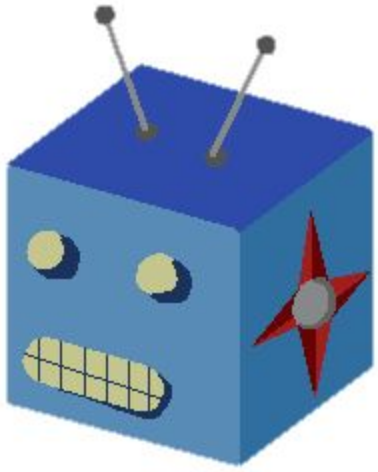

NaviBot Systems

Michael Leverington - Sponsor
Scooter Nowak - Mentor
Diva Ferrell, Logan Behnke,
Peter Aaron Giroux, George Cadel-Munoz,
Benjamin Peterson

Software Testing Plan Version 1

04/03/2020

Overview

This document will outline a test plan for the Thirty-Gallon Robot Part Deux project. We at NaviBot Systems will address how we plan to meet the expectations of our sponsor, and ensure that our implementation satisfies the functional and nonfunctional requirements described thus far.

Table of Contents

1. Introduction

Together, under the mentorship of Scooter Nowak, NaviBot Systems has been working on further developments of the "Thirty-Gallon Robot," sponsored by Dr. Michael Leverington. Dr. Leverington is a lecturer in the College of Engineering, Informatics, and Applied Sciences program at Northern Arizona University with a masters in both computer science as well as educational psychology, and a PhD in curriculum and instruction. He has proposed an affordable alternative to incorporating robotics into a learning environment. Specifically, Dr. Leverington would like to create a robot that, once complete, will be capable of providing self-guided tours throughout the Engineering Building located at Northern Arizona University, and we determined the need of a Navigation module, Mapping module, a Wi-Fi Localization module, and GUI. For a more in depth look into these components, refer to our Software Design Document.

Testing is an essential part of any large scale software project which involves writing code to check that all aspects of your application are running as intended. There are many different methods to testing a software project which lend themselves to different scenarios. For example, you may write simple functions to assert that the return value of a function is as you would expect, or you might need a focus group to test out your GUI to ensure that it is intuitive to its intended users. Regardless of the types of tests being deployed in a software system, they all contribute to a bigger picture of clean, efficient and easy-to-use code that is working as intended.

For the purpose of the Thirty-Gallon Robot Project, we will start by writing unit tests throughout each of our modules outlined above. These unit tests will test many key functions in each module to ensure that they are working as intended and any edge or failures have been properly addressed. Once our unit tests are implemented and passing, we will write integration tests to check on the interactions between all of the components. These interactions include data sent back and forth between the GUI and the Navigation module as well as data sent from the Mapping and Localization modules into the Navigation module. These integration tests will work to prove the different components of our project are able to work together efficiently and

without error. Lastly we will be performing usability tests on our GUI. In these tests, we will have various users try our GUI and provide feedback. This will ensure that the user's interpretation of our GUI is in line with the way it's been programmed.

We feel that these three testing methods will work hand in hand to create a very good test coverage of the entire project. Unit tests will ensure that the inner workings of each component run as expected, integration testing will show that the interactions among the different components are handled correctly, and the user testing will show that our GUI is intuitive and user-friendly. We will begin the discussion of our testing plan with unit testing.

2. Unit Testing

Unit tests are vital during the development stages of large scale software projects. They are used to ensure that the product is being developed to consumer specifications, and to ensure that the product is being developed correctly. These tests allow us to validate portions of our modules and ensure that the overall module performs how we expect them to run prior to the integration phase. Each module—Wi-Fi Localization, GUI and GUI server connection, Mapping, and Navigation—will have individual unit tests written in their respective language (i.e. Python, C++, etc.).

2.1 Wi-Fi Scanning

The Wi-Fi scanning module will have unit tests written in Python 2.7, as the scanner itself has been written in Python 2.7. Utilizing the built-in Python library *unittest*, tests will be looking for set requirements as dictated by the user (however, in integration, set requirements will be given to the scanning module via the Wi-Fi Localization module). For the Wi-Fi scanning module to pass all tests, the program must be capable of providing the following:

- **A comma separated value file with a header.** A CSV file stores all information about a router, including signal strengths, signal names, signal addresses. This file is a necessity for the Wi-Fi Localization module as it provides vital information to help approximate R.A.T. in the engineering building.
- **A comma separated value file with a specified name created after running the module.** In order for the CSV file to serve its purpose, the Wi-Fi Localization module should be able to locate the CSV file by a given name and location. This test ensures that the file will always be found under the specified name/system location.
- **A list of scanned routers with legal values.** Sometimes hardware can become inconsistent and provide false or incorrect information. To exclude data from routers displaying illegal values (such as values < 0% signal strength and values > 100% signal strength), this unit test will ensure that all values recorded in the table are within desired bounds, and will continue to rescan routers until the requirements are met.

- **A list of scanned routers listing greater than 3 routers.** For the Wi-Fi Localization module to provide an approximate location, the minimum requirements for the localization algorithm requires at least 3 routers. With at least 3 routers, R.A.T. may be able to triangulate its own location.
- **A list of scanned routers with signal strengths bearing values greater than or equal to a specified value.** To provide reliable approximations, the signal strengths of the routers will have to meet a minimum value in order to be usable information. This can be set via a command line flag and the unit test will ensure all recorded routers have a signal strength bearing the specified signal strength value or greater.

With these given requirements, a reliable Wi-Fi scanner will be able to provide countless amounts of information without sacrificing quality and quantity of data.

2.2 Wi-Fi Localization

The Wi-Fi Localization module will have unit tests written in Python 2.7 to match the Python version that R.A.T will be running. Testing for this module will use Python's built in library *unittest*. In order to pass the test the Localization module will need to take various pre-generated CSV files with router information and a pre-generated "map" with coordinates for each router. It will then need to correctly output the following:

- **A dictionary containing correctly calculated distances based off of signal strength which does not include distances over 25 meters.**
    - This will test both the calculation of distance from signal strength, and the removal of outliers. The distance calculation must properly use the following formula: $10^{(FSPL - K - (20 * log(f)))/20}$ where FSPL is the Free-Space-Path-Loss equation. The outlier removal function must remove all distances over 25 meters.
- **A dictionary containing correctly formatted router information with the mac address as a key**
    - This will test the function that reads a CSV file for router information. It must be able to read all of the information from the CSV and format this into a dictionary correctly.

- **A list of correctly calculated intersections of each distance circle.**
    - This will test the intersection function as well as the removal of outliers from the intersection list. Intersections must be calculated using a series of equations, and thus can be tested in reverse using the Pythagorean Theorem. Once the intersections are calculated the outlier removal function must remove all intersections which would leave the predefined building constraints.
- **A correctly calculated midpoint**
    - This will test the final part of the module, where the midpoint is calculated. The midpoint must be within 5 meters of the predefined location of R.A.T. that was used to create the test. Ideally the midpoint should equal the predefined location of R.A.T. however this is not always possible.

For this testing a module will need to be created to generate locations of R.A.T, a CSV of signal strengths being a communication between R.A.T. and the building routers, and a CSV of router locations that works properly with the signal strength CSV and the location of R.A.T.

2.3 GUI and Server Functionality

As stated previously, the GUI—named "R.A.T. Tracker"—is the component that an operator will have the most interaction with. It displays R.A.T.'s location and status, which gets consistently updated every fifteen seconds. That being said, it is imperative that it is well-tested and that we catch undefined behavior. We will check the functions that get called whenever a button gets pressed to see if they are doing their jobs. For example, the "Submit" button for choosing a room and floor for R.A.T. to go to uses the POST method, and in this method the form is checked to see if it is valid before it can be referenced in HTML. Values in the database are also checked before being shown to the user in the frontend. We will also check that the server connection is consistently stable.

Since much of the GUI functionality relies on an interaction between a database to retrieve or update certain data, it may be difficult to write unit tests since return values and parameters to these functions are likely to be different every time. In order to work around this problem, we

will use *pytest* mocking to avoid any real connection to the database. Rather than retrieving the live data from the database, we will simply mock the function that would access the database to return a value of our choosing. This will allow us to test the functionality of the GUI independently of its connection to the GUI server, which will be tested separately in our integration tests.

2.4 Mapping

For the Mapping module, much of the usability relies on real-world-testing. R.A.T. will need to be able to read a map input. To test the map readability, R.A.T. will be given a "hand-drawn" map, meaning a floor plan with defined distance. These tests will need to operate in conjunction with the navigation module, as correct map reading will prove that the Mapping module is providing correct instructions to the navigation hardware. In order for the Mapping module to be deemed complete, R.A.T. will need to perform and pass the following tests:

- **Traverse a known distance.** An example of a known distance is a hallway. This will ensure that R.A.T. will be able to travel straight pathways.
- **Provide clearance for navigation.** R.A.T. is given its own dimensions, which means that it will have to account for clearance between itself and walls. This will ensure that R.A.T. will always avoid collisions in perfect conditions (i.e. no obstacles moved in front of R.A.T. during operation).
- **Account for inaccessible areas.** This marks areas such as stairwells being inaccessible for traveling. Doorways are also to be avoided, as tours inside individual rooms are unnecessary.

Through these tests R.A.T.'s ability to read and interpret the given "hand-drawn" map will be examined. Each unit will show a different part of R.A.T.'s mapping abilities.

2.5 Navigation

Most of the Navigation module's usability also relies on real-world-testing. For the unit testing, it would be the best to run R.A.T. through a series of hallways, rooms, and situations to ensure

proper navigation. This means that our team will set up automated scripts to give commands to R.A.T for various scenarios.

- **Scenario 1**
    - R.A.T. will be placed in a straight hallway and will be told to move various amounts forward, backward, and to each side. R.A.T must be able to distinguish between open space and obstacles, and must move as directly as possible.
- **Scenario 2**
    - This will be much like Scenario 1 with the addition of un-mapped obstacles, and moving people (our team)
- **Scenario 3**
    - R.A.T. will be placed in an open space with various un-mapped obstacles like a classroom or lab space. R.A.T. will be sent various commands like the commands in Scenario 1 and must be able to distinguish between open space and obstacles, and must attempt to take the most direct route possible.
- **Scenario 4**
    - This will be much like Scenario 3 with the addition of un-mapped obstacles, and moving people (our team)
- **Scenario 5**
    - R.A.T. will be put through a series of combinations of obstacles 1-4 with various un-mapped obstacles.
- **Scenario 6**
    - R.A.T. will be told to move to the end of a hallway with stairs. R.A.T. must be able to detect the stairs and either return a warning or at the very least attempt to avoid the stairs. R.A.T may not sit at the top of the stairs, and thus must move at least 3 feet away before attempting to process any new commands or information.

These various tests will handle all of the units that comprise R.A.T's various movement and navigation capabilities. They will also test R.A.T's ability to avoid new obstacles, mapped obstacles, moving obstacles, and stairs.

3. Integration Testing

Integration tests are a key part of developing and maintaining a well-running software project. While unit tests focus on the inner workings of a project's individual components, integration tests are focused on ensuring that these components can communicate the necessary information between each other in a seamless and bug-free manner. For the purposes of our project, there are several connections between our four modules that will benefit from integration tests. These connections include those between the Wi-Fi scanning module and the Wi-Fi Localization module, the Wi-Fi Localization module and the Navigation module, the Navigation module and the GUI server, and lastly, the GUI webapp and the GUI server.

Integrating the Wi-Fi Localization and Wi-Fi scanning module involves the communication of commands from the Localization module to the scanner. The Localization module will need to provide necessary arguments such that the scanning module can account for desired information and formatting. Since signal strengths have an important role in determining the outcome for a location approximation of R.A.T., the Localization module will need to dictate the type of signal strength necessary to provide accurate location approximations to the Navigation module. In order for the Localization module to read data collected by the scanner module, the Localization module will need to provide a file name and location that it will be able to access once the scanner completes its tasks of seeking and recording Wi-Fi signals with appropriate data, that has also been tailored to the specifications of the command arguments given by the Localization module.

Once all communications between the Wi-Fi Localization module and Wi-Fi scanning module have concluded, the Localization module will compute an approximation of R.A.T.. This calculation will be passed on to the Navigation module, in an attempt to remove extraneous guesses where R.A.T. could be located. Limiting the amount of guesses the Navigation module estimates will decrease processing load and increase localization performance. Removing extra

load from localization allows R.A.T. to identify its surroundings quickly, and continue with further navigational operations.

In order to test that the communication between the GUI webapp and the GUI server are integrated correctly, we must check up on the retrieval and updates on data in the GUI server's database. This connection yields very little risk of failure since both of these components are a part of the same Django project. However, it is still very important that the communication with the database is running as expected, and therefore we will create integration tests to ensure this. These tests will be written using *pytest* and will work on an area of the database set up specifically for testing. The tests will be relatively straightforward, simply testing that an update and retrieval of data are carried out successfully when called from the webapp.

The communication between the GUI server and the Navigation module allows a user to send commands to the robot. As such, it is very important that this communication is implemented properly to ensure that the robot is receiving and interpreting the correct information in a timely manner. Failures to pass this information quickly may result in catastrophic failures such as a robot collision. In order to ensure that these failures don't occur, we will be running integration tests on this connection using *pytest*. These tests will send sample data from the server to the Navigation module and vise versa.

For tests involving data sent from the Navigation module to the GUI server, we will want to send sample data (x- and y-coordinates, robot status string) to the server on some time interval since this is how the program will run in practice. Upon receiving this data, we will check on the GUI side that the locations are within range of the map, and the status string is valid. We also want to check that the messages are received on time intervals similar to the ones that the navigation is sending them with.

For tests involving data sent from the GUI server to the Navigation module, we will send sample data to the Navigation module (an x- and y-coordinate that corresponds to a room on the map).

We will test that the Navigation module has successfully received this information. Once the information has been received, we will check that the x- and y-coordinates are valid on the map, as well as check that they correspond to the room that was specified by the GUI server.

4. Usability Testing

Unlike previous testing methods mentioned throughout this document, usability testing places its emphasis on the user experience of a project as opposed to the proper execution of code. Testing to assure that a programs front end is acceptable by an end user can't be done by writing code. Instead, usability testing is often conducted with methods such as focus groups, where developers of a project can receive real feedback from real users.

NaviBot Systems wants to provide software that will be relatively simple for our client to operate on a day-to-day basis without having to consult manuals and documentation often. Our GUI is the only part of the Thirty-Gallon Robot project that an end user will be interacting with. Usability testing will help us to ensure that our GUI is intuitive and functions according to an end user's expectations. Additionally, these tests will help show whether or not our GUI provides enough functionality for a user to operate the robot effectively.

In order to conduct usability tests on our GUI, we will enlist the help of three different groups of people, these being Computer Science professors, Computer Science students, and non-Computer-Science users. It is likely that the end users of this project will have some computer science background, however this is not guaranteed. The goal of breaking up our test groups into these three categories is to allow for three different levels of expertise to provide feedback on the project. Computer Science professors and students may provide helpful critiques on technical improvements that would create for a better experience, while non-Computer-Science users may point out design flaws that might otherwise go unnoticed.

In order to allow these groups to test our project, we will present them with our GUI and tell them to send a robot to a certain room. Once the tester has issued a command, we will ask the user to read the robot's status and location to us while it is moving to its location. If the testers are able to issue a command to the robot without asking for our assistance, this will act as a good indication that this feature has been implemented in a way that is easy for users to understand.

Contrarily, if the users are unable to send a command to the robot without asking for help, this will likely act as a sign that something about our design is in need of change. If the testers are unable to read the robot's status and location from the GUI, then this is an indication that the information is not presented clearly enough on the screen. After a tester has carried out these two tasks, we will offer them a chance to provide any feedback about the GUI that they have picked up on. To help guide this conversation, we will have questions prepared to ask the testers about their experience with the GUI.

This process will be carried out with as many testers as possible. Having a larger group of testers will allow for more diverse feedback about the GUI which will help guide us to a polished and well-working product.

5. Conclusion

NaviBot Systems is dedicated to providing an efficient, easy-to-operate, and autonomous robot for the purpose of showcasing the Engineering Building and garnering attention to both the computer science program and Northern Arizona University as a whole. The four modules being delivered are Wi-Fi Localization, Navigation, the GUI, and Mapping. Because these modules are closely integrated and built off of one another simultaneously, we will put significant time into testing these components individually in addition to how they work together as a single unit.

The testing of the Wi-Fi Localization module and scanning the routers involves using the built-in Python library *unittest,* two of several passing requirements being a list of scanned routers with legal values and a list of scanned routers listing greater than 3 routers. The Navigation module's testing will consist of traveling through straight hallways—both with obstacles and without—including avoidance for people. As for the GUI's testing, it focuses more on simulating what functions called by buttons would access within the database and ensuring that they return a value that is up to standard. Lastly, the Mapping module's tests involve the criteria of traversing a known distance, providing clearance for navigation, and recognizing areas that are off-limits to R.A.T.

With these benchmarks in mind, we plan to do a thorough job testing the Thirty-Gallon Robot in order to deliver a working system with as few problems as possible to our client. This is a project with the potential of inspiring others to inquire more about computer science and robotics, and to see what kind of learning opportunities can be found at the university. Thus, we are excited and confident in our ability to provide such a tool.