University of St Andrews

Report Briefing Vector Drawing Application

Student ID 190029087

November 23, 2019

The vector drawing application implements the Model-Delegate (MD) design pattern in that the delegate (*guiDelegate/*) responsible for providing the view is isolated from the model (*model/*). The packages, classes and their functionalities are described below:

Package	Class	Functionality
main/	SimpleSwingMain	Couples the delegate with the model.
guiDelegate/	SimpleGuiDelegate	Controls the view of the application by notifying the model.
model/	SimpleModel	Handles all user queries such as drawing shapes, undo, redo, clear, etc.
model/	DrawingShapes	Abstract parent class whose sub-classes construct shapes for drawing rectangles, triangles, parallelograms, lines, ellipses, hexagons and diamonds.
model/	MyCanvas	Maintains the buffered image whose graphics is used for drawing shapes.
tests/	SimpleModelTest	Hosts the JUnit test cases for testing the functionalities of the model.

The program can be run from *CS5001-p4/src/* as:

```
$ javac main/SimpleSwingMain.java
$ java SimpleSwingMain
```

1 Design

Inspired by the MD design pattern, the delegate notes the input supplied by the user by listening to mouse as well as key events. As soon as the user clicks on the JPanel (or requests an Undo/Redo/Clear operation), the model is triggered for processing the corresponding operation.

2 Features implemented

The model supports following possible actions:

1. **Drawing basic shapes**: If the user's click requests drawing a shape, the model first clears its redo stack to reflect that no redo operations can be carried out. It then initializes a new graphics object coupled to the original buffered image of the drawing canvas followed by either drawing the outline of the shape or filling it with specified color as indicated by the input.
2. **Undo/Redo**: The undo/redo operation implements two stacks whose working can be explained in the following steps:

Step 1: Both the stacks keep track of the states of the buffered image linked to the canvas. If the user wants to draw a shape, the redo stack is cleared and a copy of the buffered image with the shape is added to the undo stack. The clearing of redo stack reflects that the user should no longer be able to retrieve the undone operations once they have chosen to draw a new shape. The first element of the undo stack is always an empty image so that the user gets an empty canvas after having undone the first operation.

Step 2: On requesting each *undo*, the top of the undo stack is pushed to redo stack to keep track of the steps that can be redone. Subsequently, a new canvas is initialized with the buffered image retrieved from the top of the undo stack. On the other hand, if the user requests an undo as a first operation (before drawing any shape), an empty canvas is initialized.

Step 3: On requesting a *redo*, a new canvas is initialized with the buffered image at the top of the redo stack followed by pushing it back to the undo stack. This push takes into account that the user might request the current *redo* operation to be undone again.

3. **Clear**: The clear operation simply resets the image of the current canvas with an empty buffered image of the same dimensions.

4. **Load/Save file**: The load/save operations are performed on the buffered image coupled with the canvas of the JPanel. The operations perform respective read/writes on the file *"vectorDrawing.png"*.

5. **Drawing Squares/Circles**: The rectangle/ellipse buttons additionally listen to a *Shift* key press which locks the aspect ratio to draw squares and circles respectively. The locking of aspect ratio is simulated by fixing the width/radius in the original shapes. In case of circles, the radius is the distance between the current cursor position (i.e. point P2) and the point of first mouse press (i.e. point P1) while P1 being the center. On pressing the shift key again, the mode toggles back to ellipse/rectangle.

6. **Additional Shapes**: The following additional shapes are added:

Parallelogram: Figure 1 shows the idea behind drawing a parallelogram ABCD from the user's first click (i.e., point P1) and the current position of the cursor as the mouse has been dragged (i.e., point P2). Based on the distance between the initial and the final clicks, the coordinates of vertices B, C and D are calculated.[1]

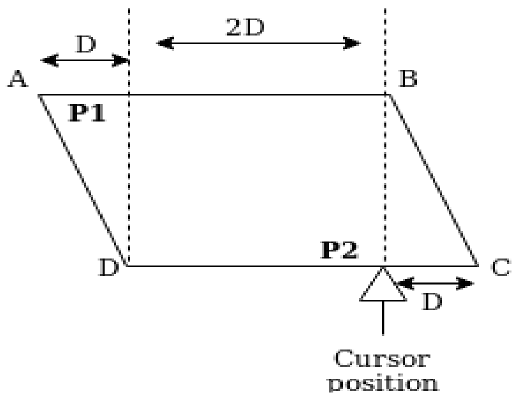

Figure 1. Figure showing the design for constructing a parallelogram.

Triangle: The idea for drawing triangles derives from parallelograms in that the third vertex (i.e. vertex C in figure 2 lies mid-way of the x-coordinate distance between the points P1 and P2.

Hexagon: For drawing a hexagon, the current cursor position is treated as the center as depicted in figure 3. The formula used is a special case of the formulae for calculating vertices of a regular polygon.[2]

Diamond: The diamond is an additional feature whose sides connect the mid-point of an imaginary rectangle with width and height determined by the difference between x and y coordinates of the points P1 and P2 (see figure 4).

[1]The formulae for calculation can be found in the code and has been purposely avoided here for a concise report.
[2]https://stackoverflow.com/questions/3436453/calculate-coordinates-of-a-regular-polygons-vertices

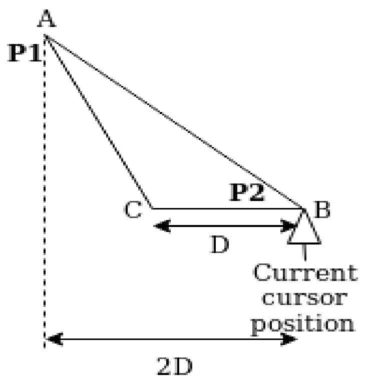

Figure 2. Figure showing the design for constructing a triangle.

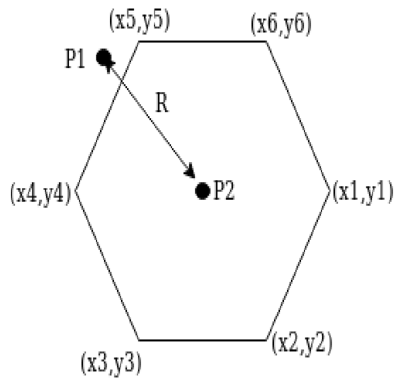

Figure 3. Figure showing the design for hexagon construction: (x1,y1) is calculated from P2 and radius R while each subsequent vertex draws upon the previously computed vertex and R.

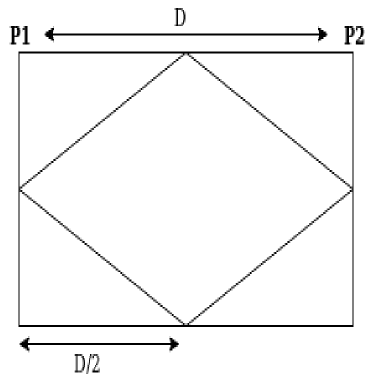

Figure 4. Figure showing the design for diamond shape construction.

3 Testing

The code for testing can be run from *CS5001-p4/src/* as:

```
$ javac -cp .:../lib/junit-4.11.jar:../lib/junit.jar: tests/SimpleModelTest.java
$ java -cp .:../lib/junit-4.11.jar:../lib/junit.jar: tests/SimpleModelTest
```

The program provides a total of eight JUnit test cases for testing the intended functionalities of the model. The evidence for these are described below:

Test Case 1: The module *testDrawShape()* tests the sizes of redo and undo stacks after drawing a shape on the canvas.

Since this is the first shape drawn, the undo stack should have a single item while the redo stack should have been flushed.

Test Case 2: The module *testLoadAndSaveFile()* tests the file I/O operations alongside the state of undo and redo stacks after carrying out these. The byte array of the pixel information contained in the image of the canvas is stored and compared with the loaded image to ensure that both are equal. Further, another shape is drawn on the canvas to ensure that the undo stack contains at least two images. After an undo operation, the size of the redo stack must be 1.