6.5 Orientación a objetos.

Debido a la naturaleza del proyecto, la orientación a objetos ha sido parte fundamental del mismo así como el polimorfismo a la hora de mantener un código limpio y efectivo.

6.5.1 Piezas

El programa consta de 48 piezas distintas y para facilitar la claridad del código todas estas heredan de la clase abstracta Pieza. A su vez esta clase Pieza implementa la interfaz FuncionesAuxPiezas.kt donde hay funciones getter (Esto se debe a que el programa base se hizo con java y luego se pasó a kotlin y así se ha mantenido, además considero que queda la estructura con más claridad con esos getters).

FuncionesAuxPiezas.kt:

```
interface FuncionesAuxPiezas {  Pruden *
    fun getForma(): Array<Array<IntArray>>  Pruden
    fun getColumnaCentro(): Int  Pruden
    fun getFilaCentro(): Int  Pruden
    fun getNumpieza(): Int  Pruden
    fun getCentro(): Int  Pruden
    fun getColor(): Color  Pruden
}
```

La clase abstracta Pieza se instancia con los siguientes atributos los cuales heredan todas las piezas:

```
Pieza(open var fila: Int, open var columna: Int,   Pruden *
      open var orientacion : Int, open var condicionEspecial_b: Boolean)
```

Además esta clase dará los siguientes métodos a las hijas:

```
abstract fun rotar(): Boolean  Pruden
abstract fun limpiar()  Pruden
abstract fun pintar()  Pruden
abstract fun clonar(): Pieza  Pruden

abstract fun bajar(): Boolean  Pruden
abstract fun derecha(): Boolean  Pruden
abstract fun izquierda(): Boolean  Pruden
abstract fun puedeRotar(nuevaOrientacion: Int) : Boolean  Pruden
```

Veamos cómo es la implementación real de una clase Pieza ya que el esqueleto de una es prácticamente el mismo para todas:

Tenemos la clase Pieza_A, de primeras vemos que en la declaración de herencia, a la hora de instanciar la pieza recibe fila y columna y luego orientación y condicionEspecial_b tienen valores por defecto.

```
class Pieza_A (override var fila: Int, override var columna: Int,  Pruden
               override var orientacion : Int = 0, override var condicionEspecial_b : Boolean = false)
    : Pieza(fila, columna, orientacion, condicionEspecial_b) {
```

Cuando instanciamos cada pieza es importante que el valor de la fila y columna no sea por defecto pues depende la pieza y el tablero hay veces que hay que hacer ajustes para que no cause errores, por ejemplo en la Pieza_E:

```
bolsaTemp.add(Pieza_E( fila: -1,  columna: columnaInicial-1))
```

Es necesario realizar este ajuste en el tablero de 12x8 ya que la columna Inicial es el centro y si empieza en el centro al ser una pieza 5x5 se sale del tablero provocando un error, entonces la desplazamos -1 en la columna.

Continuando nuestro periplo por la Pieza_A encontramos los siguientes atributos que definen las casillas de la matriz interna que se encarga de gestionar el tablero y las piezas:

```
private val CENTRO = 298
private val NUMPIEZA = 29
private val COLOR = Color.web("#ff6c58")
```

Y las formas de esa pieza:

En este caso como tiene 4 orientaciones tiene 5 formas, las 4 primeras corresponde a sus 4 posibles estados 0º, 90º, 180º y 270º y la última y quinta en este caso corresponde a la forma en la que tiene que dibujarse en los tableros secundarios. (Esto último es útil para unificarlo todo con esa lógica ya que sino habría que hacerlo de una manera más tediosa para cada pieza, así nos aseguramos la forma correcta en estos tableros de 5x5)

Algunas piezas con una simetría parcial tendrán solo 3 formas, 0º, 180º y la de la forma, cómo las piezas Z_v2, S_v2, Z_v3, S_v3 o Twin O.

```
private val FORMAS_A = arrayOf(
    arrayOf(
        intArrayOf(BLANCO, NUMPIEZA, BLANCO),
        intArrayOf(NUMPIEZA, CENTRO, NUMPIEZA),
        intArrayOf(NUMPIEZA, BLANCO, NUMPIEZA)
    ),
    arrayOf(
        intArrayOf(NUMPIEZA, NUMPIEZA, BLANCO),
        intArrayOf(BLANCO, CENTRO, NUMPIEZA),
        intArrayOf(NUMPIEZA, NUMPIEZA, BLANCO)
    ),
    arrayOf(
        intArrayOf(NUMPIEZA, BLANCO, NUMPIEZA),
        intArrayOf(NUMPIEZA, CENTRO, NUMPIEZA),
        intArrayOf(BLANCO, NUMPIEZA, BLANCO)
    ),
    arrayOf(
        intArrayOf(BLANCO, NUMPIEZA, NUMPIEZA),
        intArrayOf(NUMPIEZA, CENTRO, BLANCO),
        intArrayOf(BLANCO, NUMPIEZA, NUMPIEZA)
    ),arrayOf(
        intArrayOf(BLANCO, BLANCO,BLANCO, BLANCO,BLANCO),
        intArrayOf(BLANCO, BLANCO,NUMPIEZA, BLANCO,BLANCO),
        intArrayOf(BLANCO, NUMPIEZA,NUMPIEZA, NUMPIEZA,BLANCO),
        intArrayOf(BLANCO, NUMPIEZA,BLANCO, NUMPIEZA,BLANCO),
        intArrayOf(BLANCO, BLANCO,BLANCO, BLANCO,BLANCO)
    )
```

Y otras piezas tendrá con una simetría total tendrán solo 2 formas, 0º y la de la forma, cómo las piezas O, O_v6, o X.

Justo debajo encontramos un companion object el cual sirve para que a la hora de pintar la matriz completa según el número se pinte con un color u otro:

```
companion object {  Pruden
    const val NUMPIEZA_A = 29
    const val CENTRO_A = 298
    val COLOR_A = Color.web("#ff6c58")!!
}
```

Como apunte esto podría estar en un fichero de constante pero veo más práctico que cada pieza incorporé sus propias constantes en su companion object.

Luego tenemos las funciones limpiar y pintar que su implementación es la misma para todas las piezas:

```
override fun limpiar() {  Pruden
    limpiarPieza( pieza: this)
}

override fun pintar() {  Pruden
    val filaColumna = pintarPieza( pieza: this)
    filaCentro = filaColumna[0]
    columnaCentro = filaColumna[1]
}
```

La función rotar que recibirá un parámetro dependiendo de las posibles orientaciones de la pieza o incluso si no tiene más que una orientación simplemente devolverá true:

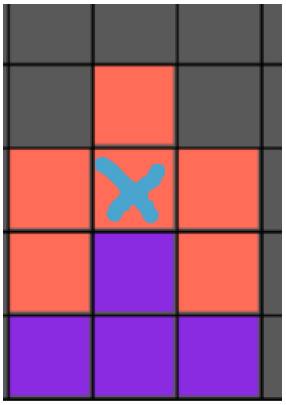
```
override fun rotar(): Boolean {  Pruden
    return rotarNormal( pieza: this, mod: 4)
}
```

La siguiente función, puedeRotar, es sin duda la más compleja y más específica de cada pieza, esta se encarga de comprobar si la pieza puede rotar a la siguiente posición y en caso negativo calcula si con un desplazamiento de fila o de columna puede rotar:

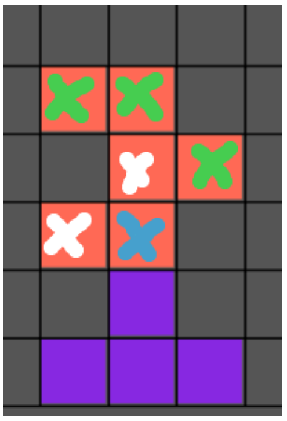
```
override fun puedeRotar(nuevaOrientacion: Int): Boolean {  Pruden
    condicionEspecial_b = false
    return if (nuevaOrientacion == 0) {
        if (matrizNumerica[filaCentro][columnaCentro + 1] == BLANCO
            && matrizNumerica[filaCentro + 1][columnaCentro - 1] == BLANCO) {
            true
        } else (condicionRotarEspecial( pieza: this, intArrayOf(0,0,1), intArrayOf(1,2,2),  columnaB: true,  movimiento: 1)
                || condicionRotarEspecial( pieza: this, intArrayOf(0,-1,-2), intArrayOf(1,-1,0),  columnaB: false,  movimiento: -1))
    } else if (nuevaOrientacion == 1) {
```

Fragmento de la función puedeRotar() de Pieza_A.

Para ello nos apoyamos también la función condicionRotarEspecial() y en el atributo condicionEspecial_b. Gracias a esta función y condición es posible esa rotación especial veamos un par de ejemplos de ese tipo de rotación especial:



Img 1



Img 2

En este caso tenemos la Pieza_A con una orientación = 0 y vemos que se queda como encajada en la pieza T de abajo, nuestra Pieza_A busca girar a la orientación 1 pero rápidamente vemos que esta condición se lo impide:

```
} else if (nuevaOrientacion == 1) {
    if (matrizNumerica[filaCentro - 1][columnaCentro - 1] == BLANCO
        && matrizNumerica[filaCentro + 1][columnaCentro] == BLANCO) {
        true
    } else {
        if (matrizNumerica[filaCentro + 1][columnaCentro] != BLANCO) {
            condicionRotarEspecial( pieza: this, intArrayOf(-1,-2,-2), intArrayOf(1,-1,0),  columnaB: false,  movimiento: -1)
        } else condicionRotarEspecial( pieza: this, intArrayOf(1,-1,0), intArrayOf(-1,0,1),  columnaB: true,  movimiento: 1) ||
                condicionRotarEspecial( pieza: this, intArrayOf(2,2,1), intArrayOf(-1,0,0),  columnaB: true,  movimiento: 1)
    }
```

Ya que la pieza comprueba que la casilla (1, 0) respecto el centro de la Pieza_A (es la cruz de la Img 1) está ocupada, por lo tanto entra en else y luego dentro del if que comprueba que justo ese casilla esté ocupada (!= BLANCO) y una vez ahí aplica la condición especial.

Analizando la Img 2, encontramos que el centro relativo en el cual se hacen los cálculos es la cruz azul (Ya que es el centro de la Pieza_A con orientación 0 originalmente). Por lo tanto ya sabemos respecto a qué casilla comprobar si el resto están libres para poder rotar la pieza. Para dar con estas casillas vemos que las que están en blanco ya estaban ocupadas por la pieza original, pues entonces no es necesario comprobarlas. Luego nos quedan las verdes que en este caso no haría falta comprobarlas todas ya que todas no se pueden cumplir juntas pero por separado sí por lo tanto para asegurar que se cumple cogemos las tres.

La función condicionRotarEspecial() recibe la instancia de la propia pieza, una lista de enteros con las coordenadas de las filas que hay que comprobar y otro con las coordenadas de las columnas, la posición 0 de uno y 0 del otro nos forman la primera casilla y así con el resto de posiciones de los dos arrays.

Por ejemplo para la condición especial que estamos analizando:

```
condicionRotarEspecial( pieza: this, intArrayOf(-1,-2,-2),
    intArrayOf(1,-1,0), columnaB: false, movimiento: -1)
```

Vemos que formaría las casillas (-1,1), (-2,-1) y (-2,0) que partiendo del centro relativo marcado en la Img 2 corresponden con las cruces verdes, entonces esta función comprueba que está libre y si lo verifica con los parámetros siguientes false y -1, que significa que se tiene que desplazar en la fila y -1 obtenemos el resultado de Img 1 a Img2.

Nota: los parámetros columnaB el cual es booleano indica si el desplazamiento es de columna (si es true) o de fila. Y el movimiento indica la magnitud de ese desplazamiento.

La función en cuestión:

```
fun condicionRotarEspecial(pieza : Pieza, dFilas : IntArray,  Pruden *
                           dColumnas : IntArray, columnaB: Boolean,
                           movimiento: Int): Boolean {
    with(pieza) {
        var condicion = true
        for (i in dFilas.indices){
            if(matrizNumerica[getFilaCentro() + dFilas[i]][getColumnaCentro() + dColumnas[i]] != Globales.BLANCO){
                condicion = false; break
            }
        }
        return if (condicion) {
            limpiar()
            if (columnaB) {
                columna += movimiento
            } else fila += movimiento
            condicionEspecial_b = true
            true
        } else false
    }
}
```

Podemos observar que en esta función el atributo de la pieza condicionEspecial_b pasa a ser true, ¿y esto por qué? Esto es necesario para indicarle a la función rotalNormal() que esa pieza ya ha sido “limpiada” por la función limpiar().

Esto es necesario para evitar errores gráficos con la pieza cuando ha habido una rotación especial.

```
fun rotarNormal(pieza : Pieza, mod : Int): Boolean {  Pruden *
    with(pieza){
        val nuevaOrientacion = (orientacion + 1) % mod
        if (puedeRotar(nuevaOrientacion)) {
            if (!condicionEspecial_b) {
                limpiar()
            }
            orientacion = nuevaOrientacion
            pintar()
            return true
        }
        return false

    }
}
```

Continuando nuestra travesía por la Pieza_A encontramos las funciones que se encargan del desplazamiento de la misma:

```
override fun bajar(): Boolean {  Pruden *
    return bajar3x3( pieza: this, intArrayOf(2, 2, 1, 1, 2, 2,  //R0
                                             1, 1, 2, 2, 1, 1,  //R2
                                             0, 2, 2, 2, 1, 1,  //R1
                                             1, 1, 2, 2, 0, 2)) //R3
}
```

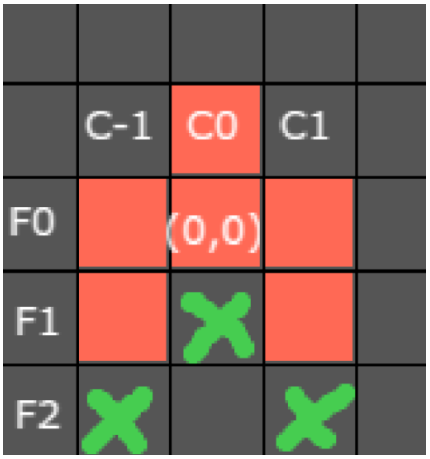
Todas las piezas tiene la acción de bajar esta se ejecuta cada x tiempo por el timeline de la partida que se encarga de hacer que estas desciendan, además el usuario también puede moverlas. Lo primero de todo es saber que devuelve un booleano que determina si esa pieza puede bajar, si se ejecuta la función y devuelve false quiere decir que la pieza no puede bajar y por lo tanto se queda fija y se genera una nueva.

Un aspecto fundamental para comprender el funcionamiento de las piezas y sus movimientos es tener en cuenta la dimensión de la matriz que forman sus casillas.

En este caso es una matriz de 3x3, pero hay otras de 3x2, 2x2, 4x4, 4x1, 5x3 y algunas más, dependiendo de estas dimensiones implementan un método bajarNxM y cada uno tiene sus diferencias con los otros.

Lo primero de todo es ver los parámetros que recibe el método bajar, la instancia de la pieza y un array de enteros que representan las casillas que hay que tener en cuenta respecto el centro de la pieza a la hora de bajar la pieza, veámoslo con un ejemplo:

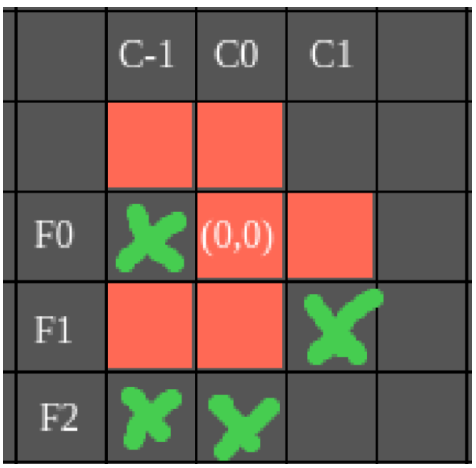
Cuando la Pieza_A está en rotación 0 (R0 en la imagen de la función arriba) recibe los parámetros 2,2,1,1,2,2 que a efectos prácticos nos podemos quedar con 2,1,2 ya que por la naturaleza de las piezas de 3x3 hay que “duplicar” los valores que recibe (esto será explicado más adelante).



Podemos ver que las casillas que hay que comprobar son las cruces verdes, si todas están libres la pieza podrá bajar.

Luego C-1, C0 y C1 representan las columnas respecto al centro de la pieza, igual que F0, F1 y F2 las filas.

Ahora se entiende mejor qué es lo que recibe la función, recibe LAS FILAS donde ha de comprobar que esté libre, las columnas ya las tiene internamente la función ya que estas en la pieza 3x3 siempre serán -1, 0 y 1.



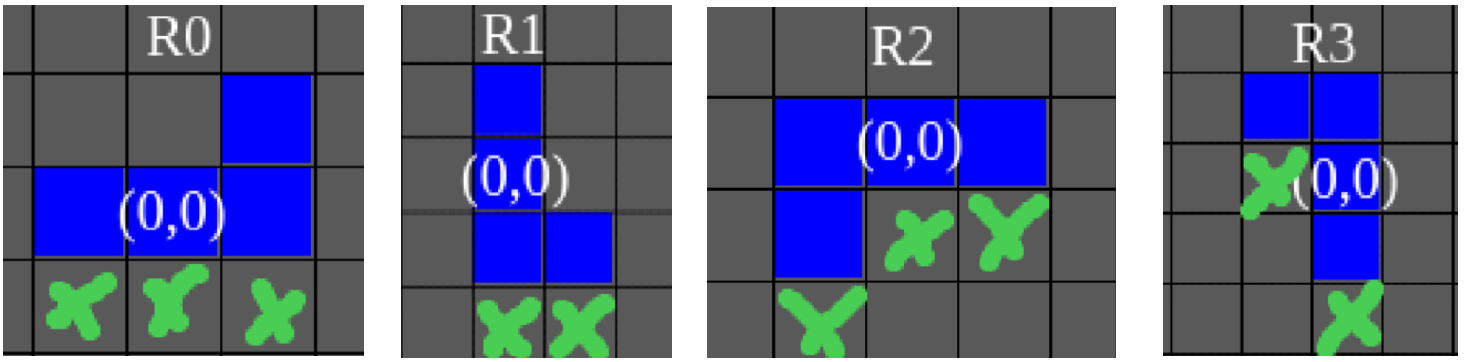
¿Y entonces, por qué se duplican los valores recibidos? Bien con la pieza A en su rotación 1 se ve muy claro:

Aquí vemos que a la hora de bajar en la columna -1 hay que comprobar una casilla más, es por ello que

los valores duplicados son necesarios, cuando los hay quiere decir que solo hay que comprobar un casilla, eso facilita que la función bajar3x3 sea más universal y abarque a todas las piezas de 3x3.
Y es por ello que para la rotación 1 recibe los valores 0,2,22,1,1.

Otro apunte esencial es que los valores respecto a la rotación actual que recibe la pieza no van en orden R0, R1, R2, R3 sino que siguen el orden R0, R2, R1, R3, esto se debe a que originalmente para las piezas 3x2 (las primeras en programarse) en las funciones de bajar, se recogían primero los valores de las rotaciones 2x3 y luego las 3x2, es decir R0, R2 y posteriormente los valores de R1 y R3.

Esto se ve claramente en la Pieza_L (3x2):




```
puedeBajar =
    matrizNumerica[getFilaCentro() + desplazamientos[0]][getColumnaCentro() - 1] == Globales.BLANCO
            && matrizNumerica[getFilaCentro() + desplazamientos[1]][getColumnaCentro() - 1] == Globales.BLANCO
            && matrizNumerica[getFilaCentro() + desplazamientos[2]][getColumnaCentro()] == Globales.BLANCO
            && matrizNumerica[getFilaCentro() + desplazamientos[3]][getColumnaCentro()] == Globales.BLANCO
            && matrizNumerica[getFilaCentro() + desplazamientos[4]][getColumnaCentro() + 1] == Globales.BLANCO
            && matrizNumerica[getFilaCentro() + desplazamientos[5]][getColumnaCentro() + 1] == Globales.BLANCO
```

Sabiendo esto vamos a ver ahora el funcionamiento interno de la función bajar3x3 para R0:
Se comprueba primero la columna -1, luego la 0 y luego la 1 (Cada una 2 veces para abarcar esos casos que hemos hablado antes)
Y para cada rotación comrpueba los desplazamientos (el array que recibe) con las columnas si la casilla está libre.

```
if (puedeBajar) {
    limpiar()
    fila = (fila + 1)
    pintar()
} else {
    borrarLinea()
    comprbarPerder()
    return true
}
return false
```

En caso de que pueda bajar, limpia la pieza, incrementa en 1 su fila y la vuelve a pintar, sino puede bajar comprueba si debe borrar la línea y luego comprueba si el usuario ha perdido.

Y este es el funcionamiento general de todas las piezas al bajar.

Esta función también es usada cuando el usuario presiona el espacio, tecla encargada de bajar la pieza todo lo que pueda usando un while y aprovechando que devuelve true si puede bajar y false si no:

```
fun moverPiezaAbajo() :Boolean{   Pruden
    if(piezaActual.bajar()){
```

```
fun moverEspacio(event : KeyEvent){   Pruden
    event.consume()
    while (moverPiezaAbajo()) {
```

Los siguientes movimientos son, para desplazarse a los lados, derecha e izquierda:

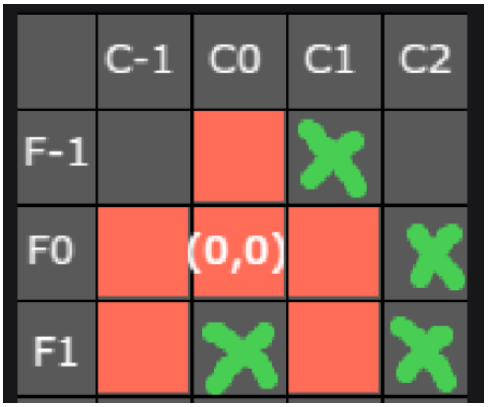
```
override fun derecha(): Boolean {   Pruden *
    return moverDerechaIzquierda_3x3( pieza: this, intArrayOf(1, 1, 2, 2, 0, 2,
                                                              0, 2, 2, 2, 1, 1,
                                                              1, 1, 2, 2, 1, 1,
                                                              2, 2, 1, 1, 2, 2),
                                                    direccion: 1)
}
```

```
override fun izquierda(): Boolean {   Pruden *
    return moverDerechaIzquierda_3x3( pieza: this, intArrayOf(-1, -1, -2, -2, 0, -2,
                                                               0, -2, -2, -2, -1, -1,
                                                               -2, -2, -1, -1, -2, -2,
                                                               -1, -1, -2, -2, -1, -1),
                                                     direccion: -1)
}
```

Prácticamente todas las matrices usan la misma función tanto para derecha como para izquierda y lo que cambia es el valor final que es la dirección, negativo para izquierda y positivo para derecha.

Para bajar lo que le pasábamos eran las filas ya que las columnas ya las comprobaba antes, aquí al revés, le pasamos las columnas porque en la función ya tiene las filas predeterminadas, veamos un ejemplo de desplazamiento de la Pieza_A a la derecha en R0:

Las cruces verdes determinan las casillas que deberían de estar libres para poder desplazarse a la derecha.



La función moverDerechaIzquierda_3x3() comprueba F-1, F0 y luego F1 cada una 2 veces para cubrir condiciones cómo en este caso el de la casilla (1,0).

Viendo el desplazamiento a la derecha, a la izquierda es igual solo que cambiando las casillas que hay que controlar y la dirección a -1.

Así es cómo se ven las comprobaciones de moverDerechaIzquierda_3x3():

```
puedeMoverse = matrizNumerica[getFilaCentro() - 1][getColumnaCentro() + desplazamientos[0]] == Globales.BLANCO
        && matrizNumerica[getFilaCentro() - 1][getColumnaCentro() + desplazamientos[1]] == Globales.BLANCO
        && matrizNumerica[getFilaCentro()][getColumnaCentro() + desplazamientos[2]] == Globales.BLANCO
        && matrizNumerica[getFilaCentro()][getColumnaCentro() + desplazamientos[3]] == Globales.BLANCO
        && matrizNumerica[getFilaCentro() + 1][getColumnaCentro() + desplazamientos[4]] == Globales.BLANCO
        && matrizNumerica[getFilaCentro() + 1][getColumnaCentro() + desplazamientos[5]] == Globales.BLANCO
```

Y siempre y cuando se pueda mover se desplazará la columna de la pieza en la dirección que se le ha pasado.

```
if (puedeMoverse) {
    limpiar()
    columna = (columna + direccion)
    pintar()
}
```

Otro detalle, es que la función si puede moverse devuelve true y si no false, esto no sería necesario de no ser por la mecánica dash que actúa como el espacio pero para los lados, este movimiento se produce cuando el usuario mueve la pieza a la derecha o izquierda pulsando además el Control del teclado. Es un mecánica que no está disponible en todos los modos de juego y en código funciona con un while al igual que el espacio y desplaza la pieza todo lo que pueda a la derecha o a la izquierda:

```
fun dashDerecha(){  Pruden
    while (piezaActual.derecha()){}
}
```

```
fun dashIzquierda(){  Pruden
    while (piezaActual.izquierda()){}
}
```

Posteriormente en una clase Pieza encontramos los siguientes getters (los cuales vienen de la interfaz antes mencionada)

```
override fun getForma(): Array<Array<IntArray>>  {
    return FORMAS_A
}

override fun getColumnaCentro(): Int {  Pruden
    return columnaCentro
}

override fun getFilaCentro(): Int {  Pruden
    return filaCentro
```

```
override fun getNumpieza(): Int {
    return NUMPIEZA
}

override fun getCentro(): Int {
    return CENTRO
}

override fun getColor(): Color {
    return COLOR
}
```

Estos getters son muy importantes, ya que el polimorfismo está muy presente y es esencial para el manejo de las piezas, pues yo tengo una instancia Pieza, y cada una tiene sus propias características, pues el polimorfismo me permite acceder a los elementos de esa pieza. Por ejemplo en la función limpiarPieza, que recibe un objeto de tipo Pieza:

```
fun limpiarPieza(pieza: Pieza){  Pruden
    val forma = pieza.getForma()[pieza.orientacion]
    for (i in forma.indices) {
        for (j in forma[i].indices) {
            if (forma[i][j] == pieza.getNumpieza() || forma[i][j] == Globales.NUMPIEZA_PICO){
                matrizNumerica[pieza.fila + i][pieza.columna + j] = Globales.BLANCO
                pintarCasilla(FONDO, (pieza.fila + i).toDouble(), (pieza.columna + j).toDouble())
            }else if (forma[i][j] == pieza.getCentro() && pieza.getCentro() != Globales.CENTRO_BLANCO) {
                matrizNumerica[pieza.fila + i][pieza.columna + j] = Globales.BLANCO
                pintarCasilla(FONDO, (pieza.fila + i).toDouble(), (pieza.columna + j).toDouble())
            }
        }
    }
}
```

Vemos que podemos acceder a las funciones de arriba ya que Pieza implementa la interfaz de los getters.

Por último al final de una clase pieza encontramos la función Clonar:

Esta función es necesaria para el modo PVP.

```
override fun clonar(): Pieza {  Pruden
    return Pieza_A(this.fila, this.columna)
}
```