Jordan Davis

Professor Joseph Rangitsch

CS-320-T4515

April 17, 2022

Project Two: Summary and Reflection

To understand what types of tests I needed to run, I looked at the description of the assignments to see what specifications needed to be followed for each variable and function listed. For example, for the ContactService class, the contact phone number entered by the user could not contain any type of characters except for digits and had to be exactly 10 digits long. For this specific variable, I ran four separate tests, each using assertThrows to verify that the incorrect Contact creation would set off the IllegalArgumentException. These IllegalArgumentException checks were implemented in the constructor of the Contact object so the very creation of the object would set off the exception if the user entered invalid information. The JUnit tests checked that a phone number that was too long, too short, had a null value, and had non-digit values were not accepted by the program. For the length tests, I had the contactPhoneTooLong (line 75 of ContactTest) JUnit test run with a phone number of eleven digits and then I ran the contactPhoneTooShort (line 83 of ContactTest) JUnit test with a phone number of nine digits. This was just to ensure that the length closest to the boundary would still set off the Exception. This type of testing is what was implemented for the rest of the variables in both the Contact and Task classes, with slight variations based on what was specified for the variable.

For the Appointment Service feature, there was a different type of variable that had to be treated specially. The classes needed a variable to store a date entered by the user for the appointment. At first, I tried using the Date object to store this variable and modify it for the tests. But my tests kept failing due to the way that I was trying to use it because it would fail the specification of the check for the date being before the current point in time. So, I instead used a Calendar object to store this date. The Calendar library allowed me to modify the test variable for the date to a future day by adding an additional month to the current date. This allowed all tests to pass and allowed for a more specific date to be set by the user for their appointment.

My coverage for the entire project, that includes the 6 running files and the 6 test files, is 89.4%. Every important part of my functionality has been tested through my JUnit tests. The 10.6% not covered in my program is if statements that were put there for contingency. It is a good thing that they are not run because that means that everything is working as it should. One if statement not run is in the AppointmentService (line 56), ContactService (line 64), and TaskService (line 62) classes and it is to check if an entry was not deleted from the ArrayList holding the specific object of that type of class when the delete function for each class is run. It does not run because the function operates as intended and deletes the object from the ArrayList correctly. I incorporated these if statements to add a layer of security to my code that would ensure that the code ran as I intended it to and not cause a major error if it did not.

The main software testing technique utilized in my code was the use of JUnit tests to test the program’s functionality. As stated above, these JUnit tests helped ensure that my program met the required specifications of the client. They check to make sure that no matter what input the user puts in, that it will determine if the input is valid for that variable or invalid. The JUnit tests checked if my functions that performed actions such as adding, deleting, and finding objects worked. They also checked that the simpler getters and setters accurately worked and did as they were intended. My JUnit tests used assertions, which tests what we expect from the program and what the program actually does (García, 2017). For example, I used the assertTrue function to test that the constructor of the Contact, Task, and Appointment classes properly assigned the variables with the user’s input. I used the assertThrows function to test that if invalid input would trip my set IllegalArgumentExceptions that were set in the constructors if input that did not meet the required specifications was entered. Through these JUnit tests, I made sure that the program would not accept null values, values that were too long in length, and values that did not have the right type of characters. JUnit testing is a good technique for any program to test its functionality and to see that your program adheres to specifications.

Another aspect of the JUnit tests that I utilized for my testing techniques was repeated tests. I implemented these repeated tests in my contactServiceTest, taskServiceTest, and appointmentServiceTest classes. For example, in the taskServiceTest class, the repeated test at line 77 runs through both the addTask and findTask functions of the taskService class. The addTask function is run 100,001 times for each of the forty repeated tests. This makes sure that the system can handle this many objects and that it does not falter with IDs being so close together. Line 85 of taskServiceTest is to gain coverage in the findTask function of the taskService class. It allows 100% coverage to be achieved for that function. These types of tests are good to test your program for repeated actions and to test it in more strenuous circumstances than just a single instance test.

There were some techniques that I did not implement in my program. Firstly, I did not incorporate assumptions into the testing. Assumptions only run under certain conditions (García, 2017). I had no reason for some of my tests to not run; I wanted all of them to run to test my entire program. In more complex systems with more options, this would be an ideal way to test certain aspects if needed for those standards. Next, I did not fully utilize boundary analysis in my JUnit tests. As stated above, I utilized it for the phone variable tests in the ContactServiceTest class. But I did not implement this strategy for other variables, especially the ones that involved strings. I did not see the necessity, but it is certainly a good strategy for integer type of inputs to make sure that the program runs correctly even with input that is supposed to be invalid but is close to valid input. The system can get confused and accidently accept that invalid input and that needs to be tested. Finally, I did not intentionally create my tests with equivalence partitioning in mind. The test for the string length for various variables would count as equivalence partitioning since it is testing the partitions of invalid length. But this was not the main mindset when I created the tests and, as a technique, is not used to its fullest potential. Thus, I do not count it as a technique used in my program. This type of testing would be good for testing integer type input to make sure that all partitions of valid integers is tested.

For this project, I had to think more about how my program was going to be used by a user. It would not be perfect input put in every time with smooth running by the program. Invalid input could cause the program to crash or react in unexpected ways. I had to work with caution to ensure that no matter what input was put in, that the program would recognize what input is valid. I also used caution in my code for the service classes to make sure that no repeat IDs would be used because of the system’s random grab for an ID. And I included an if statement in case the delete function failed to properly delete the object, whether due to its error or the find function committing an error. There were many parts that involved using other classes and that these classes worked harmoniously. I had to make sure that the program ran smoothly through all the running and testing classes. As mentioned above, I had to change my strategy with the variable type for date with the Appointment object classes because the test classes were not interacting well with the Appointment class’s constructor for that variable.

Regarding bias when I was testing, I am aware of my shortcomings as a programmer. I can have trouble with integration and getting some functions to work right or how I had intended them to. I tried to cover my code thoroughly so that I knew that it worked with evidence to back it up. I worked and added lines of code to my tests to increase coverage. An example of this is the added line to my repeat tests (line 85 in TaskServiceTest) that tested the findTask function in TaskService. This added line increased my coverage by covering an additional if statement in that function. I tried to maximize my coverage so that I knew my code worked and was implemented correctly.

Testing with caution, discipline, and with bias in mind will lead to well running code that will stand the test of users and complex runs of the program. This stronger code will also bring peace of mind to the client that the program will not crash on their customers. It can also improve how maintenance is later run on the program by expounding on how the functions work and are utilized by the user. In my tests, you can see how updating a value works and what information a user needs to input to make that happen in a more user-friendly view than the original parameters of the function. If testing is lacking, it will only lead to problematic programs that can easily crash, data leaks, can cause frustration for users, and make maintenance of the program difficult and costly later for different programmers. More testing can cause more time upfront but will save the client time and money down the road because of less need of major maintenance and more documentation of the code’s functions is in place.

**References**

García, B. (2017). *Mastering Software Testing with JUnit 5*. Packt Publishing.