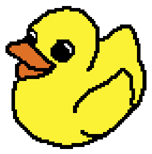
Software Testing Plan

April 1st, 2022



Team Code Duckies

Team Members: Anthony Simard, Ari Jaramillo (lead),

Daniel Rydberg, Chris Cisneros, Jacob Heslop

Sponsor: Dr. Truong X. Nghiem

Mentor: Vova Saruta

Table of Contents

1. Introduction

Cars of the future, robots at the wheel, self-driving vehicles, or Autonomous cars. These are all phrases to describe once fictional vehicles that are currently being developed and, in certain places, being deployed. Being able to develop, code, and test these vehicles is an experience that few have. The goal of team Code Duckies is to be able to expand the scope of those who can experience this thrill by simplifying the coding process for smaller, cheaper robot vehicles so that someone with little to no experience programming will be able to build instructions for these vehicles. The program that we have determined to make our Code Duckies user interface from is called Blockly, a web application that allows for blocks of simplified instructions to be dragged and dropped into a work area and reorganized to do some programming work.Our program will first have to go through software unit testing to make sure that what we have built as a user interface does communicate well with the robots that we are using, the MIT Duckiebots. These Duckiebots were built with the goal to be affordable and small while still being able to mimic conditions much larger vehicles would still have. We have also determined that the best focus group for those who would be testing our user interface would be middle and high school students; who can understand basic instructions, but probably have not had as much of an opportunity to build their own programs.

Our goal is to leave as much of the software on the Duckiebots alone, instead focusing our work on the creation and integration of our Blockly user interface. This means that the Duckietown software should not need to be tested, since it has been developed and tested extensively at MIT. Blockly is also, for the most part, a packaged programming product that we are only changing to talk to the Duckiebots. Because of this, there is not much unit testing that will be able to be done with Blockly; therefore a majority of our testing will be integration as well as usability.

Integration testing will mostly make sure that our Blockly interface connects with the Duckiebot’s software and is able to control the Duckiebot in a way that we expect. Our tests will include a variety of checks to ensure that the wireless networks can connect properly and that all commands sent from Code Duckies interface work with Duckietown software. Further tests on restrictions (one user can only connect with one robot and vice versa) will also possibly be tested.

Since our goal is to make our software simple to understand to almost anyone, usability testing will play a major role in the instructional design as well as the overall looks and functionality of our user interface. Tests such as how well someone can understand blocks of instructions, if they can find the right buttons, as well as the results of asking someone to navigate through our menu could change what blocks we put into the future.

2. Unit Testing

The purpose of unit testing is to test if each individual portion (or unit) of the software works on its own. Some canned input is provided to the unit, and it is asserted that the resulting output is what would be expected given the input. When performing unit testing, it is particularly important to identify and test edge cases which is generally done by determining what different classes of input are expected. For example, if a function does something with an integer, the test cases may be a positive, a negative, and zero. For most functions, the difference between a one and a one thousand is not meaningful, but the difference between a one and a negative one might be.

In the case of our code base, automated unit testing is a bit tricky to execute. Much of our behavior is dependent on having a properly calibrated bot to connect to, and the behavior of said bot when our software is used to control it cannot be precisely measured automatically. Much of this behavior is also dependent on the code we are interfacing with and not our code which is a black box we are unable to properly peer into.

Keeping that in mind, we will be testing the only two pieces of our code base we can directly control. First, we will be verifying that the bot connection info (name and port) the user enters is correctly transmitted from our front end to our back end for usage to connect to the bot. We can do this by simply console.logging the information from our back end and verifying it is correct.

Second, we will use the same method to verify that the code templated out by our blocks is correct. The actual templating of the code is controlled by Blockly under the hood, but we have complete control over what code is templated. This will allow us to validate that we are getting the results we expect.

3. Integration Testing

Integration testing's primary function is to combine the different modules of our software together to ensure that entered data flows through our system without any failures in communication between modules. Integration testing typically comes after unit testing, since the goal is to test specifically the transfer of data and not the modules themselves. Depending on the size and complexity of a system, there are several different approaches that can be taken to identify critical issues. These differences in approach largely affect the speed in which higher priority issues are found and where they occur. For our project, our choice came down to a BIg Bang Approach or a Incremental Top-Down Approach.

In a Top-Down Approach, modules are combined from the top module to the lowest module in stages, testing the communication between each module before further complicating the test with additional modules. This approach works fantastically in larger, complex systems since it allows for a slower overall testing process, but quick identification of higher priority critical issues and easy location of where those issues are occurring. Ultimately we decided to go with our second choice, the Big Bang Approach.

In a Big Bang Approach to integration testing, the whole system is tested at once. In larger systems this can lead to difficulty when finding critical issues, but if a system is sufficiently small this approach is the quickest and most efficient. Since our software connects to larger systems together, each with their own testing already, our system is quite small with little room for error between communication of its few modules. We only need to test the custom block library that we created for Blockly and the ROSLibJS code that we added to the webpage in order to access the ROSBridge instance already running on the bot.

Now that our approach and general idea of what we need to test has been established, we only need a plan to accurately test our system. Since we chose the Big Bang Approach, our current plan to test is to place all of our developed blocks into a single file and verify that the bot properly is receiving the commands. To do this, upon visiting our web app you are immediately prompted to connect to a bot via a name and port number. We already run a bit of test here with a print statement stating whether or not connection to the bot was established, so modifying this into a proper portion of our integration test should be rather easy. After that connection is established you are taken to our Blockly environment, which allows you to drag and drop blocks into the workspace to send commands to the bot. Since each of our blocks functions as its own command to the bot and they are designed to be run together in multiple orders, our test program will just be one program combining all of our blocks, one each and testing to make sure the bot properly executes all commands. Each block can be seen as its own module, and each module fundamentally has the same general loop. Take user input, format that user input into a ROS message, and send that ROS message to the correct corresponding ROS topic through ROSLibJS and ROSBridge. If the block can correctly format the ROS message from the input and the bot successfully completes the command, the test is a success.

Due to how software is set up, even though we are using a Big Bang Approach, locating where an error and how an error is occuring is made simple by physically being able to see which command the bot didn’t respond to. Similarly we can see if a connection issue occurred if the bot doesn’t respond to any of the commands at all.

4. Usability Testing

Usability testing evaluates the functionality of the Code Duckies application by having people of the target audience give feedback on the application either verbally or through specific tasks. The purpose of usability testing is to figure out any issues that make it difficult to use Code Duckies, collect qualitative and quantitative data, and then obtain user/client feedback to try and improve or fix the application.

In order to have proper usability testing, there are a few steps that need to occur. The first step is to choose test participants that will provide relevant data. For Code Duckies, the end users are middle-school and high-school students who are ten to eighteen years old that might have little to no programming experience. Therefore, the participants being used in the tests need to be between those ages; the team needs at least five participants in total to receive enough data to be reliable. The second step is deciding what tests each participant needs to complete. Feedback on the user interface and how intuitive it is to use is vital because that is a major objective of the product. A test participant will use Code Duckies and provide feedback on the general user interface as a whole, then provide more detailed feedback on specific areas such as the introduction page where Duckiebot information is inserted and the visual programming page where users can create their program. Also, they will be tasked with creating different programs and interacting with the application in specific ways, so the time it takes participants to complete the various tasks can be recorded. The testing data collected from the tasks and the user feedback will be used to determine where refinement is needed.

While choosing the test participants required, specific tasks to evaluate the application can be defined. First, the user will be tasked to go through the process of inputting the robot's name and port number, and then submit the form. Afterwards, the test participant will be told to create a program that repeatedly moves the bot forward, backwards, then turns it right, followed by left for an infinite amount of time. Then, the user will save their project, clear the workspace, and load their saved project. Lastly, they will press play, let the program run for thirty seconds, and then press the stop button. Each step in the process will be timed and their feedback on the general user interface and individual sections will be recorded. This test covers each use case possible from setting up the connection to the Duckiebot, creating a program, exporting/importing a project, and starting/stopping the execution of the created program.

Another test will involve having the client, Dr. Nghiem, perform the same tasks as the test participants. In doing this, the team will receive feedback on any possible issues the client sees with the Code Duckies application. It is vital that the client is happy with the product created for them.

Once the data is collected, the team will organize and analyze it carefully. Any major issues will be addressed and fixed as soon as possible, then lower priority issues will be dealt with accordingly. Finding test participants and completing the tests will take place after unit and integration testing is completed. The ultimate goal is to finalize the product in a way that assures its user-friendliness and accessibility to the target audience.

5. Conclusion

Our team will begin by unit testing each module we have developed, ensuring that the correct output is returned for a given input. These include our website, connecting to the Duckiebot and each of our custom blocks. Testing the connection to the Duckiebot is as simple as sending it a basic command and watching it execute. Since Blockly already handles invalid user input, there is no need to test that. Instead we will test that each block generates the expected code for given power and time inputs. Once these modules are shown to function separately, we can move on to integration testing the entire system.

We have decided to use the Big Bang Approach - testing the functionality of the entire system at once - rather than an incremental approach, as our system is small and simple enough to make this most efficient. We will connect to the Duckiebot using our frontend and send it a series of commands using blocks with predictable results. If the Duckiebot moves as expected, it passes the test. If this approach fails, we can always resort to an incremental approach to see where exactly the error occurred.

Finally, we open testing to people outside of team CodeDuckies, allowing them to see design errors in our user interface that we might have missed. We will request feedback from a variety of different people - including those with no idea how our project works under the hood - and use this data to improve usability. We will continue to test usability until our client is satisfied. Team CodeDuckies is confident that this testing process will reveal any lingering errors in our product, allowing a new generation of computer science students to learn how to program autonomous vehicles.