Team Lev Tours

Sponsor: Dr. Michael Leverington

Mentor: David Failing

Members

Erik Clark

Kyle Savery

Alexis Smith

David Robb

Ariana Clark-Futrell

Software Testing Plan

Date: 3/26/2021

Version: 1.0

Overview

This document provides detailed descriptions of our team's plan to perform unit, integration, and usability testing on our software product for Dr. Michael Leverington.

Table of Contents

1. Introduction

In the field of computer science, mobile robotics are capable of increasingly complex tasks, and the necessary hardware for these tasks has become far more accessible in recent years. As a result, the costs associated with these materials have also decreased, allowing for a greater number of individuals and organizations to take part in the research and development of mobile robotic platforms. Most significantly, educational organizations can give students the opportunity to interact with robotics either from a mechanical or programmable standpoint. To take full advantage of mobile robotics, one of the main goals of our project, “Thirty Gallon Robot Part III, The Smiling Tour Guide”, designed by our sponsor Dr. Michael Leverington, is to demonstrate that an institution's budget for robotics can be further reduced, and used more effectively, by utilizing inexpensive materials. For Dr. Leverington’s robot, one of the most inexpensive components is the thirty gallon barrel that all of the other components are built around. When this project is complete, the robot will be an example of how instructors can get students involved in the world of robotics, which could lead to an expansion of the field.

Beyond computer science the industry of robotics is involved with numerous fields such as mechanical and electrical engineering, medicine, agriculture, and manufacturing. In 2019, the global robotics industry was valued at $62.45 billion [1]. A subset of this industry that is of particular relevance to our project is known as mobile robotics. This field is responsible for creating robots that can move in 3 dimensional space without the need for human assistance. The market size for mobile robotics was approximately $9.34 billion in 2018 [2].

The Thirty Gallon Robot project is in its third year of development. The first team created the foundation for Robot Assisted Tours (R.A.T) by programming the robot to respond to commands from an Xbox controller. The second team built upon R.A.T by creating a way for the robot to generate its own maps by navigating around the building. A stretch goal for the second team was to develop a framework for Wi-Fi localization capable of directing the robot in a building, but due to the COVID-19 pandemic they were unable to test their design within the engineering building and so that development was not completed at the time. Therefore, this task now falls upon our team. Wi-Fi localization will allow for the robot to navigate on its own throughout a building without the need for an Xbox controller. Requiring user control defeats the robot’s purpose of being an automated tour guide.

To complete this project’s third stage of development, our team consists of 5 members, as well as a mentor and a sponsor:

- Dr. Michael Leverington: Team Sponsor
- David Failing: Team Mentor

- Erik Clark: Team Lead
- Kyle Savery: Architect, Developer
- Ariana Clark-Futrell: Communications, Recorder
- David Robb: Release Manager, Developer
- Alexis Smith: Developer

Now that our tasks are nearing completion we have developed a test plan that aims to make sure our project has the proper functional and non-functional characteristics. We have broken our testing up into three phases: unit, integration, and usability testing. This will allow us to test every aspect of our product from individual functions and modules to how easy the product is to use for non-developers.

2. Unit Testing

Unit testing is a vital component to the development of any software system. A unit test is designed to isolate an individual unit of the software and test its robustness. These tests administer carefully designed inputs that cover the edge cases (or equivalence classes) for the values that a specific unit is expected to handle without relying on other components of the software. For a very rudimentary example, if there is a function that expects an integer as an input then the unit test for that function could test the output with a negative number, zero, and a positive number. The function output is tested against the correct output which has been calculated ahead of running these tests. This is one of the major benefits to unit testing where the programmer only has to write the unit tests with associated edge cases and correct outputs once, and then the code can be tested repeatedly to ensure any changes are not producing incorrect output.

To test our software product, our team will be utilizing the Python ‘unittest’ module. This module allows us to create tests and run them all at once for every unit we designate to be unit tested. This module will output which tests fail and then we will be able to easily see which edge cases our code is not handling properly.

2.1 Backend Units

The backend portion of our software is split up into building navigation and Wi-Fi scanning. Building navigation encompasses how the data related to a given building is stored, edited, and used to calculate routes throughout that building. The Wi-Fi scanning component is responsible for retrieving Relative Signal Strength Indicator (RSSI) values for each nearby Wi-Fi access point. These values will then be used to estimate the device’s position within the building. The respective unit tests for building navigation and Wi-Fi scanning are described in the following sections.

2.1.1 Building Navigation and Storage

Our software's data structure which holds the data related to a given building, consists of each floor on that building, where each floor holds all the nodes (or locations) that have been set up. The three major operations that our unit testing will be focused on are editing this building data structure, calculating paths from any two points, and storage. These operations are made up of several smaller units that will each receive their own unit test that evaluates the details as laid out below.

- Editing the Building Data Structure
    - Adding Nodes: When adding a new node to a building, the node is actually added to the floor first and then the floor can be added to the building structure. Once the node is added, the adjacency matrix that represents the edited floor must reflect this change properly. That is, the matrix must have added a new row and column. Each entry in this new row and column should be initialized to a representation of infinity, indicating this node is not connected to anything. The node's distance to itself must be set to zero. There is also a list which holds the names of all added nodes (not the *Node* objects themselves), the new node's name must be appended to this list. The order of this list of names links up with the adjacency matrix to identify each row and column with their respective nodes.
        - Edge Cases: Adding a node with a name that does or does not already exist. If the name already exists, this procedure should exit and have no effect on the building structure. If the name does not exist then the new node should be correctly added.
        - Erroneous Input: Trying to add a node that is of an unexpected type, specifically in our structure a node needs to be of type *Node*. Similarly, adding a node with a name that is not of type *String.*

    - Removing Nodes: To test that a node can be removed properly, the adjacency matrix must have taken out the corresponding row and column. To complete this the index of the node's name within the list of names (as described previously in "Adding Nodes") will be used to identify the entries within the matrix that must be deleted. Then the list of names will remove the deleted node's name.
        - Edge Cases: Removing a node with a name that does or does not exist. If the name exists then the node should be removed. Otherwise there should be no change within the matrix and list of names.

        - Erroneous Input: Attempting to remove a node via a name that is not of type *String*.

    - <u>Adding Edges</u>: To add an edge in the building's graph, all that is needed is the names of two nodes that are present in the system and the distance between them. The adjacency matrix then updates the corresponding entry with the new edge distance.
        - Edge Cases: Adding an edge to two nodes where neither or at least one do not exist.
        - Erroneous Input: The edge distance can only be greater than zero, so positive numbers, zero, and negative numbers must all be tested. Another incorrect input would be attempting to add an edge distance that is not of type *float*.
    - <u>Removing Edges</u>: This procedure is simpler than adding an edge as the only possible input are the two names of the nodes that are to be disconnected. This is done by setting the corresponding entry in the adjacency matrix to infinity.
        - Edge Cases: Removing an edge between two nodes where neither or at least one do not exist. Two nodes can be connected or not connected when attempting to remove their edge. If they are not connected there is technically no effect as the distance between them is left unchanged and reflects an infinite distance.
        - Erroneous Input: Inputted node names are not of type *String.*

- Pathing Operations: All of the following procedures will be tested using these edge cases.
    - Edge Cases: Graphs that are either disconnected or connected. There will be three types of disconnected maps: a single node is not connected to any others, multiple nodes are separated, and all nodes are disconnected. While the connected map will contain nodes that have one, two, three, or four possible connections. These maps will also span multiple floors.

    - <u>Dijkstra</u>: This function receives a single node's name that exists within the floor's graph. The output will be a list where each element will show the node's name, the total distance to this node, and another node that must be passed through to reach this node. This function will be tested by passing in a node that falls in each of the categories described in the previous edge cases.

- Connectedness: This procedure is used to determine if a floor within a building is disconnected or connected. Only connected buildings should be displayed to the user when choosing a building to tour. This method relies on “Dijkstra” to determine the connectedness of a map. The map is disconnected if for at least one node, in terms of the list returned by “Dijkstra”, the node that must be passed through to reach this node is set to the internal constant NO_CONNECTION.

- Get Path: This function will also utilize “Dijkstra” and expects two inputs, the starting and destination node for a user’s route. The result of this function should be the shortest path between any two points in the building depicted as a list of nodes in the order they must be traversed. Testing will be conducted by passing in start and destination nodes that lie in each category from the edge cases.
    - Erroneous Input: Attempting to get a route on a disconnected map should fail and return false.

- Set Directions: This procedure relies on the assumption that any turn set up in the building will be left or right in order to display human readable directions to the user (when this software is integrated with the robot, turns will be in terms of degrees). Essentially, the purpose of this function is to output the correct direction depending on the orientation of the user. It is assumed the user is always holding the device directly in front of them. A simple example is shown in Figure 2.1.

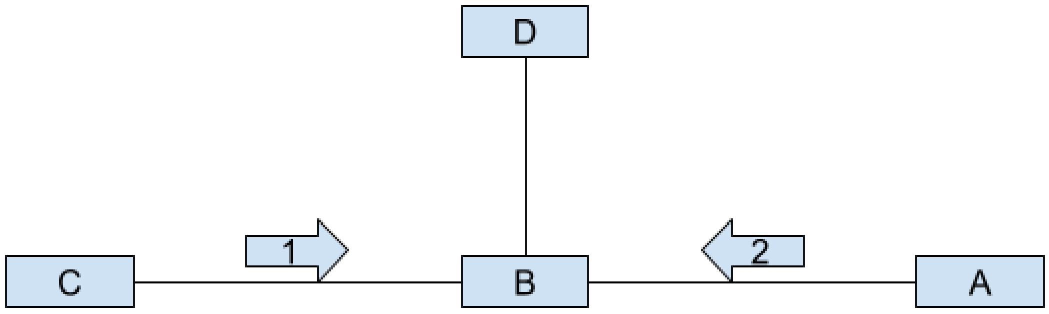

Figure 2.1: Process of outputting directions.

In the above diagram, if the user’s destination is D and they are approaching B from C, then the next direction they receive is to turn left. Whereas if they are approaching from A, then the next direction they receive is to turn right. Testing this procedure is similar to the previous functions and the output of each test will be compared to the expected directions.

- File Operations
    - Saving/Loading: The process of saving and loading a building's data will be tested together. A created building will be saved to a text file and then a new building will be created by loading the data from that text file. If these buildings hold the same information then both saving and loading is functioning as expected.
        - Edge Cases: Saving/Loading a building structure with no data.
        - Erroneous Input: For loading, incorrect input would be any malformed text file. To test this, the function will be given text files with mismatching information (i.e. edges between nodes that are never found or files that are missing expected headers).

    - Get Buildings: This function simply returns the names of all the buildings that the backend user has set up. Set up buildings are stored in folders with corresponding names and all of the building folders are stored in a folder called 'buildings'.
        - Edge Cases: No buildings have been set up or the 'buildings' folder does not exist.

2.1.2 Wi-Fi Scanning

Testing the Wi-Fi scanning module is far more brief than our tests for the building data structure. There are only two units within this module our team deemed necessary for isolated testing. That is, ensuring nearby RSSI values are scanned and stored properly, as well as using a current scan to compare against past scans.

- Scanning: When our software scans nearby Wi-Fi access points, it utilizes a command on the linux terminal called *iwlist*. This command returns a variety of information about each access point and the next step is to parse through this information and extract only the relevant data. In this case the relevant data is the Media Access Control (MAC) address for a given access point and its associated RSSI value which is an integer inclusively between 0 and -100. For Wi-Fi scanning in general, we are unable to write unit tests looking for specific values and must instead rely on a test which checks that for all of our parsed information (the MAC address and RSSI) are valid values.

- Comparison: This procedure compares a current Wi-Fi scan with our stored data to estimate the device's current location. To test this function we will use manufactured data that has an expected outcome and then check to ensure our comparison procedure functions correctly.

    - Edge Cases: Our manufactured input should cover the cases where two locations are equally likely to be chosen or no location should be chosen.

2.2 Frontend Units

A downside to any kind of User Interface (UI) is that it is not very useful to write dedicated unit tests for each segment to be tested. In order to properly evaluate a UI it is necessary to physically test edge cases and erroneous input so that the tester is able to visually see the expected output. Described below are a variety of tests and edge cases for the units of our Graphical User Interface (GUI).

- Switching Frames: In our GUI, there are numerous pages the user must be able to navigate to and from. The most straightforward way to test this process is to check that every button opens up the correct page, as well as checking that back buttons are always available and return to the desired screen (either going back a single page or navigating back to the home screen).

- Manipulating Nodes: When adding a node on the editing screen, the new node should pop up in the correct area and be the only node that is highlighted yellow**.** The correct area is centered right above the entry box for the new node's name. Node names are not allowed to be repeated anywhere within the building (even if they exist on different floors) and so our program must alert the user when a name they enter is invalid. Anytime a node is clicked on its color should switch to yellow to distinguish it from the other nodes which will remain a grey color.

- Passwords and Login: During the process of setting up a new user and password, our program should raise an alert if the username already exists. An error message should also be displayed if the new username and password are not properly added to the system. The system should grant any user with the proper credentials access to our software's administration section.

- Requesting a Destination: When a user inputs a start and destination for a given route, those nodes should be highlighted green and red, respectively.

3. Integration Testing

The goal of integration testing is to bring together components of the overall system and test their interactions in order to ensure that the product as a whole operates as expected. Since most software is developed by multiple people, integration testing mitigates possible bugs and errors in the communication between the modules that have been developed independently of one another. Once our product has been thoroughly unit tested, our team can move onto testing the integration of our building data structure, Wi-Fi scanning module, and GUI. We chose the following tests according to how each of the previous three modules communicates and relies on the others.

3.1 Wi-Fi Scanning and Building Data

The Wi-Fi scanning module communicates with our building data structure in a single way: each established node performs a series of Wi-Fi scans during setup in order to “fingerprint” its location. This process results in a set of data that describes all of the node’s detected Wi-Fi points. The data is then stored within the node for ease of access. To test this interaction, we can essentially reuse the test designed to ensure that a single scan has the correct format (2.1.2 Scanning). After we combine the data from several scans the test will need to be modified to check that the range of RSSI values for a given router (instead of a single value) are contained within -100 to 0. Once the data is verified to be correctly gathered, we will use the procedure of saving a building and then reloading it to check that the building state is equivalent before and after those operations. This will verify the building data, along with the integration of the scanned Wi-Fi data is being stored properly.

3.2 Building Data and Graphical User Interface

Our GUI heavily communicates with the backend's building data structure and expectedly this is where the bulk of our integration testing will take place. The two components must interact in the subsequent ways and each of these interactions must be tested extensively, otherwise our product will not function as intended even if the units themselves have passed all of their tests.

- Loading and/or saving a building selected by the user either for editing or touring.
    - Associated Test: In our GUI, we will create a new building and link each floor to a map image. This building will then be loaded on the editing side to ensure each floor displays the correct map. Then a number of nodes will be placed along with their corresponding edges. The building will be saved and then reloaded on both the editing and touring side to determine if the changes were made permanent.

- Displaying the correct building for selection.
    - Associated Test: The building selection screen appears twice in our GUI, once when an administrator wants to load a building for editing and again when a user navigates to the screen where they can receive a tour. On the admin side, every created building should be displayed with the capability of being selected regardless of whether or not the building’s setup has

been completed. Whereas on the tour side, only buildings that have connected maps will be displayed.

- Take the user’s input for a destination request and calculate the shortest route, then use this route to highlight the relevant nodes on the map.
    - Associated Test: Once the user presses the “Start Route” button all of the nodes in order between the start to the destination should light up yellow. The nodes that light up should equal the nodes in the shortest path a given map and route that is calculated ahead of time.
- Display directions to the user while they navigate this route.
    - Associated Test: The directions are already known to be correct via unit testing, now the integration with touring must be verified by checking to ensure the directions are displayed to the user at the proper time.

- Properly switch between floors whenever the user wants to either view a different floor or the route indicates the user has traversed a transition node (i.e. an elevator).
    - Associated Test: A user is able to switch between floors at any time while a route is not running. Since each floor is stored within the current building’s object, our GUI must correctly retrieve the requested floor and subsequently display it. This functionality will be tested by attempting to switch floors within a building that has 1 or many floors.

3.3 Graphical User Interface and Wi-Fi Scanning
These two modules interact with each other while the software is actively giving a tour. Once a user begins a route, our software will repeatedly scan nearby routers in order to estimate the device’s position within the building. Whenever the estimation changes nodes, a small robot head will be placed over that node to indicate to the user their current position in somewhat real time. These updates occur approximately every 3 to 4 seconds. To test this feature, we will simply need to operate our software in multiple test buildings and judge whether or not the positioning of the robot head is accurate enough to reflect our true current position.

4. Usability Testing
Usability testing allows us to test the functionality of our system with end users. This will evaluate how well the user can interact with our product and what functional or aesthetic changes need to be made to make sure our software is as streamlined as possible. As the creators of this software we may believe our system is sufficiently intuitive, but there are likely many things that will show up during this testing phase that highlight any

shortcomings. By letting people who have no prior knowledge of this software attempt to use it, they can give constructive feedback on how easy the system is to use.

The procedure for our usability testing will be a four step process. First, we will need to choose a group of at least 10 users with varying levels of computer skills to assess our system. Second, have each of these users complete tasks (outlined below) while interacting with our GUI and being given minimal directions. Ideally, they should only be aware of the basic function of the system. This will help us find out which areas of the GUI need more descriptive or prominent labels (i.e. names and placement of buttons). Next we will compile all of the feedback we receive from these test users. Finally, we can make all the necessary edits to our software based on that feedback and which problems a majority of the test users agreed on.

The following tasks we will require our test users to complete will cover all of the functionalities of this software product.

- Login to the administration section (they will be given credentials prior to the test).
- Set up a new admin user with a placeholder name and password.
- Create a new building, using arbitrary maps for each floor.
- Load this newly created building in order to set up nodes and edges on each floor. The user will be asked to establish 5 nodes and connect them logically.
- Return to the home screen and then open up the building they just completed in the tours page.
- Request several destinations to different points in the building.

For several users, we will forgo the administration section and just have them operate the tour section on previously set up buildings. This will account for the fact that most users will only be interacting with the tour functionality. If at any one of these tasks the user does not make progress in at least 1 minute, we will take note of why they may have gotten stuck and then show them how to complete the task.

5. Conclusion

Unit, integration, and usability testing is extremely important to producing a software product that minimizes the amount of bugs that it contains. The main focus of our unit and integration testing will be on the Wi-Fi scanning module, building data structure, and Graphical User Interface (GUI). Where the integration testing phase is concerned with ensuring each of these components is able to communicate with each other as intended. For unit testing, each component is broken down into smaller units. The Wi-Fi scanning module will be tested on if it can properly scan and store Wi-Fi information, as well as if it can successfully compare two scans in terms of our established algorithm. The building data structure needs to be able to correctly maintain an adjacency matrix to represent a floor's map, utilize Dijkstra's algorithm to calculate the shortest path between any two nodes, produce orientation dependent directions along that path, and properly save/load a building's data. While the GUI must allow a user to visually edit the stored building data and make destination requests to any node in the building from their current location.

Regardless of our software's performance during unit and integration testing, if people do not find the system to be user friendly then our product would not adequately fulfill Dr. Leverington's vision. To get high quality feedback from our test users during the usability phase, we will have each of them complete a series of tasks while taking note of the areas where they get stuck. The users will then list the parts of our software they find to be the most intuitive and the parts they had trouble understanding without any instructions.

Once the overall product has been improved by fixing the issues that appeared during each of the three testing phases, our team can move forward with constructing a user manual that best reflects the final state of the system.