- Práctica de Programación Orientada a Objetos - Junio 2011 -

May 18, 2011

Alumno: Octavio Martínez García
NIF: 71280002-N
e-mail: octavio_ mgs@yahoo.es
Teléfono: 690 36 11 55

Análisis de la Aplicación:

El funcionamiento general de la aplicación se basa en el uso de una matriz de dos dimensiones, de 25 filas por 12 columnas, la cual almacena en cada una de sus celdas los bloques que forman cada una de las posibles piezas del juego o tetriminos.

Desde esta perspectiva, la unidad básica del juego son los bloques, representados por la clase **Block**. Cada uno de estos bloques almacena internamente en sus atributos o campos, información propia sobre su color, posición en el la matriz del juego (definida esta posición por la tupla (fila,columna)), así como una bandera que determina si el bloque ha sido "apilado" al fondo de la matriz y es por tanto inmóvil.

Contiene casi exclusivamente métodos de acceso y modificación de sus campos, para acceder o modificar la posición del bloque, el color, la bandera que indica si está apilado, etc.

La existencia de esta clase se justifica por la identificación del bloque como unidad mínima del sistema de juego, la deberá ser almacenada en las celdas de la matriz y utilizada por esta última clase para representar los movimientos del juego. Esta clase será utilizada de forma directa por las clases Piece, la cual contiene la información para la inicialización de cada uno de los 7 tetriminos diferentes, y por la clase Grid para el posicionamiento y movimiento de los bloques en la matriz.

Una unidad superior dentro del diagrama de clases es la clase **Piece**, la cual representa una agregación de 4 bloques en diferentes posiciones relativas, que conforman las 7 diferentes piezas del juego o tetriminos. Realmente esta clase almacena internamente una matriz de 3 filas por 3 columnas (o 4x4 en el caso de que queramos representar la pieza "stick" o "palo"), la cual puede contener bloques al igual que la matriz principal de juego, y en la cual se pueden hacer 7 inicializaciones diferentes según el parámetro entero del constructor. La funcionalidad básica de esta clase es precisamente esa, encapsular la forma de inicializar los 7 diferentes tetriminos, para ser usada por la matriz principal. Además esta clase provee de funcionalidad a la matriz principal, para la forma de realizar los giros de las piezas, ya que mantiene información interna del número de posiciones que puede tener una pieza en un giro completo, información representada en el campo *pieceP ositions*.

Los métodos que contiene van destinados en su mayoría a la creación de cada una de las 7 piezas de juego, así como algún método adicional para el acceso o modificación de sus campos.

El uso de esta clase viene determinado por razones de abstracción y modularidad, ya que conceptualmente es más lógico y dota de mayor claridad a la estructura de clases, si la construcción inicial de las piezas se hace en una clase aparte y es utilizada posteriormente por la clase Grid que representa la matriz de juego principal. Además con esto dotamos a la aplicación de una mayor

cohesión, ya que esta tarea específica es realizada aparte por una clase dedicada a ello.

La clase que se encarga de toda la mecánica interna de los movimientos y posicionamientos de las piezas de juego está representada por la clase **Grid**, la cual está formada por la anteriormente citada matriz interna de 25 filas por 12 columnas, así como por otros campos que mantienen información sobre el número de giros hechos, la pieza actual sobre la que se trabaja, el bloque central de dicha pieza, y un objeto Random usado para la creación aleatoria de las piezas.

Esta clase hace uso directo de la clase Piece, para tener acceso a la creación aleatoria de los 7 tetriminos al inicio del juego y cuando una pieza se apile al fondo de la matriz, así como de la clase Block. Esta decisión viene dada debido a que varios de los métodos de Grid utilizan los bloques almacenados en sus celdas para realizar los diferentes movimientos y posicionamientos de dichos bloques, y si bien Grid tiene un método que devuelve un objeto almacenado en una determinada celda (*Block getObjectAt(row, col)*) que podría ser usado directamente para hacer referencia a un determinado objeto bloque (eliminando ese uso directo de Block y reduciendo por tanto el acoplamiento entre clases) en estos métodos de Grid, optamos por usar una variable local de tipo Block a la que asignamos el valor del método comentado, lo cual contribuye a una mayor claridad y legibilidad del código.

La mayoría de los métodos de Grid, están orientados a realizar los diferentes cálculos necesarios para logar el movimiento y rotación correctos de las piezas dentro de la matriz de juego, así como métodos auxiliares para el posicionamiento de bloques o la eliminación de estos.

La existencia de esta clase se debe a la necesidad de usar una clase que se ocupe de la mecánica interna del juego en lo referente a movimientos y colocación de las piezas en el área de juego, la cual por motivos de cohesión es preferible que sea independiente de la IGU del juego, facilitando esto a su vez la extensión o modificación del juego, en el caso, por ejemplo, de que quisiéramos que la interfaz de usuario fuera de otro tipo, (terminal de texto, etc.)

Para crear la interfaz gráfica de la pantalla de juego, existe la clase **GameView** que se encarga de mostrar en pantalla la matriz de juego representada por rectángulos de diferentes colores en función de si las posiciones de la matriz están ocupadas por bloques o si están vacías. Cada vez que cambia el estado de la matriz a causa de los diferentes movimientos del juego, se llama a uno de los métodos de GameView que se encarga de mostrar el estado actualizado de la matriz y del juego por extensión.

Esta clase se encarga de la creación de la ventana de juego mediante un JFrame extendido, y contiene como atributo una clase interna extendida a su vez de JPanel, la cual contiene toda la lógica necesaria para preparar la imagen a mostrar cada vez que sea necesario, la cual consiste en una rejilla formada por rectángulos de diferentes colores que indican si una posición de la matriz del

juego está vacía (color negro) o bien ocupada por una bloque de una determinada pieza (color del bloque de una pieza especifica).

Adicionalmente la ventana de juego contiene una barra de menú, que facilita ciertas funcionalidades para controlar el juego, como pausar, resetear, o salir del juego. Los elementos del menú de juego ubicados en esa barra, son variables de instancia que pueden ser accedidas por otras clases mediante métodos de acceso, lo cual es útil para que la clase principal del juego, Tetris, acceda a estos elementos para establecerlos como oyentes de eventos y poder así realizar las acciones definidas internamente. Establecemos los oyentes de eventos en la clase Tetris, y no en GameView, ya que las acciones que realizan al producirse los eventos (click en el elemento de menú) están definidas en la primera clase, de modo que si les definiéramos en GridView, ésta clase debería hacer uso de Tetris, induciendo una relación de uso que no existe del modo que lo hemos definido, y aumentando, por consiguiente, el acoplamiento entre clases.

La existencia de esta clase viene determinada por la necesidad de obtención de un mayor grado de cohesión, creando una clase específica para toda la creación y gestión de la IGU, orientando de este modo el diseño según las responsabilidades de cada clase.

Finalmente tenemos la clase principal del juego, **Tetris**, que representa la clase cliente, que utiliza todas las demás clases, bien directa o indirectamente, para crear la simulación del Tetris tal y como se especifica en el enunciado de la práctica. Básicamente mantiene en sus campos una matriz de juego, una IGU de la matriz, y un objeto Timer, utilizado principalmente para controlar la caída por segundo de las piezas. Tiene asimismo establecidos diferentes oyentes de eventos que se ocupan de que al realizar diferentes acciones por el usuario del juego (básicamente presionar las teclas de dirección) , el juego responda según la mecánica preestablecida.

La mayoría de los métodos relevantes de esta clase usan métodos de movimientos de piezas de Grid, seguidos del método *showStatus(grid)* de GameView para visualizar ese cambio de estado de la matriz del juego ocurrido después del movimiento de la pieza. Asimismo se usan métodos de inicialización del Timer para dar inicio al juego, durante el desarrollo del cual se realizan los pasos de bajar una pieza, buscar "lineas" (filas completas por bloques para eliminarlas), leer las entradas de las flechas de dirección para realziar los movimientos adecuados de las piezas, y verificar que no se haya llegado al tope superior de la matriz, o finalizar el juego en caso afirmativo.

Esta clase existe porque es necesaria para combinar todos los demás elementos de la mecánica del juego, definidos por el resto de clases del diagrama de clases, para ejecutar la simulación interactiva del juego propiamente dicha.

El funcionamiento básico de juego es el siguiente:

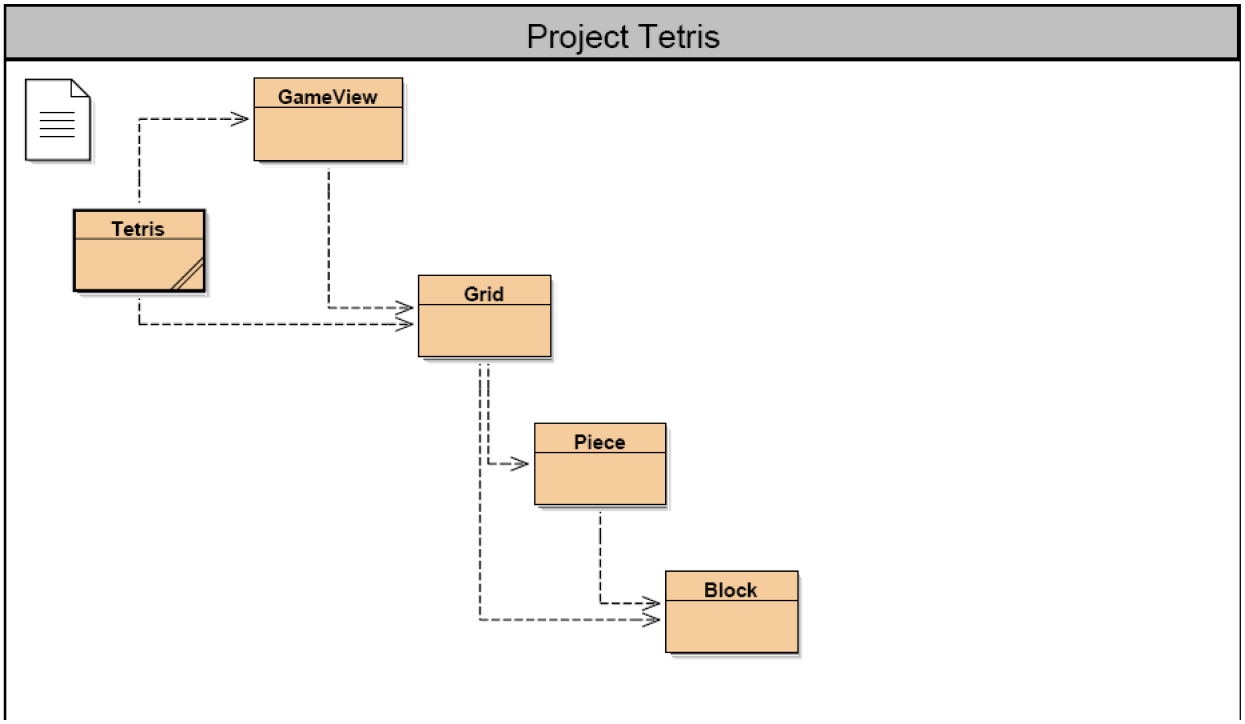
- Inicialmente se crea un nuevo objeto de la clase Tetris, la cual cuando se llama a su constructor, crea dos nuevos objetos de tipo Grid y GameView, así como los oyentes de eventos necesarios tanto para el teclado como para los menús de juego. Seguidamente se pinta en pantalla la matriz de juego, se crea un nuevo tetrimino aleatorio en la posición inicial y se da comienzo al juego por medio del temporizador.

- Una vez iniciado el juego, para realizar cualquier movimiento de las piezas, ya sea por creación de una nueva pieza, desplazamiento descendente (natural o inducido), desplazamientos laterales o giros, se invoca el método correspondiente de Grid que realiza el movimiento adecuado pintando nuevos bloques en las posiciones correspondientes y borrando los antiguos. Una vez que se realiza el movimiento mediante la llamada a los métodos de Grid, el método invocador de Tetris llama a continuación al método principal de GameView (*showStatus(grid)*) para pintar el estado actualizado de la matriz después del movimiento de las piezas.

- Durante el "ciclo" del Timer se realiza a cada paso (cada 1000 ms por defecto), aparte del movimiento descendente, una comprobación de la existencia de filas completas (llenas de bloques) para hacer lo que se conoce como "linea", que se trata de eliminar la fila completa (poniendo todas sus celdas a null) y bajar una fila el resto de bloques apilados que existan en la matriz en ese momento.

- Aparte de lo anterior, se realiza a cada paso del juego (en cada acción del Timer) una comprobación de que no hayamos alcanzado el tope superior de la matriz, para que el juego pueda continuar. En caso contrario se detiene el Timer y se muestra un mensaje de "Game Over", dándose por finalizado el juego.

Diagrama de Clases:



Como se puede observar en el diagrama de clases, la aplicación consta de 5 clases, entre las cuales se establecen las relaciones de uso mostradas en la figura.

Descripción de Clases:

Block:

Esta clase sirve de representación de la unidad mínima de juego, que son los bloques que forman las piezas de juego, proveyendo al resto de clases de un objeto que pueden usar para representar dichos bloques de juego, el cual contiene información interna sobre su color, posición y si está o no apilado en la matriz de juego.

Métodos públicos:

Color getBlockColor();
Da acceso al campo color del bloque

int getCol();
Da acceso a la coordenada columna de la tupla (fila, columna) que determina la posición del bloque en la matriz.

int getCol();
Da acceso a la coordenada fila de la tupla (fila, columna) que determina la posición del bloque en la matriz.

boolean isStacked();
Determina si el bloque está o no apilado en la matriz de juego

void setStacked();
Establece un bloque como apilado en la matriz de juego.

void setLocation(int row, int col);
Establece la posición del bloque en forma de coordenada 2D (fila, columna)

void setBlockColor(Color color);
Establece un color para este bloque

Piece:

Esta clase representa una de las posibles 7 piezas diferentes del juego o tetriminos, las cuales están compuestas por 4 bloques organizados en diferentes posiciones relativas, que se almacenan internamente en una submatriz de 3 filas por 3 columnas (o 4 x 4 en caso de la pieza "palo").

La funcionalidad básica de la clase es servir a la clase Grid de una forma sencilla de realizar la inicialización aleatoria de las piezas cuando ésta las necesite crear en los diferentes momentos del desarrollo del juego, además de proporcionar facilidades a la hora de realizar los giros en la matriz principal al mantener información sobre las posiciones diferentes que puede tener una pieza en los giros.

Su existencia se justifica desde el punto de vista de la cohesión, por la utilidad de encapsular en una clase aparte la forma de inicializar las 7 piezas del juego en vez de hacerlo directamente en la matriz del juego.

Métodos públicos:

Block getBlock(int row, int col);
Devuelve el objeto bloque almacenado en una de las celdas de la submatriz 3x3 o 4x4

int getPieceCols();
Acceso a la anchura(columnas) de la submatriz que contiene la pieza

int getPieceRows();
Acceso a la altura(filas) de la submatriz que contiene la pieza

int getPositions();
Acceso al número de posiciones diferentes que tiene la pieza en los giros

Grid:

Esta clase representa la matriz principal del juego de 25 filas por 12 columas, y se encarga de toda la lógica interna necesaria para implementar los movimientos de las piezas en el área de juego, la creación de piezas nuevas, y la búsqueda de "lineas" y la eliminación de ls mismas cuando ocurran.

Cada vez que necesita posicionar un bloque en la matriz principal, añade a la celda correspondiente una referencia a un nuevo (o existente) bloque, y establece la información interna del bloque sobre su posición para que coincida con la posición que ocupa actualmente en el Grid.

Mantiene referencias al bloque "central" de la pieza actual (la que podemos mover) para guiarse a la hora de realizar los giros.

Cada vez que se le ordena (desde Tetris) realizar un movimiento de las piezas, comprueba que existe espacio para realizar dicho movimiento, y en caso afirmativo coloca nuevos bloques en las posiciones pertinentes y borra los anteriores. (es responsabilidad de Tetris el llamar a métodos de GameView después para "pintar" el nuevo estado de la matriz de juego).

Mantiene un contador interno de los giros hechos (*int turns*) utilizado para la implementación de la limitación de 4 giros por segundo.

La existencia de esta clase se justifica por la necesidad de unificar toda la lógica del movimiento y colocación de las piezas en la matriz en una sola clase que se ocupe de este trabajo.

Métodos públicos:

boolean allStacked();

Comprueba si todos los bloques de las piezas en el grid están marcados como apilados

void clear();

Vacía el grid poniendo a null todas sus posiciones

void createTetrimino();

Crea un nuevo tetrimino en la posición inicial de la matriz de juego, es decir, en la zona superior central, mediante la creación aleatoria de un nuevo objeto Piece y la copia de sus bloques en las posiciones adecuadas de la matriz de juego.

Color getCellColor(int row, int col);

Devuelve el color de una determinada celda de la matriz

int getHeight();

Devuelve la altura (filas) de la matriz

int getWidth();

Devuelve el ancho (cols) de la matriz

Block getObjectAt(int row, int col);

Devuelve el objeto en la celda especificada (fila-columna), o null si no existe ningúno en esa ubicación

void moveDown();
Comprueba que es posible el movimiento descendente de toda la pieza y si es así la baja una fila.

void moveLeft();
Comprueba que es posible el movimiento a la izquierda de toda la pieza y si es así la mueve en ese sentido.

void moveRight();
Comprueba que es posible el movimiento a la derecha de toda la pieza y si es así la mueve en ese sentido.

void turnPiece();
Intenta girar la pieza 90^{o} si hay espacio para tal movimiento en función del número de posiciones que tenga definidas la pieza concreta para un giro.

void searchLines();
Recorre la matriz de abajo a arriba buscando líneas completas y eliminándolas, moviendo hacia abajo los bloques superiores

void setTurns(int value);
Pone el contador de giros a un valor determinado

GameView:

Esta clase encapsula la IGU del juego, que se representa como una matriz de rectángulos de dimensiones 25 filas por 12 columnas, en la cual los rectángulos que representan cada celda de la matriz están definidos por un color u otro en caso de que dicha posición en la matriz del juego definida por la clase Grid, esté vacía (null, color negro) u ocupada por un bloque (instancia de Block, color definido para ese bloque).

Utiliza una clase interna que deriva de JPanel para proporcionar una forma de pintar en el componente de la ventana una matriz formada por rectángulos que sirva de representación del área de juego.

Mantiene como atributos los elementos de menú (JMenuItem) que necesitará utilizar, mediante métodos de acceso, la clase Tetris para establecer estos elementos como oyentes de eventos que realizarán las acciones definidas en esta última clase.

Métodos públicos:

JMenuItem getQuit();
Da acceso al elemento de menú Quit

JMenuItem getReset();

Da acceso al elemento de menú Reset

JMenuItem getResume();
Da acceso al elemento de menú Resume

JMenuItem getStart();
Da acceso al elemento de menú Start

JMenuItem getStop();
Da acceso al elemento de menú Stop

void showStatus(Grid grid);
Muestra en pantalla el estado actual de la matriz que representa la pantalla de juego diferenciando por colores entre celdas vacías y ocupadas por bloques

Tetris:

Esta es la clase principal de la aplicación, que se encargará de ejecutar la simulación del juego, mediante el uso del resto de clases.

Al crear un objeto Tetris mediante su constructor se crean instancias de Grid, de GameView así como unos oyentes de eventos. Estos oyentes de eventos son bien las teclas de dirección del teclado, que al ser presionadas llaman a métodos privados de Tetris que mueven la pieza en la dirección deseada y llaman a *showStatus(grid)* de GameView para mostrar en pantalla el nuevo estado del juego después del movimiento, o bien los elementos del menú Game, que cuando son clickeados realizan las acciones pertinentes de control del juego, siendo estas parar el juego, continuarlo, reiniciarlo o salir del juego.

Métodos públicos:

static void main(String[] args);
Este es el único método público que contiene la clase, que se encarga de crear un nuevo objeto Tetris, y es a partir de la inicialización del objeto mediante el constructor de Tetris, donde se crean los objetos necesarios que van a intervenir en la simulación, es decir, un objeto Grid, un objeto GameView, y unos oyentes de eventos, así como los bloques contenidos en la matriz principal, creados mediante la construcción de nuevos objetos Piece por parte de Grid, desplazados por los movimientos que inserte el usuario por el teclado, y apilados al fondo de la matriz una vez que toquen fondo u otros bloques apilados. Asímismo con la creación de la instancia de Tetris se da comienzo a la ejecución del juego ya que la útlima sentencia del constructor invoca al método privado *startGame()*, que crea un nuevo Timer y da comienzo a la repetición de acciones que realiza dicho temporizador, consistentes en bajar la pieza, comprobar si hay que crear otra pieza, buscar lineas y comprobar si se alcanza la parte superior de la matriz, para finalizar el juego.

Clase Block

```
import java.awt.Color;

/**
 * Representa cada uno de los 4 bloques individuales que conforman cada
 * pieza del juego o tetrimino
 *
 * @author Octavio Martínez
 * @version 16.05.2011
 */
public class Block
{
   // cada bloque almacena su posición en fila y columna
   private int row, col;
   // Cada bloque individual debe contener información sobre su color
   private Color blockColor;
   // Determina si el bloque ha tocado el fondo de la matriz de juego y no se debe mover
   private boolean stacked;


   /**
    * Constructor de bloques por defecto con color predefinido
    */
   public Block()
   {
      stacked = false;
      blockColor = Color.gray;
   }

   /**
    * Constructor de los bloques que formas las piezas, que establece
    * un color definido por parámetro
    * @param color El color que establecemos para este bloque
    */
   public Block(Color color)
   {
      stacked = false;
      blockColor = color;
   }

   /**
    * Construye un bloque con un color y una posición determinadas
    * @param color El color que asignamos a este bloque
    * @param row La fila que asignamos a este bloque
    * @param col La columna que asignamos a este bloque
    */
   public Block(Color color, int row, int col)
   {
      stacked = false;
      blockColor = color;
      this.row = row;
```

```
    this.col = col;
}

/**
 * Establece una posición para este bloque en forma de coordenada
 * fila-columna
 * @param row La fila que queremos establecer
 * @param col La columna que queremos establecer
 */
public void setLocation(int row, int col)
{
    this.row = row;
    this.col = col;
}

/**
 * Acceso a la posición fila del bloque
 * @return La fila que almacena internamente el bloque
 */
public int getRow()
{
    return row;
}

/**
 * Acceso a la posición columna del bloque
 * @return La columna que almacena internamente el bloque
 */
public int getCol()
{
    return col;
}


/**
 * Establece el color de los bloques
 * @param color El color que queremos para este bloque particular
 */
public void setBlockColor(Color color)
{
    blockColor = color;
}

/**
 * Acceso al color del bloque
 * @return El color del bloque
 */
public Color getBlockColor()
{
    return blockColor;
}
```

```
    /**
     * Indica que el bloque ha sido apilado en el fondo del grid
     * y no se puede mover (excepto en caso de hacer línea)
     */
    public void setStacked()
    {
        stacked = true;
    }

    /**
     * Verifica si el bloque está o no apilado al fondo
     * @return True si está apilado al fondo, false si no lo está.
     */
    public boolean isStacked()
    {
        return stacked;
    }

}
```

Clase Piece

```
import java.awt.Color;

/**
 * Una pieza que representa una de las posibles piezas del
 * Tetris o tetriminos, formada por objetos Block los cuales se almacenan
 * en una matriz de dos dimensiones de tamaño suficiente (3x3 o 4x4) para contener
 * cada una de las 7 posibilidades de piezas existentes en el Tetris.
 *
 * @author Octavio Martínez
 * @version 16.05.2011
 */
public class Piece
{
    // El tamaño de la matriz que contiene a los tetriminos
    private int piece_rows, piece_cols;
    // La matriz que contiene todas las formas posibles de los tetriminos
    private Block[][] piece;
    // Las posiciones de la pieza en los giros según tipos de tetriminos
    private int piecePositions;

    /**
     * Constructor con 7 opciones que se invocarán mediante un random 0-6
     * para construir 7 posibles configuraciones de pieza que representan
     * cada uno de los 7 posibles tetriminos
     * @param selector El número que especifica que inicialización se hará
     */
    public Piece(int selector)
    {
        // establece el tamaño por defecto de la matriz para el caso general(3x3)
        piece_rows = 3; piece_cols = 3;
```

```
        // realiza una de 7 configuraciones de bloques en la matriz
        // que representan uno de los 7 posibles tetriminos
        switch(selector) {
            // construimos el palo. Establecemos caso especial de 4 rows 4 cols (para giros)
            case 0: piece_rows = 4; piece_cols = 4;
                piece = new Block[piece_rows][piece_cols]; makeStick(); piecePositions = 2; break;
            // construimos el cubo
            case 1: piece = new Block[piece_rows][piece_cols]; makeCube(); piecePositions = 1; break;
            // construimos la T
            case 2: piece = new Block[piece_rows][piece_cols]; makeT();  piecePositions = 4; break;
            // construimos la L izq
            case 3: piece = new Block[piece_rows][piece_cols]; makeLeftL();  piecePositions = 4;
break;
            // construimos la L der
            case 4: piece = new Block[piece_rows][piece_cols]; makeRightL(); piecePositions = 4;
break;
            // construimos la S izq
            case 5: piece = new Block[piece_rows][piece_cols]; makeLeftS(); piecePositions = 4; break;
            // construimos la S der
            case 6: piece = new Block[piece_rows][piece_cols]; makeRightS(); piecePositions = 4;
break;
            // Resto de casos no se hace nada. No puede ocurrir, usaremos rand(7) como máximo.
            default: ;
        }
    }

    /**
     * Acceso a las posiciones posibles de la pieza para realizar los giros
     * @return El número de posiciones relevantes de la pieza en un giro
     */
    public int getPositions()
    {
        return piecePositions;
    }

    /**
     * Devuelve el objeto almacenado en una de las celdas de la matriz
     * @param row La coordenada fila de la celda
     * @param col La coordenada columna de la celda
     * @return El objeto bloque almacenado en esa celda, o null si no hay ninguno
     */
    public Block getBlock(int row, int col)
    {
        return piece[row][col];
    }

    /**
     * Añade un bloque en una posición específica de la matriz de la pieza
     * @param p_row La fila donde vamos a añadir el bloque
     * @param p_col La columna donde vamos a añadir el bloque
     * @param color El color del bloque que vamos a añadir
     */
```

```
private void addOneBlock(int p_row, int p_col, Color color)
{
   piece[p_row][p_col] = new Block(color, p_row, p_col);
}

/**
 * Crea la configuración de la matriz para representar el tetrimino "palo"
 */
private void makeStick()
{
   addOneBlock(1,0, Color.red);
   addOneBlock(1,1, Color.red);
   addOneBlock(1,2, Color.red);
   addOneBlock(1,3, Color.red);
}

/**
 * Crea la configuración de la matriz para representar el tetrimino "cubo"
 */
private void makeCube()
{
   addOneBlock(0,1,Color.blue);
   addOneBlock(0,2,Color.blue);
   addOneBlock(1,1,Color.blue);
   addOneBlock(1,2,Color.blue);
}

/**
 * Crea la configuración de la matriz para representar el tetrimino "T"
 */
private void makeT()
{
   addOneBlock(0,1,Color.yellow);
   addOneBlock(1,0,Color.yellow);
   addOneBlock(1,1,Color.yellow);
   addOneBlock(1,2,Color.yellow);
}

/**
 * Crea la configuración de la matriz para representar el tetrimino "L izquierda"
 */
private void makeLeftL()
{
   addOneBlock(0,0,Color.green);
   addOneBlock(1,0,Color.green);
   addOneBlock(1,1,Color.green);
   addOneBlock(1,2,Color.green);
}

/**
 * Crea la configuración de la matriz para representar el tetrimino "L derecha"
 */
```

```
    private void makeRightL()
    {
        addOneBlock(0,2,Color.magenta);
        addOneBlock(1,0,Color.magenta);
        addOneBlock(1,1,Color.magenta);
        addOneBlock(1,2,Color.magenta);
    }

    /**
     * Crea la configuración de la matriz para representar el tetrimino "S izquierda"
     */
    private void makeLeftS()
    {
        addOneBlock(0,0,Color.cyan);
        addOneBlock(0,1,Color.cyan);
        addOneBlock(1,1,Color.cyan);
        addOneBlock(1,2,Color.cyan);
    }

    /**
     * Crea la configuración de la matriz para representar el tetrimino "S derecha"
     */
    private void makeRightS()
    {
        addOneBlock(0,1,Color.orange);
        addOneBlock(0,2,Color.orange);
        addOneBlock(1,0,Color.orange);
        addOneBlock(1,1,Color.orange);
    }


    /**
     * Acceso a la altura de la matriz de la pieza
     * @return El número de filas que tiene la matriz de la pieza
     */
    public int getPieceRows()
    {
        return piece_rows;
    }

    /**
     * Acceso a la anchura de la matriz de la pieza
     * @return El número de columnas que tiene la matriz de la pieza
     */
    public int getPieceCols()
    {
        return piece_cols;
    }

}
```

Clase Grid

```
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;
import java.awt.Color;
import java.util.ArrayList;

/**
 * Representa una matriz bidimensional de celdas en cada
 * una de las cuales se puede almacenar una de los 4 bloques
 * que forman cada tetrimino.
 *
 * @author Octavio Martínez
 * @version 16.05.2011
 */
public class Grid
{
    // variable aleatoria usada para la creación de los tetriminos
    private static final Random rand = new Random();
    // la anchura (columnas) por defecto de la matriz
    public static final int DEFAULT_COLS = 12;
    // la altura (filas) por defecto de la matriz.
    public static final int DEFAULT_ROWS = 25;
    // controla el numero de giros de la pieza
    private int turns;

    // Las dimensiones 2D de la matriz, filas y columnas que posee.
    private int rows, cols;
    // El Array bidimensional que almacenará los bloques de las piezas o tetriminos.
    private Block[][] grid;
    // La pieza actual
    private Piece tetrimino;
    // mantiene la noción en el grid de bloque central del tetrimino actual.
    private Block central;


    /**
     * Construye una matriz de las dimensiones especificadas
     * @param rows Las filas de la matriz
     * @param cols Las columnas de la matriz
     */
    public Grid(int rows, int cols)
    {
        this.rows = rows;
        this.cols = cols;
        grid = new Block[rows][cols];
        tetrimino = null;
        central = null;
    }
```

```
/**
 * Vacía el grid poniendo a null todas sus posiciones
 */
public void clear()
{
   for(int row = 0; row < rows; row++) {
      for(int col = 0; col < cols; col++) {
         grid[row][col] = null;
      }
   }
}

/**
 * Pone un bloque de la pieza en una posición
 * dada de la matriz y establece sus coords internas para que coincidan
 * con la posición real dentro del grid
 * @param element El bloque que queremos poner
 * @param row La fila en que queremos ubicarle
 * @param col La columna en que queremos ubicarle
 */
private void placePiece(Block element, int row, int col)
{
   if(element != null) {
      // establecemos los atributos del bloque para que coincidan con
      // su posición real en el grid
      element.setLocation(row,col);
      grid[row][col] = element;
   }
}

/**
 * Devuelve el objeto en la celda especificada (fila-columna)
 * @param row La fila.
 * @param col La columna.
 * @return el objeto en la posición dada, o null si no hay ninguno
 */
public Block getObjectAt(int row, int col)
{
   if(grid[row][col] != null) {
      return grid[row][col];
   } else {
      return null;
   }
}

/**
 * Devuelve el color de un determinado bloque en la matriz
 * @param row La fila donde vamos a buscar
 * @param col La columna donde vamos a buscar.
 * @return El color de un bloque presente en la matriz, o null si no hay bloque
 */
```

```
public Color getCellColor(int row, int col)
{
   Color color = getObjectAt(row, col).getBlockColor();
   return color;
}

/**
 * Vacía una celda concreta de la matriz estableciendo a null su referencia
 * @param row La fila de la matriz
 * @param col la columna de la matriz
 */
private void eraseCell(int row, int col)
{
   grid[row][col] = null;
}

/**
 * Obtiene un bloque de una ubicación del grid y lo elimina de esa
 * posición
 * @param row La fila en la que está el bloque
 * @param col La columna en la que está el bloque
 * @return El bloque que estaba en esa posición si había alguno, null de otro modo
 */
private Block cutBlock(int row, int col)
{
   if(grid[row][col] != null) {
      Block block = grid[row][col];
      grid[row][col] = null;
      return block;
   } else {
      return null;
   }
}

/**
 * Establece todas los bloques de las piezas del grid
 * como "apilados" para tenerlo en cuenta y no moverlas
 * al recorrer la matriz en otros métodos de modificación
 */
private void setStacked()
{
    for(int row = DEFAULT_ROWS - 1; row >= 0; row--) {
      for(int col = DEFAULT_COLS - 1; col >= 0; col--) {
         Block block = getObjectAt(row,col);
         if(block != null) {
            block.setStacked();
         }
      }
   }
}

/**
```

```
 * Comprueba si todas los bloques de las piezas en el grid están apilados
 * @return True si estan todos apilados, false en caso contrario
 */
public boolean allStacked()
{
   boolean check = true;
   for(int row = DEFAULT_ROWS - 1; row >= 0; row--) {
      for(int col = DEFAULT_COLS - 1; col >= 0; col--) {
         Block block = getObjectAt(row,col);
         if(block != null && !block.isStacked()) {
            check = false;
         }
      }
   }
   return check;
}


/**
 * Crea un nuevo tetrimino en la posición inicial del grid,
 * es decir, en la zona superior central
 */
public void createTetrimino()
{
   turns = 0;
   int dice7 = rand.nextInt(7);
   tetrimino = new Piece(dice7);
   // marca en bloque central de la pieza para seguirle en el grid.
   central = tetrimino.getBlock(1,1);
   // colocar en el grid un bloque si !null en la submatriz Piece
   for( int pieceRow = 0; pieceRow < tetrimino.getPieceRows(); pieceRow++) {
      for( int pieceCol = 0; pieceCol < tetrimino.getPieceCols(); pieceCol++) {
         if(tetrimino.getBlock(pieceRow, pieceCol) != null) {
            placePiece(tetrimino.getBlock(pieceRow, pieceCol), pieceRow, pieceCol + 4);
         }
      }
   }
}

/**
 * Acceso al campo tetrimino de esta clase
 * @return El campo tetrimino de una instancia de esta clase
 */
private Piece getTetrimino()
{
   return tetrimino;
}

/**
 * Devuelve la altura (filas) de la matriz.
 * @return La altura de la matriz.
 */
```

```
public int getHeight()
{
   return rows;
}

/**
 * Devuelve el ancho (cols) de la matriz
 * @return El ancho de la matriz.
 */
public int getWidth()
{
   return cols;
}

/**
 * Devuelve el numero de giros de la pieza
 * @return El número de giros efectuados por una pieza dentro del grid
 */
private int getTurns()
{
   return turns;
}

/**
 * Pone el contador de giros a un valor determinado
 * @param value El valor al que queremos establecer el campo turns
 */
public void setTurns(int value)
{
   turns = value;
}

/**
 * Devuelve un array integrado
 * por los 4 bloques de la pieza móvil actual
 * @return Una lista de los 4 bloques pertenecientes a la pieza móvil actual
 */
private ArrayList<Block> getBlocksAtGrid()
{
   ArrayList<Block> blocksAtGrid = new ArrayList<Block>();
   for(int row = 0; row <= DEFAULT_ROWS-1; row++){
      for(int col = 0; col <= DEFAULT_COLS-1; col++) {
         Block block = getObjectAt(row,col);
         // si existe bloque en esa posición y no está apilado
         if(block != null && !block.isStacked()) {
            blocksAtGrid.add(block);
         }
      }
   }
   return blocksAtGrid;
}
```

```
/**
 * Mueve toda la pieza una posición a la derecha en al grid
 */
public void moveRight()
{
   // si el movimiento a la derecha es legal
   if(canMoveRight()) {
      for(int col = DEFAULT_COLS - 1; col >= 0; col--) {
         for(int row = DEFAULT_ROWS - 1; row >= 0; row--) {
            Block block = getObjectAt(row,col);
            // si existe bloque en esa posición y no está apilado
            if(block != null && !block.isStacked()) {
               // movemos a la derecha en bloque y borramos el anterior
               placePiece(block, row, col+1);
               eraseCell(row,col);


            }
         }
      }
   }
}

/**
 * Verifica que todos los elementos a la derecha de los bloques de una pieza
 * esten libres.
 * @return True si la pieza puede moverse a la derecha, False en caso contrario
 */
private boolean canMoveRight()
{
      boolean check = true;
      // almacenamos los bloques de la pieza en una colección
      ArrayList<Block> currentPiece = getBlocksAtGrid();
      // verificamos que para cada bloque haya una posición válida a su derecha
      for(Block pieceBlock : currentPiece) {
         int row = pieceBlock.getRow();
         int col = pieceBlock.getCol();
         if(col+1 < DEFAULT_COLS) {
            Block nextBlock = getObjectAt(row,col+1);
            if(nextBlock != null && nextBlock.isStacked()) {
               check = false;
            }
         } else {
            check = false;
         }
      }
      return check;

}

/**
 * Mueve toda la pieza una posición a la izquierda en el grid
```

```
 */
public void moveLeft()
{
    // si el movimiento a la izquierda es legal
    if(canMoveLeft()) {
      for(int col = 0; col <= DEFAULT_COLS - 1; col++) {
         for(int row = DEFAULT_ROWS - 1; row >= 0; row--) {
            Block block = getObjectAt(row,col);
            // si existe bloque en esa posición y no está apilado
            if(block != null && !block.isStacked()) {
               // movemos el bloque a la izquierda y borramos anterior
               placePiece(block, row, col-1);
               eraseCell(row,col);

            }
         }
      }
   }
}

/**
 * Verifica que todos los elementos a la derecha de los bloques de una pieza
 * esten libres.
 * @return True si la pieza puede moverse a la izquierda, False en caso contrario
 */
private boolean canMoveLeft()
{
      boolean check = true;
      // almacenamos los bloques de la pieza en una colección
      ArrayList<Block> currentPiece = getBlocksAtGrid();
      // verificamos que para cada bloque haya una posición válida a su izquierda
      for(Block pieceBlock : currentPiece) {
         int row = pieceBlock.getRow();
         int col = pieceBlock.getCol();
         if(col-1 > -1) {
            Block nextBlock = getObjectAt(row,col-1);
            if(nextBlock != null && nextBlock.isStacked()) {
               check = false;
            }
         } else {
            check = false;
         }
      }
      return check;
}

/**
 * Desciende toda la pieza una posición en la matriz
 */
public void moveDown()
{
   // si el movimiento descendente es legal
```

```
    if(canMoveDown()) {
        for(int row = DEFAULT_ROWS - 1; row >= 0; row--) {
            for(int col = DEFAULT_COLS - 1; col >= 0; col--) {
                Block block = getObjectAt(row,col);
                // Si hay un bloque y éste no está apilado
                if(block != null && !block.isStacked()) {
                    // bajamos una fila el bloque y borramos anterior
                    placePiece(block, row + 1, col);
                    eraseCell(row,col);
                }
            }
        }
    } else {
        // Marcar todos los bloques como apilados
        setStacked();
        // el actual tetrimino se mezcla con el resto de bloques apilados
        // de modo que ya no existe como pieza
        tetrimino = null;
    }
}

/**
 * Verifica que todos los elementos por debajo de los bloques de una pieza
 * esten libres.
 * @return True si la pieza puede descender, False en caso contrario
 */
private boolean canMoveDown()
{
        boolean check = true;
        ArrayList<Block> currentPiece = getBlocksAtGrid();
        for(Block pieceBlock : currentPiece) {
            int row = pieceBlock.getRow();
            int col = pieceBlock.getCol();
            if(row+1 < DEFAULT_ROWS) {
                Block nextBlock = getObjectAt(row+1,col);
                if(nextBlock != null && nextBlock.isStacked()) {
                    check = false;
                }
            } else {
                check = false;
            }
        }
        return check;

}

/**
 * Intenta girar la pieza 90º si hay espacio para tal movimiento
 */
public void turnPiece()
{
    // si tiene espacio para girar, no ha girado ya 4 veces y no es ni palo ni cubo
```

```
        if(perimeterFree() && tetrimino.getPositions() > 2 && turns < 4) {
            // "cortamos" los bloques de su posición original
            Block zero0 = cutBlock(central.getRow()-1, central.getCol()-1);
            Block zero1 = cutBlock(central.getRow()-1, central.getCol());
            Block zero2 = cutBlock(central.getRow()-1, central.getCol()+1);
            Block one0 = cutBlock(central.getRow(), central.getCol()-1);
            Block one1 = central;
            Block one2 = cutBlock(central.getRow(), central.getCol()+1);
            Block two0 = cutBlock(central.getRow()+1, central.getCol()-1);
            Block two1 = cutBlock(central.getRow()+1, central.getCol());
            Block two2 = cutBlock(central.getRow()+1, central.getCol()+1);

            // los colocamos en sus nuevas posiciones excepto el central que no varía
            placePiece(two0,central.getRow()-1, central.getCol()-1);
            placePiece(one0,central.getRow()-1, central.getCol());
            placePiece(zero0,central.getRow()-1, central.getCol()+1);
            placePiece(two1,central.getRow(), central.getCol()-1);
            placePiece(zero1,central.getRow(), central.getCol()+1);
            placePiece(two2,central.getRow()+1, central.getCol()-1);
            placePiece(one2,central.getRow()+1, central.getCol());
            placePiece(zero2,central.getRow()+1, central.getCol()+1);
            // actualizamos el contador de giros
            turns++;
        // en el caso del tetrimino "stick"
        }  else if (stickPerimeterFree() && tetrimino.getPositions() == 2 && turns < 4) {
            turnStick();
            turns++;
        }
    }

    /**
     * Verifica que todas las posiciones alrededor del bloque central
     * sean válidas para poder realizar los giros correctamente
     * @return True si tiene un perímetro libre para realizar el giro, false en caso contrario
     */
    private boolean perimeterFree()
    {
        boolean check = true;
        for(int offsetY = central.getRow()-1; offsetY <= central.getRow()+1; offsetY++) {
            for(int offsetX = central.getCol()-1; offsetX <= central.getCol()+1; offsetX++) {
                // si las posiciones están dentro de los límites válidos
                if(offsetY >= 0 && offsetY < DEFAULT_ROWS && offsetX >= 0 && offsetX <
DEFAULT_COLS) {
                    Block block = getObjectAt(offsetY,offsetX);
                    // si en esa posicion hay un bloque y está apilado no puede girar
                    if(block != null && block.isStacked()) {
                        check = false;
                    }
                // si el perimetro está fuera de limites válidos no puede girar
                } else {
                    check = false;
                }
```

```
            }
        }
        return check;
    }

    /**
     * Rota la pieza específica "stick"
     */
    private void turnStick()
    {
        // "cortamos" los bloques de su posición orginal la cual puede ser:
        // (----) o bien:
        // ( |  )
        // ( |  )
        // ( |  )
        // ( |  )
        Block zero1 = cutBlock(central.getRow()-1, central.getCol());
        Block one0 = cutBlock(central.getRow(), central.getCol()-1);
        Block one1 = central;
        Block one2 = cutBlock(central.getRow(), central.getCol()+1);
        Block one3 = cutBlock(central.getRow(), central.getCol()+2);
        Block two1 = cutBlock(central.getRow()+1, central.getCol());
        Block three1 = cutBlock(central.getRow()+2, central.getCol());

        // les colocamos en su nueva posición girados 90º
        placePiece(one0,central.getRow()-1, central.getCol());
        placePiece(one2,central.getRow()+1, central.getCol());
        placePiece(one3,central.getRow()+2, central.getCol());
        placePiece(zero1,central.getRow(), central.getCol()-1);
        placePiece(two1,central.getRow(), central.getCol()+1);
        placePiece(three1,central.getRow(), central.getCol()+2);

    }

    /**
     * Verifica que todas las posiciones alrededor del bloque central
     * sean válidas para poder realizar los giros correctamente para
     * el caso específico del stick
     */
    private boolean stickPerimeterFree()
    {
        boolean check = true;
        for(int offsetY = central.getRow()-1; offsetY <= central.getRow()+2; offsetY++) {
            for(int offsetX = central.getCol()-1; offsetX <= central.getCol()+2; offsetX++) {
                if(offsetY >= 0 && offsetY < DEFAULT_ROWS && offsetX >= 0 && offsetX <
DEFAULT_COLS) {
                    Block block = getObjectAt(offsetY,offsetX);
                    if(block != null && block.isStacked()) {
                        check = false;
                    }
                } else {
                    check = false;
```

```
            }
        }
    }
    return check;
}

/**
 * Verifica si existe una fila del grid completa con bloques
 * y los elimina en caso afirmativo (hacer "linea" en el juego original)
 * @param row La fila que queremos comprobar
 * @return True si está llena de bloques, False en caso contrario
 */
private boolean line(int row)
{
    boolean check = true;
    for(int col = 0; col < DEFAULT_COLS; col++) {
        // si hay alguna posición vacía o no apilada devuelve false
        if(getObjectAt(row,col) == null || !getObjectAt(row,col).isStacked()) {
            check = false;
        }
    }
    // si la línea esta completa con bloques borra toda la línea
    if(check == true) {
        for(int col = 0; col < DEFAULT_COLS; col++) {
            eraseCell(row,col);
            //return check;
        }
    }
    return check;
}

/**
 * Recorre la matriz de abajo a arriba buscando líneas completas
 * y eliminándolas, moviendo hacia abajo los bloques superiores
 */
public void searchLines()
{
    for(int checkRow = DEFAULT_ROWS-1; checkRow > 0; checkRow--) {
        // nos aseguramos que mientras existan "líneas" se eliminen y se bajen los de encima
        while(line(checkRow) == true) {
            // desde la fila inmediatamente superior a la cual en la que hizo "linea"
            for(int row = checkRow-1; row >= 0; row--) {
                for(int col = 0; col < DEFAULT_COLS; col++) {
                    Block block = getObjectAt(row,col);
                    if(block != null && row+1 < DEFAULT_ROWS) {
                        // bajamos el bloque una fila y eliminamos el anterior
                        placePiece(block, row+1, col);
                        eraseCell(row,col);
                    }
                }
            }
        }
```

```
        }
    }


}
```

Clase GameView

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.LinkedHashMap;
import java.util.Map;

/**
 * Representa una vista gráfica de la pantalla de juego, como una matriz
 * de 25 filas por 12 columnas, en la cual cada celda puede almacenar un objeto.
 * Cada posición de la matriz se representa como un rectángulo con un color
 * definido, en función de si están vacías (null) o no.
 * Se puede establecer un color para cada tipo de objeto mediante setColor.
 *
 * @author Octavio Martínez
 * @version 16.05.2011
 */
public class GameView extends JFrame
{
    // Usamos el color negro por defecto para posiciones vacías
    private final Color EMPTY_COLOR = Color.black;
    // instancia de la clase interna que proporciona la forma de pintar la matriz
    private GridView gridView;
    // elementos del menu del juego
    private JMenuItem start, stop, resume, reset, quit;

    /**
     * Construye una vista gráfica de la pantalla de juego
     * @param height La altura de la matriz de juego.
     * @param width  La anchura de la matriz de juego.
     */
    public GameView(int height, int width)
    {
        setTitle("Tetris");
        setLocation(800, 50);

        gridView = new GridView(height, width);

        Container contents = getContentPane();
        contents.add(gridView, BorderLayout.CENTER);

        makeMenuBar();

        pack();
        setVisible(true);
```

```
}

/**
 * Crea una barra de menú con opciones básicas del juego
 */
private void makeMenuBar()
{
   JMenuBar menuBar = new JMenuBar();
   setJMenuBar(menuBar);

   JMenu game = new JMenu("Game");
   menuBar.add(game);

   start = new JMenuItem("Start");
   game.add(start);

   stop = new JMenuItem("Pause");
   game.add(stop);

   resume = new JMenuItem("Resume");
   game.add(resume);

   reset = new JMenuItem("Reset");
   game.add(reset);

   quit = new JMenuItem("Quit");
   game.add(quit);
}

/**
 * Acceso al elemento de menú Start
 */
public JMenuItem getStart()
{
   return start;
}

/**
 * Acceso al elemento de menú Stop
 */
public JMenuItem getStop()
{
   return stop;
}

/**
 * Acceso al elemento de menú Resume
 */
public JMenuItem getResume()
{
   return resume;
}
```

```
/**
 * Acceso al elemento de menú Reset
 */
public JMenuItem getReset()
{
   return reset;
}

/**
 * Acceso al elemento de menú Quit
 */
public JMenuItem getQuit()
{
   return quit;
}

/**
 * Muestra en pantalla el estado actual de la matriz que representa
 * la pantalla de juego diferenciando por colores entre celdas
 * vacías y ocupadas por bloques
 * @param grid La matriz sobre la cual mostramos su estado actual.
 */
public void showStatus(Grid grid)
{
   // hacemos visible el frame si aún no lo es
   if(!isVisible())
      setVisible(true);

   gridView.preparePaint();

   for(int row = 0; row < grid.getHeight(); row++) {
      for(int col = 0; col < grid.getWidth(); col++) {
         // si hay un bloque en esa celda lo pintamos de su color y en 3D
         if(grid.getObjectAt(row, col) != null) {
            gridView.drawMark3D(col, row, grid.getCellColor(row, col));
         // si no hay bloque pintamos negro y en 2D
         } else {
            gridView.drawMark2D(col, row, EMPTY_COLOR);
         }
      }
   }
   // llamamos implícitamente a nuestro paintComponent desde aquí para pintar el estado
   gridView.repaint();
}

/**
 * Clase interna de GameView que proporciona una forma de mostrar
 * en pantalla la matriz de juego rectangular, pintando rectángulos
 * de igual tamaño para cada celda de la matriz (posición: row-col)
 * y diferenciando por medio de colores las celdas vacías
 * (en negro) de las que están ocupadas por bloques de las piezas del
```

```
 * tetris (en otros colores).
 */
private class GridView extends JPanel
{
   // la escala de los rectángulos que vamos a pintar
   private final int GRID_VIEW_SCALING_FACTOR = 15;

   // la anchura y altura de la matriz que vamos a pintar
   private int gridWidth, gridHeight;
   // las proporciones horizontales y verticales de los rectángulos
   private int xScale, yScale;
   // las dimensiones 2D del rectángulo de juego
   Dimension size;
   // el contexto gráfico necesario para pintar en los componentes
   private Graphics g;
   // la imagen que vamos a pintar
   private Image gridImage;

   /**
    * Construye un nuevo componente interno de la matriz dibujable.
    * @param height La altura de la matriz
    * @param width La anchura de la matriz
    */
   public GridView(int height, int width)
   {
      gridHeight = height;
      gridWidth = width;
      size = new Dimension(0, 0);
   }

   /**
    * Dice al gestor de la interfaz gráfica el tamaño del área del juego
    * @return El área total en 2D del rectángulo de juego.
    */
   public Dimension getPreferredSize()
   {
      return new Dimension(gridWidth * GRID_VIEW_SCALING_FACTOR,
                           gridHeight * GRID_VIEW_SCALING_FACTOR);
   }

   /**
    * Inicializa las variables necesarias para poder pintar
    * la matriz y tiene en cuenta si hemos modificado el tamaño
    * para escalarlo adecuadamente.
    */
   public void preparePaint()
   {
      // si el tamaño ha cambiado
      if(!size.equals(getSize())) {
         size = getSize();
         gridImage = gridView.createImage(size.width, size.height);
         g = gridImage.getGraphics();
```

```
        xScale = size.width / gridWidth;
        if(xScale < 1) {
            xScale = GRID_VIEW_SCALING_FACTOR;
        }
        yScale = size.height / gridHeight;
        if(yScale < 1) {
            yScale = GRID_VIEW_SCALING_FACTOR;
        }
    }
}

/**
 * Pinta en 3D y de un determinado color cada rectángulo que representa
 * una posición de la matriz
 * @param x La coordenada x de la posición
 * @param y La coordenada y de la posición
 * @param color El color designado para esa celda.
 */
public void drawMark3D(int x, int y, Color color)
{
    g.setColor(color);
    g.fill3DRect(x * xScale, y * yScale, xScale-1, yScale-1, true);
}

/**
 * Pinta en 2D de un determinado color cada rectángulo que representa
 * una posición de la matriz
 * @param x La coordenada x de la posición
 * @param y La coordenada y de la posición
 * @param color El color designado para esa celda.
 */
public void drawMark2D(int x, int y, Color color)
{
    g.setColor(color);
    g.fillRect(x * xScale, y * yScale, xScale-1, yScale-1);
}

/**
 * Copia el componente que representa la imagen interna
 * dibujable de la matriz a la pantalla
 * para ser mostrado y le pinta.
 * @param g El objeto de contexto gráfico que permite dibujar en los componentes
 */
public void paintComponent(Graphics g)
{
    if(gridImage != null) {
        Dimension currentSize = getSize();
        if(size.equals(currentSize)) {
            g.drawImage(gridImage, 0, 0, null);
        // Si el tamaño del componente ha variado se pinta reescalado.
        } else {
```

```
                g.drawImage(gridImage, 0, 0, currentSize.width, currentSize.height, null);
            }
        }
    }
}
```

Clase Tetris

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.awt.Color;
import javax.swing.Timer;
import java.awt.event.*;
import javax.swing.*;

/**
 * Una simulación del juego Tetris con las funcionalidades
 * básicas del juego original
 *
 * @author Octavio Martínez
 * @version 17.05.2011
 */
public class Tetris
{
    // Retraso preestablecido para el temporizador que controla la caida de las piezas.
    private static final int TIME_DELAY = 1000;
    // la matriz 2D del juego
    private Grid grid;
    // La interfaz gráfica de la matriz del juego
    private GameView gameView;
    // Un temporizador para controlar la caida automática de las piezas
    private Timer timer;

    /**
     * Crea un tetris con valores de matriz por defecto 25x12
     */
    public Tetris()
    {
        // crea una matriz y una vista gráfica de la misma de dimensiones por defecto
        grid = new Grid(Grid.DEFAULT_ROWS, Grid.DEFAULT_COLS);
        gameView = new GameView(Grid.DEFAULT_ROWS, Grid.DEFAULT_COLS);
        // añade el oyente de eventos por teclado para controlar las piezas
        gameView.addKeyListener(new ArrowListener());

        // comienza el juego con el elemento del menu "Start"
        gameView.getStart().addActionListener(new ActionListener()
            {public void actionPerformed(ActionEvent e) {startGame();} });

        // para el juego con el elemento del menu "Pause"
        gameView.getStop().addActionListener(new ActionListener()
```

```
        {public void actionPerformed(ActionEvent e) {stopTimer();} });

    // continua el juego con el elemento del menu "Resume"
    gameView.getResume().addActionListener(new ActionListener()
        {public void actionPerformed(ActionEvent e) {startGame();} });

    // resetea el juego con el elemento del menu "Reset"
    gameView.getReset().addActionListener(new ActionListener()
        {public void actionPerformed(ActionEvent e) {reset();} });

    // sale del juego con el elemento del menu "Quit"
    gameView.getQuit().addActionListener(new ActionListener()
        {public void actionPerformed(ActionEvent e) {System.exit(0);} });

    // pinta la pantalla de juego
    gameView.showStatus(grid);

    // crea una nueva pieza en su ubicación inicial en la matriz
    newTetrimino();
    // comienza la ejecución del juego por medio del temporizador
    startGame();
}

/**
 * Reestablece el juego a un estado inicial, vaciando la matriz de bloques
 */
private void reset()
{
    grid.clear();
    gameView.showStatus(grid);
}

/**
 * Aparece la nueva pieza
 */
private void newTetrimino()
{
    grid.createTetrimino();
    gameView.showStatus(grid);
}

/**
 * Implementa el movimiento descendente de las piezas en la
 * matriz de juego mediante un temporizador fijado con
 * un retraso de 1000 milisegundos.
 */
private void startGame()
{
    // crea el objeto temporizador
    timer = new Timer(TIME_DELAY, new FallActionListener());
    // inicia el proceso de temporización
    timer.start();
```

```
}

/**
 * Detiene el temporizador.
 */
private void stopTimer()
{
   timer.stop();
}

/**
 * Moviemiento descendente de toda la pieza conjuntamente
 * en el Grid.
 */
private void fall()
{
   grid.moveDown();
   gameView.showStatus(grid);
}

/**
 * Mueve a la derecha una pieza en el grid si es posible dicho movimiento
 */
private void moveRight()
{
   grid.moveRight();
   gameView.showStatus(grid);
}

/**
 * Mueve a la izquierda una pieza en el grid si es posible dicho movimiento
 */
private void moveLeft()
{
   grid.moveLeft();
   gameView.showStatus(grid);
}

/**
 * Intenta girar 90º una pieza en la matriz si dicho giro es válido
 */
private void turnTetrimino()
{
   grid.turnPiece();
   gameView.showStatus(grid);
}


/**
 * Clase interna que servirá de oyente de eventos para
 * la caída de los bloques. En cada paso del timer realiza su acción,
 * que consiste en bajar una fila la pieza, poner a 0 el contador de giros
```

```
     * y hacer "linea" si se dan las condiciones
     */
    public class FallActionListener implements ActionListener
    {
       public void actionPerformed(ActionEvent event)
       {
          fall();
          // reset del contador de giros para permitir 4 por segundo
          grid.setTurns(0);
          grid.searchLines();
          // si toca fondo la pieza se crea una nueva
          if(grid.allStacked()) {
             newTetrimino();
          }
          // si las piezas llegan arriba de la matriz paramos el timer y se acaba el juego
          for(int col = grid.DEFAULT_COLS-1; col >= 0; col--) {
             if(grid.getObjectAt(0,col) != null && grid.getObjectAt(0,col).isStacked()) {
                stopTimer();
                JOptionPane gameOver = new JOptionPane();
                gameOver.showMessageDialog(gameView, "GAME OVER!!", "Game Over",
gameOver.INFORMATION_MESSAGE);

             }
          }
       }


    }

    /**
     * Clase interna que implementa KeyListener usada para realizar las acciones
     * de mover y girar al presionar las teclas de dirección apropiadas
     */
    public class ArrowListener implements KeyListener
    {
       public void keyPressed(KeyEvent e)
       {
          if(e.getKeyCode() == 39) {
             moveRight();
          } else if (e.getKeyCode() == 37) {
             moveLeft();
          } else if (e.getKeyCode() == 38) {
             turnTetrimino();
          } else if (e.getKeyCode() == 40) {
             fall();
          }
       }

       public void keyReleased(KeyEvent e2)
       {

       }
```

```
        public void keyTyped(KeyEvent e3)
        {

        }
    }

    /**
     * Rutina main para ejecutar el programa fuera de BlueJ
     */
    public static void main(String[] args)
    {
        Tetris tetris = new Tetris();
    }

}
```