7-2 Project Two

CS-320 Software Test Automation & QA

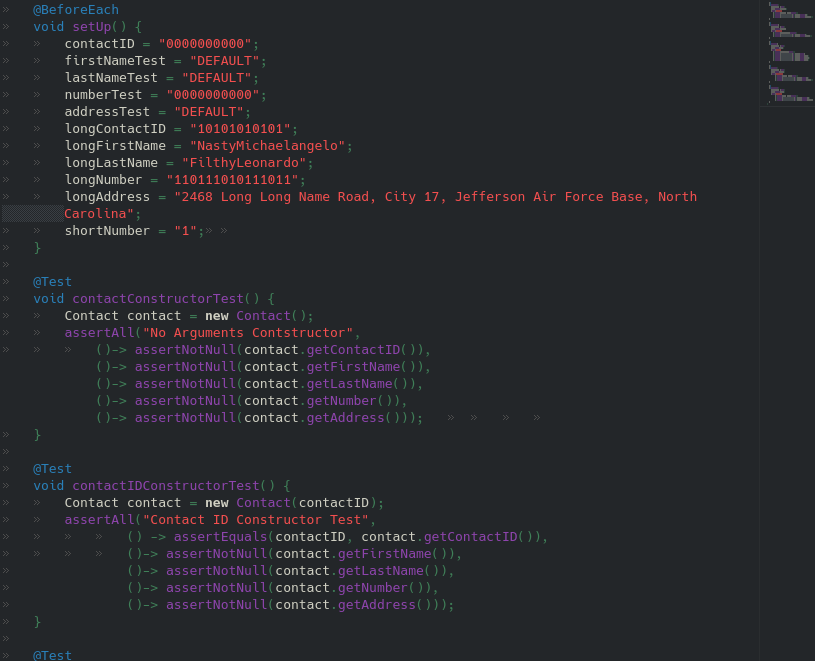
Christian J. Busca

Southern New Hampshire University

**Summary and Reflection**

When developing software solutions, unit testing is necessary. Swiftly identifying flaws and fixing them lowers the likelihood of encountering programming errors and raises the caliber of the code. I used a similar unit testing strategy for the three features because I made it compatible with the software requirements. A notable example was the Contact service, which had several conditions. The Contact object must have a distinct contact ID string, which cannot exceed ten characters in length and cannot be null or updateable. The specifications were for the firstName and lastName fields in the contact object to be required, not void, and have a maximum length of 10 characters.

Additionally, the Contact object has a service that performs some basic contact-related tasks. The software must be implemented with these criteria for the unit test to pass since they were all specific requirements for one of the three features and all the other features had identical conditions. My unit test's overall quality was good since I tried to test each field and operation to make sure the test cases were legitimate. In my contactTest I implemented parameters which checked the validity of the constraints. This was a good way to create a test case to see if the ID was too long.



The dynamic testing method involved examining the dynamic characteristics of the software code, which was one of the software strategies I employed. Using this method, I could identify the software's weak spots during runtime and examine the behaviors of the many dynamic variables that are not constant. Static testing, a way to test the code without executing it, is a software technique I did not employ. I preferred dynamic testing because I find it easier to run your code and study how the program functions within your tests. During the milestones, I also implemented black and white box testing strategies. Using black box methodologies, test cases are directly derived from the specification or some other form of a model of what the system should accomplish. Equivalence partitioning, used to test for both valid and invalid inputs, decision tables, state transition testing, events that modify the state or generate outputs, use cases derived from test cases, and boundary values, used to test borders are examples of black box techniques. The coverage tests investigate the components, and the if-then statements heavily utilize structure-based testing. It is used to divide tests into testable chunks. Statement coverage, path coverage, and branch coverage are structure-based approaches. (Automation Beyond, 2016)

I worked on this project with an analytical, experimental, and growth-oriented approach. I exercised prudence by doing a lot of research, testing, trial-and-error runs and attending tutorials. Given how much it affects the product's quality and performance, it is imperative to understand the intricacy and interconnections of the code. For instance, the coverage percentage was low when I ran my tests. The coverage percentage of my finished product, which added more tests and covered a lot more code, exceeded the required 80%. The additional tests ensured that the code was technically sound and that the result was of the highest caliber. I tested everything repeatedly to reduce prejudice in my code review, regardless of whether I was sure it would function well. We all know what occurs when we presume, so I tried to develop hypotheses rather than assumptions. Therefore, if I oversaw my code testing, I could see how bias would be a problem. For instance, I might have missed that the ID test wasn't functioning because of a missing line of code if I examined the function that checked that the length of the first name was no more significant than ten characters. When I approached this project with the proper perspective, I could guarantee that I captured all the test cases. I was cautious when testing the numerous issues that needed to be checked. Understanding the intricacy and interdependencies of the code, I was testing was crucial because each of the class objects had specific requirements to be met to vouch for the code's correct operation. One must comprehend what each code element accomplishes to execute all of this. As a professional in software engineering, discipline is crucial to assuring quality. Doing so will guarantee a seamless application development process with fewer bugs, issues, and coding mistakes. When it comes to writing or testing, it is crucial to avoid cutting corners. Testing the code can prevent millions of dollars in financial loss. This ensures proper operation, and everything functions flawlessly before shipping the code to production.

**Reference**

Test Engineering, test automation ... professional development, management, Community Building. Automation Beyond. (n.d.). Retrieved August 14, 2022, from <http://automation-beyond.com/2016/08/30/on-white-box-and-black-box-testing/>