CS 320 Project Two

December 10th , 2023

Jeremy Reaban

The project consisted of creating three classes that contain objects, task, contract, and appointment, as well as three further classes (taskservice, contractservice, and appointmentservice) that would allow the manipulation of those objects, adding them to a list or hashmap object, creating a database of sorts, and allowing the objects to be edited.

The software requirements in the task, contract and appointment classes required that objects have certain fields that are strings of a given length, longer than zero (that is, they are not null) and shorter than a specified limit. Furthermore, these objects were to be capable of being edited in all fields except their ID. The appointment class was somewhat more complex, containing a field that included a date. This needed to be compared to the current date to see if the appointment data was in the past or not. In order to make sure these requirements were complied with, I used JUnit testing.

Before the tests were created, I set those requirements into the code of the objects themselves, writing the methods that set the values of the objects so any values must obey those requirements. In writing the JUnit tests, I then used varying cases to test that the requirements were enforced. Specifically attempts to create an object with null values, with normal values, and with values larger than allowed. For the appointment class, I also created test cases with past and future dates, to see if it met the requirement that appointments could not be created in the past. If those rules were violated, then a failure was asserted. If the rules were not violated, then they passed.

For the classes that used the objects, the service classes, I did something similar, calling the methods created in those classes with varying types of input representing correct and incorrect inputs. If incorrect inputs were accepted, than a failure was asserted. For instance, I created an object, set values, and then checked to see if those values were properly set by checking the property for a correct answer.

As the project never got further than the creation of the three object classes and the classes to service those objects, meaning they stood along and were not integrated, only unit testing was used. This essentially consisted of two parts, first doing a manual code review to make sure things looked right, then much more rigorous testing using JUnit to create test cases for all the classes. This was in essence, functional testing, making sure the input and output were validated, producing results within specified requirements.

Both of these types of testing are examples of static testing, as the code itself was not run directly. The JUnit tests do run the code, but only with the provided inputs in the test, and similarly, the output is contained in the test. This is a form of black box testing. Black box testing treats the program as a hidden, opaque system but looks at the input and output to make sure they are correct and what is expected.

Had the development of the project been taken further to completion, it would have gone additional types of testing. Next would be integration testing, which starts to put the units of the program together as a whole to make sure they work together. Integration can be partial, either top down or bottom up, or the whole program integrated at once, called big bang integration. The former makes finding problems easier, but requires the use of temporary or placeholder components that don’t really work and are just used for testing purposes, and so can be more time consuming. The latter, testing the program entirely at once might save time by not requiring placeholder program components, but makes debugging more difficult since it might not be obvious just which component is causing the problem.

After that comes system testing and usually done near the end of a program’s development cycle, when it is functionally complete. The program is checked to make sure it works by using systems and values that are expected to be used by normal, regular use of the program, as opposed to the possibly extreme cases used in earlier stages in testing. It’s also checked to make sure it meets legal requirements (is data protected, does it mean export laws if it’s being exported, and so on) and can be installed on non-development systems.

When that is finished, acceptance testing is next. This is done independently of past testing and usually done by representatives of the company the program is being developed for, to make sure it meets all their needs and works to their expectations, works on their hardware, and so forth. It might also need to pass government or industry safety standards, which should have been checked for in system testing, but this is checked again independently. Once this is done, the product can be released.

Since I wrote the code to be tested, it was somewhat difficult to separate what I knew I intended with my code, and the tests of that code. For instance, for the class objects I wrote the setter methods to check for bad input data and make corrections to that bad input data. So the JUnit tests that I wrote were simply to use cases of bad input and to see if my setter methods handled the bad input correctly.

Had I instead written the code to not do that and simply throw or generate an error when it encountered bad input, I would instead have written my tests to check to see if that error was generated when bad input was provided and no error when good input was entered.

It’s important to be disciplined when testing, but I’ve learned from experience and the course material that it’s impossible to test everything. When you have limited time and budget, you must test for the most important things and test as efficiently as possible. In many ways it’s like triage, you have to ensure testing creates the most good given the limited resources.

It’s also something learned from experience. As I am a novice tester and developer, I am not aware of the more common or obvious mistakes and how to spot them, so I have to be as methodical as possible. That’s not to say more experienced testers and developers should cut corners, but they can find the more common mistakes quicker which lets them concentrate on the less mundane.

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.