Project 2
Mark Webster

The software requirements wholly guided my approach to the software testing. In design, requirements are really two statements in one: a software should do X correctly, and should not allow X to be done incorrectly. Therefore, in designing my unit tests I followed two rules: one, that tests must confirm that the software met requirements when given proper input; and two, that tests must also confirm that the software rejected bad input, therefore meeting requirements for error handling. To those ends, in each module I specifically designed certain classes of tests that confirmed functionality under proper constraints – for example, having no non-digit characters in a phone number input, or not going over the character length limit on any given input field. This testing was accomplished cimply by passing the input to the constructors and then confirming that my input information matched the newly created object's using the accessor methods I had written. Once tests confirmed that proper input was successfully used to create the desired object, it was time to "stress test" my software. As Mike Tyson is famed for quipping, "Everyone has a plan until they get punched in the mouth." How would my software handle deliberately being fed bad input?

The outcome that any developer wants to avoid is the one in which software accepts the wrong input but continues forward treating it as correct. This type of error mishandling is why security flaws such as SQL injection can exist. We must fully vet each input to ensure it is correct—if it is not, the program must not let things go forward. To achieve this filtration, I wrote error-checking into each project, sending input to a checking function to ensure it was neither empty, too long, nor failing on some other principle. If improper input was detected, runtime exceptions were written to throw, stopping the progress forward. During the testing of my error handling procedures, I focused on these exception throws using the assertThrows() function native to JUnit. By passing bad input or combinations of bad input to constructors, I tested my error-checking methods—for example, if the name field was left empty, or was three characters too long.

To accomplish near maximum level test coverage, I strove to test every possibility of input. In order to do this efficiently, I grouped my tests together based on what they aspect of the software each tested. For example, testing bad input in my task creation program was grouped together, each throw assertion testing a given field with null or too long input. In order to ensure efficiency in my testing, I made sure to cover the main errors we were trying to filter out, namely empty and too long input. Only in one program, the contact software, did I need to write a method to iterate through the string and check for non-digit characters. Aside from that, the approach was primarily to test the big picture methods (addTask() or updateContact(), for example), through a series of tests that ensure that the smaller functions that composed them worked as they should. Due to the simpler nature of the input specifications, I did not have to do any input partitioning or really filter the input beyond either its existence or its length. While this did allow my testing strucutres to be more lightweight, I still wanted to achieve maximum coverage, and spent a lot of time writing tests that would go through every method written.

The testing techniques I used in this project were dynamic techniques. As I was not working on a team or for stakeholders, there was no one to review the code in any formal setting such as a code review or walkthrough. Rather, I tried to mentally place myself in the role of a third-party, both when writing my initial software and when writing my tests. It helps that I have been doing code refactoring for security purposes as part of another course. Looking at my codebase, I tried to pinpoint weaknesses in the schema I had established. If I so desired, was it possible to get my system to accept the wrong information? If I was working as a member of a team, could other team-members understand the organization of my code and glean its function quickly? Both these practices helped me write concise and secure code with error-checking and the base requirements in mind. In the case of specification testing in a “black box” scenario, I wanted things to be very clear to a third-party tester: the system allows this, but not this; this is correct, this is incorrect. In short, I wanted the software to be intuitive, showing it follows the rules to a tester through its response, while at the same time obscuring its

methods of doing so. If you did not know how the system worked, you could quickly pick up that I had built ways to corral users into only inputting the correct forms of input.

In the future, I look forward to building and testing software with the aid of a team and a formal testing plan that includes both static and dynamic methods. I am also interested to test integration on more complex systems. While the software I wrote was not necessarily complicated – at most, perhaps two or three method calls in any given method—I have seen examples of much more complex code that calls from libraries, developer-written methods, and user input. The sheer scale that OOP codebases can reach is mind-boggling at times. Ultimately, the testing plan there will be much different from the smaller scale that I followed, but in some ways similar. We will test an a unit basis and build from there.

It is with respect to working with larger codebases that I try to refactor early and often. I'm cautious about writing more code than is necessary. Not only does more complicated code make testing more difficult, but it can often take things that can be done in simpler ways and break them into too many smaller parts. While this modularization makes sense in some aspects, especially because we want to avoid having to rewrite code within methods too many times, too much can lead to the dreaded "spaghetti code" which is a nightmare to disentangle and make sense of. In a way, learning how to test through JUnit has made me appreciate sleeker and more minimalist codebases, as they can be tested more easily and provide the same functionality in a lighter package.

As a developer, I understand that software that meets requirements is not always written correctly, nor it is by any means the most secure or efficient. In order to ensure code quality, it's important to assume that code can always be made leaner and more secure. While testing my own code, I try not to get caught up in the "Well, it works" fallacy. Yes, if I give something the correct input, it may produce the output desired. But how far off can I be before my system no longer works the way it's supposed to? What margin of error is acceptable?

In my mind, that margin of error should always be zero. Software is not correct until it works

every time when given the right things, and should not proceed at all every time it is given the wrong things. Any slippage in that standard, and we have a vulnerability. As such, the developer must also think like a hacker. How can the structure of a given piece of software be used against itself? For example, what if there was a way to circumvent the input validation in my contact program? If an external data structure was being used, we are now open to injection or other forms of penetration, and an intruder could possible take all the contacts in the system or worse When testing his or her own code, a developer must reject the notion that "I am the engineer, this is my creation, and if it works it is correct." This can lead to technical debt, as the team has to go back later and fix problems that arise when the code is incorporated in a larger system and integration testing shows errors. Units should work individually, but always with a mind to a larger-scale as well. Designing minimalist code that is not overly-engineered, and refactor early and often, we make functionality efficient and testing much easier on ourselves later. I could have broken addContact into a dozen separate functions, one each to check input, one to write each attribute to the new object, et cetera. We should group things together as rationally as we break them into smaller units. Often times, the simplest solution is most elegant—I intend to carry that forward as a mantra throughout my career.