msihapanya CS-320 Project Two

When I first developed the Contact and Task services, my approach was mostly aligned to the software requirements, but lacked meeting a couple. I continuously referenced the requirements as I developed the code. I first worked on the Contact and Task classes, ensuring that there were variables for each of the requirements, and made if statements that threw exceptions if the constructed variables were invalid. I also based their Service classes strictly to the requirements, creating methods for each of the functions – adding, updating, and deleting objects. Here is the basis for my Contact class, which my Task class mirrored:

public Contact(String id, String firstName, String lastName, String phoneNumber, String address) {

if (id == null || id.length()>10) {

throw new IllegalArgumentException("Invalid ID");

}

if (firstName == null || firstName.length()>10) {

throw new IllegalArgumentException("Invalid firstName");

}

if (lastName == null || lastName.length()>10) {

throw new IllegalArgumentException("Invalid lastName");

}

if (phoneNumber == null || phoneNumber.length()!=10) {

throw new IllegalArgumentException("Invalid phoneNumber");

}

if (address == null || address.length()>30) {

throw new IllegalArgumentException("Invalid address");

}

So, for meeting the most basic requirements, my testing approach was aligned. However, like mentioned before, I missed the mark on a few requirements. For the Contact and Task classes, my approach failed to ensure that each ID was unique. I also failed to fully account for the length requirements of the variables. I focused solely on the aspect of String fields not being longer than a certain amount of characters. In my view, I thought I only had to write unit tests for if the length exceeded the threshold. I then learned that I must write unit tests for the condition of less than and equal to along with more than. This way, the requirement is fully tested, and accounts for all possibilities of constructor input. I applied the growth from my mistakes to the Appointment feature and fully implemented all the requirements. In my final code for these services in Project One, my approach became fully aligned with the software requirements. In each of the services, I made sure to test for the uniqueness of each ID by creating an array and keeping track of all IDs constructed:

if (*allIds*.contains(id)) {

throw new IllegalArgumentException("ID exists. Please use a different ID");

}

*allIds*.add(id);

*@Test*

void testTaskIdUnique() {

Task task1 = new Task("1000000000", "name1", "description1");

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

new Task("1000000000", "name2", "description2");

});

}

This test throws an exception if a newly constructed object matches the ID of an already constructed object.

For each of the String objects, I made sure to write tests for all conditions of less than, equal to, and more than.

*@Test*

void testTaskNameTooLong() {

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

Task task = new Task("111111113", "Sharahu123456789101112", "This is a task description");

});

}

*@Test*

void testTaskNameIs20() {

Task task = new Task("2111111118", "Sharahu1237891011123", "Task description");

*assertEquals*(20, task.getName().length(), "Name length is 20");

// less

}

*@Test*

void testTaskNameLessThan20() {

Task task = new Task("2111111114", "Sharahu", "A Task description");

*assertTrue*(task.getName().length() < 20, "Name less than 20 characters.");

}

In developing Project One, my approach was completely aligned to all software requirements.

The overall quality of my JUnit tests is robust and comprehensive. I wrote my tests in isolation, which led to substantial coverage and thus higher quality tests. For each length requirement of all String variables, I could have combined all length requirements into one test case. However, by separating each requirement for each variable into their own test cases I was able to ensure that any failure of one test did not impact the other. I also isolated the getters and setters of each variable for all classes. For the length tests, I also covered boundary conditions, which is good practice when dealing with thresholds in variables. Based on the coverage percentage of my tests, which all passed higher than 80%, my JUnit tests were highly effective. Each service class test hit 100%, meaning I tested for all methods appropriately for the objects. The base classes ensure that the code runs, but the test classes all implement assertions, which ensure that all my code asserts the expected behavior of the software.

For example, in my AppointmentTest.java, I tested for if the appointment was in the past. If the appointment is in the past, an exception is thrown. The test matches when that exception is thrown in Appointment.java.

*@Test*

void testAppointDateInPast() {

String id = "19";

Date currentDate = new Date();

long pastMilliseconds = currentDate.getTime() - 999999999;

Date pastDate = new Date(pastMilliseconds);

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

Appointment appt = new Appointment(id, pastDate, "This is an appointment description.");

});

}

if (appointmentDate.before(new Date())) {

throw new IllegalArgumentException("Appointment date is in the past");

}

Like the previous milestones, I employed black-box and white-box testing techniques in the project. Boundary value analysis and use case testing proved to be the most useful out of black-box testing techniques. Boundary value analysis involves testing for values outside the boundaries of a certain threshold. In other words, it’s testing for values below, within, and above a specific boundary. It is not possible to ensure any condition by only testing one value, so boundary value analysis is often necessary. Use case testing involves writing tests based on defined use cases of an application. For example, the TestService class required operations like add task, update task, and delete task. The application is the actor enacting these use cases. I wrote test cases that directly attributed to these three methods. Once the base classes were all developed, I used white-box testing techniques. The first technique I used, statement testing, is used on the executable statements in the code. When data is input, certain statements are forced to execute. Statement testing is sure to be done when tests are run as coverage. Code in the base classes will be highlighted in green or red. Statements highlighted in green were thus tested. So statement testing constructs tests based on this process; it is a sort of trial and error. Decision testing focuses on ensuring exceptions are thrown when certain decision branches are reached. All constructors in this project involved decision branching. In the Contact.java class, an illegal argument exception is thrown when the length of the first name is either null or longer than ten characters.

if (firstName == null || firstName.length()>10) {

throw new IllegalArgumentException("Invalid firstName");

}

Black-box testing techniques are practical for when the base requirements of the project are known, but the inner code has not yet been developed or revealed. White-box testing then becomes practical when the code is already developed, and tests can be created through retroactive inspection of the code. Testing techniques should not be used in isolation and only in isolation for projects, but rather used in tandem with each other to develop thorough depth and breadth in any program.

The other kind of testing technique I did not use for this project is experience-based testing, as I have no prior experience of professional software development. I have not been taught proper testing protocols, so I did not have much experience to bring to the table. Experience-based testing is testing that comes from a tester’s past experiences. This becomes handy when time or documentation is a limitation. A popular form of experience-based testing is exploratory testing, where test objectives are constructed and focused on throughout the testing process. Testers can structurally construct tests based on their previous experience. Another form of experience-based testing is checklist-based testing. This involves creating high-level checklists from experience and outside sources. When combined with other testing techniques, experience-based testing becomes especially powerful. A continuous feedback loop can be started with experience-based testing, which instills project refinement. This is highly beneficial for real-world projects.

Iterating on the JUnit test cases for each class helped me know my code was efficient and technically sound. As I wrote my first draft of code for the tests, I saw that several lines and methods were highlighted in red, and thus not being covered. So, I changed my approach and ensured that my test code covered the base class code, and for the text to highlight in green. This way, my test code was covering for all of the base class code as much as possible. My key example is the Task.java and TaskTest.java classes. In Milestone 4, I had no tests for the Task class’ getters. As I continued on with the milestones and then finished Project One, I realized that each base class needs tests for its getters. So, in TaskTest.java, I implemented test cases for my getters:

*@Test*

void testGetTaskId() {

String id = "20";

Task task = new Task(id, "Sharahu", "This is a task description.");

String returnedId = task.getId();

*assertEquals*(id, returnedId);

}

*@Test*

void testGetTaskName() {

String name = "Sharahu";

Task task = new Task("21", name, "This is a task description.");

String returnedName = task.getName();

*assertEquals*(name, returnedName);

}

*@Test*

void testGetTaskDescription() {

String description = "This is a task description.";

Task task = new Task("22", "Sharahu", description);

String returnedDescription = task.getDescription();

*assertEquals*(description, returnedDescription);

}

Continuous iteration and increasing exposure to the code and its tests allowed me to sharpen the efficiency and technical soundness of my code.

As this is my first time acting as a software tester, caution was not necessarily the first thing on my mind, especially for the initial milestone. My approach was to get all the requirements for the base classes down, and then test for as many cases as I could think of. But as I developed more into the next milestones, I began to realize caution. My coverage was not as high as I would have liked it to be. I did not have much caution in the Module 3 milestone. My constructor test method was more careless than my later iterations. I exercised more caution in correcting this in Project One by changing my assertTrue assertions to assertEquals. AssertEquals offers a more direct comparison of the object’s constructed parameters with the attributes retrieved from the getters.

This was my previous method of testing Contact’s constructor:

*@Test*

void testContact() {

Contact contact = new Contact("1", "Kafka", "Hime", "8888888888", "1234 Star Rail Drive");

*assertTrue*(contact.getId().equals("1"));

*assertTrue*(contact.getFirstName().equals("Kafka"));

*assertTrue*(contact.getLastName().equals("Hime"));

*assertTrue*(contact.getPhoneNumber().equals("8888888888"));

*assertTrue*(contact.getAddress().equals("1234 Star Rail Drive"));

}

And here is my final test for Contact’s constructor:

*@Test*

void testContact() {

String id = "222";

String firstName = "Kafka";

String lastName = "Hime";

String phoneNumber = "8888888888";

String address = "1234 Star Rail Drive";

Contact contact = new Contact(id, firstName, lastName, phoneNumber, address);

*assertEquals*(id, contact.getId());

*assertEquals*(firstName, contact.getFirstName());

*assertEquals*(lastName, contact.getLastName());

*assertEquals*(phoneNumber, contact.getPhoneNumber());

*assertEquals*(address, contact.getAddress());

}

All in all, I exercised increasing caution the more I iterated on the milestones and then finally when I completed Project One. It is extremely important to appreciate the complexity and interrelationships of the code one is testing so there is a full understanding of the software at hand. If tests are done in isolation of each other, a holistic view of the software cannot be achieved, and scenarios will stay untested for, opening doors to software errors.

I was very biased starting out this project. Since the milestones were the basis of the project, the project really started early in the course. But as a novice developer, I brought in certain coding expectations when approaching the assignments. When reading the requirement “The contact object shall have a required unique contact ID String that cannot be longer than 10 characters,” I believed I only had to test for one direction: longer than 10 characters. Receiving feedback from the professor helped me realize my bias and shift my perspective. That also made me realize the importance of shifting our biases when we develop applications. We often work with other people when developing applications, and that allows us to expand our perspectives and limit bias when we develop and test code. This occurred when the requirements were already laid out for me. In the real world, we often come up with our own requirements before we write the code. If I was to be the only one testing the code I wrote with requirements I came up with, the analyses from my testing would be severely limited. We only see the world from our own perspective, so it is important to have others test the code we write to ensure all gaps are filled. Bias is definitely a concern when testing code, so a diverse team, or just a second eye, helps mitigate bias.

It's important not to cut corners so we can reduce the number of problems faced in the future. There are problems we can avoid if we take great care in designing and testing the code prior to releasing it to production. Touching back on my earlier thoughts about bias and limiting it with diverse perspectives, I think peer code reviews are a great way to stay disciplined in the process of developing high quality code. Not only does it build strong rapport across the team (which is greatly beneficial to any project in the long run), but it allows for many possibilities to be considered during the development process. Peer code reviews catch flaws and also ways to make code more efficient. To avoid technical debt as a practitioner in the field, I will be diligent in my own work, but frequently seek educated second opinions. A continuous feedback loop throughout the team will keep everyone on track, and reduce the overall technical debt instilled by the team.