CSCD01 - Team Name:

‘Scrum Till You Waterfall’

Deliverable 3

Team 13

Table of Contents

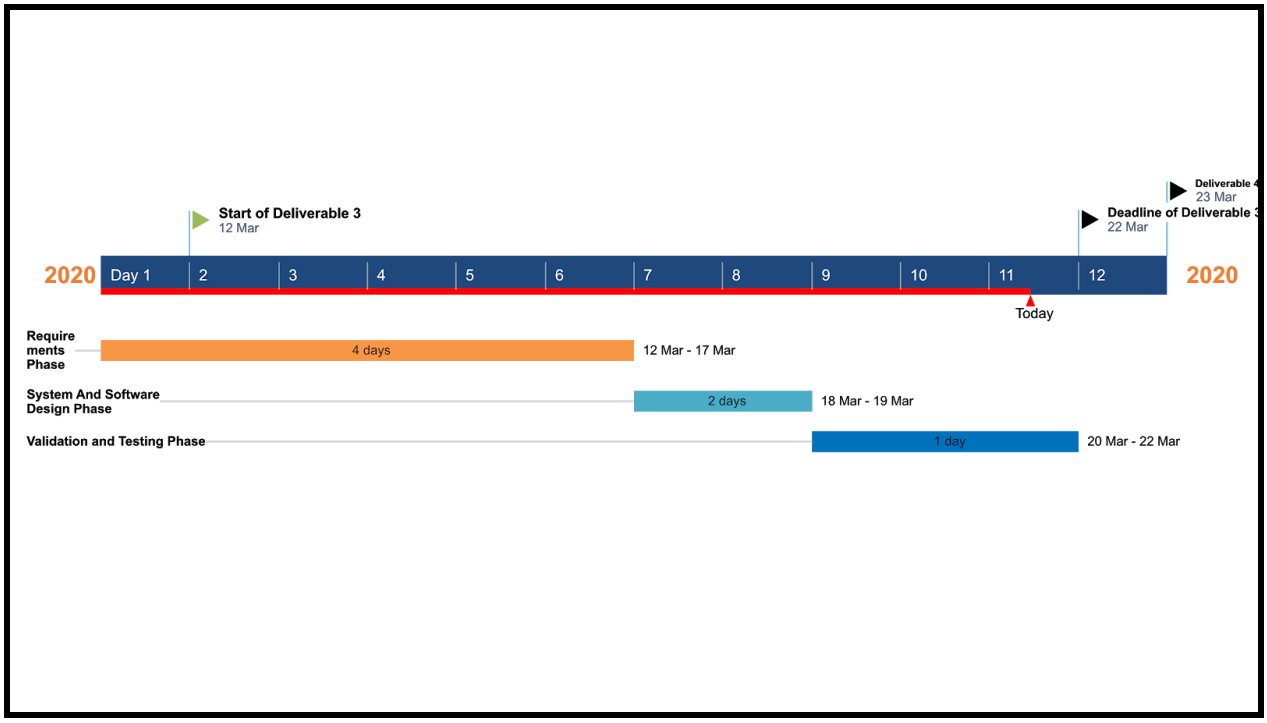
Plan of Action - Waterfall Software Development Process

Here are the five phases of Waterfall and how our team will follow them for this deliverable. Since this phase is more about planning for the development in the following deliverable, our process will be a more relaxed version of Waterfall.

- Requirements Phase (Deadline: Mar 17)
    - Investigate the 2 bugs that require substantial new development that we want to target for Matplotlib
    - For each bug,
        - we provide a description of the bug (expected vs actual, labels)
        - provide a UML for the architecture of the bug
        - provide code traces for the bug
        - and our plan of action on we are going to organize our new code and how it will interact with the existing system
    - Outcome of this phase will be a section of the document that provides all the detailed information for each of the 2 bugs chosen
- System And Software Design Phase (Deadline: Mar 19)
    - Determine which bug will be implemented and why.
    - Create acceptance test suite for the bug for if it were to be correctly fixed
    - Design the UML of the overall structure of system of Matplotlib
        - Decide what to include and what to omit
        - How the diagram will be structured
        - Describe design patterns followed + design principle
    - Talk about any solutions of interest that are derived from the UML.
    - Outcome of this phase will be a section of the document explaining why we chose one of the bugs of substantial new development to work on, an acceptance test suite for the fixed implementation of that bug, and the improved
- Validation and Testing Phase (Deadline: Mar 22) [extended due date]
    - Review of our documentation and the correctness of our UML
    - Outcome of this phase will be improved documentation via code review

Deliverable 3 Gantt Chart

This Gantt chart was based on our Plan of Action and is used to help us to keep track of our schedule for each phase of our software development process for this deliverable 3.



Created using https://online.officetimeline.com/

Documented Tasks

Task	Assigned Group Member	Start Time of Task	Estimated Time of the Task	Actual Completed Time of the Task
Description of Feature#1 - Quiver key	Abithan	Monday, March 16	10 min	10 min
Description of Feature#2 - QuadContourSet lacks remove method	Abithan	Monday, March 16	10 min	10 min
Feature#1 - Create Sequence Diagram	Anandha	Monday, March 16	2 hour	3 hour
UML/Diagram for Feature#1	Anandha	Tuesday, March 17	1 hour	1.5 hour
UML/Diagram for Feature#2	Andrew	Tuesday, March 17	1 hour	2 hour
Reason for choosing **Feature#1** to implement	Abithan	Thursday, March 19	30 min	20 min
Create **Acceptance Test** for Chosen Feature	Venkada	Thursday, March 19	1 hour	1 hour
Creating Overall Structure of Matplotlib **UML**	Venkada, Anandha, Jason	Thursday, March 19	1 hour	2 hour
Description of Overall **UML** + Design Patterns	Venkada	Thursday, March 20	3 hour	2 hour
Review of Document	Whole Team	Sunday, March 22	1 hour	1 hour

Requirements Phase

Here are the two bugs/features of substantial development that we chose to look at.

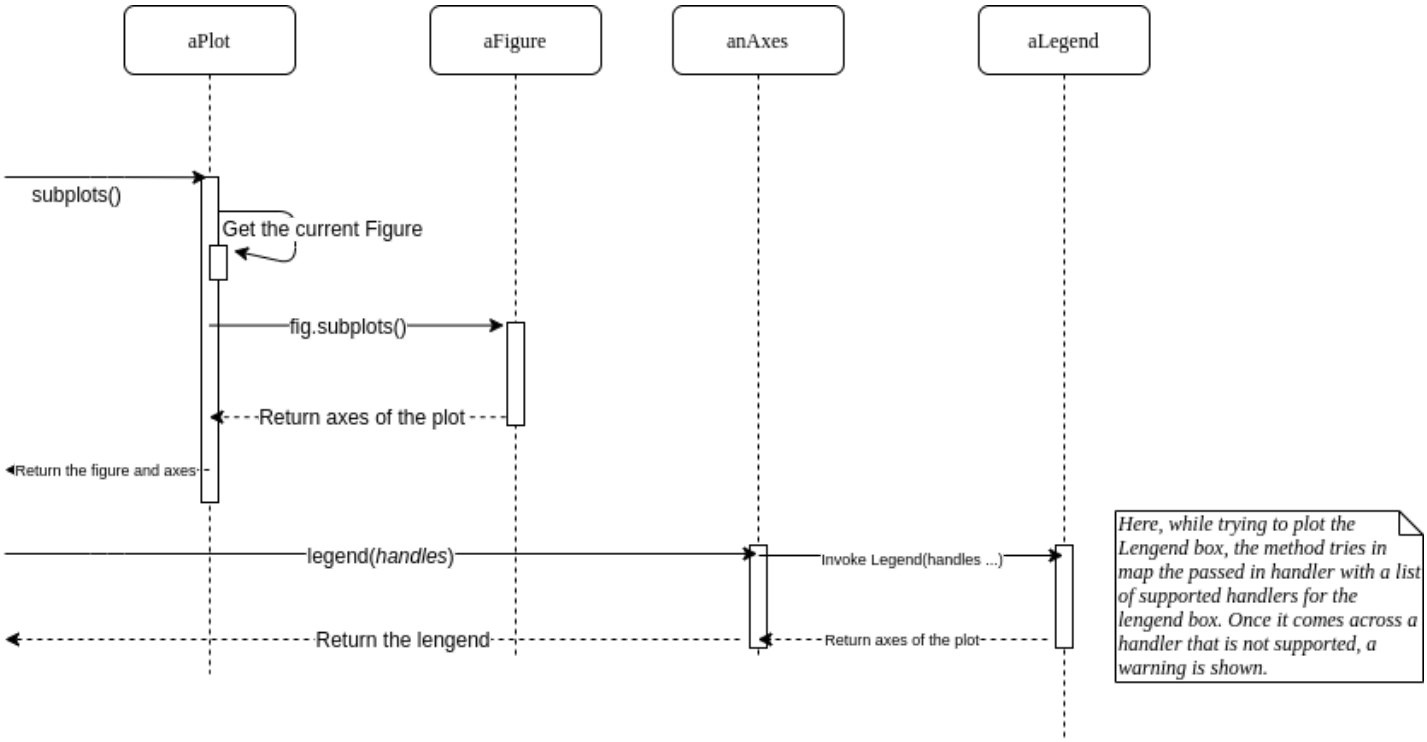
#1: Feature: https://github.com/matplotlib/matplotlib/issues/16664

Description:

This feature mainly revolves around creating a new class called QuiverKey from scratch that acts as a proxy artist. More details about proxy artists can be found here: http://matplotlib.org/users/legend_guide.html#creating-artists-specifically-for-adding-to-the-legend-aka-proxy-artists. The main purpose of this feature request was in order to display properties of a QuiverKey object in the legend which would be a consequence of passing the QuiverKey object from the proxy artist class we construct into the legend handler that Matplotlib uses. This feature will require substantial development as it involves investigating how proxy artists work and extending it to create a class that supports QuiverKey objects. Tackling this issue will consist of mainly developing a proxy artist class, documentation, as well as writing unit tests for it.

UML Diagram for Bug/Feature:

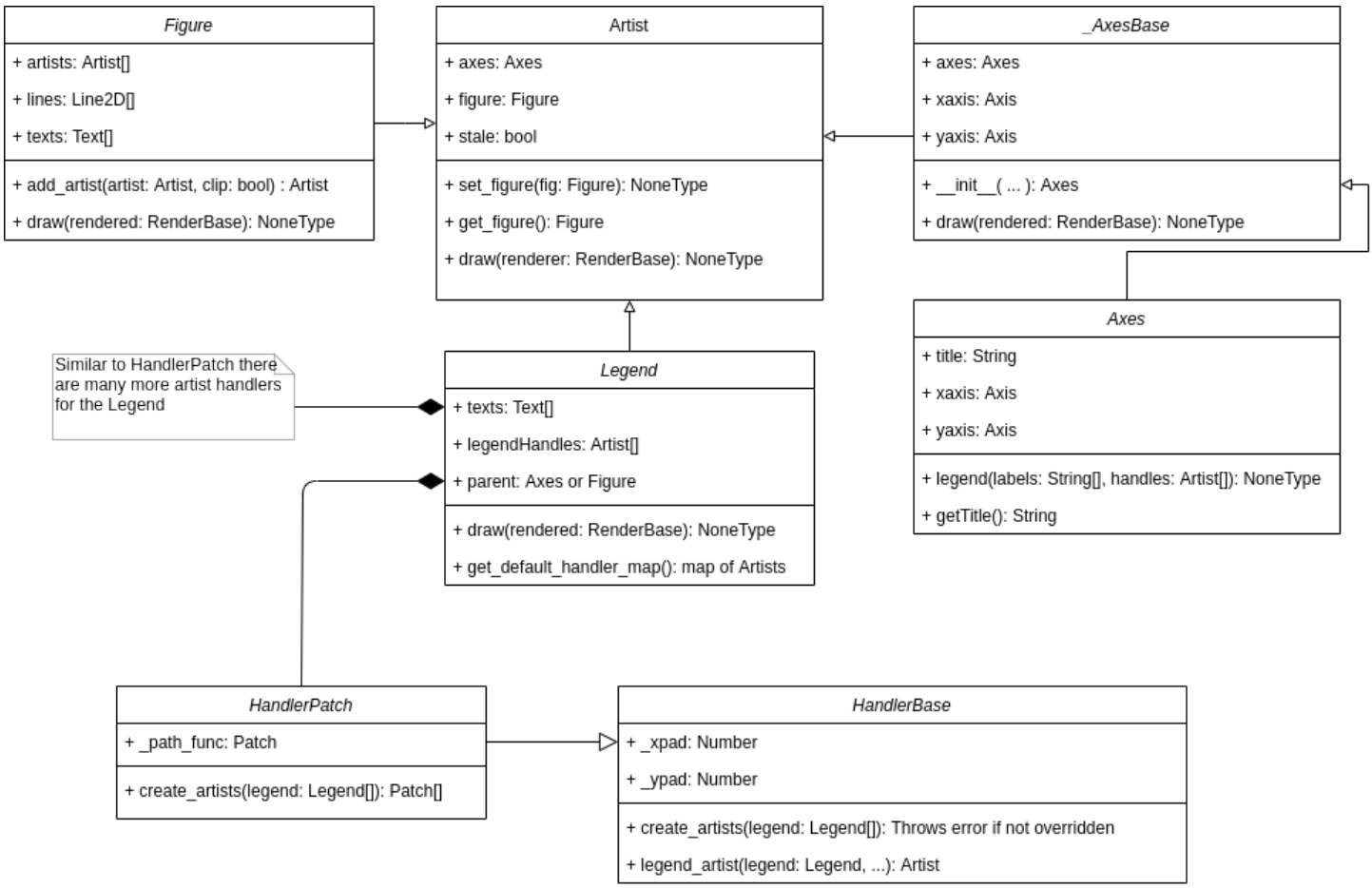
Here is an overall sequence diagram illustrating the major players during the process of generating a legend box using a QuiverKey handler.



The above diagram shows an example sequence of events that triggers matplotlib to show a warning that indicates a handler of type QuiverKey is not supported. The major players involved while generating a Legend box for a Quiver plot using a QuiverKey is as follows:

1) PyPlot
2) Figure
3) Axes
4) Legend

The relationships between the above mentioned objects can be expressed as the diagram:



The Pyplot is omitted from the above diagram as it only acts as a driver in the scripting layer and it does not help in visualizing the interactions between the other classes. The warning is thrown when the QuiverKey is passed in as an Artist Handler for the Legend since there does not exist a corresponding '*HandlerQuiverKey'* class defined *in* `*legend_handler.py*`.

To address the warning, a class named `*HandlerQuiverKey*` must be implemented in `*legend_handler.py*` and composed within the Legend Class.

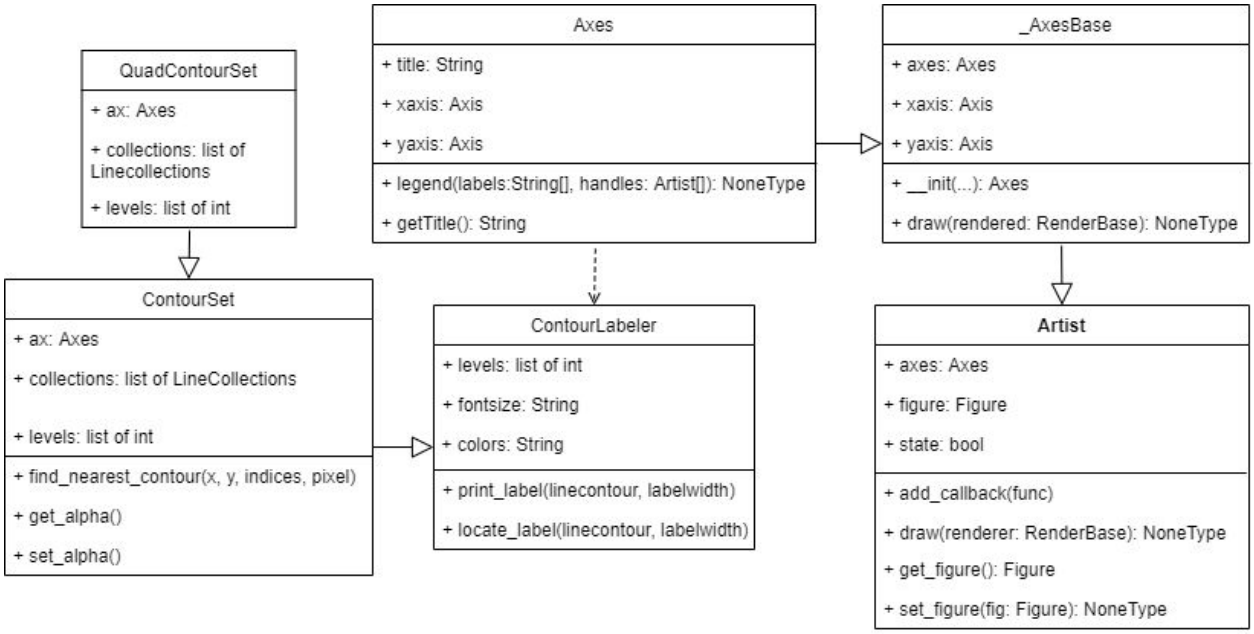
***#2: Feature*:** https://github.com/matplotlib/matplotlib/issues/5332

Description:

This feature involves enhancing an already existing class with some new functionality. Specifically, the QuadContourSet class currently lacks a remove method that is currently restricting users of the software from being able to remove objects. subplots, and contours from plots created using the QuadContourSet. With this added functionality a user of the QuadContourSet class would be able to modify existing plots that use the class. This issue we could be tackling would require adding a remove method, documentation, and the necessary unit tests for it.

UML Diagram for Bug/Feature:

Here is a generalized UML for the QuadContourSet class and the classes that relate to it.



From this UML, it can be seen that the contour classes are classes which the Axes class uses in order to support contour plotting and labeling.

If we were to implement the new feature of creating a remove method for QuadContourSet, we would either create that method in QuadContourSet or its parent ContourSet. Additionally, we would need to test to make sure that the removal of the object does not break with any of the other classes when running a matplotlib program and that it works with the pyplot class.

For example, by running the following code,

```
import matplotlib.pyplot as plt
import numpy as np
w = 4
h = 3
d = 70
plt.figure(figsize=(w, h), dpi=d)
x = np.arange(-2, 2, 0.25)
y = np.arange(-2, 2, 0.25)
x, y = np.meshgrid(x, y)
z = np.sin(x * np.pi / 2) + np.cos(y * np.pi / 3)

plt.contour(x, y, z)
plt.savefig("out.png")
```

this would plot a contour plot on a graph. An additional line of code to test this would be to add a .remove() function in order to test the functionally of the new feature we would have created. This new code would not change any of the existing functionalities of the old code as adding the new feature only affects the QuadContourSet object.

System and Software Design Phase

The feature that we chose to implement from the two that we shortlisted was “Add ability to add QuiverKey object to Legend” as seen by this link from Github, https://github.com/matplotlib/matplotlib/issues/16664.

Reason For Choosing Feature#1 To Implement

The reason we chose this feature to implement was because of the following. We considered quite a few of the issues and feature requests on Github before shortlisting the two above and then carefully deciding on this new feature to add a proxy artist class for QuiverKey objects. As we have decided to practice waterfall methodology for this project, it makes sense to implement a feature like this, where we can complete all the design work beforehand. We are able to complete a class diagram with the required methods and acceptance tests well in advance. There is also a low chance of changes to the design being made, so we will likely not have to cycle in the waterfall methodology. Furthermore, given our limited timeline of a week and a half to implement and test this feature, we believe that this feature will have our development team busy, but still be able to deliver a polished product in the end. This feature was also interesting enough to experiment with as there is already documentation on how to create a class of this nature, and the ability to test the end result is relatively straightforward, all things that developers love to hear. Although it may be a substantial and difficult feature, we have confidence in our team and software process in order to deliver.

Additionally, the reason we didn’t choose the other feature “QuadContourSet lacks remove method” from https://github.com/matplotlib/matplotlib/issues/5332, was because we thought that there wasn’t enough substantial work to implement for that feature. Even though it had a label for “New Feature”, we thought that it wasn’t that hard to just create a new remove method for an already existing object. There were also other examples and documentation for creating a remove method from other objects such as Line2D in the _axes.py that we could have based our implementation off of, which would lessen the amount of work needed to be done even further.

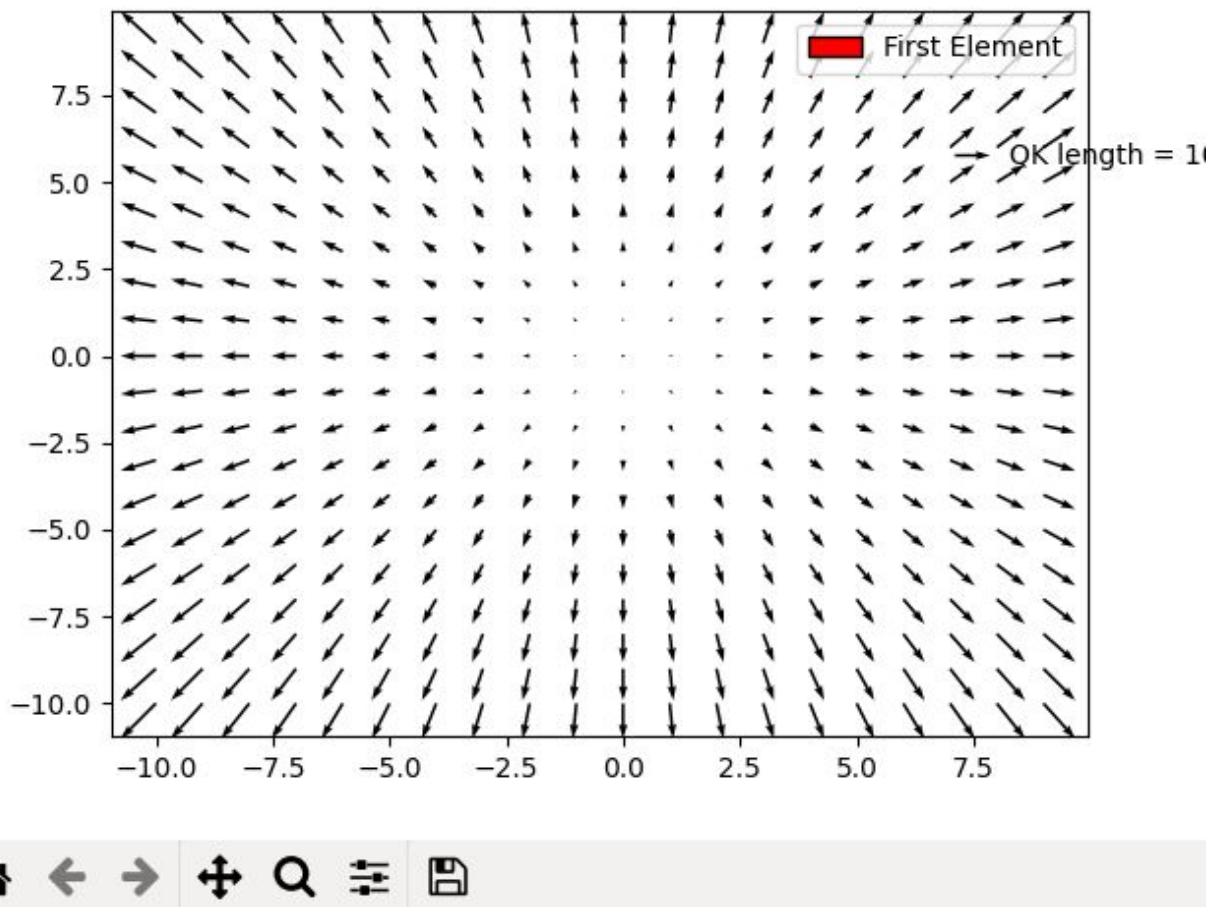
Acceptance Criterion for the Feature

```python
from matplotlib.patches import Patch
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()

X = np.arange(-10, 10, 1)
Y = np.arange(-10, 10, 1)
U, V = np.meshgrid(X, Y)

q = ax.quiver(X, Y, U, V)

qk = ax.quiverkey(q, 0.9,0.8, U=10, label='QK length = 10', labelpos='E')

legend_elements = [
    Patch(facecolor="red", edgecolor="black", label="First Element"),
    qk
]

ax.legend(handles=legend_elements)

plt.show()
```

The actual outcome from running the code above is shown below:

```
demo.py:20: UserWarning: Legend does not support <matplotlib.quiver.QuiverKey object at 0x7fd03aa5beb8> instances.
A proxy artist may be used instead.
See: http://matplotlib.org/users/legend_guide.html#creating-artists-specifically-for-adding-to-the-legend-aka-proxy-artists
  ax.legend(handles=legend_elements)
```

The warning above is given because as of now the legend does not support a constructor of type Quiverkey to be part of the Legend.



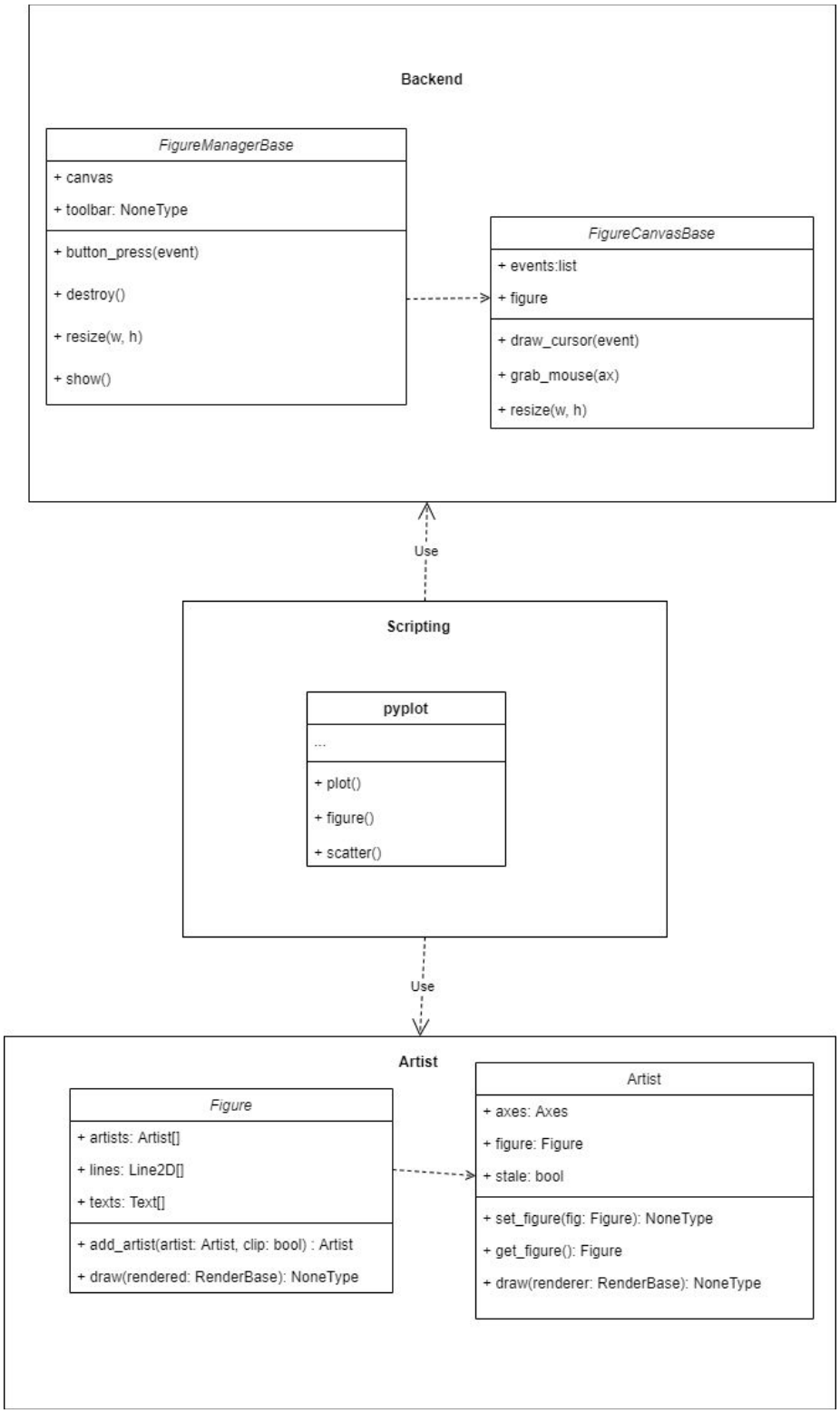
Along with a quiver plot missing the quiver key in the legend

The expected outcome after implementation:

- Quiver key should be present as another element in the legend box
- There should not be a warning in the console stating that the QuiverKey is not a support constructor for the Legend

Overall Architecture of System - UML

UML:



Description of UML:

This UML is split into 3 different sections. The architecture of Matplotlib uses a backend layer which will serve to prepare the canvas of the graphs and all the information to be displayed. The artist layer which will work to display the information onto the screen. Lastly the scripting layer which acts as a middle layer to communicate with the artist and backend layer, allowing users to write code for using Matplotlib.

The artist layer is tied to the backend via the draw method. An artist subclass must implement this method and take in a renderer obtained from the backend. The artist will not know what renderer nor backend is used but it will use the renderer's methods to appropriately display the information onto the screen.

Design Patterns:

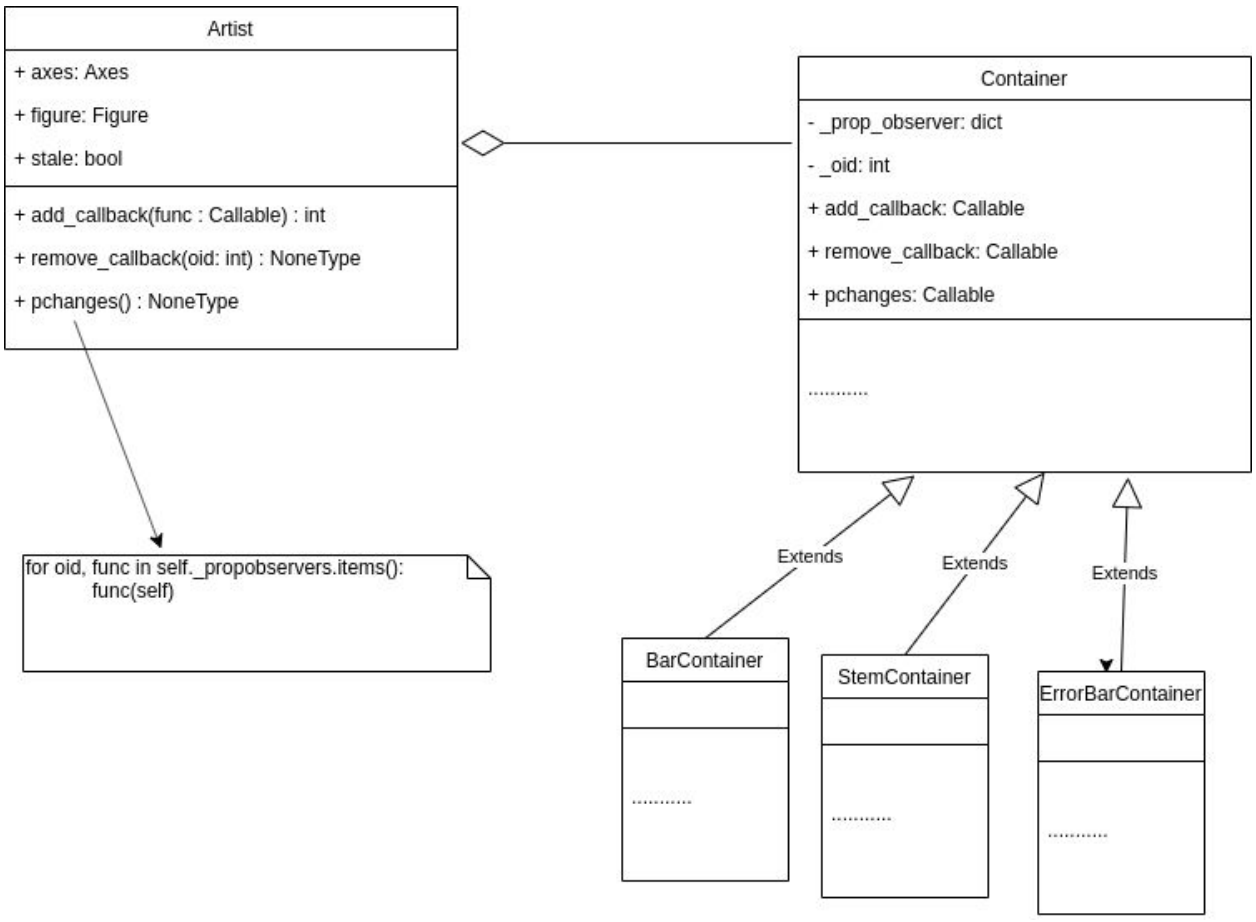
Facade Design Pattern:

Pyplot which is the scripting layer of the matplot application acts as the facing object. It creates an user interface to make it easier for the user to interact with other backend and artist layers. It hides the complexity that has to happen behind the scenes and therefore makes it plain and simple for the user to use.

Iterator Design Pattern:

The iterator pattern can be seen in some of the classes in the matplotlib. For example the **FixedArtistHelper** in **grid_helper_curvelinear.py** uses a helper class called GridHelperCurveLiner in the same file which has a method called get_tick_iterator. This get_tick_iterator() returns an iterator for the container to iterate over the ticks.

Observer Pattern:



There are various places where the observer patterns are present in the project. One particular example is as follows:

The **Artist** class keeps record of the observers and the callbacks for when the property of the artist changes. The method add_callbacks() adds the callbacks, remove_callbacks() removes the callbacks, pchanged() notifies the observers by executing the callbacks. The **Container** class is the **Observer** which can use callables add_callback, remove_callback, pchanges to make necessary updates. BarContainer, StemContainer, and ErrorBarContainer are some classes that inherit from Container and will be the **concreteObservers.**

Interesting Solutions in the Architecture:

The three layers can be stacked on top of each other in the order scripting, artist and then finally backend from top to bottom. Through this stacking we can recognize that the layers at any layer will only know about itself and the layers below it. This is also known as open layered architecture.

An open architecture is used for easier maintainability of the matplotlib. As it is an open source project, several developers will need to work on the software and will need to learn the basics without it being a huge learning curve. Secondly, open architecture is known for its performance. Compared to a closed architecture which can only access the layer below, an open architecture can call any layer below it making it faster.

One interesting solution for the artist layer is that there are two different types of artists. One is the primitive artist component such as text, basic shapes, and lines. Another is the composite artist components which utilize multiple artists inside of them. This creates many components stemming from the artist class and dependencies among the components. This does help to separate the responsibilities of the components but might be difficult to manage for components that depend on a lot of other components.