Kendal Guizado O’riley

Project Two

Summary:

Our testing approach for the software requirements were such that we tested each of the specifications to ensure that they worked as intended. Starting with contact service we tested to ensure that contact service was able to add contacts with a unique ID, delete contacts per contact ID, and update contacts based on that contact ID. Furthermore, we tested the contact class to ensure that contact ID was no longer than ten characters or null, that first and last name were not longer than ten characters long or null, that contact phone number is no longer than ten characters or null, and lastly that the contacts address is no longer than 30 characters or null. We ensure that if any of these requirements were broken that an exception would be thrown. We then started testing task. With task service we started testing to ensure that we could add a new task with a unique task ID, delete task per task ID, and finally tested to ensure we were able to update the name and description fields per task ID. We then ran tests to ensure that task ID strings were no longer than ten characters or null, also that task name string was not longer than twenty characters or null, and lastly that the task description string was no longer than fifty characters or null. We ensured that if any of these requirements were not mad that an exception would be thrown. Lastly, we began testing on appointment service. We began testing appointment service by ensuring that we were able to add appointments with a unique appointment ID, and also tested that we could delete appointments per appointment ID. We then tested the appointment class by ensuring that the appointment ID string was no longer than 10 characters or null, also that the appointment date field was not in the past or null, and lastly that the appointment description string field was no longer than 50 characters or null. Throwing exceptions in the event that any of these fields failed the tests. Our tests ensured that exceptions were thrown if any of those conditions on the appointment class did not meet the requirements.

The quality of our tests was strong. We ran tests with test coverage to verify how much of the methods and classes were tested. Coverage for our class, method, and line % all reached 100%. Given us confidence in the test that we ran.

To ensure that my code was technically sound I first started with research on industry standards and best practices. I then went on to test the code against the requirements to ensure that what was written in fact performed as expected. The code below shows an example of a test performed. In this test, we ensured that an illegal argument exception would be thrown if the appointment ID was too long. The code below that, is the actual lines of codes we are testing to ensure they work properly. This line of code was written with the specification given that appointment ID should not be longer than ten characters or null.

@Test  
void testAppIDTooLong() {  
 Date testingDate1 = new Date(2024, 12, 03);  
 Assertions.*assertThrows*(IllegalArgumentException.class, () -> {  
 new Appointment("1234567891011", testingDate1, "Testing Description");  
 });  
}

if(appID == null || appID.length()>10) {  
 throw new IllegalArgumentException("Invalid ID");  
}

To ensure that my code was efficient again I started by research. I defined what was being asked of me, and then proceeded to gather information on how to achieve this. Early on, I realized that creating a display menu, with menu items would be the most efficient way for a user to navigate the program. We gave the users options on a screen, and based on the input of the user, the program would behave in a certain way. In order to achieve this, we used switch and cases. The code below is for case 2 of the contact service class. In this case, the program is performing the task of deleting a contact based on the contact ID, as efficient as possible. We used this template of cases in each of the programs to navigate between requirements given to us, that the user would need to have access too. This made the process of adding, deleting, and altering information based on an ID number very simple.

case '2':  
 System.*out*.println("\nEnter contact ID");  
 Scanner in = new Scanner(System.*in*);  
 String info = in.nextLine();  
 contact contact = *findPerson*(*contactList*, info);  
 if(contact != null) {  
 System.*out*.println(contact.getFirstName() + contact.getLastName());  
 System.*out*.println("\nAre you sure you want to delete contact (Y/N)");  
 String delete = in.nextLine();  
 if(delete.equalsIgnoreCase("y")) {  
 *contactList*.remove(contact);  
 System.*out*.println("\nContact deleted");  
 *displayMenu*();  
 }  
 else {  
 System.*out*.println("contact not found");  
 *displayMenu*();  
 }  
 }

Reflection:

For this project both white box and black box testing techniques were employed. White box testing was used to test the inner working of the system. For this to be performed properly I was required to know the code very well. Since I wrote it, I was very familiar with the inner workings of the program. Black box testing was used to ensure all requirements were met. I set up tests to ensure that each one of the requirements functioned as expected.

Testing not performed in this project were for security, performance, and readability. They were not part of the requirement, and in the normal testing cycle of the development process, I believe that in a real-world scenario these tests would be performed by someone independent from the development team.

Black box testing practical use is that it tests that the specified requirements were met. An implication of not performing these tests could cause you to deliver software that is not as specified or requested by the user. White box testing is more for the inner workings of the code. The tester most be very knowledgeable of the code to find bugs or issues that may present themselves once the program is released. Not performing these tests could lead to deliver of software that can be full of bugs and issues. Non-functional testing practical use is to ensure delivery of secure software. Without security the implications are that external forces could hack into a system and steal or alter sensitive data.

This was my first time working as a software tester on a project. Prior to this I really did not have caution when writing code. I simply wrote what I thought was correct, try to run the program and if it worked great, and if it did not, I went back to the drawing table. Of course, this led to writing lines of code that I believed to work correctly, but in fact could have given wrong results, or the illusion of working properly when in fact there were bugs. This project changed my mind set in the sense that now I understand that when we write code, we must test that the classes work as intended, the methods as well, and lastly our lines of code.

Limiting bias was a little difficult. When you write code and spend hours of your time doing research to write something you believe to be correct, it can be difficult to look at it with an open mind. But after reading so much on this subject, I tried to keep an open mind, and try to find as many issues and solutions as possible.

Being disciplined will be not only important for the code I write, but also important to the career I chose to embark on. Without being disciplined I am of the believe that it will be very difficult to make it in the industry I want to join. It is important to not cut corners for this very same reason. Your career path will be linked to the quality of code that you write. Not only that, but your clients and users of the software you are writing are depending on you as the developer to deliver code that is functional for their needs and aspirations. To avoid technical debt in the field I will try to keep a learning mind. By this I mean, I will do as much research as possible on this subject, and continue to keep with up industry standards, best practices, and new technology that becomes available to us as time progresses.