Exercise snakes and ladders

> *Snakes and ladders* is a board game for two or more players originated in ancient India. It is played on a game board with numbered squares. A number of ladders and snakes are pictured on the board, each connecting two specific board squares. The object of the game is to navigate one's game piece, according to die rolls, from the start (bottom square) to the finish (top square), helped by climbing ladders but hindered by falling down snakes.
> https://en.wikipedia.org/wiki/Snakes_and_ladders

In our version, additionally, if a player rolls the die and falls into a square already occupied by another player, he/she goes to square 1.


36 35 34 33 32 31
25 26 27 28 29 30
24 23 22 21 20 19
13 14 15 16 17 18
12 11 10 9 8 7
1 2 3 4 5 6

We have built a **simulator of games**.

Input

- number of players and their names
- number of squares
- pairs of from-to positions of snakes and ladders. [3,16], [5,7]... for ladders, [12,2], [17,4]... for snakes, in the board above

Output: the computer simulates the players playing the game until one of them wins

- number rolled by die
- who is and moves of current player
- current state of game
- who's the winner

There are 12 squares
ladder from 2 to 6
ladder from 7 to 9
snake from 11 to 5
Players are :
1. Ana
2. Blanca
3. Carles
4. Didac
Initial state :
Ana is at square 1
Blanca is at square 1
Carles is at square 1
Didac is at square 1

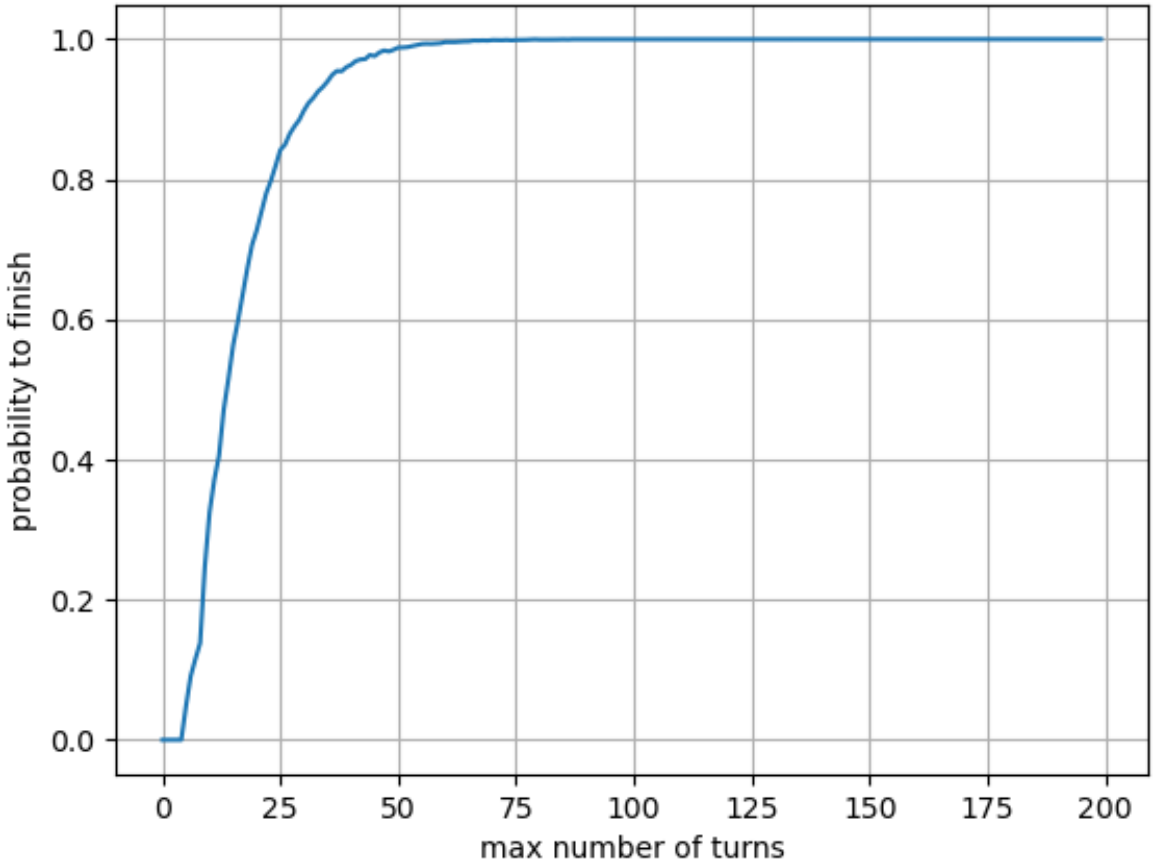
Current player is Ana and rolls 4
move from 1 to 5
land at 5
State :
Blanca is at square 1
Carles is at square 1
Didac is at square 1
Ana is at square 5

Current player is Blanca and rolls 4
move from 1 to 5
square 5 is occupied, go to square 1
State :
Carles is at square 1
Didac is at square 1
Ana is at square 5
Blanca is at square 1
:
:
:
Current player is Ana and rolls 4
move from 8 to 12
land at 12
State :
Blanca is at square 4
Carles is at square 9
Didac is at square 1
Ana is at square 12

Ana has won after 9 rounds

The simulator is useful on its own to calculate the frequency of a player wins after *at most* N turns , for $N = 1 \ldots 200$ = cumulative probability distribution of a game duration.

From it one can approximate the mean and variance of game duration and answer questions like, for a certain board, is the game too fast? too slow ? how many times, on average, a player goes back to square 1 ? ...



Approximated cumulative distribution for a small board with 12 squares, two ladders and one snake. **We are not going to compute it. Only simulate a game.**

Goals

- practice Java and UML by studing the source code and design of the finished application that simulates games
- because this is necessary to perform the two changes we ask you to design and code
- larger design than n-body exercise, with inheritance
- see how have we applied some Grasp patterns and object-oriented principles
- apply the *don't repeat yourself* principle

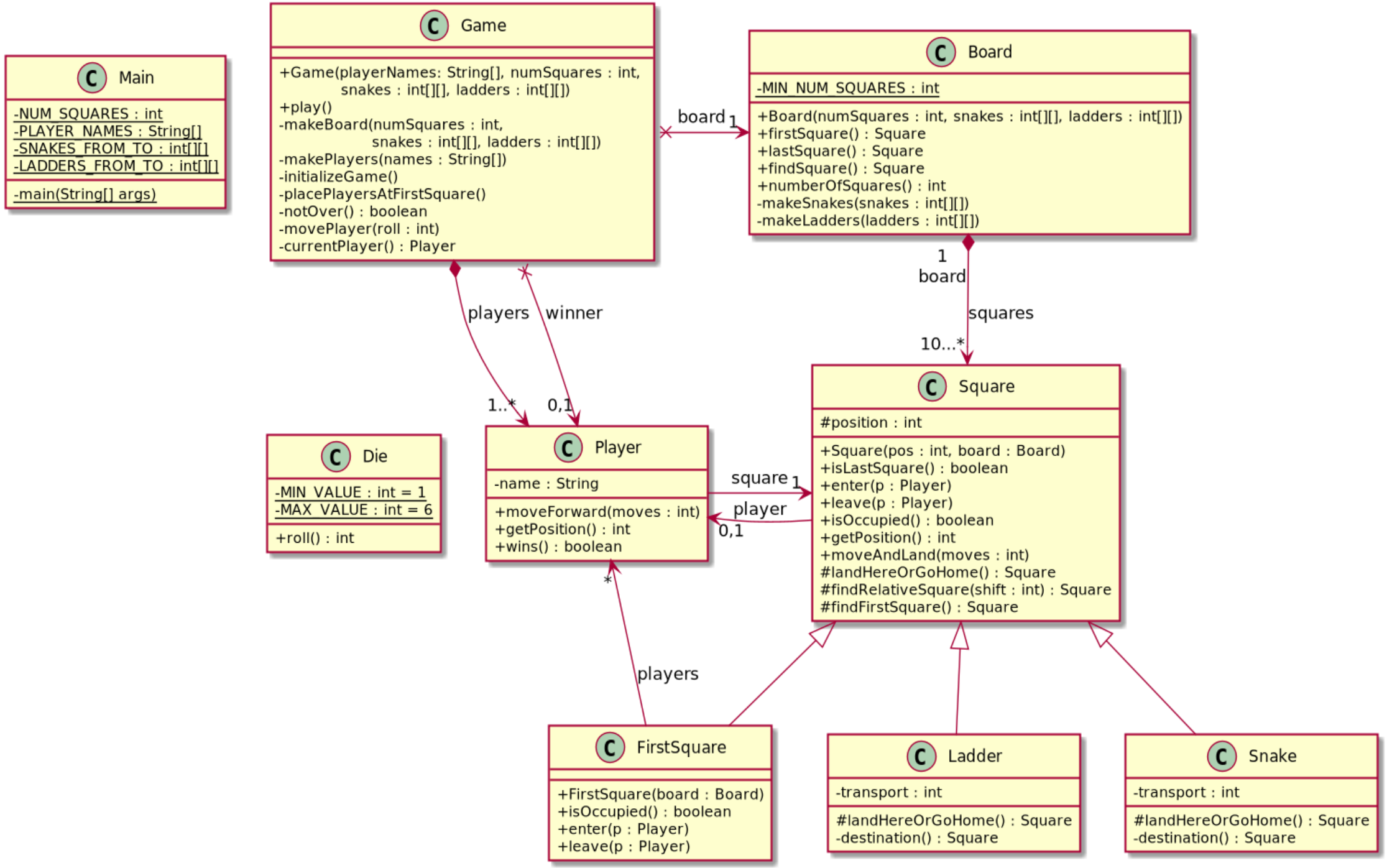
Design and code

Find it here

Study the design and explore the code:

- open file `Main.java` and follow the chain of method invocations
- move the cursor over `game.play` and Ctrl + 1st button, this will open `Game.java` in the editor and go to the implementation of `play()`
- you can go back with Alt + Shift + ←
- and forward with Alt + Shift + →

See how the different **responsibilities** of making the players, the board, the squares, initialize the game and play the game are assigned to the classes.


Main
-NUM_SQUARES : int
-PLAYER_NAMES : String[]
-SNAKES_FROM_TO : int[][]
-LADDERS_FROM_TO : int[][]
-main(String[] args)
Game
+Game(playerNames: String[], numSquares : int,
snakes : int[][], ladders : int[][])
+play()
-makeBoard(numSquares : int,
snakes : int[][], ladders : int[][])
-makePlayers(names : String[])
-initializeGame()
-placePlayersAtFirstSquare()
-notOver() : boolean
-movePlayer(roll : int)
-currentPlayer() : Player
board
1
Board
-MIN_NUM_SQUARES : int
+Board(numSquares : int, snakes : int[][], ladders : int[][])
+firstSquare() : Square
+lastSquare() : Square
+findSquare() : Square
+numberOfSquares() : int
-makeSnakes(snakes : int[][])
-makeLadders(ladders : int[][])
players
winner
1..*
0,1
1
board
squares
10...*
Die
-MIN_VALUE : int = 1
-MAX_VALUE : int = 6
+roll() : int
Player
-name : String
+moveForward(moves : int)
+getPosition() : int
+wins() : boolean
square
1
player
0,1
Square
#position : int
+Square(pos : int, board : Board)
+isLastSquare() : boolean
+enter(p : Player)
+leave(p : Player)
+isOccupied() : boolean
+getPosition() : int
+moveAndLand(moves : int)
#landHereOrGoHome() : Square
#findRelativeSquare(shift : int) : Square
#findFirstSquare() : Square
*
players
FirstSquare
+FirstSquare(board : Board)
+isOccupied() : boolean
+enter(p : Player)
+leave(p : Player)
Ladder
-transport : int
#landHereOrGoHome() : Square
-destination() : Square
Snake
-transport : int
#landHereOrGoHome() : Square
-destination() : Square

What to do : two changes

1. Apply the principle *don't repeat yourself*

- have a look at the code of classes `Snake` and `Ladder`
- also, at methods `Board.makeSnakes()` and `Board.makeLadders()`
- they are almost the same, the only difference between a snake and a ladder is the sign of `transport`
- replace classes `Snake` and `Ladder` by a new class `SnakeOrLadder`
- same with the two methods, write a new `Board.makeSnakesOrLadders(int[][] toFrom)` that does the same
- run and make sure the program works by checking the output messages
- update the plantUML design (don't try too hard to position well the classes)

2. New `Death` square

- add new type of square (ie. a class) `Death`, and put one or more death squares in the board, for instance at positions 4 and 8 of the small board with 12 squares
- if a player falls into a death square, the game is over for him/her, the others keep playing
- it may be the case that all players have fallen into the Death square, then the game is over

As before, update the UML design, and check the code is right by reading the printed messages that trace a game.

Steps for 2

1. Add new file `Death.java`, accept the constructor suggested by the editor
2. Override method `enter(Player p)` so that it just "kills" the player `p`, meaning sets it as "dead" $\Rightarrow$ new boolean attribute `dead` in `Player`, getter `public boolean isDead()` and setter `public void setDead(boolean d)`
3. Redefine end of game condition: game not over = no winner yet *and* there is still one player alive at least . Make a new method to check the second condition
4. In `main` create a new constant vector with the position of one or more death squares, `[4,8]` for instance, to be passed to `Board`
5. Add a `makeDeaths()` to `Board`
6. When the current player is dead, do not put it back in the list of players (in `Game`)

Typical errors you may come across

Exception in thread "main" java.lang.NullPointerException

you are trying to call a method of, or access, an object that is null = no object

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

you are trying to access a position of a vector or array out of its limits, for instance a negative index or equal or larger than its size

Array and vector indices are 0 to num. elements - 1

Deliverables

A .zip file containing:

1. text file named `autors.txt` with name and NIU of authors responsible of this work
2. a plain text `explicacio.txt` briefly explaining how have you done items 1 and/or 2 and the problems you have had
3. a PNG image of the new design (exported from plantUML)
4. a text file with the output of your program, showing it works well after the addition of two death squares, if you have done it
5. a zip file with the project, including the new design in plantUML