Daulton Truett

08/13/2023

CS-320: Software Testing Automation, QA

Southern New Hampshire University

**Project Two**

1. **Summary**
   1. Describe your unit testing approach for each of the three features.
      1. **To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.**

Writing good tests is a critical aspect of writing good code to produce a program that aligns with the requirements that the software is intended to meet. My approach to testing was directly aligned with the software requirements for the Contact, ContactService, Task, TaskService, Appointment, and AppointmentService classes as the I created the tests to specifically address the requirements that these classes were designed to achieve. These classes model specific objects and perform very focused tasks because they are smaller parts of an even larger program that will be created. The Junit tests I created had these focused tasks in mind and aimed to test the classes in order conform that they resulted in the desired results while also confirming that the classes would perform as expected in the event that undesired results were presented. In my TaskServiceTest.java file on lines 8 – 17, I have written a test that ensures that the TaskService class is able to add a new task to the in-memory data structure array list. This test ensures that requirements are met because the test validates that a new task object can be created with the right attributes from the Task class and that the task object is able to be stored within the in-memory data structure for later use in the program.

* + 1. **Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were effective based on the coverage percentage?**

The Junit tests that I created for all classes in the application I believe were good quality tests that all passed without error and had optimal coverage. When running Junit tests, an option is available to run the tests under “coverage as Junit test” which produces the percentage of the code covered in the .java file it is testing against and displays a visual color over the lines of code it tested. The color coding will highlight the code green that was successfully covered, yellow will show missed conditional branches, and red will show code that failed to run under the Junit tests. It is considered good practice to ensure that Junit tests produce a minimum of 80% coverage and in my case all tests achieved greater than 90% coverage for the classes they were designed to test. When conducting Junit tests there will also be a test results window that will display which tests passed, failed, and any errors encountered. All of my Junit tests passed without error for the entire application.

* 1. Describe your experience writing the JUnit tests.
     1. **How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.**

Ensuring my code was technically sound was based on following industry standards and best practices. The methods I followed to ensure industry standards and best practices involved avoiding writing complex code, implementing readable code, providing insightful comments where required to enhance understanding of code infrastructure, initializing all variables correctly and using all variables declared in the application, and reducing code repetition.

Writing overly complex code does not make an application more impressive, in fact it makes the application harder to maintain for future developers who may maintain the code and potentially increases runtime and memory complexity in certain situations. In my “AppointmentTest.java” class file, lines 9 – 15, is a Junit test that creates an Appointment object from the Appointment class and ensures that the Appointment object has been created utilizing a series of “assertTrue” methods for each of its attributes. The method is simple, has very few lines of code, and achieves the intended purpose of the Junit test. All of my tests follow this format which provides a simple, yet effective, suite of Junit tests for my application that are easy to understand.

The way I went about ensuring my code was easy to read involved two basic strategies, utilizing camel casing for variable names and class methods as well as spacing my lines of code so that only code that are directly related to each other for particular sets of functionalities are clustered together. Camel casing variable names and class methods is an industry standard and best practice that is used amongst developers across almost all programming languages. It provides an agreed upon standard format for variable and class method names that consist of more than one word to improve readability. At the top of all three of my class files, “Contact”, “Task”, and “Appointment”, all variable names that consist of more than one word follow the camel case convention. This same convention can be seen in all class methods for these class files. Effective spacing of code is not a requirement, but again is an industry standard and best practice as it improves readability substantially. This can be seen in all files of my application since I implement this in all code that I write.

Comments can be seen throughout my application as they provide insight into how the application was designed and what each method is intended to do. As an example, in the “TaskSService.java” file, comments can be seen above the class constructor, the data-structures, and every method. Line 11 says “In-memory data structure to store tasks”, line 27 says “Method to delete task from task list”, line 49 says “Method to update description”. As shown above, these comments provide insight into how the application works on an individual level for each aspect of the code to provide valuable knowledge for future developers or myself when maintaining the code base at a later time.

Initializing variables correctly involves initializing them close to where they are going to be used to reduce confusion and without specific declaration until declaration is required to reduce unnecessary memory usage. Using all declared variables in the application prevents the potential to waste resources, reduces runtime and time complexity, and reduces code complexity as unused variables may cause confusion in large code bases by contributing to more code that isn’t required. In my “Appointment.java” file, lines 9 – 11, the private string variables are initialized at the top of the class and are declared within their respective setter methods. This methodology reduces confusion and follows best practices.

Implementing effective class methods helps prevent code repetition by providing functionality that can be reused any time a particular process is needed. For example, in my “ContactService.java” file, lines 22 – 29, the “addContact” method provides a means to call this function any time a new contact is created and added to the contact list data structure. Without this method, the code to add a contact to the contact list data structure would need to be written every time this functionality was needed, resulting in a huge amount of code repetition that causes the code base to become larger and more complex to read than it should be.

* + 1. **How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.**

Ensuring my code was efficient involved writing my classes and tests to perform their intended purpose without redundant or wasted code that had little to no contribution to the requirements of the application. Within my TaskServiceTest.java file at lines 31 – 41 is the test for deleting a task from the array list data structure. This test starts with the required “@Test” annotation and defines the test as “testDeleteTask(). The function starts by creating a new task object and adding that task object to the array list. I capture the size of the array list after adding the task object to it, I call the delete method, and capture the size after deleting the task object from the list. The test then calls the assertTrue() function to validate that the size of the array list before the delete method was called is larger than the size of the array list after deleting the task object from the array list. This function is effective because all lines of code are executed during the test and it reduces the need for iterating the array list and using conditional statements, which prevents missed branch execution and iterating a list that only contains a single object. Both of these situations for the case of deleting an object is a waste of resources and time to complete the test when it doesn’t require this sort of code infrastructure.

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

The software testing techniques that I used in this project were equivalence partitioning and decision testing. Equivalence partitioning involves dividing variable input into different partitions, which includes a partition that is an acceptable range of input and a partition that is an unacceptable range of input. There can be more than two partitions when testing since the range can, and normally will, have input ranges on both sides of the accepted input range. For example, if a variable input requires a range between 1 and 10, then that is the partition of accepted input, and anything in the range of 0 or less, such as negative values, is a partition of unaccepted values and anything greater than 10 is also a partition of unaccepted values. In my application, I utilized equivalence partitioning for all of the attributes defined in the “Contact.java”, “Task.java”, and “Appointment.java” classes as all of these attributes had a range requirement that the variable could not exceed. In regard to “Contact.java” as an example, the attributes “contactID”, “firstName”, “lastName”, were not to exceed 10 characters in length with the partition of acceptable input in the range of 1 and 10 and the partition of unacceptable input any length greater than 10. The “phoneNumber” attribute was required to be 10 characters in length exactly with the partition of acceptable input exactly 10 characters in length and the partition of unacceptable input any length between 0 and 9 or greater than 10. Finally, the “address” attribute was not to exceed 30 characters in length with the partition of acceptable input in the range of 1 and 30 and the partition of unacceptable input any length greater than 30.

Decision testing is implemented by ensuring that all branches of conditional statements are taken during testing to ensure that all code operates as intended when a particular branch is taken. For example, in my application in the “Task.java” file, lines 10 – 25 contain the constructor for the Task class to create a Task object. Within the constructor, there is a conditional statement that will take one of three branches depending on the condition of the “taskID” attribute. If the attribute is null, an exception will be thrown, if the attribute is greater than 10 characters in length, an exception will be thrown, and of neither of these conditions are met, the attribute is officially assigned to the passed value when constructing the Task object. This same principle is also applied to the “setName” and “setDescription” methods for the Task object. The way in which these conditional statements are tested is in the “TaskTest.java” file across all of the tests I have written.

* + 1. **What are the other software testing techniques that you did not use for this project? Describe their characteristics using specific details.**

Other software testing techniques that I did not use in my project include state-transition testing, statement testing, and flow charts, among many others.

State-transition testing involves diagramming the application from one behavior to another behavior, or from one state to another. AN example of this could be described as going from one level of a game to the next. We first describe the state of the game at level one, and then describe the state of the game when advancing to level two. When we perform this testing, or diagramming, we expect the application to behave in a particular manner at level one and then behave in another manner at level two, so it is critical to ensure that the application operates correctly before, during, and after this transition phase.

Statement testing involves testing every line of code in the application at least one time to ensure that the code operates as it was designed. This form of testing is a good practice to adhere to as it allows the analysis of the full spectrum of the applications behavior and provides insight into application security and overall behavior regardless of which code is executed or not.

Flow charts are used throughout the SDLC for both application design and application testing purposes. For the purpose of testing, a flow chart can provide a visual representation of program behavior and depict how the program will behave in particular conditions.

* + 1. **For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**

All of the software testing techniques that I have discussed fall under either black-box testing or white-box testing. Black-box testing is concerned with observing and testing the behavior of the application to ensure that the stated requirements are met. The benefit to black-box testing is that the developer who is conducting the testing does not need to understand the internal structure of the code.

Equivalence partitioning is a form of black-box testing that can be used to ensure that the program provides the intended output from a defined input range or range of inputs. This can be useful for projects where the requirements have defined the expected output of the application from a set of inputs known to produce the defined output. The other benefit that this testing technique provides is allowing a separate team of developers to test the program as they do not need to understand the underlying program structure.

Decision testing is a form of white-box testing as it depends on testing and analyzing the behavior of the underlying program structure. The major benefit of this testing technique is unit testing can be conducted by the developers writing the program as it is being created. This helps streamline the development and testing process as testing does not need to wait until development is complete. It is also important to ensure that the varying conditional paths that the program can take is thoroughly tested to ensure the complete behavior of the program is analyzed.

* 1. Mindset
     1. **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.**

The mindset that I adopted while working on this project was two sided. First, I was concerned with writing the program in a professional manner that met all requirements that the application specified. Second, I wanted to write my Junit tests in the most thorough way to obtain as much coverage as possible without writing them in a way that targeted a passing condition. The extent to which I employed caution when acting as a software tester for my project was ensuring that I did not leave code untested or tested in a way that allowed my code to pass on purpose. Having a full understanding of how the various aspects of the code was interconnected and the complexity of the code was very important to the Junit tests that I wrote for the application. Without this understanding, it would be easy to leave code untested or discover failed tests without understanding why they are failing. An example of understanding the code complexity and interconnectedness is understanding that class objects cannot be created with duplicate unique IDs. The test to ensure that a duplicate object cannot be created required an understanding of the class structure and methods used to create and add objects to an in-memory data structure.

* + 1. **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.**

The way I tried to limit bias when testing my code was attempting to view my code through the lens of a different developer and not taking it personally if my code wasn’t passing a test.. This isn’t always easy to do, but I found it helped to try and “decouple” myself from my code and prevent myself from reacting emotionally when it didn’t operate as planned. The idea of creating code that works as defined in the requirements instead of code that I feel proud of is more important and ultimately results in a better application.

After writing the code and Junit tests for this project, I can see how bias is a concern when a developer is testing their own code. When a developer invests many hours into a project, it is almost inevitable that an emotional connection is formed between the developer and the code in the form of pride in their work. So, it is reasonable to assume that a bias can be formed between a developer and their code, resulting in Junit tests that may not be as effective had a developer conducted the same testing that was not a part of the development process as personally.

* + 1. **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.**

Discipline is a very important trait to possess as an individual and is particularly useful as a software engineering professional to ensure that the work being produced is of the highest quality. Software engineering is a highly technical field that requires strict attention to detail and a lack of discipline to our commitment to quality can have devastating consequences to data, finances, trust, and in extreme cases, life. Cutting corners creates a perfect pipeline to some of these undesired consequences, is a direct result from a lack of discipline, produces some of the lowest quality software engineering and results in technical debt. Reducing technical debt is achieved by maintaining discipline and resisting the temptation of cutting corners. Technical debt tends to pile up after a short time and results in more work in the long run. It is wise to do the extra work up front and take the time to ensure that the application is of the highest quality as this will reduce future code refactoring and unforeseen errors that, in order to fix, escalates exponentially in both time and cost.