Jack Coster

12/10/23

SNHU CS 320 Project 2

Summary:

The software requirements for this project were outlined clearly at the start, so I tried to align my testing approach to be as in line with the expectations as possible. For instance, when writing the J Unit test for the “Contact” object, we were instructed to make sure that the object “shall have a required phone string field that must be exactly 10 digits. The phone field shall not be null.” You can see below how I set the phone string to hit the desired requirements below:

protected void updatePhoneNumber(String phoneNumber) {

      String regex = "[0-9]+";

      if (phoneNumber == null) {

        throw new IllegalArgumentException("Phone number cannot be empty.");

      } else if (phoneNumber.length() != CONTACT\_PHONENUM\_LENGTH) {

        throw new IllegalArgumentException(

            "Phone number length invalid. Ensure it is " +

            CONTACT\_PHONENUM\_LENGTH + " digits.");

      } else if (!phoneNumber.matches(regex)) {

        throw new IllegalArgumentException(

            "Phone number cannot have anything but numbers");

      } else {

        this.phoneNumber = phoneNumber;

      }

    }

I then made sure when writing the corresponding test for the contact object to create a contact that created a compliant contact and one that was designed to fail. I wrote my tests in a way that first tested that every field was not null, then test by test, would confirm that field equals its corresponding id as shown below.

 @Test

  void contactIdAndFirstNameConstructorTest() {

    Contact contact = new Contact(contactId, firstNameTest);

    assertAll("constructor two",

              ()

                  -> assertEquals(contactId, contact.getContactId()),

              ()

                  -> assertEquals(firstNameTest, contact.getFirstName()),

              ()

                  -> assertNotNull(contact.getLastName()),

              ()

                  -> assertNotNull(contact.getPhoneNumber()),

              () -> assertNotNull(contact.getAddress()));

This way we make sure we get full coverage on our J unit test because it checks line by line in the contact object that every field is compliant before moving onto the next piece. It may have been redundant for some of the tests, but I prefer to be thorough to get complete coverage.

To make sure my code was efficient throughout my build, I tried to use concise, easy to read code that adhered to modern code standards with proper indentation and spacing. I tried to keep my naming conventions consistent and not include any filler that was not needed to make the code run as intended and pass the required tests. For example, in my code for the appointment object, I first set my parameters for the “appointment id and description length”, then initiated a new instance of an appointment. I then wrote effective if/else statements to throw errors if the requirements were not met with clear error messages. Then I laid out the remaining CRUD operations below to get, update, delete the appointment so that all outcomes were available. You can see this below with an example of my update appointment:

public String getAppointmentId() { return appointmentId; }

  public void updateDate(Date date) {

    if (date == null) {

      throw new IllegalArgumentException("Appointment date cannot be null.");

    } else if (date.before(new Date())) {

      throw new IllegalArgumentException(

          "Cannot make appointment in the past.");

    } else {

      this.appointmentDate = date;

    }

  }

Reflection:

Throughout this project I tried to keep my testing techniques consistent across the build. I stuck to using individual unit testing before moving to integration testing then running system wide tests. As described above, this way I could check each individual piece of the object before confirming the entire object was compliant. This way when the object was ready to be tested system wide, I would be more confident that my different pieces of code would work together as intended and less errors would be found. I could have included other testing techniques like performance and acceptance testing, but they were not practical for this use case. In the real world they would be standard practice to make sure my code met all customer requirements and met performance standards. My tests running in sequence would take longer and could be slimmed down, but since it is only me contributing to this build and I wanted to make sure all aspects were included for review, the individual unit tests were the right decision to use. When working with a larger development team, each coder would probably be responsible for a smaller part of the overall build, so you would see a variety of tests run on the code before it was submitted for review as a complete project.

When going into this project and class, I was cautioned that it would be an intimidating process to build all these different tests, so I tried to plan from the beginning on how to make sure my code and tests were cohesive overall. Luckily, I have work experience writing J Unit tests and had a good foundation to use when considering requirements on this project, so it was not as daunting as expected. I did write my code cautiously at first by writing my individual unit tests in a way that they would not pass unless the test before passed to limit any bugs appearing at the end. With these types of tests, I believe it removes a lot of inherent bias by leaving out any doubt the tests cover everything needed from the requirements. I could have skipped over some of the smaller tests to save time and it would have still passed, but sticking to the plan of testing everything kept me from having any unknown bias in the build. This goes into the importance of being disciplined in my coding experience, in the beginning I would always be looking for short cuts to finish builds like this as quickly as possible. But I have found that by taking my time and checking each test as I went and keeping to a system of set standards, I was able to move more efficiently and make less mistakes and not having to go back and fix mistakes later on in the build helped the whole project run as intended.

Citations:

Hamilton, D. (2023). Junit tutorial with examples: Setting up, writing, and running Java Unit tests. Retrieved from https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/

(N.d.). Retrieved from https://www.javatpoint.com/junit-tutorial

(N.d.-a). Retrieved from https://www.headspin.io/blog/junit-a-complete-guide#:~:text=JUnit%20is%20a%20software%20testing,of%20automation%20testing%20with%20JUnit.