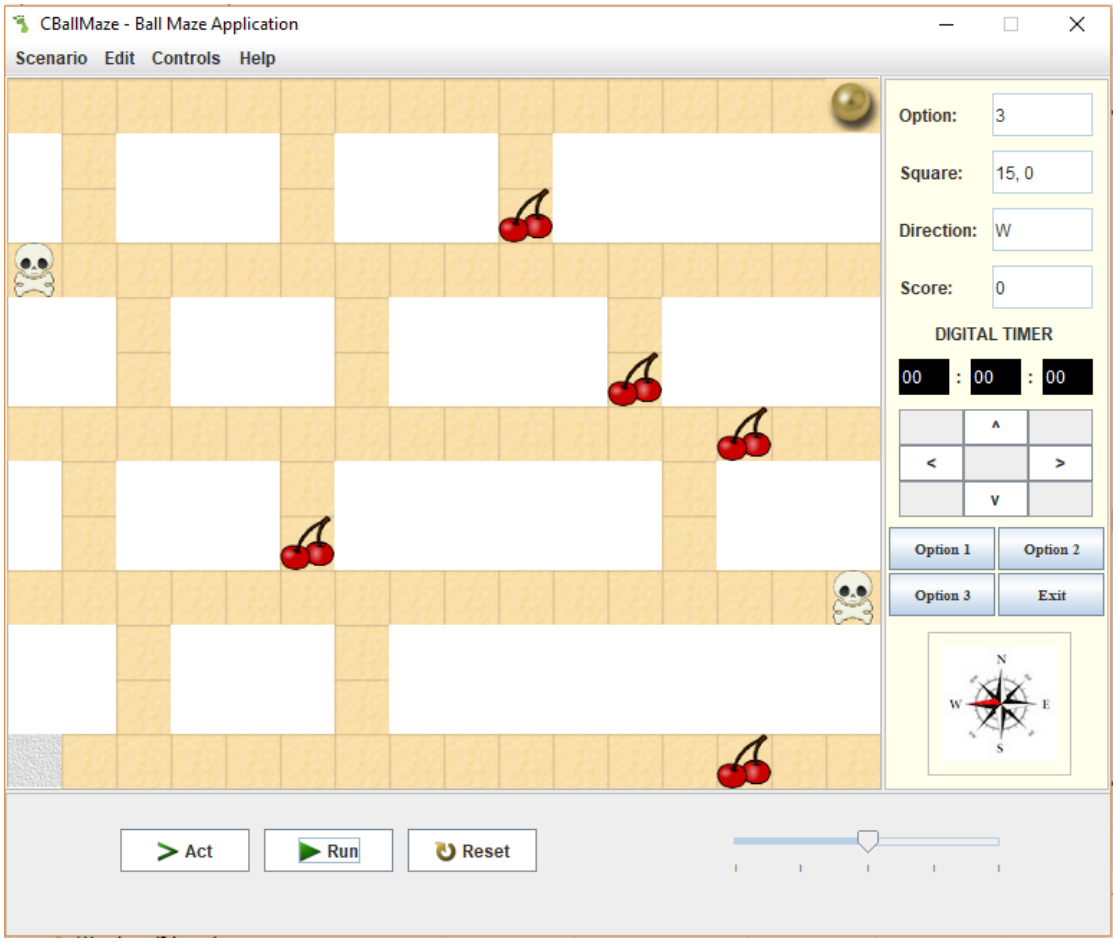
Diwas Lamsal
UN ID - 18406547

Problem Solving and Programming – Second Term Assignment

Application – CBallMaze.java



CBallMaze – Ball Maze Application

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Project Background:

The Term – II of the CSY1020 module mostly deals with creating GUI based Java applications using swing. The previous ways to solve different problems using the console has been replaced with the excessive usage of GUI. The provided assignment assesses the level of understanding and implementation abilities of the subject matter covered within this term.

This project will involve coming up with solutions to the different sets and levels of tasks provided in the brief using java swing. The package javax.swing, which is a part of Oracle's Java Foundation Classes, provides sets of components that can be used to create GUI applications. Using swing, along with other different libraries, the solution was completed. The finished project contains almost all the features listed in the brief including some other extra features.

1.2 Aims and Objectives:

The brief provides a fair set of tasks to work on. This section covers the provided tasks in the brief along with the rules.

The Graphical User Interface requirements (Taken from the brief):

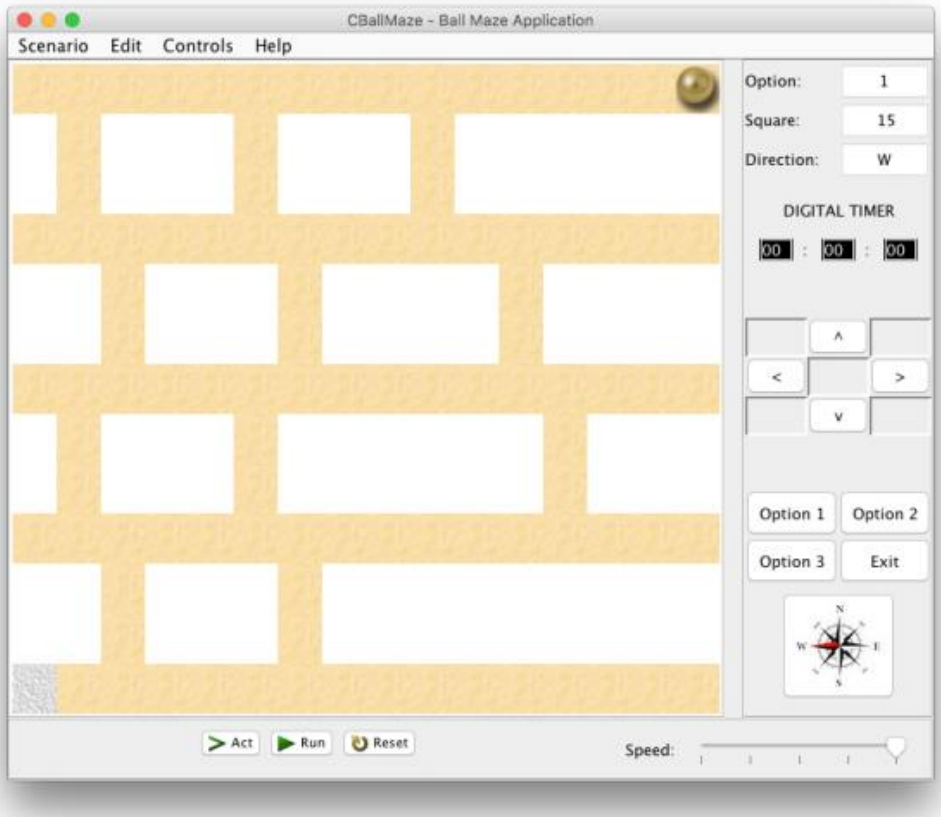


Figure 1.1 – Provided GUI Screen

Essential Functionality:

- 13 x 16 grid of **JButtons** or Icons.
- 4 **JButtons** for the game options 'Option 1, Option 2, Option 3' and 'Exit'.
- 3 **JButtons** for 'Act', 'Run' and 'Reset'.
- 9 **JButtons** for 'Forward >', 'Backwards <', 'Up ^', 'Down v' should move the ball in the appropriate direction by one square for each press (plus 5 blank).
- The compass icon (**JButton**) should illustrate the current direction for the ball.
- 3 **JLabels** for 'Option', 'Square' and 'Direction'.
- 3 **JTextFields** for the current 'Option', Location/'Square' and 'Direction' of the ball. Use the square identification method e.g. 0 to 207 and N, E etc.
- 3 **JLabels** for the '**DIGITAL TIMER** and the two :', with 3 **JTextFields** for the hours, minutes and seconds.
- Create a **JFrame** application, which opens to the set size (775 * 650).
- **JFrame** title set as "*CBallMaze – Ball Maze Application*".

Additional or the Advanced Functionality and Complexity:

- Application icon for the **JFrame** used (Windows only).
- Application dock icon.
- The 'Run' **JButton** should show the ball moving between the continuously from the initial position (Option 1 – default opening state – ball top right-hand corner) to the end position at the grey square/tile (bottom left-hand corner.
- The 'Reset' **JButton** should clear/reset the application to its starting/default opening state.
- The 'Act' **JButton** should step through the above 'Run' sequence one move at a time.
- Discuss and implement the different options for the 3 configurations.
- The 'Option 1, Option 2, Option 3' **JButton**s should display different tile/object configurations/locations.
- A **JMenuBar** could be included with **JMenu**s for the *Scenario, Edit, Controls* and *Help*, which include **JMenuItem**s of *Exit (Scenario), Help Topic* and *About (Help).*
- Additional **JButton**s may be used to improve the applications usability e.g. ball movement – in random/predefined direction, jump objects/obstacles in Option 2 or 3 etc.
- The ball drops down the maze.
- A sound effect is heard when the ball drops down to the next level.
- Create a **JFrame** application, which is not resizable.
- Create a **JFrame** application, which centers itself on the monitor.
- Discuss the possibilities for incorporating intelligence/checks for whether moves are valid.
- Digital Timer should start and stop when 'Run' is pressed and stopped when a ball gets to the end.
- Implement intelligence/checks for whether moves are valid.
- A **moveBall()** method should be used to solve the problem. The **moveBall()** method should include **move(MOVE_LEFT),**

move(MOVE_RIGHT), move(MOVE_UP), move(MOVE_DOWN) methods (see below).

```
public void moveBall()
{move(MOVE_LEFT);
        .....
        move(MOVE_RIGHT);
}
```

The source code file containing the **main()** method and the compiled byte code **class** files should be named as follows: **CBallMaze.java** & **CBallMaze.class**

The Rules (Taken from the brief):

Rules (Basic) Create a simulation of the ball moving around the pitch, where:

- The ball must only travel when on the 'sand'-coloured blocks otherwise it should not move.
- The ball must move one whole 'sand'-coloured blocks at a time every time a movement key is pressed - via a direction button (<, > v ^)) - (when movement is possible).
- Must use the scenario provided.
- Must stop when it the ball reaches the grey block at the end of the maze.
- The basic solution must be completed using the 'act' button (accessing the **moveBall()** method within the **CBallMaze.class**).

Rules (Intermediate and advanced) Create a simulation of a ball moving around the maze, where:

Rules (Intermediate)

- Whilst maintaining the features of the basic solution add the following
- When there is a block below the block the ball is automatically go down. In other words, if the ball can drop it 'falls' down until a white space is below it.
- Add a sound effect when the ball drops.
- The ball must not fall into the white spaces.

Rules (Advanced)

- For higher grades on the solution part of the assignment see the marking scheme/rubric. You must NOT change the layout and all changes should still meet the criteria of **Rules (Basic**).

Besides these rules, the brief asks to follow different conventions and standards used in Java, and asks for excessive usage of comments in the codes.

1.3 Report Structure:

This part contains the structure of the report.

- **Introduction**

The introduction section includes the project background, aims and objectives of the project, the tasks provided in the brief and an introduction to the report.

- **Analysis**

The analysis section is a thorough analysis of the task functionalities as well as the rules provided and in what way these will be handled. It also includes some sets of inputs and outputs required.

- **Design**

The design section contains flowchart forms of different logics implemented for performing various tasks in the program. It also contains some information about the language tools utilized.

- **Implementation**

The implementation section is like a commentary of the code and the resulting effects in the GUI. The screenshots of the code and the result have been displayed in this section.

- **Testing**

The testing section includes tests, expected outcomes and the actual outcomes for various tests performed in the program. This section also contains the screenshots of the test results.

- **Conclusions and Recommendations**

This section includes the evaluation of the task, as in what part of the provided requirements have been fulfilled, what has not been fulfilled. The strengths and weaknesses of the solution, future work required to improve the solution and the approach that would be used, if the program was to be redesigned from scratch.

2. ANALYSIS

2.1 The Problem:

The task has been divided to have different levels of problems. These problems can be analyzed step by step with increasing difficulty. This section comprises of a thorough analysis of the functionalities included in the solution and the inputs and outputs involved with the player.

2.1.1 The Frame and Components:

As shown in the provided GUI Screen – Figure 1.1, this is the layout the application should have when it is complete. The application would consist of the main JFrame, and contain many different JLabels, JPanels, JMenuBar, JMenus, JButtons, Icons and a JSlider. These components should be added to the frame and their respective features, in order to meet the brief.

2.1.2 The Essential Functionalities:

i. The 13 x 16 grid of JButtons or Icons:

The main panel should contain a 13 by 16 grid which contains the maze. This could and has been solved by creating a two-dimensional array of JLabels which would represent the blocks. These can be added into the panel using layout managers such as **GridBagLayout**. The **GridBagConstraints** can be used to define the position of different JLabels with the Icons set as the sandblock icons, golden ball icon and the grey block.

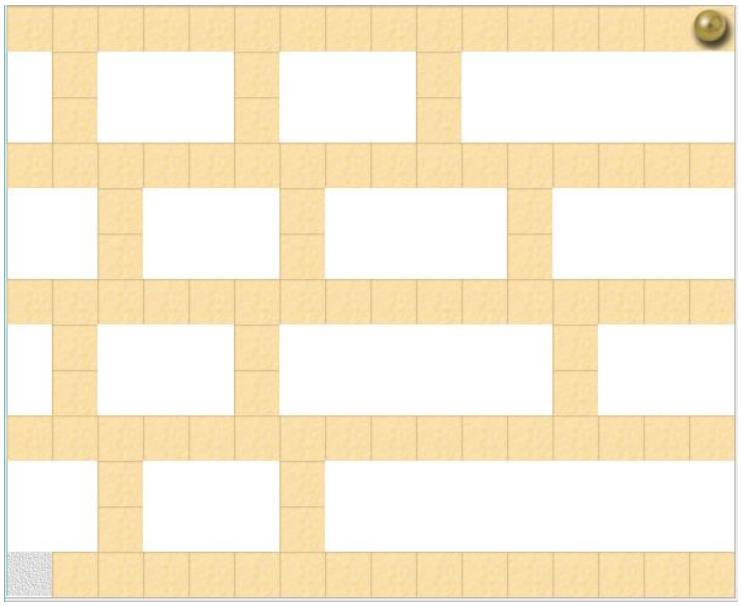


Figure 2.1 – The 13x16 grid

ii. 4 JButtons for the game options 'Option 1, Option 2, Option 3' and 'Exit':

There should be four buttons in the right panel which display "Option 1", "Option 2", "Option 3" and "Exit" as their text. This can be done by creating four

JButtons - jBOption1, jBOption2, jBOption3 and jBExit, setting respective texts on each button and then adding them into the right panel in their given positions. To add these buttons into the panel, the panel's layout could be set to null and each button's positions and sizes could be defined by using the **setBounds** method.

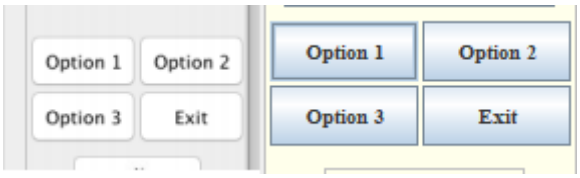


Figure 2.2 – The Option and Exit Buttons

(Note: From left to right, provided and implemented)

iii. 3 JButtons for 'Act', 'Run' and 'Reset'.

Similar to the three Option buttons and the Exit button, the bottom panel should contain three buttons labelled "Act", "Run", and "Reset". These could be added in a similar way but into the bottom panel.

iv. 9 JButtons for 'Forward >', 'Backwards <', 'Up ^', 'Down v' should move the ball in the appropriate direction by one square for each press (plus 5 blank).

The 9 JButtons:

The 9 Buttons could be added to the right panel in a similar way as the other buttons have been added. Additionally, some of these buttons need to be configured to be disabled or have a color set as the background color.

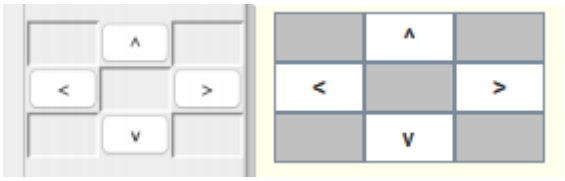


Figure 2.3 – The 9 JButtons

(Note: From left to right, provided and implemented)

Moving the Ball:

The ball could be moved by setting up action listeners associated with the buttons. Whenever the buttons are clicked, the icon towards the direction that the button represents would be set as the ball's icon and the one previously having the ball's icon would be set as the sand block. More on how this has been done in the **4. IMPLEMENTATION** section.

v. 3 JLabels for 'Option', 'Square' and 'Direction':

The right panel should contain three JLabels labelled "Option:", "Square:", and "Direction:". These could be created and added into the right panel using the setBounds method (how the buttons were placed) to make them appear in the required positions.

Figure 2.4 – The Option, Square, Direction labels

vi. 3 JTextFields for the current 'Option', Location/'Square' and 'Direction' of the ball (Use the square identification method e.g. 0 to 207 and N, E etc.):

Similarly, for the three JTextFields, which appear against the three JLabels of Option, Square and Direction, the defining and setBounds method would follow with a default value for them. As given in the scenario, the first selected option would be 1 and hence the text that appears in the Option JTextField would be 1. The **setText** method can be used on these JTextFields to specify some default values they have at startup. Likewise, these JTextFields are set to be disabled for editing by the user or the player of the game. The **setEditable** method can be used and set as false to make these text fields non-editable.

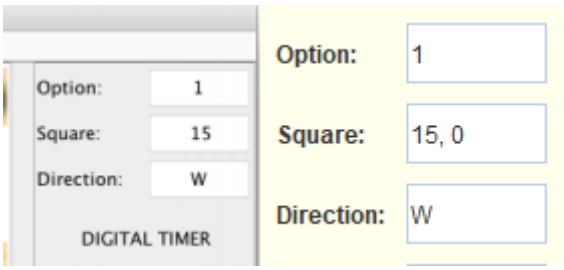


Figure 2.5 – The 3 JTextFields for Option, Square, Direction

(Note: From left to right, provided and implemented)

vii. 3 JLabels for the 'DIGITAL TIMER and the two :', with 3 JTextFields for the hours, minutes and seconds:

As provided in the brief, there should be three JTextFields which would represent the time as hours, minutes and seconds, and three JLabels which would represent the "DIGITAL TIMER" and the two colons which act as the separators for the timer. These could be defined and added using the setBounds method. Additionally, the timer JTextFields are shown as having background as black and the text color as white. The background of these JTextFields should be changed to black with the use of setBackground method and the font color should be changed to white with the use of setForeground method.

Figure 2.6 – The Digital Timer

(Note: From left to right, provided and implemented)

viii. Create a JFrame application, which opens to the set size (775 * 650):

The JFrame application is to be created, which has the size set as 775 x 650. This would be done by extending the class **CBallMaze** as **JFrame**. This would allow the class to inherit all the JFrame properties. The CBallMaze properties are edited in the constructor where the **setSize** method is used to set the size of the JFrame to 775x650.

setSize(775, 650); //Code that sets the size of the frame

ix. JFrame title set as "CBallMaze – Ball Maze Application":

Like the setSize method, the class can now use **setTitle** method to set the title of the frame.

setTitle("CBallMaze – Ball Maze Application"); //Code that sets the title of the frame

2.1.3 The Additional Functionalities:

i. Application icon for the JFrame used (Windows only) and application dock icon.

The application icon for the JFrame can be set in the constructor. The setIconImage method, available for JFrames can be used to set the image for both the dock and the application. The icon can be defined first which could be passed on as an argument in the method to set the icon as the application's icon.

```
Icon iconName = new ImageIcon("images/greenfoot.png");
setIconImage(iconName);
```

ii. The 'Run' JButton should show the ball moving between the continuously from the initial position (Option 1 – default opening state – ball top right-hand corner) to the end position at the grey square/tile (bottom left-hand corner.

To make the Run button show the ball moving continuously from the initial position to the grey block, first, an action listener should be added to the Run button. Some methods would be written which would make the ball move towards left, however, when the ball is able to move in the downward direction, the ball should not move left and move downwards (Figure 2.7). Adding a timer to this method when the Run button is clicked would solve the problem.

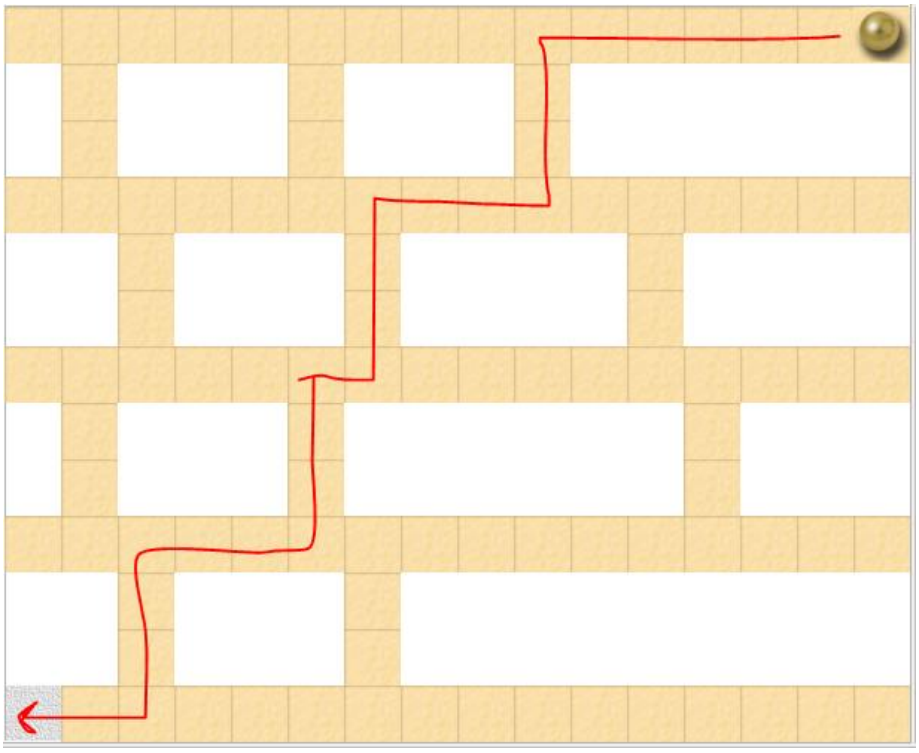


Figure 2.7 – The Ball Movement

iii. The 'Reset' JButton should clear/reset the application to its starting/default opening state.

This can be done simply by setting up the action listener, and, calling a method for the reset button where the frame is disposed (dispose() method) and another instance of the class CBallMaze is created. This would exit the older instance and create a fresh one.

iv. The 'Act' JButton should step through the above 'Run' sequence one move at a time.

After the action listener is set up, unlike the Run button where a timer is enabled on button click, the same method which makes the ball move used for the Run method can be called each time the Act button is clicked.

v. Discuss and implement the different options for the 3 configurations.

There could be lot of possible configurations for the different options. The solution has all the three options utilized wherein the option 1, 2 and 3 would configure the game state following basic, intermediate and advanced rules respectively. The option 1 would be the default run state where the ball would move on its own from the starting position to the end position. Option 2 would have the ball dropping effect as asked by the intermediate rules and the falling sound effect added as this happens. Option 3 would have enemies across the maze and cherries to eat that would increase the score. Also, in the option 3, there would be features that would enable using the arrow keys on the keyboard to move the ball.

vi. A JMenuBar could be included with JMenus for the *Scenario, Edit, Controls* and *Help,* which include JMenuItems of *Exit (Scenario), Help Topic* and *About (Help)*.

To add a JMenuBar, the required JMenuItems (these could be taken from the Greenfoot application used in the first term) would be created and added to the created JMenus which would then be added into the JMenuBar. The frame or the CBallMaze class has a special method setJMenuBar which can set its menu bar as the created JMenuBar.

vii. The ball drops down the maze.

To drop the ball down the maze, a detector could be added which is triggered each time movement occurs. This detector detects if the ball can move down, and when the ball is able to move down, its property of moving left is disabled and a timer could be added which would make the ball move down until it is no longer able to do so (until the block below is not whitespace but the sandblock).

viii. A sound effect is heard when the ball drops down to the next level.

When the detector detects and makes the ball drop down, a sound effect could be added at this exact moment. There are different available sets of libraries to add audio into the application. The audio will be played within the function which makes the ball drop down, but outside the timer. Playing the audio inside the timer would add unwanted echo and noise.

ix. Create a JFrame application, which is not resizable.

The setResizable method could be added into the constructor to make it resizable/ non-resizable according to the argument passed. In this case, it would be setResizable(false).

x. Create a JFrame application, which centers itself on the monitor.

Similarly, to center the application on the monitor, there is a separate method which could be called – setLocationRelativeTo(null). Adding this into the constructor would provide the required results.

xi. Digital Timer should start and stop when 'Run' is pressed and stopped when a ball gets to the end.

A separate timer can be set up for the run button action event. When the run button is clicked, this starts the timer and calls a method every second. Whenever this method is called, the second timer value is incremented by one. Whenever the value reaches 59, in the next second, this would be reset to 0 and the minute value would get incremented. The same would happen for the hour value when the minutes are about to reach 60.

xii. **A moveBall() method should be used to solve the problem. The moveBall() method should include move(MOVE_LEFT), move(MOVE_RIGHT), move(MOVE_UP), move(MOVE_DOWN) methods (see below).**

```
public void moveBall()
{move(MOVE_LEFT);
        ..................
         move(MOVE_RIGHT);
}
```

A method called moveBall() would be used which is the requirement of the solution. In this moveBall() method, another method called move is called. This move method would take a string argument and contain switch case statements. The variables MOVE_LEFT, MOVE_RIGHT and the respective direction indicators would contain string values of "left", "right", etc. Inside the move method, these string values are processed into the switch case statement which would call another function moveBallLeft(), moveBallRight(), etc. according to the argument value. These newer methods would have the code to move the ball in the set direction (name indicates which direction the ball would move).

Similarly, the main class name has been set as **CBallMaze** and the source code file containing the main() method is named as **CBallMaze.java** as per the brief's requirements.

2.1.4 The Rules

According to the criteria provided, the following will be implemented for the different provided rules and the option configurations.

2.1.4.1 Basic (Option 1 Configurations):

- The ball will travel only within the sand blocks and not reach the whitespace.
- The ball will move one whole sandblock and not make partial movements every time it is moved.
- The arrow buttons will make the ball move towards the respective direction which the button indicates.
- The ball stops moving once it reaches the grey block at the end. This is also true for when the run button is pressed.

  Note: Run button makes the ball move continuously from the starting position to the end.
- Pressing the act button would move the ball one step each (this being the same function as of the run button, but only called once instead of making the ball move continuously).
- Sound effect is played upon reaching the end grey block.
- The slider changes the speed of the ball's movement.

2.1.4.2 Intermediate (Option 2 Configurations):

- The ball would appear to be dropping down whenever there is a sandblock under it until it reaches the end (the sandblock above whitespace).
- Sound effect would be added for this event.
- The ball would not fall into the whitespaces.

2.1.4.3 Advanced (Option 3 Configurations):

- This would be the continuation from the Intermediate rules, i.e. it follows all the rules mentioned in the intermediate portion and would have extra features added.
- The ball can be moved using arrow keys from the keyboard (key listeners have been implemented).
- Enemies would be added into the maze which would move continuously across the maze. When the enemies touch the ball, the game is over and reset.
- Sound effect would be played when the enemy touches the ball, i.e. game over.
- Cherries would be placed across the maze which would add to the score when touched by the ball.
- A separate text-field and label would be added to the right panel to display score.
- Some of these cherries would be placed randomly and not in a static place which would make them change their location every time the game is reset.
- Winning or losing the game (touched by the enemy) would display a message saying you won or lost the game while also displaying the score the player was able to secure.
- The about menu-item from the help menu shows a message showing some information about the application.

2.2 Inputs and Outputs

2.2.1 Inputs

Below is a list of inputs to be taken from the user/player:

- Clicks across different buttons
- Clicks for the menu-bar
- Slide for the slider (dragging the level using the mouse or using arrow keys to move it)
- Key presses for moving the ball

2.2.2 Outputs

Below is a list of outputs provided to the user across multiple events:

- Whenever the application is started, the frame which would comprise of all the solutions listed above would display.
- The ball moves when some events are triggered – when the run/act button is clicked when options 1 or 2 are selected and when the left, right, up and down arrow keys are pressed or buttons are clicked.

- When the ball moves, the direction icon, direction text-field and the square values change as well according to the ball's movement and position.
- The run button also triggers the digital timer where the time is displayed from when the run button has been clicked.
- The slider changes the ball's speed for the Run button in options 1 and 2.
- The option 1, 2 and 3 buttons change the option text-field value.
- The option 3 displays a different object configuration where there would be enemies and cherries across different places in the maze.
- The ball touching the cherries would add 10 to the score value in the score text-field and make the cherry disappear.
- An enemy touching the ball would play a sound effect and display a new window displaying a you lose message which also contains the score.
- Reaching the goal in option 3 would play a sound effect and display a new window displaying a congratulations and game won message which also contains the score.
- The about menu-item would display a message showing some information about the application.

3. DESIGN

This section contains the information about how the program was designed to implement the analyzed solutions. To complete the application, every mentioned solution should be coordinated with one another and should be called in a proper hierarchy. This section shows the different techniques available for Java and swing that were implemented throughout the program. It also shows how the program's features were broken down into pieces each being called when necessary.

3.1 Breaking Down the Major Tasks

For improved efficiency, an application is not written in a single method, but, it is a combination of a lot of methods. These methods would have certain features added to them and can be called whenever required. Many different methods have been written and utilized within the program. Not only the methods, but the functionalities can be broken down according to their features. The upcoming sections show some of the major features of the solution and how they work in the form of flowcharts. These features are the broken-down versions of the whole application and together, make up the application.

3.1.1 The Ball's movement (Left Direction used as example)

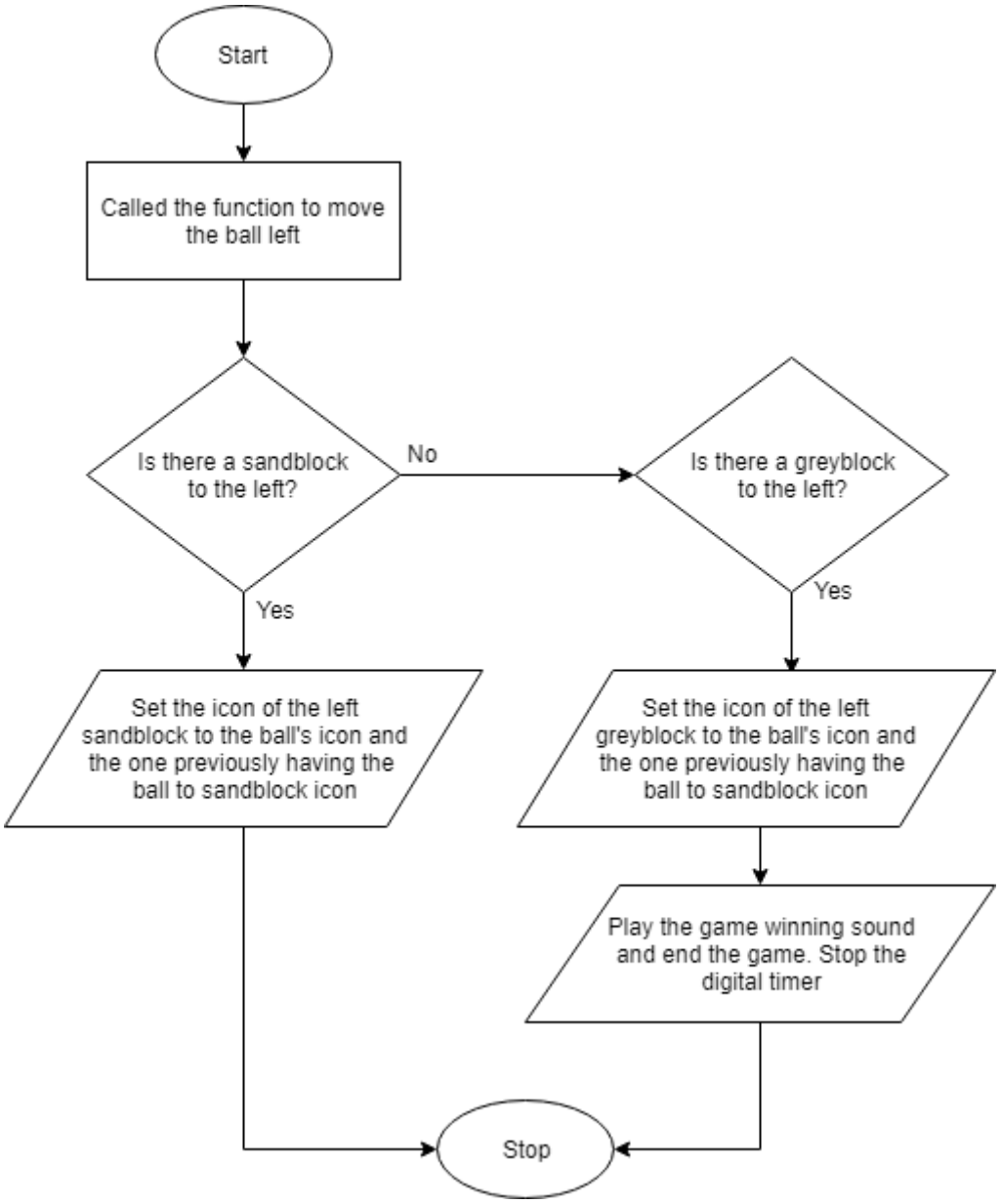


Figure 3.1 – Ball Movement Flowchart

The ball moves in a way that it is not actually the ball moving, but the icon of the next block being changed to the one with the ball. This example takes left movement as a reference. Whenever some actions trigger the call to a function that moves the ball left, if a sandblock exists towards the ball's left, the sandblock's icon is changed to be the one with the ball and the block which previously had the ball would be changed to a normal empty sandblock. Similarly, if the block the ball moved to is a grey block, same things would happen but the game would end. The program would be designed to play a game winning sound when this happens.

3.1.2 The Run button simulation

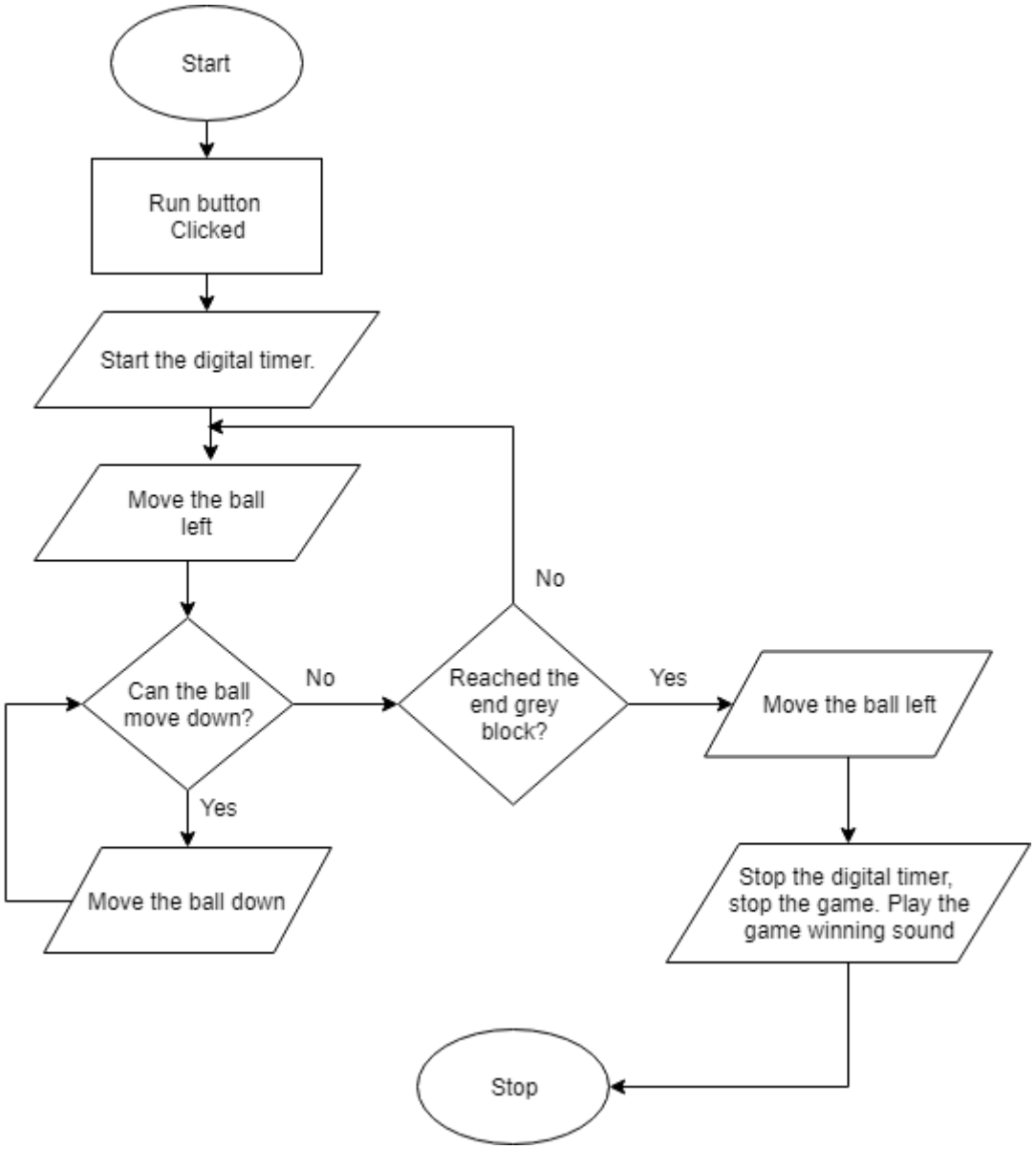


Figure 3.2 – Run Button Simulation Option 1

As the brief asks for, the run button should be a simulation of the ball moving from the starting position to the end, this can be done by moving the ball continuously towards the left and whenever it can move down, moving it down. This logic would work because the maze is set up in such a way that moving left and downwards would make a complete movement from the starting position to the end. Whenever the ball reaches the grey block however, it ends the game.

3.1.3 Intermediate (Option 2) Simulation

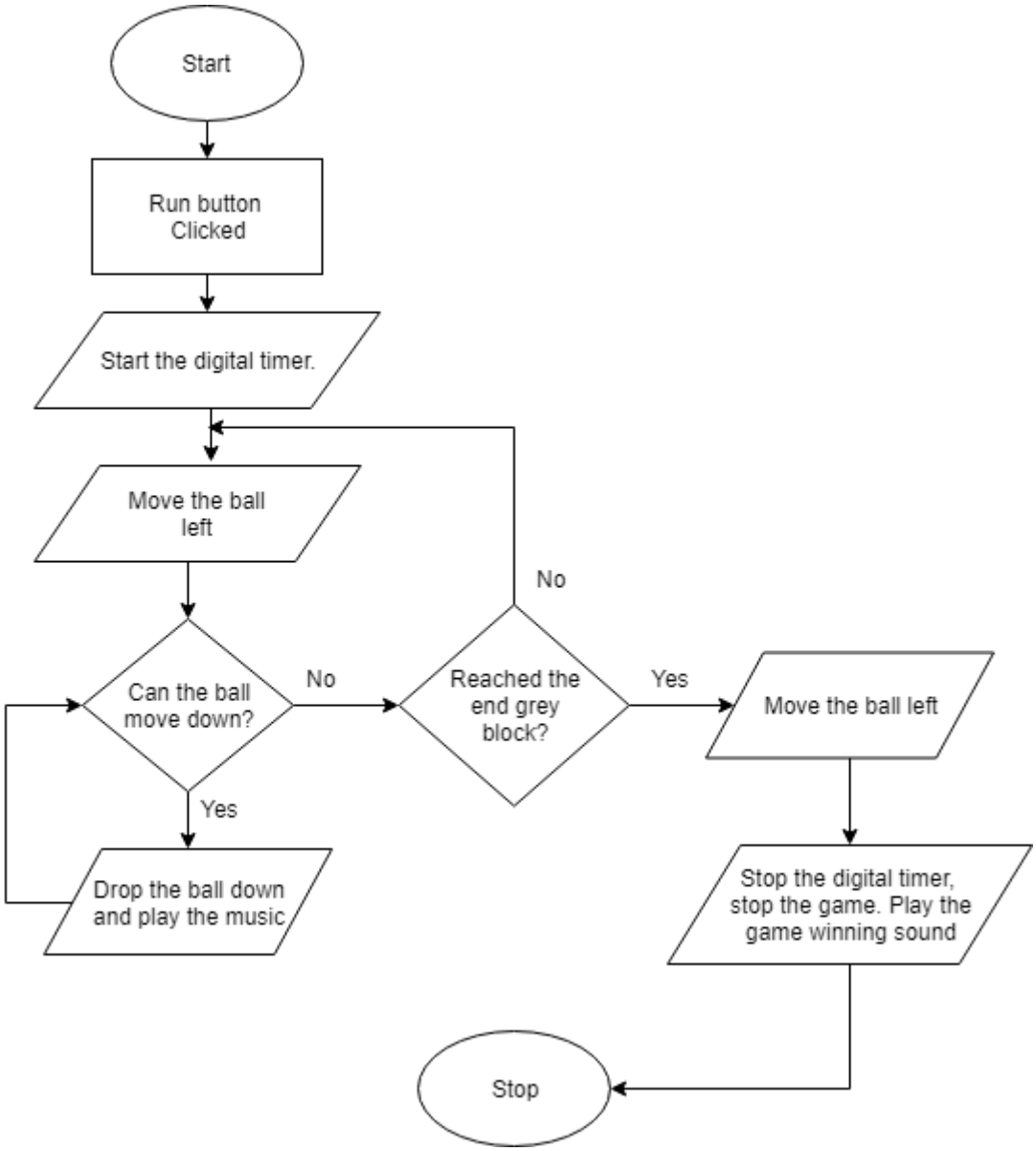


Figure 3.3 – Run Button Simulation Option 2

The intermediate or the option two rule asks to add sound effect and a ball dropping effect whenever the ball moves down. The run button simulation for option two would be almost the same as in option 1 but with two changes. Whenever the ball moves down, it does not move step by step but has a falling down effect, and a sound effect is played when this happens.

3.1.4 Option 3 Layout and Enemy Movements

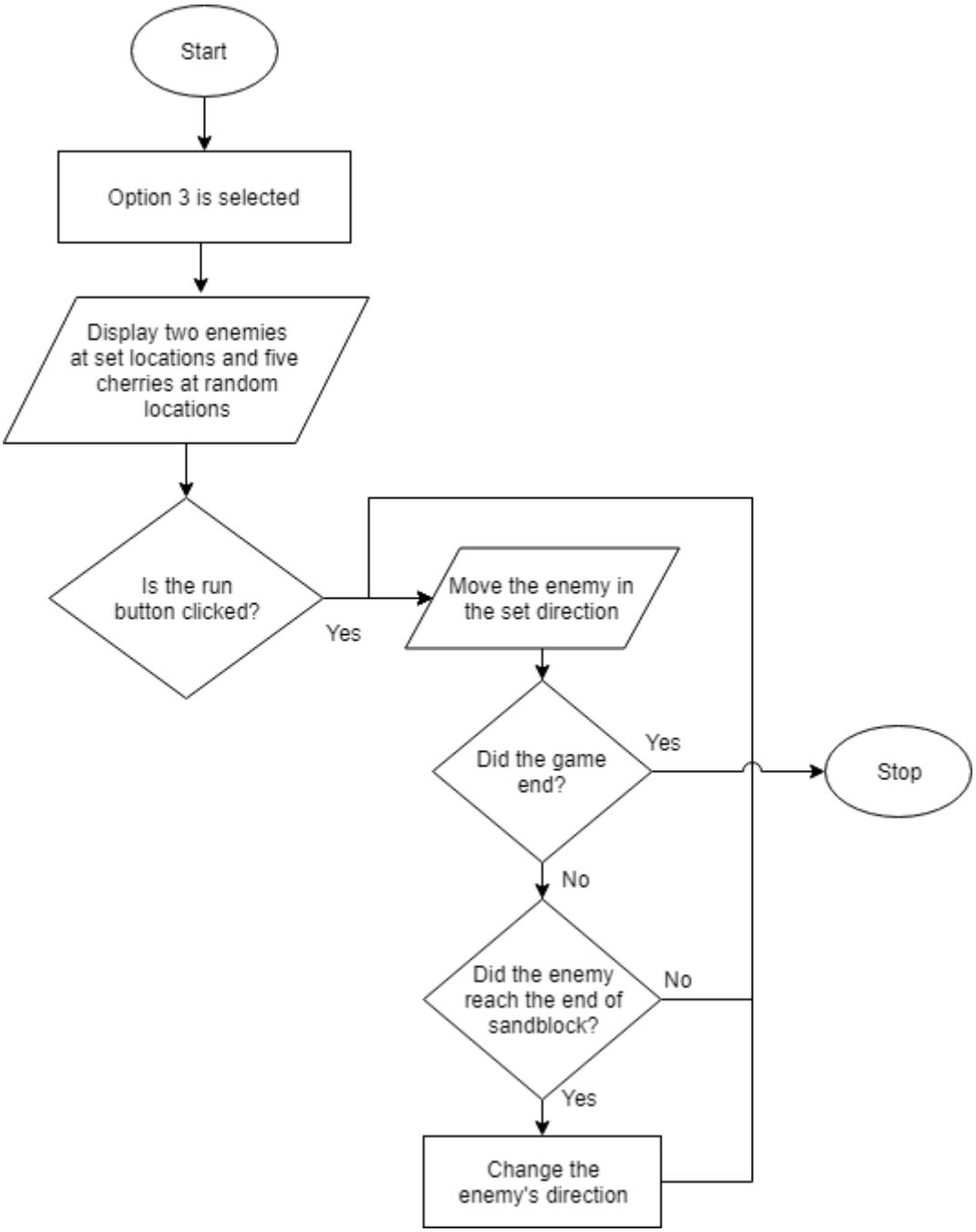


Figure 3.4 – Option 3 Layout and Enemy Movements

The advanced features ask to add additional features for different option configurations. Enemies and cherries are added into the program which make the game more interactive. The enemies continuously move from left to right and change direction upon reaching a corner. Two of the five cherries will be placed at random locations each time option 3 is selected.

3.1.5 Advanced (Option 3) Simulation

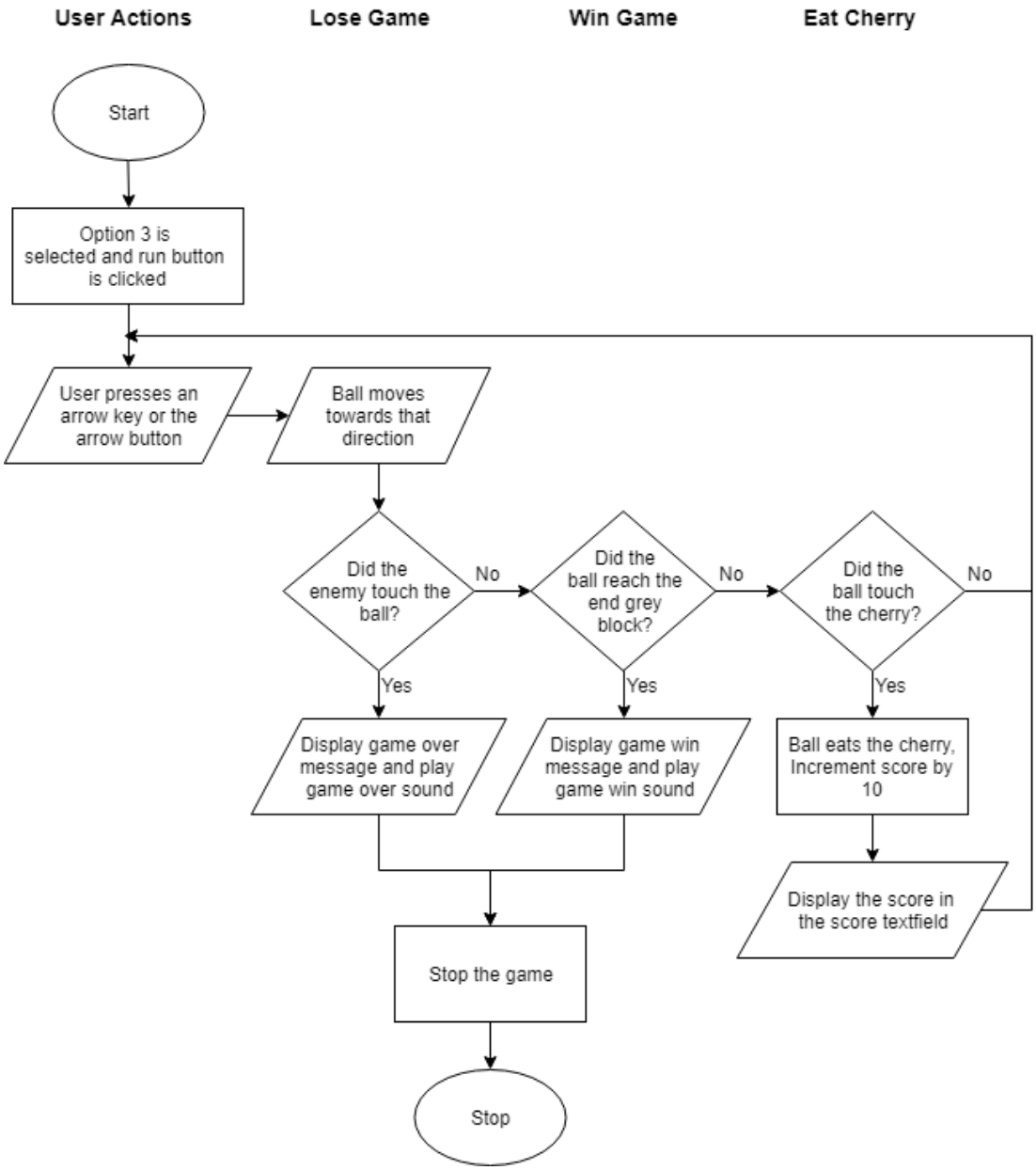


Figure 3.5 – Run Button Simulation Option 3

The added enemies and cherries react to the ball's movements and collision using this logic. The user moves the ball by triggering the movement keys or the arrow buttons. Along the path, the ball might touch the cherries or the enemy. The game should end with a game over message if the enemy touches the ball. The score counter should increment the score by 10 when the ball touches or eats a cherry. Like in any other configurations, the game would end once the ball reaches the grey block. A message appears saying "You Win" when this happens.

3.2 Language Tools Utilized:

Java has many language tools to help the coder write efficient and non-repeating codes. Different such tools have been used to complete the solution. Some of these include loops and nested loops, conditional statements – if…else and switch case, two dimensional arrays and functions. Without the usage of these tools, much part of the solution would be too long, some even being not possible to complete. Similarly, other properties such as inheritance and layout managers have been used. The main class **CBallMaze inherits** all the properties of the **JFrame superclass**, likewise, the layout manager used for the main Maze panel's layout is **GridBagLayout**.

4. IMPLEMENTATION

Like the logics of the solution have been divided to have separate features working together, the code has been written in a similar manner. The code contains different parts, together forming the complete solution. Below are sets of the codes and their use displayed as screenshots.

The first part is the title of the program where some info has been added. [Figure 4.1]

```
/**
 Program: Assignment 2: Application – Ball Maze
 Filename: CBallMaze.java
 @author: Diwas Lamsal
 Course: BSc. Hons Software Engineering Year 1
 Module: CSY1020 Problem Solving & Programming
 Tutor: Kumar Lamichhane
 @version: 1.0 Assignment Solution
 */
```

Figure 4.1 – Title

This is followed by the sets of imports of different libraries used in the program. Next is the class itself, which inherits and implements different properties. Within the class are the declarations for different variables. [Figure 4.2]

```
10
11⊕import java.applet.Applet;
43
44 public class CBallMaze extends JFrame implements ActionListener, KeyListener, ChangeListener{
45
46 //Declaring different objects and variables that make up the program
47     private JButton jBActBtn, jBRunBtn, jBResetBtn;
48
49 //The three main panels within the frame
50     private JPanel jPMaze, jPBottomPanel, jPRightPanel;
51
52 //Borders for the different panels or anywhere necessary
53     Border lightGreyBorder = BorderFactory.createLineBorder(Color.LIGHT_GRAY);
54     Border etchedBorder = BorderFactory.createEtchedBorder(EtchedBorder.RAISED);
55
56 //Icon declaration for different buttons
57     private Icon actIcon, runIcon, pauseIcon, resetIcon;
58
59 //The slider for the bottom panel and labels for the slider
60     private JSlider slider;
61     private JLabel sliderLabel1, sliderLabel2, sliderLabel3, sliderLabel4, sliderLabel5;
```

Figure 4.2 – The import, class and declarations

The constructor for the class contains the necessary code for setting the size of the GUI, making it non-resizable, calling different functions, and many other purposes. The features are shown as comments right next to the piece of code. [Figure 4.3]

```
165     CBallMaze(){
166         super ("CBallMaze - Ball Maze Application"); //Sets the title of the program
167         setSize(775, 650); //Sets the size of the program to 775x650
168         setIconImage(new ImageIcon("images/greenfoot.png").getImage()); //Sets the dock icon
169         setLayout(null); //Sets the layout of the main frame to null
170         setDefaultCloseOperation(EXIT_ON_CLOSE); //Sets default close operation as exit
171         createMenuBar();
172         setJMenuBar(mb); //Sets the menu bar to mb
173         setResizable(false); //Makes the window non-resizable
174         setLocationRelativeTo(null); //Brings the window to the monitor's centre
175         createjPMaze();
176         createjPBottomPanel();
177         createjPRightPanel();
178         addKeyListener(this); //Adds keylistener for this jframe
179         setFocusable(true); //Makes this frame focusable
180         setVisible(true); //Makes this frame visible
181     }
```

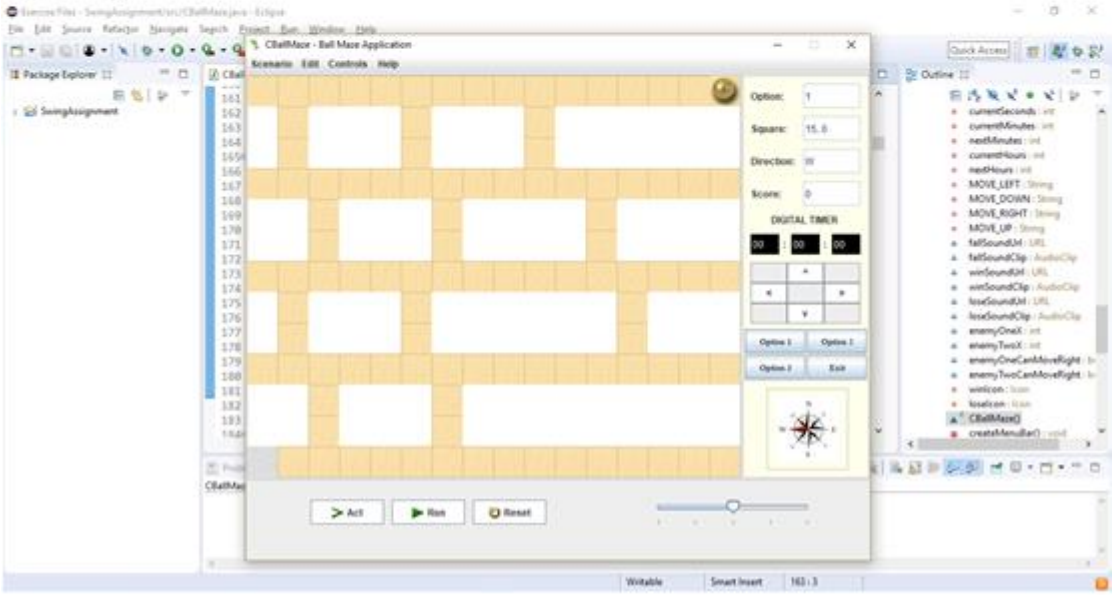


Figure 4.3 – The constructor code and result

This is followed by a method named createMenuBar. This method is called to create a menu-bar. Setting the menu-bar as the JMenuBar for the frame (which is done in the constructor itself) would enable the visibility of this menu-bar at the top of the frame.

```
/*
 *
 *  This method deals with Creating the Menu bar and adding items to the menu bar.
 *  It also deals with the different events associated with the Menu items.
 *
 */

    private void createMenuBar() {
        mb = new JMenuBar();

    //The Scenario Menu and its menu-items

        scenario = new JMenu("Scenario");

        newStrideScenario = new JMenuItem("New Stride Scenario");
        newJavaScenario = new JMenuItem("New Java Scenario");
        open = new JMenuItem("Open");
        close = new JMenuItem("Close");
        saveAs = new JMenuItem("Save as");
        saveAs.addActionListener(this);

        quit = new JMenuItem("Quit");
        quit.addActionListener(this);
```

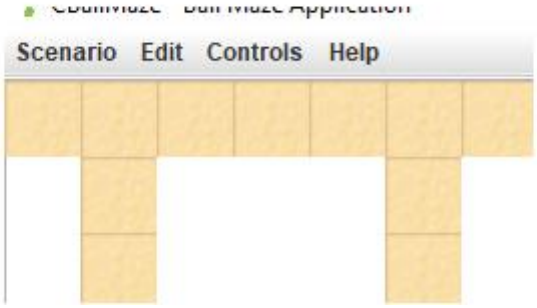


Figure 4.4 – Creating the Menu-bar

The main method contains only a single line of code that creates a new object of the class CBallMaze.

```
/* ------------------------- THE  MAIN METHOD -------------------------*/
    public static void main(String[] args) {
        new CBallMaze();
    }
```

Figure 4.5 – Main method

Next up is the code that creates the main Maze panel. While the way the panel is created is not the most efficient and non-repetitive way, it does its work. The sandblocks created using this method are easy to access and manipulate. The gridbaglayout has been used to create the panel and add the sand blocks.

```
272 //This method creates maze panel
273     private void createjPMaze() {
274         Container window = getContentPane();
275         jPMaze = new JPanel();
276         jPMaze.setBackground(Color.white);
277         jPMaze.setBounds(0, 0, 610, 497);
278         jPMaze.setBorder(etchedBorder);
279         jPMaze.setLayout(new GridBagLayout());
280         createjPMazeLayout();
281         window.add(jPMaze);
282     }
283
284 //This method creates the maze panel's layout (the sand-blocks and the ball)
285     private void createjPMazeLayout() {
286         c.fill = GridBagConstraints.HORIZONTAL;
287         c.gridx = 15;
288         c.gridy = 0;
289
290     //The initial position of the ball
291         jLMazeLabels[c.gridx][c.gridy] = new JLabel(goldenBallIcon);
292         jPMaze.add(jLMazeLabels[c.gridx][c.gridy], c);
293
294     //The first full row of the sand-blocks and the columns follow
295         c.gridx = 0;
```

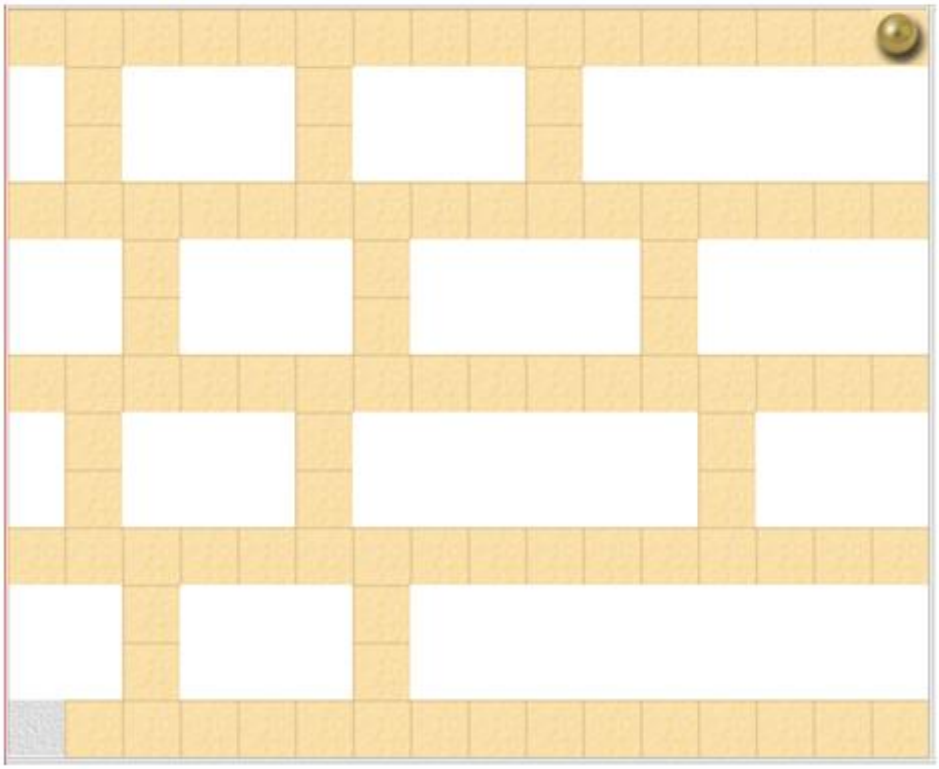


Figure 4.6 – jPMaze panel

This method creates the bottom panel that contains the slider and the three buttons.

```
439 // This method creates the Bottom Panel, also adds buttons and slider to the panel.
440     private void createjPBottomPanel() {
441         Container window = getContentPane();
442         jPBottomPanel = new JPanel();
443 
444     //Icons for the buttons
445         actIcon = new ImageIcon("images/step.png");
446         runIcon = new ImageIcon("images/run.png");
447         pauseIcon = new ImageIcon("images/pause.png");
448         resetIcon = new ImageIcon("images/reset.png");
449 
450     //Act button
451         jBActBtn = new JButton("Act", actIcon);
452         jBActBtn.setBackground(Color.white);
453         jBActBtn.addActionListener(this);
454         jBActBtn.setBounds(80, 25, 90, 30);
455         jBActBtn.addKeyListener(this);
456 
457     //Run button
458         jBRunBtn = new JButton("Run");
459         jBRunBtn.setIcon(runIcon);
460         jBRunBtn.addActionListener(this);
461         jBRunBtn.addKeyListener(this);
462         jBRunBtn.setBackground(Color.white);
```

Figure 4.7 – jPBottomPanel panel

There is a similar method for creating the right panel. The panel components are created first and added into the panel which is then added into the window. This method is called in the constructor itself and so is the createjPBottomPanel method.

```
508 //This method creates the right panel, also adding different components to it.
509 
510     private void createjPRightPanel() {
511         Container window = getContentPane();
512         jPRightPanel = new JPanel();
513 
514     //Panel three the first three labels and TextFields
515         jLOption = new JLabel("Option: ");
516         jLSquare = new JLabel("Square: ");
517         jLDirection = new JLabel("Direction: ");
518         jLScore = new JLabel("Score: ");
519 
520 
521         jLOption.setBounds(10, 10, 70, 30);
522         jLSquare.setBounds(10, 50, 70, 30);
523         jLDirection.setBounds(10, 90, 70, 30);
524         jLScore.setBounds(10, 130, 70, 30);
525 
526         jTFOptionText = new JTextField("1");
527         jTFSquareText = new JTextField("15, 0");
528         jTFDirectionText = new JTextField("W");
529         jTFScoreText = new JTextField("0");
530         jTFOptionText.setEditable(false);
531         jTFSquareText.setEditable(false);
```

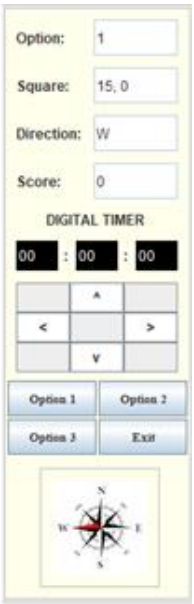


Figure 4.8 – jPRightPanel panel

The digital timer labels and the text-fields have been added and the code is included in the right panel creating method.

```
554
555     //Panel Three digital timer labels and text-fields
556         jLDigitalTimer = new JLabel("DIGITAL TIMER");
557         jLDigitalTimer.setBounds(35, 165, 150, 25);
558
559         timerHours = new JTextField("00");
560         timerHours.setForeground(Color.white);
561         timerHours.setBorder(etchedBorder);
562         timerHours.setBackground(Color.black);
563         timerHours.setBounds(10, 195, 35, 25);
564
565         jLTimerSeparatorOne = new JLabel(":");
566         jLTimerSeparatorOne.setBounds(50, 195, 5, 25);
567
568         timerMinutes = new JTextField("00");
569         timerMinutes.setForeground(Color.white);
570         timerMinutes.setBorder(etchedBorder);
571         timerMinutes.setBackground(Color.black);
572         timerMinutes.setBounds(60, 195, 35, 25);
573
574         jLTimerSeparatorTwo = new JLabel(":");
575         jLTimerSeparatorTwo.setBounds(100, 195, 5, 25);
576
577         timerSeconds = new JTextField("00");
578         timerSeconds.setForeground(Color.white);
579         timerSeconds.setBorder(etchedBorder);
580         timerSeconds.setBackground(Color.black);
```

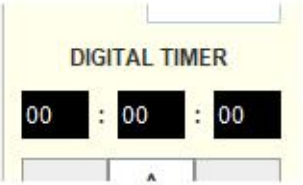


Figure 4.9 – Digital Timer

The digital timer is followed by the code to create the arrow buttons. The empty buttons are created in a similar manner. This is followed by the code to create the Option buttons. The option buttons are created and added in the same way as the arrow buttons.

```
594     //Panel Three arrow buttons
595         upButton = new JButton("^");
596         upButton.addActionListener(this);
597         leftButton = new JButton("<");
598         leftButton.addActionListener(this);
599         rightButton = new JButton(">");
600         rightButton.addActionListener(this);
601         downButton = new JButton("v");
602         downButton.addActionListener(this);
603
604         upButton.setBackground(Color.white);
605         leftButton.setBackground(Color.white);
606         rightButton.setBackground(Color.white);
607         downButton.setBackground(Color.white);
608
609         upButton.setBounds(55, 230, 45, 25);
610         leftButton.setBounds(10, 255, 45, 25);
611         rightButton.setBounds(100, 255, 45, 25);
612         downButton.setBounds(55, 280, 45, 25);
613
614         jPRightPanel.add(upButton);
615         jPRightPanel.add(leftButton);
616         jPRightPanel.add(rightButton);
617         jPRightPanel.add(downButton);
618
619     //Panel Three empty buttons
```

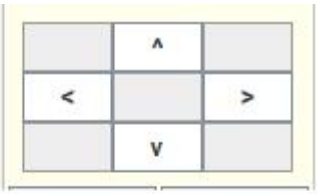


Figure 4.10 – Arrow buttons

This is the moveBall method as asked in the solution. This contains method callings like move(MOVE_RIGHT) and move(MOVE_LEFT). This method checks what the option numbers are and based on that calls the move method. For example: - it is because of this method that in option 2, the ball has a fall down effect.

```
/*
 * moveBall method for the act and run buttons the move(MOVE_RIGHT).. have been
 * implemented.
 *
 * Different sounds have been added to be used across multiple events
 * Such as: winning the game, the ball falling and touching the enemy.
 *
 * Additionally, the different options have different move configurations set up
 * Option 1 is the default
 * The option 2 would activate the intermediate move methods
 * In option 3, the ball is set to be moved by the user
 */

    private void moveBall() {
        boolean canMoveLeft = true;
        JLabel jLCheckDown = jLMazeLabels[xCounter][yCounter+1];
        String optionNumber = jTFOptionText.getText();

        try {
            fallSoundUrl = new URL("file:images/fall.wav");
        } catch (MalformedURLException e1) {}

        fallSoundClip = Applet.newAudioClip(fallSoundUrl);

        switch(optionNumber) {
     //When the option selected is Option 1
        case "1":
            if(jLCheckDown != null) {
                canMoveLeft = false;
                move(MOVE_DOWN);
            }
            if (xCounter>0) {
                JLabel jLCheckLeft = jLMazeLabels[xCounter-1][yCounter];
                if(jLCheckLeft != null && canMoveLeft &&
                        jLCheckLeft.getIcon().equals(sandBlockIcon)) {
                    move(MOVE_LEFT);
                }
                if(jLCheckLeft != null && canMoveLeft &&
                        jLCheckLeft.getIcon().equals(greyBlockIcon)) {
                    move(MOVE_LEFT);
                    win();
                }
            }
            break;
```


```
        switch(optionNumber) {
     //When the option selected is Option 1
        case "1":
            if(jLCheckDown != null) {
                canMoveLeft = false;
                move(MOVE_DOWN);
            }
            if (xCounter>0) {
                JLabel jLCheckLeft = jLMazeLabels[xCounter-1][yCounter];
                if(jLCheckLeft != null && canMoveLeft &&
                        jLCheckLeft.getIcon().equals(sandBlockIcon)) {
                    move(MOVE_LEFT);
                }
                if(jLCheckLeft != null && canMoveLeft &&
                        jLCheckLeft.getIcon().equals(greyBlockIcon)) {
                    move(MOVE_LEFT);
                    win();
                }
            }
            break;
```


Figure 4.11– moveBall method

The move method is called from the moveBall method with arguments. These arguments decide which direction to move the ball.

```
private void move(String direction) {
    switch(direction) {
    case "left":
        moveBallLeft();
        break;
    case "down":
        moveBallDown();
        break;
    case "up":
        moveBallUp();
        break;
    case "right":
        moveBallRight();
        break;
    }
}
```

Figure 4.12– move method

This is the way the ball has been moved across the maze. The example shows the way to move the ball left, but a similar logic has been used to move the ball towards any direction. If the ball can move left, and the left button is clicked or this method is called, the ball has to move into the left sandblock. Similarly, reaching the end grey block should result in win condition, which is by default a win sound and some addition in the option 3.

```
785 //  Moving ball to the Left With the respective button or method calls
786     private void moveBallLeft() {
787         if (xCounter>0) {
788             JLabel jLCheck = jLMazeLabels[xCounter-1][yCounter];
789             if(jLCheck != null && (jLCheck.getIcon().equals(sandBlockIcon)
790                     ||jLCheck.getIcon().equals(cherryIcon))) {
791                 jLMazeLabels[xCounter][yCounter].setIcon(sandBlockIcon);
792                 jLMazeLabels[xCounter-1][yCounter].setIcon(goldenBallIcon);
793                 xCounter--;
794                 jLDirectionIcon.setIcon(directionImageWest);
795                 jTFDirectionText.setText("W");
796                 jTFSquareText.setText(""+xCounter+", "+yCounter);
797             }
798             else if(jLCheck != null && jLCheck.getIcon().equals(greyBlockIcon)) {
799                 jLMazeLabels[xCounter][yCounter].setIcon(sandBlockIcon);
800                 jLMazeLabels[xCounter-1][yCounter].setIcon(goldenBallIcon);
801                 xCounter--;
802                 jTFSquareText.setText(""+xCounter+", "+yCounter);
803                 win();
804             }
805         }
806     }
```

Figure 4.13– moveBallLeft method

Enemies and cherries have been added for the option 3 and a method creates this layout. The two of the cherries are placed randomly which would affect the score of the player.

```
853 //This method creates the option three layout
854     private void createOptionThreeLayout() {
855 //      Create the five cherries two of the cherries are placed randomly
856         int secondlastRandom = (int)(1 + Math.ceil(Math.random()*14));
857         int lastRandom = (int)(1 + Math.ceil(Math.random()*14));
858 
859         jLMazeLabels[9][2].setIcon(cherryIcon);
860         jLMazeLabels[11][5].setIcon(cherryIcon);
861         jLMazeLabels[secondlastRandom][6].setIcon(cherryIcon);
862         jLMazeLabels[5][8].setIcon(cherryIcon);
863         jLMazeLabels[lastRandom][12].setIcon(cherryIcon);
864 
865 //      Create the five enemies
866         jLMazeLabels[0][3].setIcon(enemyIcon);
867         jLMazeLabels[15][9].setIcon(enemyIcon);
868         enemyMoveTimer = new Timer(200, new ActionListener() {
869             public void actionPerformed(ActionEvent e) {
870                 moveEnemy();
871                 detectCollision();
872             }
873         });
874     }
```

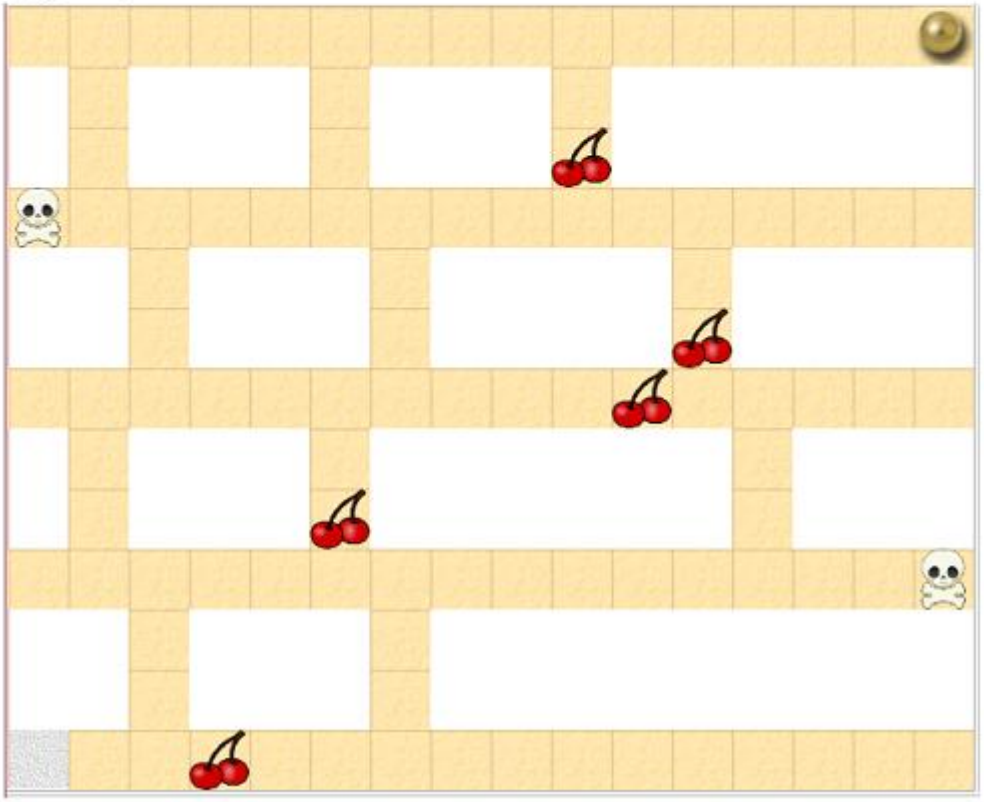


Figure 4.14 – Option 3 layout

This method moves the enemies created in the option 3 layout after the run button is clicked.

```
877 //Moving the enemies in option 3 layout
878     private void moveEnemy() {
879         if (enemyOneX<15 && enemyOneCanMoveRight) {
880             jLMazeLabels[enemyOneX][3].setIcon(sandBlockIcon);
881             enemyOneX++;
882             jLMazeLabels[enemyOneX][3].setIcon(enemyIcon);
883         }
884         else if(enemyOneX>0) {
885             enemyOneCanMoveRight = false;
886             jLMazeLabels[enemyOneX][3].setIcon(sandBlockIcon);
887             enemyOneX--;
888             jLMazeLabels[enemyOneX][3].setIcon(enemyIcon);
889         }
890         else {
891             enemyOneCanMoveRight = true;
892         }
893 
894         if (enemyTwoX<15 && enemyTwoCanMoveRight) {
895             jLMazeLabels[enemyTwoX][9].setIcon(sandBlockIcon);
896             enemyTwoX++;
897             jLMazeLabels[enemyTwoX][9].setIcon(enemyIcon);
898         }
899         else if(enemyTwoX>0) {
900             enemyTwoCanMoveRight = false;
901             jLMazeLabels[enemyTwoX][9].setIcon(sandBlockIcon);
902             enemyTwoX--;
903             jLMazeLabels[enemyTwoX][9].setIcon(enemyIcon);
```

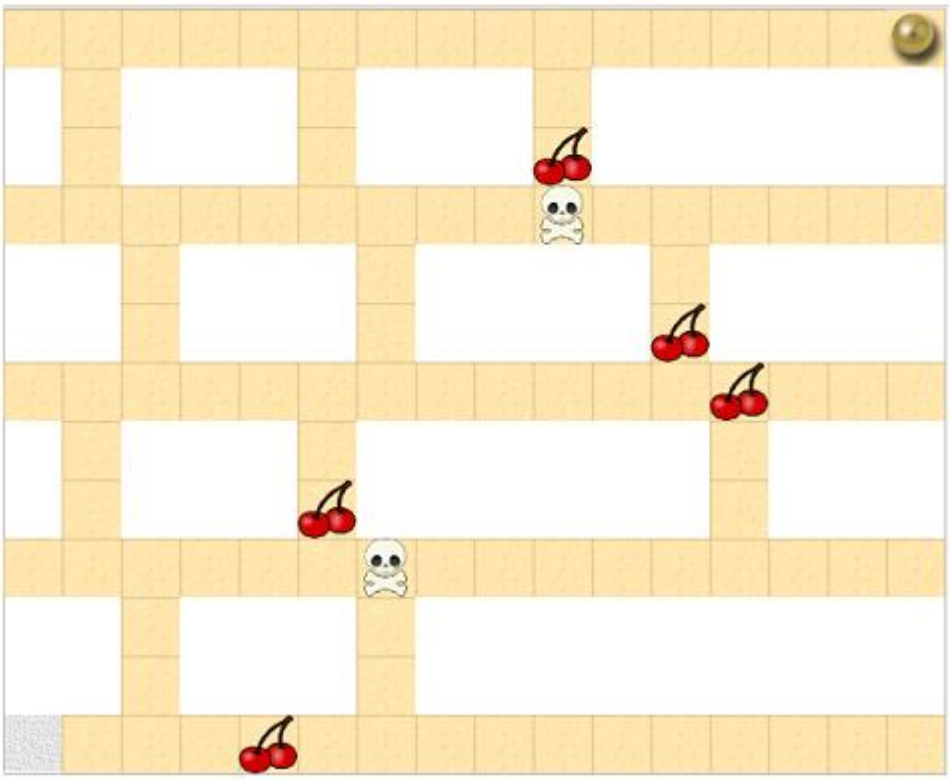


Figure 4.15 – Move Enemy

The method is used to see if the ball has touched any cherry or has been touched by the enemy. Touching the cherry would add a score of 10 in the score text field, whereas being touched by enemy would bring in the game over effect. Although this is not the correct way to detect collisions, the default collision detection logic brought in lots of bugs. This is regarded as a weakness of the program and will be discussed later in the evaluation section.

```
910 /*
911  * Detecting that the ball is touched by the enemy or the cherries are touched by
912  * the ball
913  *
914  */
915     private void detectCollision() {
916         JLabel jCheckIcon;
917         int ballCounter = 0;
918         int cherryCounter = 0;
919         for (int i = 0; i<16; i++) {
920             for (int j=0; j<13; j++) {
921                 jCheckIcon = jLMazeLabels[i][j];
922                 if(jCheckIcon!=null && jCheckIcon.getIcon().equals(goldenBallIcon)) {
923                     ballCounter = 1;
924                 }
925                 if(jCheckIcon!=null && jCheckIcon.getIcon().equals(cherryIcon)) {
926                      cherryCounter++;
927                 }
928             }
929         }
930         jTFScoreText.setText((5-cherryCounter) + "0");
931         if(ballCounter==0)gameOver();
932     }
```

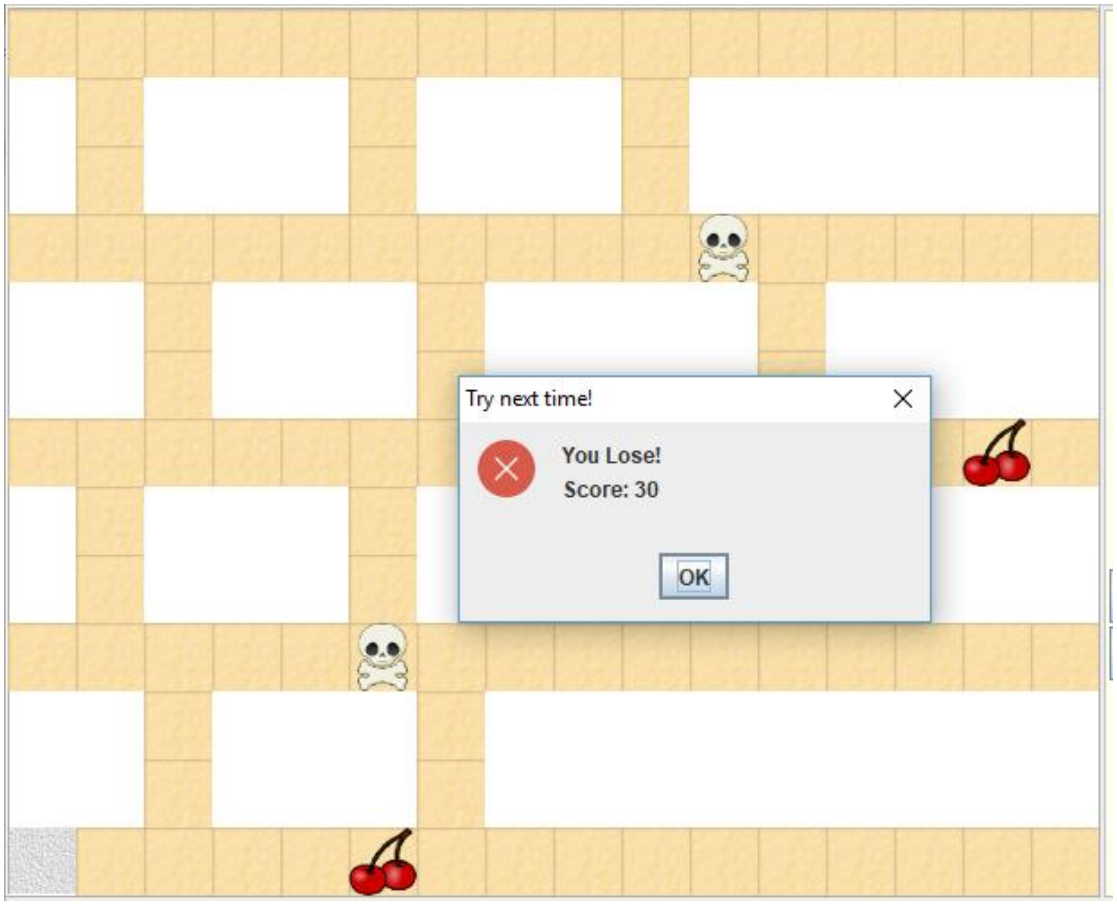


Figure 4.16 – Collisions

The run method is for starting the timer for the digital timer, as well as moving the ball from the start position to the end position (which is not true for the option 3 however, the run button must be clicked in option 3 in order to move the ball and the enemies). Clicking the run button will set the text of the button to pause and the icon to pause icon.

```
934 //Start moving the ball. For the ball to start moving, the run button must be pressed.
935 //This is true for all the options.
936
937     private void run() {
938         jBRunBtn.setText("Pause");
939         jBRunBtn.setIcon(pauseIcon);
940
941         timerForRun = new Timer(1000, new ActionListener() {
942             public void actionPerformed(ActionEvent runClick) {
943                 startTimer();
944             }
945         });
946
947         timerForBall = new Timer(750, new ActionListener() {
948             public void actionPerformed(ActionEvent runClick) {
949                 moveBall();
950             }
951         });
952
953         timerForBall.start();
954         timerForRun.start();
955         if(enemyMoveTimer!=null)enemyMoveTimer.start();
956     }
```

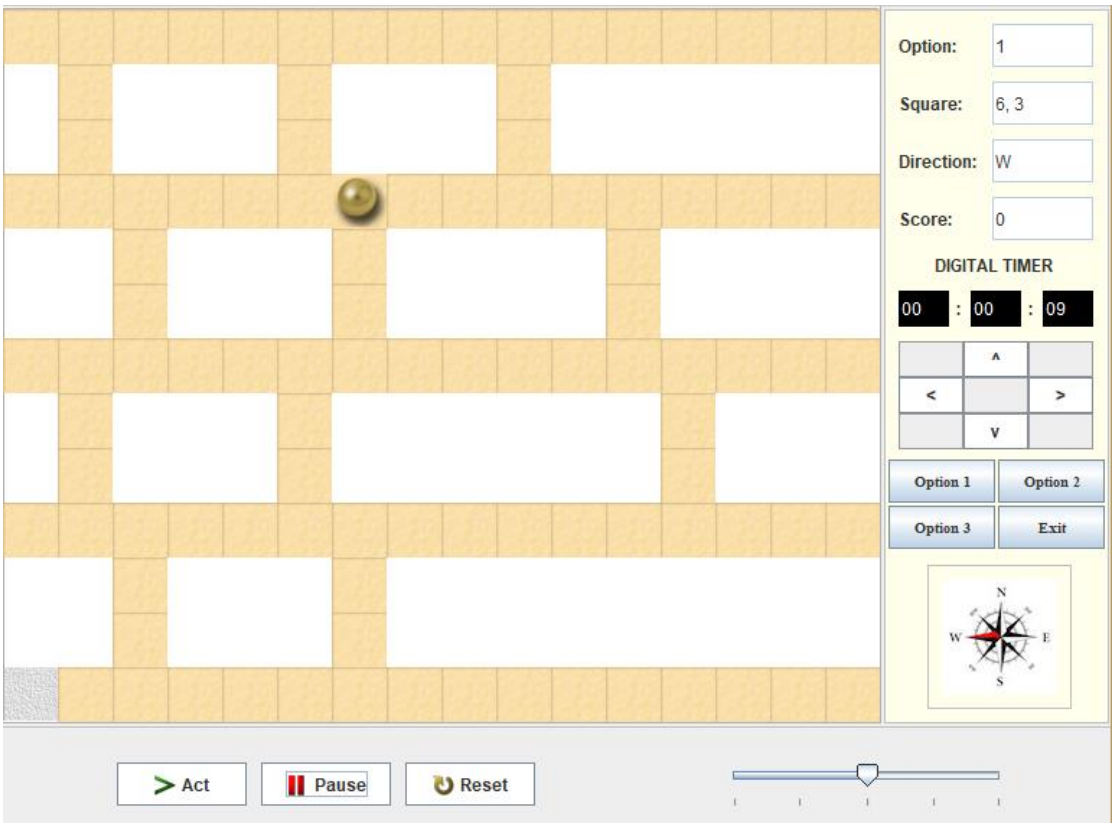


Figure 4.17 – run Method

The pause method stops the timers called in the run method. It has an effect of pausing the game.

```
958 //Pause the game, stop the ball from moving.
959     private void pause() {
960         jBRunBtn.setText("Run");
961         jBRunBtn.setIcon(runIcon);
962         if (timerForRun!=null && timerForRun.isRunning())timerForRun.stop();
963         if (timerForBall!=null && timerForBall.isRunning())timerForBall.stop();
964         if(enemyMoveTimer!=null && enemyMoveTimer.isRunning())enemyMoveTimer.stop();
965     }
```

Figure 4.18 – pause Method

The win method is used to end the game when the ball reaches the end grey block. This plays a winning sound, stops all the timers and displays a game winner message if the selected option is option 3.

```
967 //Win The Game
968     private void win() {
969         if (timerForRun!=null && timerForRun.isRunning())timerForRun.stop();
970         if (timerForBall!=null && timerForBall.isRunning())timerForBall.stop();
971 
972         try {
973             winSoundUrl = new URL("file:images/win.wav");
974         } catch (MalformedURLException e1) {}
975         winSoundClip = Applet.newAudioClip(winSoundUrl);
976         winSoundClip.play();
977         pause();
978 
979         if(jTFOptionText.getText().equals("3")) {
980             JOptionPane.showMessageDialog(null, "You Win\n"
981                     + "Score: "+ jTFScoreText.getText() + "\n \n",
982                     "Congratulations!!",
983                     JOptionPane.INFORMATION_MESSAGE,
984                     winIcon);
985             reset();
986         }
987     }
```

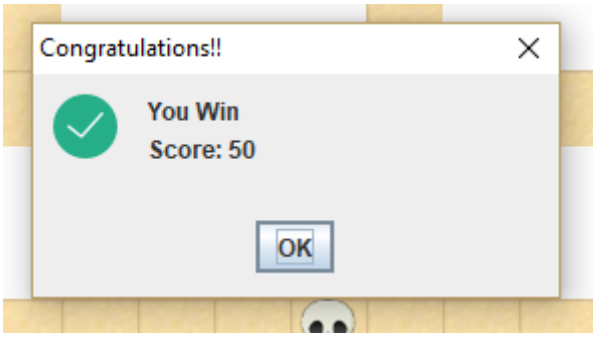


Figure 4.19 – win Method

The game over method is called in the option 3. This brings up a popup showing the game over message.

```
989 //Game Over
990     private void gameOver() {
991         try {
992             loseSoundUrl = new URL("file:images/lose.wav");
993         } catch (MalformedURLException e1) {}
994         loseSoundClip = Applet.newAudioClip(loseSoundUrl);
995         loseSoundClip.play();
996         pause();
997         JOptionPane.showMessageDialog(null, "You Lose!\n"
998                 + "Score: "+ jTFScoreText.getText() + "\n \n",
999                 "Try next time!",
1000                JOptionPane.INFORMATION_MESSAGE,
1001                loseIcon);
1002        reset();
1003    }
```

Figure 4.20 – gameOver Method

The aboutHelp method is called when the About menu item is clicked from the help menu. This displays a message box showing some information about the program.

```
1005 //The about description JOptionPane method
1006    private void aboutHelp() {
1007        JOptionPane.showMessageDialog(null,
1008                "CBallMaze - Ball Maze Application \n"
1009                + "Version 1.0 \n"
1010                + "By Diwas Lamsal \n",
1011                "About CBallMaze", 1, null);
1012    }
```

Figure 4.21 – aboutHelp method

The reset button exits and creates a new instance of the game, setting all the conditions to the default opening state.

```
1014 //Exit and restart the game
1015    private void reset() {
1016        if (timerForRun!=null && timerForRun.isRunning())timerForRun.stop();
1017        if (timerForBall!=null && timerForBall.isRunning())timerForBall.stop();
1018        dispose();
1019        new CBallMaze();
1020    }
```

Figure 4.22 – reset method

The startTimer method starts the digital timer. This method is set in a timer in the run method. The timer starts whenever the run button is clicked.


```
1023 //Start Digital Timer (This happens when the run button is clicked).
1024     private void startTimer() {
1025         currentSeconds = Integer.parseInt(timerSeconds.getText());
1026         currentMinutes = Integer.parseInt(timerMinutes.getText());
1027         currentHours = Integer.parseInt(timerHours.getText());
1028 
1029         nextSeconds = currentSeconds + 1;
1030         if(nextSeconds<10)timerSeconds.setText("0" + String.valueOf(nextSeconds));
1031         else timerSeconds.setText(String.valueOf(nextSeconds));
1032 
1033         if(nextSeconds>59) {
1034             nextSeconds = 0;
1035             timerSeconds.setText("0" + String.valueOf(nextSeconds));
1036 
1037             nextMinutes = currentMinutes + 1;
1038             if(nextMinutes<10)timerMinutes.setText("0" + String.valueOf(nextMinutes));
1039             else timerMinutes.setText(String.valueOf(nextMinutes));
1040         }
1041 
1042         if(nextMinutes>59) {
1043             nextMinutes = 0;
1044             timerMinutes.setText("0" + String.valueOf(nextMinutes));
1045 
1046             nextHours = currentHours + 1;
1047             if(nextHours <10)timerHours .setText("0" + String.valueOf(nextHours));
1048             else timerHours .setText(String.valueOf(nextHours));
```

Figure 4.23 – startTimer method

The action listener for the CBallMaze. This calls different methods or performs different actions according to the button clicked.

```
1053 //Different action listener events for different buttons or menu items
1054     public void actionPerformed(ActionEvent click){
1055         if (click.getActionCommand().equals("Quit")||click.getActionComm
1056             System.exit(0);
1057         }
1058         if (click.getActionCommand().equals("Act")) {
1059             moveBall();
1060         }
1061         if (click.getActionCommand().equals("Run")) {
1062             run();
1063         }
1064         if (click.getActionCommand().equals("Pause")) {
1065             pause();
1066         }
1067         if(click.getActionCommand().equals("Reset")){
1068             reset();
1069         }
1070 
1071         if(click.getActionCommand().equals("<")) {
1072             moveBallLeft();
1073         }
1074 
1075         if(click.getActionCommand().equals(">")) {
1076             moveBallRight();
1077         }
1078 
1079         if(click.getActionCommand().equals("v")) {
```

Figure 4.24 – Action Listener

The keylistener key press event gets the keyboard inputs (arrow keys) to move the ball in the required direction. This is especially used in the option 3 to move the ball.

```
1116 //KeyListener key press event for up down left right keys
1117 //The ball moves with these only when the option 3 is selected.
1118     public void keyPressed(KeyEvent e) {
1119         if (jTFOptionText.getText().equals("3") && jBRunBtn.getText().equals("Pause")){
1120             if (e.getKeyCode()==37) {
1121                 moveBallLeft();
1122                 jBRunBtn.grabFocus();
1123             }
1124             if (e.getKeyCode()==38) {
1125                 moveBallUp();
1126                 jBRunBtn.grabFocus();
1127             }
1128             if (e.getKeyCode()==39) {
1129                 moveBallRight();
1130                 jBRunBtn.grabFocus();
1131             }
1132             if (e.getKeyCode()==40) {
1133                 moveBallDown();
1134                 jBRunBtn.grabFocus();
1135             }
1136         }
1137     }
1138
```

Figure 4.5 – Key Listener

The state changed event detects the change in slider value and sets the ball's speed accordingly. This has been implemented in the option 1 and option 3 configuration. The slider has been disabled in the option 3.

```
1146 //State Change even for the slider. The timer values are set with this event
1147     public void stateChanged(ChangeEvent e){
1148         if (jBRunBtn.getText().equals("Pause")) {
1149             int timeGap = 1501 - slider.getValue();
1150             if (timerForBall.isRunning()) {
1151                 timerForBall.setDelay(timeGap);
1152             }
1153         }
1154     }
1155 }
```

Figure 4.26 – Slider Change

5. TESTING

In a programming project, documented or not, testing is an important process to find out whether or not the written code works. Various tests have been performed throughout the coding stage before finally reaching the end of the solution. Even so, further testing was required to find out what works well and what does not. During the code writing phase, the tests mainly included the direct effects on the application across different events and the console results. For example: - If a block of code was not running or functioning correctly, a console message would be added in the same block of code to find out where the debugging was required. Also, the variable values could be displayed in the console to know how they are being manipulated across multiple events. Besides these, the console also displays the errors generated during the runtime.

This section includes various test plans which display the expected and actual results from the tests performed. The latter part shows the screenshots of the tests, followed by how the bugs discovered during the test phase were corrected at the end.

Test Plan for GUI Results (Additional Functionalities not included):

	The Test Performed	Expected Result	Actual Result
01.	Starting the application	The application opens up to the set size 775x650 and appears across the center of the screen.	Same as the expected result. [Figure 5.1]
02.	Attempting resize on the application.	The application cannot be resized.	Same as the expected result.
03.	Dragging the application away from the center	The application should be draggable.	Same as the expected result.
04.	Clicking the option text-field and editing the text as “Test”	The text-field should not be editable	The text-field was editable and the text could be changed. [Figure 5.2]
05.	Clicking the timer text-field and editing the text as “Test”	The text-field should not be editable	The text-field was editable and the text could be changed. [Figure 5.2]
06.	Clicking the menus from the menu-bar.	The menu-items for the respective menu-bars should dropdown from the parent Menu.	Same as the expected result. [Figure 5.3]

07.	Clicking the exit menu-item from the Scenario menu.	The application should exit.	Same as the expected result.
08.	Clicking the close menu-item from the Scenario menu.	The application should exit.	The application does not exit.

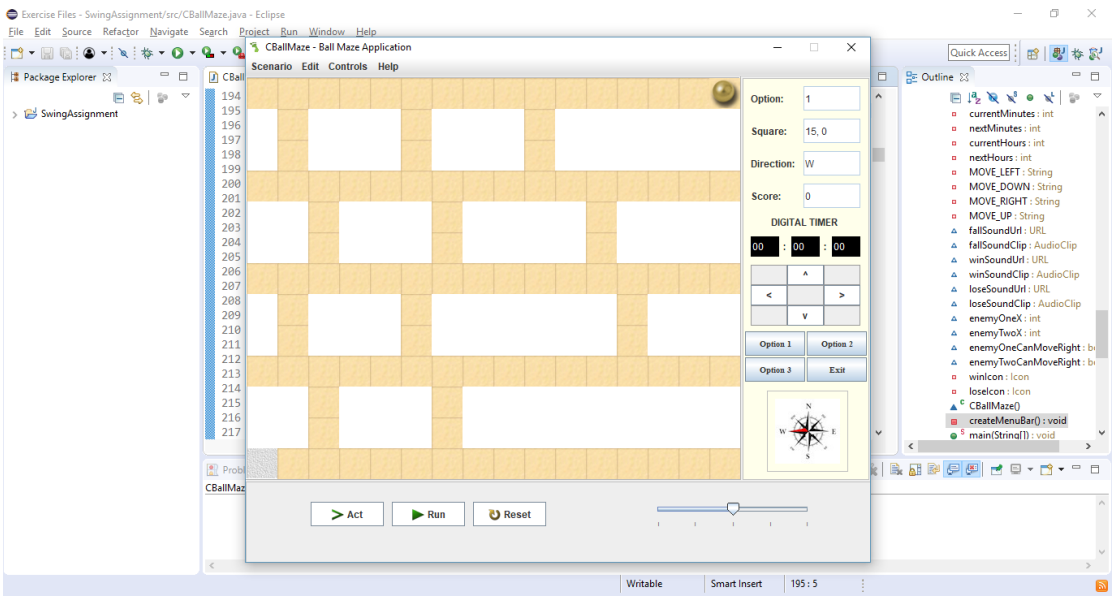


Figure 5.1 – Starting the Application

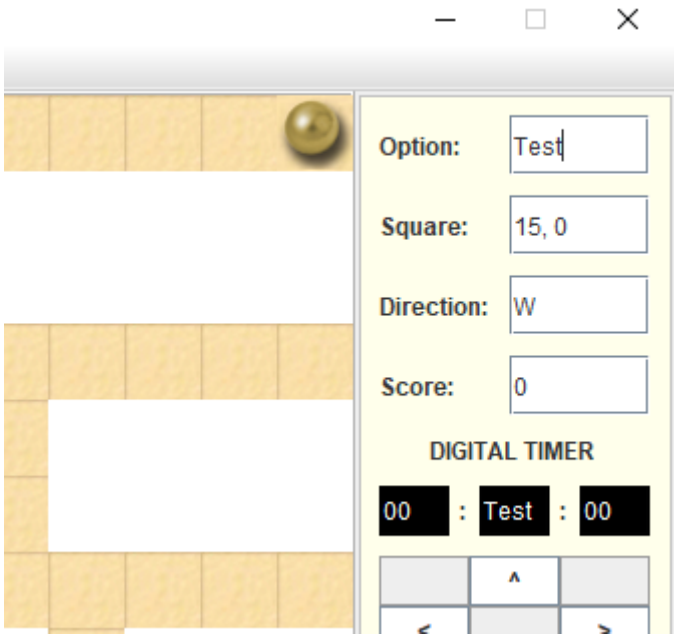


Figure 5.2 – Editing text-fields

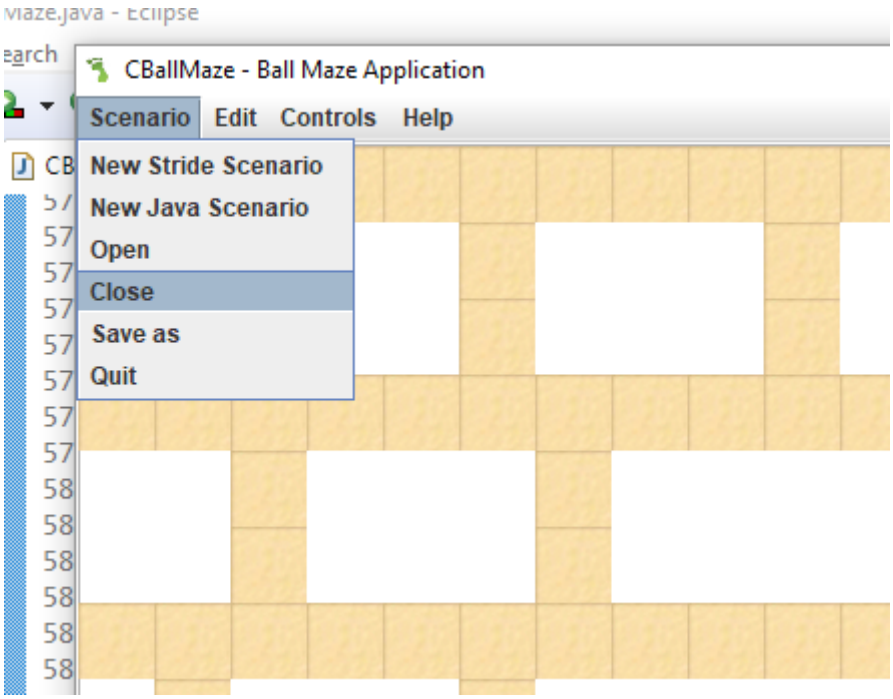


Figure 5.3 – Clicking the menu

Test Plan for Option 1:

	The Test Performed	Expected Result	Actual Result
01.	Run button is clicked	The digital timer should start and the ball should start to move continuously at regular intervals from the start position to the end grey-block. The text of the run button should be changed to "Pause" and the icon should be changed to pause icon. The direction and square text-field values also change according to the position and movement of the ball.	Same as the expected result. [Figure 5.4]
02.	The ball reaches the end block after the run button simulation.	Winning sound effect is played, the text and icon of the run button are changed back to normal, the digital timer is paused.	Same as the expected result. [Figure 5.5]

03.	The run button is clicked once again after the ball reaches the end.	The scenario should reset and the ball should start from initial position.	No changes take place. Not as expected result.
04.	The Act button is clicked.	The act button should run the run button simulation one step each time the act button is clicked.	Same as the expected result.
05.	The slider is dragged towards right and the run button is clicked.	The ball should move faster.	No changes take place. Not as expected result.
06.	The run button is clicked, and when the ball starts to move, the slider is dragged towards the right.	The ball should move faster.	Same as the expected result. Dragging the slider to the rightmost corner would complete the movement within 1 second in the digital timer.
07.	The run button is clicked, and when the ball starts to move, the slider is dragged towards the left.	The ball should move slower.	Same as the expected result. Dragging the slider to the leftmost corner would complete the movement within 39 second in the digital timer.
08.	The reset button is clicked.	The application should restart at the default opening state.	Same as the expected result.
09.	The exit button is clicked.	The application should exit.	Same as the expected result.
10.	The direction buttons are pressed.	The ball should move towards the respective direction if a sandblock exists.	Same as the expected result. [Figure 5.6]
11.	The direction buttons are pressed during the run button simulation.	The ball should not move while the run button simulation is occurring.	Not as expected, the ball moves when the arrow keys are pressed.

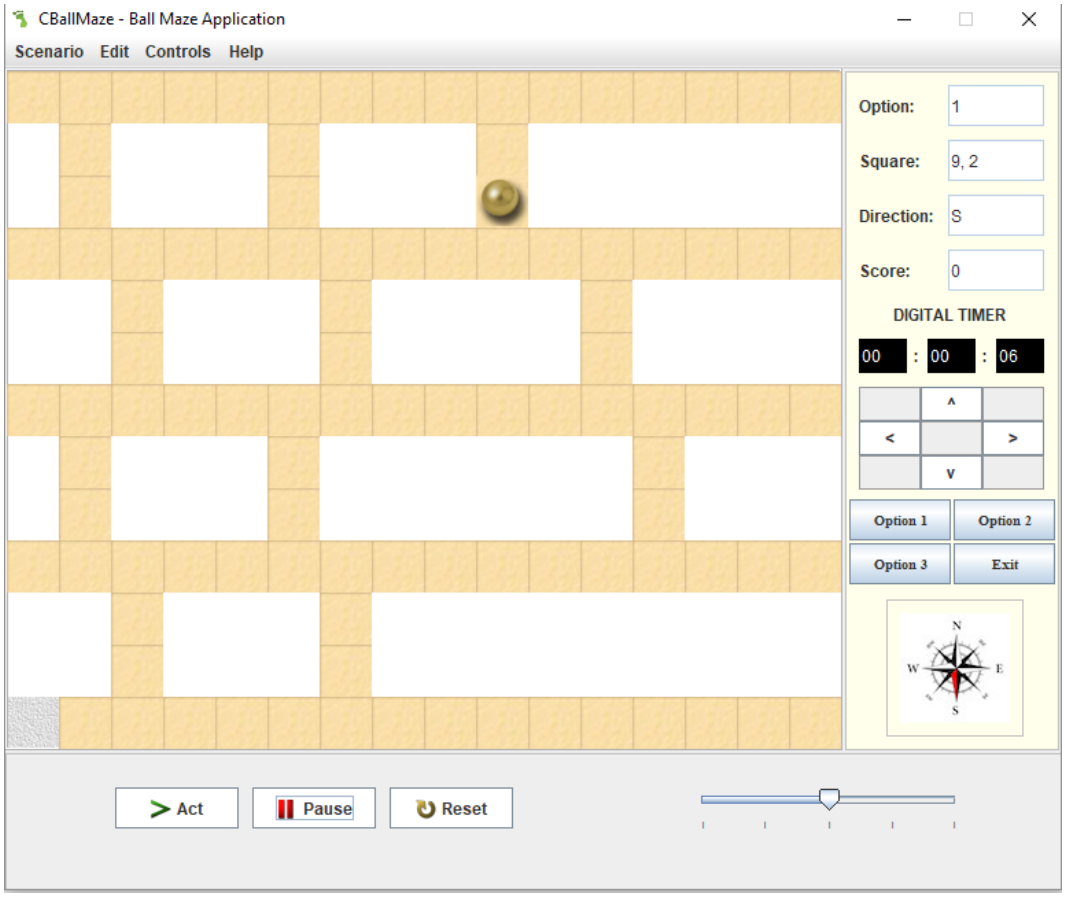


Figure 5.4 – Run Button Simulation

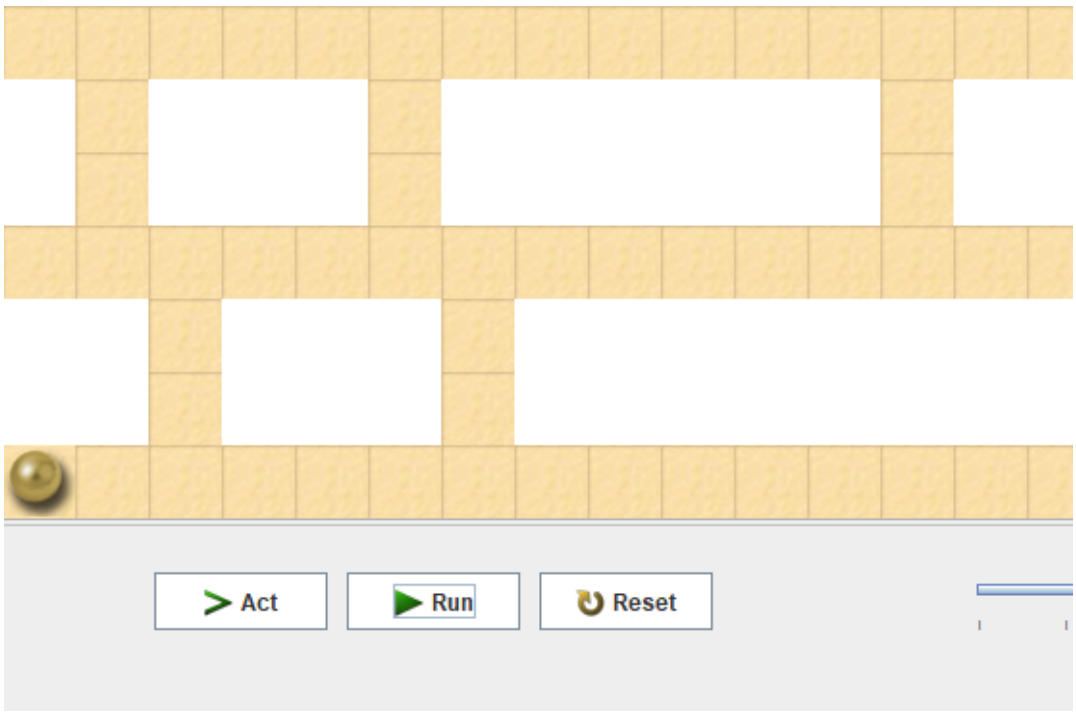


Figure 5.5 – Ball Reached the end

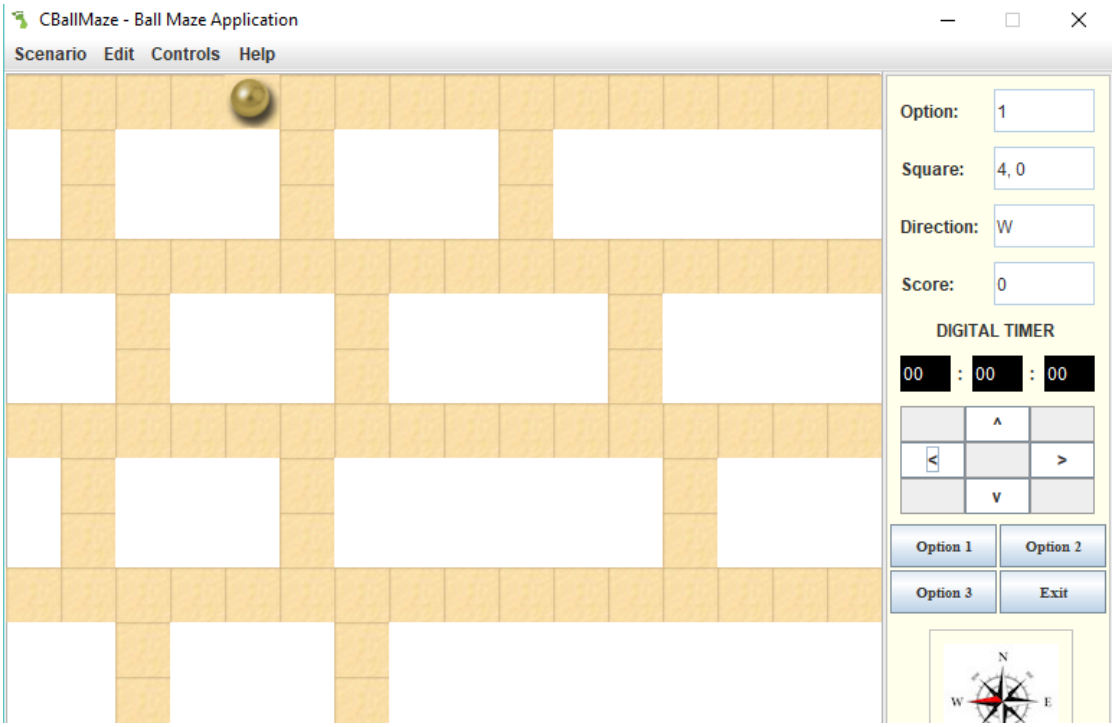


Figure 5.6 – Clicking the Left Arrow button

Test Plan for Option 2:

	The Test Performed	Expected Result	Actual Result
01.	Option 2 button is clicked.	The option text-field text would be changed to 2.	Same as expected. [Figure 5.6]
02.	Run button is clicked	Everything else same as the option 1, but the ball should fall (drop) down when it has a movable block in the South or the downward direction. A sound effect should be played as this happens.	Same as the expected result. [Figure 5.4]

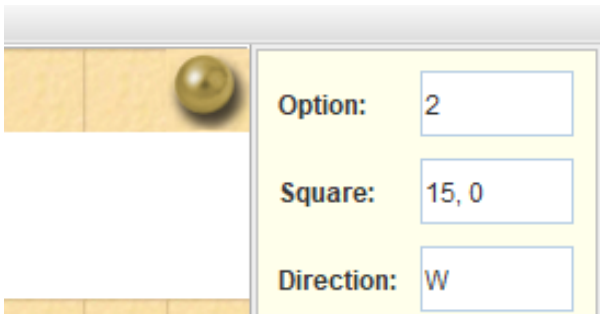


Figure 5.7 – Option 2 selection

Test Plan for Option 3:

	The Test Performed	Expected Result	Actual Result
01.	Option 3 is clicked	Like in the Option 2, the text-field text would be changed to 3. Additionally, a different kind of layout can be seen where there are two skulls on the sandblocks and five cherries. The slider would be disabled.	Same as expected. [Figure 5.7]
02.	Run button is clicked	Nothing happens to the ball itself, the timer starts, the enemies start moving from end to end	Same as expected. [Figure 5.8]
03.	Arrow keys are pressed without clicking the run button	Nothing happens to the ball	Same as expected.
04.	Arrow keys are pressed after the run button is clicked – "Pause" is the text on the run button	The ball moves to the direction that arrow key represents if a sandblock exists in the direction. This could also be done by using the arrow buttons. Like in the option 2, the ball drops when it can move down.	Same as expected. [Figure 5.9]
05.	Ball touches the cherry	The cherry disappears, 10 is added to the score text-field.	Same as expected. [Figure 5.10]
06.	Enemy touches the ball	A message is displayed saying game over and the game resets after "OK" button is clicked.	Same as expected. [Figure 5.11]
07.	Option 3 is clicked twice and then once again	The game is reset and option 1 is selected after the button is clicked twice, after clicking the button third time, the option 3 layout would reappear wherein the cherries should be placed in a	Not as expected, only two of the cherries are randomly changing positions. [Figure 5.12]

		random position, different from before.	
08.	Option 3 is clicked, followed by another option, i.e. 1 or 2, and option 3 again	The game is reset and option 1 is selected.	Not as expected, the option 3 layout adds two more cherries to the existing layout without resetting it. [Figure 5.13]
09.	Same as test 07 but is repeated 10 times	The game is reset and option 1 is selected.	Not as expected, the option 3 layout adds more cherries which sometimes may overlap to the existing cherries in the layout, without resetting it. [Figure 5.14]
10.	Option 3 is selected, option 1 is selected and then run button is clicked	The game should be reset and option 1 should be selected.	Not as expected, the ball starts to move from the start position to the end position. The other conditions are the same as in option 3, i.e. the game is over when enemy touches the ball.
11.	Ball reaches the grey sand block	A message is displayed saying you win and the game resets after “OK” button is clicked.	Same as expected. [Figure 5.15]

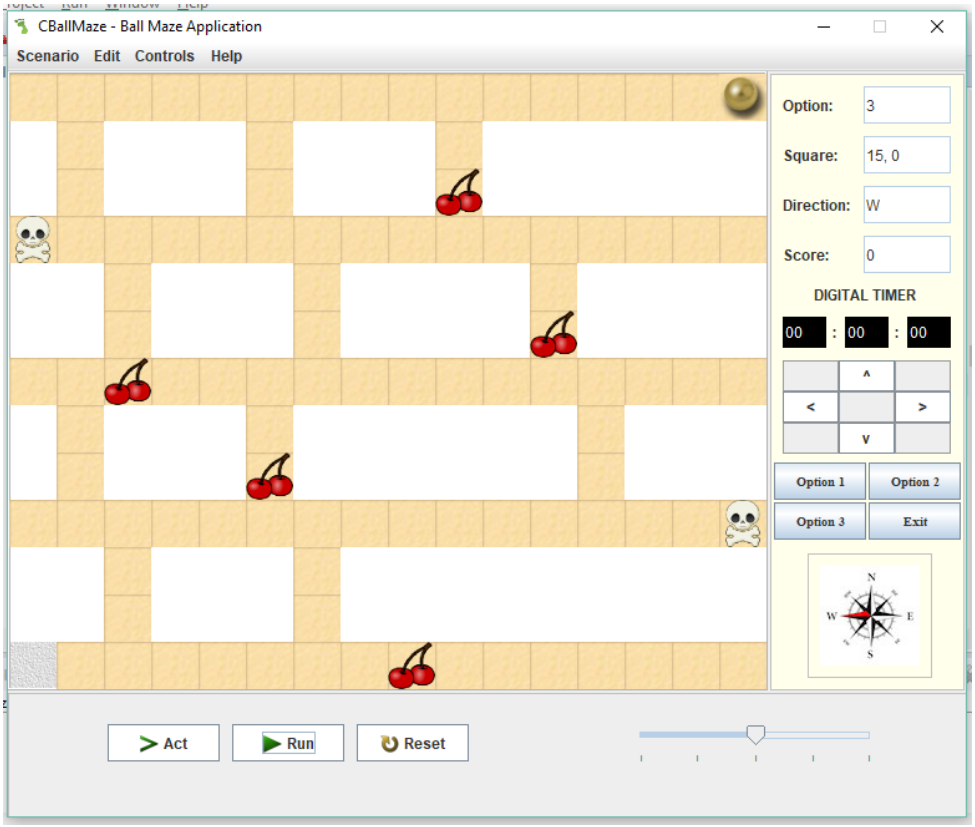


Figure 5.8 – Option 3 layout

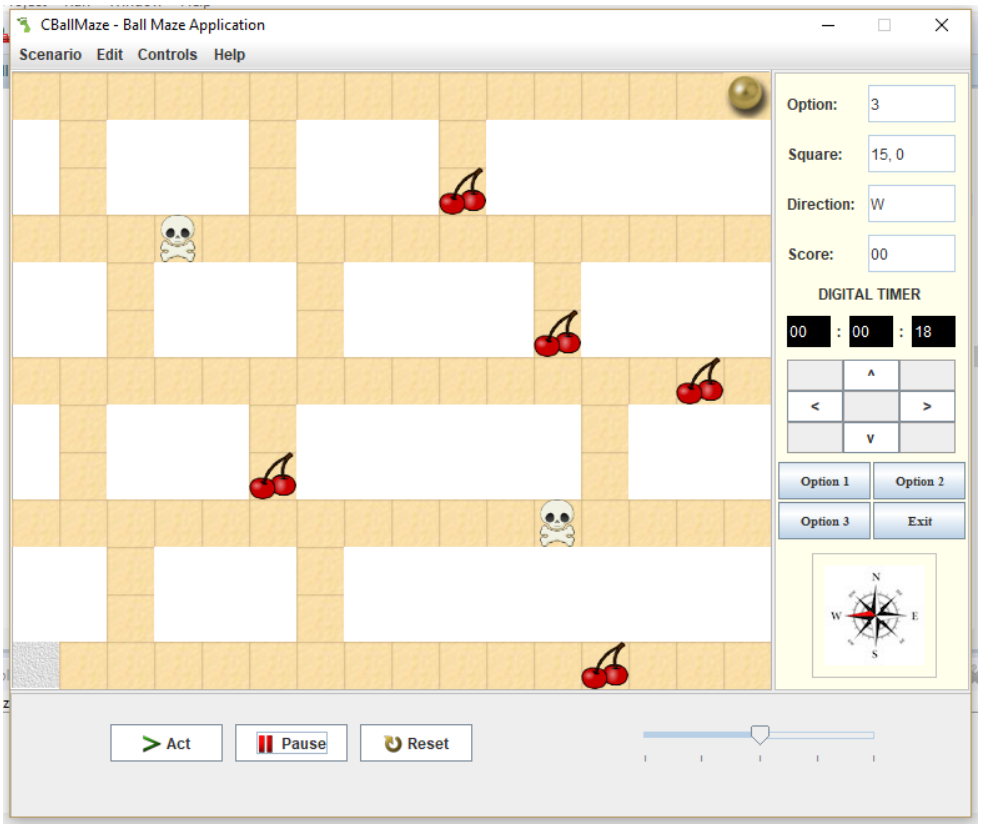


Figure 5.9 – Option 3 run button clicked

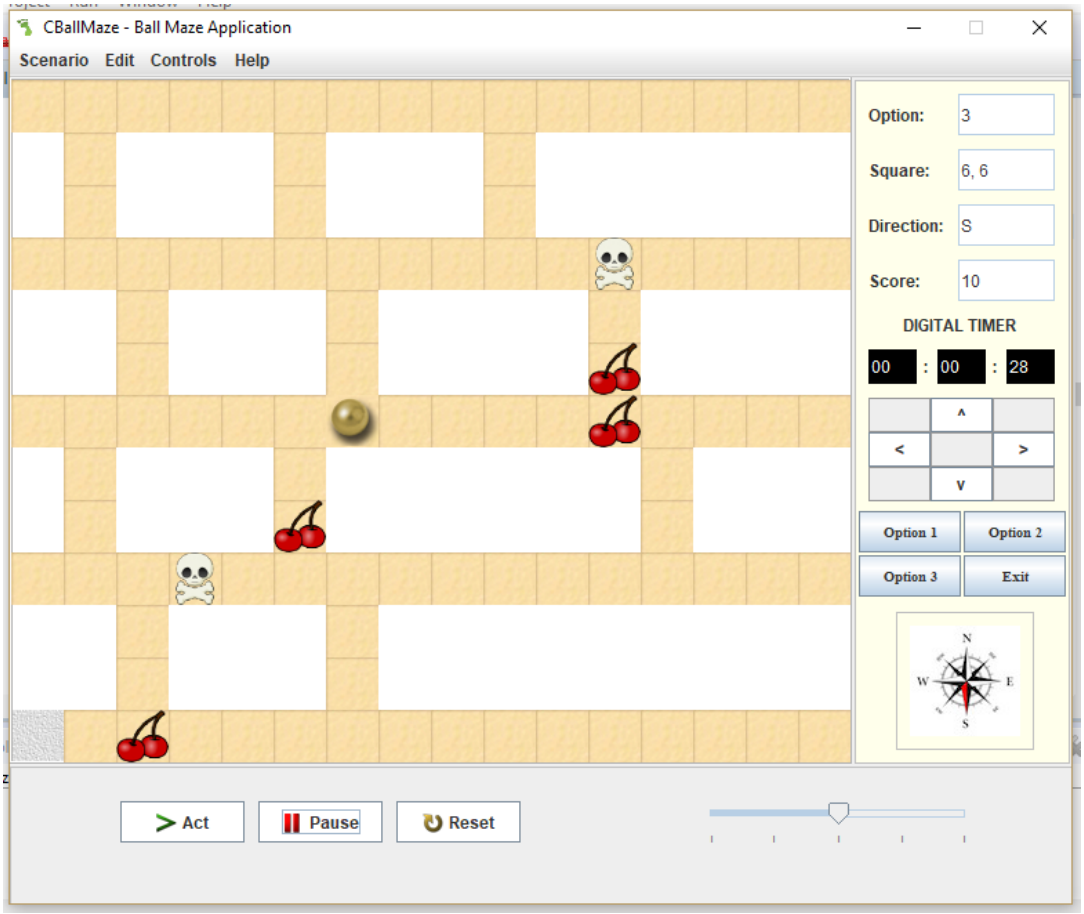


Figure 5.10 – Option 3 Arrow key pressed

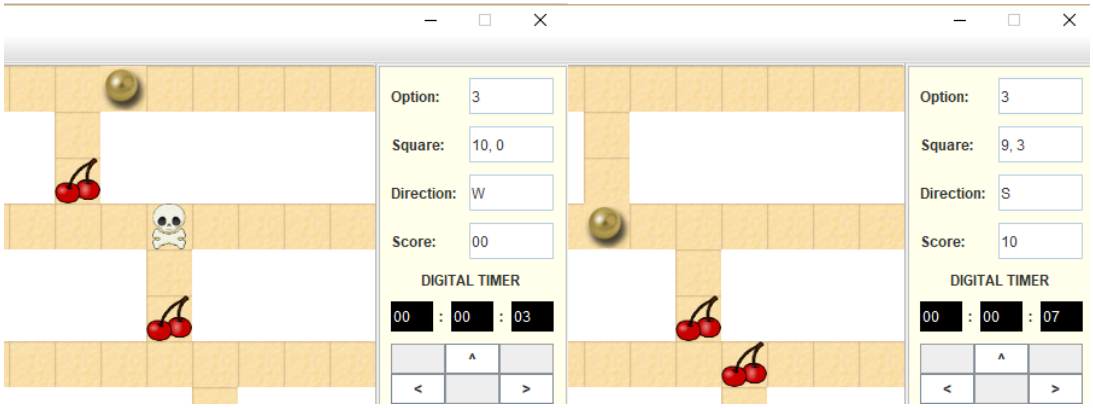


Figure 5.11 – Eating cherry

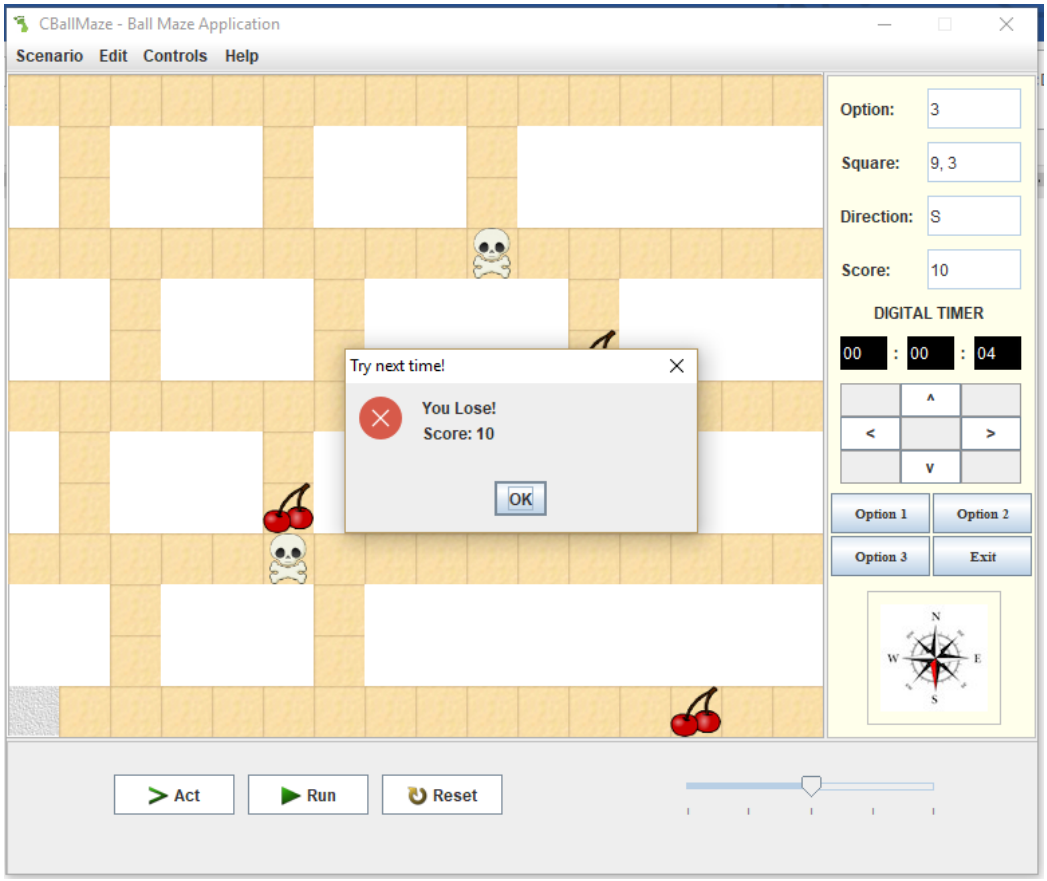


Figure 5.12 – Game Over

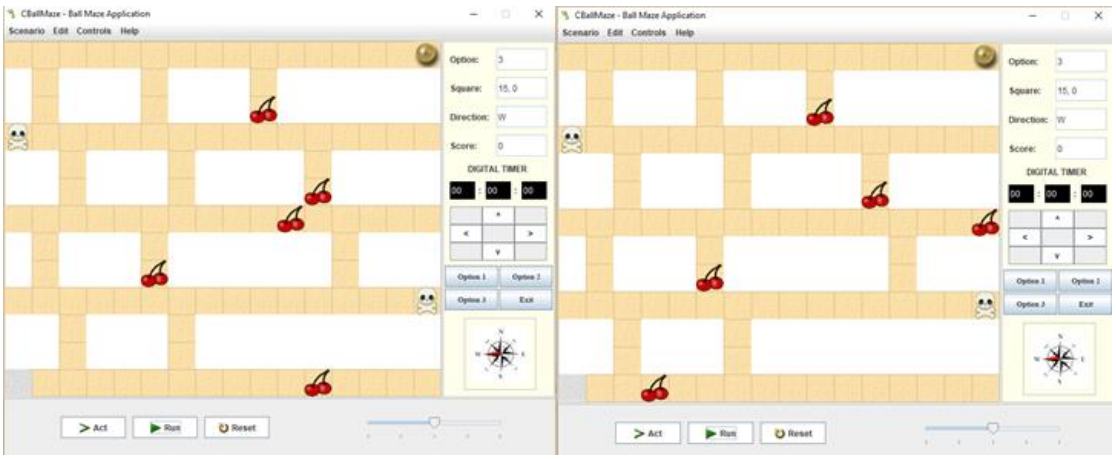


Figure 5.13 – Clicking option 3 once vs thrice

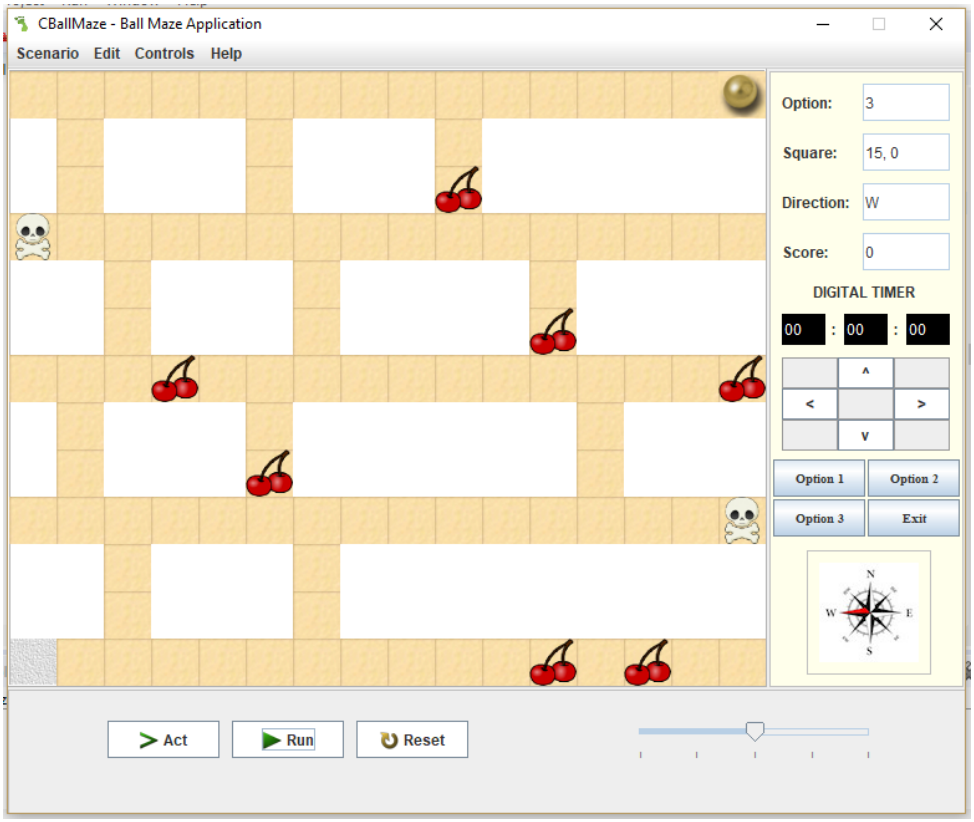


Figure 5.14 – Clicking option 3, 1 and 3 again in order

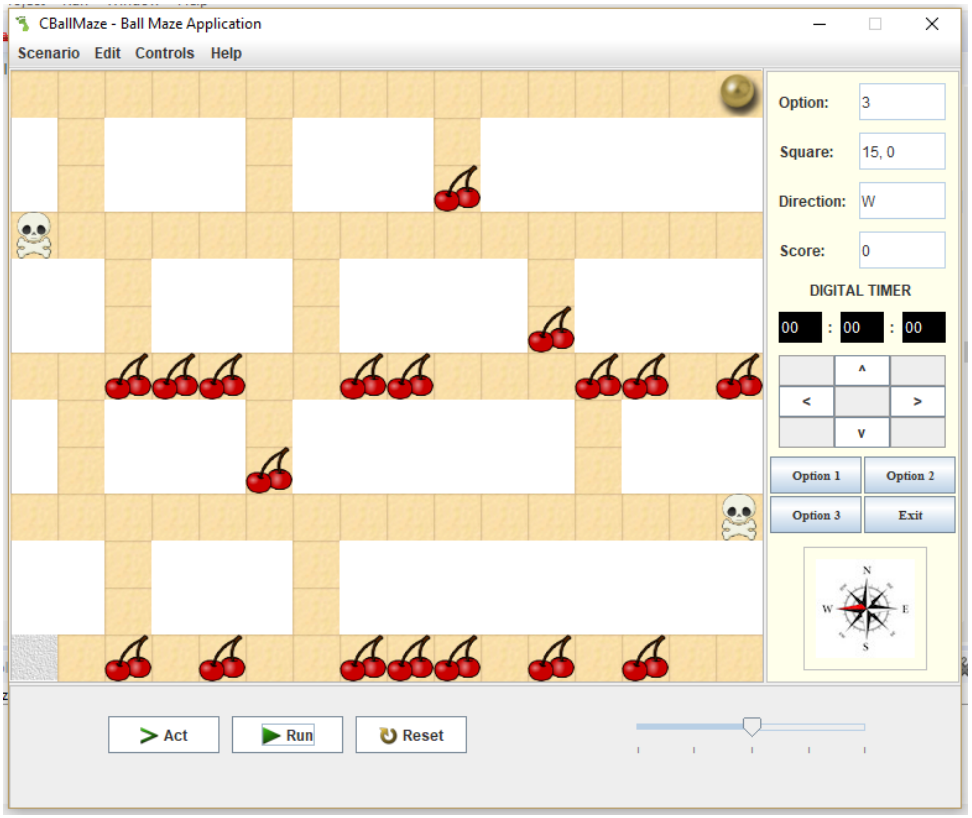


Figure 5.15 – Clicking option 3, 1 and 3 in order 10 times

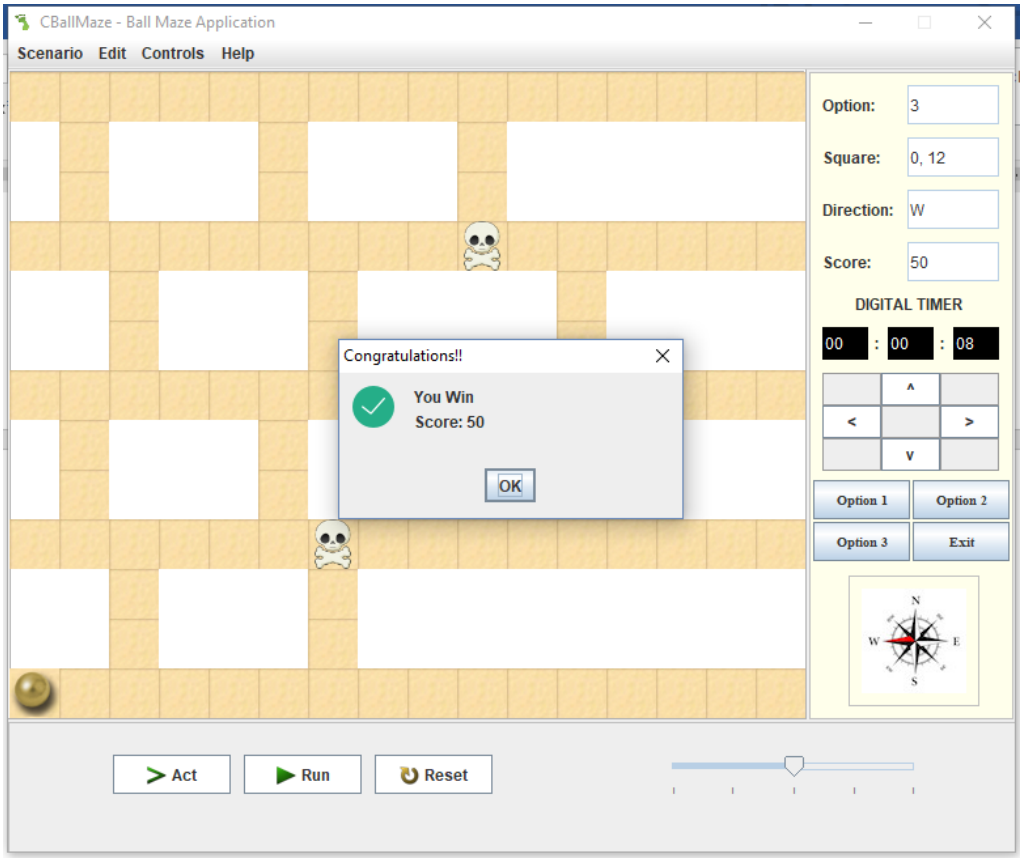


Figure 5.16 – Game won

Some notable bugs:

Some notable bugs found during the testing phase include:

- Editable text-fields of the digital timer and the option, square direction.
- Close menu-item not closing the application.
- The slider value before run button is clicked is not executed.
- The option 3 does not generate random positions for all the cherries, but only so for two of them.
- When the option 3 is clicked, followed by clicking option 1 and then again clicking the option 3 would bring up more cherries in the maze but not reset the layout.

Some of these bugs have been fixed in the newer versions of the application. Also, some improvements, weaknesses and strengths of the solution have been discussed in the **6. CONCLUSION AND RECOMMENDATIONS** section.

6. CONCLUSION AND RECOMMENDATIONS

6.1 Evaluation

6.1.1 Aims and Objectives

The brief provides a fair set of tasks to work on. This section covers the provided tasks in the brief along with the rules.

The Graphical User Interface requirements (Taken from the brief):

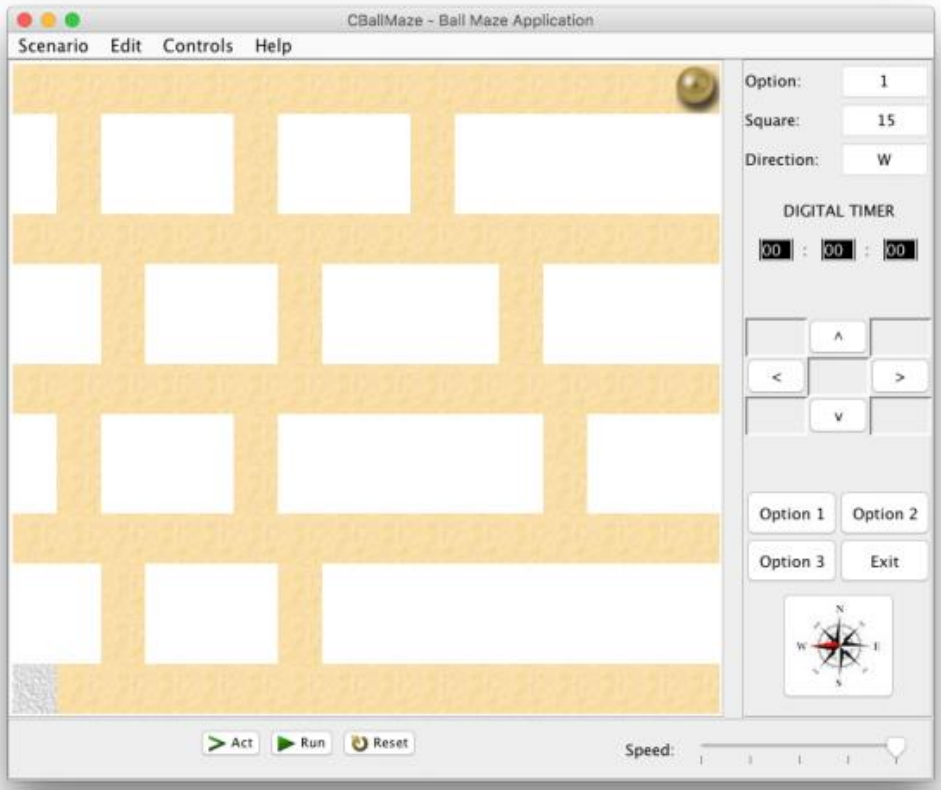


Figure 6.1 – Provided GUI Screen

Essential Functionality:

- 13 x 16 grid of **JButtons** or Icons.
- 4 **JButtons** for the game options 'Option 1, Option 2, Option 3' and 'Exit'.
- 3 **JButtons** for 'Act', 'Run' and 'Reset'.
- 9 **JButtons** for 'Forward >', 'Backwards <', 'Up ^', 'Down v' should move the ball in the appropriate direction by one square for each press (plus 5 blank).
- The compass icon (**JButton**) should illustrate the current direction for the ball.
- 3 **JLabels** for 'Option', 'Square' and 'Direction'.
- 3 **JTextFields** for the current 'Option', Location/'Square' and 'Direction' of the ball. Use the square identification method e.g. 0 to 207 and N, E etc.
- 3 **JLabels** for the '**DIGITAL TIMER** and the two :', with 3 **JTextFields** for the hours, minutes and seconds.
- Create a **JFrame** application, which opens to the set size (775 * 650).

- **JFrame** title set as "*CBallMaze – Ball Maze Application*".

Additional or the Advanced Functionality and Complexity:

- Application icon for the **JFrame** used (Windows only).
- Application dock icon.
- The 'Run' **JButton** should show the ball moving between the continuously from the initial position (Option 1 – default opening state – ball top right-hand corner) to the end position at the grey square/tile (bottom left-hand corner.
- The 'Reset' **JButton** should clear/reset the application to its starting/default opening state.
- The 'Act' **JButton** should step through the above 'Run' sequence one move at a time.
- Discuss and implement the different options for the 3 configurations.
- The 'Option 1, Option 2, Option 3' **JButton**s should display different tile/object configurations/locations.
- A **JMenuBar** could be included with **JMenu**s for the *Scenario, Edit, Controls* and *Help,* which include **JMenuItem**s of *Exit (Scenario), Help Topic* and *About (Help).*
- Additional **JButton**s may be used to improve the applications usability e.g. ball movement – in random/predefined direction, jump objects/obstacles in Option 2 or 3 etc.
- The ball drops down the maze.
- A sound effect is heard when the ball drops down to the next level.
- Create a **JFrame** application, which is not resizable.
- Create a **JFrame** application, which centers itself on the monitor.
- Discuss the possibilities for incorporating intelligence/checks for whether moves are valid.
- Digital Timer should start and stop when 'Run' is pressed and stopped when a ball gets to the end.
- Implement intelligence/checks for whether moves are valid.
- A **moveBall()** method should be used to solve the problem. The **moveBall()** method should include **move(MOVE_LEFT), move(MOVE_RIGHT), move(MOVE_UP), move(MOVE_DOWN)** methods (see below).

```
public void moveBall()
{move(MOVE_LEFT);
        ..................
         move(MOVE_RIGHT);
}
```

The source code file containing the **main()** method and the compiled byte code **class** files should be named as follows: **CBallMaze.java** & **CBallMaze.class**

The Rules (Taken from the brief):

Rules (Basic) Create a simulation of the ball moving around the pitch, where:

- The ball must only travel when on the 'sand'-coloured blocks otherwise it should not move.
- The ball must move one whole 'sand'-coloured blocks at a time every time a movement key is pressed - via a direction button (<, > v ^)) - (when movement is possible).
- Must use the scenario provided.
- Must stop when it the ball reaches the grey block at the end of the maze.
- The basic solution must be completed using the 'act' button (accessing the **moveBall()** method within the **CBallMaze.class**).

Rules (Intermediate and advanced) Create a simulation of a ball moving around the maze, where:

Rules (Intermediate)

- Whilst maintaining the features of the basic solution add the following
- When there is a block below the block the ball is automatically go down. In other words, if the ball can drop it 'falls' down until a white space is below it.
- Add a sound effect when the ball drops.
- The ball must not fall into the white spaces.

Rules (Advanced)

- For higher grades on the solution part of the assignment see the marking scheme/rubric. You must NOT change the layout and all changes should still meet the criteria of **Rules (Basic**).

Besides these rules, the brief asks to follow different conventions and standards used in Java, and asks for excessive usage of comments in the codes.

6.1.2 What has been Completed

Essential Functionality:

- 13 x 16 grid of **JButtons** or Icons.
- 4 **JButtons** for the game options 'Option 1, Option 2, Option 3' and 'Exit'.
- 3 **JButtons** for 'Act', 'Run' and 'Reset'.
- 9 **JButtons** for 'Forward >', 'Backwards <', 'Up ^', 'Down v' should move the ball in the appropriate direction by one square for each press (plus 5 blank).
- The compass icon (**JButton**) should illustrate the current direction for the ball.
- 3 **JLabels** for 'Option', 'Square' and 'Direction'.
- 3 **JTextFields** for the current 'Option', Location/'Square' and 'Direction' of the ball. Use the square identification method e.g. 0 to 207 and N, E etc.
- 3 **JLabels** for the '**DIGITAL TIMER** and the two :', with 3 **JTextFields** for the hours, minutes and seconds.
- Create a **JFrame** application, which opens to the set size (775 * 650).
- **JFrame** title set as "*CBallMaze – Ball Maze Application*".

Additional or the Advanced Functionality and Complexity:

- Application icon for the **JFrame** used (Windows only).
- Application dock icon.
- The 'Run' **JButton** should show the ball moving between the continuously from the initial position (Option 1 – default opening state – ball top right-hand corner) to the end position at the grey square/tile (bottom left-hand corner.
- The 'Reset' **JButton** should clear/reset the application to its starting/default opening state.
- The 'Act' **JButton** should step through the above 'Run' sequence one move at a time.
- Discuss and implement the different options for the 3 configurations.
- The 'Option 1, Option 2, Option 3' **JButton**s should display different tile/object configurations/locations.
- A **JMenuBar** could be included with **JMenu**s for the *Scenario, Edit, Controls* and *Help,* which include **JMenuItem**s of *Exit (Scenario), Help Topic* and *About (Help).*
- Additional **JButton**s may be used to improve the applications usability e.g. ball movement – in random/predefined direction, jump objects/obstacles in Option 2 or 3 etc.
- The ball drops down the maze.
- A sound effect is heard when the ball drops down to the next level.
- Create a **JFrame** application, which is not resizable.
- Create a **JFrame** application, which centers itself on the monitor.
- Discuss the possibilities for incorporating intelligence/checks for whether moves are valid.
- Digital Timer should start and stop when 'Run' is pressed and stopped when a ball gets to the end.
- Implement intelligence/checks for whether moves are valid.
- A **moveBall()** method should be used to solve the problem. The **moveBall()** method should include **move(MOVE_LEFT), move(MOVE_RIGHT), move(MOVE_UP), move(MOVE_DOWN)** methods (see below).

```
public void moveBall()
{move(MOVE_LEFT);
        ..................
        move(MOVE_RIGHT);
}
```

The source code file containing the **main()** method and the compiled byte code **class** files should be named as follows: **CBallMaze.java** & **CBallMaze.class**

The Rules (Taken from the brief):

Rules (Basic) Create a simulation of the ball moving around the pitch, where:

- The ball must only travel when on the ‘sand’-coloured blocks otherwise it should not move.
- The ball must move one whole ‘sand’-coloured blocks at a time every time a movement key is pressed - via a direction button (<, > v ^)) - (when movement is possible).
- Must use the scenario provided.
- Must stop when it the ball reaches the grey block at the end of the maze.
- The basic solution must be completed using the ‘act’ button (accessing the **moveBall()** method within the **CBallMaze.class**).

Rules (Intermediate and advanced) Create a simulation of a ball moving around the maze, where:

Rules (Intermediate)

- Whilst maintaining the features of the basic solution add the following
- When there is a block below the block the ball is automatically go down. In other words, if the ball can drop it ‘falls’ down until a white space is below it.
- Add a sound effect when the ball drops.
- The ball must not fall into the white spaces.

Rules (Advanced)

- For higher grades on the solution part of the assignment see the marking scheme/rubric. You must NOT change the layout and all changes should still meet the criteria of **Rules (Basic**).

Besides these rules, the brief asks to follow different conventions and standards used in Java, and asks for excessive usage of comments in the codes.

6.1.3 What has not been completed:

- Additional JButton’s may be used to improve the applications usability e.g. ball movement – in random/predefined direction, jump objects/obstacles in Option 2 or 3 etc.

This was the functionality not included, and this is because, other extra features have been added instead which add to the strengths of the solution. Except the one mentioned, all the features listed in the brief have been added to make a complete solution.

6.1.4 Strengths of the Solution:

- Completed all the points in the brief except the one mentioned.
- Enemies added to the game to make it more interactive.
- The player can eat cherries and gain score.
- Suitable sound clips for different events.
- Different configurations for option 1, option 2 and option 3 where option 1 would refer to basic rules, option 2 would refer to intermediate rules and the option 3 would refer to advanced rules.
- The ball can be moved using keyboard arrow keys. (Use of keylistener, not just actionlistener)
- A complete JMenuBar with many menus and items.
- Windows open up when game is lost or won.
- A window shows some information about the application when the about menu-item from help menu is clicked.

6.1.5 Weaknesses of the Solution:

- Existence of some minor bugs that were not corrected due to the unavailability of time.
- For the run button simulation, the ball's movement is predefined as in it only goes left and down. This logic would not work if the maze had a different kind of structure.
- The code for creating the maze panel has lot of redundant codes. These could have been well organized and placed inside methods, calling them when necessary.
- The consecutive timers running everywhere causes lags or delays at times.
- The option 2 has win sound played in a loop which creates noise. (Could not be debugged)
- No proper collision detection technique used to detect enemy-ball and ball-cherry collisions. When the collision detection was used earlier, many bugs were introduced and thus has been replaced by a separate technique. But this is not the solution for a program to be extended further.

6.2 Future Work and Improvements:

- The game can be designed to have lesser repeating codes and have all the bugs fixed.
- New levels can be added when the first option 3 round is complete. For example: - more enemies, key to enable the grey block, time limit to complete the game, etc.
- Implementation of Artificial Intelligence on the run button simulation where the ball could determine its path no matter where it is.
- Proper way to detect collisions.
- Use of threads would divide the work of timers and thus could be utilized.

6.3 Difference in Approach Next Time

This solution was produced as a first experience of this kind of project, this would be used to come up with a more efficient way to write the program, also correcting the bugs that were seen on the first round. The author would basically divide the task as much as possible, for example: - having separate classes for each panel and adding them into the main frame. This kind of approach would greatly increase efficiency in debugging and managing the codes. Also, the program would be designed to be more interactive and improved version of the current solution.

This completes the CSY 1020 Problem Solving and Programming – Term II assignment. The major GUI design principles and not writing codes haphazardly have been the greatest achievements from this term's coursework. Also, many other important principles of java have been known which would be implemented as the course goes on further.

REFERENCES

Adding Image to JOptionPane

Nicholson M. (2012), Add Image to JOptionPane - Stackoverflow. Available from: https://stackoverflow.com/questions/13963392/add-image-to-joptionpane [Accessed: 11/06/2018]

Adding Sound to the Application:

edu4JAVA (2018), Adding Sound to our game - Play sounds using AudioClip. Available from: http://www.edu4java.com/en/game/game7.html [Accessed 09/06/2018]

Game won sound from soundbible.com

Soundbible (2011), Electrical Sweep Sound. Available from: http://soundbible.com/1795-Electrical-Sweep.html [Accessed 21/03/2018]

Game over sound from soundbible.com

Soundbible (2011), Laser Cannon Sound. Available from: http://soundbible.com/1771-Laser-Cannon.html [Accessed 24/03/2018]

Cross Icon Flaticon

Flaticon (2018), Cancel free icon. Available from: https://www.flaticon.com/free-icon/success_148767 [Accessed: 11/06/2018]

Tick Icon Flaticon

Flaticon (2018), Success free icon. Available from: https://www.flaticon.com/free-icon/success_148767 [Accessed: 11/06/2018]

Educational Resources:

Javatpoint:

Javatpoint (2018), Java Swing Tutorial. Available from: https://www.javatpoint.com/java-swing [Accessed: 10/06/2018]

Oracle Tutorials:

Oracle (2018), Trail: Creating a GUI with JFC/Swing Available from: https://docs.oracle.com/javase/tutorial/uiswing/ [Accessed: 10/06/2018]