



CHECKERS

PRÁCTICA 2

Nataly Yhislaine Jaya Salazar

GPraLab4

24/04/2024

UNIVERSIDAD DE LLEIDA

Escola Politècnica Superior

Grado en Ingeniería Informática

Programación II

ÍNDICE

Apartado 1: Position	1
Apartado 2: Direction	3
Apartado 3: Cell.....	4
Apartado 4: Board	5
Apartado 5: Geometry.....	8
Apartado 6: Game	9
Conclusiones	14
Referencias	15
Anexo de Imágenes	16

APARTADO 1: POSITION

CLASE PLAYER

En esta clase no se debían realizar modificaciones ya que se trataba de un código implementada que está basada en el Enum Types¹ [1].

CLASE PLAYER

Esta clase representa las posiciones en un tablero de juego identificadas por sus coordenadas X e Y. Es inmutable, lo que significa que sus valores no pueden ser alterados una vez establecidos. Las coordenadas X e Y están definidas con el modificador *final*, y una vez inicializadas en el constructor, no pueden ser modificadas.

En general, esta clase ha sido relativamente sencillo. La función del constructor y las funciones de obtención fueron fáciles de implementar. Para otras funciones, inicialmente creaba una variable donde realizaba la operación y luego la retornaba. Sin embargo, tras considerarlo más detenidamente, me di cuenta de que era más eficiente retornar la variable directamente para reducir el código.

```
public Position(int x, int y) {  
    //throw new UnsupportedOperationException("TODO: Step1");  
    this.x = x;  
    this.y = y;  
}  
  
public int getX() {  
    //throw new UnsupportedOperationException("TODO: Step1");  
    return this.x;  
}  
  
public int getY() {  
    //throw new UnsupportedOperationException("TODO: Step1");  
    return this.y;  
}
```

Además, las clases *distance* y *middle* requerían de la clase Math [2], y descubrí un nuevo método llamado *floorDiv* que utilicé. Aunque la función *sameDiagonalAs* necesitaba un nivel más de abstracción, pude identificar un patrón y derivar el código en consecuencia.

```
public static int distance(Position pos1, Position pos2) {  
    //throw new UnsupportedOperationException("TODO: Step1");  
    int distanceX = Math.abs(pos2.x - pos1.x);  
    int distanceY = Math.abs(pos2.y - pos1.y);  
  
    return distanceX + distanceY;  
}  
  
public static int distance(Position pos1, Position pos2) {  
    //throw new UnsupportedOperationException("TODO: Step1");  
    return Math.abs(pos1.getX() - pos2.getX()) + Math.abs(pos1.getY() - pos2.getY());  
}
```

¹ se utiliza para definir un tipo enumerado que representa una clase con exactamente dos instancias únicas, las cuales son predefinidas internamente y accesibles públicamente. Este patrón combina las características de un tipo enumerado (enum) con el comportamiento de un Singleton, asegurando que solo existan dos instancias específicas de la clase.

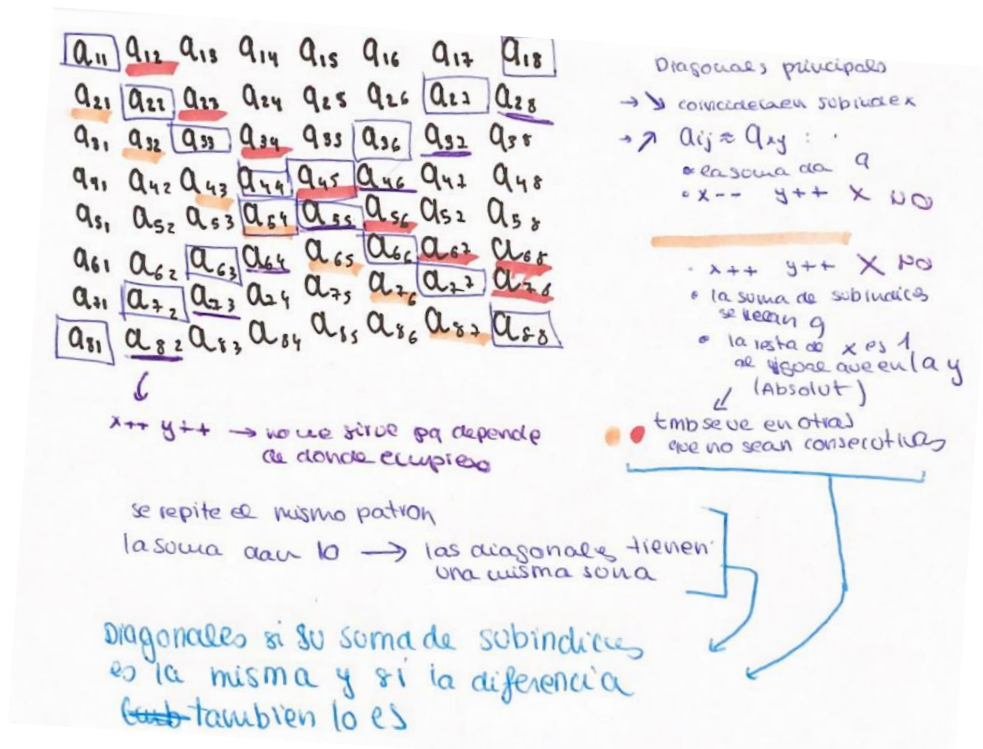
```
public static Position middle(Position pos1, Position pos2) {
    // throw new UnsupportedOperationException("TODO: Step1");
    int middleX = (pos1.x + pos2.x) / 2;
    int middleY = (pos1.y + pos2.y) / 2;

    return new Position(middleX, middleY);
}
```

```
public static Position middle(Position pos1, Position pos2) {
    return new Position(Math.floorDiv(pos1.x + pos2.x, 2), Math.floorDiv(pos1.y + pos2.y, 2));
}
```

Durante mi análisis inicial de la función, pensé en un principio que estaba diseñada para comparar los ejes y asegurarme de que pertenecieran a la misma diagonal. Sin embargo, tras examinar más de cerca los *tests*, me di cuenta de que la función tenía requisitos más complejos. Específicamente, era necesario analizar diagonales ubicadas tanto por encima como por debajo de las diagonales principales y no necesariamente dos elementos consecutivos.

Para abordar este desafío, desarrollé un esquema matricial que asignaba índices únicos a cada diagonal, lo que me permitió identificar patrones en los datos y consolidar la expresión en un solo código. Al implementar este nuevo enfoque, logré alcanzar el resultado deseado y asegurar que la función operé de manera efectiva y eficiente. El procedimiento se puede evidenciar en el siguiente esbozo ls que realicé al momento de proceder en la búsqueda de patrones.



```
public boolean sameDiagonals(Position other) {
    // throw new UnsupportedOperationException("TODO: Step1");
    return (this.x + this.y == other.getX() + other.getY()) || (this.x - this.y == other.getX() - other.getY());
}
```

APARTADO 2: DIRECTION

Esta clase define las direcciones para las fichas del tablero de juego: NW y NE para las blancas, SW y SE para las negras. Incluye un constructor privado, atributos públicos, estáticos y finales, además de un array de direcciones. El método *apply* transforma posiciones basándose en las modificaciones almacenadas.

El transcurso de esta clase se centró en una función constructora y la función *apply*, las cuales no fueron particularmente complejas. En las etapas iniciales, definí variables y luego las retorné, pero más tarde perfeccioné el proceso al retornarlas directamente.

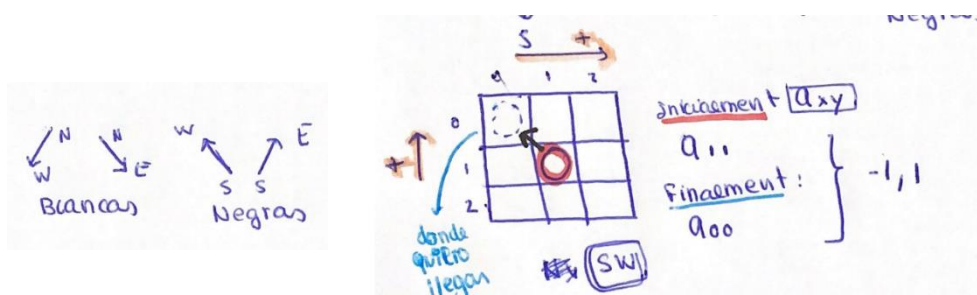
```
private Direction(int dx, int dy) {
    //throw new UnsupportedOperationException("TODO: Step2");
    this.dx = dx;
    this.dy = dy;
}
```

```
public Position apply(Position from) {
    //throw new UnsupportedOperationException("TODO: Step2");
    int newX = Math.abs(from.getX() + dx);
    int newY = Math.abs(from.getY() + dy);
    return new Position(newX, newY);
}
```

```
public Position apply(Position from) {
    //throw new UnsupportedOperationException("TODO: Step2");
    return new Position(Math.abs(from.getX() + dx), Math.abs(from.getY() + dy));
}
```

Además, desarrollé un enfoque sistemático para comprender el comportamiento de los índices al definir las direcciones, y a partir de ahí, derivé una solución. A continuación, incluyo el esquema para la dirección NW, que fue formulado utilizando la misma metodología aplicada a las otras direcciones.

```
public static final Direction NW = new Direction( dx: -1, dy: -1);
2 usages
public static final Direction NE = new Direction( dx: 1, dy: -1);
2 usages
public static final Direction SW = new Direction( dx: -1, dy: 1);
2 usages
public static final Direction SE = new Direction( dx: +1, dy: 1);
```



APARTADO 3: CELL

Esta clase representa cada celda del tablero de juego. Cada celda puede ser prohibida, vacía, blanca o negra. Las celdas pueden cambiar de estado entre estar ocupadas y vacías según los movimientos y capturas realizados durante la partida. Habrán solo cuatro instancias dentro de la clase y definiremos un constructor privado.

La clase Cell presentó una complejidad mínima, involucrando principalmente la devolución de variables definidas² y la implementación de una sola función con condicionales.

```
private Cell(char status) {  
    //throw new UnsupportedOperationException("TODO: Step3");  
    this.status = status;  
}
```

```
public boolean isForbidden() {  
    //throw new UnsupportedOperationException("TODO: Step3");  
    return this.status == C_FORBIDDEN;  
}
```

Es notable que inicialmente, la estructura de las declaraciones *if-else* en el código no me satisfizo. Como resultado, realicé una exploración de metodologías alternativas, encontrando finalmente la estructura *switch-case* [3]. Sin embargo, debido a la falta de experiencia directa en la implementación junto con un conocimiento limitado sobre su uso, opté por no emplear esta orientación.

```
public static Cell fromChar(char status) {  
    //throw new UnsupportedOperationException("TODO: Step3");  
    if (status == C_FORBIDDEN) return FORBIDDEN;  
    else if ( status == C_EMPTY) return EMPTY;  
    else if (status == C_WHITE) return WHITE;  
    else if (status == C_BLACK) return BLACK;  
    else return null;  
}
```

² En este caso solo mostraré la función *isForbidden* ya que esa misma estructura es para el *isEmpty*, *isWhite* y *isBlack*

APARTADO 4: BOARD

La clase *Board* es crucial en el juego, ya que representa el tablero de juego. Tiene atributos para las dimensiones del tablero, las celdas y contadores de fichas blancas y negras.

Para inicializar el tablero, se utiliza un *string* donde cada línea corresponde a los caracteres de las celdas del tablero. Es importante asegurarse de que la cadena coincida correctamente con las dimensiones del tablero para inicializarlo correctamente.

Tras la finalización de la clase y la aprobación de todas los *tests* asociados, se creyó que se habían implementado todas las modificaciones necesarias. Sin embargo, durante el desarrollo de la clase *Game*, quedó claro que se requerían más alteraciones en el *Board*. Se hizo evidente que simplemente funcionar correctamente no garantizaba que la clase lograra todo lo que debía.

En la función *Board*, se llevó a cabo una revisión del proceso de *tets* según lo indicado en el enunciado, que señalaba que la construcción de un tablero se establecía utilizando un parámetro *string*. En consecuencia, determiné que la forma más efectiva de manejar el parámetro pasado a la función. Durante este proceso, se observó que caracteres específicos, como "\n", debían ser excluidos. La solución final fue utilizar un *StringTokenizer*³ ya que también permitía facilitar el conteo del número de piezas.

```
public Board(int width, int height, String board) {
    //throw new UnsupportedOperationException("TODO: Step4");
    this.width = width;
    this.height = height;
    this.numBlacks = 0;
    this.numWhites = 0;
    cells = new Cell[height][width];

    structureBoard(board);
}
```

Es significativo destacar que inicialmente se utilizaron dos bucles *for* para iterar a través de la matriz *cells*. Sin embargo, por simplicidad, se tomó la decisión aprovechar un bucle *while* para recorrer la matriz con un solo bucle *for* [4]. Si bien esta decisión puede no haber sido revolucionaria, me hizo salir de mi zona de confort y facilitó un proceso más simple.

```
private void structureBoard(String board) {
    StringTokenizer token = new StringTokenizer(board, delim: "\n");
    int j = 0;

    while (token.hasMoreTokens()) {
        String item = token.nextToken();
        for (int i = 0; i < width; i++) {
            cells[j][i] = Cell.fromChar(item.charAt(i));

            if (Cell.fromChar(item.charAt(i)) == Cell.WHITE) numWhites++;
            else if (Cell.fromChar(item.charAt(i)) == Cell.BLACK) numBlacks++;
        }
        j++;
    }
}
```

```
".b.b.b.b\n" +
"b.b.b.b.\n" +
".b.b.b.b\n" +
" . . . \n" +
". . . . \n" +
"W.W.W.W.\n" +
".W.W.W.W\n" +
"W.W.W.W.";
```

³ Prefiero realizar esta acción en una función auxiliar para mejorar la legibilidad del código y distinguir entre la acción del constructor de la clase *Board* y la función de estructuración de la tabla.

Las funciones presentadas consistieron en *getters*, que se ejecutaron utilizando un proceso consistente.

```
public int getWidth() {  
    //throw new UnsupportedOperationException("TODO: Step4");  
    return this.width;  
}
```

En un esfuerzo por ayudar a la clase *Game*, se realizaron modificaciones en las funciones *isForbidden* y los *setters*. Por ejemplo, al agregar un booleano para indicar si está fuera de rango (*outOfBounds*⁴) en la función *isForbidden*, asegurando así que la función cumpliera un doble propósito. Fue diseñada para evitar que las piezas se colocaran en una posición prohibida o fuera del rango permitido, lo que también podría considerarse una posición prohibida.

```
public boolean isForbidden(Position pos) {  
    //throw new UnsupportedOperationException("TODO: Step4");  
    int posX = pos.getX();  
    int posY = pos.getY();  
  
    boolean outOfBounds = (posX < 0 || posX >= getWidth() || posY < 0 || posY >= getHeight());  
  
    boolean cellForbidden = !outOfBounds && cells[posY][posX].isForbidden();  
  
    return outOfBounds || cellForbidden;  
}
```

```
private boolean outOfBounds ( Position pos) {  
    return posX < 0 ||  
           posY < 0 ||  
           posX >= getWidth() ||  
           posY >= getHeight();  
}
```

Más adelante, durante la ejecución de las funciones *isBlack*, *isWhite* y *isEmpty*, se realizó la declaración correspondiente y una llamada a *isForbidden* que era opuesta a la función original⁵.

```
public boolean isBlack(Position pos) {  
    //throw new UnsupportedOperationException("TODO: Step4");  
    return (!isForbidden(pos)) && cells[pos.getY()][pos.getX()].isBlack();  
}
```

Aunque parecía que los *setters* simplemente aplicaban el cambio solicitado, me di cuenta que eran útiles para contar y gestionar el número de piezas. En la clase *Game*, era necesario acceder a este recuento. Sin embargo, no era posible realizar modificaciones, ya que una función pública con este propósito no era válida. Después de una cuidadosa consideración, determiné que una función complementaria debería ser privada, de acuerdo con las reglas.

```
public void setBlack(Position pos) {  
    //throw new UnsupportedOperationException("TODO: Step4");  
    cells[pos.getY()][pos.getX()] = Cell.BLACK;  
}
```

⁴ Inicialmente iba a consistir en una función aparte, pero decidí unirla a la función *isForbidden* porque considere que una celda fuera del rango también puede gestionarse como celda prohibida

⁵ Esto es debido a que se buscaba una celda válida, es decir, que no esté prohibida

MÉTODO INVÁLIDO

A pesar de la validez funcional de la función mencionada anteriormente, se consideró incorrecta porque no cumplía con las especificaciones predefinidas de la función descritas. En consecuencia, no debería haber sido clasificada como una función pública en su momento.

```
public void capture(Player player) {  
    if (player == Player.BLACK) {  
        this.numWhites--;  
    } else if (player == Player.WHITE) {  
        this.numBlacks--;  
    }  
}
```

MÉTODO VÁLIDO

```
private void positionEdited(Position pos){  
    if(isWhite(pos))numWhites--;  
    else if(isBlack(pos))numBlacks--;  
}
```

```
public void setEmpty(Position pos) {  
    //throw new UnsupportedOperationException("TODO: Step4");  
    positionEdited(pos);  
    cells[pos.getY()][pos.getX()] = Cell.EMPTY;  
}
```

Al vaciar la casilla,
decremento el
número de fichas
correspondientes

```
public void setWhite(Position pos) {  
    //throw new UnsupportedOperationException("TODO: Step4");  
    positionEdited(pos);  
    cells[pos.getY()][pos.getX()] = Cell.WHITE;  
    numWhites++;  
}
```

Al pintar una casilla, el
número de fichas se
mantiene estable, ya que
se decrementa e
incrementa

APARTADO 5: GEOMETRY

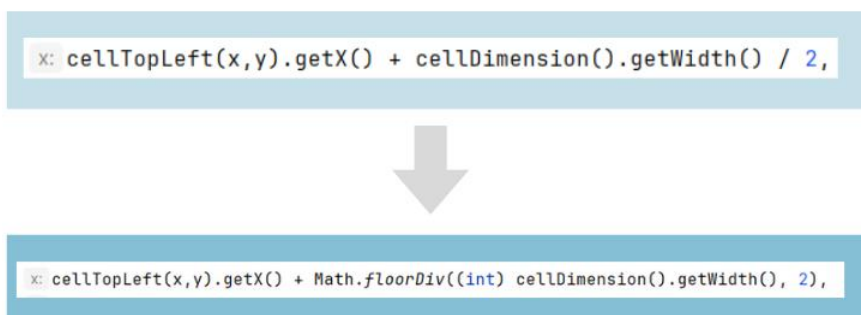
La clase *Geometry* utiliza las clases *GPoint* y *GDimension* de la biblioteca ACM para calcular las posiciones y dimensiones de elementos gráficos en la interfaz. *GPoint* representa coordenadas en la pantalla, y *GDimension* representa dimensiones de elementos, ambas con métodos *getters* que devuelven sus valores respectivos como números decimales (*doubles*).

Me encontré con esta clase relativamente más compleja que las anteriores, lo que me obligó a depender de los diagramas de la práctica para comprender los pasos necesarios para lograr el resultado deseado. Sin embargo, lo más esencial fueron los *tests* que resultaron ser fundamentales, ya que tuve dificultades para comprender y formular el código de manera independiente.

Las funciones iniciales que se trataban de un constructor y *setters*, no resultaron un gran desafío. A medida que avanzaba, me encontré con nuevos retos en los cuales pude aprovechar el conocimiento adquirido y reimplementar un nuevo método de la clase *Math*, aunque requería de un *cast* a enteros.

En el momento del uso del *padding*, este me permitió obtener un entendimiento más profundo del concepto. Había encontrado este concepto anteriormente en el código del Problema 2 [5], el cual lo revisé para saber cómo operarlo debidamente.

Por otro lado, creía que estaba enfrentando problemas de redondeo al tratar con números decimales, pero después de consultar con compañeros, me di cuenta de que no era el caso, ya que las operaciones arrojaban números exactos. A pesar del contratiempo, perseveré a través de un largo proceso de prueba y error hasta que finalmente concluí las operaciones que necesitaba realizar.



APARTADO 6: GAME

La clase *Game* representa el estado de un juego, incluyendo el tablero de juego, el jugador actual, y un valor booleano que indica si el jugador ha ganado. La clase *Game* comprende las reglas del juego y puede determinar si una posición es seleccionable como inicio de un movimiento, si otra posición es seleccionable como su final, y puede ejecutar un movimiento en consecuencia. En esta clase expondré los tres métodos más complejos que he encontrado, principalmente se entran en las tres funciones principales que son el *isValidFrom*, *isValidTo* y *move*. Para exponer estas funciones, explicare el resultado derivado y el proceso mediante el uso de diagramas.

Al principio lo que se me dificultó fue entender la diferencia u objetivo de estas funciones, pero pude resolverlo gracias que en el fórum del campus se había tratado el tema, asimismo también tuve que comparar los *tests* para ver que revisaban y entre ellos que los diferenciaba. También cabe mencionar que en el proceso fui entendiendo aún más gracias a que como mi inicial objetivo básico era que el código me pasara el *test* y indagando porque según que *tests* no me funcionaba me ayudaban a comprender el propósito y como aplicaba la lógica del programa.⁶

Como tal las primeras funciones tampoco hubo dificultad ya que eran un constructor, un *getter* y un *return*.

Cuando comprendí por primera vez cómo funcionaban las funciones *isValidFrom* y *isValidTo*, se me ocurrió la idea de utilizar la función *isValidTo* para adaptarla y utilizarla en *isValidFrom*. Básicamente, pretendía asignarle dos tareas diferentes a la función *isValidTo*. Sin embargo, más tarde me di cuenta de que esto no era factible debido a cómo estaba estructurada la función principal, que requería que la función *isValidTo* realizara una tarea específica. A pesar de esto, en ese momento creé un diagrama para ilustrar este plan inicial.⁷

public boolean *isValidFrom* (Position position)

```

public boolean isValidFrom (Position position) {
    Quiero obtener la lista de direcciones → Esta depende del jugador
    Direction[] validDirections = (get currentPlayer() == Player.WHITE) ? WHITE_DIRECTIONS : BLACK_DIRECTIONS;
    Me aseguro que en la posición que estoy NO ESTE VACÍA (tengo pieza)
    NO ESTOY EN ZONA prohibida (y en la misma función me aseguro que estoy dentro del rango)
    if (board.isEmpty(position) || board.isForbidden(position)) return false;

    El siguiente bloque de código NO ES prescindible, ya que sin esta pasa los tests, pero al momento de jugar sí es necesaria ya que no quiero que se mueva el cuadro de un jugador que no se puede mover.
    if (isOpponentPiece(position)) return false;

    Una vez descartada esas inconvenientes delego el trabajo a una función para ver de mis direcciones cuales son válidas
    return isValidPosition(position, validDirections);
}

una función sencilla para ver del actual turno cual es su oponente
private boolean isOpponentPiece (Position position) {
    return (get currentPlayer() == Player.WHITE) ? board.isBlack(position) : board.isWhite(position);
}
    veo actual jugador      si es blanco → EL negro es el oponente      sino es el blanco → eso significa que es blanco de negro

```

⁶ El estudio de algunos *tests* queda evidenciado también en anexos de imágenes

⁷ Consulte anexos de imágenes

```

private boolean isValidPosition (Position position, Direction[] validDirections) {
    // Itero las direcciones válidas → FOR-EACH este método lo vimos en clase de prácticas y me gustó aplicarlo
    for (Direction direction : validDirections) {
        Position nextPosition = direction.apply(position);
        if (nextPosition != null) {
            if (isOpponentPiece (nextPosition)) {
                Position enemy = direction.apply (nextPosition);
                if (enemy != null && board.isEmpty (enemy) && position.sameDiagonalAs (enemy)) {
                    // Esta no está en el código actual lo considere anteriormente hasta que me acordaba que no es necesario
                    // esto me da cuenta gracias al texto de "Invalid From"
                    // Este caso también me da cuenta gracias a los tests
                    // También ando en cuenta cuenta de este error ya que el test daba a entender que como tal era válido pero no se podía evaluar true por el falso debido a que no formaba parte de la misma diagonal
                    // si la siguiente posición es vacía la pieza puede moverse → true
                    return true;
                }
            } else if (board.isEmpty (nextPosition)) return true;
        }
    }
    return false;
}

```

Como se menciona en el diagrama se ha implementado el bucle *for-each* el cual quise usarlo sobretodo por la nitidez del código y porque a gustos personales ayuda a la estética visual del código, no obstante, antes de usar este método usé un *for*. [6]

public boolean isValidTo (Position validFrom, Position to)

A continuación, explicaré visualmente el código implementado, en este caso me ayudó bastante el esquema inicial que hice, con ese me di cuenta otra vez de como explicar lo que buscaba⁸.

```

public boolean isValidTo (Position validFrom, Position to) {
    // quiero ver que el jugador que comienza es el blanco (así se juegan las damas)
    boolean isWhite = getCurrentPlayer() == Player.WHITE;
    // declaro las variables mid y dist, esto es debido a que en un caso que veremos mas adelante las necesitaremos
    Position mid = Position.middle (validFrom, to);
    int dist = Position.distance (validFrom, to);

    if (
        !this.board.isEmpty (to) ||
        !validFrom.sameDiagonalAs (to) ||
        ! (isWhite ? to.getY() <= validFrom.getY() : to.getY() >= validFrom.getY())
    ) {
        // hago la búsqueda de lo contrario a lo que quiero para usar OR y hacerlo mas optimo
        return false;
    }

    return dist == 2 || (dist == 4 && (isWhite ? this.board.isBlack (mid) : this.board.isWhite (mid)));
}

```

CASO NORMAL

CASO DE CAPTURA

CASO PROXIMO

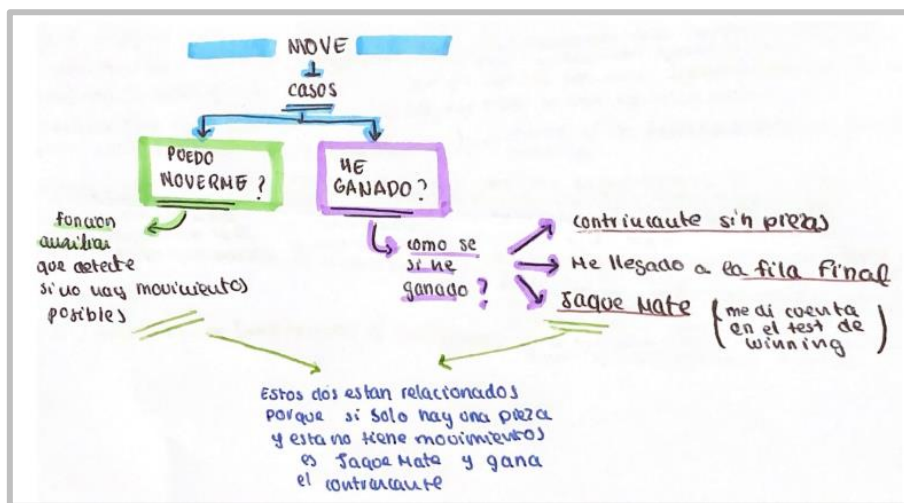
CASO LEJANO

me aseguro que lo del medio sea pieza enemiga para que pueda haber ataque

⁸ Se trata del esquema donde quise asignarle dos tareas al *isValidTo*, gracias a este la tarea de realizar el *isValidTo* no fue tan tediosa como pudo haber llegado a ser

public Move *move*(Position validFrom, Position validTo)

La función *move* fue una de las funciones complejas en su momento ya que debía realizar el movimiento y tener aspectos en cuenta. Pero gracias a la experiencia de anteriores practicas pude realizar un enfoque estructurado de lo que quería y que métodos auxiliares debía realizar para conseguir el objetivo. Una vez acabado el esquema el código fue perfeccionando gracias a la demanda de los tests y posteriormente al tener una idea clara el código fue optimizado⁹. Ahora explicare el esquema original con las ideas base y como lo optimice para posteriormente explicar la idea central del código resultante. Como he dicho antes, lo complicado de esta función fue como enfocar el movimiento junto los posibles casos.



```

public Move move (Position validFrom, Position validTo) {
    Position mid = null;
    // dedaró una variable mid que sea la del medio, la declaro
    // null porque tal vez no hay una posición intermedia y de este
    // modo no afecta al retorno

    if (Position.distance (validFrom, validTo) > 2) {
        mid = Position.middle (validFrom, validTo);
        board.setEmpty (validFrom);
    }

    if (board.isWhite (validFrom)) board.setWhite (validTo);
    else board.setBlack (validTo);
    board.setEmpty (validFrom);

    this.hasWon = hasPlayerWon (get (currentPlayer()) == Player.WHITE);
    // En el caso de haber ganador este sera segun lo que diga la funcion de si hay
    // ganador. Hace falta una funcion que la estructura de este booleano lo vice
    // jugando con el isWhite o !isWhite (al revés)

    if (!hasWon()) currentPlayer = get (currentPlayer()) == Player.WHITE ? Player.BLACK : Player.WHITE;

    return new Move (validFrom, mid, validTo);
}

```

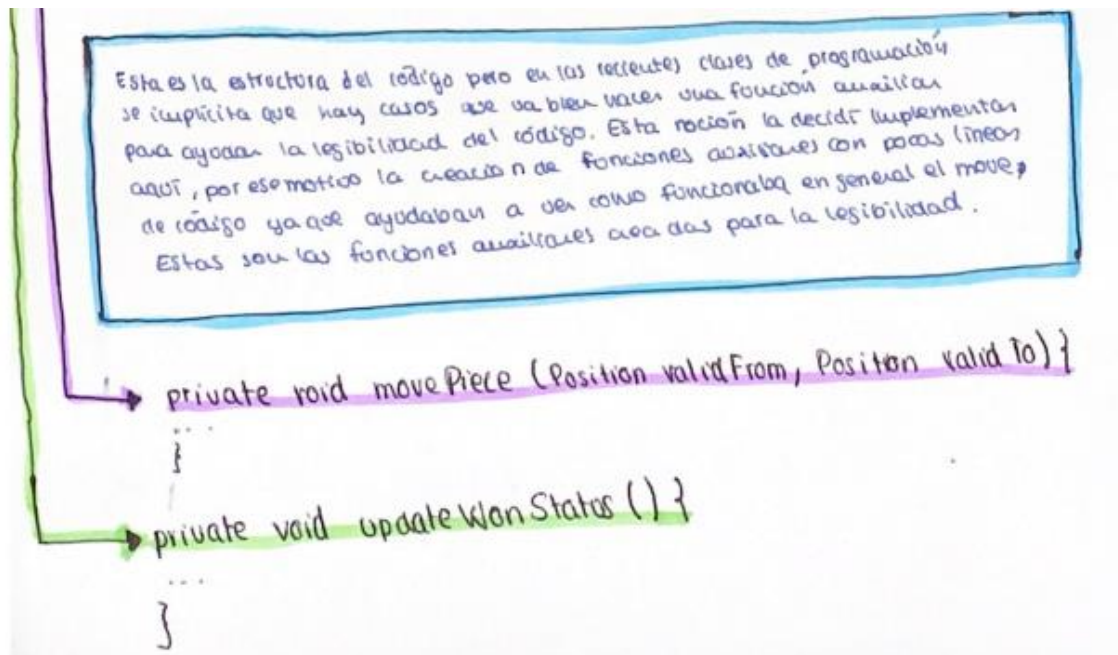
CASO LESANO
→ hago el calculo del mid
→ como la pieza se va a mover borro donde estaba

MOVIMIENTO
basicamente miro que pieza es (color) pinto donde va a ir y borro donde estaba haciendo la ilusion de movimiento

CASO DE GANADOR
seguros el jugando haciendo cambio y viendo el turno del jugador

CASO QUE AUN NO TENEMOS GANADOR

⁹ Al acabar el test tenía un total de más de 6 funciones auxiliares, pero posteriormente pude optimizarlo quedando menos de la mitad de las funciones iniciales



¿HE GANADO?

Funciones iniciales

```
private boolean whiteHasWon() {
    int y = 0;
    if (board.getNumBlacks() == 0) {
        return true;
    }
    for (int x = 0; x < board.getWidth(); x++) {
        if (board.isWhite(new Position(x, y))) {
            return true;
        }
    }
    return blackHasNoMoves();
}
```

```
private boolean blackHasWon() {
    int y = board.getHeight() - 1;
    if (board.getNumWhites() == 0) {
        return true;
    }
    for (int x = 0; x < board.getWidth(); x++) {
        if (board.isBlack(new Position(x, y))) {
            return true;
        }
    }
    return whiteHasNoMoves();
}
```

Función resultante

```
private boolean hasPlayerWon(boolean isWhite) {
    int startY = isWhite ? 0 : board.getHeight() - 1;
    int opponentPieces = isWhite ? board.getNumBlacks() : board.getNumWhites();

    if (opponentPieces == 0) {
        return true;
    }

    for (int x = 0; x < board.getWidth(); x++) {
        Position pos = new Position(x, startY);
        if ((isWhite && board.isWhite(pos)) || (!isWhite && board.isBlack(pos))) {
            return true;
        }
    }

    return playerHasNoMoves(!isWhite);
}
```

Ubico la fila en la que debería estar, dependiendo de si es blanca o negra. Además, observo el número de piezas, lo cual también puede ser un factor para determinar si gano, según el esquema

Itero el tablero y verifico si la pieza en la posición pos corresponde al color especificado y retorna false si esa pieza no puede realizar ningún movimiento válido.

¿PUEDO MOVERME?

Funciones iniciales

```
private boolean whiteHasNoMoves() {
    Position pos;
    for (int y = 0; y < board.getHeight(); y++) {
        for (int x = 0; x < board.getWidth(); x++) {
            pos = new Position(x, y);

            if (board.isWhite(pos) && !canMove(pos)) {
                return false;
            }
        }
    }
    return true;
}
```

```
private boolean blackHasNoMoves() {
    Position pos;
    for (int y = 0; y < board.getHeight(); y++) {
        for (int x = 0; x < board.getWidth(); x++) {
            pos = new Position(x, y);
            if (board.isBlack(pos) && !canMove(pos)) {
                return false;
            }
        }
    }
    return true;
}
```

Función resultante

```
private boolean playerHasNoMoves(boolean isWhite) {
    Position pos;
    for (int y = 0; y < board.getHeight(); y++) {
        for (int x = 0; x < board.getWidth(); x++) {
            pos = new Position(x, y);

            if ((isWhite && board.isWhite(pos)) || (!isWhite && board.isBlack(pos))) {
                if (!canMove(pos)) {
                    return false;
                }
            }
        }
    }
    return true;
}
```

Recorro la tabla y anoto cada posición con sus coordenadas correspondientes

Observo qué tipo de pieza es y llamo a la función auxiliar de movimientos para verificar si hay movimientos disponibles (búsqueda del caso contrario)

LLAMADA A LA FUNCIÓN DE MOVIMIENTOS HABILITADOS

```
private boolean canMove(Position pos) {
    if (board.isWhite(pos)) {
        return board.isEmpty(WHITE_DIRECTIONS[0].apply(pos)) ||
            board.isEmpty(WHITE_DIRECTIONS[1].apply(pos)) ||
            (board.isBlack(WHITE_DIRECTIONS[0].apply(pos)) && board.isEmpty(WHITE_DIRECTIONS[0].apply(WHITE_DIRECTIONS[0].apply(pos)))) ||
            (board.isBlack(WHITE_DIRECTIONS[1].apply(pos)) && board.isEmpty(WHITE_DIRECTIONS[1].apply(WHITE_DIRECTIONS[1].apply(pos))));
    } else if (board.isBlack(pos)) {
        return board.isEmpty(BLACK_DIRECTIONS[0].apply(pos)) ||
            board.isEmpty(BLACK_DIRECTIONS[1].apply(pos)) ||
            (board.isWhite(BLACK_DIRECTIONS[0].apply(pos)) && board.isEmpty(BLACK_DIRECTIONS[0].apply(BLACK_DIRECTIONS[0].apply(pos)))) ||
            (board.isWhite(BLACK_DIRECTIONS[1].apply(pos)) && board.isEmpty(BLACK_DIRECTIONS[1].apply(BLACK_DIRECTIONS[1].apply(pos))));
    }
    return false;
}
```

Los arrays *WHITE_DIRECTIONS* y *BLACK_DIRECTIONS* contienen las direcciones en las que una pieza blanca o negra puede moverse en el tablero. Estas direcciones se utilizan para verificar movimientos válidos desde una posición dada "pos" en el tablero.

Para una pieza blanca, *WHITE_DIRECTIONS[0]* representa un movimiento hacia arriba y a la izquierda, mientras que *WHITE_DIRECTIONS[1]* representa un movimiento hacia arriba y a la derecha.

Para una pieza negra, *BLACK_DIRECTIONS[0]* representa un movimiento hacia abajo y a la izquierda, mientras que *BLACK_DIRECTIONS[1]* representa un movimiento hacia abajo y a la derecha.

El método *canMove(Position pos)* utiliza estas direcciones para verificar si las posiciones en estas direcciones están vacías o contienen piezas del oponente. Esto ayuda a identificar si una pieza puede realizar un movimiento válido.

CONCLUSIONES

Esta práctica de programación en Java ha sido un gran paso hacia mi conocimiento respecto al lenguaje. He obtenido un entendimiento mucho más amplio de conceptos teóricos como métodos públicos o privados y la importancia de seguir un enfoque estructurado.

El aspecto más intrigante de esta práctica fue la conexión entre operaciones matemáticas y codificación. Tuve la oportunidad de pensar en el proceso de traducir operaciones matemáticas en código dentro de clases, lo que me permitió apreciar más profundamente cómo estos campos aparentemente diferentes están interconectados. A través de búsquedas independientes y lectura de diferentes artículos, pude mejorar mi conocimiento de metodologías y funciones de clases, mejorando así mi comprensión general del tema.

Programar un juego fue una experiencia emocionante que me permitió abordar esta práctica con entusiasmo. Fue una oportunidad para probar y jugar con lo que estaba creando, fomentando un enfoque más optimista en comparación con prácticas anteriores. Los conceptos que me llevo de esta práctica incluyen el estudio previo de funciones, como la búsqueda de patrones que hice para diagonales o para la clase *Geometry*. Además, crear un marco para organizar cómo proceder en cada función ha sido un paso significativo para mí. Anteriormente, solía tardar días o incluso semanas en resolver un código, pero gracias a adaptar esta metodología, ahora puedo resolver problemas de manera mucho más eficiente.

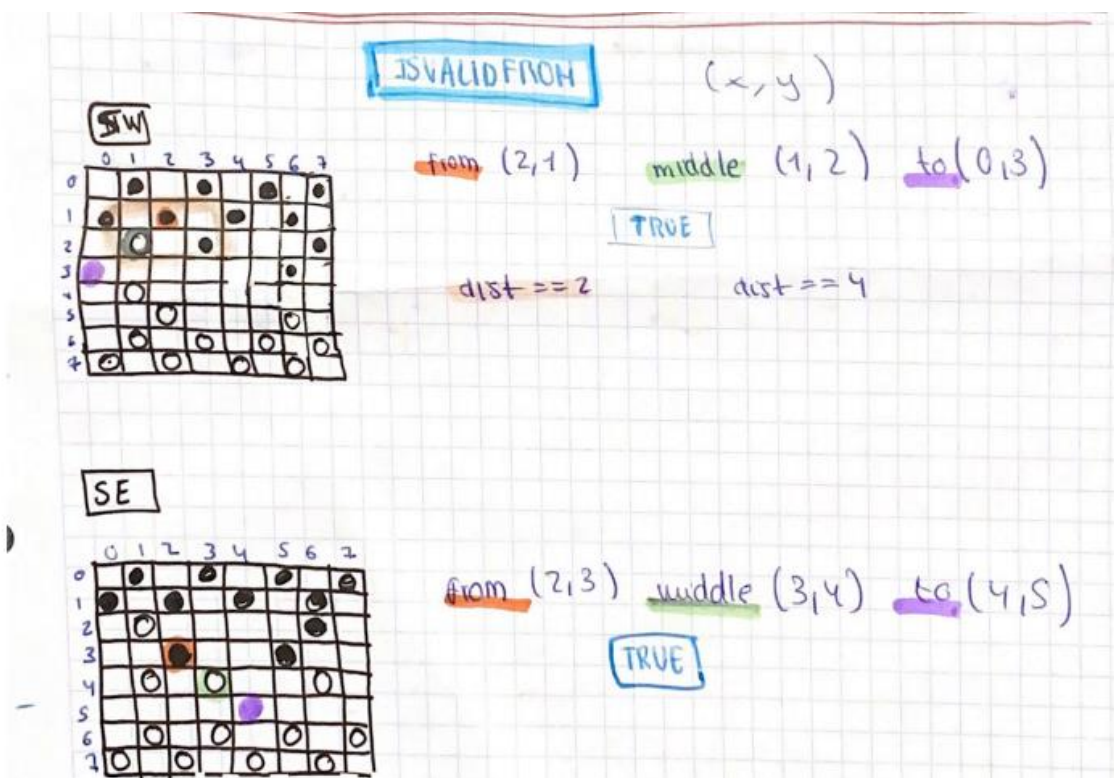
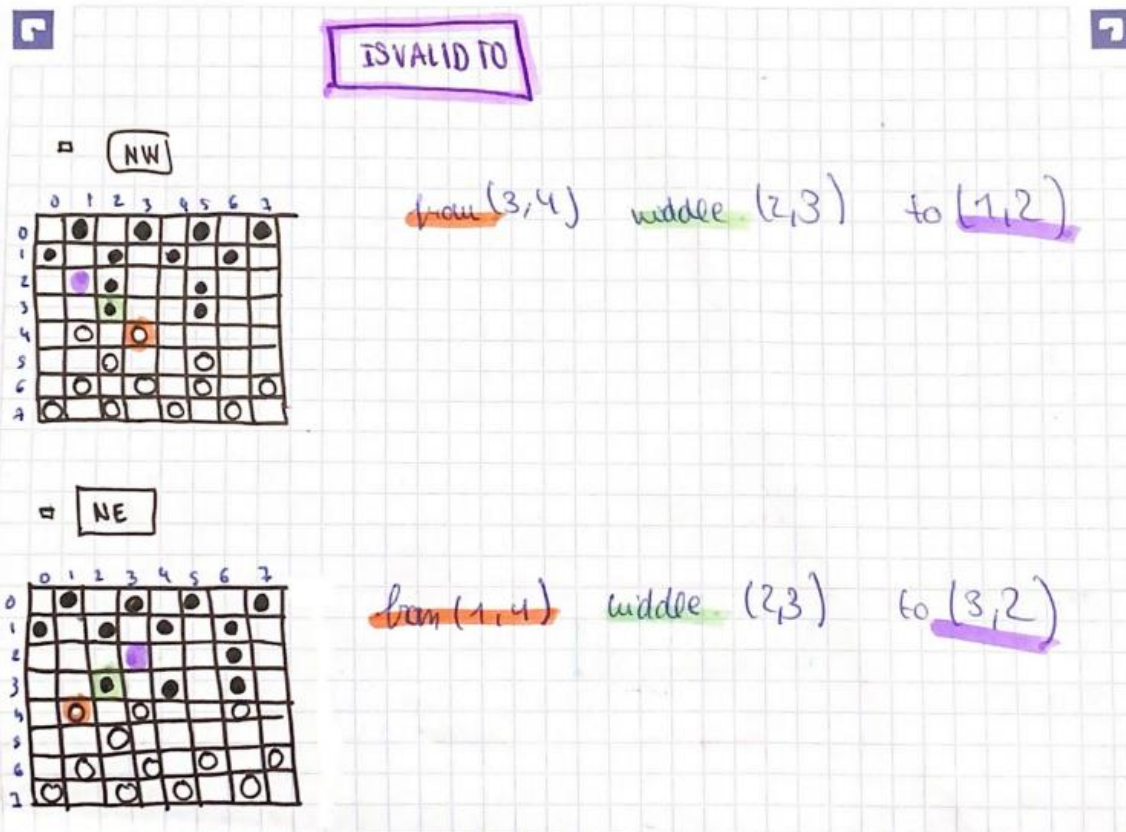
En retrospectiva, me doy cuenta de que podría haber reducido el tiempo dedicado a esta práctica atreviéndome a hacer más preguntas en el foro del campus en lugar de depurar y analizar prueba por prueba. Sin embargo, estoy orgullosa de cómo he podido mejorar la legibilidad del código al equilibrar lo que es un código óptimo y un código legible. Solía creer que cuantas menos líneas y funciones tuviera el programa, mejor sería. Sin embargo, esta práctica me ha enseñado que la legibilidad es primordial al programar.

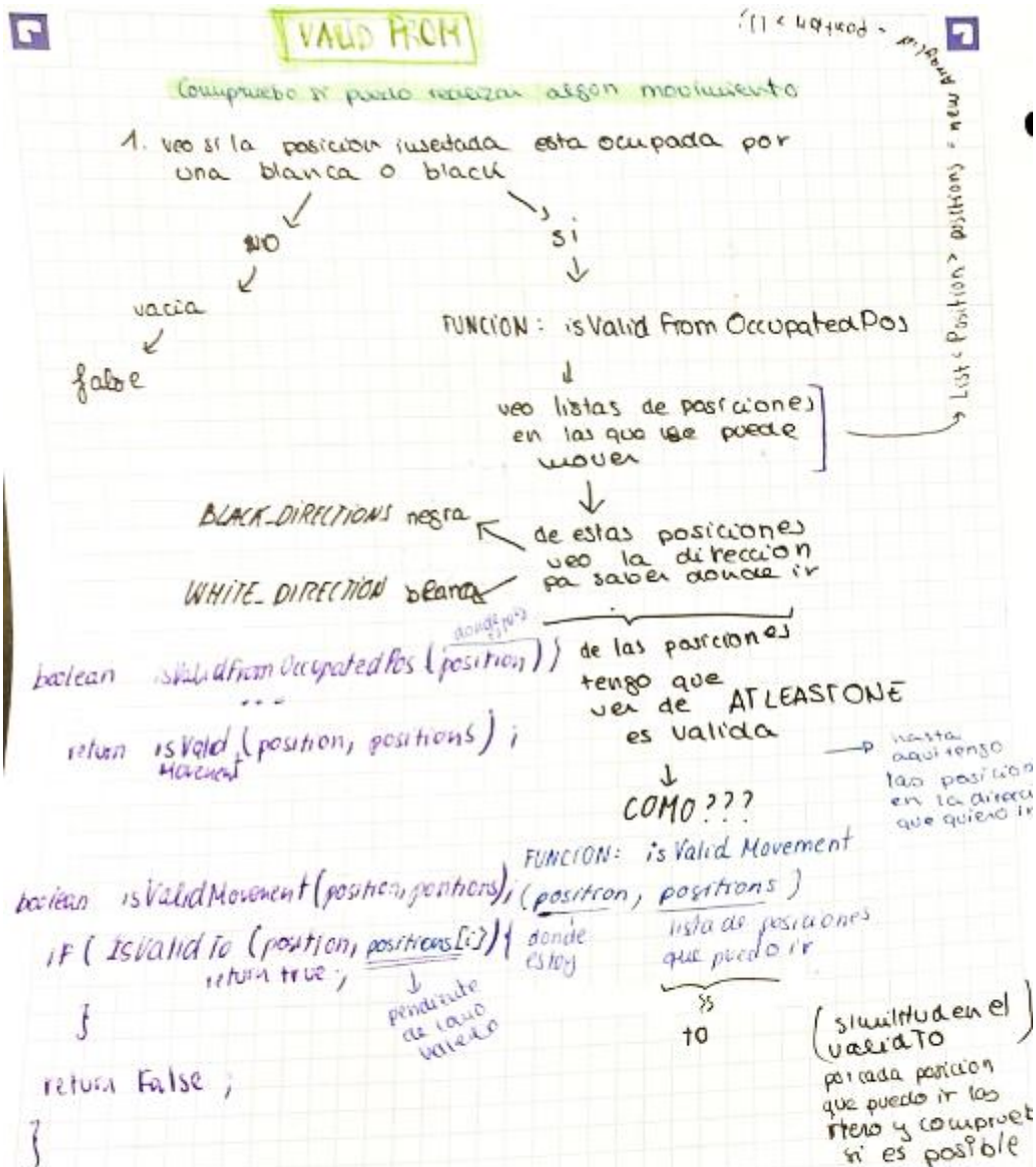
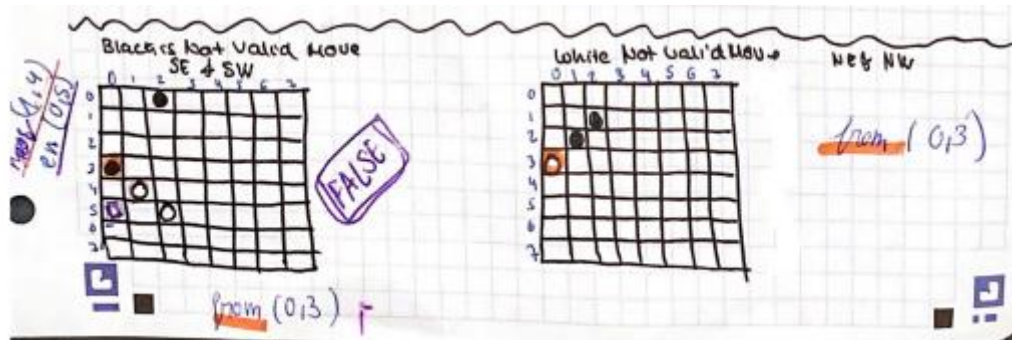
En resumen, esta práctica ha sido gratificante en mi progreso como programadora, y espero aplicar los conocimientos y habilidades adquiridos en futuras prácticas.

Referencias

- [1] The Java Tutorials, «Oracle,» Java Documentation, 2022. [En línea]. Available: <https://docs.oracle.com/javase/tutorial/java/javaOO/QandE/enum-questions.html>. [Último acceso: Abril 2024].
- [2] Microsoft, «Microsoft Build,» 2024. [En línea]. Available: <https://learn.microsoft.com/es-es/dotnet/api/java.lang.math?view=net-android-34.0>. [Último acceso: Abril 2024].
- [3] Shiksha Online, «Conditional Statements in Java,» 21 Agosto 2023. [En línea]. Available: <https://www.shiksha.com/online-courses/articles/conditional-statements-in-java/>. [Último acceso: Abril 2024].
- [4] Disco Duro de Roer, «Recorrer una Matriz con un Bucle en Java,» 2017.
- [5] Programación 2, *Problemas de Programación Orientada a Objetos (v1)*, Problemas Universidad de Lleida, 2023/2024.
- [6] J. Hartman, «GURU 99,» 26 Diciembre 2023. [En línea]. Available: www.guru99.com/es/foreach-loop-java.html. [Último acceso: Abril 2024].

Anexo de imágenes





PISTA:

ISVALIDFROM: Compruebo si puedo realizar algun movimiento

ISVALIDTO: Selecciono los movimientos posibles

VALID TO

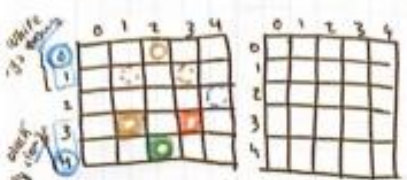
(position valid from, position to)

Selecciona los movimientos posibles

+
RESTRICCIONES (lógica del juego)

$\left\{ \begin{array}{l} \text{position middle} \\ \text{int distance} \end{array} \right\}$

$\text{position mid} = \text{Position.middle}(\text{validfrom}, \text{to});$
 $\text{int dist} = \text{Position.distance}(\text{validfrom}, \text{to});$



- To No ha de estar vacía $! \text{this.board.isEmpty}(\text{to})$ (el contrario a lo que busco pa usar el + optimo)
- validfrom y to NO estan en la misma diagonal $! \text{validfrom.sameDiagonalAs}(\text{to})$

→ Direction?

$\left\{ \begin{array}{l} \text{is White} \\ \text{is Black} \end{array} \right\}$

$\Rightarrow \text{to.get}Y > \text{validfrom.get}Y$
 $\Rightarrow \text{to.get}Y < \text{validfrom.get}Y$

¿is white?

$\left\{ \begin{array}{l} \text{to.get}Y() < \text{validfrom.get}Y() \\ \text{to.get}Y() > \text{validfrom.get}Y() \end{array} \right\}$

RETURN y FALSE

por ahora "deberia" haber descartado casos cercanos. Ahora gestionar los saltos y distancia

DISTANCE

Black

- pos 1 (2, 4) $dx = |2-3| = 1$
- pos 2 (3, 3) $dy = |4-3| = 1$
- distance $1+1 = 2$

White

- pos 1 (2, 4) $dx = |2-1| = 1$
- pos 2 (1, 3) $dy = |4-3| = 1$
- distance $1+1 = 2$

el que es mas proximo

$\text{dist} == 2$ (true)

o SALTO dist es 4 tengo que ver

analisis del espacio

↳ ven que soy (blanca o negra)

debe haber algo si o si

(descarte el proximo con dist = 2)

PENDIENTE

optimizar caso de un solo vacio?

dormir

esto se rompe return true;