

Designing a Block Puzzle in Java Swing

Final project — Programming II

Universitat de les Illes Balears

Authors:

Andreu Massanet Felix
Marc Morlà Maimó

2023 - 2024

Contents

1	Introduction	2
2	General structure of the project	2
3	Design	3
3.1	Class Tetris	3
3.1.1	inicializacion()	3
3.1.2	iniciarPartida()	3
3.1.3	nuevaPartida()	4
3.1.4	configuracion()	4
3.1.5	historial()	4
3.1.6	informacion()	4
3.1.7	salir()	4
3.1.8	Events	4
3.1.9	Class Display	5
3.2	Class Tablero	5
3.2.1	Constructor	5
3.2.2	Paint	5
3.2.3	Others	5
3.3	Class Pieza	6
3.3.1	Constructor	6
3.3.2	generarFormaAleatoria()	6
3.3.3	Others	6
3.4	Class Casilla	6
3.4.1	Constructor	6
3.4.2	Others	6
3.5	Class Partida	6
3.5.1	Attributes	6
3.5.2	Constructors	6
3.5.3	Others	7
3.6	Class PartidaObjetoFicherosLectura	7
3.6.1	Constructor	7
3.6.2	lectura()	7
3.6.3	cierre()	7
3.7	Class PartidaObjetoFicherosEscritura	7
3.7.1	Constructors	7
3.7.2	escritura()	7
3.7.3	cierre()	7
3.8	Other classes	7
3.8.1	AdicionObjectOutputStream	7
3.8.2	LTGraficos	8
4	Graphical design	8
5	Conclusions	10

1 Introduction

For the Programming II course we were asked to build **Tetris UIB**, a desktop application written in Java. The emphasis of the exercise was on designing and implementing a graphical user interface, so the game was the vehicle rather than the goal: what the project exercises is object-oriented design, event handling and reading and writing objects to a file.

The application had to offer four things from its menu: starting a new game, configuring several aspects of it, consulting the history of matches already played, and showing instructions on how to play. It also had to be intuitive to use, with clearly identifiable buttons and icons.

It is worth stating plainly what the game turned out to be, because the name is misleading. Nothing falls under gravity, the pieces are not the seven tetrominoes, and a line clears along columns as well as along rows. The player drags an arbitrary shape from the tray and drops it wherever it fits. That places the result closer to a drag-and-drop block puzzle than to Tetris proper, and the design that follows only makes sense read that way.

This report describes how the application was built, class by class, then the layout of the interface, and finally what the exercise left us with.

2 General structure of the project

The project is a NetBeans Ant project. All source files live in the package **entregafinal**, next to two resource folders: **imagenes**, holding the cell textures, the backdrop and the title image, and **iconos**, holding the menu icons.

The classes are the following:

- **Tetris**: the entry point. Builds the whole interface, wires up every event, keeps the score and the clock, and decides when a match ends.
- **Tablero**: the board. Draws itself, records which cells are occupied, and detects completed rows and columns.
- **Pieza**: the piece in play, a random shape on a 3x3 grid.
- **Casilla**: a single cell, used both by the board and by the pieces.
- **Partida**: a finished match, ready to be serialised.
- **PartidaObjetoFicherosLectura** and **PartidaObjetoFicherosEscritura**: reading and writing matches to file.
- **AdicionObjectOutputStream**: an output stream that writes no header, which is what allows matches to be appended to an existing file.
- **LTGraficos**: a dialog for entering several fields at once.

3 Design

3.1 Class Tetris

3.1.1 inicializacion()

The main entry point of the application, and where the whole interface is set up.

UIManager establishes the look of the interface, defining the background colour and the fonts for the different components, such as the panels and the dialog boxes.

The components are then declared and configured: panels, labels, buttons, progress bars, menus and menu items. Each one is instantiated and given the properties it needs, such as its text, its background colour and the events attached to it.

Actions are assigned to the buttons and menu items through **ActionListener**. This defines what happens when the user clicks on start a game, configuration, history, information or exit. The clock handler is responsible for writing the match to file once it has finished, and for showing the corresponding message.

Events are also defined for the actions of the game itself, such as rotating a piece or asking for a new shape, together with the timer that governs the length of a match and what happens when the time runs out: showing the final score and saving it.

The components are arranged inside panels using several layouts, namely **BorderLayout**, **GridLayout** and **FlowLayout**. A card layout over **panelInterfaces** is what allows the application to swap between views, that is between the game itself, the history of matches and the information panel. Finally the main window (**JFrame**) is configured: its size, whether it can be resized, and what closing it does.

3.1.2 iniciarPartida()

- The score is reset to zero.
- The game state is set to active (**jugando = true**).
- The clock and the progress bar tied to the playing time are reset.
- The board is reset, ready for a new match.
- The points shown in the interface are updated.
- The player's name is written into the corresponding field.
- The board is added to **panelTablero**.
- The clock starts counting the playing time.
- The game panel (**panelInterfaces**) is shown in the window.
- Certain cells of the board are marked as invalid, blocking the areas that cannot be played on.
- Mouse listeners are attached to the window, so that the board can be interacted with during play. Messages agree in number, singular or plural, with what they report.

3.1.3 nuevaPartida()

Checks whether a match is already in progress. If it is not, the player's name is requested through a dialog box (`LTGraficos`) and a new match is started by calling `iniciarPartida()`. If a match is already in progress, the player is told to finish it before starting another.

3.1.4 configuracion()

Checks whether a match is already in progress. If it is not, a dialog box opens offering to configure either the scoring or the length of a match.

Depending on the option chosen, the points awarded for specific actions or the duration of a match can be set. Any field left untouched keeps its previous value, and so does the cell texture if the image named does not exist. Every setting is shown with its current value in the prompt, in the form “[1 point]”, and is only applied if what was typed is a valid number. Here too the messages agree in number with what they report.

If a match is already in progress, the player is told to finish it before reaching the configuration.

3.1.5 historial()

Checks whether a match is already in progress. If it is not, the history of matches is read from file and displayed in a text area in the interface, again with messages agreeing in number.

If a match is already in progress, the player is told to finish it before reaching the history.

3.1.6 informacion()

Checks whether a match is already in progress. If it is not, the information panel is shown in the interface. If a match is already in progress, the player is told to finish it first.

3.1.7 salir()

Calls `System.exit(0)` to close the application with exit code 0.

3.1.8 Events

Beyond the buttons already described, the application responds to the mouse.

mouseDragged(): if the variable `DESPLAZAMIENTO` is true, updates the coordinates of the piece on the board so that it follows the cursor, and repaints the window.

mousePressed(): if the click falls inside the area of the piece, sets `DESPLAZAMIENTO` to true, which is what lets the piece follow the cursor. If the piece is in its insertion position, its coordinates are reset and the window is repainted.

mouseReleased(): if `DESPLAZAMIENTO` is true, a piece was being moved. This method works out the coordinates at which the mouse was released and checks whether that is a valid position on the board. If it is, the piece is placed, and the score is updated according to the rows and columns that have been completed. It then switches to a new piece and repaints the window.

3.1.9 Class Display

An inner class of `Tetris`, used to draw the UIB image.

3.2 Class Tablero

3.2.1 Constructor

A nested loop creates every cell of the board. Each cell is placed at a specific position derived from the constants that have been defined, namely the cell size (`DIMCASILLA`) and the offset (`DESFACE`). A matrix of cells (`casillas`) represents the board, each element being an instance of `Casilla`, and the x and y coordinates of each cell are updated on every iteration so that the cells line up correctly.

A new instance of `Pieza` is created to represent the piece currently in play, and the image of the piece is passed as an argument to its constructor.

3.2.2 Paint

A nested loop walks every cell of the board.

Each empty cell is drawn with the `drawImage` method of `Graphics2D`, painting the image of an unoccupied cell in the corresponding position. The horizontal and vertical lines of the board are drawn with `drawLine`.

A further loop draws the invalid cells, that is the ones that pieces cannot occupy, as black rectangles. A backdrop image is then drawn, again with `drawImage`.

Finally, each cell is checked. If it is occupied, the image of the occupied cell is drawn. If it is not, and the piece currently in play has a filled cell at that position, the image of the piece is drawn at the cursor position, relative to the position of the cell on the board.

3.2.3 Others

- Getters and setters, to read and set values.
- `piezaTieneCosa()`: reports the layout of the current piece.
- `cambioPieza()`: swaps the piece.
- `rotarPieza()`: rotates the piece.
- `filaCompleta()`: walks a row to see whether it is complete.
- `eliminaFila()`: clears a row.
- `columnaCompleta()`: walks a column to see whether it is complete.
- `eliminaColumna()`: clears a column.
- `reiniciar()`: frees every cell and switches to a new piece.

Both `filaCompleta()` and `columnaCompleta()` scan only the playable range, never the invalid border, so the black cells around the edge can never be counted towards a completed line.

3.3 Class Pieza

3.3.1 Constructor

Assigns the image given as a parameter to the cells and generates a random shape.

3.3.2 generarFormaAleatoria()

Uses an array of booleans to assign random shapes. Every cell of the 3x3 grid is decided independently, and the whole grid is generated again if it comes out empty, so that a piece always has at least one filled cell.

3.3.3 Others

- Getters and setters, to read and set values.
- `rotar()`: transposes the matrix of cells.

The rotation is written into a second matrix and only then copied back over the first. Rotating in place would overwrite cells that the operation still had to read.

3.4 Class Casilla

3.4.1 Constructor

Marks the cell as free.

3.4.2 Others

Getters and setters, to read and set values.

3.5 Class Partida

3.5.1 Attributes

- `nombre`: the name of the player.
- `puntuacion`: the score obtained in the match.
- `tiempo`: the length of the match in seconds.
- `FechaYHora`: the date and time at which the match was played.

3.5.2 Constructors

- `Partida()`: constructor without parameters.
- `Partida(String nombre, int puntuacion, int tiempo, String FechaYHora)`: constructor with parameters, initialising every attribute of the match.

3.5.3 Others

- Getters and setters, to read and set values.
- `obtenerFechaYHoraPartida()`: uses the `Calendar` class to obtain the current date and time and format it into a string.
- `toString()`: returns a string representation of the match, including the date and time, the playing time, the name of the player and the score.

The class implements `Serializable`, which is what allows a match to be written to file as an object rather than as text.

3.6 Class `PartidaObjetoFicherosLectura`

3.6.1 Constructor

Opens the link to the file.

3.6.2 `lectura()`

Reads a `Partida` object from the file.

3.6.3 `cierre()`

Closes the link to the file.

3.7 Class `PartidaObjetoFicherosEscritura`

3.7.1 Constructors

Open the link to the file. There are two: one that writes a new file, and one taking a second boolean argument that opens an existing file in append mode.

3.7.2 `escritura()`

Writes a `Partida` object to the file.

3.7.3 `cierre()`

Closes the link to the file.

3.8 Other classes

3.8.1 `AdicionObjectOutputStream`

Adds `Partida` objects without overwriting what the file already holds.

An `ObjectOutputStream` writes a header as soon as it is opened. Opening an existing file in append mode and writing normally would therefore leave a second header in the middle of the file, and a reader would stop dead when it reached it. This class overrides `writeStreamHeader()` so that it does nothing, which is what makes a match appendable to a file that already holds matches.

3.8.2 LTGraficos

Creates windows for entering data.

4 Graphical design

The interface is built from nested panels. The figures below name each region with the identifier it carries in the code, so that the diagrams and the source can be read against each other.

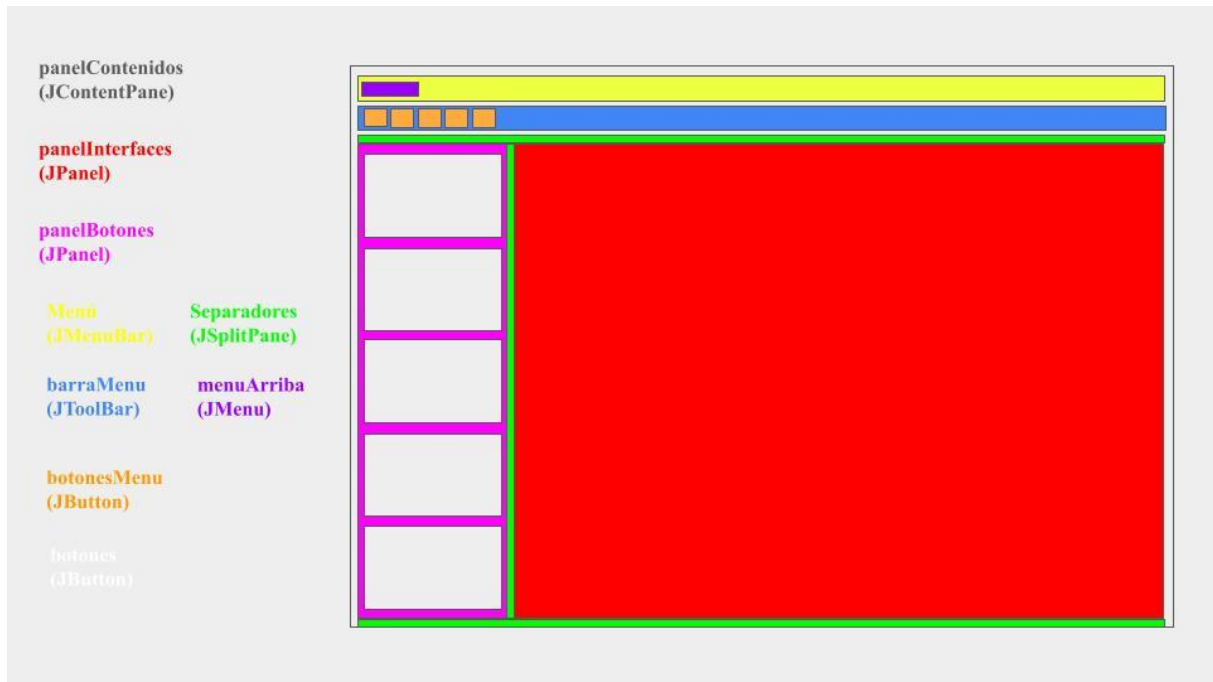


Figure 1: Layout of the main menu.

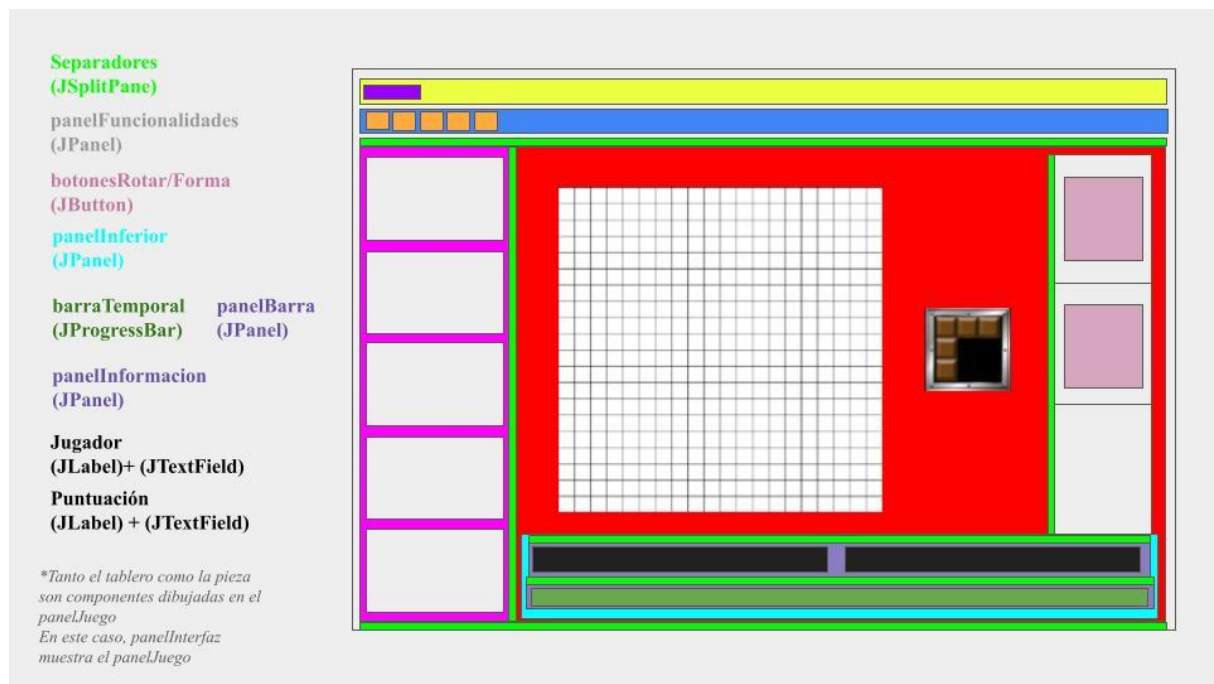


Figure 2: Layout of the game interface. Both the board and the piece are components drawn inside `panelJuego`, which in this case is what `panelInterfaz` is showing.

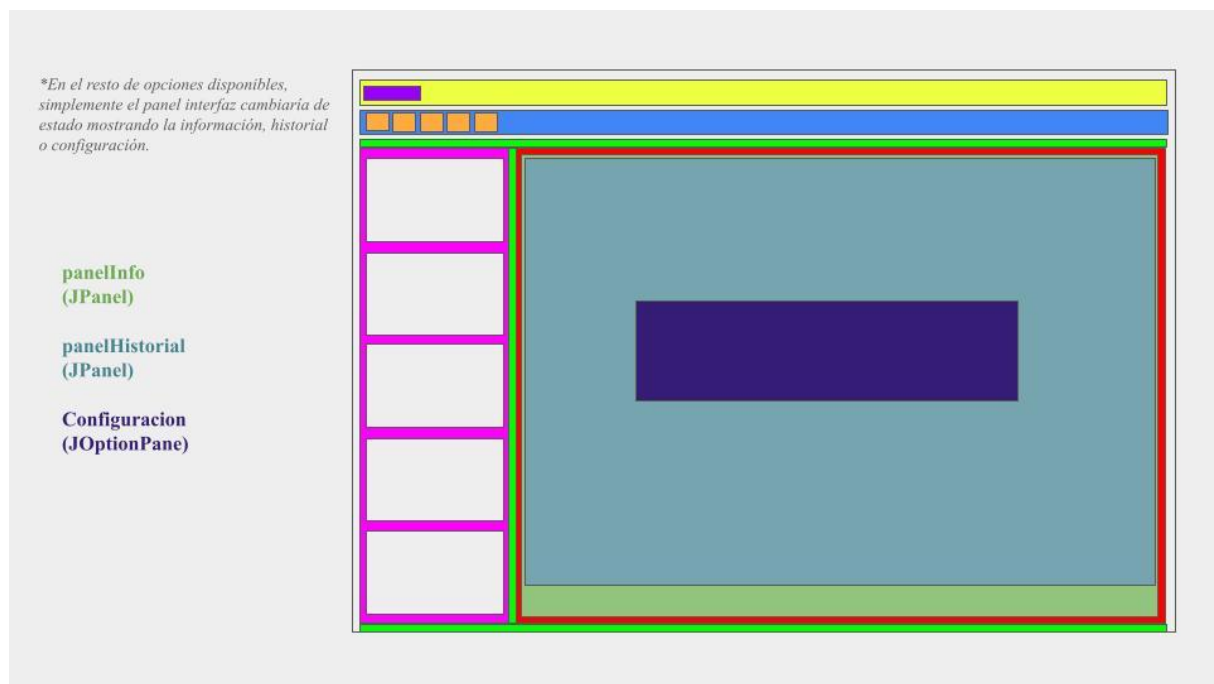


Figure 3: Layout of the remaining interfaces. For the other options available, the interface panel simply changes state to show the information, the history or the configuration.

5 Conclusions

Building this application consolidated the fundamentals of object-oriented programming and of graphical interfaces in Java. The result meets the requirements it was given and is reasonably pleasant to use, which was the point of the exercise.

The parts that took the most work were not the ones we expected. Generating and managing the pieces was straightforward; deciding whether a shape could be dropped where the cursor had left it, and translating between window coordinates and board coordinates, was where most of the effort went. Serialising matches was the other, and the header problem described above is a good example of a difficulty that only appears once the file already has something in it.

Beyond the specific techniques, the exercise made the case for organising and documenting code as you go. A structured, documented project is easier to come back to and considerably easier to work on with someone else, which on a project split between two people is not a small thing.