CSC D01

11/10 (aka 110%)

Deliverable 2

By: Ben Cooper, Kirisanth Ganeshamoorthy, Matthew Kewarth, and Viola Ou

Table of Contents

1. Development Process 2
2. Architecture of the System 3
3. Observer Pattern 5
    a. Structural View
    b. Behavioural View
    c. Explanation
4. Iterator Pattern 7
    a. Structural View
    b. Behavioural View
    c. Explanation
5. Singleton Pattern 9
    a. Structural View
    b. Explanation
6. Factory Method 10
    a. Structural View
    b. Explanation

Development Process

As a team we have decided upon a unique hybridization of the Waterfall and Agile design processes in an attempt to merge in our own predetermined feature list while maintaining an Agile development cycle to conform to deliverable deadlines. Following the requirements and design phases of a classic waterfall process allows us to draft a prepare a backlog of requested features and bug fixes to deploy and solve over the lifespan of the project; comparatively an Agile development cycle provides us with the resources needed to meet deadlines that Waterfall would otherwise would leave lacking.

Our proposed hybrid process design:

To give you an idea of our design process here is a outline of stages:

[Waterfall]

1) Research requested features and bug fixes, add possible candidates to the - product backlog.

2) Analyze candidates, assigning cost and value. Plan for future Agile-based sprints.

[Agile]

3) Execute sprints, adjusting plans with each sprint for a more accurate - representation of the team capabilities.

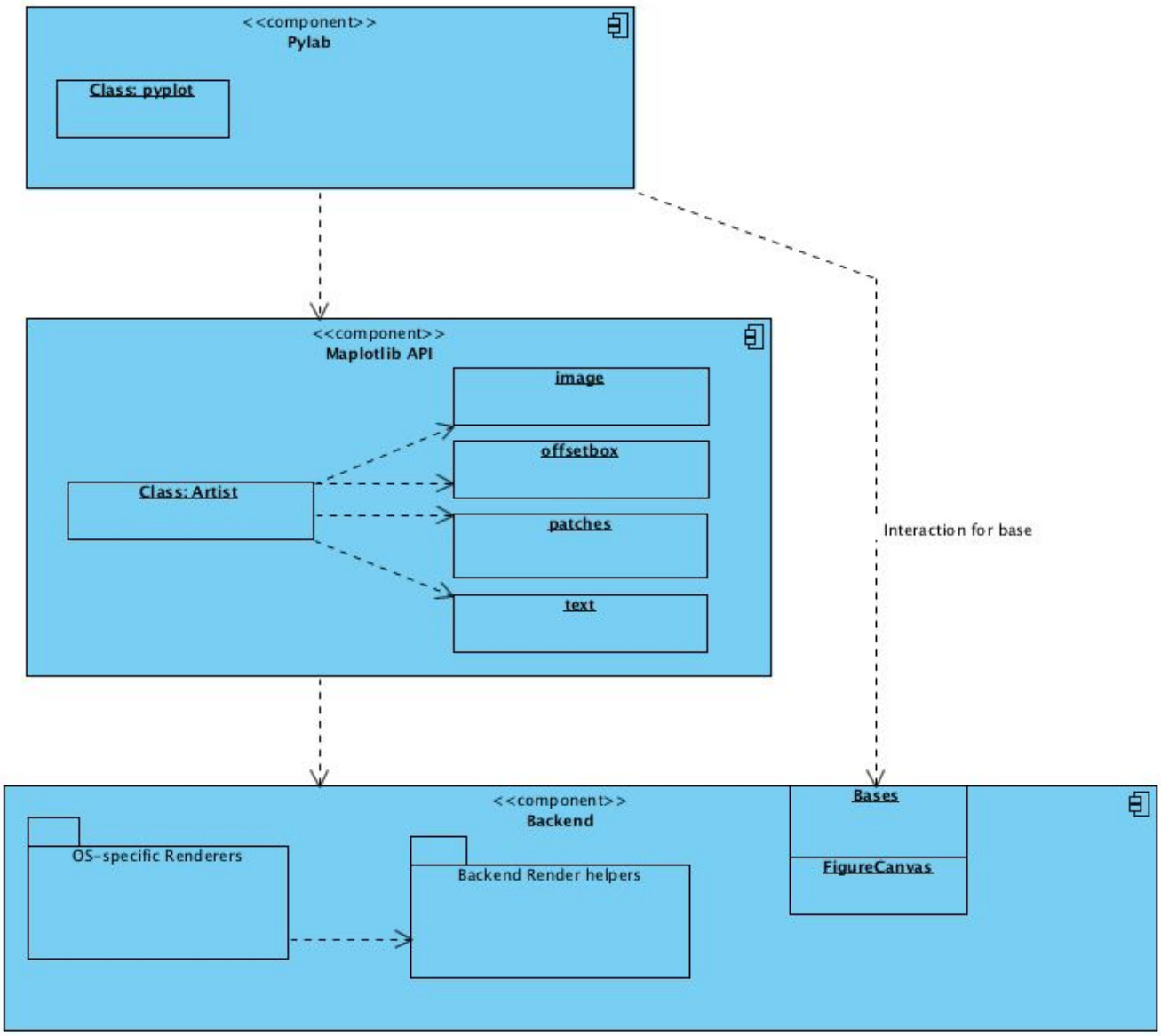
At this point is time it is unclear how we will implement a Validation stage but by following the Agile process will allow us to adapt for Anya's plans. Under this new process the two prescribed team meeting still remain. When we transition from Waterfall to Agile our Thursday meeting will take the role of a code jam in which all team member participate to aiding others in fixes others issues/headaches.

System Architecture

(note: The files pyplot.py and artist.py induced errors in the Visual Paradigm reserve engineering software - therefore we were unable to see the full scope of pyplot's and artist classes within the system architecture)

Components

There are three conceptual parts of matplotlib (src); the Pylab Interface, that manages a MATLAB-like plotting framework, the Matplotlib API, the frontend system that creates and manages figures and its related features, and the backend of the system that renders the figures into a displayable state.



Using these 3 parts, Matplotlib follows a general open 3-layer architecture:

Presentation layer: Pylab interface

This interface interacts with the user: its MATLAB-like frontend representation can be directly accessed with commands like `plot`. In Pylab, the interface uses features from both the Matplotlib API and backend layer to achieve its output. Figure management and setup is done using the Matplotlib API and the device dependent renderers that requires the confirmation of a backend and the graphical visualization bases like `FigureCanvasBase`.

Application Logic Layer: Matplotlib API

This layer has an abstract interface, `Artist` that manages the logic and data management that the software will use in the form of figures of plot information, text styles, line placement and all the instructions and specifications of a plot that the backend will need to visualize data. The Artist class and its related classes modify, create and add specifications to a figures rendered into `FigureCanvas`.

Backend Layer:

This layer is focused on the rendering of the graphical representation of a plot according specifications of the figures within it. The usage of this layer is dependent on the operating system used to execute matplotlib as well as the type of graphics it will be rendered in. There are many classes in the backend layer dedicated to specific OS renders as well as dedicated classes to output the render into different graphic types.

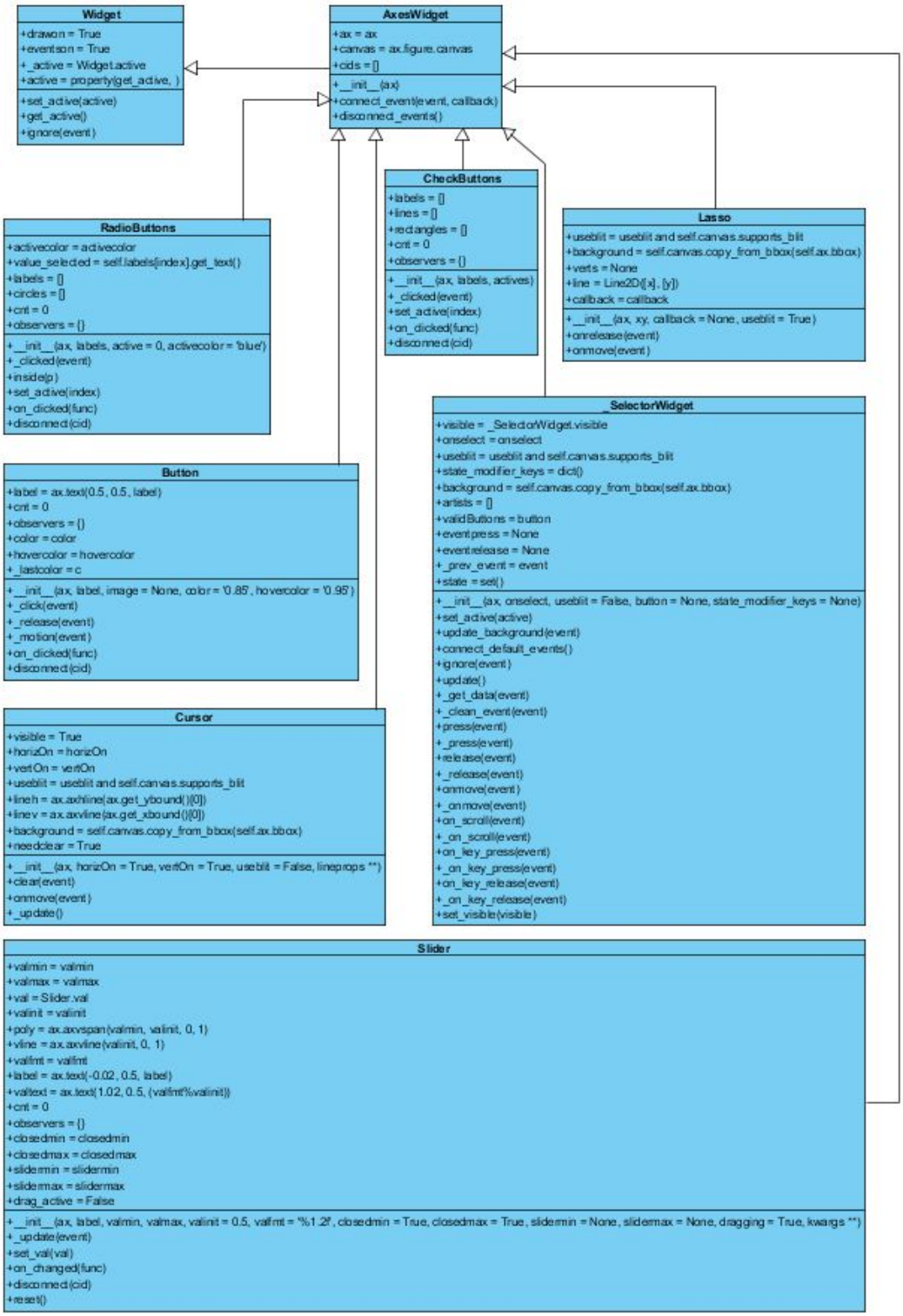
Cohesion & Coupling:

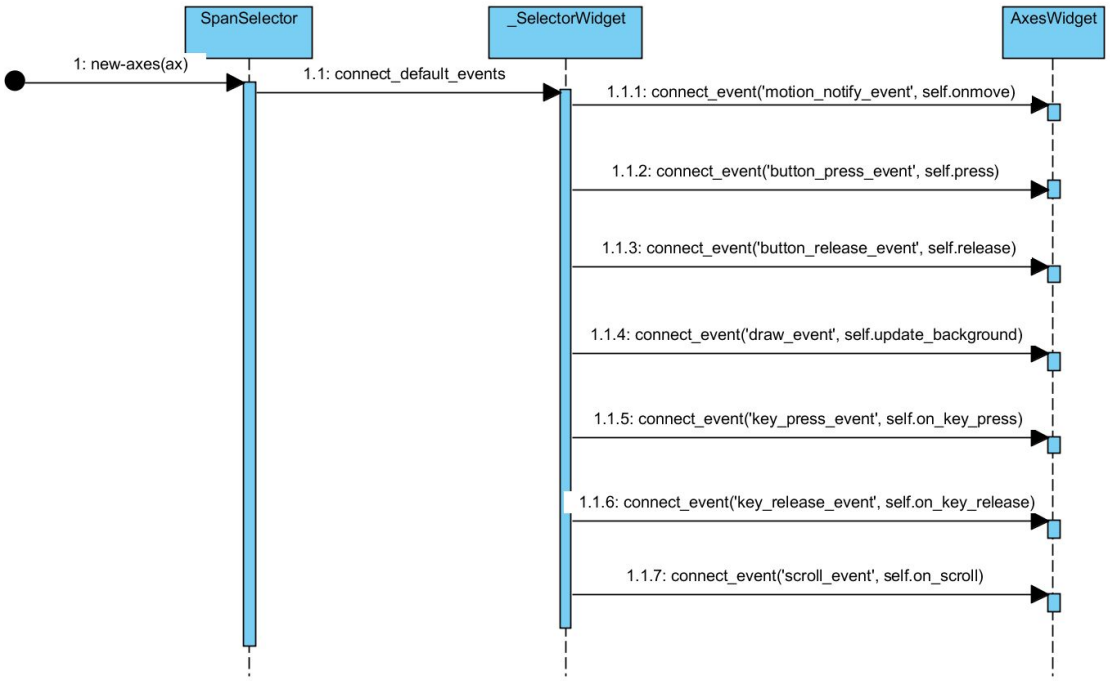
Since matplotlib system architecture is open (based on the fact the presentation layer interacts with both layers below it) there is quite a lot of dependencies. For example, in order for pyplot to carry out its standard `plot(*args, **kwargs)` function, pyplot needs to at least establish a `FigureCanvasBase` from the backend layer and initialize an instance of `Artist` to initialize the basic figure style and plot its arguments before having the figure rendered by the backend. This high dependency on every layer makes the rate of coupling high, and therefore, the structure does not have particularly good cohesion. It could be argued however, that the inclusion of the pyplot and artist interfaces in this structure is an example of good architecture because of the modularity of the interfaces.

Critiques:

The system design would benefit from more packaging of components.

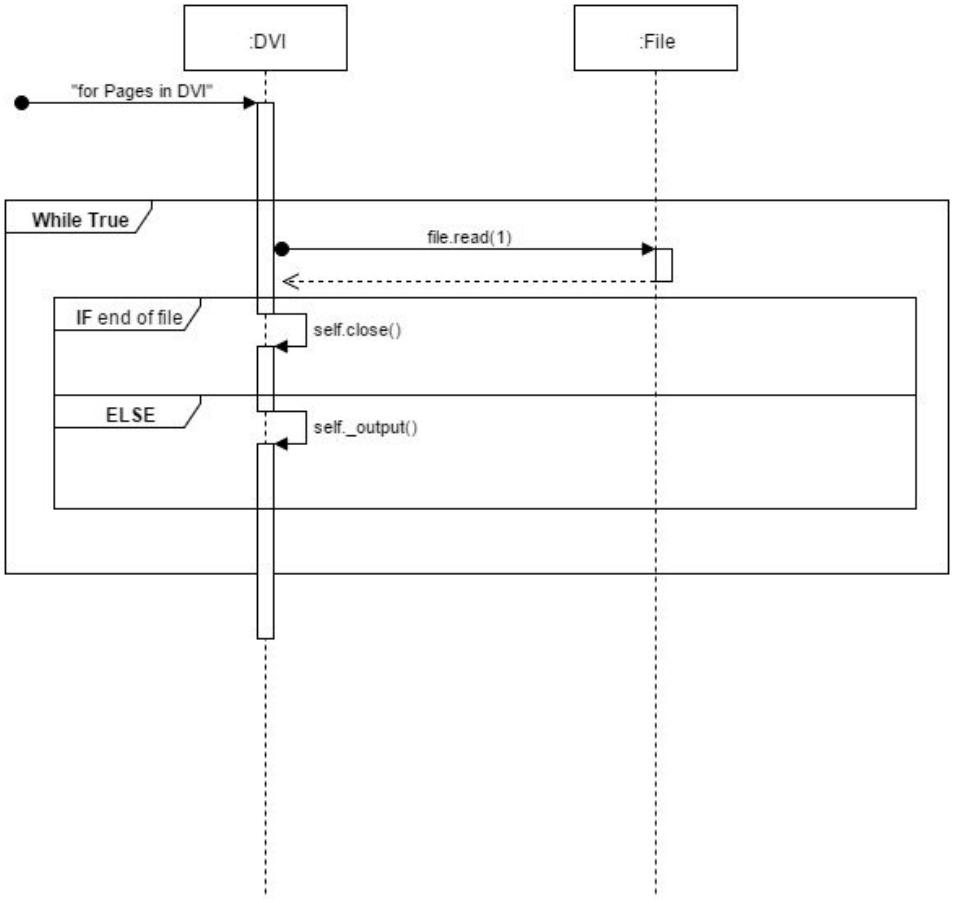
Design Pattern #1 - Observer





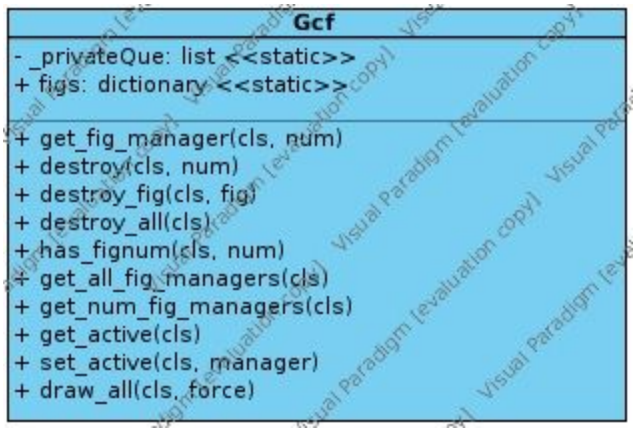
As seen above in the structural and behavioural diagrams the observer design pattern is present with into matplotlib. In matplotlib, observer is used as an event handler. It allows events to be added to a listener(Widget) and if chances occur, such as key press or key release, an appropriate action is taken. Though unlike the observer design pattern that is regularly seen, this implementation does not use an observable class. This leads to the observer doing a bit more work by checking for changes in the events. Though this particular way of implementing the observer design pattern could be improved, by using an observable class, the way it is used (event handler) is a good choice.

Design Pattern #2 - Iterator



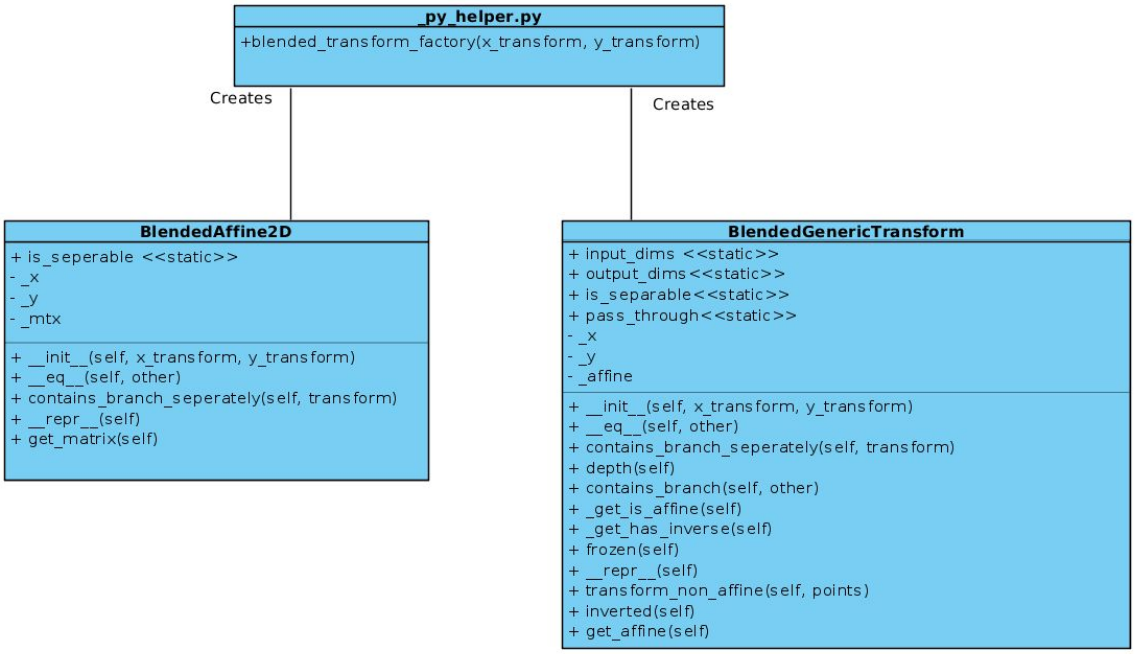
The DVI class serves its roles in matplotlib by translating text in the Tex format files into text readable by Adobe Reader and, with a noticeable lack of a next method, implements the Iterator design in order to iterate over the pages of the given file provided upon initialization of the object. This iterator is able to discard the need for a next method by using the file class' read method as a surrogate thus improving code efficiency. The Vf child design caters to translating virtual font files; requiring a modified read loop dispatch algorithm but still capable of using the same iteration design.

Design Pattern #3 - Singleton



The Gcf class is _pylab_helpers.py is part of a singleton design pattern. This class manages the current figures being used by matplotlib by storing all of them in a dictionary with a number identifier, while also storing the the Queue of active manager figures in a list. This design pattern has an interesting representation in python as a class with no __init__ method. This means that it inherits object's blank constructor and has no instance data. It's data and methods are all static within the class (although they still take self as a parameter), therefore giving it the functionality of a class with only one instance.

Design Pattern #4 - Factory Method



This is a function that is part of transforms.py. Its responsibilities include deciding which transformation to instantiate. Depending on its parameters, it will either return a BlendedAffine2D object or a BlendedGenericTransform. These classes mention that they are only to be instantiated using this factory method. This design pattern is slightly different from the general solution, where a method like this would be part of a factory class but, its function is the same.