Design Document Chutes And Ladders

TABLE OF CONTENT

Description of the solution

- Description of the package , main Interfaces, classes and Exceptions

package com.example.snakeAndLadder

App : The main class instantiates a Game, passing to the constructor the necessary parameters (players names, number of squares and snakes and ladders position) to create the board and start the game.

Game : The game controller class, which handles the dice throw, making the board ready to play, the queue to keep the players to play in round robin fashion, the game play till the game is won.

Player : Gives the details of each player like, the name, the position where the player is. I.e the Square associated with this player. A player can be told to move a certain number of squares, and land into the last square, a ladder, a snake or a regular square already occupied (or not).

package com.example.snakeAndLadder.Exceptions;
InitialSetupException : to handle exceptions that occur during the game board set up.

package com.example.snakeAndLadder.GameSetup;
Board : Handles the board setup to play with, initializing with the types of squares and its position associated with it, marking the first and last squares. Creating the ladder and snakes by setting those positioned square with LadderRole and SnakeRole respectively to support the appropriate jumps of positions. Makes the actual move to appropriate squares on dice throw **Dice** which generates a random number from 1-6.

Square: defines the behavior of the game, giving the position on the board it is associated with, the player present and the type of Role that square is, allowing the player to enter and leave the square and checking if the player is already occupying the square or if it is the last square thus helps to find a valid square position to move upon die throw and if the square is occupied or not.

SquareRole: An abstract class that helps to define the square type providing details and rules that each type of square should behave as like if the square is isOccupied, isFirstSquare, isLastSquare, i.e,
if the player to enter and leave the square;
check if It can be occupied, if player is present or not;
check if It can be Mid squares or first or last;
moveAndLand : To find a valid initial square position to move upon die throw. i.e cannot be beyond the last quare; landHereOrGoHome :

FirstSquareRole : FirstSquareRole extends **SquareRole** and can contain List of players. It is a FirstSquare.

LastSquareRole : extends **SquareRole** and it is a LastSquare.

RegularSquareRole: Any square that is of type SquareRole

LadderRole: extends **SquareRole** and it will find the square position it should jump/ladder shift(+) to upon reaching the square of this type on dice throw.

SnakeRole: extends **SquareRole** and it will find the square position it should jump/snake shift(-) to upon reaching the square of this type on dice throw.

Motivation and Consideration of the design pattern

Factory method to define each Square behavior. The players are played in around robin fashion taking turns one after the other until one of them wins the game.

Requirements:

Total number of squares, name and order of the players are provided.

Assumptions:

Each square can be occupied by only one player. i.e if occupied, stay/return to firstSquare else move to new place.

Constraints:

BoardInitialization Issue:

If there is an issue while creating the new game with specified board details - InitialSetupException

Business Logic: actions to be implemented:

Make the board ready to play with rules of the game characterized with the types of square present in the board for given total number of squares, ladders and snakes.

Make the players list and queue them to play in a round robin fashion.

Play the game by rolling the dice and making the player to move forward till someone wins by entering and leaving the squares with distinct play rule behaviors.

Benefits:

Factory method to define each Square behavior marked along with the position ensures encapsulation without exposing the behavior and rules on how the square works and allows Player to move forward till one of the player wins.

Testing Strategy:

```
<<INITIAL BOARD SETUP>> There are 12 squares
<<INITIAL BOARD SETUP>> ladder from 2 to 6
<<INITIAL BOARD SETUP>> ladder from 7 to 9
<<INITIAL BOARD SETUP>> snake from 11 to 5
<<INITAL SETUP - PLAYERS >> Players are :
1.  Monica
2.  Albert
3.  Noemi
4.  Jaume

 Monica : 1--> 4
 Albert : 1--> 6
 Noemi : 1--> 6--OCCUPIED--> (return to) 1
 Jaume : 1--> 6--OCCUPIED--> (return to) 1
 Monica : 4--> 5
 Albert : 6--> 9
 Noemi : 1--> 6
 Jaume : 1--> 7 --LADDER--> 9--OCCUPIED--> (return to) 1
 Monica : 5--> 9--OCCUPIED--> (return to) 1
 Albert :  Should go to 13 beyond last square 12 so don 't move
 Noemi : 6--> 11--SNAKE --> 5
```

```
Jaume : 1--> 2 --LADDER--> 6
Monica : 1--> 2 --LADDER--> 6--OCCUPIED--> (return to) 1
Albert : 9--> 11--SNAKE --> 5--OCCUPIED--> (return to) 1
Noemi : 5--> 9
Jaume : 6--> 11--SNAKE --> 5
Monica : 1--> 6
Albert : 1--> 4
Noemi :  Should go to 15 beyond last square 12 so don 't move
Jaume : 5--> 10
Monica : 6--> 7 --LADDER--> 9--OCCUPIED--> (return to) 1
Albert : 4--> 5
Noemi :  Should go to 14 beyond last square 12 so don 't move
Jaume :  Should go to 13 beyond last square 12 so don 't move
Monica : 1--> 5--OCCUPIED--> (return to) 1
Albert : 5--> 11--SNAKE --> 5
Noemi : 9--> 11--SNAKE --> 5--OCCUPIED--> (return to) 1
Jaume :  Should go to 16 beyond last square 12 so don 't move
Monica : 1--> 4
Albert : 5--> 9
Noemi : 1--> 2 --LADDER--> 6
Jaume : 10--> 11--SNAKE --> 5
Monica : 4--> 8
Albert : 9--> 12
Albert  has won.
```

Design decision and data structure

Arrays (2D) of integers and String : playerNames, snakesFromTo and snakesFromTo are initialized giving the board details.
Int : position
Random : to generate a random dice throw from 1-6
String : Player name
ArrayList<Square> : Board containing
LinkedList<Player> : Game players

Flow Chart of use cases / Description of the methods

Step 1: Initial players and board setup to start the game with necessary parameters (players names, number of squares and snakes and ladders position). During initial board setup, make the board and list of players playing.

```
private void makeBoard(int numSquares, int[][] ladders, int[][] snakes)
private void makePlayers(String[] playerNames)
```

Step 2: While making board, setup the Squares in the board as ArrayList<Square> making the necessary squares of types (RegularSquareRole, FirstSquareRole, LastSquareRole);
ladders (LadderRole)and snakes (SnakeRole) by calculating the shift count positive/negative respectively by getting the current position.

```
private void makeSquares(int numSquares) throws InitialSetupException
private void makeLadders(int[][] ladders)
private void makeSnakes(int[][] snakes)
```

Step 3: Start the play, Game.play() starting with initial position and move forward until the game is not over for every roll of Dice.roll().

```
public int roll()
private void moveForward ( int moves ) throws InitialSetupException
```

Helper Methods:

```
private void play() throws InitialSetupException
private void startGame()
private void placePlayersAtFirstSquare()
private boolean notOver()
private void movePlayer(int roll) throws InitialSetupException
```

5. UML Diagram