Jonathan Pun

CS-320

Prof. Pettit

10/16/2022

Project 2

The testing approach aligned to the software requirements because it tested the required

methods that were implemented in the base code. In the contact class, the requirements stated

that the first name and last name cannot be longer than 10 characters. The ContactTest

JUnit implemented assertions to test for inputs that are too long. The requirements for the task class was that the task ID cannot be longer than 10 characters. In the TaskTest JUnit, the “void testTaskIdTooLong()” method was implemented to throw an illegal argument flag if the task ID is greater than 10 characters.

My comfortability with JUnit testing is still not that great. The resources that were provided in the modules helped a lot during the process, but I still need more practice to solidify the knowledge. The coverage percentage was higher in the contact test than in the task test. However, in both functions, the service tests had a much higher coverage percentage. The positive percentage let me know how much of the function that the test covered. I intend to use these more often to ensure that each statement is being implemented in at least one test case.

There were a couple measures that I took to ensure that the code was secure and sound. I attempted to use data structures such as arrays. In the contact class, I used “private static List<String> CONTACT\_IDS = new ArrayList<String>();” to make a list for the strings. I tried to use common algorithms such as equals, add, and length. For example, “taskId.length()” in the task class, “getFirstName().equals("Jane"));” in the contact test JUnit, and “contactService.add(c1));” in the contact service JUnit. I also used “assertequals”, “asserttrue”, and “assertthrow” assertions.

Coding is not my strong suit, so I used multiple measures to ensure that the code was

efficient. I watched so many tutorials, spent a lot of time on google, and used the provided resources, and fixed errors in the code one at a time with trial and error runs. I made sure to declare variables before initiating them. In the task service JUnit, “if (!alreadyPresent) { tasks.add(task); System.out.println("Contact added successfully!"); return true;” tested to see if the task id was already used prior to allowing it to be added. In the contact service test, I tested correct contact input and incorrect contact input.

Reflection

The software testing techniques that I employed for the milestones would be under the

black box and white box categories because they were all specification-based or structure-based.

Black box techniques “derive test cases directly from the specification or from some other kind of model of what the system should do” (Hambling et al., 2015). Black box techniques include equivalence partitioning to test for both valid and invalid inputs, decision tables to test conditions and actions, and state transition testing used to test events that change the state or generate outputs, use cases that are made from test cases, and boundary values to test boundaries. Structure-based testing was used a lot in the coverage tests to analyze components and in the if then statements. It is used to break down the tests into sections to be tested. Structure-based techniques include statement coverage, path coverage, and branch coverage. Structure-based testing techniques “are used to explore system or component structures at several levels” (Hambling et al., 2015).

The testing techniques that I did not use for the milestones are the experience-based techniques. According to Knovel, these techniques “use the users’ and the testers’ experience to determine the most important areas of a system and to exercise these areas in ways that are both consistent with expected use (and abuse) and likely to be the sites of errors – this is where the experience comes in” (Hambling et al., 2015). The specific techniques include error guessing and exploratory testing. Error guessing is where prior experience is used to determine which tests would be best to analyze the code. Exploratory testing is used to test areas that are lacking specifications. I omitted these techniques due to my limited experience with testing.

The techniques described above include black-box techniques, white-box techniques, and

experience-based techniques. Black-box techniques are used when code segments have clear-cut

functionality and is generally used for outsourced testing. White-box techniques are used when

the desired outcome of the product being tested is thoroughly understood. Experience based techniques are to “identify special tests that may not be easy to capture by the more formal techniques” are implemented “where specifications are either missing or inadequate and where there is severe time pressure” (Hambling et al., 2015). Black box and white box techniques are implemented based on their practical uses and implications for different software development projects and situations.

My mindset while working on this project was analytical and trying to ensure that everything was correct as I went so I would have to do as little bug squashing as possible. I supported this mindset through extensive research and testing by watching tutorials and implementing many trial and error runs. It is critical to appreciate the complexity of the code because of how much it impacts the quality and performance of the product. When I originally conducted my coverage tests, the coverage percentage was low. My finished product, which implemented more tests and covered much more of the code, pushed the coverage percentage up over the 80% requirement. The additional tests ensured that the code was technically sound and the overall product was high quality. I attempted to limit bias in my review of the code by testing everything multiple times regardless of whether I knew it would work correctly or not. I tried to form hypotheses rather than assumptions, because we all know what happens when we assume. Therefore, I could see how bias would be an issue if I were responsible for testing my own code. For example, if I only tested the function which checked that the length of the first name was no longer than 10 characters and chose not to test the ID, I may not have caught the fact that the ID test was not working due to an omitted line of code. Testing multiple inputs such as both valid input and invalid input versus just one or the other also helped to limit bias.

It is important to be disciplined in the commitment to quality as a software engineering professional because it helps uphold and “advance the integrity and reputation of the profession” (Software Engineering Code, 2018). It is important not to cut corners to avoid compromising quality and performance of the finished product. The expectations of a software engineer are outlined in the software engineering code of ethics which states that “software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest” (Software Engineering Code, 2018). To avoid falling behind, I plan to implement agile development techniques which includes constant testing and communication to keep the code secure and working as it is developed. The agile methodology will help mitigate technical debt as anyone develops code in any scenario. This is more for coding new things that aren’t normal or standard, but is still a good option to keep yourself honest during the development process.

References

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015).

Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) -

4.1.1.1 The Test Development Process (K3). BCS The Chartered Institute for IT.

<https://app.knovel.com/hotlink/pdf/id:kt00UC2IS4/software-testing-an-istqb/testdevelopment-process>

Software Engineering Code. ACM Ethics. (2018, December 19).

<https://ethics.acm.org/code-ofethics/software-engineering-code/>.