Summary and Reflections Report

Similoluwa Adeyemi

CS-320

Ramsey Kraya

8-10-2023

To meet the requirements given by the client, all fields that require user input are actively validated. For example:

public void setId(String id) {

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

throw new IllegalArgumentException("Invalid ID");

}

this.id = id;

}

These lines of code ensure that a user ID is no longer than 10 characters as specified by the requirements. In the event that the ID given is longer than 10 characters, the code generates an error message saying, “Invalid ID”. Checks like these are prevalent across the entirety of the code all to ensure that then requirements are being met.

Each JUnit test had a 100% coverage of its corresponding .java file. In other words, every JUnit test completely verifies every aspect of the .java code under test.

The code was designed to be technically sound by checking for, and throwing appropriate exceptions at various points such as line 11 of the Task.java file:

throw new IllegalArgumentException("Invalid Task Id");

This line throws an exception after checking that the task input given is valid and without error, and if it fails this check, an exception is thrown. A similar task is accomplished in TaskTest.java:

*assertAll*("Id tests", () -> *assertEquals*(task.getId(), idTest),

() -> *assertThrows*(IllegalArgumentException.class,

() -> {task.setId(longId); }),

() -> *assertThrows*(IllegalArgumentException.class,

() -> {task.setId(null); })

);

This block of code tests the setId() method under various conditions to ensure that they meet the requirements of not being longer than 10 characters, and also not being “null”.

Code efficiency was achieved by avoiding redundancy and automating various tasks. For example, I developed a method to efficiently search through all contacts or tasks stored in their respective lists. This is given in ContactService.java as:

private Contact contactSearch(String id) throws Exception{

for(int i=0; i < contactList.size(); i++) {

if(id == contactList.get(i).getId()) {

return contactList.get(i);

}

}

throw new Exception ("Contact not found");

}

Various design elements such as this were implemented to streamline the code, reduce the number of lines needed to accomplish various actions, and keep the code organized and readable with industry standard naming conventions and practices such as commenting to describe what certain blocks of code are meant to accomplish.

*@Test*

void deleteContactTest() {

ContactService contact = new ContactService();

contact.newContact();

*assertThrows*(Exception.class, () -> contact.deleteContact(idTest));

*assertAll*(() -> contact.deleteContact(contact.getContactList().get(0).getId()));

}

This test tests one of such features. In this case, the deleteContact() method is tested, to ensure that the code can efficiently and accurately remove a contact object when given the appropriate contact id.

The major software testing technique utilized for all three modules was unit testing. Unit testing involves testing a specific block of code to ensure that a bare minimum functionality is met. This can involve testing a single method, class, or even a single line of code. For example, in my module 5 submission “AppointmentTest”, each test goes over a specific method from the “Appointment” class, testing the base functionality of each one and ensuring that they produce the expected results.

As of this module, we are yet to employ a system test. A system test takes all units and tests them as a collective. The goals are to test the entirety of the software to ensure that each part is working in unison to achieve whatever goals have been set for that piece of software.

For most large-scale systems, unit testing on a grand microscopic scale is largely impractical. The scope of what a “unit” is, is largely subjective, and for larger projects, a unit would be much larger than a smaller projects unit. Unit tests and system tests should be used together regardless of the scope of a project. Using them together ensures that the project is consistently tested at all levels and that most bugs and errors are caught and fixed before a product is shipped.

Taking on the role of Developer-Tester was rather challenging. It wasn’t enough to just write passable code, the code had to be secure while meeting all requirements consistently. Caution went into every facet of the code, both writing and testing. I had to be cautious to make sure the code was achieving what was intended, and even more careful to ensure that the tests were relevant and achieving what I needed them to accomplish within the scope of the project. I would say that my personal bias toward my code really shone in this project. There were multiple failed tests which I believed should have been successful. I thought my code was flawless, so I spent more time than I’m proud to admit searching through the wording of my tests to find what error I was making as a tester, rather than questioning my skills as a developer. As a professional, I imagine this kind of ego would be even stronger and more difficult to shake off in order to view the work done objectively. One specific mistake that I made consistently was failing to test boundary values. I found myself not considering this as necessary and simply choosing not to include those tests because “surely my code was good”. This sort of attitude is what leads to much bigger issues and missing very important, very avoidable, errors while writing code. After eventually deciding to add the boundary tests I did find some things that would be important to include, such as ensuring that the code didn’t accept any negative values and making sure that the code accepted values at the boundary. In the future I will strive to ensure that all relevant tests, no matter how big, small, or insignificant they might seem, will be made, and run to ensure that all code developed in my purview are to industry standards and never cut any corners to get their green checkmark.