CS501 Assignment 8

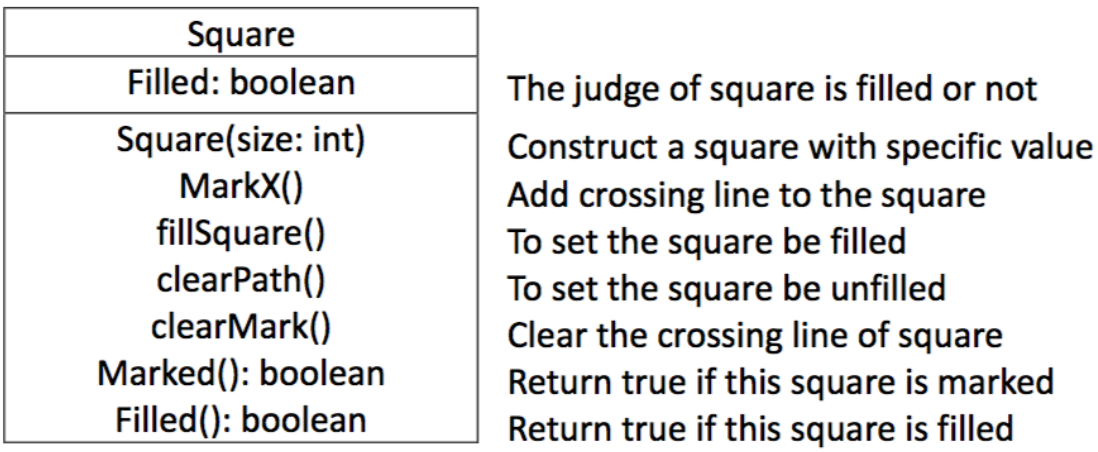
Abstract

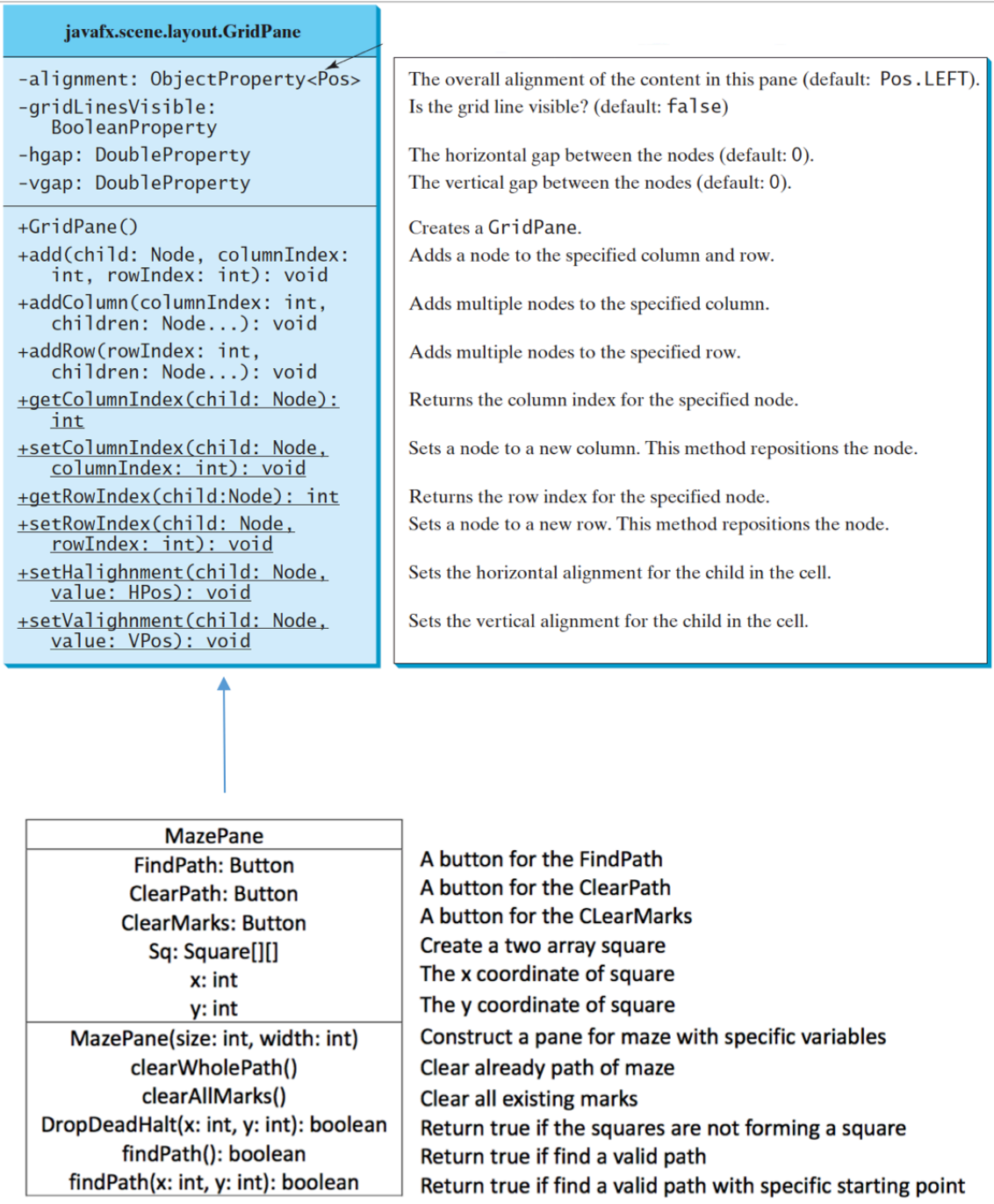
This program can find a valid path from starting point to goal point in a maze, the size can be designed by user by using recursion.

Instruction

When run the java file CS501final with class Square and MazePane, input the size of the maze, for example 8, that will create 8*8 square maze. The path starting point is located at the left-up square and the ending point is located at the right-down point. Then click one time on the square to mark it to be blocked and click one more time to unblocked the square. When all the blocks are set, click the "Find Path" button to find path of maze. If you want to reset the maze, you can click the "Clear Path" and "Clear All Marks", then all the path and marks will be clear and you can set a new maze.

The UML diagram for the class is shown as following:





In the file CS501final.java, I ask user to define the maze size and arrange maze pane and buttons to the scene.

```
public void start(Stage primaryStage) throws Exception{
```

```
        System.out.println("This project can let user to input the
size of maze and create own maze by marking the block by clicking
square.");
        int size = 8;
        //prompt user to define the size of maze
        System.out.println("Please input the size of maze:");
        size = input.nextInt();
        //create a pane
        MazePane pane = new MazePane(size, 50);
        //create a borderPane
        BorderPane borderPane = new BorderPane(pane);
        //add button to hBox
        HBox hBox = new HBox(30, pane.FindPath, pane.ClearPath,
pane.ClearMarks);
        //set position to the center
        hBox.setAlignment(Pos.BASELINE_CENTER);
        //set the gap
        hBox.setPadding(new Insets(10));
        //set buttons to the bottom of borderPane
        borderPane.setBottom(hBox);
        //add borderPane to scene
        Scene scene = new Scene(borderPane);
        primaryStage.setScene(scene);
        //set the window's title
        primaryStage.setTitle("Maze");
        //show result
        primaryStage.show();
    }
```

For the file Square.java, firstly, set a defined number square empty maze, and set every mouse click make square to be marked.

```
    Square(int size) {
        //set the maze pane length and width
        setPrefSize(size, size);
        //set the maze pane border color and background color
        setStyle("-fx-border-color: black;" +"-fx-background-
color: transparent;");
```

```
        //every click on mouse can mark a square
        this.setOnMouseClicked(e -> MarkX());
    }
```

Secondly, define the method MarkX, using the ObservableList class which can update every changing when mark is changed. Every time using this method, if the list is marked, then clear the crossing line, else add the crossing line X to the square.

```
    private void MarkX() {
        //create a observablelist to follow every change
        ObservableList<Node> list = this.getChildren();

        //if list's size is not zero, clear all
        if (list.size() > 0) {
            list.clear();
        } else {
            //else add crossing mark to the square
            list.addAll(
                    //line from left up to right down
                    new Line(0, 0, getWidth(), getHeight()),
                    //line from right up to left down
                    new Line(getWidth(), 0, 0, getHeight())
            );
        }
    }
```

In the MazePane.java
For the MazePane method, at first, create a two arrays square and add every square to the the sq. Then set the starting point and ending point to be disabled. And set the function of the buttons.

```
    public MazePane(int size, int width){
```

```
        //create a new empty square
        sq = new Square[size][size];

        //add square to every unit
        for(int i = 0; i < sq.length; i++)
            for(int j = 0; j < sq[i].length; j++){
                sq[i][j] = new Square(width);
                add(sq[i][j], j, i);
            }

        //set the starting square to be disabled
        sq[0][0].setDisable(true);
        //set the exiting square to be disabled
        sq[size-1][size-1].setDisable(true);
        //set the method clearPath to button ClearPath

        ClearPath.setOnMouseClicked(e -> clearWholePath());
        //set the method findPath to button FindPath
        FindPath.setOnMouseClicked(e -> findPath());
        //set the method clearAllMarks to button ClearMarks
        ClearMarks.setOnMouseClicked(e -> clearAllMarks());

        //set width border of pane is 10
        setPadding(new Insets(10));
    }
```

By using the clearPath method in the class Square to clear all of the square marked as the path.

```
    private void clearWholePath(){
        //from the starting point to clean
        x = 0;
        y = 0;
        //for every point
        for(int i = 0; i < sq.length; i++)
            for(int j = 0; j < sq[i].length; j++)
                //set the square to be not filled
                sq[i][j].clearPath();
    }
```

Using the clearMark method in class Square to clear all of the marked squares.

```
    private void clearAllMarks(){
        //from the starting point to clear
        x = 0;
        y = 0;
        //for every square
         for(int i = 0; i < sq.length; i++)
            for(int j = 0; j < sq[0].length; j++){
                //if this square is marked
                if(sq[i][j].Marked()){
                    //clear the mark
                    sq[i][j].clearMark();
                }
            }
    }
```

In case of the situation when path form a square, if the finding point's around has three or more filled square, this means that this will lead to a wrong path or drop-dead halt. Counting every around square of this point, only return true if the count is equal or less than three.

```
    private boolean DropDeadHalt(int x, int y){
        int count = 0;
        //if there are three or more square is filled around this
square[x][y], this means that the path lost in endless loop
        //right
        if(x < sq[0].length-1 && sq[y][x+1].Filled())
            count++;
        //down
        if(y < sq.length-1 && sq[y+1][x].Filled())
            count++;
        //left
        if(x > 0 && sq[y][x-1].Filled())
            count++;
```

```
        //top
        if(y > 0 && sq[y-1][x].Filled())
            count++;
        //top left
        if(y > 0 && x > 0 && sq[y-1][x-1].Filled())
            count++;
        //top right
        if(x < sq[0].length-1 && y > 0 && sq[y-1][x+1].Filled())
            count++;
        //down right
        if(y < sq.length-1 && x < sq[0].length-1 &&
sq[y+1][x+1].Filled())
            count++;
        //down left
        if(x > 0 && y < sq.length -1 && sq[y+1][x-1].Filled())
            count++;

        //return true if there are 3 around filled squares
        return (count >= 3);
    }
```

In this method, using the recursion to judge and find a valid path toward the goal point. For every step, finding around square is there a valid path or not by using the same method findPath(x, y), if all of the recursioin is true that means this path is correct, and every method has already set path square to be filled. If all of the paths are false, the unfilled the square and return false, which means there are no valid path for this maze.

```
    private boolean findPath(int x, int y){
        //set this square to be filled
        sq[y][x].Filled =true;

        //the square is lost in drop-dead halt
        if(DropDeadHalt(x, y)){
            //unfilled this square
            sq[y][x].Filled = false;
```

```
            //this path is false
            return false;
        }

        //when arriving to the ending square
        if(x == sq[0].length-1 && y == sq.length-1){
            //fill the square and return true
            sq[y][x].fillSquare();
            return true;
        }
        //find the path toward right
        if(x < sq.length-1 && !sq[y][x+1].Marked() &&
!sq[y][x+1].Filled()){
            //if this square has next valid path
            if(findPath(x+1, y)){
                //fill square
                sq[y][x].fillSquare();
                //this path is correct and return true
                return true;
            }
        }
        //find the path toward down
        if(y < sq.length-1 && !sq[y+1][x].Marked() &&
!sq[y+1][x].Filled()){
            //if this square has next valid path
            if(findPath(x,y+1)){
                //fill square
                sq[y][x].fillSquare();
                //this path is correct and return true
                return true;
            }
        }
        //find the path toward left
        if(x > 0 && !sq[y][x-1].Marked() && !sq[y][x-1].Filled()){
            //if this square has next valid path
            if(findPath(x-1, y)){
                //fill square
                sq[y][x].fillSquare();
                //this path is correct and return true
                return true;
            }
```

```
    }
    //find the path toward top
    if(y > 0 && !sq[y-1][x].Marked() && !sq[y-1][x].Filled()){
       //if this square has next valid path
       if(findPath(x, y-1)){
          //fill square
          sq[y][x].fillSquare();
          //this path is correct and return true
          return true;
       }
    }
    //if all of above is not valid, unfilled this square
    sq[y][x].Filled = false;
    //no valid path for this square, return false
    return false;
}
```