During the programming of the appointment, contact, and task services, I employed many different testing techniques. These testing techniques included libraries from the “java.text,” “java.util,” and “org.junit.” With these libraries, I was able to effectively test one-hundred percent of all available information in the appointment, contact, and task services. This means that my code was perfect according to JUnit, and this is to say that no issues could be found for what I was testing for. I stayed to the rules that were given to me. I made sure that each one of the variables were the correct length, weren’t null, had the proper naming conventions, that dates to be scheduled weren’t in the past, and that noneditable variables were not updatable. One such example is:

@Test

void testAppointmentDate() {

Date oldTest = new GregorianCalendar(2021, Calendar.APRIL, 25).getTime();

Appointment test = new Appointment("1", new Date(), "sample");

assertTrue(test.getAppointmentDate().toString().equals(new Date().toString()));

assertFalse(test.getAppointmentDate().toString().equals(null));

assertFalse(test.getAppointmentDate().after(oldTest));

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

new Appointment("1", new GregorianCalendar(1993, Calendar.APRIL, 25).getTime(), "sample");

});

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

new Appointment("1", null, "sample");

});

}

In this example, we see that I use assertions. I use these assertions to make sure that the date follows ALL rules given. These rules were that the date can not be in the past and it cannot be null. This example shows that I check the provided date to make sure that it is a real date, that it is not null, and that it is not in the past. I could have used a different command for the past check, but this will suffice for what I was using. I could have used, “new Date()-1,” which would be yesterday. This same snippet of code above shows how technically sound my code is as well. It shows that I followed all conventions, while maintaining the solidity of my JUnit tests, which shows when the tests are ran, and I achieve one-hundred percent coverage. One assertion is all that is needed to be efficient. This is also shown in the above snippet. It tests each area of the given requirements one time each.

As for the software testing techniques that I employed, I can give a simple list. This list includes examples of each as well:

* assertTrue() – Practical Use: Make sure an object is true.
  + Contact contact = new Contact("0000000001", "Douglas", "Bolden", "4233002190", "248 England Road");

assertTrue(contact.getFirstName().equals("Douglas"));

* + - This will not throw an exception because “Douglas” is equal to “Douglas.”
* assertFalse() - Practical Use: Make sure an object is false.
  + Appointment test = new Appointment("1", new Date(), "sample"); assertFalse(test.getUniqueAppointmentID().equals(null));
    - This will not throw an exception because “1” and “null” are not equal.
* Assertions.assertThrows() - Practical Use: Make sure an object throws an exception.
  + Assertions.assertThrows(IllegalArgumentException.class, () -> { new Task(null, "Call Meeting", "Call Daily Standup Meeting"); });
    - This throws an exception because of “null.”

Some of the other software testing techniques that I did not use include:

* assertEquals() – Practical Use: Check two strings to make sure they are the same string. (Banks could use this for duplicate transactions)
  + Asserts that two objects to see if they are equal.
* assertAll() - Practical Use: Checking of active game users for cheating/bug abuse.
  + Asserts that all supplied executables do not throw exceptions.
* assertNotNull() - Practical Use: Make sure that information is actually there.
  + Asserts that an object is not null.
* assertThat() - Practical Use: Making sure something happens. RECURSION Testing is a good example here.
  + Asserts that something has satisfied a condition. (Defined in the assertion call)

There is truly only one mindset that needs to be learned for any project. If you were a user of what you are making, would you be satisfied with the results? The insight that you need, can and will be gained from treating each project this way. You need to get out of your comfort zone and act as if you are a user of the software to truly get the full picture for this. Most programmers will act like they can easily achieve this, but “these same DP professionals propose "user-oriented" facilities, they are often subconsciously proposing systems that are more suited to their own needs and abilities than to those of the real end user”(McLean, 1979). With this information, it is easy to see how being a user is harder than meets the eye.

Caution should be taken while working among the complexity of any test. I suppose the best example would be:

* Constructs
* Setters
* Getters

I know that these can be extremely simple, but you must see it from a larger point of view. These can cause the biggest issues because of their ability to change code, or limit what code can do. Caution must be taken while setting or getting any bit of memory, as the memory can leak if not properly performed and allow software manipulators to see information that they shouldn’t see.

Bias is a true problem in programming, as it is human nature to see things the way that we want to see things. Even if you believe your own code to be perfection, there will usually be something that can be improved on. This can cause frustration to some. I never believe that anything I make is perfect. I have even seen some problems along the way while I am programming. This is human nature, and it is okay. People make mistakes. Other programmers can possibly find something that you didn’t see before-hand, possibly saving your job.

Technical debt can sometimes not be avoided, but it is very important to not cut corners. When cutting corners, it is easy to see that there will be problems. Being disciplined in quality as a software developer will help you point out more problems as you become a stronger programmer. Spotting issues becomes easier and seeing that allowing technical debt becomes more of an apparent issue. If you cut corners, you will most likely cause yourself to potentially lose your job, cost your company big money losses, and cause grief for the users of your product. As a programmer that is disciplined, I will abide by all standards set in place for me, I will follow naming conventions, and I will gather more information than is needed to make sure that my customer is completely and thoroughly joyful while using any program or feature that I produce. This should be the way that all programmers see this topic. Thank you for reading my report.

REFERENCES

McLean, E. R. (1979). End Users as Application Developers. MIS Quarterly, 3(4), 37–46. https://doi-org.ezproxy.snhu.edu/10.2307/249047