TWENTY 6IXERS

DELIVERABLE 3

Stefan Mitic

Kevin Bato

Chia Anamekwe

Zongda Wang

Byron Leung

TABLE OF CONTENTS

Feature #1

Feature request: add edit_parameters to toolbar for all GUI backends #13195
https://github.com/matplotlib/matplotlib/issues/13195

Description

The feature should allow users to edit the parameters of a Qt, GTK, Tk, or WX-embedded figure through the GUI. The user will be able to edit figure parameters including the title, x/y-axis limits, labels, and line/marker styles.

Changes Required

in matplotlib.backends.backend_gtk3:

- This file contains classes for many components that can be included in a GTK window including the NavigationToolbar, which currently contains buttons that allow the user to do actions such as pan, zoom, and save the figure. We will add our parameter editor tool to this toolbar, so we must create and add a button for it, which can be done in the initialization function of the class. The button will call another function within the class that we need to implement which will display and configure the GUI element for our parameter editor.

in matplotlib.backends._backend_tk:

- Same as above but for Tk backend

in matplotlib.backends.backend_wx:

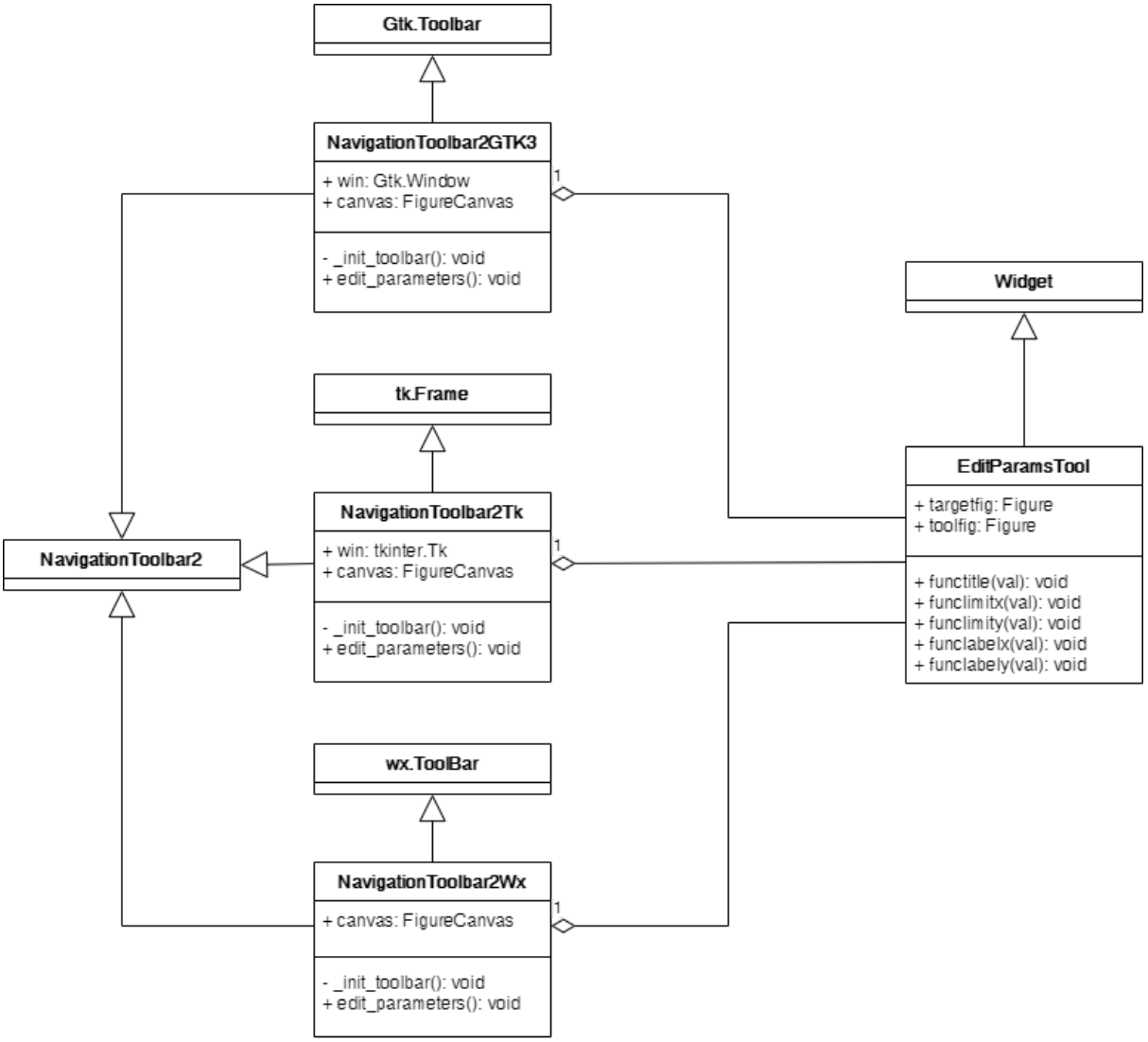
- Same as above but for WX backend

in matplotlib.widgets:

- Here we will define a new Widget class for our tool which creates the actual GUI element containing all the textboxes, dropdowns, checkboxes, etc., that the user will use to edit the figure. Fortunately there are some base widget classes for some of these elements that we can use within ours.
- Within our new widget class we must also implement and attach the functions that will actually interact with the figure and modify it.

in matplotlib.backend_bases:

- If we successfully implement the parameter editor for all the backends, we integrate it for the base NavigationToolbar2 class which is inherited by the navigation bars for all the backends.


Gtk.Toolbar
NavigationToolbar2GTK3
+ win: Gtk.Window
+ canvas: FigureCanvas
- _init_toolbar(): void
+ edit_parameters(): void
1
Widget
tk.Frame
EditParamsTool
+ targetfig: Figure
+ toolfig: Figure
+ functitle(val): void
+ funclimitx(val): void
+ funclimity(val): void
+ funclabelx(val): void
+ funclabely(val): void
NavigationToolbar2Tk
+ win: tkinter.Tk
+ canvas: FigureCanvas
- _init_toolbar(): void
+ edit_parameters(): void
1
NavigationToolbar2
wx.ToolBar
NavigationToolbar2Wx
+ canvas: FigureCanvas
- _init_toolbar(): void
+ edit_parameters(): void
1

Feature #2

Colormaps should have a `_repr_html_` that is an image of the colormap #15616
https://github.com/matplotlib/matplotlib/issues/15616

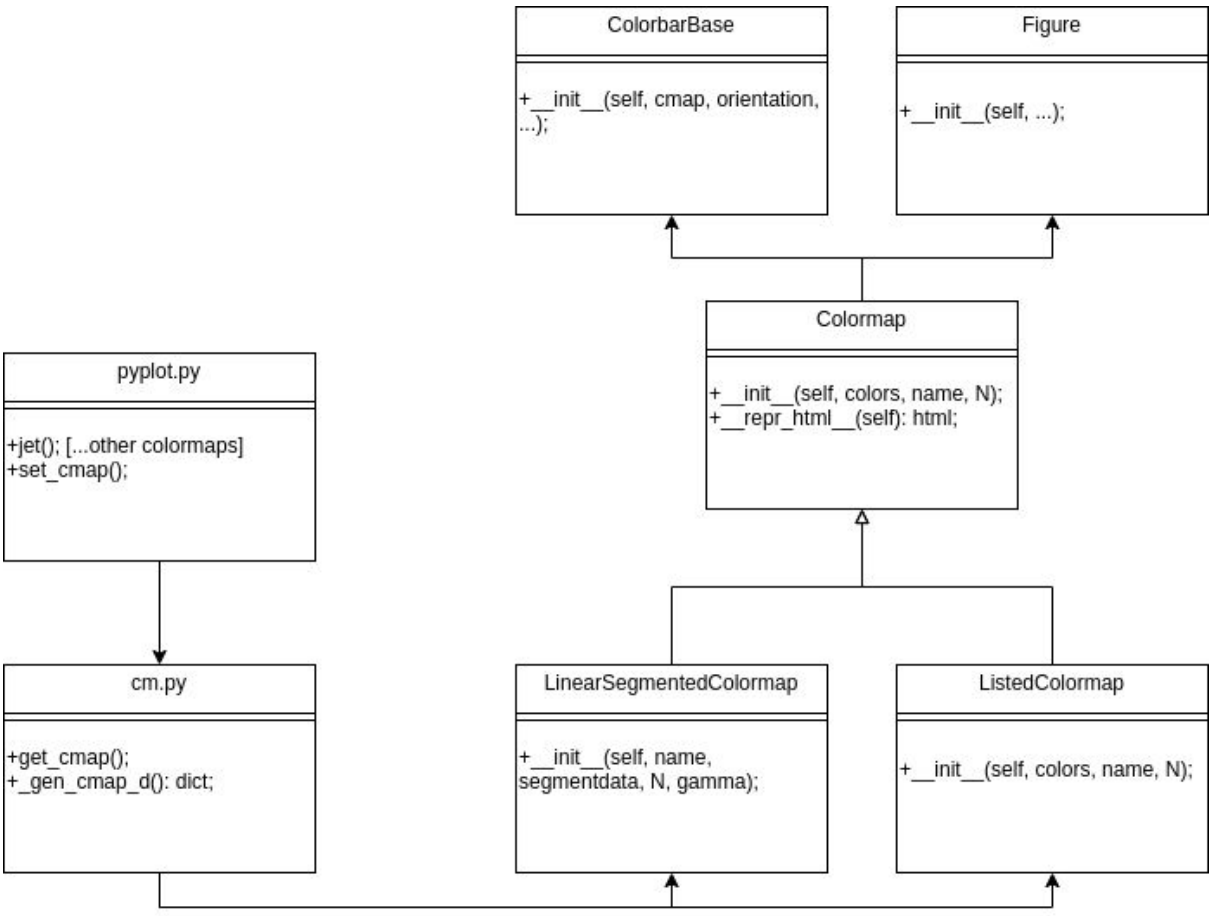
Description

The feature will allow the ability for a user that uses an external platform such as jupyter notebook to display inline images of colorbars from matplotlib. This feature will use html to work with browsers giving the user an image of the colorbar inline with the notebook platform of their choice.

Changes Required

The feature will be integrated using the default structure from iPython that is a library used by notebook platforms to display inline figures from matplotlib. The code that will be changed to integrate such a feature will be within the ColorMap class that will include '_repr_html_' to represent the ColorMap through inline. This method is invoked by iPython when displaying Figure objects.

Within pyplot there exists a function that sets the corresponding map that is included with the different Colormap objects existing within matplotlib. In order to implement a representation for notebook platforms, we must define a function within Colormap that creates a Colorbar object and a Figure object in order to display the Colormap with inline.


ColorbarBase
+__init__(self, cmap, orientation, ...);
Figure
+__init__(self, ...);
Colormap
+__init__(self, colors, name, N);
+__repr_html__(self): html;
pyplot.py
+jet(); [...other colormaps]
+set_cmap();
cm.py
+get_cmap();
+_gen_cmap_d(): dict;
LinearSegmentedColormap
+__init__(self, name, segmentdata, N, gamma);
ListedColormap
+__init__(self, colors, name, N);

Feature #3

NonUniformImage does not respond to ax.set_scale('log') #13442
https://github.com/matplotlib/matplotlib/issues/13442

Description

When an image created by NonUniformImage is plotted on a logarithmic scale along the x-axis, the image does not change due to scaling. It is plotted as if the x-axis were on a linear scale which is not intended.

The main issue with this bug is that NonUniformImage does not apply any transformations on the image when rendered. By default it is rendered on a square coordinate system. When ax.set_scale('log') is executed, the transformation to logarithmic scales are applied only on axes and not on the image.

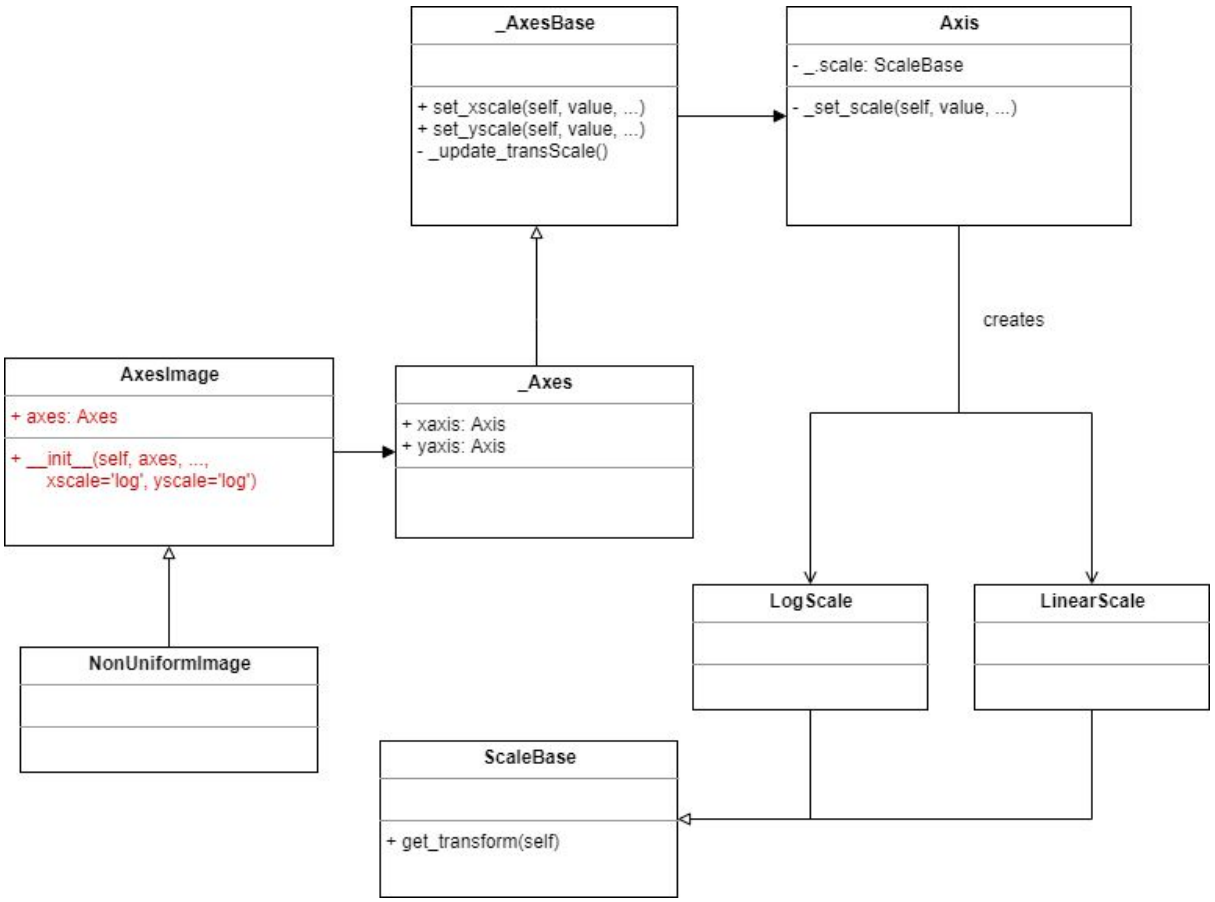
Changes Required

Looking into how Axis class transforms their axis, a property called _scale stores the current scale of the axis. It calls a factory method Scale.scale_factory(...) which returns a scale class, for example: LinearScale, LogScale, etc. Later into the function calls, Axes._update_transScale() will use the stored Scale class to transform the axis accordingly. Replicating this design onto AxesImage will provide a synchronized behavior to both axis and image.

in matplotlib.image.AxesImage:

- This class is primarily used for mounting an image onto the axes using an extent property that takes a tuple (left, right, bottom, top) and fixes the position of the image according to the provided coordinates. It does not account for any image transformations.
- First, we need to provide a transformation method to transform the image directly from AxesImage using the provided scale factory method. Here, we do not need to provide an Axis object since AxesImage will have access to it through an Axes object, which means _scale property is provided as well and setting the _scale property using _set_scale method where the scale factory resides in. This will help set the scale accordingly. We implement the transformation to the image upon creation then mount the image on the axis.

- Lastly, to synchronize both image and axis to the same scaling, the axis needs to be transformed. This is done by reusing their method, Axes._update_transScale() by using the same scaling property _scale that we previously set.



The red text on the UML is modified or added code into the class. The addition of the two parameters are optional in the __init__ method of AxesImage. By default, the scale will be linear, if the user chooses to set the scaling differently they can specify which axis they choose to scale. Keeping in mind that after creating this image on a plot and setting the axis after using ax.set_scale will overwrite the current scale, this will desynchronize both axis and image.

The design of this fix is to reduce the amount of code written and reuse methods already provided to reduce the risk of breaking the overall architecture. An added bonus or unintended feature of this fix is all classes inheriting AxesImage, images (not only NonUniformImage) will be able to transform with the axis.

Feature Choice

The feature we chose to work is Feature #1: add edit_parameters to toolbar for all GUI backends #13195". We chose to implement this feature because it is very useful allowing users an option to real-time edit their plots by providing a toolbar. The parameter editor is currently available on one backend (Qt) and not the others, presumably because of the difference in implementation, but the feature has proven to be useful and it makes sense to standardize it throughout the remaining GUI backends, which also lends to consistency. Furthermore, it requires us to define new classes and several functions which will be integrated into the system, and we feel like it is an appropriate amount of work for the next deliverable.

Feature #2 does not seem to be as useful as it is more of a "fancier" feature that may be used only by several users because it's scope includes colormaps, not plots in general.

Feature #3 is more of a major bug fix and due to the lack of information, there isn't a direct approach to implementing such a major change. Therefore, Feature #1 provides a good amount of work for our team and we are confident in implementing this feature with our skills.

Acceptance Tests

Navigation toolbar

Tests:

1. Create and display a GTK-embedded matplotlib figure with a navigation toolbar.
2. Create and display a Tk-embedded matplotlib figure with a navigation toolbar.
3. Create and display a WX-embedded matplotlib figure with a navigation toolbar.

Expected outcome:

- A GUI window should pop up containing the matplotlib figure and navigation toolbar.

Opening the parameter editor

Tests:

1. Click the parameter editor tool icon on the GTK navigation bar.
2. Click the parameter editor tool icon on the Tk navigation bar.
3. Click the parameter editor tool icon on the WX navigation bar.

Expected outcome:

- The parameter editor tool should pop up in a new window.

Modifying axis properties

Tests (for each backend):

1. Change the x or y-axis limits of an axes through the parameter editor.
2. Change the x or y-axis label of an axes through the parameter editor.

Expected outcome:

- The axis properties on the figure should change accordingly.

Modifying line properties

Tests (for each backend):

1. Change the label of a line through the parameter editor.
2. Change the style of a line (dashed, dotted, etc.) through the parameter editor.
3. Change the width of a line through the parameter editor.
4. Change the color of a line through the parameter editor.

Expected outcome:

- The properties of the line in the figure should change accordingly.

Modifying marker properties

Tests (for each backend):

1. Change the style of a marker (point, star, etc.) through the parameter editor.
2. Change the size of a marker through the parameter editor.
3. Change the color of a marker through the parameter editor.

Expected outcome:

- The marker in the figure should change accordingly.

Generating a legend

Tests (for each backend):

1. Generate a legend for an axes with lines or markers.

Expected outcome:

- A legend should be generated for the desired axes containing all the labeled lines and markers in it.

Multiple toolbars and axes

Tests (for each backend):

1. For a figure with multiple axes and navigation toolbars, open the parameter editor for one of them and change some parameters.

Expected outcome:

- The desired axes should change leaving the others unaffected.

Multiple lines in an axes

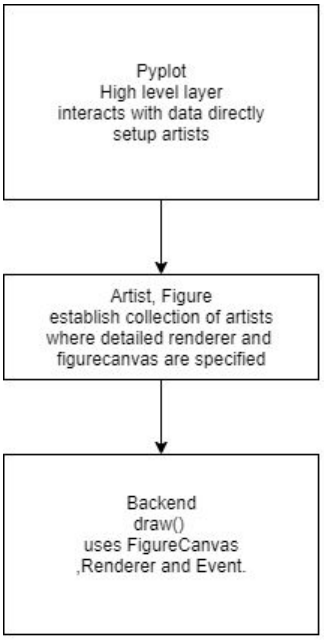
Tests (for each backend):

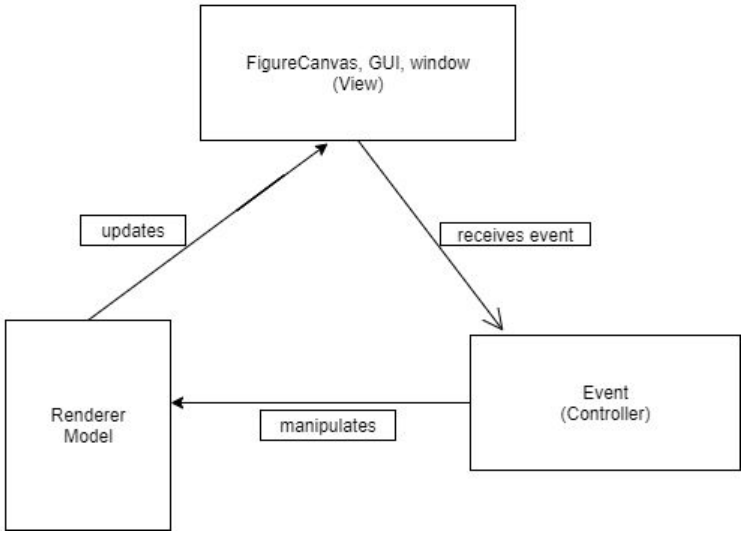
1. For an axes with multiple lines, select and change the properties of one of them.

Expected outcome:

- The desired line should change, leaving everything else unaffected.

Architecture Overview

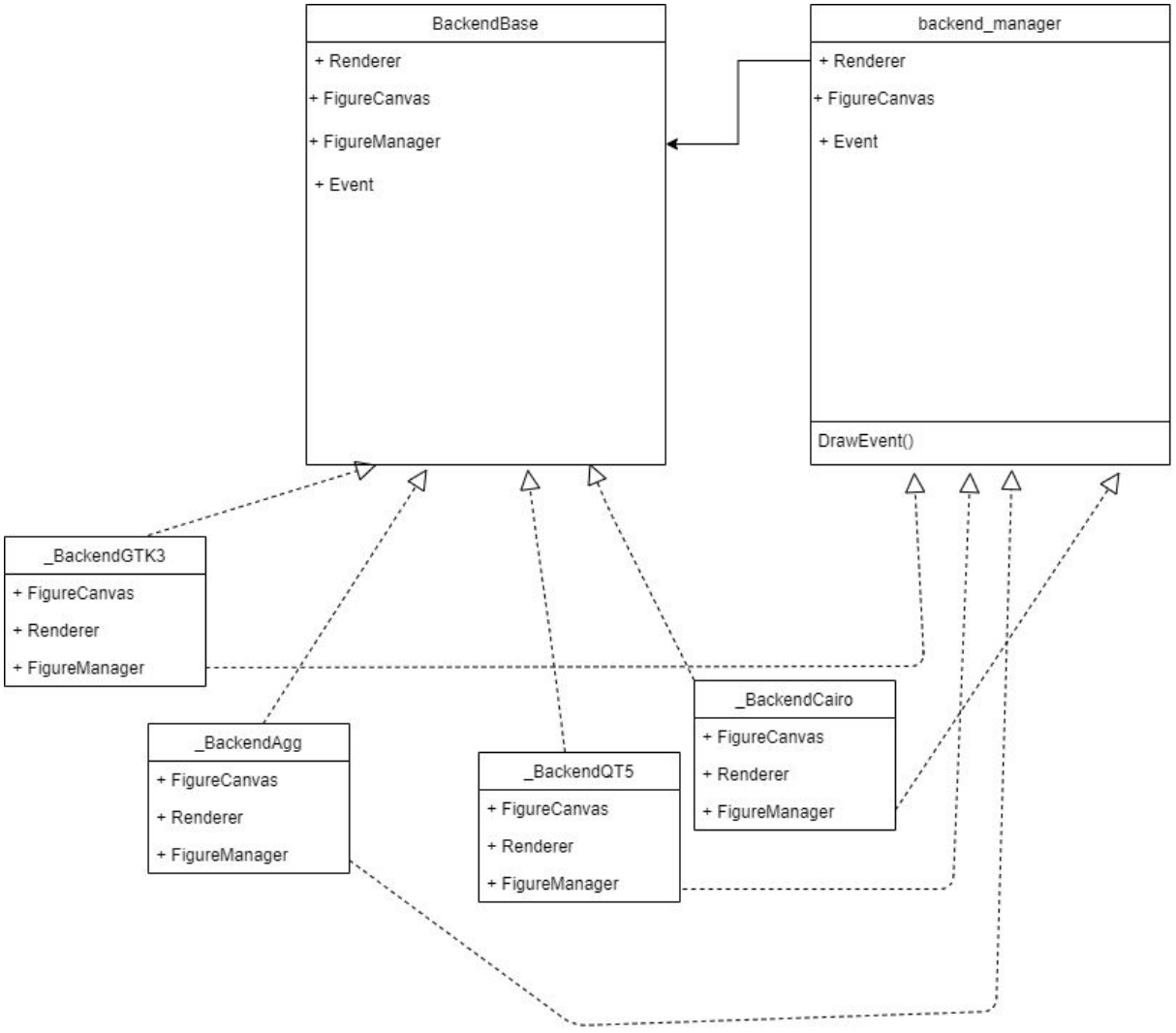




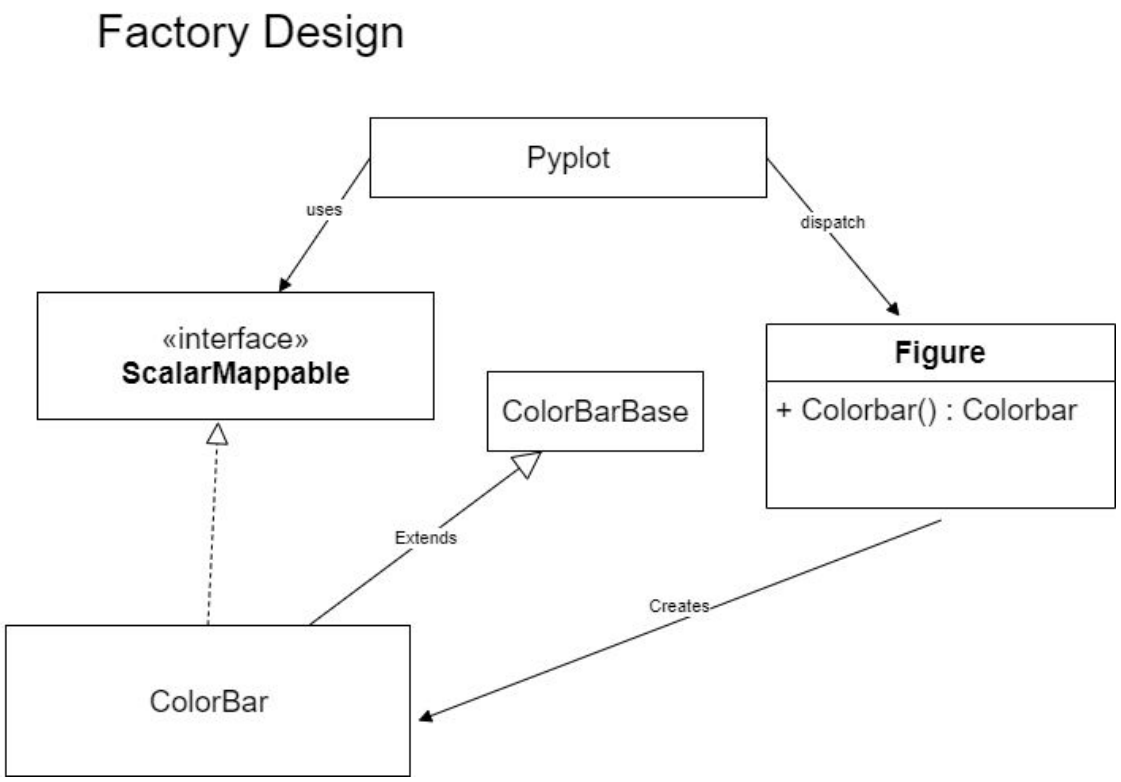
After pyplot.show() is called, the backend itself has the behavior of a MVC model where FigureCanvas together with GUI acts like view for users, by interaction with the window the user is “creating” the event where it acts like controller to manipulate Renderer in backend affecting and updating the view.

Twenty 6ixers

Matplotlib Backend

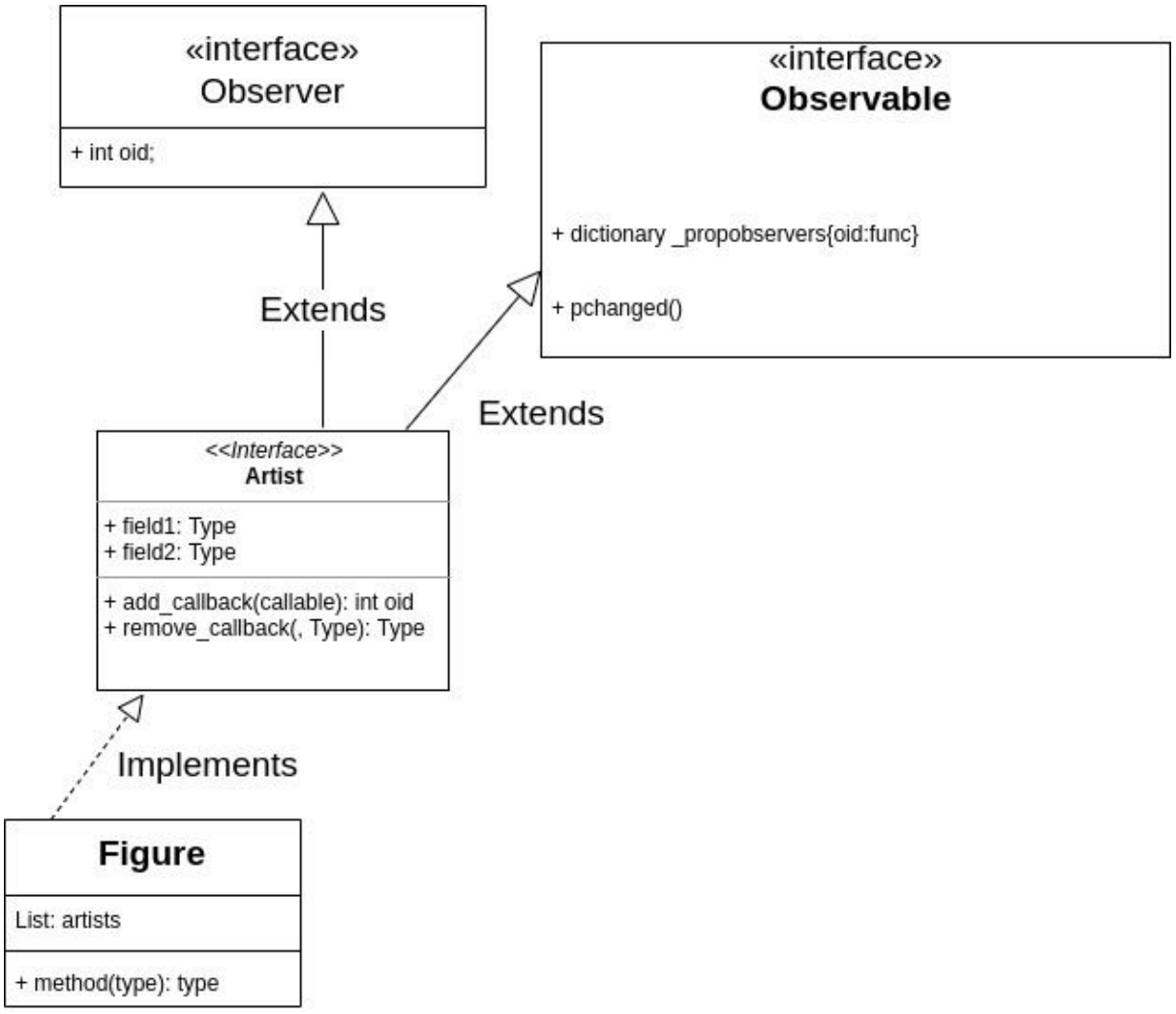


Backends used in different GUI and platforms have different adapters allowing them to be mapped by Artist objects by implementing the BackendBase interface, where each specific backend class has an instance of FigureManager that implements an interface, backend_manager with similar concept.



Pyplot acts as a client which sends requests to a Figure object where it acts like a factory, in this case, to create a colorbar instance within the figure object. It will then be able to provide it back to Pyplot without calling colorbar's own constructor directly.

Observer Design Pattern



The artist class is both an observer and an observable to each other, allowing updates when one of the artists changes. This will toggle function pchanged(), hence, it invokes the callback functions with respect to their observer stored in the dictionary.