University of Toronto – Scarborough
CSCD01 – Engineering Large Software Systems

Deliverable 02 – Report

Prepared by ***GoonSquad*** Team

Swarnajyoti Datta
Nikki L. Quibin
Junil Patel
Beiyang Liu
Laine London
Hajoon Choi
Leo Li

Date: February 2, 2018

Table of Contents

Commentary on Architecture

Introduction

According to official documentation, Matplotlib's (MPL) architecture is comprised of a closed stack of 3 logical layers. This means that components in each layer can generally, at most, access the layer directly below it (not above or two levels away). This has allowed for (among other things) the complete encapsulation and separation of key operations like the designing and rendering of an MPL Figure (the top-level Matplotlib object). Each layer functions cohesively with one another with little coupling. With lower layers abstracted away from higher ones, extensibility of the project is inviting to developers. Additionally, users can interact with the software through one of the two top layers depending on their needs and expertise (as shown in Figure 1).

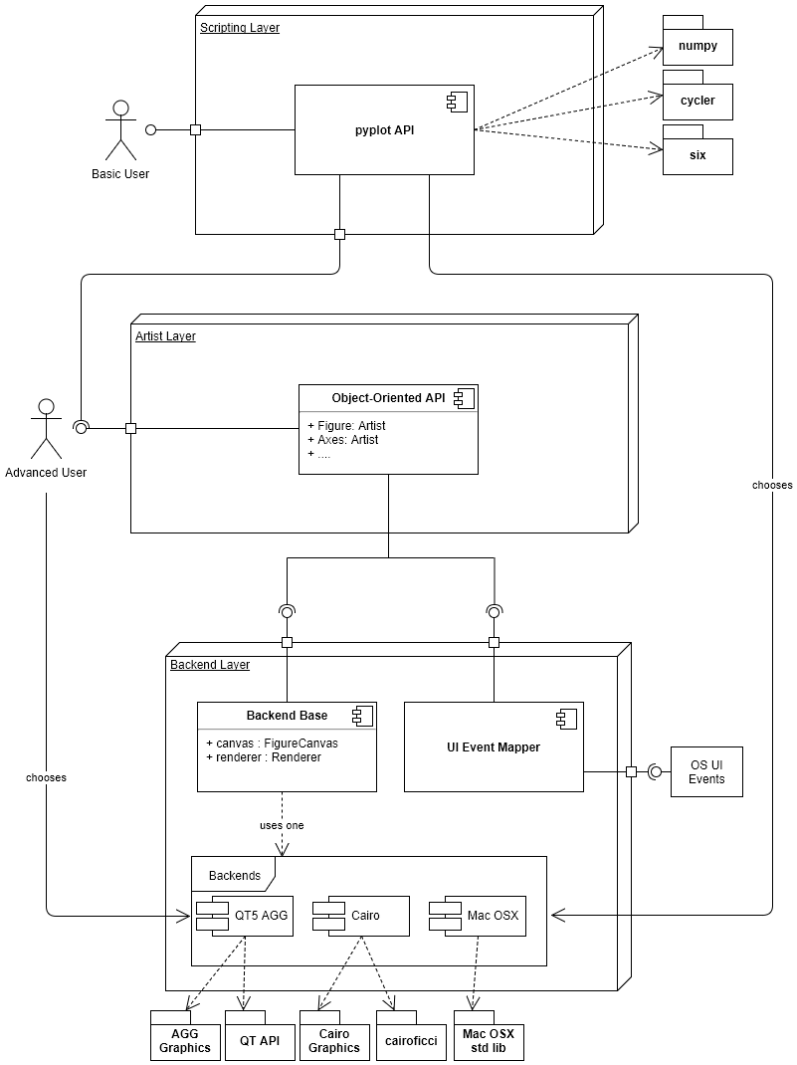


Figure 1: A component diagram of Matplotlib showing the three layers and key components

Bottom Layer: The Backend

The backbone of MPL consists of the FigureCanvas and Renderer pair. The FigureCanvas component is an encapsulation of an object on which drawings occur. The Renderer knows how to draw onto it. These are highly coupled components, as any implementation of either of them requires knowledge about how the other is implemented. There are several selectable backends currently present in MPL (each as a FigureCanvas/Renderer pair) and Figure 1 shows a few of them along with the external libraries they utilize (neither is exhaustive). Other than the requirement that a user or pyplot must choose one to use, the details of these components and their implementations are abstracted away. They only expose a relatively simple interface to the artist layer. This is integral to the high extensibility of MPL mentioned earlier, as rich features and logic can be focused on without much attention to the backend.

Event handling is also done on this layer. Like the style of abstraction used for the backend base, an interface is exposed to the artist layer to allow for the mapping of user input actions (like mouse clicks or movements) to the manipulation of Figures and its data. Developers and users only need to pass in callback functions through the object-oriented artist API.

From Figure 2, note that the FigureCanvas keeps a reference to a Figure (an artist layer object) that gets modified (by the Renderer). This is a violation of the general closed 3-layer architecture described earlier. We have components from a lower layer directly interacting with one from above, also creating a dependency cycle. When rendering a Figure, its draw() operation will be called. This will then use Renderer methods that will ultimately reference the details contained within the Figure itself. To avoid this roundabout chain of operations, one option could have been to pass in the Figure to be drawn instead. This would resolve the structural violation and result in the removal of the cycle. Mind, a non-trivial restructuring of logic would have to occur.

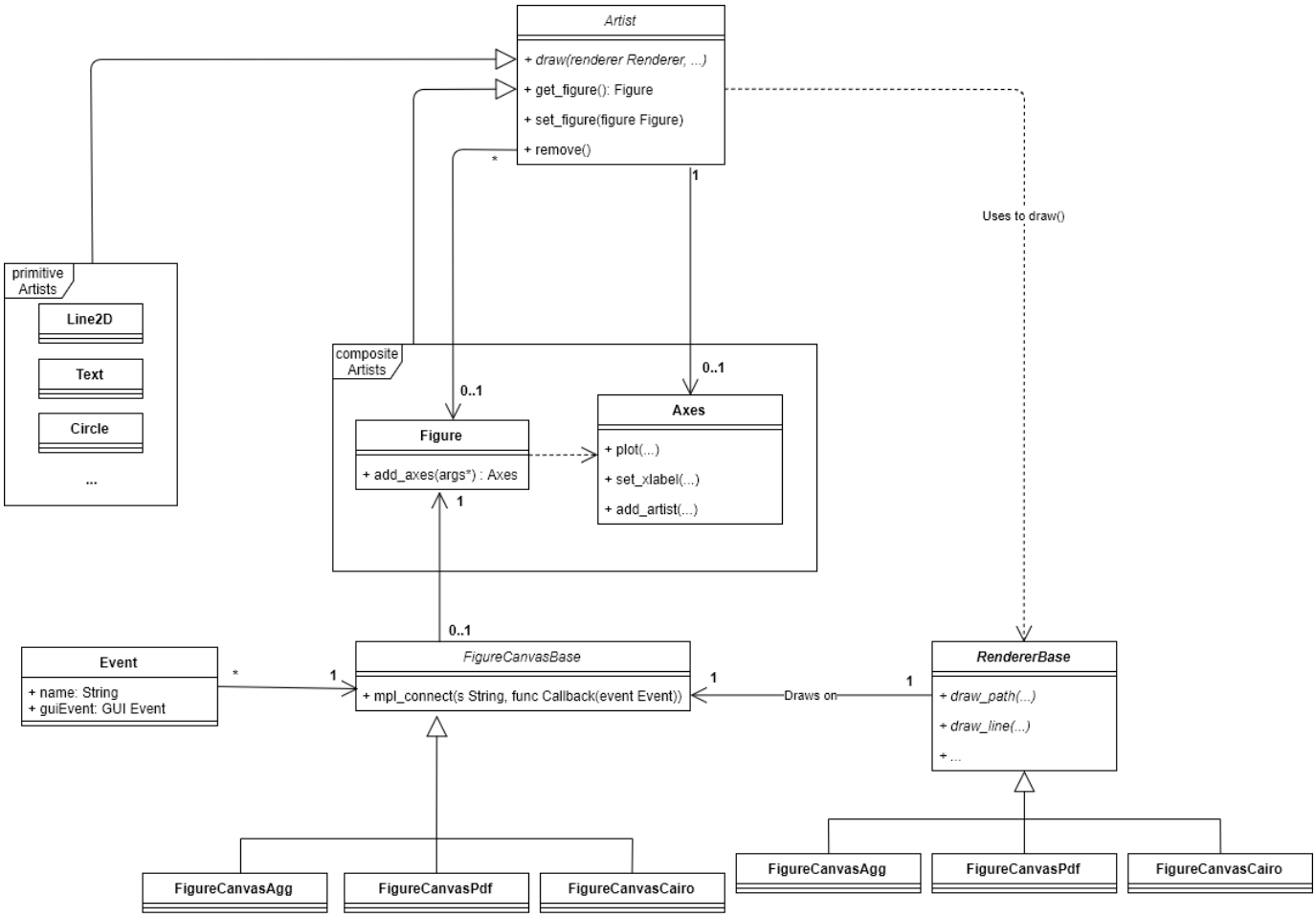


Figure 2: A class diagram showing the relationships between Artists in the artist layer and the backend.

Middle Layer: The Artist

If the FigureCanvas is a sheet of paper and the Renderer is a paintbrush, the Artist is the one who uses the paintbrush to create things on the paper. This is the analogy used in the official documentation, but there exist many classes realizing the base Artist class. Furthermore, they are named as the objects of creation rather than a creator. Since these classes are where the definitions of matplotlib plotting operations are located (each specializing in one thing), it may be more accurate to think of each of them as different artists specializing in a specific shape or component. Artists can be separated into primitives and composites; the latter being composed of other primitives and/or composites. Primitives include basic shapes like circles or lines while a Figure would constitute of a composite. Figure 3 shows an example of an MPL Figure along with the Artist classes that it is composed of (not all composites were expanded). The most important among all composites is the Axes class which contains the lion's share of responsibility and operations when plotting Figures. Axes objects directly correlate with the concept of a sub-plot in an MPL Figure, and as such contains many of the other Artist elements that make up a complete graph in the final product as well as the means to add them to itself. Artist objects allow for an object-oriented approach to using MPL, being more natural for programmers to wield and ultimately yielding better results for power users. Figure 4 shows an example script from the code base showing the sub-plotting feature along with the equivalent code when working directly with artist objects.

An oddity to the Figure object is that it keeps track of a "current" axes (retrievable via the gca() function). It

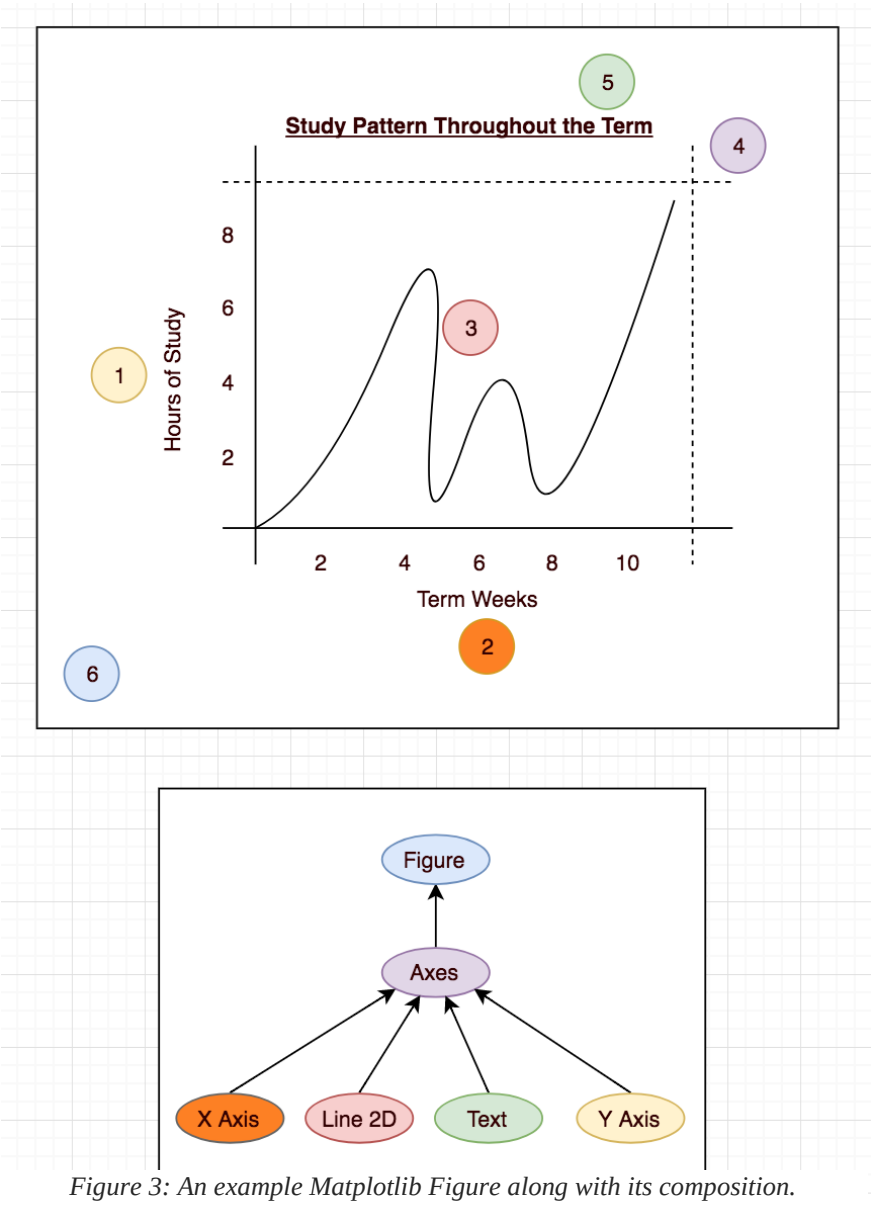


Figure 3: An example Matplotlib Figure along with its composition.

makes no sense to have this until you realize this is bookkeeping for the stateful pyplot module, located in the layer above. Instead of coupling this functionality into the Figure object, abstracting this away with an interface or directly handling the feature within pyplot would keep the object-oriented design of the Artist layer more consistent.

```
"""
==================
Pyplot Two Subplots
==================

"""
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(121)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(122)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```

```
# This script emulates the pyplot_two_subplots.py script
# included in the folder matplotlib/lib/mpl_examples
# from the Artist layer

import numpy as np

# Choose a FigureCanvas from any backend, attach figure to it
from matplotlib.backends.backend_pdf import FigureCanvasPdf as FigureCanvas
from matplotlib.figure import Figure

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)


# initialize FigureCanvas and attach it to the selected Figure()
fig = Figure()
canvas = FigureCanvas(fig)


# Call Figure method to add subplot, creates Axes
ax1 = fig.add_subplot(121)

# Use Axes method plot() to plot
ax1.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

# repeat for second subplot
ax2 = fig.add_subplot(122)
ax2.plot(t2, np.cos(2*np.pi*t2), 'r--')

# save
fig.savefig('pyplot_two_subplots.pdf')
```

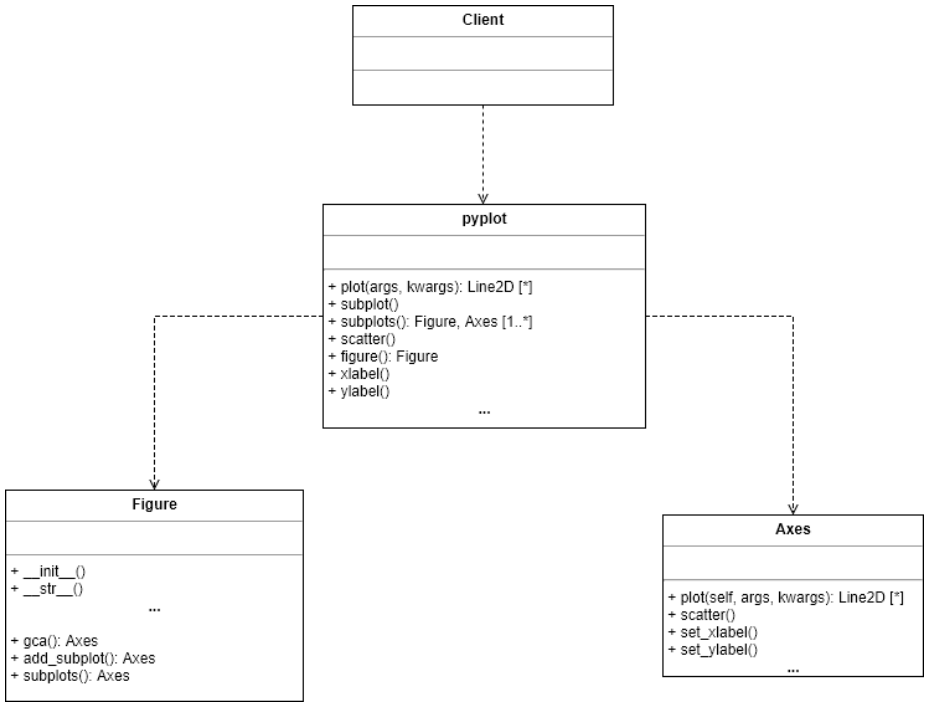
Figure 4: Use of pyplot (left) and the lower-level code that replicate the behavior (right) of plotting two simple sub-plots onto a figure. Note the explicit choice of the PDF backend, Axes operations, and how pyplot works with a single Axes at a time.

Top Layer: pyplot

Packing the artist layer concisely into a stateful user-friendly interface is pyplot. All the technical details of manually manipulating Artist objects is abstracted away. The interface handles one Axes at a time. Selection of the backend base is done based on user settings or inputs.

Examples of Design Patterns

Façade



Module:

matplotlib/lib/matplotlib/pyplot.py

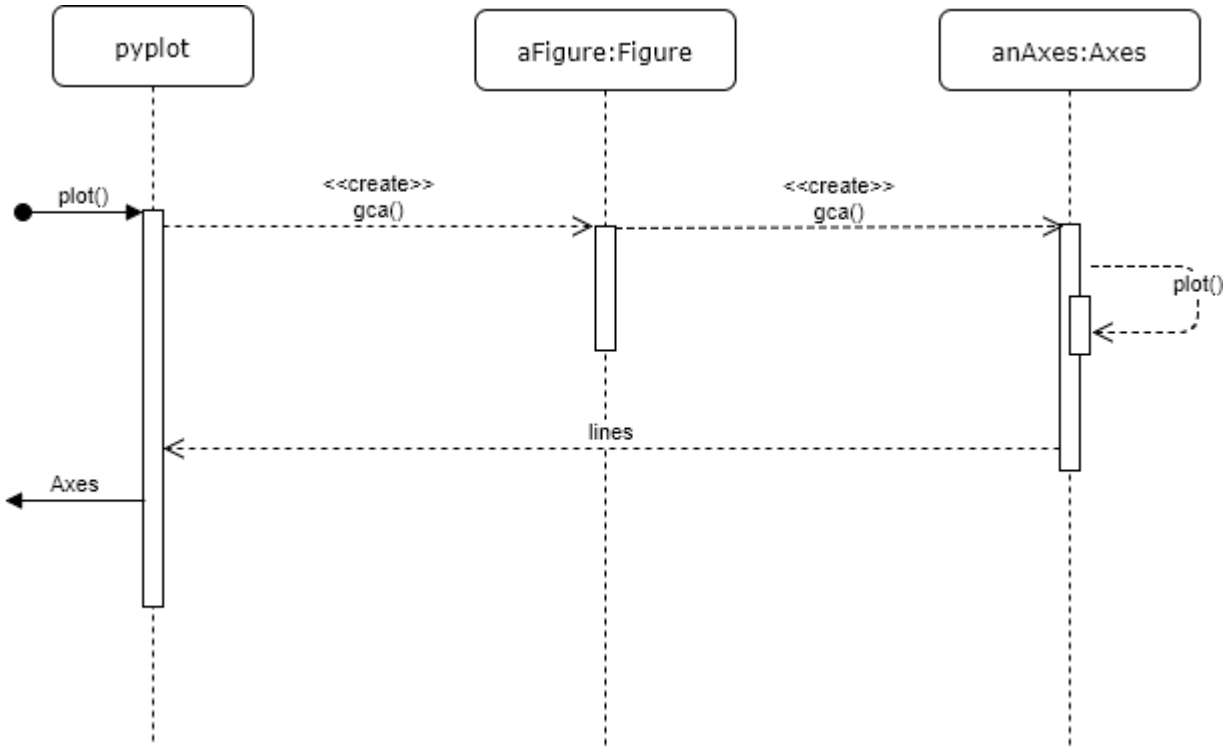
Description:

pyplot implements the façade design pattern, acting as an interface for a more complex subsystem involving classes like Figure, Axes, Subplot, Artist, and others.

By offering clients the option to interact with a single interface rather than multiple classes, the system becomes much easier to use. Instead of using methods from the façade's various dependencies, clients can just use pyplot's methods, which have the classes of the subsystem plot points, draw bars, label axes, save figures, etc. This also weakens coupling between the clients and the subsystems.

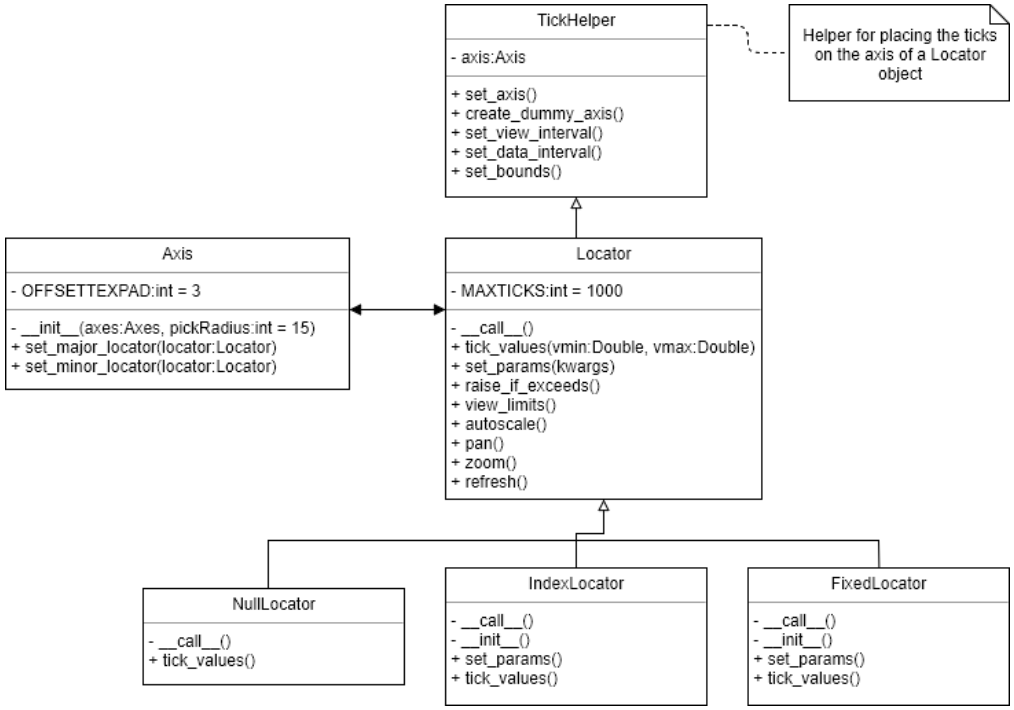
Justification:

pyplot, like all façades, is an alternative interface, not a replacement – subsystem objects are shielded, but clients are not restricted from accessing them. Clients can still directly interact with the subsystem to manually create Figures, Axes, and plots if they wish.



The façade design pattern may be confused with the mediator design pattern because they both abstract the functionality of other classes. However, the mediator facilitates communication between peers (and it is therefore known to the peer classes), while façade provides interface to a subsystem, and isn't known to the subsystem classes. Although there are some helper methods in the subsystem made to help pyplot act as a façade, like Figure's _gci(), the functionalities of the subsystems are independent from pyplot, and so pyplot is firmly an implementation of the façade design pattern.

Bridge



Modules:

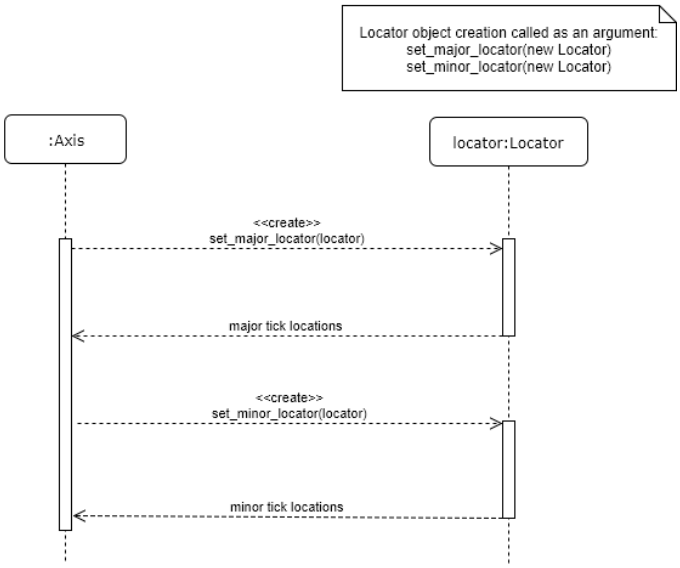
matplotlib/lib/matplotlib/ticker.py

matplotlib/lib/matplotlib/axis.py

Description:

The Locator class is used to design and determine the locations of the major ticks (scale numbers to explicitly show) and the minor ticks (scale numbers that are not shown and appears between the major ticks) of an axis. In other words, the overall purpose of the Locator class is to define the scale that is used and seen on an axis of a graph. The Axis class is used to define an axis of a graph.

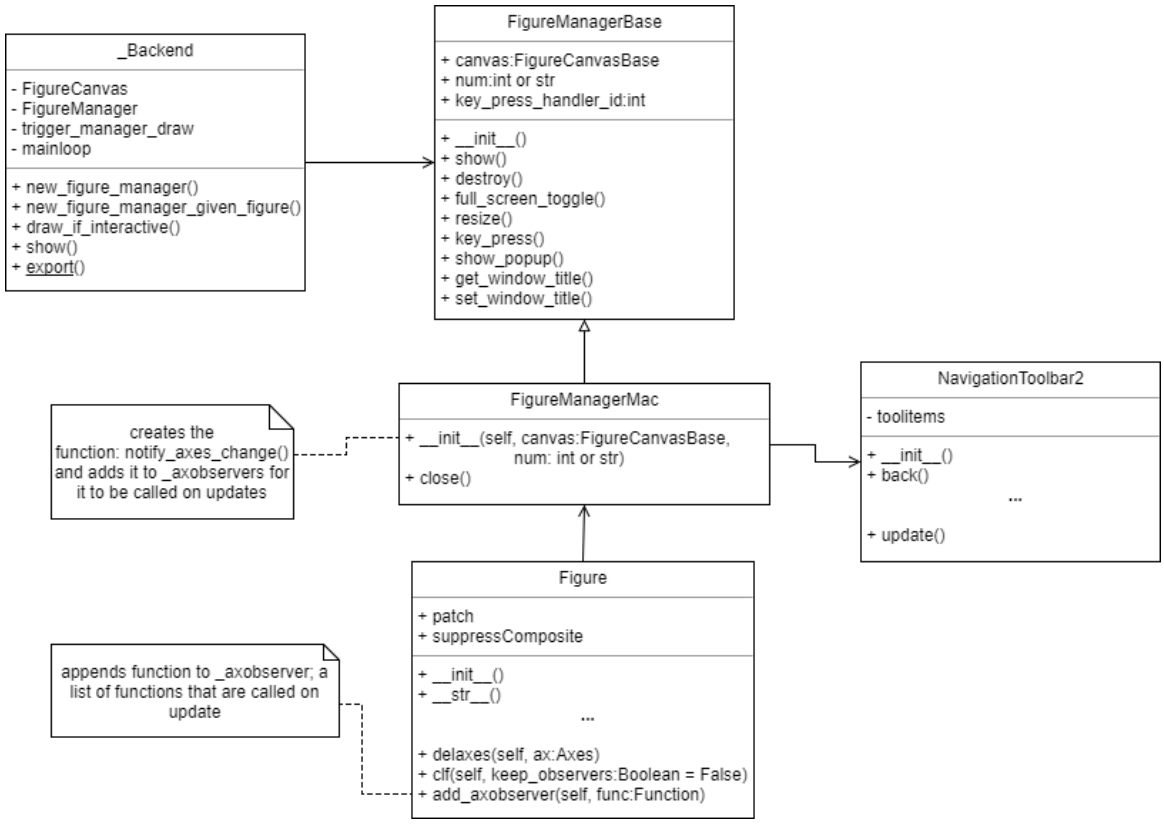
The Locator class utilizes the Bridge design pattern. To recap, the Bridge design pattern is a structural design pattern that prefers composition over inheritance so that two abstract implementations can vary independently. An example is to have a single webpage with different selection of themes. One approach is to create two WebPage objects with one having a light theme, and the other a dark theme. However, by using the Bridge Design Pattern, a new Theme class is created. Theme objects are then used to change the theme of a WebPage object without creating multiple WebPage objects with different themes. In this case, each Axis object can use one Locator object and it can be interchanged with other Locator objects. Keep in mind that any Axis object can use any Locator object. The Axis object is analogous to the WebPage object and the Locator objects are analogous to the Theme objects.



Justification:

The Locator and Axis classes design pattern can be confused with the Composite or Decorator design pattern. The pair does not follow the Composite design pattern since the two classes cannot be treated as a single object and that the two classes must be independent from each other. It cannot be the Decorator design pattern because when a behaviour to a Locator object is added, it will change the behaviour of the underlying Locator objects. Moreover, an Axis object doesn't create a Locator object, it is assigned to it. The two classes have two different responsibilities, Locator class for setting the tickers of the axis, the Axis class to define the axis of the graph. This does not conform to the Decorator design pattern.

Observer



Module:

matplotlib/lib/matplotlib/backend_bases.py
matplotlib/lib/matplotlib/backends/backend_macosx.py
matplotlib/lib/matplotlib/figure.py

Description:

The Observer pattern is used to notify relevant classes whenever a change is made to a figure's axes. FigureManagerMac, upon initialization, creates a toolbar and an accompanying operation notify_axes_change() that updates the toolbar. notify_axes_change() is then added onto the active Figure's list of observers. Whenever the axes state of the Figure changes, its observed functions are automatically called. notify_axes_change() will update the toolbar.

Justification:

Matplotlib allows working with multiple axes, and it is possible to create any number of axes onto given figures. It keeps track of current axes, and all plotting commands apply to those current axes. For this reason, it is crucial that consistency is maintained between the relevant classes whenever the axes are changed, making the observer pattern the correct pattern to utilize.

This implementation has a small difference to the observer pattern from class/textbook. Instead of having a list of observers and calling the observer's update function, it keeps a list of functions (notify_axes_change) and calls it with the parameter self in the figure.