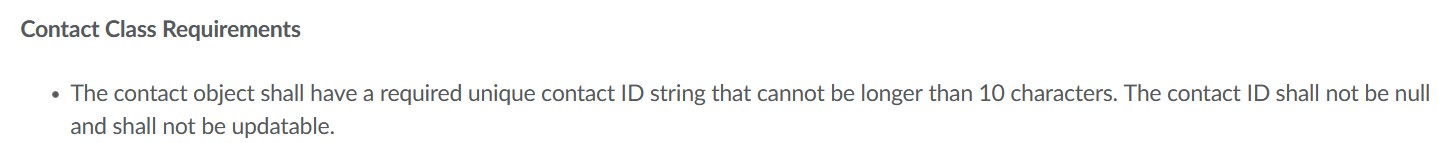
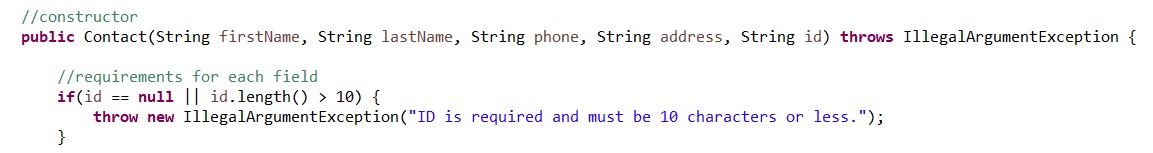
Alexandra O’Donnell

CS-320 Project Two

6/14/2021

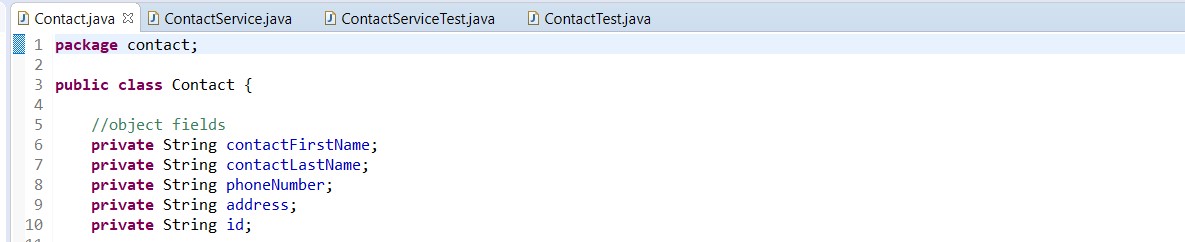
When programming each of the three main requirements for this project – the contact service, the task service, and the appointment service – I used the requirements given to me for the application as a direct guideline for my code. Every data field listed in the requirements document became a “field” in my class – either a variable or an object. Every restriction on one of those required fields became logic in the constructors of my contact or task or appointment objects. Here is an example of the restrictions on required fields for a contact object, and my code for the contact object constructor:

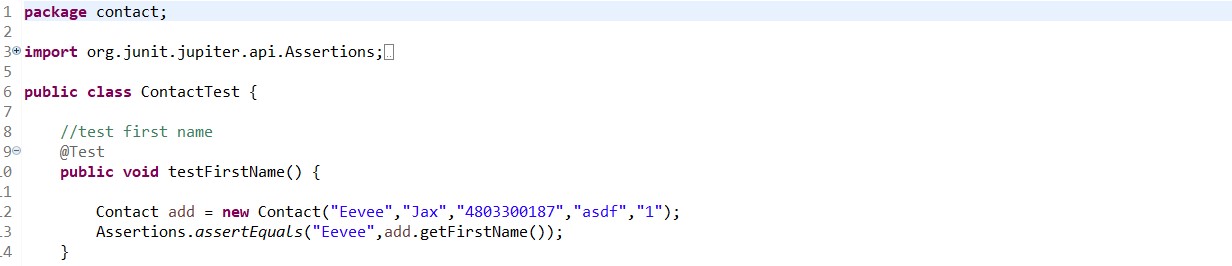




The overall effectiveness of my jUnit tests is actually very easy to find out. Eclipse has a useful feature that allows you to see the coverage percentage of a jUnit test in relation to all of the java files in your project. So, if you run a coverage-jUnit test on your “ContactService” object, Eclipse will show you what percentage of the ContactService class was used in the test, and it will also show you line-by-line what was used and what was skipped over. This feature is really important for testing, because it allows you to see exactly which parts of your code you may not be testing at all. This means you can go back to your jUnit tests and add more tests or more test logic to ensure that you are testing for as many potential use cases as you can.

When I first received the requirements for each portion of the application (contact, then task, then appointment) I wrote my jUnit tests just like I wrote the actual classes – I used the requirements as a guideline. I also went through every single variable and method in an object when writing my jUnit tests – for example, my contact class has fields for first name, last name, phone number, address, and a unique ID, and getters and setters for each field – and I wrote a specific test for every one of those items. Having a high percentage of coverage and using the requirements and class fields as a guide while writing my tests has helped to ensure that my code is both technically sound and efficient. Here is an example of my task class and one of the tests that I wrote for it, based on the task class’s attributes:





The testing library that I have used while putting together my application project is called JUnit 5, and it employs a programming model called Jupiter. Jupiter has quite a few useful features that help to simplify test-writing, which is very helpful for people like me who do not have a lot of experience with writing unit tests. Some of the features that I’ve used in my code are described in detail in Garcia’s *Mastering Software Testing with JUnit 5.* The easiest way to declare a method as a test is to annotate it, with “@Test” immediately before the start of the test method so that Eclipse knows to treat it as a unit test (Ch. 3). I used the “@Test” annotations before my test methods, and instead of writing any methods with the “@BeforeAll”, or “@AfterAll”, etc. annotations, I put all the method calls that I would need for each test within the test itself. Another testing method that I used in my program is Assertions. About Assertions, Garcia writes “An assertion (also known as a predicate) is a boolean statement typically used to reason about software correctness” (Ch. 3). I’ve used assertions in all of my test methods to make sure that my method calls and inputs are getting the results that I’m looking for.

One testing method that I did not use in my program testing was nested tests. According to Garcia, to write a nested test, you must “...annotate inner classes with @Nested and all test methods in there will be executed as well, going from the regular tests (defined in the top-level class) to the tests defined in each of the inner classes” (Ch. 3). Because I went through my classes and requirements while I was writing my unit tests, and since this is the first time I have ever written unit tests, I decided to keep my tests simple and I wrote a separate test for each type of parameter I wanted to test. For example, I wrote a test to create a task object with a task-name under twenty characters, and a separate test to create a task object with a task-name greater than twenty characters so that my task constructor would throw an exception.

I think that the use of tests on your own code is a sort of way to limit bias. At least for your own code, I think the bias might be that you feel you’ve covered all the requirements and that you’ve already thought of and accounted for all the possible use-cases. So, by testing your code, and especially by checking the code coverage when you run your tests, you have a non-biased view and direct feedback on your program. This allows you to make any necessary changes, and end up with a polished finished product.

Being disciplined and cautious while writing your code is a skill that I believe all developers should have. I think that being cautious has its time and place, since as a software developer you will always be working on a timeline. But, being at least a little cautious while writing your code will likely help you avoid mistakes that could cost you more time later on. Being disciplined to me means thinking out as many possible use-cases as you can. Using unit testing as a tool to test those use-cases against your code is essential, in my opinion. Even though when you’re pressed for time in terms of getting your program or deliverables done and out the door, taking the time to test make sure your code is sound and complete is way more important. Thorough testing could save you time in the later stages of your development lifecycle, and it could save your company from releasing a defective product and having to deal with the repercussions.

References

García, B. (2017). *Mastering Software Testing with JUnit 5*. Packt Publishing.