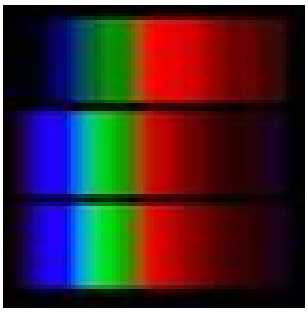

Software Testing Plan Version 1.0

2023-02-17

Project Sponsor: Joe Llama, Lowell Observatory

Faculty Mentor: Rudhira Talla, Northern Arizona University

Team: Empyrean- Henry Fye, Nhat Linh Nguyen, Jakob Pirkl, Jacob Penney, Kadan Seward

Overview

This document outlines different testing strategies that will make up the quality assurance of this project, which includes unit testing, integration testing, and usability testing.

Table of Contents

1.0 Introduction

Our team's product is a web application commissioned by Dr. Joe Llama at Lowell Observatory for the purpose of improving his and his fellow researchers' user experience when working with spectrographs. His work depends on the usage of spectrographs and the software that manages them, but the software they currently use, which is purportedly the industry standard, is unwieldy because it does not provide abstractions for details that they never truly need to concern themselves with, making its usage unnecessarily complicated. This program is also closed source, making contribution impossible for the very users that depend upon it for their research. The specification that he provided to our team asks for an open source application that firstly provides a simple interface through which to use, monitor, and manage the hardware they depend on and, secondly, for this application to be built using popular, well-supported technologies that will be more likely to be accessible for the average programmer so that the likelihood of it being supported as an open source tool for a niche target community is higher.

As with all software, this application should have thorough testing. Software testing is the practice or process of evaluating the efficacy of software at completing its intended purpose. Testing can be performed manually, for example by clicking the button of a website to ensure that it functions, or automatically, using a variety of types of tests built as software that execute various functionalities or parts of a software system and check what the output of that execution was. Automatic testing is protean: it can be performed at various scales and granularity, allowing software engineering teams to test fine pieces of functionality, such as guaranteeing the output of a single function, up to the system level, testing how massive tools integrate with each other. Our product, a web application with a relatively typical structure for such a product, requires a few different types of testing in several places.

Firstly and in order of granularity, we intend to perform unit testing on all major functionalities of the frontend and backend of our application, as well as the hardware interfaces that we have inherited and refactored. This includes interfaces for ZWO ASI-brand astronomy cameras and Shelyak-brand spectrographs. This will ensure that the core functionalities that comprise our application will be guaranteed by verifying that the lowest levels of our application perform correctly and in a predictable manner. Moving a step higher, we will perform integration testing at the junctures of our subsystems, such as between the frontend and backend of our application or between the backend and external tools such as hardware interfaces and our database. This testing will ensure that these various subsystems will communicate as intended, such as when passing user commands and data between themselves. Lastly, we will perform usability testing, which will exercise the end-user interface. For us, this is a website which attempts to expose all of the aforementioned functionalities to the user in a simple, abstract manner. This testing will attempt to guarantee those desirable qualities, looking at how easy it is to navigate our tool.

The motivation behind this testing regime is to ensure that our application meets the ethos or spirit that our client is searching for in commissioning this product. As mentioned above, this tool should abstract away the complexities of operating the hardware our client depends upon and should itself be simple so that it is easy to maintain and lives a long life as an open source tool which a niche professional demographic may depend upon. To attempt to fulfill these characteristics, our testing should emphasize searching for places of complexity, whether that complexity is in the end-user interface itself, how a single function operates (such as

whether it is too large or performs more than one job), or how a module operates (such as violating the single responsibility principle) or communicates with other modules. Achieving this goal demands that, materially, we emphasize testing the components that we built that interact with each other. For example, testing the interface between our backend and database is a lower priority, because the database and its interfaces are professionally developed and maintained, meaning the only aspect we can reasonably focus on in that area is how we access it, not what it provides. On the other hand, the communication between our web interface and backend are functionalities all crafted by our team. While the interfacing between those two modules depends on standards such as HTTP requests and websockets, whether they communicate correctly depends upon us (as opposed to how our database works, for example). Below, we explore our plans for unit, integration, and usability testing at length, discussing how we plan to ply these methods for the outcomes that I've described here.

2.0 Unit Testing

2.1 Introduction to Unit Testing

Unit testing is one of the most common ways of testing code, relying on concepts such as modularity to drive assurance. Unit tests are written at the smallest level of code, usually a function or method, testing their validity with known outcomes. These tests are designed to test the smallest pieces of code, without interference with other functions or other parts of the system, such as a database, or other servers.

The goals of unit testing are multifaceted. One reason is to increase documentation. The tests themselves provide documentation, as future writers will have some idea of what inputs to the function will return. Going back to write tests for older functions also forces writers to read their functions and add documentation to sections that were not as obvious. This can also lead to rewrites of code in order to increase clarity. Unit tests also allow programmers to check their work, since once a function is written, the function can be checked by writing a unit test for the function to ensure proper functioning.

Unit tests are measured in several different ways to ensure that they are adequate. Code coverage is one of these metrics which details how much of the code in a unit was “covered,” or run, through a set of tests. This is a valuable metric for unit testing, and Empyrean will have 100% code coverage for each unit tested in this way. Breaking that down, each unit in which tests are run will have enough test cases so that each line of code in said unit will be run at least once. Another metric that Empyrean will hit 100% in is test pass %. This is a more obvious metric, measuring the percentage of tests that were written that passed. There will also be more qualitative metrics that we will keep track of such as Test Duration. Test duration will not have concrete goals to hit, but it will be obvious during runtime when there are failures in this department.

The next two sections will detail the process for testing each language used in the makeup of the code, python and javascript. This will begin with a breakdown of options for testing, followed by a list of units that will be tested, along with the boundaries of those units and what erroneous values will be provided to test robustness.

2.2 Python Unit Testing

Due to the popularity of python, there were many available options for unit testing. The three most common options for writing unit tests in python are doctest, unittest, and pytest. Doctest is by far an outlier when compared to the other two. This testing framework works via native docstrings. Through this, it can run the code and verify that the outputs are what the docstrings specify. We decided against this framework however, as it bulks up each function's code, increasing the length of files, and is unintuitive to learn. Another option was unittest, python’s native unit tester. This is a more popular option, being based on JUnit for Java. That means it has common syntax. However, we also decided against this option as default operation of this module lacks feedback for the user running these tests. It is very common for the only results being the percentage of tests that pass without any further information. Pytest is even more popular than unittest, despite being a third-party package. Additionally, there is less boilerplate as Pytest is a python package through and through. It also provides more robust feedback from tests, even going so far as trying to find the line of code that offset the results of

the function. Because of these features, and in the spirit of our project, if made more sense to test python with pytest.

The following will be the units that will be tested on as well as the boundaries and incorrect values that will be provided to each unit. Notable exceptions from the list of units are functions that interact with our database, as that is a kind of integration test, one that is conducted by the creator of whatever library we have chosen to use. Additionally, similar functions, such as the constructors of model classes will all be the same. Included will be the code in instrument files which control the spectrograph and camera if they are entirely internal. Thus:

We will focus code on the backend of our application. Most of this code deals with database access, and thus is not in the scope of unit testing. These are the constructors and methods of each of our models, and one function which deals with file writing.

Instrument Model's Constructor: This method creates an Instrument object which can be submitted to a database.

Equivalence Partitions: The only argument for this method is a name for the instrument, which is a string, so the partitions are along the type of the given argument. The acceptable arguments are non-empty strings, such as "ZWO ASI2600MM Pro". Then blank arguments (strings composed of whitespace) will err. Empty strings will also err e.i. " ". Finally, objects that are not strings, including None will err, as a name is required for our database.

Observation Model's Constructor: This method creates an observation object to be submitted to the database to be used in the future. Unit testing this will also test the class's set_attrs method, as the constructor is a wrapper for the function.

Equivalence Partitions: This function takes in an initialization dictionary, as there are many fields that need to be set in the database. This means that many of the fields will need to be checked in the same way as the previous unit, such as obs_type, and date_obs, will need to be checked. The dictionary will also need to be checked. Entries that are not in the attribute list of the class will need to be checked, such as "NOTREALATTR": "HELLO", which will need to fail. Finally, different types of objects will be the most obvious failure of this function, where any non-dictionary will err.

User Model's Constructor: This method creates a user object that can be used to login, which will be stored in the database.

Equivalence Partitions: This function takes in two strings as arguments. Refer to the Instrument Model's Constructor to learn about the partitions of one string. This is the same, except that there are two strings, both of which are required, so any combinations of the above scenarios can happen.

Status Model's Constructor: This method will create a status keeping track of the instruments during the operation of the app, which is stored in the database.

Equivalence Partitions: Like the above function, two of the values are strings, both required, so they have the same partitions. There is also an InstrumentID value, which refers to the instrument to which the status is connected. This is a positive integer, so correct values would be, for example, 5. Incorrect values would be other numerical types, like floats like 5.3, non-positive numbers like -1 or other types, like "Hello".

File Writer's __init_fits_abspath: This function figures out where to store data collected by the camera and returns the file path where it will go.

Equivalence Partitions: This function takes in two arguments, the file path to the directory where the FITS files are stored, and the name of the temporary file that the camera has provided. These are both strings, so they can first be tested in the same ways as with the Instrument model's constructor, but these strings are more specific. They must be a filepath and file name, respectively, and so all of the restrictions to those will be applied. As an example, a file path with whitespace in it cannot be used, like " / home/user/ Desktop/Data". A correct input may be "/home/user/Desktop/Data". For the file name, forward slashes cannot be contained, such as "file/fits.fits". Whereas a correct input may be "obs1".

2.3 Javascript Unit Testing

Similarly to python, there are many available options for unit testing for React and Javascript as a whole. While there are far many more options for JS, the three most widely used options for writing unit tests in Javascript are Jest, Mocha, and Storybook. Jest is the largest and most frequently used unit testing interface for React on the market. It is easy to use and has no setup, meaning unit test development is quick and easy. Its main feature is being able to take snapshots of large amounts of test cases, allowing for a larger variety of test variables for specific functions. Next is Mocha, which was the most commonly used testing framework until recently. The main drawback with Mocha is the overly complex setup process, however the testing suite is incredibly clean and can write complex and flexible tests for both front and back end JS. Finally there is Storybook, which acts more like a development tool with a built in testing suite. It allows you to create independent components that interact with each other and allow for ease of testing and documentation. It is commonly used for large complex projects with many pages, each with many components. Because of all these different choices, there was a discussion on which interface would be most useful for our specific project. We ended up going with Jest for its ease of use and hassle free setup.

The following are all units that will be tested on with correct, incorrect, and invalid inputs for a given unit. As our frontend is a large and complex system, with several interlocking components and pages, some unit tests might include 2 functions or more, so long as no major processing occurs between functions. This will not affect the test conditions as all function inputs and outputs will be logged. There will not be any testing for any React specific components, as they have already been thoroughly tested and documented. The front-end code is entirely based on the website and the UI, and as such will be closely coordinated with other integration tests involving the front-end so that there is not any extraneous overlap, whilst also testing all features and functions.

attemptRegistration - Registration Page: This function sends a post request to the backend and contains the logic to allow a new user to be added to the system.

Equivalence Partitions: The arguments for this function are the fields that exist on the web page in the form of asynchronous values. These fields are username, observer name, and password. The acceptable arguments are determined by the backend and as such any non-NULL ASCII string is a valid input.

attemptLogin - Login Page: This function sends a post request to the backend and contains the logic to login to the system with a valid username and password. The function will navigate back to the login page in the case of an incorrect username and password. Otherwise, the user is directed to the observation page.

Equivalence Partitions: The arguments for this function are the fields that exist on the web page in the form of asynchronous values. These fields are username and password. The acceptable arguments are determined by the backend and as such any non-NULL ASCII string is a valid input.

getChipProps - Log Sheet Component: This function gathers the data of a row in the logsheet table for later display.

Equivalence Partitions: The arguments for this function is a row from the log sheet table in the backend. This includes name, current progress, and icon. The acceptable arguments are pulled from the database and therefore will always be compliant in form and content.

showProgress - Log Sheet Component: This function takes a valid row's completion status and displays it as a loading bar.

Equivalence Partitions: The arguments for this function is a property of current progress in an observation. As this information is pulled from the database using getChipProps, the data will always be properly formatted and easily accessible.

initResolution - Observe Component: This function sends a post request to the backend to resolve an astronomical object into valid observable coordinates.

Equivalence Partitions: The argument for this function is the text field of the object to be requested. The backend deals with whether this is an acceptable astronomical object, and as such the valid arguments are any non-NULL ASCII string.

validateObservationRequest - Observe Component: This function checks that there are no current issues with any values being requested for observation. This checks the coordinates set by initResolution as well as user input.

Equivalence Partitions: The input for this function are all the fields dealing with an observation. Each field has regex checking that will return an error in case of incorrect formats. Thus, this function will check if there are any errors, and as such there are no incorrect input values.

initObservation - Observe Component: This function sends a post request to the backend to begin an observation with the current observe field values.

Equivalence Partitions: The argument for this function is the text fields of the object coordinates to be observed as well as observation duration and number of exposures. The validateObservationRequest deals with whether the values are acceptable astronomical inputs, and as such the valid arguments are any non-errored string.

endObservation - Observe Component: This function sends a post request to the backend to end an observation and log the results.

Equivalence Partitions: There are no input values for this function, as it is simply sending a post request to the backend and logging the response, or sending an error if one occurs.

updateData - Status Component: This function updates the statuses for the instrument. This involves setting color, text, and label for all of the current statuses.

Equivalence Partitions: The inputs to this function are all of the current statuses in object form. Each object is checked from a request to the database and its label, current status, and color are updated. There are no incorrect inputs as they are all status objects.

3.0 Integration Testing

3.1 Introduction to Integration Testing

It is undeniable that a typical software project consists of multiple software modules, coded by different programmers. Because of this, developers need to test the integration or interaction between modules and make sure that they work properly. Thus, integration testing becomes an ideal software testing that verifies the proper functioning of individual software components when they are combined and integrated into a larger system. The main objective of integration testing is to identify and eliminate any issues or defects that may arise when different software modules are integrated together.

The main goal of integration testing in software engineering is to verify that the different modules or components of a software system can work together correctly as a cohesive unit. This involves testing the interfaces and interactions between the individual modules or components to ensure that they behave as expected and that data and control flow smoothly between them. Integration testing helps to identify any issues that may arise when different modules are integrated together, such as compatibility problems or inconsistencies in data formats or communication protocols. By detecting and resolving these issues early in the development cycle, integration testing helps to reduce the risk of software failure, increase the quality and reliability of the software, and improve the overall efficiency and effectiveness of the development process.

Integration tests are a common method to verify the interaction between modules to meet requirements and correct data flow. The target of this type of testing is the modules in a software product, so the first step is to identify the components or modules to be tested including front-end, back-end, third party components or interaction with the device's system. Next, developers must define test cases by following an integration strategy, which designs test cases to test the interactions between the different components. Test cases should be developed to cover a variety of scenarios, including normal and abnormal use cases, and should be based on the system requirements and design documents. After defining test cases, the next step is to develop test scripts and execute test cases, which test scripts are developed by using Pytest package and the results of test cases are recorded and any issues or defects are documented. Finally, the testing comes to the analyzing and fixing stage where developers ensure that any issues or defects identified during the testing process are fixed and the tests are re-executed to verify that the fixes have been effective. This stage not only identifies any defects or bugs that need to be addressed, but it is also used to improve the quality of software products.

3.2 Front-end to Back-end Integration Testing

The software related to the spectrograph requires the connection or interaction between front-end, back-end and devices' systems. Thus, team Empyrean has discussed and illustrates that the integration testing is important because it helps to ensure that the two parts of the application work together correctly and provide a seamless user experience.

The front-end is responsible for presenting the user interface and allowing the user to interact with the application, while the back-end is responsible for processing requests, retrieving and storing data, and performing any necessary calculations or logic. With the

integration testing, someissues can arise when the front-end and back-end are combined, such as data transfer, compatibility, communication, and security vulnerabilities.

More specifically, the integration testing for Front-end to Back-end used the bottom up approach.That allows the team to start at the lowest-level components of the system, such as backend APIs or database queries. These test cases concentrate on the accuracy of response from front-end to back-end and the interaction between the back-end and database.

Valid User Login/Signup: This function indicates the test cases for the login and signup service of the product. More specifically, when the client sends their information including email, name and password through the login form, the function in the backend will receive the same information package and create a database object 'Account' for that client. Thus, it is clear that this test case will test for correctness of the request sent through the backend and whether the database server creates an "Account" object for the client.

Resolve Functionality: This function indicates the test cases for the resolve functionality in the Observe form. The resolve function is responsible for generating the coordinates of the object. The test case creates a driver function that sends a request to the back-end API and verifies the response data. Moreover, the test case checks the correctness of response back from the library.

Observation implementation: The test cases verify that the observation implementation in the product is functioning correctly. The test case sends a POST request including observation data to the path "/observation", then it checks whether a database object is created for the observation. The expected results are that the observation is displayed correctly in the component log sheet. The observation has a timer to countdown the observed time and displays "Completed" when the observation runs out of time.

Log Sheet Implementation: the test cases verify that the log sheet implementation in the product is functioning correctly. The test case sends a POST request including observation data to the path "/observation", then the test verifies the request package is received in the backend. There are 2 cases when checking the log sheet, and they are the current log sheet or new log sheet message. In the case of the current log sheet, the test checks the existence of that log sheet database object; otherwise, the test checks the log sheet object initialization. The expected result is that the chosen log sheet even the created one, is expected to display in the component log sheet

3.3 Back-end to Instrument Interface Integration Testing

One of the stretch goals that our team accomplished with this project was the implementation of an interface to external astronomical instruments. This interface defines a series of methods that must be provided by all instruments that desire to connect to the backend of our tool and function properly. Our current implementation allows us to narrow and focus our integration testing between these two categories of subsystems on these provided points of access. To test the integration between these tools, we will rely upon a test harness called PyTest. Pytest is one of the most popular Python testing frameworks that currently exists and is claimed to be simple and approachable. The fact that it is both popular and purportedly simple fits the project ethos discussed in our introduction. The following is a discussion of the interface and its endpoints, and how we can exercise them to ensure their proper functioning, and what outcomes should be guaranteed.

The “instrument” abstract class defines the common interface used by all external instruments. It contains several “magic methods” (special methods capable of being provided by Python classes) as well as public methods, cumulatively allowing for one to instantiate, delete, and manipulate an object of this class type.

Initialization: We must provide a test which ensures that instances of the instrument class are initialized properly and completely. During initialization, the instrument must

1. connect to the host (the backend of our application)
2. assign its name and, using that name, retrieve its unique instrument ID from the application
3. Initialize its hardware statuses in the application’s database
4. Initialize its unique callbacks

To test this functionality, we must guarantee that steps 1, 2, and 3 transpire properly. This a contract assumption that we can verify by

1. Checking the status code returned by the socket emission which connects the two devices
2. Checking that the instrument receives the proper ID
3. Checking that the database contains initialized statuses after connection

Deletion: We must provide a test which ensures that instances of the instrument class can be deleted without side effects or or the system crashing. To guarantee this, a simple test which deletes an instance of an instrument and checks the resulting status will suffice.

Retrieving the instrument name: This method is defined in the interface and guaranteed to be accessible to the back-end, but its implementation is hardware-specific. To guarantee its proper functioning, we must ensure that we receive the correct instrument name and a success status. The alternative is to provide an error status if the name cannot be reached for some reason. Because instrument names are defined in their firmware, we can check returned names of specific instruments against their known and confirmed names in the test source.

Updating the instrument’s status: This method is defined and implemented in the interface and has no relationship to the firmware other than relying upon the statuses that the firmware emits as the hardware operates. To test this method, we must compare statuses received on the back-end for submission to the database with the statuses that exist in the firmware at the same moment. The test does not need to check if the statuses themselves are sensical, given that the firmware decides which status contents are emitted during a given internal event, but just that the back-end is receiving the emitted data correctly. For example, we can guarantee this functionality by provoking discrete events, such as the camera initializing, and comparing the status received from the firmware in the backend and the status actually in the firmware with the status that we know to be given during this event.

4.0 Usability Testing

4.1 Introduction to Usability Testing

In conjunction with unit and integration testing, a very important piece of the puzzle includes usability testing. Usability testing is where a team gathers actual feedback from real users to understand if their product is working not only as intended but in a way that is natural for their users. Often, the users will “play” with the system in front of observers while certain questions may be asked or user stories played out. This helps to ensure that the user is happy with the system and that everything is working as intended while also aiding the team in finding new problems that they may have overlooked or new recommendations that could be made. By conducting these usability tests as soon as possible and over a period of time, issues within the system will be exposed and fixed while the client's happiness in the overall product will increase.

4.2 User Testing Pool

Now that usability testing has been defined it is time to explain how this form of testing will fit into this specific project. As this project has a very niche clientele, the range of users that will be testing the product is very limited. Our main contributor towards usability testing will be our client, Dr. Llama. Testing with our client has already been occurring over the previous weeks and will continue for the remainder of this project. The environment for the project has been set up at Lowell at about the halfway point for development and our client has been working to provide us feedback as he uses the product. As we continue to update the software, our team meets with the client every two weeks to test out the user stories detailed below. The only other contributors to usability testing will be any astronomer that uses this open source software. This will be outside of the scope of the project for Empyrean but will allow users to continuously provide feedback for the product over Github.

4.3 User Stories

Outlined below are the user stories that will be observed when conducting usability testing with our client:

Logging In/Out: The user will initially be prompted to enter a username and password. This will be created in preparation for usability testing and provided to the user. The user will also log out at the end of the testing session to ensure functionality.

Registering New User: Once logged in, the user will be on an admin account that allows the creation of new users. The user will navigate to the web page that allows registration and create a new user. The user will be asked to log in with these new credentials to test the system and then log back into the admin account.

Changing User Permissions: While the user is on the admin account, they will navigate to another web page where they will change the user permissions of the previously registered account. The user will again log in to this account to ensure changes were correctly made.

Requesting Observations Using Object: The user will be asked to make an observation using the Object tab. This will require the user to input an object they would like to

view, press the resolve button to populate fields, and input the number of exposures/exposure duration. The user will click the start button and the log sheet will be populated with new entries

Requesting Observations Using Dark: The user will be asked to make an observation using the Dark tab. This section only requires a number of exposures and exposure duration. User will click start and the log sheet will display the observations.

Requesting Observations Using Flat: The user will be asked to make an observation using the Flat tab. This section only requires a number of exposures and exposure duration. User will click start and the log sheet will display the observations.

Requesting Observations Using ThAr: The user will be asked to make an observation using the ThAr tab. This section only requires a number of exposures and exposure duration. User will click start and the log sheet will display the observations.

Sorting Observations in Log Sheet: Now that the logsheet is full of observations, the user will be asked to filter and sort the logsheet to their liking. This includes hiding/showing columns, filtering specific observations, and changing the density of the table. This will also include downloading the current state of the table to a CSV.

Searching for Previous Observations: The user will finally be prompted to search for previous observations on the page labeled “Explore Logs”. The user is given two calendar components that represent a range in time and when two times are selected the user will be able to search for all observations in that time frame. The user can then navigate the entries using the functions mentioned in the previous section.

4.4 Continued Usability Testing

This project is going to continue expanding once our team leaves it at the end of this semester and our client will eventually take full control. With this, our client will be able to make specific changes to the project that are deemed necessary and will have to conduct his own usability testing in the future. Due to the open-source nature of this project, it is likely that other users will come across problems or recommendations and will have the ability to make recommendations/contributions to the software through Github. This will ensure that the project is continuously evolving to fit the needs of astronomers all across the globe.

5.0 Conclusion

With a lot of hard work from everyone at Empyrean, it is feasible to complete this project to satisfy our client, Lowell Observatory and Dr. Llama. Upon completion, Dr. Llama will be able to operate his new spectrograph remotely, making it much easier and much faster to collect the important data he needs to do his research. To make our client's experience positive, it is undeniable that a software testing plan is necessary to ensure the quality and reliability of the software product. By creating a well-defined testing plan, the team can systematically identify and address any potential issues before releasing the software to our client.

A comprehensive testing plan includes three major testing components including Unit Testing, Integration Testing, and Usability Testing. Each testing component is responsible for testing different levels of functionality from the lowest to the highest level which is the entire system. It also specifies the tools, techniques, and resources required for testing and defines the roles and responsibilities of the team members involved in the testing process.

In conclusion, a well-defined software testing plan is crucial to ensure that our software product is reliable, high-quality, and meets the client's Minimum Valuable Product. The testing plan should be considered as the final stage and comprehensive, well-structured to industry standards and best practices to ensure that the software product meets the requirements and is free from errors and defects.