Mark Meyer

Angelo Luo, M.S.

CS-320-T1023

Southern New Hampshire University

October 17, 2021

**7-2 Project Two**

The testing approach I used closely aligned to the software requirements such that all requirements were met and tested against. For both Contact Service and Task Service, I created an array list to hold and maintain the class objects. When the objects were created, each parameter of the constructor ran through specific conditions before the object could be successfully created and the parameters could be assigned to their respective variables of the corresponding class object. For example, when a Task class object was created, if the task id was null or more than 10 characters, an illegal argument exception would be thrown. I used JUnit 5 testing to make sure the illegal argument exception was thrown for both the event that the task id was null in the constructor and whether the task id was too long. I used this same method for each parameter for the respective services. The Contact object required a contact id, first name, last name, address, and phone number, whereas the Task object required a task id, task name, and task description.

Given the nature of the conditions of each project, I was able to easily modularize the code to be able to use the same logic and only had to change the variable names, slight modifications to the testing conditons, and changed the number of tests to align with the number of parameters and their required test conditions. Based on coverage percentage, the Contact Service project tested with an overall coverage of 90.7%; on the other hand, the Task Service project tested with an overall coverage of 95.7%. The consensus for successful test coverage is 80%, so I am confident that the overall quality of my JUnit tests is exceptional as all requirements have been tested against with ample coverage.

To ensure that my code was technically sound, I used software best practices to be proactive in testing the parameters against the required conditions. For example, in the constructor for each class object, if any fail conditions were met, an exception was thrown, and then (and only then) was the parameter able to be updated in the constructor:

// throw illegal argument exceptions for invalid inputs

if (contactId == null || contactId.length() <= 0 || contactId.length() > 10) {

throw new IllegalArgumentException("Invalid contact ID, must be a length of 1 to 10 characters.");

}

else {

this.contactId = contactId;

}

In addition, to combat the ability of updating a unique id of the class object, I did not create a setter (only getters) for either id, and I also declared the String itself to have access modifiers ‘private’ and ‘final’ as you can see here:

private final String contactId; // > 0 && < 11 chars, !null, !updatable

Most importantly, the ‘final’ access modifier essentially makes the variable a constant—disallowing the value of the variable to change. Furthermore, in the functions that allow the user to update the respective Contact or Task, there is no option given to update that particular parameter.

To ensure that my code was efficient, I gave each service function a return type of boolean:

public boolean deleteTask(String id) {

// Look up task using id to find index

for (int k = 0; k < taskList.size(); k++) {

if (taskList.get(k).getTaskId().equalsIgnoreCase(id) ) {

taskList.remove(k);

return true;

}

}

return false;

}

That way when I created the JUnit tests, I could easily use the ‘assertEquals’ assertion to quickly determine whether the function was successful in its operations. Furthermore, for each condition tested, I utilized a similar format of inputs to increase readability and future maintainability. Noticeably, to make my functions efficient as possible, I used ‘for loops’ (shown above) to iterate through each array list to quickly locate the object using a String id, then executing the necessary operation upon success of locating the correct object. Finally, in formulating my assertions for the test cases, I tested only what was necessary:

@Test

void testDeleteTaskSuccess() {

TaskService tService = new TaskService();

Task one = new Task("Task TST 1", "Twenty character tsk", "Fifty character task description testing here 1234");

Task two = new Task("Task TST 2", "Twenty character tsk", "Fifty character task description testing here 5678");

Task three = new Task("Task TST 3", "Twenty character tsk", "Fifty character task description testing here 9123");

assertEquals(true, tService.createTask(one));

assertEquals(true, tService.createTask(two));

assertEquals(true, tService.createTask(three));

assertEquals(true, tService.deleteTask("Task TST 1"));

assertEquals(true, tService.deleteTask("Task TST 2"));

This was to ensure proper, logical, and practical behavior of the program rather than doing unnecessary and exhaustive testing for such a simple project. Conclusively, the software requirements ultimately determine the overall design and implementation, and JUnit 5 certainly helps in making that process efficient at a higher standard of quality.

Software testing is a fundamental cornerstone of the software development lifecycle. Specific techniques are utilized in optimizing potential for error-free software. The techniques I use in the ‘appointment service’ program are boundary value analyses and state transitions. A boundary value analysis involves the testing of values surrounding the limits created by the conditions of customer requirements. For example, the appointment ID in this program has a requirement to be less than 10 characters in length; therefore, it is only necessary to test the values of the appointment ID just within the boundary of an acceptable appointment ID length and a single increment outside of this boundary such that the appointment ID length first becomes unacceptable according to the conditional statement. This same concept applies to testing the fifty-character limit of the appointment description. Using boundary value analysis, the appointment description string length would only need to be tested at a length of 50 characters and tested again at a length of 51. This technique is sufficient and efficient in testing whether the condition of the requirement is operating correctly.

Using state transitions is a technique used to validate correct inputs that modify the state of a program. For example, when adding, updating, or deleting appointments, the list of appointments changes its state by either a change in length or a change in a list item value. To effectively test these state transitions, an appointment is created that passes all constructor conditions, and then an assertion is made to ensure that what was added, updated, or deleted matches the expected result. In the case of appointment updates, there are potentially several tests that need to be performed because each option requires testing the passage and failure of each state change. There are other testing techniques that were not used in this program because they were not necessary for the job.

These techniques include equivalence class partitioning, decision-table-based testing, and error guessing. In equivalence class partitioning, domains of inputs are typically defined to identify data sets that may be considered the same as other values of the same class. Doing so may allow you to identify and test equivalence among classes. An example of this would be testing the values 100-110 as valid, 111-119 as invalid, 120-129 as valid, and so on. Decision-table-based testing is used to identify combinations of inputs and events using the concept of cause and effect. This program was not complex enough for a decision table to be necessary, but a good example of this concept would be creating a truth table to identify conditions which may allow correspondence with or activation of a single event given a certain combination of truths. Lastly, error guessing is a technique that uses the experience and/or knowledge of the developer to quickly isolate typical issues that may arise from the given situation. For example, a developer might begin by listing all the possible defects and turn that into a series of tests and it might test such things as how a program handles special characters or blank inputs.

The mindset I adopted in working on this project was to modularize the code for reusability, keep it simple for readability and maintenance, be thorough as possible for the sake of meeting the client’s requirements and achieving a quality product, and be practical in the sense of testing only where necessary (ie. utilizing boundary value analysis) yet sufficient to meet the requirements. In acting as a software tester, I employed caution to the extent that I paid close attention to the requirements such that I did not add features or leave out significant boundaries. I made sure all IDs were unique, all fields were within their specified length requirements, and that each service was able to add, delete, and/or check for conditions such as making sure the appointment date was in the future.

It was important to appreciate the complexity and interrelationships of the code I was testing because when testing, it is important that your tests are actually reaching the inner depths of the code. For example, in my function “updateTask” in the file TaskService.java, this functon takes in 3 arguments, checks through the task list to match the id, then depending on the menu option provided will proceed into a switch operation that allows you to either set the task name or task description. In order for the JUnit test to reach the switch operation in this function, specific assertions have to be made and tested against. If a portion of the function is not reached, it means that the test code did not address that specific line of code (often shown in yellow).

In my review of the code, I tried to limit bias by maintaining the mindset that the ultimate goal is to make the client happy by meeting and adhering to all of their requirements. Any addition or removal of requirements should be first discussed with the client before changes are made in code. I imagine that bias may be a concern if I was responsible for testing my own code for the simple fact that when I design a program, what I put in “writing” as code is what I came up with as my current solution for the task at hand. More importantly, there could always be a better, faster, more robust solution out there which is why peer reviews can help tremendously, especially if those peers have more experience.

Finally, the importance of being disciplined in my commitment to quality as a software engineer professional is that we are human and mistakes happen. Software engineers are one of the first lines of defense in the safety of the public when it comes to this ever-growing world of technology. We are the reason why neat things like robot vacuums occupy your cats for hours, and the reason why people are still able to work and go to school through a pandemic, and the reason why I can find my puppies if they ever run away because of an embedded microchip. These things exist because an engineer made it happen. It is important not to cut corners when it comes to writing or testing code because the domino effect of doing so can be devastating—too often resulting in severe monetary loss and/or death—and the later the stage that a mistake is caught in the software development lifecycle, the more it costs to fix (otherwise known as technical debt). In order to avoid technical debt as a practitioner in the field, I plan on being thorough in my requirements gathering, analyses and designs, development, testing, and I will make my best effort to perform as much static and dynamic testing as possible throughout each stage.

**References**

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 October 17, 2021 from  
<https://app.knovel.com/hotlink/pdf/id:kt00UC2GB1/software-testing-an-istqb/fundamenta-introduction>