**7-2 Project Two**

Amanda Purnhagen

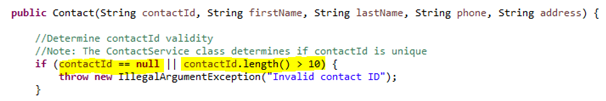
Southern New Hampshire University

CS 320: Software Test Automation and Quality Assurance

Professor Jonathan Norman

August 13, 2023

Throughout my creation of the contact, task, and appointment management application, I aligned my testing approach with the software requirements to the best of my ability by reading the software requirements carefully. For example, the contact service required each contact object to have a contactId that cannot be null, cannot be updated, and cannot be longer than 10 characters.



I ensured that contactId was not updateable by using the final keyword in declaration. As seen above, the code then checks if contactId was null or over 10 characters long. If contactId is null or over 10 characters long, an exception for an illegal argument is thrown which stops the constructor from creating the contact object. To check that only a valid contactId is accepted, the following tests were included in my ContactTest class:

A computer code on a white background

Description automatically generated

Both tests shown above pass if the illegal argument exceptions are thrown correctly, thereby preventing a contact object with an invalid contactId from being created.

Furthermore, I successfully reached a satisfactory rate of test coverage. I will take my task service as an example. At first, the TaskTest class only reached 64.3% coverage of my Task class. I realized that I had not included tests for my setter methods, so I added them. Now, the TaskTest class covers 88.1% of the Task class. My TaskServiceTest class covers 93.2% of my TaskService class and even 82.1% of my Task class.

To ensure that my code is technically sound, I made good use of object-oriented programming (OOP) properties such as solving the problem at hand with the use of objects. I made good use of code conventions with the proper use of indentations, comments, white space, and naming conventions. This can be seen in the code snippet below.

A computer code with text

Description automatically generated

As for my code’s efficiency, I listened to feedback about which data structure I should use to store and manipulate the Contact, Task, and Appointment objects. Originally, I was using a HashSet. After reading feedback, I switched to using a HashMap. This led to more efficient mapping of each object to their unique ID. My use of the HashMap data structure can be seen in the code snippet above.

The software testing techniques that I employed for my contact, task, and appointment services were very similar. As shown below, I imported the relevant packages needed for the unit testing:

A black text on a white background

Description automatically generated

These allowed me to create tests with the basic structure of arrange, act, and assert (AAA). Then, I annotated each separate test with @Test. Next, I used the AAA concept for each test. In the code snippet shown below, the “arrange” step consists of creating two separate, valid dates and an Appointment object containing the first valid date. The “act” step calls the setDate() method with the second valid date. The “assert” step uses the assertTrue() method to confirm that the date associated with our Appointment object is in fact the second valid date.

A computer code with text

Description automatically generated with medium confidence

Sometimes the “assert” step of my tests used Assertions.assertThrows() to show that I expect an exception to be thrown due to there being invalid input. Besides that, the only time my testing really varied from this is when I was required to create LocalDate objects in my appointment service to check whether an input date was in the past and therefore invalid. An example of me creating a LocalDate object for a test can be seen above. I also used the testing technique of checking my tests’ code coverage.

There are other software unit testing annotations I did not use for these milestones such as @BeforeAll, @AfterClass, @RepeatedTest, and @ParameterizedTest. Personally, I felt that these annotations were unnecessary for unit testing in this scope. However, these annotations can be very useful in different software development projects where significant decision coverage is needed. In that scenario, tests are run multiple times with different decisions. There are also other software testing techniques I did not use such as mocking. Mocking allows you to mock the dependencies needed for the class being tested. This strategy allows you to isolate the testing of a specific unit. This would be exceedingly useful in situations where the unit being tested has a dependency that is complex such as an API or a database.

I also did not employ integration, system, acceptance, or maintenance testing techniques. Integration testing is how software professionals check whether all components that make up a system work together as expected. System testing looks at how the system functions as a whole. Acceptance testing makes sure the system satisfies the initial requirements of the end users. Maintenance testing checks whether patches or additions to the live system created any bugs or broke any pre-existing functionalities. These software testing techniques would be useful for multifaceted development projects in the real world. When you want your system to meet the requirements of end users when it goes live, it is highly recommended to do integration, system, acceptance, and maintenance testing.

While working on this project, I adopted an unbiased and cautious mindset. Over the course of the project, I learned new ways to test some of the interrelationships of the code. For example, it was brought to my attention that my set() methods were not properly configured. If we look at my appointment service, I originally had tests in the AppointmentServiceTest class checking the validity of updated attribute values. However, there were no input limitations set for those new values. Therefore, I added some in the set() method in the Appointment class to prevent a valid Appointment object being created and then updated to include invalid values, as shown in the code snippet below.

A screen shot of a computer code

Description automatically generated

As demonstrated above, it is vitally important to remain unbiased while testing software. If you are testing your own code, you must not be overconfident. For example, when I began to code for this project, it was tempting to believe that I knew my code and all its potential flaws. Nevertheless, I was required to test functionalities I took for granted and to push my software to the limit. This led me to discover important shortcomings. Practicing discipline by refusing to cut corners, being meticulous, and being unyielding in your testing methods will prevent the creation of technical debt. These lessons are an invaluable addition to my tool belt, further preparing me to begin my professional software career.