

Jeffry - Project Portfolio

Project: Piconso

Overview

Piconso is a powerful general-purpose image editing tool. The user interacts with it through a flexible command-line interface and receives instant visual feedback through the Graphical User Interface (GUI) written in JavaFX. The application is mainly written in Java and spans a considerable 10k Lines of Code (LoC). The codebase is well-maintained with a rigorous system of checks and tests in place to ensure that code quality is consistently high. A comprehensive set of guides are also provided to ensure a smooth on-boarding process for both users and contributors alike.

Summary of contributions

As the point man of the group, I led the group through discussions from idea generation to delegation of work. I placed extra emphasis on engineering optimal solutions to real pain points. As a result, my team was able to morph the given codebase from a trivial application into a polished product.

I put my considerable experience to use by being available at all times to help out with various tasks like design considerations and debugging. I believe that my contribution has improved the unlying engineering of the codebase which in turn sped up the application manyfold.

My major contributions are as follows:

Designed and implemented the unlying data structures and models for the manipulation of canvases and layers.

This allowed other team members to maintain a clear separation of concerns and reduce dependencies which in turn made the code base less prone to breaking changes. This is a major change that affects virtually all of the code base and future developers looking to add or improve Piconso will have to build upon my models.

In fact, I have already extended my models to provide end-users with a hassle-free and easily understandable way to navigate more intricate image editing features such as layer ordering and canvas size. This improves the user experience significantly as these features are expected of any modern image editing tool. It also reduces the friction of users migrating to Piconso from other competitors.

Added a sub-command targeted at tech-savvy users : `raw`.

In line with our goal of being a powerful and flexible tool, I have added a feature that allows users to tap into the full power of Piconso. This is a notable departure from the other commands in the application which can be fairly restrictive but easy to learn. The raw command allows users to quickly experiment and iterate upon their image editing process without excessive hand-holding from the application.

By giving advanced users the freedom to do as they please, I have brought Piconso closer in terms of feature parity as compared to commercial heavy-weights like Photoshop.

Developers will also save time on manually implementing obscure functionalities that the majority of the end-users will likely never use. This results in a more streamlined code base unlike Photoshop which clocks in at over 10 million LoCs.

Besides that, I have also made multiple smaller contributions which significantly improves the user experience.

Implemented the ImagePanel

The ImagePanel is at the heart of Piconso, displaying the output of Piconso to the user visually. It was carefully engineered with reuse in mind and its usage painstakingly documented for future developers.

Improved upon the HistoryListPanel

Building upon the work of other teammates, I have improved the HistoryListPanel to reflect past operations on across multiple layers. Now users can quickly see what operations they have performed on each layer.

Implemented the LayerPanel

Similar to the HistoryListPanel, the LayerPanel displays information that that users might have trouble keeping track of in a clean and easily grokkable fashion. Users can clearly see which layer they are operating on as well as the order of layers.

Executed upon the change in layout

I rearranged the layout of the user interface components from the initial code base to one that was designed by the team to be more user-friendly.

Here is the code that I have written for this product: [\[All commits\]](#) [\[Dashboard\]](#)

Other contributions

- Project management:

I commented on critical pull requests. I also extensively tested and reported bugs along with a comprehensive reproduction reports: [#170](#) [#175](#)

- Documentation:

I wrote a tutorial to walk users through the on-boarding process with a sample project that demonstrates most of the key features of our application. This light-hearted tutorial also serves to show that our application is relevant and solves problems that the reader can relate to.

I also contributed multiple new illustrations to the developer's guide to help future contributors visualise what each command does. The illustrations are included in my contributions below.

Contributions to the User Guide

Given below are sections I contributed to the User Guide. They showcase my ability to write comprehensive but simple documentation that addresses the needs of users.

Getting started

Let's work through a simple project in Piconso from start to finish. In this tutorial, you will learn how to create some of the dankest memes starting from a blank template.

If you ever get stuck with a command, pop down to the Features section for reference!

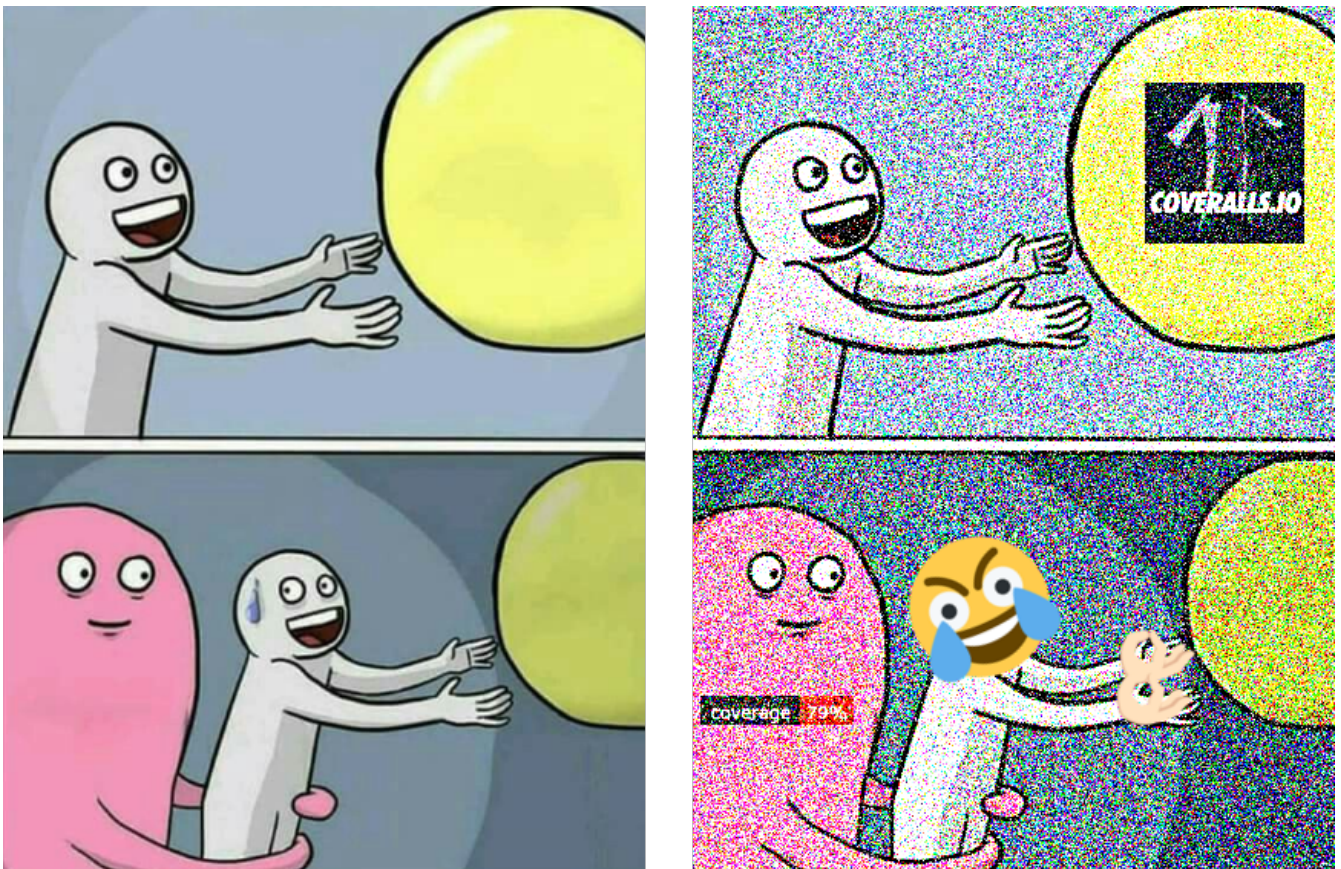


Figure 1. The Piconso team knows how it feels.

Ready to start your Piconso journey?

1. Head over to the link [here](#) to get a zip file with all the assets you need to get started!
2. Unzip the files into somewhere easy to access, like your Pictures folder.
3. Start Piconso!
4. Use the `cd` and `ls` commands to get to the folder with your unzipped assets. Your current file location is as noted on the bottom right of the application. You can use the `TAB` key to auto-complete pesky folder names! If everything goes well, you should be able to see all the images show up in the Image Panel on the lower left.

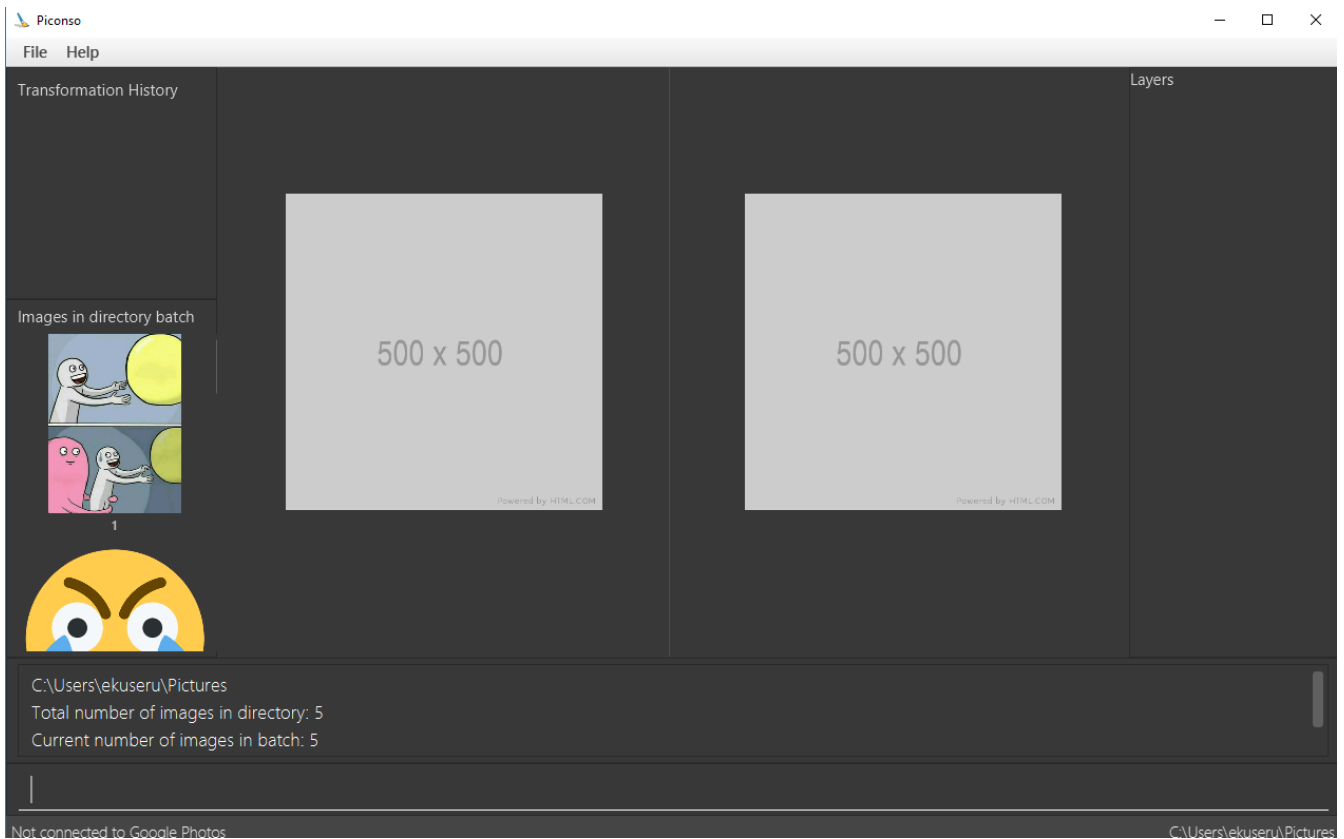


Figure 2. Images and their corresponding indexes are showing up!

5. See the number below the base image? That's the *index* of the image. It might be different depending on your folder's content. If you don't see it within the first ten images, use the **next** and **prev** command to navigate!
6. Open the image simply by typing **open [index]**. (Substitute index with the number you noted down just now)
7. You should see the base template in all its glory.

The tutorial is truncated to account for the page limit.

User guide command documentation

Canvas operations : **canvas**

Changes the size of the canvas: **canvas size**

Format: **canvas size (NEW_SIZE)** → You can change the size of the canvas to with this command. Remember that the effects of **canvas auto-resize** takes precedence!
If **canvas auto-resize** is **off**, cropping might potentially occur.

NOTE

If the optional parameter NEW_SIZE is not provided, the current size will be displayed in the output instead.

Examples:

- `canvas size 800x600` - Sets the canvas to have a height of 800 pixels and a width of 600px.
- `canvas size` - Prints the current size.




Changes the background color of the canvas: `canvas bgcolor`

Format: `canvas bgcolor (NEW_COLOR)` → You can change the background color of the canvas with this command. This is only visible if the canvas is larger than all the images placed on it. See the examples for the formats that your colors can take.

NOTE

If the optional parameter `NEW_COLOUR` is not provided, the current colour will be displayed in the output instead.

Examples:

- `canvas bgcolor none` - Sets the canvas to have a transparent background.
- `canvas bgcolor #0f0` - Sets the canvas to the hex colour #00ff00 .
- `canvas bgcolor #00ff00` - Sets the canvas to the hex colour #00ff00 .
- `canvas bgcolor rgba(0,255,0,0.7)` - Sets the canvas to the hex colour #00ff00 but with 70% opacity .
- `canvas bgcolor` - Prints the background colour.

Allowing the canvas to auto-resize: `canvas auto-resize [ON|OFF]`

Format : `canvas auto-resize [ON|OFF]` → This command allows you to turns the auto-resize for the canvas on or off.

When auto-resize is on, it can potentially override the size manually specified with the `canvas size` command. When `on`, the canvas expands as required to ensure that no clipping occurs.

NOTE

New canvases default to having auto-resize off.

Examples:

- `canvas auto-resize on` - Allows the canvas to expand and prevent cropping.
- `canvas auto-resize off` - The height and width of the output canvas will remain as is.

Contributions to the Developer Guide

Given below are sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.

Layer (layer) commands

The `layer` command follows a similar pattern as the `canvas` command. The following sub-commands inherit from `LayerCommand`:

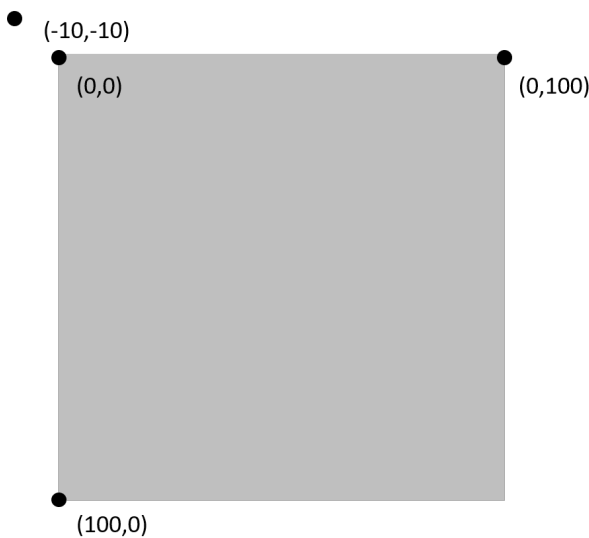
- **add** - Adds a layer from the **Canvas** and generates a layer name.
- **delete** - Removes a layer from the **Canvas**. The current layer and the last remaining layer cannot be removed.
- **select** - Selects a layer to work on.
- **swap** - Swaps the order of two distinct layers.

Current implementation

Layer implements a few key accessors and utility functions. Some of them include:

- **addTransformation(Transformation)** - Adds a given transformation into its **PreviewImage's TransformationSet**.
- **getName()** - Gets the name of the layer.
- **setHeight(int)** - Sets the height of the layer.
- **setWidth(int)** - Sets the width of the layer.
- **setPosition(int, int)** - Sets the x and y coordinates of the layer.

The following image illustrates the coordinate system adopted in Piconso.



The default anchor point is the top-left corner, this means that a **layer position -10x-10** command will set that layer's top left corner at **(-10, -10)**. It is possible to have negative co-ordinates although clipping will occur unless the canvas is set to auto-resize.

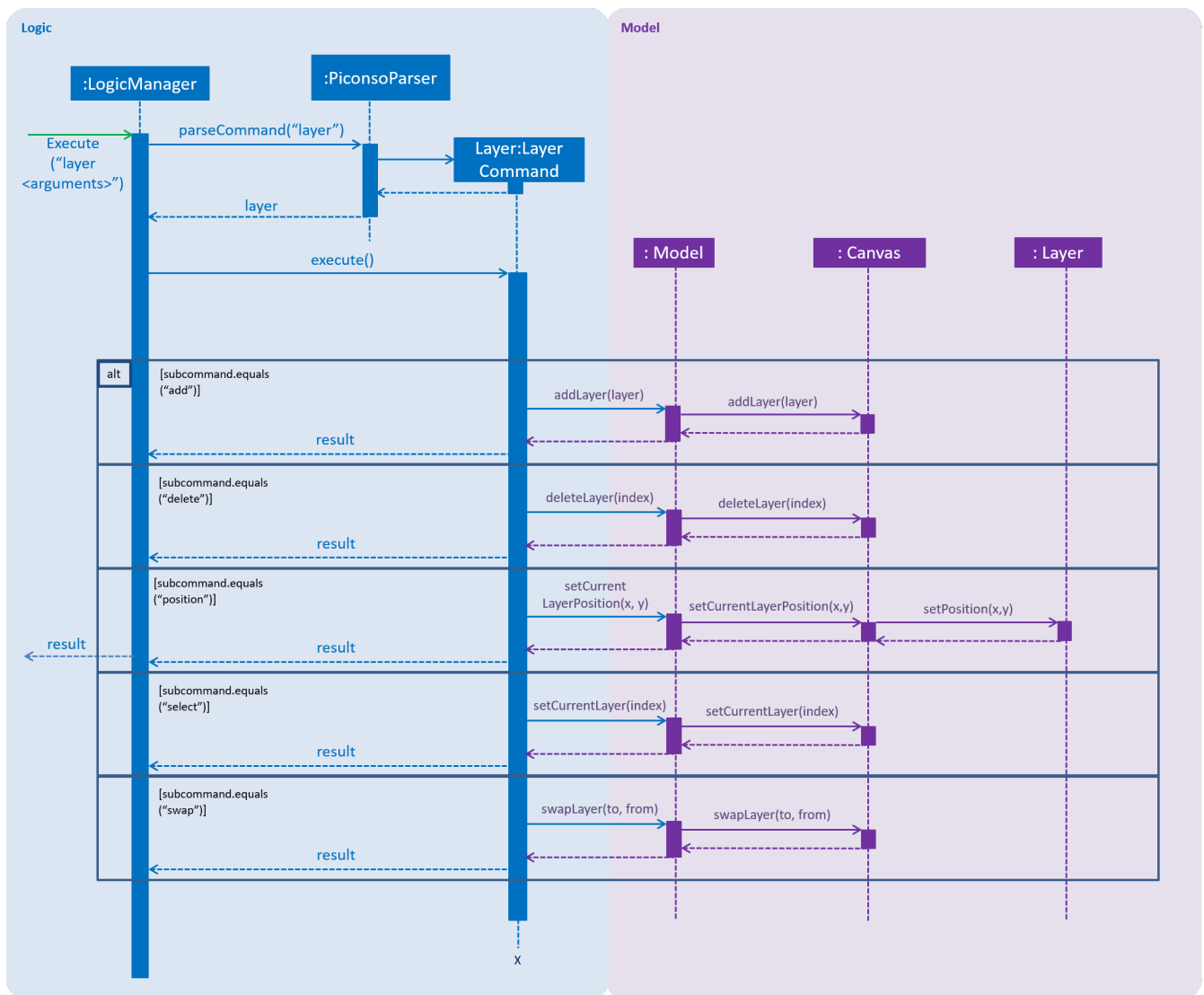


Figure 3. Sequence Diagram for Layer Command

The above diagram illustrates the process of parsing and executing of **canvas** commands.

1. The user executes any command beginning with **layer**.
2. The command is first parsed by **PiconsoParser** which picks up the **layer** keyword and passes any remaining arguments to **LayerCommandParser**.
3. **LayerCommandParser** determines the appropriate sub-command being executed and performs the requested operation on the canvas and current layer.

Design Considerations

Further manipulation of the **ProcessBuilder**

Alternatives

It is actually possible to nest **ImageMagick** commands which means that it is possible to keep separate **List<T>**s and conjugate them when the canvas needs to be rendered. The resultant ImageMagick command will take the form :


```
magick [overall canvas flags] {[canvas flag] (individual layer flags)} [overall canvas flags]
```

Where blocks enclosed by `{ }` need to be repeated per layer.

Evaluation

This solution is the most straight-forward and results in no intermediate files which is usually desirable.

However, the resultant ImageMagick command will short circuit and cause the entire expression to fail if any of flags are incompatible or incorrect. Caching is also impossible, causing the whole canvas and all of its layers to be composed again from scratch. This results in a poor user experience and hence we have decided against it.

Canvas and Layers in action

As the `canvas` and `layer` commands compliment each other, let's walk through a typical user's session in Piconso from start to finish.

1. The user executes a valid `open` command: A `Canvas` is constructed holding exactly one `Layer` with the default height and width being that of the image selected.
2. The user adds a new layer to the canvas: `Canvas#addLayer` is executed and the helper functions in `ModelManager` refreshes the UI.
3. The user swaps the order of the two layers to such that the original image is back on top: `Canvas#swapLayer` swaps the two entries in the list of layers.
4. While still on the original layer, the user moves it to the top left: `Layer#setPosition` moves the first layer out of the way.
5. The user decides to work on the second layer and enters `layer select 2`: `Canvas#setCurrentLayer` changes the current layer to the given index and internally keeps track of the index as well.
6. A transformation is applied by the user to the current layer: `Canvas#addTransformation` appends the new Transformation to the current Layer. Note that no image processing occurs until a new render is requested.
7. After a `canvas size 80x60` command: The new `Canvas#setSize` is applied upon the next render and it crops the image to a fraction of what it used to be. Remember that resizing and scaling is accomplished with the `apply resize` command.
8. Realising his mistake, the user sets the canvas to fit his all of his layers: `Canvas#setCanvasAuto` toggles a boolean.
9. The user fills in the default transparent background with a `canvas bgcolor #707070`: a hex color code is accepted by `Canvas#setBackgroundColor`.



Figure 4. Results of various commands in sequence. The black bounding rectangle indicates the canvas size.