Dalton Gollihue

SNHU  
CS-320

10-19-2025

We started this project by reviewing the requirements that were given to us in the early modules to get a feel for what was coming. As well as the instructions that came with each part of the process. The contact class begins with a feature that requires the user's first name to be no greater than 10 characters. So, we have this requirement, now we make our code to allow the user to make a first name. We then test the length of their first name by creating a test that fails if the name is >10 characters. It’s worth noting that the code will also fail if no name is entered. This is to make sure that a name is entered, and that it won’t exceed the 10-character requirement.

In the task service portion, we have a similar way of setting up tests and satisfying the requirements. We set up code that allows the user to create tasks, along with some logic that can update descriptions, change names of tasks, remove tasks, and add them. To test to see if the feature of updating the description is functional, we simply write a test that tries to use the update description functionality and if the new description and the old one match, it triggers a failure.

In the appointment portion, we have a very simple and similar way of testing the class. We set up the logic to make appointments, complete with dates, a unique appointment ID, names, and descriptions. To test a feature such as adding appointments requiring a date, as opposed to leaving the field blank, we simply trigger the failure when the date field is null in the printout of the appointment.

I feel like I got strong at making these tests over the course of the term. My contact class in module 3 was very much a struggle, and I wasn’t quite understanding the concepts very well. It was enjoyable to have the problem of writing code to test code. I had never known that such a thing would exist, but I’m really glad that it’s a skill that I’ve come to love.

I know my code was very technically sound because of some of the features I was able to add to it. For example, private Date futureDate(int offsetDays) {

Calendar cal = Calendar.*getInstance*();

cal.add(Calendar.***DAY\_OF\_YEAR***, offsetDays);

cal.set(Calendar.***HOUR\_OF\_DAY***, 0);

cal.set(Calendar.***MINUTE***, 0);

cal.set(Calendar.***SECOND***, 0);

cal.set(Calendar.***MILLISECOND***, 0);

return cal.getTime();

}

This will access the system's information to determine what day and time it currently is. This allows me to do some cool stuff in the appointment feature. I can use it to make sure the appointment dates and times are compatible with the present time and date. A possible way this could fail is if the user is using a device with some weird proprietary calendar and clock, or they are set to the wrong time, or perhaps they are in a different time zone. All of these things can be accounted for, of course, but I wanted to keep things simple. It’s always possible to force the user to set a time and date upon setup, but that would be too complicated when you can just import it.

I was very efficient at writing simple tests that would make sure that the requirements were intact, and that the limits of possibilities were reasonable. For example, *@Test*

*@DisplayName*("Test to Update First Name.")

*@Order*(1)

void testUpdateFirstName() {

ContactService service = new ContactService();

service.addContact("Jeff", "Danny", "1111111111", "123 Street ST");

String id = service.contactList.get(0).getContactID();

service.updateFirstName("Sam", id);

service.displayContactList();

*assertEquals*("Sam", service.getContact(id).getFirstName(), "First name was not updated.");

This is taken from the contact service test file, and it demonstrates efficiency. It simply looks to see if the input for the first name field changes in the array when the update first name functionality is used. If it changes, then the test will pass. If the first name remains the same, it will fail. I’m sure you can make more robust and more complex tests that will be as airtight as possible, but when you bring more complexity to the table, you are going to suffer the consequences of that complexity. Efficiency is often a short, elegant solution that won’t give rise to new issues, and I believe I demonstrated that will code like this, as well in other areas where I just have the test read the number of characters in an appointment ID for example. Very simple solutions for real and complex problems that engineers will run into all the time.

A lot of the process for me lies in working backwards. Often the most effective way to code is to start with large outlines and have things mapped out, but oftentimes I get stuck and will inevitably have to go backwards. It’s really easy to state that you want to have a big plan before you begin, but it’s important that you understand that you’re capable of making errors and accept that it’s totally fine to work backwards. It’s still important to plan and optimize ahead of time. That way, you aren’t restarting over and over again. I was really worried about getting started because I would worry, I didn’t have it figured out well enough, but I just needed to start. There is a ton of complexity that demands respect in the code. Particularly in this project, we have the long arrays of information that a user puts in which will then get tested in the test file. The long arrays are used in the tests to see what changes when certain functionalities are called on.

When I was making the appointment components of the project, I had an idea of setting hard dates for the date fields, but instead, I decided to use the systems information to tell the time. This added a layer of complexity, but I would also argue that it’s a way that I took my bias out of the code. I also could use this feature as a means to test itself when it came time to develop the test code. I had become used to making code that just checked for less than a certain number of characters, and it was important to step outside of that conditioning that I had set for myself. It can also be really hard to design code to test code that you made. You can overlook huge errors and assume errors don’t exist when they in fact exist.

It’s really important to take your time when you are doing the planning code, and make sure you take time to read the requirements. The big planning phases are so important that you aren’t doing the running backwards to fit in requirements that you had forgotten about. It’s equally important in testing because you might not be adequately testing code. It could come time to roll out the code, and you might suffer the consequences of your ego. It’s important to code without bias and ego, so that you may develop code that is as free from bugs as possible.