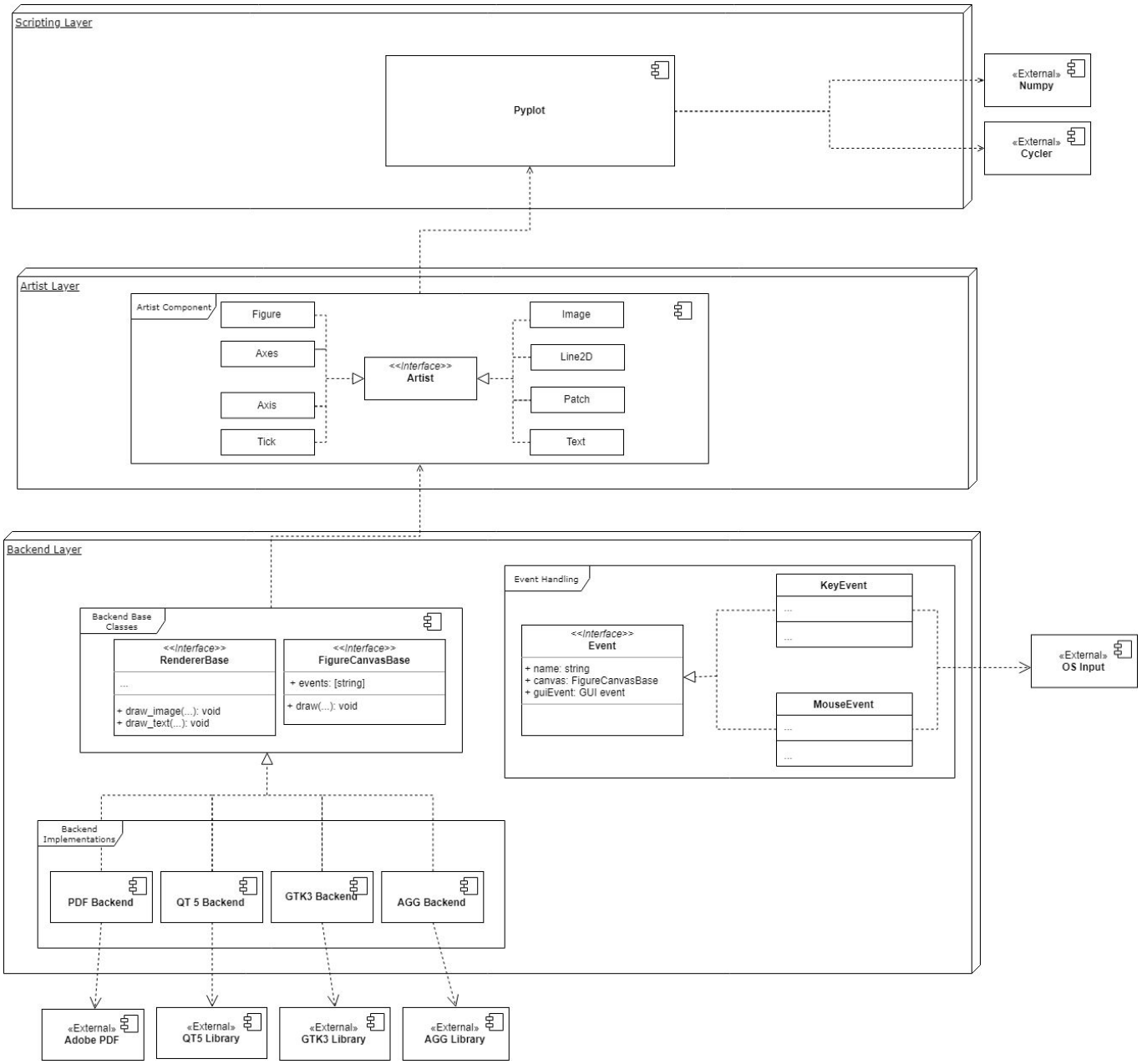
Deliverable 1: Matplotlib Architecture

Benjamin's Grooper
CSCD01
Anya Tafliovich
David Gao, Simon Ngo, Andrei Grumazescu, Raunak Mathur, Tony Ng
Winter 2019

Table of Contents

Software Architecture

Matplotlib's architecture is structured into three layers. These layers from top to bottom are Scripting, Artist, and Backend. These layers are encapsulated such that any one layer is generally only able to access the layer immediately below it.



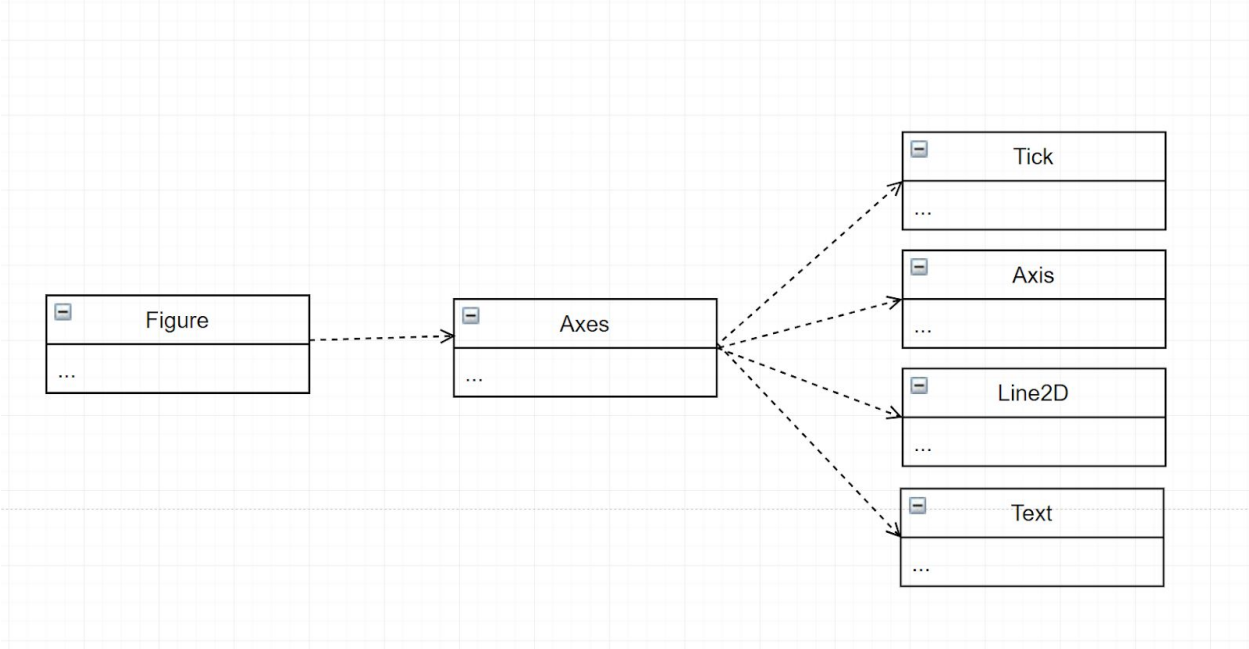
Backend Layer

This layer essentially handles the I/O. MPL was designed to work with a variety of different use cases such different user interface libraries. This is manifested in the many different implementations of the backend. There are two types of backends, “User Interface Backends” and “Hardcopy Backends”. User Interface Backends allow for user interaction and live on-screen output. In contrast, Hardcopy Backends generate files as their outputs.

All of these backends implement a few key Base classes. Amongst the most important classes, is the **FigureCanvas** serves as the “Canvas” on which the figure is drawn on, it contains the specific user interface’s details. The **Renderer** is responsible for taking drawing instructions and creating a render of the figure. Additionally this layer handles **Event**s which are primarily inputs. A few examples of these includes **KeyEvents** and **MouseEvents**.

Artist Layer

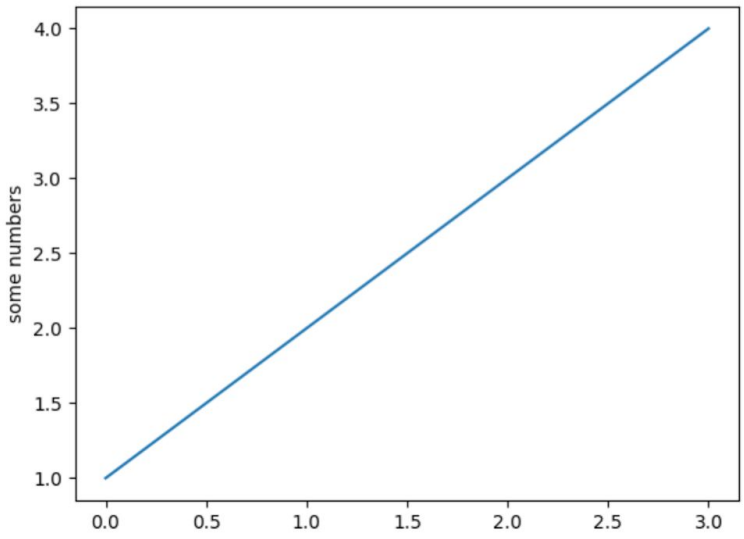
The artist layer is in the middle of the stack. **Artists** are visible elements that are rendered onto the **FigureCanvas**. An **Artist** creates images using the selected **Renderer** from the backend to create the graph. There are two types of **Artists**: primitives and containers. Primitives represent graphical objects. Containers hold and arrange **Artists** into a larger image. **Figure** is the top level container which is utilized by the Backend to generate the final output. It can hold multiple **Axes** containers which is where most of the elements in the figure are stored. Some of **Artists** that **Axes** contains are **Axis**, **Ticks**, **Lines**, **Polygons** and **Texts**.



Scripting Layer

The scripting layer is responsible for wrapping the features from the underlying layers into a simpler interface for basic end-users. Pyplot offers a suite of command-like functions for use with interactive plots and simple static plots. It allows the user to create plots without understanding the underlying object oriented structure.

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```



Interesting Aspects of Architecture

A well designed architecture, as defined in class, should maximize the cohesion within modules and minimize coupling. As mentioned previously the architecture of Matplotlib is split into three different layers for each layer of the application. Each layer, however, holds many interconnected objects that share resources to complete a goal.

An example would be Figure 2.1 show below, where Artist is an interface implemented by many artists in the Artist layer and FigureCanvasBase is an object used by Event as well as RenderBase in the Backend layer. Looking at each layer as a module, the backend layer *only* handles processing, the artist layer *only* creates the necessary objects related to rendering and passes it to the backend layer and the scripting layer is *only* responsible for input and output processing through an easy to use user interface. Not only does this guarantee that the elements inside a module belong together, cohesion, but it also makes it so that components that are no relevant to each other are separated by modules, coupling. Out of all the classes within Matplotlib, only a select few interact between two layers and not a single class that interacts with all three layers has so far been discovered, reducing coupling. The classes that do interact between two layers are mainly used to act as a gateway to relay data between the layers, such as the class **Figure** in figure 2.1.

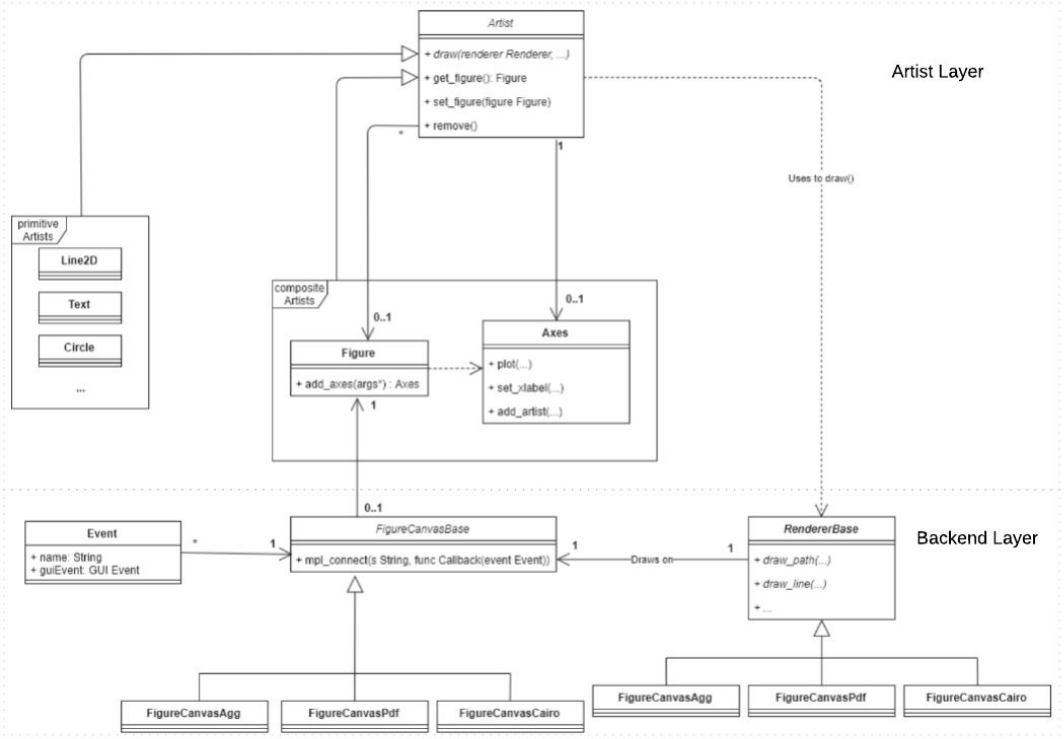


Fig 2.1: A simplified class diagram of the Artist layer and its interconnected objects. This figure is used as an example of the Artist layer and many classes have been exempted for clarity.

Critiques of the Architecture

Although Matplotlib has managed to separate majority of their functionalities into three main layers, they also introduced their version of abstract data types that they use throughout their programs. These abstract data types are all under the module **cbook** and the main ones that are used throughout the program are **_OrderedSet**, **maxdict,** and **silent_list**. These classes are used to replace ordered set, dictionaries and lists in their respective uses. In the method parameters they accept the data type it is trying to represent, but it does not inherit from the Collections class. This means if the Collections class were to update in the future, Matplotlib would possibly be required to update their implementation of these classes, which could affect their entire code base.

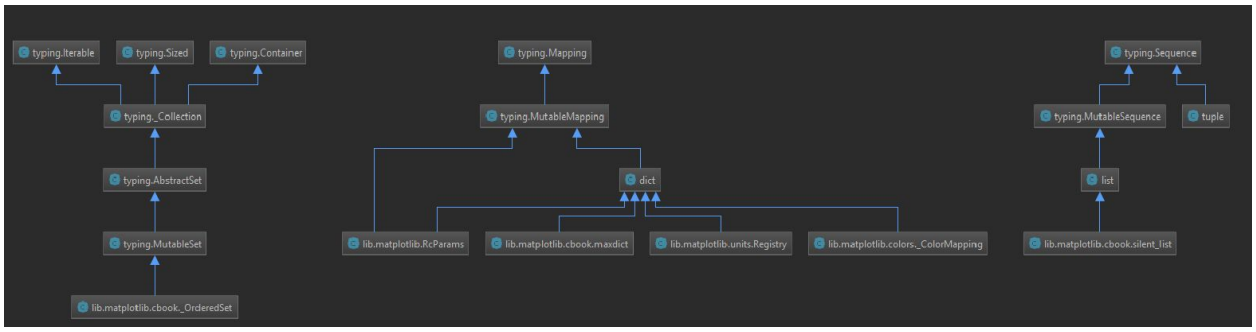


Fig 3.1: **_OrderedSet**, **maxdic**, **and silent_list** are completely independent of other modules within Matplotlib. Code indicates the relationship is association and not inheritance.

Suggestions for Possible Improvements

Matplotlib should create adaptor for the MutableSet, dict and list classes rather than rewriting them entirely. This way if the functionality within the adaptee were altered, there would not be much work to maintain the working status of the code base.

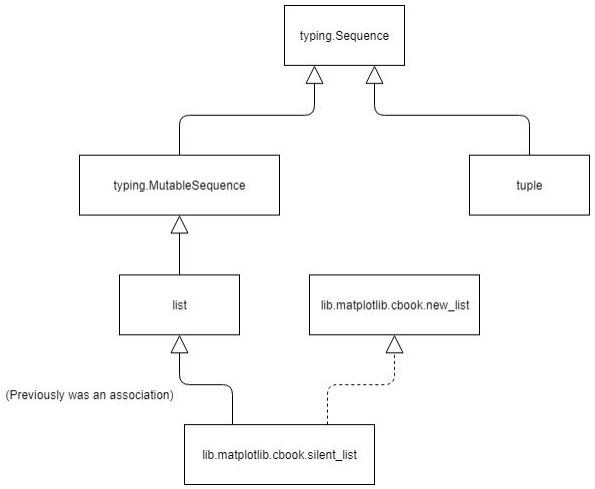


Fig 4.1: A UML diagram for the suggested design pattern of **_OrderedSet**, **maxdic**, **and silent_list**.

Design Patterns

Decorator Design Pattern

An example of the Decorator design pattern can be observed in /lib/matplotlib/animation.py inside the MovieWriterRegistry class. This class handles the registration of MovieWriter objects. A MovieWriter object grabs image information from a figure and saves it as a movie frame to be eventually sent to a pipe. Some examples of MovieWriters include PillowWriters which saves animations as endless looping gifs and FFMpegWriters which stream to FFmpeg. Since there are many types of MovieWriters, a MovieWriterRegistry class was created to register them.

The method “register” in MovieWriterRegistry utilised the Decorator design pattern by taking a name as a parameter and a writer class that it wraps up. Normally, a writer would have to be named and added to a dictionary of available writers on its own at some point during its initialization. It may also need to check that the writer class is actually available to be added. MovieWriterRegistry will instead provide a MovieWriter class the ability to add itself to the registry upon initialization with a simple and human readable name.

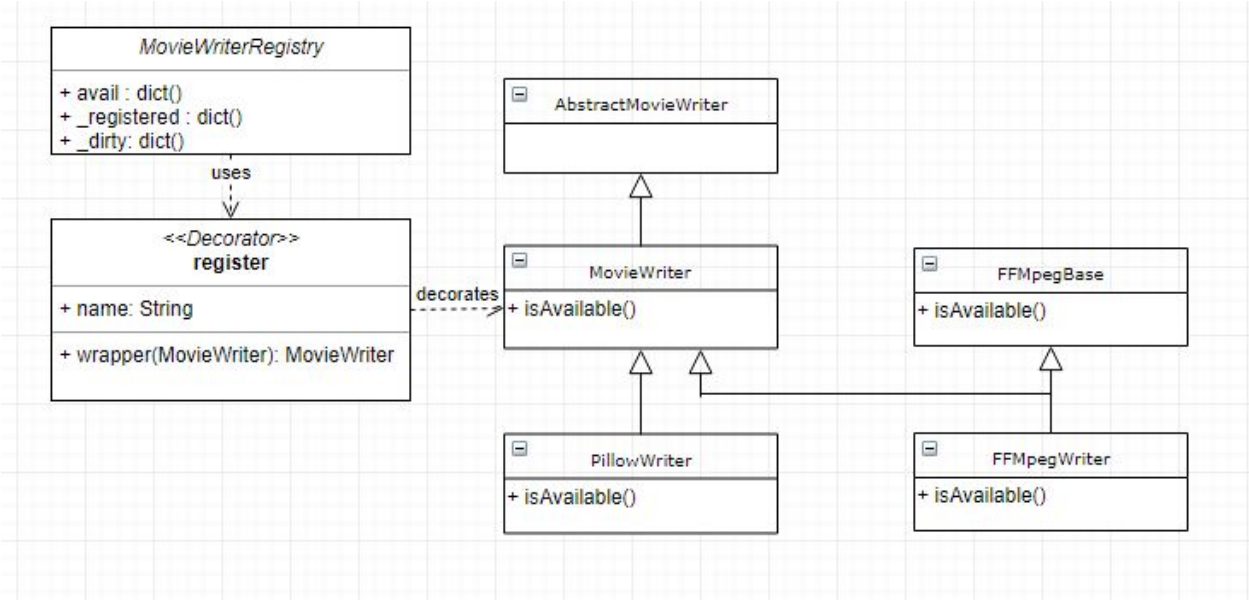


Fig 5.1: A UML diagram for the decorator design pattern used in animation.py

The classes mentioned above can be found at
https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/animation.py

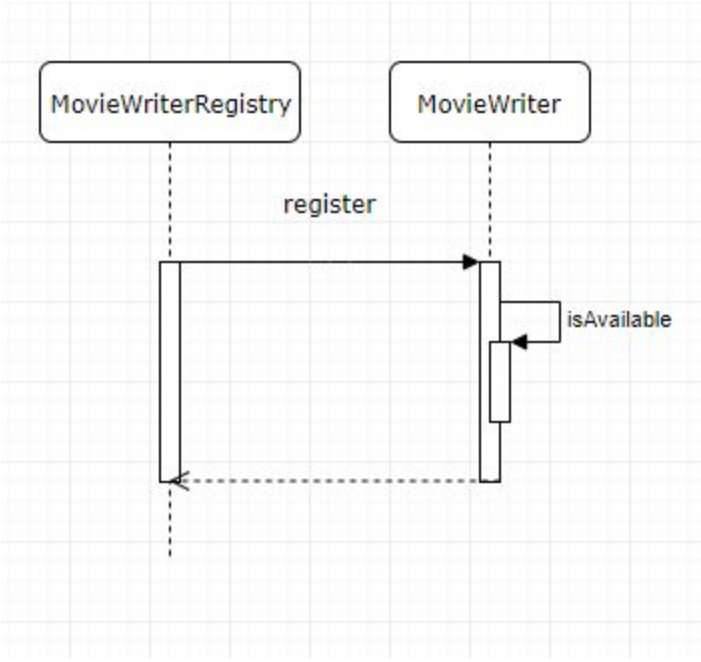


Fig 5.2: A sequence diagram of the decorator design pattern used in animation.py

Strategy Pattern

A use of the Strategy Pattern was also found within the transforms.py file. This pattern was evident through the interaction between Affine2D class and its subclasses IdentityTransform, BlendedAffine2D, CompositeAffine2D, BboxTransform, BboxTransformTo, BboxTransformToMaxOnly, BboxTransformFrom, ScaledTranslation which inherited attributes from Affine2DBase. Each class implemented their own algorithm for the get_matrix() method and allows the application to select the desired choice during runtime. In this case the Affine2DBase class is decoupled from its subclasses mentioned above.

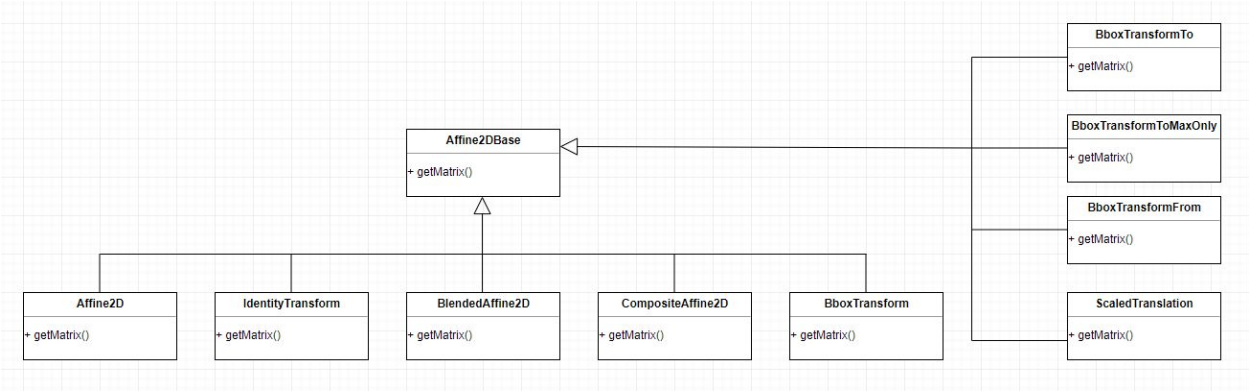


Fig 5.3: A UML displaying the Strategy Pattern where only the getMatrix() method is shown within each class for simplicity.

The classes mentioned above can be found at
https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/transforms.py

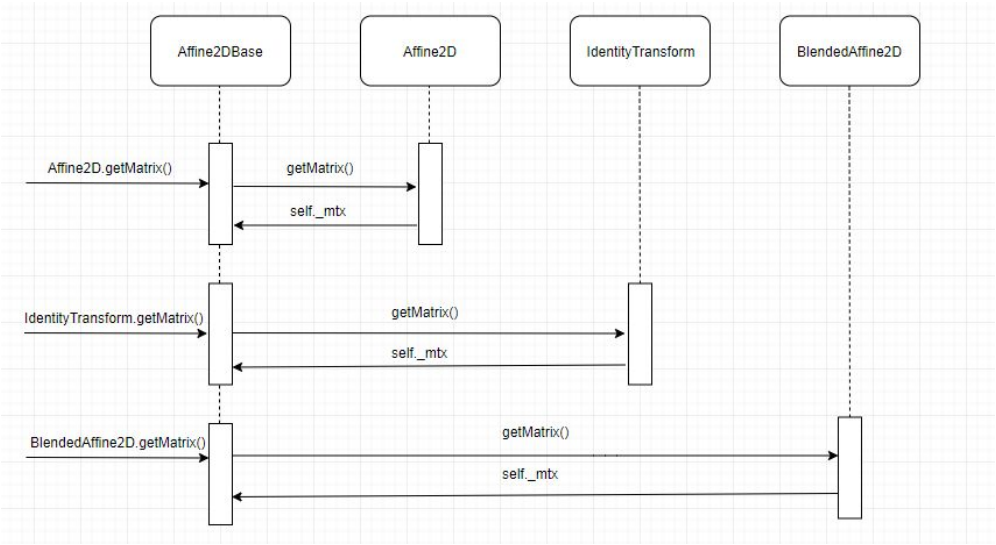


Fig 5.4: A sequence diagram for the Strategy Pattern in transforms.py. Only Affine2D, IdentityTransform and BlendedAffine2D are shown as the other classes follow the same process.

Facade Pattern

The pyplot file implements the facade design pattern by acting as an interface for the subsystem of classes which includes classes such as Artist, Figure, and Axis. pyplot allows for interaction with important parts of the whole system through functions defined within the pyplot file. This allows the system to be much easier to use by having to only use this single interface when performing actions such as deciding whether or not to display grid lines on the plot that is displayed to the user. The user can simply import pyplot and call the grid function with a boolean value as the argument and pyplot will know to use the grid function within the Axis class. Having these useful functions within pyplot also reduces the coupling between the users and the subsystem.

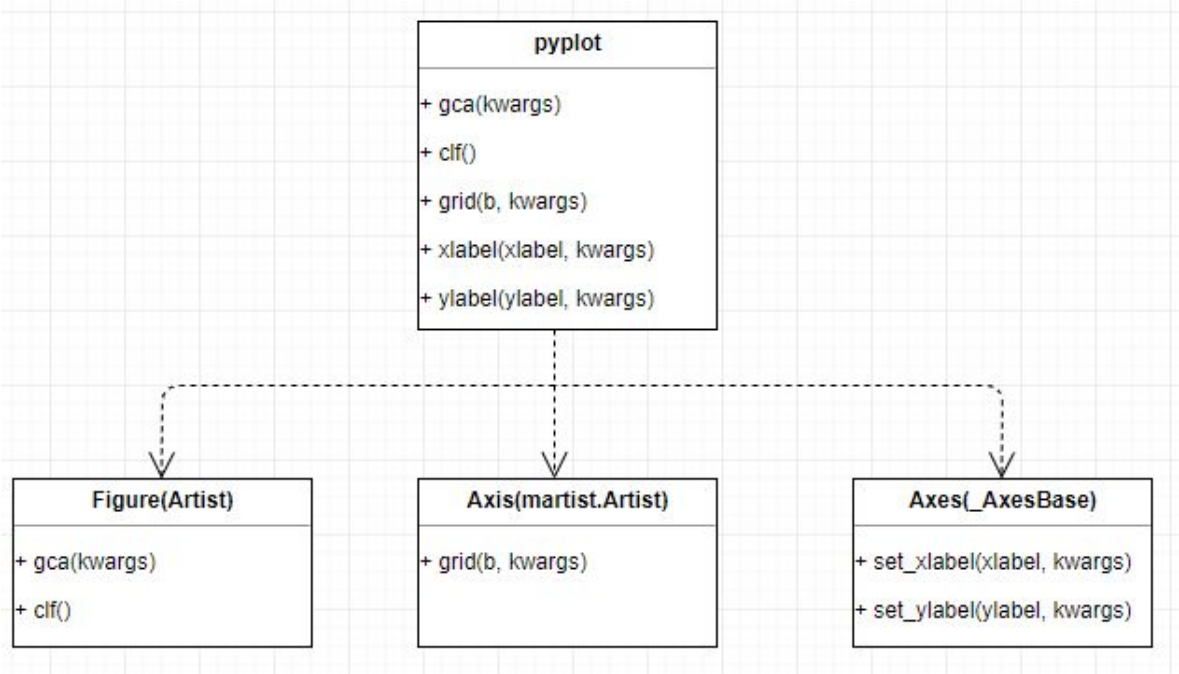


Fig 5.5: A UML displaying the interaction between pyplot and some classes within the subsystem along with example methods.

The file mentioned above can be found at
https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/pyplot

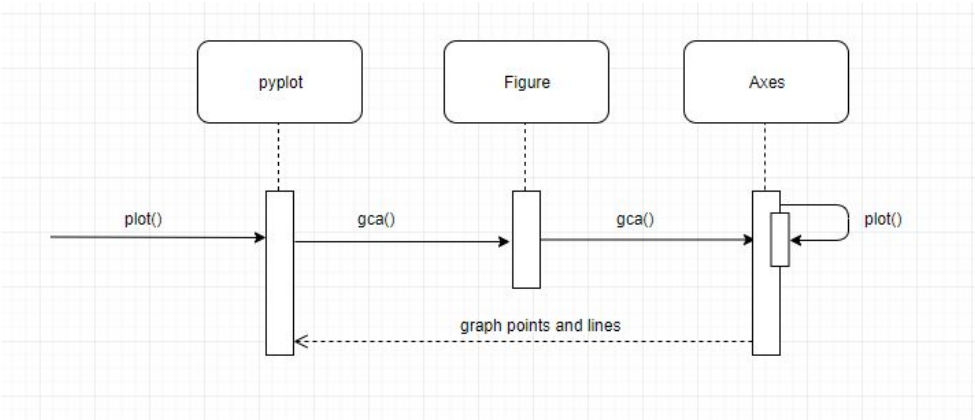


Fig 5.6: A sequence diagram showing the Facade Pattern for pyplot using the plot method for graphing.