Universidad Nacional de
Educación a Distancia

Práctica de Programación Orientada a Objetos

Junio 2011

Nombre: Manuel Ángel Rubio Jiménez
DNI: 30.956.906-X
Email: manuelr@bosqueviejo.net

Universidad Nacional de
Educación a Distancia

Contenido

Universidad Nacional de
Educación a Distancia

Introducción

Este documento responde al requisito de documentación presente en la práctica de la asignatura de Programación Orientada a Objetos, impartida en los estudios de Grado en Ingeniería Informática de la UNED.

La práctica consiste en implementar el juego Tetris, en su nivel más básico, y empleando una estructura orientada a objetos en la que se emplee herencia, agregación, composición y otros elementos propios de la orientación a objetos.

Universidad Nacional de
Educación a Distancia

Inicio Rápido

Como la evaluación de la práctica consiste, principalmente, en que funcione. En este apartado voy a dar instrucciones básicas de cómo hacer la ejecución de la práctica en todos los ámbitos en los que se ha probado.

Ant

Lo más simple, es entrar a nivel de consola y ejecutar *ant*, este sistema (similar a Makefile de UNIX), se encarga de construir la aplicación en dos niveles, la compilación de todas las clases en un directorio *build* y la creación de un fichero de distribución en *dist* llamado *tetris.jar*.

BlueJ

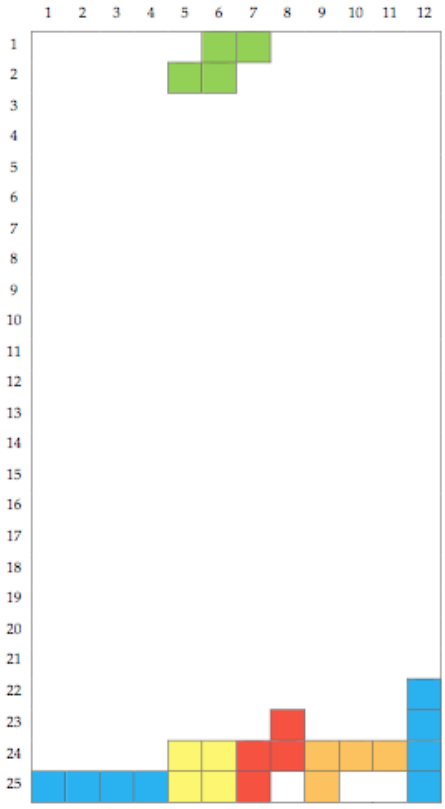
La práctica ha sido creada en BlueJ, y se han mantenido los ficheros de proyecto, por lo que se puede abrir, en el directorio *src*, a través del fichero de proyecto, en el programa BlueJ la práctica.

Para ejecutar, tenemos dos posibilidades:

- **Tetris**: ejecutar el método *main* de esta clase, para lanzar la aplicación de escritorio.
- **Web**: ejecutar el applet a través de la opción *Run Applet*.

Universidad Nacional de Educación a Distancia

Enunciado y Análisis de la Práctica



La práctica del presente curso va a ser diseñar e implementar una versión del conocido juego Tetris. Esto servirá para estudiar y practicar los mecanismos de la programación Orientada a Objetos.

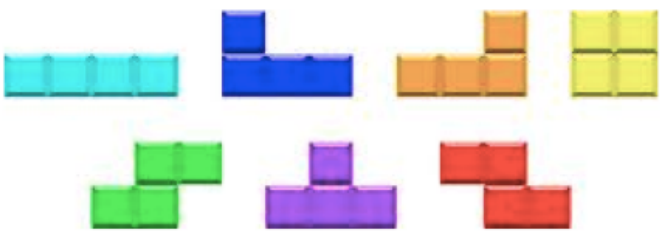
Historia

Tetris es un juego desarrollado por Alekséi Pázhitnov. Este juego se basaba en otro previo que se llamaba pentaminós. El número de cuadros que compone cada una de las piezas (siempre constante a cuatro) es lo que da nombre al juego (tetra, de cuatro, da el nombre de tetris). Su éxito vino con su portabilidad a IBM PC y a otras plataformas, como Apple II, Commodore 64, Atari ST y Sinclair ZX Spectrum.

Tetris ha sido históricamente uno de los videojuegos más versionados. Después del éxito de este juego, muchos otros trataron de imitarlo. Existe incluso una versión gratuita en muy popular en Internet denominada TetriNET, que proporciona una versión multijugador en arquitectura cliente-servidor en el que se pueden enfrentar a través de la red de 2 a 6 personas con la posibilidad de crear equipos.

Mecánica del Juego

Distintos tetriminos, figuras geométricas compuestas por cuatro bloques cuadrados unidos de forma ortogonal, caen de la parte superior de la pantalla. El jugador no puede impedir esta caída pero puede decidir la rotación de la pieza (0º, 90º, 180º y 270º) y en qué lugar debe caer. Cuando una línea horizontal se completa, esa línea desaparece y todas las piezas que están por encima descienden una posición, liberando espacio de juego y por tanto facilitando la tarea de situar nuevas piezas. La caída de las piezas se acelera progresivamente. El juego termina cuando las piezas se amontonan hasta salir del área de juego.



Para simplificar el juego, se van a hacer las siguientes suposiciones:

- **[R1]** La pantalla está formada por 25 líneas (contadas desde la parte inferior a la parte superior).
- **[R2]** Puesto que las piezas caen de la zona superior, y aparecen inicialmente en el formato que se muestra en la figura, el área de juego efectivo son 23 líneas (restando dos líneas por la colocación inicial de las piezas).
- **[R3]** El ancho de la pantalla será de 12 casillas.
- **[R4]** Una pieza nueva no aparece hasta que no ha caído otra.
- **[R5]** No se produce aceleración en la caída de las piezas a lo largo del juego. Estas caerán a una velocidad de 1 fila por segundo, de tal manera que una pieza tarda 23 segundos en caer de la parte superior a la inferior, si no se producen rotaciones.
- **[R6]** Como máximo el juego deja hace cuatro rotaciones por segundo.
- **[R7]** Las piezas se desplazarán a izquierda y derecha con los cursores, y **[R8]** rotarán en un solo sentido (en el de las agujas del reloj), mediante la flecha superior de los cursores.
- **[R9]** No se habilitará la opción de dejar caer la pieza como ocurre en el juego original.
- **[R10]** No se establecerá sistema de puntuaciones.
- **[R11]** No se establecerán niveles diferentes en el juego.
- **[R12]** Las piezas siempre aparecen por la parte superior y central del tablero.

Ampliando requisitos con supuestos

La lista de requisitos es algo escasa, por lo que, para tener presente cada elemento, se hará una descripción de lo que se desarrollará con todos los supuestos que se han entendido, deben estar presentes en la práctica, y los que se han expuesto de forma explícita.

El juego de tetris se basa en la definición ya mencionada anteriormente de: *la consecución de caída de* **[R13]** *piezas de cuatro bloques dispuestas de 7 formas diferentes, que pueden* **[R8]** *rotarse y* **[R7]** *desplazarse lateralmente hasta que termine su caída.* **[R4]** *Cuando una pieza se ha posado, aparece otra, y así hasta que ya no hay posibilidad de que aparezcan nuevas piezas. Igualmente,* **[R14]** *cuando una línea horizontal se completa, la línea desaparece y todos los bloques superiores bajan una posición*.

Con esto tenemos la definición a alto nivel, que se completa con algunos requisitos vistos antes. No obstante, quedan algunas preguntas en el aire:

- **[R15]** ¿Cómo rotamos las piezas? En principio, tomaré como suposición que la rotación de la pieza en 90, 180 y 270 grados, tienen que resultar en que la primera fila y la primera columna tiene que tener, al menos, un bloque, es decir, que la figura de la recta, debe de estar en posición inicial completando en la posición X de la pieza en el tablero, e igualmente en su rotación de 180 grados; en cambio, cuando se rote 90 ó 270 grados, debe ocupar los cuatro bloques a partir de la posición Y relativa al tablero.

- **[R16]** Dos bloques no pueden ocupar el mismo espacio, por lo que, en los movimientos de giro, desplazamiento a izquierda y derecha y el movimiento de caída, se debe de tener en cuenta que no se produzca el caso de que dos bloques ocupen el mismo espacio.
- **[R17]** Límites del tablero. Ningún bloque puede dibujarse fuera de los límites laterales e inferior del tablero, por lo que se tendrán que controlar en el giro, los desplazamientos laterales y la caída, que no se salga ningún bloque del tablero.

Universidad Nacional de
Educación a Distancia

Diseño de la Práctica

La práctica se ha realizado con idea de emplear, en lo máximo posible, la orientación a objetos, tal y como se pedía en el enunciado de la misma, por lo que, centrándonos en el diseño que se puede ver en la figura siguiente, podemos llegar a ver cuatro zonas centrales en las que el código se ha concentrado, y la potencia de la orientación a objetos que ha hecho que el código, en muchas áreas sea extensible y fácil.

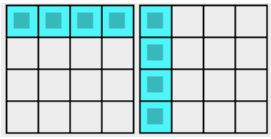
La Idea de las Piezas

Cada pieza se dibuja sobre el tablero en un color predefinido que, para cada pieza es diferente, pero igual entre los mismos tipos de piezas. Por la cantidad de aspectos que hay tan similares entre las piezas, se decidió unificar en una clase abstracta todo este código similar.

Por tanto, se crea una clase por cada tipo de pieza. **[R13]** Hay siete piezas en total, por lo que habrá siete clases que hacen referencia a cada una de las piezas. La definición de cada clase tiene, además de la pieza en sí, sus giros **[R15]**.

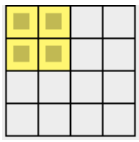
Recta

La pieza de línea recta presenta dos formas. Se ha hecho de forma que, las rotaciones máximas que ofrece son dos, ya que la rotación de 0º y 180º son iguales, al igual que la de 90º y 270º. Por lo tanto, en la matriz presentará la forma adjunta.



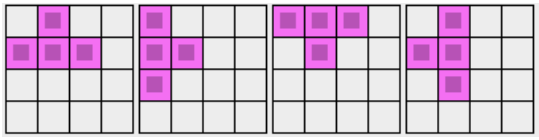
Cuadrado

La figura del cuadrado, sin embargo, no tiene nada más que una forma, ya que, aunque se girase, siempre permanecería en forma de cuadrado. Por lo tanto, esta figura solo presenta una rotación que se emplea para el formato de 0º, 90º, 180º y 270º.



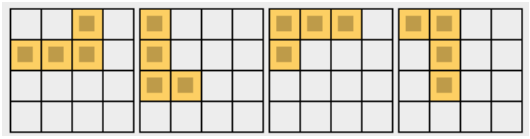
Te

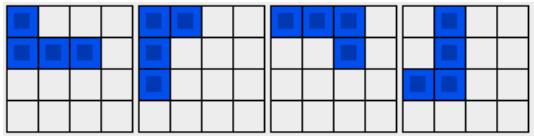
La figura en forma de T posee las cuatro formas según su posición. La rotación de la figura se ha realizado, como se mencionó en los supuestos, realizando la rotación de la figura y ajustándola al margen superior izquierdo, con lo que la forma en la que queda dicha figura en la adjunta.



Ele

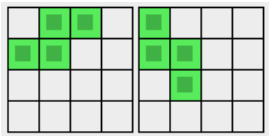
Las figuras en forma de L, tanto hacia izquierda como a derecha (o L invertida y normal), también disponen de cuatro formas al ser giradas. La forma en la que se visualiza cada una de ellas se puede ver de en las siguientes figuras adjuntas:

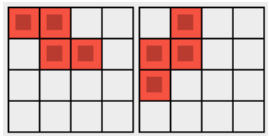




Zeta

El último tipo de figuras son las que si visualizan en forma de Z. Estas figuras, por su forma, y al girarse, tan solo presentan dos rotaciones posibles, que son las que se presentan en las figuras:





Generación de Piezas

Como puede verse en los gráficos anteriores, todas las piezas, en su posición inicial ocupan de alto 2 bloques como máximo, el caso de la recta es una excepción al **`[R2]`**, no obstante, se cumple en el resto de piezas.

El haber pensado en la clase Pieza, de forma abstracta, da pie a generar métodos tanto del objeto como de clase, que facilitan mucho la gestión de las piezas, y cosas como la creación. Para ello, se empleó un patrón de diseño llamado *factory*, que nos facilita la instanciación de las clases hijas, de forma aleatoria. Para ello se ha hecho:

```
private static Random random = new Random(new Date().getTime());
```

Esto nos asegura que la *semilla* será diferente para la mayoría de veces que se inicie el juego.

La instanciación nos plantea un problema, y es el hecho de que Java es fuertemente tipado y esto tan solo se puede resolver mediante *reflection*. En sí, el beneficio al principio, por mantener la jerarquía de clases y agregar una facilidad bastante interesante, se implementa el sistema de reflection y todo bien. Un ejemplo:

```
private static Class[] piezas = {
        EleIzquierda.class,
        EleDerecha.class,
        Cuadrado.class,
        Recta.class,
        Te.class,
        ZetaIzquierda.class,
        ZetaDerecha.class
    };
[...]
pieza = (Pieza) tipoPieza.newInstance();
```

Durante la implementación surge el problema de que, en base a encapsulación e inversión de control, cometí el fallo de agregar los tipos de clase (el fallo sería el mismo poniendo el nombre de las clases en modo texto) dentro de la clase abstracta, lo que limita el uso de la clase abstracta a sus implementaciones conocidas y no pudiendo expandir este uso a más clases que se creen nuevas. En mejoras se comenta junto con la solución que podría haberse implementado, pero quedaba fuera de ámbito.

El Tablero de Juego

Una vez se ha diseñado el sistema de piezas, su generación y rotación, comenzamos a diseñar el tablero donde estas piezas van a cobrar vida. El tablero de juego se concibe para que sea una matriz de colores (de `java.awt.Color`) de modo que los colores usados sean:

Vacío	Recta	Cuadrado	Te	L-Izq	L-Der	Z-Izq	Z-Der
BLACK	CYAN	YELLOW	MAGENTA	ORANGE	BLUE	RED	GREEN

El tablero, por tanto, establece una matriz en la que almacena, en principio, solo colores negros, y en el momento que se fija una pieza al tablero (que no la pieza de en juego), estos bloques se fijan con su color al mismo.

La Pieza en Juego

Cuando se genera una pieza con la que jugar. Esta pieza se mantiene *volante* sobre el tablero, de modo que no forma parte, realmente de él. La pieza en juego, por tanto tiene la clase propia instanciada de forma genérica dentro del tablero (Pieza actual), y su posición relativa dentro del tablero (xp, yp).

Por tanto, el desplazamiento de la pieza afecta solo a estas coordenadas (xp, yp) y cuando se fija la pieza al tablero, se hace que la matriz de puntos se escriba (solo los 1) en el tablero, cambiándose por el color de la pieza activa o actual **`[R7]`**.

Un pequeño ejemplo de esto:

```
Tablero tablero = new Tablero(10, 20); // Tablero de 10x20
Pieza actual = new EleDerecha();

tablero.setPieza(actual);
tablero.desplazaPieza(0, 18); // Bajamos hasta el fondo
tablero.fijaPieza(); // se fija la pieza
```

En este ejemplo, el tablero se ha creado de 10x20 cuadros, se ha creado una pieza de tipo EleDerecha, se ha agregado al tablero como pieza en juego, se ha bajado 18 posiciones (hay de 0 a 19 y todas las piezas, nada mas iniciar, ocupan una altura de 2 posiciones, por lo que, se baja 18 posiciones y la 19 estará ocupada por la pieza, es decir, la base), y se ha fijado la pieza.

Con esto, la pieza ha sido fijada, pero no se ha quitado del juego, y podría modificarse su posición y volverse a fijar, por ejemplo, agregando el código:

```
tablero.desplaza(4, 0); // Mueve a izquierda 4 posiciones
tablero.fijaPieza();
```

Con este código, el tablero tendría dos piezas fijas ya en el tablero. En caso de querer girar la pieza, solo habría que llamar al método del tablero específico y se giraría en sentido de las agujas del reloj **[R8]**.

La lógica del Juego

Tenemos el tablero, tenemos las piezas... ahora nos falta la lógica del juego. Según los supuestos y requisitos, el juego debe de iniciarse con el tablero vacío, con una pieza en juego y esta debe de ir cayendo a una velocidad de 1 cuadro por segundo **[R5]**. Agregamos un reloj (un `javax.swing.Timer`), que llamará a una función cada segundo.

Así mismo necesitamos recibir las acciones del teclado para hacer que la pieza se desplace lateralmente y gire. Pero se nos plantea otro problema, y es que hay una limitación temporal para los giros **[R6]**. Ya tenemos un cronómetro, pero mejor, aislamos todo lo que tenga que ver con el teclado en una nueva clase **Teclado**, para simplificar. En esta clase Teclado, agregamos los eventos propios del teclado y creamos el reloj para el requisito planteado del giro limitado a 4 por segundo.

Esta clase, controlará que, cada vez que el Teclado le pida mover una pieza o girarla, no choque con los bordes, ni con otros bloques presentes en el Tablero.

Controlando las colisiones

El tablero, como ya se ha mencionado antes, es una matriz de elementos de Color, que según el color, puede indicar que la casilla está ocupado por un bloque de color, o esta vacía. Las colisiones que se han tenido en cuenta y se han desarrollado son las siguientes:

- **Abajo**: comprueba que la pieza, al avanzar a la siguiente posición, ni se salga del tablero por su parte inferior, ni ocupe un espacio ya ocupado por alguno de los bloques presente en la matriz propia de la pieza.

- **Derecha**: comprueba que la pieza, al moverse lateralmente hacia la derecha en la siguiente posición, ni se salga del tablero por su parte derecha, ni ocupe un espacio ya ocupado por alguno de los bloques presente en la matriz propia de la pieza.
- **Izquierda**: comprueba que la pieza, al moverse lateralmente hacia la izquierda en la siguiente posición, ni se salga del tablero por su parte izquierda, ni ocupe un espacio ya ocupado por alguno de los bloques presente en la matriz propia de la pieza.
- **Giro**: comprueba que, al realizar el siguiente giro, la pieza no se salga del tablero ni por su parte derecha, ni por su parte izquierda, y que la nueva posición de la pieza, no ocupe bloques que ya estén ocupados por otros bloques ya presentes en el tablero.

Con esto, nos aseguramos, para todos los movimientos, que sean realmente posibles, según la lógica del juego **`[R16]`** y **`[R17]`**.

Eliminando líneas completas

Una vez que una pieza en juego ha tocado fondo, según la función de colisión abajo, esta pieza se fija **`[R4]`** y se llama a la función de limpieza de líneas, que se encarga de recorrer el tablero, de abajo hacia arriba para, en caso de que haya alguna línea completa, es decir, que todas sus posiciones horizontales estén ocupadas por un bloque, del color que sea, esta línea es eliminada copiando todas las que están por encima y desplazando por tanto una posición hacia abajo todo el tablero, al mismo tiempo que se configura la primera posición como vacía **`[R14]`**.

Presentación del Juego

Una vez tenemos ya la correspondencia del teclado con la lógica del juego, y todas las reglas, e incluso los relojes para el juego, solo nos queda representar el tablero de forma gráfica. Para esto, teniendo en cuenta de que Java es un elemento que es soportado en muchas plataformas de diversa índole, se quiso mostrar el juego, no solo como aplicación de escritorio, sino también como applet en la web. Por ello, se generó la **ZonaJuego**, donde se abstraen los elementos básicos que requiere la lógica de **Juego** para desarrollar el juego en sí.

Como se ha dicho anteriormente, en varias ocasiones, el juego ha sido desarrollado como aplicación de escritorio a través de JFrame, y como applet para la web a través de JApplet, por lo que, a través de ZonaJuego, se permite la interfaz entre uno y otro.

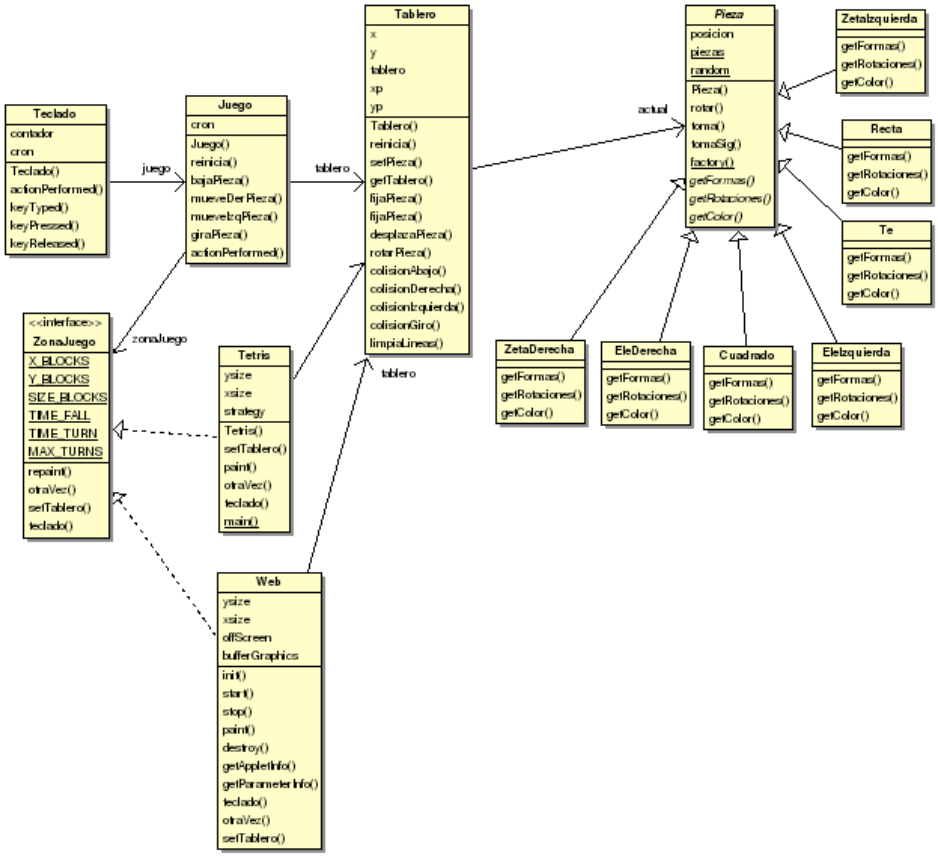
Desde la Web

Para que el sistema funcione desde la web, además de generar el fichero *jar*, se necesita de un fichero HTML con el siguiente código dentro:

```
<applet
            code="net.bosqueviejo.uned.lsi.poo.Web.class"
            width="240px"
            height="460px"
            archive="tetris.jar"
></applet>
```

Diagrama de Clases

Como se comentaba antes, se han creado cuatro zonas centrales en la que se aglomera el código principal, estas son:



- **Pieza**: es la parte del código que se encarga de las acciones base de las piezas, su rotación, la forma de cada una de sus posiciones a los 0º, 90º, 180º y 270º, así como su color. La clase, a través de un patrón de diseño *factory*, se encarga, también de la generación de las piezas de forma aleatoria.
- **Tablero**: se encarga de mantener la forma propia del tablero, los bloques que contiene en cada casilla, la pieza actual que hay en juego y su posición relativa al tablero, así como el movimiento de la pieza. Tiene también las funciones para comprobar si el movimiento es posible o no, pero sin estar enlazadas con las opciones propias de movimiento.
- **ZonaJuego**: es una interfaz que es como una plantilla de cómo debe de ser la clase que se encargue de realizar la visualización del juego. A modo de ejemplo, se han realizado dos implementaciones:
  - **Tetris**: implementación con JFrame, como aplicación de escritorio. Tiene un método *main*, por lo que se puede lanzar directamente desde consola.
  - **Web**: implementación con JApplet, como aplicación incrustada en una página web. En la página web se debe de usar una etiqueta especial, que haga referencia a la clase dentro del paquete (fichero *jar*).

- **Juego**: es la clase que tiene la lógica del juego. Se encarga de iniciar el juego, crear el tablero, notificar al tablero de que debe de bajar la pieza cada segundo, y tiene los métodos necesarios para mover y girar la pieza en juego (llamando a la comprobación del tablero, antes de invocar la acción en sí).

Análisis de Clases

Se ha visto a nivel general el esquema de clases y se han detectado los principales núcleos del programa, ahora vamos a ver en detalle cada una de las clases, su descripción y papel dentro del software.

Pieza

Como se había dicho antes, la clase pieza se encarga de la forma de cada pieza, las rotaciones y la creación aleatoria de las mismas. Para ello dispone de los métodos:

- **rotar**: realiza una rotación sobre la pieza.
- **toma**: da una matriz de 4x4 en la que se indica con 0 y 1, si hay bloque que dibujar en esa posición o no.
- **tomaSig**: retorna la pieza pero como si se hubiese realizado la siguiente rotación. Es útil para clases como *Tablero*, que le ayuda a comprobar si una pieza colisiona al ser girada o no.
- **factory**: es un método de clase, que se encarga de generar una instancia hija de las piezas: Recta, Cuadrado, Te, EleDerecha, ElelzquierdaZetaDerecha y ZetaIzquierda; lo hace de forma aleatoria.

Piezas

Recta, Cuadrado, Te, EleDerecha, ElelzquierdaZetaDerecha, ZetaIzquierda: las *hijas* de la clase Pieza, implementan los métodos abstractos que completan el código necesario para la pieza:

- **getFormas**: da las matrices de las formas de la pieza.
- **getRotaciones**: da el número de giros posibles que tiene cada pieza. Esto es porque piezas como Cuadrado, por ejemplo, solo tienen una posición, y por tanto no tienen posibilidad de girar, mientras que otras como Zeta y Recta tiene dos posiciones y el resto tienen las cuatro.
- **getColor**: da el color de la pieza. Cada pieza tiene su propio color, esto se comentó más arriba.

Tablero

Esa clase ya se ha comentado con anterioridad, sus métodos son los siguientes:

- **reinicia**: se encarga de iniciar el tablero con los valores por defecto de `BLACK` y crear la pieza en juego o actual.
- **setPieza**: se le pasa una pieza como parámetro y la configura como pieza en juego o actual, colocándola en la posición de salida, que es la primera fila, en la columna central `[R12]`.
- **fijaPieza**: toma la información de la pieza actual y la fija en la posición relativa al tablero en la que se encuentre.
- **desplazaPieza**: desplaza la pieza en juego de forma relativa, según los valores que se pasen como parámetros. Si como parámetro X se pasa 0, la pieza no se moverá lateralmente, si se pasa un número negativo, se moverá a izquierda y si el número es positivo, lo hará hacia la derecha. Igualmente se trata para el parámetro Y.

- **rotarPieza**: indica a la pieza actual que gire, en sentido de las agujas del reloj, para que el siguiente dibujado del tablero aparezca girada **[R8]**.
- **colisionAbajo**: retorna verdadero si hay colisión en caso de mover la pieza hacia abajo en el siguiente movimiento y falso en caso contrario.
- **colisionDerecha**: retorna verdadero si hay colisión en caso de mover la pieza hacia la derecha en el siguiente movimiento y falso en caso contrario.
- **colisionIzquierda**: retorna verdadero si hay colisión en caso de mover la pieza hacia la izquierda en el siguiente movimiento y falso en caso contrario.
- **colisionGiro**: retorna verdadero si hay colisión en caso de realizar el siguiente giro de la pieza como siguiente movimiento y falso en caso contrario.
- **limpiaLineas**: realiza el barrido del tablero de abajo hacia arriba para eliminar todas las líneas que estén completamente llenas por bloques. Esto se comentó anteriormente **[R14]**.

Juego

Esta clase se encarga de la lógica del juego, sus métodos son:

- **reinicia**: se encarga de reiniciar el tablero y activar los relojes.
- **bajaPieza**: comprueba por la colisión correspondiente que el movimiento es posible, y en caso de que lo sea, desplaza la pieza. Si el movimiento no fuese posible, fija la pieza, y genera una nueva. En caso de que la nueva colisione nada más comenzar, da el juego por acabado, preguntando si desea que se vuelva a comenzar.
- **setPieza**: configura una pieza como la pieza en juego.
- **mueveDerPieza, mueveIzqPieza**: se asegura por las funciones de colisión correspondientes de que la pieza puede ser desplazada lateralmente y, en caso de que se pueda, la desplaza **[R7]**.
- **giraPieza**: se asegura por la función de colisión de giro de que la pieza puede ser girada y, en caso de que se pueda, la gira **[R8]**.
- **actionPerformed**: el método requerido por el reloj, para ser llamado cada segundo. Este método llama internamente a *bajaPieza*, ya que la lógica está ahí **[R5]**.

Teclado

Como se dijo anteriormente, la dinámica del teclado ha sido desplazada a esta clase para simplificar su desarrollo, ya que se ha implementado un requisito que implica una acción de escucha de un reloj sobre lo que puede hacer el teclado. Los métodos de esta clase son (solo se comentan los que se usan):

- **actionPerformed**: inicializa el contador a cero, para que se puedan hacer otros 4 giros en este segundo nuevo **[R6]**.
- **keyPressed**: la tecla pulsada. Este evento es el que recoge la pulsación de las teclas del cursor y, según lo que se presione (el cursor hacia arriba, izquierda o derecha) se envía la petición de hacer un desplazamiento horizontal o un giro sobre la pieza **[R7]** y **[R8]**.

ZonaJuego

La zona de juego define los valores base del juego que se usan como constantes en todas las clases, valores como:

- **X_BLOCKS**: tamaño horizontal del tablero en bloques. Fijado en 12 **`[R3]`**.
- **Y_BLOCKS**: tamaño vertical del tablero en bloques. Fijado en 25 **`[R1]`**.
- **SIZE_BLOCKS**: tamaño de cada bloque en píxeles. Fijado en 20.
- **TIME_FALL**: tiempo que tarda en caer un bloque una pieza, en milisegundos. Fijado en 1000 **`[R5]`**.
- **TIME_TURN**: tiempo cada cuanto se reinicia el contador de giros, en milisegundos. Fijado en 1000.
- **MAX_TURNS**: número de giros que se permiten por el tiempo configurado. Fijado en 4 **`[R6]`**.

Los métodos para la parte gráfica del juego que se deben de desarrollar son los siguientes:

- **repaint**: es un método que está presente en JFrame y en JApplet, y se encarga de forma la llamada a las funciones de redibujado del elemento en sí. Se emplea cada vez que se realiza un movimiento lateral, de giro o cuando la pieza cae una posición.
- **otraVez**: se encarga de realizar la pregunta al usuario de si quiere volver a jugar o no. En caso de entornos web, esta pregunta no se realiza, sino que se reinicia el juego de nuevo.
- **setTablero**: se configura el tablero sobre el que se jugará. A este tablero se le pedirá información para su dibujado en la pantalla.
- **teclado**: agrega la clase que contiene los métodos del teclado para que gestione el teclado durante el juego.

Tetris

Es la clase principal del juego, desde el punto de vista de aplicación de escritorio, ya que es la clase que contiene el lanzador inicial del programa, el método de clase: *main*. Esta clase implementa los métodos de ZonaJuego, por lo que no los comentaré de nuevo, el resto de métodos:

- **main**: un método estático de clase que se encarga de inicializar el programa y lanzarlo.
- **Tetris**: el constructor, que se encarga de configurar toda la ventana inicial.
- **paint**: método llamado cada vez que se debe de redibujar la ventana (o se llama a *repaint*).

Web

Para la publicación a través de web, en formato applet, esta clase requiere que se implementen todos los métodos de JApplet, por lo que, en base, esos son los métodos de que dispone, a demás de los que requiere para poder se implementación de ZonaJuego.

Esta clase contiene en *start* todo lo que Tetris contiene en *main*, básicamente, y el contenido en *paint* es muy similar, por no decir igual. Tiene algunas optimizaciones que varían de uno a otro, pero el resto es idéntico.

Universidad Nacional de
Educación a Distancia

Conclusiones y Comentarios

En esta sección, como cajón de sastre, se introducen algunos elementos que se han solicitado deben de aparecer dentro del documento, como son comentarios sobre la práctica, mejoras que se podrían haber realizado y problemática encontrada en el desarrollo propio de la práctica, por tanto, damos pie a comentar todos estos temas en esa sección y ampliar con otros comentarios.

Mejoras

Se enumerarán las mejoras que se han contemplado posibles, pero fuera de alcance por tiempo:

- En el diseño propio de las clases, el uso del patrón factory dentro de Pieza, ha hecho que la encapsulación no sea tal y como debería, según los modelos de encapsulación de objetos en los que se dice que una clase no debe ser manipulada cada vez que se requiera o necesite ampliar su funcionalidad a través de la herencia. Esto podría haberse solucionado creando otro paquete hijo llamado *piezas*, donde se incluyesen las clases que se deben de incluir dentro de Pieza para ser llamadas cuando se requiera, así como agregar una variable de entorno o un fichero *property* para permitir la configuración de más rutas donde buscar más piezas, en caso de que se desease desarrollar de forma paralela, sin modificar el código original, un paquete agregado de piezas.

- El tablero configura colores para ser representados, tomando como base que si el color es negro (`BLACK`), ese cuadro no se dibuje. Esto podría haberse realizado igualmente con *null*, es decir, si el espacio en la matriz no es un objeto, sino que es *null*, entonces no se dibuja.

- El paso de mensajes en algunas partes en la que se ha realizado mediante una configuración (*setTablero*, por ejemplo) podría haberse realizado mediante una obtención de información en sentido contrario (*getTablero*) de modo que se habrían empleado menos variables de objeto.

Problemática encontrada

En principio, la realización del tablero a nivel básico, consta del uso de Java a un nivel básico, es decir, no se sale de emplear funciones base que modifican tablas (o matrices) y cambian datos de una forma u otra. Quizás el factor más problemático, por decirlo de alguna forma, sea el uso en sí de la interfaz de usuario gráfica (GUI en inglés), ya que plantea el problema de que hay que adentrarse, conocer y manejar, una jerarquía de clases y una ideología planteada por Sun Microsystems de uso de unas gráficas, así como del teclado y los eventos de tiempo.

Personalmente, no he visto gran problema en la realización de la práctica, por mi trayectoria y mi bagaje con Java, pero sí entiendo que para una asignatura de primero de carrera, en la que haya gente que la programación sea algo que no haya visto con más tiempo de 4 meses, puede ser algo bastante complejo, y sobretodo teniendo en cuenta que hay que emplear Orientación a Objetos, y Orientación a Eventos, que son dos *doctrinas* que chocan en concepción con la programación estructurada.

Comentarios o Crítica sobre la Práctica en sí

La práctica me ha resultado curiosa, realmente, porque permite realizar un programa que de seguro, muchos, van a probar hasta asegurarse de no tener fallos y durante horas. No obstante, plantea ciertas dudas sobre la asignatura en sí, ya que, disponiendo de tan solo 4 meses de asignatura, para alguien con poco conocimiento de orientación a objetos supone:

- Aprender un lenguaje Orientado a Objetos como Java. Que no supone un gran problema, pero si un gran cambio cuando se viene de lenguajes como C o C++, donde el tipo de los datos se debe de *mimar*, para evitar violaciones de segmento, por ejemplo.
- Aprender bien Java. Lo cual veo que no se ha hecho mucho énfasis, en sentido de que, por ejemplo, la carencia de la etiqueta *package* en el programa, ocasiona un aviso y es considerado una mala práctica, al igual que el uso de asteriscos en los *import*, y otras cosas más.
- Patrones y anti-patrones. Es importante, muy importante, hacer énfasis en programar con patrones de diseño para facilitar el código, hacerlo más robusto, comprensible y elegante. Java, de por sí, tiene muchos patrones que implementa de forma nativa. Así mismo, es importante reseñar las cosas que NO se deben de usar nunca.
- Interfaz Gráfica de Usuario. Habría que comenzar con la ideología, ¿por qué está organizada así?, ¿por qué hay awt y swing?, para pasar a ver las jerarquías de clases, los eventos típicos y los objetos base.

Solo con eso, ya habría para más de 4 meses... y nos hemos dejado fuera lo más importante, que sería aprender la Orientación a Objetos teórica, a través de UML, por ejemplo.

Considerando la práctica, con toda la materia que está detrás, considero que, si bien, quienes lo conseguimos ya sabiendo Java no nos aporta gran cosa, los que no saben y llegan, puede que no hayan conseguido tampoco gran aporte, por ser tanto lo visto en tan poco tiempo... pero es solo mi opinión.