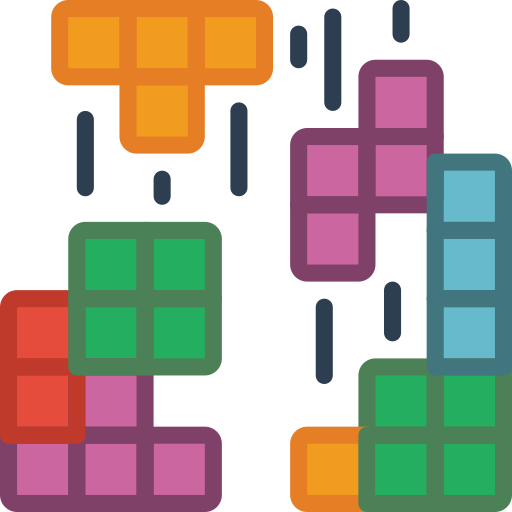
**Reto 2: ABSTRACCIÓN**

David Kessler Martínez

Isabel Morro Tabares

Antonio Javier Rodríguez Romero

****

**class Pieza {**

private:

/\*\*

\* @brief Guardará el alto que podrá ocupar una pieza como máximo. Se podrá

\* modificar en función de la rotación de la pieza.

\*/

int **height**

/\*\*

\* @brief Guardará el ancho que podrá ocupar la pieza. Se podrá

\* modificar en función de la rotación de la pieza.

\*/

int **width**

/\*\*

\* @brief Guardará el color de la pieza en un *string.* Este tendrá que ser uno de los

\* disponibles para las piezas, que se especificarán antes de empezar la partida

\*/

string **color**

/\*\*

\* @brief matriz de tipo bool donde se guardarán las posiciones que la pieza ocupa

\* (*true* si está ocupada y *false* si está libre)

\*/

bool **\_data**[height][width]

/\*\*

\* @brief Guardará la imagen de la pieza que aparecerá en la cola.

\*/

Image **img\_pieza**

public:

// Constructores

/\*\*

\*@brief Creará una pieza aleatoria, asignando una al azar de la lista de 7 disponibles

\* y rellenará la matriz según las posiciones que ocupe la pieza, guardará el color

\* correspondiente a esta y la imagen de esta en el campo *img\_pieza.*

\* post: width=height=0

\*/

**Pieza** ()

// Métodos

/\*\*

\*@brief Nos informa de si una casilla de la pieza está ocupada o no

\* @param i Fila de la posición

\* @param j Columna de la posición

\*@return *true* si está ocupada y *false* si no lo está

\*/

bool **get\_pos**(int i, int j)

/\*\*

\* @brief Pone una de las posiciones de la matriz de la pieza en ocupada o en libre,

\* @param i Fila de la posición

\* @param j Columna de la posición

\* @param occupied Valor que se le asignará a la posición (i,j)

\*/

void **set\_pos**(int i, int j, bool occupied)

/\*\*

\* @brief Gira la pieza a la derecha, modificando los campos alto y ancho a los

\* correspondientes una vez se realiza el giro.

\*/

void **rotate\_right**()

/\*\*

\* @brief Gira la pieza a la izquierda, modificando los campos alto y ancho a los

\* correspondientes una vez se realiza el giro.

\*/

void **rotate\_left**()

}; // class Pieza

**class Tablero {**

private:

/\*\*

\* @brief Guardará el color que tendrán las casillas vacías.

\* @pre Tendrá el valor predeterminado GRIS

\*/

static const string **COLOR\_EMTPY**

/\*\*

\* @brief Ancho del tablero. Se podrá modificar antes de comenzar la partida, pero

\* durante esta se mantendrá constante.

\*/

int **width**

/\*\*

\* @brief Alto del tablero. Se podrá modificar antes de comenzar la partida, pero

\* durante esta se mantendrá constante.

\*/

int **height**

/\*\*

\* @brief Almacena las casillas ocupadas en el tablero por fichas y el color de cada una \* de ellas. En el caso de que no estén ocupadas, se le asignará un color por defecto a \* las vacías (la constante COLOR\_EMPTY)

\*/

string **\_data**[height][width]

/\*\*

\* @brief Es la pieza que está cayendo ahora mismo

\*/

Pieza **pieza\_fall**

public:

// Constructores

/\*\*

\*@brief Creará un tablero vacío con el ancho y el alto igual a 0.

\* @post: width=height=0

\*/

**Tablero** ()

/\*\*

\*@brief Creará un tablero vacío con el ancho y el alto igual a **w** y **h**, respectivamente,

\* pasados como argumentos.

\* @post: width=w height=h

\*/

**Tablero** (int w, int h)

// Métodos

/\*\*

\*@brief Nos da el valor del ancho del tablero

\*@return el valor del ancho

\*/

int **get\_width**()

/\*\*

\*@brief Cambia el valor del ancho del tablero a **w**

\*@param **w**, valor del ancho del tablero

\*@pre w > 0

\*/

void **set\_width**(int w)

/\*\*

\*@brief Nos da el valor del alto del tablero

\*@return el valor del alto

\*/

int **get\_height**()

/\*\*

\*@brief Cambia el valor del alto del tablero a **h**

\*@param **h**, valor del alto del tablero

\*@pre h > 0

\*/

void **set\_height**(int h)

/\*\*

\*@brief Comprueba si una posición está ocupada o no consultando su color y viendo

\* si coincide con la constante **COLOR\_EMPTY**

\*@param **i**, coordenada de la posición del tablero a consultar

\*@param **j**, coordenada de la posición del tablero a consultar

\*@return *true* si está ocupada y *false* si no lo está

\*@pre i > 0, j > 0

\*/

bool **get\_pos**(int i, int j)

/\*\*

\*@brief Copia la pieza que se le pasa como argumento al campo *pieza\_fall*, es decir,

\* añade una pieza al tablero en la parte superior que empezará a caer.

\*@param n, pieza que será la nueva *pieza\_fall*

\*/

void **add**\_**pieza** (Pieza n)

/\*\*

\*@brief mueve una posición hacia abajo la pieza del campo actual, dentro del tablero.

\*@pre Antes de hacer el movimiento, comprobará que este se puede hacer.

\*@post Si no se puede realizar este movimiento, las posiciones que ocupa la pieza

\* actual serán marcadas dentro de \_data con el color de esta pieza.

\*/

void **move\_down**()

/\*\*

\*@brief Mueve una posición hacia la izquierda la pieza del campo actual, dentro del tablero.

\*@pre Antes de hacer el movimiento, comprobará que este se puede hacer.

\*/

void **move\_left**()

/\*\*

\*@brief Mueve una posición hacia la derecha la pieza del campo actual, dentro del tablero.

\*@pre Antes de hacer el movimiento, comprobará que este se puede hacer.

\*/

void **move\_right**()

/\*\*

\*@brief Resetea la posición de la pieza, es decir, la mueve a la posición donde aparece

\*/

void **reset\_pieza**()

/\*\*

\*@brief Borra una fila entera, se usará cuando esta esté llena.

\* @param num\_row Número de la fila a borrar

\*/

void **erase\_row** (int num\_row)

/\*\*

\*@brief comprueba si una fila está completa (devuelve 1 si sí lo está y 0 si no lo está).

\* @param num\_row Número de la fila a comprobar

\* @return *true* si está completa o *false* si no lo está

\*/

bool **check\_row** (int num\_row)

/\*\*

\*@brief Comprueba si las fichas han llegado a lo alto del tablero, de manera que

\* terminaría el juego

\* @return *true* si ha terminado o *false* si no lo ha hecho

\*/

bool **check\_finish**(): comprueba si las fichas han llegado a lo alto del tablero, de manera que terminaría el juego.

}; // class Tablero

**class Marcador {**

private:

/\*\*

\* @brief Campo de tipo string que guarda el título de la partida.

\* @pre Como valor por defecto tendrá “Tetris”.

\*/

string **title**

/\*\*

\* @brief Campo de tipo int que guarda el nivel actual.

\*/

int **level**

/\*\*

\* @brief Campo de tipo int que guarda el número de filas que hemos completado.

\*/

int **rows**

/\*\*

\* @brief Campo de tipo int que guarda el número de piezas totales insertadas.

\*/

int **piezas**

/\*\*

\* @brief Campo de tipo bool que guarda *true* si todavía se sigue jugando o *false* si ya   
\* se ha perdido.

\*/

bool **state**

public:

// Constructores

/\*\*

\*@brief Crea una imagen por pantalla vacía, con los siguientes valores

\* predeterminados.

\* post: tittle=”Tetris” ; level=0; rows=0; state=*true*

\*/

**Marcador** ()

/\*\*

\*@brief Asigna todos los valores por defecto menos al campo *tittle*, que se le asignará

\* el recibido como parámetro.

\*@param t Título del juego

\* post: tittle=t ; level=0; rows=0; state=*true*

\*/

**Marcador** (string t)

// Métodos

/\*\*

\* @brief Cambia el estado del juego según un booleano: jugando (true) o finalizado

\* (false)

\*/

void **set\_state** ( bool estado )

/\*\*

\*@brief Nos informa del estado del juego

\*@return El estado de la partida, jugando (*true*) o finalizado (*false*)

\*/

bool **get\_state**()

/\*\*

\*@brief Cambia el número de líneas completadas

\*@param n Número de líneas completas

\*@pre n >= 0

\*/

void **set\_rows** (int n)

/\*\*

\*@brief Nos informa del número de líneas completas por el jugador

\*/

int **get\_rows**()

/\*\*

\*@brief Cambia el número de piezas insertadas en el acumulador

\*@param n Número de piezas

\*@pre n >= 0

\*/

void **set\_piezas** (int n)

/\*\*

\*@brief Nos informa del número de piezas insertadas en el acumulador

\*@return El número de piezas

\*/

int **get\_piezas**()

/\*\*

\*@brief Cambia el nivel del juego

\*@param n Nivel del juego

\*@pre n >= 0

\*/

void **set\_level** (int n)

/\*\*

\*@brief Informa del nivel del juego

\*@return El nivel de la partida

\*/

int **get\_level**()

/\*\*

\*@brief Pausar momentáneamente el juego, para poder ser resumido a continuación

\*/

void **pause()**

/\*\*

\*@brief Resume el juego una vez que ha sido pausado

\*/

void **resume()**

**};** // class Marcador

**class Cola {**

private:

/\*\*

\* @brief Número de piezas que aparecerán en la cola

\* @pre Por defecto, este será 4.

\*/

int **num\_piezas**

/\*\*

\* @brief Cola que contiene las piezas que próximamente saldrán al tablero

\* @post \_data.size()=num\_piezas siempre.

\*/

queue<Pieza> **\_data**

public:

// Constructores

/\*\*

\*@brief Constructor sin parámetros, crea un elemento tipo *Cola* con los valores por

\* defecto

\* @post num\_piezas=4; \_data con 4 piezas aleatorias.

\*/

**Cola** ()

/\*\*

\*@brief Constructor con parámetros. Rellena con n piezas aleatorias

\*@param n Número de piezas a mostrar

\*@pre 0<=num\_piezas

\*/

**Cola** (int n)

/\*\*

\*@brief Constructor con parámetros

\*@param queue\_piezas Cola que contiene las piezas

\*@post num\_piezas=queue\_piezas.size()

\*/

**Cola** (queue<Pieza> queue\_piezas)

// Métodos

/\*\*

\*@brief Obtenemos la siguiente pieza que saldrá al tablero

\*@return elemento tipo Pieza

\*/

Pieza **get\_next**()

/\*\*

\*@brief Obtenemos la pieza de la posición i-ésima

\*@param i Índice de la pieza que queremos obtener

\*@pre 0 <= i < num\_piezas

\*@return Elemento de tipo Pieza, en particular el de la posición i del vector

\*/

void **get\_pieza**(int i)

/\*\*

\*@brief Añadimos una pieza al final de la cola

\*@param p Pieza a añadir

\*@pre La cola debe tener al menos un elemento menos de *num\_piezas*

\*/

void **add\_pieza**(Pieza p)

/\*\*

\*@brief Elimina la pieza que está al principio de la cola

\*/

void **delete \_pieza**()

**};**

Para que las imágenes que generemos puedan tener color, crearé un tipo de dato *struct* muy básico pero en el que podamos guardar el código *RGB* de un color.

**struct Color {**

byte **red;**

byte **green;**

byte **blue;**

**};**

**class Image {**

private:

/\*\*

\* @brief Matriz de tipo *Color* que guardará cada pixel a color de la imagen

\*/

Color **\_data**[rows][columns]

/\*\*

\* @brief Número de filas de la imagen

\* @post 0<=rows.

\*/

int **rows**

/\*\*

\* @brief Número de columnas de la imagen

\* @post 0<=columns.

\*/

int **columns**

public:

// Constructores

/\*\*

\*@brief Se crea una matriz vacía

\*/

**Image** ()

/\*\*

\*@brief Se crea una imagen vacía con los parámetros dados

\*@param f Filas de la matriz

\*@param c Columnas de la matriz

\*@pre f >0

\*@pre c>0

\*/

**Image** (int f, int c)

// Métodos

/\*\*

\*@brief Crea una imagen a partir de los datos de un fichero

\*@param file\_path Nombre del archivo que contiene los datos

\*@return La imagen generada a partir de los datos proporcionados

\*/

Image **OpenImage**(const char\* file\_path)

/\*\*

\*@brief Muestra la imagen por pantalla

\*/

void **ShowImage**()

**};** // class Image

/\*\*

\*@brief Sobrecarga el operador <<. Imprime por pantalla la imagen

\*/

ostream& **operator<<** (ostream& flujo, const Imagen& img)

/\*\*

\*@brief Sobrecarga el operador >>. Permite cargar la información al disco

\*/

istream& **operator>>**(istream& flujo, const Imagen& img)

Podría hacerse una clase **Juego** que agrupara cada uno de los elementos que hemos creado para generar el juego. Otra opción es directamente generar el juego en un main, usando un objeto de cada clase generada.