Matthew Oliphant

CS-320: Software Test Automation & QA

Southern New Hampshire University

October 15, 2021

**Summary & Reflections Report (Grand Strand Systems)**

**Summary:**

**a**. The system requirements for the contact class applied to each of the class instance variables. Contacts each contained a contact id, first name, last name, phone number, and a street address. The constructor checked that each of these attributes contained an acceptable value and threw an exception if any were null or longer than required. The contact id was also final and cannot be changed. The contact id, first name, last name, and phone number are required to be strings and cannot exceed ten characters in length. The phone number must be ten digits exactly. Lastly the address is required to be a string that cannot be longer than thirty characters in length.

The contact service class required for contacts to be added, deleted, and updated from an in-memory data structure. The contacts have to be added with a unique id, deleted with a contact id, and allow for the first/ last name, phone number, and or address to be updated. I believe my code aligned to the software requirements by accounting for all of the above during object construction and JUnit testing. I’ve included the check on the contact phone number below as an example of adhering to software requirements:

if (phone == null || phone.length() != 10 || !phone.matches("[0-9]+"))

throw new IllegalArgumentException("Invalid phone number");

The other two features (task and appointment) work exactly the same way, except for the number/ names of attributes and their required length. The task class requires a task id, name, and description that cannot be null or longer than ten, twenty, or fifty characters respectively. The task service class works the same way as contact service class, and must be able to update task names and descriptions. The appointment class required an appointment id, date, and description and cannot be longer than ten, current date, and 50 characters respectively. The task id and appointment id cannot be changed once set. The following code ensures the appointment date is in the furture:

if (appointmentDate == null || appointmentDate.before(new Date()))

throw new IllegalArgumentException("Invalid appointmentDate");

My JUnit tests checked the boundaries and conditions for the software requirements, and were effective; the coverage for the overall project was 84.8%. Each of the final features utilized a hash map for the internal data structure.

**b**. When I first wrote the JUnit tests, the coverage was around 60% and they performed “false tests” that passed, but also didn’t do anything. I realized this when I performed regression tests which led to the tests failing. I ensured the code was technically sound by simplifying it and running *many* tests. Some some unit tests are designed to fail and lead to unavoidable dead code (constructing the object), as outlined below:

@Test

void testAppointmentClassAppointmentIdNull() {

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Appointment(null, new Date(), "This is a description");

});

}

This code utilizes a lambda expression to create a new appointment, but will fail to construct the object due to a null id. The JUnit test passes (the code throws an illegal argument exception), but since the object cannot be constructed, the new appointment instantiation is dead code. To ensure my code was efficient I performed a check for each object that it’s properly constructed and can be added, deleted, and or updated from the internal data structure, as demonstrated below with the task instantiation:

@Test

void testTaskServiceClass() {

TaskService taskService = new TaskService();

Task task = new Task("12345", "Name", "This is a description");

assertFalse(taskService == null);

assertFalse(task == null);

}

**Reflection:**

**a**. I employed both white-box and black-box tests in this project. White box testing techniques included path testing and condition testing. Loop testing wasn’t applicable to this application at this stage. The black box testing techniques included functional (unit and acceptance tests) and regression techniques. Functional testing can be summarized as unit tests. Non-functional testing checks aspects such as performance, usability, and reliability to name a few, which I did not perform for this project. Below is an example of a unit test that tests for an exception if the provided first name is too long:

@Test

void testContactClassFirstNameTooLong() {

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Contact("12345", "FirstNameIsTooLong", "LastName",

"7890123456", "1234 Main Ave");

});

}

White box tests are useful for programmers while black box tests are useful for testers. Path testing is a test that checks each path in the control flow graph (GeeksforGeeks, 2020). This is useful because it ensures each path within the code is ran, thus preventing most dead code. Loop testing (not performed here) checks the functionality of loops and ensures the increment and exit conditions perform as expected. Condition coverage essentially checks if-else statements to also prevent dead code. Testing early and often can reduce errors and implies the current code functions as intended, which means it most likely isn’t the issue when new code is added (Hambling, 85-126).

**b**. While working on this project, my mindset that I adopted was to be as mindful as possible when designing and developing code. I worked very hard on trying to identify my bias as a developer and tried to focus more on the testing aspect of the project before coding. The design thus incorporated testing which made identifying issues while testing a breeze. I employed a lot of caution, but also feel as if I could have done more testing; I could have tested for a change in feature ids, but assumed the final keyword was enough.

I tried to limit bias in review of the code by admitting I could probably do more ensure the code is rock solid. I also stated out loud what the requirements were and voiced my thought process so that I could challenge any assumptions or identify any confusion. I absolutely agree that bias is an issue when writing code because I probably think code works possibly better than it does. It’s easy to write JUnit tests that do not perform as intended; I used an assertTrue instead of an assertFalse in a test and it failed.

Quality is difference between failed and successful code and is ensured through testing. It’s easy to be lazy as a software developer so staying disciplined to a commitment of creating quality software is a must for success. Failing software can put lives in jeopardy. JUnit testing is a very good tool that allows for hundreds of tests in a couple seconds. I plan to perform JUnit tests on all code in the future to ensure the quality meets industry standard; I also plan on studying IEEE documentation to learn more about standards.

**Citations:**

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition). BCS The Chartered Institute for IT. Retrieved from https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb

GeeksforGeeks. (2020, July 2). *Path Testing in Software Engineering*. https://www.geeksforgeeks.org/path-testing-in-software-engineering/