**Summary and Reflections Report**

Justin Haby

Southern New Hampshire University

02/17/2022

Professor Rangitsch

Project 2 Submission

**Summary and Reflection Report**

**Summary**

This unit testing approach aligned with the application requirements will cover the testing of the business rules for the attributes and services. For example, it ensured that each appointment had a unique id, date, and description, which did not exceed 50 characters using assert Equals,assertNotNull among other methods.

For example, in Appointment Test.

These are the business rules and code snippets used to achieve that:

The appointment object shall have a required unique appointment ID string that cannot be longer than 10 characters. The appointment ID shall not be null and shall not be updatable.

public void updateDescription(String description) {

   if (description == null) {

     throw new IllegalArgumentException(

"Appointment description cannot be null.");

   } else if (description.length() > APPOINTMENT\_DESCRIPTION\_LENGTH) {

     throw new IllegalArgumentException(

         "Appointment description cannot exceed " +

APPOINTMENT\_DESCRIPTION\_LENGTH + " characters.");

   } else {

     this.description = description;

   }

 }

//The appointment object shall have a required appointment Date field. The appointment Date field cannot be in the past. The appointment Date field shall not be null.

 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;

   }

 }

//Appointment Service Requirements

//The appointment service shall be able to add appointments with a unique appointment ID.

void setUp() {

   id = "1234567890";

   description = "The appt object shall have a required description.";

   date = new Date(2022, Calendar.JANUARY, 4 );

   tooLongDescription =

       "This description is too long for the appointment requirements but good for testing.";

   pastDate = new Date(0);

 }

 @Test

 void testNewAppointment() {

   AppointmentService service = new AppointmentService();

service.newAppointment();

assertNotNull(service.getAppointmentList().get(0).getAppointmentId());

assertNotNull(service.getAppointmentList().get(0).getAppointmentDate());

assertNotNull(service.getAppointmentList().get(0).getDescription());

service.newAppointment(date);

assertNotNull(service.getAppointmentList().get(1).getAppointmentId());

   assertEquals(date,

                service.getAppointmentList().get(1).getAppointmentDate());

assertNotNull(service.getAppointmentList().get(1).getDescription());

…

}

//The appointment service shall be able to delete appointments per appointment ID.

void setUp() {

   id = "1234567890";

   description = "The appt object shall have a required description.";

   date = new Date(2022, Calendar.JANUARY, 4 );

   tooLongDescription =

       "This description is too long for the appointment requirements but good for testing.";

   pastDate = new Date(0);

 }

 @Test

 void testNewAppointment() {

   AppointmentService service = new AppointmentService();

service.newAppointment();

assertNotNull(service.getAppointmentList().get(0).getAppointmentId());

assertNotNull(service.getAppointmentList().get(0).getAppointmentDate());

assertNotNull(service.getAppointmentList().get(0).getDescription());

service.newAppointment(date);0

assertNotNull(service.getAppointmentList().get(1).getAppointmentId());

   assertEquals(date,

service.getAppointmentList().get(1).getAppointmentDate());

assertNotNull(service.getAppointmentList().get(1).getDescription());

To achieve unit tests, all the components should be independently verifiable. It involves this pattern arrange, act and assert. If the behavior is consistent with the expectations, the test passes; otherwise, the test fails (Sohan et al., 2017). The state is checked as is, and the interactions are based on its interactions with other components' activities. This ensured that the tests were verifiable and easy to follow, as shown in the snippet below.

Secondly, the code was written to have 100 percent code coverage. As observed in the code snippet below, it covered all the situations which could be found to ensure interactions with other components. For example, we managed to test the getters and setters in the test class, which included the first name and last name, phone number, address, and all the situations where the above attributes can be null.

class ContactTest {

              //test cases for the following

              @Test

              void testContactNullArguments() {

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

                                              new Contact(null, null, null, null, null);

                              });

              }

              @Test

              void testContactAndGetters() {

                              Contact contact = new Contact("123456", "Lawrence", "Dominguez", "7025554900", "6029 Elm Street");

                              assertTrue(contact.getFullName().equals("Lawrence Dominguez"));

                              assertTrue(contact.getPhoneNumber().equals("7025554900"));

                              assertTrue(contact.getAddress().equals("6029 Elm Street"));

                              assertTrue(contact.getId().equals("123456"));

              }

              @Test

              void testSetFirstAndLastName() {

                              Contact contact = new Contact("123456", "Lawrence", "Dominguez", "7025554900", "6029 Elm Street");

                              contact.setFirstName("Lorenzo");

                              contact.setLastName("Day");

                              assertTrue(contact.getFullName().equals("Lorenzo Day"));

              }

              @Test

              void testSetPhoneNumberAndAddress() {

                              Contact contact = new Contact("123456", "Lawrence", "Dominguez", "7025554900", "6029 Elm Street");

                              contact.setPhoneNumber("7024595355");

                              contact.setAddress("1881 W Alexander Rd");

                              assertTrue(contact.getPhoneNumber().equals("7024595355"));

                              assertTrue(contact.getAddress().equals("1881 W Alexander Rd"));

              }

              @Test

              void testNullSetAttributes() {

                              Contact contact = new Contact("123456", "Lawrence", "Dominguez", "7025554900", "6029 Elm Street");

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

                                              contact.setFirstName(null);

                              });

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

                                              contact.setLastName(null);

                              });

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

                                              contact.setAddress(null);

                              });

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

                                              contact.setPhoneNumber(null);

                              });

              }

              @Test

              void testAllGetters() {

                              Contact contact = new Contact("123456", "Lawrence", "Dominguez", "7025554900", "6029 Elm Street");

                              assertTrue(contact.getFullName().equals("Lawrence Dominguez"));

                              assertTrue(contact.getId().equals("123456"));

                              assertTrue(contact.getPhoneNumber().equals("7025554900"));

                              assertTrue(contact.getAddress().equals("6029 Elm Street"));

              }

}

**Reflections**

Unit testing was done in this testing process using the white box mechanisms. Here, the attributes were known to provide a result that was known to the user. So, for example, the contact was first created then checked if the results produced by the test are the same as that which was the input.

Contact contact = new Contact("123456", "Lawrence", "Dominguez", "7025554900", "6029 Elm Street");

                              assertTrue(contact.getFullName().equals("Lawrence Dominguez"));

                              assertTrue(contact.getId().equals("123456"));

                              assertTrue(contact.getPhoneNumber().equals("7025554900"));

                              assertTrue(contact.getAddress().equals("6029 Elm Street"));

              }

Mockito could also be used to create mock objects but stubbing the object's behavior or method (Acharya, 2014). Mockito uses a black box testing mechanism where we check on the behavior of the method without using live data

Mockito is a behavior-driven approach to testing applications. It can be used when the application is coded to interfaces rather than the classes. Using mockito, we can add dummy functionalities and test the behavior, for example, when you don't want to directly work with the database.

As software testers, we need to ensure 100 percent code coverage to cover all situations on the code. Therefore, each code should be tested separately, independently, and collaborated. For example, we have the contactTest and ContactServiceTest classes. Although these classes work together, they must be tested independently to ensure that the application works correctly.

When using tools like CICD, like CirceCI, we can avoid code bias because the workflow will show tests that fail or not. Tools like SonarQube can also give the statistics of code

test quality.

All software developers must use [TDD.ie](http://tdd.ie/) test-driven development where all the tests are written before we code the application's functionality. Test-driven development will help reduce the technical debt accrued due to development indiscipline (Acharya, 2014)—for example, writing contactServiceTest before writing contactService logic.

**References**

Acharya, S. (2014). *Mastering unit testing using Mockito and JUnit*. Packt Publishing Ltd.

Sohan, S. M., Maurer, F., Anslow, C., & Robillard, M. P. (2017, October). A study of the effectiveness of usage examples in REST API documentation. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 53-61). IEEE. https://doi.org/10.1109/VLHCC.2017.8103450