Project Two

CS 320

Kyle Ogle

For this project, I developed three features – the contact, task, and appointment services and their associated objects. For each feature I developed unit tests with the following approach: I first wrote a test for each attribute in an object, ensuring that inputs that failed to meet the requirements for each attribute would cause the application to throw errors. For instance, in the Contact object, I wrote a test that ensured that a “null” input for address would fail, and that an input greater than 30 characters would fail. After writing tests for each type of “bad” input for each object attribute, I wrote a test that ensured that “good” inputs for all fields would result in a successful instantiation. In this way, the test for each object ensured that the object met all requirements. For the services, I wrote one test for each method specified by the requirements. Because the object attributes were covered by the object tests, the service tests only had to ensure that the service methods were executed correctly. Because each object and service had 100% unit test coverage, the tests were effective in showing the code met the requirements.

To ensure my code was technically sound, I intentionally wrote tests with bad inputs and checked that these tests caused the application to throw errors. For example, in ContacTest.java:

// test that null ID throws an exception

Exception nullException = *assertThrows*(IllegalArgumentException.**class**, () -> {

**new** Contact(**null**, "firstName", "lastName", "1234567890", "address");

});

String nullExpectedMessage = "Contact ID must not be null and must be no longer than " + Contact.***NAME\_NUMBER\_ID\_LENGTH*** + " characters";

String nullActualMessage = nullException.getMessage();

*assertTrue*(nullActualMessage.equals(nullExpectedMessage));

This test showed that passing null for the Contact ID caused an error to be thrown as expected by the requirements.

The primary technique I used to test the objects was boundary value analysis. In this technique, errors are found at the boundary values of acceptable inputs rather than testing only for common or expected inputs. This was very useful in determining whether the boundaries were set correctly in the objects being tested.

The testing mindset was very important for this project. I had a problem in my mindset that I didn’t catch until the final milestone – which was that I was validating inputs during object instantiation but not during update executions. So it was possible to create an object with good inputs, and then update the object to have bad inputs. I was able to fix this by moving the attribute validation to the attribute setters rather than doing the validation in the constructor. Since the service tests didn’t validate input before attempting to update, the unit tests failed to catch this defect. This example shows how important it is to avoid bias and be disciplined in writing software and tests, because exhaustive testing is impossible and testing cannot show the absence of defects.