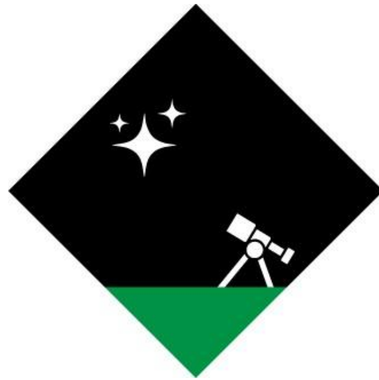

Operation Dark Sky

Requirements Specification

April 9th, 2022

Team: Justin Ceccarelli, Jordan Tatum, and Luke Thompson

Sponsored by: Jim Clark, Peter Kurtz, and Henrique Schmitt

Mentored by: Anirban Chetia and Tomos Prys-Jones

Accepted as baseline requirements for the project

By the client:__________________________ By the team:__________________________

Table of Contents:

1 - Introduction: 2

2 - Problem Statement: 3

3 - Solution Vision: 4

4 - Project Requirements: 5

4.1 - Functional Requirements: 5

- 1 - Develop user interface 5
- 2 - Present astronomers with star data 8
- 3 - Fix previous issues 11

4.2 - Performance Requirements: 11

- 1 - Detailed and rapid data retrieval 11
- 2 - Creating an easy to use interface 12

4.3 - Environmental Requirements: 12

5 - Potential Risks: 14

- 1 - Risk Mitigation 14
- 2 - Risk Overview 16

6 - Project Plan: 16

7 - Conclusion 18

1 - Introduction:

The Navy Precision Optical Interferometer (NPOI) is an astronomical observatory operated by a partnership between the Lowell Observatory and the U.S. Naval Research Laboratory (NRL). The NPOI conducts research on unique astronomical phenomena like binary stars as well as carrying out a range of practical functions that support the U.S. Navy's navigational and communication capabilities, such as monitoring the satellites used for GPS navigation. The Navy still relies on star charts provided by the NPOI to serve as a form of navigation that is entirely independent of technology or political influence.

The NPOI is primarily dedicated to astronomical research with the aim of improving our understanding of the stars and planets in our galaxy. The NPOI is an interferometer, which means that it uses a network of mirrors to collect and redirect starlight from up to six locations before merging the light together to create an interference pattern. The NPOI is the world's largest baseline interferometer, making it especially well suited to studying binary stars, or pairs of stars that orbit each other. Like many observatories, the NPOI can only operate at night, during fair weather, and beneath a cloudless sky. Since these circumstances are out of the observatory's control, they have to take advantage of every opportunity to collect data. This means planning each night of observation well in advance to make sure that the equipment is positioned and then continuously re-positioned to capture the appropriate stars at precise times over the course of an evening. Properly planning an event of this scale poses a significant challenge, especially considering the constant movement of the stars in the sky relative to the earth. The astronomy team at the NPOI initially needed to manually perform an exhausting amount of technical work to determine the optimal timing and ordering of their observations. This planning process requires an experienced professional to dedicate a significant amount of time every night the NPOI observes the stars.

In order to handle this organizational challenge, the NPOI created a software application they called "obsprep", short for observation preparation. Obsprep consisted of two complementary components: the backend and the frontend. The backend contained a catalog of stars, including their location in the sky, as well as a library of astronomical functions called the Naval Observatory Vector Astrometry Subroutines (NOVAS). The frontend of obsprep was a Graphical User Interface (GUI) that allowed a user to access the data in the star catalog and to use the functions in NOVAS to perform calculations on that data. Together, obsprep allowed astronomers to create a list of the stars they wish to observe before calculating the optimal time and order to observe those stars to maximize the observatory's uptime.

When obsprep was originally written in the late 90s, it met the NPOI's needs for planning and organizing the observatory's nightly work. Unfortunately, in the decades since then the software's frontend has become outdated to the point where it requires frequent maintenance simply to avoid crashes and unplanned downtime. The GUI is also an outmoded relic that requires a great deal of experience and training both to operate and to install the software on a new machine. The NPOI has estimated that a night of observations missed costs the observatory an estimated $12,000. Hence, they have decided to retire the original implementation, and has hired team Dark Sky to create a new frontend application for the

obsprep software. The NPOI envisions a solution that can be easily installed on a user's work machine regardless of operating system, that allows the user to select multiple desired stars for a specific night, as well as generate and save graphs of the stars' progress through the sky to better visualize their location as time passes.

In this document we will consider the problem in greater detail, as described to us by our client, as well providing greater detail on our solution vision. . Following this, we create a list of essential functional, performance and environmental requirements. Once we have clearly established those, we will consider the potential risks facing our project, and create a project plan complete with milestones that we will aim for as we build the software.

2 - Problem Statement:

This section will describe the problems facing our client that our project will address. The researchers at the NPOI have been stymied by flaws in the installation and operation of the current form of obsprep, particularly its graphical user interface. The researchers require access to obsprep to plan their nightly observations, analyze the quality of their observations, and interpret the results. The backend through utilities like NOVAS and Simbad provide computational support on subjects such as the observable angle of a star or the current corrected Julian date through NOVAS, or identifying and describing information about a given star through simbad. However access to this information must come through the frontend. The inefficiencies and flaws in the current GUI make it difficult for researchers at the NPOI to share their work with their partners and collaborators at the Navy Research Lab (NRL), US Naval Observatory (USNO), and Lowell Observatory.

The current edition of the GUI for obsprep is difficult and time consuming to install, has too many environmental requirements and installation steps for many members of its intended user base. This creates a drain on NPOI time and resources that are being devoted to making the old edition of their GUI work, and creates frustration and delays for the observers. The software required constant maintenance, and suffered frequently from unplanned downtime. Until recently predicting star locations over a night of observation required researchers to perform a number of calculations by hand, but obsprep has been updated to provide this information. The dissemination of these star plans is still difficult due to the current design of obsprep. The following is a more comprehensive list of problems caused by deficiencies in the current obsprep GUI.

- Lack of Access to upcoming predicted observation angles
- Inability to save reports for review and distribution
- Excess downtime on observation nights when issues arise
- More time spent to onboard a new employee, or even simply spent when an employee purchases a new computer
- Difficulty planning nights
- Software works poorly on new machines, or even not at all

3 - Solution Vision:

This section will describe our vision for a solution that will address the problems described above. Team Dark Sky will create a new graphical user interface that will serve as the front end for the obsprep program. This new GUI will replace the current front end with a product that is easier to install, use, and maintain. Decreasing the time the users are forced to invest in learning and operating the software will increase the time they have available for more important tasks, like conducting research. Our product will be consistent with the latest edition of python and will be easier to configure for installation on new systems. The new front end will be cross-platform and fully functional on the Windows, Linux, and macOS operating systems.

A new GUI will also allow astronomers at the NPOI to more effectively record and distribute their work. This project will reduce operational down time, time spent onboarding new employees, and time spent updating workstations. It will improve troubleshooting capacity and communication between research partners. The following is a more comprehensive list of the features of our solution.

- Codebase that uses an up-to-date language and meets modern code standards
- Easy to use graphical user interface - an intuitive and functional design
- Access reports using local files - self declaring file storage for more intuitive access
- Easily installed on a new system - reduce the number steps required of the end user
- Support cross platform use - account for differences in operating system architecture and commands, and use cross platform supported functions.
- Access a library of Astronomical Functions for observation planning - maintain accurate and reliable access to information on astronomical bodies and stellar calculations.

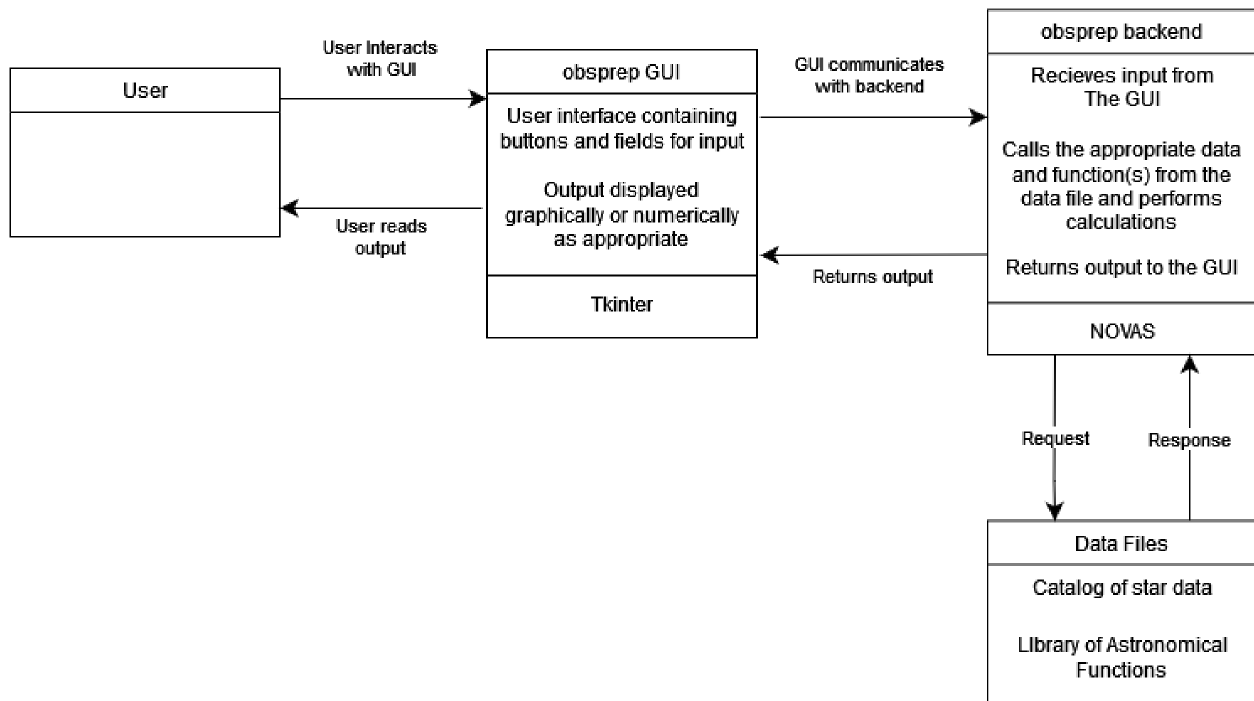

Figure 3: Architecture

4 - Project Requirements:

In the following sections we will discuss the functional, performance, and environmental requirements our project must meet.

4.1 - Functional Requirements:

Functional requirements are the technical capacities that a program would be expected to provide to fulfill our solution vision. More than anything else, functional requirements define what a program is and what it does in quantitative terms. We have divided our functional requirements into higher level requirements which are then decomposed into a series of necessary capacities needed to fulfill that requirement.

1 - Develop user interface

One of our major domain requirements is to develop the user interface for the obsprep software. We have managed to break this requirement down into three functional requirements. These functional requirements are to call the astronomical functions, implement our tkinter framework and to design an intuitive interface. Through implementing these requirements we will be able to successfully develop our user interface.

1.1 - Design an intuitive user interface

When creating a user interface one of the most important things to consider is to make it as intuitive as possible. One of our main goals with the application is to reduce operational downtime and increase observation time, with an interface that will be intuitive to learn. Some aspects that help create an intuitive interface are easy maneuverability and creating a uniform area classification. Through implementing these two functionalities we will be able to develop an intuitive interface.

1.1.1 - Easy maneuverability

Graphical interfaces can become complicated and hard to maneuver when developers implement too many widgets within a single page. This is why our plan is to create a specific page for each function we need to implement. For example if a user is looking to have the star data displayed to them they will navigate to the star data page. As well we will have clear labels for each page so that they know exactly which page does what function. Another feature we will add to help make it maneuverable is to have sorting buttons for information. This will include ascending, descending and by observation stations. We also want the user to be able to access any functionality they need within three clicks or less. This will help keep the interface simple as well as making it easy to maneuver through. With our project layout plan we believe that this will help make our interface intuitive for all users.

1.1.2 - Establish uniform area classifications

Another way we can help make the interface as intuitive as possible is to create a uniform area classifications for each page on the application. This means that we want the

same general layout per page. Some classifications we will need to consider are the screen, tab, field area, pop-up / overlay, and widgets. The first consideration is how big the screen size is going to need to cover. Through the use of the GUI framework the application will be reactive to the user's screen size. The next step is to pack the screen in a non cluttered way. This is important because we don't want too much white space on our program but we also don't want too many objects on the screen at once. We are going to have a max of four objects on the screen at once so that we can have equal spacing between them. We will be putting the user input boxes in the top half of the screen and then have the output received under it with a few line breaks in between so the output can be easily readable. In the situation where the user is inputting a star to receive information and needs a graph we will have the output field below the input and have the graph displayed to the right side of the input field. This way they can see the text data representation and the graph at the same time without overcrowding the screen. Finally we will create a formatted header for each page that will hold all the different tabs for each page, as well as the application name and other information such as date and time. Through implementing this format on each page the user will be able to understand each page and functionality swiftly.

1.2 - Implement GUI framework

Through the use of a framework we will be able to develop and design our interface. With the use of a library there are different parts that we will need to implement to be able to optimize our interface. These parts include user input box, submit/delete/reset buttons, embedding graphs within the interface and list boxes. We will need to create classes for specific aspects of the GUI, deploy widgets, implement tooltips and help options and implement units. With these requirements in mind we will be able to implement and utilize our framework to the best of its ability.

1.2.1 - Create classes

With tkinter, our framework of choice, there are few options for setting up specific layouts for necessary tasks. Some of these layouts include menu buttons, frames, and lists. This is why it is crucial to create classes for each layout option. If we did not create classes for specific tasks we would not be able to design each one specifically to the task at hand. Through creating classes we will be able to detail each layout's look and functionality oriented to the task.

For instance we will need to create a class to embed our graph display within our frame layout.

This will allow us to display a graph with the optimal observation time to the observers. We will be able to initialize the graph with the correct layout, date and containing frame. This will not only help us create more detail oriented graphs but also give the observers the option to adjust the size of the graph. If we did not create a class for our graphs they would not be considered a framework object and would be unable to be implemented and displayed within our interface.

Another class we will need to create is a class for data entry that will also be within our frame layout. With this class we will initialize it with the date, check if the date is in the current

date or standardized one and implement a given command sent to it. With this class this will help us implement a frame layout that is different from the previous frame but allows for user input and saves the current date for the observers. This will be important because they will need to date their findings so they can share when their findings were discovered.

One of the other classes we will need to implement for our interface is a multiple list box that is also part of the frame layout. This list box will display data to the observers based on their selection of star and observation station. It will also be able to save the retrieved data for the observers. With this class they will be able to mutate the data to their specific needs such as ascending order, reverse order, delete and insert entries. Through this class the observers will be able to have full access to any data and be able to navigate through the list with ease. This class is one of the most important and will need to be very precise to our clients needs.

The final class implementation that will need to be done is a pick list that is a menu button layout. This class will be able to add sub lists to the pick list as well as create label and data entries to the list. The observers will then be able to choose what the menu label will be as well as any sub menu labels they will need. They will also have access to setting a selected value and retrieving a selected value as well. This class expands the capabilities of the default menu button offered by tkinter and allows us to make the users job easier.

All of these classes are important to expanding the functionality of the main layouts tkinter offers. Taking a class approach give us access to implement these specific needs into the framework and allows us to have a more functional interface for our clients.

1.2.2 - Deploy Widgets

Widgets are the main part of user interfaces, with tkinter they offer basic widgets that we will need to implement within our interface. Widgets are the main part of an interface; objects the ones we will be using are buttons, input boxes, list box, menu buttons, scrollbar, menu and labels. Without the use of widgets it would be impossible to interact with the interface itself. While implementing our widgets it is also important to make sure that we are choosing the right widget for a job. For instance if we need a user input we will need to use an input box widget such as a text box and a button to be able to submit the entry. Some of our widgets will be more advanced then the default ones included in the framework. This is because as discussed in 1.2.1 we will have classes that give each widget the ability to have more functionality then what is offered. Through deploying widgets we are giving the interface the ability to interact with the user, accept input and make the application maneuverable. .

1.2.3 - Implement tooltips and help options

One of the tasks we have received from our clients is to implement tooltips into the interface. This will help the users of the program be able to get a grasp on exactly what each tool's functionality is. Our plan to create a tooltips is to create a tool tips class for a specific widget. The tooltips icon will be located right next to the widget. This will allow the user to scroll over the tooltips icon and get a text box that describes how to use the specific function. With this

being implemented this helps us make our interface user friendly answering any questions the user might stumble upon while using aspects of our interface.

Another part we want to implement is a help option. This will be a more broad help item that will give you navigation help. In the current iteration of obsprep their help option is a long 233block of text that breaks up into different sections based on what you need help with. We are going to make our help options more readable with smaller text blocks that are more accurate to the exact functionality you need. We will be adding a help button to each page instead of one generic help button about the whole application. This will help keep the help text readable and make it more oriented to the specific page it's located on.

1.2.4 - Implement units

Implementing units is an obvious requirement to include in a computational application but learning from the past iteration of obsprep it is very necessary. We plan to include units on all necessary calculations. This will be an easy implementation by just appending the unit to the returned value. We will also offer different units and a conversion between them as necessary.

1.3 - Call astronomical functions

The interface will be taking user input and retrieving specific information based on their requests. Through our interface the naval research team will be able to access a broad range of observation information such as zenith angle, optimal time for observation, and location in the sky. This will help them simplify the process of conducting their nightly observations. The NPOI has provided us with access to a file containing all the astronomical functions we will need to implement. This will help us because from our front end we will just have to call the specific function depending on the request from the user. For example if the observers were trying to get the modified julian date they would put in the current date in the input field and we would then take that input and pass it into the julian date to modified julian date function as the parameter and be able to get the updated date as the response. Then if they wanted to find the observed place of a solar system body they would put in the body their looking for, the modified julian date, the site their at, and if they need to add refraction or not into the input box and then we would call the observed planet function and return the observed place of the given solar system body. Another function we will have to use is the apparent star function. We will be calling this function when the user is looking to find the apparent place of a star at a given date. The user will input the modified julian date and the star they want to see and it will then return the star's apparent place. We also have functions which are helper functions that ensure the input is formatted correctly so we get the correct output from a function. Some of these are the body function which retrieves the exact format and name of a given solar body, equation of equinox function, npoi function which gives us the location of the NPOI site, and air mass to zenith degree function which returns the zenith distance in degrees for a specified air mass. Through calling these helper functions before accessing the main ones we guarantee that our parameters are correctly formatted for it. Through accessing these functions we will make our interface able to perform all necessary calculations and tasks needed by the observers.

2 - Present astronomers with star data

One of the major goals of our project is to give the astronomers at the NPOI access to the stellar data they need to plan their observations of celestial bodies. To this end, our product will maintain a library of astronomical functions, generate graphical representations of this data, and manage the information already requested by the astronomers such that it can be recalled and distributed.

The astronomical calculations themselves and the database of stellar bodies are maintained by organizations outside of the NPOI. Simbad is an online repository of stellar bodies that the NPOI uses to access current and extremely precise star information, and NOVAS is a library of astronomical functions. Access to these services are maintained primarily by two programs named astro.py and catalogue.py. We will not be required to perform a re-write of these functions to determine when, where, and how they are implemented in obsprep.

2.1 - Maintain a library of astronomical functions

Prior to displaying and saving the information requested by the astronomers, our product will calculate the needed information based on data from Novas and Simbad. Novas is a technology developed by the Naval Research Lab (NRL) to provide support for astronomical calculations, and Simbad is a database of stellar bodies accessible through the internet. These capabilities will be supported through two files named astro.py and catalogue.py. Both of these files already exist in obsprep and much of their particular calculations will be maintained, save that they will be updated to the latest edition of python and adapted for use with our obsprep frontend.

2.1.1 - Access to astro.py

Astro.py is the file that provides the backbone of the astronomical calculations needed by obsprep. It provides access to the Novas library to obsprep through a series of methods, as well as accounting for the variables that are dependent on time and location. In other words, it adapts Novas for use at the coordinates and time of the NPOI in Flagstaff. Astro.py also provides a few other functions not based on Novas to obsprep and the astronomers.

2.1.2 - Access to catalogue.py

Simbad is the primary source of data on astrological bodies used by obsprep to help the observers plan their observations. Simbad is an online repository of stellar bodies that is queried by the catalogue.py file, the results of these queries are then passed on to obsprep and the observers. Catalogue.py is a file that provides a series of methods with which to access simbad to obsprep. Catalogue.py and astro.py are both fundamentally helper files, they provide a series of methods to obsprep for use by the astronomers.

2.2 - Display star data graphically

Obsprep must provide a graphical representation of the visible times and angles of observation for stellar bodies at the request of the observers at the NPOI. To this end our

product must have the capability to generate graphs of the times when observations will be possible from any given observation station on site and when they won't be. Matplotlib is the tool we have chosen to support this functionality. The observers at the NPOI will be able to take the stellar bodies retrieved from Simbad, apply the calculations of Novas to them, and then receive a graph of what times that stellar body will be visible from each individual station at the NPOI. The stations at the NPOI each have a separate series of angles they can observe on a given night, and the user will be able to select which stations they want to use and receive a graph detailing when a stellar body is visible from those stations.

This functionality is necessary for the observers at the NPOI to plan what bodies they will be observing on what nights, and aids in their ability to plan work flow around a series of operations at the NPOI including not just when astrometric data is to be collected and analyzed, but also when observation stations need to be prepared for use and moved.

Matplotlib provides functionality to store its graph as simple text that can then be reinterpreted into a graph again by a matplotlib parser. This is preferable both for ease of storage and for distribution. This makes the files simple to create, store, and share with others. For the convenience, our client also requests that graphs be storable as an image file, png, pdf, or jpeg. We will be providing them with the ability to save their graphs as png files, these files will not be integrated into a matplotlib frame as the text files are, but they will be more conveniently viewed by the user from their file explorer or equivalent.

2.2.1 - Embed matplotlib into the tkinter tool

Displaying these graphs will involve developing an embed frame for them. TKinter has the capacity to place a series of images on the screen through use of what it calls frames. These frames can hold a myriad of information that we discuss in section 1.2.1 and often makeup the general layout of a GUI, we will be using a frame for the graphical display of information both because it allows to dynamically place these images for the user, but also because it facilitates the conversion of the graph from text to image and vice versa.

2.3 - File management

The observers at the NPOI expect to be able to generate their observational plans and be able to refer to them later. To this end we intend to facilitate file generation and retrieval for the user. The current version of obsprep requires the user to boot obsprep from the command line with reference to the file they wish to review. We plan to resolve this inconvenience by creating and accessing files through the use of our GUI. The user will be able to open files after opening obsprep. This will eliminate the need for repeated openings and closings of obsprep to access different files, and make the whole process much more streamlined.

2.3.1 - Save data

Obsprep will store the graphical data for a specific observation as a formatted text file that Matplotlib and the methods we design can parse into a graphical display. These files will be generated in a folder in the directory of obsprep that is appropriately labeled for its function and the user will be able to directly access these files through that folder with their OS's file explorer.

This will enable the astronomers to pull their files for distribution more easily and simplify the organization of these files.

2.3.2 - Access saved data

Obsprep will be capable of accessing files stored in its appropriate directory, in order to open a file with obsprep the user will simply need to have the file in that folder when opening obsprep. The user will then be able to access any file in that folder and view it graphically. The files in this folder will all be selectable for use from within obsprep. This will allow for more dynamic workflow from the observer as they do not need to commit to a single file before opening obsprep.

3 - Fix previous issues

Our product is intended to replace the version of obsprep that the NPOI is currently using. This previous version has been in use for almost three decades, and has received a number of updates and modifications in that time. A core requirement for this project is that we consider the significant quantity of user feedback generated over the years, and create a product that accommodates our clients particular needs for convenience and ease-of-use.

3.1 - Update installation process

A major issue with the original obsprep program was the difficulty involved in simply installing the program on a new workstation. The program was designed specifically for windows, and making it run on any other system was a complicated process that required an IT professional to conduct. Our client wants the program to not only be cross-platform, but to be simple enough to install that any employee can set it up on their own machine without assistance.

3.1.1 - Implement installer script

Our team decided that the most convenient way to make our program easy to install was to create a script to handle the process. The program will include this script, which can be run to carry out the entire installation process automatically.

4.2 - Performance Requirements:

This section will explain the quantitative performance requirements associated with the functional requirements described above. What are performance requirements?

1 - Detailed and rapid data retrieval

We have two performance requirements for calculating and presenting data to the user. First, every function presented to the user must have a clear and readily available description. Second, the software must communicate with the backend very rapidly. We decided that data should be retrieved in less than a second.

1.1 - Every Function has a commented description

We will ensure that every function performed by the software has a clear and correct description of exactly what it does. Many of these functions are quite complex, so we will coordinate with our client to make sure that we have an accurate description for each one readily available within the program itself.

1.2 - Retrieve data in less than five seconds

We wanted to set a baseline for the speed and technical performance of the software. Corresponding with the backend in less than 5 seconds is a realistic goal, and will make user's much less likely to become frustrated or impatient with the software. A user will not wait more than 5 seconds to receive an output following an input.

2 - Creating an easy to use interface

For performance it is very important that our GUI is able to give the user all necessary performance functions that they need. We plan for our interface to be easily able to be learned and to have any kind of help needed within a few clicks. This will help create an optimal performing interface.

2.1 - Requires less than a half hour of training

Within the development of our project we want to create it so the user only needs very little training. Our goal is to have the users only need less than a half hour of training. We will be able to do this through implementing all of our user interface functional requirements. Testing the rate of learning will also be easy because we will give it to our clients and give them a very thorough walkthrough of how to use it. This will involve invoking specific functions that they will need such as inputting and retrieving data as well as getting a graphical representation of the stars data. Then to test it on the user we will deploy a workshop that we will be presenting to them and give them a 25 minute tutorial. Afterwards we will test them by asking them to retrieve and display when a specific star will be available for observation. We will have achieved our goal if 90% of the users are able to complete this test after the 25 minute training they have been given.

2.2 - Help text/tooltip reachable within 3 clicks

Another helpful performance requirement is to have any help text or tooltips that they'll need within 3 clicks. This will help keep our interface simple and find any help they may need easily. Testing this specific requirement will be very easy by being on a page and trying to access the help text without going over three clicks. If we cannot reach the help text then we will simply move it to the place that is easier to reach. Through implementing this performance requirement our application will produce better results for our clients.

4.3 - Environmental Requirements:

In this section we will explore the environmental requirements of our product. Environmental requirements are what is necessary on behalf of the environment for our

program to be able to run, as well as what is necessary of our program to conform to the desired environment.

1 - Environmental Requirements placed on our product

Our product is required to function cross platform on home and work computer desktops. We anticipate use on Linux, Windows, and macOS systems, this means that we do not need to make specific hardware requirements for our program as our product is intended to be used in a resource bountiful and readily configurable environment such as a workstation computer.

The Navy has developed the obsrep C based backend both with a python wrapper around it and as bare C code. The development of this wrapper is a step the navy has taken to improve the implementation of Obsprep across their research groups, the NPOI included. To this end it is expected of our program to provide access to the Navy's latest release of the obsprep backend, which has a python wrapper around it. This means that it is expected that our product will be interacting with the backend through python.

Most of the challenges of cross platform development are mitigated by working on the major operating systems as opposed to other platforms such as mobile platforms, and by using the latest edition of python. A python program written and compiled on a mac computer will execute on a windows machine and vice versa, the same goes for Linux. While other cross platform projects will require separate code bases for each platform ours will not. There is an exception to this however in the portions of our program that reference the OS module of python. This module makes use of OS functions and interactions to achieve various tasks in our program. Not all of the methods in the OS module will function on all platforms however it is possible to detect what platform our program is running on and account for this problem. This problem can be accounted for by only executing the portion of the code relevant to the user's OS and not executing the portions relevant to other OS's. The reason we do not plan to maintain a separate codebase or separate distributable for separate OS's is because the portion of our design that executes OS functions is diminutive compared to the GUI creation and deployment, astrological computation, and other sections. The vast majority of design will focus on python based information management and TKinter based GUI design, which will both execute equally well on all three platforms. Additionally, minimizing the number of distributables will ease the use of our program for our client.

The previous edition of obsprep also relied on a pre-established environment variable. However this is a dependency we hope to remove from our product either by handling the creation and modification of this variable ourselves, or annulling its necessity through simplified file management within obsprep.

2 - Environmental Requirements necessary for our product to run

The expectations for our program to function are fairly generic. The host machine must be running a compatible operating system such as ubuntu, mac OS, or windows. The host machine must have sufficient storage and memory to execute the program, these requirements will not be beyond one gigabyte each and is easily attainable on any home or work computer.

The same is true for computational resources, the program does not need more than one thread to operate efficiently and will not need a significant quantity of speed from the processor that can't be found on the average home or work machine. The program will need to be installed in a working directory that should not be modified except in small and particular ways. It is useful for the user to be able to access and personally modify their folder of saved observational plans and graphs, however rearrangement of files used by the program is highly inadvisable and would result in failure to execute, unexpected behavior, and loss of functionality.

5 - Potential Risks:

In this section we will consider potential risks to the project's success, and how these risks could impact project development and the client's operation. Each risk will be judged by two factors: the likelihood of it occurring, and the severity of the consequences if it does occur. Both of these factors will be ranked low, medium, or high. For likelihood, low means that the risk is unlikely to occur, medium means the risk has a moderate chance to occur, and high means that the risk will occur more than half the time. For severity, low means that the risk could cost our team or the client several hours of work and very little to no financial loss, medium would cost several days of work and little financial loss, and high would cost more than a week of work and big financial loss. These are the risks we have deemed most relevant to the project:

1 - Risk Mitigation

Rapid depreciation: Learning from the past iteration of obsprep it has been proven that an easily maintainable interface is very important. With this past iteration of the obsprep software it was given years of small little patches to try to help keep it up to date. This proved to not be as helpful because through all these tiny updates it became unfixable and rendered useless. The previous developers of the software did their best with upkeep of the code structure but through years of different developers coming in and trying to fix it, the past code is very hard to follow along with what file performs what actions. It is very unlikely for code to rapidly depreciate but there is a chance of it happening if Simbad the online database holding the stars information becomes unsupported or if Tkinter the gui framework becomes unsupported then our code could depreciate rapidly. For these reasons we are giving it a likelihood of low. Our plan to help prevent this is to have a clean and uniform code structure so any new eyes that come on to maintain the project can follow the code path easily. As well we plan to have thorough documentation so it will be easy to figure out exactly which functions are performing what actions. Through these plans to help future developers who may need to maintain it we are giving it a low severity because it'll be easy to figure out where the problem is and how to fix it.

Protracted training period: The next risk we have with our project is the interface not being intuitive and user friendly. If our application is not able to be easily used by the observers the operational time will increase and the developers will have to go in and fix the layout to make it more user friendly. For these reasons we have given the severity a medium. This is because they will have to learn how to use our new application taking time away from their work as well as having to bring in a developer to fix the layout. Our project's goal is to decrease

operational downtime so if our interface does not increase efficiency this will mean we have failed. The less time they have to take fidgeting with our program the better. Our plan to mitigate this risk is to make sure that we are showing our clients the interface as we develop to get active feedback as we work on it. Once we have the interface setup the way they believe will be best this will help ensure that it is intuitive and user friendly. With our plan to build the interface off their feedback the likelihood of having a non intuitive interface is low.

Incorrect Calculations: One of the core functions of our project is to maintain a library of astronomical functions essential to the observatory's operation. Users can access these functions through the graphical user interface to predict star locations and aim the observatory's mirrors. It is possible that our program could implement these calculations incorrectly, as many of them are extremely complex. The project also relies on data collected from an online database, which then needs to be processed. It is unlikely to happen because the calculations are already done for us. We just have to create calls for the specific calculations we will need. There are a few ways we could get wrong calculations though, we could call the wrong function for the wrong task, the user could input the information in the wrong order, or when returning the calculation to the user it could be formatted wrong. All of these possibilities could happen but are unlikely so we are giving it a low likelihood. If anything goes wrong and our software performs a calculation incorrectly, the users are unlikely to realize until the night they actually plan to observe those stars. In that case, the observatory will most likely waste a full night of operation, as the calculations will be extremely difficult to perform by hand on short notice.While observing stars if you miss a night you are not guaranteed to see it the next night it could be months or even years till you can get that reading again. As well, each night of research cost 12,000 dollars so missing a night is a massive waste of money. With a potential missed night of observation we are giving it a severity of High. To help reduce the chance of this risk occurring we will be doing through testing on function calls making sure that we are calling the right one for the given task. We will also be making sure to check that the output we are receiving is correctly formatted for the observers to use. Finally we will be labeling the input boxes on our program to make sure there's no confusion as to what input goes where.

Failure to save data: Another possible risk involves our software's ability to save a record of the collected data for later viewing. This would seriously compromise our clients productivity. There also is the problem with the day crew not being able to get access to the data the observers got at night. The likelihood of not being able to save the data is low because there is not a lot of functionality included in saving data so we are giving it a low likelihood. If the application isn't saving the data files correctly there could be either a complete loss of data or a lack of data. With the loss of data it will make our clients job strenuous and make it so they have to manually record the data themselves increasing occupational time. For these reasons we are giving it a severity of high. This is why we are making sure we have a neat file tree and our documents are all properly stored and formatted. As well we will be testing the application to make sure that anytime you try to save the data it saves. With these methods invoked we believe we will be able to mitigate any kind of risk that comes with improperly saved documents.

2 - Risk Overview

Risk	Likelihood	Severity
Rapid depreciation	Low	Low
Protracted training period	Low	Medium
Incorrect calculations	Low	High
Failure to save data	Low	High

Figure 5.2: Risk Analysis

6 - Project Plan:

Our team's project plan is broken up into five different milestone goals. Each goal builds off the previous milestone and with this structure we believe that we'll be able to create an intuitive interface that will help aid in the organizational challenges of setting up a nightly observation. These goals are a completed architectural outline, a prototype interface, interface reacting with test data, a successful connection with the backend, retrieving and submitting data to the backend and finally the complete working project. With these six goals We believe that we will be able to meet our clients standard given our time constraints..

Our next milestone is to create a prototype interface. Our prototype interface will show off our possible design features we would like to implement. It will have very little functionality but will be more used for showing what the application itself will look like. In this stage we will be showing our clients our take on the application design and getting their feedback. Our goal is to have a skeleton interface that has an organized layout that will call a novas function and return the output to the user. As well, we want to display a graph with our matplotlib library that will just create a graph based off test data we give it.This will help us in later steps because we will not have to worry about design layout later. As you can see in figure 8.1 we have started development on the prototype and plan to have it done by May 2nd. At the end of this milestone we will have our interface layout nailed down with our clients approval.

The second milestone we have setup in place is having our prototype interface being able to interact and display test data that we feed it. This will help us start to implement functionality within our interface. Our test data will closely resemble the actual data that we will be retrieving from the backend. Having our application interacting and working with test data will help us know that we are moving in the right direction with development. We plan to break this down into two parts. The first part is having the interface return the data correctly to the user. The next part is for the user to be able to format the data in the way they'd like it such as a table, list or just a set of numbers. The important part to this milestone is making sure that the test data we are testing it with is the same structure as the data in the backend. From figure 8.1 we can see that our plan is to start development on this in later May all the way through July.

We want to take our time on this step because once we have it working perfectly with the test data it will be an easy process to transfer it to the actual data. Once we have the interface working flawlessly with the test data we will be able to move on to our next milestone.

Our third milestone will be to set up and establish a connection with the current obsprep backend. This is going to be one of the biggest and most important milestones. The whole purpose of this project was to be able to communicate and receive information from the backend. Our user interface will already be set up to display similar data structures so we will just need to develop our function calls to the correct function given the specified needs. Our plan for this step is broken up into two parts. The most important part is to be able to call and retrieve information from all the novas, backend, functions. The next step is to be able to retrieve the star data, location, zenith angle and time of observation from the Simbad online database. Our goal is to start this around the time we get back into classes for the fall semester which will be later August and work on this part till the end of September. Once we can successfully create a connection from the user interface to the backend data that will mean we have successfully completed this milestone and are ready to move on to our next one.

The fourth milestone we have is to actually retrieve and submit data to and from the backend. We will already have a successful connection to the backend so all we will have to do from there is retrieve the data from it as well as store data that we get into it. Assuming we have successfully completed the last milestone this one shouldn’t be too difficult we will just have to involve some commands to retrieve and store data. This will involve grabbing data from the backend and then finally displaying it to the user on the interface. We plan on testing each function individually to make sure that we’re getting the correct information back from it. As well we will be doing tests to make sure that the data is stored in the correct place and being saved in the correct format. Development on this part will start in late September and we plan to finish in early November. After we finish this milestone that will mean we are just one step away from a completed project.

Our final milestone is to have the fully functioning application done. Through our previous milestones we have a mostly functioning application but we just need to add a few more things to the end product. This includes developing an installer script to ease the installation process, implementing some helpful tool tips so the observers can get help on how to utilize certain features, creating and displaying graphs, and some other small details to implement such as color formatting and clearing empty space. This will start in late October and finish up in early December. All of these last implementations will be easy to implement once we have a mostly functioning application, that is why we waited to include them within our last milestone.

Through our carefully created milestones we believe that we will be able to deliver the best possible product to our clients. It is important that we follow these steps in order to help avoid potential development issues we may run into. This is why we have each milestone built off of one another. Our road map is very thorough and well thought out leaving very little room for error and at the end of it our application will be able to perform all necessary tasks for setting up an observation.

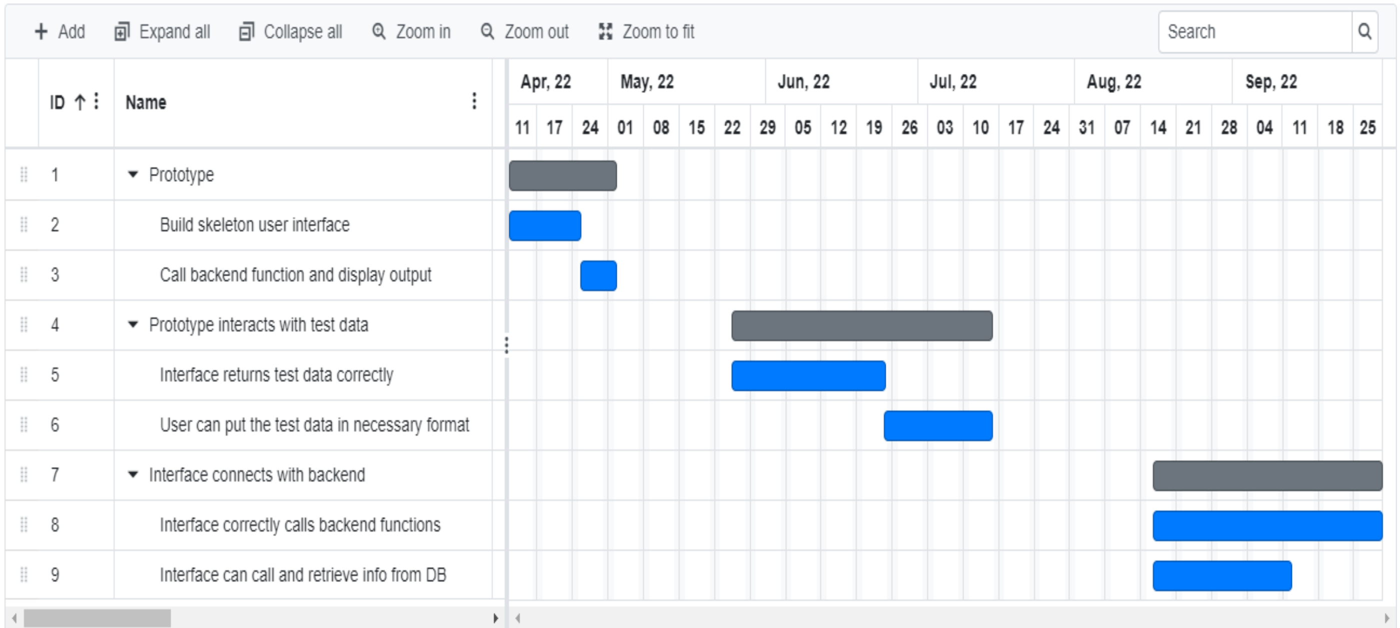

Figure 8.1 - Gantt Chart

7 - Conclusion

The NPOI has developed an alternative to the widely used single-aperture telescope design by combining the light from an entire array of telescopes. This innovation has allowed them to achieve a degree of precision far beyond that of a traditional telescope, resulting in the highest angular resolution of any observatory in the world. The NPOI's research has contributed to GPS systems, star charts, and revolutionary research into distant astronomical phenomena, particularly binary star-systems. The observatory is expensive to operate, and requires very specific weather conditions, so the NPOI needs to make the most of every opportunity they have. Planning out an observation night is a complex and time-consuming task by hand, but the software application designed for the purpose no longer works as intended, so they are left with no choice.

The solution outlined in this document is designed to enable the NPOI to make the most of their time by simplifying the organizational challenge of planning out their nightly observations. Our project will allow a user to perform complex calculations and comparisons that would take hours by hand in a matter of minutes. It will replace an outdated software application with one that is easy to learn and use, that displays data quickly and accurately, and that specifically addresses the flaws of its predecessor, such as the complex installation process or the inability to save outputs for later viewing

While there are risks inherent in our design, we are confident in our ability to manage them and handle their potential impact. We are currently working on a technical prototype of our product that meets most of the requirements described here. By the end of the year we will have created something that will contribute not just to our client, but to the astronomical community at large.