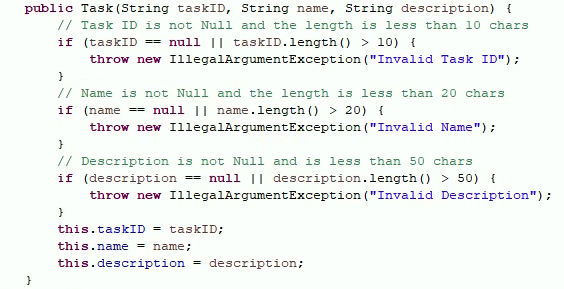
Branden Langhals

CS-320 Project 2

April 11th, 2023

7-2 Project Two

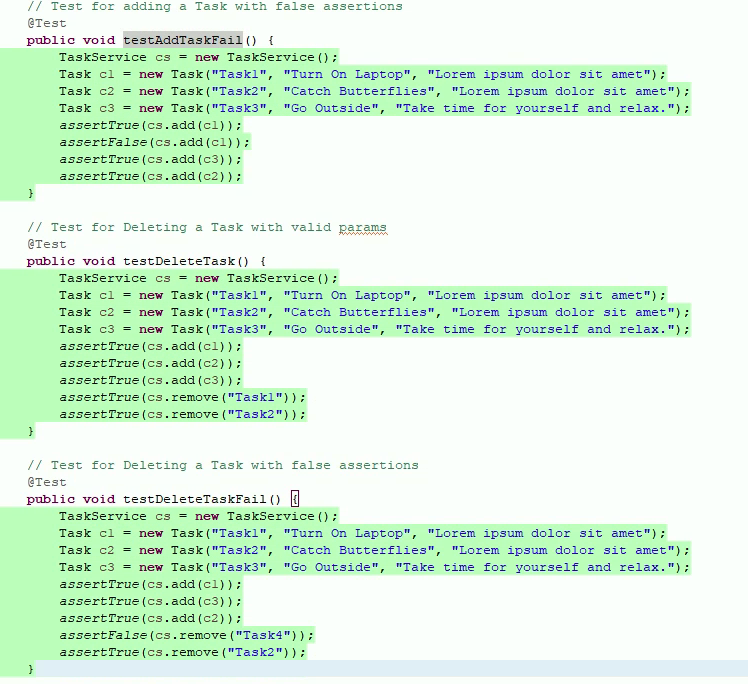
For each milestone, the same approach was used to effectively test the software requirements, I first created the two necessary classes of Contact and ContactService, Task and TaskService, also Appointment and AppointmentService. Within these classes, the software requirements are defined. For example, the Task class requirements were that the TaskID, Name, and Description cannot be longer than 10 characters, and not be null. The code for this area is as follows:



By creating a public class of Task within this file, my Junit test files can access the class file’s test requirements. I created two new Junit test files called TaskTest and TaskServiceTest. By creating a test for each class file, I can accurately test the requirements of each class.

My Junit tests focused directly on the project requirements and addressed specifications as stated in the class files. I created tests with correct data and tested against false assertions. Using false assertions lets me know that if data is passed that is outside of the requirements the test should fail, which in this case a failure means that the test passes. I monitored test coverage by checking the Coverage tab and observing that my tests were covered 100%. 

By properly naming all classes and constructors, I created clean and easy to follow code with corresponding test cases. By using logical naming for test methods, I can check what each method is covering and then follow through with its respective requirements. It is easy to assume what testDeleteTask and testDeleteTaskFail will do because of proper naming.



Test creation for Contact, Task, and Appointment services were done using Junit. Unit tests were created to validate assertions that were created for each category. For example, one requirement was to create a new Contact with character count restrictions. Code was written to create methods that will physically create a new contact. A separate java file with Junit tests was implemented to make sure that the method was working as expected and that the data entered met requirements. These Junit tests use both valid and false assertions. Valid assertions would pass data that would match program expectations to ensure that if a user enters the correct data within parameters that the test will pass. Alternatively, we want to know that if a user enters data that is not valid that the appropriate response is given, this is when we test using false assertions.

Throughout this process, we did not implement Integration, System, or Acceptance testing. All these stages are conducted once the product is more refined and typically when it has a UI built out and is called Black Box testing. The next step in the process after these unit tests would be to implement Integration tests which would test the same areas as the units but utilize the built-out application and not access the codebase. After Integration testing is done, System and Acceptance testing is conducted using the final product. At these later stages, QA Analysts and members of the product team will check product requirements and test them, assuming the role of an end-user.

Each of these testing methods have a place in the SDLC, but Unit tests are typically conducted during the development phase whereas Black Box tests will be done later using a refined product. Realistically each of these phases will be touched on during Agile development and before a product is released to its users. However, there are times when unit testing can be done after a product has been made. Unit tests can be run each time new code is merged with the main branch to ensure that the updates do not interrupt other areas of the code and usually this is done in the form of API tests and can use some form of continuous integration.

My mindset throughout this course was to maintain an open perspective. As a full-time QA Engineer, I was looking forward to taking this course to see what additional knowledge I could gain and bring into my professional career. Because of this, I wanted to thoroughly look through the course materials provided and take my time with each assignment. When coding a test, I would make sure that my assertions were valid and run the code over as many times as necessary until the tests pass. Taking time to be meticulous in software development will always equate to a better final product. Working at an elevated level in this industry means staying disciplined, focused on requirements, and matching them to the end user’s experience. One key resource I fell back on throughout this class was Testim.io. I currently create automated tests using this software, but they also have a lot of great best practices regarding the prevention of regressions and how to rigorously test your code.

Sources:

Testim. (2021, March 11). *Unit testing best practices: 9 to ensure you do it right*. AI-driven E2E automation with code-like flexibility for your most resilient tests. Retrieved April 16, 2023, from <https://www.testim.io/blog/unit-testing-best-practices/>