Ziad Hussein

CS 320

Project Two

**Summary**

The testing approach was very suitable to the software requirements since it tested the required methods that were used in the code. The requirements for the contact task state that the first name and last name cannot be longer than 10 characters. Therefore, the ContactTest JUnit implemented the assertion “Assertions.assertThrows (IllegalArgumentException.class, ()-> {new Contact("CID12345678", "Frederickson", "Schrefflerski", "12345678901","45678 The Longest Rd. Farawayville, KS 45678");” to test for input to make sure it is not too long. The same requirements are applied to task id so it is handled the same there.

The quality of the JUnit tests is still a work in progress. I made use of the resources available in these modules and they have helped me out a lot. The coverage percentage for the contact test was higher than the one in the task test. On the other hand, for both tasks, the service tests had a much higher coverage percentage. Since they showed a positive percentage that’s a good indicator for how much the tests is covered the functions.

I took a bunch of measures to make sure that the code is technically sound. I tried using data structures especially arrays. For example, 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.

I’m still learning coding, so I made sure to use multiple measures to ensure that my code was efficient. I watched a bunch of tutorials and guides, as mentioned I made use of the modules’ resources, and fixed errors in the code using trial-and-error. 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.

**Reflection**

The software testing techniques that I used 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 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 smaller sections to be tested.

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). Each of these techniques are implemented based on their practical uses and implications for different software development projects and

situations.

The mindset that I adopted working on this project was analytical, experimental, and

aimed toward growth. I employed caution through extensive research and testing by watching

tutorials and implementing many trial and error runs. It is critical to appreciate the complexity

and interrelationships of the code because of how much it impacts the quality and performance

of the product. For example, 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 if 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 technical debt as a practitioner in the

field, I plan to implement agile development techniques which test code often, consistently push

for high-quality, demonstratable software production, and keep communication between the

client and the developers open. The agile methodology will help mitigate technical debt.

**Citation**

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.

Retrieved from

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

process

*Software Engineering Code*. ACM Ethics. (2021, August 15). https://ethics.acm.org/code-ofethics/

software-engineering-code/.