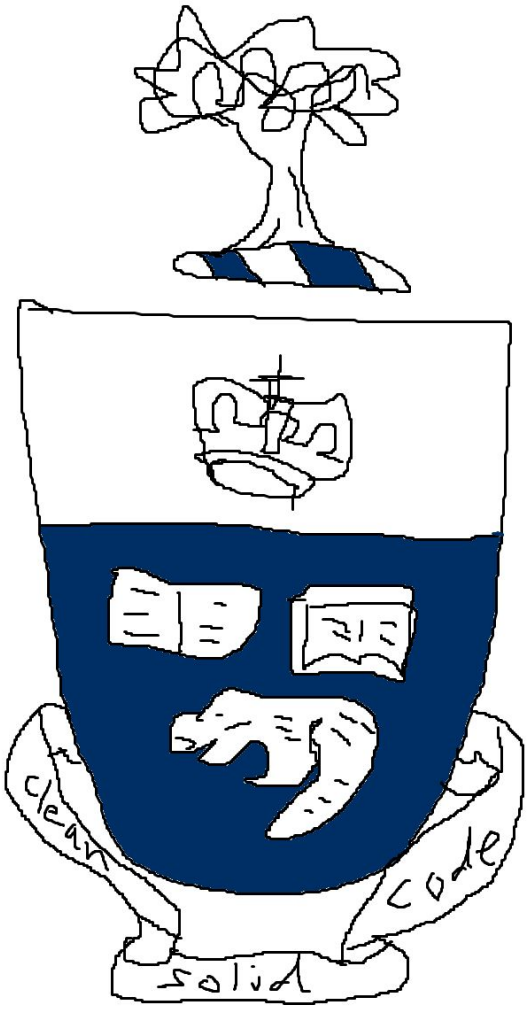


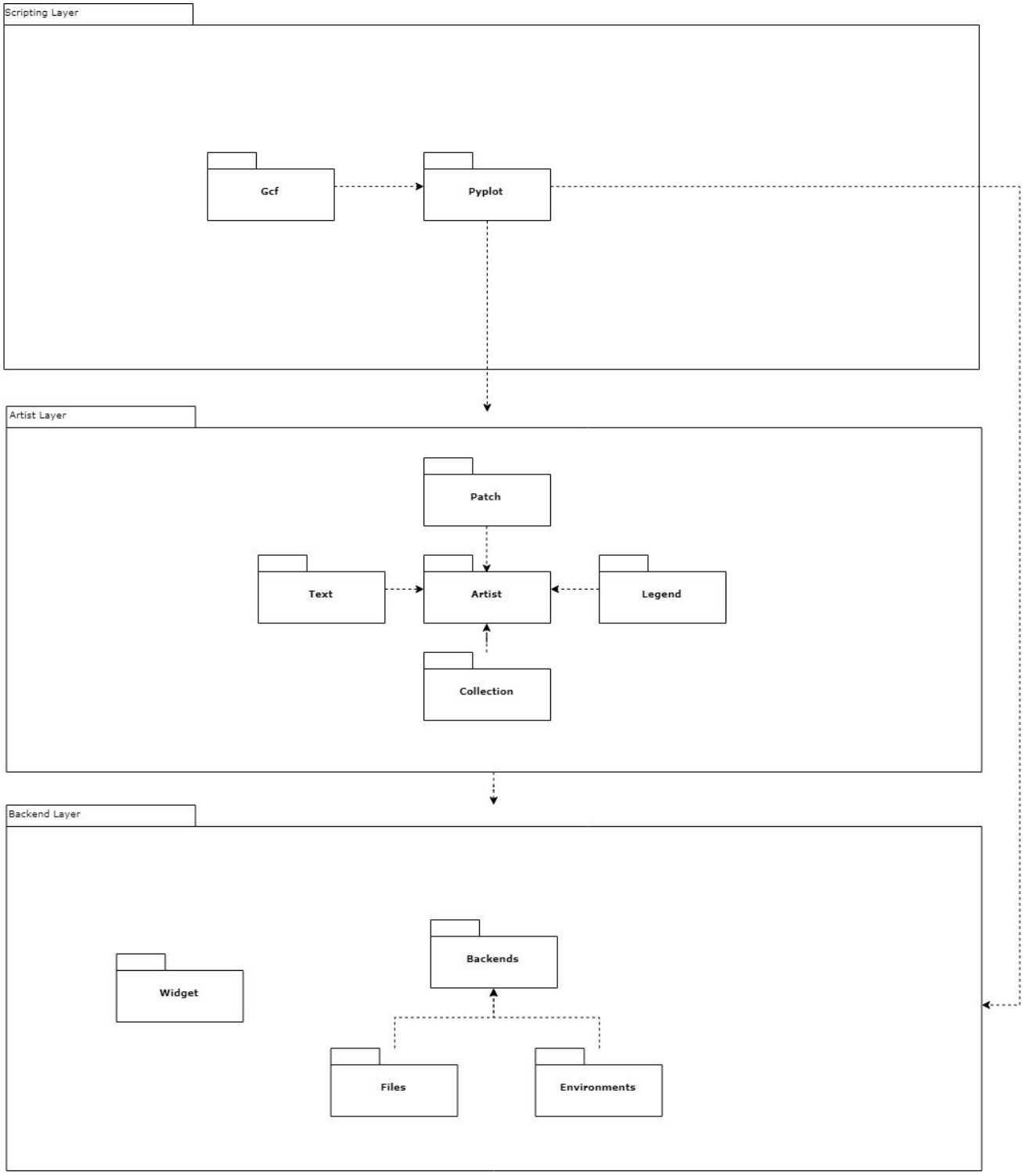
Deliverable 1: Team Plagiarism

Deliverable #1: Understanding Matplotlib
February 5th 2019
Team Members: Minqi Wang, Shuang Wu,
James Nicol, Xinrui Tong, Zixing Gong

Table of Contents

General Architecture:

Structure UML



Commentary:

The overall architecture is three layers (similar to any good cake) one for scripting, one for frontend, and one for backend. The architectural pattern immediately apparent is the open layered architecture with 3 big layers: the scripting layer, artist layer, and backend layer. The scripting layer communicates with both the artist and backend layer, hence open. The scripting layer allows users of Matplotlib to easily plot their data through simple API calls. The artist layer is responsible for "drawing" the different components of what is displayed. The backend layer is responsible for outputting the "drawings" from the artist layer and handling user interaction with said "drawings" if interactive mode is enabled. The usage of facade pattern also greatly reduces coupling between classes, seeking to pass through a singular interface (e.g. pyplot and figure). Matplotlib chooses backend from either of the types non-interactive (aggregated to be 'Files' in the UML) and/or interactive user environment (aggregated to be 'Environment' in the UML). An example of non-interactive backend is backend_pdf.py for pdf output. An example of interactive output is backend_gtk3.py for GTK. Matplotlib can do exceptional things with widgets combining interactive and non-interactive plotting. Matplotlib has ability to plot virtually every kind of graph, thus covering all features is frivolous, but every type of graph uses the architecture in the same way. There is always the pyplot, then the figure.

An interesting thing about Matplotlib architecture is that while the scripting layer uses the facade design pattern so that an end user does not have to worry about the other backend and artist layers. But if you wanted to use Matplotlib as a developer (ie. using Matplotlib in an app) then it would be better to use the backend and artist layers directly.

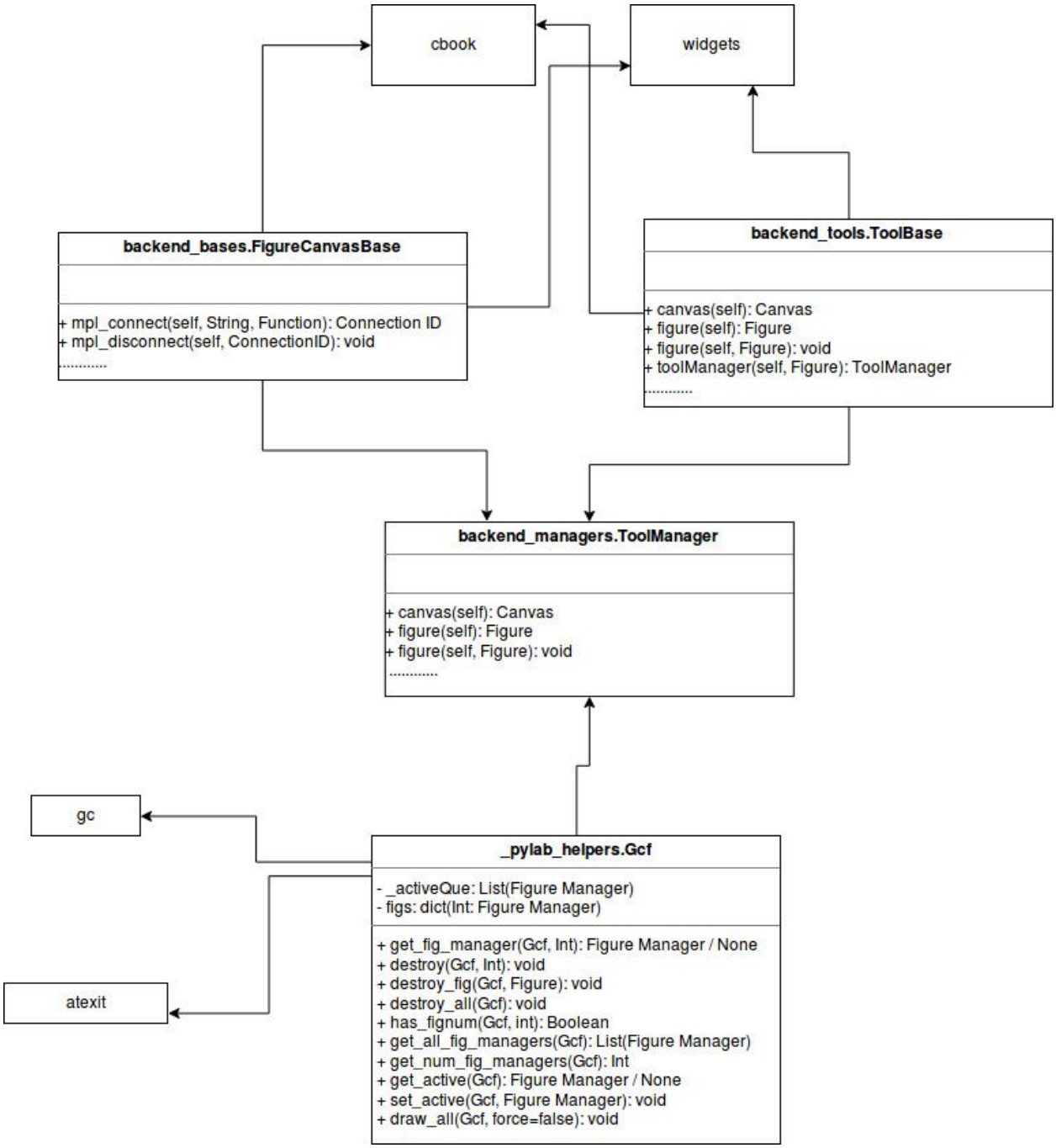
Matplotlib is large and complex, but what seems to be a bug as a result of the design is there is no real way to dictate order. There is zorder in plot, but that seem to be unreliable in cases because Matplotlib seems to plot before the zorder is introduced and can cause problems. This also results in a problem with graph legends getting stuck behind items plotted and becoming obscured. A plausible solution would require an overhaul of the system and perhaps introducing another layer in the architecture purely responsible for ordering and organizing the display layer of elements.

A special mention to widgets, Matplotlib has interesting capabilities with widgets as all backends support widgets. This means the user can incorporate some interesting interactive components using widgets.

Design Patterns:

Singleton Design Pattern:

Structural UML:



- All classes in the diagram are under matplotlib/lib/matplotlib/
- Overview of Gcf:
    - Gcf is a singleton that is never instantiated. It consists of a list of figure managers, with the "active" one being at the end, and a dictionary where the keys are integers and the values are figure managers. A figure manager is a type of tool manager that is used to link all of the user interactions with a figure to the backend functions. The backend base that is linked with this figure manager is the FigureCanvasBase, which is "an abstraction layer that separates the matplotlib.figure.Figure from the backend specific details like a user interface drawing area".
    - Gcf allows us to add, view, modify, and destroy all of the current figure managers, and as an extension, all of the current figures, using static methods such as get_fig_manager() and get_num_fig_managers().
    - Gcf is used to organize a list and dictionary of Figure Managers using static methods. It is linked to the backend tools and managers which allows it to connect and disconnect figures from the canvas using mpl_connect() and mpl_disconnect() found in the FigureCanvasBase class.
- Location of the classes in the overall system structure:
    - The Gcf class sits on the scripting layer as it contains containers (a list and a dictionary) that organize access to Figure Managers which deal with the artist and backend layers. The backend classes (tool, base, manager) sit in a low-level layer which is responsible for interaction with various GUIs. These classes themselves mainly manages the relation between GUI events (i.e. user operations such as keystrokes, mouse moves etc.) and the event listener functions (i.e. what to do on recipient of events).
- Explanation of how the Singleton design pattern is shown in this example:
    - Matplotlib's implementation of Gcf is an unorthodox approach to singleton design pattern. Where the singleton design pattern allows only "one instance of a class" does this by allowing no instantiation of gcf(), but since gcf() can store data in its global variables and modify its global variables using class/static methods it effectively achieves only "one instance". Therefore, Gcf achieves the desire of the singleton design pattern which is access to one set of data information within Gcf 's global variables.
- Links to the classes involved in this example:
    - Class Gcf
      https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/_pylab_helpers.py
      Gcf is the singleton class that contains the figure managers.
      All of the methods contained in the .py file are associated with the class, these are static methods used to interact with the data structures owned by the class (the list and dictionary
    - Class ToolManager
      https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/backend_managers.py
      Line 45
      The methods involved: canvas() and figure() located at lines 78 and 85 respectively
    - Class ToolBase
      https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/backend_tools.py

Line 37
The methods involved: figure() and canvas() located at lines 88 and 96 respectively

- Class FigureCanvasBase
  https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/backend_bases.py
  Line 1520
  The involving methods: mpl_connect(), mpl_disconnect() located at lines 2091 and 2139 respectively

- Notable code snippets:
  - Since the class itself is never instantiated, it is a singleton and only allows static access, like in the below code snippet from pyplot.py:

```
def gcf():
    """Get a reference to the current figure."""
    figManager = _pylab_helpers.Gcf.get_active()
    if figManager is not None:
        return figManager.canvas.figure
    else:
        return figure()
```

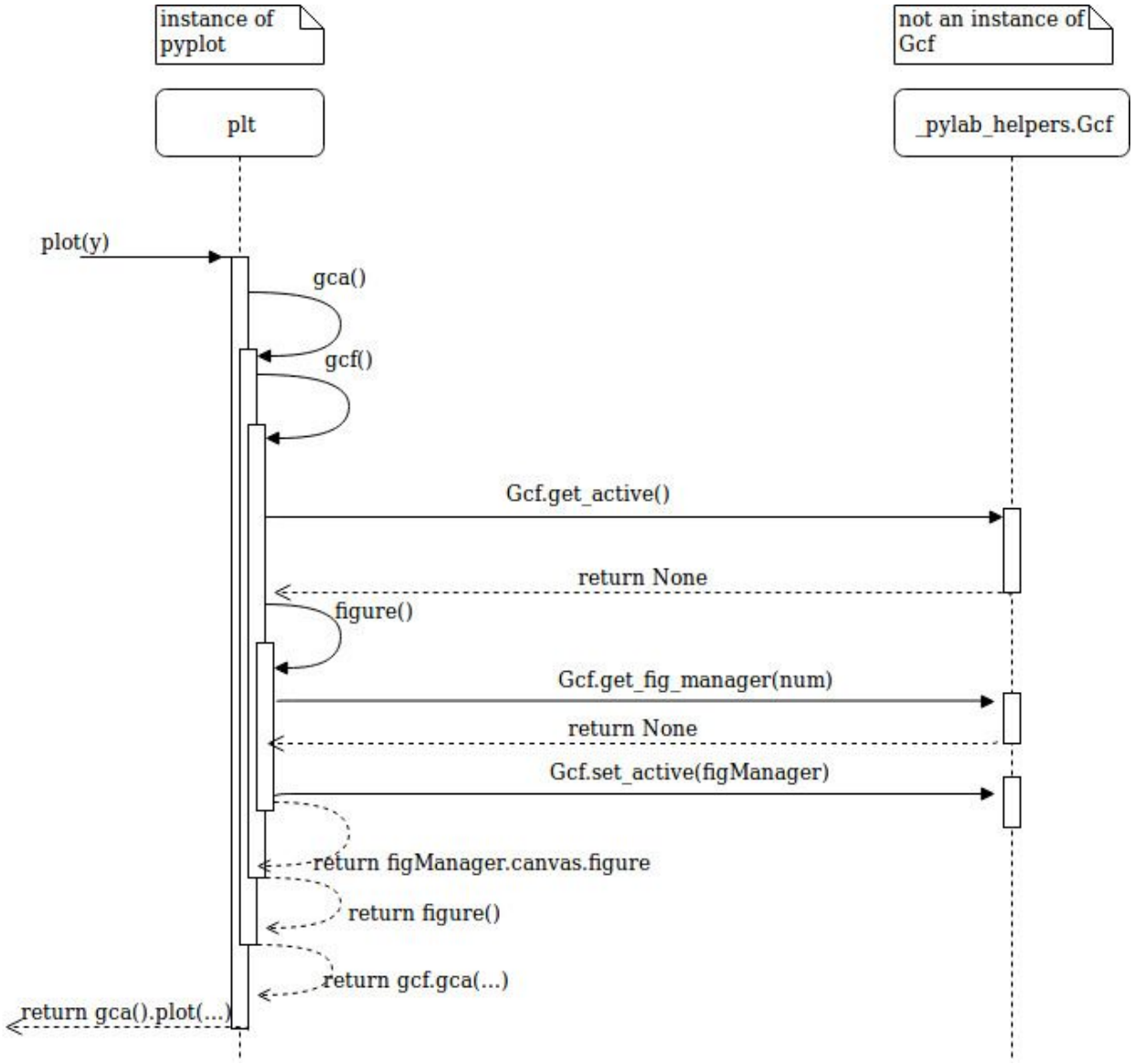
Figure 1: Accessing the current figure.
In this snippet, we are trying to access the current figure (**g**et **c**urrent **f**igure) by seeing if there are any Figure Managers in the Gcf class. If there is one then we access its figure, otherwise we make a new one.

Sequence Diagram:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

y = np.random.rand(100000)
y[50000:] *= 2
y[np.logspace(1, np.log10(50000), 400).astype(int)] = -1
plt.plot(y)
plt.show()
```

Figure 2: Sequence diagram example.

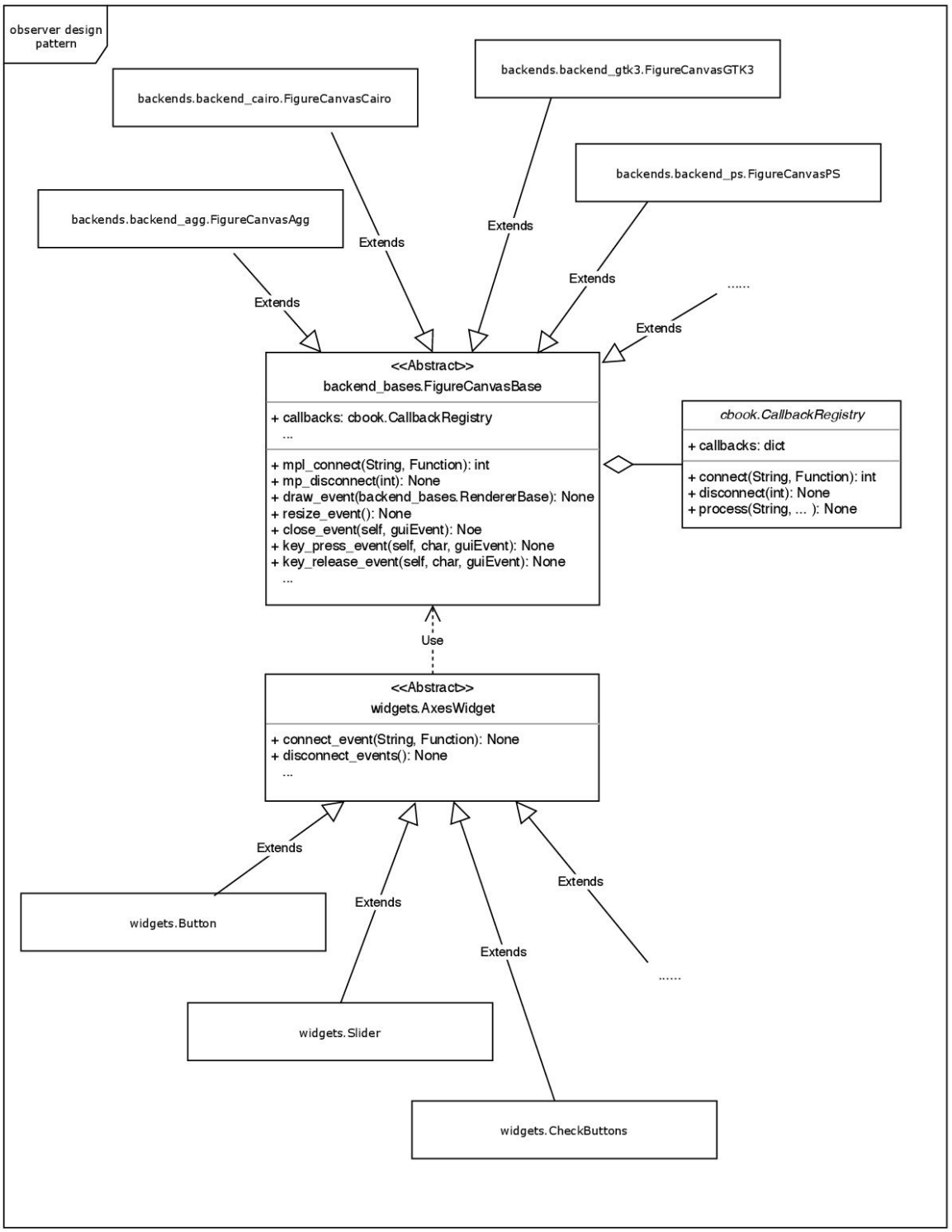
- The following sequence diagram will show the plot(y) command from the above code snippet.



- Explanation of sequence diagram:
    - Since there is no current figures or figure managers, they will have to be created. This is shown by the None returned by Gcf.get_active(), since there are no figure managers, there are no figures, so pyplot creates a new figure using figure() and then creates a new number, and creates a new Figure Manager, and sets it to the active one using Gcf.set_active(Figure Manager). After this code runs, a new figure manager is created and that figure can now be drawn on the canvas using mpl.show().
- Location of the classes involved in this sequence diagram:
    - Class Gcf
      https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/_pylab_helpers.py
      Gcf is the singleton class that contains the figure managers.

Observer Design Pattern:

Structural UML:



- All classes in the diagram are under matplotlib/lib/matplotlib/.
- "guiEvent" denotes events dispatched from the GUIs.
- "Function" denotes python functions.
- Location of the classes in the overall system structure:
    - These classes sit in a low-level layer which is responsible for interaction with various GUIs. These classes themselves mainly manages the relation between GUI events (i.e. user operations such as keystrokes, mouse moves etc.) and the event listener functions (i.e. what to do on recipient of events).
- Explanation of how the observer design pattern is shown in this example:
    - The FigureCanvasBase class and its subclasses are observables; The AxesWidget class and its subclasses are observers that observes the canvas classes.
    - The CallbackRegistry class is a utility class which maintains a mapping from event names to event handlers, which are the registered observers. The FigureCanvasBase class uses the CallbackRegistry class to manage its observers.
    - An observer can call mpl_connect() or mpl_disconnect() to register to or to unregister from the observable. The observable uses process() method of its attribute "callbacks" to notify the registered observers.
- Links to the classes involved in this example:
    - Class AxesWidget and its subclasses: https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/widgets.py Line 92 -- AxesWidget class
    - Class FigureCanvasBase: https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/backend_bases.py Line 1520 – FigureCanvasBase class
    - The folder where FigureCanvasBase subclasses are implemented (Roughly each backend GUI integration code implements a subclass): https://github.com/matplotlib/matplotlib/tree/master/lib/matplotlib/backends
    - Class CallbackRegistry: https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/cbook/__init__.py Line 88 – CallbackRegistry class
- Notable code snippets:
    - How FigureCanvasBase uses CallbackRegistry: (In class FigureCanvasBase https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/backend_bases.py)

```
2090
2091    def mpl_connect(self, s, func):
2092        """
2093        Connect event with string *s* to *func*.  The signature of *func* is::
2094
2095          def func(event)
2096
2097        where event is a :class:`matplotlib.backend_bases.Event`.  The
2098        following events are recognized
2099
2100        - 'button_press_event'
2101        - 'button_release_event'
2102        - 'draw_event'
2103        - 'key_press_event'
2104        - 'key_release_event'
2105        - 'motion_notify_event'
2106        - 'pick_event'
2107        - 'resize_event'
2108        - 'scroll_event'
2109        - 'figure_enter_event',
2110        - 'figure_leave_event',
2111        - 'axes_enter_event',
2112        - 'axes_leave_event'
2113        - 'close_event'
2114
2115        For the location events (button and key press/release), if the
2116        mouse is over the axes, the variable ``event.inaxes`` will be
2117        set to the :class:`~matplotlib.axes.Axes` the event occurs is
2118        over, and additionally, the variables ``event.xdata`` and
2119        ``event.ydata`` will be defined.  This is the mouse location
2120        in data coords.  See
2121        :class:`~matplotlib.backend_bases.KeyEvent` and
2122        :class:`~matplotlib.backend_bases.MouseEvent` for more info.
2123
2124        Return value is a connection id that can be used with
2125        :meth:`~matplotlib.backend_bases.Event.mpl_disconnect`.
2126
2127        Examples
2128        --------
2129        Usage::
2130
2131            def on_press(event):
2132                print('you pressed', event.button, event.xdata, event.ydata)
2133
2134            cid = canvas.mpl_connect('button_press_event', on_press)
2135        """
2136
2137        return self.callbacks.connect(s, func)
```

Figure 3: Registration of event listeners on observable side.

```
2138
2139     def mpl_disconnect(self, cid):
2140         """
2141         Disconnect callback id cid
2142
2143         Examples
2144         --------
2145         Usage::
2146
2147             cid = canvas.mpl_connect('button_press_event', on_press)
2148             #...later
2149             canvas.mpl_disconnect(cid)
2150         """
2151         return self.callbacks.disconnect(cid)
```

Figure 4: Unregistration of event listeners on observable side.

```
1634
1635     def key_press_event(self, key, guiEvent=None):
1636         """Pass a `KeyEvent` to all functions connected to ``key_press_event``.
1637         """
1638         self._key = key
1639         s = 'key_press_event'
1640         event = KeyEvent(
1641             s, self, key, self._lastx, self._lasty, guiEvent=guiEvent)
1642         self.callbacks.process(s, event)
1643
1644     def key_release_event(self, key, guiEvent=None):
1645         """
1646         Pass a `KeyEvent` to all functions connected to ``key_release_event``.
1647         """
1648         s = 'key_release_event'
1649         event = KeyEvent(
1650             s, self, key, self._lastx, self._lasty, guiEvent=guiEvent)
1651         self.callbacks.process(s, event)
1652         self._key = None
1653
1654     def pick_event(self, mouseevent, artist, **kwargs):
1655         """
1656         This method will be called by artists who are picked and will
1657         fire off :class:`PickEvent` callbacks registered listeners
1658         """
1659         s = 'pick_event'
1660         event = PickEvent(s, self, mouseevent, artist,
1661                           guiEvent=mouseevent.guiEvent,
1662                           **kwargs)
1663         self.callbacks.process(s, event)
1664
1665     def scroll_event(self, x, y, step, guiEvent=None):
1666         """
1667         Backend derived classes should call this function on any
1668         scroll wheel event.  x,y are the canvas coords: 0,0 is lower,
1669         left.  button and key are as defined in MouseEvent.
1670
1671         This method will be call all functions connected to the
1672         'scroll_event' with a :class:`MouseEvent` instance.
1673         """
1674         if step >= 0:
1675             self._button = 'up'
1676         else:
1677             self._button = 'down'
1678         s = 'scroll_event'
1679         mouseevent = MouseEvent(s, self, x, y, self._button, self._key,
1680                                 step=step, guiEvent=guiEvent)
1681         self.callbacks.process(s, mouseevent)
```

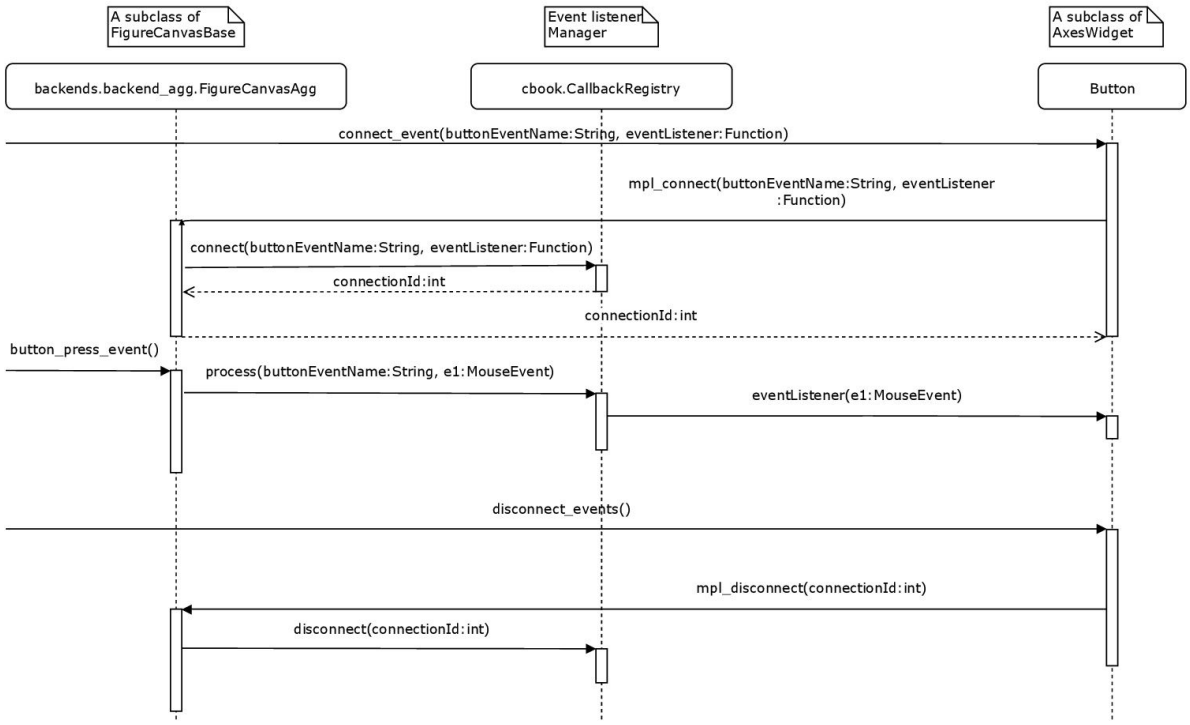
Figure 5: Notifying registered observers by various state updates.

- How observers register and unregister event listeners – Note that they are not registering or unregistering themselves but event listeners instead: (In class AxesWidget https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/widgets.py)

```
118
119     def connect_event(self, event, callback):
120         """Connect callback with an event.
121
122         This should be used in lieu of `figure.canvas.mpl_connect` since this
123         function stores callback ids for later clean up.
124         """
125         cid = self.canvas.mpl_connect(event, callback)
126         self.cids.append(cid)
127
128     def disconnect_events(self):
129         """Disconnect all events created by this widget."""
130         for c in self.cids:
131             self.canvas.mpl_disconnect(c)
132
```

Figure 6: Register and unregister event listeners on observer side (calling methods in the observable).

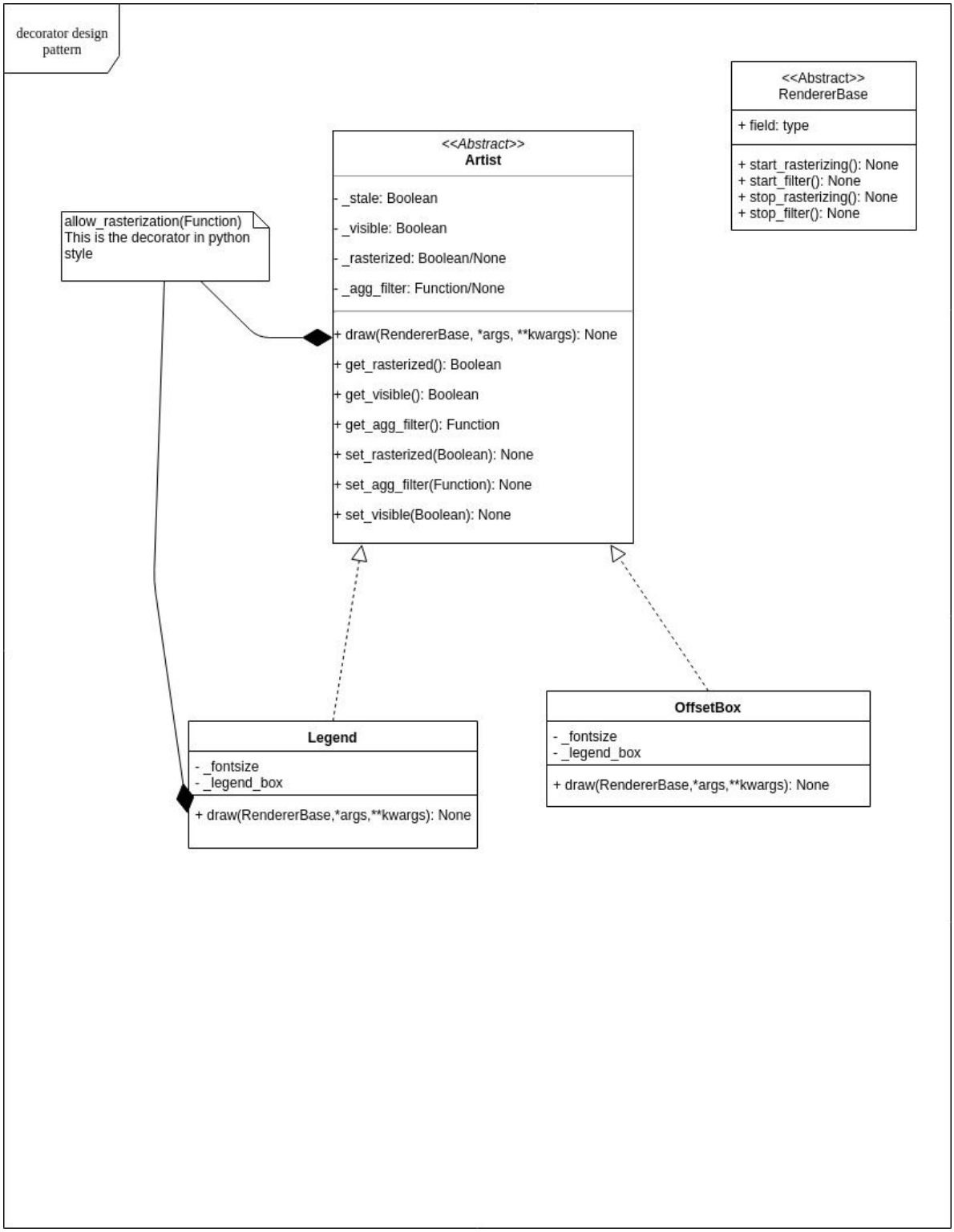
Behavioural UML:



- Explanation of the sequence diagram:
    - The above sequence diagram illustrates how this observer pattern is used by taking the interaction between 2 concrete classes, the Button class and the FigureCanvasAgg class, as an example.
- Location of the classes involved in this sequence diagram:
    - Class Button: https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/widgets.py Line 134
        - The methods involved
            - The methods connect_event() and disconnect_events() inherited from class AxesWidget (Location is shown in a screenshot above)
            - Method eventListener – An example is Button._click() in https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/widgets.py Line 200
    - Class FigureCanvasAgg: https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/backends/backend_agg.py Line 363
        - The methods involved
            - Method button_press_event() inherited from class FigureCanvasBase in https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/backend_bases.py Line 1683
            - The methods mpl_connect() and mpl_disconnect() inherited from class FigureCanvasBase (Location is shown in a screenshot above)
    - Class CallbackRegistry: As mentioned above
        - The methods involved
            - The methods connect(), disconnect_events() and process() in https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/cbook/__init__.py Line 162, Line 189 and Line 204

Decorator Design Pattern:

Structural UML:



- All classes and functions are under Matplotlib/lib/matplotlib/
- "Function" in allow_rasterization denotes callable draw function/method
- "Function" in method set_agg_filter and field _agg_filter denotes filter python function
- "RendererBase" denotes class RendererBase declared in backend_bases.py
- Explanation of where the classes are in the overall system structure:
    - These classes sit in the mid-level layer (Artist layer) which is responsible for handling the status before/after rendering, manipulating various graphical objects that are going to be painted on figure's canvas. These graphical objects includes, but are not limited to: Tick, Text, Patch, Legend, Line2D, Table, Collection…
- Explanation of how the decorator design pattern is shown in this example:
    - The function "allow_rasterization" is a decorator for the method "draw" declared in abstract class Artist and "decorates" the overwritten method "draw" in Legend. It uses the wraps function in the functools module (https://docs.python.org/2/library/functools.html) to implement the decorator function.
    - "allow_rasterization" enables the specific graphical object to start rastering(https://en.wikipedia.org/wiki/Rasterisation) before rendering and stop it after rendering.
- The benefits of the decorator design pattern
    - It provides additional functionality to the "draw" function while adhering to the Single Responsibility Principle. It does not use inheritance to achieve that! Therefore, the flexibility, convenience, cleanness the decorator design pattern offers benefits a lot on the future maintenance and development of classes it involves.
- Links to the classes involved in this example:
    - Class Legend
      https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/legend.py
      Line 321 Legend class
    - Class OffsetBox
      https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/offsetbox.py
      Line 131 OffsetBox class
    - Class RendererBase
      https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/backend_bases.py
      Line 128 RendererBase class
    - Class Artist and Decorator function allow_rasterization
      https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/artist.py
      Line 57 Artist class, Line 20 allow_rasterization
- Some important code snippets:
    - How the decorator function allow_rasterization works with draw method of class Legend:

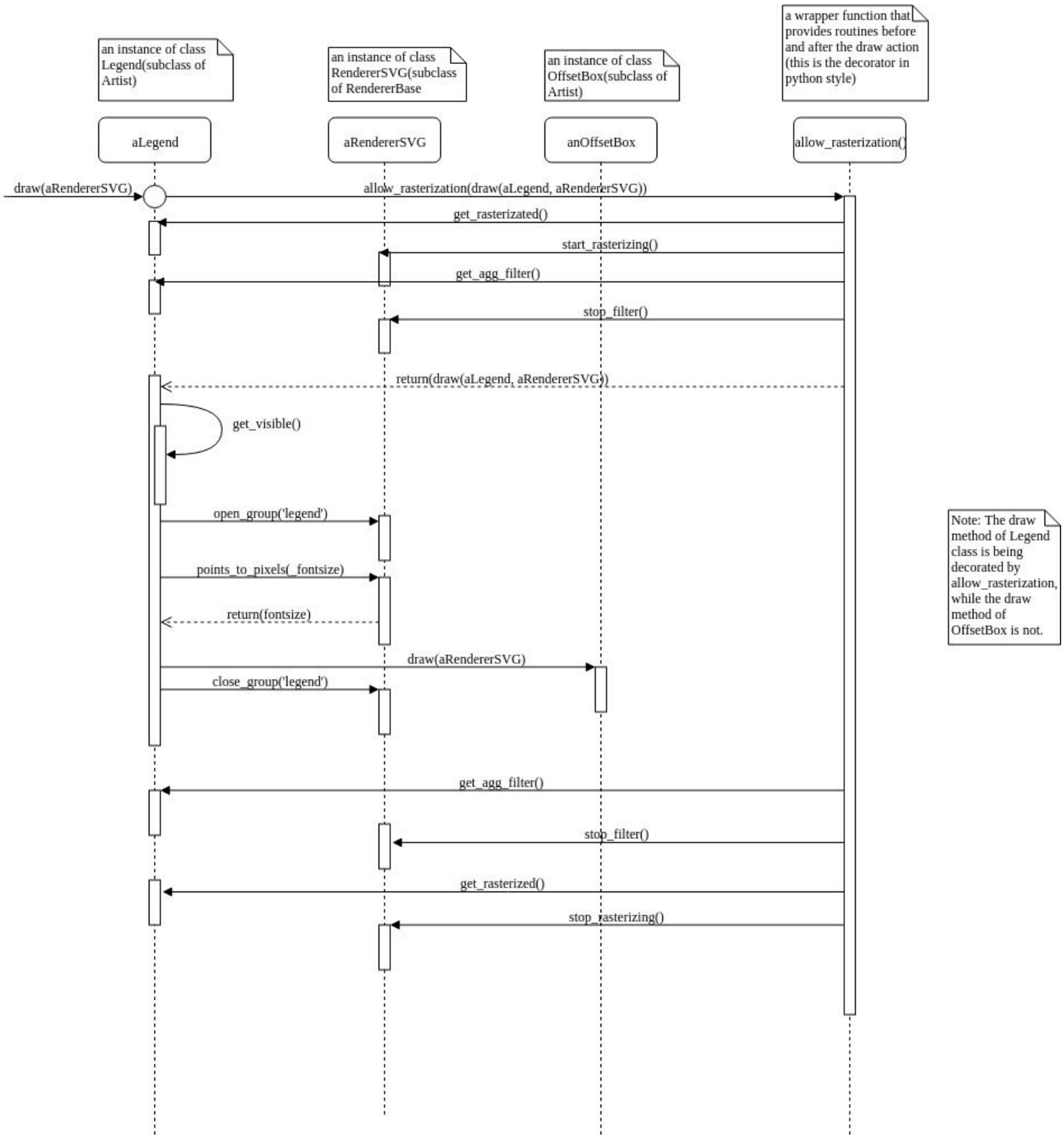
```python
def allow_rasterization(draw):
    """
    Decorator for Artist.draw method. Provides routines
    that run before and after the draw call. The before and after functions
    are useful for changing artist-dependent renderer attributes or making
    other setup function calls, such as starting and flushing a mixed-mode
    renderer.
    """

    # the axes class has a second argument inframe for its draw method.
    @wraps(draw)
    def draw_wrapper(artist, renderer, *args, **kwargs):
        try:
            if artist.get_rasterized():
                renderer.start_rasterizing()
            if artist.get_agg_filter() is not None:
                renderer.start_filter()

            return draw(artist, renderer, *args, **kwargs)
        finally:
            if artist.get_agg_filter() is not None:
                renderer.stop_filter(artist.get_agg_filter())
            if artist.get_rasterized():
                renderer.stop_rasterizing()

    draw_wrapper._supports_rasterization = True
    return draw_wrapper
```

Figure 7: The decorator allow_rasterization using wraps function in functools module.

```python
    @allow_rasterization
    def draw(self, renderer):
        "Draw everything that belongs to the legend."
        if not self.get_visible():
            return

        renderer.open_group('legend')

        fontsize = renderer.points_to_pixels(self._fontsize)

        # if mode == fill, set the width of the legend_box to the
        # width of the parent (minus pads)
        if self._mode in ["expand"]:
            pad = 2 * (self.borderaxespad + self.borderpad) * fontsize
            self._legend_box.set_width(self.get_bbox_to_anchor().width - pad)

        # update the location and size of the legend. This needs to
        # be done in any case to clip the figure right.
        bbox = self._legend_box.get_window_extent(renderer)
        self.legendPatch.set_bounds(bbox.x0, bbox.y0,
                                    bbox.width, bbox.height)
        self.legendPatch.set_mutation_scale(fontsize)

        if self._drawFrame:
            if self.shadow:
                shadow = Shadow(self.legendPatch, 2, -2)
                shadow.draw(renderer)

            self.legendPatch.draw(renderer)

        self._legend_box.draw(renderer)

        renderer.close_group('legend')
        self.stale = False
```

Figure 8: How to decorate a draw method using allow_rasterization in Python3 way, it is equal to say draw = allow_rasterization(draw).

Behavioural UML:



- The above sequence diagram illustrates how this decorator pattern is used by taking the workflow after calling draw method of concrete Legend class as an example.
- We can clearly see the flexibility and additional functionality the decorator function allow_resterizaiton provided in the draw method of Legend (the one being decorated) compared with the draw method of OffsetBox(the one not being decorated)
- Location of the classes and decorator involving in this sequence diagram:
    - Class Legend: https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/legend.py Line 321
        - The methods involved
            - Method draw() from itself Line 642
            - Methods get_visible(), get_rasterizated(), get_agg_filter() inherited from Artist class https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/artist.py Line 748, 816, 836
    - Class RendererSVG: https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/backends/backend_svg.py Line 273
        - The methods involved
            - Methods open_group('legend'), close_group('legend') from itself Line 505, 513
            - Methods points_to_pixels(), start_rasterizing(), stop_rasterizing(), start_filter(), stop_filter() inherited from RendererBase class https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/backend_bases.py Line 664, 689, 696, 704, 711
    - Class OffsetBox: https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/offsetbox.py Line 131
        - The methods involved
            - Method draw() from from itself Line 247
    - Decorator function allow_rasterization: https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/artist.py Line 20