Team Controller
Northern Arizona University
Flagstaff, Arizona
April 5th, 2024

Zachary Parham (Team Lead): zjp29@nau.edu
Tayyaba Shaheen (Mentor): ts2434@nau.edu
Bradley Essegian: bbe24@nau.edu
Brandon Udall: bcu8@nau.edu
Dylan Motz: djm658@nau.edu

Software Testing Plan

Northrop Grumman

Weapon System Support Software

Harlan Mitchell

Laurel Enstrom

Version: 1.0

1.0 - Table of Contents

2.0 - Introduction

Software testing is the last crucial step in the weapons system support software. As the team finishes up development it's important to include testing as it helps find any unknown bugs and have a good way to test any functionality with the software. Our team will soon be handing over our project to our clients at Northrop Grumman and it's important to have testing done as the engineers at Northrop Grumman will be continuing our project. The engineers will be able to easily test any functionality with the program as the framework will be completely set up.

The team’s plan for software testing is using the QT testing framework for creating unit tests for all functions. Each individual function in our software will have a unit test that will run the correct data and the wrong data through it to catch any bugs or errors. It's important to make sure that each function can handle any data that comes into the function. Even if it's the wrong data the software should be able to recognize the issue. Then for our front end testing we will be conducting user cases and testing with many different users. The front end shouldn’t have any issues and each user that tests out our software will be asked to try every single feature on the graphical user interface.

This testing strategy is the most efficient way to test our software since it allows us to check back to the requirements and make sure that each requirement is tested. For any back-end requirement it's best to make a script that automatically tests since it is the fastest way to test any function. Then for the front-end it's best to do user cases since it's a bit tricky to do automated testing however it's also beneficial because it tests for any bugs and gives the team feedback on how the user likes or dislikes the software. Based on the user experiences the team can then make small adjustments to the software to improve the overall quality of the application. So for each requirement it will have a test that confirms if the requirement has been met. Some tests can confirm multiple requirements so this document will highlight the different parts of the software that is going to be tested.

3.0 - Unit Testing

Unit tests are small software tests that measure the effectiveness and correctness of that code section. Often, unit testing is performed on functions or a vital section of code. Throughout this document and the wider overall project, the team will be implementing unit tests that test an individual function. Alongside this, we will also be testing the error handling of each function by inputting bad inputs.

To carry out these tests, we will use the QT Framework, more specifically the QTest library. This library has the following functions that we will utilize to carry out the tests:

- QCOMPARE - used to check the accuracy of a value against the correct value;
- QVERIFY - used to check a boolean value.

3.1 - Electrical Class

The *electrical.cpp* class contains logic to create a linked list holding the electrical data associated with the weapon.

3.1.1 - Default Constructor

As the name implies, this unit test will test the effectiveness of the default constructor and verify the values created are default.

3.1.2 - addNode

This function will test the functionality of adding a node to the linked list. To ensure total testing capacity, we will implement two paths:

- Normal functionality - This pertains to the normal function of AddNode, which is adding one electrical node at a time;
- Failure functionality - This is primarily to test the error handling of the function. This type of test includes providing the incorrect type or value into the function.

3.1.3 - freeLL

This test will verify that the linked list is being freed correctly. Since the function's primary use is to destroy the linked list there are only two cases to be tested: freeing a linked list containing a single node and freeing a linked list with multiple nodes. These cases mimic the actual uses in the main project.

3.2 - Status Class

The *status.cpp* class is responsible for taking in controller data and converting it to usable class variables.

3.2.1 - loadData

This function is responsible for converting a deserialized message into status data. Due to its importance, we are testing this function multiple times with various messages:

- Normal functionality - This case sees a correctly formatted message imputed into the function which should create the correct status data.
- Incorrect input - This case is the incorrect input being entered into the function and can be categorized as the following:
    - Too many/too few items inside the input message;
    - Incorrect type of an item.

3.2.2 - loadVersionData

Like the loadData function, loadVersionData takes in a message and outputs status data. This function is called just after a connection to the simulated controller has been established. We are testing this function the same way as loadData:

- Normal functionality - An expected input is passed to the function and an expected output is received.

- Incorrect input - A wrong input message is passed. Again, this can be categorized as the following:
    - Too many/too few items inside the input message;
    - Incorrect type of an item.

3.3 - Event Class

Testing the *event.cpp* will be straightforward and similar to testing the electrical class. The Events class will create a linked list to hold the events and error messages.

3.3.1 - Default Constructor

This test case will ensure that the default values are being assigned correctly. Specifically, this test sees a creation of the linked lists and verifies that the values associated with the linked list are default.

3.3.2 - addEvent And addError

These test cases are combined into one section because the approach is not different from each other.

In addEvent's test case, a linked list object is being created and the specific event data is added. The data added is a random ID, a timestamp, and a test message. Once addEvent is called, we verify that the head and tail nodes are not null, and that the data associated with the event object is what we declared it as.

The test case addError is similar to addEvent. Event data is included such as the random ID, timestamp and message but also error specific data such as the number of cleared/uncleared errors. The test case verifies that the function worked as expected and adds the data to the linked list object.

In both test cases, after the values were verified, the linked list object is deleted to avoid a memory leak. Because this function takes in an input, we will be testing bad inputs the same way as the status and electrical classes. This includes too many:

- Inputs that have too many items;
- Inputs that have the incorrect type.

3.3.3 - freeLinkedLists

Similar to electrical class' freeLL test case, this test case will add both an error node and an event node to the linked list object. Now that the linked list has actual nodes, the test will verify that the head and tail nodes are not null, indicating that the linked list contains nodes. The test case will call the freeLinkedList function and then verify that the head and tail nodes are now null.

3.3.4 - clearError

The clearError will clear an error inside the linked list, given a valid ID. The test case will create an uncleared error message and call the clearError function with the uncleared error's ID. The test case will then verify that the node's cleared variable returns true. Since this function takes an input, we will be testing with bad data. This includes too many arguments, too few arguments, and incorrect data types.

3.3.5 - loadEventData/loadErrorData

Similar to the load data functions in *status.cpp*, these functions will add the data located inside the linked list nodes and create a new event or error node. Since this function takes in input, we will be testing bad input as well. There are two paths to test these functions:

- Normal functionality - function is working as it would in the real execution of the software;
- Bad input - the function is given bad data as the input that includes the types below.
    - Incorrect message - message does not have all items needed
    - Incorrect type - message has required items, but items have mismatched types

3.3.6 - loadEventDump/loadErrorDump

This function handles the creation of a data dump, used to capture data being transmitted but not accepted by the user computer. Like the functions above it, it also takes input in the form of node messages. To testing the error handling of the function we will be conducting the following types of tests:

- Normal functionality - the function is given inputs that it expects
- Bad input - the function is given either an incorrect message or the message has an incorrect type.

4.0 - Integration Testing

Integration testing is a crucial phase in the software development life cycle, where we must focus on verifying the interactions and data exchanges between major modules and components of our system. The main goal of our integration testing will be to validate that different modules in our program can interface with each other to achieve the desired functionality. The most important modules that we will be looking at are the Data Display Module (DDM) and the Controller Simulator (CSIM), since it is a requirement that these two must communicate with each other via RS422 serial protocol. Our testing will also ensure seamless integration of saving and retrieving data from a log file. All of these backend components will interact with the graphical user interface (GUI) to display data, so we must also verify that the user's interactions with the GUI properly trigger the correct signals and their respective slots.

4.1 - Serial Communication

First, we must fully test the interface between the data display module and the controller simulator. Our custom test harness will:

1. set up the serial connection between two ports on the user's computer,
2. mock typical controller behavior by generating messages to send,
3. and monitor the expected outcome.

This test will verify that the data display module correctly receives event and error updates from the controller simulator and is properly processed by the DDM. Since all incoming messages are expected to have a leading identifier character, we must test every possible scenario and response the DDM should have. The enumerated values are defined in our constants file (constants.h), where each identifier should cause the DDM to do something different. These include an event dump, error dump, incoming electrical data, a singleton event, a singleton error, incoming current status data, an incoming clear error message, a listening message, a begin message, and finally a closing message.

4.2 - File System Architecture

Another important part of our system that we must test is the logging of event data to a log file and the retrieval/viewing of historical data dumps. The test harness for this component will:

1. load an existing log file's data into the respective linked lists,
2. verify that data,
3. manipulate the data in some way,
4. and then save new log files on the system.

At the same time, we will also verify that the number of autosaves does not go over the user-specified capacity. The intended behavior is that the program will overwrite the oldest autosave file when this occurs. To validate the overwrite capabilities, the harness will attempt to save N + 1 log files, where N is the maximum number of log files allowed, specified by the user (default 5).

4.3 - User Interface Interaction

Our program allows the user to interact with the GUI in many different ways. There are buttons that trigger certain actions, filters the user can set, settings that can be modified from their defaults, and more. Most of these interactions will trigger a "signal" in our program, which tells our backend to do something with a method we created referred to as a "slot". Each signal has a respective slot that should be run when the user does something in the GUI. Our integration test for the GUI will follow the most typical flow the user will might follow while running the program:

1. validate incoming messages are displayed in the events tab (including counters),
2. change filters for the output and validate the display again,

3. verify functionality of the download and upload log file buttons,
4. validate status information is updated as messages are received in the status tab,
5. validate that electrical data has been dynamically loaded in the electrical tab,
6. modify serial connection settings,
7. and verify functionality of the “save as default” and “restore defaults” buttons.

5.0 - Usability Testing

Usability testing is measuring how well an end user can navigate through a user interface (UI). Commonly, usability testing is done by seeing how a normal user interacts with the application. This allows the development team to identify the blockers that an end user is running into or improve the application based on user feedback.

The main characteristics of this project and its intended end users include:

- Background of the user;

Northrop Grumman identified the end user of this project to be service technicians who are conducting repairs or maintenance on the weapon. It is important to note that the end user will be supplied a user manual.

- Novelty of the application;

This application, while building off of an existing procedure Northrop Grumman engineers follow, will be the first software-based solution intended for maintenance technicians.

- Impact of bad design.

The impact of bad UI design will affect the efficiency of the maintenance process. If a commonly used item is difficult to reach or is placed in an improper place, this can affect the job of Northrop Grumman’s customers.

5.1 - Connection Page

The connection page is the jumping off point for the users’ interaction with the application. Here, the user will enable the connection to the controller. Based on a user with an average level of technical knowledge, the team will conduct the following steps to prove usability. Given prior knowledge of the correct ports to select, the user will be able to do the following:

1. Connect to the simulated controller.
2. Change the connection settings.
3. Revert connection settings back to default.
4. Update default connection settings.

5.2 - Events Page

The Events page is the page users will be greeted with upon starting the application. It contains the log file functionality, displays, and filters the events and errors. Based off of a user with an average level of technical knowledge, the team will conduct the following steps to prove usability:

1. Once connected to the controller (either simulated or wired), the user will navigate to the Events page from the tabs at the top of the application.
2. The user should be able to complete the following steps without knowledge of how to do them:
    a. Set the log file folder;
    b. Manually save a log file;
    c. Filter the events and errors

5.3 - User Settings

The user settings page is where the user can update and change application settings. These include colored output, connection timeout duration, and the limit of auto saved log files. To fully test the application’s usability for this page, the team will conduct the following steps:

1. From the starting page (Events), navigate to the user settings.
2. Change the Colored Events Output boolean to false and check the Events page.
3. Next, navigate back to the user settings page, and change both the connection timeout duration and auto log file limit.

4. If the user is connected to a physical device, ask them to unplug the controller device. The timeout duration will expire at the set time.
5. Now, ask the user to start and stop the controller. This will generate auto saved log files. The count of saved log files will never exceed the user set variable in user settings.

6.0 - Conclusion

In conclusion, the comprehensive testing strategy outlined in this document aims to ensure the delivery of a robust, error-free, and highly usable software product. By implementing unit tests using the QT Framework's QTest library, we are poised to thoroughly evaluate the functionality and error handling capabilities of critical code sections across various classes.

The unit tests, designed for each class, will assess both normal functionality and error-handling scenarios. Each test case is crafted to simulate real-world usage and potential edge cases.

Integration testing is used for verifying the interaction between major modules and components of the system. With a focus on serial communication, file system architecture, and user interface interaction, we aim to ensure that different modules interface effectively, data exchanges occur accurately, and the graphical user interface responds appropriately to user interactions.

Usability testing is integral to guaranteeing that the software meets the needs and expectations of its end users. By simulating user interactions with application pages such as the connection page, events page, and user settings, we aim to identify potential usability issues and improve the user experience.

Through the use of unit tests, integration tests, and usability tests, we are confident that our software development process will result in a maximally error-free, functional, and highly usable product. This comprehensive testing approach not only ensures the reliability and stability of the software but also enhances its usability and user satisfaction, ultimately meeting the objectives set forth by Northrop Grumman.