**Project Two – Summary and Reflection**

By Joseph Mead

The features for this software were so similar that I was able to reuse most testing techniques for each class. This was a great time saver, but it also limited the diversity of techniques employed. I used a testing plug-in that allowed me to learn more about how effective my tests were. In some cases, I had features that were passing my tests, but actually contained errors. I would not have discovered them without the help of the plugin.

The features were nearly identical in that each required a class with a unique ID and contained objects which could not be null and were limited by how many characters they could contain. In addition, they required the ability to add new members of each class and the ability to modify objects in each class. In order to test the creation of a class object, I first created my object and I used assertTrue to check each that each element was present. Then, for each object in the class, I used assertThrows and IllegalArgumentExceptions to check that each violated condition did in result in an error. I tested to check if the object was null and verified that if the number of characters were exceeded, it would result in an error. I should point out that the only object that differed was the phone number in the Contact class which needed to have exactly ten characters.

For the Services classes, I verified that I had the ability to add classes with assertTrue. To ensure that I couldn’t add duplicate IDs, I first added an object and then I used assertThrows and IllegalArgumentExceptions to adding that same object again. I used assertTrue after editing an element for verification. The Appointment class and Appointment Service classes contained a Date object which was to be implemented with the Date utility. The date utility requires one to enter the date using Unix time which requires the number of seconds that have passed since January 1st, 1970 to enter a date. I find this cumbersome, so for my version, I used a string entered in this format: yyyy-MM-hh. I then used parse and SimpleDateFormat to convert it to a Date. This presented a problem for testing because that process can throw a ParseException so I had to include that as an exception for each test in addition the error that I was actually looking for.

I think I could have put some other assertions to use such as assertEquals (looks for equality) and assertNull (looks for null values). There are others that wouldn’t apply to this project but might have been useful in other projects such as assertSame, which looks for the same object. When I was reviewing the annotations, I noticed @Before and @After, which by themselves don’t help me, but it made me realize that I could have introduced my class object before all of the tests and removed it after which would have been much more efficient than what I did. I added a class object and removed it inside of each individual test. While writing the code I found I was getting some errors when I forgot to remove an object at the end of a test because duplicates are not allowed.

While it would not have helped for these features, I think many of my other programs would have benefitted from using Parameterized tests. This is where one can generate a whole array of data for testing purposes which could save one a lot of time. I’ve also read about using Junit Rules which allows one to specify exception messages. I could have employed that in my AppointmentService testing rather than writing out the Parse exception for each test. It also appears that one can use rules to add data to the results of the test, which could also be very helpful.

I found that using a plugin to determine coverage was one of the most effective methods that I have encountered for improving my code and reducing errors. I used a plugin called EclEmma to determine the coverage. EclEmma shows the number of lines that were utilized in the java files being tested and also highlights every line of code that was used to perform the tests. This made it very easy for me to find places where errors could exist.

My familiarity with the structure of this code gave me a false sense of confidence, so I did not initially employ much caution. I believed that my code was perfect and error free, so my initial tests did not appreciate the complexity and interrelationships of the code. For example, each feature has a class with a mutator. The service class then uses that mutator in another function. I originally didn’t test the mutators in the original classes because I assumed that they would be covered in the test for the service class. While, that is true, the part I didn’t appreciate is that those tests were written to address issues specific to the service and not issues with the original class. In other words, I assumed that those mutators were right. I discovered that they were actually incorrect and allowed elements to be changed in ways that were supposed to be prohibited.

Bias is obviously a problem for me because, as I have mentioned, my original code was littered with mistakes and I didn’t even notice. I try to test everything at least once, and because of my undue confidence, I moved on after the passage of one test. Obviously there are a lot of bugs don’t fail every test. Once I noticed that this was a mistake I was making, I dealt with my bias by proposing a new challenge. I wanted to try to account for every possible scenario in my tests. It became a challenge. The testing plugin helped with that.

That challenge I gave myself to address my bias is related to developing a commitment to quality. Rather than assume that my code had no problems, I instead assumed that there was a hidden problem and my goal was to find it. The testing plugin contains a feature that determines how complete the testing classes address the class to which they are assigned. Initially, some of mine scored low. I was aware that would be the case because my service tests were testing some of the functions from the other class, but in order to be a completionist, I wrote more tests to improve my scores. I found one more error in my code using this technique. This is the technique I would employ to avoid technical debt. Those tests were very easy to write and, at times, felt redundant, but they allowed me to see errors that would have been very elusive to track down later on.