My testing approach was aligned with the software requirements because it tested the required methods. For example, Appointment, Contact, and Task all required an id with a maximum length of ten characters. One of the tests could then be to check for an id that is too long. If an entry with an id length of eleven or more throws an Illegal Argument Exception, the test passes. A similar case can be made for each JUnit test I made.

Overall, I believe my JUnit tests were quite effective, once I figured out how they worked. My tests had a coverage percentage of 83.6 percent, with at least 80 being our target. The high percentage lets me know that my tests covered a large portion of the application.

My experience with JUnit tests was mildly frustrating. I did not understand them at first, so I went with my usual testing method. I would manually check that I was receiving the Illegal Argument Exceptions, then comment them out so the rest of the tests could run. For them, I created a function that would print out the current object’s information. In the case of Contact, that meant printing the id, first name, last name, phone number, and address, letting me make sure everything was created and would update properly. Ultimately, I got the same information the JUnit tests provided, but not in the way I was supposed to. Finally, I found out how to use them, then things went a lot better. I don’t remember what I initially did wrong, but things fortunately clicked before Project One.

My code is technically sound, being modular and syntactically correct. I made sure to use appropriate data structures. For example, addContact in my ContactService class utilizes a for loop and an if-else statement. First, we create a Boolean variable that defaults to saying the new contact does not exist in the contacts array. The for loop then checks each item in the contacts array and compares each of their ids with the id of the new contact. If the new contact’s id matches any of the ones in contacts, the Boolean variable changes to true. Once the for loop ends, we enter the if-else block, based on the Boolean variable. If it is true, meaning we found a contact with a matching id, we exit addContact without adding the contact. This is because one of the requirements was to prevent having contacts with matching ids. If we got through the for loop without finding a matching id, the Boolean variable remains false, so we add the new contact to the array.

{

// Check if contact exists

for (Contact c : contacts) {

if (c.getContactId().equals(contactId)) { // if contact exists

contacts.remove(c); // remove the contact

return true;

}

}

return false;

}

Whether my code is efficient or not is a matter of perspective. In the addContact example, the variant of the for loop I used is known as for-each. The line can be read as “for each Contact in contacts, do the following”. Because of this, I didn’t have to find the size of the array to iterate through or declare an iterator. I also reduced the clutter and thus the chance for error. On the other hand, I don’t know if there’s a performance difference between a normal for loop and a for-each. It may be that a sufficiently large number of contacts would cause a noticeable difference in performance with the for-each being slower.

For this project, I utilized white-box testing. White-box testing has the programmer or someone otherwise familiar with the written code, in this case, me, perform various tests with input that I know should or should not work. If everything goes according to what I know about the program, the tests should succeed.

I did not utilize black-box testing. Black box testing has someone who is unfamiliar with the written code and only the final product perform the tests. As the programmer, I would not be able to do this myself. Theoretically, I could have handed the project over to my roommate, but I don’t think that would have gone over well, unless I provided a UI. A black-box testing is typically used by major consumer products. One example is someone releasing a game in a beta state and having the people who want to play the game do so and report issues that can be fixed in the final product.

The mindset I had to apply was one who was learning how to use JUnit tests. I had never done so before, so I had to learn how they worked. I’m not entirely sure how caution applies as a tester, unless it just means that I made sure to only write relevant tests and didn’t break anything. Regarding the complexity of the final product, it was ultimately rather simple, which made writing the tests easier than if we took a much larger project, then started testing. This basically came down to writing a struct, adding a couple functions, testing it, then repeating two more times.

Bias can be a problem when you’re both the developer and tester. Someone might be very confident in what they wrote, which can make them think there are not any problems, thus cutting corners in testing. If I had missed too many possible tests in my program, I wouldn’t have hit the 80 percent coverage mark. That said, you are free to take shortcuts, so long as you do so properly. I copied large amounts of code from Contact to Task to Appointment, making sure to use find and replace to remove any instance of the previous names and replace them with the new one (for instance, finding all instances of “contact” and replacing them with “task”). addContact and addTask were basically the same function. testContactServiceAddDuplicateId and testTaskServiceAddDuplicateId were basically the same test.

I had to be careful, though, because they were different enough to cause problems if I didn’t do this properly. Contact has a first name and a last name, while Task has a name. They’re similar, but removing last name and keeping first name could cause confusion for someone looking at the code for Task to see just a first name. They also had different lengths. If I just kept the first name from Contact and did nothing else with it, it would still be capped at a length of ten characters, instead of the required twenty. Shortcuts are okay, but cutting corners is not. This, along with utilizing good programming practices, helps to avoid technical debt.