Jacob Sullivan

CS320 Project Two Final Paper

For my final project, I revisited my past assignments involving my task service and contact service classes and applied the same testing techniques I implemented in my appointment service class. As I noted in past journals, much of my testing utilized the white box testing technique known as decision testing. As a refresher, decision testing is a scrutinous testing of all decisions made by a program during runtime (Hambling et al, 2015). I also made use of a black box testing technique known as decision table testing to plan out my tests by drawing up some decision tables for my past two assignments, as well as a flow chart for each outlining how the program would function given certain inputs. Decision tables are tables drawn out to detail how a program will handle all possible combinations of input, both valid and invalid (Hambling et al, 2015). I used my appointment service project as a base to revisit and revise my task and contact service classes and, using the aforementioned techniques, was able to attain a testing coverage of 90.6% on my final project when combing all service classes and their respective object classes into one program. I would credit my success largely in part to my utilization of decision table testing, as drawing everything out on paper gave me a clearer vision of the overall program and allowed me to be more precise with my decision testing methods. Using a breakdown code after each test by emptying my created array lists in each class to start fresh also contributed to this value, as before I had implemented code to conduct said breakdown, I was getting spill over from past tests, preventing the testing code from reaching certain lines in my classes.

The two above techniques can both be considered examples of functional testing, which is testing based upon the specifications given for the program to have (Hambling et al, 2015). The client set forth limitations as to what each attribute of each object should have. Many of these specifications were limitations on how many characters could be in the passed string values. None of my attributes were allowed to be null, no duplicate identification numbers were allowed, and more specific to the appointment service classes was the request that the program not accept date that had already passed to be used as a date for a created appointment object. My testing was virtually based completely on these specifications, as one would see in reading through my written Junit test cases for each class. It appeared to me, given the specifications, that the goal of my testing should be around validation of passed values for the attribute of each class, as the client did not specify much else other than what an object’s attribute could and could not be.

I did not make much use of nonfunctional testing methods as this was not a complete program and I did not feel it was necessary to apply such testing methods to these classes. For example, stress testing did not make much sense here because they were simply classes that handled the creation and storage of objects. There was no main method to execute that would implement each class, so testing how all of these classes would work in tandem in a main method under heavy stress seemed more of a future test to be run on a more complete program than on my current version. Without a main method, I could not test how this program would interact with a user to the extent I felt necessary to justify the use of nonfunctional testing methods. However, concerning other software development projects, nonfunctional testing is a necessary part of the testing process as it tests a systems behavior when encountered by end users, who are the prospective consumers and the reason the software is being developed in the first place. If a company developed a software to handle online banking but failed to test the accuracy of functions or how the system handled thousands if not hundreds of thousand of users using it at once, the results could be quite destructive. Banking information is a very sensitive classification of information, and to release a system to handle the utilization of such personal information that could not perform correctly could end in improper handling of transactions, loss of funds, crediting of funds inaccurately, or the entire system crashing and no end users able to access their money, to name a few scenarios. The basis of these concerns applies to any software meant for mass consumption. If a program does not function properly under stress or is rendered unusable by a user simply interacting with it, the software, regardless of its function, would be considered a failure. Failing software means less customers, bad reviews, and detrimental financial implications for the producing company.

In regard to my code being technically sound, I can say that I am confident in my delivery possessing that quality. Exceptions are thrown when the need to be, valid and invalid inputs are tested, and a breakdown after each test is performed to ensure spillage between tests does not occur. One such line in my code that adds to this aspect is below:

Graphical user interface, text

Description automatically generated

I included this block in each of my test classes for my service classes. This block ensures that after each test is performed, the array lists I created in each service class are emptied so subsequent tests start with fresh data. In each of these service classes, I also utilize tests for valid inputs for all attributes, as well as specific tests for each attribute in which I provide invalid inputs. For example, in my contact service class, a contacts address cannot be null or longer than 20 characters. Based on these specifications, I wrote a test passing a null address to my constructor, as well as a test passing a 25 character string for an address. In both tests, I instruct the compiler to assert an exception is thrown upon the constructor receiving invalid input, shown below:

Graphical user interface, text, application

Description automatically generated

I utilize the above method in all of my Junit test classes to ensure all attributes passed to the constructor and methods to update those attributes handle invalid input of any nature correctly. Not only do I validate the handling of bad input, I also test the success of functions given valid input to ensure the program works as desired.

In terms of efficiency, however, I am not so confident. This is largely in part due to the fact that this is my first experience with Junit tests and, as was the case with learning coding languages, I am unaware of ways that some of my code could be consolidated. I made no use of methods such as nested testing or using tags to group tests as I was not comfortable with their use when completing the assignments. Looking back, I’m certain that with a little more research and some practice in my own time, I could return to these test cases and perhaps rewrite them in a much more efficient manner, eliminating a large amount of redundant code that I am already aware exists in them, such as the code below:

Graphical user interface, text, application

Description automatically generated

This is simply an example of one of many tests I wrote that begins with the creating of a valid object of the appropriate class. After finishing the project, I couldn’t help but wonder how I could eliminate this redundancy that I had written over and over again into a much simpler, yet still just as accurate, manner. I do feel that, for a first delving into the world of unit testing, I did a proficient job. However, I cannot say that my tests are as efficient as they could be because, in truth, they aren’t. With time and practice, though, I know this will improve.

Building upon my explanation of testing focused on decision making, I feel this is a vital building block for any software development process as it tests the very nature of what every software is built to do. That is, take in input from a user, process said information, make decisions based on that input, and react in a way the user desires. If decision testing was not implemented in the development of any software, the resulting product could be wildly inaccurate if it even functioned at all. I would be impressed if someone was able to bring to me even one piece of commercial software that, at no point, makes a decision using user input. It’s one of the main points of any produced software, which is what makes decisions testing that much more important to any software being produced.

I approached this project with a very practical mindset. I was given a set of requirements for attributes that would be assigned to an object. This seemed, to me, very straightforward. If all the attributes met those requirements, they may be used. If they did not, the program must throw an exception. If I am being honest, I did not really use much caution in my approach as the end goal here was simply doing my best to achieve an A. I feel, while this is an important aspect to me in my current pursuit of a degree, that the implications of a lower yet passing grade did not prompt the caution I feel the prompt is referencing. To elaborate, I feel this caution will be much more prevalent when developing code with real world applications. If one’s job is on th line, the level of caution would of course be much higher as their livelihood is at stake. If the software is being designed for a system that directly affects many others, that, too, would elevate the amount of caution used in writing the code. The straightforward nature of the assignments from this course also did not prompt much of a cautious mindset in my approach as they were very direct. “Here’s the prompt, here’s what it should do, here’s what it shouldn’t do”. With a more vague prompt, I feel I would have been a little more cautious as it would have prompted me to think a bit more critically about what the assignment is asking of me. However, with such blunt directions, it was easy to simply write based on the requirements and, thus, eliminated the element of caution for me entirely. I was following the directions and there was not much else to think on.

I believe this mindset helped me to achieve the 90.6% coverage I mentioned above. However, I did feel that I may have not been able to test as effectively as someone who didn’t write the program themselves. I was feeling, towards the end of the assignments and even now, the bias that I had been reading about for the past few weeks. The issue was that I knew the program because I wrote it. I knew what it should do, how it performs those tasks, and how it handles invalid input. Thus, I wrote my tests to interact with my program in the manner I knew would be successful. In my opinion, my testing of my own software was still an important piece of the overall development of this software if it were to continue. If given to testers with no knowledge of the inner workings of my classes, I feel it would be a better “starting point” for their tests as it is a properly functioning code. The big hindrance I kept feeling, however, was this bias. I could not, hard as I would try, put myself in the viewpoint of someone who was testing this software and knew nothing about it. Each test I wrote was very specific to the instructions. I had a very difficult time imagining what else to test aside from the specifications put forth by the client because I knew my code worked and that was good enough for me. This was a difficult pill to swallow, so to speak, as it did worry me that I may be missing a number of tests that could and should also be run here. However, I feel that upon entering a team of developers in whatever career I land in, this bias will be much easier to mitigate with more than one person writing, testing, and reflecting on a piece of code. Have team members to help and bounce ideas off will be a very useful tool to mitigate this bias.

Being disciplined in a dedication to the quality of a product is not unique to the software industry. This is one of the most important aspects of producing any item meant for consumption by customers. The point of any production for this purpose, to be blunt, is economic success. If a company cuts corners and pushes out a faulty, hard to use product, no one will purchase it. This has been exacerbated by the rise of the internet and social media due to the excess of reviews and easy access to said reviews. The release of a bad product due to lack of care about its quality will lead to bad reviews, which leads to lower sales, if any. What would be the point of developing a product that will not succeed? All it would accomplish is a waste of resources and a failure to generate any profit. The difference between many products and software, however, lies in the purpose of some software. Banking information, on-board vehicle systems, rocketry, chemical plant ventilation systems; all of these involve software to help automate processes that many institutions don’t have the manpower to perform manually. Imagine if a company produce a software to automate a chemical plant’s ventilation system but haphazardly developed the software. In cutting any corners into a software that is being designed for the distinct purpose of protecting the life of those working in the plant, the results could be disastrous. The same goes for a banking system that was developed quickly and cheaply and, through trying to pinch pennies, may be released with security risks that weren’t seen in the midst of trying to save money by shaving off some of the testing rounds on the software before release. Software often handles very important information and processes and, as such, should never be written without dedication to a safe, functional, and easy to maintain system that benefits users, not harms them.

Sources:

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - 4.6.4 Decision Testing and Coverage.* BCS The Chartered Institute for IT. Retrieved from  
<https://app.knovel.com/hotlink/pdf/id:kt00UC2JF1/software-testing-an-istqb/decision-testing->coverage