TEAM SQUAAD

Deliverable 2

Table of Contents

System Architecture of Matplotlib

Introduction to Matplotlib

Matplotlib is an Python based, open source library which is able to plot and produce various different types of figures and graphs, both in a static format (such as an image) and in an interactive format. Matplotlib includes a module named pyplot, which is the main interface used to configure and create needed graphs/diagrams.

Matplotlib Architecture

Matplotlib as a whole uses an open layered architecture. However, despite the high coupling which may come with such a architecture choice in the layers, it is important to note that lower layers are not aware of the layers above them - lower layers do not depend on the upper layers, reducing coupling. In Matplotlib, there are three main abstraction layers - the Façade Layer, the Artist (Presentation in UML) Layer, and the Backend (Application Logic in UML) Layer.

The topmost layer is exposed through pyplot, and is referred to as the Façade Layer. As its name suggests, it follows the Façade design pattern - it is the layer that most users of Matplotlib will interact with, providing various high-level API functions and methods so that the user can configure, edit, and render diagrams and figures. This layer takes in the user's commands and configurations, and performs the low-level calls associated with creating and rendering the figures, axes, and customizations.

The middle layer is the Artist Layer (also known as the Presentation Layer). As its name suggests, much like an artist, this layer knows how to paint the requested axes and data onto figures. There are two types of artists present within this layer:

1. The primitive artist type - these artists represent primitive graphical objects, such as basic lines, text, and shapes.
2. The container artist type - these artists hold multiple primitive artist types in meaningful groupings to be rendered.

In order to paint things, this layer uses the correct, platform specific, Renderer and FigureCanvas from the bottom layer. Coupling between this and the bottom layer occurs within

the draw() method, allowing for displaying onto the canvas. In general, most things in Figure are an instance of Artist and most of the “heavy lifting” is done within this layer.

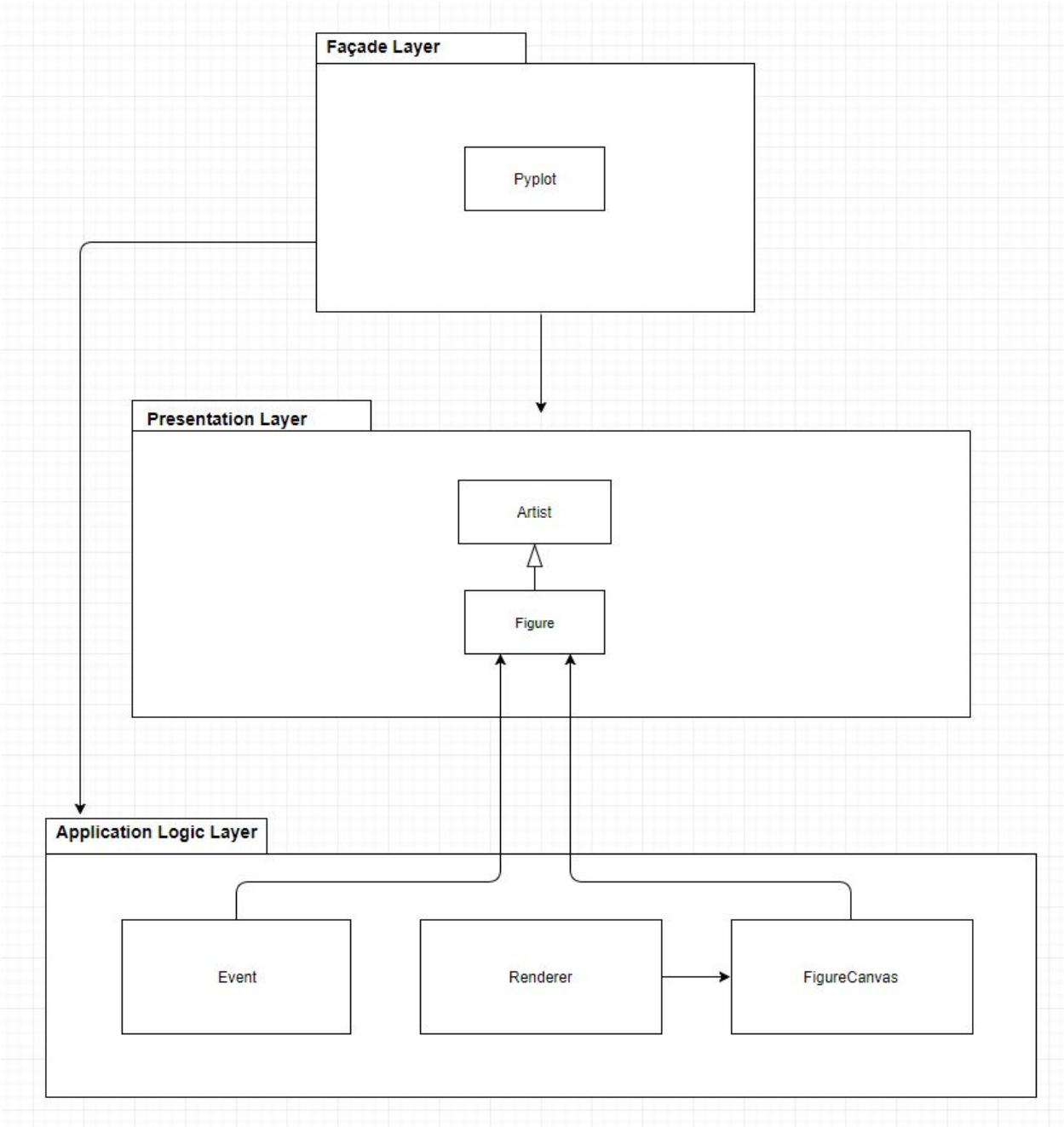
The bottommost layer is the Backend Layer (also known as the Application Logic Layer). This is the layer that contains the abstract base classes for FigureCanvas, Renderer, and Event that a renderer/framework needs to implement in order to serve as a backend for Matplotlib. There are two types of backends: interactive and non-interactive. Interactive backends serve as a wrapper for various application frameworks such as Qt, GTK, tkinter, etc., allowing for graphics to be displayed in a GUI. Non-interactive backends implement ways to save graphics to a file, for example, PDF, SVG, PNG, etc. As an Application Logic Layer, it provides the necessary drawing board (FigureCanvas) and the tools necessary to draw (Renderer), while the Event class handles user inputs, such as keystrokes and mouse movement. Cohesion within this layer is quite sizeable due to multiple instances of communication with multiple application frameworks.

Comments/findings/notes:

- Due to the one-way architecture of Matplotlib, coupling between layers is reduced, which is good.
- The abstraction of event toolkit allows devs to write UI event-handling code such that it applies everywhere and is reusable.
- Axes minimize the Figure’s dependency and allow it to manage parts of the Artist.
- The Figure is an important object that manages all elements in a graphic. The architecture of Matplotlib allows for more complex features to be built into Figure while avoiding complicating the backends.
- Matplotlib is built to be multiplatform.
- Pyplot allows the users to not worry about the complicated areas of Matplotlib; providing a simple, easy to use interface.

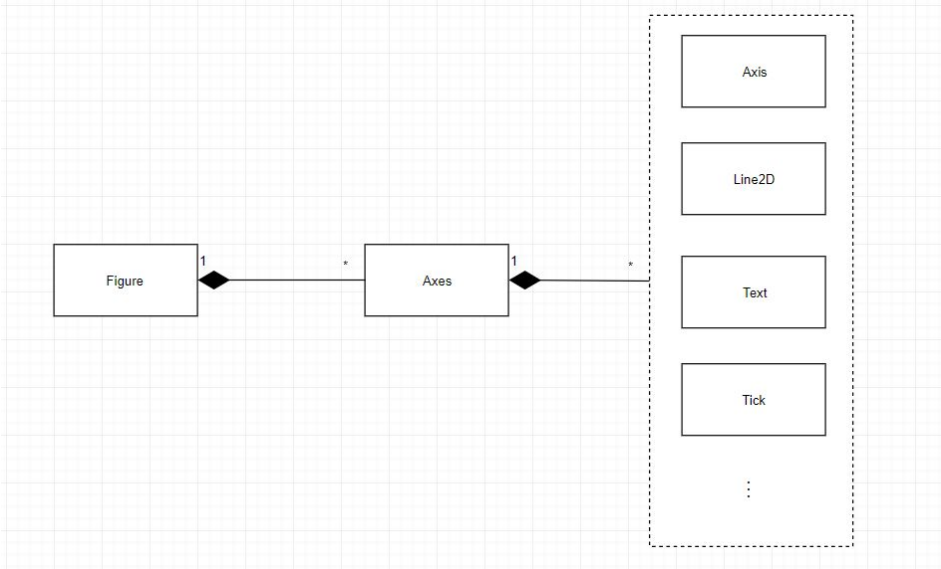
UML Diagrams

Overall Architecture of Matplotlib



(Figure.0)

Important class/area of code in Matplotlib (Axes)



The Axes is the region of the figure where data is visualized. Consequently, it is the object that is most frequently interacted with by a user.

Our team has chosen the Axes (matplotlib.axes.Axes) as one of the most important parts of Matplotlib. To the user, the Axes are the “axes” that the user sees when figures and diagrams are rendered, and in the code, the Axes provide the container which interacts with the most different Artists in creating a figure/diagram.

When a diagram is drawn/rendered, the Axes is the class which not only provides the area in which objects and drawn, but also the class by which the user is able to plot their data to (similar to an axis of a graph). Hence, from a user's viewpoint, the the Axis class is visually a foundation of almost any diagram produced in Matplotlib.

From the code's perspective, the Axes class is what has the needed helper methods in order to carry out the work needed to process the user's data, create the needed Artist instances, and to add them to the relevant containers when requested.

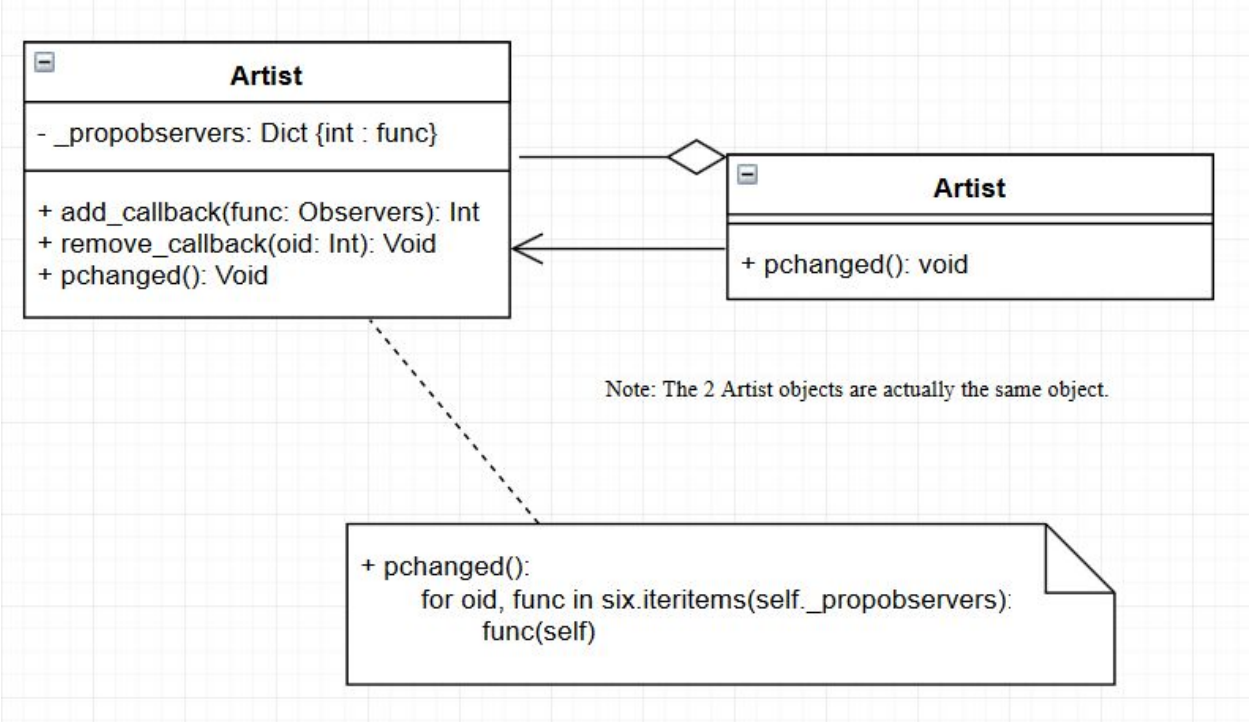
This is the class which conducts the formation of figures and diagrams in Matplotlib, and hence we believe it is one of the most crucial parts in Matplotlib.

Observer Design Pattern (Artist.py)

An observer design pattern can be seen in “matplotlib.artist.py”. In artist.py, there is a dictionary named _propobservers; where the keys are integers (oid) and the mapped values are functions. The functions are added and removed through the method Artist.add_callback() and Artist.remove_callback(). Whenever a property of Artist has been changed, pchanged() is called. This will iterate through all of the oid within the dictionary, calling upon each associated function (i.e. it will update all “observers” in the dictionary). An example of this can be seen in (*Figure.2)*.

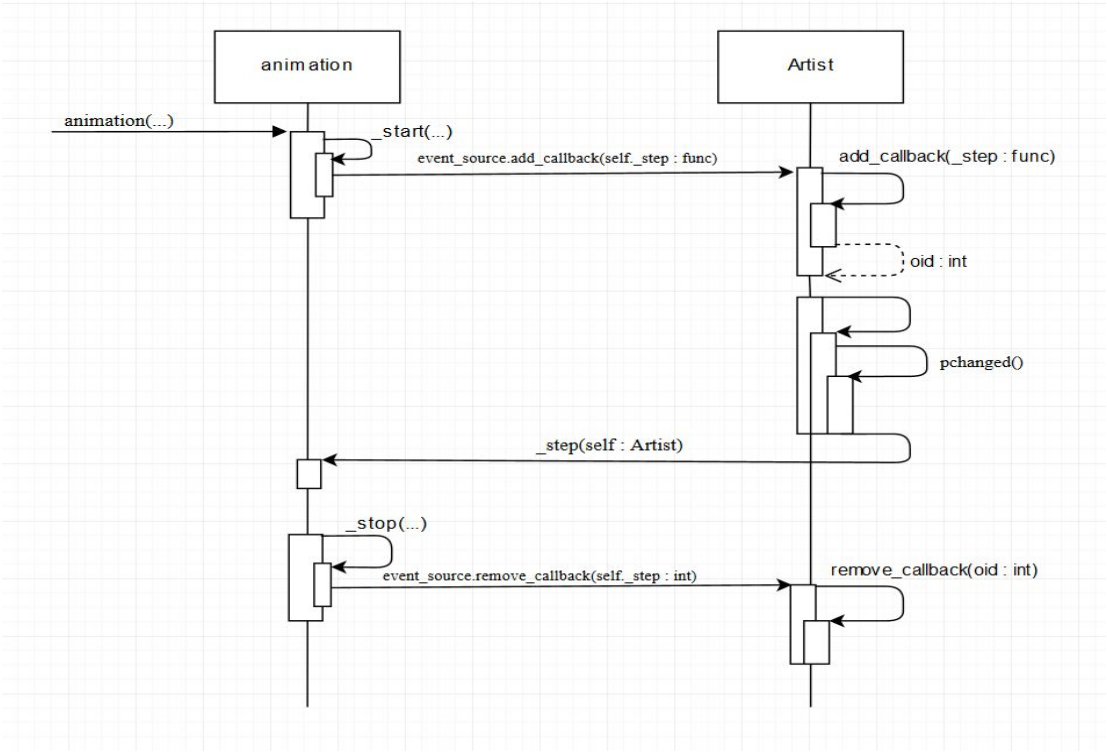
Note: We believe this is done most likely because when a property of Artist is changed, the Artist object must re-update itself, other parts of the systems, and the image on the canvas/figure to give the most up-to-date, and accurate data.

Observer UML Class Diagram



(Figure.1)

Observer UML Sequence Diagram



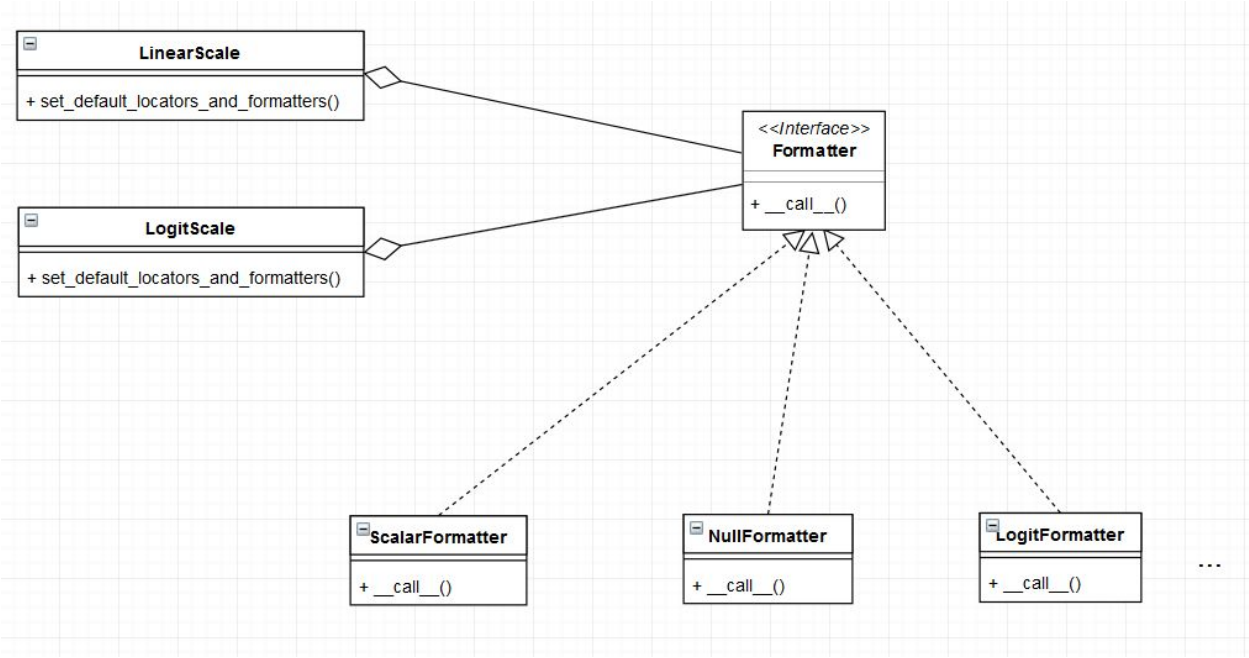
(Figure.2)

Note: event_source in Figure.2 is an Artist object.

Strategy Design Pattern: (Ticker.py and Scale.py)

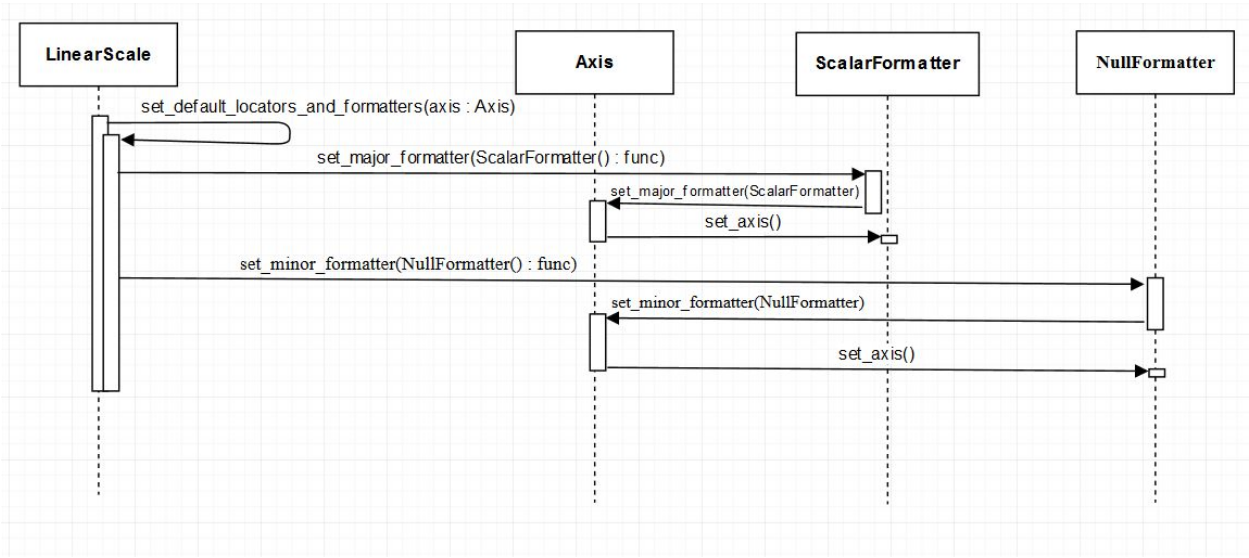
A strategy design pattern can be seen from "matplotlib.ticker.py", and "matplotlib.scale.py". Scale.py contains various scales, such as LinearScale and LogitScale, which are used to mark the axis of generated graphs depending on the type of data that is being graphed. Each scale class in scale.py contains a method named set_default_locators_and_formatters which sets the axis to use different tick formatting strategies by default according to the scale class used. An example of this can be seen in Figure.3 and Figure.4, where LinearScale uses ScalarFormatter and NullFormatter. The tick formatting strategies (e.g. ScalarFormatter, NullFormatter, LogitFormatter, etc.) are derived from the Formatter class in ticker.py. We believe this strategy/design pattern is used because what is required can simply be done by changing a method by the class being used. In this case, the many Formatter classes, and it's method.

Strategy UML Class Diagram



(Figure.3)

Strategy UML Sequence Diagram

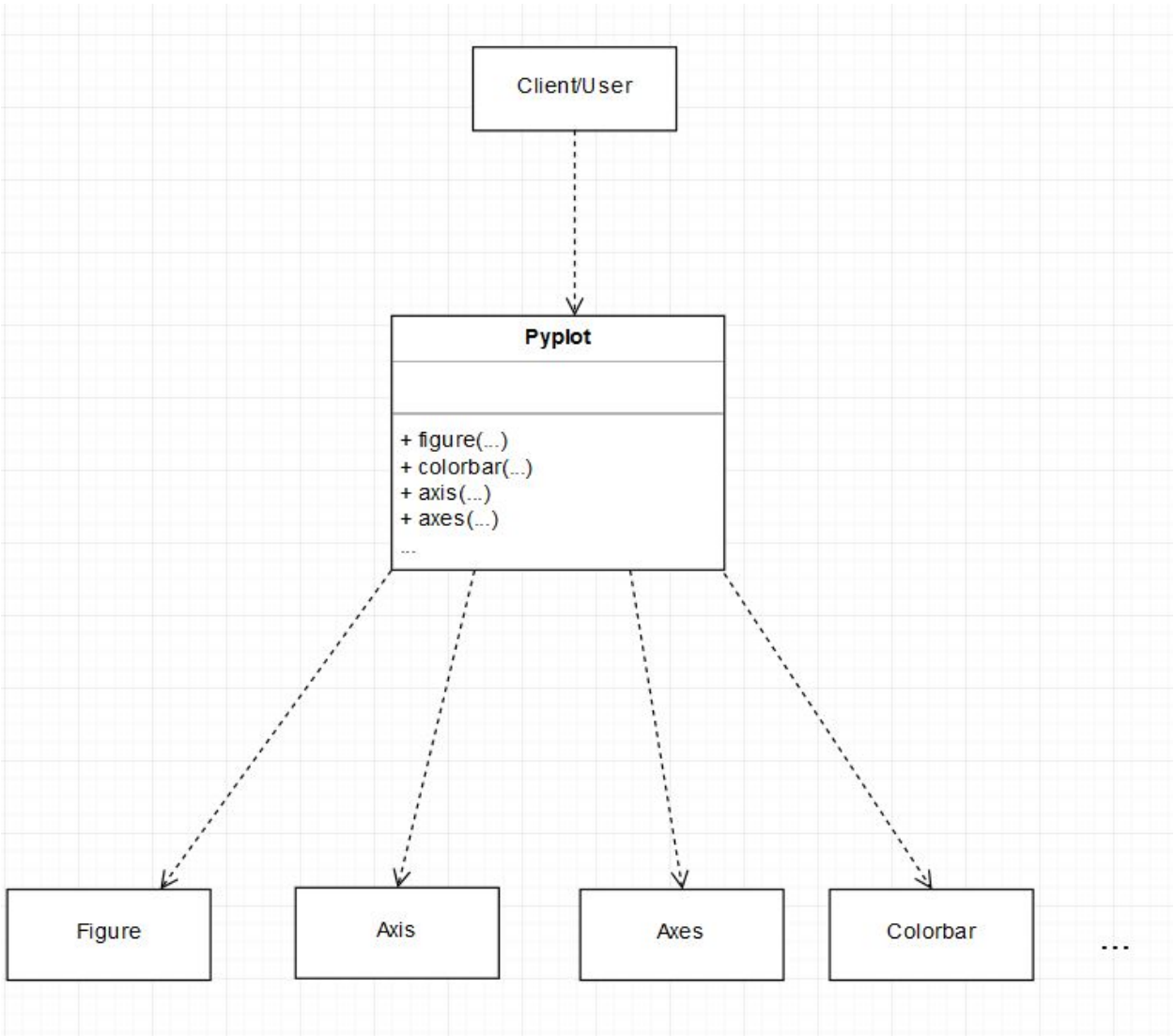


(Figure.4)

Façade Design Pattern: (pyplot.py)

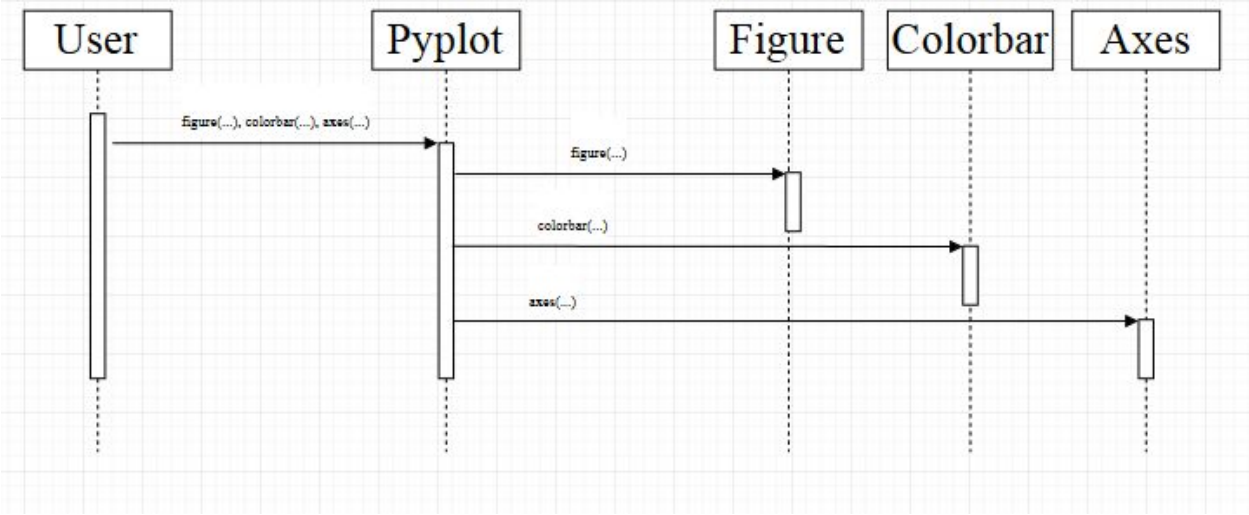
A façade design pattern can be seen in “matplotlib.pyplot.py”. Note that there doesn’t exist any Pyplot class. It’s the entirety of the file being imported to be used by the user. Pyplot acts as an interface for the user to create the figure/graph/plot/etc. that they want without having to directly work with the more complicated areas such as the backend of Matplotlib. Pyplot instead interacts with the other 2 layers of the API on the user's behalf. Examples of this can be seen in the UML diagram where pyplot functions, Figure() and colorbar() are called. We believe this design pattern was used so as to have an easy to use interface for the user for a complex large system that is Matplotlib.

Façade UML Class Diagram



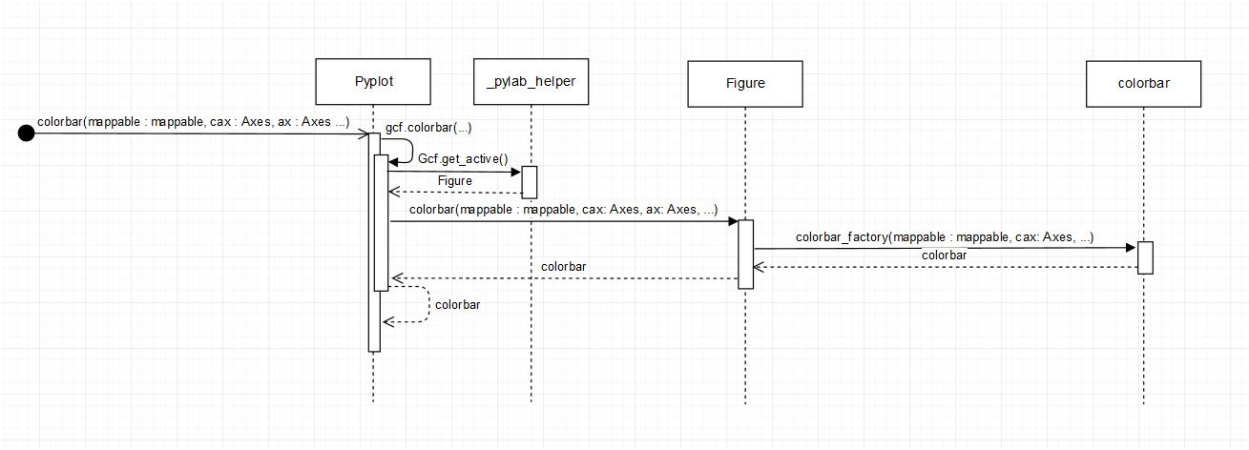
(Figure.5)

Facade UML Sequence Diagrams



(Figure.6)

Note: This is the UML sequence diagram of the facade like behaviour pyplot exhibits



(Figure.7)

Note: This is the UML sequence diagram of a inner behaviour of pyplot calling one of its functions in figure.6. In this case, colorbar()