1. Introduction

The United States Coast Guard (USCG) oversees 296,000 nautical miles of sea utilizing the Rescue21 system for Search and Rescue missions. General Dynamics Mission Systems (GDMS) assists the USCG by creating and maintaining the Rescue21 system. The Rescue21 system consists of roughly 250 remote fixed facilities (RFFs) which are strategically placed around bodies of water to pick up distress signals over a very high frequency. While the Rescue21 system works as designed, GDMS requested an additional system that collects a variety of data from the RFF sites to determine the cause of radio frequency (RF) interference, predict damage from severe weather, and improve engineer safety by using live weather data to know if it is safe to perform maintenance. An electrical engineering capstone project created a device called the Site Weather and Power Recorder (SWAPR) which would sit on an RFF site and record weather conditions and antenna power. This project solved the data collection problem, but the data was hard to read and inaccessible outside of the RFF sites. Our project will improve the SWAPR solution by collecting, storing, analyzing, and displaying collected data in graphical interfaces through a secure website. Through easily readable graphics and access to SWAPR data, GDMS and USCG will be able to detect, schedule, and predict maintenance requirements to reduce outage times, decrease maintenance costs, and improve safety for engineers.

A project of this importance needs significant software testing before we can even begin to implement it into General Dynamics current system. Software testing is when you put a program or small piece of a program through tests to see if it works as expected. The most common of these, which we will use on our own project are unit testing, integration testing, and usability testing. Though General Dynamics will be doing their own testing and refactoring of our solution, we will be creating a series of unit, integration, and usability tests to ensure our project works as intended under a wide range of conditions and meets our project requirements.

Unit testing is when small pieces of code, or units, are put through tests to ensure they are working properly. Unit testing will be extensive but will only be done on our Website Subsystem. We will test if the Website properly receives the data objects from the Database in the correct format. There will be tests for each view in our Website Subsystem: List View, Map View, and Historical View and check that they display the correct values from the database. We will test if graphs are visible with historical SWAPR data.

Integration testing can be summarized as testing the interactions between interconnected systems or programs to ensure proper functionality. Considering the complicated nature of our multiple subsystem solution, we have integration tests for every Subsystem to ensure they all work well together. We will be using both Big Bang and Hybrid Integration testing techniques which we will describe later in our document. For the Simulator we will test if it properly creates data imitating the SWAPR device’s output and test if it sends the data to the Reader Software.

We will check if the data inside the database matches that which was sent by the Reader. This tests the ability to store and serve SWAPR data in a secure manner. Next, we will test if the Website is retrieving the entries from the Database. This will show we are able to take data from a SWAPR device and send it securely off-site. There will be a test for Orchestra, which emulates the entire SWAPR device network for stress testing. Our solution currently has a way to notify an operator when there is a problem with an RFF site. The connections of our notification system will be tested.

For Usability testing, we will look at our solution as if we were a basic user. This testing is limited to the Website frontend, as that is the only part the user has access to. This ensures that our team has established a secure website environment with authentication. We will test the List View to ensure a specified number of SWAPR devices are shown. For the Map view, we will test if points are placed properly and visibly on the map to correspond to SWAPR location. We will test the ability to export data from the database as a .csv file. Then we will check that the user can see and understand the Wind Rose and Line Graphs for a SWAPR. We will test if notifications are viewable, and closable.

We have a greater amount of integration testing and usability testing because it is vital that the weather data produced and received is accurate and in the proper format. We have fewer unit tests because the core function of our project lies in the interconnection of our Subsystems rather than standalone functions. In the following sections, we will go in further detail on our unit Testing, Integration Testing, and Usability Testing and explain the purpose of each.

2. Unit Testing

Unit testing is when small pieces of a program are tested individually to see if they are functioning as planned. In our case we will be testing small functions such as ones that create randomized values for our weather information, and other small tests that ensure the proper formatting of our output is met. We will possibly use unit testing C# in .NET Core using dotnet test and xUnit to create our solutions and tests, but we do not have extensive experience with those yet so we will most likely create our own unit tests, since we have a clear understanding of our project and its inner workings already. As previously mentioned, our unit testing will be extensive for the Website Subsystem as is essential the data is produced properly and in the intended format. Ensuring this will make our integration testing and interconnectivity between subsystems much easier. The Simulator and Reader Subsystems are unnecessary to provide unit tests for. The function of these systems do not face variability in input or procedure. The generation of data in the Simulator is generated based on boundaries defined by our team and are deterministic meaning we always know what values will be generated and the appearance of the data output by the Simulator. The Reader Subsystem handles a standard and straight-forward process when transmitting data over a comport. Because the Reader uses the RS-232 protocol, a standardized and simple procedure, we do not consider it necessary to unit test this process. We will break down the different unit tests for the Website Subsystem, and the lack of unit testing for the Simulator, Reader, and Database Subsystems.

2.1 Simulator Unit Testing

In introducing unit testing, our team has stated the unnecessity in unit testing the Simulator Subsystem code. Our team evaluated the change in states and the behavior in Program.cs and Entry.cs; we concluded the methods and state changes are hands off for the user therefore there is no benefit from creating unit tests. The data generated by the Simulator does not involve user input capable of impacting the operations or outcomes of this subsystem.

2.2 Reader Unit Testing

Similarly, the Reader Subsystem code also does not require unit testing. Our team evaluated the relevant files to this subsystem and determined that the objects and methods do not need unit tests created for them. The methods for sharing information over the com0com RS-232 protocol connection do not require testing to evaluate accurate data transmission, however this will be a necessary part to perform integration testing.

2.3 Database Unit Testing

Our Database Subsystem does not need unit testing. Our team assessed the Database Subsystem and determined that data transmission was our main concern with this subsystem so we will solely focus on integration testing of the Database Subsystem.

2.4 Website Unit Testing

The Website Subsystem involves all the front-end interaction the user experiences. Testing each individual component of the data access, views, and notifications is integral to guaranteeing a robust and dependable system. The List view does not provoke any opportunity for erroneous values, because it depends on the data in the database as well as the effectiveness of the data retrieval method. This view will however be important for larger, modularized testing.

The Data Manager is vital to the operations of many of the components. To retrieve data to be used in the Website Subsystem the GetDatabaseEntries() must be called. Specifically, the variant that takes an integer and two DateTime objects as arguments. The integer represents a site identification number which is necessary for identifying sites. The DateTime objects are used to specify a set of entries based on a range of time with the first DateTime object being the starting time and the second being the ending time. The DateTime value can be converted to from both a string or a long data type variable. The usage for this function requires the DateTime objects include the Year, Month, Day, Hour, Minute, and Second. The Year is in the range of 0001 to 9999. The Month is in the range 1 to 12. The Day is in the range 1 to the number of days in the month. This is dependent on what month as the total can vary. The Hours are in the range 0 to 23. The Minutes are in the range 0 to 59. The Seconds are in the range 0 to 59. This is the valid partition our unit test will test for. Values that are negative or 0 in the case of the Year are considered invalid as are values that are over the stated upper bound. The DateTime object as the argument protects the method from having to handle erroneous input such as integer or floating point types. The DateTime is distinct and structured in a specific way. The Id parameter may be any integer in the range of 0 to 2,147,483,647 because of the memory capacity of an int data type. The Id must be a valid number for a site in the database. For this unit test, sites will be used with known Id’s to accurately test this method. We will also attempt to use an Id that is not in the database to verify nonexistent data cannot successfully be retrieved. This will give us three partitions for the case of a valid Id and a fourth partition to test an invalid Id.

To update notifications, the method SetNotificationStatus() is available for use by an Admin. This method enables the display of select notifications for all user roles. The first parameter is the notificationId. This is an int data type that represents a distinct value for the notification message for reference. The second parameter is a boolean value. The given status will be inverted by the method then enabling or disabling the display of the notification in the notification list. The notificationId must be a valid Id for an existing notification in our database.

A unit test will be created to test a valid Id, a known Id found in the database table, as well as an invalid Id to verify a nonexistent reference is appropriately handled. This will be what consists of our two partitions. One with a valid Id and one with an invalid Id. It is unnecessary to test the boolean argument, because it is in no way capable of being erroneous.

In the Website Subsystem, the views we feature are of high importance. Our team will need to test user interactive components in our system to evaluate the accuracy. To accomplish these operations, a data manager object is used to handle data services and the accessing of data.

The Map view provides a geographical display with site markers signaling the location of SWAPR devices. When clicking a marker on the map a redirect occurs that loads the historical view page for that site. The method NavigateInNewTab() has a single parameter, an Entry object. The Entry object contains a site Id value used to determine the page to load. A unit test will be created to test that an entry object can successfully load the historical view page for the given site. One valid partition for the case where an existing site Id is referenced and an invalid partition for the case where a site Id is given for a nonexistent site. Due to the entry object only using the site Id, there is no concern for other erroneous valued attributes in the Entry object. Additional to the map site marker locations being visible, the color of the marker will be shown. The color is associated with the status of the site. The color is set by a status variable where it can be: Green, Orange, Yellow Orange, Yellow, or Red. To determine this status, a function GetColor() is used. This function has a single parameter that is an Entry object. The Entry provides a status attribute and is checked to determine and return the status color of the site. A unit test will be necessary to evaluate the case where a passed in Entry object with a valid status and the case where a passed in Entry object with an invalid status. If the status code is not one of the listed codes, then the color should not be set and the exception should be handled.

The Historical view features a line, bar, and radar as available graphics to visualize entries from the sites in the database. All these views utilize a number of operations to perform their functions. The method CreateDataSet() is used to take data from the database and transform it into a usable list for the website. It has four parameters: an int data type datasetType, string data type label, Color data type colorType, and a list of doubles data. The datasetType parameter is used to determine the type of graphical view whether it be line, bar, or radar. The label parameter is used to provide a display name on the graph. The colorType parameter specifies the color of the line or bar depending on the graph type. Finally, the data parameter is a list of the values for the points that will be placed on the graph. The label and colorType are considered arbitrary in the context of this unit test, because they do not impact the outcome of the graphs. The list of doubles is important; however this method only passes the data along for a later operation. The datasetType must be one of the three mentioned values otherwise it is invalid. This unit test will include four partitions with one partition handling an invalid datasetType value and the other three partitions handling the three valid datasetType values.

The three Historical view graphs also utilize a method CreateTimeDataSet(). This method takes the same parameters as the CreateDataSet() method excluding the list parameter data. Instead of a list of doubles, this method takes a list of TimePoint objects. The list contains data for the timestamp of each value placed in the graph. The same operations are completed for this method otherwise. A list of the data is returned with the set label and color.

2.5 Orchestra Unit Testing

The Orchestra Subsystem does not require unit testing due to its purpose. The Orchestra is a batch script performing a set sequence of operations to run the simulation of the SWAPR data and the reading of the data to the database. With no input or test cases to handle our team will prioritize testing this subsystem in integration testing.

3. Integration Testing

For our integration testing, we will be using a combination of the Hybrid and Big Bang methods, which will accurately assess the work we have done so far.

Hybrid integration testing is an approach used to test a system of modules by testing the communication between each connected module. The test is structured in three layers: the main layer, top layer, and bottom layer. The approach utilizes two other approaches, the top-down and bottom-up approach. The top layer represents the top-down approach, testing from the highest level and down in the system, and the bottom layer represents the bottom-up approach, testing from the lowest level and up in the system. The main layer is the central component for communication in the system. The goal of the hybrid integration approach is to test a connected and completed application's working system components in the early stages of development.

Big Bang Testing is an approach to integration testing in which all the components or modules are brought together simultaneously and then tested as a single unit. During testing, the integrated set of features will be treated as a single object. The integration procedure will not run unless all the components in the unit have been completed. This ensures that our system works as a whole and will be able to detect connection issues between the main features.

The Hybrid integration approach will be implemented to test the communication in our system from both sides of the database. For our usage of this approach, we consider the main layer to represent the database as it is central to the functionality of our application. The top layer represents the website, because it is the interface for the user to communicate with the database. The bottom layer represents the simulator and reader, because these two subsystems communicate with the database to provide it data. The database is at the center of our system because it is the storage and access point of all information used by the system. The database contains data tables for the notifications, site entries, and sites. These database tables are populated using generated data from the Simulator that has been sent by the Reader and received by the Amazon Web Services (AWS) Simple Queue Service (SQS) lambda functions. Our database schema defines these data tables for usage on the website. The website will utilize data retrieval functions to pull data from the database and generate pages.

The Simulator will begin by generating random numbers within valid and invalid ranges. The range values we will generate are four antenna power ranges, humidity, temperature, rainFall, wind speed and direction. The simulator will create data every 5 seconds to send the reader. In the functions GenerateValidData and GenerateInvalidData, the System.random class will be used. The System.IO.Ports.SerialPort class is used to send the data to the Reader after it has been generated. We'll use the System.IO.Ports.SerialPort class in the Reader software to accept data from the virtual com port. When called on the SerialPort object, the function Open() establishes a connection to the Simulator. The data will be sent to the Reader where the reader will serialize the data into a JSON string. The string will be printed to the console to check for

the correct values and serialization happens. We will compare the generated simulator data to the json strings for accurate code

The Reader will then send the data to be used in AWS. Using the AWS.SQS C# class, the Reader will send the message to the Amazon Web Services (AWS) Message Queuing Service (SQS). AWS SQS will then store the data in a queue. Once in the queue, AWS lambda functions will be alerted of new entries, pull them, and utilize them to generate SQL queries to add rows to the Database's Entry and Notification tables. The SQS Lambda function is expected to take and deserialize a JSON string into an Entry object. The Entry object provides the site identification number with all of the weather and power information at the given timestamp. This information will then be evaluated for its status and a Notification will be generated if the status is anything other than green. To verify the outcome, we must input an established JSON string into the Lambda function and evaluate the database tables to confirm the new entries are identical to the expected outcome. The test harness in this module includes creation of database entries input as a JSON string. By verifying input into the Lambda function produces distinct entries in the database we can assert that data placed in the AWS message queue will be accurately stored in the database.

With a database populated with site, entries, and notifications, the website is then integrated to communicate with the database. The Website Subsystem features the website and the multiple graphical user interface elements. The Historical, List and Map Views, and Notification components will all be tested and evaluated using pulled entries from the database. The creation of a notification, as we previously overviewed, is completed using the AWS SQS Lambda functions and message queue. The data loaded into the database is reviewed and established by our team for the purposes of this integration testing. The notifications are stored in a notification datatable and are accessed from website functionality. It is necessary to verify all notifications pulled from the database are identical to the data stored in the database. The test harness includes the created notifications and entries stored within the database notification and entry datatable. This is the same case for the Historical View. The Historical View pulls the entries from the entries datatable.

The website uses the various weather and power fields from a range of entry dates and visualizes these in either a line, bar, or radar graph. The chart will have multiple tests to ensure proper display of these graphics. In this integration testing, viewing these graphs through the website pages will be done to evaluate the accessing and implementation of the entry data. The data, like the notification testing, will be previewed to certify our team knows the expected generated graphs. The generation of these graphs is specified by various parameters defined by a user. This means the Historical View must be carefully evaluated to guarantee the user entering graph specifics can produce the desired graph. With accurate graphs, we will have verification that our website is able to pull data from the database correctly and use this data to generate graphs. For the test harness, the Historical View requires a user to select the parameters of the graph which adds one extra step of testing. The test harness also includes the created entries

stored within the database entry datatable which must be retrieved using the input parameters. In addition to the Historical View, the List view must also accomplish a similar task.

The List View does not utilize user interaction like the Historical View, but instead implicitly accesses the database to retrieve the latest entry from all sites. The List view will be tested to determine if the access and retrieval of data from the entries works appropriately while also verifying every available site and its latest entry is provided in the form of an informative panel. To ensure this, as we will do with the other views, the entry data in the datatable will be known and verified in the website List View page to ensure the known sites are visible with accurate entry information. The test harness for the List View includes the latest created entries stored within the entry datatable from each site. Finally, the Map View will be tested in a similar fashion to the List View with an additional check.

The Map View accesses and retrieves site and entry data from the datatable, but only a certain amount of information is relevant to this view. The purpose of the Map View is to display the sites on a map of the United States based on the declared latitude and longitude of the physical site in the real-world. In addition to showing the location of the sites, a color system is used based on the status of the site. This status is determined at another location of the system, but the importance here is to give the user a clear understanding of the operational state of the site. The database site data retrieved will need to be tested to verify they utilize the latitude and longitude data to accurately place the site marker on the map. To color the marker the accurate operational status color, we will need the status of the entry being evaluated which should be the latest entry from the site. The Map View will need to provide one additional feature, interactivity from the user through the website GUI, that must be tested. By clicking on a marker the site must use the site identification number to determine the URL path to send the user to. This page the user is redirected to must be a page of the Historical View already set to the site Id and ready to generate a desired graph. Verifying this will be important to not only ensure the Map View displays accurate information from the database, but also guarantee the click of a site marker on the map redirects the user to the appropriate Historical View page. The test harness includes the stored sites within the database site datatable. By verifying these entries utilized throughout the website are accurately accessed and pulled to the website pages, our team can confirm the website retrieves correct data from the database in both the case of the site datatable as well as the entry datatable. This is integral for our system, because the primary purpose of the website is to provide methods for viewing information from the database.

Our project's Orchestra Subsystem will be used to test the system as a whole. This subsystem enhances the simulator and reader subsystems' capabilities by allowing multiple instances of each to be generated. Put another way, the Simulator Subsystem and Reader Subsystem pair represents a single SWAPR device. A simulator and reader pair will connect to the database using the reader software and send the SWAPR device data generated by the simulator to AWS. We can scale this up to hundreds of SWAPR devices using the Orchestra. For a full system test we will use the Orchestra to create 250 RFF Sites using the Simulator

Subsystem and Reader Subsystem pair to populate the data for each SWAPR device. The pair will be continually running as the rest of the process runs; over time, we will be testing the data generation and storage part of our system to ensure that it can handle the heavy load of putting data in the database while also being able to give data to the website. Next we will discuss how we will implement usability testing with our system.

4. Usability Testing

Now that we have explained our integration testing, we will explain usability testing and our plans for carrying out our tests. Usability testing is the testing of the experience of an end-user with the software from an outside perspective. The hypothetical end-user has no inside knowledge of how code works other than functionality built into the user interface. The goals of usability testing are to ensure that the integrated systems are functioning correctly from the end users standpoint and to iron out any bugs that occurred during the integration of our system. Usability testing is done by having a hypothetical user carry out a number of operations to ensure that the functionality that our system should provide is being carried out in the correct and appropriate way.

For our usability testing, we envision two different usability tests, from two different users. One user will be a standard user, and the other user will act as an administrator. This will be done because our system has two main roles which are the user and administrator which have very slight differences in the permissions they are given. This will in turn slightly affect the way that functionality is provided to the two roles. The main difference is that the administrator has access to the user and notification management page where they can manage the users and notifications in the system. Besides this difference, the administrator and user roles are practically the same and can do all other functionality on the system. We will split the testing into two parts, one with the user role and the other with the administrator role. The testing will be done on a visual confirmation basis where we do an action and confirm that the result is correct through manual checks. For example, we will open the list view and confirm that the data being shown is done in the correct manner with only the latest entry being used from each site. We will manually confirm this by having manually entered data for each test that way we can be sure that the results are correct. This will allow us to have more control over the results making it easier to confirm proper functionality. Once we have confirmed that the testing is done correctly with static data then we will simulate the network and confirm that nothing weird occurred. If nothing weird occurs then we know that the system functions correctly.

To test the system, a member of our team will first register an account and login as that new user. They will then access each page of our website. This includes the list view, map view, historical view, and notifications. When the user accesses the historical view, they will also test the csv exporting functionality by downloading the data used to create the historical view. The user will then attempt to access the two restricted pages of the website being the admin account management and admin notification management pages. Both pages should fail to load for the user with a message saying that they need to be administrators to view the page.

We will have a second team member act as an administrator. We have preconfigured a username and password in our code and they will login with the admin username and password. Then they will access the list view, map view, historical view, and notification pages. They will

test to make sure they can see each of the graphs properly and that data exporting works properly on the historical view. To test the administrative abilities, our team member posing as an administrator will access the admin account management page and will be able to view all existing accounts and modify them. They will test deletion by deleting an existing non-administrator account. They will also access the admin notification management page where they will be able to view all notifications and disable notifications and ensure they are correctly disabled by viewing the change in the database and admin notification management page.

We plan on doing the usability testing over a one week period that will happen during the week of UGRADS which starts April 18th. By doing the testing at this time, we will know what we can show to the general public while displaying our project as well as making our final list of tasks to complete by the end of the semester.

5. Conclusion

Overall, we started with an introduction of Unit Testing, Integration Testing, and Usability testing and we broke down each of those into the designated subsections. We discussed how these tests would verify the validity of our user requirements. Our Software Testing Plan is organized in a way where the more critical parts of our solution are emphasized. To exemplify this, we have more extensive integration tests and usability tests than unit tests because the interconnection of subsystems in our solution is absolutely critical to work properly. All Unit Tests are explained in detail, in their own subsections. We described our method for Integration Testing which will be a combination of Big Bang Testing, and Hybrid Approach testing since the majority of our functionality is based around our database. We are currently finalizing our code, and after that we will commence our testing using aspects from this document. Once we complete the tests explained throughout our testing plan, we are confident that our project will be highly functional, usable, and be as error-free as possible. We hope to have our testing done by UGRADS which will happen on April 22nd. By doing this, we allow ourselves time to fix any issues that we discover as well as having a detailed understanding of what works in our system for presenting to the public at UGRADS. Overall, we are confident in our ability to have a completely working and tested solution to give to our client by the end of the semester.