Austin Scarinza

Software Test, Automation QA

June 20, 2024

**Summary**

Before I began planning and writing out each of the three features, I made sure to read through and fully understand the given requirements. This led me to follow a more structured development approach that better aligned with the requirements opposed to going back and changing my own approach to meet the requirements. This was my tactic for writing all three features, the ContactService, TaskService, and AppointmentService. Each set of requirements were very clear which made creating test cases also very straight forward.

All three features shared a similar structure. Each feature was composed of an object class, and a service class that was used to manipulate and store the objects. The requirements for the object class were quite simple, each attribute stored in the object was required, could not be null, and could not be longer than a certain length of characters. This was the case for the contactID, firstName, lastName, phone, and address variables for the Contact class. The Task and Appointment objects shared a similar attribute structure. To ensure that these attributes met the requirements, I used if statements to check the criteria, and throw exceptions when they failed, I then incorporated tests to see if valid inputs would pass, and bad inputs would trigger the exception throw. Once all attributes were verified, the constructor would then create a new object with the valid information. For example, verifying that the contactId was no longer than 10 characters and was not null was written, and tested using,

        //Check for valid contactId

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

            throw new IllegalArgumentException("Invalid contact ID");

        }

    @Test

    void testContactIdTooLong() {

        Assertions.assertThrows(IllegalArgumentException.class,  () -> {new Contact("1234567891011", "Austin", "Scarinza", "1234567891", "12 Fake Road");

    });

    }

    @Test

    void testContactIdNull() {

        Assertions.assertThrows(IllegalArgumentException.class,  () -> {new Contact(null, "Austin", "Scarinza", "1234567891", "12 Fake Road");

    });

    }

This type of testing was used to test each attribute when creating a new object, and later in the service classes when updating an existing object.

The three common requirements for the ContactSerice, TaskService, and AppointmentService was to add, delete, and update the respective object from an array list of objects. In doing so, the requirements stated that only unique Ids could be added, no duplicates. Some other requirements were not directly stated, but I added to ensure functionality such as deleting a non-existent Id and updating non-conforming variable inputs. Similar tests were used in updating variables as before in the constructor.

    //Used to update an existing contact

    public boolean updateContact(Contact contact, String firstName, String lastName, String phone, String address) {

        if (firstName != null && !firstName.isEmpty() && firstName.length() <= 10) {

            contact.setFirstName(firstName);

        }

    @Test

    void TestUpdateFirstName() {

        Contact contact = new Contact("12345", "Austin", "Scarinza", "1234567891", "12 Fake Road");

        contactService.addContact(contact);

        assertTrue(contactService.updateContact(contact, "Bob", "Scarinza", "1234567891", "12 Fake Road"));

        assertEquals("Bob", contactService.getContact(contact.getContactId()).getFirstName());

    }

Throughout testing, I made sure that each requirement was met with tests from all directions, a test with bad input to ensure the provided exceptions were thrown, and tests with good input to ensure the tests passed and functioned as planned. In doing so, I was able to reach an overall test coverage of 96.1% for all the provided object, and service features.

Through the process of manually debugging, recognizing patterns, and using the built in tools, I was able to create a technically sounds application. The code I created was modularly created for simple integration, testing, and future improvements. This also led to fast processing times by eliminating redundant code and inefficient practices. The features were all well commented to allow others to read and understand the functionality. Using simple and concise code made finding logical and syntax errors quicker as seen in the addTask and removeTask functions of the TaskService feature.

    //Adds task to contact list if it passes all the constructor requirements

    public boolean addTask (Task task) {

        if (!taskList.isEmpty()) {

            for (Task t : taskList) {

                if (t.getTaskId().equals(task.getTaskId())) {

                    return false; //This will ensure only unique taskIds are allowed to be added

                }

            }

        }

        //If taskId does not exist it will be allowed to be entered and we pass true back to the Contact constructor

        taskList.add(task);

        return true;

    }

    //Removes the given task via taskId if it exists

    public boolean removeTask(String taskId) {

        for (Task t : taskList) {

            if (t.getTaskId().equals(taskId)) {

                taskList.remove(t);

                return true;

            }

        }

**Reflection**

For each feature, Black-Box was used to guide the general testing. Under the large Black-box umbrella of requirement-based testing, I used manual, functional, non-functional, unit, integration, and acceptance testing. Manual testing is done in almost all testing environments, and these features were no different. I manually searched the codebase to see if I could find any issues, before and after integration. This led to the later, functional tests, to show less errors due to simple typos. The functional testing took the most time and led to the greatest overall increase in test coverage percentage. The functional testing conducted ensured that the code written met the requirements and gave feedback to prove it. I used functional testing alongside acceptance testing to ensure that what I had written was going to meet the client’s expectations. In writing each feature, and the application, the code was broken up into units such as the Contact and ContactService classes. Testing each unit individuality allowed me to quickly locate issues before integrating them together. I then also used integration testing when testing the contact Service and combing them. While non-functional testing was limited in these milestones, ensuring performance and usability was still a factor in ensuring the code met requirements.

Some testing strategies that I did not use in these features would include automated, regression, security, and user acceptance testing. These are all very important testing strategies that are by no means not useful, just not needed in these milestones. Given the size and complexity of the three milestones, there was no need to incorporate automated testing as test cases could be quickly written and changed. There was not much code altering done which led to the absence of regression testing. I also did not use any sort of security testing as nothing was getting deployed and the only person using the code this far was myself. I also have no end user which makes incorporating user acceptance testing difficult and there is no end user input.

When it comes to any software application, using a Black-Box approach and the testing strategies outlined will be beneficial in ensuring the applications meet the client’s requirements. All the tests I did conduct could, and should, be used in testing other types of software as they ensure the written code is truly targeting the client’s problem. Creating manual code reviews, functional test cases, and acceptance tests to prove that what has been written is valid is a needed step. When delivering software to a client, the team needs to ensure that it is secure, which can be done by using automated tests and regression tests when large areas of the code is changed. Black-Box is not the only answer, however. Other strategies such as White-Box testing that ensure proper internal structures are just as important and validating the output when creating an all-around sound application. As testers become more familiar with testing techniques and applications, they will know which tests work best and when to apply them.

As I became more familiar with the application I was creating, I began to pick up on areas that needed more testing, and how those tests should be written. I was cautious throughout the whole process, but the familiarity and patterns made creating and executing tests much quicker. I began to feel very comfortable with the patterns towards the end and felt as though no testing was even needed as the services were all so similar and the previous ones worked flawlessly. I had to put my own personal bias of how my code was written to the side and test it as though someone else had written it, testing everything I would in any other case. On the other hand, I felt as though I needed to test it thoroughly to the best of my ability to ensure that the product I was submitting was as perfect as possible. I am someone who likes to take accountability and be proud of my work, so turning int an unpolished version is not an option. Cutting corners in the development or testing processes only leads to costly and more complicated fixes later when someone inevitably goes astray.

The more thorough the testing, the more likely it is that issues can be found and resolved early on eliminating wasted time on developing an issue prone feature. The quicker a problem is found, the cheaper, quicker, and easier it is to fix. Having a structured manual review process and conducting peer reviews are some examples of steps one can take to ensure that the product being delivered or integrated into a large project is sound and free of as many errors and possible.