2025

University of St Andrews | FOUNDED 1413

Graphics - Mandelbrot Set Explorer

COURSEWORK 03

PALOMA PÉREZ DE MADRID LAGUNA – STUDENT ID 250016561

Student ID 250016561

INDEX

CS5001: COURSEWORK-03 REPORT

OVERVIEW

The goal of this application is to visually render the Mandelbrot Set through a graphical user interface.
The project is built around the provided `MandelbrotCalculator.java`, whose output is displayed and managed through a clean object-oriented architecture.

The implementation emphasizes **core** Object-Oriented Programming principles (encapsulation, abstraction, inheritance, and polymorphism) while maintaining clear separation of concerns. To achieve this, the application follows the MVC (Model–View–Controller) design pattern, described in detail in the next section.

DESIGN AND IMPLEMENTATION

The Model–View–Controller (MVC) pattern is a software architecture that separates an application into three distinct, interconnected components. Its main purpose is to improve modularity, maintainability, and scalability by isolating the user interface from the underlying logic. This separation makes the system easier to extend and test, while also reducing coupling between parts of the code. As described by GeeksforGeeks (2024), MVC helps organize applications so that each component has a clear and focused responsibility.

MODEL

The Model represents the application's data, logic, and rules.
In this project:

- It stores the Mandelbrot state (centre coordinates, zoom window, max iterations, colour map).
- It uses `MandelbrotCalculator` to compute iteration values.
- It notifies listeners when the state changes, enabling automatic UI updates.

The model also manages full undo/redo stacks through immutable `MandelbrotState` snapshots, so every change to the view window or iteration depth is reversible and traceable in order. All mutations (`setViewWindow`, `setMaxIterations`, `pan`) validate inputs up front, push the prior state onto undo, clear redo when appropriate, and immediately recalculate the fractal data to keep data consistent with the current parameters. Persistence is built in: `saveToFile`/`loadFromFile` serialize or restore the current bounds, iteration limit, and colour map name so sessions can be resumed. For user feedback, `getMagnification` reports the zoom level relative to the initial frame, helping the UI display how deep you've zoomed. Colour map names are normalized to a sensible default when missing or blank, and every state change calls `notifyListeners`, guaranteeing that any registered view (like `MandelbrotPanel`) repaints in sync without manual refresh calls.

VIEW

The View is responsible for what the user sees.
In this project:

- It renders the Mandelbrot fractal on the screen.
- It displays UI components like buttons, spinners, and colour selectors.
- It updates its visuals automatically when the Model changes.

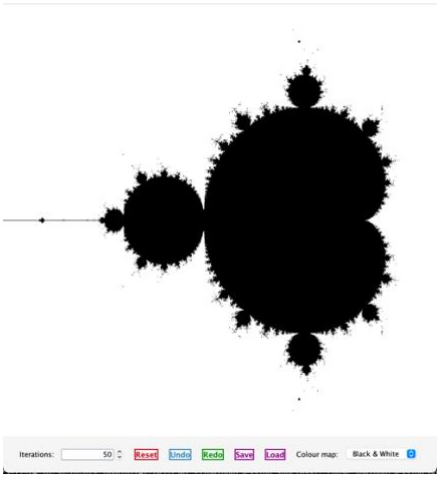

Figure 1: Example of UI (Black & White)

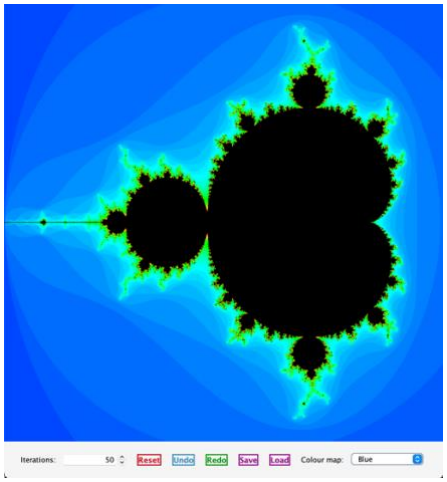

Figure 2: Example of UI (Blue)

`MandelbrotPanel` paints every pixel using the active `ColourMap`, overlays a green selection box while dragging to zoom, and repaints automatically when the model fires `modelChanged()`. The `ControlPanelMandelbrot` groups user controls (iteration spinner, reset/undo/redo, save/load, colour map picker) with consistent styling from `UIStyler`, while `MandelbrotApp` arranges the panel and controls in the main window. Because the panel listens to the model via `ModelListener`, any state change—new data, palette switch, undo/redo—triggers a repaint so the UI stays in sync without manual refreshes.

CONTROLLER

The Controller handles user interactions.
In this project:

- It interprets mouse actions (zooming, selecting rectangles, panning).

- It listens to UI controls (iteration spinner, undo/redo buttons, colour map combo box).
- It updates the Model based on user input.

The Controller acts as a translator between the View and the Model, ensuring the correct logic is executed when users interact with the interface. It wires every UI signal to the model and view: it registers mouse listeners on `MandelbrotPanel` to detect drag-zoom versus pan (Shift or right-click) and translates pixel rectangles into complex-plane bounds before calling `setViewWindow` or `pan`. It also listens to the iteration spinner, undo/redo/reset, save/load, and colour-map combo, validates spinner edits, updates the model, and keeps the spinner/selection in sync when the model changes. Errors during save/load surface via dialogs, and every model change triggers a repaint through the panel's `ModelListener` hook, so user actions immediately reflect on screen

STRUCTURE

The project is organized in a simple layout. At the top level, the makefile handles building, testing, running the program, and generating documentation. The main source code lives inside the `src/` folder, split into model, view, and controller packages to match the MVC design. All JUnit tests are grouped under `src/test/`, keeping testing separate from the main code. The `doc/` directory stores the generated documentation and the PlantUML diagrams. There's also a `JUnit/` folder that includes the standalone JUnit jar needed to run the tests, along with the UML file generated by an external tool.

```
.
├── makefile              # Build, test, run, javadoc targets
├── README.md             # Project overview and usage
├── doc/                  # Generated docs and PlantUML diagrams
├── src/
│   ├── controller/       # ControllerMandelbrot (wires UI + model)
│   ├── model/            # ModelMandelbrot, calculator, state, listeners
│   ├── view/             # Swing UI, panels, colour maps, styling
│   └── test/             # JUnit tests for model, controller, view
└── JUnit/                # JUnit standalone jar (test runtime)
```

Figure 3: Project file structure

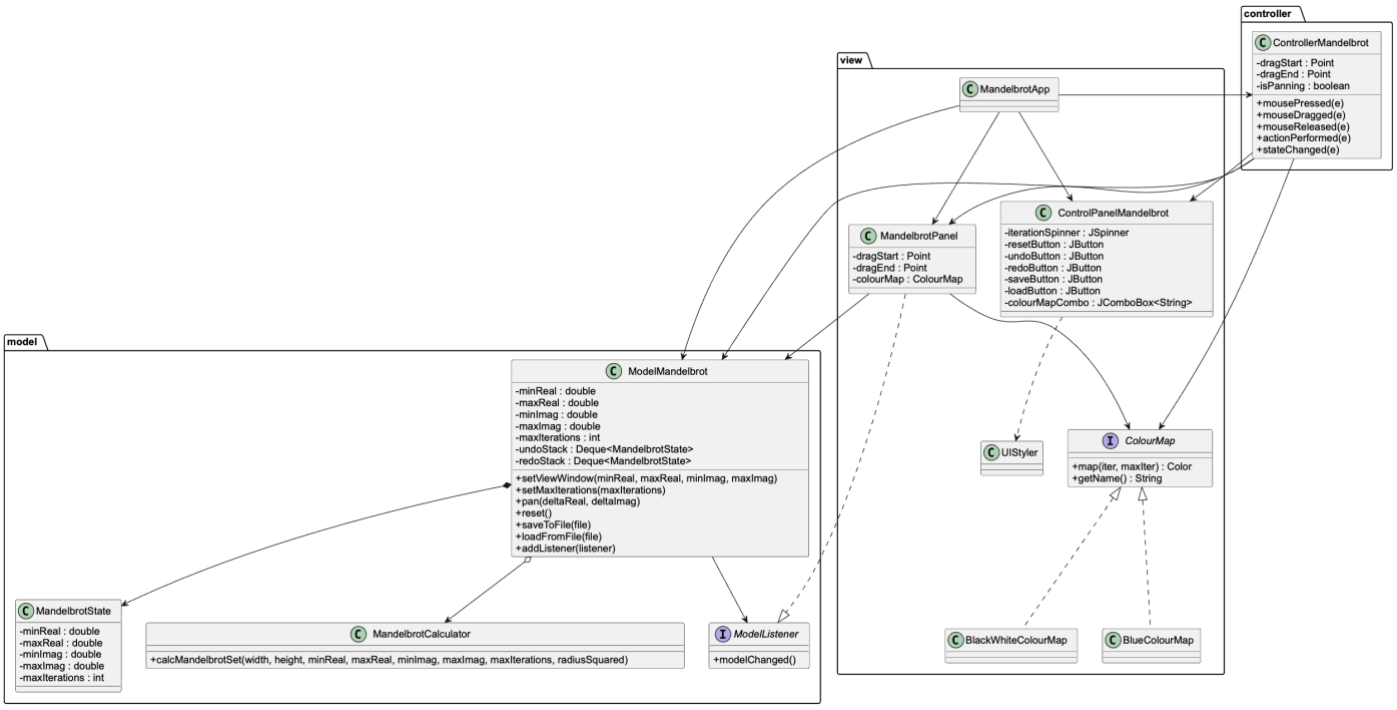

Figure 4: Generated UML of the code ("src" folder)

Student ID 250016561

The UML diagram for this project was first generated using the Visual Studio Code extension "UML Generator for Java" created by 0SMA0 (0SMA0, n.d.). This tool produced an initial diagram saved as `doc/src.puml`. After reviewing and refining the structure to better reflect the design and relationships within the project, I created an improved version stored as `uml.puml`. The final diagram image was then rendered using the online PlantUML editor at *https://plantuml.app/editor*.

OBJECT-ORIENTED PROGRAMMING PRINCIPLES

The code demonstrates the four fundamental pillars of Object-Oriented Programming (OOP):

Encapsulation: The project maintains strict encapsulation by keeping internal state private within classes such as `ModelMandelbrot`, `ControllerMandelbrot`, and the Swing-based UI components.
State is accessed and modified only through controlled getters, setters, and helper methods (e.g., undo/redo support), ensuring that internal logic remains protected from unintended external interference.

Abstraction is achieved through interfaces like `ModelListener` and `ColourMap`, which define what the system can do without exposing how it is implemented. These abstractions allow components to interact through well-defined contracts, promoting loose coupling and easier extensibility.

Inheritance appears in several parts of the system, most notably in the colour map hierarchy:
`BlackWhiteColourMap` and `BlueColourMap` implement the shared `ColourMap` interface, inheriting its contract for mapping iteration counts to colors.
In testing, `FakeModel` extends `ModelMandelbrot` to override or mock specific behaviours, enabling isolated and predictable test scenarios.

Polymorphism enables multiple implementations to be used interchangeably.
Both `MandelbrotPanel` and `ControllerMandelbrot` work against the `ColourMap` and `ModelListener` interfaces, meaning different colour maps or listeners can be swapped in without altering client code. This allows for flexible rendering, alternative behaviours, and improved testability, all through uniform method invocation.

RUNNING THE PROGRAM

This project already includes the JUnit standalone jar that was used in previous coursework. It is stored inside the `JUnit/` folder under `JUnit/lib-alone/junit-platform-console-standalone-1.13.4.jar`, so there is no additional setup required to run the tests.

To start the application, you can simply run **`make run`**. This command compiles all the source files into the `bin/` directory and then launches the main class `view.MandelbrotApp`, which opens the graphical interface. If you want to run the test suite instead, you can use **`make test`**. This will build both the main code and the test classes and then execute them using the JUnit jar included in the project.

The makefile also includes a few other useful commands. Running make compile compiles the source files without launching the program. Using **`make javadoc`** generates the documentation inside the `doc/` folder. Finally, **`make clean`** removes the `bin/` and `doc/` directories to give you a fresh workspace.

Note: The program assumes you already have Java JDK 17 or higher installed and available on your system's PATH. It also requires that make is installed on your machine, since the provided makefile depends on it for building, running, and testing the project.

Student ID 250016561

TESTING

The tests focus mainly on the core logic of the application. Most of the testing effort is placed in the Model because it contains the actual behaviour of the Mandelbrot viewer: state updates, calculations, undo/redo logic, and save/load functionality. These parts are deterministic, isolated, and easy to verify with automated tests. In contrast, the View and Controller contain more UI-focused code, which depends heavily on Swing events and visual rendering. These components are harder to test automatically and often require interaction or visual confirmation. For this reason, the tests in those areas are lighter and focus only on the behaviours that can be reliably checked in code.

Notes:

- *Temporary files created during the tests are cleaned up automatically.*
- *The controller test uses simulated mouse events, so no GUI interaction or window display is required.*

MODEL TESTS

`src/test/model/ModelMandelbrotTest.java`

- `testResetRestoresInitialValues` checks that after modifying the view window, max iterations, and colour map, calling `reset()` restores all initial constants and triggers a recalculation.
- `testUndoRedoStacks` exercises several actions (`setViewWindow`, `setMaxIterations`, `pan`) and verifies the correct behaviour of the undo and redo stacks, including clearing redo history after new actions.
- `testSaveAndLoadRoundTrip` saves the model's state to a temporary file, loads it into a fresh model, and confirms that all fields and data match.
- `testLoadFromFileRejectsMalformedData` writes an invalid properties file and ensures that `loadFromFile` throws an IOException when it encounters malformed input.
- `testSetColorMapNameNullResetsToDefault` confirms that passing null to `setColorMapName` safely resets the model to the default colour map.
- `testListenersAreNotifiedOnStateChanges` attaches a recording listener and verifies that all relevant state changes (iterations, panning, colour map updates, and resets) send the expected notifications.

CONTROLLER TEST

`src/test/controller/ControllerMandelbrotTest.java`

The behaviour is checked using a FakeModel.

- `modelDelegationFromUiActions` simulates spinner updates, colour map selection, and Shift-drag panning. It verifies that the controller forwards these actions to the model exactly once and with the correct parameters.

VIEW TEST

`src/test/view/ViewMandelbrotTest.java`

The tests focus only on the behaviours that can be validated programmatically.

- `blueColourMapProducesGradient` checks that BlueColourMap produces a smooth gradient, correctly reports its name as "Blue", and maps the maximum iteration count to black.
- `blueColourMapHandlesZeroMaxIterations` makes sure the colour map handles the case where maxIterations is zero without errors, returning black as a safe default.

RESULTS

Figure 5: Result of command "make test"

REFERENCES

[1] GeeksforGeeks. (2024). MVC design pattern. https://www.geeksforgeeks.org/system-design/mvc-design-pattern/

[2] 0SMA0. (n.d.). *UML Generator for Java* [Visual Studio Code extension]. Retrieved from https://marketplace.visualstudio.com/items?itemName=0SMA0.uml-generator