de conexión:

1. Ejecutamos el *Run.java* del servidor, quedándose a la **espera de conexiones**.

```java
public class Run {
    // CONSTANTES DE CLASE
    private final static int PUERTO = 3000;

    // VARIABLES DE INSTANCIA
    public static ArrayList<Partida> arrayPartidas;

    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(PUERTO);
            arrayPartidas = new ArrayList<>();
            Socket socket;
            Thread t;
            // flujos de conexión del cliente hacia el servidor
            ObjectInputStream flujoLectura;
            ObjectOutputStream flujoEscritura;
            // creamos variables de instancia necesarias para gestionar jugadores
            Jugador jugador = null;
            int idPartidaAux = 0;
            String idJugador, nickName, idPartida;
            // variables booleanas
            boolean control = true; // controla vida del hilo
            boolean encuentraPartida; // controla si hay partida disponible o hay que crear una nueva
            while (control) {
                // damos formato a idJugador, e inicializamos encuentraPartida
                encuentraPartida = false;
                // esperamos conexiones e inicializamos su flujo de lectura para recibir nickname
                socket = serverSocket.accept();
```

2. Iniciamos el *Run.java* del cliente, abriéndose una interfaz de usuario en la que deberemos incluir un nombre de usuario y clicar sobre el botón conectar.

   Se ejecutará el método *run()* de *HiloCliente.java*, clase del cliente que gestiona su conexión respecto al servidor y en la que se controlan las posibles excepciones de error que puedan salir al tratar de conectarse. En este método, **el cliente se conecta al servidor y le envía el *nickname* escrito en la interfaz**.

```
@Override
public void run() {
    try {
        // generamos conexión al servidor e implantamos modoBusqueda en interfaz
        socket = new Socket(DIRECCION, PUERTO);
        interfaz.setModoBusqueda();
        // creamos flujos de lectura y de escritura
        flujoEscritura = new ObjectOutputStream(socket.getOutputStream());
        flujoLectura = new ObjectInputStream(socket.getInputStream());
        // creamos flujo escritura y mandamos nickname al servidor
        flujoEscritura.writeUTF(interfaz.getNickname()); // enviamos nickname
        flujoEscritura.flush();
```

3. El **servidor recibe el *nickname* del cliente en cuestión**, generando previamente el correspondiente flujo de lectura y flujo de escritura de la conexión pertinente.

```
// esperamos conexiones e inicializamos su flujo de lectura para recibir nickname
socket = serverSocket.accept();
flujoEscritura = new ObjectOutputStream(socket.getOutputStream());
flujoLectura = new ObjectInputStream(socket.getInputStream());
nickName = flujoLectura.readUTF();
```

   En este punto comprendemos dos posibilidades: que el servidor esté recién arrancado y por tanto no haya ninguna partida creada a la que vincular al jugador, o bien que ya existan una serie de partidas y haya que bucear en ellas para saber si esta conexión tiene cabida en alguna.

   Si no existe ninguna partida, o en la búsqueda de partidas ya existentes el jugador no tiene cabida porque ya hay 2 clientes en cada una de ellas, se ejecuta esta parte del código en la que se crea una nueva partida y se añade la conexión entrante con el identificador “J1” y como local, es decir, como poseedor de las figuras blancas.

```
if (arrayPartidas.isEmpty() || encuentraPartida == false) { // si no encuentra, creamos una nueva
    // aumentamos secuencialmente idPartidaAux y damos formato a idPartida
    idPartidaAux += 1;
    idPartida = "P" + idPartidaAux;
    // creamos la nueva partida y le añadimos el jugador
    Partida partida = new Partida(idPartida);
    idJugador = "J" + (partida.getArrayJugadores().size() + 1);
    jugador = new Jugador(idJugador, nickName, socket, flujoEscritura, flujoLectura);
    jugador.setLocal(true);
    partida.añadirJugador(jugador);
    // añadimos partida a arrayPartidas
    arrayPartidas.add(partida);
}
```

   Si, por el contrario, se encuentra una partida con un hueco libre para la conexión entrante, se presupone que esta conexión tiene que ser la segunda en la partida y se le asigna el identificador “J2” y las figuras negras.

```
for (Partida p : arrayPartidas) {
    if (p.getArrayJugadores().size() != 2) { // si encontramos partida con hueco
        idJugador = "J" + (p.getArrayJugadores().size() + 1);
        jugador = new Jugador(idJugador, nickName, socket, flujoEscritura, flujoLectura);
        jugador.setLocal(false);
        p.añadirJugador(jugador);
```

A continuación, el camino vuelve a dividirse en dos posibilidades puesto que **puede ser que "J1", en el proceso de espera de conexión de un rival, se haya desconectado del servidor**. Por ello, antes de dar comienzo a la partida, **a cada jugador de la partida se le debe mandar una instancia *Jugador.java* con sus datos representativos para corroborar que las conexiones de ambos clientes siguen en pie**. Si saltara alguna excepción, la variable *booleana encuentraPartida* se pondría a *false*.

```
// si la partida cuenta con 2 jugadores QUE NO HAN CERRADO SUS CONEXIONES, debe empezar
if (p.getArrayJugadores().size() == 2) {
    Jugador jug = null;
    for (int i = 0; i < p.getArrayJugadores().size(); i++) {
        try {
            jug = p.getArrayJugadores().get(i);
            // enviamos a cada conexión su objeto Jugador, aprovechando así para comprobar
            // si el cliente/jugador local todavía no ha cerrado la conexión y sigue disponible
            // para la partida
            jug.getFlujoEscritura().writeObject(jug);
            jug.getFlujoEscritura().flush();
            encuentraPartida = true;
        } catch (SocketException e) {
            // excepción que salta cuando al pasarle el objeto Jugador al cliente
            // este ya ha cerrado la conexión
            encuentraPartida = false;
            p.eliminarJugador(jug); // eliminamos el jugador correspondiente de la partida
            break; // salimos del bucle que recorre jugadores
        }
    }
```

Si la conexión de "J1" siguiera disponible, es decir, si *encuentraPartida* estuviera a *true*, a cada uno de los jugadores de la partida correspondiente se les mandaría por red un *booleano* de valor verdadero para darles a entender que la partida debe comenzar. Si *encuentraPartida* está a *false*, el jugador "J2", que controlaba figuras negras, pasaría a ser "J1" con figuras blancas y se le mandaría un *booleano* a valor falso para darle a entender que su emparejamiento todavía está pendiente de encontrar un rival.

```
//
// se manda un booleano a las conexiones para saber si comienza la partida
// comenzará si no ha saltado el socketException de arriba
if (encuentraPartida) {
    for (Jugador j : p.getArrayJugadores()) {
        j.getFlujoEscritura().writeBoolean( val: true);
        // ARRANCAMOS HILOSERVIDOR PARA CADA JUGADOR
        t = new Thread(new HiloServidor(j, p));
        t.start();
    }
    break; // salimos del bucle en la primera coincidencia de partidas
} else {
    // actualizamos idNickname del jugador y lo seteamos como local
    idJugador = "J" + (p.getArrayJugadores().size());
    jug.setIdJugador(idJugador);
    jugador.setLocal(true);
    // enviamos objeto jugador porque el bucle de HiloCliente así lo requiere.
    jug.getFlujoEscritura().writeObject(jug);
    jug.getFlujoEscritura().flush();
    jug.getFlujoEscritura().writeBoolean( val: false);
}
```

4. El **cliente**, ajeno a la lógica de emparejamiento del servidor, está programado de tal manera que queda **bloqueado en un bucle *while()* hasta que no recibe por red un *booleano* de valor verdadero**, lo que significaría el pistoletazo de salida de la partida habilitando y deshabilitando los elementos de la GUI correspondientes para empezar la partida.

   Como vemos, en *HiloCliente.java* **se recibe una instancia *Jugador.java* que determina nuestro identificador y si jugamos con figuras blancas o negras**, lo que hace que nuestra interfaz cargue a nuestra disposición un color u otro.

```
// creamos flujo lectura y recibimos nuestros parámetros como jugador, así
// como un booleano que indica si comienza la partida
boolean comienzaPartida = false; // hasta que no sea true, la partida no empieza
while (!comienzaPartida) {
    jugador = (Jugador) flujoLectura.readObject();
    comienzaPartida = flujoLectura.readBoolean();
}
interfaz.setModoPartida(jugador.isLocal());
interfaz.setNicknames(jugador, (Jugador) flujoLectura.readObject());
```

5. Una vez vinculados dos jugadores en una misma partida, **por parte del servidor ejecutamos el método *run()* de *HiloServidor.java* para cada uno de los jugadores de la partida**. Se encarga de recibir instancias de *Mensaje.java* de los clientes de una partida para, posteriormente, reenviar dichos objetos a cada uno de los jugadores de la partida.

   En el momento que saltara una excepción en la recepción de los mensajes de red, eliminaríamos la partida desconectando a los clientes oportunos del servidor.

```
@Override
public void run() {
    Mensaje mensaje;
    // enviamos al cliente info sobre su rival para que actualice la interfaz
    for (Jugador j : partida.getArrayJugadores()) {
        if (!j.equals(jugador)) {
            try {
                flujoEscritura.writeObject(j);
            } catch (IOException e) {
                // excepción que se controla en Run.java
            }
        }
    }
    // comenzamos con bucle para escuchar los mensajes del Jugador en cuestión
    while (control) {
        try {
            mensaje = (Mensaje) flujoLectura.readObject();
            enviarMensaje(mensaje);
        } catch (IOException | ClassNotFoundException e) {
            control = false;
        }
    }

    // si salimos del bucle debemos cerrar conexión
    eliminarPartida();
}
```

```
/**
 * Método por el cual el servidor recibe por parámetro un mensaje y lo redirige
 * a los jugadores de la partida, enviándolos a sus correspondientes flujos de escritura.
 */
private void enviarMensaje(Mensaje mensaje) {
    for (Jugador jugador : partida.getArrayJugadores()) {
        try {
            jugador.getFlujoEscritura().writeObject(mensaje);
            jugador.getFlujoEscritura().flush();
        } catch (IOException e) {
            eliminarPartida();
        }
    }
}
```

6. Finalmente, **por parte del cliente la conexión queda atrapada en un bucle *while()* mientras el *socket* esté disponible**. Para poder identificar si el mensaje que se recibe del servidor lo hemos mandado nosotros o lo ha mandado el rival, nos apoyamos en el identificador que asignamos a cada uno de los jugadores de la partida.

```
while (!socket.isClosed()) {
    try {
        mensaje = (Mensaje) flujoLectura.readObject();
        // determinamos si el mensaje lo ha enviado el mismo cliente o el rival
        if (mensaje.getJugador().getIdJugador().equals(jugador.getIdJugador())) {
            interfaz.procesarMensaje(mensaje,  enviadoPorMi: true);
        } else {
            interfaz.procesarMensaje(mensaje,  enviadoPorMi: false);
        }
    }  catch (SocketException e) {
        // caso desconexión del cliente hacie el servidor
        desconexion();
        JOptionPane.showMessageDialog(interfaz,
                message: "Se ha desconectado con éxito.",
                title: "Información",
                JOptionPane.INFORMATION_MESSAGE);
    } catch (EOFException ex) {
        // caso pérdida de conexión con servidor
        desconexion(); // salimos del bucle
        JOptionPane.showMessageDialog(interfaz,
                message: "Se ha perdido la conexión con la partida.",
                title: "Error",
                JOptionPane.ERROR_MESSAGE);
    }
}
// si salimos del bucle cerramos conexiones y habilitamos modoInicio en interfaz
desconexion();
```

En este punto del documento, una vez hemos simulado el arranque del servidor y su gestión de las conexiones entrantes hasta el punto de emparejarlos, debemos proporcionar a cada cliente las herramientas necesarias para poder generar instancias de *Mensaje.java* y enviarlas al servidor. A continuación, **nos centramos en el servicio chat de la aplicación**.

Si recordamos la GUI, nos acordaremos de que hay un *JTextField* dedicado a que los jugadores escriban los mensajes que quieren enviar por chat a sus rivales. Es por ello que, cada vez que pulsemos la tecla *Enter* sobre este *JTextField* invocaremos al método ***enviarMensajeChat()*** que hemos definido en la clase *HiloCliente.java* para enviar instancias de *MensajeChat.java*.

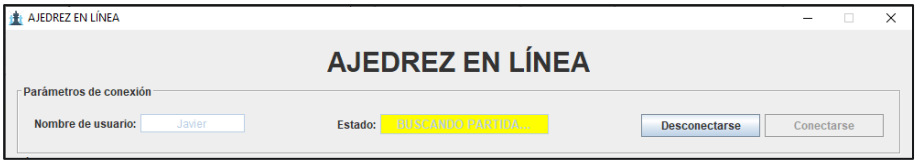
En las siguientes imágenes enseñamos ***MensajeChat.java***, que hereda de *Mensaje.java* ***getJugador()*** e **implementa los *getters* y *setters* para reconocer la cadena enviada en el mensaje de chat en la variable *txt*,** además del método ***enviarMensajeChat()*** de *HiloCliente.java*.

```
public class MensajeChat implements Mensaje, Serializable {

    // VARIABLES DE INSTANCIA
    private Jugador jugador; // jugador que lo envía
    private String txt; // cadena de texto a enviar

    public MensajeChat(Jugador jugador, String txt) {
        this.jugador = jugador;
        this.txt = txt;
    }
```

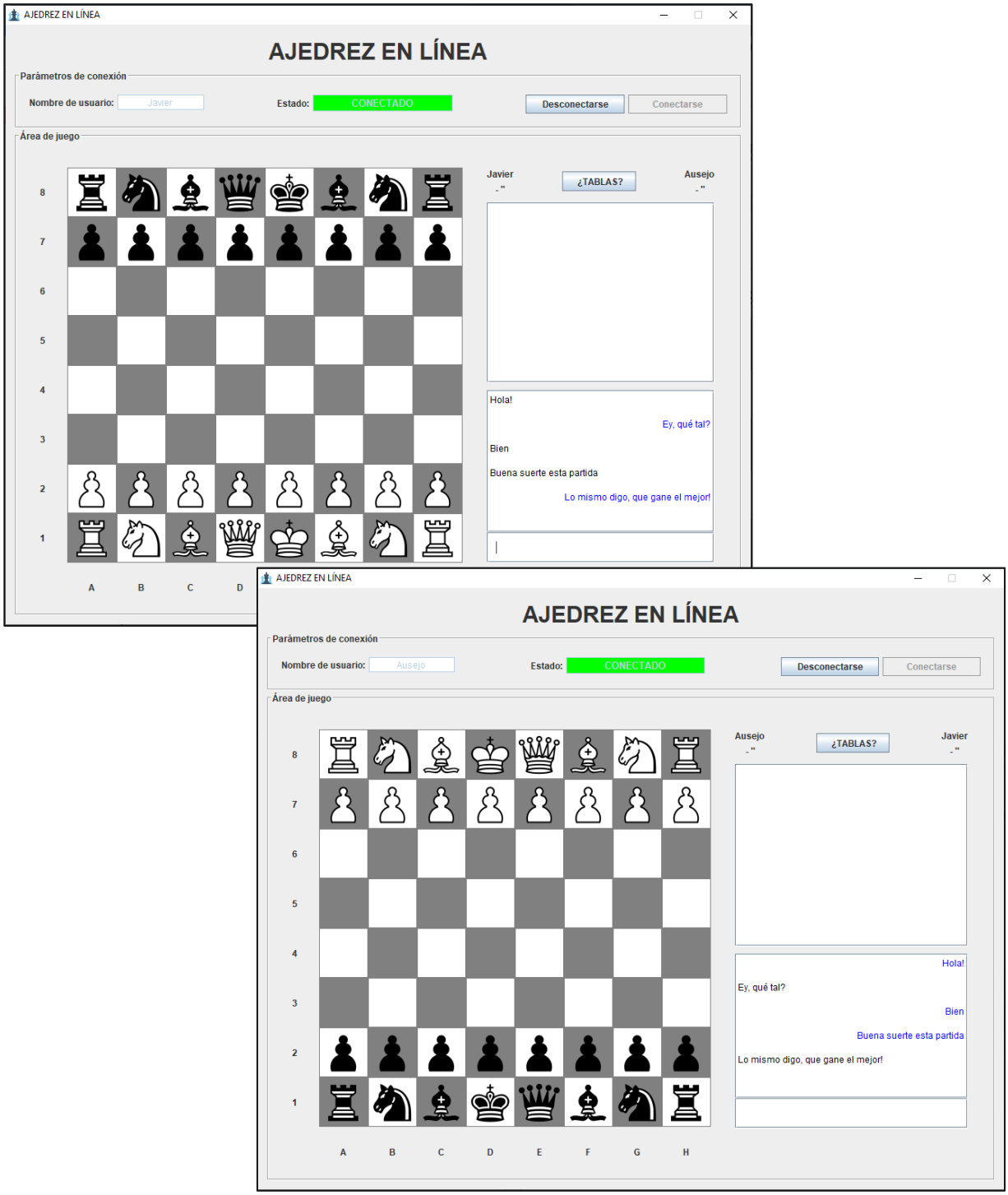
```
public void enviarMensajeChat(String txt) {
    try {
        Mensaje mensaje = new MensajeChat(jugador, txt);
        flujoEscritura.writeObject(mensaje);
        flujoEscritura.flush();
    } catch (IOException e) {
        desconexion();
    }
}
```

Para acabar con este sprint, observamos el aspecto final que debe tomar nuestra interfaz durante las diferentes fases del emparejamiento, así como una vez cargadas las figuras en el tablero y habiendo establecido la comunicación vía chat entre los diferentes jugadores:

*VISTA JUGADOR ESPERANDO RIVAL:



*VISTA JUGADORES EMPAREJADOS INTERPRETANDO MENSAJES DE CHAT:



d. Sprint 4: Programación de movimientos, del jaque y del jaque mate.

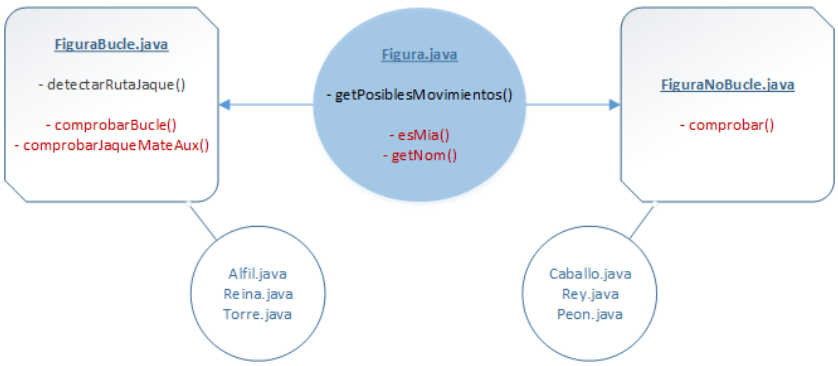
Tareas a cumplir:

- Evolución de *Figura.java* y programación de los movimientos.
- Programación de *MensajeMovimiento.java*.
- Programación del jaque y del jaque mate.

Desarrollo:

A la hora de diseñar la programación de las **figuras**, nos encontramos que en un tablero de ajedrez podemos "diferenciar" **dos tipos** de estas: **aquellas cuya lógica de movimiento responde a bucles (alfil, reina y torre), y aquellas que no (caballo, peón y rey).** Es por ello, que he decidido que de *Figura.java* cuelguen dos nuevas clases abstractas, ***FiguraBucle.java*** y ***FiguraNoBucle.java***, en vistas a la posibilidad de implementar una **programación de los movimientos de las figuras en función de su posición y del estado del tablero a lo largo del transcurso de la partida**. Además, a la hora de comprobar el jaque mate, también influye qué tipo de pieza sea pues, por ejemplo, un caballo que haga jaque puede ver frustrado el jaque mate si se lo pueden comen, mientras que para una reina que ponga en jaque al rival, el jaque mate puede verse frustrado tanto si se la comen como si taponan/bloquean determinado movimiento de la reina. Por lo tanto, de esta circunstancia, nace el método ***detectarRutaJaque()***.

(*Métodos abstractos en negro; método propio de la clase en rojo.)



Como acabamos de explicar, la valoración de los posibles movimientos de las figuras se hace en función de la casilla en la que se encuentren y del estado correspondiente del tablero a lo largo de la partida. Por ello, **es necesario programar un método de comprobación** que reciba estos parámetros en el constructor y nos devuelva un *HashSet* de tipo *Posicion.java* con los posibles movimientos que pueden hacer las figuras sobre las que hemos clicado. Es decir, que el método de comprobación se trataría de un método auxiliar de ***getPosiblesMovimientos()*** que serviría para depurar si dichos movimientos detectados se pueden ejecutar o no comprobando si las casillas ya están ocupadas, y si es así si por figuras nuestras o del rival. Además, la comprobación heredaría de ***getPosiblesMovimientos()*** la variable *booleana* de detección del jaque mate, modificando su lógica levemente en función de si hay que comprobarlo o no.

La implementación del método de comprobación difiere según si se trata de una *FiguraBucle.java* o de una *FiguraNoBucle.java*. ¿Por qué? Porque para las *FiguraNoBucle.java* también pasaremos una variable *booleana* por constructor que representará la dirección hacia la que pretenden moverse las figuras (norte, sur, este, oeste…), haciendo que parte del código particular de los alfiles, reinas o torres se ejecute o no en virtud de si devuelve verdadero o falso.

Vamos a explicar, en primer lugar, cómo comprobaríamos/depuraríamos los movimientos para *FigurasNoBucle.java*. Pongamos el caso de ***Peon.java***.

El peón es una figura particular, puesto que respecto al resto de *FiguraNoBucle.java*, su lógica de movimiento es diferente: únicamente puede desplazarse hacia una casilla norte (o dos, en función de si es su primer movimiento) si está/n vacía/s, y sólo come una casilla en dirección noreste o noroeste, por lo que implementa un método particular que es ***comprobarComer()***.

```java
public class Peon extends FiguraNoBucle {

    // VARIABLES DE INSTANCIA
    private HashSet<Posicion> hsPosiblesMovimientos;
    private boolean primerMovimiento;

    public Peon(boolean local) {
        super( nom: "PEÓN", local);
        primerMovimiento = true;
    }
```

Como muestran las siguientes imágenes, **el método para detectar los movimientos consiste, en esta clase de figuras, de una serie de operaciones manuales que calculan un objeto *Posicion.java*** el cual pasan al método ***comprobar()*** junto a otra serie de parámetros (el *HashSet* de potenciales movimientos, el estado del tablero y la variable *booleana* de detección de jaque mate). También debemos subrayar que los únicos movimientos que se añaden al *HashSet* resultante en caso de buscar el jaque mate, son solo aquellos en los que puede comer.

```java
@Override
public HashSet<Posicion> getPosiblesMovimientos(Posicion posicion, Casilla[][] arrayTablero, boolean detectarJaqueMate) {
    hsPosiblesMovimientos = new HashSet<Posicion>();
    int fila = posicion.getFila();
    int columna = posicion.getColumna();
    int nFila, nColumna;
    Posicion p;
```

```java
    if (!detectarJaqueMate) {
        // una fila NORTE
        if (esMia()) {
            nFila = fila + 1;
        } else {
            nFila = fila - 1;
        }
        nColumna = columna;
        p = new Posicion(nFila, nColumna);
        comprobar(hsPosiblesMovimientos, arrayTablero, p, detectarJaqueMate: false);

        // si se trata de primer movimiento
        if (esPrimerMovimiento() && !hsPosiblesMovimientos.isEmpty()) {
            // dos filas NORTE
            if (esMia()) {
                nFila = fila + 2;
            } else {
                nFila = fila - 2;
            }
            nColumna = columna;
            p = new Posicion(nFila, nColumna);
            comprobar(hsPosiblesMovimientos, arrayTablero, p, detectarJaqueMate: false);
        }
    }
```

```java
                comprobar(hsPosiblesMovimientos, arrayTablero,
            }
        }

        // comprobar si puede COMER de modo tradicional
        if (esMia()) {
            nFila = fila + 1;
        } else {
            nFila = fila - 1;
        }
        nColumna = columna - 1;
        p = new Posicion(nFila, nColumna);
        comprobarComer(p, arrayTablero, detectarJaqueMate);
        nColumna = columna + 1;
        p = new Posicion(nFila, nColumna);
        comprobarComer(p, arrayTablero, detectarJaqueMate);

        return hsPosiblesMovimientos;
    }
```

Aquí podemos consultar cómo es el método auxiliar ***comprobarComer()***, que, insisto, es único y propio de los peones.

```java
private void comprobarComer(Posicion posicion, Casilla[][] array, boolean detectarJaqueMate) {
    int fila = posicion.getFila(), columna = posicion.getColumna();
    if (fila >= 0 && fila < 8 && columna >= 0 && columna < 8) {
        // detectamos si en dicha posición hay una figura y si es nuestra o no
        if (array[fila][columna].getFigura() != null) {
            if (array[fila][columna].getFigura().esMia() != esMia()) {
                hsPosiblesMovimientos.add(posicion);
            } else {
                if (detectarJaqueMate) {
                    hsPosiblesMovimientos.add(posicion);
                }
            }
        } else {
            if (detectarJaqueMate) {
                hsPosiblesMovimientos.add(posicion);
            }
        }
    }
}
```

En la siguiente imagen mostramos el **método *comprobar()* de *FiguraNoBucle*.java**. Como el método para comprobar si come o no del peón ya lo implementa este mismo, hacemos una distinción entre esta figura y las demás de este grupo.

Esta comprobación, básicamente, depura si las posiciones que recibe como movimientos respetan las dimensiones del tablero y, en caso de que así sea, comprueba si esa posición está ocupada o no. Si está ocupada por el rival, procede a añadirse a los posibles movimientos puesto que puede comerse; si está ocupada por nosotros, se añade o no en función del *booleano detectarJaqueMate*, ya que esa posición podría ser uno de los potenciales movimientos del rey rival si este se encontrara en jaque, pero eso es algo que explicaremos después.

```
public void comprobar(HashSet<Posicion> hsPosiblesMovimientos,
                      Casilla[][] array, Posicion posicion, boolean detectarJaqueMate) {
    // detectamos si la fila y la columna enviadas entran dentro del tablero
    int fila = posicion.getFila();
    int columna = posicion.getColumna();
    if (fila >= 0 && fila < 8 && columna >= 0 && columna < 8) {
        if (this instanceof Peon) { // si se trata de un peón, el comportamiento varía
            if (array[fila][columna].getFigura() == null) {
                hsPosiblesMovimientos.add(posicion);
            }
        } else {
            if (array[fila][columna].getFigura() != null) {
                if (array[fila][columna].getFigura().esMia() != esMia()) {
                    hsPosiblesMovimientos.add(posicion);
                } else {
                    if (detectarJaqueMate) {
                        hsPosiblesMovimientos.add(posicion);
                    }
                }
            } else {
                hsPosiblesMovimientos.add(posicion);
            }
        }
    }
}
```

A continuación, pasamos al **método de comprobación para *getPosiblesMovimientos()* en *FigurasBucle.java*** basándonos en el ejemplo más sencillo: ***Torre.java***.

La torre es una figura que puede moverse tanto en sentido horizontal, como vertical, hasta toparse con la casilla límite del tablero, o con una casilla ocupada. Si la casilla está ocupada por una figura rival, podrá comérsela; en caso contrario, se conformará con establecer su límite en la casilla previa a la de su figura compañera. Considerando también que en el siguiente sprint vamos a implementar los movimientos de enroque, en los que interfieren rey y torre, añadimos la **variable *booleana primerMovimiento*** puesto que nos será útil.

```
public class Torre extends FiguraBucle {

    // VARIABLES DE INSTANCIA
    private HashSet<Posicion> hsPosiblesMovimientos;
    private boolean primerMovimiento;

    public Torre(boolean local) {
        super( nom: "TORRE", local);
        primerMovimiento = true;
    }
```

Como hemos dejado caer antes, **para estas figuras pasaremos una variable *booleana* para representar las diferentes coordenadas hacia las que puede moverse la figura**. En el caso de la

torre son norte, sur, oeste y este, por lo que incorporará en su ***getPosiblesMovimientos()*** sus correspondientes variables *booleanas* inicializadas a true.

Una vez se establecen estas condiciones, como vemos en el método, se inicia un bucle *for* en el que, dependiendo del valor de las variables *booleanas* antes citadas, se entrará a generar una nueva posición que, posteriormente, se depurará con los métodos verificadores de posiciones *comprobarBucle()* y *comprobarJaqueMateAux()*.

```java
@Override
public HashSet<Posicion> getPosiblesMovimientos(Posicion posicion, Casilla[][] arrayTablero, boolean detectarJaqueMate) {
    hsPosiblesMovimientos = new HashSet<Posicion>();
    int fila = posicion.getFila();
    int columna = posicion.getColumna();
    int fAux, cAux;
    Figura figura;
    Posicion p;
    // variables para calcular movimientos
    boolean N = true;
    boolean S = true;
    boolean O = true;
    boolean E = true;
```

```java
boolean E = true;
for (int i = 1; i < 8; i++) {
    // dirección norte
    if (N) {
        fAux = fila + i;
        cAux = columna;
        p = new Posicion(fAux, cAux);
        N = comprobarBucle(hsPosiblesMovimientos, arrayTablero, p, N, detectarJaqueMate);
        // determinamos que se respetan las dimensiones del tablero
        if (!N && detectarJaqueMate && fAux < 7) {
            figura = arrayTablero[fAux][cAux].getFigura();
            fAux += 1;
            p = new Posicion(fAux, cAux);
            comprobarJaqueMateAux(figura, p, arrayTablero, hsPosiblesMovimientos);
        }
    }
    if (S) {
        // dirección sur
        fAux = fila - i;
        cAux = columna;
        p = new Posicion(fAux, cAux);
        S = comprobarBucle(hsPosiblesMovimientos, arrayTablero, p, S, detectarJaqueMate);
        // determinamos que se respetan las dimensiones del tablero
        if (!S && detectarJaqueMate && fAux > 0) {
            figura = arrayTablero[fAux][cAux].getFigura();
            fAux -= 1;
            p = new Posicion(fAux, cAux);
            comprobarJaqueMateAux(figura, p, arrayTablero, hsPosiblesMovimientos);
        }
    }
    // dirección oeste
```

```java
    // dirección oeste
    if (O) {
        fAux = fila;
        cAux = columna - i;
        p = new Posicion(fAux, cAux);
        O = comprobarBucle(hsPosiblesMovimientos, arrayTablero, p, O, detectarJaqueMate);
        // determinamos que se respetan las dimensiones del tablero
        if (!O && detectarJaqueMate && cAux > 0) {
            figura = arrayTablero[fAux][cAux].getFigura();
            cAux -= 1;
            p = new Posicion(fAux, cAux);
            comprobarJaqueMateAux(figura, p, arrayTablero, hsPosiblesMovimientos);
        }
    }
    if (E) {
        // dirección este
        fAux = fila;
        cAux = columna + i;
        p = new Posicion(fAux, cAux);
        E = comprobarBucle(hsPosiblesMovimientos, arrayTablero, p, E, detectarJaqueMate);
        // determinamos que se respetan las dimensiones del tablero
        if (!E && detectarJaqueMate && cAux < 7) {
            figura = arrayTablero[fAux][cAux].getFigura();
            cAux += 1;
            p = new Posicion(fAux, cAux);
            comprobarJaqueMateAux(figura, p, arrayTablero, hsPosiblesMovimientos);
        }
    }
}

return hsPosiblesMovimientos;
}
```

Ya entendido el esquema de ***getPosiblesMovimientos()*** de la torre, entramos a ver cómo es el método ***comprobarBucle()*** que valora si las posiciones que se le pasan pueden agregarse al *HashSet* de posibles movimientos de las *FigurasBucle.java*. Como vemos en el código anterior, además, es un **método al que se le pasa un *booleano* para saber si se busca el jaque mate**.

```java
public boolean comprobarBucle(HashSet<Posicion> hsPosiblesMovimientos, Casilla[][] array, Posicion posicion,
                              boolean direccion, boolean detectarJaqueMate) {
    // detectamos si la fila y la columna enviadas entran dentro del tablero
    int fila = posicion.getFila();
    int columna = posicion.getColumna();
    if (fila >= 0 && fila < 8 && columna >= 0 && columna < 8) {
        // casilla sin ocupar
        if (array[fila][columna].getFigura() == null) {
            hsPosiblesMovimientos.add(posicion);
        } else if (!array[fila][columna].getFigura().esMia() == esMia()) { // casilla ocupada por figura del rival
            hsPosiblesMovimientos.add(posicion);
            direccion = false;
        } else if (array[fila][columna].getFigura().esMia() == esMia()) { // casilla ocupada por nosotros
            if (detectarJaqueMate) {
                hsPosiblesMovimientos.add(posicion);
            }
            direccion = false;
        }
    } else {
        direccion = false;
    }
    return direccion;
}
```

Como vemos, para *FiguraBucle.java* el método de comprobación devuelve un valor *booleano* que será el que adopten las coordenadas de movimiento de las figuras. De la misma manera que para *FiguraNoBucle.java*, se comprueba que la posición pasada cumple con los requisitos de dimensión del tablero y si la posición está ocupada ya o no por una figura. En caso de que no, dicha posición se añade al *HashSet*; en caso contrario, distinguimos dos posibilidades: la primera, que la figura que ocupa esa posición sea una figura rival, añadiéndose la posición al *HashSet* y poniendo la variable *booleana* a devolver a *false* (significando que ya no se puede avanzar más en esa coordenada); y la segunda, que la figura que ocupe esa posición sea nuestra. Para esta segunda condición ponemos sí o sí la variable *booleana* a devolver a *false*, y es donde entra a actuar la variable *booleana detectarJaqueMate*: en caso de que esté a true, añadimos la posición al *HashSet*; en caso de *false*, no.

```
public boolean comprobarBucle(HashSet<Posicion> hsPosiblesMovimientos,
                              Casilla[][] array, Posicion posicion,
                              boolean direccion, boolean detectarJaqueMate) {
    // detectamos si la fila y la columna enviadas entran dentro del tablero
    int fila = posicion.getFila();
    int columna = posicion.getColumna();
    if (fila >= 0 && fila < 8 && columna >= 0 && columna < 8) {
        // casilla sin ocupar
        if (array[fila][columna].getFigura() == null) {
            hsPosiblesMovimientos.add(posicion);
        } else if (array[fila][columna].getFigura().esMia() != esMia()) { // casilla ocupada por figura del rival
            hsPosiblesMovimientos.add(posicion);
            direccion = false;
        } else if (array[fila][columna].getFigura().esMia() == esMia()) { // casilla ocupada por nosotros
            if (detectarJaqueMate) {
                hsPosiblesMovimientos.add(posicion);
            }
            direccion = false;
        }
    } else {
        direccion = false;
    }
    return direccion;
}
```

Si volvemos al *getPosiblesMovimientos()* de la torre, vemos que cuando una coordenada se pone a *false*, si se dan las condiciones de respeto a las dimensiones del tablero y *detectarJaqueMate* está a true, deberemos darle una vuelta de tuerca más a los posibles movimientos en esa dirección con *comprobarJaqueMateAux()*. Esta lógica de método, que no el mismo método, se repite para todas las *FiguraBucle.java.*

En *comprobarJaqueMateAux()* comprobamos si en la última posición válida respecto a esta coordenada se encuentra un rey de color diferente a la pieza cuyos movimientos estamos comprobando. De ser así, habrá que comprobar si la instancia de *Posicion.java* que le hemos pasado por parámetro representa una casilla sin figura o del mismo color de la pieza en cuestión para así valorar si añadirla al *HashSet* de posibles movimientos.

```
public void comprobarJaqueMateAux(Figura figura, Posicion posicion,
                                  Casilla[][] arrayTablero,
                                  HashSet<Posicion> hsPosiblesMovimientos) {
    // comprobamos que la figura sea el rey que tiene que ser
    if (figura != null && figura instanceof Rey && figura.esMia() != esMia()) {
        int fila = posicion.getFila();
        int columna = posicion.getColumna();
        figura = arrayTablero[fila][columna].getFigura(); // figura de la posición que recibimos por parámetro
        if (figura == null || figura.esMia()) { // la añadimos si es null o nuestra
            hsPosiblesMovimientos.add(posicion);
        }
    }
}
```

Una vez programados los movimientos de las figuras, debemos pensar en **cómo transmitirle al servidor que hemos movido determinada figura a determinada posición, y si hemos comido o no.**

Para ello, generamos, a partir de la ya mencionada interfaz *Mensaje.java*, **una nueva clase llamada *MensajeMovimiento.java*** que se encargue de crear instancias de este tipo y enviarlas al servidor.

La clase sería la siguiente, con sus respectivos *getters* y *setters*:

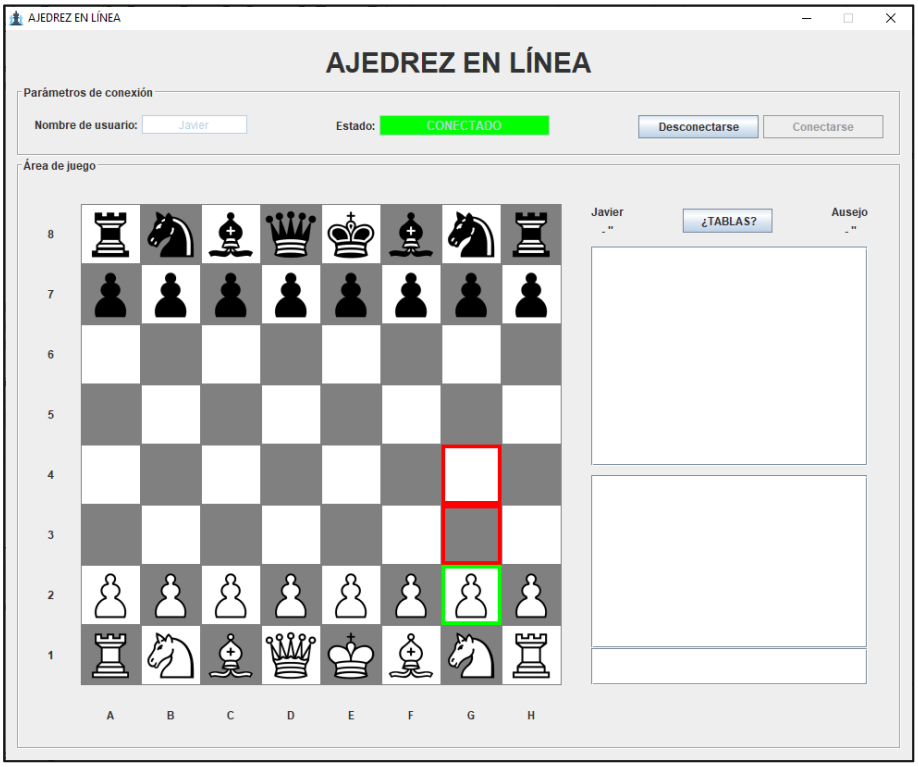
```java
public class MensajeMovimiento implements Mensaje, Serializable {

    // VARIABLES DE INSTANCIA
    private Jugador jugador; // jugador que lo envía
    private Casilla casillaOrigen;
    private Casilla casillaDestino;
    private boolean seCome;

    public MensajeMovimiento(Jugador jugador, Casilla casillaOrigen, Casilla casillaDestino, boolean seCome) {
        this.jugador = jugador;
        this.casillaOrigen = casillaOrigen;
        this.casillaDestino = casillaDestino;
        this.seCome = seCome;
    }
```

Por tanto, desde ***Tablero.java***, cada vez que sea nuestro turno y decidamos mover nuestras figuras, se invocará al método ***enviarMovimiento()*** **de** ***HiloCliente.java*** para que genere una instancia de *MensajeMovimiento.java* y la mande al servidor.

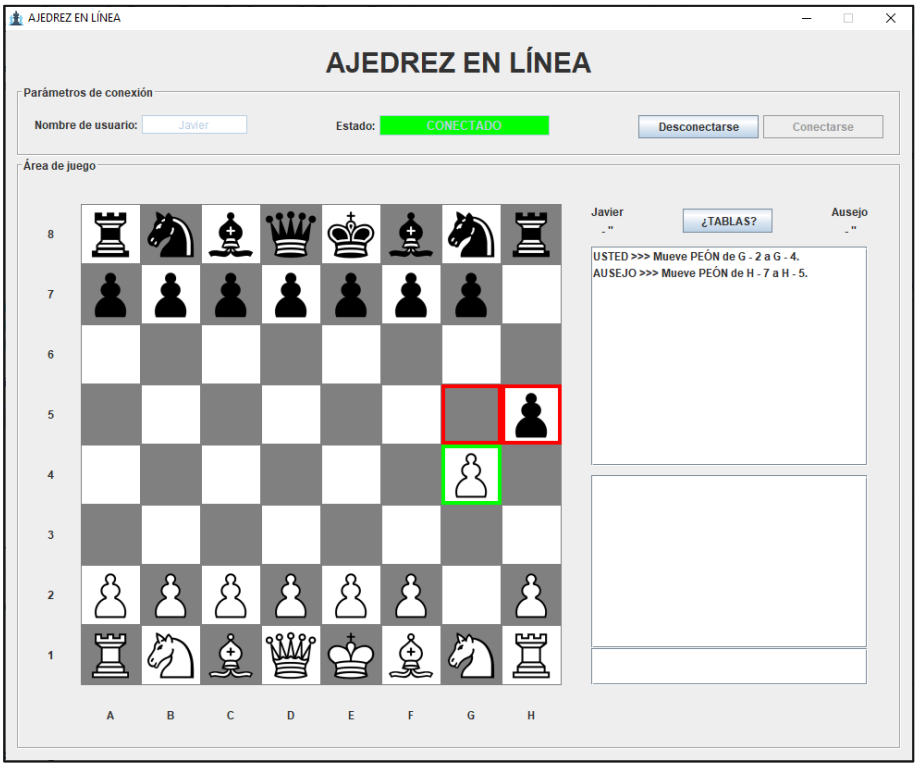
```java
public void enviarMovimiento(Casilla casillaOrigen, Casilla casillaDestino, boolean seCome) {
    try {
        Mensaje mensaje = new MensajeMovimiento(jugador, casillaOrigen, casillaDestino, seCome);
        flujoEscritura.writeObject(mensaje);
        flujoEscritura.flush();
        flujoEscritura.reset(); // reseteamos flujo de escritura a estado inicial, de lo contrario no transmite bien la figura
    } catch (IOException e) {
        desconexion();
    }
}
```

Una vez *HiloCliente.java* detecta el mensaje y lo envía a la interfaz, esta lo procesa y los refleja en el *JList* de incidencias de la partida. En las siguientes capturas, mostramos el resultado final de lo llevado a cabo en este sprint.

*VISTA PEÓN PRIMER MOVIMIENTO:



*VISTA PEÓN NO PRIMER MOVIMIENTO, ADEMÁS EN EL QUE PUEDE COMER:



*VISTA TORRE:

Ahora que hemos programado *getPosiblesMovimientos()* para cada una de las figuras e implementado cómo *Tablero.java* se comunica con el servidor y cómo *Interfaz.java* interpreta los *MensajeMovimiento.java*, la programación del jaque se torna en algo sencillo: simplemente deberemos ejecutar de nuevo *getPosiblesMovimientos()* para la figura que se acaba de mover y comprobar si entre las posiciones incluidas en el *HashSet* encontramos el rey rival antes de proceder a cambiar de turno y dárselo al otro jugador.

Con vistas a la programación de la detección del jaque mate, estableceremos este ***comprobarJaque()*** como un método de *Tablero.java* que, o bien nos devuelva la posición del rey rival, o nos devuelva un valor *null*. El método sería el siguiente:

```
public Posicion comprobarJaque(Casilla casilla) {
    Figura figura = casilla.getFigura();
    Figura figuraAux;
    int fila, columna;
    Posicion pos = null;
    // limpiamos hsPosiblesMovimientos y detectamos los posibles movimientos de la figura indicada
    hsPosiblesMovimientos.clear();
    hsPosiblesMovimientos = figura.getPosiblesMovimientos(casilla.getPosicion(), arrayTablero, detectarJaqueMate: false);
    // recorremos hsPosiblesMovimientos y comprobamos si hay posibilidad de comerse al rey rival.
    for (Posicion posicion : hsPosiblesMovimientos) {
        fila = posicion.getFila();
        columna = posicion.getColumna();
        figuraAux = arrayTablero[fila][columna].getFigura();
        if (figuraAux instanceof Rey) {
            pos = posicion;
            break; // salimos del bucle
        }
    }

    return pos;
}
```

Como hemos subrayado antes, la comprobación del jaque la haremos después de que en nuestro tablero se mueva una figura (independientemente de si es nuestra o no) y justo antes del cambio de turno.

Si la predicción de jaque es cierta, debemos comprobar si existe el jaque mate, puesto que como dice la Wikipedia "**Jaque mate es una posición del ajedrez en la que el rey se encuentra amenazado (en jaque) y esta situación no puede cambiarse mediante ninguna jugada legal**.". Es decir, que dicha comprobación es obligatoria **SIEMPRE** después de cada jaque.

Por tanto, entendemos que nuestra *Interfaz.java* tiene que terminar de interpretar los mensajes de movimientos de una manera acorde a lo siguiente: **si después de mover la figura correspondiente indicada en la última instancia de *MensajeMovimiento.java* el método *comprobarJaque()* nos da un valor de retorno diferente a *null*, debemos entrar a comprobar si se da jaque mate invocando al método *comprobarFinPartida()*.** Este método, veremos en el siguiente sprint cómo nos ayudará también a la hora de detectar un empate por ahogado.

```
private void comprobarMovimiento(String jug, Posicion posicion) {
    // comprobamos si hay jaque
    Posicion jaque = pnlTablero.comprobarJaque(arrayTablero[posicion.getFila()][posicion.getColumna()]);
    if (jaque != null) { // significa que hay jaque y puede haber jaque mate
        boolean jaqueMate = pnlTablero.comprobarFinPartida(jaque, detectarJaqueMate: true);
        if (jaqueMate) {
            String mensaje = null;
            String titulo = null;
            if (pnlTablero.esMiTurno()) {
                mensaje = "¡Jaque Mate! Has ganado.";
                titulo = "Victoria";
            } else {
                mensaje = "¡Jaque Mate! Otra vez será.";
                titulo = "Derrota";
            }
            JOptionPane.showMessageDialog( parentComponent: this,
                    mensaje,
                    titulo,
                    JOptionPane.INFORMATION_MESSAGE);
            desconectar();
            return;
        } else {
            modelo.addElement(jug + " >>> Aviso de JAQUE.");
        }
    }

    // cambiamos turno y movemos jlist
    pnlTablero.cambiarTurno();
```

Pero, ¿en qué consiste el método ***comprobarFinPartida*() orientado al jaque mate**? Vayamos por partes, y es que, **para librarse del jaque mate**, remitiéndonos de nuevo a la Wikipedia, **existen 3 posibilidades**:

- "Capturar la pieza atacante".
- "Interponer una de las piezas entre el atacante y el rey" (caso de *FiguraBucle.java)*.
- "Alejar el rey del ataque".

Como iremos viendo a lo largo del resto de este sprint, intentaremos poner solución a estas excepciones.

El método ***comprobarFinPartida()***, a la hora de buscar el jaque mate, recibe por parámetro la posición del rey que se pone en jaque y un *booleano* que indica la búsqueda del jaque mate como activa. **Si nos centramos en el último de esos 3 puntos, en "alejar al rey del ataque", debemos comprobar dos cosas:**

- Las potenciales posiciones del rey.
- Si otras figuras (o la misma que pone en jaque) son capaces de llegar a las potenciales posiciones del rey.

Para comprobar las posiciones del rey, es tan fácil como **ejecutar su correspondiente método de detección de movimientos** y almacenarlos en un *HashSet* particular en el que se incluirá, además, la posición que actualmente tiene.

```
public boolean comprobarFinPartida(Posicion posicion, boolean detectarJaqueMate) {
    boolean fin = true;

    // detectamos los posibles movimientos del rey rival
    int filaRey = posicion.getFila();
    int colRey = posicion.getColumna();
    Figura reyRival = arrayTablero[filaRey][colRey].getFigura();
    HashSet<Posicion> hsMovimientosReyRival = reyRival.getPosiblesMovimientos(posicion, arrayTablero, detectarJaqueMate: false);
    hsMovimientosReyRival.add(posicion); // añadimos la actual posición del rey, además (aunque no sea necesaria para ahogado)
```

Para comprobar si se pueden llegar a esas salidas del rey, recorreremos el tablero en busca de figuras rivales de este. A dichas figuras, les aplicaremos el método de detección de movimientos, pero poniendo la variable *booleana* ***detectarJaqueMate*** a true para que en su lógica de búsqueda se incluyan aquellas posiciones ocupadas por figuras del mismo color, o incluso aquellas casillas que estén vacías pero pueda llegar el rey en cuestión (recordemos los *getPosiblesMovimientos()* del peón y de la torre); **la idea de la aplicación de dicho *booleano* es, principalmente, incluir en el *HashSet* de movimientos aquellas casillas (vacías o no) susceptibles de ser ocupadas por el rival, porque si son ocupadas por el rival puede que sean ocupadas por el rey que se ha puesto en jaque.**

Con el *HashSet* de posiciones del rey y el *HashSet* de posiciones de la figura que en ese momento se tome mientras se recorre el bucle, **recorreremos ambos para buscar coincidencias y, si es así, añadir dicha coincidencia a un *HashMap* cuya clave sería la posible posición del rey, y cuyo valor sería un *ArrayList* de tipo *Casilla.java* en el que se incluyan aquellas casillas cuya figura podría llegar a la posición que hace de clave**.

```
// creamos un HashMap con las Casillas de hsMovimientosReyRival como clave, y una variable array de Posicion
// como clave que comprende las posiciones de las figuras que pueden llegar a la posición que está de clave.
// si el tamaño de ese array es de 1, debemos comprobar si la figura en cuestión que puede llegar, puede ser
// comida previamente por equipo rival desmantelando el JAQUE MATE
HashMap<Posicion, ArrayList<Casilla>> hmJaqueMate = new HashMap<>();

Figura figura;
Posicion pos;
HashSet<Posicion> hsMovimientosAux;
ArrayList<Casilla> arrayCasillas;
```

```
for (int fila = 0; fila < DIM_TABLERO; fila++) {
    for (int col = 0; col < DIM_TABLERO; col++) {
        figura = arrayTablero[fila][col].getFigura();
        pos = arrayTablero[fila][col].getPosicion();
        if (figura != null && (figura.esMia() != arrayTablero[posicion.getFila()][posicion.getColumna()].getFigura().esMia())) {
            hsMovimientosAux = figura.getPosiblesMovimientos(pos, arrayTablero, detectarJaqueMate: true);
            // recorremos hsMovimientosAux y hsJaqueMate en busca de coincidencia
            for (Posicion p : hsMovimientosAux) {
                for (Posicion p1 : hsMovimientosReyRival) {
                    if (p.getFila() == p1.getFila() && p.getColumna() == p1.getColumna()) {
                        // si no existe un registro para p1, inicializamos arrayCasillas
                        if (hmJaqueMate.get(p1) == null) {
                            arrayCasillas = new ArrayList<>();
                        } else { // de lo contrario, tomamos el valor del array y deberemos actualizarlo
                            arrayCasillas = hmJaqueMate.get(p1);
                        }
                        arrayCasillas.add(arrayTablero[fila][col]);
                        hmJaqueMate.put(p1, arrayCasillas);
                    }
                }
            }
        }
    }
}
```

La parte del código dedicada entonces a comprobar si se puede "alejar al rey del peligro" en ***comprobarFinPartida()***, se basaría en, si se dan las condiciones, **comprobar si el número de posibles movimientos que hay en el *HashSet* del rey, coincide con el número de claves almacenadas en el *HashMap***.

```
// la lógica cambia según si detectamos jaque mate o ahogado
if (detectarJaqueMate) { // comprobación de jaque mate
    // si el tamaño de hmJaqueMate es igual al de hsMovimientosReyRival, cubre todas las posibles posiciones
    // que pueda tomar el rey y aparentemente puede ser jaque mate
    arrayCasillas = hmJaqueMate.get(posicion);
    Casilla casilla = arrayCasillas.get(0);
    figura = casilla.getFigura();
    pos = casilla.getPosicion();
    boolean comprobar;
    comprobar = (hmJaqueMate.size() == hsMovimientosReyRival.size()) ? true : false;
```

Dada estas circunstancias, si no pudiese alejarse, **deberíamos comprobar si se puede taponar el avance de la figura que ha puesto en jaque al rey, o directamente comérsela para sortear el jaque mate**. En vistas a este cometido, recogemos la ruta de la figura que podría comerse al rey y la almacenamos en un *HashSet*: en las *FigurasBucle.java* aplicamos un método abstracto (***detectarRutaJaque()***) para saber cuál sería esa ruta, mientras que para las *FiguraNoBucle.java* añadiríamos simplemente su posición actual porque la única manera de evitar el jaque mate sería comiéndosela.

```
// comprobamos el valor del booleano, y si está a true entramos al if
if (comprobar) {
    // comprobamos que figuras pueden comerse al rey en su actual posición
    // si el tamaño de dicho array es de 1, debemos comprobar si a esa figura se le puede "taponar" el
    // movimiento y/o, directamente, comérsela para evitar el jaque mate
    // (partimos de la base de que no se pueden taponar 2 movimientos a la vez)
    if (arrayCasillas.size() == 1) {
        HashSet<Posicion> hsRutaJaque = new HashSet<>();
        // las figuras que se pueden "taponar" son aquellas cuya lógica de movimiento responde a un bucle
        // que les permite avanzar más de una casilla adyacente para comer (Torre, Alfil, Reina)
        if (figura instanceof FiguraBucle) {
            FiguraBucle figuraBucle = (FiguraBucle) figura;
            hsRutaJaque = figuraBucle.detectarRutaJaque(pos, arrayTablero);
        } else { // debemos añadir en hsRutaJaque pra las FiguraNoBucle su posición actual
            hsRutaJaque.add(pos);
        }
        // comprobamos si las figuras rivales pueden evitar el jaquemate buscando coincidencias entre
```

Para comprobar si el jaque mate se puede evitar taponando, o bien comiéndose la figura que ha hecho el jaque, debemos recorrer el tablero en busca de las figuras (excluyendo al rey) del jugador amenazado. **En cuanto encontremos una coincidencia, sabremos que la jugada en cuestión no se trata de un jaque mate**.

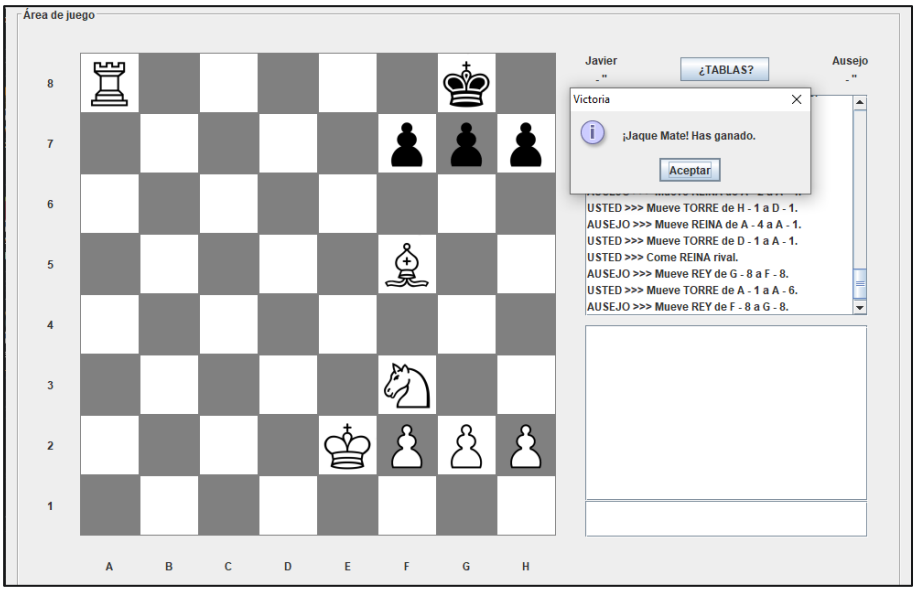
Y hasta aquí la programación del jaque mate.

(*Se hace uso del método ***detectarRutaJaque()***, que no se explica en este documento. No obstante, decir que se trata de un bucle en el que, una vez detectada la posición del rey, se traza la ruta que llega hasta dicha figura.)

```
            } else { // debemos añadir en hsRutaJaque pra las FiguraNoBucle su posición actual
                hsRutaJaque.add(pos);
            }
            // comprobamos si las figuras rivales pueden evitar el jaquemate buscando coincidencias entre
            // sus posibles movimientos y los de hsMovimientosAux
            for (int fila = 0; fila < DIM_TABLERO; fila++) {
                for (int col = 0; col < DIM_TABLERO; col++) {
                    if (arrayTablero[fila][col].getFigura() != null
                            && arrayTablero[fila][col].getFigura().esMia() ==
                            arrayTablero[posicion.getFila()][posicion.getColumna()].getFigura().esMia()) {
                        figura = arrayTablero[fila][col].getFigura();
                        if (!(figura instanceof Rey)) {
                            for (Posicion p : hsRutaJaque) {
                                for (Posicion p1 : figura.getPosiblesMovimientos(new Posicion(fila, col), arrayTablero, detectarJaqueMate: false)) {
                                    if (p.getFila() == p1.getFila() && p.getColumna() == p1.getColumna()) {
                                        fin = false;
                                        break;
                                    }
                                }
                                if (!fin) {
                                    break;
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    return fin;
}
```

*VISTA JAQUE (CABALLO BLANCO HACIA REY NEGRO):

*VISTA JAQUE MATE (“MATE DEL PASILLO”):



e. Sprint 5: Programación de diferentes reglas y recursos del ajedrez.

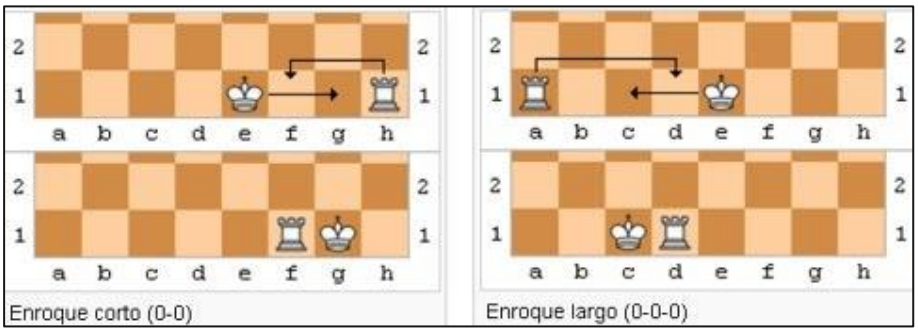
Tareas a cumplir:

- Programación del enroque, del ahogado, de la coronación y de la función comer el paso.
- Programación de la solicitud de empate.
- Programación de un cronómetro.

Desarrollo:

Afrontando ya el último tramo de nuestro proyecto respecto al diseño y programación de la aplicación, parece claro que lo que le queda a nuestro ajedrez es **integrar esa serie de recursos que existen en una partida de ajedrez real**, tales como el enroque, el ahogado, la coronación, o la posibilidad del peón de comer al paso.

El primer recurso a programar será el **enroque,** que es una técnica de movimiento en la que **se desplaza al rey dos casillas hacia una torre, desplazando a la par la torre justo una casilla al lado contrario del sentido de dicho movimiento**. Dada las dimensiones del tablero y de la propia distribución de las figuras, distinguimos dos tipos de enroques: **enroque corto y enroque largo**.



Para que el rey pueda enrocarse, deben darse las siguientes condiciones:

- Ni rey ni torre pueden haberse movido.
- No puede haber piezas entre medias del rey y la torre mediante.
- El rey no puede enrocarse si está en jaque o si el enroque atraviesa casillas a las que tiene acceso el rival.

Explicada la lógica del enroque, **programamos un método para cada uno tipo de los enroques que NO se ejecuta dentro de *getPosiblesMovimientos()*, sino que se ejecuta después de este cuando es nuestro turno y clicamos sobre nuestro rey** (lo consideramos movimiento de rey, no de torre). Es decir, lo invocamos dentro del ***actionPerformed()*** que desarrollamos en *Tablero.java*. Por otro lado, aunque no se muestre su implementación, puesto que el enroque es el único movimiento en el que dos figuras se desplazan a la vez, **debemos configurar el tablero para que dado un movimiento del rey en el que se desplace dos casillas, mueva también la torre pertinente**.

Como ya sabemos, según si juguemos figuras blancas o juguemos figuras negras, la posición en la que se encuentra el rey en el tablero cambia. Por ende, desde la perspectiva de las figuras blancas el rey se mueve hacia la derecha para hacer un enroque corto, mientras que hacia la izquierda para hacer un enroque largo; en el caso de controlar figuras negras, sería al revés.

Tal y como vemos en las siguientes capturas, a ambos métodos les pasamos la casilla en la que se encuentra el rey, el estado del tablero y una condición booleana que nos indica si jugamos con casillas blancas o negras. Con estos datos, comprobamos si se cumplen las condiciones antes mentadas y si añadimos (o no) el enroque al *HashSet* de posibles movimientos.

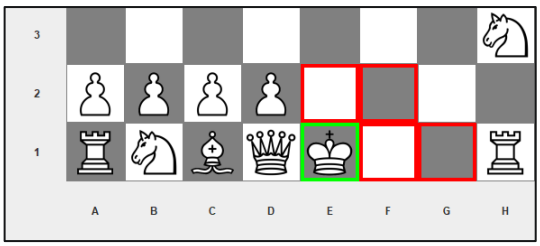
(***comprobarEnroqueLargo()*** es el mismo método pero valorando otras coordenadas a la hora de comprobar que no hay casillas mediante entre el rey y la torre en cuestión.)

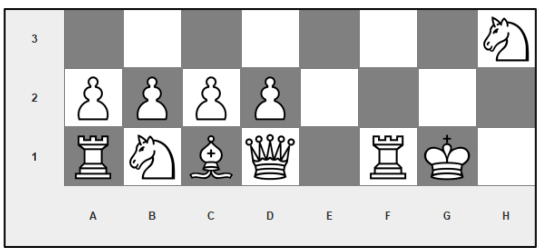
```
public void comprobarEnroqueCorto(Casilla casillaRey, Casilla[][] arrayTablero, boolean local) {
    boolean enroqueCorto = false;
    int colAux1, colAux2;
    Figura fAux, fAux1, fAux2;

    if (esPrimerMovimiento()) {
        if (local) {
            fAux = arrayTablero[0][7].getFigura();
            colAux1 = 6;
            fAux1 = arrayTablero[0][colAux1].getFigura();
            colAux2 = 5;
            fAux2 = arrayTablero[0][colAux2].getFigura();
        } else {
            fAux = arrayTablero[0][0].getFigura();
            colAux1 = 1;
            fAux1 = arrayTablero[0][colAux1].getFigura();
            colAux2 = 2;
            fAux2 = arrayTablero[0][colAux2].getFigura();
        }

        if (fAux != null && fAux instanceof Torre) {
            HashSet<Posicion> hsAux;
            Torre torre = (Torre) fAux;
            if (torre.esPrimerMovimiento()
                    && fAux1 == null
                    && fAux2 == null) {
                enroqueCorto = true;
                // recorremos tablero para determinar si se puede hacer enroque o no dependiendo
                // de si el rey termina el movimiento en una posición que le deja en jaque, o atravesando
                // una casilla a la que puede llegar el rival
                Casilla c;
                for (int i = 0; i < DIM_TABLERO; i++) {
                    for (int j = 0; j < DIM_TABLERO; j++) {
                        c = arrayTablero[i][j];
                        // si la figura no es nuestra, detectamos si puede bloquear el enroque dado que nuestro propio
                        // rey está en jaque o la ruta que atraviesa es un posible movimiento de una ficha rival
                        if (c.getFigura() != null && c.getFigura().esMia() != esMia()) {
                            hsAux = c.getFigura().getPosiblesMovimientos(new Posicion(i, j), arrayTablero,  detectarJaqueMate: false);
                            for (Posicion p : hsAux) {
                                if ((p.getFila() == 0 && p.getColumna() == colAux1)
                                        || (p.getFila() == 0 && p.getColumna() == colAux2)
                                        || (p.getFila() == casillaRey.getPosicion().getFila()
                                        && p.getColumna() == casillaRey.getPosicion().getColumna())) {
                                    enroqueCorto = false;
                                }
                            }
                        }
                    }
                }
            }
        }
        // añadimos las posiciones correspondientes al HashSet
        if (enroqueCorto) {
            hsPosiblesMovimientos.add(arrayTablero[0][colAux1].getPosicion());
        }
    }
}
```

1 - Comprobamos que no hay piezas mediante el rey y la supuesta torre, siendo además el primer movimiento de ambas figuras.

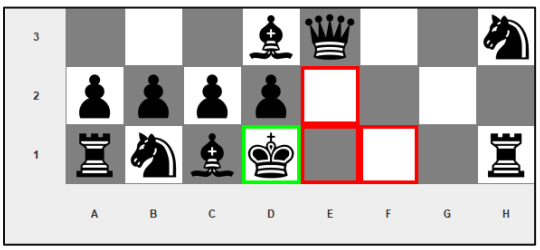
2 - Recorremos tablero para comprobar si el rey o las posiciones que intervienen en el enroque están en peligro para decidir si añadir el enroque al *HashSet* de movimientos.

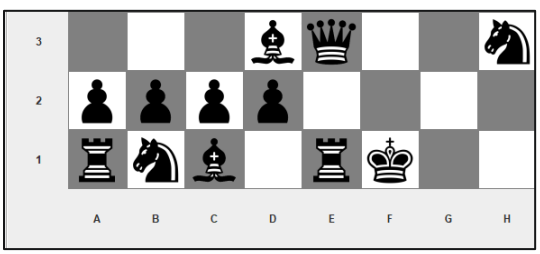
*EJEMPLO ENROQUE CORTO:



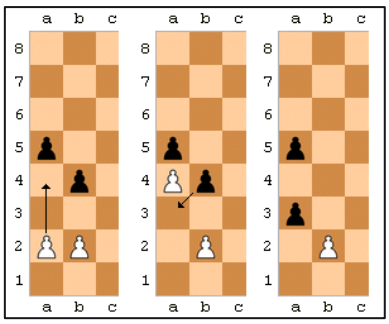


*EJEMPLO ENROQUE LARGO





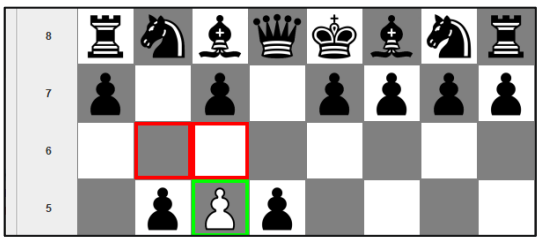
El segundo recurso que vamos a trabajar en este sprint es la función del peón denominada **comer al paso**. Se trata de una técnica de movimiento en la que **un peón, dada una posición determinada, tiene la posibilidad de capturar a un peón rival si este es la última figura que ha movido el rival habiendo aprovechado su primer movimiento para situarla dos casillas por encima**.

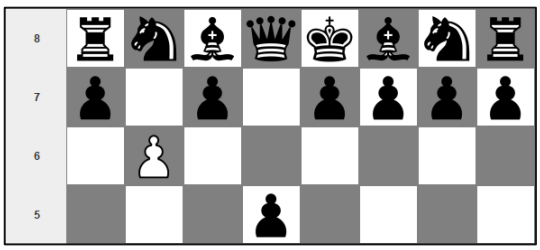


Vista la teoría, decido crear para *Peon.java* un método llamado ***comprobarComerAlPaso()*** que se integra dentro del ***getPosiblesMovimientos()*** de esa misma clase. Se dedica a comprobar si, cuando vamos a detectar los posibles movimientos de uno de nuestros peones, podemos encontrar en la misma fila de las columnas adyacentes un peón rival que se acabe de mover. Para comprobar si este peón se acaba de mover, añadiremos a la clase *Peón.java* una variable booleana llamada ***comerAlPaso*** que se ponga a true cuando dicho peón sea la última figura que haya movido el rival.

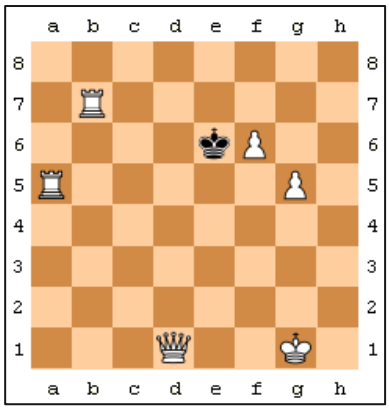
```
private void comprobarComerAlPaso(Posicion posicion, Casilla[][] arrayTablero) {
    int fila = posicion.getFila();
    int columna = posicion.getColumna();
    Peon pAux;
    // comprobamos si las casillas correspondientes se pueden comer al paso
    if (columna - 1 >= 0) {
        if (arrayTablero[fila][columna - 1].getFigura() instanceof Peon) {
            pAux = (Peon) arrayTablero[fila][columna - 1].getFigura();
            if (pAux.sePuedeComerAlPaso()) {
                hsPosiblesMovimientos.add(new Posicion( fila: fila + 1,  columna: columna - 1));
            }
        }
    }
    if (columna + 1 <= 7) {
        if (arrayTablero[fila][columna + 1].getFigura() instanceof Peon) {
            pAux = (Peon) arrayTablero[fila][columna + 1].getFigura();
            if (pAux.sePuedeComerAlPaso()) {
                hsPosiblesMovimientos.add(new Posicion( fila: fila + 1,  columna: columna + 1));
            }
        }
    }
}
```

Como último detalle de programación al respecto de este **comer al paso**, recordar que es el único movimiento en el que se come, pero en el que la figura que come no se desplaza a la casilla de la figura que es comida. Es por ello que, retrotrayéndonos a ***MensajeMovimiento.java***, debemos marcar que se come, pero configurar la interfaz para que cuando reciba un movimiento en el que se coma y la casilla destino no contenga ninguna figura, mire en las filas adyacentes de la misma columna para eliminar el peón pertinente del tablero.





El tercer recurso a implementar, será el del **ahogado**. El ahogado **significa un fin de partida que surge cuando NO HAY JAQUE y el jugador con turno no tiene movimientos legales posibles que no condenen a su rey a ser comido por el rival**. Este fin de partida no da la victoria a ningún jugador, finalizando en tablas.



Para comprobar si se da esta situación, **debemos ordenar a *Interfaz.java* que después de recibir un *MensajeMovimiento.java* y en el caso de que *comprobarJaque()* devuelva *null*, ejecute** un método que nos ayude a comprobar esto. Este método es ***comprobarAhogado()***, que se apoya en el recién explicado ***comprobarFinPartida()***.

```java
public boolean comprobarAhogado() {
    // variables usadas a lo largo del método
    boolean mia = false;
    int contador = 0;
    Figura f;
    Casilla casillaRey = null;
    Rey rey;
    HashSet<Posicion> hsPosicionesRey = new HashSet<>();
    boolean ahogado = false;
    // definimos el color de la figura a buscar dependiendo de si es nuestro turno o no
    if (miTurno)
        mia = false;
    else
        mia = true;
    // recorremos el tablero en busca del rey y definimos un contador de movimientos
    for (int i = 0; i < DIM_TABLERO; i++) {
        for (int j = 0; j < DIM_TABLERO; j++) {
            f = arrayTablero[i][j].getFigura();
            if (f != null && f.esMia() == mia) {
                contador += f.getPosiblesMovimientos(new Posicion(i, j), arrayTablero, detectarJaqueMate: false).size();
                if (f instanceof Rey) {
                    casillaRey = arrayTablero[i][j];
                    rey = (Rey) arrayTablero[i][j].getFigura();
                    hsPosicionesRey = rey.getPosiblesMovimientos(new Posicion(i, j), arrayTablero, detectarJaqueMate: false);
                }
            }
        }
    }
    // una vez recorrido, comparamos si el número de movimientos posibles coincide con el número de movs del rey
    // lo que significa que solo puede moverse el rey
    if (contador == hsPosicionesRey.size()) {
        ahogado = comprobarFinPartida(casillaRey.getPosicion(), detectarJaqueMate: false);
    }
    return ahogado;
}
```

En el código podemos leer que, además de tener en cuenta de quién es el turno para determinar la propiedad del rey a comprobar si se ahoga, recorremos el tablero en búsqueda de dicha figura para poder detectar sus posibles movimientos y almacenarlos en un *HashSet* auxiliar.

Mientras se recorre dicho tablero, usamos la variable local *contador* como sumatorio del número de movimientos posibles de determinadas figuras, para finalmente comprobar si *contador* y el número de registros del *HashSet* de movimientos del rey coinciden. Si así, fuere, se entraría al método *comprobarFinPartida()* con *detectarJaqueMate* como *false* para comprobar si se cumple el ahogamiento en función de si el rey puede escaparse a alguna casilla a la que no llegue el rival.

Dicho *booleano* determina que el comportamiento del método mute en función de si lo que hay que comprobar es el jaque mate o si se da el ahogado, cobrando en este instante el sentido que pretendíamos darle.

Como ya hemos hablado de *comprobarFinPartida()*, no hace falta hacer incidencia en exceso en este método, pero cabe recordar lo siguiente: se recorre el tablero almacenando en un *HashMap* como claves las posiciones que puede tomar el rey, y como valor un *ArrayList* con las casillas cuya figura puede llegar a la clave.

```
public boolean comprobarFinPartida(Posicion posicion, boolean detectarJaqueMate) {
    boolean fin = true;

    // detectamos los posibles movimientos del rey rival
    int filaRey = posicion.getFila();
    int colRey = posicion.getColumna();
    Figura reyRival = arrayTablero[filaRey][colRey].getFigura();
    HashSet<Posicion> hsMovimientosReyRival = reyRival.getPosiblesMovimientos(posicion, arrayTablero, detectarJaqueMate: false);
    hsMovimientosReyRival.add(posicion); // añadimos la actual posición del rey, además (aunque no sea necesaria para ahogado)
    HashMap<Posicion, ArrayList<Casilla>> hmJaqueMate = new HashMap<>();

    Figura figura;
    Posicion pos;
    HashSet<Posicion> hsMovimientosAux;
    ArrayList<Casilla> arrayCasillas;
    for (int fila = 0; fila < DIM_TABLERO; fila++) {
        for (int col = 0; col < DIM_TABLERO; col++) {
            figura = arrayTablero[fila][col].getFigura();
            pos = arrayTablero[fila][col].getPosicion();
            if (figura != null && (figura.esMia() != arrayTablero[posicion.getFila()][posicion.getColumna()].getFigura().esMia())) {
                hsMovimientosAux = figura.getPosiblesMovimientos(pos, arrayTablero, detectarJaqueMate: true);
                // recorremos hsMovimientosAux y hsJaqueMate en busca de coincidencia
                for (Posicion p : hsMovimientosAux) {
                    for (Posicion p1 : hsMovimientosReyRival) {
                        if (p.getFila() == p1.getFila() && p.getColumna() == p1.getColumna()) {
                            // si no existe un registro para p1, inicializamos arrayCasillas
                            if (hmJaqueMate.get(p1) == null) {
                                arrayCasillas = new ArrayList<>();
                            } else { // de lo contrario, tomamos el valor del array y deberemos actualizarlo
                                arrayCasillas = hmJaqueMate.get(p1);
                            }
                            arrayCasillas.add(arrayTablero[fila][col]);
                            hmJaqueMate.put(p1, arrayCasillas);
                        }
                    }
                }
            }
        }
    }
```

Una vez recorrido el tablero, **se producirá la situación de ahogado si el tamaño del *HashMap* es menor por 1 al del *HashSet* de movimientos del rey en situaciones de *detectarJaqueMate* a *false***. Es menor porque, recordemos, añadimos en este método añadimos la propia posición del rey a su *HashSet* de movimientos, siéndonos totalmente innecesario en el ahogado porque no se necesita el requisito previo de que se haya hecho jaque.

```
        } else {
            fin = false;
        }
    } else {
        // caso ahogado: no hace falta comprobar si se puede taponar porque previamente,
        // antes de la llamada a este método, se comprueba que solo puede moverse el rey
        // recordamos que se añade al hs del rey su posición actual cuando no es necesaria
        if (hmJaqueMate.size() == hsMovimientosReyRival.size() - 1) {
            fin = true;
        } else{
            fin = false;
        }
    }

    return fin;
}
```

*VISTA REY AHOGADO:

La cuarta y una de las técnicas más conocidas del mundo del ajedrez, es la de la **coronación**. **Se trata de un proceso en el que cuando un peón logra alcanzar la otra fila extrema del tablero, su dueño es recompensado con la posibilidad de promocionar su peón y cambiarlo por un caballo, un alfil, una torre o una dama**.

Su programación empieza en *Interfaz.java*, cuando esta detecta un *MensajeMovimiento.java* en el que hay un peón que ha llegado lograr a la primera fila enemiga. En función de si promocionamos nosotros, o lo hace el rival, la interfaz reacciona de una manera diferente: o simplemente recibiendo una instancia de ***MensajePromocion.java***, o con la emergencia de un ***JOptionPane*** que nos dará a elegir la figura a la que promocionar, convirtiendo dicha selección en un ***MensajePromocion.java*** en el que indicamos la nueva figura y la posición que ocupará.

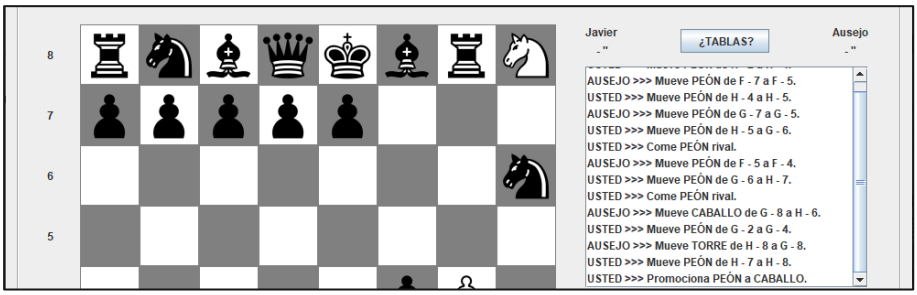
```java
public class MensajePromocion implements Mensaje, Serializable {

    // VARIABLES DE INSTANCIA
    private Jugador jugador; // jugador que lo envía
    private Figura figuraPromocion;
    private Posicion posDestino;

    public MensajePromocion(Jugador jugador, Figura figuraPromocion, Posicion posDestino) {
        this.jugador = jugador;
        this.figuraPromocion = figuraPromocion;
        this.posDestino = posDestino;
    }
```

```java
public void enviarMensajePromocion(Figura figuraPromocion, Posicion posDestino) {
    try {
        Mensaje mensaje = new MensajePromocion(jugador, figuraPromocion, posDestino);
        flujoEscritura.writeObject(mensaje);
        flujoEscritura.flush();
    } catch (IOException e) {
        desconexion();
    }
}
```

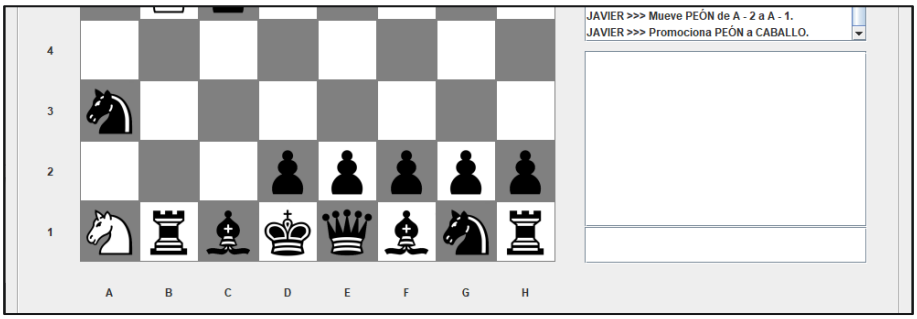
Como vemos en las siguiente capturas, de la interfaz nace un ***JOptionPane*** que indicará nuestra selección y la posición a ocupar al método ***enviarMensajePromocion()*** que encontramos en *HiloCliente.java*. Posteriormente, la interfaz lo interpreta como tal y aplica los cambios pertinentes, cambiando el icono del botón y la imagen figura que ocupa dicha casilla.

```
public void seleccionarFiguraPromocion(Posicion posDestino) {
    String[] figuras = {"Alfil", "Caballo", "Reina", "Torre"};
    Figura figuraPromocion = null;
    while (figuraPromocion == null) {
        int opcion = JOptionPane.showOptionDialog( parentComponent: this,
                message: "Seleccione la figura a la que desea promocionar:", title: "Promoción",
                JOptionPane.DEFAULT_OPTION, JOptionPane.QUESTION_MESSAGE, icon: null, figuras, figuras[0]);
        switch (opcion) {
            case 0:
                figuraPromocion = new Alfil( local: true);
                break;
            case 1:
                figuraPromocion = new Caballo( local: true);
                break;
            case 2:
                figuraPromocion = new Reina( local: true);
                break;
            case 3:
                figuraPromocion = new Torre( local: true);
                break;
            default:
                figuraPromocion = null;
                break;
        }
    }

    hiloCliente.enviarMensajePromocion(figuraPromocion, posDestino);
}
```

*VISTA JUGADOR PROMOCIONANDO PEÓN BLANCO EN H8 A CABALLO:



*VISTA JUGADOR VIENDO PROMOCIÓN RIVAL (CABALLO A1):



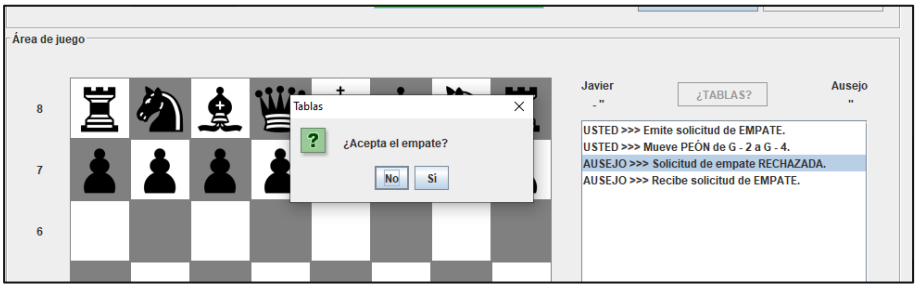
Casi llegando al final, del sprint, nos ponemos con la **posibilidad de solicitar un empate**, terminando así con el abanico de clases que derivan de *Mensaje.java* y diseñando ***MensajeEmpate.java***. **La idea es que cada jugador tenga un botón para solicitar empate el cual puede usar 1 vez siempre que sea su turno**, por lo que habilitamos el botón empate de la interfaz según este criterio para que cuando toque se invoque al método ***enviarMensajeEmpate()*** de la clase *HiloCliente.java*.

```
public class MensajeEmpate implements Mensaje, Serializable {

    // VARIABLES DE INSTANCIA
    private Jugador jugador; // jugador que envía
    private boolean tablas, iniciaSolicitud;

    public MensajeEmpate(Jugador jugador, boolean tablas, boolean iniciaSolicitud) {
        this.jugador = jugador;
        this.tablas = tablas;
        this.iniciaSolicitud = iniciaSolicitud;
    }
```

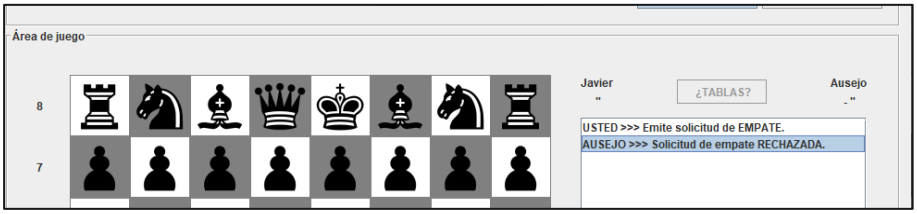
```
public void enviarMensajeEmpate(boolean empate, boolean inicioSolicitud) {
    hiloCliente.enviarMensajeEmpate(empate, inicioSolicitud);
}
```

En *Interfaz.java* programamos un comportamiento para este tipo de mensajes de acuerdo a la siguiente premisa: si se inicia la solicitud de empate, se manda un *MensajeEmpate.java* al servidor en el que las variables *booleanas aceptaTablas* e *iniciaSolicitud* están a true; en cuanto la interfaz rival detecte que le llega un *MensajeEmpate.java* no emitido por él, le aparece por pantalla un *JOptionPane* que le obliga a responder si acepta el empate o no con un nuevo *MensajeEmpate.java*: *iniciaSolicitud* se pone a *false*; si está de acuerdo con las *tablas*, se pondrá a *true*, de lo contrario, a *false*.

*VISTA RECIBIMIENTO SOLICITUD DE EMPATE:



*VISTA SOLICITUD DE EMPATE ACEPTADA Y RECHAZADA:



Llegamos ya, por fin, al último elemento a programar de la aplicación: el **cronómetro**. El propósito de este cronómetro es **establecer una especie de *time out* al jugador que tenga el turno**: si pasados 30 segundos no ha movido figura, se entiende que no está en la partida y se desconecta a ambos jugadores para evitar pérdidas de tiempo intencionadas.

Para ello, crearemos una clase llamada ***HiloCronometro.java*** que funcionará tal que así:

- **Cuando empiece la partida, *Interfaz.java* iniciará el cronómetro para el jugador de figuras blancas** (que es el que tiene el primer turno).
- **Cuando se cambie de turno, este cronómetro se apagará y se iniciará una nueva instancia para el jugador rival**.
- **Cada segundo que corra del cronómetro, se actualiza la interfaz indicando el tiempo de turno restante**.
- **Cuando se esté a la espera de elección de figura de promoción o de respuesta a la solicitud de empate, el cronómetro se pausa a la espera de la decisión. Cuando se tome, este vuelve a funcionar.**

```java
public class HiloCronometro implements Runnable {

    // CONSTANTES DE CLASE
    private final static int TIEMPO_TURNO = 30;

    // VARIABLES DE INSTANCIA
    private Interfaz interfaz;
    private boolean esMiTurno;
    private boolean control;
    private boolean pausa;

    public HiloCronometro(Interfaz interfaz, boolean esMiTurno) {
        this.interfaz = interfaz;
        this.esMiTurno = esMiTurno;
        control = true; // variable que controla la vida del hilo
        pausa = false; // variable que bloquea hilo en caso de espera de coronación
                       // o espera respuesta solicitud empate
    }
```

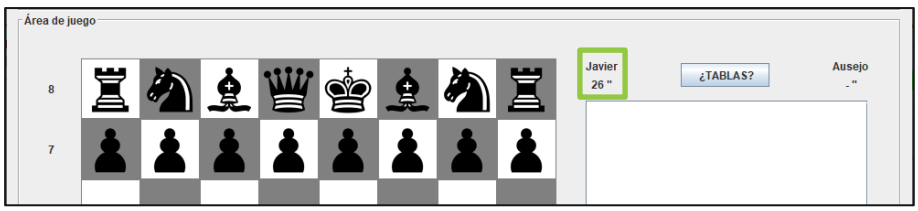
```java
@Override
public void run() {
    int tiempo = TIEMPO_TURNO;
    interfaz.reiniciarTiempos();
    interfaz.setTiempoTurno(esMiTurno, tiempoRestante: tiempo + "");
    while (control) {
        try {
            Thread.sleep( millis: 1000);
            if (!pausa) {
                if (tiempo != 0) {
                    tiempo -= 1;
                    interfaz.setTiempoTurno(esMiTurno, tiempoRestante: tiempo + "");
                    if (tiempo == 0) {
                        apagarCronometro();
                        interfaz.desconectar();
                    }
                }
            } else {
                synchronized (this) {
                    wait();
                }
            }
        } catch (InterruptedException e) { }
    }
}
```

```java
public void apagarCronometro() {
    control = false;
    pausa = true;
}
```

```java
public void pausar() {
    pausa = true;
}
```

```java
public void resume() {
    pausa = false;
    synchronized (this) {
        notify();
    }
}
```

*VISTA CRONÓMETRO FUNCIONANDO:



*VISTA CRONÓMETRO A 0:

4. AMPLIACIÓN Y POSIBLES MEJORAS.

Respecto a la programación del proyecto, creo que hay muchos elementos que se han quedado en el tintero y que podrían haberse implementado haciendo de este ajedrez una aplicación mucho más funcional y atractiva.

Por ejemplo, habría sido interesante **construir una aplicación en torno a una base de datos** que registre una serie de usuarios con sus respectivas credenciales para poder almacenar de ellos información acerca del número de partidas jugadas, el ratio de victorias etc. Además, en torno a esto, se podrían establecer elementos de personalización tales como la posibilidad de escoger una foto de perfil representativa en la partida. También, al hilo de este inventario de partidas de cada jugador, podría haber sido interesante estudiar la posibilidad de diseñar un *matchmaking* en torno a estos datos.

Dado que una parte importante del proyecto tiene que ver con la comunicación, me habría gustado también estudiar la viabilidad de **implementar un chat de voz**, suponiendo, si se pudiera, varios pasos adelante respecto al servicio de chat de texto que hemos incorporado en la aplicación.

En lo que refiere al funcionamiento de la aplicación tal y como se encuentra diseñada, tengo algunas **dudas acerca de programar según qué elementos** porque al final se trata de una aplicación orientada a **jugar al ajedrez de forma distendida y amateur**. Un ejemplo de ello, es que no se programa cuando un jugador ejecuta un movimiento en el que mete a su propio rey en la boca del lobo suponiendo jaque mate (entrando en juego si el jugador rival se da cuenta y come al rey). Aunque, por otro lado, estoy satisfecho con la implementación del método que detecta el fin de partida, ya sea por jaque mate o por rey ahogado.

Aunque contento con el resultado, me quedo con la espina clavada de **no haber trabajado más según qué elementos estéticos** como puede ser el propio *JPanel* que se convierte en el corazón del servicio de chat. También, echo en falta algún recurso sonoro que ayudara al jugador a asimilar el *modus operandi* de la aplicación (que tampoco es que sea complicado, pero hace del programa algo más trabajado e intuitivo si se programa correctamente).

En definitiva, también se tiene que tener en cuenta que se trata de un proyecto que debe tener una duración estimada máxima de 40h, que creo que en este caso han sido rentabilizadas porque se han tocado muchos y diferentes elementos.

5. CONCLUSIÓN.

Una vez terminado, se puede tomar cierta distancia con el proyecto para ver si estás satisfecho con el resultado o no. A pesar de las ampliaciones señaladas en el anterior punto, puedo decir que, sin subirme a la parra y desde una perspectiva humilde, he terminado por programar un minijuego de ajedrez online que tiene muchos de los elementos que podemos encontrar en otros juegos de mesa masivos de internet.

Aunque, repito, hay elementos mejorables, estoy contento con el resultado del proyecto porque no ha sido una simple recapitulación de información en clave *Java* asimilada durante estos 2 años de grado, sino que ha sido un ejercicio de relación de lo aprendido y que en muchos casos me ha llevado a darle una vuelta de tuerca a los contenidos vistos en clase implementando tecnologías con las que no habíamos trabajado (por ejemplo, en clave de comunicación de red el uso de *ObjectInputStream* y *ObjectOutputStream*).

Como se ha visto a lo largo del documento, en definitiva, **este ajedrez se ejecuta, principalmente, entre hilos (*threads*), comunicación en red a través de sockets TCP, y clases abstractas e interfaces**.

También, por iniciativa propia, me ha servido para medirme con un nuevo entorno de trabajo: acostumbrado a trabajar en *NetBeans*, este proyecto ha sido íntegramente programado en *IntelliJ,* vinculado, además, a *GitHub*, donde se puede encontrar el proyecto completo en "*https://github.com/javierausejo/Ajedrez*".

Como último detalle a subrayar, decir que ha sido un PFC que da fuerza a la intuición que tengo de que programar, normalmente, no es diseñar algo de manera abstracta sin tener en cuenta el camino recorrido. Esto que explico, para mí, se ve reflejado en lo que rodea a *Figura.java*: hay que sentarse a pensar cómo quieres que sea tu aplicación para que en función de tu diseño (*FiguraBucle.java* y *FiguraNoBucle.java*) puedas controlar todas aquellas circunstancias que puedan darse mientras se ejecute (jaque, jaque mate o ahogado). **En este proyecto, en cada paso que se daba se debían revisar los 5 anteriores al respecto porque, creo, la principal dificultad de este ajedrez tiene que ver con la coordinación de los pasos, el procedimiento a seguir para gestionar las excepciones y la capacidad de saber cómo tienes que hacer lo que quieres hacer**.

6. BIBLIOGRAFÍA.

- https://www.codeproject.com/Tips/991180/Java-Sockets-and-Serialization
- https://stackoverflow.com/questions/1453028/how-to-send-and-receive-serialized-object-in-socket-channel/1453163
- https://stackoverflow.com/questions/56405854/java-client-server-application-throws-java-net-socketexception-connection-rese
- https://stackoverflow.com/questions/910374/why-does-java-have-transient-fields
- Libro "*Programación de Servicios y Procesos"*, 2ª Edición, Ed. Garceta.