Matt Jackson

Project Two

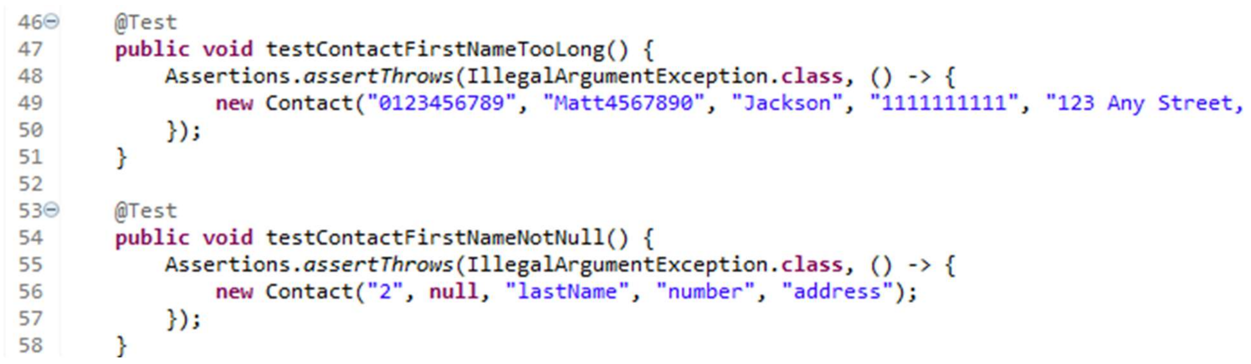
CS-320

Tuft

SNHU

1. **Summary**
   a. Describe your unit testing approach for each of the three features.
      i. To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence.

I made sure to design my tests to test the requirements. For example, in the Contact Class, the firstName String cannot be longer than 10 characters and shall not be null. I created two tests, one with a name that was too long, and one with a name that was null on lines 46-58:



```
46	@Test
47	public void testContactFirstNameTooLong() {
48		Assertions.assertThrows(IllegalArgumentException.class, () -> {
49			new Contact("0123456789", "Matt4567890", "Jackson", "1111111111", "123 Any Street,
50		});
51	}
52
53	@Test
54	public void testContactFirstNameNotNull() {
55		Assertions.assertThrows(IllegalArgumentException.class, () -> {
56			new Contact("2", null, "lastName", "number", "address");
57		});
58	}
```

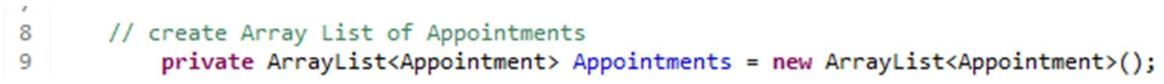
ii. Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were **effective** based on the coverage percentage?

My tests covered 100% of the class code. This means my tests used each method and constructor. The appropriate failures were asserted (null inputs when they weren't allowed) and successful operations were confirmed (attempting to get a task by ID after it was deleted returns null).

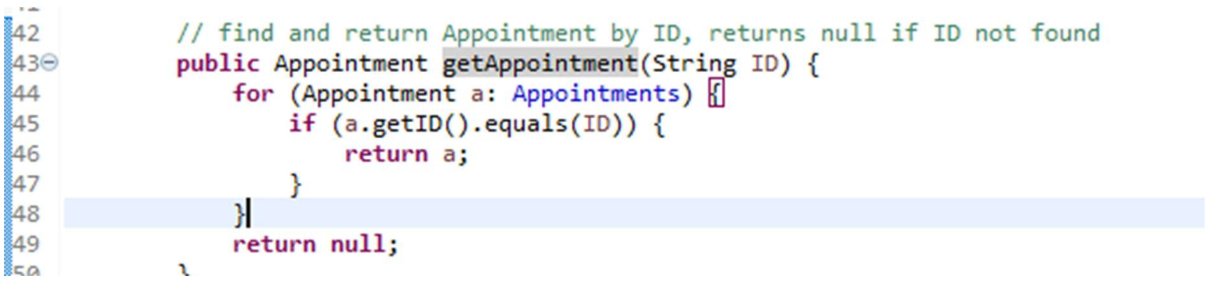
b. Describe your experience writing the JUnit tests.

i. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate.

I used private ArrayLists to store the various classes in memory as they were created in the Service classes. I then created a public getter for each class type that allowed me to access only the individual elements of the lists in my test classes, rather than pulling the entire list into my test class and accessing the elements from there.



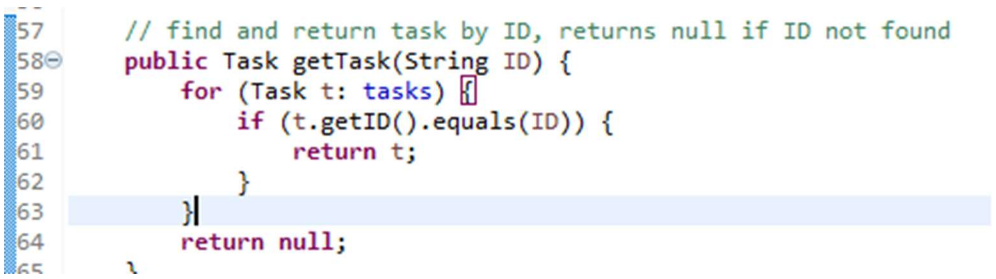
```
    // create Array List of Appointments
        private ArrayList<Appointment> Appointments = new ArrayList<Appointment>();
```



```
        // find and return Appointment by ID, returns null if ID not found
        public Appointment getAppointment(String ID) {
            for (Appointment a: Appointments) {
                if (a.getID().equals(ID)) {
                    return a;
                }
            }
            return null;
```

ii. How did you ensure that your code was **efficient**? Cite specific lines of code from your tests to illustrate.

One efficiency I used was to return immediately upon matching when searching for a class ID rather than continuing to iterate through the entire list. Only if the list was exhausted did it return null.



```java
    // find and return task by ID, returns null if ID not found
    public Task getTask(String ID) {
        for (Task t: tasks) {
            if (t.getID().equals(ID)) {
                return t;
            }
        }
        return null;
```

2. **Reflection**
    a. Testing Techniques
        i. What were the **software testing techniques** that you employed in this project? Describe their characteristics using specific details.

I used primarily automated, white-box, dynamic, unit testing for the milestones. But analysis and design of the tests also required static testing and a little manual testing to make sure my automated tests were functioning properly.

White-box testing lets you see into and analyze the code itself, as opposed to black-box testing where only inputs and outputs are known. Because I wrote it, I had access to the source code which allowed me to test specific restrictions like instantiating objects with null fields or improper lengths.

Dynamic testing is where the code is tested by running it and seeing the results. There was no visible output to the console or any kind of user interface except for the testing interface. This showed that the desired behavior was occurring like a deleted object returning null when searched for or an illegal argument exception thrown when a field was too long.

Unit testing ensures “that individual pieces of source code are tested to verify that the design and implementation for that unit” are correct. (Garcia, 2017) Each of the required objects had it’s own test service associated with it.

Analysis of requirements and examination of code fall under static testing. “Instead of executing the code, static testing is a process of checking the code and designing documents and requirements before it's run to find errors.” (Gillis, n.d.) Building tests to verify requirements requires such analysis.

To trouble shoot some of my automated tests, I had to resort to manually outputting statements to the console to verify portions of my test code were being reached as expected.

ii. What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details.

There were no performance standards for this project (for example: must be able to create n contacts in t seconds or search a list of n contacts in less than t seconds) so performance testing was not conducted. There was no user interface, either, so usability and user acceptance testing were not conducted. Usability is the “ease with which users can engage with the system.” Tools to measure mouse clicks, capture screenshots and other user feedback can be used to measure usability. Accessibility for users with disabilities is a consideration when measuring usability. (Hambling, Morgan, Samaroo, Thompson, & Williams, 2019)

iii. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.

White box testing is useful for looking at security from an inside “blue-team” perspective. Code can be reviewed for security best practices and vulnerabilities to known threats. Black box testing is helpful for complex systems where many pieces are integrated and the overall output of various inputs is measured. Black box testing can also be useful from a “red-team” perspective in that a malicious actor may not have much knowledge of the inner workings of a system. Dynamic system testing may not always be feasible for large projects with long compile times; smaller unit testing and some integration testing may suffice. Perhaps the most important type of testing is acceptance testing- ensuring the results meet the user needs and any regulatory requirements.

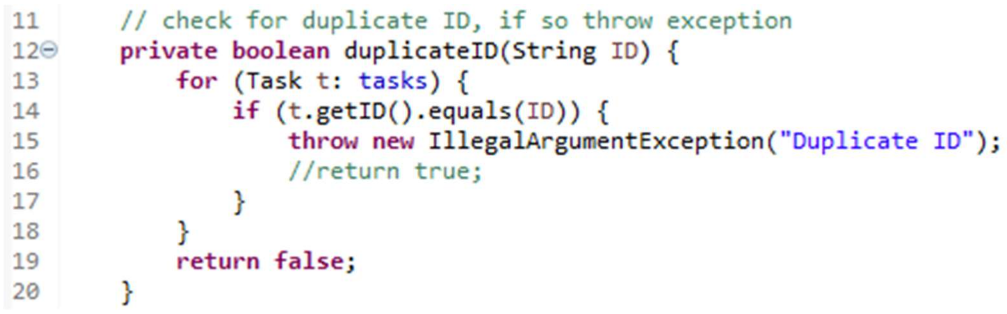
b. Mindset

i. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution**? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.

I used caution by taking care to fully understand the requirements of the software being developed. Understanding that pieces of the code had to build on each other to create a functioning unit was important. For example, I had to test and ensure that a Class could be created and added to a List properly before being able to delete it.

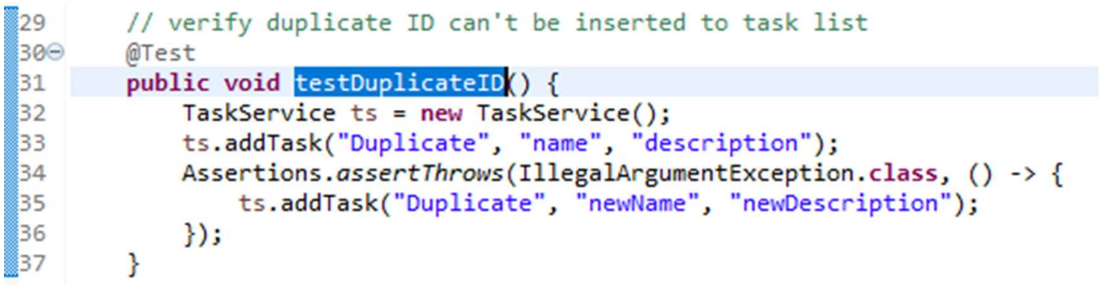
ii. Assess the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.

By not assuming the code worked properly and designing tests that would make the code fail, I avoided the confirmation bias of only writing working code. For example, when creating a new task, it can't have a duplicate ID. This is written in the creation of a Task which checks for a duplicate ID and returns an IllegalArgumentException if it finds one.



```java
    // check for duplicate ID, if so throw exception
    private boolean duplicateID(String ID) {
        for (Task t: tasks) {
            if (t.getID().equals(ID)) {
                throw new IllegalArgumentException("Duplicate ID");
                //return true;
            }
        }
        return false;
    }
```

When writing my tests, I made sure to test for this failure condiditon:



```java
    // verify duplicate ID can't be inserted to task list
    @Test
    public void testDuplicateID() {
        TaskService ts = new TaskService();
        ts.addTask("Duplicate", "name", "description");
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
            ts.addTask("Duplicate", "newName", "newDescription");
        });
    }
```

iii. Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

Being disciplined and committed to quality as a software engineer can save lives! Perhaps this assignment is not life or death, but setting a date in the past could have huge repercussions for a program determining GPS coordinates in a fighter jet. Especially if the code is going to be used in an open-source project, there's no telling how it will be reused in the future, so testing code to ensure it does what it's intended to do and writing it (and documenting it well) is critical. While clean, efficient, well documented code can avoid technical debt in the future, it can also build technical debt in the present by over-engineering code. A balance needs to be made between not cutting corners and not over-engineering a project.