Program Design

The following pages describe one possible design for this program. You are not required to use this particular design.

The MazeSquare Class

This class represents a single square of the maze.

Suggested Data Members

- A public `SquareType` `enum` that specifies the different possible types of squares that can be drawn: `WALL`, `SPACE`, and `PATH`.
- A symbolic constant that specifies the dimensions of a square (the sample screenshots use 15 x 15 pixel squares).
- Two integers that store the row and column of the maze square within the maze.
- A `SquareType` variable that stores the type of the square.
- A Boolean variable that specifies whether this square has been visited by the solution algorithm. This should be initialized to false.

Suggested Methods

- A constructor that creates a `MazeSquare` with a specified row, column, and type.
- `clearSquare()` – This method will be called for every square in the maze when the "Clear Solution" button is pressed. It should set the square's `visited` data member to false and if the square's type is currently `PATH`, it should be changed back to `SPACE`.
- `markVisited()` – This method will be called by the solution algorithm. It should set the square's `visited` data member to true.
- `getVisited()` – This method will be called by the solution algorithm. It should return the square's `visited` data member.
- `isWall()` – This method will be called by the solution algorithm. It should return true if this square's type is `WALL` and false if it is not.
- `setToPath()` – This method will be called by the solution algorithm. It should set the square's type to `PATH`.

- `void drawSquare(Graphics g, int startX, int startY)` – This method will be called for each square as part of drawing the maze. The `Graphics` object can be used to call various drawing methods. The parameters `startX` and `startY` are the x and y coordinates of the upper left corner of the maze.

  The method should set the current drawing color based on the square’s type, draw a filled rectangle of that color, and then draw a black rectangle as the border of the square. You will need to compute the x and y coordinates of the upper left corner of these rectangles based on `startX` and `startY`, the row and column of the square, and the square’s dimensions.

The Maze Class

This class represents the maze as a whole.

Suggested Data Members

- Symbolic constants for the maximum number of rows and columns in the maze. The sample program uses the value 30 for both of these.

- Four integers to store the row and column of the start and end of the maze.

- A reference to a two-dimensional array of `MazeSquare` objects.

Suggested Methods

- An accessor method to return the actual number of rows in the maze.

- An accessor method to return the actual number of columns in the maze.

- `void readMaze(File inputFile)` – This method will be called to read an input file of maze data (see **Input** below for a description of the file format). It will allocate the maze array storage and create all of the `MazeSquare` objects. It will also save the row and column coordinates of the start and end of the maze when they are encountered in the input.

  This method could throw `NumberFormatException` or `NoSuchElementException` if the input file is not in the correct format, or `FileNotFoundException` if the file can't be opened.

- `clearMazePath()` – This method will be called to clear the maze solution. It should call `clearSquare()` for every `MazeSquare` in the maze array.

- `void drawMaze(Graphics g, int startX, int startY)` – This method will be called to draw the maze. The parameters `startX` and `startY` are the x and y coordinates of the upper left corner of the maze. The method should call `drawSquare()` for every `MazeSquare` in the maze array.

- `boolean solveMaze()` – This non-recursive method will call the recursive `solveMaze()` method described below, passing it the row and column of the start of the maze. It should return the value returned by the recursive method.

- `boolean solveMaze(int row, int column)` – This private recursive method attempts to solve the maze. It returns true if a solution is found and false if not.

  The method will try to find a solution for the maze by attempting to move north, south, east, and west. If a move reaches the end of the maze, a valid path has been found, and the squares along that path will have their type changed from `SPACE` to `PATH` as the algorithm backtracks. If a move hits a wall or ends up retracing its steps, that is not a valid path. If no valid path is found, the eventual return value will be false.

  Here is the recursive logic for this method, which has three base cases and four recursive cases:

```
// If we’ve reached the end of the maze, we have solved it.
if (the row and column of this MazeSquare is the end of the maze) {
    Call setToPath() for this MazeSquare
    return true;
}

// If we hit a wall or we've already visited this square, this square is
// not part of a valid path to the end of the maze.
if (this MazeSquare is a wall or has been visited)
    return false;

Mark this MazeSquare as visited

// If we're not on the top edge of the maze, try to go north.
if (row is not the first row of the maze) {
    if (call solveMaze() with row-1 and col as arguments) {
        // There is a valid path to the north.
        Call setToPath() for this MazeSquare
        return true;
    }
}

// If we're not on the bottom edge of the maze, try to go south.
if (row is not the last row of the maze) {
    if (call solveMaze() with row+1 and col as arguments) {
        // There is a valid path to the south.
        Call setToPath() for this MazeSquare
        return true;
    }
}

// If we're not on the left edge of the maze, try to go west.
if (col is not the first column of the maze) {
    if (call solveMaze() with row and col-1 as arguments) {
        // There is a valid path to the west.
        Call setToPath() for this MazeSquare
        return true;
    }
}

// If we're not on the right edge of the maze, try to go east.
if (col is not the last column of the maze) {
    if (call solveMaze() with row and col+1 as arguments) {
        // There is a valid path to the east.
        Call setToPath() for this MazeSquare
        return true;
    }
}

// If we haven't returned true by this point, no valid path through the
// maze has been found.
return false;
```

The MazePanel Class

This is a custom panel class that supports drawing a maze on its surface. This class should extend `JPanel`.

Suggested Data Members

- An object reference to a `Maze` object. This should be initialized to `null`.
- A pair of Boolean variables, `solutionAttempted` and `solutionFound`, both of which should be initialized to false.

Suggested Methods

- `void readMaze(File inputFile)` – This method will be called to read an input file of maze data. It should set `solutionAttempted` and `solutionFound` to false, create a new `Maze` object, use that object to call the `readMaze()` method of the `Maze` class, and then call `repaint()` to draw the new maze. Any exceptions thrown by the `readMaze()` method of the `Maze` class can simply be rethrown by this method.
- `clearMazePath()` – This method will be called to clear a maze solution. It should set `solutionAttempted` and `solutionFound` to false, call the `clearMazePath()` method for the `Maze` object data member, and then call `repaint()` to redraw the maze.
- `solveMaze()` – This method will be called to try to solve a maze. It should set `solutionAttempted` to true, set `solutionFound` to result of calling the non-recursive `solveMaze()` method for the `Maze` object data member, and then call `repaint()` to redraw the maze.
- `protected void paintComponent(Graphics g)` – Override this method to draw the maze on the panel's surface (if it exists) and potentially a message about whether or not the maze has been successfully solved.

  Logic for this method might look something like this:

  1. Call the superclass version of `paintComponent()` to draw the panel.
  2. Get the dimensions of the panel by calling `getSize()`.
  3. Draw a filled rectangle in `LIGHT_GRAY` over the entire panel to erase whatever was previously drawn there.
  4. If a `Maze` object exists (i.e., its object reference is not null), compute the x and y coordinates of its upper left corner so that the maze is centered in the panel. Then call the `Maze` object's `drawMaze()` method to draw it.

5. If both `solutionAttempted` and `solutionFound` are true, draw a “Solved!” message in the lower part of the panel. Otherwise, if `solutionAttempted` is true but `solutionFound` is false, draw a “No solution exists for this maze.” message in the lower part of the panel. In both cases, the message should be horizontally centered, with its baseline about 20 pixels above the bottom of the panel.
   (If neither Boolean variable is true, the “Solve Maze” button has not been pressed yet so no message needs to be displayed.)

- You may want to override the methods `getPreferredSize()` and/or `getMinimumSize()` to ensure that your panel has the appropriate dimensions to accommodate a maze of the maximum possible number of rows and columns. Since the panel does not contain any other components, its preferred size will likely default to an unacceptably small value.

  There are other ways to set the dimensions of the panel, such as setting its size when its created or setting the size of the enclosing frame.

The MazeApp Class

This class represents the entire application It will contain the bulk of the user interface and will be responsible for handling events from the user. It should extend `JFrame`.

Suggested Data Members

- A `MazePanel` object.

- Three `JButton` objects to represent the “Open Maze File”, “Solve Maze”, and “Clear Solution” buttons.

- A `JFileChooser` object which will be used to choose an input file.

Suggested Methods

- `main()` – Creates an instance of `MazeApp` and then uses it to call method(s) to set up the user interface elements (e.g., `createAndShowGUI()`). Remember that Swing is not thread-safe, so your Swing objects must be created on the Event Dispatch Thread. See Assignment 0, Part 2 for an example of the code pattern that is typically used to do this.

- A constructor that sets the title on the application’s title bar.

- `actionPerformed()` – Handles `ActionEvents` from the buttons. It’s entirely up to you whether you have a single version of this method that handles events from all three buttons or whether you create a separate handler for each button.

  In general terms, here’s what the handler for each button should do:

  “Open Maze File” button

  - Call the `showOpenDialog()` method of the `JFileChooser` object to allow the user to choose an input file of maze data.
  - Call the `readMaze()` method for the `MazePanel` object to read the chosen file.
  - Enable the “Solve Maze” button.
  - Disable the “Clear Solution” button.

  `readMaze()` may throw one of several possible exceptions; if it does, display an appropriate error message using `JOptionPane.showMessageDialog()` as on Assignment 2.

  “Solve Maze” button

  - Call the `solveMaze()` method for the `MazePanel` object.
  - Disable the “Solve Maze” button.
  - Enable the “Clear Solution” button.

“Clear Solution” button

- Call the `clearMazePath()` method for the `MazePanel` object.
- Enable the “Solve Maze” button.
- Disable the “Clear Solution” button.