CS 320

Professor Kraya

Jeremy Snow

24 February 2024

**Project Two**

In order to create unit tests for each of the milestones, it was necessary to ensure that each requirement was covered. Not only that, but that edge cases had a unit test for the boundary, one before the boundary, and one after the boundary. Each requirement also included that null was not allowed, and thus a unit test verifying the proper handling of a null case was also needed. Finally, in the case of the ID variable for each service, the ID is not allowed to be changed after initialized and this required a test case to show that the service would properly handle this scenario as well. Factoring each detail of the requirements for each service into a specific test case allowed the testing approach that was taken to align as closely to the requirements as possible.

When inspecting the overall quality of the JUnit tests that were created by using coverage percentage, one can see that each piece of the code has been covered by a test case. The test suite for each service has 100% code coverage and also covers boundaries, one before, and one after. Although this doesn’t guarantee that the code is error or bug free, it does guarantee that each piece of the code has been exercised and thus helps to reduce the probability that a faulty piece of code will slip through the cracks.

Making sure that my code was technically sound involved creating test cases that dealt with edge cases on the character length of strings for object initialization, as well as invalid input, such as null. Test cases were also created for instances such as when a non-unique ID is supplied for a contact, task, or appointment which is seen starting at line 44 of the ContactServiceTest.java file in the givenDuplicateContactId\_whenAddContact\_thenThrowIllegalArgumentException() method. This test case uses assertThrows to check for an IllegalArgumentException when a contact is added containing an ID that is already stored in the contactList. This is also tested in a similar way in the TaskServiceTest.java file regarding task IDs as well as in the AppointmentServiceTest.java file regarding appointment IDs. Other things that were checked for were situations such as attempting to delete a nonexistent contact/task/appointment, or when trying to update a contact/task/appointment that doesn’t exist.

Regarding efficiency, the test cases that were designed focused specifically on the essential functions of the code and avoided testing for things that were not requirements or related to the proper functioning of the code. Targeted test cases were designed to test edge cases eliminating the need to test for situations that fit within the given boundaries. This can be seen in the givenTaskIdAtBoundaryOfOuterEdgeCase\_whenGetTaskId\_thenReturnTaskId() test unit as well as the givenTaskIdOneLessThanBoundaryofOuterEdgeCase\_whenGetTaskId\_thenReturnTaskId() test unit starting at line 12 in the TaskTest.java file. Combined with the tests for an invalid taskId, name, and description, these test units verify that the only allowed input adheres specifically to the requirements given.

Overall, the generated test units for the task service and the contact service efficiently tested 100% of the code generated for each service. These test cases verified that proper input is required, and that improper input is handled correctly. In addition, the test cases verified that each file created met the requirements laid out in the Rubric.

In module three, the first milestone was to create a ‘Contact’ class and a ‘ContactService’ class. The ‘Contact’ class created ‘Contact’ objects containing an ID, first name, last name, phone number, and address to be created. The ‘ContactService’ class created a list of contacts and allowed ‘Contact’ objects to be added, deleted, and updated. The second milestone was to create very similar code, this time creating a ‘Task’ and ‘TaskService’ class. The ‘Task’ class contained objects with an ID, name, and description. The ‘TaskService’ class initialized a list of tasks and allowed tasks to be added, deleted, and updated, similar to the ‘ContactService’ class. Finally, in the third milestone, the ‘Appointment’ and ‘AppointmentService’ class were created. The ‘Appointment’ class created ‘Appointment’ objects with attributes for an ID, date, and description. In the same light as the last two milestones, the ‘AppointmentService’ class initialized a list of appointments and allowed new appointments to be added or deleted from the list.

In each module, in order to verify the proper functioning of the code, Junit testing was utilized. Junit testing is a form of white-box testing which derives the test cases directly from the code that it is testing. For example, in each of the milestones, an ID attribute was used to identify individual objects, in order to guarantee that each object was unique. This ID attribute was required to be no more than 10 characters. Thus, several test units were created that made sure that ID’s 10 characters long and shorter were allowed, and ID’s longer than 10 characters were not allowed. This is done through assertions which is where we assert that what is expected to happen, actually happens. One instance is when an ID longer than 10 characters is given, we assert that the program will throw an ‘InvalidArgumentException’ error. If the assertion is correct, the code is functioning properly for this test case. When the assertion fails, we know that something isn’t working as expected in the code.

Although the focus in these milestones was on white-box testing and more specifically, Junit testing, there are several other types of testing that exist. Some of these testing techniques include static testing, black-box testing, and experience-based testing. Static testing involves examining the code without executing it. This can include code reviews, walkthroughs, and inspections to identify defects or issues early in the development process. Unlike dynamic testing, static testing does not require code execution. Instead, it focuses on analysis and review of the code and documentation. For example, code reviews involve team members examining the code line by line to identify errors, inefficiencies, or potential improvements. This technique can be particularly useful for catching logic errors or inconsistencies in the codebase before they manifest as runtime errors.  
 Black-box testing, also known as behavioral testing, focuses on testing the functionality of a system without knowledge of its internal workings. Test cases are derived from the system's requirements and specifications, rather than the code itself. Testers interact with the system as an external user would, inputting specific inputs and observing the outputs. This approach ensures that the system behaves as expected from the end user's perspective. For instance, in a web application, black-box testing might involve submitting forms, clicking buttons, and navigating through different pages to ensure that the functionality meets the specified requirements.

Experience-based testing relies on the tester's expertise and intuition to identify potential issues in the software. This technique draws on the tester's past experiences and knowledge to anticipate where defects might occur. Testers may use techniques such as exploratory testing, where they interact with the software in an improvised manner to uncover unforeseen bugs or usability issues. Unlike formalized testing methodologies, experience-based testing is less structured and relies heavily on the tester's judgment. For example, a tester with extensive experience in mobile app development might intuitively focus on areas such as performance optimization or compatibility with different devices.

Each of these testing techniques has practical uses and implications for different software development projects and situations. For instance, static testing can be invaluable for identifying issues early in the development process, reducing the cost and effort required for bug fixes later on. Black-box testing is essential for validating that the software meets the specified requirements and behaves correctly from the end user's perspective. Experience-based testing complements formalized testing methodologies by leveraging the tester's expertise to uncover subtle issues that may not be caught through automated or scripted tests alone. Depending on the project's requirements and constraints, a combination of these testing techniques may be employed to ensure thorough coverage and quality assurance throughout the software development lifecycle.

While working on this project, I attempted to have the mindset of a tester for the majority of the time. Adopting this mindset even while creating the code for the services allowed me to analyze what was necessary for this code to meet the requirements and to function properly. This also required that I employ caution as I created both the code for the services and the test cases designed to test the code. By being cautious and examining each piece of the requirements and ensuring that the code captured all of the required functionality, I was able to make sure that the code would function as intended. The same process was used while creating the test cases and then verified by running the test files with coverage. In breaking down both the requirements and the necessary test cases required, I was able to appreciate the complexity and the interrelationships of the code that was being tested. This can be seen in each test file where very specific test cases have been created testing only one piece of functionality at a time.

When creating the test cases for each service, it was necessary to approach the testing with as little bias as possible. Pretending like I did not create the code and approaching the testing with the intention of finding defects and bugs allowed me to effectively create test cases to verify the functionality of the code. On the software developer side, bias is definitely something to consider when creating testing for code that you have developed. Allowing an unbiased party to create the testing instead is more than likely a better option. However, creating test cases effectively while also having developed the code is definitely possible, as demonstrated in this scenario.

As a software engineering professional, being disciplined in the commitment to quality is extremely important. Creating code is a meticulous endeavor and the tiniest of errors can cause massive problems. There are many examples in the real world of coding failures causing catastrophic harm including the loss of human life. One example is the issue that I discussed in my discussion post regarding when Porsche had to recall a significant number of their Taycan vehicles due to software issues that caused complete power loss to the engine. These types of issues are avoidable and should be avoided, especially when human lives are at stake. Not cutting corners is a given if one is determined to be successful in this field. This applies to both writing code and testing it. With this approach in mind, I intend to do my best to avoid technical debt as a future programmer in this field. It will always be my goal to create code that is ethical, efficient, and well thought out. In addition, my testing efforts will aim to be as unbiased as possible and will attempt to find the bugs and defects in code if they exist.

References

Garcia, B. (2017). Mastering software testing with JUnit 5: Comprehensive guide to develop high quality Java applications. Packt Publishing.

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). Software Testing: An ISTQB-BCS Certified Tester Foundation guide (4th ed.). BCS Learning & Development Limited.

Hoffman, C. (2021, July 4). Porsche recalls taycan EV for total loss of power. Car and Driver. <https://www.caranddriver.com/news/a36904751/porsche-taycan-recall-announced-power-loss/>