Project Two

Laci Walsh

CS-320-X6135 Software Test Automation & QA

Professor Ramsey Kraya

August 13, 2023

**Summary:**

When determining what I should include in my JUnit tests, the most important thing to consider was the requirements. This informed me about what the program should and should not be able to do. For example, in the Contact Class, one of the requirements was: “The contact object shall have a required lastName String field that cannot be longer than 10 characters. The lastName field shall not be null.” From this requirement, I knew that I needed to include tests that verify the length and nullity of the the lastName field. After receiving feedback about testing the boundaries, I included more tests than I initially had. In my ContactTest.java, these tests included testLastTooLong(), which included a last name with more than 10 characters, testLastLessThan10(), where the lastName was less than 10 characters, testLastEqualTo10(), where the lastName field was exactly 10 characters, and testLastIsNull(), where the last name is null. Two of these tests, the testLastTooLong() and testLastIsNull(), should throw Illegal Argument Exceptions stating that the input for the last name is invalid. The other two tests should not throw any exceptions as they meet the requirements for the lastName.

I used similar tests as shown above to test the other contact attributes such as the firstName, ID, and address. The phone number field had an additional test that the other fields did not require: testing to verify that all characters entered were digits. This was a step that I originally did not consider but was necessary to ensure that the phone number was valid. In the other classes, the Task and Appointment, I also did boundary testing to ensure that the input meets the requirements. In each of the service tests, AppointmentService, ContactService, and TaskService, I included additional testing to ensure that when a user updates information, that information is valid. For example, I included tests such as TestUpdateInvalidName() where an attempt to update the first name field with a name that is more than 10 characters is made. This attempt should throw an exception, where the TestUpdateValidFirstName() test should not, as the two inputs tested are equal to and less than 10 characters. This ensures that a user cannot create a contact with valid input, and then change it so that invalid input is accepted.

The quality of the JUnit tests can be seen in the coverage percentage of the tests. The coverage in JUnit testing shows the percentage of instructions that have been tested. When running the coverage test for my Task class, the coverage is 100%, meaning that all the lines in the Task class are tested in some part of the TaskTest test file. The coverage in my Contact, Appointment, and AppointmentService files are also at 100%, while my TaskService and ContactService files both have a coverage of over 99%. By having such high coverage percentages, I know that the JUnit tests were effective for what the program requires.

To ensure that my code is technically sound, one thing I did was ensure the proper creations of objects as these objects are the base for the connected classes. For example, a Contact object should have an ID, firstName, lastName, phoneNumber, and address; an Appointment object should have an ID, date, and description; and a Task object should have an ID, a name and a description. To test these, I created a new object in my test classes. For example, in the TaskTest file, there is a testTaskClass() test where I created a Task object with an ID, a name, and a description. To test the task name, I used the assertTrue() method to verify that the task name was properly stored and could be retrieved using the getTaskID() method found in the Task.java file. I performed the same tests with the rest of the Task attributes and used the same process to test the Contact and Appointment objects. Another thing I did to ensure my code was technically sound was including comments and using proper syntax. While this sounds like a basic thing to include, without these, the code would be difficult to understand and change for another person.

When ensuring my code was efficient, one thing I did was go through the code to make sure there was no redundant code or code that was unused. For example, prior to my final inspection of this, in my Task class, I had included functions to set the task name and descriptions as, normally, I would have had these in the same section. However, after inspecting the requirements more thoroughly, I realized that the ability to update a Task’s name and description actually comes from the TaskService class and including them in the Task class would be redundant as they would never be used.

Another situation where I increased the efficiency of my code was preventing duplicate objects with the same information. For example, in the TaskServiceTest file, there is a TestDuplicateTask() test where a new Task is created with the name “New Task 1,” and then an attempt is made to create another Task object with the same name. Previously, this would have been possible as there were no checks for this, and it could have potentially caused issues in the future. After realizing that this needed to be checked, I had to modify the TaskService file so that the AddNewTask() method checked the taskList for an existing object with the same name. I included similar tests with the ContactService, and AppointmentService classes.

A third situation where I ensured the efficiency was setting up my code so that it could be reused in other places. For example, when creating the Task, TaskService, Appointment, and AppointmentService classes, I reused a lot of the code that I had written for the Contact and ContactService classes. This included the structure to create the objects, new ID’s and the setters/getters for the attributes as needed. By having similar functions in the requirements, it was much easier to use my existing code, only changed to fit the new class, rather than creating the code from scratch. Having reusable code increases efficiency because the previous code had already been tested to ensure it works logically. I also did this with the JUnit tests, as many oth the test that I had previously defined in the Contact class, such as testing for attribute lengths and nullity, also applied to the Task and Appointment classes. This also provides some starting points for any code that I write in the future that might have similar functionality.

**Reflection:**

Throughout the projects in this course, I used JUnit tests to perform automated unit tests. In unit testing, each unit is tested to ensure the objects and functions of the unit meet the requirements of the project. This is a form of white-box testing as the tester must know how the program should work and its internal structure. By using JUnit testing, the process is automated so that it is faster and produces more reliable results than if testers checked these manually. An example of the unit tests would be the creation of the Appointment object. In the appointment class, the appointment object is created using an ID, a date, and a description and requires all three of these components to be valid for the creation of an appointment. In the AppointmentTest file, there are 11 tests performed that ensure every appointment object will be created with acceptable information. In the first testApptClass() test, the focus is to make sure that the parameters used to create the appointment object are valid and stored in the correct location. Without a deep understanding of how the program works and is coded, it would be very difficult, if not impossible, to accurately test this.

An example of how white-box unit tests are used to test the methods that a class uses can be found in the AppointmentService and AppointmentServiceTest files. One of the methods in the AppointmentService class is AddNewAppt() and is tested with the TestAddNewAppt() test. In this test, the first thing that is checked is that the apptList is empty, which is followed by the creation of a new appointment object. This is tested by checking that the size of the list now is 1. I included the creation of a second object to verify that the apptList gets updated as expected and should now have a size of 2. To ensure that the tests work properly, I used a @BeforeEach function to clear the apptList before each test occurs. This prevents objects that were created in previous tests from affecting the results of the current test, as the apptList would contain more objects that intended. Another benefit of this white-box testing is that the code coverage can be determined. This helps testers know ow much of the code is being tested, if there are any branches that haven’t been covered, and if all conditional statements are properly tested.

Each testing technique is useful for different aspects of a program and should be used according to what needs to be tested. Unit testing is useful in the beginning stages of development to ensure the program has a strong base to build on. For example, if objects aren’t being created properly from the start, this may cause many issues later on. Another testing technique that I did not use in the project is Black-box testing. This form of testing is a test of the functionality of the program and the tester does not need to know the strucures of the code. Black-box testing is useful after white-box testing has been done to verify that the internal structures work properly as it will use these structures to determine if input does not yield the expected results. Black-box testing might also include the testing of the GUI to ensure that it is working properly based on the events that the user will cause, such as clicking on a specific button, scrolling, or clicking on a text box. Black-box testing is not relevant to the testing that the project required because the program is not yet at the level that things such as the GUI can be tested.

Another testing technique that I did not use during the project is non-functional testing of the system. This form of testing includes performance testing, load testing, volume testing, and security testing. Performance testing can determine if the system is capable of handling multiple users at a time and can determine the response time of the system. Load and volume testing are similar in that they both focus on determining how much the system can handle when icreasing a specific part of the system: the load and the data, respectively. During security testing, the security features of the system are tested. This includes having secure confidential information, ensuring this information is accurate, and testing user authentication and authorization. Like black-box testing, these non-functional tests were not relevant to the project due to the low level that was being tested in them. It would not be possible to test the performance or overall security if the program is incomplete.

When working on the project as a software tester, something that I had to do was employ caution when it came to designing the tests. This included appreciating the complexity and interrelationships of the code as understanding how the classes interacted was crucial in creating efficient tests. When testing the service classes, it was important to understand how they used the object classes and how the requirements for the object class should also be implemented in the service class. For example, one thing that I overlooked was verifying the input that is used when updating attributes. As I had already included these tests in the object creation, it did not occur to me that the restrictions could be worked around after the initial creation. In the ContactService class, if a user were to update the first (or last) name field, there were no checks in place to verify that the length of the name meets the requirements, and the updated contact name could be greater than 10 characters long. By understanding the interrelationships of the code that was being tested, I was able to create tests for updating Contact attributes through the ContactService class in the same way that I tested the attributes in the ContactTest file.

Limiting bias is another aspect of testing that can be difficult, especially if the developer is the one doing the testing. As developers tend to get “attached” to the code they create and feel confident in how well it will work, testing becomes more difficult, and it is easier to miss things that should be obvious. In the tests of this project, testing boundaries was confusing as I was confident that my code would allow the creation of objects when using valid input, so I did not originally test for these things. For example, in my original ContactTest tests, I only tested input that was greater than the 10-character limit but did not test for input that was less than or exactly 10 characters. I did the same for the rest of the tests in the other classes, as well, and I had to go back in and include more boundary tests. Had I not been the one that developed the code for the classes, I likely would have automatically included these tests. This is a good example of how it can be more difficult to test your own code rather than somebody else’s.

Being disciplined in my commitment to quality is another trait that was important in my experience as a tester this term. It is important that corners are not cut when writing or testing code because it can leave an opening for errors to occur. As mentioned before, when creating the service classes, I originally did not consider verifying the input that is used when updating attributes. When this did occur to me, it was tempting to just not include these checks because it wasn’t “that big of an issue.” Though after considering it some more, I decided to just include it because it is likely that an error like this would cause an issue somewhere down the line. While it was time consuming to make changes to each of the updating methods in each of the classes and to include tests for these in the JUnit tests, it is better to make these changes now rather than when they cause problems. In the future, I will avoid technical debt by changing my code when I realize there is a mistake and including additional tests to ensure that the program functions as intended. While it might be easier to skip this, if it becomes an issue later on, it will take much more time to find and fix these issues and might cause distrust between the users and the company if there needs to be multiple bug fixes after the release.