Operation Dark Sky

Software Testing Plan

November 11, 2022

Team: Justin Ceccarelli, Jordan Tatum, and Luke Thompson

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

Mentored by: Daniel Kramer

Version 1.1

Table of Contents:

1 - Introduction:

Operation Dark Sky has repaired and updated a software application called obsprep, short for observation preparation. Obsprep was previously created by the Navy Precision Optical Interferometer (hereafter referred to as the NPOI) to help astronomers calculate the precise calibrators necessary for their observatory to find and capture the light from a specific star in the sky. The obsprep application had been in use for over a decade when it finally encountered a critical bug that forced the NPOI to retire the program. Early in 2022 they hired Operation Dark Sky through NAU's capstone program to fix their software. Our goal with this project is to restore all of obsprep's original functionality while remaining faithful to the original design and interface. We have worked closely with two members of the NPOI over the past year to make sure that obsprep correctly performs complex astrometrical calculations, and functions properly on the NPOIs particular hardware.

In order to ensure that our client's requirements are fully met, we will be performing a battery of tests. These tests measure the performance of our project's individual components, the interfaces between those components, as well as the user experience. We will begin with unit tests, which measures the accuracy and functionality of the individual modules and methods. This will primarily involve comparing our program's output to a set of precalculated values to ensure that each of our program's individual components are working properly. The next step will be integration testing, or a test of the interaction between obsprep's major modules. This will also be done by comparing obsprep's output to a value calculated by hand. The final tests will measure usability, or the quality of interaction between users and the software.

2 - Unit Testing:

Operation dark sky will be conducting a series of unit tests in order to ensure that each distinct module within the program functions correctly. Each module will be subjected to a series of tests that compares that module's output under different conditions to a hand-calculated result. If the module output matches the expected result, then the test has been successfully passed. If the test is not passed, then there must be a flaw or bug in that particular module, and the team will review the code related to that module before running the tests again. This will continue until the module passes all provided unit tests. The unit testing process will include test inputs that should result in errors. In these cases it is important to confirm that the correct error is thrown, and that there aren't any unintended effects outside of the error message.

In this section we will describe each individual module within the application, and give a brief description of the tests required to ensure their proper function. In general, each module will be subjected to a battery of five tests. Each test will be made with a

variety of different and distinct inputs intended to catch unique interactions and edge cases that are likely to create conflicts or errors within the module. There are two different types of failure: errors, and inaccuracies. An error occurs when the test cannot generate an output at all, or that output is in an unacceptable format. Errors can stem from a broad variety of sources, but unit testing allows us to narrow the issue down to a small subset of the program's codebase, making them dramatically easier to resolve. An inaccuracy is when the test returns an output that does not match the manually calculated output. An inaccuracy is typically more difficult to resolve than an error, and is also potentially more harmful than an error, since it isn't always obvious that something is wrong in the first place. By detecting inaccuracies through unit testing, the team can isolate the source of an inaccuracy to a specific module and fix the source.

Our software application, obsprep, implements a large quantity of complex astrometric functions alongside a catalog of star data. Since the mathematics used are so complex and specific to their particular field of study, we will be cooperating significantly with our client to create tests that accurately reflect each module's intended function. We will be relying on our client to help us make sure that our test outputs are accurate.

2.1 - Add Star to Target Table

The user can add a star to the target table by typing its proper name or designation into the field below the target list. If obsprep can find the provided name within the catalog, it will add the star to the target list. For this test, we will attempt to input a variety of different star designations to confirm that the appropriate star is always returned for a valid entry, and that incorrectly input names consistently result in an appropriate error message and will not add data to the table. We will attempt to find edge cases by inputting extremely short names, extremely long ones, and names containing unusual characters, including parentheses and quotation marks.

2.2 - Delete Star from Target Table

The delete button removes a selected star from the target table. This function is very basic and should not be liable to errors, but we will still perform several tests to ensure that it performs consistently by deleting a variety of stars with different traits from the table.

2.3 - Calculate Calibrators

After a star has been added to the target table the user can select it to bring up a list of calibrators for that star. We will cooperate with our client to calculate several calibrators by hand, and confirm that the values provided by obsprep are correct.

2.4 - Save Obslist

A user can press the save button to create a .txt file containing all the information currently displayed by obsprep, including the full list of chosen target stars and their calibrators. Testing this functionality will involve comparing the data captured in the output file against what is displayed on the GUI and a hand-calculated output.

2.5 - Read Obslist

This function allows the user to input a previously created obslist file into the program to automatically load the stars and calibrators into the application. Testing this will involve creating a selection of obslist files, reading them into the application, and confirming the accuracy of the data added to the GUI.

2.6 - UV Plot

The ultraviolet plot button generates a graphical representation of a star based on a provided time, station, and spectrometer. We will work with our client to establish a mix of stars and specifications that should test the limits of the plotting and display software.

2.7 - Visibility Plot

The visibility plot button generates a graphical representation of a star based on a provided time, station, and spectrometer. We will work with our client to establish a mix of stars and specifications that should test the limits of the plotting and display software.

2.8 - Star Plot

The star plot button generates a graphical representation of a star based on a provided time, station, and spectrometer. We will work with our client to establish a mix of stars and specifications that should test the limits of the plotting and display software.

2.9 - Uptime Plot

The uptime plot button generates a graphical representation of a star based on a provided time, station, and spectrometer. We will work with our client to establish a mix of stars and specifications that should test the limits of the plotting and display software.

2.10 - Coverage Plot

The coverage plot button generates a graphical representation of a star based on a provided time, station, and spectrometer. We will work with our client to establish a

mix of stars and specifications that should test the limits of the plotting and display software.

2.11 - Siderostat

The siderostat object is capable of storing astronomical position data and checking if the astronomical position is visible from a particular geographical location.

2.12 - station

The station object is capable of storing accurate geographical location data.

2.13 - NPOI object

The NPOI object stores a series of stations representing the actual NPOI array of telescopes.

2.14 - Target

The Target object contains the information about a star necessary to aim a telescope at that star and to determine valid calibrators for it. It also successfully writes itself to a file and contains the method to add it to the target list in the GUI.

2.15 - calibrator

The calibrator object stores the information on a star used as a calibrator for a target, associates itself with that star, and writes itself to an output file. Notably, it writes itself to the output differently from the target class and does need to add itself to a list.

2.16 - GUI

Contains widgets associated with most of the other units above. It contains the buttons that those units use to call their associated methods. The Save obslist for example contains a button used to call a function that writes out the current obslist, This unit contains that button and all the others. It represents the visual appearance of the GUI and can be tested by simply using demo output from every button to ensure each widget properly calls its method.

3 - Integration Testing:

Integration testing is a process designed to ensure that the distinct modules within the program interact correctly with other modules. It makes sure that the input and output of each module is formatted appropriately, so that the data shared between modules is correct and accurate. Integration testing bridges the gap between unit testing and usability testing. It ensures that the entirety of the product works as intended

on a large scale, similar to usability testing but from a designers perspective, instead of a users perspective.

There are 77 methods across 6 classes in our program. Integration testing will require us to examine how each of these classes interact with the others. The methods will be tested and approved through unit testing, and the classes having both internal and external integration. Internal integration refers to the circumstances in which a class's own methods interact with each other, and external integration refers to the circumstances in which multiple classes interact.

As part of all of our harnesses we will be “mounting” each of our classes individually and then in relevant combinations together to test their integration. Mounting is the process of isolating these classes from other elements of the program and providing the necessary environment for them to be run, receive test input, and receive test output. In every integration point the GUI class will need to be mounted so that it can run without access to any classes not within the scope of that integration point. We will now summarize some of the integration points for our program.

Integration Point 1: Target Object Class and calibrator class to Backend

Modules involved: The target Object class and the computational backend

Harness: A large array of target objects and an array of calibrator objects will be created based on known stars using the relevant methods for the back end, and those target objects will be compared to the existing database attributes of those stars.

Correct integration: The Objects created are cohesive with the known qualities of those stars.

Contract: The Target Object class and calibrator class are able to correctly store and manipulate their own relevant data.

Integration Point 2: Target Object and calibrator Class to GUI

Modules involved: The target Object class and the GUI class

Harness: A list of Target Objects for the GUI to interact with, manipulate using the targets own getters, setters, and functions, and for the GUI class to place in the targets table, place in the calibrators tables, and create graphs from. Stubs for the siderostat and station objects.

Correct integration: The GUI is able to use the list of target objects and calibrators to accurately fill the targets table, the calibrators table, and various graphs with the information from the targets list.

Contract: The lists of targets and calibrators are accurately generated and fed to the GUI.

Integration Point 3: Siderostat to GUI integration

Modules involved: Siderostat class and the GUI class

Harness: Stubs for the station, target, and calibrator objects. An array of siderostat objects for the GUI to interact with that is created using the siderostats constructors. Output method for the information within the siderostat object and the ability to output the result of the siderostats visibility check from itself and from the GUI object.

Correct integration: The GUI is able to receive and utilize the information from the siderostat object. The siderostat object constructor is able to be used to create complete siderostat objects.

Contract: The results of the siderostat visibility check are correctly used later in the GUI.

Integration Point 4: Station object to GUI class integration

Modules Involved: The Station object and the GUI object.

Harness: The array of station objects that represents the stations at the NPOI is accurately created and fed to the GUI class which has been mounted to allow it to run with only the station list as input (bypassing the NPOI class).

Correct Integration: The Stations can be correctly selected and their baselines and valid spectrometers also selected in and only in valid configurations for how the real world NPOI operates. The GUI can successfully display station objects.

Contract: The list of station objects is accurate to the position of stations at the NPOI.

Integration Point 5: NPOI object to GUI class integration

Modules Involved: NPOI object, GUI Object, and station Object.

Harness: The Gui class has been mounted to allow it to run without the Target and calibrator objects.

Correct Integration: The NPOI class successfully and accurately creates the station object list and integrates it with the GUI class. This is the same successful output as integration point 4 however the station information is created dynamically from the NPOI object instead of hard coded.

Contract: The GUI can successfully display station objects.

Integration Point 6: GUI class and Obsprep Backend

Modules involved: All.

Harness: Received information from the backend, and the information being sent to the back end are printed to the terminal or a file as it is processed to allow us to confirm it's validity. Aside from this for more precise analysis this test can largely be done by running the whole program and examining the entire product.

Correct integration: The GUI class is able to successfully send and receive information from novas, vizier, and other backend modules to formulate the output for our user. There are also edge cases of indirect integrations that will be checked here, for example the visibility check for the siderostat object is now able to checked within the context of the NPOI and station inputs.

The Integration is examined both for accuracy of transmission through the terminal output, and accuracy of actual output through the GUI display.

Contract: The GUI class is able to correctly integrate with all constituent members of the obsprep.py file, being the siderostat, station, NPOI, target, and calibrator objects.

4 - Usability Testing:

For our usability testing it was clear that we had a niche crowd of possible end users who would use our program. This program is only going to be used by the Flagstaff NPOI observers. They employ a fairly small team of observers, so our options are very limited as far as people who can actually test this. We have sent our clients our current program to pass on to the observation team with a request for feedback. This

allows us to get a variety of feedback from our primary user demographic on how easy it was to set up/install, the interface's intuitivity, and if the program outputs are accurate and useful to their work. Our usability testing plan involves both direct discussion with our potential user base, as well as a short survey. Our client will eventually provide the observers with several “challenges” to find different astronomical values using only obsprep. The final test for our prototype will involve using it to actually observe a star using data provided by obsprep. Through these tests we believe we will receive the most productive feedback for our program.

For our user testing it was important to us that we choose the correct users to test this program. This is because our program is specifically designed for a very niche group of users that have access to tools and education related to the program. In addition, it returns information specifically calibrated to the NPOI’s geographical location here in flagstaff. Users in other observatories could attempt to use the program, but the values output would not be correct. Since the output is so specific to location, the only way we can be absolutely sure of our program’s functionality is to test it at least once. Since operating the observatory is an expensive undertaking, we want to be as sure as possible of our program’s functionality before moving to this stage.

Our ideal user would be someone unfamiliar with the previous iteration of obsprep, so that they can view the program without any bias or previous expectations. Unfortunately it appears that the majority of the observers are at least somewhat familiar with the program, but we have asked our client to let us know about anyone who has been hired in the months since the original program was retired. We are unsure of the total size of the observation team, but we are attempting to get as many of them to test the program as we can. This should give us a reasonably wide range of feedback.

We met with the head observer for the NPOI team to discuss our testing strategy. During our meeting we discussed a list of tests they could perform on the prototype. The first usability issue is the setup and installation process. The project contains a readme file that describes what needs to be done in simple terms, as well as a more technical description of the dependencies that need to be installed. We have a few questions for the observers to answer: we want to know how long the installation process took, if the readme file was sufficient for easy setup, and what steps they had trouble with. This part is important because installation difficulty has been a major issue at the NPOI in the past. We hope this process will provide useful feedback that our team can use to ensure the installation process is simple and intuitive, and to discover ways to improve it within our final project.

The next test we will run is to have the lead observer tell the team to get the observation information for a star on a specific date. This test will have the end user go

through a possible use case which will be useful for testing our programs functionality and intuitivity. During this test they will add a star or stars to the interface, set their observation details (station, reference station, spectrometers, tracking baseline, and the apertures of their interferometer) and document the data they receive back. During this test we also provide a survey to the users. We want to know how easy it is to maneuver around the program, how easy it is to fill in fields and provide input, what steps were difficult to understand, and any particular features the users believe could improve their experience.

The next step is for them to save their observation information and to load the saved file into their interface. This test is very important because saving an observation file and then loading that up at a later date is a major part of their job. A lot of times they will plan their observation the night before then load up the observation file the current night they plan to observe. If they are unable to load or save the file this will make our program much less effective and possibly have them lose out on a night of observation. What we want to know from this test is was the observation information able to be saved successfully, did the saved file contain accurate information, were you able to load in that saved file, and was the information that was loaded into the interface accurate. This will let us know exactly how well our save and read functions are performing and if any changes need to be implemented. The final test that they will run on our program is to then use that observation information to attempt to observe their star. This is a very critical test because this will let us know if the information we are returning to them is accurate. If the information is not accurate, that will let us know that we have a big problem within our code that needs to be fixed immediately. The reason we are making sure to include this in our testing is because this is the real way to know if the information we are calculating is accurate to their observations. We only have one thing we want to know from this test is if they were able to successfully observe their star from the information provided to them. Through going through these tests we will receive critical information about our programs functionality.

We plan to receive feedback in a week on November 18th 2022. The lead observer will give us the observers notes and answers to the questions we have sent them. We will then as a group sit down and review the observers feedback and create a plan based on what we receive. The8 will give us enough time to implement any changes that may be necessary. Our user testing has been carefully planned to make sure that we receive information pertaining to our interface.

5 - 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.

Our goal is to present our client with an easy-to-use program that saves NPOI observers time and effort. For it to accomplish that, it needs to work consistently and accurately. Requiring the NPOI's technical team to update or maintain the program is not acceptable, so it needs to be as reliable and bug-free as we can possibly make it. We hope that the tests described in this document will provide us with the feedback we need to prove our programs reliability and identify any potential issues so that we can resolve them before shipping the final product.