Daniel Giannatsis

6/14/2022

Software Testing and Automation

Project 2 Summary and Reflections Report

1. Summary

My approach to testing was very aligned to the software requirements. I made sure that I tested every class and data requirement outlined for each feature. For the contact class, I made sure to test that phone numbers had to be exactly 10 characters and not null or that addresses had to be 30 characters or less and not null. Sinilarly, for the task class, I made sure to test data requirements such as testing the ID was ten characters or less and not null, or testing the description was 50 characters or less and not null. The appointment class had similar requirements for the ID and description, but I also had to test that the dates being passed into the constructor were before the current date. I was aligned with software requirements when it came to the ContactService class as I made sure to test that the functions, such as addContact and deleteContact were implemented correctly. This testing checked that the addContact function threw an error if a duplicate contact was added or that the deleteContact function deleted the contact that contained the given ID argument. I tested to validate very similar functions in both the TaskService and AppointmentService classes.

The overall quality of my JUnit tests were solid as all of the tests had at least 80% coverage for the class they were testing. I know that my tests were effective based on the coverage because it means that for each class, at least 80% of the code was executed. For the first component, my test for the Contact class had 84% coverage and my test for the contactService class had 87% coverage. For the second component, my test for the Task class had 88% coverage and my test for the taskService class had 96% coverage. For the third component, my test for the Appointment class had 88% coverage and my test for the AppointmentService class had 87% coverage.

One example that my code was technically sound when writing the unit tests was that I used the correct assertions. For instance, when testing that an input would be an incorrect argument, I used the assertThrows with the correct exception type to make sure the code threw the error.

A picture containing graphical user interface

Description automatically generated

Another example that my code was technically sound for the unit tests was to write certain functions that commonly wouldn’t need a return type with a return type in order to compare values with testing. I used this when testing the delete functions for all 3 features.

Graphical user interface, text, application, email

Description automatically generated

An example that my code was efficient for the unit tests was when I tested the constructor for the contact class by adding an assertion for each of the entered fields to check the values in the new object were the same as the arguments passed to the constructor. All of the assertions were grouped in one test to cut down on the number of tests being run.

Graphical user interface, text, application, email

Description automatically generated

1. Reflection

Testing Techniques

The major software testing techniques used were functional, specification-based, black-box testing. I wrote JUnit test cases to test out that the code fulfilled the requirements. This involved writing assertions that checked that the classes’ constructors were accepting the correct data and throwing exceptions for incorrect data, such as null data or data that had too many characters. There were also tests to make sure functions such as adding or deleting for the contactService, taskService and appointmentService worked correctly, such as the add function throwing an exception if there was a duplicate entry, the delete function returning null if the object to delete wasn’t found or checking to make sure the changeLastName changed the last name of the correct Contact. There were a number of testing techniques that weren’t used for the project, some were specification, black-box techniques and others were structural, white-box techniques. Equivalence partitioning, a specification-based technique wasn’t used because the classes in the project didn’t take in numerical input that and behave differently depending on those inputs. This testing technique is useful in those situations because equivalence partitioning drastically cuts down on the different input values that need to be tested ; only the equivalence classes that cause the program to exhibit different behavior need to be used. Boundary analysis was another related technique that wasn’t used. It involves using three values around a limit value to test different behavior when those values are inputted. Both of these techniques are great when the code being tested makes a lot of decisions based on the values of numerical data.

Another specification-based technique is called decision table testing, where the system is defined in terms of combinations of business rules, which are conditions, that result in actions to occur. In a decision table, each test case details the combinations of business rules that need to be either true or false in order for a certain action or actions to occur. This technique comes in handy when testing systems where different combinations of inputs/actions can result in the same outcomes. State transition testing is another black-box technique that focuses on events that cause a system to change states through the use of a state change diagram and state table which is where we derive the test cases from. When looking at a system that changes states frequently based on events, state transition testing is effective. Another form of specification-based testing, use case testing, focuses on sequence of actions close to how an average user would use the system. This testing is used to write tests to address the most used system functions are working properly, usually when the project has limited resources and can’t do more in depth testing. Other test techniques that I didn’t use were white box, structural techniques. These techniques focus on code’s structure and are concerned with reaching a high coverage percentage for both statement coverage and decision coverage. We create control flow graphs by analyzing the decision points in the code so that decision coverage can be tested. Tests are written to reach all of the different branches of code in the control flow graph. This test technique is valuable for code with many decision points and can be used not only at the component level, but also at the system level and integration level.

Mindset

I employed a significant amount of caution in writing not only the tests but also in writing functions that I’d written before, since I found out that although they were pretty basic, they had to be written in a way that I could use assertions with them. This way of writing code where I had to be aware of making it testable took a bit longer than it would have been without testing. One example was making sure that where I threw exceptions in the code were correct and that all branches could be reached. It was important to appreciate the complexity and interrelationships of the code being tested because they need to be understood in order to write effective tests. For example, when testing contactService, one needs to know that it’s made up of contacts and in certain tests, contacts needed to be created along with the service to test properly. Bias can be a huge problem for those testing code they’ve written themselves because one believes in their code much more than another tester ever would. One way I limited bias was to separate my coding and testing sessions. When I would return to write the tests, I’d try to pretend that I was testing somebody else’s code where my only goal was to prove that the code was viable and met the requirements imposed on it. Similarly, cutting corners when testing code, such as not writing for enough test cases means that defects will go untested and unaddressed. One way that I’ll avoid technical debt is, when possible, to use test-driven development where I write all the unit tests first so there is a thorough definition of what the code should do. I’ll also frequently check the code coverage to make sure that the tests are actually testing enough of the code.