Practica 2: Checkers

Programación 2

Nombre: Denis Emanuel, Lazarian

NIE: Y0804111-E

Pass: 060960434

Grupo: GPraLab4

Fecha: 17/04/2024

Descripción general

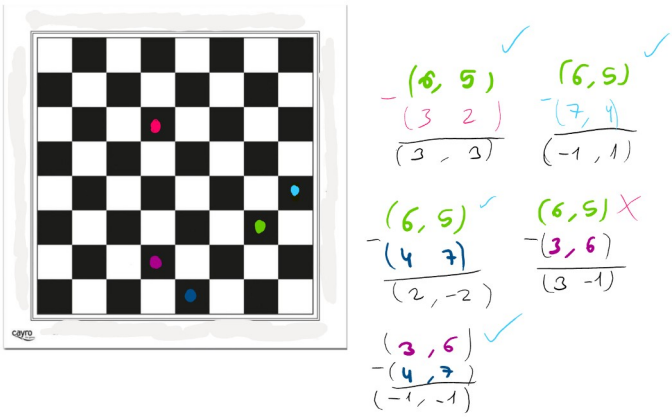
Al principio, para el desarrollo de las primeras clases he hecho uso de los test unitarios disponibles, en el proyecto, para garantizar su correcto funcionamiento.

En la clase *Position*, para el desarrollo método *sameDiagonalAs* había pensado en realizar 4 bucles para un recorrido en las dos diagonales de un punto, de esta forma lo comparo con la otra posición y devuelve la expresión conforme están en la misma diagonal, ya sea la normal o la inversa. Pero he visto un problema, el proyecto en si tiene contemplado el hecho que se pueden modificar el tamaño del tablero, por ende no existe un tamaño fijo.

Por este hecho he visto que era incorrecto, dejando de lado el hecho que era ineficiente, por ende me he documentado mas sobre el calculo de coordenadas cartesianas y ver si existe una forma de verificar que dos puntos estén en la misma diagonal. Al final he empleado la siguiente formula:

$$|(x_1 - x_2)| = |(y_1 - y_2)|$$

Aquí resto las coordenadas X e Y de los dos puntos, realizo el valor absoluto y comparo sus resultados, si dan el mismo resultado significa que están en la misma diagonal. En la siguiente imagen se pueden ver los test:



En la clase *Cell* para el desarrollo de la función *fromChar*, consiste en comprobar los 4 estados posibles de una variable:

- **Forbidden** (Prohibido)
- **Empty** (Vacía)

- **White** (Blanca)
- **Black** (Negra)

Por tanto, para esta función, he considerado mas adecuado, a nivel de legibilidad, utilizar la estructura de control *Switch-case*, para comprobar los 4 estados dependiendo del carácter que nos llega través de la variable *status*.

Otro de los problemas que he tenido es en los test de la clase *Game*, sobre todo a la hora de comprobar la cantidad de fichas existentes en el tablero. Al principio he pensado de realizar este conteo dentro de la clase *Game*, pero no disponía de los permisos para ello ya que esos atributos se declararon en estado privado. Entonces caí en la cuenta que esa gestión la tenia que hacer a través de los *setters* de la clase *Board*.
Por tanto cuando establezco una casilla a vacío, compruebo previamente el tipo de ficha y reduzco el contador, cuando establezco la ficha al tablero aumento el contador a ese tipo de ficha en concreto.

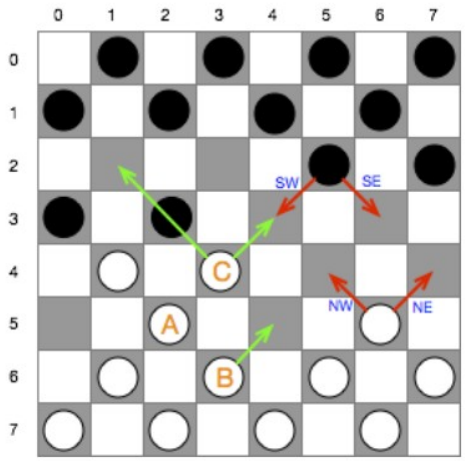
La clase Game

La clase *Game* es la encargada de manejar e imponer las reglas del juego. La clase *Game* es el apartado mas complejo en toda la aplicación y por ende aquí explicare las 3 funciones mas complejas, que son las funciones *isValidFrom*, *isValidTo y move*.

La instancia de la clase es un constructor normal en la que se asigna el tablero, se establece la variable de victoria a falso, y por defecto el jugador de piezas blancas dispone del primer turno.

Las reglas del juego son las siguientes:

- Cada ficha solo se pueden desplazar en diagonal.
- Las fichas solo disponen de 2 direcciones de desplazamiento, de forma respectiva, ya que ambas fichas se pueden mover al este y oeste, pero al norte solo se mueven las fichas blancas y las negras solo se mueven por el sur, por ende las fichas negras disponen de 2 movimientos posibles (*South-West, South-East*) y las blancas (*North-West, South-East*).
- Por los general cada ficha solo se mueve 1 casilla de distancia, pero si se trata de una captura su desplazamiento consiste en dos casillas de distancia. Una captura se efectúa cuando una de las fichas tiene a otra de su color contrario ocupando su primera celda de desplazamiento, siempre y cuando la segunda celda de desplazamiento este vacía.



Funciones principales

`public Player getCurrentPlayer()`

Nos permite obtener el jugador que disponga del turno actual.

`public boolean hasWon()`

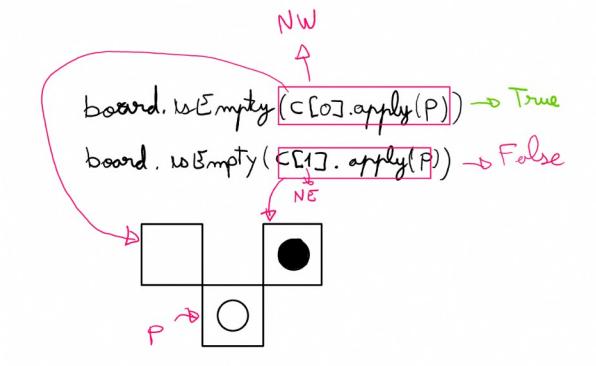
Nos permite saber si la partida actual dispone de un ganador.

`public boolean isValidFrom(Position position)`

- position: Posición inicial

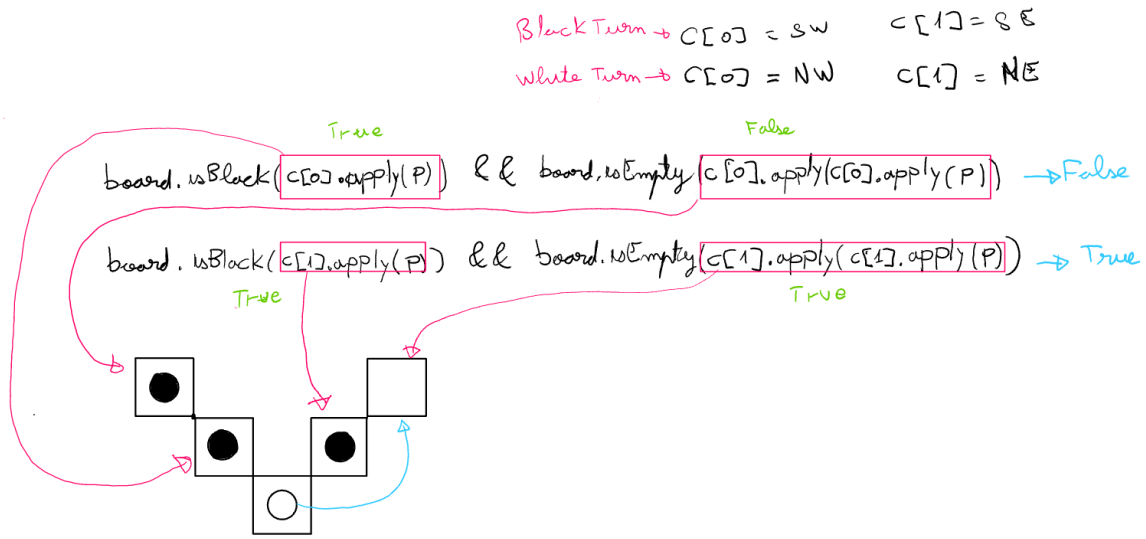
Su función es indicar al usuarios que fichas disponen de movimiento posible.
Para eso hay que cumplir, de forma obligatoria, una serie de condiciones:

1. La partida no dispone de un ganador.
2. La posición actual no puede estar vacía.
3. La ficha seleccionada pertenece al propio jugador, no puede seleccionar las fichas de un jugador donde no es su turno o los dos tipos de fichas a la vez.



Hay que tener en cuenta que ***c*** *varia su valor en función del turno.*

4. Teniendo en cuenta que vector c dispone de 2 direciones, si la direccion 1, aplicado sobre la posición, está vacío o bien la direccion 2 lo está.
5. Si la direccion 1 de una ficha, en su siguiente casilla está ocupada por una ficha contraria, adicionalmente comprobamos que la siguiente casilla, a la mencionada y en la misma dirección, no este ocupada por cualquier otra ficha, o bien comprobamos esas condiciones para su dirección 2.



Entre las condiciones 1, 2, 3 han de cumplirse todas y entre las condiciones 4 y 5, una de ellas también ha de cumplirse de forma obligatoria junto a las primeras, puede ser la 4 o la 5, o bien las dos.

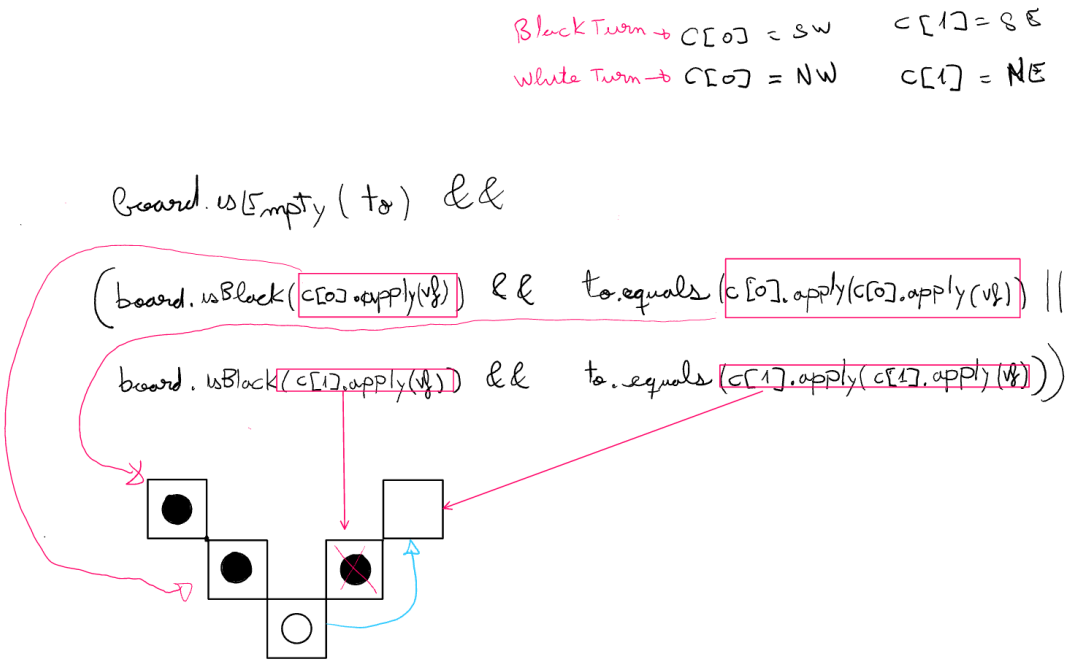
`public boolean isValidTo(Position validFrom, Position to)`

- *vf: Posición* inicial.
- *to: Posición* final.

La función que desempeña, es la de indicar los movimientos que se pueden realizar. Una vez comprobado que se pueden realizar algún movimiento, la función indica cual de esos movimientos se puede realizar.

Una vez validado el hecho que una pieza disponga de movimiento posible, señalizamos el movimiento posible tras comprobar una serie de condiciones:

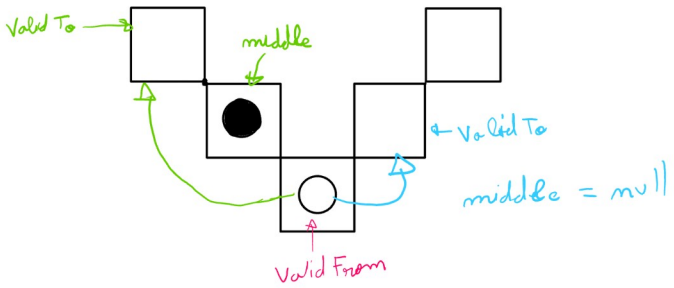
1. Si la posición inicial (*vf*) y la final (*to*) estén en la misma diagonal.
2. Si la casilla donde marca la posición final (*to*) esta vacía.
3. Si el movimiento que realizo lo hago en la dirección correcta en el eje *Y*, es decir, si la función auxiliar *iAmNotReturningBack(vf, to)* evalúa a *true. R*evisen la explicación de dicha función par mas detalles.
4. Si la distancia entre posición inicial y posición final es igual a 2, ya que con eso se comprueba que avanzamos una casilla.
5. Si la siguiente casilla a la inicial (vf) está ocupada por el color contrario, y adicionalmente compruebo si la posición final es la misma que la segunda casilla respecto a la ficha seleccionada. Esto solo se aplica para la dirección “oeste” (West) de los dos jugadores.
6. Lo mismo que en la condición 6, aplicada para la dirección “este” (East) disponible para la casilla seleccionada.



Para validar la expresión booleana que devuelve la función, entre la expresión 1, 2, 3 han de cumplirse todas y también tiene que cumplirse mínimo una de las condiciones compuesto por la 4, 5, 6, si esas tres evalúan a *false*, pues toda la expresión lo hará.

`Public Move move(Position validFrom, Position validTo)`

Teniendo garantizado que isValidFrom e isValidTo evalúen a cierto, esta función se encarga de efectuar el movimiento. Primero la función comprueba que el movimiento que queremos realizar se trata de una captura o un movimiento normal. Para ello comprobamos la distancia entre la posición inicial y la posición final sea igual a 4, en caso que sea cierto, se trata de un movimiento de captura, por eso dispone de una casilla de en medio, por tanto lo podemos agregar en la nueva instancia *Move* que declaramos. De lo contrario, simplemente declaramos la instancia con el atributo *middle* con valor nulo.



La función de forma adicional realiza otras acciones como la de modificar el tablero (*updateBoard(move)*), la de comprobar si el movimiento realizado le da la victoria al jugador (*checkAndDeclareWinner(move)*) y adicionalmente también cambia el turno del jugador por cada movimiento realizado (*changePlayerTurn()*), observad la explicación de cada función para ver las acciones que se realizan por detrás cuando se realiza un movimiento.

Funciones auxiliares

`private boolean iAmNotReturningBack(Position vf, Position vt)`

- vf: Posición inicial.
- vt: Posición final.

La función principalmente se encarga de comprobar que las fichas de cada jugador no se mueven en la dirección equivocada en el eje de las *Y*, es decir, que una casilla negra no se mueva hacia el norte y una ficha blanca no se mueva por el sur. Para ello compruebo los atributos *Y* de las dos posiciones. Cuando la posición inicial dispone de una *Y* mayor a la de la posición final *(vf.getY() > vt.getY())*, quiere decir que se mueve al norte, y cuando es al contrario significa que se mueve al sur.

Por tanto, en función del turno, devuelvo esas expresiones, es decir, si el turno es del jugador de fichas blancas devuelvo la expresión conforme se mueve al norte, y cuando el turno es del jugador de fichas negras pues devuelvo la expresión conforme se mueve al sur.

`private void isPieceColor(boolean[] t, Direction[] c, Position vf)`

- t: Es quien contendrá el resultado de las expresiones explicadas mas tarde.
- c: *S*on las constantes de dirección disponibles para cada jugador. En la constante se guardan la dirección oeste y este de los dos jugadores. La dirección norte y sur se modifican en función del turno.
- vf: Posición inicial.

En función del turno del jugador dentro del vector c guarda las expresiones, que evalua si la siguiente casilla esta ocupada por alguna ficha del color contrario en las dos direcciones (West, East).

`private void updateBoard(Move move)`

- move: Movimiento realizado.

Dependiendo del turno del jugador, dentro de la posición final añadimos una ficha blanca o negra. Después cambiamos el estado de la celda en vacío a la celda del medio y la celda de procedencia. En caso que la celda de en medio sea nulo, los *setters* de la clase Board por dentro comprueba si la celda esta prohibida donde una de las condiciones que valida dicha prohibición, es cuando la posición es un valor nulo.

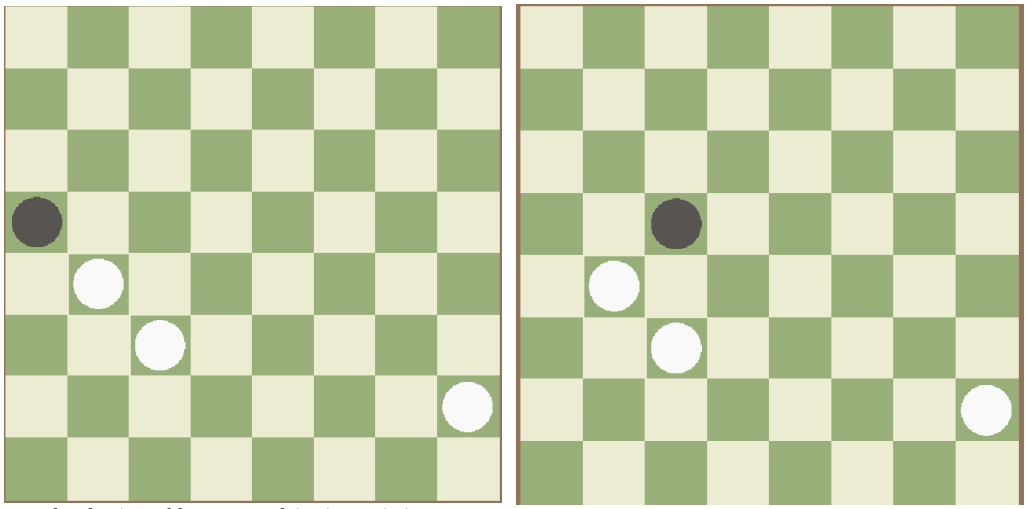
`private void checkAndDeclareWinner(Move move)`

- move: Movimiento realizado.

El rol que toma la función es la de comprobar si el movimiento realizado le concede la victoria al jugador con el turno actual.

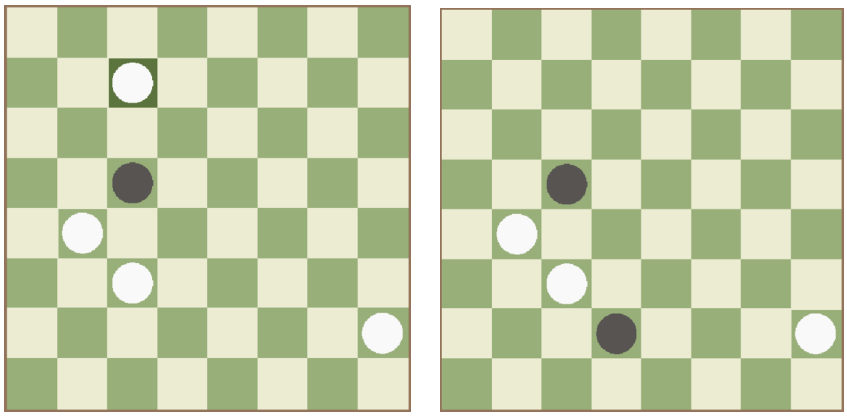
Las condiciones que se tienen que cumplir para declarar la victoria a un jugador, en este proyecto, son las siguientes:

- Si las fichas del jugador contrario ya no disponen de movimiento posible, privándolo de realizar cualquier acción. Aunque también si un jugador ya no dispone de fichas en el tablero, tampoco dispone de posibles movimientos, por ende esta expresión comprueba los dos caso a la vez. La función que gestiona la cantidad de fichas disponibles con posible movimiento se llama *totalMoves*, y por cada movimiento se actualizara su valor.



Jugador de piezas blancas que deja sin movimiento posible a la ultima ficha de color negro.

- Si alguna de las fichas llegan al extremo contrario. Es decir para el caso del jugador de las fichas blancas, si una de sus fichas llega al extremo norte (en el eje Y = 0), se le concede la victoria. Lo mismo para el jugador de fichas negras en el extremo sur (en el eje Y = 7).

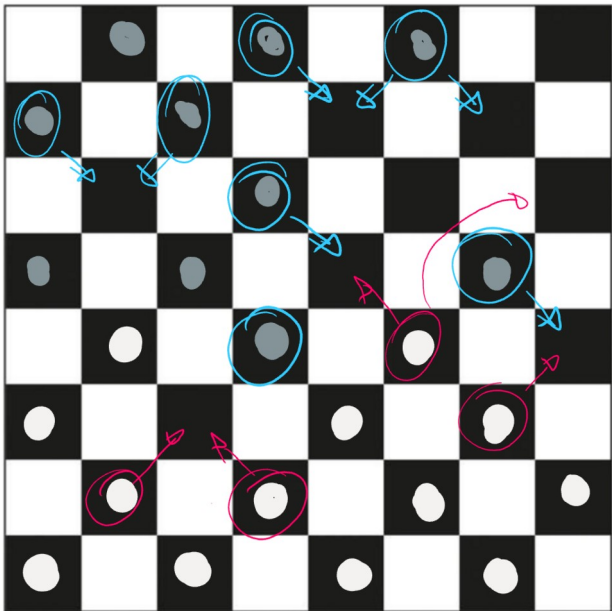


Si se cumple una de las condiciones el jugador con el turno actual obtendrá la victoria de la partida.

`private int totalMoves(char c)`

- c: carácter que queremos contabilizar.

La función realiza un recorrido por todo el tablero y según el carácter que quiero comprobar (‘b’ o ‘w’) compruebo si dispone de movimiento posible mediante la función *isValidFrom* y contabilizo las fichas cuya expresión evalúa a *true*.



Uno de los problemas que he notado con respecto al diseño del código es que funciona mucho en función de los turnos de los jugadores, para evitar duplicar mucho código. Esta función de por si contabiliza las fichas dependiendo del turno del jugador, pero da problemas cuando quiero contabilizar fichas negras si el turno es del jugador de fichas blancas. Por ende, he considerado conveniente, de forma temporal, modificar los turnos para realizar el conteo de fichas.
Primero guardo el jugador que disponga del turno, después dependiendo de la ficha que quiero contabilizar, cambio el turno, es decir, si quiero contar el carácter ‘b’, cambio el turno al jugador de fichas negras y realizo el conteo. Finalmente cambio de nuevo el turno al jugador antes de ejecutar la función y retorno la cantidad de fichas con movimientos posibles.

`private void changePlayerTurn()`

Si no disponemos de un ganador, la función simplemente se encarga de cambiar el turno del actual al contrario cada vez que lo llamamos. Esto lo hacemos cada vez que realizamos un movimiento en el tablero.

Conclusión

He observado la magnitud que puede alcanzar una aplicación desarrollada en Java, así como su complejidad durante su proceso de desarrollo. He aprendido la importancia de planificar cuidadosamente el funcionamiento de cada clase y cada función de la aplicación. Es crucial realizar exhaustivas pruebas para asegurarse de que no se arrastren errores a otras partes del programa, lo que dificultaría su detección.

Si tuviera que volver a realizar este proceso, me aseguraría de verificar meticulosamente cada función o aspecto que programo. En ocasiones, no había considerado algunos casos específicos o no me había percatado de su posible ocurrencia, lo que resultaba en errores al implementar nuevas funcionalidades. En numerosas ocasiones, me vi obligado a rehacer varias partes de la aplicación debido a una comprensión insuficiente de su comportamiento y los motivos de ciertos errores.

Bibliografía

https://stackoverflow.com/questions/18192255/regarding-java-switch-statements-using-return-and-omitting-breaks-in-each-case.

https://cv.udl.cat/access/content/group/102001-2324/Apunts/2-Programación%20Orientada%20a%20Objetos%20_v1_.pdf

https://www.programarya.com/Cursos/Java/Objetos-y-Clases