MATPLOTLIB: PROJECT DELIVERABLE 2

Team 16 (Do It Tomorrow)

CSCD01

February 10th, 2017

Team Members:
Mengzhe Lu
Jubin Patel
Derek Chow
Daniel Karas
Gauravjeet Kala

Table of Contents

Project Tools

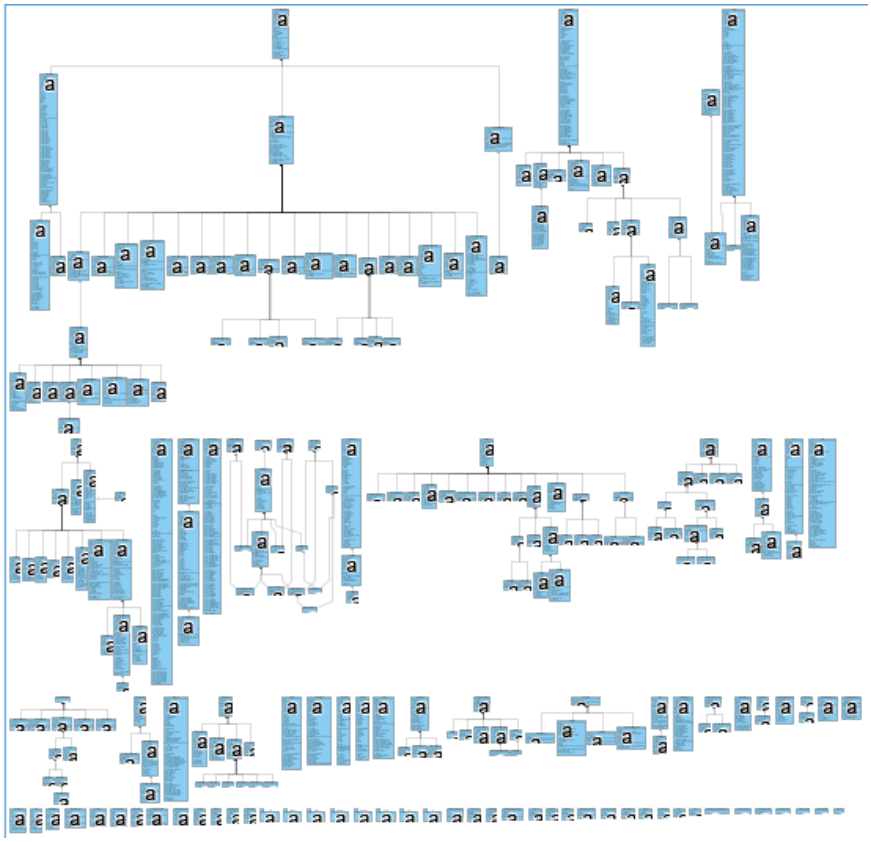
As a team, we have discussed and decided to have the following applications to aid us in developing our diagrams:

1. Visual Paradigm
    - We will be using Visual Paradigm as a reverse-engineering tool to a UML class diagram representing the classes, subclasses, relationship and associations.
    - After looking at other alternatives such as PyCharm, our team decided on Visual Paradigm as it was more simpler to use and easier to reverse-engineer the entire matplotlib library

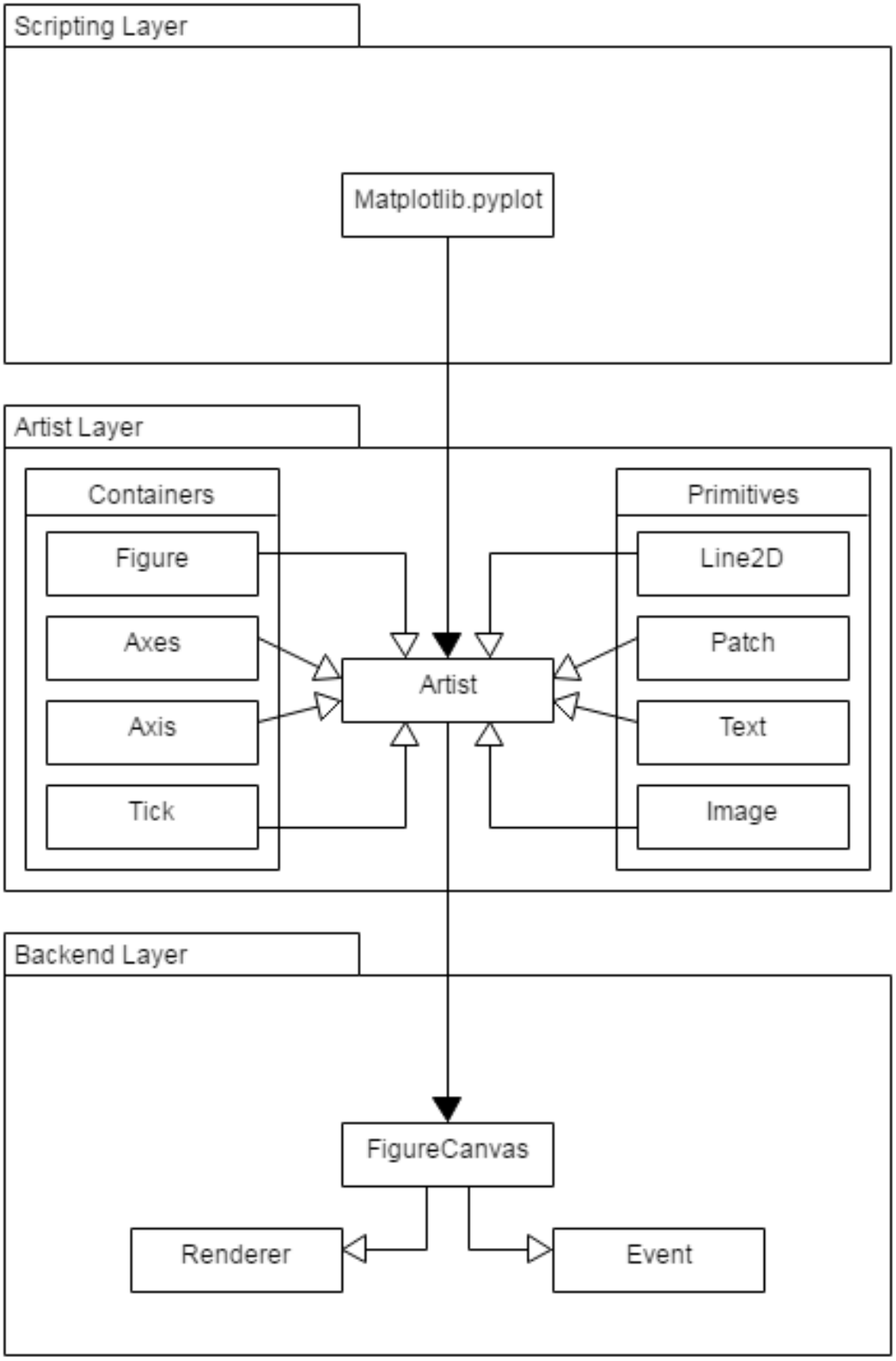
2. UMLetino
    - We will be using UMLetino to build some of our structural diagrams for our discovered design patterns. Several members found it much easier to use from other alternatives such as VisualParadigm and Gliffy

3. Gliffy
    - We will be using Gliffy to build our behavioral diagrams for our discovered design patterns. Several members found it much easier to use from other alternatives such as VisualParadigm and UMLetino lacked the capabilities to produce such diagrams

Visual Paradigm's Reverse Engineering of matplotlib:



System Architecture Diagram



System Architecture

The system architecture for matplotlib is divided into three layers. The three layers are organized in a stack, where each layer only knows the existence of the layer that's strictly below it. The three layers that complete the matplotlib's architecture, from top to bottom, are: **the scripting layer, the artist layer and the backend layer.**

Backend Layer:

The backend layer's primary task is the handling of the user interface. In this case, the backend layer provides matplotlib a **FigureCanvas** surface and allows the **Renderer** to provide the user a "paintbrush" to paint onto this **FigureCanvas** surface. The backend layer is mainly composed of two types, the user interface backend as well as the hardcopy backend. The user interface backend handles UI **Event**s such as key presses, button presses and mouse clicks and movements. Whether matplotlib is embed into a graphical user interface or being used interactively from python shell, the backend is capable of distinguishing these use cases and return corresponding outputs. The hardcopy backend is mainly used for creating file representations of images (PNG, SVG, PDF, PS, etc.).

Artist Layer:

The Artist layer's main responsibility is for the creation of a **Figure**. The Artist communicates and uses the **Renderer** (*paintbrush*) of the backend layer to provide image representations of the **FigureCanvas** surface produced by the backend layer. This is the layer where plots are constructed by combining several Artist objects (separated by primitives and composite [see system diagram]) such as **Line2D** and **Rectangle** to create a **Figure**. Everything displayed under a matplotlib **Figure** (title, lines, ticks, images, etc.) is an instance of an **Artist**, where it uses its *draw()* method to paint the created **Figure** onto the **FigureCanvas** surface.

Scripting Layer:

The scripting layer's primary responsibility is to simplify the common tasks of working with other layers including the drawing and visualization of a **Figure**. For instance, when importing pyplot, it'll select the default backend for your system or the one that's previously configured as well as calling a setup function. Subsequently, the pyplot interface defines several functions in its setup function, such as *plot()* and *title()* to aid in the development of a **Figure** rather than generating a **Figure** with only the other two layers.

Quality of Architecture

- matplotlib's was designed to what we define as a good system architecture as it minimizes the coupling between modules as well as maximizing the cohesion between each module.

matplotlib displays these good architecture attributes by:

Coupling:

- matplotlib's 3-layer system architecture plays a large role in minimizing the degree of coupling.
- matplotlib's layers only know of the existence of the layer that's strictly below it instead of having multidimensional coupling. As seen in the system architecture diagram, the scripting layer only knows the existence of the artist layer and the artist layer only know the existence of the backend layer.
- Therefore, it produces minimal coupling by only having the top layer depend on the layer below it and so forth.

Cohesion:

- matplotlib's 3-layer architecture displays a high degree of cohesion within the system. The artist layer and backend layer maximizes cohesion of each module by having their module's content be strongly inter-related.
- For example, the artist layer is strongly inter-related in the sense that it contains many sub-components that shares the same goal of drawing onto the FigureCanvas provided by the backend.
  These sub-components are shown in the artist layer of our system architecture diagram.
- The backend layer also demonstrates similar cohesion with its ability to handle user events and convert such events to an Event class, as well as rendering commands onto a FigureCanvas instance.

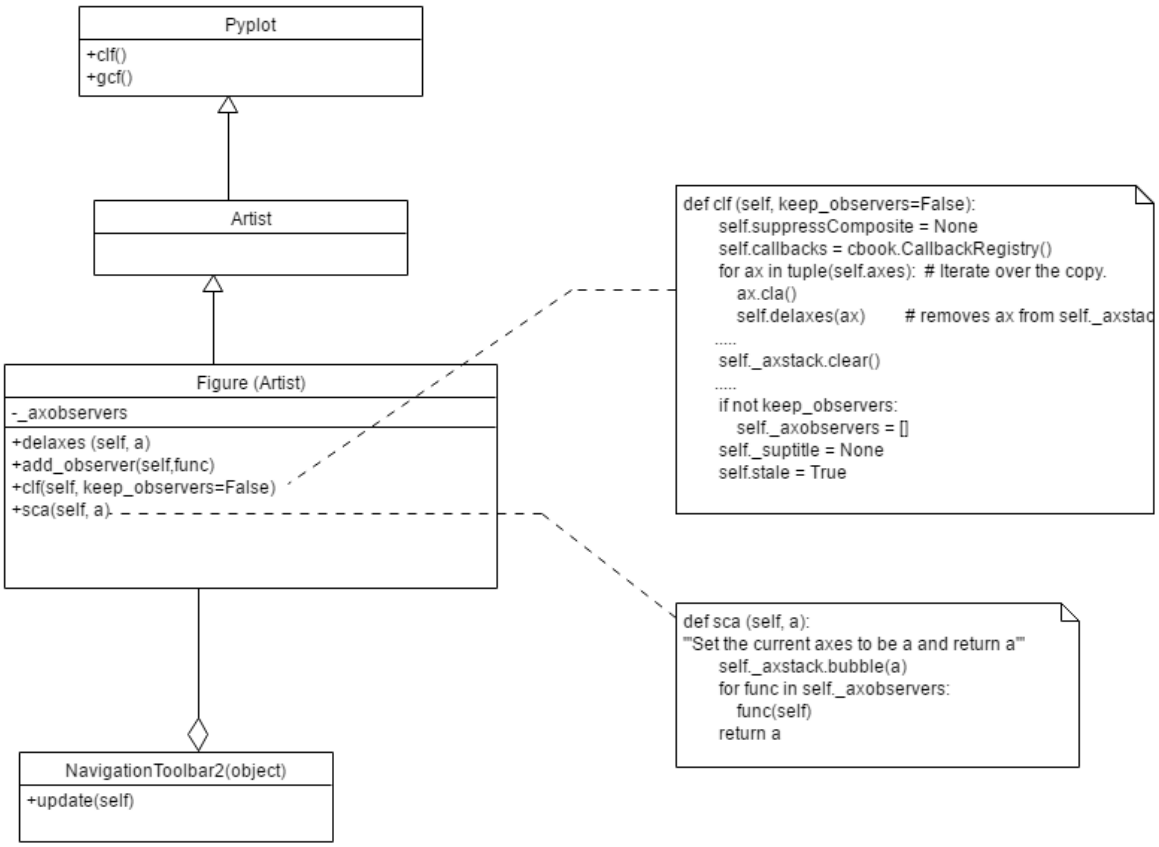
Observer Pattern

- An observer design pattern was evident in figure.py as it was executing functions (mainly notify_axes_change()) whenever the axes were changed in a Figure. The functions added to _axobservers' list comes from the observers that were using the Figure class. Note that in the backend files, whenever a Figure is created, the FigureManager will append the function, notify_axes_change() to _axobservers of that Figure instance.

- This design pattern is very different to the observer design patterns we've seen, and what makes it interesting is that it maintains a list of functions of the observer classes (mostly notify_axes_change()) rather than our traditional way of the observer design pattern.

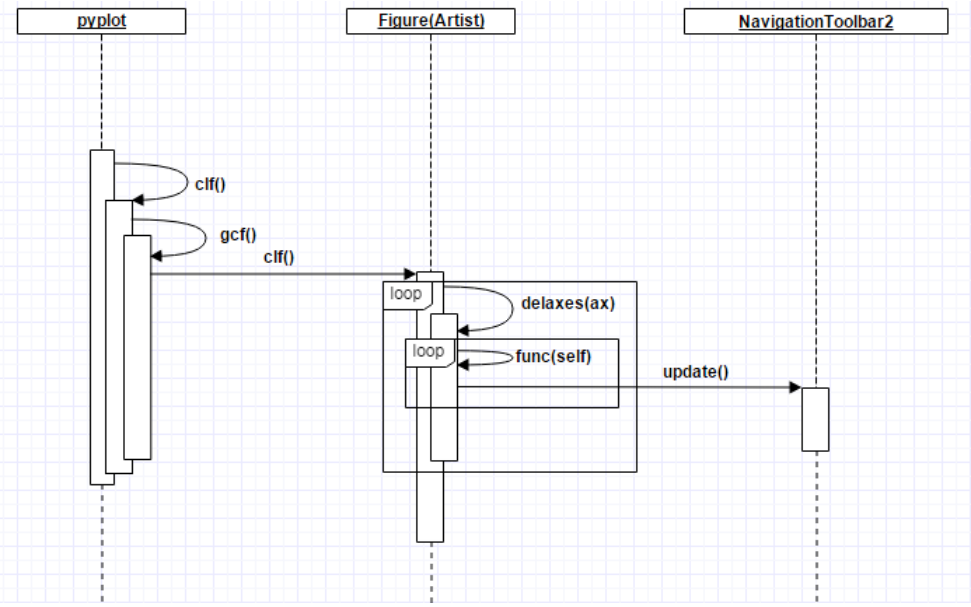
UML Diagram:

For our structural diagram, we've included the relationships between pyplot, the Figure class and the NavigationToolbar2 class. In this case, when pyplot calls clf(), it calls the current Figure's clf() method and in turn it loops to delete all the Figure's axes. This notifies all the observer classes by calling all the functions in self._axobservers.



Sequence Diagram:

The sequence diagram below shows the calls when pyplot calls its clf() method:



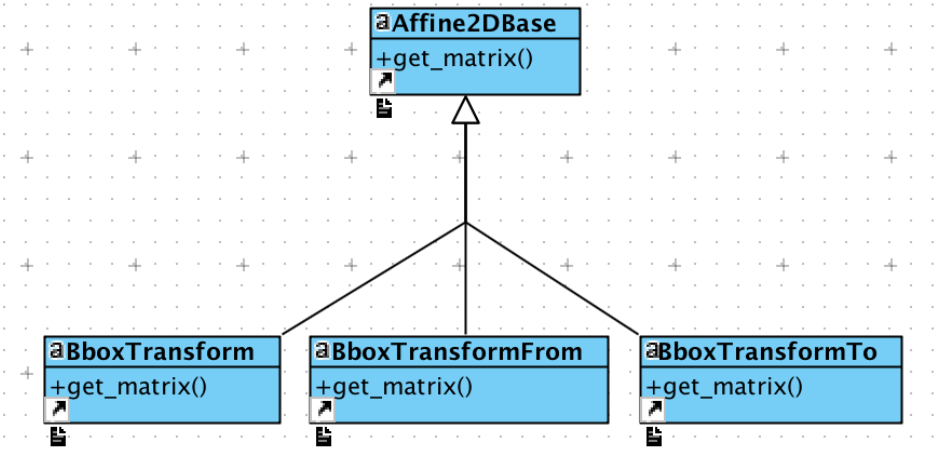
delaxes() method executes all functions in _axobserver:
https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/figure.py - L813

Strategy Pattern

- A strategy pattern was found in transforms.py where the BboxTransform, BboxTransformFrom and BboxTransformTo classes inherit most of their attributes from the Affine2DBase class. Each of the classes implement their own get_matrix() method, thus following the strategy design pattern.

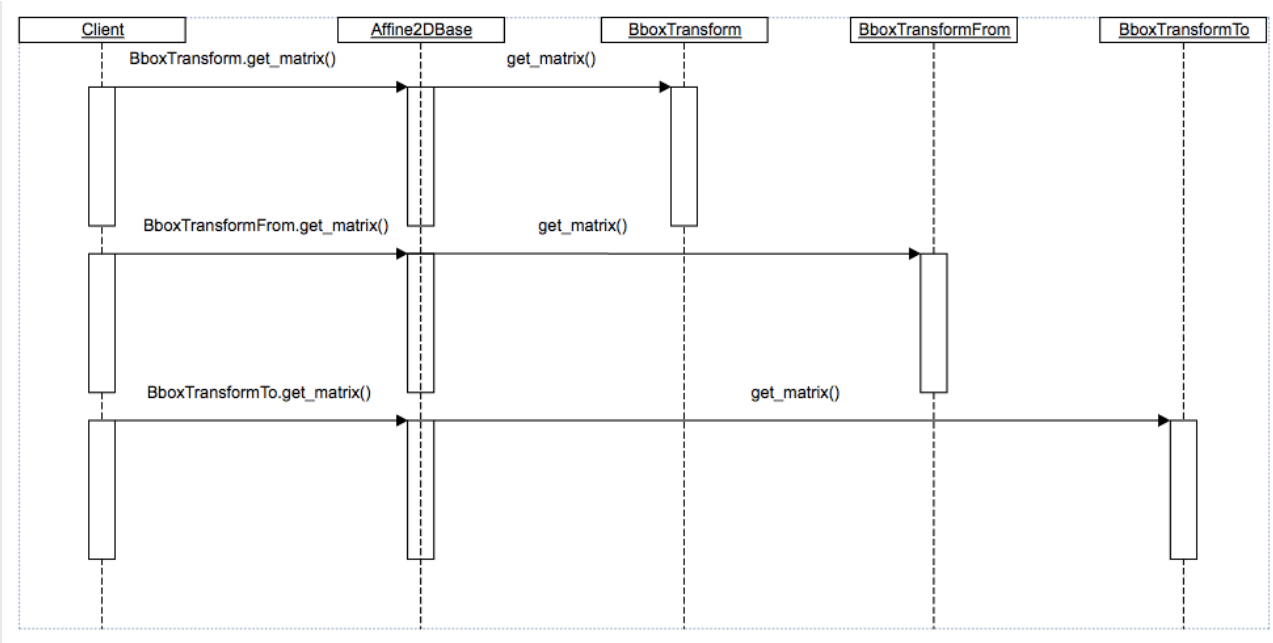
UML Diagram:

For our structural diagram, we've included the relationship between BboxTransform, BboxTransformFrom and BboxTransformTo. We've excluded excess attributes and methods for a simplistic view of the UML diagram.



Sequencing Diagram:

The sequencing diagram below shows the calls of BboxTransform.getmatrix(), BboxTransformFrom.getmatrix() and BboxTransformTo.getmatrix():



class BboxTransform, BboxTransformFrom and BboxTransformTo in transforms.py: https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/transforms.py - L2487

Composite Pattern

- In the file figure.py we noticed that the composite design pattern was present. We noticed that the the draw function on lines 1210 to 1250 in figure.py was the composite part of the composite design pattern.
- This pattern involves the following classes: Figure, Artist, Axis, _ImageBase, Patch, Legend, Line2D and Text.
- The draw(self, renderer) function referenced works by sorting all of the following types of objects Artist, Axis, _ImageBase, Patch, Legend, Line2D and Text. Then function then proceeds to call the _draw_list_compositing_images function in image.py which then loops over the sorted list of objects and calls each objects' implementation of the draw function.

In this design pattern we have three components:

Component:

- Abstraction for all components
- Declare the interface for objects in the composition

Leaf:

- Represents leaf objects in the composition

Composite:

- Represents a composite Component
- Implements methods to manipulate children
- Implements all Component methods

In the code the this is how the classes fall into the design pattern:

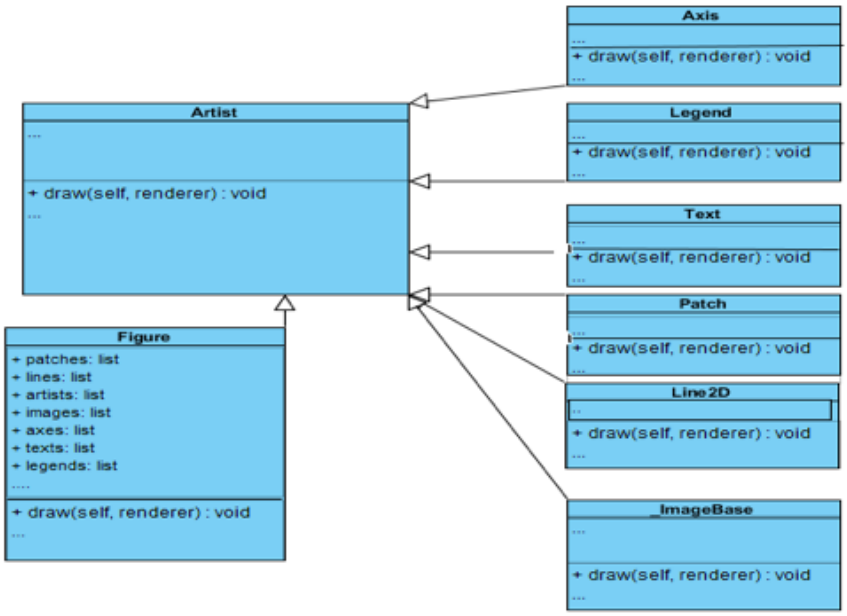
Component: Artist

Leaf: Axis, _ImageBase, Patch, Legend, Line2D and Text

Composite: Figure

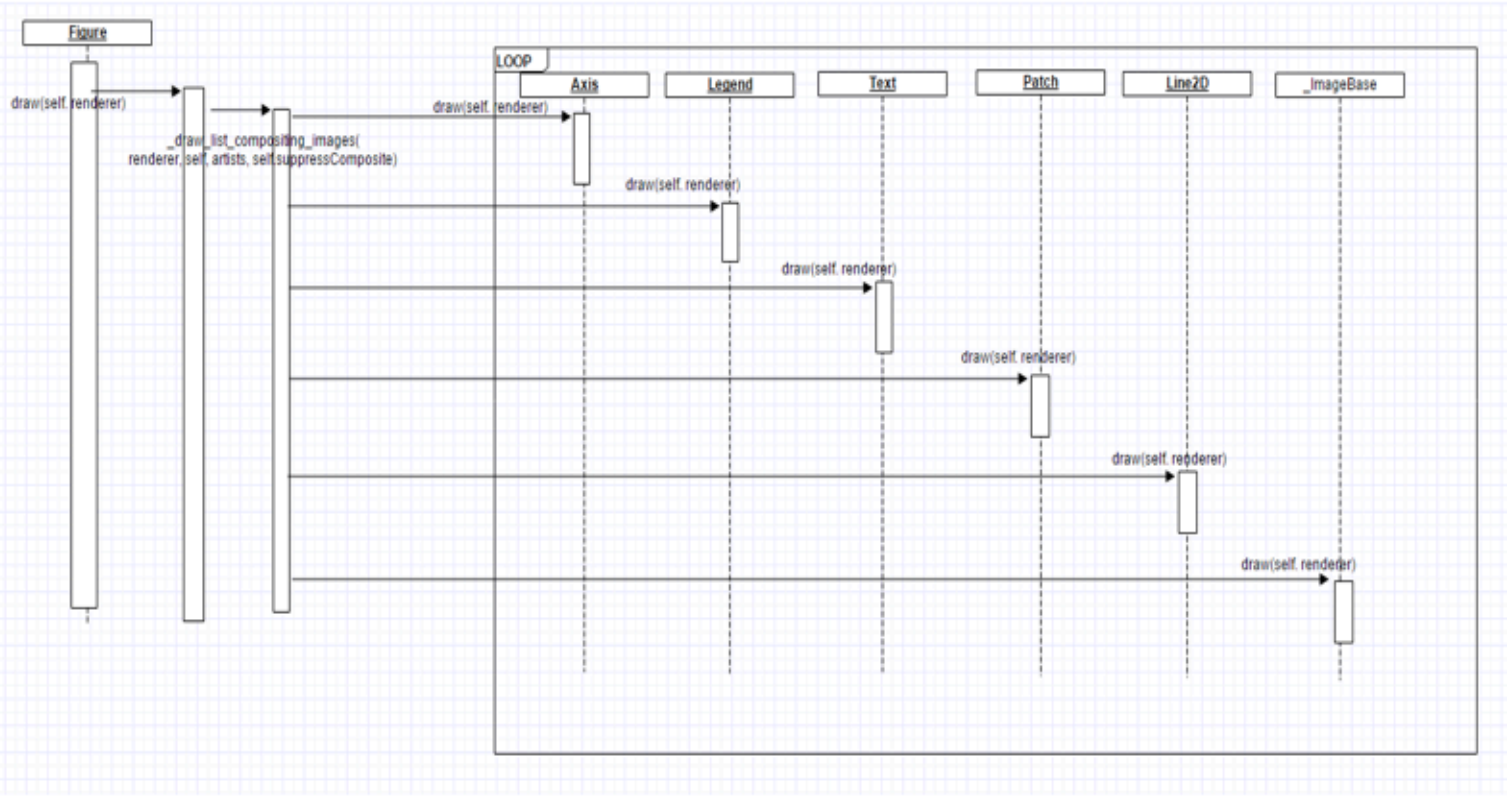
UML Diagram:

For our structural diagram, we've included the relationship between the component Artist and many of its leaf components including Axis, Patch, Legend, etc.



Sequencing Diagram:

The sequencing diagram below shows the calls of draw(self.renderer) of the Figure class and how it interacts with the leaf components of the design pattern.



draw() method in figure.py:
https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/figure.py - L1212