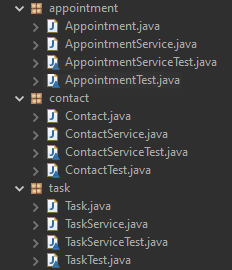
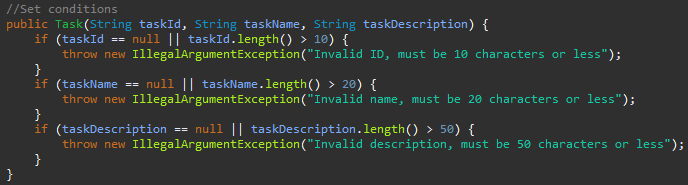
My unit testing approach was purposefully aligned with all the software requirements.

I tested each requirement individually, and created specific test cases for each class that was associated with a specific feature. With the feature and class Contact, for instance, I created a JUnit test class specifically for that feature, and within the hierarchy of files and packages.

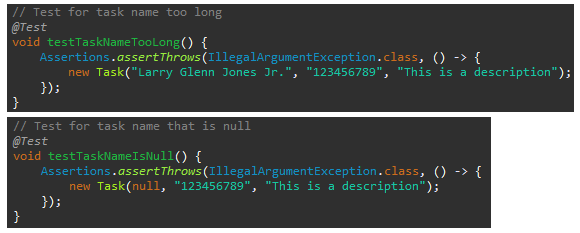


"Contact" feature included as an individual class (Contact.java), and subsequent JUnit test case (ContactTest.java) is included within the hierarchy of files within a "contact" package in the source folder.

As far as individual objects, their characteristics and conditions, I created blocks of test code for each as well, within the test class for each feature. For instance, a requirement for the Task feature was that it included an object taskName, whose required characteristics were that it shall not be null, and be 20 characters or less. For this requirement, the code and test code for the condition was written as such:



"taskName" object code in Task class



"taskName" test code in TaskTest class

Based on the coverage percentage, JUnit testing done with which all characteristics were tested with blocks of test code was effective, as all the characteristics were covered with tests. The goal is to reach 80% coverage with the test methods. In some instances, less than the total of all characteristics were tested.

With the above graphic examples given, it can be demonstrated that the code written was technically sound, as the required characteristics were tested individually. Efficiency was reached, as only the required characteristics were tested. In Eclipse, there is actually an alert for writing code blocks with specific variables more than once. I did encounter this alert, as I copied and pasted certain blocks and later changed their content.

One of the most used software testing techniques in this project was assertions. Assertions are included in a collection of utility methods, and they are used to assert conditions in tests. They can be seen in the graphic examples above. Assertions give unit testing a self-validating characteristic, and they are considered very powerful and useful tools for determining pass or fail status of a test case.

Not used in my project to any degree are assumptions. In contrast to assertions, a failed test with assumptions actually results in an aborted test. These are included in JUnit 5 as a static method. Assumptions are used in cases where continuing with a given test method does not make sense (Gupta, 2017). Testing with an assumption is done on a condition that is not present within the current run-time environment.

I was very cautious in writing the code and test code for this project. There are interrelationships between the blocks of test code in their test classes and respective blocks of code and methods in the feature classes themselves. It was important to be detail-oriented with “.” dot operators, parameters, etc. because these details were connecting items between files. For example, the “getAppointmentId” interrelationship between the following getter code and subsequent test code for the Appointment feature:



I can see, as far as software development is concerned, why bias could potentially create “blindspots” in assessing the soundness of testing methodologies and their effectiveness. For instance, we can be under the influence of “resemblance bias”, with which we judge a situation based on the resemblance of a similar situation (Geethanjali, 2018). The problem with this is that we can become acclimated to recognizing a certain type of error, or become familiar with patterns, and we fail to recognize nuisance in our code and test code. In this project, for example, I was given requirement criteria for description length for each feature. In writing the test code, I had to be careful not to repeat patterns with parameters, as the description strings for distinct objects were not always the same. Some required keeping a maximum of 30 characters, while others required a maximum of 50.

Along with what I have learned in this course, I also carry some personal experience as a quality advocate. With something as intricate and complex as software code, we cannot afford to cut corners. We can be efficient, but we have to maintain integrity and functionality. Along with using excellent tools that help to cut down on technical debt such as Scrum, we can be disciplined and diligent analyzers and better communicators. We can remind ourselves that time is money. We can also keep in mind the fallout of carelessness in the examples we have discussed in this course from the real world. Perhaps most importantly, we can take ownership of quality. Each individual who works on or manages a project or product is responsible for that item’s quality, just as though they are handing it off themselves to the end user or customer.

References

Hambling, B., & Al, E. (2019). *Software testing : an ISTQB-BCS certified tester foundation guide*. Bcs.

García, B. (2017). *Mastering software testing with JUnit 5 : comprehensive guide to develop high quality Java applications*. Packt Publishing, Birmingham ‌

Jakubiak, N. (2022, December 6). *JUnit Tutorial With Examples: Setting Up, Writing, and Running Java Unit Tests*. Parasoft. <https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests>

Gupta, L. (2017, April 30). *JUnit 5 Assumptions*. HowToDoInJava. <https://howtodoinjava.com/junit5/junit-5-assumptions-examples/#:~:text=JUnit%205%20assumptions%20class%20provides>

Geethanjali, S. (2018, October 25). *Cognitive Bias in Software Testing: Why Do Testers Miss Bugs?* Software Testing Help.  <https://www.softwaretestinghelp.com/cognitive-bias-in-software-testing>‌

.