Courtney Thim

Project 2

The unit testing approach I used for each of the three features was that of a white-box testing approach. My strategy was completely aligned with the software requirements as I gave meeting these requirements the top priority. I used JUnit tests to ensure that each requirement was satisfied and that the boundaries were tested for each specification. For example, the specification of the Task says that the name is required, cannot be null, and must be a maximum of 20 characters in length. To ensure these requirements were met, a function with three branches was created to throw exceptions if the task name did not meet the specifications, as shown below.

A computer code with text

Description automatically generated

As you can see, each branch handles a situation where one of the requirements is not met. JUnit tests were then created to test whether these branches worked properly. I created tests that tested a valid task name, a null task name, and a task name that was too long. Both tests that tested task names that did not meet the requirements threw assertions like the example below:

@Test

@DisplayName("Task name is too long")

**void** testTaskNameTooLong() {

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

**new** Task("T10010", "Test Software 1111111", "Test software requirements.");

});

}

Functions and tests like the ones above were created for all the other requirements in the Task class, such as the requirements for the task ID and description. The same process was repeated for the Contact and Appointment classes as well.

A very similar process was repeated for the Task Service, Contact Service, and Appointment Service classes. These classes were focused on the functionality of the system and produced their own set of requirements. Again, individual functions were designed to meet the requirements for each of these classes. JUnit tests were then designed to test that the functions worked properly. For example, the Contact Service class was meant to enable the addition of a contact to a list. The addContact function accomplished this, while multiple JUnit tests ensured the function actually added a contact. The tests were designed to test different situations that could possibly occur that might cause the function to behave strangely, like adding one contact, adding multiple contacts, or adding a duplicate contact. A similar process was repeated for all other class requirements for each of the three classes.

I’m confident my JUnit tests were effective because the test coverage for each of the six classes individually exceeded 80%. Although there is no standard for the exact percentage that makes tests effective, a common number is 80% (Heusser, 2021). Because I exceeded 80% in every case, oftentimes achieving 100% test coverage, I believe I created effective JUnit tests. I did not pay much attention to the test coverage percentage for the JUnit test classes because their only purpose was to test the actual system classes. The JUnit tests did not contribute to the functionality of the software, and without them, the software would still function normally. As you can see in the screenshots below, the test coverage for the Contact class was 100%, while the ContactService class had 98.8% coverage.

A screenshot of a computer

Description automatically generated

I was able to achieve 100% coverage for the Appointment class and 98.4% coverage for the Appointment Service class, as shown below.

A screenshot of a computer

Description automatically generated

Finally, I achieved 100% test coverage for the Task class and 98.8% coverage for the Task Service class.

A screenshot of a computer

Description automatically generated

To ensure that my code was technically sound, I tried to test “to the spirit of the code.” For example, the Contact Service was to be able to delete contact(s) from a contacts list. There were various scenarios that could occur when attempting to accomplish this that might cause the system to behave strangely. The user could attempt to delete a contact that has a null ID, attempt to delete a contact from an empty contacts list, or attempt to delete a contact that doesn’t exist. To ensure that the system knew what to do in these scenarios, assertions were used to throw flags as follows:

A screenshot of a computer code

Description automatically generated

Additionally, line 97 double-checks that the contact was successfully deleted. This was done for each of the other classes as well. An additional thing I could have done was check the size of the contacts list before and after deletion. This would have increased the technical soundness of the system.

To ensure my code was efficient, I used clear and meaningful variable names, like taskId, taskName, taskDescription, contactFirstName, contactLastName, etc. Using meaningful variable names makes it easier for others to understand the code. I also tried to write simple and concise code and not over-complicate methods. For example, the addTask method only utilizes one for loop and two if-branches. I tried to use the same simplicity when writing the unit tests as well. I was able to keep them simple and focused by designating only one test condition for each JUnit test. For example, the Contact Service was to be able to update a contact's first name, last name, phone number, or address. Each of those had their own individual unit test. Here are two of those tests as an example.

@Test

@DisplayName("Test updating a contact - change address")

**void** testUpdatingContactAddress() {

ContactService newContact = **new** ContactService();

Contact newContact1 = **new** Contact("C10005", "Dave", "Manning", "2225550940", "16 Chestnut Ave, Grey, CA");

newContact.addContact(newContact1);

newContact.updateContact("C10005", "Dave", "Manning", "2225550940", "17 Chestnut Ave, Grey, CA");

*assertEquals*(newContact1.getAddress(), "17 Chestnut Ave, Grey, CA");

}

@Test

@DisplayName("Test updating a contact - change phone")

**void** testUpdatingContactPhone() {

ContactService newContact = **new** ContactService();

Contact newContact1 = **new** Contact("C10005", "Dave", "Manning", "2225550940", "16 Chestnut Ave, Grey, CA");

newContact.addContact(newContact1);

newContact.updateContact("C10005", "Dave", "Manning", "6234518473", "16 Chestnut Ave, Grey, CA");

*assertEquals*(newContact1.getPhone(), "6234518473");

}

As I stated earlier, I used white-box testing techniques for this system. White-box testing tests the internal structure and logic of an application. I employed unit and static testing techniques to ensure the applications ran correctly. Static testing involves examining the code and comparing it to the specifications to ensure all requirements are met. It also involves studying the code to identify errors. I used static testing frequently to determine why some of my JUnit tests failed. Unit testing is a type of testing “that focuses on individual units or components of a software system” (GeeksforGeeks, 2023). The purpose of unit testing is to ensure that each individual “unit” works properly and meets its individual requirements before integrating it into the rest of the system. I utilized the JUnit framework to help perform the unit tests. JUnit testing was used to break down the code into small sections focused on one specific functionality, feature, or specification. Dividing the code in this way allowed me to pinpoint exactly where errors were occurring in the program.

Several software testing techniques that were not used for the project include system testing, acceptance testing, integration testing, and regression testing. System testing involves verifying a software’s behavior and functionality as a whole system. It’s typically conducted after integration testing and is intended to identify errors within the system and ensure all requirements are met. Integration testing was also not incorporated, as its purpose is to verify the interfaces and interactions between components or modules. The next step in the process would probably be system/integration testing to see how the Contact, Task, and Appointment classes worked together as one application. Acceptance testing is a technique used to ensure the system is working correctly for the user and meets the user’s needs. It’s similar to unit testing, except it generally has a binary result, pass or fail (Doyle et al., 2023). Users of the system would typically test out the system themselves and then provide feedback. Regression testing occurs when an update is made to an existing system to ensure that the update/new feature or functionality did not cause errors elsewhere in the code. This testing technique did not need to be used as this was the initial creation of the system. Finally, no type of non-functional testing was performed, like security dependencies, performance, usability, reliability, etc. Security testing is always an important part of any project to ensure that the software isn’t vulnerable to attack and that data is secure/uncompromised. Tools like OWASP Dependency Check could be used to help perform security testing.

There are many different applications and practical implications for each of the different software techniques. Unit testing allows developers to understand what functionality is provided by a unit and how to use it to gain an understanding of the unit API (GeeksforGeeks, 2023). It also enables some parts of the software to be tested before others so that you don’t have to wait until the entire software is complete. The main advantage of unit testing is that it can uncover bugs and errors early on in the project. Static testing is the process of reviewing the code without executing it. This is done through things like peer reviews, code inspections, and walkthroughs. Like unit testing, the main goal is to identify bugs and errors early. Integration testing is the process of testing the interface between software units or modules, while system testing is the process of testing the components of an entire system together. Both integration and system testing are important for larger projects because they have so many more components and parts. The main purpose of both of these types of testing is to expose the faults of the interactions between components or modules. Different approaches to integration testing include Big Bang Integration, Bottom-Up Integration, Top-Down Integration, and Mixed Integration. Regression testing is a type of system testing that ensures that any changes that have been made to a system during development or updating hasn’t introduced new bugs. Finally, acceptance testing is typically done after system testing and is focused on satisfying the end-user. Beta-testing is the most common type of acceptance testing and involves users testing the software in a real-world environment.

Because I had no experience with software testing prior to this course, my mindset going into the project was extremely open. I had little knowledge of what it meant to be a software tester, so I wanted to gain as much knowledge as possible. As I progressed in the project, however, I began to notice that I was thinking more about the JUnit tests than the actual code I was writing. I was thinking about the boundaries of the specifications and what specific scenarios would need to be tested for. At the beginning of the project, I did not apply much caution when writing the code. However, as I progressed and started having a better understanding of what software testers do, a moderate amount of caution crept into my mindset. Probably not as much as would be needed for a professional software tester, but enough so that when writing the code, I was wondering what it would do to the system. This illustrates how important it is to consider the complexity of the interrelationships of the code because it directly impacts code quality and performance. For example, at the beginning of the project, I didn’t have nearly enough tests, and my code coverage percentage was low. But as I added more and more tests, the coverage increased, improving the overall quality of the system.

When developing your own code, it can be easy to allow bias into the code review. You’re the one who wrote it, so obviously, you think it will pass most, if not every, test. However, I attempted to mitigate this by testing everything multiple times, despite whether or not I knew it would pass. I tried to form hypotheses instead of assumptions as well. Additionally, ensuring that you test both bad and good input can help eliminate biases. For example, when testing that the Contact's first name was 10 characters or less, if I had only tested a bad input, I would have missed that the exception was always being thrown, whether it was good input or not.

It’s vital to be disciplined in the commitment to quality as a software engineering professional because it helps uphold and “advance the integrity and reputation of the profession” (Software Engineering Code, 2018). It’s important not to cut corners when developing or testing a project to avoid compromising quality. The quality of your work is what people will remember, so it’s important to take pride in that and not throw something together lackadaisically. Avoiding technical debt is also an important task, and this topic should be revisited frequently. Sometimes technical debt is not always a bug either, but a missed use case or incorrect interpretation of requirements. To avoid stacking up technical debt, user satisfaction should be continually monitored and polled, and a combination of both static and dynamic testing tools should be used to fix errors. Additionally, adopting an agile approach when appropriate can help to avoid technical debt as it tests early and often and pushes for open communication channels between the client and the developers.

References

Doyle, A., Button, B., & Boos, P. (2023, August 25). *What is Acceptance Testing?*. Agile Alliance . https://www.agilealliance.org/glossary/acceptance-testing/

GeeksforGeeks. (2023, November 16). *Unit Testing - Software Testing*. GeeksforGeeks. https://www.geeksforgeeks.org/unit-testing-software-testing/

Heusser, M. (2021, December 15). *What unit test coverage percentage should teams aim for?: TechTarget*. Software Quality. <https://www.techtarget.com/searchsoftwarequality/tip/What-unit-test-coverage-percentage-should-teams-aim-for>

*Software Engineering Code - ACM ethics*. ACM Ethics - The Official Site of the Association for Computing Machinery’s Committee on Professional Ethics. (2022, June 8). https://ethics.acm.org/code-of-ethics/software-engineering-code/