



Grado en Ingeniería informática  
Programación II

**Práctica 2:**

“Checkers”

Barreto García, Maurys Nicole  
Y-7005580R  
GPrLab1

## ÍNDICE

Introducción.....	3
Descripción de las Clases.....	4
1. Clase Positions:.....	4
2. Clase Directions:.....	4
3. Clase Cell:.....	4
4. Clase Board:.....	5
5. Clase Geometry:.....	6
6. Clase Game:.....	7
Métodos Complicados .....	8
Conclusión.....	13
Bibliografía.....	14

## INTRODUCCIÓN

El objetivo de esta práctica es construir una aplicación para jugar, entre dos jugadores, a una versión simplificada del conocido juego de las damas. Las simplificaciones son las siguientes:

- En cada turno solamente se realiza un movimiento o una captura, es decir, las capturas no pueden encadenarse.
- Cuando una ficha llega a la última fila, no se convierte en dama, sino que la partida acaba, siendo ganador el jugador que lo ha logrado.
- En ningún momento es obligatorio capturar.

El objetivo de la práctica será construir un conjunto de clases que implementen esta versión simplificada del juego:

- El sistema mostrará el tablero y permitirá seleccionar la ficha que se quiere mover. Teniendo en cuenta que solamente podrán seleccionarse fichas válidas
- Permitirá seleccionar el hueco al que se quiere mover, solamente podrán seleccionarse huecos válidos.
- El sistema indicará si el juego ha terminado.

Este informe ofrece una breve descripción de cada clase, junto con un análisis detallado de los tres métodos considerados más complejos, sus funciones auxiliares y una explicación de la estrategia empleada para abordar el problema planteado. En la conclusión, nos centraremos en los conocimientos adquiridos a través de esta práctica.

## DESCRIPCIÓN DE LAS CLASES

### 1. Clase Positions:

Esta clase representa las posiciones, parejas de X e Y, dentro del tablero de juego. Dichas posiciones servirán para identificar cada una de las celdas que hay en él. Los métodos que se han implementado son los siguientes:

- **public Position(int x, int y)** = construye la posición a partir de sus dos coordenadas.
- **public int getX()** = devuelve la coordenada x (eje horizontal) de una posición.
- **public int getY()** = devuelve la coordenada y (eje vertical) de una posición.
- **public boolean sameDiagonalAs(Position other)** = indica si la posición receptora (this) y la posición que se recibe como parámetro (other) están sobre la misma línea diagonal, ya sea en sentido / como en sentido \.
- **public static int distance(Position pos1, Position pos2)** = calcula la distancia de Manhattan (o taxicab) entre las posiciones pos1 y pos2. La distancia de Manhattan entre dos posiciones se define como la suma de los valores absolutos de las diferencias entre las coordenadas de ambas posiciones.
- **public static Position middle(Position pos1, Position pos2)** = devuelve la posición intermedia entre las posiciones pos1 y pos2. La posición media es aquella que tiene por coordenadas a las medias de las coordenadas de las posiciones.

### 2. Clase Directions:

Esta clase representa las posibles direcciones en las que podemos mover las fichas; dos de ellos serán válidos para las blancas, NW y NE; y dos para las negras, SW y SE. Viene representada por las modificaciones en x e y que hacemos sobre las coordenadas. Métodos implementados:

- **private Direction(int dx, int dy)** = constructor privado que inicializa las variables de instancia.
- **public Position apply(Position from)** = devuelve la posición consistente en habernos movido según la dirección del objeto receptor, desde la posición pasada como parámetro. En este apartado, tomamos en cuenta que se mantenga dentro del margen del tablero con el módulo de celdas que hay.

### 3. Clase Cell:

Esta clase representa a cada una de las celdas del tablero. Cada una de ellas puede ser:

- **Prohibida (FORBIDDEN)**, es decir, que no puede albergar ficha alguna.

- **Vacía (EMPTY)**, es decir, aunque pudiera ser ocupada por una ficha, no lo está en estos momentos.
- **Blanca (WHITE)**, ocupada por una ficha blanca.
- **Negra (BLACK)**, ocupada por una ficha negra. Las celdas pueden ir pasando de ocupadas (por una ficha blanca o negra) a vacías y viceversa, según los movimientos y capturas que se realicen a lo largo de la partida.

Métodos implementados en esta clase:

- **private Cell(char status)** = inicializa el atributo status con el parámetro que se le pasa.
- **public static Cell fromChar(char status)** = devuelve la instancia de Cell correspondiente al carácter: ‘.’ para FORBIDDEN, ‘ ’ para EMPTY, ‘w’ para WHITE, ‘b’ para BLACK y, finalmente, null en caso de que el carácter pasado como parámetro no sea uno de esos cuatro.
- **public boolean isForbidden()** = indica si el objeto receptor se corresponde con la celda prohibida.
- **public boolean isEmpty()** = indica si el objeto receptor se corresponde con la celda vacía.
- **public boolean isWhite()** = indica si el objeto receptor se corresponde con la celda ocupada por una ficha blanca.
- **public boolean isBlack()** = indica si el objeto receptor se corresponde con la celda ocupada por una ficha negra.

#### **4.Clase Board:**

Esta clase representa el tablero de juego. Consta de atributos para sus dimensiones y una matriz para sus celdas y, además, mantiene contadores sobre el número de fichas blancas y negras que contiene. El tablero se inicializará a partir de un String que contiene, en cada línea, los caracteres correspondientes a cada una de sus celdas. A partir de esta descripción, los métodos implementados son los siguientes:

- **public Board(int width, int height, String board)** = construye una instancia de tablero con un tamaño de width celdas de anchura, height de altura y con las celdas según los caracteres de board, que consta de height líneas (acabadas en ‘\n’) de width caracteres cada una.

Para poder llenar las celdas del string *board* se usa un método auxiliar “*fill*”, y para contar el número de piezas que hay de cada color se usa *count*.

- **public int getWidth()** = devuelve la anchura del tablero.
- **public int getHeight()** = devuelve la altura del tablero.
- **public int getNumBlacks()** = devuelve el número de fichas negras que hay en el tablero.

- **public int getNumWhites()** = devuelve el número de fichas blancas que hay en el tablero.
- **public boolean isForbidden(Position pos)** = indica si la posición dada está prohibida, ya sea porque está fuera del tablero, o porque la celda que contiene está prohibida.
- **public boolean isBlack(Position pos)** = indica si la posición dada está ocupada por una ficha negra. Las celdas fuera del tablero nunca están ocupadas.
- **public boolean isWhite(Position pos)** = indica si la posición dada está ocupada por una ficha blanca. Las celdas fuera del tablero nunca están ocupadas.
- **public boolean isEmpty(Position pos)** = indica si la posición dada está libre. Las celdas fuera del tablero nunca están libres.
- **public void setBlack(Position pos)** = ocupa con una ficha negra la celda correspondiente a la posición pos (que podéis suponer que no es prohibida). Además, se agregará al número de fichas de este color cuando se llame a este método.
- **public void setWhite(Position pos)** = ocupa con una ficha blanca la celda correspondiente a la posición pos (que podéis suponer que no es prohibida). Tendremos en cuenta que se agregarán al número de fichas de color blanco cuando se llame a este método.
- **public void setEmpty(Position pos)** = desocupa (marca como vacía) la celda correspondiente a la posición pos (que podéis suponer que no es prohibida). Cuando se llame a este método se restará al número de fichas del color de la celda que se vacíe.

## 5. Clase Geometry:

En esta clase se han agrupado todos los métodos que calculan las posiciones y dimensiones de los diferentes elementos de la interfaz gráfica. Esta clase utiliza las clases GPoint y GDimension de la librería de la acm:

- **GPoint**: representa las coordenadas de un punto en la pantalla (con getters para la x y la y, que son doubles).
- **GDimension**: representa las dimensiones de un elemento en la pantalla (con getters para height y width que son doubles).

Los métodos implementados en este apartado son los siguientes:

- **public Geometry(...)** = construye una instancia de Geometry a partir de los siguientes valores:
  - `windowWidth`: anchura de la pantalla.
  - `windowHeight`: altura de la pantalla.
  - `numCols`: número de columnas del tablero.
  - `nunRows`: número de filas del tablero.
  - `boardPadding`: porción, en tanto por uno, del margen entre la parte interna del tablero y la pantalla.

- `cellPadding`: porción, en tanto por uno, del margen entre cada una de las celdas del tablero y el tamaño de la ficha que la ocupa.
- **public int getRows()** = devuelve el número de filas.
- **public int getColumns()** = devuelve el número de columnas.
- **public GDimension boardDimension()** = devuelve las dimensiones de la parte interna del tablero.
- **public GPoint boardTopLeft()** = devuelve las coordenadas del extremo superior izquierdo de la parte interna del tablero.
- **public GDimension cellDimension()** = devuelve las dimensiones de cada una de las celdas en las que se divide el tablero.
- **public GPoint cellTopLeft(int x, int y)** = devuelve las coordenadas del extremo superior izquierdo de la celda que ocupa la columna x y fila y (comenzando desde 0).
- **public GDimension tokenDimension()** = devuelve las dimensiones de los ovales que representan cada una de las fichas.
- **public GPoint tokenTopLeft(int x, int y)** = devuelve las coordenadas del extremo superior izquierdo de cada uno de los ovales que representan la ficha con columna x y fila y.
- **public GPoint centerAt(int x, int y)** = devuelve las coordenadas del centro de la celda con columna x y fila y.

## 6. Clase Game:

Esta clase representa la situación de la partida, que es básicamente el tablero, el jugador al que le toca el turno de juego y un booleano indicando si el jugador ha ganado la partida.

La diferencia con la clase Board es que Game sí conoce las reglas del juego y, por tanto, es capaz de indicar si una posición es seleccionable como inicio de un movimiento, si otra es seleccionable como fin del mismo y es capaz de realizar un movimiento.

- **public Game (Board board)** = construye una partida y guarda una referencia al tablero sobre el que se jugará. El turno es el del jugador de fichas blancas, para simplificar podemos suponer que ningún jugador ha ganado la partida.
- **public Player getCurrentPlayer()** = devuelve el jugador que tiene el turno de juego.
- **public boolean hasWon()** = devuelve true si el jugador actual ha ganado la partida, false en caso contrario
- **public boolean isValidFrom(Position from)** = indica si la posición es un posible inicio de un movimiento para el jugador actual, es decir, es una posición ocupada que, en una de las dos direcciones posibles, puede avanzar o capturar

- **public boolean isValidTo(Position validFrom, Position to)** = indica si la posición to es una posición de llegada partiendo de la posición validFrom, que se supone válida, es decir, la posición está vacía y es la casilla de destino al avanzar o al capturar desde la posición validFrom.
- **public Move move(Position validFrom, Position validTo)** = dadas las posiciones validFrom y validTo que representan el inicio y final de un movimiento, lo ejecutan modificando el estado del tablero. Ambas posiciones se suponen válidas y devuelve un objeto Move que indica las posiciones involucradas en el movimiento realizado.

## MÉTODOS MÁS COMPLICADOS

En esta sección, proporcionaré un análisis detallado de los métodos auxiliares más desafiantes, detallando mi enfoque inicial para resolverlos, los aspectos que requirieron especial consideración y, por último, describiré la estrategia final que he implementado.

- **public boolean isValidFrom (Position from) =>**

Al principio, subestimé la importancia de este método, lo que conllevó problemas recurrentes en los métodos posteriores. Este en cuestión delega sus procesos a funciones auxiliares, un aspecto que no prioricé hasta después de revisar mi primer borrador, cuando aún había numerosos errores por corregir.

**BORRADOR INICIAL**

Palabra clave => "Posible inicio de movimiento"

1. Tener en cuenta que el tipo de dirección variará según la ficha.

White directions

Black directions

2. Separar los posibles casos:

Caso de un movimiento simple:  
Es necesario para PODER moverme tener **una** celda en vacío, teniendo en cuenta la dirección, ver si la posición de dicha celda es válida.

board.isEmpty(movimiento simple)

Caso captura:  
En este caso, nos encontramos al **oponente en la posición que debería ser un movimiento simple**, ahora tenemos que calcular que la siguiente está vacía, este sería posible movimiento de captura.

board.isBusy (con el oponente) y boardIsEmpty (con mi nueva posición).

Para comenzar, noté que la clase inicializa las direcciones en un array basándose en el color de las fichas. Por lo tanto, creé un array similar que almacena las mismas direcciones correspondientes al color del jugador actual.

Luego, procedí a iterar y determinar las posiciones iniciales. Estas posiciones representan las casillas potenciales para realizar movimientos simples, es decir, suponiendo que estén vacías, y respetando las restricciones de movimiento asociadas a cada color, podríamos movernos.

Sin embargo, no es el único escenario que debemos considerar. Si una casilla contiene una ficha del oponente, debemos avanzar una casilla más para determinar si podemos realizar una captura. Esto se logra mediante un condicional que evalúa esta posibilidad y ajusta la posición en consecuencia, manteniendo la misma dirección que el movimiento simple.

Si encontramos un movimiento posible, la función devuelve true.

Con el objetivo de optimizar este método, decidí trasladar todas las condiciones mencionadas anteriormente a una función auxiliar denominada “*validDirections*”, permitiéndome concentrarme en mejorar algunos detalles que no funcionaban correctamente.

Inicialmente, me enfrenté a problemas en el cálculo de los posibles movimientos de captura, ya que en algunas ocasiones estos se ubicaban fuera de la diagonal del movimiento simple, lo que generaba múltiples inconvenientes. Para abordar este problema, añadí un método auxiliar llamado “*opponentsPiece*”, el cual permite determinar los movimientos de captura según el oponente del jugador actual. Además, decidí emplear un método de la clase “*position*” que había pasado por alto en mi primer borrador, el cual verifica si ambas posiciones se encuentran en la misma diagonal. Después de aplicar estos cambios, el método cumplió con la mayoría de requerimientos pedidos.

Otro cambio importante relacionado con los posibles movimientos de captura fue en el método de la clase “*position*” mencionado anteriormente. En el método “*apply*”, modifiqué la suma de las coordenadas. Si estas se salen de los límites del tablero, se calculará el módulo para evitar errores. Tras este ajuste, el método funcionó correctamente.

Considero que esta función es una de las más difíciles, ya que influye en gran medida en el funcionamiento general del juego. Aunque inicialmente parecía funcionar correctamente, los problemas surgidos en funciones posteriores estaban relacionados con ella. No obstante, decidí intentar priorizar la simplicidad y claridad en la misma. Con el fin de simplificar la función, creé una variable booleana llamada “*hasValidMove*” que se activa únicamente si se cumplen todas las condiciones necesarias, esta variable nos aporta fluidez a la hora de evaluar la función.

- **public boolean isValidTo ( Position validFrom, Position to ) =>**

Considero que este enunciado fue confuso no tanto por su implementación, sino por su planteamiento.

Inicialmente, me costaba diferenciarlo del método anterior. Ambos verificaban si la posición de llegada era válida. Sin embargo, el método “*isValidFrom*” examina si hay un posible punto de partida para el movimiento. Al mismo tiempo, se comprueba si las casillas de destino cumplen con las condiciones necesarias, ya sea para un movimiento simple, donde la casilla debe estar vacía, o para un posible movimiento de captura, donde se necesita una ficha oponente en la casilla inicial y una casilla vacía después de ella para saltar. Por lo tanto, me surgía la duda: ¿qué verificaba “*isValidTo*” que “*isValidFrom*” no hacía?

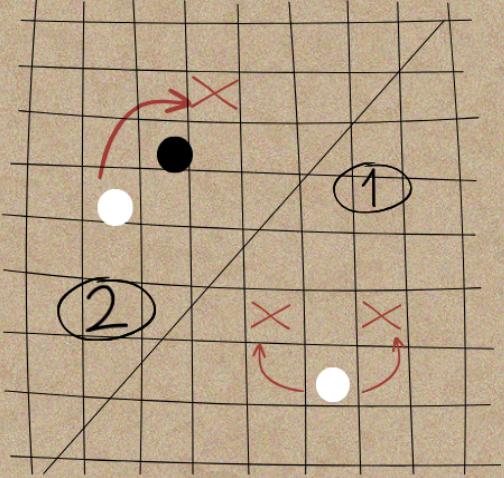
Gracias al ejemplo proporcionado, logré entender el objetivo de este enunciado. Sin embargo, considero que si el planteamiento se hubiese realizado de una manera diferente, la ejecución de este hubiese sido más óptima. En mi implementación para este método me basé en lo aplicado anteriormente en "isValidFrom", pero comparando las posibles casillas con el parámetro "to". No obstante, en este punto de la práctica, me surgió la pregunta de, ¿cuán necesario era esta función?, interrogante que resolvería en el apartado siguiente.

- **public Move move (Position validFrom, Position validTo) =>**

Dada su importancia y extensión, este apartado es crucial. Por ello, mi primer borrador prioriza el uso de funciones auxiliares y una estructura clara.

## BORRADOR INICIAL

Prioridad => Hacer casos, auxiliares y usar el diseño descendente



Estructura del método

- 1 Movimiento simple.
- 2 Movimiento de Captura.
- 3 Winning Detection:
  - Todas Capturadas.
  - Llegar al límite del oponente.
  - No legal moves ?

Comenzamos utilizando los métodos previamente explicados, los cuales deben cumplirse para el correcto funcionamiento del juego, además de un booleano que representa si el jugador ha ganado, del cual haremos uso en la implementación de *Winning Detection*.

El primer escenario que abordamos es el de un movimiento simple. Para su ejecución, invocamos a una función auxiliar llamada "SimpleMove", la cual se encarga de mover la ficha en este movimiento sencillo, trasladándola a su destino y eliminándola de su posición inicial.

Para acceder al segundo escenario, empleamos un condicional que verifica si la distancia entre mi posición inicial y la de destino es mayor que la de un movimiento simple, indicando así un movimiento de captura.

En este caso, calculamos la posición exacta de la pieza de nuestro oponente, que se encuentra entre la posición inicial y la de destino. Luego, llamamos a la función auxiliar "captureMove", la cual, al igual que "simpleMove", moverá nuestra pieza, pero además elimina la pieza del oponente de la posición intermedia.

El primer problema de implementación surgió al detectar que necesitaba tener en cuenta el número de fichas de cada color en el tablero. Sin embargo, no tenía acceso directo a estos atributos de la tabla, ya que eran privados. Esto me llevó a reconsiderar mi estrategia para manejarlos.

Para solucionarlo, decidí modificar la clase “Board”, puesto que estaba utilizando sus métodos que modificaban las celdas del tablero, ya sea vaciándolas o llenándolas después de un movimiento. Cada vez que llamábamos a uno de estos métodos utilizando “set (Color)”, incrementábamos el número de fichas de ese color, mientras que si llamábamos a “setEmpty”, disminuíamos el número de piezas del color correspondiente que estábamos eliminando.

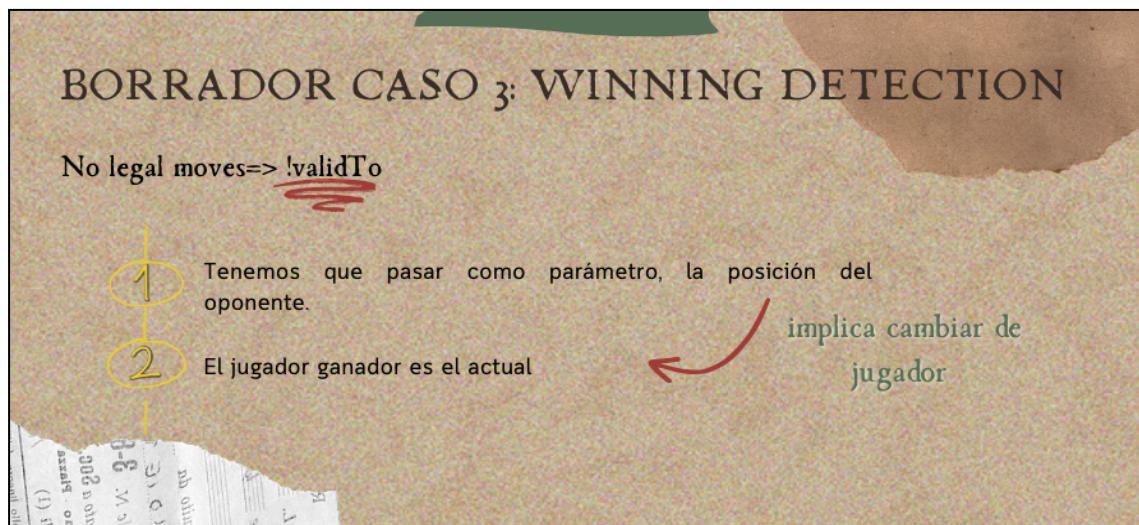
Fue en este punto cuando comprendí la facilidad que aportaba a la resolución haber creado la función “validTo”. Gracias a ella, tenía la capacidad de modificar la posición destino de manera individual, lo cual habría sido mucho más complicado sin esta.

Siguiendo la estructura de mi borrador inicial, el siguiente paso es implementar los criterios para ganar el juego. Los he dividido en 3 casos: primero, ganar al llegar al extremo de la fila del oponente; segundo, capturar todas las fichas del jugador oponente; y por último, verificar si el oponente tiene movimientos posibles. En cuanto a este último caso, no tenía muy claro cómo implementarlo.

La lógica para manejar la posibilidad de llegar al extremo opuesto se basa en primer lugar en llamar a la función auxiliar “winningLimits”. En esta función, utilizamos el jugador actual para determinar si sus coordenadas se encuentran en la fila superior del tablero, en el caso del jugador con fichas blancas, o en la fila inferior, para el jugador con fichas negras. Si ocurre alguno de estos dos casos, la variable que controla al ganador del juego, inicializada al principio de la clase, se establecerá como verdadera, lo que marcará el fin del juego.

Otra condición de victoria alternativa sería capturar todas las fichas de nuestro oponente, y esta es la base del segundo caso. Para su implementación, utilizamos la variable "OpponentPieces", la cual, según el jugador actual, se le asigna el número de piezas en el tablero correspondientes de su oponente. Si este valor es 0, se marca el fin del juego.

El tercer caso requería un poco más de tiempo para su desarrollo. Este caso se presenta cuando el oponente ya no tiene más movimientos disponibles, lo que resulta en la victoria del jugador actual. Mi borrador inicial para esta situación era el siguiente:



Para mejorar el funcionamiento de la clase, he creado una variable booleana privada llamada "moved", que se establecerá como verdadera después de realizar un movimiento, ya sea simple o de captura. Además, he implementado una función auxiliar llamada "switchPlayer", encargada de cambiar al siguiente jugador y reiniciar la variable "moved". Este cambio de jugador es necesario, ya que necesitaremos los parámetros del nuevo jugador.

El condicional verifica si "validTo" es falso y si el jugador tiene solo una pieza. Si estas condiciones se cumplen, se cambia nuevamente al siguiente jugador y se declara como ganador. En caso contrario, el juego continúa.

## CONCLUSIÓN

Personalmente, esta ha sido mi primera experiencia con un proyecto de esta magnitud, siendo aún una principiante en el mundo de la programación. Encontré esta práctica sumamente interesante, ya que me enseñó la importancia de abordar los problemas con optimismo, incluso cuando parecen carecer de solución inicialmente. Aprendí que ante la dificultad inicial, es fundamental buscar diferentes enfoques o recursos que nos ayuden a encontrar una solución.

Un ejemplo claro de esto fue el manejo de las fichas en el tablero. Debido al encapsulamiento de estas variables y otros métodos, tuve que cambiar mi enfoque a uno alternativo para lograr la funcionalidad deseada.

Si tuviera la oportunidad de comenzar este proyecto nuevamente, no subestimaría la importancia de las funciones que en apariencia parecen ser simples. En su lugar, consideraría los posibles casos de error que estas podrían implicar. Creo firmemente que adoptar este enfoque ampliaría mi nivel de conciencia al estructurar un problema y me permitiría abordarlo de manera más eficiente en el futuro.

## BIBLIOGRAFÍA

- *GDimention* . (s. f.).  
<https://cs.stanford.edu/people/eroberts/jtf/javadoc/student/index.html>
- *GPoint*. (s. f.).  
<https://cs.stanford.edu/people/eroberts/jtf/javadoc/student/acm/graphics/GPoint.html>