Project Two

Scott M. Laboe

[CS-320-T3224 Software Test Automation& QA 21EW3](https://learn.snhu.edu/d2l/home/636628)

Southern New Hampshire University

February 21st, 2021

Project Two

**Summary**

When I first started organizing my thoughts about how I wanted to write the tests for the services in project one, my first consideration was the requirements given to me. At a minimum, I needed to ensure my services possessed every quality and constraint provided in the requirements. An example of this can be found in the contact storage class is the “setFirstName(string t\_firstName)” function. The requirements constrained the first name to not exceed more than 10 characters, which I verified by first implementing it in a test and making sure the test failed. I then went into the code and implementing the necessary checks in the set name function until I got a passing test. I repeated this process for every requirement provided for each class.

To ensure my tests had good coverage, I checked not just the limits, but also the area around them. Again, I evidence this technique in the set first name function in the contact class. When writing the check for this, it’s important to consider that only checking a string with a length above 10 does not provide adequate converge. For my tests, I would check the string just before the limit, which would be ten, just after the limit, which would be eleven, and a number beyond the limit like twelve. This ensures you get a valid result with inputs that are within the requirement and an invalid result when you are violating the requirement. There is typically a discrepancy of one between how you index an object in an array and the number of objects in the array, that necessitates you to check the limit and the area one index above and below it.

To ensure my code was technically sound I explicitly defined setup and tear down functions that ran before and after each test as shown below.

public void setUp() throws Exception{

contactService = new ContactService();

}

@After

public void tearDown() throws Exception{

contactService = null;

}

When I wanted to test things in series, I kept all the test code in one test case and when I wanted to start from a new known state, I created a new test case. This allowed me to set my tests in a predictable environment that wouldn’t unintentionally influence each other and ensured technical proficiency.

There isn’t much going on in this program that would require efficiency, but I did implement some techniques to ensure the code readability was efficient and slight improvements to computational efficiency (in the order of a few clock cycles). For both code readability and computational efficiency perspective, I simplified test checks to a single “if” check as shown below.

if( t\_ID.length() > 10 || t\_ID == "")

return false;

When there are only two checks it is more readable, and sometimes more computationally efficient (depending on the compiler) to put both checks into a single “if’ statement. If the first condition is true, there would be no need to check the second condition.

**Reflection**

Throughout the development of the classes and tests for this project, I used several software testing techniques that guided the form of my tests and improved their coverage. I’ve already mentioned the verification of boundary cases and the initialization of test cases as some of the techniques I utilized. Another technique I found useful in my project was to initialize every variable in the object’s constructor. This ensures none of the member variables are undefined, which can be difficult to test if the value they take on is inconsistent.

These techniques are universal and find application in a diverse range of software. They are not only useful for functional testing but also to ensure a minimum level of security. One of the greatest security vulnerabilities in software is improper input validation. Where there is space for users to input data into the system, there is potential for an attacker to inject malicious code. Input validation is done to make sure data is formatted correctly and isn’t malicious. It’s important to understand what the requirements are trying to accomplish, and not just fulfill them technically without any thought. An example of using intuition to expand the requirements to fit what is trying to be accomplished was done in my project with the verification that the string isn’t empty. String member variables can be empty initially but cannot be changed to be empty. This is an example of testing the correct formatting of user input.

It’s important to be cautious when adding new code, to ensure you aren’t compromising security or introducing new bugs into old code. For the application I was developing, there is a lot of interdependent functionality between various libraries and modules. This inherent complexity requires serious consideration even when writing something as simple as my program. For this project, I didn’t choose any of the libraries, but I did, it would be important to check how each module interacts with each other to ensure there aren’t any security vulnerabilities or potential functional faults such as running a threaded non-thread safe object.

Limiting bias in design reviews is always a challenge in any engineering field. If possible, you should have someone else examine your work to provide a new perspective in potential problems you might face. In this project, I didn’t have anyone other than my instructor look over my code, so I had to use some techniques to help manage my bias as the designer and tester. Writing tests allows me to switch my perspective from writing an internal use API to writing an implementation of the API. This perspective shift allows you to better understand the quality of your interface.

One of the greatest challenges and most desirable accomplishments in software engineering is designing bug-free software. Trying to accomplish this freehand is inefficient if the software is expected to evolve and adapt to changing expectations. Writing tests allows you to verify the software you produce is achieving the requirements defined before it reaches the user. Depending on the application, there could be serious potential to do harm to the user. (Schneider, 2002) Thoughtful design and thorough testing become a necessary course for development when there is any risk of harm to the user. Potential harm can be physical such as a life-saving device, or financial and mental with security risks. When developing software, a level of discipline must be taken to ensure the user has complete protection and is padded from any serious risks.

# References

Schneider, K. (2002, February 7). Clear and present danger: why we refused to give up. *Computer Weekly.*