PROJECT DELIVERABLE 2

Team 4 (Sky Blue)

CSCD01

February 8, 2016

Team Members:
Chun Cho
Richard Luo
Yuxuan Hu
Geoffrey Hong
Kai Lin

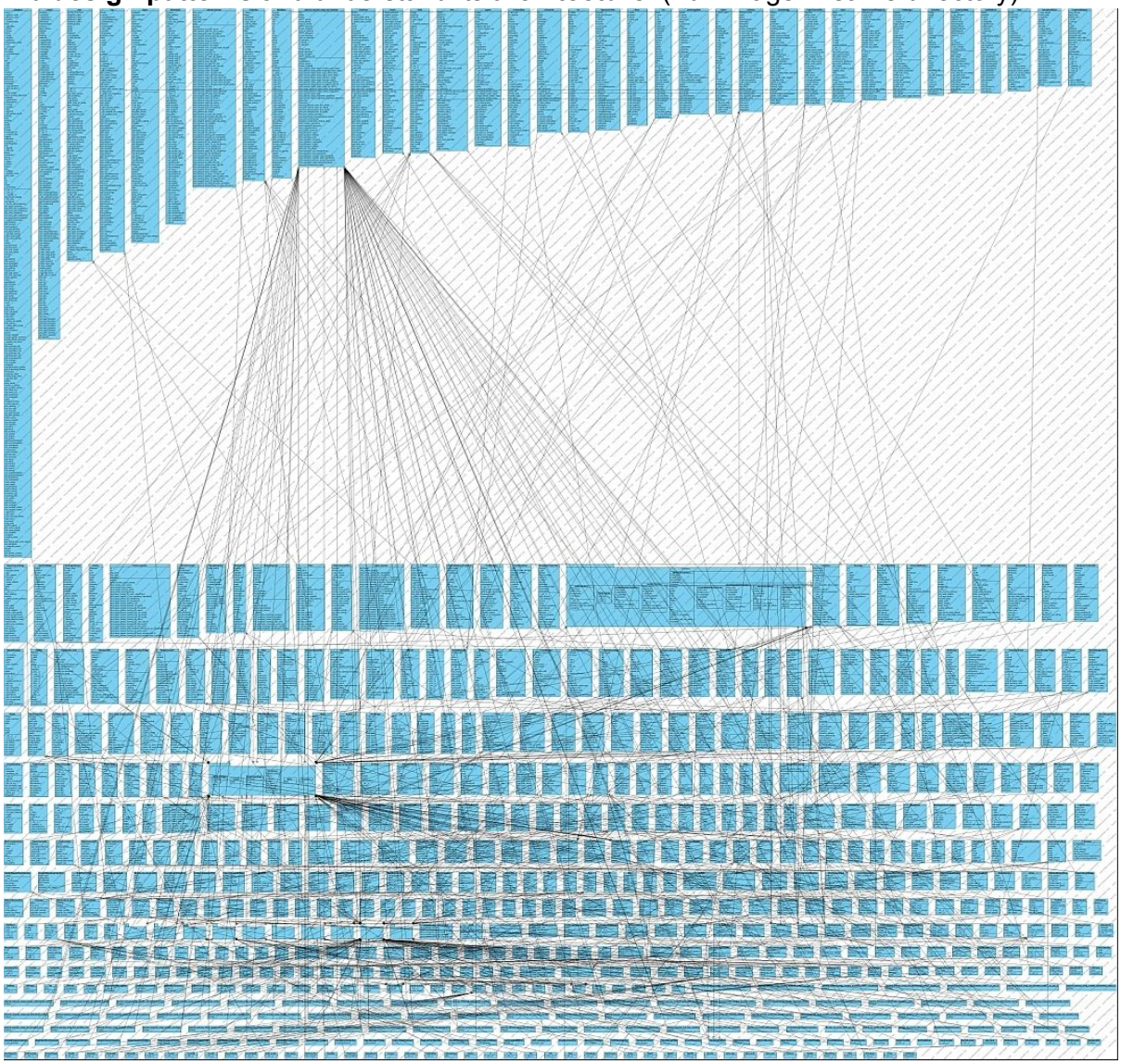
Table of Contents

Tools

Reverse Engineering Tool

For reverse engineering the system, multiple tools were tried. These tools include: Epydoc, PyNSource, and Visual Paradigm. Ultimately, the team decided to use Visual Paradigm as its GUI was intuitive and clean, and it was simple to use. Unlike PyNSource, where you are unable to import the entire project at once, Visual Paradigm allows users to import the entire code base, which greatly saves time. When compared to EpyDoc, Visual Paradigm had much better documentations and guides to using the software. EpyDoc had shown flaws due to poor documentation on their website. An example of this is incorrect options for running on command line. The documentation provided on the website was incorrect, and thus, caused the command to fail. In addition, EpyDoc is outdated (last updated 2008) and proven difficult to setup and use for both their GUI and command line options. Visual paradigm did not need to use the command line, and had a clean and intuitive user interface.

Below is the diagram generated by **Visual Paradigm,** which we have explored to help us find **design patterns** and understand its **architecture.** (Full image in same directory):



UML Drawing Tool

Gliffy was used for producing UML diagrams for the project as mentioned in the previous deliverable. Gliffy is an excellent online UML drawing tool which has a simple and efficient UI, enabling us to effectively create visual diagrams of matplotlib's system and design patterns. Since Gliffy is an online UML drawing tool, it can be easily shared amongst users because Gliffy allows for multiple access to the same images. Another tool, draw.io, was also considered; however, the team chose to stick with Gliffy as most of the team were more experienced with Gliffy.

System Architecture

The top-level matplotlib object that contains and manages all of the elements in a given graphic is called the Figure. matplotlib's architecture implements a framework for representing and manipulating the Figure object while separating the data rendering. This allows users to build sophisticated features into the Figures, without worrying about the backend.

matplotlib encapsulates drawing interfaces to allow rendering to multiple devices, event handling and windowing of user interface toolkits. Thus users can create rich interactive graphics and toolkits.

matplotlib's abstract UI event framework enables developers and end-users to write UI event-handling code once and run on all UI toolkits implemented by the framework. An example would be a developer writes interactive drag and move matplotlib figures, which will then work across all supported user interface toolkits.

Architectural Style (3-layer)

The architecture is separated into three layers. Each layer knows how to talk to the layer below it. However, the lower layers are not aware of the layers above it. The names of the three layers from bottom to top are **backend, artist, and scripting.**

Backend Layer

Main responsibilities: The Backend layer gives matplotlib an area to write texts and draw, and the tools necessary to accomplish this. This layer is also able to detect user events such as mouse clicks and keyboard strokes from different user interface toolkits and handle them accordingly.

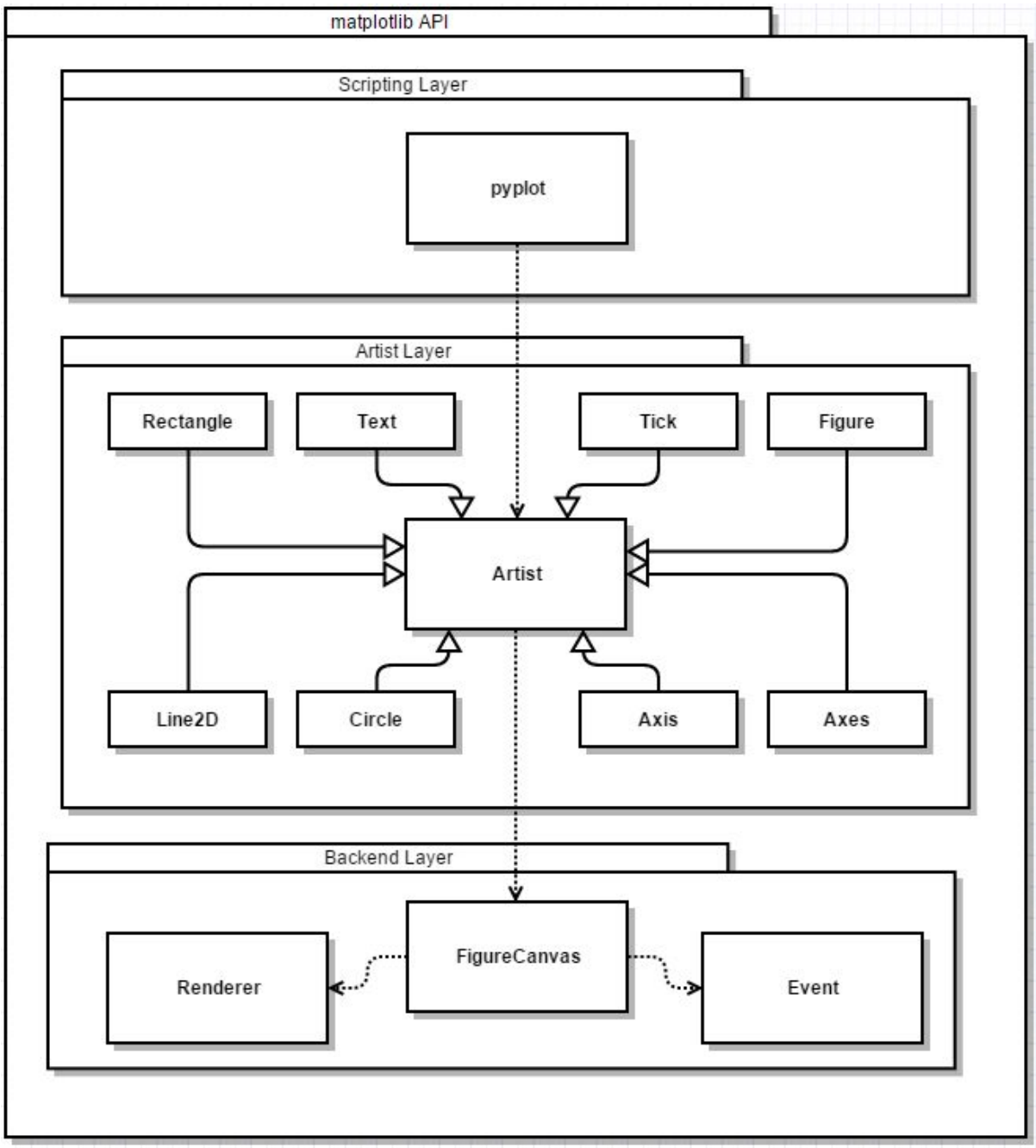
Artist Layer

Main responsibilities: The Artist layer uses the backend's rendering capability to draw the visuals on the area provided by the backend. It is important to note that everything seen on the Figure (text, lines, graphs, etc) are all instances of an Artist class, which has a draw() method to display themselves onto the canvas.

Scripting Layer

Main responsibilities: The Scripting layer talks to the Artist layer to provide users the ability to draw complex figures by calling the script in pyplot, that uses various instances of the Artist class. For example, to draw a simple line graph, one would need the following instances of Artist class: Axis, ticks, Line2D, text, etc.

Layer diagram



Quality of Architecture

matplotlib has a good architecture which successfully minimizes coupling between modules, while maximizing the cohesion of each module.

Coupling:

The system shows a low degree of coupling because the dependencies of each module only go in one direction. Namely, the dependencies go from the scripting layer to artist layer to backend layer.

Cohesion:

The system shows a high degree of cohesion between each class inside the artist layer and the backend layer. This is because every thing artist layer is strongly interrelated, sharing the same goal of drawing objects onto the canvas. In the backend layer, classes abstract themselves to work with supported matplotlib GUIs, transferring matplotlib renderer commands onto the GUI canvas and translating GUI user events (i.e. keystrokes, mouse clicks) to the matplotlib event framework.

Reusability:

As mentioned before, the abstraction of the UI toolkit's event framework makes many code reusable with multiple UIs. The backend layer provides capability for components to be suitable for use in other applications and in other scenarios. It also minimizes implementation time.

Maintainability

Maintainability is the ability of the system to undergo changes with a degree of ease. matplotlib's architecture allows users to build features and logic into Figures without worrying about the backend, which provides a great degree of ease in adding or changing functionality to meet new requirements.

Design Pattern

Composite (Parent-Child)

The composite pattern describes that a group of objects is to be treated in the same way as a single instance of an object.

Component :

- The abstraction for all components, including composite ones.
- Declares the interface for objects in the composition

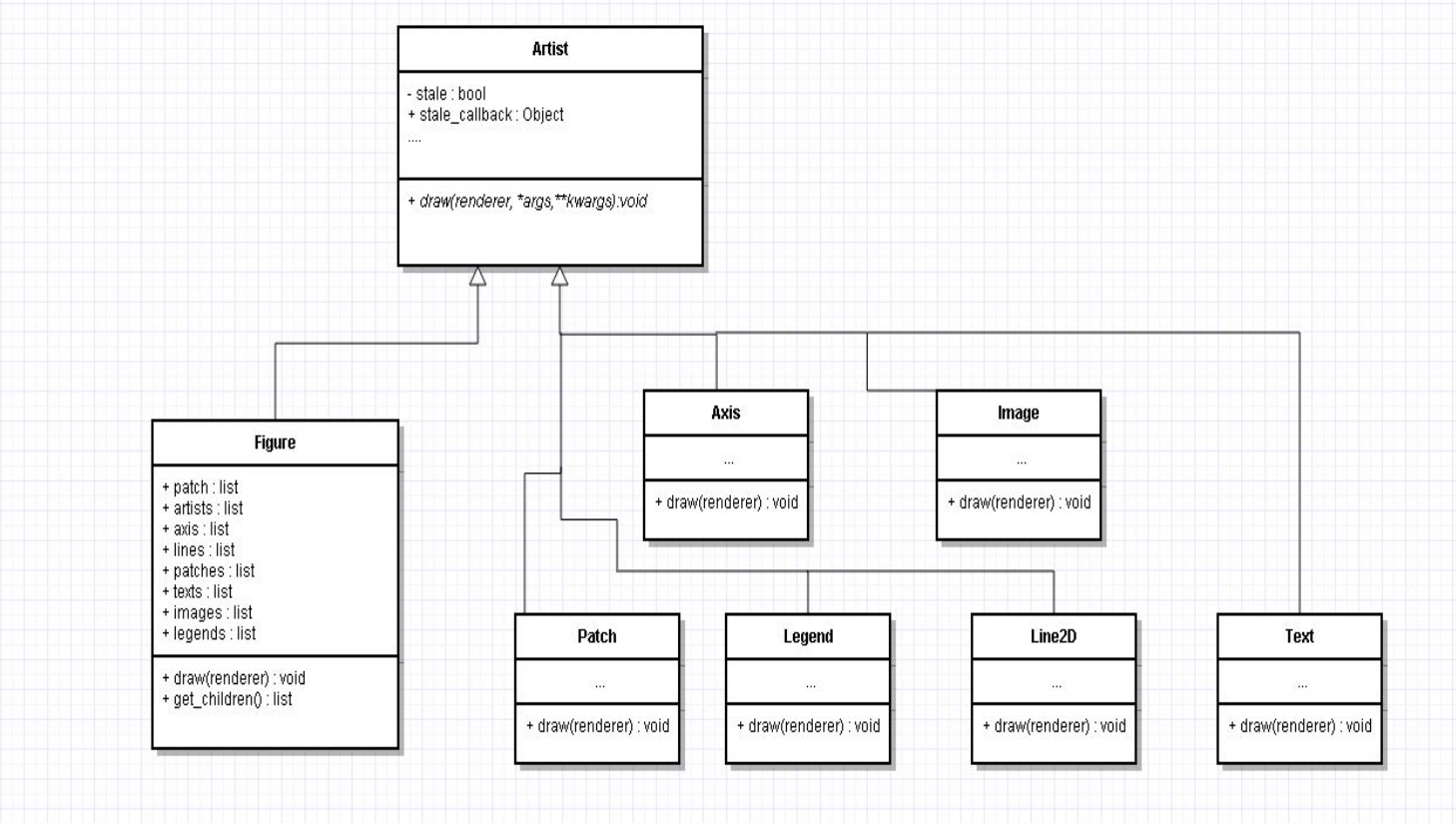
Leaf :

- Represents leaf objects in the composition
- Implements all Component methods

Composite :

- Represents a composite Component (component having children)
- Implements methods to manipulate children
- implements all Component methods, generally by delegating them to its children

In matplotlib, the composite pattern is within classes Artist, Figure, Axis, Image, Patch, Legend, Line2D and Text where Figure is composite and Axis, Image, Patch, Legend, Line2D and Text are all leaves. Draw method in Artist is an abstract method and have been implemented among the leaves and composite as shown in the class UML diagram below:



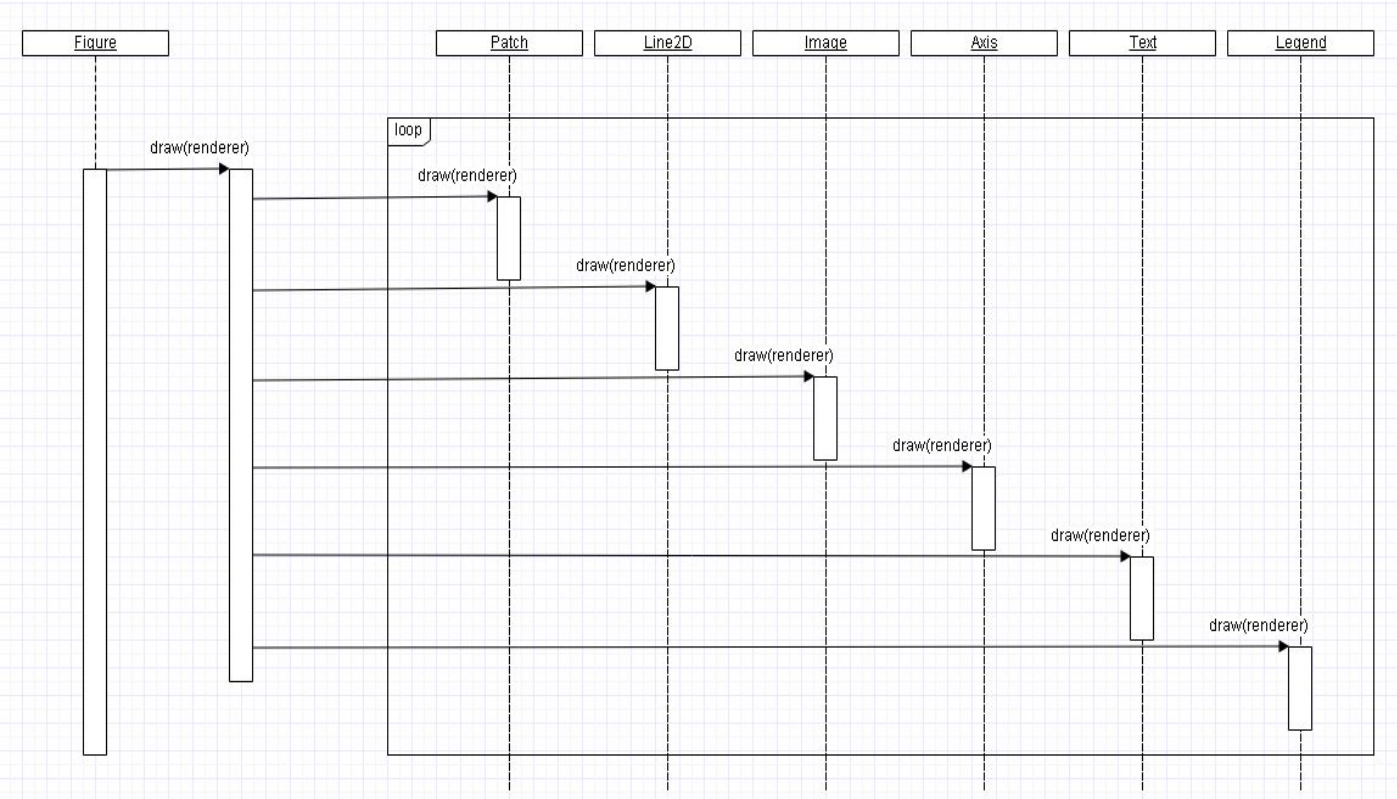
Code:

In figure.py (line 1209-1298):

This a composite implementation where Figure class contains a collection of Patch, Line2D, Image, Axis, Text and Legends. Moreover, Patch, Line2d, Image, Axis, Text and Legends all have their own implementation of draw(renderer). By using the composite pattern we can easily treat all of them uniformly in the collection within Figure.

```
# a list of (zorder, func_to_call, list_of_args)
dsu = []

for a in self.patches:
    dsu.append((a.get_zorder(), a, a.draw, [renderer]))

for a in self.lines:
    dsu.append((a.get_zorder(), a, a.draw, [renderer]))

if self.suppressComposite is not None:
    not_composite = self.suppressComposite

if (len(self.images) <= 1 or not_composite or
        not cbook.allequal([im.origin for im in self.images])):
    for a in self.images:
        dsu.append((a.get_zorder(), a, a.draw, [renderer]))

# render the axes
for a in self.axes:
    dsu.append((a.get_zorder(), a, a.draw, [renderer]))

# render the figure text
for a in self.texts:
    dsu.append((a.get_zorder(), a, a.draw, [renderer]))

for a in self.legends:
    dsu.append((a.get_zorder(), a, a.draw, [renderer]))
```

Sequence Diagram



When draw function is called in Figure, all its collections are iterated and called the draw function respectively. According to the code listed above, the draw trigger the leaves class in a particular order.

Strategy

Strategy pattern is a software design pattern that enables an algorithm's behaviour to be selected at runtime. It defines a family of algorithms, encapsulates each algorithm, and makes the algorithms interchangeable within that family. Strategy Design Pattern is often used for multiple implementations of a certain behaviour.

One of the examples that Strategy Design Pattern was used in matplotlib library is the ToolBase class in the backend_tools package. In the package, it is clearly that ToolBase is the base class for a number of different types of Tools. From the Strategy Design Pattern perspective, the ToolBase class is the algorithm we want to perform, and the subclasses are the different ways to implement the algorithm. In this implementation, the ToolBase class is responsible for trigger tools when certain tools are used. Based on the subclass specified at run time, trigger() will be called if a tool gets used. For example, if the ToolQuit subclass is used, then trigger method inside ToolQuit will be executed.

This design pattern is good for matplotlib because it inherit all of the tools that will be used and they share the same method and attribute. If I need to implement a new tool all I need is extended to the base ToolBase class or any subclass of ToolBase, It makes the code maintainable for future usage. This pattern enable the ability to maintain code without redesign the system.

Code:

backend_tools.py

```
class ToolBase(object):
        ...
        def trigger(self, sender, event, data=None):
            ...
            pass

class ToolToggleBase(ToolBase):
        def trigger(self, sender, event, data=None):
            if self._toggled:
                self.disable(event)
            else:
                self.enable(event)
            self._toggled = not self._toggled
        ...
class RubberbandBase(ToolBase):
        def trigger(self, sender, event, data):
            if not self.figure.canvas.widgetlock.available(sender):
                return
            if data is not None:
                self.draw_rubberband(*data)
            else:
                self.remove_rubberband()
```

```
        ...
(and all other tools.)
```

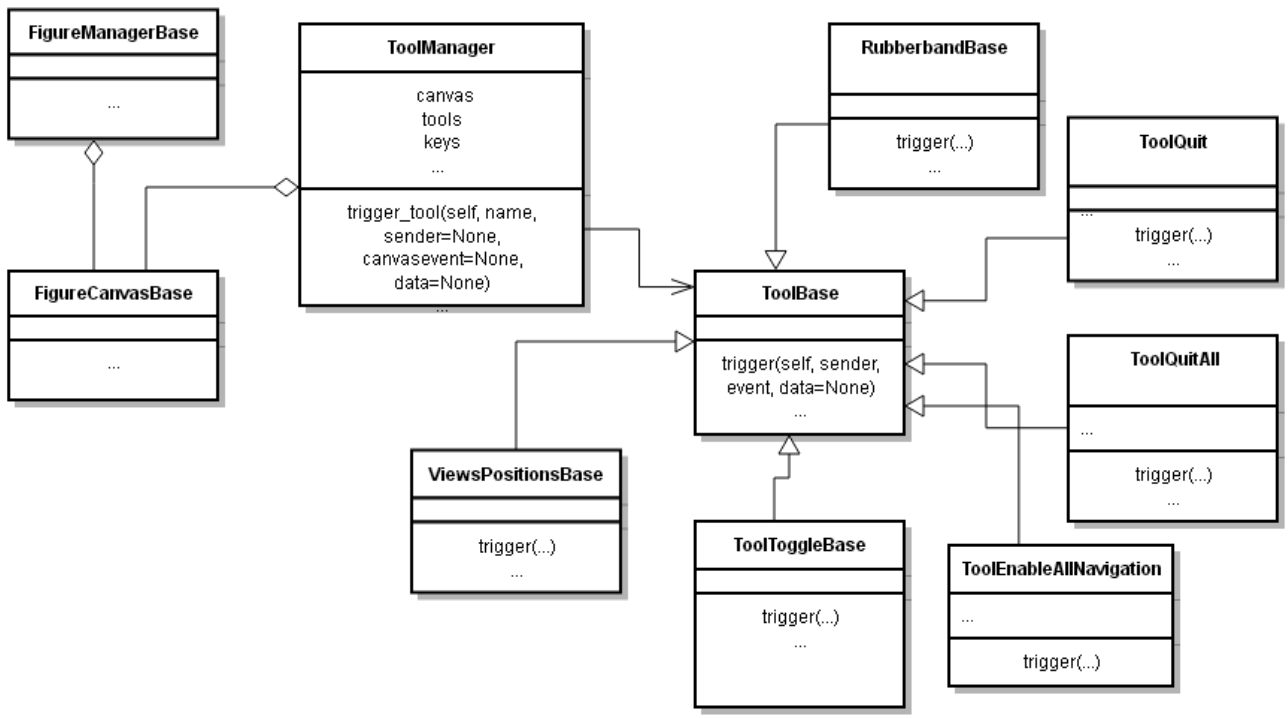
backend_managers.py

```
class ToolManager(object):
    def _trigger_tool(self, name, sender=None, canvasevent=None, data=None):
        ...
        tool.trigger(sender, canvasevent, data)
```

UML Diagram:

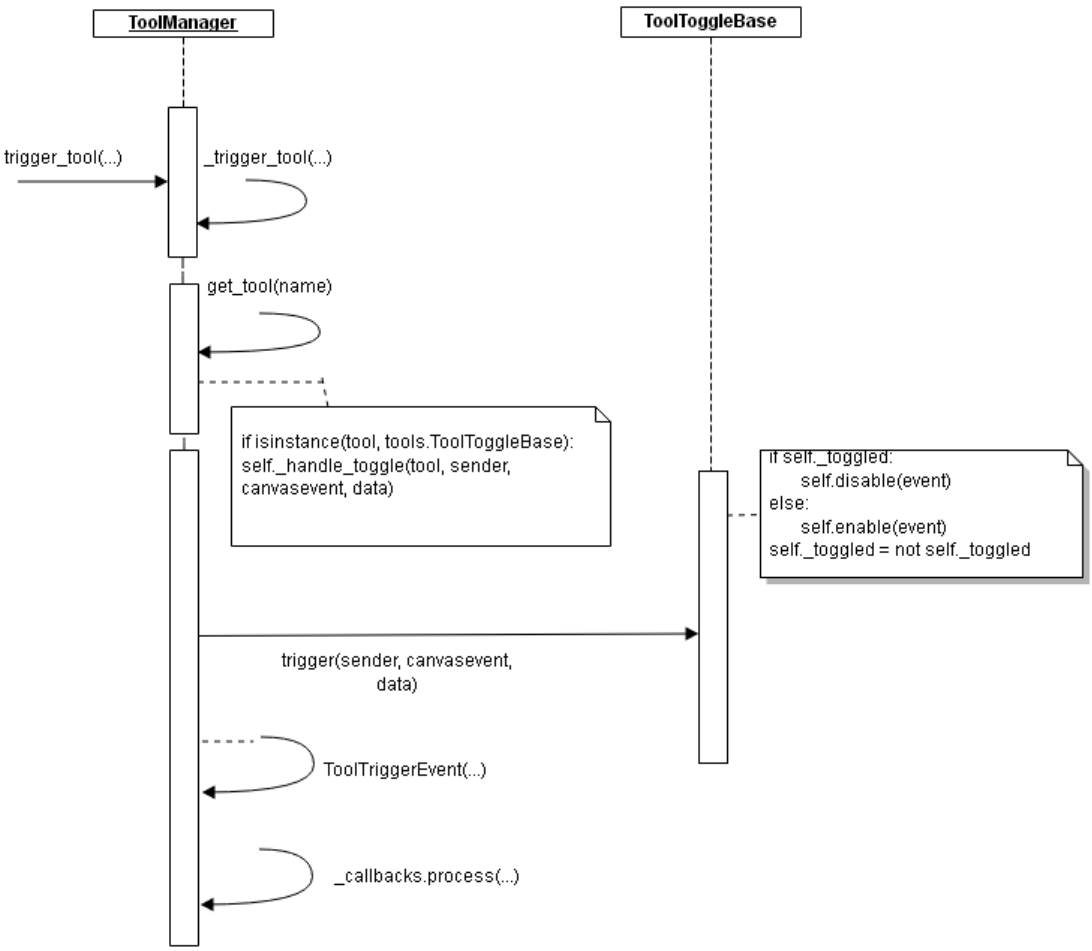
https://fossies.org/dox/matplotlib-1.5.1/classmatplotlib_1_1backend__tools_1_1ToolBase.html

The UML class diagram shows the relationship between FigureManagerBase, ToolManger, ToolBase, and the 6 different Tool Class. The diagrams shows multiple different strategies that extends the ToolBase class. This strategy greatly reduces the coupling within the package by creating a single base class. The sub-strategies themselves also have base classes so they can be extended for future uses.



Sequence Diagram

The sequence diagram shows a simple call to ToolManager.trigger_tool(...). Once a tool is selected it will do all the following step and trigger the tool. Also, based on the tool object, a different trigger function may be executed based on the type of tool.

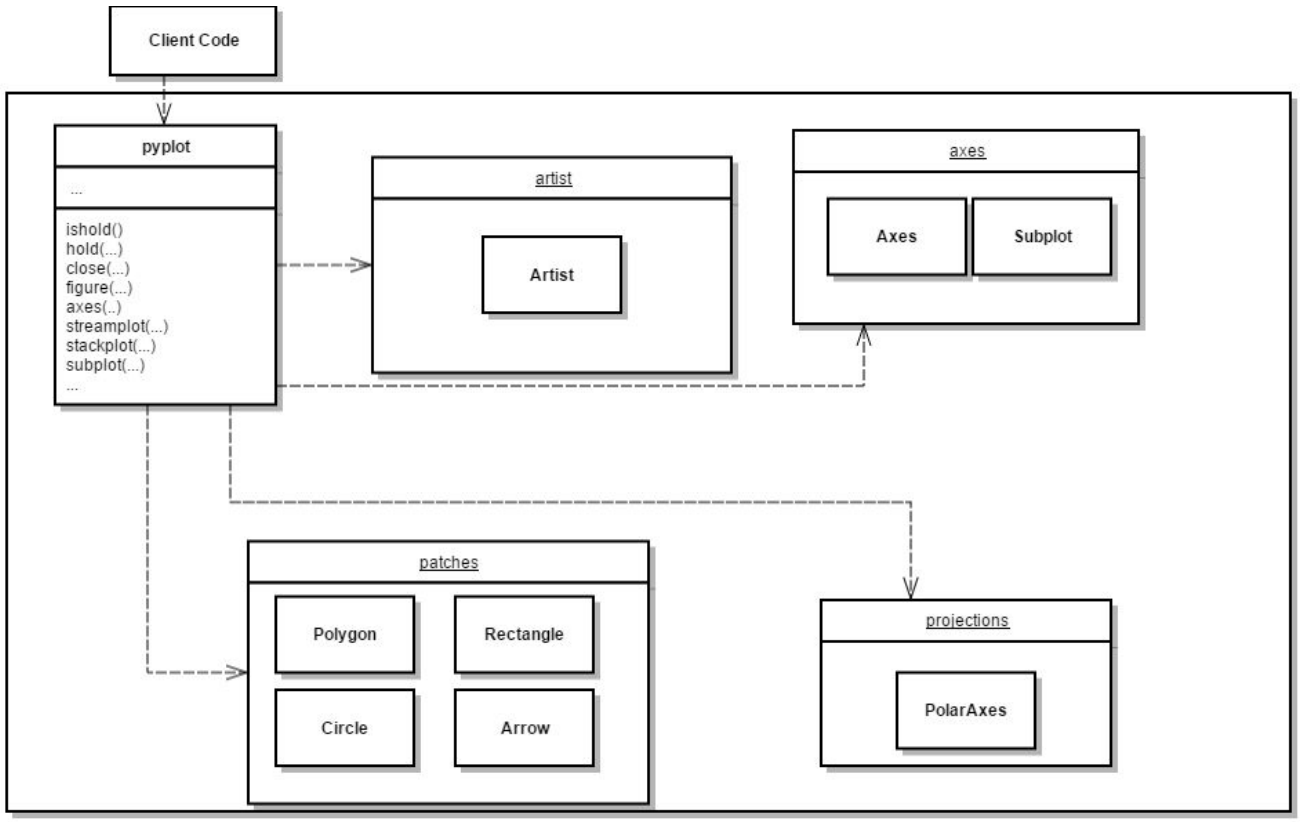


Facade

Facade design pattern is a pattern that uses a “facade” as a way to provide a simplified interface to hide a larger body of code, such as having a class that users can access to use all of the software library. The facade allows the library to become more readable and appears more clean to the users. More importantly, it reduces dependencies of outside code on the inner workings of a library.

UML Diagram for matplotlib (simplified version) :

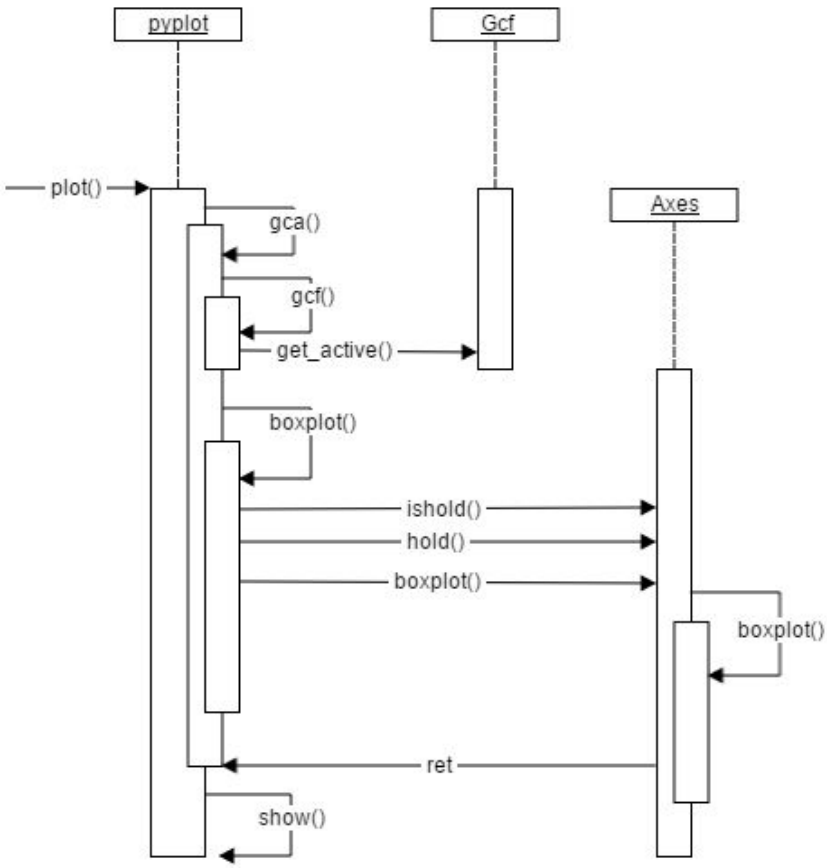
One way matplotlib uses facade design development is in pyplot.py, users calls pyplot and it uses gca() as an adapter to call other system functions such as axes.py, artist.py, patches.py and more. pyplot in this case acts as the facade class, the user uses the plotting functions from pyplot without calling the system function itself as wanted.



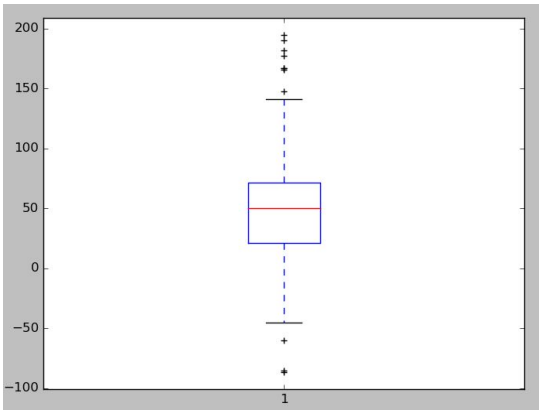
Example.
This function return the second x axis. twinx is from pyplot, it uses gca to call twinx from Axes class

```
def twinx(ax=None):
    """
    Make a second axes that shares the *x*-axis.  The new axes will
    overlay *ax* (or the current axes if *ax* is *None*).  The ticks
    for *ax2* will be placed on the right, and the *ax2* instance is
    returned.

    .. seealso::

       :file:`examples/api_examples/two_scales.py`
          For an example
    """
    if ax is None:
        ax=gca()
    ax1 = ax.twinx()
    return ax1
```

Sequence diagram of calling boxplot for example, sample code shown below:



Example code for calling a boxplot():




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

# some random points
spread = np.random.rand(50) * 100
center = np.ones(25) * 50
flier_high = np.random.rand(10) * 100 + 100
flier_low = np.random.rand(10) * -100
data = np.concatenate((spread, center, flier_high, flier_low), 0)

# plot it
plt.boxplot(data)

plt.show()
```

boxplot function from pyplot:
As soon as the user code calls boxplot(...) from pyplot, it uses gca() to call boxplot(...) from Axes class. When there is nothing in hold that is.

```
@_autogen_docstring(Axes.boxplot)
def boxplot(x, notch=None, sym=None, vert=None, whis=None, positions=None,
            widths=None, patch_artist=None, bootstrap=None, usermedians=None,
            conf_intervals=None, meanline=None, showmeans=None, showcaps=None,
            showbox=None, showfliers=None, boxprops=None, labels=None,
            flierprops=None, medianprops=None, meanprops=None, capprops=None,
            whiskerprops=None, manage_xticks=True, hold=None, data=None):
    ax = gca()
    # allow callers to override the hold state by passing hold=True|False
    washold = ax.ishold()

    if hold is not None:
        ax.hold(hold)
    try:
        ret = ax.boxplot(x, notch=notch, sym=sym, vert=vert, whis=whis,
                         positions=positions, widths=widths,
                         patch_artist=patch_artist, bootstrap=bootstrap,
                         usermedians=usermedians,
                         conf_intervals=conf_intervals, meanline=meanline,
                         showmeans=showmeans, showcaps=showcaps,
                         showbox=showbox, showfliers=showfliers,
                         boxprops=boxprops, labels=labels,
                         flierprops=flierprops, medianprops=medianprops,
                         meanprops=meanprops, capprops=capprops,
                         whiskerprops=whiskerprops,
                         manage_xticks=manage_xticks, data=data)
    finally:
        ax.hold(washold)

    return ret
```

Software Development Process:

Our team will focus on **Agile Software Development**

There are many reason to why we decided to use Agile:

1. The goal of fixing bugs depends entirely on the user/customer concern and satisfaction, so user collaboration is very important in the development.
2. Different size of bugs can be fixed and tested with different time and effort, so group management and allocation will be relevant during each of our iteration and sprint.
3. By the nature of the project, other groups might overlap our work and vice versa, we have to be quick to respond and make change to decision which Agile development process excels at.
4. Since there is limited amount of time for the project, have working prototype during each iteration is very important to ourselves and the customers.

Our group will follow an agile development process similar to the one in CSCC01:

- We will not be creating personas as we do not think that it will be useful in this situation (bug fixes and implementing features).
- Create user stories which correspond to bug fixes and features that need to be done.
- Assign priorities, time estimates, confirmation (testing validity) to each user story.
- Create a release plan which states what needs to be done in each release.
- Create iteration plans which states what needs to be done in that iteration, time re-estimates, task assignment and project velocity.
- Create task boards for each iteration which keeps track of what needs to be done, currently worked on and completed.
- Create and keep track of a burndown chart to see if we are progressing as planned.
- Verify user stories when they are completed and also perform internal code reviews.