**Summary**

Unit Testing Approach

To align with the software requirements, I ensured that the correct character limit and the inability to be null for each task, contact, and appointment were included. Furthermore, when testing these restrictions I included sample data that should, and did, come out as false. As an example, when testing the invalid delete portion – I created a task with the ID of “A123”, but when calling the deleteTask function I input the ID I wished to remove as “12345.” This resulted in an error when set to true, but correct when set as false; generally meaning that the function worked correctly. For visual reference, I will be showing Tasks restrictions that align with the software requirements below:

*Ensuring Correct Character Limit and Inability of Null : Task.java*

public Task(String taskId, String taskName, String taskDescription) {

if (taskId == null || taskId.length() >= 10) {

throw new IllegalArgumentException("Invalid ID.");

}

if (taskName == null || taskName.length() >= 20) {

throw new IllegalArgumentException("Invalid Name.");

}

if (taskDescription == null || taskDescription.length() >= 50) {

throw new IllegalArgumentException("Invalid Description.");

}

*Testing Invalid Delete : TaskServiceTest.java*

@Test

void testInvalidDelete() {

TaskService tasks = new TaskService();

Task test5 = new Task("A123", "Test", "Adding to test invalid Delete");

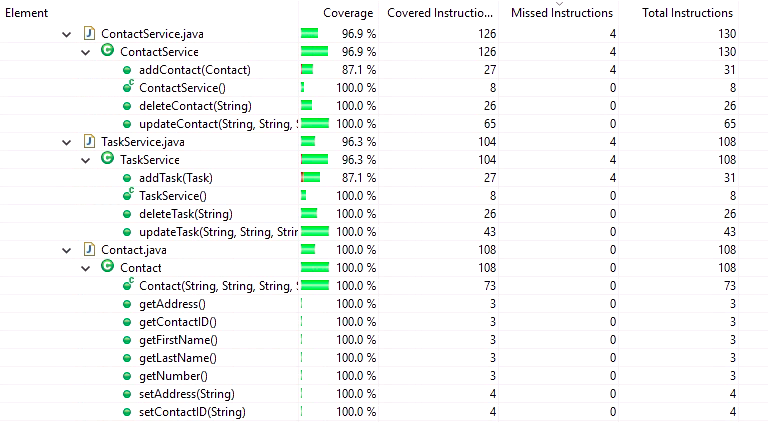
tasks.addTask(test5);

assertEquals(false, tasks.deleteTask("12345"));

}

To determine the quality of the overall JUnit testing, there is the ability to see the coverage percentage of the specific code. My overall coverage percentage was 72.8%. This is not an ideal percentage, so rectifying certain areas is needed before the final deployment. The areas that need to be combed through are Appointment and Appointment Service with their corresponding tests due to their 48.9% or less coverage. Task, Task Service, Contact, and Contact Service were reported having a 96.3 coverage percentage or more – which is more of an ideal percentage.

Experience with JUnit Testing

To ensure my code was technically sound, I made sure that each variable and function name related to what object and/or action it related with. Choosing to do this allowed me to easily combine the previous milestone together easily and efficiently. As an example, instead of just using add(), I chose to put addContact(), addTask(), and addAppointment() allowing them to be differentiated in the future. This will help ensure the reliability and readability of the future, completed system. For visual reference and to get an overview of the functions/variables, below is the coverage percentage output showing some of the given names:  To ensure my code was efficient, the JUnit tests gave me the ability to determine that only valid inputs will work – leaving less of a possibility of errored inputs. Similarly to what I stated in the paragraph above, the given variables allow the code to be readable and organized. Also, the coverage percentage allowed me to determine that there was not any ‘useless’ code when the percentage was high (shown in the screenshot directly above). To avoid repetition, the character restriction in the Task class was, for the most part, able to be carried over to the task service and appointment service functions (such as shown above in the first paragraph).

**Reflection**

Testing Techniques

For these milestones, the software testing technique used was white box testing – specifically unit testing. Unit testing is used to test individual functions to ensure that the output given is what’s expected. Other motions I did whenever running the Junit tests were including both valid and invalid inputs. This allowed me to determine that if something fit the criteria it would continue running but when it went over the character count or tried to complete a task that did not fit the restrictions (such as trying to delete an appointment with an appointment ID that was not on file) – it would not complete the process and show an error.

Some of the testing techniques we did not use for these milestones are integration testing, system testing, and acceptance testing. Integration testing is done after unit testing and is when the programmer checks the interface functionality. System testing would be done after integration testing and when the program is functioning as one. System testing will be used to determine if the software meets the systems requirements. Finally, acceptance testing is the last testing technique we have yet to use and is completed after system testing. Acceptance testing is similar to a unit test as it will show whether it passes or fails and it will be used to determine if the software meets the customers’ requirements.

These techniques differ. Unit testing allows a programmer to test individual functions at a time. This will not only help at catching errors early in development, but can also be used at a later date. In the event that future changes happen to a program, rerunning a JUnit test (which would be error free from the previous testing) would allow the programmer to easily identify which methods are causing an error instead of having to comb through it all again. Integration testing would be done by a developer or tester. System testing is used to check if the software meets the systems requirements given by either the developer or a tester. Finally, acceptance testing will be done by the stakeholders and/or client to determine if it fits all our their requirements.

Mindset

To employ caution, I made sure that each variable had their given class in the name. In the instance I just used Add(), there would be a bigger possibility of the other tests adding an object into an incorrect array. Due to it still functioning and adding, it would pass in a unit test – creating an invalid testing process. The interrelationships of these codes is that a customer can add, delete, and/or update a contact, task, and appointments. These will all come together to create a mobile application with the lists stored in an in-memory data structure called an array.

Programmers who create a code may have the possibility of facing biases when testing it. According to Tolga Mirmirik, there are four instances that may affect the quality of software testing. The first bias is confirmation bias. Confirmation bias is creating tests that the programmer/tester knows will execute and produce the expected output. Second, the bandwagon effect is when one person in a team believes something is correct – which pushes the views to the other team members making them thing it is error free. Following bandwagon effect, congruence bias is when the programmer/tester creates tests that only follow the written expected behavior (Mirmirik, 2018). Finally, inattentional blindness can occur when someone overlooks very obvious defects from either reviewing self-made code and reading incorrectly (from what they expected it to say in their head) or just from lack of attention. To try to limit confirmation and congruence bias, it is important to test all aspects – not based on just what the written expectancy is and what you know will pass. While working alone, the bandwagon effect was not an issue but inattentional blindness was. To avoid the possibility of inattentional blindness, it was necessary to spend days completing the code and later dates going back to review for a “fresher” set of eyes.

It is very important to be disciplined in your commitment to the quality of a code. If you produce a product you do not feel completely confident in, there is a possibility that small errors can turn into big problems. These big problems can range from unhappy clients, insecure program, consequences when released (which may not follow the code of ethics and/or result in program failure), and more. According to our textbook, *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition),* errors found in the earlier stages of development will be much less costly than an error found in a deployed program. To avoid technical debt, early testing will be used to identify and eliminate errors throughout the development of a program. This will ensure that errors are not repeated from a previous stage to a current, newer one. Early testing will, also, be recommended when there is a time sensitive due date. If a due date is set and they do not do early testing, there is a great chance that time pressure will increase and cause thorough testing to be incomplete.

**References**

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter.

(2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition).* BCS The Chartered Institute for IT. Retrieved from  
https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb

Mirmirik, T. (2018, November 11). Cognitive Bias in Software Testing. Retrieved from

https://www.testeryou.com/cognitive-bias-in-software-testing/