Madeline Wessels

UW/CSE User ID: mwes

CSE 490 E1

Assignment #1

1/5/17

**Brainstorming About Software Development Difficulties**

Difficulty #1: Addressable LED Array – IMPLEMENTATION/DESIGN Problem

Solving problems when the problem is not well defined is a classic engineering nightmare. This problem arose when writing code for an addressable LED array (as a side project) without the hardware- which had not been created. In cases like these, the spec sheet for the hardware is unknown. The difficulty of the problem lies within its generality, and teams need to focus on functionality and developing abstract methods to accomplish desired behavior- in this case, blinking the LEDs, creating LED patterns, and other functions. Current tools and techniques don't necessarily solve the problem because the solution arises on a case by case basis through thoughtful reasoning and decision making. To generalize, if a company were to write code for hardware that was in development there are many ways to work around the problem. The company could hire more engineers in attempt to seed up the process, using valuable money. Alternatively, the company could wait until the hardware is complete, wasting time in which a competitor could release a competing product. A better solution, used in this example, is to concurrently develop the hardware and the software. While it is challenging, the resulting code becomes more modular and is abstracted from the hardware. More developed tests are created which can be used when the hardware is available to check hardware behavior.

I used CppUTest for the testing framework. This open source suite offered a variety of macros in C++ for testing C and C++ code – which fit my low-level hardware application well. Without a large user-base this testing harness is not well-known. Developers might be unaware of this resource which makes concurrent development much riskier without tests to validate design/implementation.

Difficulty #2: LEDs with a Time Element – TESTING Problem

This second difficulty extends upon the first as it arose while working on the same project- an addressable LED array. However, in this case a time element was added so that the LED strip would be turned on at a specific time- perhaps to turn the lights on when away on vacation or to create flashy lights for fun parties. The problem arose during testing. It is difficult to fake time as it is often taken from a system clock that cannot be controlled. Moreover, it is harder to test when the input cannot be faked. People often take time for granted. However,

company employees testing code with time elements must meet development schedules. There is no luxury of waiting, as it wastes time and ultimately money. Once again, because the problem is general, there are multiple solutions, each specific to applications. Going back to the problem of not wanting to wait, it would be ideal to have a tight feedback loop so that tests can be run and results obtained quickly. In the case of the addressable LED array, the solution (one of many) was to build a spy that would inject time into the code. The result was instantaneous time changes due to the controller (spy).

I used CppUTest for the mocking framework. The tests were seamlessly executed with the C++ macros and fit into the project library ecosystem well. Continuous Integration made the process go smoothly. As a still-developing open source framework, this repository has not picked up a massive following. For this reason, I believe many developers are unaware of the resources available for testing.

Difficulty #3: Creating Comprehensive Tests for a Graph – TESTING Problem

Writing a comprehensive test suite is challenging and tedious. My first exposure to developing tests was in the software design and implementation class taken after introductory programming. Part of the large project to build a map of the university campus was to design, build, and test a graph abstract data type. It is easy to create tests that are obvious, but challenging to remember to account for all scenarios of input. Moreover, as a beginner, I made the mistake of coding first, so when I created tests, I made assumptions about my code that turned out to be false. Creating a comprehensive test suite is important because they help with eliminating bugs and ensuring confidence in program behavior. Writing tests requires careful thought, and the types of tests to make are situation dependent. Making tests later is something many developers default to. Yet, the tests are really the checks on our assumptions and when we don't check our assumptions we make all sorts of mistakes. Tools provide an environment in which tests can be written. Certain techniques, such as writing tests before attempting to code, help create a thorough testing suite. Another possible way to handle testing could be to ask multiple developers to write tests, although in a company this could prove to be time-consuming.

I wrote the code for the graph in Java with the Eclipse IDE. Tests were created and run through the JUnit testing framework. These tools were used as mandated by the course and are more well known.