CS320: Project 2

Edward Johnson

In a recent assignment, I was asked create and present deliverables for a customer mobile application. I was asked to design the classes and tests for the core framework of three separate sections of the application, as well as the corresponding services that manage those sections. I was asked to include a testing suite along with each of these sections, one that included full test coverage for every method in each of the classes and their services. My approach for developing a unit testing suite for this application, was to start by working from the software requirements that I was given. The software requirements were the framework that I used in order to implement all of the functionality for each class and its corresponding service class. In this regard I first built out the features within each of the 4 core classes, and then laid out the service classes that would be managing those core classes. I was left with a total of 6 class files. These core class files included the Task class, the Contact class, and the Appointment class. For the service classes, I had the TaskService class, the ContactService class, and the AppointmentService class.

When writing out the Contact class, there were a total of 5 member variables that the requirements the customer provided detailed we must include. The first of these member variables was the contact ID number. This ID was designated to have a maximum character limit of 10 characters, it was stated that the value must not be null, and must be a unique value. Additionally, the ID value must be immutable, or shall not be able to be updated once the object has been instantiated with the ID value. The next member variable was defined as the contact first name, it could not be null, and it was designated in the requirements to have a 10 character maximum limit. The last name was the third member variable and is constrained to the same

limits as the first name. The fourth variable is the phone number, and it was also stated that it could not be null. Additionally, the phone number is constrained to only accept values that are exactly 10 characters in length. The last member variable is the contact address, and it also cannot be null. The address was defined in the requirements as having a 30 character maximum limit. These were the requirements defined for each variable, and they went on to state that each of the member variables with the exception of the ID should have a method in which they could be mutated, or have their values updated with a value that met the guidelines laid out in the requirements. Using these requirements, I created a full Contact class, with getter (retrieval) methods for each member variable. Using the limitations from the requirements, I then created a mutator method for each of the variables with the exception of the Contact ID variable. Each of these mutator methods accepts a string value as a parameter that is the respective value to be updated for their object instance. The methods also include a conditional statement that throws an exception if the parameter value for its respective variable does not meet the guidelines laid out in the requirements. If the exception is not thrown, and the value is valid, then the method sets the value of its respective variable instance to the value passed in as an argument. There is a private default constructor for the class, which prevents an object instance without any valid values from being instantiated. The public constructor for the class accepts parameters for a string variable that corresponds to each of the five member variables. In the constructor, we have a single conditional statement that verifies that the Contact ID is a valid value, in the same way as each of the mutable variables do in their respective mutator method. This ID verification throws an exception which prevents the creation of the object if the ID isn't valid. Writing the constructor in this way ensures that the ID value for a contact may only be set at the creation /

instantiation time of an object. Each of the 4 mutable variables for the class then have a call to their respective validation / mutator method, which effectively validates and then sets the value of their variable using the values passed in to the constructor at the time of instantiation.

The service class to manage the Contact class has the following characteristics defined in the requirements, that it must be able to add contacts with a unique ID , that it must be able to delete contact objects based off of an ID value, and that it must be able to update each of the mutable member variables for a contact object instance.

Starting off, I created a method to handle generating a singleton instance for the service class. I then created an arrayList of contact objects that would hold each of the objects that were created while the service was running.

In order to create a unique ID value, I created a method that generated a random UUID value. This value was cleaned up and shortened to 10 characters to meet the customer's requirements. The method checked the list of existing contact objects, and if no match was found, then the value was returned as valid. Otherwise, it generated a new ID value and repeated the process. The create contact method accepted parameters for the four mutable member variables, and then with a call to the create ID method, generated a valid ID number. The delete method calls the findObject method with the value of a contact object ID. FindObject searches through the list of objects, and if it finds a match, it deletes it from the list. I then created an update member variable method for each of the 4 mutable member variables of the contactObject. These updateVariable methods accepted 2 parameters, the first being the object ID, and the second being the new string value to update the variable with. It accomplishes this task by calling the findObject method with the ID, and then calling the respective Contact class validate method for

the object that is returned from the list. If the value is valid, then the variable is updated to the new value, otherwise an exception is thrown.

I repeated the same creation process for the requirements I was given for the Task, TaskService, Appointment, and AppointmentService classes. I ended up with 6 functioning classes that matched up to the requirements that we were given by the customer. I demonstrated that each of the 6 class files was built in a way that was technically sound, and provably so by testing the possible conditions for the requirements.

For each of these 6 class files, I built out their corresponding test class file using JUnit 5.

My goal when building this test suite, was to ensure that I thoroughly provided testing of all of the functional requirements as defined by the customer. I used a combination of black-box testing, and white-box testing to ensure that my unit tests covered both the behavioral aspects of the packages, as well as thoroughly testing the code implementation for each class. For each of the member variables of each core class object, I created a unit test that validated each of the values edges set out in the requirements given by the customer. For member variables that had a character limit imposed by the customer, this included validation and instantiation with valid and invalid values. An example of these tests can be seen on lines 59, 74 and 86 of the TaskTest.java file. These values included those that were within the requirement guidelines (line 59), and that would pass through without an exception being thrown. They also included values that exceeded the character limit (line 86), or that were null and as such would throw an exception when the act portion of the test was attempted (line 74). The tests assert that a value will not throw an exception in the case of valid values, and will throw an exception for invalid values. For testing immutable member variables such as the ID values, I used a constructor call to attempt

instantiation with the appropriate valid or invalid values (such as on line 39 and line 49). A new value is passed through the mutator method in the case of mutable variables. This is because they may be changed after creation of an object. I then asserted using the getter method for that mutable variable that the value had indeed been changed using the assertEquals method.

Black-box testing is a set of techniques in which you treat the system under test as a "black-box", in which we do not know any of the inner workings or what is contained within. This style of testing focuses on testing the behavior or the specifications of the system. Black-box testing is stronger suited towards refactoring since the focus is on the behaviors of the system instead of the implementations, but is less resistant to regressions than white-box testing techniques.

For the service classes, I tested each of the methods that were defined as requirements by the customer. I first tested them using black-box methods by simulating a live environment through my test assertions. I carried out these tests through the use of limited entry decision tables that were composed of combinations of decisions that lined up with the expected results of creating or modifying a valid object instance. These expected results were those that were laid out in the requirements by the customer. I also made sure to include tests that tested the edges of each case in which an exception would be thrown for trying to instantiate an object with an invalid variable value, as well as trying to modify a mutable variable with an invalid value.

White-box techniques are generally focused on testing the inner workings of the code and how it functions. It does this by testing the implementations of the code as opposed to black box which focuses on behaviors. White box tests are generally better suited towards handling regressions, but are typically much less flexible to refactoring due to their focus on the implementations.

I used white-box methods such as statement testing in order to place the application into a state

in which it would be ready to test the functionality of various methods. This was carried out by first creating the configurations needed in order to prepare the system for the test that was about to start. Such an example would be from the AppointmentTest class where I create a valid appointment object by calling the private factory method createAppointmentWithDate() with a createFutureDate() method call for the test class. The first time that the createFutureDate() method is called in a test, it creates a static future date for the tests to use. All of this test takes place within an assertion that an exception will not be thrown by any of the assertions or actions taken during this test. Using the values passed in during the object's instantiation, I assert that the member

variables are what they are believed to be, and valid.

Regarding the overall quality of my unit tests, I believe strongly that with the refactoring I have done across each of the class test files, that I have ensured that the provided unit tests are up to industry standards. I reduced and removed the use of duplicate code within the ServiceTest methods by utilizing the setup and tear-down methods to achieve the same goal as was previously being done during the arrange step of each test. This can be seen in action on line 17 and line 32 of the TaskServiceTest.java file. The validTestTask, and testTaskService are called and referenced in each test of the service file, and at the beginning of each setup they are re-initialized as null and then recreated. At the end of each test, any instances are deleted as seen with the method annotated @AfterEach. Each of the ServiceTest classes has the base needs of the tests met by the instantiation of a fresh object, the resetting of any static service instances before each test, and the deletion of any created objects after the test during the tear-down. This is all done and executed in order to remove any unnecessary coupling between my unit tests. For

the core object test classes, I implemented a private factory method(s) in order to handle the setup before each test, as the arrangement phase for each test is less complicated and does not require any modification of the system state.

Writing the test classes in this way lends towards the efficiency of the code by consolidating the majority of the arrange section of each unit test. It does this by containing the arrangement and set up of the tests into a method that automatically runs before each test, or into a single factory that can be called once to set up the test. I used the setup and private factory method instead of the test class constructor, again, to reduce the coupling between any of the tests. An example of this simple factory method that is used in the core class files can be seen on line 15 of the TaskTest.java file. The majority of my unit tests for both the core classes and for the service classes is then left to a single arrange call that consists of retrieving the object instantiated in the setup or factory method. This is then followed by calling whichever function is responsible for handling the act phase of the test. Following the act method call, I make assertions about the validity of the act that was the focus of the test.

Regarding methods for test development that I did not include in these tests, I was unable to implement any of the tests that are primarily experience based techniques. This is solely due to my limited experience in unit testing. As I am new to writing unit tests, I did not feel qualified to explore deeply into techniques that rely on a developer having picked up experience from working on a large number of projects. These techniques include the white-box techniques of error guessing, checklist based testing, and exploratory testing. For error guessing, the developer uses their own wisdom and experience with systems in order to intuitively predict the areas of

code that are of value, and that may be weak / susceptible to failures. As the number of projects that I have worked on is limited, I did not feel that it was something I could feasibly rely on in order to develop tests. Exploratory testing is typically carried out under heightened stress, time constrained projects, and it is an attempt to get as high of a degree of test coverage as possible in as short of a time frame as can be. This is done by prioritizing the development of tests for the core, essential features of the application. Checklist based testing typically involves the pooling of experience between the team in order to determine a check list of what needs to be testing in a system, this can also be influenced by documentation and outside sources for general solutions. As the scope of these classes was rather limited, and I was not operating under a shortened timeline, there was no need for me to implement these strategies with our mobile application. I opted to take a more structured approach, and to attempt to test the functionality of each method, and all of the behaviors of each class.

As the tester for this application, I attempted to employ caution where it was possible.
I constantly referred back to the documentation that we were provided for this application, in order to be sure that my understanding of what the requirements were matched up to how they were laid out by the customer. Additionally, I did my best to implement the feedback that I was given into the final deliverables. It is always important to take time to understand the complexity and the internal relationships within the code we are testing, by doing so, we are better able to test areas of the code and interactions that may have been overlooked.

As a developer, I can see how bias can limit us when we are designing tests for our own code. Because we are the ones who are writing the code, it is very easy to focus on the

implementations that we used to write the system, and potentially overlook ways in which the the code doesn't meet the value that the customer is expecting to receive from the software that we have written for them. This phenomena is known as confirmation bias, and can occur when we are expecting a set of goals. Say our objective is to design product features that meet a certain standard by a set date, our bias will trend towards reaching this goal, and we all modify our actions when we are attempting to meet a goal. We can seek to limit our bias by ensuring that we use good communication skills, and that we seek to clarify objectives whenever questions may arise. Defects in the code aren't necessarily a problem that is directly caused by the developer, it may be a problem in the specification documents. This is why constant, objective communication should be maintained and discussions should be held in impersonal ways so that the focus always remains on the product and its improvement. We should welcome the advice and feedback of our colleagues, whether that colleague is a tester, a developer, or a product manager.

Being disciplined in our commitment to quality is one of the most important aspects of being and becoming a professional software engineer. There are no real shortcuts in writing software, and it is important to truly understand this. In order to produce high quality work, it takes great time, and great effort, and it is important to be mindful of the code that we write, and to be mindful of the interactions within the systems that we design. As much time and effort as we put into the design and implementation, it is equally as important to ensure that our code is functional by pairing our code with thorough, and well designed tests. Only by pairing our quality code with equally well designed tests can we hope to reduce the amount of technical debt that accumulates over time within our systems. The best way that I can think of to continuously reduce technical

debt, is to make sure that the code and refactors that we write are written in a way that make them flexible to change. One way that I can think of to achieve this, is to implement the design strategies of Test Driven Development, and to use other agile tools such as Continuous Integration. By breaking our system up into manageable, flexible units of code, we can reduce the weight of our code. What I mean by that, is that it becomes much easier to refactor our code and tests, and that it also becomes more flexible to adding features into the future without having to worry about a full or partial system re-write when we do.

Sources:

- Morgan, P., Samaroo, A., & Hambling, B. (2010). Software testing: an ISTQB-ISEB Foundation guide. In *British Computer Society eBooks*. https://dl.acm.org/citation.cfm?id=1942046
- Khorikov, V. (2020). *Unit testing: principles, practices, and patterns*. Manning Publications Co.