CS 320 Final Project

Testing is paramount for the success of a project. It is important to be thorough with tests and keep the projects requirements top priority – for example, in Appointment.java, appointment ID length could not surpass 10 characters. In AppointmentTest JUnit I implemented

“assertThrows(IllegalArgumentException.class, () -> appt.updateApptId(longId));” to test for long input. To keep my tests organized, I name-matched the primary classes with their test class, this way I could ensure every class was accounted for. Similarly, I kept naming conventions similar throughout the project to maintain good readability.

I followed a step-by-step approach to the testing by working on one requirement at a time to maintain the integrity of my tests. By testing one thing at once, I can verify that I am testing every required feature. Since I am newer to JUnit testing, you can see my tests got better over the course of the project. At first, my tests had poor structure but as I practiced and studied JUnit more, I was able to build a solid test program with good coverage.

It is important to create technically sound code. To ensure my code was sound, I took measures like using ArrayList data structures. For example, in the AppointmentService class I used “final private List<Appointment> apptList = new ArrayList<>();” to keep a list of the appointments. I used simple algorithms (add, remove, equals) to append the appointment list. For example, in AppointmentService I used ‘remove’ to remove an appointment based on its ID: “apptList.remove(findAppt(id));”.

To keep code efficient, I removed any imports that were unused, and I cut down on any code or functions that were redundant or not used. I ensured all variables were declared before they were initiated. For input tests, I gave the program examples of good and bad input to run through testing. For example, in AppointmentTest I ran “assertEquals(appt.getApptId().length(), 10);” to test for the length of the ID based on its parameters.

**Reflection**

The software testing techniques I used for this project fall under black and white box testing since we were specifically instructed what to test for. Black box testing is usually functional and is a testing method which the internal function of the item is unknown to the tester – usually in this case, the tester did not build the project and only tests specific functions. A lot of structure-based testing was also involved. Structure-based testing was used for coverage testing and to break the test down into sections.

For this test I did not use any experience-based testing. Experience-based testing relies on the testers real-world experience to test functions that a traditional test may not look for, or parts of a program that the tester usually knows will receive a lot of use/abuse. There are different types of experience-based testing: error guessing and exploratory testing. Error guessing uses prior experiences to determine the best test to use for the program. Exploratory testing is used more for testing areas that do not receive much testing, or do not have many specifications. Because of my vague JUnit testing, I did not use these methods.

The practical uses and implications of the testing methods I mentioned vary between the three. Black-box testing is used when functionality is clearly defined and most used in out-sourced testing because it does not require knowledge of the greater codebase. In white-box testing – the product is fully understood by the tester and is mostly used in cases where testing is done in-house. Experience-based testing are typically used to identify issues that are not always captured by traditional tests. Each of these tests are best practically used in different situations.

I limited my bias by never assuming a test will work. By forming hypotheses rather than making assumptions I can use logic to build a test case upon. Testing requires neutral thinking and must be devoid of bias to get good results. If I was responsible for testing my own code, it would be quite easy for me to make mistakes, for example: not remembering to test for a null ID field, and only testing for the ID length. This could result in an issue later where the computer accepts and empty ID field. Self-testing software also takes focus away from building solid code, whereas having a dedicated QA team test software, removes bias and allows developers to focus more on developing.

It is important to maintain discipline in testing. As a software engineer, it is my job to consistently deliver solid work that users enjoy and/or find helpful. Cutting corners and creating programs that have a lot of issues is an easy way to lose the trust of your products users. The software engineering code of ethics states that “software engineers shall act in a manner that is in the best interest of their client and employer”, meaning, as a developer, it is important for me to build code that meets users’ expectations. When working on a team, not maintaining discipline would also end up affecting your team, as more errors being found and sent back would cause more time and technical debt. Developing in an agile environment is great for building reliable code because it requires testing often. My goal is to always deliver consistent and stable code that functions the way my stakeholders expect.

References:

Shah, D. (2020, September 3). *Should developers test their own code?* Medium. Retrieved February 28, 2022, from https://medium.com/tenets/should-developers-test-their-own-code-c0a921de1a9c

Stf. (2020, September 17). *Black box testing*. SOFTWARE TESTING Fundamentals. Retrieved February 28, 2022, from https://softwaretestingfundamentals.com/black-box-testing/