For my Project One, I utilized appropriate tools and methods to implement a systematic approach to Junit testing for a Java-based application that serves to provide appointment service, contact service, and task service. More specifically, my deployment of Junit testing technique helped the application to verify each function and method on a per-block basis as well as to test the application in its entirety.

My testing approach mainly focuses on two areas: input authentication, and data synchronization. For input authentication, my testing approach was to satisfy all the requirements stated in the prompt and perform all possible tests to ensure the function or method can pick up errors in input and catch exceptions. One specific example is in the TaskServiceTest.java, taskName should not be null and should be less than 20 characters. My testing approach successfully enlisted possible inputs and their corresponding results:

  // This should return an IllegalArgumentException

    @Test

    public void testUpdateTaskNameFailure() {

        Task task = new Task("123456789", "Allan's Third Task", "This is a test on Allan's third task");

        taskService.addTask(task);

        taskService.updateTaskName("123456789", "Name Updated AND THIS EXCEEDS 20 CHARACTERS");

        Task updatedTask = taskService.getTask("123456789");

        Assertions.assertEquals("Name Updated AND THIS EXCEEDS 20 CHARACTERS", updatedTask.getName());

    }

For data synchronization, I made sure all required input to stay inside of one single constructor so that all input could be picked up and assessed based on their requirements and limitations:

@Test

    public void testTaskCreation() {

        Task task = new Task("1234567", "Allan's Task", "This is a test on Allan's task");

        Assertions.assertEquals("1234567", task.getTaskId());

        Assertions.assertEquals("Allan's Task", task.getName());

        Assertions.assertEquals("This is a test on Allan's task", task.getDescription());

    }

By successfully achieving the two goals, my testing approach was thorough and effective because it not only included all the necessary input data but also tested all the potential violations of the input limitations. When a failed test was expected, a proper exception stating the reason of failure would be outputted to the console. To further elaborate, if this Java-based application were to transform into a web-based application, then instead of user receiving 500 status code, they will receive 400 instead because my testing approach successfully threw the proper exceptions.

My overall project source code and Junit testing approach were technically sound because they both addressed input requirement and limitations. For example, the taskID cannot be an empty value, null, or longer than 10 characters. I specified all of these requirements in my source code to address such limitations:

public Task(String taskId, String name, String description) {

        if (taskId == "" || taskId == null || taskId.length() > 10){

            throw new IllegalArgumentException("Max ID length: 10 characters and CANNOT BE NULL");

        }

        this.taskId = taskId;

        this.name = name;

        this.description = description;

    }

On the other hand, my source code and testing approach were efficient as well because I only included the necessary getters and setters methods in my source code and eliminated the unnecessary ones so that the code package stayed lean in size without any redundancies.

// setters to be used in TaskService... otherwise no visibility

    public String getTaskId() {

        return taskId;

    }

    // setTaskId is not used because "the task ID shall not be updatable"

    public String getName() {

        return name;

    }

One software testing technique I have used throughout the three milestones is unit testing. By incorporating specific annotations from JUnit Jupiter, I was able to perform tests for each method and see which ones would pass and which ones would fail. I added specific tests that would intentionally trigger exceptions to ensure the code passes positive and negative tests.

    @Test

    void failedTestOversizedDescription(){

        appointmentService = new AppointmentService();

        appointment = new Appointment("1234567890", new Date(), "This is an appointment but this is longer than the maximum of 50 characters so the test will return an error message");

        appointmentService.addAppointment(appointment);

    }

    @Test

    void testAddAppointment() {

        Appointment newAppointment = new Appointment("123456789", new Date(), "Test adding new appointment");

        appointmentService.addAppointment(newAppointment);

        Assertions.assertEquals(2, appointmentService.getAppointments().size());

Another software testing technique I used was UAT *(User Acceptance Testing)*. In this case, I played around with the parameter limitations to see if any specific exception could be triggered. For example, when making an appointment, the length of appointmentId should be at most 10 characters. If I test with 11 characters then the test should fail.

@Test

    void failedTestOversizedId(){

        appointmentService = new AppointmentService();

        appointment = new Appointment("12345678901", new Date(), "This is an appointment");

        appointmentService.addAppointment(appointment);

    }

Some techniques that I did not use in my milestones are automated testing and regression testing. Automated testing is a technique that automatically tests the functionality of the source code once built. Jenkins pipeline with a linked GitHub webhook might be a good choice in this case to achieve automated testing. Regression testing is very similar to automated testing that as the code changes, the tests are carried out. Since my overall source code is not very complicated and there is no real-world traffic coming in for challenge, regression testing would not be a desired technique to be implemented.

There were certain areas that I practiced caution during this project development. The most important one was testing without actual credentials. Since this project was only done locally, usage of my basic personal info could be accepted, but based on the privacy laws *(US DOE)*, I would need to be more cautious not only when sharing my personal info but also when trying to access other’s information in future projects.

Bias would probably occur in my code since it was all developed by myself, but I have tried to make my code as inclusive as possible to consider all possible events from happening. For example, one flaw in my source code was that I prevented the appointmentID to be less than 10 characters and not null, but I did not check whether it was blank or included space characters. Being more inclusive and minimizing bias would be something I will work on in my future project.

Some people might say that simpler code causes less trouble, but I would say that if a software engineer keeps writing simple code, he/she might not be able to grow at all. Meanwhile, there are sample source code online for almost everything we might need for our future project. My take is that we can feel free to copy and paste the source code, but at the meantime we also need to make sure we can fully understand the code so that it transforms into our own knowledge that we can use in the future.

Reference:

User Acceptance Testing *| University IT*. (n.d.). Uit.stanford.edu. <https://uit.stanford.edu/pmo/UAT>

U.S. Department of Education. (2018). Protecting student privacy | *U.S. department of education.* Ed.gov. <https://studentprivacy.ed.gov/>