CS 320

Professor Kalysa Wilson

Michael Devin Moore

9 December 2023

**Project Two: Testing Summary and Reflection**

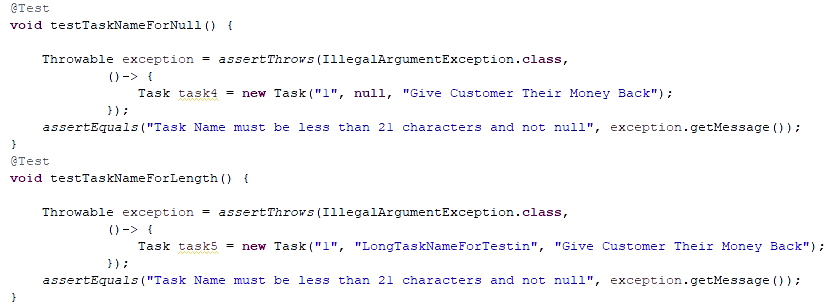
**Summary**

Testing Approach:

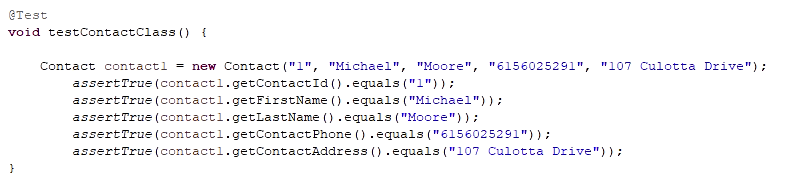
* To what extent was your testing approach aligned to the software requirements?

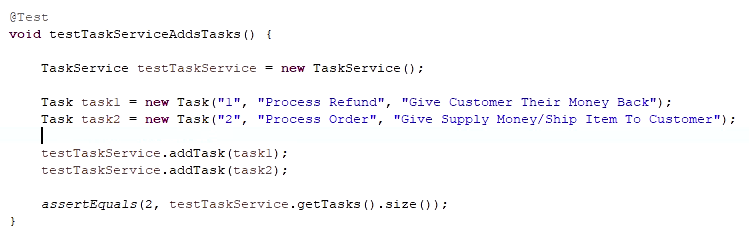
I wrote three sets of two classes and associated JUnit tests to show that I was meeting the requirements for Project One. The first class pairing asked for a class, called contact, that held 5 variables with various length requirements and another requirement that none of them could be null. One specific variable also had to be unique and non-updateable, therefore, could not be repeated or changed. Another variable had to be a specified length. Additionally, the class pairing called for a second class to act as the service medium for adding, updating, and deleting the contact class objects from a contact list. The second class pairing asked for similar requirements but did differ slightly in the requirements. The first class of the second pairing, called task, held 3 variables with one being unique and non-updateable as before. The 3 variables had character length limits and could not be null, similar to before. The task service had to be able to add, update, and delete task objects from a task list. The third class pairing was called appointment and also had corresponding service class. The variable requirements were similar with the addition of a date data type for scheduling appointments, only in the future, it was not allowed to be null or contain a date in the past.

Firstly, I test each individual variable, of each of the base classes, to verify it meets requirements. I perform this testing one variable at a time, one requirement at a time. Each variable had a length limit of some sort, had to be after the present date, and could not be null, so a test is run to test each variable’s requirement individually similar to as shown:

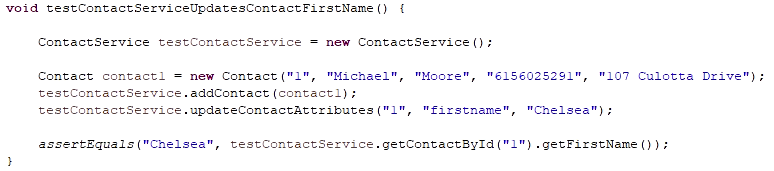


Once I have tested the base variables, I move on to test the service class which will hold a list of the variable holding class. The test approach tests each service class such that it can store the classes in a list. Shown below are screenshots of JUnit code for testing the contact service class as mentioned:





For the contact service and task service I tested for the ability to add, remove, and update the corresponding class type for each of the two services. The appointment service did not call for the ability to update data. An example for the task service was shown above for adding tasks and an example for the contact service updating bariables is shown below:



* How do you know that your JUnit tests were effective on the basis of coverage percentage?

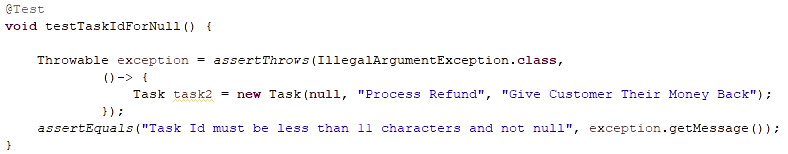
I know that my JUnit tests were effective as all tests passed. As I wrote a test for each individual requirement of each variable and class, the JUnits are my proof that I met them. I wrote my JUnit tests to be at least 80% test coverage of the corresponding class. Several of the JUnit tests achieve 100% coverage and my lowest coverages are for the different service classes at about 87%. That 87% comes down to portions of the code with multiple choices not being tested for null when different choices are selected, but the null value is already tested as part of class variable requirements, so I found it not pertinent to test for null there. All of the tests I ran ended up with 100% passing of all tests which is proof of meeting the requirements. All 6 of my JUnit test classes resulted in all passing runs with no errors or failures. Shown below is the appointment and appointment service classes JUnit test coverages respectively:

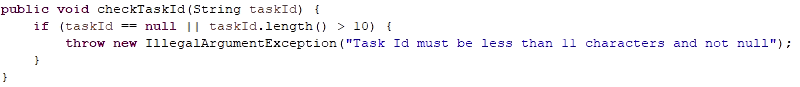




JUnit Tests:

* How did you ensure that your code was technically sound?

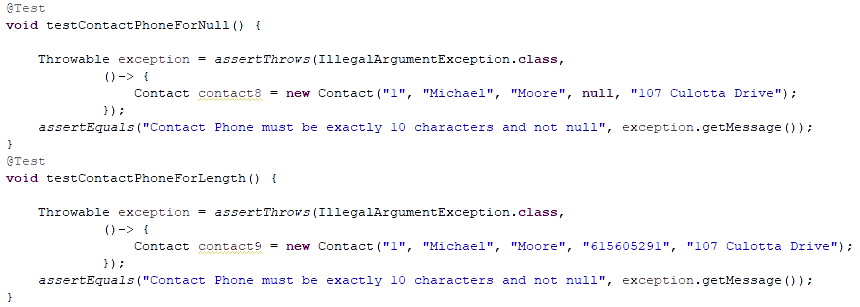




My JUnit code is technically sound as I tested for only one requirement at a time, therefore I could ensure my pass and fail criteria, or what I was verifying, could not get mismatched results. For example, if I were to write tests that searched for multiple criteria at a time, a singular test pass or fail may mask another pass or fail in a combined test. In my level of experience, I find it prudent to separate tests such that I do not have the possibility of one JUnit test, with multiple points of testing, be setup poorly and reflect innacurate test results. Shown above is a JUnit test for taskId. The JUnit creates a task object with the taskId that is null, which is not allowed. It references the task class method to create a new task, which references a length and null check on the data input for each of the variables, taskId included. The second screenshot shows that if taskId is null or length is greater than 10 characters, it will throw an exception that I made the JUnit look for, as an invalid taskId value was utilized in the JUnit. That is the general scheme of how I enacted all of my tests.

* How did you ensure that your code was efficient?

I made sure my code was efficient by minimizing the number of tests I had to run to prove that my code was within requirements. Each variable had to be not null and under a certain length, or at a specified length. A test was run for each variable for those two requirements, that of not null and length. The special variables that had to be unique values and non-updateable were tested for such. The service classes had to be capable of adding, removing, and updating variables for the classes held within. These were each tested as simply as possible. There would be no benefit to testing variables that do not meet requirements more than once. A variable that is 1 character over length limit or 100 over does not need to be tested, that would provide no useful data. Creating tests that show data within requirements passing and then data just outside of requirements failing meets the intent of proving a program’s abilities to filter input within specified limits. A test should be developed to test each requirement individually and then done once, that keeps the testing shorter and more efficient. Below is another example of how one variable is tested for each of its two requirements and only done once as more testing is not necessary unless requested:



**Reflection**

Testing Techniques:

* What were the software testing techniques that you employed in this project? Describe their characteristics using specific details.

From Software Testing, by Hambling et al. (2019), there are three main types of testing techniques. Those types of testing techniques are Black-box testing, White-box testing, and Experience-based testing. Within those groupings there are a handful of techniques that fall under one of the three categories. In the Black-box techniques, which are techniques meant to verify requirements and specifications of the program, I utilized the boundary value technique. This technique is carried out by checking that data passes when within a boundary specified by program requirements. More specifically, for the JUnit testing I accomplished in the last three modules I was to test that variables were shorter than specific lengths and not a value of null. I ran a test that would pass if all input data was valid, and it did. I then ran tests, changing only one variable at a time, setting them to values that would not pass. I would set the length of variables one value above the allowed maximum and expect a fail, and then run a test where it was null and expect a fail. From the combination of these tests, I verified data just out of range would fail and data within would pass.

I also utilized a White-box technique, which are those specifically meant to test the program that functions execute and interactions between components happen correctly. For my work in the past three modules, I had to not only create a class that held variables, but also make a service class that allowed for creating objects of the original class, storing them in a list, and either updating or deleting said objects. I was to verify those capabilities via JUnit tests, which is in line with statement testing. It would run specific lines of code that I chose and verify the proper results afterwards. The whole intent of statement testing is just that, testing a statement via verifying output results were what they were expected to be when ran through a particular statement or method. This type of testing was done for all portions of the class and service class which verified all of the methods would run correctly.

* What are the other software testing techniques that you did not use for this project? Describe their characteristics using specific details.

I did not utilize a whole handful of techniques, but they are useful in the proper situation. From Black-box testing, I did not utilize equivalence partitioning, decision table testing, state transition testing, or use case testing. I did not utilize equivalence partitioning as the string variables were bound to either a character value, such as 10, or to be less than 50, for example. The variables then were bound between 1 and 50 characters as 0 characters would make it null, which variables were not allowed to be. The point of equivalence portioning is to test values from generally middle regions of a range of numbers, such as testing -5000, 0, and 5000 in an allowed range of -10000 to 10000. The bounds of this assignment did not warrant this type of testing, but it could have been done by testing a 50 length, or other maximum values from the modules, passed as a 1 length variable did. The modules did not have any decision making for the program to do, so decision tables would not have been possible. However, decision tables are developed when logic exists in a program such that varying input changes how a function or program executes, thus changing the resultant output. They can be developed and then help guide testing to verify logic meets intent within the program. State tables were not utilized in my testing, but they can be utilized to generate test cases to check functionality of a program is behaving a certain way when a specific input occurs in a dynamic system. I also did not utilize use case testing, as the intent of the modules was to utilize JUnit testing on variables and class integrations. However, use case testing is vital to ensure that each different type of user can perform their specific actions on a program or system, and it largely done on large scale systems with many different types of users.

From White-box testing, I did not utilize decision testing. I do not believe I utilized it as there was no decision logic within the program to work with, other than that of testing my inputs for validity in length and being not null. Either of those would have resulted in a fail or a pass, but that was to verify input was valid per requirement, which White-box is not meant to verify requirements are met in the code. They are meant to verify program logic itself. The decision testing itself is utilized to verify a program takes the correct course of action when varying inputs and choices are fed into a program. It is to verify the correct actions occur when certain inputs are made and that the resultant output also makes sense from those executed actions of the program.

Lastly, I did not utilize and experience-based techniques as I have no experience in this regard and I was to just verify variables and class integration. The experience-based techniques rely on the tester’s past experience to look in areas that would commonly have problems and develop test methods to look for inadequacy in meeting specifications. Experience-based techniques are those of error guessing, exploratory testing, and checklist-based testing. Each of these could not have been utilized in such simple programs such as the ones I made.

* For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.

As previously mentioned, Black-box testing has five main techniques consisting of equivalence partitioning, boundary value testing, decision table testing, state transition testing, and use case testing. Black-box testing exists to prove the meeting of requirements and specifications. The tests formulated are formulated such that the end result shall prove that a program requirement or specification is met. Equivalence partitioning and boundary value testing is done to make sure that the program reacts accordingly to different inputs and other data handling. If the input is within what the program is meant to accept, it shall take it and utilize it whatever way it is meant to. If it was out of range or invalid, then a proper response will occur to inform of such. Decision table and state transition testing are meant to be performed by analyzing program design products and code and discerning how the program is to behave under dynamic choice and result situations. They both involve drafting a table to aid in presenting logic such that a program can be tested to determine whether or not the program logic meets the drafted tables, which then meet the requirements for the program. This type of testing can only really be performed with dynamic cause and effect type programs, implementing this in the modules the past weeks would not make sense for storing a couple variables in a class object in a list. The use case testing is performed to verify that different types of users can in fact perform the functions that they are meant to be able to. They should be able to utilize the program to meet the intent and purpose of the program based on what type of user they are. The use cases guide the use case testing to develop testing methods that effectively pretend you are one type of user or another trying to perform all functions they would need out of a program.

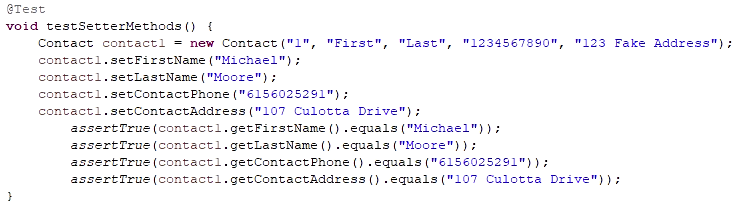
White-box techniques test that a program behaves within what is expected without regard to any specification. The techniques that fall in this branch are statement and decision testing. The statement testing is verifying that a statement being executed does what it is supposed to. This is extremely important for any program as it proves that the program does what it is built to. The decision testing verifies that the logic within the program for handling certain inputs, choices, or data is sound and executes as desired. If logic is found to not be working properly, it can throw off all other portions of a program.

The experience-based techniques were all roughly the same, but they were varying approaches on letting experienced testers take the reins on finding ways to test a program utilizing their experiences, resources, and each other. The error guessing technique relies on the tester to develop tests that are more unique or specific to a program and mainly involve testing suspected weaknesses or checking up on problem areas in a program. The exploratory testing relies on the tester to test around to discern where specifications are not being met or are inadequate in fully meeting them. It involves a structured test charter to get results in a short time to maximize knowledge accrual about the program. The checklist-based testing then relies on the testers to follow checklists decided upon from different resources and their own experience to guide their testing and their development of tests. These checklists also typically are used for both functional and non-functional testing.

Mindset:

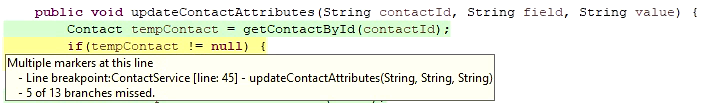
* Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution**? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.

I employed caution when I was finalizing up my Project One for submission. All I had to do was get my coverage percentage above 80% as per requirements of the project. I found that I had some untested methods for some of my variable classes. I was trying to rationalize not testing them as I was already above my requirements in terms of coverage percentage. I also test these specific methods when I am testing the corresponding service class as well, so I thought against adding specific tests for these methods again. I then realized I should test each testable thing in its most basic form to accurately assess its performance, so I ended up adding a test for those methods and got my coverage to 100%. Testing things across classes can cause false results if the test is not built well and may cause issues if not tested individually and then later implemented to a release build. The very slight extra work to make the test to check setter methods is shown below:



* Assess the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.

I gave a perfect example of limiting bias above when explaining how I utilized caution in my testing approach which in turn eliminated the bias that my code was all fine and dandy. I do have another example of bias that I left in my code, that I would need to make sure I tested for in a similar role in the future:



This update attributes method from my contact service class was not explicitly tested for all of the possible input combinations it could have received. I only tested them in the specific instance all input was acceptable and allowed the method to be carried out. I did this because I tested the setters via this test, and the other test I added in the above section, so I felt it did work as intended when all input was satisfactory. However, that bias led me to ignore the other possible input combinations that could possibly happen. In the future, I will ensure I tests all possible logic paths that the code may possibly take, or at least prove other tests will nullify the need of a more thorough test such as this situation. This issue can lead to leaving a lot of testing out which segways into the final topic.

* Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

Cutting corners while coding or testing can create a myriad of problems from a non-functioning product, a product that misses the mark, or a product that is believed to be effective, but then after release turns out to be useless. None of these situations are where anyone desires to be. Maintaining a strong level of discipline when coding and testing will aid in creating a thorough product that minimizes overall work through the whole design and creation process. If one sticks to industry standards, references documentation, and keeps that high level of discipline, they will ensure a level of first-time quality which then reduces later rework. That is the epitome of reducing technical debt, or time spent reworking a product due to overlooked issues within the product. The fact I did not test all the different logic paths for my above function could have led to many issues for a release build of a product that might have been complete failure of that portion of the program. I would have then had to accept the fact I cut corners and messed up big time, and then had to go back and fix whatever might have been inadequate, test it like I should have, and re-release it. That stands true if I still had my job after that blunder!

References:

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software Testing : An ISTQB-BCS certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.