Conor Steward

CS 320-Project 2

conor.steward@snhu.edu

4/18/24

The features of this application varied from class to class and include the ability to update, delete, or add various components of various classes. Below is a detailed description of these classes, their functions, and how unit testing is used to verify their functionality and requirements.

In the appointment class Junit testing is utilized to verify that the id, date, and description of the appointment are not null, are shorter than a specific number of characters, and that date is not in the past. This is done with JUnits @Test tag and creating a function that calls the constructor for appointment, which has been setup to verify each parameter requirement and throw an exception in the case that the JUnit test has parameters set to fail or to assertEquals when the full test is meant to pass. In AppointmentService ID is set up to throw an illegalArgumentException when an ID has been utilized already while using the addAppointment method, which adds an appointment by ID. deleteAppointment searches for an existing ID and throws an exception when that ID is not found and deletes the appointment if the ID is found. @Test tag is then used in AppointmentServiceTest class to create JUnit tests that attempt to add and delete appointments using either assertThrows when the test is supposed to fail or assertEquals when it is supposed to pass to verify the addition or deletion of an appointment.

Contact class uses a different approach. Instead of building verification into the constructor with if statements, like the Appointment class is built, verification methods are implemented after set and get methods to verify that address, ID, first/last name, and phone number follow the desired parameters. @Test methods are then created using constructors which are used against the validate methods for the various parameters to assertFalse/assertThrows if the test is supposed to fail or assertTrue if the test is supposed to pass. The tests cover the length, updateability, and validity of each parameter requirement. ContactService class contains 4

methods. One for adding a contact, one for deleting a contact, one for getting a contact, and one for updating a contact. These are all utilized in ContactServiceTest with @Test tags and are set up with assertTrue/assertEquals if the tests are supposed to pass or assertFalse if they are not.

The Task class is set up in a similar way to the Appointment class but the two are not identical. Instead of handling each parameter in its own if statement, the parameters are grouped for the null check and then have separate if statements to check their length in the constructor. @BeforeEach is also utilized in TaskTest to setup the constructor before each test run. @Test tags are added to each test and assertThrows is used to see if an exception is thrown when the constructor is used incorrectly. TaskService has add, delete, get, and update methods that are called in TaskServiceTest with @Test tags with assetEquals if the test is supposed to pass, assertNull for checking deleted tasks, and assertThrows when the test should fail.

Each of these approaches differ in the way they are implemented but ensure that parameter requirements are met in each case. For example, in Task each parameter cannot be null which is setup in the constructor to throw an IllegalArgumentException and called in TaskTest where each is left null in a different test to verify the function of the constructor and where length is made to throw an exception as well as each parameter is input incorrectly. In ContactService the addContact method is setup to throw exceptions if the ID already exists, is null, or is too long. This is tested in ContactServiceTest where each of these branches are tested to ensure that an exception is thrown in each case. In the Appointment class each parameter is set up to throw an exception if it is null, is too long, or date is set in the past. In the AppointmentTest class all of these branches are tested to verify that exceptions are thrown when parameters have been input incorrectly or assertEquals is used to verify that a properly input appointment has

been logged. These along with all other parameters are met using the same methodologies utilizing either constructors or methods to validate requirements.

The JUnit test coverage was 83.2%, which indicates that a fairly large coverage area was enacted to cover most possible branches of each element of each class. The quality of these JUnit tests could be improved up to 99%, where branches that verify correct input would be implemented to bring the % up. Invalid input have all been checked, which covers mistakes or malicious input from users. Overall the quality of these tests is sound.

Writing JUnit tests was fun and pretty straight forward compared to many of the other implementations required for a Computer Science degree. Utilizing different constructors and methods for validation was interesting, since it opened my eyes to just how many paths can be taken when validating code. The cleanest and most effective approach was utilizing the constructor with throwsIllegalArgument and where each parameter was checked for its own requirements. Writing the tests for these was the simplest since assertThrows can be used each time and the constructor is very close to the same each time, allowing for a copy/paste methodology. If I were to remake the tests, I would use @BeforeEach tag so that the constructor would not have to be called in each test, making the code a bit cleaner and shorter.

The code was technically sound in spite of the variety in which it was built. For one it produced no errors or warnings. Secondly, the java library was used when it served an available purpose instead of creating my own methods to handle various tasks. See .length() used to check lengths of inputs or IllegalArgumentException() in all of the standard classes. Constructors were utilized as well as get/set methods to build the initial classes, keeping in line with industry standards. For testing, JUnit libraries were used to validate parameter requirements in all test classes instead of creating my own test methods. It is well commented and runs smoothly.

This code was efficient mostly because of its small computational needs and low resource utilization. Variables were stored locally and called for execution in test cases where constructors were called for each test case. I think TaskServiceTest was the most efficient due to its usage of @BeforeEach on line 16, where a constructor was called, allowing each @Test to be relatively small and simple to write. Even the others, which called constructors in each case, were still efficient due to their limited usage of resources. Instead of creating new instances of a class to test, the same instance would be reutilized to save space as seen on line 48 of AppointmentTest. Overall these tactics were used throughout the assignment to ensure efficient code.

Unit testing was used to check each functional unit of a class, testing for parameter requirements to ensure good test coverage. Boundary value testing was done to check the length of certain parameters that required length limitations, as seen in line 35 of ContactTest to check first name length. Equivalence partitioning was utilized to ensure that values are as they should be when building an object incorrectly, as seen in AppointmentTest to check how the system handles invalid date inputs. Positive testing was used to verify that a test returns true when it is supposed to pass and negative testing was used to return true when a test should fail, as seen in testValidAppointment() of the AppointmentTest class or testPhoneValid() of the ContactTest class.

Very little if any exploratory testing was done for this assignment, which is where testing is used to uncover unexpected inputs and see how the system reacts to them. This is valuable to aid in creating requirements for the validation of parameters to keep users on track when making inputs. Security testing is not utilized and as the system stand it is very vulnerable to attacks. Security testing aims to identify vulnerabilities and weaknesses in the code making your data safer for having mitigated them. Performance testing was also not done, where evaluating

responsiveness and throughput are tracked to create baseline performance measures. Doing performance testing lets maintainers know when there is a need to scale resources and sets performance baselines to help keep up with user demand. Integration testing, where different systems interactions are tested, was not done either.

Unit testing should be the initial testing case for all systems to verify that each unit performs its function as designed. If unit testing is not done there is not a chance that integration testing will go well or that there is even a point in continuing into the next steps of software development. Boundary value testing is valuable to ensure that inputs or outputs are not longer than a system can handle, keeping local memory, resource utilization, and performance within normal limits. Another benefit is in keeping security measures in place with boundary value testing, disallowing for certain injection attacks. Equivalence partitioning is a software testing technique used to divide input data into different partitions or classes based on the assumption that all members of a partition are equivalent with respect to the test case. The goal of equivalence partitioning is to reduce the total number of test cases while still providing adequate test coverage. Positive and negative testing are utilized in a wide variety to check expectations of true/false values against an object. The goal of positive/negative testing is to validate that the system responds correctly to inputs/outputs.

I cannot say that I was cautious in building these tests except to write more tests when my coverage was not adequate. Outside of this caution would be useful when writing tests that could potentially break the software or are inadequate to completely cover potentially harmful inputs. My mindset was to cover as many branches as possible using as little code as possible, ensuring efficient code and adequate coverage. Appreciating the complexity of interoperating systems is vital for sound testing because without that appreciation the tester will most likely be unable to

thoroughly test for the systems complex functions and will leave some important minutia out of their testing. A good example is something I saw in a discussion post about a piece of radiation technology that ended up burning or killing the individuals it was supposed to help. The designers did not appreciate the complexity of the interactions between the hardware, software, and end user and that cost people their lives.

Limiting bias in my code review was difficult and required me to come back to look it over several times. Since I wrote the code and tested the code I was confident that branches were covered and that requirements were met. After coming back for a third time to review the code for good measure I found that several mistakes had been made. It was only through carefully parsing the requirements outlined in my constructors and testing that I found that some of my boundary validations did not check the correct length for certain parameters. Also, some of the positive and negative tests were found to be backwards, which I only found after scanning my code rigorously after finding the initial mistakes of boundary validation. For this very reason, testing your own code should be avoided if possible.

Writing tests without cutting corners is essential for well functioning, performing, and compliant code. Not doing so can often result in code that doesn't work as intended or even can lead to terrible loss of life and property. Code quality ensures that the company is trusted by the public and stakeholders, the users are kept safe, and that the software itself functions as time goes on. Code quality is not only essential for initial release but countless stories of legacy code spaghetti that cannot be refactored due to its lack of modularization and general lack of quality exist to back up quality code writing.