Patrick D. Pattison

CS-320 Software Test Automation and Quality Assurance

Southern New Hampshire University

19 June 2022

My unit testing approach was, in my eyes, rather simple. I tried to map each requirement to an individual test case. In doing so, I feel that I achieved close to 100% coverage on the tests. The requirements for the project were fairly simple, so there were not really any edge cases. As an example, let’s look at the Contact class requirements. The required member fields are a unique ID string, firstName, lastName, and phone strings that cannot be longer than 10 characters and cannot be null, in addition to an address field that cannot be longer than 30 characters and cannot be null. To test these requirements I coded up three separate tests for each field. The first test was just setting the field to a known value, and then verify that I can read the value back out. The next test case was to attempt to set the field to a null value, and verify that the Contact class throws the IllegalArgumentException. The final test case was to attempt to set the field to a value that was longer than the allowed character limit and verify that it threw the IllegalArgumentException as well. The only requirement that I was unable to test, involved the ID string being unique. When testing just the Contact class, the class itself wasn’t generating its own ID string, and with the ContactService class testing this would have involved generating every single possible ID string and verifying that it couldn’t generate an additional ID string. The alphabet I chose for the ID string was 80 characters deep, ‘\*’ (42 Dec) to ‘z’ (122 Dec). This would have required generating 8010 instantiations of the Contact class.

Being as I mapped every requirement to it’s own test case, I feel that my test coverage is near complete. Again, I didn’t test the uniqueness of the ID string generation in the ContactService class, or any of my Service classes. However, all other requirements I feel were adequately tested. I ensured that my code was technically sound by using the JUnit Assertion to test that an exception is thrown. Doing so allows to test that an exception is thrown in a completely safe matter, that doesn’t actually cause an exception and therefore end the execution of the code.

The testing techniques that I employed in this project were really only three different assertions. The first assertion used was assertTrue. This was used to validate that a string value in the class was equal to a known value. I used this assertion over assertEqual, because in Java, strings cannot be directly compared for equality, instead one must use the String.equals(otherString) method which returns a true or false Boolean value making the assertTrue assertion the appropriate choice. Another assertion used was the assertThrows as discussed earlier. This assertion was used to test that a specific test actually throws an exception expected without the thrown exception causing the code to stop executing. The final assertion used was the assertAll. The assertion assertAll allows multiple tests to be ran completely despite failures. The tests I used assertAll for could have just been ran all in series, however doing it that way, if there were multiple failures the test would have stop as soon as the first failure occurred. By using assertAll the test case will actually run all tests, and report all the failures.

The additional assertions that were not utilized during the coding of this project were many. The assertion assertArrayEquals is used to validate if an array is populated as it is expected to be. The assertion assertDoesNotThrow will test that a piece of code runs successfully without throwing any exceptions. The assertion assertNotEquals will test that two values are not equal. The assertion assert False is the opposite of the assertTrue I used. This one will validate that a Boolean test is false. The assertion assertInstanceOf would be useful in testing a factory class, as it will test an object and validate that it is an instance of a specific class. The assertion assertIterableEquals will test an iterable and validate equality. The assertion assertLinesMatch test is useful when multiple lines of text must be validated for equality. The assertions assertNull and assertNotNull will validate a return value is or is not null respectively. Finally there is an assertTimeout test that is useful to validate that a piece of code doesn’t take too long to execute.

While working on this project the mindset I adopted was very similar to the mindset I adopt at work. I have always been the type that would rather find my own errors then have them pointed out to me. As such, I try to systematically make sure that all requirements are tested if possible. My process is fairly old-school, in that I prefer to actually print out the requirements, and actually mark up the printout while I write the tests. That way, for me, it is easier to validate that my tests fully cover the requirements. As far as limiting bias, that is an exceptional hard task to accomplish. Especially when it comes to reviewing and testing your own code, it is much like your own research papers. You know what you intended to code, so small gaps are easy to overlook. It is for this reason that most professional software engineering groups, like the one I work for, have a dedicated testing group. This way the testers are completely separate of the development process. Software Engineers should still write and use Unit Tests, but having a separate testing group helps to ensure that no bias was injected into the test process. This is also why professional software engineers will have code reviews, not just of the code they wrote, but also of the Unit Test process. This allows other engineers to review the test procedure and verify that no gaps are present.

As written by Humberto Farias in InfoWorld, “Good developers test early and often. Errors happen in every project, and they’re easiest to fix when identified early. The cost of fixing a bug found in the late stages of a project are 30 to 100 percent higher than the same bug discovered in the first stage.” (Farias, 2018). Basically this is saying that the later in the Software Development Life Cycle that a bug is found, the more expensive in time and money it will cost to correct the bug. It is absolutely important to maintain vigilance while Software Engineering not to cause bugs. Too buggy of code is worse than not having the code in the first place. In addition, by passing bugs from version to version is one way to accumulate what is called technical debt. Just as you wouldn’t want to have to fix a bunch of bugs caused by someone else using sloppy coding practices, nobody wants to have to fix a bunch of your bugs caused by sloppy coding practices.

**References**

SNHU. (n.d.). *Project One Guidlines and Ruberic*. Project One Guidelines and Ruberic. Retrieved June 19, 2022, from https://learn.snhu.edu/d2l/le/content/1069803/viewContent/18229364/View

Farias, H., & Farias, H. (2018, July 2). *Where you should never cut corners in software development*. InfoWorld. Retrieved June 19, 2022, from https://www.infoworld.com/article/3286265/where-you-should-never-cut-corners-in-software-development.html