Derek Kwasniewski

Professor Toledo

CS320 Software Test Automation & QA

15 October 2023

Project 2

For the contact service and task service units, after creating each respective class and having the necessary attributes associated with that class, I took a step-by-step approach to developing the tests. The first test I did was a broad test to ensure that I can create an instance of the class with values that will be allowed in the final product. After creating this first test, I moved on to tests that would ensure that attributes were aligning to the specified requirements. For example, in the latest task service assignment, the first requirement is as follows,” The task object shall have a required unique task ID String that cannot be longer than 10 characters. The task ID shall not be null and shall not be updatable”. To make this easier, I split this line into 4 different test cases. The test cases for this requirement are as follows: having a unique string named ID, ID must be 10 characters or less, ID must not be null, and the ID cannot be updated. The first check happens in the TaskService.java object file in the addTask method. This method is called when we want to create a new task, so to create a new task the user needs to supply an ID number for the new task. To ensure that the ID provided is unique, we check every task currently in the array list. If another task is found with the same ID, an illegal argument exception is created to prevent the creation of the new task with this id. In the Task.java object file for the Task Service unit we started with adding a private id string to help ensure that the value cannot be changed from outside sources. After this, to ensure that the ID is 10 character or less and is not null, we add an if statement in the constructor for the task object with the following condition, “id == null || id.length() > 10”. If the id matches one of these conditions, then an illegal argument exception is created. Lastly, to ensure that the ID cannot be updated I ensured at the beginning that the variable was private, and there were no methods within the class that could change said variable.

To ensure the quality of testing, I went through each function in both classes and created tests for each branch in the functions. After doing this, I ran a coverage test to check that the tests covered all areas of its respective class. Once completed, I checked which functions and/or branches may not have been tested and added them to the respective test file and retested until 100% coverage was achieved.

To ensure that my code was technically sound, I ensured that I was using coding best practices to ensure the readability of my code, for example in my Contact.java class, I used descriptive names and camel case for variables and functions such as firstName string to hold the contact’s first name, or setFirstName() so any programmer can understand what the function is meant to do. I also used comments in areas that may be more difficult to understand what is going on. For example, at line 28 in the Contact.java class, I created a comment to let anyone looking at the code know that the try…catch statement that follows will “Test if entered phone number includes any character that aren't digits “. As mentioned previously, after adding all necessary methods needed for testing, I ran a coverage test to see how much of the file was being tested. I continued to edit/add tests until the file was 100% covered in the test. With the addition of annotations that signified a method used for testing and the use of coverage tests, I was able to ensure that my code was technically sound.

To make sure that my code was efficient, I took a quick glance at the tests completed and made sure that they were not taking too long to complete. Also, during development I made sure to keep in mind other methods created so that redundant code was not developed creating an extended run time that was not needed.

For my 3 milestones, I used the white-box method, specifically JUnit testing to help test the software created. Using the white box testing technique, we can test the functionality of the system as well as the decision flow to ensure it is operating properly. Other techniques I applied to testing the functionality was using the JUnit testing I implemented equivalence partition, specifically the input partition to test if groups that are valid work in the system, and inputs that aren’t valid are properly producing exceptions. For example, the phone number for a contact in the could only have a length of 10 digits. To verify, I tested with values that had less than 10 characters, and values with more than 10 characters, and tested with values that included letters to ensure is returned an exception. Then tested a valid number to ensure inputs were partitioned correctly. As more changes were made to the program, I also utilized regression testing to ensure that changes made to the class were also tested to make sure no errors were introduced. Further levels of testing include integration testing, which tests how different parts of the system work together. The next level would be systems testing which may include testing how the system reacts to user input.

The other testing techniques that I did not invoke for the milestones were black-box testing and experience-based testing. Both techniques have their own pros and cons for use in software development and at different stages. For example, black-box testing is mainly used to test the specification documents for a particular system and the functionality of the system as a user that has no knowledge of the internal workings. Experience-based testing can be done throughout the development of the system by utilizing someone who has experience with a similar system and testing areas known to have been an issue.

Some practical uses for the black-box method could be to test the inputs and outputs of a particular part of the system. For example, we could utilize input and output partitioning to create valid test cases for user inputs. Say we are developing a system that creates an invoice for a 3D printing company. If the product uses less than 300 grams of filament, a base charge of $7.50 is applied. If the product uses 300 and <600, a charge of $15 is applied and if it uses 600+ a base charge of $30 is applied. Using this information, test cases could be created to test inputs in the valid input partition: <300 grams, 300 grams to <600 grams, and greater than 600 grams. Input values within these partitions would be expected to produce values of $7.50, $15, and $30 respectively. White-box testing uses would include testing the functionality of the code in detail such as decision flow. By using flow charts, we could create test cases, so the expected branch is executed with the correct output. We want to keep this simple so it may be best to only create a flowchart for a particular method at first. To create a flowchart, we would want to work our way through the method being tested and include all the decisions the code must make to show the different branches of decision it could make. Using this we can create test cases to test the functionality of the method to make sure the method is producing expected results. Lastly, we have experience-based testing which could be used in a variety of projects and in many parts of the development process. Using the experience of a given person or team, we can ensure that areas that have been prone to cause issues are tested thoroughly to make sure no errors are produced. For example, when working on a project, we will have a list of areas that should be tested as they may be subject to producing errors. However, Person A, who is an experienced programmer has created similar systems to this one and feels that a particular part of the code should also be tested as it has been known to cause issues in previous systems they have worked on.

While acting as a software tester, I applied caution diligently and evenly by making sure that as much of the program was tested as possible. While creating unit tests, I would constantly make sure to run the tests to make sure there were no errors, as well as run a coverage test to ensure that more of the system was being covered. I feel appreciating the complexity and interrelationship of the code is important because having code that is less complex but still related to each other, allows you to know that if a part of the code uses function X, and the test passes, then you could be almost certain that if another part of the code using function X fails the test, that it is something with that part of the code instead of function X which allows you to narrow down where the error could be.

For me, limiting my bias towards my own code when testing wasn’t very difficult. I know that if I am biased towards my own code when testing I may miss out on things that could be potential problems later which for right now, only has an effect on my grade, whereas if I were to have that attitude towards testing my own code in the future, could be detrimental depending on what the software is to be used for so if I find something wrong when testing, I take the responsibility and correct the issue.

I feel it is important to not cut corners in general as doing so will most likely have you going back and redoing what you did the correct way anyway. Whereas, if you don’t cut corners and do it properly the first time, you can walk away without having to worry about not doing it wrong the first time and wasting time later. In programming, this is also very true as some software being developed could be for systems that may hold confidential information or may control something that could cause possible injury if corners are cut and not all areas of the system are addressed in testing.