**Project Two**  
CS-320 UI/UX Software Test Automation & QA 22EW5  
by. Corey Nance  
6/19/2022

The first part of JUnit testing is to test each of the class, Contact Class, Appointment Class, and Task Class based on the requirements that were assigned. The testing approach was to set up each test to reflect both sides of the requirement parameters, meaning that the goal is to set up testing for when the action is successful and for when the action fails. One example of this is with the Contact Service, Appointment Service, and Task Service requirements for their perspective objects. The requirements state that the object Id must be unique, be 10 or less characters, cannot be null and cannot be updateable. To properly align the test scenarios with the requirements, the test must first check that the object can be created within all the parameters and then test the time when the object can’t be created. To achieve this the tests will verify the Id length not being greater than 10, the id not being null, the id being unique, and the id not being updateable. Within Test code that tests each class, there are tests that check to make sure an object is created and test to check every reason for the object failing. If the object fails, the test cases will be able to see where it failed by throwing an Illegal Argument Exception. This can be seen by attempting to create objects like the testContactIdToLong, where the ID is set up to be longer than the requirement, this test should fail and throw an exception error. Since this test is set up to check and see if it does fail, the error would confirm that the program is working correctly. This approach covers the full extent of the software requirements by testing the code to make sure that it is requiring the software requirement parameters and throwing errors when the parameters are not met. The approach or plan as to how the tests are going to align with the requirements is the first step, the next step is the overall quality of those tests. To figure out the effectiveness of the test scenarios, a JUnit coverage test is run. This test gives the percentage of actions that are covered by the created tests, the higher the percentage, the better quality your JUnit tests are. With the Contact Service, the Task Service, and the Appointment Service test the JUnit test coverage was one hundred percent. With having a coverage percentage rate that high, that means all passing and failing scenarios as well as all decision scenarios or if statements are covered.

With every area of the code being covered by the test, the next area is to go over how the code is technically sound. In general, to achieve the goal of having a technically sound code, was to first try to keep all the test function naming conventions representing the task they were testing, one example on line 53 of TaskTest.java, the test called testTaskIdNotNull which tries to create a new object with the task ID being set to null. Even though IDs cannot be updated because the functions use the ID to update the other parameters, to help achieve a technically sound code, another test is created to test an update function by name. This method does not allow for IDs to be updated as well as a test to make sure they cannot be duplicated. This can be seen on line 163 and 182 of the TaskService.java where objects are created, then an object update is attempted, one with changing the ID and the other making the ID the same as an existing one. Both tests look for an Illegal Argument Exception and if found the test passes meaning that the code throws a failure when either of those actions are attempted. Making sure to critically look at every little detail within the created code is the key factor to ensure that the code is technically sound.

To ensure that the code runs efficiently, looking at how long the test must run was the key factor. Most of the function test ran remarkably close to 0.00 with one test on line 100 of ContactTest.java, testContactAddressNotNull, took 0.000 seconds to complete. The longest test within the same file on line 16 ContactTest, which is the object creation test took 0.016 seconds. Analyzing the speed of each test and then seeing how low every test is and with a completion of all tests being 0.001 seconds for the first file and 0.040s for the second file shows that the entire code runs efficiently and ensures that the code is efficient.

With using JUnit testing, this allows the developer to test every action that their code is doing, through this testing, the code can be verified to meet all system requirements. Using this testing method can not only be used to verify code actions, but it also shows how quickly the code is able to be run. Using these two verification methods and making sure the code is technically sound ensures that there is a quality code that meets all the software requirements.

These three programming assignments all had the same code design, but with just minor changes to call instance variables and constructors. JUnit 5 is the testing framework that was used to code the different testing methods that will test the actual code verifying that it meets the software requirements. The main type of testing done was unit testing, a methodology of writing out small tests by creating various test cases to verify the behaviors or output of an individual unit of code (Bansal, 2021). Each testing method is designed to test the outcome of the main code page methods based on the software requirements. The first set of methods are designed to assert data that fits the requirements and give a passing sign if everything goes through. The second set of methods asserts data that should cause an error, and if an error is successfully thrown, then a passing sign is also given. Because the main set of code has more than one function that in some cases rely on each other, a little bit of regression testing should be used. Regression testing is when new code and already existing are all tested to ensure the existing functionalities still work fine with the new code changes (*What is regression testing? definition, test cases (example)* 2022). This testing method was used to ensure that not only the code method works as intended but also the entire code base also works together as intended. For these assignments, unit testing and light regression testing was all that was needed.

Unit testing and regression testing are not the only style testing that there is even though that was all that was used here. Some of the testing techniques that are available but were not used were Integration testing and black box testing just to name a few. Integration testing is when software modules are created locally and tested as a group (*Integration testing: What is, types, top down & bottom up example* 2022). This is usually seen when sections of code are designed separately and then later combined and tested. For this reason, is why integration testing is under the not used category, but the argument could be made that the Service.java pages interacting with the Object Class pages could be a type of integration. With black box testing, the test is done from an external or end user perspective (*What is white box testing? techniques, example & types* 2022). Because this program has never hit the end user or was done externally, the black box testing is not needed at this phase of the programming. These two are just examples of the many testing techniques that were not used.

With the various techniques discussed, each one with its practical use. First Unit testing, which is practical for testing small sections of code to make sure they are working as intended and in doing this will help catch errors early on within projects. Regression testing’s most practical use is really testing how the new small sections of code affect the already existing sections of code which can help catch unintended output changes early in the development cycle. Black box testing is when the code is tested without knowing the actual code. A practical use for this technique is when the focus needs to be on the behavior of the code while not knowing the actual code which is helpful for knowing how a user will interact with the code. Lastly is integration testing where its practical use in a project would be once all the codes are combined. As an example, if Contact Service, Task Service, and Appointment Service all must be combined, then integration testing would have to take place.

As far as the mindset while developing and testing this code caution, bias and discipline have to be taken into account. Caution is employed when trying to verify every angle where a requirement could not be met within the code. The main example of this was with the Contact Service Class. Within this code, it was found that even though the ID could not be updated due to the updateContactID method using the actual ID to update the contact, so the ID could not be changed, But the ID itself was changeable is some way. The resulting effect from taking this caution was to create a method that updates by name and then creating a block of code that checks to make sure the contact ID is not able to be updated.

When it came to trying to limit bias, the strategy that was to try and test and successful cases when the code works at designed and also test the failure cases to try and find all the ways the code is supposed to fail. This can be seen through the entire code; one example is with addTask method in the Task Service class. When a task information is entered correctly, the test passes, but now it must be checked to see what happens when information is not entered correctly. This is where other test methods are made to try and enter information that does not fall within the requirements criteria, like entering an ID that is too long, entering a duplicate ID, or just not entering an ID at all. There is a test for each one of those scenarios testing to make sure that the code throws an error if the error is thrown the test passes. This strategy of testing passing a failing scenarios seems to be an effective way to limit code test bias.

In the area of discipline, cutting corners is where most of the discipline faulters. The most common reason for cutting corners when it comes to testing is that the task is too time-intensive (Farias & Farias, 2018). It is important to be very disciplined when developing a code and try to find errors early. Fixing a bug found in the later stages of a project is at least 30% higher than fixing the same bug earlier on (Farias & Farias, 2018). When it comes to technical debt, this is when teams use an easy solution now instead of using a better approach that would take longer (Wolpers, 2019). The main strategy used is within the naming conventions, as well as limiting code redundancies. Using the most efficient code that can get the task accomplished with fewer lines of code as well as using easy to read naming conventions helps for scalability and readability. This can take a longer time trying to design the code to handle multiple tasks within a few lines of code but overall, it will lead to having less technical debt.

**References**

Bansal, A. (2021, November 16). *Best practices for unit testing in Java*. Baeldung. Retrieved June 3, 2022, from <https://www.baeldung.com/java-unit-testing-best-practices>

Farias, C. S. D. B. H., & Farias, H. (2018, July 2). *Where you should never cut corners in software development*. InfoWorld. Retrieved June 19, 2022, from <https://www.infoworld.com/article/3286265/where-you-should-never-cut-corners-in-software-development.html#:~:text=Given%20the%20risks%2C%20there's%20no,software%20from%20the%20ground%20up>.

Hamilton, T. (2022, April 16). *Integration testing: What is, types, top down & bottom up example*. Guru99. Retrieved June 3, 2022, from <https://www.guru99.com/integration-testing.html>

Hamilton, T. (2022, April 23). *What is white box testing? techniques, example & types*. Guru99. Retrieved June 3, 2022, from <https://www.guru99.com/white-box-testing.html>

Hamilton, T. (2022, March 17). *What is regression testing? definition, test cases (example)*. Guru99. Retrieved June 3, 2022, from <https://www.guru99.com/regression-testing.html>

Wolpers, S. (2019, June 17). *Technical debt & scrum: Who is responsible?* Scrum.org. Retrieved June 19, 2022, from <https://www.scrum.org/resources/blog/technical-debt-scrum-who-responsible>