170021928
CS5001-P4 Report

How to Run

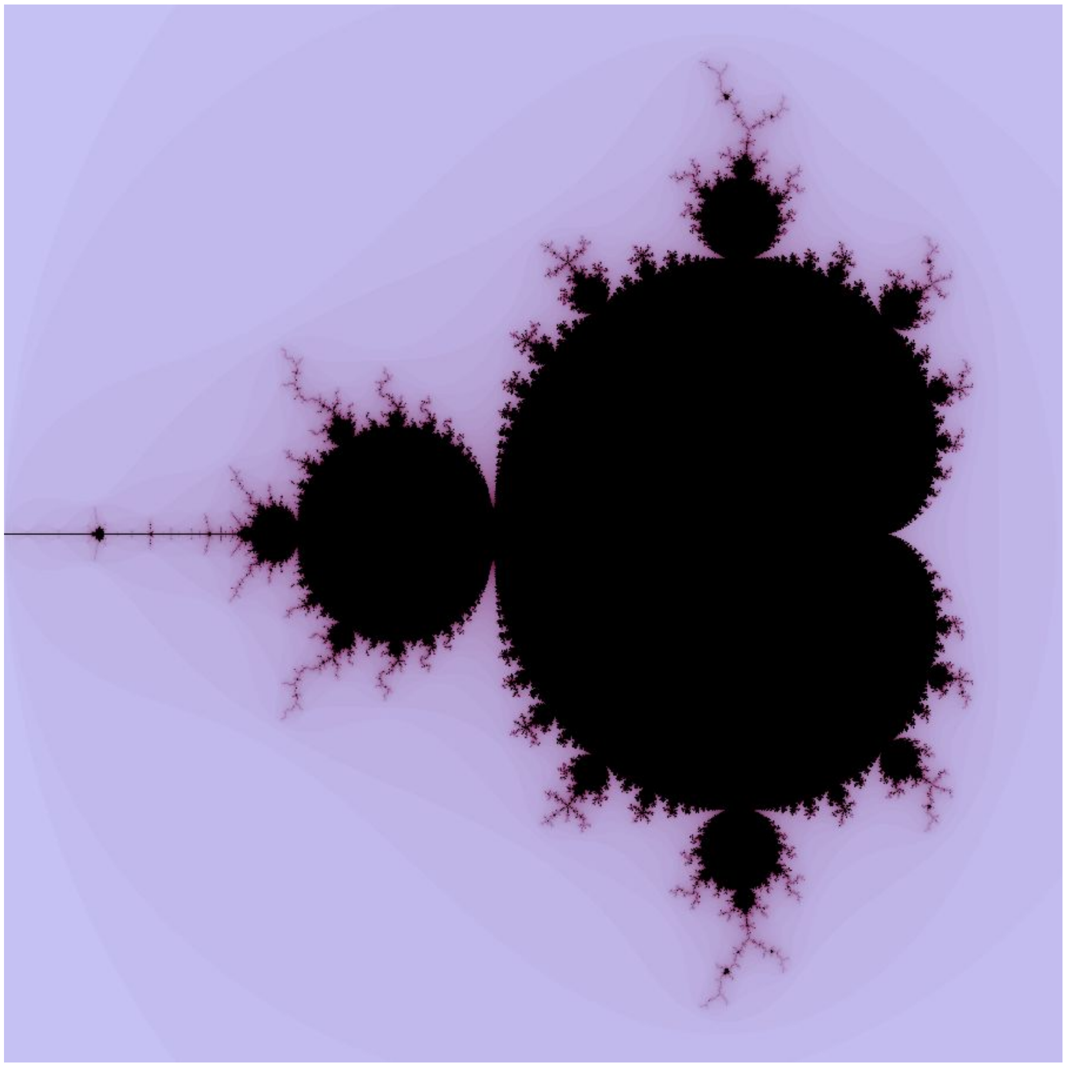
You may compile and run from the src folder, or use the runnable jar file I have provided thus:

```
java -jar Mandelbrot.jar
```

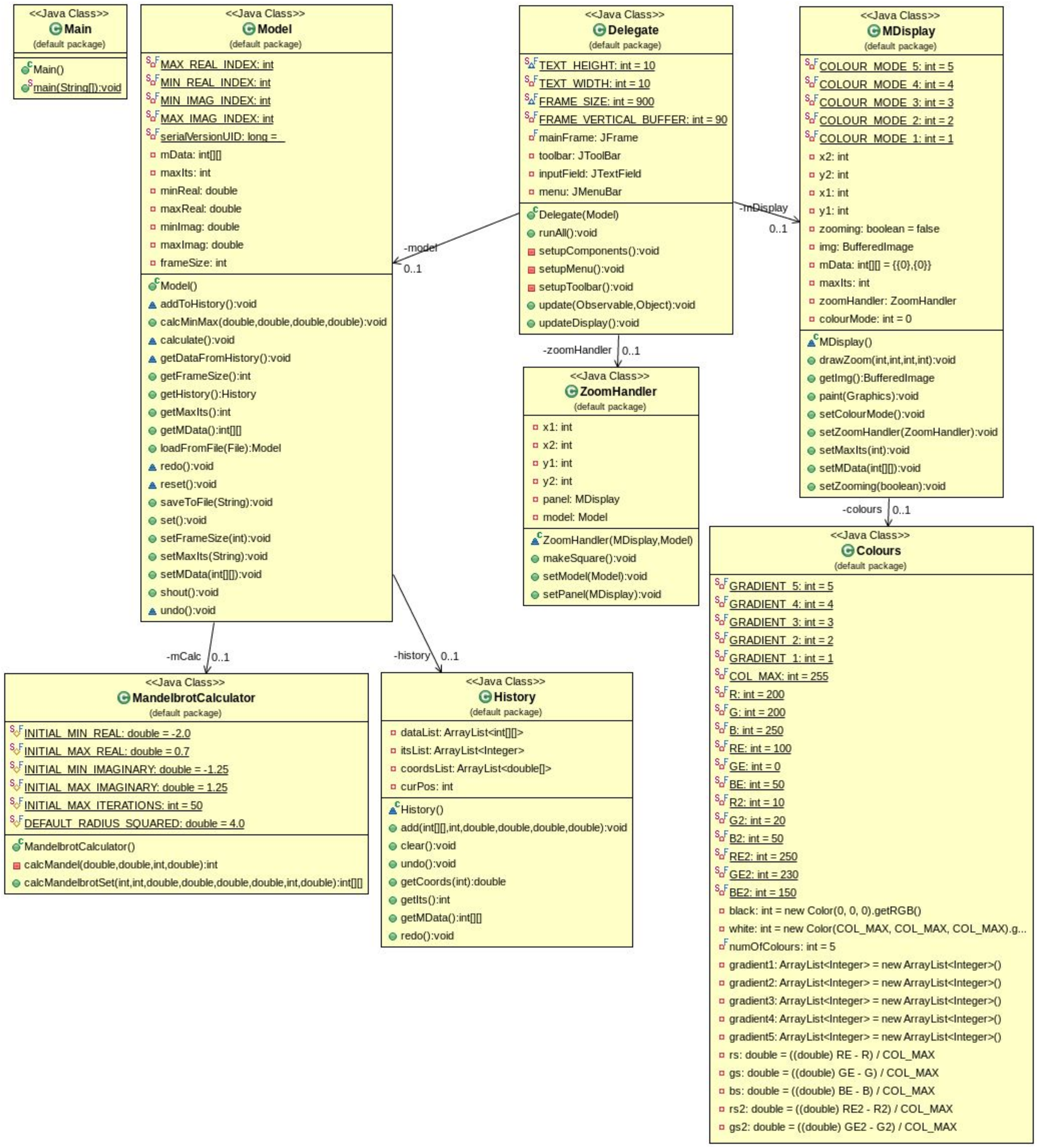
Extensions

I have implemented all of the basic requirements, plus:

- Different colour mappings
- Save and load
- Additional operations (save as .png) and colour mappings



Design



My program implements the Model-Delegate design pattern, with two main classes (Model and Delegate), each supported by helper classes. There is also a Main class that creates the initial model and delegate to run the program. Keeping the model and the delegate separate is useful, as it means that if in the future we wanted to use a different GUI library, the model code could be reused as it stands.

Model

Consists of Model, MandelbrotCalculator, and History.

Model is observed by Delegate. It is responsible for keeping track of all the variables needed to calculate the mandelbrot image, and for doing this calculation when prompted. When its variables have been updated according the the calculation, or in response to undo/redo etc, it alerts the delegate which then updates the view with the new information. MandelbrotCalculator was provided to me in the spec and handles the calculation itself, and History keeps track of the current state of the Model's variables, plus any past or previous states, to enable undo and redo.

Delegate

Consists of Delegate, Colours, MDisplay and ZoomHandler

Delegate observes Model. It is responsible for setting up the GUI, listening for user input and passing it to the model. When the delegate observes that the model has been updated it redraws the image accordingly. The main Delegate class is assisted by three helper classes: Colours, which generates and stores arrays of RGB values to use for the different colour modes; MDisplay, which extends JPanel to draw and redraw the mandelbrot data (and/or the zoom marquee) to a buffered image according the current colour mode; and ZoomHandler, which listens for the user's mouse input, parses the co-ordinates and passes them to the model.

Please see code & javadoc comments for fine detail of each class and method implementation.

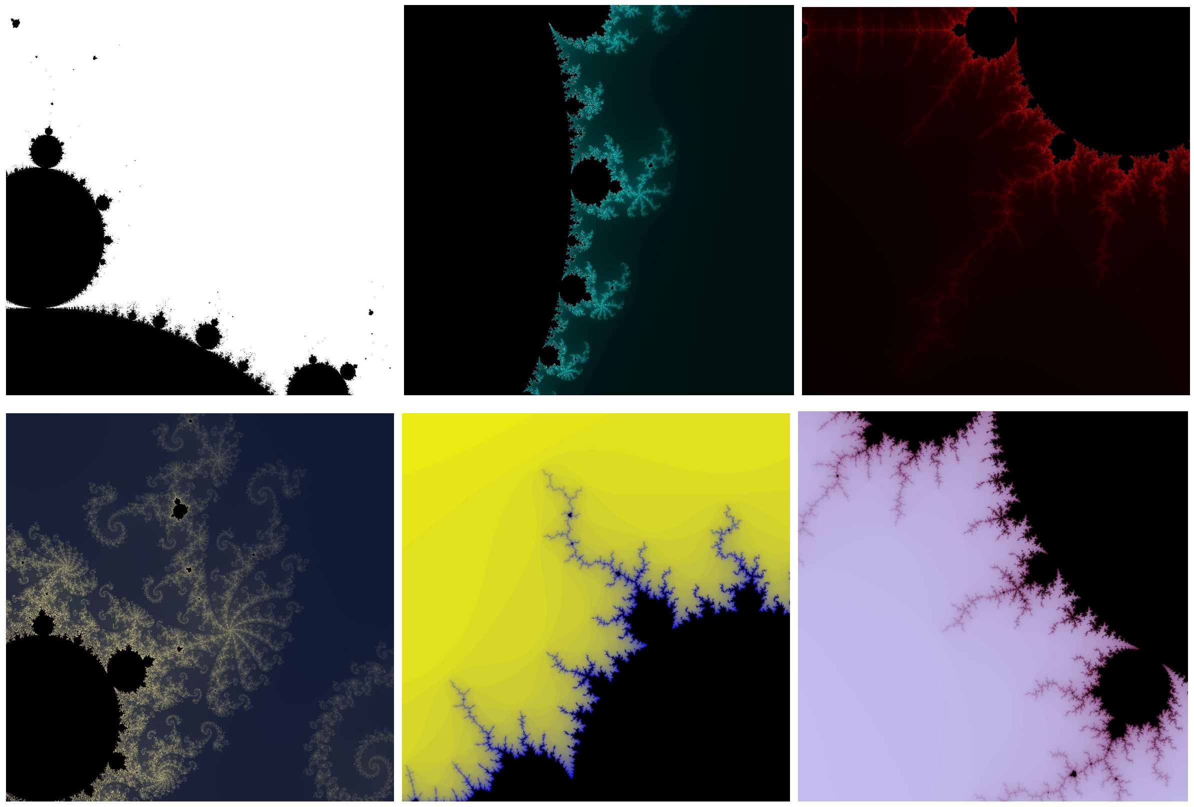
Initial Calculation & Display

When the program is launched, the main class creates a new Model instance and a new Delegate instance, passing the Model to the Delegate. When the Delegate constructor is called, it sets up the GUI components, adds itself as an observer to the model, and calls the model's set() method. This method sets the model's variables to the default values provided in the MandelbrotCalculator class, and creates a new History instance. The model then calls its shout() method, which calls setChanged() and notifyObservers(). As Delegate observes Model, it is notified.

In Delegate's update() method, runAll() is called which tells the model to calculate the mandelbrot set. The result of this calculation is a two dimensional array of ints, which is stored in the model and then passed by the delegate, along with the number of maximum iterations, to delegate's instance of MDisplay. MDisplay extends JPanel, and within its paint method simply creates a new buffered image and iterates through the mandelbrot data, colouring each pixel of the image according to the colour mode.runAll() then calls MDisplay's repaint() method, before prompting the model to add its values to its History object to allow for future undoing and redoing.

Colour Views

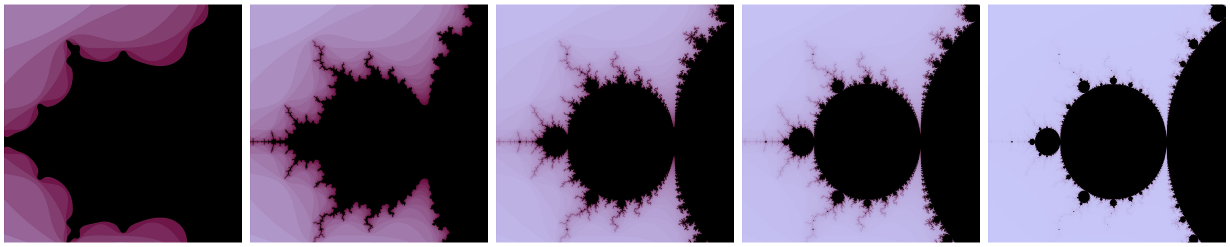
There are six different colour views - one basic black and white without a gradient, red and blue as mentioned in the spec, and three others of my own creation. The user may toggle between these. In the MDisplay class, each pixel in the image is coloured based on the data calculated by the MandelbrotCalculator. This data takes the form of a 2d array with each value being some number between 0 and the maximum number of iterations. These values are used to calculate which colour a given pixel should be, by mapping the value to a set range, and then using that to index into a preset array of integer colour values. These arrays are generated and stored when a new instance of the Colours object is created, using a loop to gradually increment the colour values before adding them to the array, to get a smooth gradient.



Max Iterations

By entering a number in the text field, the user can adjust the number of iterations. If the user increases the number of iterations as they zoom in, the mandelbrot could be explored indefinitely.

10, 20 50, 100 and 500 iterations:



Zoom

The user may click and drag a marquee on any part of the image to zoom in. This is handled by the ZoomHandler class, which is responsible for listening to the user's mouse clicks and parsing their co-ordinates. To ensure that the image does not get distorted, the marquee is constrained square. When the mouse is dragged, MDisplay's drawZoom() method is called to update the coordinates of the marquee and repaint the image to include the rectangle. When the mouse is released, the new coordinates are sent to the model, which uses them to update its minimum real, maximum real, minimum imaginary and maximum imaginary values.

Undo, Redo & Reset

When a new model is created, it in turn creates its own History object which is used to store and retrieve the parameters used to create the mandelbrot image (namely the mandelbrot data and the number of iterations), plus the minimum and maximum real and imaginary values, which are needed to create the *next* image if the user zooms again. The History object also contains an int curPos, which is used to keep track of the current position within the history so that the user may undo and redo multiple times. curPos is incremented by the model whenever a new set of values is added to the history and whenever redo() is called, and decremented whenever undo() is called. When reset() is called, the history is cleared and all the model's values are set to their defaults.

Save & Load

The image can be saved in two ways - as a .mandelbrot file, which is a custom file type that can be loaded back into the program, and as a png image file. To save as a .mandelbrot, the delegate's model instance (which contains all the information necessary to redraw the current image) is written to a file using an object output stream. Similarly, .mandelbrot files are loaded by simply updating the delegate’s model with the one read in from file. Save as png is implemented by getting the display panel’s buffered image and writing it to file using ImageIO.write().

Testing & Evaluation

It’s hard to write tests for GUIs! I used print statements and eclipse’s debugger to ensure that at each stage my code was working as expected, which was an effective strategy. Had I more time, I would have liked to implement an animated zoom, and also the functionality for the user to pan around the image after zooming. It would also be really cool to export an animated zoom to a gif! In terms of implementation, I think my code is fairly robust and well decomposed into methods which makes it easy to isolate bugs. I feel that my implementation of the Model-Delegate design pattern is successful, and that the classes are well encapsulated and interact sensibly. I’m also very proud of my breathtakingly beautiful colour schemes!