CS 320 Project Two Summary and Reflections Report

Shayne Rushton
matthew.rushton@snhu.edu
Southern New Hampshire University

Summary

My approach to testing each feature was to start with the object itself and then test the construction of the object. Once I got past all the reuqirements of the object, I would write and test the code for the actions required, sucha s add, update and delete a contact.

I made sure to pay attention to the requirements in the code so that i could use exceptions for when failing code in a test. That way I stayed cogniznt of the requirements while also considering how my Junit test would be written, minding the exceptions forh when failures were attempted and assertTrue to ensure valid values worked as well, I did that with all the actions for each class, also. Here, the Appointment object was created with the intention of not having duplicate appointments and no dates entered that were before the current day, which I was able to do with the following tests:

```
class AppointmentTest {

// Create a string variable to hold a date to used and changed if needed for a global application Format is yyyy, m, d

LocalDate aDate = LocalDate.of(2023, 1, 9);

@Test
public void testAppointment() throws ParseException { // checks if creating a new instance of appointment works correctly
	Appointment newAppt = new Appointment("123456789", aDate, "Because it's time");
		assertTrue(newAppt.getAppointmentId().equals("123456789"));
		assertTrue(newAppt.getAppointmentDate().equals(aDate));
		assertTrue(newAppt.getAppointmentDesc().equals("Because it's time"));
}

@Test
public void testAppointmentIdNull() { // passes if passing null value for ID fails
	assertThrows(NullPointerException.class, () ->{
		new Appointment(null, aDate, "Becuase it's time");
	});
}

@Test
public void testAppointmentIdToLong() throws ParseException { // passes if id is too long
	assertThrows(IllegalArgumentException.class, () ->{
		new Appointment("1234567891111", aDate, "Because it's time");
	});
}

@Test
public void testAppointmentDateNull() { // Passes if a null date fails
	assertThrows(NullPointerException.class, () ->{
		new Appointment("123456789", null, "Because it's time");
	});

}

@Test
public void testAppointmentDateToEarly() throws ParseException { // Passes if a date before today throws an error
	LocalDate earlyDate = LocalDate.of(2022, 1, 3);
		assertThrows(IllegalArgumentException.class, () ->{
```

```
                        new Appointment("123456789", earlyDate, "Because it's time");
                });
}

@Test
public void testAppointmentDescNull() throws ParseException { // Passes if the null description fail
                assertThrows(NullPointerException.class, () ->{
                        new Appointment("123456789", aDate, null);
                });
}

@Test
public void testAppointmentDescToLong() throws ParseException { // Passes if the descript is too long and fails
                assertThrows(IllegalArgumentException.class, () ->{
                        new Appointment("123456789", aDate, "Because 50 characters is way too many for this excercise so why are
you trying to write a book here and now?");
                });
}

@Test
public void testSetAppointmentDesc() {
                String myDesc = "stuf";
                Appointment.setAppointmentDesc(myDesc);
}

}
```

Because my tests covered 95.4% of my code, I am fully confident that it was covered. I noticed some places where branches in the code weren't reached, but my tests covered every possible condition, so I know the coverage was actually 100%. Using the color coding in Eclipse showed me where tests failed, and all tests were green. The only anomaly I noticed was in the code, but again, those lines were tested though the code was saying otherwise.

Writing JUnit tests was new for me. I have not yet tested code, so these exercises were great. I wasn't sure what run as coverage meant, but after a few searches and some hands on, I realized that it meant all the possible failures and successes being tested would be 100%. I also read that it is not always feasible to get 100% and that 80% is pretty good.

To make my code techincally sound, I made sure to closely follow the requirements and consider tests in my code. That way I could throw specific exceptions for null excpetions verses illegal arguments. It helped to make testing easier by knowing what excpetions I could expect in each scenario and made testing ultiamately easier.

I could have made my code more efficient. I have a code that looks for duplicate records. This code is not in the DIY realm as I wrote it for each object and it could have been a template code to allow any of the classes to accomplish duplicate object verification, but I can say that if this code was limited to just the one object, using this function was better for finding duplicates, and having had more time, I would have made another class so as to use one function to verify duplicates for all classes. The mehtod was called getClassName and accepted the ID for that object. This one is for finding a duplicate Contact:

```
public static Contact getContact(String id) {

            // Find the Contact object in the list and return it based on the
id given or return null;
            for(int i = 0; i < contactList.size(); i++) {
                  if(contactList.get(i).getID().equals(id)) {
                        return contactList.get(i);
                  }
            }
            return null;
}

I would call this script when adding a record so as to check the list for the
object before adding.
```

2. **Reflection**

I used assert for all of my tests. I mainly used assertTrue and assertThrows to determine if the correct values worked using assertTrue which returns true if the value is valid, and assertThrows which looks for a specific exception type and returns true if the expected exception is thrown by the incorrect data. I also used assertNull in some cases for ensuring that an object wasn't in the list and that helped ensure the delete function was working correctly. In the even that a null value was passed, I used assertThrows to test.

```
Void testAddTask(){
assertThrows(NullPointerException.class, () -{
         Task.addTask(null,"Booking","Reservation");
        });
}
```

With this test, if the id was passed as null, the NullPointerExcpetion was thrown and that would count as a passed test since the test was testing if an exception would be thrown in event of a null ID.

I mainly stuck to testing each function using asserts and tested for excpetions, null values and valid values. So mostly, just white box testing was done here. I didn't do any black box testing which may not have been practical anyway since I wrote the code and know how it works, but I could have had someone else test this for that technique. I also did not do any code review. I basically wrote the code to the specifications and tested it immediately after.

If I could have created a user interface and have someone try to enter specific data to test the outcome, that would have been another layer.

I would like to have someone do a code review as I think I have a lot of room to improve upon here.

For white box testing, it’s a great way to get the coverage in testing that you want. Even if you could possibly miss bugs, you at least know exactly how you want to code to work and what to do if failure happens. I think this approach is great for an initial test. After this, I think the black box testing would be a great asset, especially if the application was a much more complex app.

Black box testing is great becuase someone can break the code by now knowing what it is supposed to do, so they will be more likely to try things that perhaps the tester didn’t think of originally.