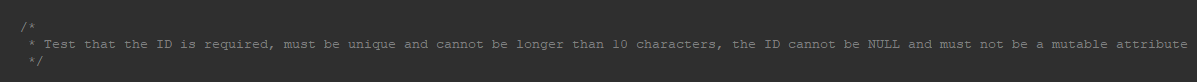
**Project Two Summary**

**Aaron McDonald**

**10/15/2023**

 To begin summarizing my project for module six – I will discuss how I achieved completeness with ensuring that the software aligned with the software requirements. When writing the unit tests, I would ensure that I am referencing the software requirements by creating comments at each section in the code where I believe the expected functionality will be tested. As an example I will use this screenshot here:

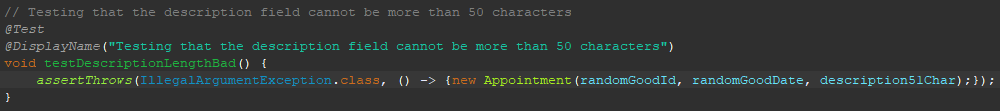
This is one example of many comments that I have laid throughout my code to help myself ensure that I am maintaining the standards as set by the module. I then continue to make a unit test for each of these parts of the object as well as some additional tests to test boundaries of certain objects.

 As for the effectiveness of my code – I was able to run my Junit tests in coverage mode to discover that I cover 84.6% of the code in my objects as shown below:

A screenshot of a computer program

Description automatically generatedThis is above 80%, meaning that most of the code has been covered by my tests with the remaining code being code that may not truly need to be tested. As an example I will use the below screenshot:

The code that is highlighted in red is code that is not being tested or getting any coverage. This screenshot is that of a test and in this case the only reason this code would get coverage is the case of the test failing. This is an example of what most of the code not being tested is – therefore establishing that the most important parts of my code are being tested thoroughly.

 To reflect on my experience writing Junit tests – I would say that the experience overall is quite positive. I think that the Junit framework is quite intuitive and very easy to use – having the ability the write every test in its own independent function, having test naming, test looping, and easy to read annotations. To ensure that my code was technically sound I tried to create unit tests that were very specific to what piece of code they were testing. I will use this snippet of code as an example:

This is a simple function that will test the description attribute to ensure that the description cannot be more than 50 characters. This is just one example of the simplicity with testing that you are able to produce with Junit tests to ensure that your code is technically sound. These small unit tests allow you to do boundary testing such as what I am doing here, ensuring that your code is effective in a different way than just being executed – but acting in expected ways depending on the data being given to it.

To ensure the efficiency of my code I ran each test class separately and looked at the timing that is measured for my tests such as shown below:

A screenshot of a computer

Description automatically generated

These tests all ran in a very small fraction of a second – indicating that the code and the tests themselves are running efficiently. One particular test that I wrote to help stress test my code was the repeated test of adding an appointment 50 times. Since this test only took 0.002 seconds we can conclude that the main functionality of the Appointment Service object works very efficiently. If I wanted to fully test the efficiency of my code I could create bigger and bigger looped tests with randomized inputs to truly see the timescale of my code to ensure that it is truly running to maximal efficiency.

As for software testing techniques, there were a few different ones that I used throughout this project – the main one being Test Driven Development. Most of how I created my classes was by first creating the unit tests and determining how I expect the code to run based on the unit tests. Additionally, we were mainly performing a type of White Box testing. Since we were the creators of the code we know how the code is supposed to work and therefore have a small bias for our expectations when creating the tests for that code. White Box testing is useful in its own way, such as knowing how the code works allowing us to gain decent coverage of that code – rather than just hoping that our tests had covered the code fully.

A couple of other testing techniques that I used were from our book: *Software Testing: An ISTAB-BCS Certified Tester Foundation Guide.* Equivalence partitioning and boundary value analysis are both testing techniques that I used to limit how many tests I needed to perform while ensuring the code is effective. Equivalence partitioning is the process of limiting data into certain groups and then testing one part of that group to ensure the expected behavior occurs for any data in that group. As an example, in my code there was a case where a phone number must contain only digits. From this requirement we can parse out two groups of data, phone numbers with only digits and phone numbers that contain any character. I created an example case of each category and tested it against my code to ensure that I got the expected outcome.

For testing with boundary analysis – I created test cases just on the edge of each boundary, the valid boundary, and the invalid boundary. This ensures similar data sets to equivalence partitioning in that if the closest unacceptable data doesn’t work and the closest acceptable data works, then all of the other cases should also work. Again, this limits how many tests need to be written to ensure a level of confidence with our code – ensuring that the code works properly on the basis of these well-defined tests.

Finally, for the mindset of a software engineer in creating this code, the mindset of caution is very important in creating robust tests for thoroughly testing code. The whole purpose of testing is to act in a proactive manner, rather than reacting to issues that arise when code is deployed. In the case of this project, faulty deployment of code will cost the company money that we are working for, time for our coding team and a loss of customer trust in our abilities as a company. In bigger and more industrial settings – faulty deployment could release code that could be harmful or deadly to other people. This is why it is crucial to understand the relationship between different pieces of code and the complexities of different systems that must be tested in order to create very effective tests. These systems may not just include the software – the environment that software or hardware may be deployed in may also have an effect on how the system operates and so understanding the deployment environment and testing under similar conditions is also crucial in developing secure and robust tests.

Bias is another consideration when developing code and testing that code. As a developer, we have an inherit bias in wanting our code to work and therefore we could fall into the trap of just testing that it works instead of testing to show that it doesn’t work. One of the principles that I try to keep in mind when writing tests is to try to “break” my system. Proving that a system works is easy since we know how we wrote the system and can write tests to simply meet our expectations. The ultimate goal in testing, however, should be to try to break the system, and only when we have sufficiently tested it enough to prove that it is too difficult for us to break can we have the confidence that it can be released. With that in mind, when writing my tests for this project I wrote tests for positive and negative scenarios. I wrote tests to ensure that the code would execute as expected under optimal conditions, but I also wrote tests to ensure the code would respond correctly under non-optimal conditions. This helps to cover the basis of positive and negative interactions with the code and attempts to show that the code will not break under stressful environments.

With all of these mindsets in mind the last consideration is discipline. To be able to write tests that ensure coverage of our code we must have the discipline to investigate the code thoroughly and the deployment setting thoroughly to be able to write tests. We must also discipline ourselves to write all of the tests that we feel that we need to have a high level of confidence. Writing tests is not as interesting as creating the framework that we are testing, but it is possibly more important than the tool we are developing. A tool is only as good as it is robust, and if the tool breaks after a normal day of use then the tool will not be very useful to anyone. This becomes truer as more stakeholders become involved in what is output from a tool being created. Software is a very expensive tool to create and is deployed into very important jobs; these are cases where it is absolutely vital to have quality testing which requires the disciple to write thorough tests.

**Citation:**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.