My testing approach was consistent throughout each module of the assignment. Since the requirements provided by the customer were clear and concise, I was able to write my method tests in a way that clearly represented each specification, treating the requirements like a checklist.

For example, this test ensures the requirement that states “The contact object shall have a required firstName String field that cannot be longer than 10 characters. The firstName field shall not be null.

| @Testvoid testContactClassFirstNameNull() {  Assertions.assertThrows(IllegalArgumentException.class, () ->  new Contact("11111111", null, "Cawley", "7027027002", "1 Ghostrider Ct LV, NV");  }); }  @Test voi d testContactClassFirstNameToLong() {  Assertions.assertThrows(IllegalArgumentException.class, () ->  new Contact("11111111", "Larryyrrall", "Cawley","7027027002", "1 Ghostrider Ct LV, NV");  }); } |
| --- |

According to Software Testing (Hambling, 2015, p. 151), this approach is defined as a methodical approach. In Section 5.4, this method is described as “failure-based (including error guessing and fault attacks), checklist based and quality characteristic based.” I ensured the code was technically sound by using JUnit assertions, which, in the case above, ensures an expected error is thrown when certain negative conditions are met. This allows us to consistently match our test constraints with the requirements while allowing us to properly handle errors when they occur. Efficiency was maintained by only testing the requirements set forth by the customer and using the functions within the test that allow unique lookups via a primary ID. This is called a “key:value” relationship, hashmap, or dictionary in some languages. It avoids the need to iterate over lists with potentially many items which will slow down retrieval as the list grows.

The techniques I used are described as black-box techniques. This means the tests match the requirements in a one-to-one relationship based on customer-provided information. One example of a technique I didn’t utilize throughout writing tests for this software is the white-box technique. Because clear requirements were provided, the tests were able to be conformed to the customers wishes initially. As requirements change, other types of validation can work to keep data in line in our real software and in testing.

Acting as a software tester let me fully realize how unit tests can effectively be written by being provided the requirements and this allowed me to consider each required test case. It was important to appreciate the complexity and interrelationships because unit-testing should focus on the smallest testable parts of software. I tried to limit the bias by focusing on the customer-provided requirements and this made unit-testing the right technique in this case. In being responsible for testing my own code, I often go into it assuming I’ve written something that will cause a side-effect. This helps go through potential cases of missing data, data that doesn’t fit required types, and data that doesn’t fit within certain prescribed limitations.

It’s important to practice discipline when creating products that others will interact with. If your customer base loses faith in the ability to deliver most bug-free code, this may lead to loss of revenue and hurt the relationship. For this reason, testers must not cut corners as testing software is one of the primary tools available to ensure you release a stable product. Time-estimation is a difficult portion of writing software but it’s important to make sure time is considered for conducting thorough testing and code-reviews.

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition)

- 5.4 Test Approaches (Test Strategies). BCS The Chartered Institute for IT.