Dre’ Scheetz

CS 320

Mr. K

12/11/2021

Grand Strand Systems: Report and Summary

This follow-up summary and reflection on the mobile application development of contact, task, and application services will explain and analyze approaches to software testing for Grand Strand Systems. The summary will focus on requirement and JUnit-based testing, providing an overview of testing techniques and the derived quality. In addition, the mindset necessary for satisfactory software testing will be examined.

In brief, the methodology for testing the services began with establishing test conditions, test cases, then writing the test procedure (Hambling et al., pp 79). Identifying the test conditions was straightforward, as most were described in the requirements and rubric for each file submitted. Occasionally the test conditions were more subtle. For instance, to ensure the updating requirement's proper functionality, validating the updated variable's correctness was necessary. Test Cases began with writing the steps as comments inside the Java files, including the preconditions, instantiating the class to be tested. The inputs were then selected to produce the expected outputs. At this point, I started writing the test procedure. A before each statement was written for set-up. The comment pseudocode test cases were then transformed into JUnit tests. Many of the tests are similar, only differing in the context used. While these tests could lead to a pesticide paradox, I think the simplicity of the program deliverable renders the concern unnecessary. Since all testing conditions were correctly identified, the resulting approach aligns with the requirements. For instance, variables were tested with a valid, non-valid, and null input.

The quality of code is the best that I can deliver to my knowledge based on the coverage of the JUnit tests. All branches are tested, aside from a while loop, when generating a new ID. The method used to create the ID should have a minimal chance of generating the same key; however, confirming that a unique ID is valuable ensures the code's quality. Aside from the branches, I confirmed that line coverage was 100% by using the three services' class tests.

The soundness of the code is demonstrated through tests of correct and incorrect inputs. Validating both provides for robustness as seen in the following, “public void testUpdateTaskName() {taskService.updateTaskName(testKey, "New Name");

assertEquals("New Name", taskService.getTaskName(testKey));

assertThrows(IllegalArgumentException.class, () -> {

taskService.updateTaskName("Not a Key", "Task Name"); });”

Efficiency of the code can be seen through the usage of a HashMap and UUID. The HashMap provides for ideal Big O time complexity (“public Map<String, Contact> contactList = new HashMap<String, Contact>();”) and UUID allows for cryptographic generation of IDs (“String initialID = UUID.randomUUID().toString(); return initialID.substring(initialID.length() - 10);”). Refactoring also increased efficiency through reduction of unnecessary variables and proper utilization of the Java constructor as suggested by Professor Kalinowski:

“public Appointment(String appointmentID, String appointmentDescription, LocalDate appointmentDate) {

this.appointmentID = setAppointmentID(appointmentID);

this.appointmentDescription = setAppointmentDescription(appointmentDescription);

this.appointmentDate = setAppointmentDate(appointmentDate); }”.

Software testing techniques were employed straightforwardly to meet and validate requirements. Fake objects are represented by the service HashMap instead of a database. I used mock objects when testing validation methods. For example, I instantiated the appointment class object to call the setAppointmentDate method to be tested for correctness. Interaction tests were also used to validate the appointmentService class add method calling the appointment constructor. State testing was used for validating variable assignments in the classes through static and non-static Junit Methods while utilizing EclEmma for code coverage.

Performance and security testing were some testing methodologies not implemented. Examining the Big O of the HashMap vs. other internal memory storage solutions to evaluate the software's speed, responsiveness, and memory consumption is an example of performance testing. Security testing involves methodologies to test data protection and functionality, which can be done through simulating external hacking penetration tests.

To explore the addressed concepts, consider a banking application. If a customer's account is not verified for accuracy, the program may not behave appropriately, allowing customers to spend money they do not have or funnel funds into a fake account. Penetration testing would be necessary to protect against fraud and identity theft. Performance testing would ensure that the network could maintain functionality with the volume of customers, and transaction speed is also desirable. Interactions between bank accounts would also need to be tested as most people use more than one account, and different methods would need to be executed, such as withdrawals and transfers.

Regarding the mindset of a tester, I attempted to remain as cautious and disciplined while being as unbiased as possible. The provided example of a banking application examines some of the many reasons why a tester needs to be cautious, as the potential damages can be immeasurable if escapement or defects occur. Caution can be inferred through the extensive testing and code coverage. The best example of complexity and interrelationships is the test of adding an appointment: “public void testAddAppointment() { assertFalse(appointmentService.appointmentList.keySet().isEmpty()); assertEquals(1, appointmentService.appointmentList.keySet().size()); assertFalse(appointmentService.appointmentList.containsKey(testKey) &&

appointmentService.appointmentList.get(testKey) == null);

assertEquals("This Description", appointmentService.getAppointmentDescription(testKey));

assertEquals(LocalDate.now().plusDays(1), appointmentService.getAppointmentDate(testKey)); }” Through assigning a new appointment to the HashMap, three of the Java files are implemented while the verification methods and ID generation are executed.

As I was the developer and tester for this project, an inherent bias exists. I attempted to limit bias through openness to feedback and critical examination of the logic and code written. Initially, I had not considered writing a test for valid inputs and only tested for non-correct input, which was included when I incorporated EclEmma and received feedback.

Lastly, discipline is apparent by adopting a standard style and inline comments. Readability is essential for collaboration and maintenance when writing code. Readability and reduction of refactoring are two of the main ways I plan to avoid technical debt. One principle to avoid cutting corners, thus reducing refactoring, is the adoption of non-pessimization. Per Muratori, non-pessimization ensures the computer is only doing what is necessary (2021, 6:00). An excellent example would be memorization inside a recursive function, like a Fibonacci algorithm. Although exhaustive testing is impossible, being as thorough as possible will demonstrate commitment to responsible testing. Keeping these concepts in mind for the future will enable me to be a valuable addition to a software development team.

Reference

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). *Software Testing: An ISTQB-BCS Certified Tester Foundation guide - 4th edition* (3rd edition). BCS, The Chartered Institute for IT.

Muratori, C. [Molly Rocket]. (2021, October 18). *Refterm Lecture Part 1 - Philosophies of Optimization* [Video]. YouTube. https://www.youtube.com/watch?v=pgoetgxecw8