CS5001 - Practical 4: Vector Drawing

Table of Contents

Summary

As part of the Vector Drawing practical I implemented the drawing software which I named Frida after the Mexican painter Frida Kahlo (hence the name of some of the classes).

Compile and run instructions

To compile all files (including the JUnit tests), from the project root directory run the following command:

```
javac -cp ./tests/junit.jar:./tests/hamcrest.jar:./tests/:. $(find . -name '*.java')
```

To run Frida, `cd` into the `src` directory and from there run the following command:

```
java main.FridaMain
```

Finally, to run the JUnit tests, `cd` back into the project root directory and run the command:

```
java -cp ./tests/junit.jar:./tests/hamcrest.jar:./tests/:./src/:.
org.junit.runner.JUnitCore modelTests.ModelTestSuite
```

All 95 Tests should be run.

Project structure

Overall, the submitted zip folder contains two sub-directories, `src` and `tests`. All the actual source code is in the former while the latter contains the JUnit tests.
I followed the MVC pattern, so the `src` directory contains the packages `model`, `view`, `controller`, and `main`, the last of which only contains a single class `FridaMain` that starts the programme with the GUI view.

Model

The `model` package contains the shapes as well as classes for reading and writing vector drawings from/to a file.

Shapes

Each shape implements the interface `IShapeModel` which specifies certain methods that are required from all shapes. Note that the interface implements `Serializable` so all of the shapes can actually be serialised.
I further added two abstract classes: `ShapeModel` is the more general one and it is directly inherited by the models for lines and ellipses, as I calculate their properties in a very similar way (by defining the start and endpoint of a framing rectangle, similar to the way they are represented in java.awt.geom). The second class `ShapeModel2D` extends `ShapeModel` and adds properties and methods that are used for any polygon-like shape (similar to java.awt.Polygon), i.e. everything from a rectangle to a star. Hence, all other shapes extend `ShapeModel2D`.

IO

The `model` package also contains two classes to read and write from/to a file. There is nothing special about them, they simply take a filename and either write an ArrayList of serialisable `IShapeModel` objects to file or read such a file in and convert it back into an ArrayList. Possible exceptions are thrown in order to let the view handle them appropriately.

View

The main part of the `view` package is the `FridaView` class which defines the components of the GUI and implements `Observer`. All components are stored in the JFrame `mainFrame`. Upon user input, the class passes the input to the controllers of the models.
While `FridaView` sets up some components of the `mainFrame` directly (such as the menu bar and toolbar), I implemented the `DrawPanel` class separately as it is responsible for painting the shapes to screen. Crucially, the two ArrayLists `shapes` and `models` of `DrawPanel` keep track of all the shapes and models currently present on screen. Furthermore, I defined two short classes to let the user pick a colour (using JColorChooser) and the line thickness (using JOptionPane). Lastly, I define the abstract class `HelpText` that simply defines a String containing the help text that is shown to the user if they click on the corresponding menu item.

Controller

Similar to the way I structured the `model` package, I again define an interface, `IShapeController` that is implemented by the other controllers. However, this time not every shape gets its own class but instead it is sufficient to use the `ShapeController` for lines, the `EllipseController` for ellipses, and the `Shape2DController` for all other Polygon-like shapes. The separate controller for ellipses is necessary as, even though they are only defined by two points (start and end point of a framing rectangle), they are still two dimensional and hence can be filled with a colour.

Functionality

I implemented all basic requirements as well as four out of the five advanced requirements.

Basic requirements

The user has a toolbar available from which they can select different shapes to be drawn, open a colour picker dialog for the line and fill colour, or choose the thickness of the lines. Undoing (removing the shape that was drawn last) is possible either by pressing the `U` button on the keyboard or through the `Edit` menu. The same goes for redoing (the keyboard shortcut is `R` here). Note that once the user clears the drawing panel (key `C`), the history is cleaned and undoing/redoing is reset.

Advanced requirements

By holding down the `L` key while drawing an ellipse or a rectangle, the user can lock the aspect ratio to draw a circle or a square instead.
From the toolbar, the user can also select some more shapes, triangle, hexagons, and parallelograms.
By pressing `S` or by selecting `File > Save`, the current drawing can be saved to file. This opens a file chooser dialog to let the user pick a destination and name.
To read in a previously saved file, one can press `O` or selecting `File > Open` to open a file chooser dialog to pick the file to be opened. This file can then be modified just as any regular drawing. The user can even undo the recently created shapes from the loaded file in the reverse order they were originally created.
To move a shape, the user can select the `Move` button in the toolbar to move shapes around as desired. Changing the colour of previously created shapes is unfortunately not possible.
Another small feature I implemented is a help dialog that informs the user about the functionality and the keyboard shortcuts. The help dialog can be opened by selecting `File > Help` or by clicking the `H` key.

Tests

All classes in the `models` package have been tested extensively using JUnit with a coverage of 100%. I tested the GUI manually.

JUnit

See the Compile and run instructions to see learn how the JUnit tests can be run.
Basically I wrote a separate test class for each of the models (combining `ReadFromFile` and `WriteToFile` in one test class). I made sure to test all methods of all classes as to their expected behaviour including expected exceptions etc.
In total, I wrote 95 JUnit tests, all of which passed.

Manual tests

The manual tests were performed by playing around with the programme and trying out as many possible combinations and possible error sources as possible. Generally, the programme works as expected. The only issue that sometimes appeared was that small parts of the graphics were only rendered partly (such as a part of a line) when I moved the window to a second monitor and resized it. I couldn't resolve this issue, although I suspect it to be due to the change in aspect ratio when moving from one screen to another.

References

For the calculation of the corner points of the 5-point star, I based my implementation on the formula I found here: https://math.stackexchange.com/q/3582355 (Last access, 18/11/2020, 22:50).