Project Two

CS-320

Brady Steele

10/16/2022

My testing approach was very aligned to the software requirements that have been outlined for this application. For both the contact service and task service, specific requirements for classes and services were laid out, to meet the functionality of the application. As an example, for the task class, it was necessary to have a task ID less than or equal to ten characters. In the code, I have implemented a check that will throw an exception if the ID does not meet this requirement. As an example, for the contact class, it was necessary to have a contact ID less than or equal to ten characters. In this code, just as the above, I implemented a check that throws an exception if the ID does not comply. For each of the above examples, a Junit test has been created to check if an exception is thrown. I know that my Junit tests are effective for the contact service and task service. I can confidently say that based off the specified requirements for each service, I have implemented checks and Junit tests to make sure that the data complies with the requirements. I have coverage that will ensure that the application produces and stores the correct data for the application.

I have ensured that my code is technically sound, through debugging and making sure that my tests match the requirements and produce the expected functionality. In the Task.java file, I will cite the previously mentioned code for checking the task ID. The below code ensures that the task ID complies the with requirements to stay at or below ten characters.

private void checkTaskId(String *taskId*) { *// Check task id*

if (*taskId* == null || *taskId*.length() > 10) { *// Check if task id is null or longer than 10 characters*

throw new IllegalArgumentException( *// Throw exception*

"Error. Task ID must be between 1 and 10 characters." *// Message*

);

} else { *// Else*

this.taskId = *taskId*; *// Set task id*

}

}

This code throws an exception if the condition is not met, and my Junit test checks for the thrown exception, as demonstrated below.

@Test

void TaskIdTooLongTest() { *// Test for task id too long*

Assertions.assertThrows( *// Check if exception is thrown*

IllegalArgumentException.class,

() -> new Task(longId)

);

}

I follow the same protocol for all the tests and checks that I have put into the code to ensure functionality.

I can check the efficiency of my code to perform these tests, by running a build on my code and seeing the output of my Junit tests. Keeping the runtime down on the code will increase the responsiveness of the application and make the experience for the client even better. Below is an example of the Junit test runtime for my task service code. The code performs the vital functions from the requirements and does not include any unintended functionality.



For each of the project components, I employed both black box and white box testing techniques. Both techniques can be utilized to obtain more complete testing coverage for the application. Black box testing assumes that the tester is essentially blind to the internal structure or functions of the application. Black box testing is based on the tester not needing to know or understand how the application will be finally implemented and released as a product. With this type of testing, the tester only needs to care about the input into the application, and the outputs that are created from the application. This type of testing applies to me because I was ensuring that the functionality of the code was seamless, and the application was able to deliver the intended outputs, whilst taking in the necessary inputs from the requirements. White box testing assumes a much deeper knowledge and understanding of the code and application. Having been the only person that has worked on developing the code base for these milestones and possessing a deeper understanding of the technology, I am able to do white box testing as well, as I am taking on the roles of both the developer and the tester. The main method that I used to test the application was through Junit testing, which provided a very easy and convenient way for me to run the tests and produce results. Another testing technique that I did not utilize, was automated testing. Automated testing requires employing the use of automated testing tools such as Selenium, to run the tests and produce results for you. For these milestones, the tests were written and carried out within the application, in separate testing files. For automated testing, a tool would be able to carry out the testing based on the parameters that are set within the tool. Black box testing is used in situations where the developer and tester are not the same person. Within these milestones, I was playing both roles, but in a real-world scenario, there will typically be different people in these roles. The tester may not know the exact details of the technology and application solution, but black box testing allows them to still ensure efficiency and correctness. White box testing is utilized in a scenario like these milestones, where the tester fully understands the underlying technology in the application. Automated testing can be utilized in a production environment, where you need to ensure functionality across multiple different machines, while making sure that human error is not present in the results.

During the development of this project, I had to ensure that I was employing caution throughout. Taking on the role of a software tester means that I needed to ensure that I was practicing caution and making sure that I was covering all the facets that needed to be tested and doing it accurately. It is important to appreciate the complexity of the code because these relationships between different objects and components can leave vulnerabilities and gaps that will be passed over if not taken into consideration.

I had to make sure that I limited bias in this code, since I was taking the role of not only the tester, but also the developer. Being the developer of the code can lead to bias because you want to always think that your code and application is flawless. This can be troubling, because a developer may not see apparent mistakes because they think their code is very good and they don’t make mistakes. I had to look at my code from the standpoint of a tester, where I could look for holes and vulnerabilities that I may not have considered when writing the code for the application. Being able to step out of the developer role and look at the code from a different perspective helps to limit bias and helps to gain a deeper understanding of the overall functionality.

It is very important to not cut corners as a software developer, because these can be extremely costly for the organization. Trying to save time for yourself can and will create more of a demand on your time in the long run due to having to correct mistakes. As a developer, you are being trusted to create quality applications for your organization, and you need to make sure not to abuse that privilege. I plan to avoid technical debt by creating efficient, reusable, modular code. This will ensure that I am not the only developer that can work on my code base, and that I can port over packages of code, so that I can save time by implementing code that has already been tested in one environment. I believe that creating clean code is always the way to go, even if it takes longer, because it will save yourself and your fellow developers a lot of time overall.