CS5001 Object-Oriented Modelling, Design and Programming
Practical 3 – Vector Drawing

School of Computer Science
University of St Andrews

Due Friday week 11, weighting 45%
MMS is the definitive source for deadlines and weightings.

In this practical you are required to write a simple vector graphics drawing program using Java Swing. Your program should use either the Model–View–Controller (MVC) or the Model–Delegate (MD) design pattern. You should use JUnit to test your Model, and you should include these tests in your submission.

You may not use any third-party libraries in your code, except for the javax.json library for JSON parsing and generation and JUnit libraries for testing.

At a minimum, your program should support the following feature requirements:

Requirements

- Drawing straight lines
- Drawing rectangles
- Drawing ellipses
- Drawing triangles
- Undo/redo
- Different border/line color, border/line width, or fill colors (if applicable) for each shape
- Rotation of the shapes
- Support for drawing squares and circles. One way of implementing this feature would be using a key (say the Shift key) to lock aspect ratio during the drawing of rectangles and ellipses.
- Exporting the drawing as a jpeg or png file
- Select a previously drawn object and change its location, color or size
- Use networking to sharing drawings with other users using the protocol specified below

Implementation

For this practical you will have to think very carefully about the classes you will require for your program. You should design a suitable set of classes to represent the shapes which can be created with a specified color, fill, etc. and whose dimensions may be defined by a start and end position. Consider that you will probably want to be able to deal with a collection

of abstract shapes in your program code in a polymorphic manner rather than dealing with different collections for each shape. You should think carefully about where to place the code which specifies how to draw each shape in the GUI.

Whether you choose MD or MVC, you should aim to provide a Model component of your program comprising classes that model the shapes that have been drawn. The Model should provide operations to select the current drawing shape, create and manipulate these shapes. The model should also provide methods to undo and redo various operations (including shape creation) as well as methods to save or load a shape collection from file, etc. depending on how many features you implement. The classes comprising the Delegate or View/Controller should translate button presses to the appropriate calls to methods defined in the Model. You should use an appropriate notification structure such as PropertyChangeListener, so that the drawing canvas is redrawn when the model has changed.

See the figure below for an example of what your GUI might look like.

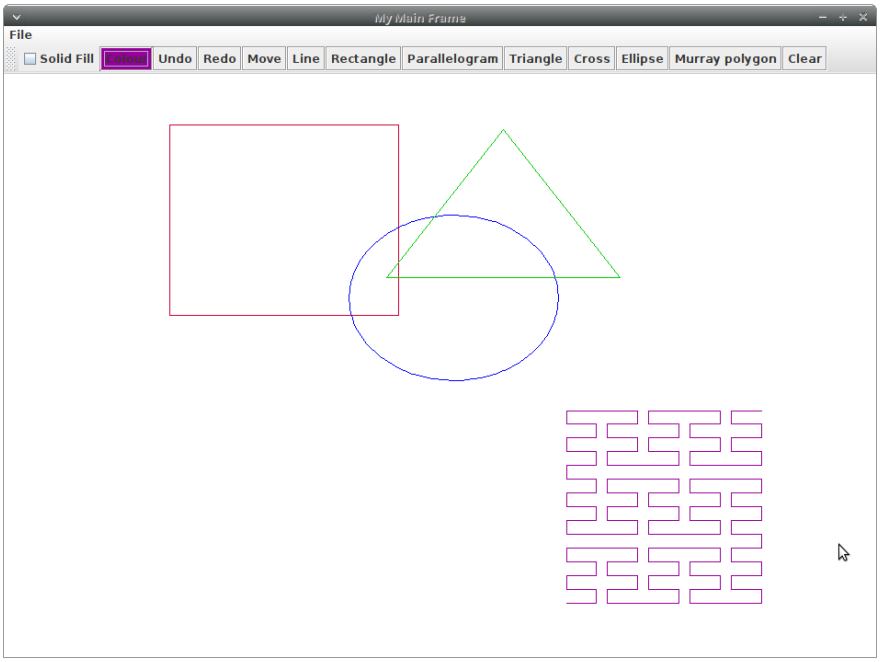

Figure 1: Example GUI layout, showing several shapes and colors. Please don't take this design as a requirement, feel free to be more creative! (Note you are no longer required to draw some of the shapes in the screenshot)

Network Procotol

You need to use the following protocol to communicate with the server.

Connect to the server at cs5001-p3.dynv6.net on port 8080. The communication protocol will be as follows. Remember that the communication is unencrypted, so you should not send any sensitive information.

The following operations are supported:

Each command should be sent on one line. The server will reply with one line. Make sure you do not close the connection, otherwise you will need to login again.

- `login`: The client sends a JSON object to the server. The JSON object has the following format:

```
{"action": "login", "data" :{"token": "token data"}}
```

The server will reply with ok or error:

```
{"result": "ok"}
```

or

```
{"result": "error", "message": "error message"}
```

- `getDrawings`: The client sends a JSON object to the server. The JSON object has the following format:

```
{"action": "getDrawings"}
```

The server will reply with a JSON object with the following format:

```
[{"id": "0aed29ae-1099-4d95-813f-bd88ad456a45", "created": "timeString",
↪  "modified": "timeString", "type": "rectangle", "x": 10, "y": 10,
↪  "properties": {"width": 10, "height"
: 10, "lineColor": "red", ...}}, { "id": 2, ...}]
```

- `addDrawing`: The client sends a JSON object to the server. The JSON object has the following format:

```
{"action": "addDrawing", "data": {"type": "rectangle", "x": 10, "y": 10,
↪  "properties": {"width": 10, "height": 10, "lineColor": "red", ...}}}
```

The server will reply with ok or error:

```
{ "result": "ok" }
```

or

```
{ "result": "error", "message": "error message" }
```

- `updateDrawing`: The client sends a JSON object to the server. The server will update the shape with the specified id using the properties specified. Any non-specified properties will remain unchanged. The JSON object has the following format:

```
{"action": "updateDrawing", "data": {"id":
↪  "0aed29ae-1099-4d95-813f-bd88ad456a45", "x": 10, "properties": {"color":
↪  "red"}}}
```

The server will reply with ok or error:

```
{"result": "ok"}
```

or

```
{"result": "error", "message": "error message"}
```

- `deleteDrawing`: The client sends a JSON object to the server. The JSON object has the following format:

```
{"action": "deleteDrawing", "data": {"id":
↪  "0aed29ae-1099-4d95-813f-bd88ad456a45"}}
```

The server will reply with ok or error:

```
{"result": "ok"}
```

or

```
{"result": "error", "message": "error message"}
```

Drawing Object JSON Format

As seen above, the JSON representing each drawing element has the following format:

```
{"id": "0aed29ae-1099-4d95-813f-bd88ad456a45", "type": "rectangle", "x": 10,
↪  "y": 10, "properties": {"width": 10, "height": 10, "lineColor": "red"}}
```

The *id* will be generated by the server on adding a shape. The *type* can be one of the following: `rectangle`, `ellipse`, `triangle`, `line`. The *properties* depend on the type of the drawing object. The following table shows the properties for each type:

Type	Properties
rectangle	width, height, rotation, borderColor, borderWidth, fillColor
ellipse	width, height, rotation, borderColor, borderWidth, fillColor
triangle	x2, y2, x3, y3, rotation, borderColor, borderWidth, fillColor
line	x2, y2, lineColor, lineWidth

The *borderColor* and *fillColor* properties are strings representing the color. For both, you should support the following colors: `red`, `green`, `blue`, `yellow`, `black`, `white`, `cyan`, `magenta`, `orange`, `pink`, `gray`, `darkGray`, `lightGray`. You could also support other color strings or RGB values, but you should do that on another property of your choosing and still keep the color property as described above as fallback for clients not implementing your extension. The *rotation* is in degrees between 0 to 359. The *lineWidth* is an integer representing the width of the line, between 0 to 100. The positional co-ordinates and dimensions are floating point numbers between -10000 and 10000.

As a minimum, your program should support the above properties and values. You can add more properties (up to a maximum of twenty) if you want to support more features though this might not be shared by other student's clients. If you receive a type of shape or property that you do not support, you should ignore it. If you do not get a property that you expect, you should use a default value. The server will do rudimentary validation, so you should implement robust validation in your client, and discard any invalid data.

If you add any extra properties (up to the maximum of twenty properties), you should ensure that they are either a string or an integer/floating point number. If it is a string, it can be a maximum of 42 characters long. If it is an integer or floating point number, it should be between -10000 and 10000.

Note that to prevent the list of shapes from becoming too big, if you add too many shapes to the server, the server might delete older shapes from your user. You should not also add shapes too frequently, as the server might rate limit you and respond slowly, or return an error.

Testing

Along with your program, you should create a test suite that tests the functionality of your Model. This test suite should be written in JUnit, and should test the broadest possible range of inputs to your Model. You should aim to execute every line of code in your Model somewhere in this test suite, and it should also include edge cases and invalid inputs. Your JUnit tests do not need to test the view/controller or delegate parts of your program, but you should test these thoroughly by hand to make sure that they work.

As well as running on your own computer, the tests should be runnable on the lab computers or host servers. Make sure to run your JUnit suite on the School infrastructure and ensure that they work before submitting. You should include everything necessary to run the tests in your submission, and your readme should explain clearly how to execute them.

Deliverables

For this practical you must include a readme file. This file must:

- List all the features you have implemented;
- Explain clearly how to run your program and how to use all its features;
- Give clear instructions on how to run the JUnit tests you have written.

When you are finished, you should submit a zip archive containing your source code, your JUnit tests, and your readme, and submit it to MMS in the usual way.

Marking

Grades will be awarded according to the General Mark Descriptors in the Feedback section of the School Student Handbook. The following table shows some example descriptions for projects that would fall into each band.

Grade	Examples
1–3	Little or no working code, and little or no understanding shown.
4–6	Acceptable code for part of the problem, but with serious issues such as not compiling or running. Some understanding of the issues shown.
7	A running stand-alone program that implements some of the basic functionality, but may contain bugs, poor style, and many missing features. Little or no object-oriented design, and little or no documentation.
8–10	A running stand-alone program that implements most of the basic functionality. Possibly lacks some of the more complex features, has problems with error-handling and poor style. Some documentation is provided.
11–13	Implements all shapes and their properties, as well as providing an attempt at sockect communication, with reasonable error-handling and acceptable program design and code quality. Some object-oriented design, and few bugs. Reasonable documentation.
14–16	Implements all shapes and their properties and the socket communication with a well-structured object-oriented design, and very little code repetition. Possibly has some minor bugs. Good documentation is provided.

Grade	Examples
17–18	Implements the requirements with correct behaviour as well as socket communication and error-handling with a clean object-oriented design. The provided code makes proper use of inheritance, association, polymorphism and encapsulation, with very good method decomposition.
19–20	Outstanding implementations of all features as above, and possibly some extra features not mentioned in the specification, with exceptional clarity of design and implementation.

Any extension activities that show insight into object-oriented design and test-driven development may increase your grade, but a good implementation of the stated requirements should be your priority.

Lateness

The standard penalty for late submission applies (Scheme A: 1 mark per 24-hour period, or part thereof):

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

The University policy on Good Academic Practice applies:

https://www.st-andrews.ac.uk/students/rules/academicpractice/