Issue 1: “size” ignored if placed before fontproperties #16389
Issue arises when creating any text object. If the “size” kwargs is passed in before the “fontproperties” kwargs, then the size kwargs is ignored. Upon tracing the code, it can be seen that this happens because fontsize (among many others) is a part of the fontproperties attribute which is managed by the FontManager class. When fontproperties is passed in as kwargs, a new copy of the FontManager class is created with default values set and replaces the old fontproperties with the copy. This means that all attributes of fontproperties that were set prior to this change will be overridden by the new object. Since this is an issue in the Text class itself, it is expected that this bug can be reproduced through other instantiations of the Text object rather than just the x and y label as suggested by the bug report.

Estimation: The amount of work that is needed to fix this issue should be minimal. We can accomplish it by first saving any fontproperties attribute before we replace it with a new copy when set_fontproperties is called. Afterwards, we can use the setter functions to set those attributes we saved.

Issue 2: Matplotlib save to pdf not showing hatch marks in bar plot — potential bug? #16052

When adding hatch marks to graphs, the hatches can be seen in the matplotlib display. However when saving to a PDF. It does seem to work while saving to other formats such as PNG. There seems to be a similar issue that was opened a year ago, and it is still open now. Inside a BarContainer, the ax.bar produces the Rectangles and the hatching. There is a test which sets the hatchings after the object had already been created, and it appears to pass. So this may be an issue with the barContainer. Others suggested that depending on different browsers and pdf version (1.4 vs 1.5), some do show the hatches.

Estimation: Based on initial inspection, the issue stem from the pdf backend generation, so at first we thought this would be doable since we can locate the issue down to a single file. However, upon further inspection, the issue could be dependent on the pdf-version and browser type. A work around using inkscape was mentioned, so we decided to drop this issue.

Issue 3: *Request: for non-interactive backends make fig.canvas.draw() force the render #16558*
Currently, for non-interactive backends, specifically the PGF backend, calling the following results in different things happening:

- fig.canvas.draw()
- fig.draw(fig.canvas.get_renderer())

Currently, the first line doesn’t do anything for these backends, while the second one forces a draw. It is useful to force a draw because if you are trying to position the axes and legends (+ other Artists) in your figure using a constrained layout (or other layout), it will only work when a draw takes place. Therefore, if you want to use these layout options for your figures in a PGF backend, you must use the second line and not the first.

This is inconsistent and if one is not aware of this issue it would not be obvious why the layouts were not being calculated, as one would assume the first line will trigger a draw.

Estimation: The first step is to figure out what separates the two lines from each other, the next step is to actually rectify these changes so that the first line does the same thing as the second one (this means not just copying implementation, but actually finding a way to call the same method from within the canvas). Finally, the other non-interactive backends must be checked for this too. It is not clear from the issue whether this affects only the PGF backend, or all non-interactive backends, so they must all be checked as well. Performing the above 3 steps, as well as writing tests, initially reproducing the bug, getting used to the codebase should take 10-15 man hours.

Issue 4: *imshow() should not modify axes aspect if transform != ax.transData. #14118*
In imshow(), the aspect kwarg represents the axes aspect (i.e. the ratio of x-unit to y-unit). If it is set to None (default value), then it would use image.aspect, whose default is 'equal' (i.e. aspect ratio of 1, meaning that pixels will be squares). If it is 'auto', then the aspect will be adjusted to fit the data in the axes (typically will result in pixels not being squares).

Another property in imshow() is **kwargs, which are Artist properties, where the "transform" kwarg could be included. If it is included, the call to the update method (which updates the artist's properties from the dict) will trigger a call to set_transform, which will in turn set the artist transform.

If the transform kwarg is passed in as something other than the default ax.transData, then it would be pointless to set the aspect. It is because the image would not be drawn in data coordinates, and thus having no effect on the ratio of x-unit and y-unit.

Estimation: This issue relates to a lot of other issues and may need to restructure the code in a lot of other areas (as it is an issue with imshow(), which is used everywhere in matplotlib). Two examples include issue #14057 and #14117. There are also a lot of "hidden" dependencies that could cause the whole matplotlib to crash (if this is changed). That was the reason why ImportanceOfBeingErnest removed the "Good first issue label" after one of the team members commented (but later deleted the comment, as ImportanceOfBeingErnest did not answer the question). This issue could take months to solve. Given our limited time, it would be absolutely impossible for us to understand the codebase of the entire matplotlib, reproducing bugs (everywhere in matplotlib where the transform kwarg is passed), and writing tests (i.e. testing the places just mentioned). In conclusion, it is not feasible to attempt solving this issue.

Issue 5: *directory-of-files MovieWriter subclass #7679*
This final issue is a new feature rather than a bug. As this issue was one that was marked with the "Good First Issue" tag on the matplotlib github, we thought this would be a good feature for us to tackle. The new functionality that is desired is a way for matplotlib animations to end up in the animation module if the end user does not have any of the third party modules that

matplotlib uses installed to save the animations to a file. Essentially, we would like to be able to have an option in matplotlib to save animations to a local directory on our machine, as a png file, directly using matplotlib (i.e without the use of any third party modules). This will allow users of matplotlib to keep a folder of pngs containing their animations and figures without having to install a third party module to save them.

Estimation: This feature does not seem too difficult to implement at a first glance, however it does require some research into the structure of the matplotlib code. Specifically, we will need to have a good understanding of the file animation.py, which contains implementations for the MovieWriter and FileMovieWriter classes. The FileMovieWriter class is the one that will let us know how to write these animations to files, so we will need to take time to understand how these classes are implemented, and then we will need to design a solution that will allow us to save these files as pngs to a local directory. We will also need to test that these files are saved as intended in the directory that we want them to be, and make sure that the third party modules still behave as expected with matplotlib, so that users that do have the third party modules installed are not affected. In total, we estimate that this implementation will take at least 10 ideal hours.

Selected Issues:

Issue #1:

This is the first issue that we decided to work on. As described above, the issue here is that the “fontproperties” kwarg can overwrite previous kwargs which were used to update a Text object, since the kwargs are processed sequentially.

The reason that we decided to select this issue, is that it seems like we can issue a simple fix without changing too much of the code. As we are still new to this codebase, it seems like this was a good introductory bug for us to fix, and we could do it without worrying too much about dependencies in the code. As mentioned above, we do not envision this bug taking much time to fix, and we estimate the majority of time will be spent testing rather than coding.

Potential risks with this bug, is that the fix we are proposing will be used all over the code, anytime that a text object is updated, so we will need to make sure that the fix behaves the way we want it to in all situations so that the text is not affected in unexpected ways in other areas of the code.

Software Development Process:

Our work can be found in chrling/matplotlib branch *Issue#16389*

Issue #1 Analysis:

The issue occurs in the method set_fontproperties inside text.py. Based on its implementation, any kwargs passed before the fontproperties kwarg will be overridden (not just the size kwarg). It is because this method creates a copy of all the defaults and then directly replaces all of them back to their defaults afterwards. To fix the issue, we will get all the font properties beforehand (using getters from font_manager.py) to get all properties, and then set all of them back after setting the font properties.

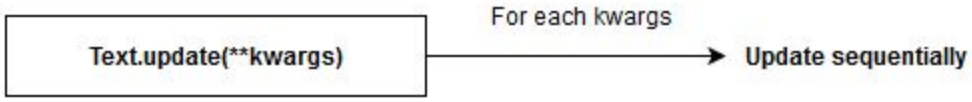
Reproduction of error:

```
data = np.random.randn(100)
plt.hist(data)

[Note: works]
plt.xlabel(“Days”, fontproperties=”Comic Sans MS”, weight=”bold”, size=16)

[Note: weight, size will be set back to default]
plt.xlabel(“Days”, weight=”bold”, size=16, fontproperties=”Comic Sans MS”)

plt.ylabel(“Food Consumption”, fontproperties=”Comic Sans MS”, weight=”bold”, size=16)
plt.show()
```

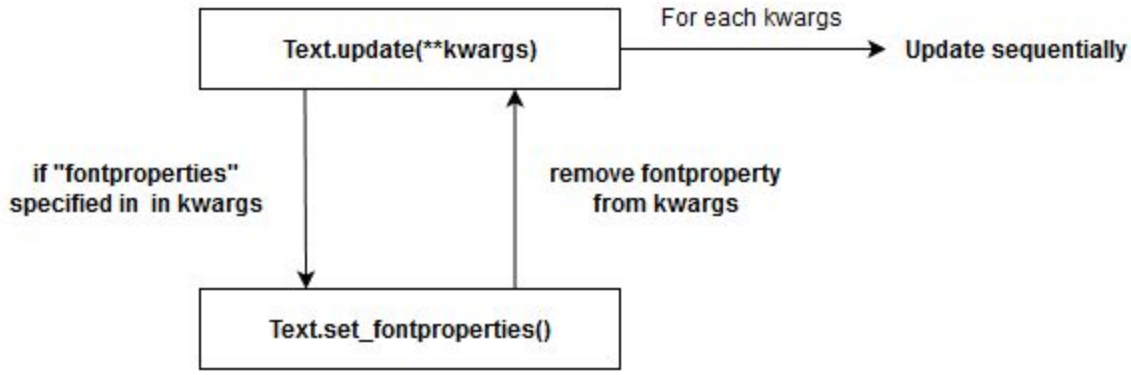
Issue #1 Design:

See **Issue16389FixPlan.pdf**. Since we are following the waterfall process the planning was done priority to writing this document.

Issue #1 Commentary:
The issue arises when **update()** does a sequential update on each of the kwargs that is passed in by the user. And since font size is an attribute of font properties itself, setting font properties after font size (or any other attributes of font properties) will have its values overwritten by default values. The default behaviour of update can be seen below.



To resolve this, we forced **update()** in the **Text** class to always update font properties first (if specified by the user). The changed behaviour can be seen in the diagram below.



Since our change to the update method is small, we expect the impact to the overall code base to be minimal. Issues could arise when a user passes in a **FontProperties** object with their own non-default font properties and also specify font properties in the kwargs. It is not clear in this case what the user wants in this scenario, so we decided that they probably meant to have the font properties in the kwargs take precedence.

The files we have changed are as follows.
To fix the issue:
text.py
For testing:
tests/test_text.py
tests/test_fix16389.py (for acceptance test)

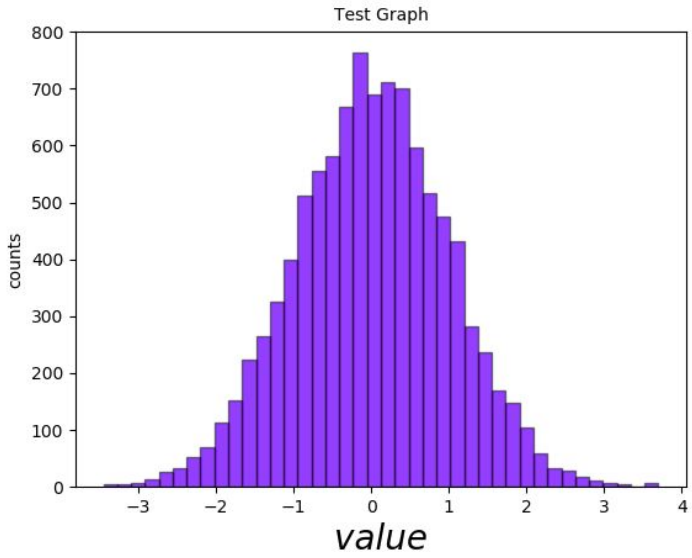
Acceptance Tests:

This assumes one of the following conditions is true:

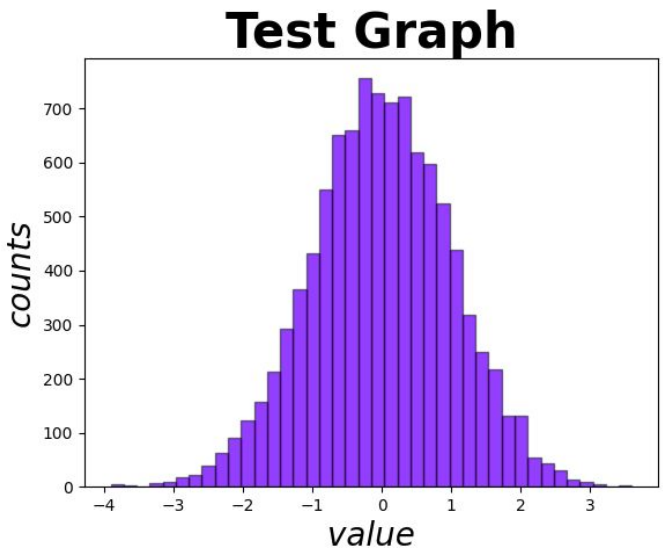
- The issue has been successfully merged into matplotlib, and user has installed the package
- User clones the fork, switches to appropriate branch, and sets up development environment as per:
    - https://matplotlib.org/3.1.3/devel/contributing.html#installing-matplotlib-in-developer-mode

The purpose of these tests is to verify that we can pass the arguments for updating a Text object in any order, including fontproperties, and that the Text object has the properties that we set for it originally, without it being overwritten. This will allow us to verify that the software behaves as intended.

The following files are included in the acceptance test folder for this issue:

- graph_prefix.png
    - This is an image of a plot that was generated before the fix was implemented
    - The x-axis label set fontproperties as the first argument, while the y-axis label and title set fontproperties after all other arguments
    - Notice how the x-label is formatted properly, while the y-label and title are not formatted
    - 

- graph_postfix.png
    - This is an image of a plot that was generated after the fix was implemented
    - The same exact code was used to generate this image and the one above
    - Notice how the y-axis and title are formatted as expected



- test_fix16389.py
  - This file contains multiple tests for Text objects to ensure that the fix applies well in all cases
  - Tests include creation of a Text object, update of a Text object, updating with multiple properties before and after fontproperties, and the original code from the bug report
  - When testing, we are setting properties like size before fontproperties, and after fontproperties, and asserting that they are the same either way

To run the tests, run *python ./test_fix16389.py*
A graph should be produced, and it should resemble the one shown above (graph_postfix.png). This verifies that the code updates the text properly for the plot. Also, upon closure of this Figure, verify that the “All tests passed” message is received. This will verify that all the test cases described above passed as well.

Issue #3:
This issue was one of the issues selected to be worked on. The reasoning behind this decision was the fact that the issue had a decent amount of information with which to start with. Compare this to issue #2 where there is some discussion, but not much regarding the issue. Also, this issue is relatively recent. Issue #5 was posted over 3 years ago, but we do not know why so much time has passed without the issue being resolved (or maybe it was resolved with an update and was not marked as such).

The estimation was already made above, 10-15 man hours should be enough time to resolve this issue (while adhering to the guidelines of our software process).

There are some risks with this issue, as it appears to affect all non-interactive backends. While we want to minimize the amount of things we change, this is even more important as we climb higher up the dependency tree. Therefore, this may potentially involve changing a parent for the interactive backends (ex. **FigureCanvasBase**). Also, while the issue talks about the pgf backend having this problem, we will have to investigate all backends for this issue, and when fixing it for these backends we will have to ensure that we do not do something with one of the backends that we are not supposed to.

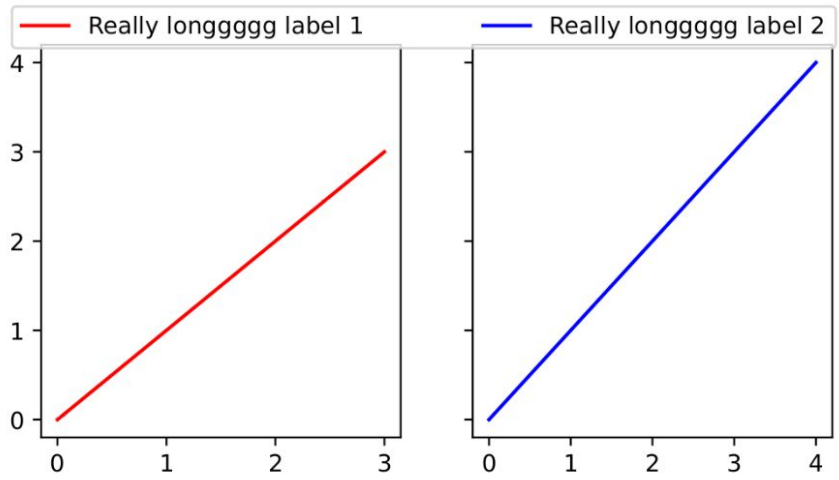
Software Development Process:

Our work can be found in oopsidoodles/matplotlib branch *dev*

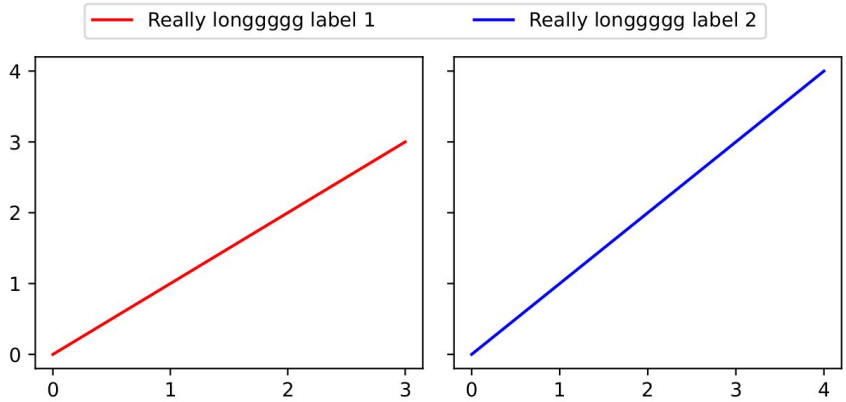
Issue #3 Analysis:
Reading the issue, it is clear that the problem lies in the fact that **canvas.draw()** does not work. Looking at the source code, it appears that **FigureCanvasPgf** does not actually implement the **draw()** method defined in its parent (in **FigureCanvasBase** it is defined but does nothing). Other backends are similar, such as **FigureCanvasPdf** which defines the method but it does not do anything.

Issue #3 Design:
See **Issue16558FixPlan.pdf**. Since we are following the waterfall process the planning was done priority to writing this document.

Issue #3 Commentary:
This issue is ultimately an inconsistency in the api, as can be seen below when **figure.canvas.draw()** is called, the incorrect image below is produced.



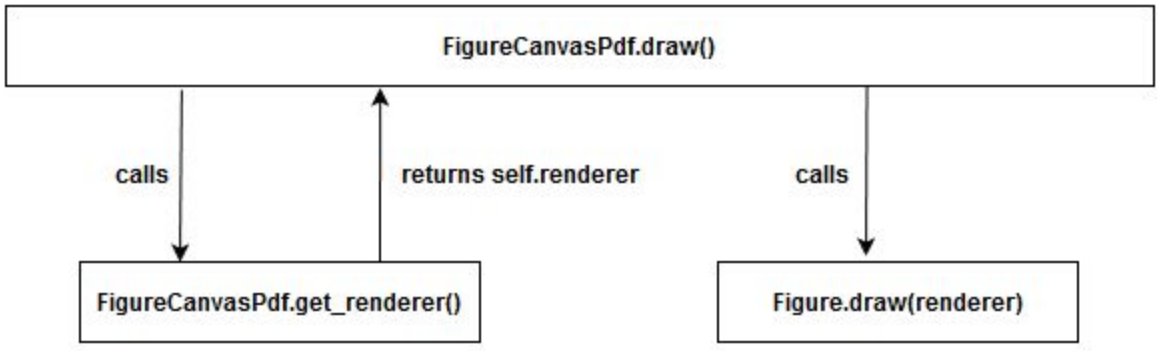
However, when **figure.draw(canvas.get_renderer())** is called, the correct image below is produced.



This happens because **figure.canvas.draw()** is a no-op in the base class and the pgf, pdf, and svg backend does not provide their own **draw()** method and the sequence UML diagram can be seen below:

To solve the issue and to keep the api consistent, we had to implement the **draw()** method of **FigureCanvasSvg**, **FigureCanvasPdf**, and **FigureCanvasPgf** using **self.figure.draw(renderer)**. However, since **figure.draw(renderer)** method requires a renderer to function, we also had to implement the **get_renderer()** method of **FigureCanvasSvg** and **FigureCanvasPdf** (The **get_renderer()** method has already been implemented in **FigureCanvasPgf**. Most of the **FigureCanvas** classes already had to create a renderer at

runtime to render the image. However for our purposes, we could not leverage these existing methods since they have a side effect of creating a file even if the user did not specify to output one. To solve this issue, we passed in a **ByteIO** object to the method used to create a renderer so that data is written to stream instead, thus no files will need to be created. After our change, the sequence UML diagram is now updated to:



Since our change involves implementing a previously unimplemented function that did nothing, we do not expect any side-effects to the existing code base. However, it is possible for it to have created a new bug if other modules have been calling **figure.canvas.draw()** when using a non-interactive backend and expecting it to be a no-op. However, this seems unlikely as we do not see any purpose in doing so.

The files we have changed are as follows.
To fix the issue:
backends/backend_svg.py
backends/backend_pdf.py
backends/backend_pgf.py
For testing:
tests/test_backend_svg.py
tests/test_backend_pdf.py
tests/test_backend_pgf.py

Acceptance Tests:
This assumes one of the following conditions is true:

- The issue has been successfully merged into matplotlib, and user has installed the package
- User clones the fork and sets up development environment as per:
    - https://matplotlib.org/3.1.3/devel/contributing.html#installing-matplotlib-in-developer-mode

There are 6 tests, although all perform the same task just with a different backend and method of drawing.
The test is to generate a plot with a constrained layout specified. The plot generated will look different if it is generated with and without a constrained layout.

The following files are included inside the acceptance tests folder for this issue:

- tests.py
    - Generates plots using both canvas.draw and figure.draw and uses all 3 backends
    - Notice that the plot has constrained layout set
    - *Note*: The generation of the plot is mostly copied from the discussion about the issue on Github, changes were made but that is the original source
- nolayout.png
    - What the plot looks like if constrained layout is not applied
    - This is included because it is not possible to generate this anymore (since it was fixed)
- expected_outcome.png
    - expected outcome for each test, each test should produce an image that is identical to this

The following backends are tested:

- pdf
- svg
- pgf

The following methods of drawing are done:

- Figure.draw
- Canvas.draw

The purpose is to run both methods of drawing on each of the 3 backends, and receive the same, correct result each time.

Run the test file with the following command (use Python3):

```
python ./tests.py
```

Once completed there will be 6 files in the directory of the form ***'test_{backend}_{drawmethod}.png'***:

- {backend} corresponds to the backend used
- {drawmethod} corresponds to the draw method used

Finally, open each of the images and verify that they are all identical and look like expected_outcome.png and do not look like nolayout.png

Since the images are all identical, this means that the Figure.Draw and Canvas.Draw are consistent. Also, since the images have the element properly positioned (unlike nolayout.png where top label clips inside plot), it means the constrained layout was applied. Therefore, the issue has been resolved.

Our Software Development Process:

Our team is following the Waterfall development process for this project, and the concrete evidence supporting this is in the Git commit history. More notes on our Waterfall process are in our team meeting notes and our plans on our git repo. As explained in our first deliverable, we don't really have a requirements phase for these bug fixes, as most of the requirements are already outlined in the matplotlib Github, and we don't expect these requirements to change much, since they are bugs. We were also excluding the maintenance phase, as the project is open source and will be maintained by the product owners. Therefore, our process started with selecting bugs, then proceeding immediately to the design phase.

Design/Planning Phase

The Design Phase for us was the most critical step of our process. This phase first allowed us to take a closer look at the bugs which we decided to fix, and get a better idea of how much work was needed to solve each of the 2 bugs that was selected. Once the team had a good idea of how much work was required for each bug, we came to the conclusion that we did not require a whole team's effort to fix any of the individual bugs, so we decided to split into smaller groups within ourselves. We initially had 3 team members work on Issue #1, and 2 team members work on Issue #3. Once we split into individual teams, each team dived into the actual code relating to the bug, so that we could identify how exactly to fix the issue and identify any dependencies that might need to be addressed while we fix the bug. This was critical for us as we are still new to this codebase, so having time to closely analyze what we had to do before actually implementing anything was very good, which is a huge upside of the Waterfall process. The outputs of the Design phase can be seen in the Issue #1 Design and Issue #3 Design written above, which point to a location in our git repo which contain the plans for our implementation.

Implementation Phase

Once we finished the designs mentioned above, we proceeded to the implementation of the plans that we came up with in the previous phase. As these were small bug fixes for the most part, we didn't need to have multiple team members writing the actual code. We could just have one team member focus on writing the code, which gave the other team members time for things like documentation and code reviews. Reviewing the code of the team members who actually implemented the code ensured that we got the highest quality code possible, and making sure we are following matplotlib's guidelines for coding. The result of this phase was clean code that was ready to be tested, and the output of this can be seen in our forked repos, as mentioned above.

Verification Phase

The testing phase occurred after the implementation of the code mentioned above. For testing, the first thing that was done was making sure the test case that was provided in the actual bug report was fixed. After making sure that that was fixed, we could move on to more

general cases that made sure that our code would work in all possible scenarios. These tests were done in the form of acceptance tests and are outlined in the “Acceptance Tests” section in this document.

How Waterfall Helped Us

This deliverable was the team’s first time working following a Waterfall process, and we found that it was very helpful, specifically for this kind of project. The major way we found that Waterfall helped us be more efficient is by limiting the amount of in-person interaction that was necessary to further our development. Our initial design phase was one meeting where we could take the time we needed to really explore the issues and make sure that we have everything ironed out before we actually proceeded to development. An Agile process could have potentially been worse here as the planning is less involved than a process like Waterfall. So it would have been a very likely scenario that we could have done a short planning phase, and then tried to work on development, only to realize that we missed something and then have to meet with the team again to identify that issue which is currently blocking our development. This is especially an issue since we are a group of students, and we all have different schedules, so it is quite difficult to find a time where we are all free for a meeting. This is one way that Waterfall helped us, by limiting the amount of times that we had to meet as a group to identify issues, which helped us flow smoothly as a team. The Waterfall process also helped our efficiency by having an accurate plan of what needed to be done before actually implementing any code. By the time the design phase was done, it was extremely clear what exactly needed to be changed in the code and where. This made the actual implementation very simple, the implementation phase was actually the shortest phase for us for this deliverable, since we had a clear plan of what we needed to do. This is one clear advantage of Waterfall, by having a concrete plan of implementation, it makes actual coding very simple, so we can spend more time on other things (eg. testing) to make sure that everything works properly.