Eyoel Tesfu

CS-320 Software Test Automation& QA

SNHU

Module Seven Project Two

Summary

The degree of alignment between the software requirements and my unit tests were closely aligned. The degree of this alignment significantly affects the quality of the software that is being developed. Firstly, I designed each unit test so that they verify that the individual components of the software are functioning as expected pertaining the system requirements.

For example,

@Test  
void testAddingDuplicateAppointments() {  
 String id = "32786423";  
 Date date = new GregorianCalendar(2026, Calendar.*JUNE*, 16).getTime();  
 String description = "Doctor's appointment";  
  
 appointmentService.addAppointment(id, date, description);  
 *assertThrows*(IllegalArgumentException.class, () -> appointmentService.addAppointment(id, date, description));  
}

The above test verifies whether adding a duplicate appointment into the appointment list instantiates an Illegal Argument Exception or not.

Each requirement specified in the system requirements was broken down into individual components that can be tested. Additionally, I ensured that every requirement was covered by the tests. This leaves no functionality unchecked.

Test coverage is a useful metric that helps us understand the degree to which the source code was tested. While I had written out tests for a line coverage of close to a 100%, it doesn’t signify the quality or the effectiveness of our tests. To improve the effectiveness of my tests, I designed them in a way that they effectively handle edge cases, and test for failure as well as success.

For example,

@Test  
void checkSuccessfulDeletionOfAnAppointment() {  
 String id = "32786423";  
 Date date = new GregorianCalendar(2026, Calendar.*JUNE*, 16).getTime();  
 String description = "Doctor's appointment";  
  
 appointmentService.addAppointment(id, date, description);  
 ArrayList<Appointment> appointments = appointmentService.getAppointments();  
 appointmentService.deleteAppointment(id);  
  
 *assertTrue*(appointments.isEmpty());  
}  
  
@Test  
void testDeleteingNonexistentAppointmentBringingException() {  
 String id = "1234567890";  
 *assertThrows*(IllegalArgumentException.class, () -> appointmentService.deleteAppointment(id));  
}

The above two tests handle a successful deletion if an appointment as well as a failed attempt of deletion of an appointment.

Another example,

@Test  
void checkCreationOfSuccessfulAppointmentInstance(){  
 String id = "3278642351";  
 Date date = new GregorianCalendar(2026, Calendar.*JUNE*, 29).getTime();  
 String description = "Doctor's appointment";  
  
 Appointment appointment = new Appointment(id, date, description);  
  
 *assertEquals*(id, appointment.getId());  
 *assertEquals*(date, appointment.getDate());  
 *assertEquals*(description, appointment.getDescription());  
}

@Test  
void checkInvalidIdOfMoreCharacters() {  
 String id = "12345678901"; // More than 10 characters  
 *assertThrows*(IllegalArgumentException.class, () -> new Appointment(id, new Date(), "Testing an appointment"));  
}

The first test checks for an id with 10 characters and the second one checks for more than 10 characters. These two tests verify that the edge cases function as expected.

I ensured that a piece of code is technically sound by effective unit testing (like the examples mentioned above), code reviews from my instructor, and adhering to team specific coding standards.

I ensured that my tests were efficient by using efficient and targeted assertions in small, quick and isolated unit tests.

Reflection on Testing Techniques

Some of the specific unit testing techniques I used in the milestones include the following:

* I implemented boundary testing. This technique focuses on testing the boundaries and limits of inputs and ensures that they behave correctly in these edge cases. For example, for a string that is supposed to have less than 10 characters, I would test the behavior of a string with 9, 10, and lastly 11 characters to exhibit boundary testing.
* I used Exception testing. This verifies that the component that is being tested throws an Exception especially when an illegal input is trying to modify component behavior beyond the requirements.
* Some other techniques I used include positive and negative testing. The positive testing technique focuses on testing the expected and valid behavior of a component. For example, the addAppointment, addTask, and addContact methods were tested by adding appointments with unique IDs and verifying that they were successfully added to the list. The negative testing technique tests the handling of invalid and unexpected inputs. I used this testing technique on scenarios that attempted to add a duplicate appointment/task/contact, and when deleting a non-existing contact/appointment/task.
* I also used a BeforeEach setup using the @BeforeEach annotation to set up the initial state before each of the rest of the test cases were executed. For example, in AppointmentServiceTest, I initialized a new instance of the AppointmentService class to ensure that each test starts with a clean state.
* I also used assertions like ‘assertEquals’ and ‘assertTrue’ to check and verify the expected results.

The other software testing techniques I didn’t use in the three previous milestones include the following:

* Performance testing, where it is used to test the software when it is under heavier loads. It helps in ensuring that the software can handle an unexpected wave of user traffic. This technique is very important for systems that support many users simultaneously like web servers, multiplayer online games, large scale transaction systems, etc.
* Stress testing, where it is used to test the software under extreme loads in order to identify the software’s limit or failure point. This technique also helps when trying to modify the software to fail gracefully when it hits its limits. It is useful in systems that handle large amounts of data or requests, such as social media apps, banking apps, etc.
* Security testing, where it is used to ensure that the software cannot be compromised. It reveals the vulnerabilities of a program or a system. It is vital in any kind of software but is a primary concern for systems that harbor sensitive user information such as banking apps and healthcare systems.
* Integration testing, where it is used to test for integration defects between different and combined modules and/or functions. This testing technique is important in software that is developed by multiple developers on its different components, such as microservices.
* Usability testing, where it is used to test how user-friendly the system is. It is a primary driver and concern when developing projects that depend highly on user experience (such as Adobe Photoshop for example).
* Compatibility testing, where it is used to check whether the program can successfully run on different platforms. This is important in software that is meant to run on a multitude of platforms; products like mobile apps and websites.

Reflection on Mindset

When software testing, it is important to adopt a meticulous mindset. We have to keep an eye out for bugs and coding errors. Additionally, testing should be about a cautious understanding of the effects of those bugs on the overall system. All scenarios should be thoughtfully considered and tested.

Software systems are composed of numerous components interacting with each other. A system of interacting components can be bug free individually but can have problems during interaction. Understanding these complexities and interaction relationships is important to identify problems that we may encounter during testing. This understanding can guide testers in the creation of test cases that can handle edge cases and various testing complexities. Having a cautious approach combined with an understanding of software complexity, therefore as I understand, is vital for successful software testing.

During development, I attempted to limit my bias by testing a variety of scenarios in the source code, including edge and corner cases. I would do this by listing all the criteria that span all cases for a component. For example, if I were to test a function that divides two numbers, I won’t just test it with positive numbers; I would also test it with zeros, negative numbers, and null values to ensure that it handles all these cases correctly.

Discipline in commitment to quality is important because it ensures high quality software that plays a critical role in widespread systems such as healthcare systems that could potentially be lifesaving, banking systems that hold sensitive information, and space programs where a tiny mistake can result in a multi-million-dollar failure.

Cutting corners in writing or testing code will often lead to substantial cost accruement in the future (technical debt). For example, if I skip writing unit tests for a particular function to save time, and a bug in that function surfaces later in development, the cost of fixing the bug, deploying the fix, and potential damage to the company's reputation could potentially far exceed the time saved by not writing the tests initially. I would try to avoid such technical debt by implementing automated testing methods, by regularly refactoring and restructuring the code for it to be better, and by providing relevant documentation on the codebase.

References

García, B. C., Rodríguez, L. E., & Yagüe, A. (2016). Managing software process measurement: A metamodel-based approach. Journal of Systems and Software, 115, 143-162. <https://doi.org/10.1016/j.jss.2016.02.014>

Stamelos, I., Angelis, L., Oikonomou, A., & Bleris, G. L. (2002). Code quality analysis in open-source software development. Information Systems Journal, 12(1), 43-60. <https://www.researchgate.net/publication/220356602_Code_quality_analysis_in_open_source_software_development>

Visaggio, G., Baldassarre, M. T., & Lavazza, L. (2010). A process metrics set for measuring defects. Software Process: Improvement and Practice, 15(3), 225-242. <https://doi.org/10.1002/spip.448>