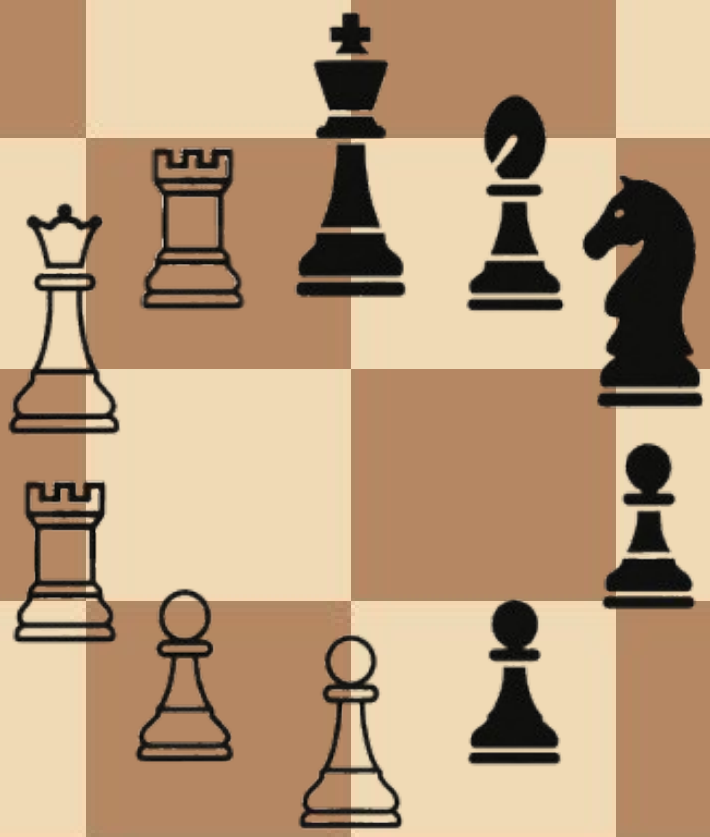
Memoria Trabajo Final Grado Superior



Chess Game

Tutor del Proyecto: Néstor Fabio Muñoz García
Tutor del Curso: Fernando Barandalla Fernández
Autor del Proyecto: Javier Gutiérrez Ezquerro

Sumario

♚ ¿De que va el trabajo? ♚

El trabajo trata sobre Ajedrez, con el puedes jugar una partida contra alguien en el mismo ordenador o guardar tus partidas que tienes en físico para luego poder verlas en la propia aplicación. También si quieres jugar contra alguien más cómodamente tiene una opción para crear una partida en LAN y que se pueda conectar otra persona con otro ordenador, y por último tiene un sistema de problemas.

♕ ¿Por que lo quise hacer? ♕

Desde 4º de primaria me ha gustado el ajedrez y he estado jugando desde entonces, teniendo clases, jugando torneos. Cuando llegue a grado superior pensé en que podría llegar a hacer un ajedrez por mi mismo pero nunca conseguía tener todo el tiempo que me hubiera gustado tener para hacer el ajedrez y entonces llego cuando tuve que elegir que tipo de TFG quería hacer y se me ocurrió que podría hacer el plan que llevaba mucho tiempo que quería hacer y así poder hacer un buen trabajo final sin perder la motivación. Y como un extra ya que también he aprendido bases de datos con este programa podría guardar las partidas que tras tantos años he estado teniendo en papel.

♕ Herramientas de Desarrollo ♕

Para la creación de esta aplicación se ha utilizado principalmente **Java puro** como lenguaje de programación. Java fue elegido debido a su portabilidad y los dos años que he estado programando en él.

Además, para mejorar la apariencia visual de la interfaz de usuario, se ha incorporado **HTML** en ciertos componentes. El uso de HTML ha permitido añadir decoraciones y estilos adicionales, logrando así una experiencia de usuario más atractiva y moderna.

En cuanto al entorno de desarrollo, se ha utilizado **Eclipse IDE**. Eclipse es uno de los entornos de desarrollo integrado más populares para Java y con el que más familiarizado me encuentro.

♕ Tiempo Empleado ♕

Actividad	Horas Estimadas
Lógica del ajedrez	60
Arrastrar las piezas	10
Jugar en LAN	35
Base de Datos	10
Pantallas de guardar-cargar partida y problemas	20
Memoria del trabajo y documentos para el usuario	25
Total:	160 horas

♔ Creación del tablero ♔

A la hora de crear el tablero de ajedrez, se genera un array bidimensional de botones de 9x9, el tablero de ajedrez básico es de 8x8 pero al añadir una columna y fila extra se pueden poner las coordenadas alineadas perfectamente con el tablero, lo cual da un efecto visual atractivo.

```
public void crearTablero(JButton[][] casillas, JButton casilla,JPanel panelTablero,
        ArrastraPieza arrastraPieza, JLabel textoFlotante,
        boolean verTiempo, boolean verMovimientos, boolean esProblema) {

    String nombreCoordenadas = null;
    int numeroFila = 0;
    char letraColumna = ' ';
    for (int fila = 0; fila < 9; fila++) {
        for (int columna = 0; columna < 9; columna++) {
            casilla = new JButton();
            casilla.setPreferredSize(new Dimension(60, 60)); // Tamaño de cada casilla
            casilla.setOpaque(true);
            casilla.setBorderPainted(false);
            casilla.setBorderPainted(false);
            if (fila == 8 && columna == 8)
                casilla.setEnabled(false);
            else if (fila == 8) {
                // casilla.setText(columna+"");
                casilla.setEnabled(false);
                letraColumna = (char) ('A' + columna);
                casilla.setText("" + letraColumna);
            } else if (columna == 8) {
                // casilla.setText(""+fila);
                casilla.setEnabled(false);
                numeroFila = 8 - fila;
                nombreCoordenadas = "" + numeroFila;
                casilla.setText(nombreCoordenadas);
            } else if ((fila + columna) % 2 == 0)
                casilla.setBackground(Colores.CASILLAS_BLANCAS);
            else
                casilla.setBackground(Colores.CASILLAS_NEGRAS);
            PonerPiezasTablero.colocarPiezasNormal(casilla, fila, columna);
            casillas[fila][columna] = casilla;
            arrastraPieza = new ArrastraPieza(panelTablero, casillas, textoFlotante,verTiempo,verMovimientos,esProblema);
            casilla.addMouseListener(arrastraPieza.new BotonMouseListener());
            panelTablero.add(casilla);
        }
    }
```

Figura 1: Creación del Tablero

El tablero se crea de dos formas diferentes que se diferencian entre lo que el usuario ve y el como se programa en él.

Para que el tablero se vea se le insertan JButtons, además de cambiarles su color de fondo para que se vea como un tablero de ajedrez. Se desactivan los botones que simbolizan las coordenadas del tablero, y a los botones del tablero se le añaden un addMouseListener para poder arrastrar las piezas (se explica más detalladamente adelante). En el siguiente apartado se habla sobre como se ponen las piezas en el modo clásico del ajedrez.

♚ Colocación de piezas ♚

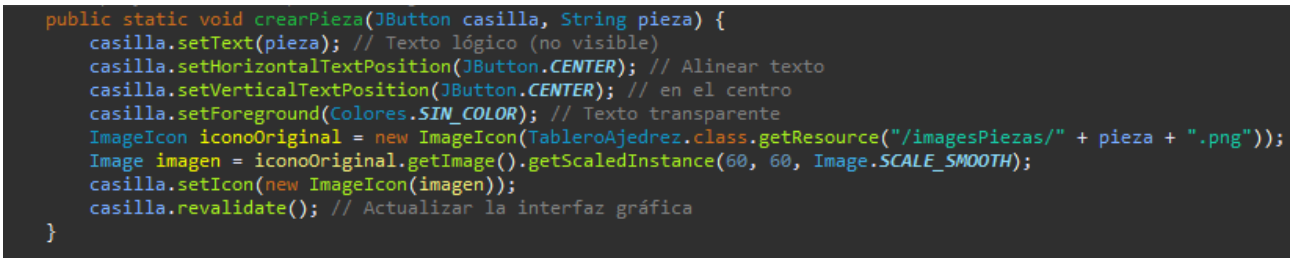
Aquí hago toda la colocación de las piezas para un tablero normal. Les pongo un nombre identificativo y les doy una apariencia para el jugador.

```
public static void colocarPiezas(JButton[][] casillas, JButton casilla, int fila, int columna) {

    String pieza="";
    if (fila==0)
        pieza="b";
    else if (fila==7)
        pieza="w";
    if (columna < 8) {
        // Filas principales (0 y 7)
        if (fila == 0 || fila == 7) {
            if (!pieza.equals("")) {
                // Añade color según la fila
                pieza = (fila == 0 ? "b" : "w");
            }
            switch (columna) {
                case 0:
                case 7: pieza += "T"; break; // Torres
                case 1:
                case 6: pieza += "C"; break; // Caballos
                case 2:
                case 5: pieza += "A"; break; // Alfiles
                case 3: pieza += "D"; break; // Dama
                case 4: pieza += "R"; break; // Rey
            }

        }
        // Peones
        else if (fila == 1) {
            pieza = "bP";
        }
        else if (fila == 6) {
            pieza = "wP";
        }

        // Solo llamas a crearPieza si hay una pieza para colocar
        if (!pieza.equals("")) {
            crearPieza(casilla, pieza);
        }
    }
}
```

Figura 2: Colocación de las piezas

Dependiendo de la posición inserta una pieza o no, le da un color a la pieza y so nombre, tras esto llama a la función "*crearPieza*".

♚ Función para Crear Piezas ♚



```
public static void crearPieza(JButton casilla, String pieza) {
    casilla.setText(pieza); // Texto lógico (no visible)
    casilla.setHorizontalTextPosition(JButton.CENTER); // Alinear texto
    casilla.setVerticalTextPosition(JButton.CENTER); // en el centro
    casilla.setForeground(Colores.SIN_COLOR); // Texto transparente
    ImageIcon iconoOriginal = new ImageIcon(TableroAjedrez.class.getResource("/imagesPiezas/" + pieza + ".png"));
    Image imagen = iconoOriginal.getImage().getScaledInstance(60, 60, Image.SCALE_SMOOTH);
    casilla.setIcon(new ImageIcon(imagen));
    casilla.revalidate(); // Actualizar la interfaz gráfica
}
```

Figura 3: Creación de Piezas

La función lo que hace es añadirle el texto al botón con el nombre pasado por parámetros. El texto solo es para la lógica, visualmente no se ve. Lo que si se ve es la imagen de la pieza que se le añade, para ello con el mismo nombre de la imagen y entrando en su carpeta ya se puede insertar y luego se escala para que se vea bien en el tablero.
Y este es el tablero con las piezas colocadas para poder iniciar una partida de ajedrez de la modalidad normal:

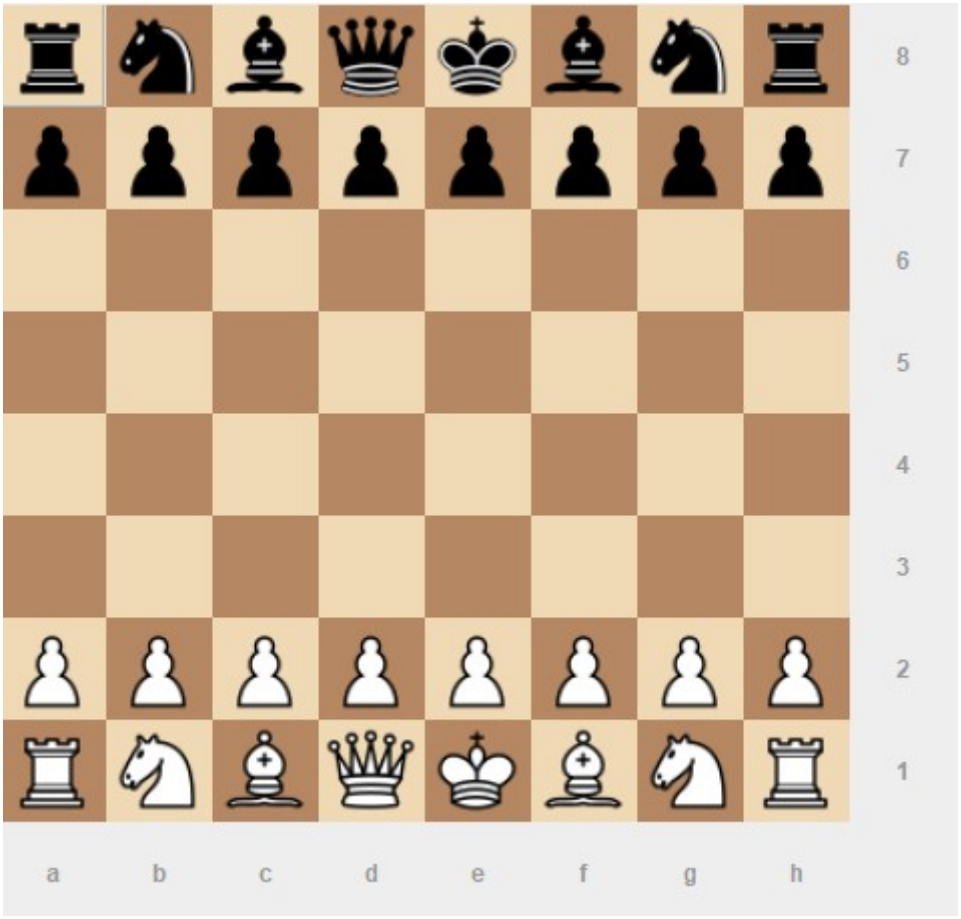


Figura 4: Apariencia del Tablero

Al ver la apariencia del tablero, lo siguiente que queda es ¿Cómo se mueven las piezas?
Pues en el siguiente apartado se enseñará esto.

♔ ¿Come se mueve? ♔

♕ Gestión de turnos ♕

Cuando el jugador toque una casilla del tablero llamara automáticamente a la función "*colorAMover*" para que juegue el jugador correcto.

```
public static boolean colorAMover() {

    if (jugadasTotales%2==0) //Mueven blacas
        return true;
    else //Mueven negras
        return false;
}
```

Figura 5: Jugador a mover

Tras saber a que jugador le toca jugar se llama a la función "*ArrastrarPieza*", que como su nombre indica, sirve para arrastrar/mover las piezas.

Esta última se divide en tres partes:

♕ Presionar ♕

Con esta función detecta que casilla se ha seleccionado, tras esto se llama a la función "*identificarPiezaParaMover*" (ver más adelante). Se guardan los movimientos posibles en una variable. Y tras terminar empieza la siguiente parte del movimiento de las piezas.

```
class BotonMouseListener extends MouseAdapter {
    @Override
    public void mousePressed(MouseEvent i) {
        FuncionesVisualesTablero.resetColores(casillas);

        origen = (JButton) i.getSource();
        if (origen.getText().equals(""))
            return;

        textoArrastrado = origen.getActionCommand();
        fichaSeleccionada = origen.getText();
        posicionOrigen = obtenerPosicion(origen);
        // Si no es el turno del color de la ficha seleccionada, no hacer nada
        if ((CalculosEnPartida.colorAMover() && !fichaSeleccionada.contains("w")) ||
            (!CalculosEnPartida.colorAMover() && !fichaSeleccionada.contains("b"))) {
            return;
        }

        // Aqui se le añade la funcion de poder arrastrar la pieza
        origen.addMouseMotionListener(new BotonMouseMotionListener());
        movimientos=MetodosMoverPiezas.identificarMovimientosDePieza(fichaSeleccionada, posicionOrigen, casillas);

        puntoInicialClick = i.getPoint();

        textoFlotante.setText(null);
        textoFlotante.setIcon(origen.getIcon());
        textoFlotante.setOpaque(false);
        textoFlotante.setBorder(null);
        textoFlotante.setSize(textoFlotante.getPreferredSize());
    }
```

Figura 6: Función Arrastrar Pieza

♕ Arrastrar ♕

```java
public void mouseDragged(MouseEvent c) {
    // Si no hay origen, el texto está vacío o no se ha registrado el punto inicial del click, salimos del método
    if (origen == null || origen.getText().equals("") || puntoInicialClick == null)
        return;

    // Cambia el cursor a una mano para indicar que se está arrastrando una pieza
    panelTablero.setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));

    // Calcula la distancia movida desde el punto inicial del click
    int dx = Math.abs(c.getX() - puntoInicialClick.x);
    int dy = Math.abs(c.getY() - puntoInicialClick.y);

    // Si se ha movido más de 5 píxeles en cualquier dirección, se considera un arrastre válido
    if (dx > 5 || dy > 5) {
        // Si el origen está habilitado y no está en jaque (color rojo), cambia el color para indicar arrastre
        if (origen.isEnabled() && !origen.getBackground().equals(Colores.JAQUE_ROJO))
            origen.setBackground(Colores.ARRASTRAR_PIEZA);

        // Obtiene el JFrame principal desde el panel del tablero
        JFrame frame = (JFrame) SwingUtilities.getWindowAncestor(panelTablero);

        // Convierte la posición del mouse relativa al origen a coordenadas del LayeredPane del JFrame
        Point p = SwingUtilities.convertPoint(origen, c.getPoint(), frame.getLayeredPane());

        // Centra el texto flotante respecto al cursor
        int x = p.x - textoFlotante.getWidth() / 2;
        int y = p.y - textoFlotante.getHeight() / 2;

        // Mueve y muestra el texto flotante en la nueva posición
        textoFlotante.setLocation(x, y);
        textoFlotante.setVisible(true);
    }
}
```

Figura 7: Arrastra Pieza

Con este método se consigue que la imagen de la pieza persiga el ratón del usuario por todo el tablero, esto da un efecto de que se esta arrastrando la pieza, y para potenciar el efecto se cambia el cursor para que parezca que se esta arrastrando la pieza.

♕ Soltar ♕

Esta función mira si la pieza esta dentro del tablero, tras comprobar eso llama a la función "*moverPiezas*" a la cual se le pasa los movimientos ya generados en la función de Presionar, y limpia el tablero de los movimientos posibles de la pieza que el jugador a movido.

```
    public void mouseReleased(MouseEvent a) {
        if ((puntoInicialClick == null))
            return;
        int dx = Math.abs(a.getX() - puntoInicialClick.x);
        int dy = Math.abs(a.getY() - puntoInicialClick.y);

        if ((origen != null && textoArrastrado != null && textoFlotante != null)
                && (dx > 8 || dy > 8)) {
            // Convertimos la posición del ratón al panel del tablero
            Point puntoEnTablero = SwingUtilities.convertPoint((Component) a.getSource(), a.getPoint(), panelTablero);

            // Obtenemos el componente (en este caso, el botón de destino) en la nueva posición del ratón
            Component destino = panelTablero.getComponentAt(puntoEnTablero);

            if (puntoEnTablero.x < 0 || puntoEnTablero.y < 0 ||
                    puntoEnTablero.x >= panelTablero.getWidth() ||
                    puntoEnTablero.y >= panelTablero.getHeight()
                    || !destino.isEnabled()){
                    // Está fuera del tablero
                    textoFlotante.setVisible(false);
                    textoArrastrado = null;
                    origen = null;
                    posicionOrigen = null;
                    fichaSeleccionada = null;
                    puntoInicialClick = null;
                    panelTablero.setCursor(Cursor.getDefaultCursor());
                    return;
                }

            // Si el destino es un botón y no es el mismo que el origen
            if (destino != origen && destino != null && destino instanceof JButton && destino.isEnabled()) {
                JButton botonDestino = (JButton) destino;
                String posicionDestino = obtenerPosicion(botonDestino);

                MetodosMoverPiezas.moverPiezas(posicionOrigen, posicionDestino, casillas, fichaSeleccionada,movimientos);
                textoFlotante.setVisible(false);
                textoFlotante.setIcon(null);
                origen = null;
                posicionOrigen = null;
                fichaSeleccionada = null;
            } else {
                fichaSeleccionada = null;
            }

            FuncionesVisualesTablero.resetColores(casillas);

            textoArrastrado = null;
            origen = null;
            posicionOrigen = null;
            fichaSeleccionada = null;
        }

        puntoInicialClick = null;
        panelTablero.setCursor(Cursor.getDefaultCursor());
    }
}
```

Figura 8: Soltar Pieza

El siguiente paso es saber que pasa cuando una pieza termina de hacer su movimiento, sea legal o ilegal.

♚ Movimientos en Tablero ♚

Aquí se explicará como se ejecuta el desplazamiento de las piezas, y que pasa en las casillas tras el movimiento de las piezas. Además de ver que otras funciones están ligadas a los movimientos. Para que un movimiento sea posible se necesita saber que movimientos puede hacer la pieza, los cuales se explican más detalladamente más adelante, y si ese movimiento no deja al propio rey en jaque.

♕ Movimientos normales ♕

Si el movimiento que el jugador a hecho es un movimiento posible de la pieza la casilla anterior de la pieza se vacía, y la nueva posición de la pieza se queda con todos sus valores

```
String[] movimientosValidos = movimientos.split(" ");

for (String movimiento : movimientosValidos) {
    if (movimiento.equals(destino)) {

        // Este metodo sirve para comprobar si un peon ha llegado a su casilla de
        // coronacion
        // Si ha llegado cornona si no, no hace nada
        String fichaOriginal = ficha;

        ficha = JugadaEspecialPeon.coronarPeon(filaDestino, colDestino, ficha, casillas);



        JButton casilla = casillas[filaDestino][colDestino];
        boolean hayPieza=false;
        if (!casillas[filaDestino][colDestino].getText().isEmpty())
            hayPieza=true;
        PonerPiezasTablero.crearPieza(casilla,ficha);

        casillas[filaOrigen][colOrigen].setText("");
        casillas[filaOrigen][colOrigen].setIcon(null);
```

Figura 9: Movimientos normales

Si el movimiento es legal pero se tienen que mover más de una pieza se les considerá movimientos especiales, los cuales son son el comer al paso y el enroque. Además de ver como se hace para cambiar el peón de pieza en su casilla de coronación.

♕ Movimientos especiales ♕

Cuando el movimiento es especial se llama a la función que controla cada movimiento especial, este consigue las piezas necesarias y las desaparece de una casilla y las genera en otro (todo siguiendo la lógica del ajedrez).

```
        // Esta parte solo es para ver si se puede comer al paso con el peon (porque
        // tiene que actualizar otras casillas)
        if (casillas[filaDestino][colDestino].getText().equals("bJa")
                && casillas[filaOrigen][colOrigen].getText().equals("wP")) {
            casillas[filaDestino + 1][colDestino].setText("");
            casillas[filaDestino + 1][colDestino].setIcon(null);
        } else if (casillas[filaDestino][colDestino].getText().equals("wJa")
                && casillas[filaOrigen][colOrigen].getText().equals("bP")) {
            casillas[filaDestino - 1][colDestino].setText("");
            casillas[filaDestino - 1][colDestino].setIcon(null);
        }
        // Aqui ya sigue con normalidad

        // Esto es para el enroque
        // Detectar si es un movimiento de rey de enroque (dos columnas de diferencia)
        if (mirarMoverEnroque(casillas, filaOrigen, colDestino, colOrigen))
            moverEnroque(casillas, filaOrigen, colDestino, colOrigen);

        JButton casilla = casillas[filaDestino][colDestino];
        boolean hayPieza=false;
        if (!casillas[filaDestino][colDestino].getText().isEmpty())
            hayPieza=true;
        PonerPiezasTablero.crearPieza(casilla,ficha);

        casillas[filaOrigen][colOrigen].setText("");
        casillas[filaOrigen][colOrigen].setIcon(null);
```

Figura 10: Como se llaman a los Movimientos Especiales

♕ Tras el Movimiento ♕

Aquí se ven todas las funciones que son llamadas tras los movimientos de las piezas. Dos son para guardar las jugadas, otro para calcular “movimientosPosibles”, otro para actualizar visualmente la interfaz del usuario con el texto del nuevo movimiento y la última se usa para ver si la partida a llegado a su fin. Luego se vacían los movimientos.

```
    CalculosEnPartida.guardarMovimientos(origen, destino, ficha);

    // Crea un hilo para realizar la tarea de decorar las jugadas
    ConvertirAJugadasAceptables tarea = new ConvertirAJugadasAceptables(ficha, fichaOriginal, casillas, destino, origen,hayPieza);
    Thread hilo = new Thread(tarea);
    hilo.start();
    FinPartida.IdentificarFinPartida(casillas, CalculosEnPartida.getJugadas());

    JugadaEspecialPeon.comerAlPaso(filaOrigen, filaDestino, ficha, casillas, colDestino,
            CalculosEnPartida.getJugadas());

    DetectarJaqueEnPartida.detertarPosicionJaque(casillas);
    SwingUtilities.invokeLater(() -> {
        ConvertirAJugadasAceptables.actualizarJugadasEnTablero(CrearTablero.getLabelDeMovimientosPartida());
    });
    movimientos = "";
}
```

Figura 11: Después de la jugada

♔ Movimientos de las Piezas ♔

```java
public abstract class Piezas {
    protected static String jugadasTotales = "";
    protected static int filaActual = 0;
    protected static int columnaActual = 0;

    public abstract String calcularMovimientos(String posicion, JButton[][] casillas, String ficha, boolean usarMovimientosEspeciales);

    protected static boolean verPeonesAlPaso(JButton[][] casillas, int fila, int columna) {
        if ((casillas[fila][columna].getText().equals("wJa")) || (casillas[fila][columna].getText().equals("bJa"))) {
            return false; // Si se ve un peon temporal (para comer al paso) devuelve true
        }
        return true;

    }

    protected void inicializarPosicion(String posicion) {
        filaActual = Integer.parseInt(posicion.substring(0, 1));
        columnaActual = Integer.parseInt(posicion.substring(1, 2));
        jugadasTotales = "";
    }

    protected String conseguirPosicion(String posicion) {
        return posicion;
    }

    protected static boolean mismoColor(JButton[][] casillas, int fila, int columna, String fichaColor) {
        String contenidoCasilla = casillas[fila][columna].getText();

        // Mira si las fichas son del mismo color
        boolean esMismoColor = (contenidoCasilla.contains("b") && fichaColor.contains("b"))
                || (contenidoCasilla.contains("w") && fichaColor.contains("w"));

        if (esMismoColor) {
            return true;
        } else {
            return false;
        }
    }



    protected static void conseguirJugadasLogicas(int fila, int columna) {
        if (fila >= 0 && fila < 8 && columna >= 0 && columna < 8) {
            //Guardar todas las posiciones posibles a las que puede ir una pieza
            jugadasTotales += fila + "" + columna + " ";

        }
    }
    protected static String VerconseguirJugadasLogicas() {
        return jugadasTotales;
    }
```

Figura 12: Clase de las piezas

Aquí se va a explicar de donde salen todos los movimientos de las piezas que se usan para luego poder moverlas. Primero todas las piezas heredan de la clase “Piezas” estos métodos y variables:

♟ CalcularMovimientos todas las piezas lo tienen es el que usan para calcular sus movimientos.

♟ VerPeonesAlPaso: sirve para ver los peones temporales que dejan los peones que se pueden comer al paso.

♟ InicializarPosicion: Limpia los valores comunes de las piezas.

♟ MismoColor: Mira si la casilla pasada contiene una pieza del mismo color que la pieza que quiere ir a esa casilla (a esta solo se le llama cuando hay una pieza en la casilla).

♟ ConseguirJugadasLogicas: guarda los movimientos totales a los que las piezas pueden ir.

♟ VerConseguirJugadasLogicas: te devuelve todas las jugadas de la pieza.

♕ Movimientos Especiales Peón ♕

♖ Comer Al Paso ♖

```java
public static void comerAlPaso(int filaOrigen, int filaDestino, String ficha, JButton[][] casillas, int columna,
        HashMap<Integer, String> jugadas) {

    Integer claveMasNueva = jugadas.size();
    String valorMasNuevo = jugadas.get(claveMasNueva);

    // Regex para saber si blancas o negras han hecho un movimiento doble con el
    // peon
    String regexBlancas = "wP-6\\d-4\\d";
    String regexNegras = "bP-1\\d-3\\d";

    // Este comprueba si el regex es correcto, y si hay un peon de color contrario
    // al lado del peon (esto se hace para ver si es posible ejecutar
    // el comer al paso)

    boolean puedeComerAlPaso = false;

    if (valorMasNuevo.matches(regexBlancas)) {
        // Peón blanco: busca peón negro a izquierda o derecha
        boolean izquierda = (columna - 1 >= 0) && casillas[filaDestino][columna - 1].getText().equals("bP");
        boolean derecha = (columna + 1 < 8) && casillas[filaDestino][columna + 1].getText().equals("bP");
        puedeComerAlPaso = izquierda || derecha;
    } else if (valorMasNuevo.matches(regexNegras)) {
        // Peón negro: busca peón blanco a derecha o izquierda
        boolean derecha = (columna + 1 < 8) && casillas[filaDestino][columna + 1].getText().equals("wP");
        boolean izquierda = (columna - 1 >= 0) && casillas[filaDestino][columna - 1].getText().equals("wP");
        puedeComerAlPaso = derecha || izquierda;
    }

    if (puedeComerAlPaso) {
        // Este if sirve para saber a qué color se le hace el comer al paso, creando un
        // peon temporal que si no se come en ese turno desaparece
        if (ficha.equals("wP")) {
            if (filaOrigen - 1 >= 0) { // Control de límites
                casillas[filaOrigen - 1][columna].setText("wJa");
                casillas[filaOrigen - 1][columna].setHorizontalTextPosition(JButton.CENTER);
                casillas[filaOrigen - 1][columna].setVerticalTextPosition(JButton.CENTER);
                casillas[filaOrigen - 1][columna].setForeground(Colores.SIN_COLOR);
            }
        } else {
            if (filaOrigen + 1 < 8) { // Control de límites
                casillas[filaOrigen + 1][columna].setText("bJa");
                casillas[filaOrigen + 1][columna].setHorizontalTextPosition(JButton.CENTER);
                casillas[filaOrigen + 1][columna].setVerticalTextPosition(JButton.CENTER);
                casillas[filaOrigen + 1][columna].setForeground(Colores.SIN_COLOR);
            }
        }
    } else {
        // Si no se puede comer al paso, limpia los peones temporales
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                if (casillas[i][j].getText().equals("wJa") || casillas[i][j].getText().equals("bJa"))
                    casillas[i][j].setText("");
            }
        }
    }
```

Figura 13: Clase Comer al Paso

Esta función registra si el peón se ha movido dos casillas adelante, si es así comprueba si tiene algún peón de distinto color a su derecha o izquierda, si esto es verdadero crea un peón temporal de su mismo color, el cual solo puede ser capturado por otro peón y que desaparece en el siguiente movimiento.

Para hacer que desaparece se tiene que mirar el historial de la partida para saber si ha pasado un turno, si es así borra el peón temporal

♖ Coronar ♖

Primero mira si la casilla a la que va el peón es la última casilla a la que puede avanzar, tras comprobarlo le pregunta al usuario en que pieza quiero coronar el peón

```
public static String coronarPeon(int fila, int columna, String ficha, JButton[][]casillas) {
    // Verifica si es una casilla de coronación
    JButton casilla=casillas[fila][columna];
    String color = ficha.substring(0, 1);
    if ((fila == 0 && ficha.equals("wP")) || (fila == 7 && ficha.equals("bP"))) {
         String[] tipos = {"D", "T", "A", "C"};
```

Figura 14: Clase Coronar Peón

♕ Movimiento Especial Rey ♕

♖ Comprobar Enroque ♖

```
public static String enroque(String ficha, HashMap<Integer, String> jugadas, JButton[][] casillas, String pos, boolean verEnroque) {
    String movimiento="";
    if (!verEnroque)
        return "";
    boolean torreCortaMovidaBlancas = false;
    boolean torreLargaMovidaBlancas = false;
    boolean torreCortaMovidaNegras = false;
    boolean torreLargaMovidaNegras = false;
    boolean reyMovidoBlancas = false;
    boolean reyMovidoNegras = false;

    for (String jugada : jugadas.values()) {
        // Torres blancas
        if (jugada.startsWith("wT-77-")) torreCortaMovidaBlancas = true;
        if (jugada.startsWith("wT-70-")) torreLargaMovidaBlancas = true;

        // Torres negras
        if (jugada.startsWith("bT-07-")) torreCortaMovidaNegras = true;
        if (jugada.startsWith("bT-00-")) torreLargaMovidaNegras = true;

        // Reyes
        if (jugada.startsWith("wR-74-")) reyMovidoBlancas = true;
        if (jugada.startsWith("bR-04-")) reyMovidoNegras = true;
    }

    if (!reyMovidoBlancas && !torreLargaMovidaBlancas) {
        String movimietoEnroque = enroqueLargo(casillas, ficha, pos);
        if (!movimietoEnroque.isEmpty())
            movimiento +=movimietoEnroque+" ";
    }
    else if (!reyMovidoNegras && !torreLargaMovidaNegras) {
        String movimietoEnroque = enroqueLargo(casillas, ficha, pos);
        if (!movimietoEnroque.isEmpty())
            movimiento +=movimietoEnroque+" ";
    }
    if (!reyMovidoBlancas && !torreCortaMovidaBlancas) {
        String movimietoEnroque = enroqueCorto(casillas, ficha, pos);
        if (!movimietoEnroque.isEmpty())
            movimiento +=movimietoEnroque+" ";
    }
    else if (!reyMovidoNegras && !torreCortaMovidaNegras) {
        String movimietoEnroque = enroqueCorto(casillas, ficha, pos);
        if (!movimietoEnroque.isEmpty())
            movimiento +=movimietoEnroque+" ";
    }
    return movimiento;

}
```

Figura 15: Comprobar si se puede Enrocar

Esta función comprueba si se ha movido las torres o el rey.
Para hacerlo comprueba en el historial de la partida si algún movimiento coincide con los regex.
Tras comprobarlo llama a las funciones correspondientes.

♖ Enroques ♖

Aquí se comprueba si las casillas del enroque están vacías y si en esas casillas el rey no está en jaque (esto se verá más adelante), tal y como en las reglas del ajedrez.

```
private static String enroqueCorto(JButton[][] casillas, String ficha, String pos) {
    int fila = ficha.contains("b") ? 0 : 7;
    int colActual = Integer.parseInt(pos.substring(1, 2));

    // 1. Verifica que las casillas intermedias estén vacías
    if (!casillas[fila][colActual + 1].getText().isEmpty() ||
        !casillas[fila][colActual + 2].getText().isEmpty()) {
        return ""; // No se puede enrocar
    }

    // 2. Verifica que el rey no pase por jaque en ninguna de las casillas involucradas
    for (int i = 0; i <= 2; i++) {
        String posTest = fila + "" + (colActual + i);
        String prueba = DetectarJaqueEnPartida.controlJugadasPorJaque(
            posTest, casillas, ficha, fila + "" + colActual, false
        );
        if (prueba.isEmpty()) {
            return "";
        }
    }


    // 3. Si todo está bien, devuelve la casilla final del enroque corto
    return fila + "" + (colActual + 2);
}
```

Figura 16: Dar movimiento en el enroque corto

```
private static String enroqueLargo(JButton[][] casillas, String ficha, String pos) {
    int fila = ficha.contains("b") ? 0 : 7;
    int colActual = Integer.parseInt(pos.substring(1, 2));
    // 1. Verifica que las casillas intermedias están vacías (colActual - 1, -2, -3)
    if (!casillas[fila][colActual - 1].getText().isEmpty() ||
        !casillas[fila][colActual - 2].getText().isEmpty() ||
        !casillas[fila][colActual - 3].getText().isEmpty()) {
        return ""; // No se puede enrocar largo
    }

    // 2. Verifica que el rey no pase por jaque en ninguna de las casillas involucradas (colActual, colActual - 1, colActual - 2)
    for (int i = 0; i <= 2; i++) {
        String posTest = fila + "" + (colActual - i);
        String prueba = DetectarJaqueEnPartida.controlJugadasPorJaque(
            posTest, casillas, ficha, fila + "" + colActual, false
        );
        if (prueba.isEmpty()) {
            return ""; // Si alguna casilla es insegura, no se puede enrocar largo
        }
    }

    // 3. Si todo está bien, devuelve la casilla final del enroque largo
    return fila + "" + (colActual - 2);
}
```

Figura 17: Dar movimiento en el enroque largo

♖ Jugadas de las Piezas ♖

En esta parte se explicaran los movimientos de las piezas, además de poner imágenes para que se pueda ver bien los movimientos

Caballo

El caballo tiene el movimiento más raro del ajedrez, su movimiento es una “L” desde su posición hasta las 8 posiciones en las que puedes hacer una “L”. Además de ser la única pieza que puede saltar piezas

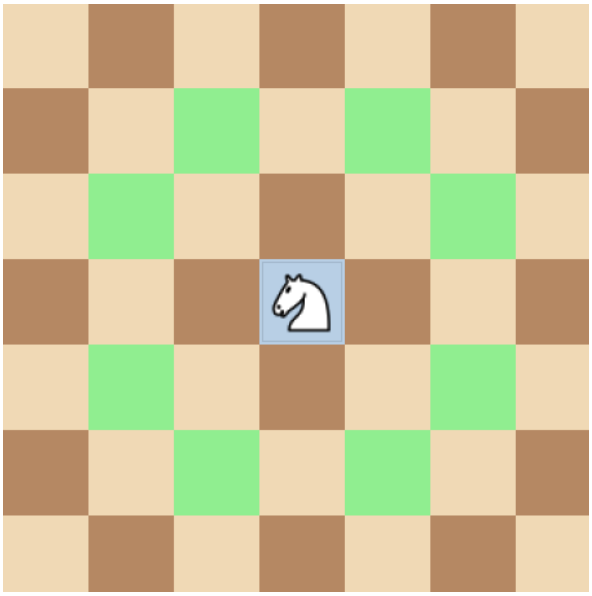


Figura 18: Movimientos del Caballo

Rey

El rey puede mover una casilla en todas las direcciones.

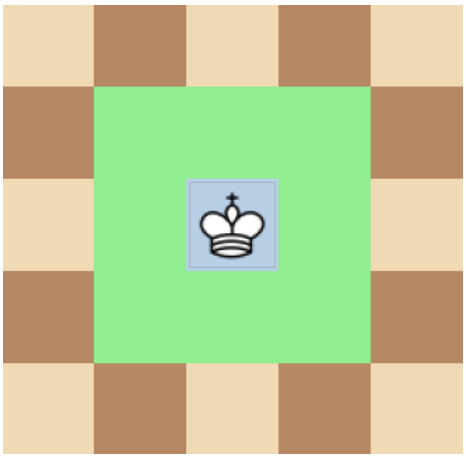


Figura 19: Movimientos normales del Rey

Dama

La Dame es parecida al rey, la diferencia es que la dama no se limita a una casilla, se puede mover todas las casillas hasta que se tope con el fin del tablero o una pieza.

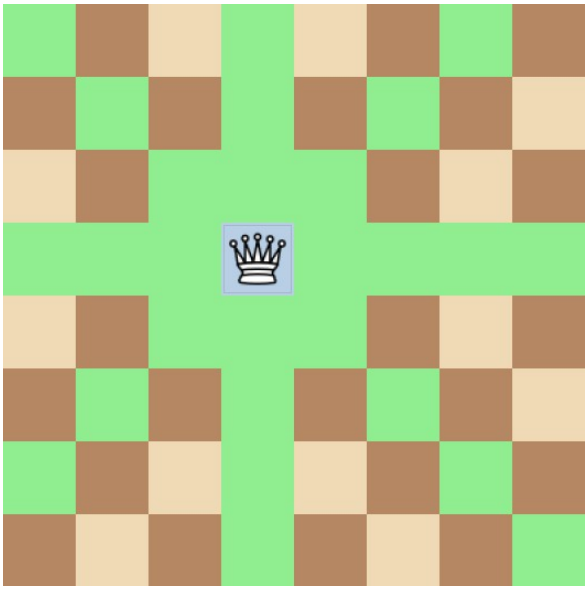


Figura 20: Movimientos de la Dama

Alfil

El alfil solo se pude mover en diagonal.

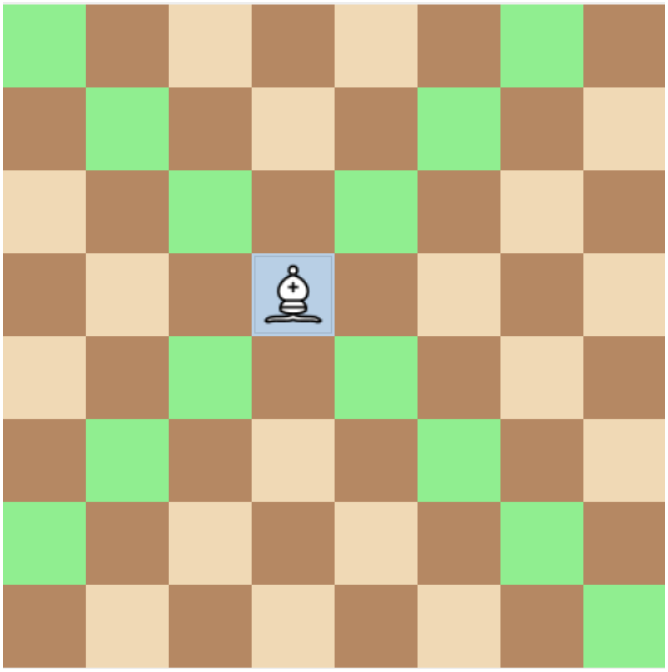


Figura 21: Movimientos del Alfil

Torre

Al contrarío que el alfil la torre solo se puede mover horizontalmente y verticalmente:

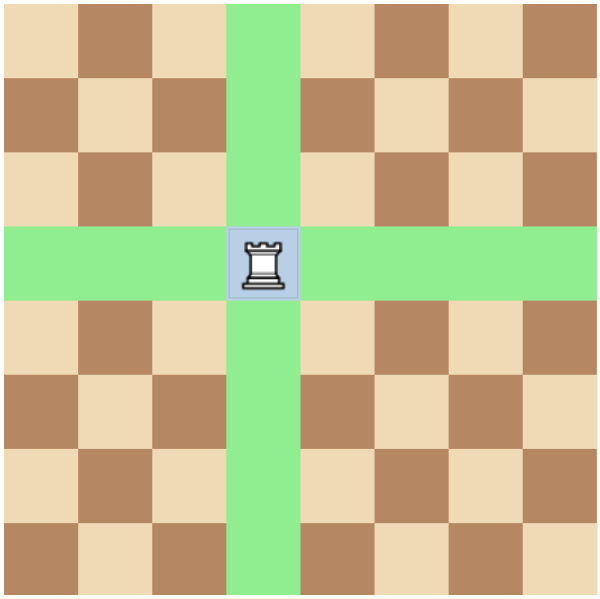


Figura 22: Movimientos de la Torre

Peón

Pese a ser simple el peón tiene varias reglas especiales, las cuales son:

- Mover dos veces en la salida

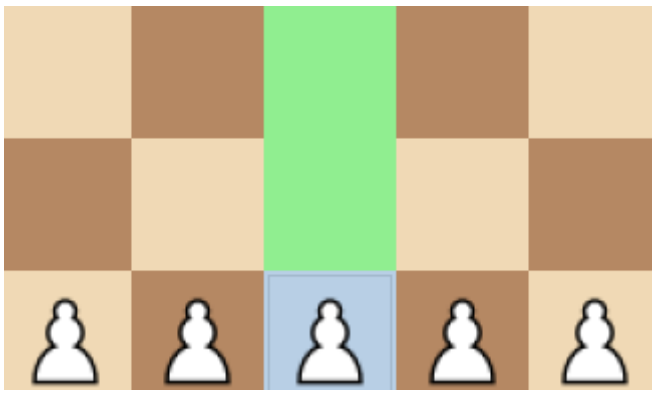


Figura 23: Movimientos salida del Peón

♟ Coronar en la última casilla a la que puede avanzar

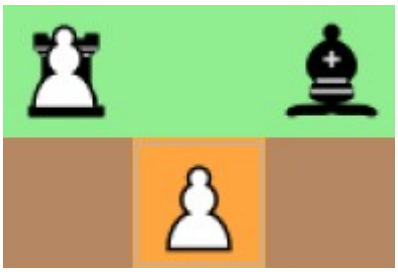


Figura 24: Movimiento Coronar Peón

♟ Comer al paso

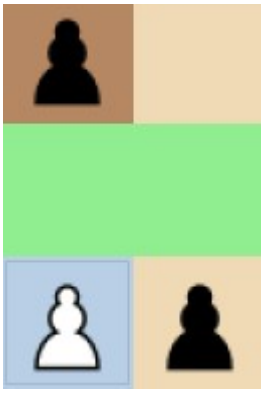


Figura 25: Movimiento comer al Paso Peón

En el siguiente apartado se verá como estos movimientos pueden cambiar dependiendo si son legales o no.

♚ Detectar Jaques ♚

Aquí se verá como se trata los jaques al rey. Además de ver como se filtran los movimientos que devuelven las piezas. Se filtran tras ver la posición futura del tablero y comprobar si el rey esta en jaque.

♕ Mirar si el Rey está en Jaque ♕

La función "*mirarReyEnJaque*" consigue el color del jugador y consigue la posición de su rey. Si detecta que su rey esta en una casilla roja (es como se marca que esta en jaque) devolverá false.

```
public static boolean mirarReyEnJaque(JButton[][] casillas, String color) {
    String colorPropio = color.substring(0, 1); // "w" o "b"
    String rey = colorPropio + "R";
    int reyFila = -1, reyCol = -1;

     // Recalcula jaques y colorea
    detertarPosicionJaque(casillas);

 // Busca la posición del propio rey tras el movimiento
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (casillas[i][j].getText().equals(rey)) {
                reyFila = i;
                reyCol = j;
            }
        }
    }


    if (reyFila != -1 && reyCol != -1) {
        Color bg = casillas[reyFila][reyCol].getBackground();
        if (bg != null && bg.equals(Colores.JAQUE_ROJO)) {
            return false;
        }
    }
    FuncionesVisualesTablero.resetFullColores(casillas);
return true;
}
```

Figura 26: Función Mirar Rey en Jaque

♕ Detectar Jaque en Posición ♕

Esta función pinta la casilla del rey en rojo. Para hacerlo consigue la posición de ambos reyes, y hace un bucle que mira toda la posición del tablero y si detecta una pieza consigue sus movimientos posibles y si uno de esos movimientos coincide con el rey rival su casilla se resalta de rojo.

```
public static void detertarPosicionJaque(JButton[][] casillas) {
    String posicionReyNegro = "";
    String posicionReyBlanco = "";

    // 1. Buscar la posición de ambos reyes
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (casillas[i][j].getText().equals("wR")) {
                posicionReyBlanco = i + "" + j;
            }
            if (casillas[i][j].getText().equals("bR")) {
                posicionReyNegro = i + "" + j;
            }
        }
    }
    boolean jaqueAlNegro = false;
    boolean jaqueAlBlanco = false;

    // Recorrer todas las piezas
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            String pieza = casillas[i][j].getText();
            if (pieza.equals("") || pieza.equals(null))
                continue; // Casilla vacía


            // Crear las piezas
            Piezas piezaObj = MetodosMoverPiezas.identificarFuncionPieza(pieza);
            if (piezaObj==null)
                return;
            String movimientos = piezaObj.calcularMovimientos(i + "" + j, casillas, pieza,false);
            String[] movimientosValidos = movimientos.split(" ");

            // Ver si alguno de los movimientos da jaque
            for (String movimiento : movimientosValidos) {
                if (movimiento.equals(posicionReyNegro)) {
                    // Jaque al negro
                    int fila = Integer.parseInt(posicionReyNegro.substring(0, 1));
                    int col = Integer.parseInt(posicionReyNegro.substring(1, 2));
                    casillas[fila][col].setBackground(Colores.JAQUE_ROJO);
                    jaqueAlNegro = true;
                }
                if (movimiento.equals(posicionReyBlanco)) {
                    // Jaque al blanco
                    int fila = Integer.parseInt(posicionReyBlanco.substring(0, 1));
                    int col = Integer.parseInt(posicionReyBlanco.substring(1, 2));
                    casillas[fila][col].setBackground(Colores.JAQUE_ROJO);
                    jaqueAlBlanco = true;
                }
            }
        }
    }
```

Figura 27: Detectar Posición en Jaque

Calcula las jugadas de los dos colores porque un rey puede mover una pieza de su mismo y que al mover esa pieza se quede en posición de jaque.

♕ Control de Jugadas Por Jaque ♕

Esta es la clase fusiona las clases explicadas anteriormente y hace un filtro de las jugadas de los movimientos de las piezas.
Tras pasarle los movimientos de la pieza esta función crear una copia del tablero con el movimiento de la pieza ya hecho, si ese movimiento hace que la función “detectarPosicionJaque” ilumine una casilla de rojo y la función “MirarReyEnJaque” detecte que hay una casilla esta en rojo ese movimiento se desecha y pasa al siguiente. Si el movimiento tras pasar por las funciones no genera casillas rojas el movimiento sera válido.

```
public static String controlJugadasPorJaque(String mov, JButton[][] casillas,String ficha,String posicionInicial, boolean visual) {
    String jugadasTotales="";
    // Divide el string por los espacios
    String[] movimientos = mov.split(" ");
    int filaInicial = Integer.parseInt(posicionInicial.substring(0, 1));
    int colInicial = Integer.parseInt(posicionInicial.substring(1, 2));

    for (String movimiento : movimientos) {
        if (movimiento.length() == 2) { // Asegúrarse de que el movimiento tiene dos caracteres
            int fila = Integer.parseInt(movimiento.substring(0, 1));
            int col = Integer.parseInt(movimiento.substring(1, 2));
            JButton[][] casillasCopia = crearTableroDePrueba(casillas,fila,col,ficha,filaInicial,colInicial);
            if (mirarReyEnJaque(casillasCopia,ficha)) {
                if (visual)
                    FuncionesVisualesTablero.resaltarCasilla(fila, col, casillas);
                jugadasTotales += fila + "" + col + " ";
            }
        }
    }
    return jugadasTotales;
}
```

Figura 28: Aplicar filtro de jugadas legales

Ejemplo visual de lo dicho anteriormente:

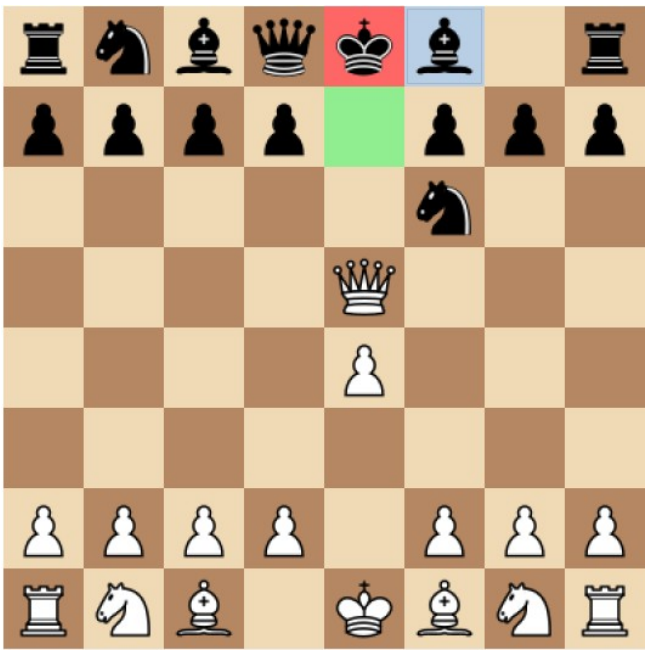


Figura 29: Ejemplo de Jaque y limitación movimientos

♕ Variables de la Partida ♕

Aquí se verá como se guardan las jugadas que los jugadores hacen, como se lleva el conteo de los movimientos que lleva la partida y como con ellos se puede saber a que jugadores le toca y como se el programa lleve la cuanta del tiempo de los jugadores.

♕ Cálculos en la Partida ♕

Esta clase lleva la cuenta de a que color le toca, los movimientos totales de la partida y llevar un conteo de todas las jugadas de la partida, guardando el movimiento como llave del HashMap, y en el valor guarda la posición original de la pieza, su posición final y la ficha que se ha movido(color y ficha).

```java
public class CalculosEnPartida {
    private static int jugadasTotales=0;
    public static HashMap<Integer, String> jugadas = new HashMap<Integer, String>();

    public static boolean colorAMover() {

        if (jugadasTotales%2==0) //Mueven blancas
            return true;
        else //Mueven negras
            return false;
    }
    public static void sumarMovimientos() {
        jugadasTotales++;
    }


    public static int getJugadasTotales() {
        return jugadasTotales;
    }


    public static HashMap<Integer, String> getJugadas() {
        return jugadas;
    }

    public static void guardarMovimientos(String posInicial, String posDestino, String ficha) {
        sumarMovimientos();
        String jugada=ficha+"-"+posInicial+"-"+posDestino; //Con un .split ya tengo un array del nº de movimiento, origen, final y ficha
        jugadas.put(jugadasTotales,jugada);
        System.out.println(jugadas);
    }
}
```

Figura 30: Ejemplo de los cálculos internos de la partida

♕ Tiempo ♕

Crea un hilo cuando el jugador de blancas hace el primer movimiento, que cada segundo le resta un segundo al jugador al que le toque y actualiza el tiempo restante del jugador, si la partida tiene incremento por jugada, ese incremento se sumará tras cada movimiento.

Luego actualiza el JLabel que contiene el tiempo (se verá más adelante).

```java
public class TiempoPartida {
    private volatile int tiempoBlancas;
    private volatile int tiempoNegras;
    private volatile boolean enPartida;
    private final JLabel label;
    private Thread hilo;

    public TiempoPartida(JLabel label, int minutosIniciales) {
        this.label = label;
        this.tiempoBlancas = minutosIniciales * 60;
        this.tiempoNegras = minutosIniciales * 60;
        this.enPartida = true;
    }

    public void iniciar() {
        hilo = new Thread(() -> {
            while (enPartida) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    break;
                }

                if (CalculosEnPartida.colorAMover()) {
                    if (tiempoBlancas > 0) tiempoBlancas--;
                } else {
                    if (tiempoNegras > 0) tiempoNegras--;
                }
                //Decorar el como se ve el tiempo
                SwingUtilities.invokeLater(() -> {
                    label.setText(
                        String.format("%02d:%02d-%02d:%02d",
                            tiempoBlancas / 60, tiempoBlancas % 60,
                            tiempoNegras / 60, tiempoNegras % 60
                        )
                    );
                });

                if (tiempoBlancas == 0 || tiempoNegras == 0) {
                    enPartida = false;
                    //Poner fin partida
                }
            }
        });
        hilo.start();
    }
```

Figura 31: Hilo del tiempo

Y para terminar el como se ha programado una partida de ajedrez simple queda el como se finaliza.

♚ Finalizar la Partida ♚

Aquí se mostraran las funciones que vigilan si la partida a acabado, ya sea por jaque mate, por ahogado o por que la posición se haya repetido tres veces.

```
public static void IdentificarFinPartida(JButton[][] casillas, HashMap<Integer, String> jugadas) {

    if (!MovimientosPosibles.tenerMovimientosPosibles(casillas, CalculosEnPartida.colorAMover())) {
        // Crea un Timer que espera 1000 ms (1 segundo) antes de ejecutar la acción
        if (DetectarJaqueEnPartida.mirarReyEnJaque(casillas,
                (CalculosEnPartida.colorAMover() ? "blanco" : "negro"))) {
            texto = "<html><b>¡Victoria!</b><br>El jugador "
                    + (CalculosEnPartida.colorAMover() ? "blanco" : "negro")
                    + " no tiene movimientos posibles.<br><i>Jaque mate.</i></html>";
        }

        else {
            texto = "<html><b>¡Tablas!</b><br>El jugador " + (CalculosEnPartida.colorAMover() ? "blanco" : "negro")
                    + " no tiene movimientos posibles.<br><i>Es tablas.</i></html>";
        }

    } else if (PosicionRepetida.posicionRepetidaTresVeces(jugadas)) {
        texto = "<html><b>¡Tablas!</b><br>La posicion se ha repetido tres veces<br><i>Es tablas.</i></html>";
    }
    if (!texto.isEmpty()) {
        mensajeTerminarPartida(texto, casillas,true);
        texto = "";
    }

}
```

Figura 32: Verificaciones de Fin de Partida

♚ Tablas por Repetición ♚

La “PosicionRepetida.posicionRepetidaTresVeces” mira si la posición actual del tablero se ha repetido dos veces en la partida si es así se termina la partida.
Para esto se crea un tablero lógico secundario que va recorriendo todos los movimientos de la partida, tras cada movimiento crea un String de la posición en ese momento, si esa posición es igual a la de la posición actual suma uno y si en otro momento se vuelve a repetir la posición son tablas. Si llega a la posición original sin que la posición se haya repetido tres veces el contador se reinicia.

♚ Jaque Mate ♚ - ♚ Rey Ahogado ♚

Para esto se hab tenido que usar las funciones “MovimientosPosibles.tenerMovimientosPosibles” y “DetectarJaqueEnPartida.mirarReyEnJaque”, estas funciones lo que hacen es ver si el jugador rival tiene algún movimiento legal por hacer tras ver que no tiene movimientos mira si el rey está en jaque si lo está es jaque mate si no es rey ahogado.

♕ Tiempo ♕

Para saber esto se llama a la función “comprobarFinPartida”. Aquí se comprueba si el reloj de algún jugador es igual a 0 si es así la partida termina.

```
private static void comprobarFinPartida() {
    if (tiempoBlancas == 0 || tiempoNegras == 0) {
        enPartida = false;
        String texto = "<html><b>¡Victoria!</b><br>El jugador "
                + (CalculosEnPartida.colorAMover() ? "blanco" : "negro")
                + " se le ha acabado el tiempo.<br><i>Caida de bandera.</i></html>";
        FinPartida.mensajeTerminarPartida(texto, casillas,true);
    }

}
```

Figura 33: Verificación fin de Tiempo

♕ Mensaje de fin Partida ♕

Esta es la base del mensaje que sale al terminar una partida, los cambios que hay son el texto que pone porque has ganado o perdido.

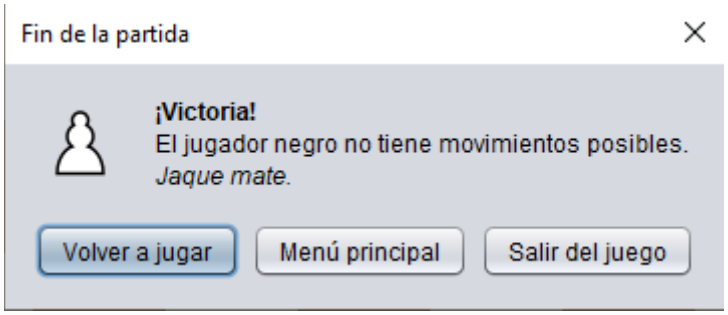


Figura 34: Mensaje Final de Partida

En la próxima página se hablará de como esta hecha la interfaz de usuario del tablero en el que se juegan las partidas, y como funciona la lógica que va actualizando los elementos que se muestran dentro de cada JPanel que hay al lado del tablero.

♚ Decoración Visual Tablero ♚

Aquí se va a mostrar como se ha hecho la decoración de la ventana en la que se disputa la partida de ajedrez. Solo aparece en los modos de jugar individual y jugar en LAN.

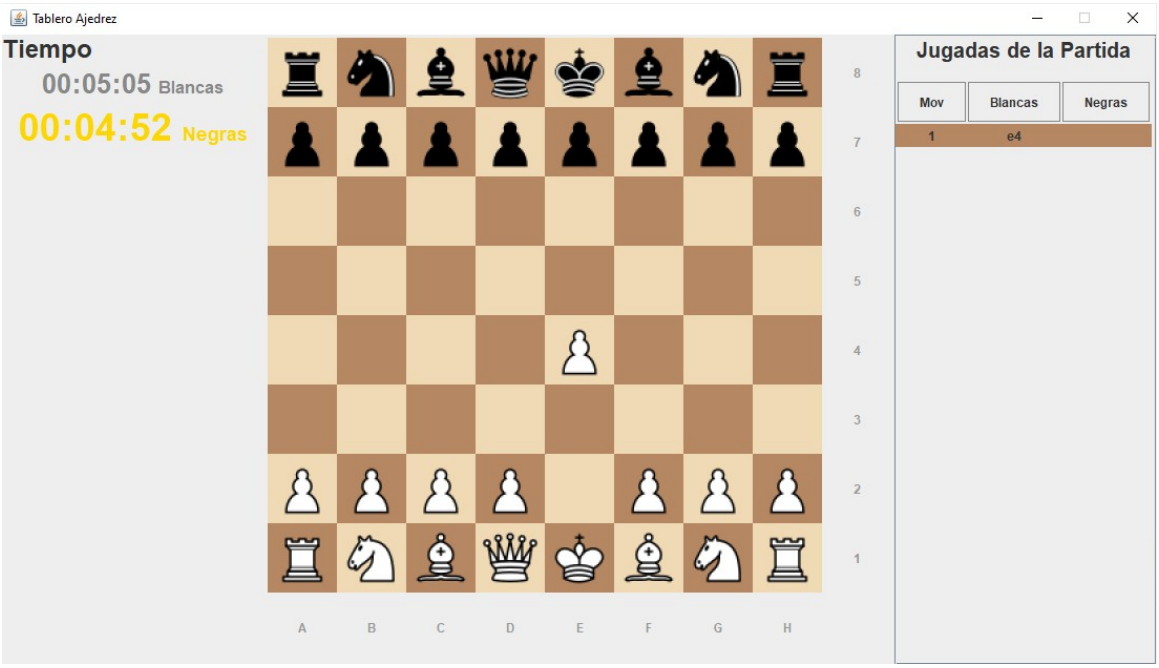


Figura 35: Visión tablero en Partida

♚Jugadas de la Partida ♚

En este apartado de la pantalla se muestran los movimientos tal y como se anotarían en una partida reglamentaria.

Para conseguirlo se ha hecho pasando los movimientos que el jugador a realizado, y llamando a la función “pasarJugadasParaGuardar”, consigue la ficha que se ha jugado, su origen y destino, tras esto se le añade las jugadas que no van ligadas a todas las fichas (enroque, jaques y coronación)

```java
private static void pasarJugadasParaGuardar(String ficha, String fichaOriginal, JButton[][] casillas,
        String destino, String origen, boolean hayPieza) {
    // Detecta si es JAque para poner un "+" en la lista de movimientos
    char colorContrario = ficha.charAt(0) == 'w' ? 'b' : 'w';
    String fichaContraria = "" + colorContrario; // Solo interesa el rey
    String destinoBonito="";
    destinoBonito=destino;

    if (!tenerJugadaEnroque(ficha, origen, destino).isEmpty())
        destinoBonito = tenerJugadaEnroque(ficha, origen, destino);
    if (fichaOriginal.contains("P") && !ficha.contains("P")) {
        destinoBonito = destino + "=" + ficha.substring(1, 2);
        ficha = fichaOriginal;
    }
    if (!DetectarJaqueEnPartida.mirarReyEnJaque(casillas, fichaContraria)) {
        destinoBonito += "+";
    }
    if (!MovimientosPosibles.tenerMovimientosPosibles(casillas, CalculosEnPartida.colorAMover())) {
        destinoBonito += "+";
    }
    generarMovimientosBonitosDesdeJugadas(destinoBonito, ficha,hayPieza, origen);
}
```

Figura 36: Jugadas Especiales a Guardar

Tras esto se llama a la función “`generarMovimientosBonitosDesdeJugadas`”:

```java
private static void generarMovimientosBonitosDesdeJugadas(String destino, String ficha,boolean hayPieza, String origen) {
    String jugada = "";

    // Convertir destino a notación algebraica (soporta coronación y jaque/mate
    String destinoAlgebraico = convertirACoordenadaAlgebraica(destino);

    if (ficha.substring(1, 2).equals("P")) {
        jugada = destinoAlgebraico;
    }else if (destino.contains("O-O") || destino.contains("O-O-O")) {
        jugada=destino;
    }else if(destino.contains("=")) {
        jugada = destino;
    }
    else {
        jugada = ficha.substring(1, 2) + destinoAlgebraico;
    }
    if(hayPieza) {
        jugada =ficha.substring(1, 2)+"x"+ destinoAlgebraico;
    }
    if(hayPieza && ficha.substring(1, 2).equals("P")) {
        int col = origen.charAt(1) - '0'; // 0 a 7 (a-h)

        char columnaLetra = (char) ('a' + col);
        jugada =columnaLetra+"x"+ destinoAlgebraico;
    }
    jugadasBonitas.put(CalculosEnPartida.getJugadasTotales(), jugada);
    //System.out.println(jugadasBonitas);

}
```

Figura 37: Generación de Jugadas Bonitas

Esta función toma la información interna de una jugada de ajedrez y la transforma en una notación algebraica estándar, teniendo en cuenta si es un movimiento normal, una captura, un enroque o una coronación, y la guarda para su posterior visualización.

♕ Tiempo ♕

El tiempo se inicia tras jugar el primer movimiento. Cada segundo que pasa se va actualizando el JLabel en el que esta, además se ilumina de amarillo el tiempo en el que se jugando, tras hacer un movimiento el tiempo cambia al del oponente el cual es ahora de color amarillo mientras que el tuyo cambia a negro con la letra más pequeña, dando un efecto visual que hace no dudar de a quien le toca.

```
private void actualizarLabel() {
    boolean blancasJuegan = CalculosEnPartida.colorAMover();

    String tiempoBlancasStr = tiempoVisual(tiempoBlancas);
    String tiempoNegrasStr = tiempoVisual(tiempoNegras);

    String tiempoBlancasHTML, tiempoNegrasHTML;

    if (blancasJuegan) {
        tiempoBlancasHTML = "<span style='color: #FFD700; font-size:28px; font-weight:bold;'>"
            + tiempoBlancasStr + " <small style='font-size:14px;'>Blancas</small></span>";
        tiempoNegrasHTML = "<span style='color: #888888; font-size:20px;'>"
            + tiempoNegrasStr + " <small style='font-size:12px;'>Negras</small></span>";
    } else {
        tiempoBlancasHTML = "<span style='color: #888888; font-size:20px;'>"
            + tiempoBlancasStr + " <small style='font-size:12px;'>Blancas</small></span>";
        tiempoNegrasHTML = "<span style='color: #FFD700; font-size:28px; font-weight:bold;'>"
            + tiempoNegrasStr + " <small style='font-size:14px;'>Negras</small></span>";
    }

    label.setText(
        "<html><div style='text-align:center;'>"
        + tiempoBlancasHTML + "<br>" + tiempoNegrasHTML +
        "</div></html>"
    );
}
```

Figura 38: Actualización del Tiempo

En la siguiente página se hablará de como se ha conseguido hacer que dos jugadores puedan jugar una partida a la vez desde diferentes ordenadores.

♚ Jugar en LAN ♚

Para conseguir esto se tiene que crear una sala y algún cliente que se quiera conectar y empezar a jugar.

♕ Creación de la Sala ♕

La sala se crea después de que el servidor configure datos de la partida, como son el color, tiempo e incremente.
Tras esto el programa crea un serverSocket con un puerto aleatorio (para evitar colisiones)

```
// Elegir un puerto TCP libre en el rango 10000-10999
int puertoTCP = 0;
while (jugando) {
    puertoTCP = 10000 + (int) (Math.random() * 1000);
    try {
        serverSocket = new ServerSocket(puertoTCP);
        break;
    } catch (IOException e) {
        // Puerto en uso, intenta otro
    }
}
```

Figura 39: Creación del ServerSocket

Después tiene que esperar con una conexión abierta para recibir mensajes broadcasts. Cuando el cliente mande un mensaje de “LIST_SALAS” todos los servidores que escuchan por este puerto le darán su información para que se puedan conectar.

```
// Hilo para responder broadcasts
new Thread(() -> {
    DatagramSocket broadcastSocket = null;
    try {
        broadcastSocket = new DatagramSocket(PUERTO_BROADCAST, InetAddress.getByName("0.0.0.0"));
        broadcastSocket.setBroadcast(true);
        byte[] buffer = new byte[1024];
        while (true) {
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
            broadcastSocket.receive(packet);
            String msg = new String(packet.getData(), 0, packet.getLength()).trim();

            if (msg.equals("LIST_SALAS")) {
                byte[] data = info.toString().getBytes();
                DatagramPacket respuesta = new DatagramPacket(data, data.length, packet.getAddress(),
                        packet.getPort());
                broadcastSocket.send(respuesta);
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (broadcastSocket != null && !broadcastSocket.isClosed()) {
            broadcastSocket.close();
        }
    }
}).start();
```

Figura 40: Escuchar por BroadCast

♕ Mensaje del Cliente ♕

El cliente manda el mensaje “LIST_SALAS” por toda la red y los servidores que le respondan le darán la información para que se pueda conectar.

```
DatagramSocket Datasocket = null;
try {
    Datasocket = new DatagramSocket();
    Datasocket.setBroadcast(true);
    byte[] mensaje = "LIST_SALAS".getBytes();

    DatagramPacket packet = new DatagramPacket(
        mensaje, mensaje.length,
        InetAddress.getByName("255.255.255.255"), PUERTO_BROADCAST
    );
    Datasocket.send(packet);

    Datasocket.setSoTimeout(2000);

    byte[] buffer = new byte[1024];
    while (true) {
        try {
            DatagramPacket respuesta = new DatagramPacket(buffer, buffer.length);
            Datasocket.receive(respuesta);

            String recibido = new String(respuesta.getData(), 0, respuesta.getLength()).trim();
            SalaInfo sala = SalaInfo.fromString(recibido);
            salas.add(sala);
            salaIPs.put(sala.puerto, respuesta.getAddress());
        } catch (SocketTimeoutException e) {
            break;
        }
```

Figura 41: Enviar mensaje por BoradCast

Tras esto el cliente ya tiene toda la información necesaria para poder crear la conexión con el servidor. Además de todos los datos necesarios para poder crear el tablero y poder jugar en igualdad de condiciones.

```
socket = new Socket(ipServidor, sala.puerto);
in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
out = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
```

Figura 42: Crear Socket

♕ Jugar la Partida ♕

Para jugar la partida ambos jugadores van cambiando entre escuchar y hablar entre ellos, de este modo cada uno puede puede jugar sus jugadas en su turno y en su tablero y que al otro le lleguen las jugadas.

♖ Hacer las jugadas ♖

Aquí se ve como se usa el outputStream para enviar las jugadas al contrincante. Además se hace una comprobación para ver si el oponente a dejado la partida.

```
   public void hacerJugadas(BufferedWriter out){
       try {
       out.write(MetodosMoverPiezas.datosDeMovimientos.getOrigen() + "\n");

       out.write(MetodosMoverPiezas.datosDeMovimientos.getDestino() + "\n");

       out.write(MetodosMoverPiezas.datosDeMovimientos.getFicha() + "\n");

       out.write(MetodosMoverPiezas.datosDeMovimientos.getMovimientos() + "\n");


       // out.write(MetodosMoverPiezas.sensorDeTurnosDosJugadores + "\n");
       out.flush();
   }catch(SocketException  a) {
       FinPartida.mensajeTerminarPartida("El oponente se ha retirado",Movimientos.getCasillas(),false) ;
   } catch(Exception e) {
       e.printStackTrace();
   }
   }
```

Figura 43: Función mandar jugadas

♖ Escuchar las Jugadas ♖

Aquí se ve como se escuchan las jugadas que el contrincante le manda, tras tener la jugada se manda a la función para mover las piezas.

```
public void escucharJugadas(BufferedReader in){
    try {
    FuncionesVisualesTablero.setVerCasillas(false);
    String origen = in.readLine();

    String destino = in.readLine();

    String ficha = in.readLine();

    String movimientos = in.readLine();
    MetodosMoverPiezas.moverPiezas(origen, destino, Movimientos.getCasillas(), ficha,
            movimientos,true,true,false);
    FuncionesVisualesTablero.resetColores(Movimientos.getCasillas());
    FuncionesVisualesTablero.setVerCasillas(true);

    }catch(SocketException  a) {
        FinPartida.mensajeTerminarPartida("El oponente se ha retirado",Movimientos.getCasillas(),false) ;
    } catch(Exception e) {
        e.printStackTrace();
    }
}
```

Figura 44: Función escuchar jugadas

♕ Interfaz ♕

♖ Crear la Sala ♖

Tras crear la sala se cambiarán los botones de la pantalla y se creará este spinner, dando un efecto óptico de estar esperando, cuando un usuario se conecte se cerrará este Jframe y se abrirá el del tablero con todos los datos que se han puesto antes.

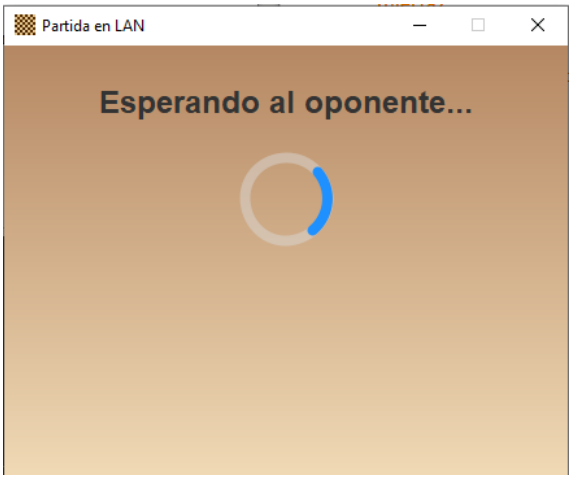


Figura 45: Esperar Oponente

♖ Unirse a Sala ♖

Cuando el usuario le de al botón de unirse, el programa cargará todas las respues dadas por los servidores y se le mostrarán en pantalla para que el usuario solo tenga que clickar a la que queiere unirse.

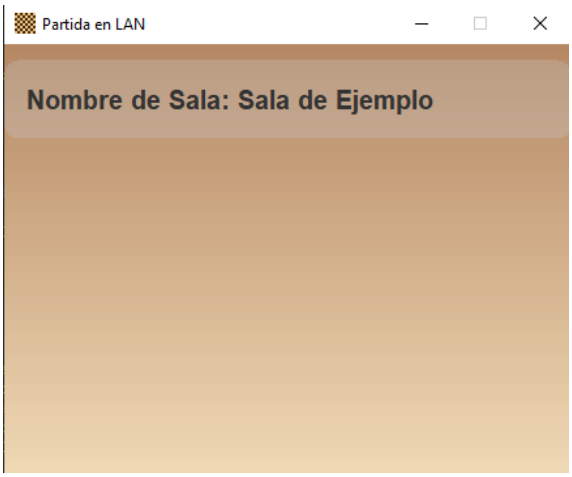


Figura 46: Unirse al Servidor

♖ Tableros ♖

♟ Tablero del cliente

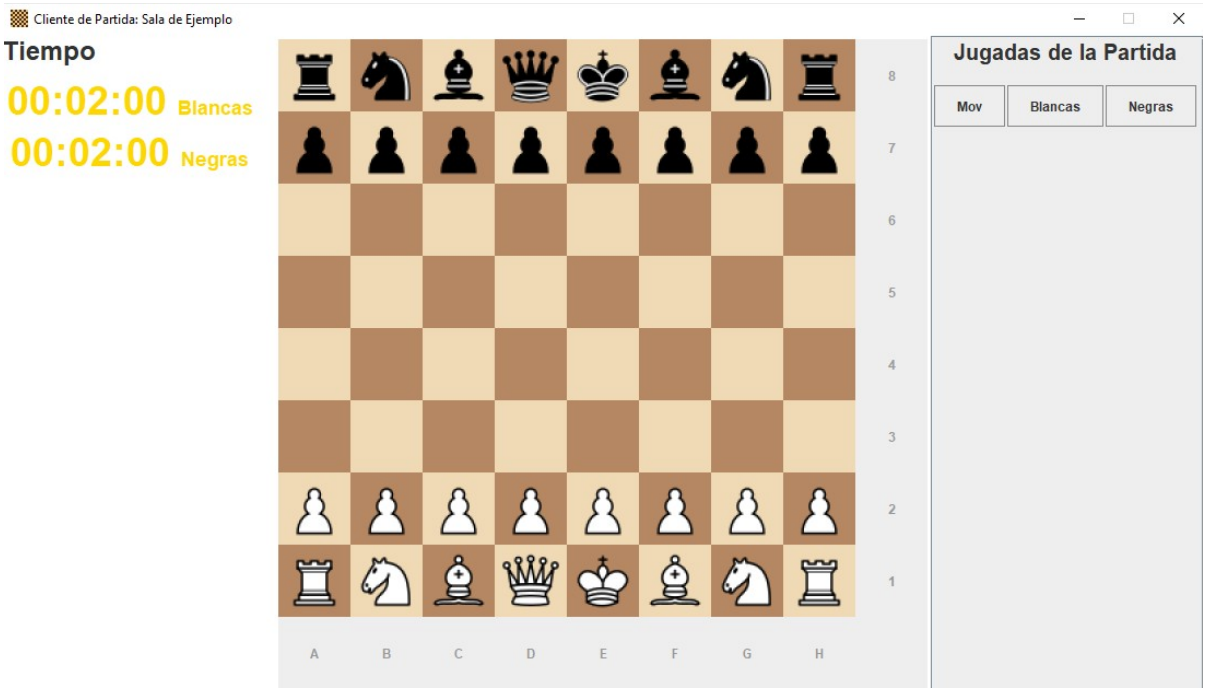


Figura 47: Tablero Cliente

♟ Tablero del servidor

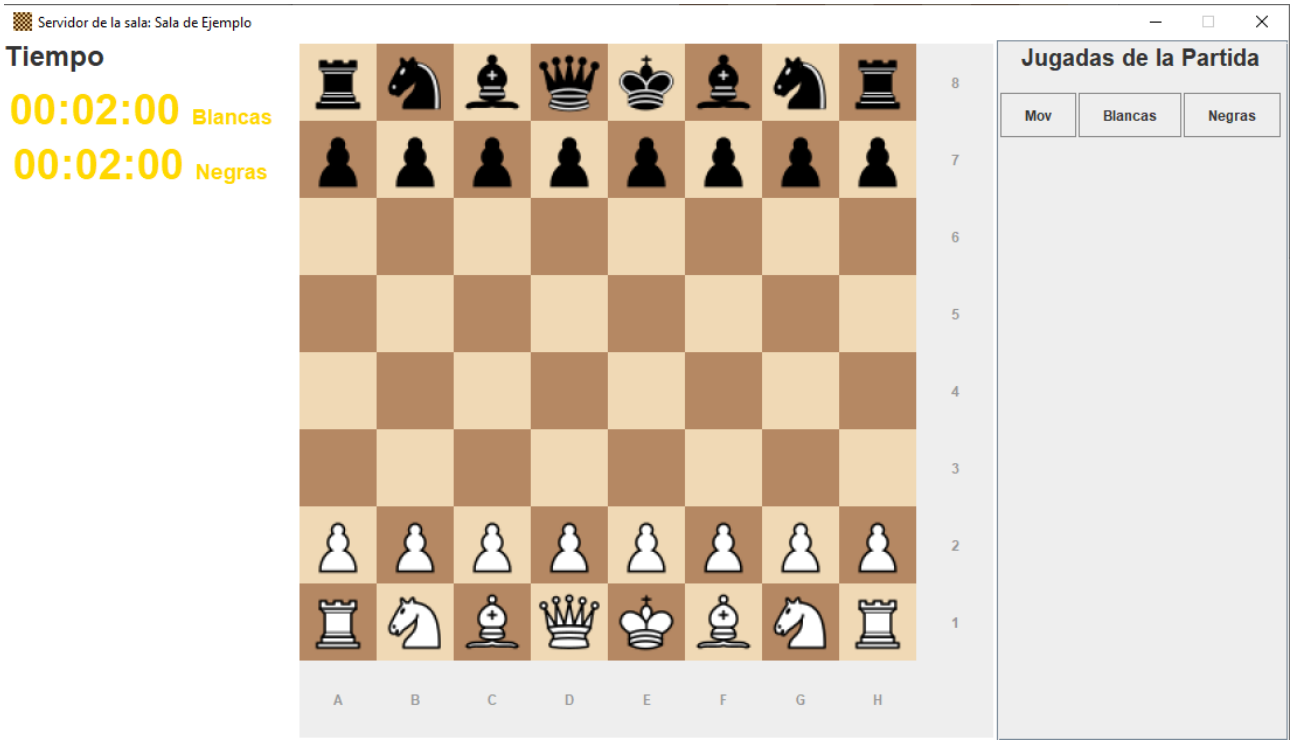


Figura 48: Tablero Servidor

♕ Problemas ♕

En esta parte del juego se explica como se guardan los problemas y los movimientos que hacen que puedas completarlo. Tras hacer el movimiento que se cree que es el correcto saldrá un mensaje de si se ha adivinado o no.

♕ ¿Donde se guardan? ♕

Los problemas se guardan en un JSON, y después la función “leerProblemasJSON” va posicionando las piezas en las posiciones que el JSON le marca y con la misma lógica con la que se crea un tablero normal se crea el problema (cambiando las posiciones de las piezas por las del JSON). Y para saber el movimiento correcto se usa la función “leerSolucionProblema”. Esta es la parte del JSON para el nivel 1.

```
{
  "nivel": 1,
  "piezas": [
      { "pieza": "wD", "casilla": "66" },
      { "pieza": "wR", "casilla": "50" },
      { "pieza": "bR", "casilla": "70" }    ]
},
```

Figura 49: Posición Problema

```
{ "nivel": 1, "movimiento": "61", "pieza": "wD" },
```

Figura 50: Movimiento Correcto Problema

♕ Detectar si la Jugada es Correcta ♕

Para esto se usa la función “problemaLogrado”, si el movimiento que el usuario a hecho es el correcto devolverá true si es incorrecto false.

```
public static boolean problemaLogrado(String movimiento, String pieza, int nivelActual) {
    LeerJSON lector = new LeerJSON();
    List<SolucionProblema> problemas = lector.leerSolucionProblema("/problemas/solucionProblema.json");
    for (SolucionProblema p : problemas) {
        if (p.getMovimiento().equals(movimiento) && p.getPieza().equals(pieza) && p.getNivel()==nivelActual) {
            return true;
        }
    }
    return false;
}
```

Figura 51: Ver si se ha logrado el problema

♕ Mensajes del Problema ♕

♖ Fallo ♖

Da tres opciones al usuario, volver a intentar el problema, volver al menú o salirse de la aplicación.

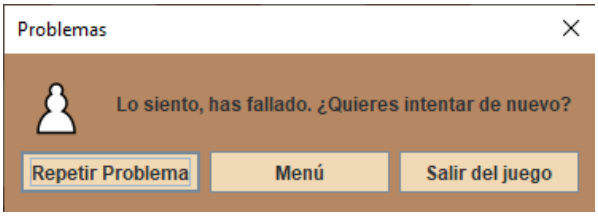


Figura 52: Problema no Conseguido

♖ Correcto ♖

Da tres opciones al usuario, pasar al siguiente problema, volver al menú o salirse de la aplicación.

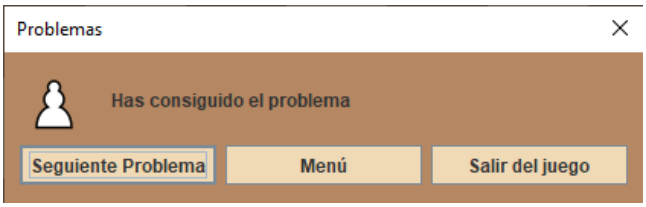


Figura 53: Problema Conseguido

♕ Interfaz ♕

Así es como se ve la pantalla de problemas

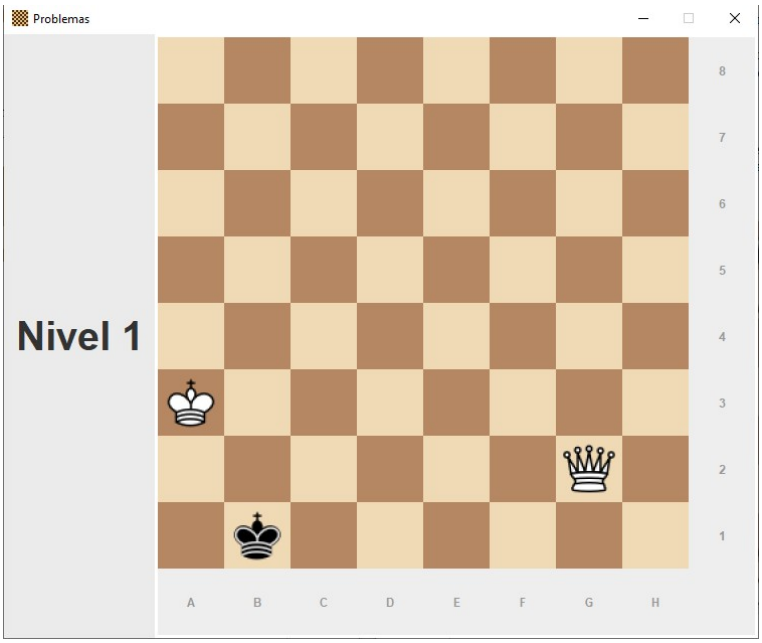


Figura 54: JFrame Problemas

♕ Bases De Datos ♕

En este apartado se abordará el tema de las bases de datos en este trabajo. El como se guardan los usuarios y las partidas. También habrá un archivo aparte que mostrará como poder instalar MySQL en Windows o Linux, y otro archivo .sql esencial para la creación de la base de datos.

♕ Inserción de Usuarios en MySQL ♕

Simplemente tienes que tener la base de datos activa en un ordenador y tener configurado el archivo para que el programa se pueda conectar y tras esto te vas a iniciar sesión

Figura 55: Inicio De Sesión-Regitrarse

Y creas una cuenta o inicias sesión.. Tras esto el programa avanzará a la pantalla de “Menú de Ajedrez” la novedad que hay por ahora al iniciar sesión es en el tema de guardar-cargar partida las cuales se explicarán a continuación.

Por privacidad a las paridas de los usuarios las contraseñas se guardan con un hash.

♚ Guardar-Cargar Partida ♚

Ahora se hablará de la última funcionalidad del proyecto, guardar las partidas que has jugado en la aplicación o pasarlas al programa en el modo de guardar partida para después irte a la parte de cargar partida y poder ver todas tus partidas guardas y poder revisarlas en la propia aplicación. A la hora de guardar las partidas hay dos formas diferentes para guardar:
- La primera es sin tener una cuenta iniciada, lo malo de esta forma es que todas las personas que quieran guardar una partida sin cuenta en un mismo ordenador se guardarán en un mismo archivo y no solo se tendrá las partidas de uno mismo, si no de todos los que guardes.
- La segunda es tener una cuenta iniciada, lo bueno es que las partidas están vinculas a tu usuario y solo uno mismo con su contraseña podrá tener acceso a las partidas, lo malo es que tienes que tener MySQL en tu ordenador e instalar la base de datos.

Ahora se verán las pantallas para guardar y cargar partida:

♕ Guardar Partida ♕

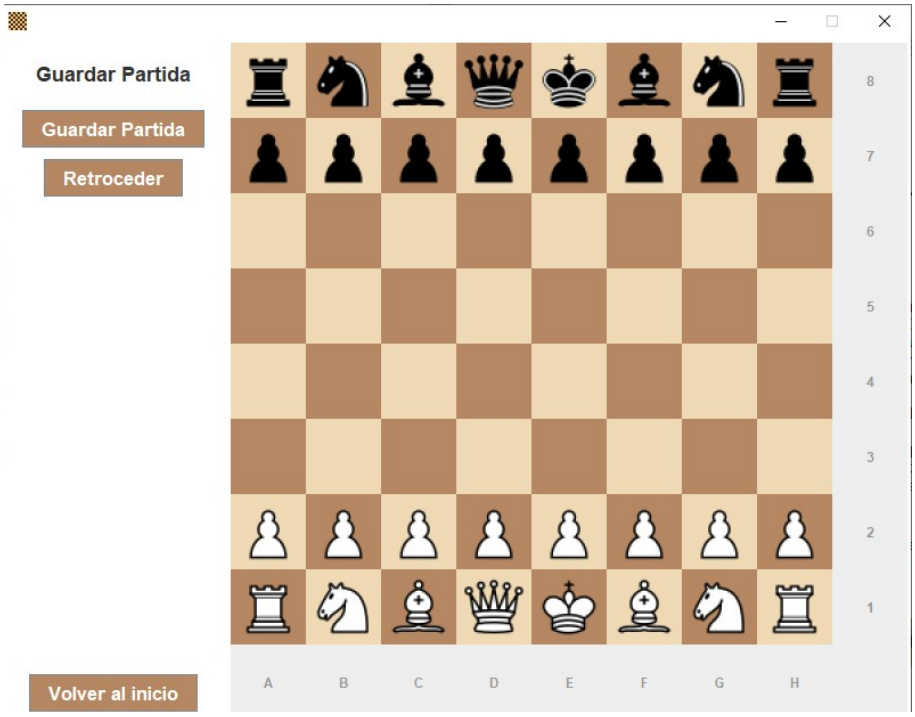


Figura 56: Interfaz de Guardar Partida

Como se ve es
un tablero normal con tres botones al lado izquierdo, uno para volver al inicio, otro para retroceder un movimiento (por si el usuario se ha equivocado guardando la partida) y el último para guardar la partida, tras darle se le tendrá que poner un nombre a la partida y de esta forma la partida se guardará o en un archivo o en la base de datos dependiendo si se tiene la cuanta iniciada o no.

♕ Cargar Partida ♕

En este apartado se mostrará como se pueden visualizar las partidas, lo primero que hay que hacer es darle al botón que aparece arriba a la derecha y elegir la partida que se quiera ver. Tras esto se activarán los botones que anteriormente estaban desactivados. Ahora se explicará que hace cada uno

- Cargar Parida: Volverá a dar la opción de cargar otra partida.
- Ir al Principio: Llamará a la misma función que se usa para crear el tablero inicial, así se puede volver a ver la partida desde el inicio.
- Avanzar: Mira el siguiente movimiento que se tiene guardado en el hashmap y lo ejecuta.
- Retroceder: Tiene la misma función que el Guardar Partida, y funciona con una combinación del botón de Volver al inicio y Avanzar, lo que hace es volver a la posición inicial y avanza todas las jugadas hasta que llega al movimiento anterior. Esto se hace porque en el hashmap en el que se guardan las jugadas solo se guarda la pieza, posición de origen y destino, no se guarda si había una pieza en la posición de destino.
- Ir al Final: Avanza todas las jugadas hasta el final de la partida.
-Volver al inicio: Te llevará a la pantalla anterior del programa.

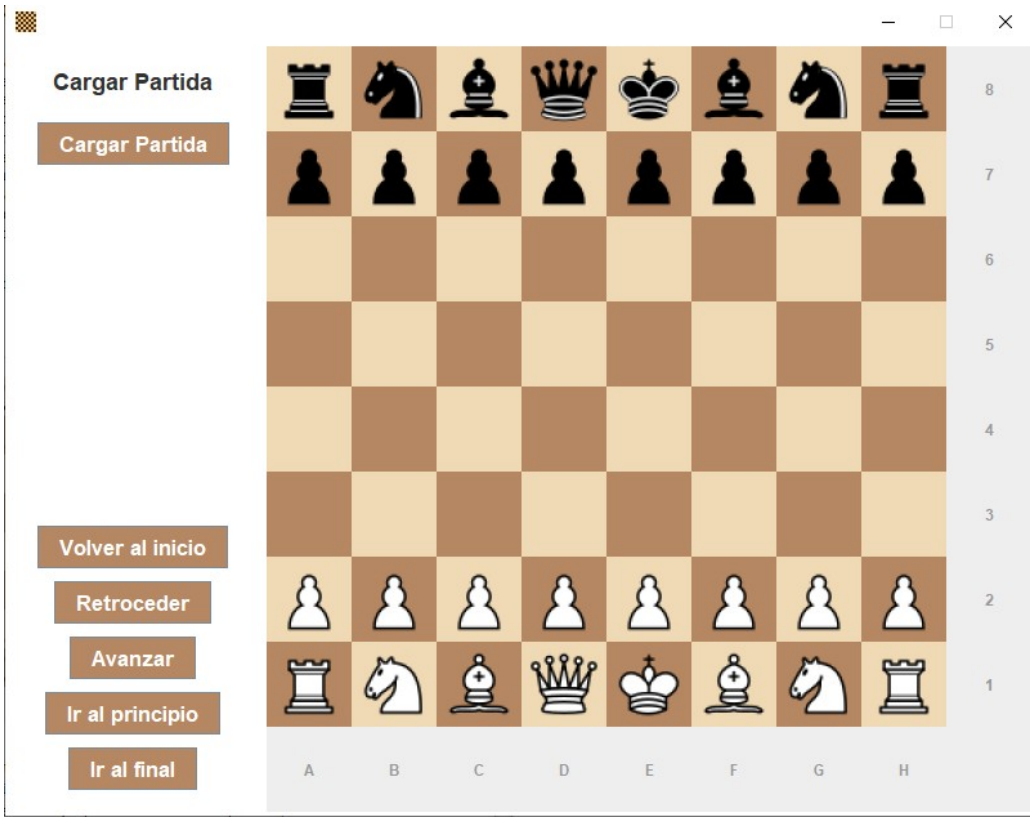


Figura 57: Interfaz de Cargar Partida

♚ Problemas en la Creación ♚

♕ Movimientos de las piezas ♕

Una de las partes más complejas del desarrollo ha sido la gestión de los movimientos y validaciones en el tablero. Durante este proceso, me encontré con numerosos tipos de errores, como por ejemplo: casillas nulas, casillas que, aunque no eran nulas, se comportaban como si no existieran, o la posibilidad de soltar una pieza en la misma casilla de origen, lo que provocaba la duplicación de la imagen de la pieza. También surgieron problemas al soltar piezas en la casilla correcta y que, aun así, se produjera una excepción, o al soltar piezas fuera del tablero sin que se generara ningún error, permitiendo que la pieza quedara en una posición inválida.

Otro fallo importante fue la posibilidad de mover piezas del color contrario durante mi turno, a pesar de que la detección de turnos funcionaba correctamente. Además, al pulsar dos veces sobre la misma pieza, se lanzaba una excepción inesperada. También detecté varios bugs visuales relacionados con las casillas donde era posible dejar una pieza. Finalmente, observé un comportamiento anómalo en el array que representa el tablero, especialmente al arrastrar piezas hacia la fila 1, lo que provocaba que el array funcionara de manera incorrecta.

♕ Peón ♕

Sin duda, el peón ha sido la pieza que más dificultades me ha presentado a la hora de programar sus movimientos. Los peones tienen un comportamiento muy particular que varía significativamente según su color, su posición en el tablero y la posición de otros peones. Esta complejidad provocó numerosos errores, especialmente cuando realizaba cambios en el comportamiento de otras piezas, lo que a menudo me obligó a rehacer la lógica de los peones en varias ocasiones.

♕ Rey ♕

El rey es la pieza con los movimientos más sencillos, pero su lógica es la más compleja de todo el juego. Sin duda, ha sido el mayor reto a la hora de implementar el sistema de movimientos de las piezas. La necesidad de comprobar constantemente si el rey está en jaque me obligó a replantear por completo la forma en que las piezas generan sus movimientos posibles. Para que el rey pudiera verificar si alguna pieza rival le daba jaque, fue necesario que esta comprobación se realizara desde las propias clases de las piezas. Si no lo hacía así, al intentar filtrar los movimientos del rey, se producía una llamada recursiva no deseada, ya que el rey dependía de los movimientos de las demás piezas y, a su vez, estas volvían a consultar al rey, generando así un bucle de recursividad.

♕ Jugar en LAN ♕

Este aspecto ha sido, sin duda, el segundo mayor foco de problemas durante el desarrollo. El hecho de tener que transmitir los movimientos a través del socket provocó que las clases responsables de procesar estos movimientos fallaran en el momento de escuchar las jugadas, lo que dificultaba identificar el origen exacto del error: ¿era un fallo de Windows, del cliente, del servidor o de la propia conexión de red?
Cuando finalmente conseguí que la transmisión de movimientos funcionara correctamente, surgieron nuevos retos, como la desincronización del tiempo entre ambos jugadores, el cierre incorrecto de los sockets (lo que provocaba que el contrincante detectara el cierre y mostrara una alerta de retirada tras terminar la partida), o que la sala permaneciera abierta tras finalizar la partida, permitiendo que otros jugadores se unieran y forzando al servidor a seguir gestionando la sala.
Estos han sido, sin duda, algunos de los problemas más complejos a los que me he enfrentado en el desarrollo del proyecto.

♕ Interfaces de la App ♕

Tuve que readaptarme al uso de las interfaces gráficas, pero lo más complicado de trabajar con ellas ha sido la gran cantidad de información que deben mostrar, proveniente de múltiples fuentes. Esto provocaba que, en ocasiones, alguna variable fuese nula justo en el momento en que la necesitaba, obligándome a replantear cómo y desde dónde obtener esos datos, a menudo con varias variables al mismo tiempo. Además, el hecho de que ciertos elementos de la interfaz cambien en función de si se ha realizado un movimiento, ha transcurrido un segundo, o el usuario ha salido y regresado al menú, hacía que la gestión de la interfaz resultara aún más confusa y compleja.

♕ Menciones honoríficas ♕

♖ La memoria del proyecto ♖

Lo increíblemente larga que ha quedado me ha quitado mucho tiempo y pensar que nunca iba a acabar, además si cambiaba algo en el código también lo tenía que cambiar en la memoria.

♖ El array bidimensional del tablero ♖

Ha habido muchos bugs relacionados con este array, como que no se podían jugar piezas en las las filas 0 y 7 y mantener la información de las casillas en constante actualización tras cada cosa que pasaba en el tablero.

♖ El hilo del tiempo ♖

Este hilo al ser independiente a casi todas las partes del programa ha sido bastante complicado reiniciarlo, hacer que actualice bien la pantalla y que al jugar en LAN funcione la más coordinado posible los dos jugadores.

♚ Bibliografía ♚

Stack Overflow. (2025). Para varios temas relacionados al proyecto RDurante varios días mientras se hacía el proyecto , de https://stackoverflow.com/questions
Github.. Chess pieces SVG set. Recuperado el 15 de abril de 2025, de las piezas visuales que se ven en el ajedrez, https://github.com/lichess-org/lila/tree/master/public/piece/cburnett
Perplexity AI. (2025). Perplexity [Modelo de lenguaje IA]. Durante varios días mientras se hacía el proyecto, de https://www.perplexity.ai
Sora. (2025). Logo generado para proyecto de ajedrez. Recuperado el 18 de mayo de 2025, de https://sora.chatgpt.com/explore
Github. Se ha usado para el control de versiones de la aplicación de https://github.com/javierg11/ChessGame

♚ Futuras Mejoras ♚

1. Poner un modo de jugar contra una I.A ya sea propia o con una API.
2. Hacer que las partidas entre los jugadores no sean solo en la misma LAN, si no que se pueda jugar cada uno desde su casa.
3. Que la gestión de partidas guardadas no dependa unicamente de una base de datos local, que puedas acceder a estas partidas desde cualquier lado.
4. Crear una aplicación para Android.
5. Un sistema de ELO.

♚ Conclusión ♚

Ha sido todo un reto poder llevar a cabo este proyecto. Al principio pensaba que no iba a ser tan complicado y que podría permitirme tener el código algo desordenado, ya que creía que no sería necesario mantener un orden muy riguroso. Sin embargo, estaba completamente equivocado. La complejidad ha sido extrema, especialmente al tener que coordinar todas las piezas, tanto a nivel lógico como visual, respetando todas las restricciones propias de cada una, las del tablero y, especialmente, las del rey para evitar cualquier movimiento ilegal. Sin ninguna duda, este es el código más largo y complejo que he realizado hasta ahora.

También sé que puede parecer que me he pasado haciendo la memoria con una cantidad de 44 páginas, pero como he explicado antes, gran parte del código es lógica de programación y la única forma para que quede demostrado todo lo que me ha costado hacer el programa ha sido ir poniendo las cosas más importantes en el código, además de las bases que han hecho que todas las clases puedan llegar a coordinarse bien.