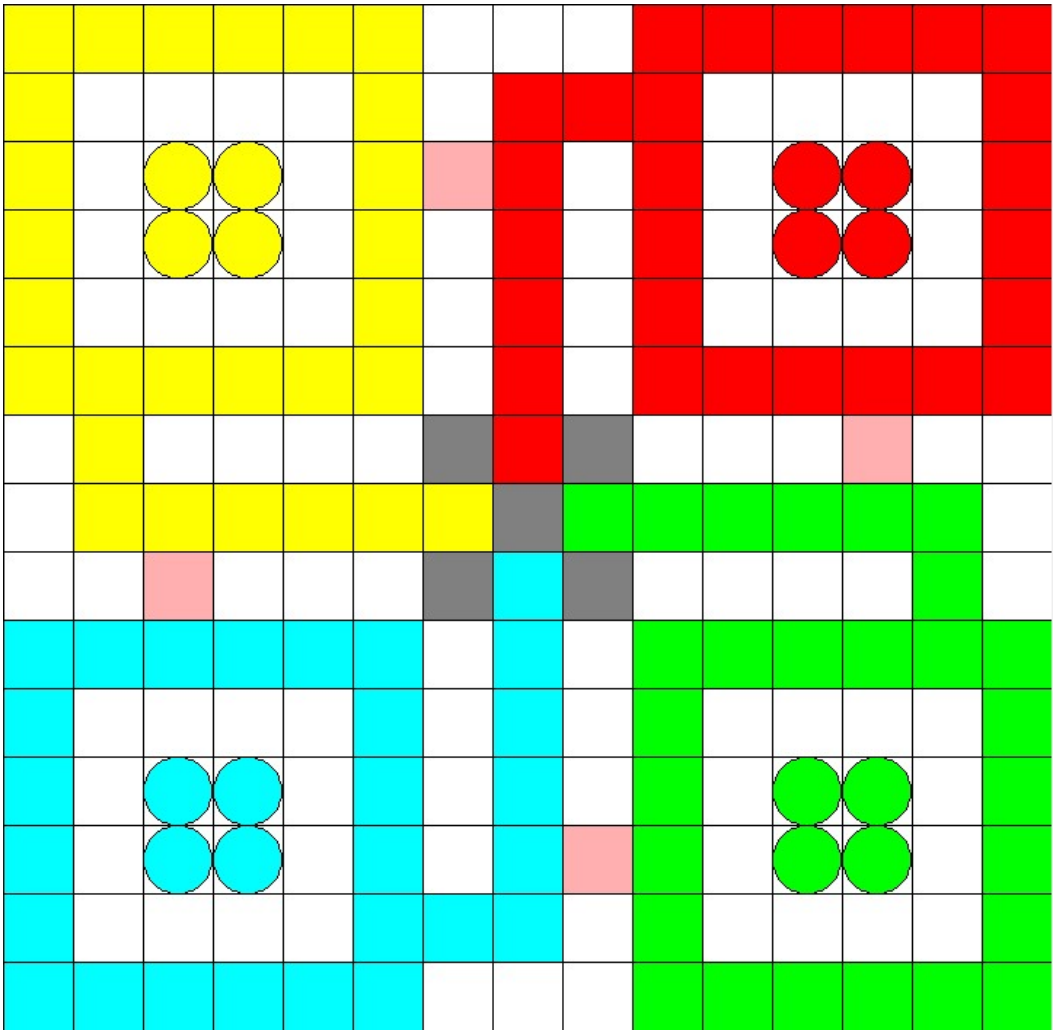
Aubry Youri LP2A

Project report :

Ludo Game



26/04/2021

Introduction

Ludo King is a game in which four players fight to make their four pieces travel through a cross-shaped board. Pieces can be captured or can form blocks which prevents other players pieces to go through them. Pieces move according to a dice roll, and players can choose which one of their pieces they want to move.

There also are a few side rules of the game :

- Some of the cases of the board are safe, which means pieces can't be captured on them.
- Players need to roll a 6 to be able to put a piece on their starting case.
- a 6, an opponent piece captured or a piece reaching its final case means the player can perform another turn.

Now that you've got the basic concepts of the game, let's see how I represented each one of them in my project. Starting with the board, we will then see how I made pieces move, how all the elements come together in the project and the design choices I made to get to the final result.

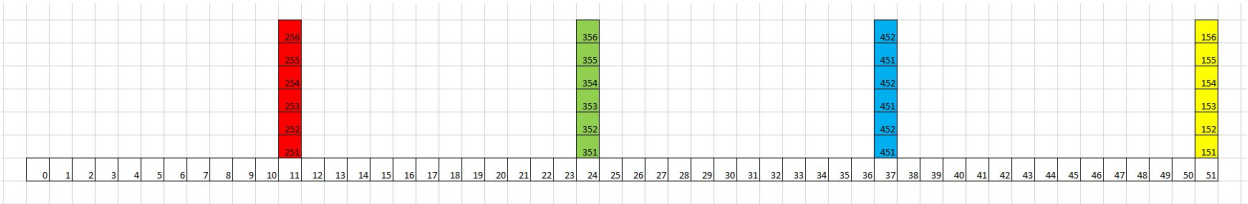
Representation of the board

A) Map

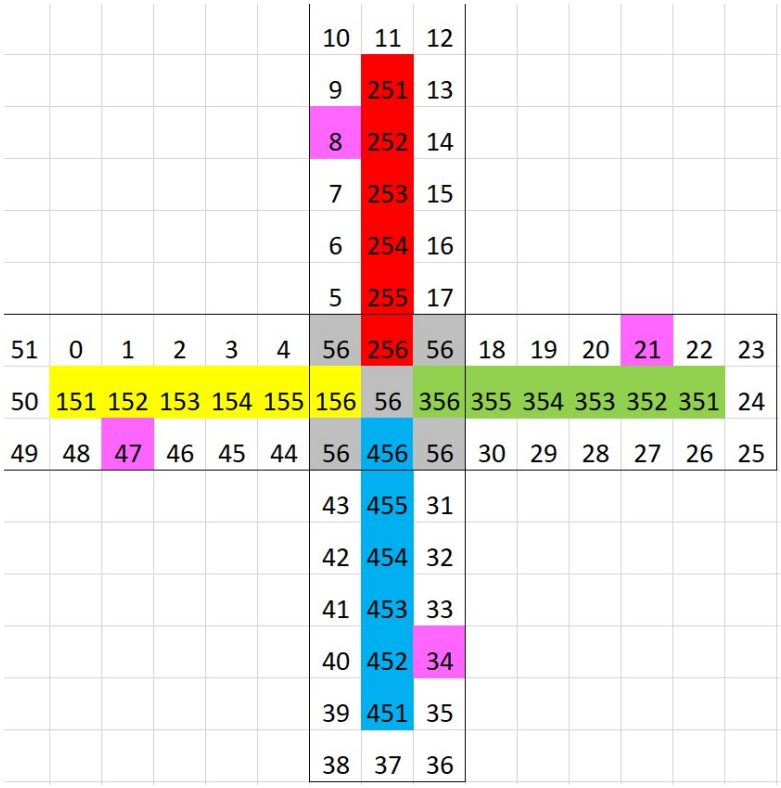
The first step into the project was to represent the board. I decided to represent it as a linear path which loops around once it's over. This path has multiple entries and exits which only can be accessed by one player. In order to be able to manipulate easily the board, I mapped each case of the 15*15 grid, giving them indexes.

The linear path is mapped from 0 to 51, and the exits are mapped from 51 to 56 to which is added 100 multiplied the index of each player. The remaining cases of the grid are mapped according to their color.

Here is what the linear representation of the board looks like :



And here is the board mapping :



This way, every case can be accessed from its index. Moreover, using a modulo, the path shared by all the players can be represented in the same way using an index representing each player (0 for yellow, 1 for red, 2 for green and 3 for blue.

Starting case : 13*playerIndex

Case : (index + 13*(4 – playerIndex)%4)%52

Example : red player, case 44 : (44+13*(4-1)%4)%52=31. Case 44 is the 31st case for red player.

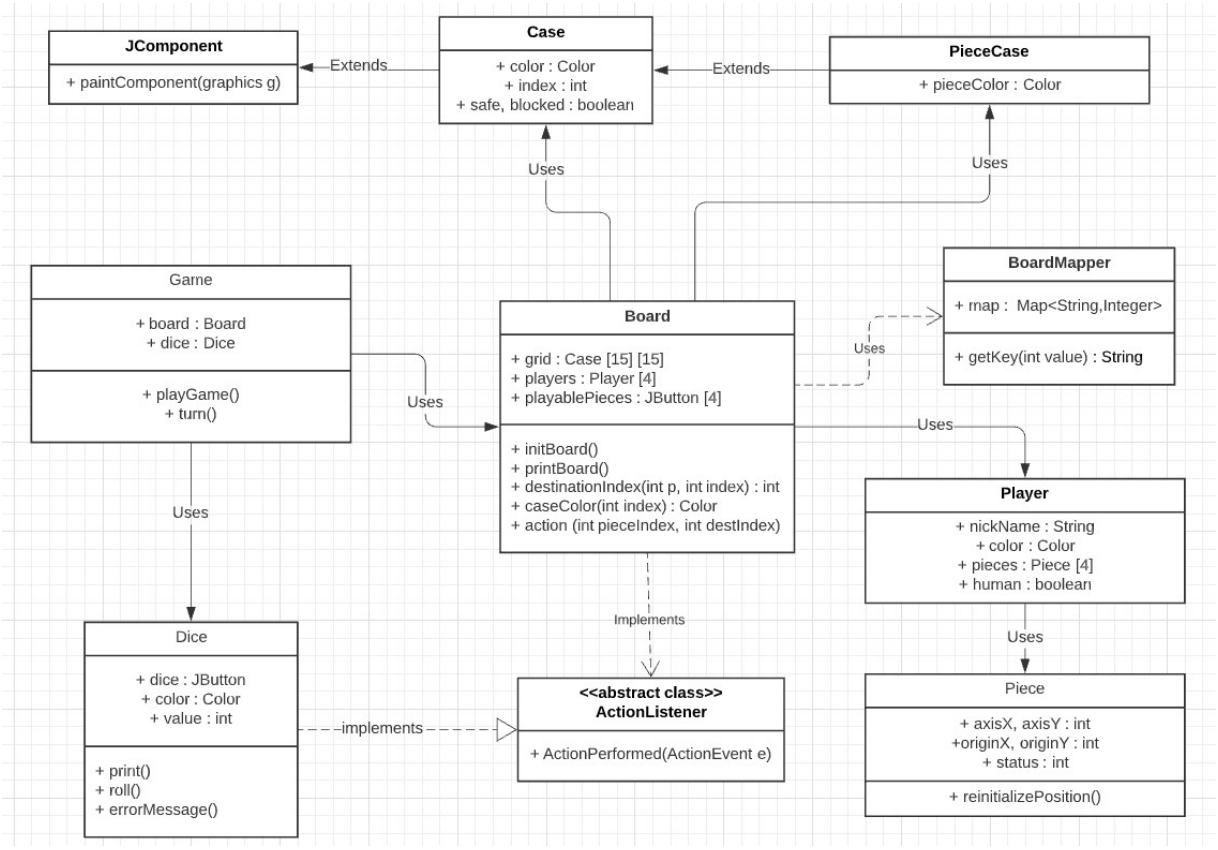
The map is located in the “BoardMapper” class.

B) Cases

As the board is represented as a 15*15 grid of cases, the class “Case” needed to be created. In order to put the board on a frame, cases were represented as graphical components which could be individually added to the board. Each case contains an integer representing its index, a color and two Boolean representing whether the case is a safe, a blocked or a normal case.

To represent cases containing a piece, a class extending “Case” was created. It contains another color, the piece’s color, and the drawing method is overridden in order to draw a circle in the middle of the square.

UML Diagram and explanation



Over this project, one technique was mainly used : composition. This is represented on the diagram by arrows labeled “Uses”. Indeed, each time one you see one of these arrows on the diagram, it means the class at the origin of the arrow contains as data at least one object of the class on the other end of the arrow.

For the graphical interface, I used inheritance and implementation of an abstract class. Inheritance was used to represent both types of cases. In both cases, the method “paintComponent” was overridden. First type draws a square, and second adds a circle with a different color.

To be able to use buttons, I decided to make two classes as implementations of the abstract class “ActionListener”. This way, “Dice” and “Board” contain buttons users can interact with and which, when pressed, modify the value of the data of the associated object.

Pieces movement

In order to move the pieces, two options were considered. The first one consists in looking around the current position where the case containing the next index is and move to its position, and repeat this process as many times as the dice roll. The second option consists in calculating the index of the final case and directly move to its position.

In order to make the process faster, the second option was kept. But the problem with this method is that it doesn't take into account blocked cases. To patch this problem, I had to add to create a method which get through the cases and checks if the path is blocked. Therefore, as it uses both options, the model isn't as fast as it could.

In addition to the board, classes needed to be created to define players and pieces. A piece contains its coordinates as two integers, its status and the coordinates of its origin. These are used to reinitialize its position once it has been captured.

A player contains 4 pieces, a nickname, a color and a Boolean indicating if the player is a user or a computer. This would have obviously been used to play against an AI. Another solution to do so would have been to extend the "Player" class into a "PlayerAI" class and to put all the functions dealing with piece movements into these classes. But the design choices I made kind of forced my hand into putting buttons into the "Board" class, therefore "Player" only contains as methods getters and setters.

How does a turn work?

To understand how the project works, let's break down a turn.

First things first, the board is initialized. If it hasn't been initialized before, all the cases are initialized to rather a "PieceCase" or a "Case" if a piece is located on the case. The color of the case comes from the index from the BoardMapper and the color of the piece is the color of the player it belongs to.

To optimize the project, instead of initializing the board this way every time, once the board has been initialized once, only the cases which were modified by the previous move are reinitialized. These cases are : landing and starting case of the moving piece, origin case of the taken pieces, cases containing buttons (moveable pieces).

Once the board is initialized, it is printed on the frame, and every case containing a moveable piece owned by the current player is replaced by a button. Buttons work this way :

- They first move the piece, by setting its coordinates to those get from its destination index using the BoardMapper.
- Then, it saves its starting and landing coordinates as cases which will be modified.
- Finally, a series of test is ran upon the case, and modifications are applied (unsafe cases are captured, ending cases update the status of the piece, the player plays again in certain occasions, etc…).

Once the player presses one of the buttons, its turn is over, the playing player switches and it starts all over again.

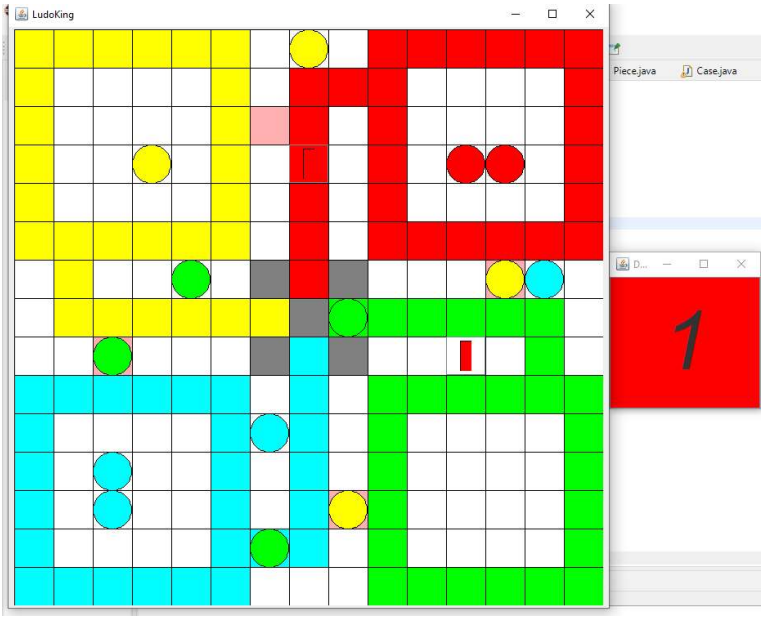
In case none of the current player's pieces can move, an error message is printed on the dice, and he needs to press this button to end its turn.

Design choices

I decided at the beginning of the project to create the game such that it could be played using only the mouse. I felt like playing this game by entering coordinates and numbers on a terminal wouldn't quite match its concept. Therefore, even though it wasn't supposed to be the priority of the project, I dedicated a lot of time to the graphical interface. The other reason I did so is because I never actually created an interface on Java and I wanted to learn how it worked.

This choice had a few consequences on the final layout of the project. two frames are used on the game : the board and the dice. Besides printing the result of the dice roll, the dice frame has two other purposes : it is a button which allows us to interact with the game when none of the pieces can be played. Moreover, it displays the color of the player which is playing.

Here is what the user interface looks like :



The other frame only represents the board, which is a 15*15 grid of colored cases. Cases containing pieces the playing player can play are replaced by buttons which activate the associated piece.

This way, my project is very close from the mobile game LudoKing. The only difference is that I made the dice roll automatically instead of making it roll on the press of a button. I consider it as an improvement from the original game, because I found kind of annoying the obligation to click each turn at the same spot to do the exact same action.

To sum up my design choices, I decided to prioritize two aspects in the project : the gameplay and the fact of learning new things.

Improvements

The board printing isn’t optimized at all. Indeed, each move requires the program to print every case of the board. The correct way would have been to print pieces on a defined board. But as I didn’t manage to modify a specific case on the grid layout of a JFrame and that I didn't find another way to print the board from the classes I created, I kept it this way.

I also could have added a menu, in which users could choose their nickname and their color.

I didn’t have time to develop an AI, but I had several ideas on how to do so, and how to set different level of difficulty. I listed all the different things that could influence a move :

take another piece, develop a new piece, move to the final case, move to a safe zone, run away from a close piece, attack a piece, block the path, move first piece, move last piece, move to an “attackable” zone, get out of a safe zone.

These types of move could have been represented by methods returning Booleans. Then, every method would have been applied to each piece in a specified order, ant the first one which returns a true value determines the piece to move. The order in which methods are applied represent the level of difficulty (example : an easy bot wouldn’t mind getting out of a safe zone, whereas a difficult bot would take your pieces whenever it can). These difficulty levels could have been a part of an adaptative difficulty level, which gets harder and harder the more you’re winning.

I had kind of a hard time trying to give to buttons the appearance I wanted them to look like, so all I managed to do was to give it the colors of the case and the piece it represents. I could have taken more time to deal with this issue, but as the graphical interface wasn’t the main focus of the project, I decided to leave it as it was.

On the graphical interface, there is no way to print a case containing multiple pieces differently from a case containing a single one. If one of the players which have a piece on this kind of case is playing, its piece appears on the case, otherwise the first to come up in the order yellow-red-green-blue appears.

Conclusion

As you might have guessed reading this report, my priorities went a little bit away from what we were told in the beginning of the project. I focus a lot more than we were asked to on the graphical interface, and as I didn't have time to finish the AI players, I guess it was a bad choice.

But even though it's not finished, I'm really proud of what this project turned out to be in the end. The player versus player mode works perfectly, and the graphical interface makes the game really pleasant to play. I don't regret this choice as I learnt a lot about a topic I never experimented with before.

I hope I made myself clear through this report, and you understood how I realized this project. Thank you for reading.