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

My testing approach for ContactTest and ContactServiceTest was a bit *too* aligned with the requirements. I wrote one test for each requirement (testContactServiceReq1, testContactServiceReq2, etc). This was bad form- it would have been better to separate each part of the requirement out into separate tests. Based on the feedback I got there, I wrote TaskTest and TaskServiceTest to have separate tests for each sub-requirement (req1ContainsIdSuccess, req1IdOver10CharactersCreateFails, etc), and I did the same for Appointment and AppointmentService. There are additional tests in some cases where I added something not explicitly in the requirements and needed to test it.

* + 1. 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?

I achieved 100% test coverage for each file, which means that the tests cover every possible branch of code. This means that my tests are effective, because they cover every possible scenario.

* 1. Describe your experience writing the JUnit tests.
     1. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate.

1. I made sure that my code had plenty of comments, which would make it far easier for another developer to review.
2. I did do some things to avoid redundancies (such as having a function for each class to validate strings instead of having that code written over and over for each string), but I do think those redundancies could’ve been reduced if the assignment’s submission allowed for a SuperClass.
3. To my knowledge, all of the variables declared in my project were used.
4. I followed standard Java naming conventions for variable and class names. My test names were less standardized, but I made sure they all contained relevant information (like numbers for each requirement, and whether they were meant to succeed or fail).
   * 1. How did you ensure that your code was **efficient**? Cite specific lines of code from your tests to illustrate.

I used HashMaps for ContactService and TaskService:

private final HashMap<String, Task> taskList = new HashMap<>();

This made my code more efficient because the keys make it easier to find tasks and contacts by id. This does mean there’s a redundancy in data, but I think it was a better choice than having an ArrayList I would have to search more manually.

Also, I have a checkString() function in Task and Contact (with a modified version in Appointment):

private void checkString(String string, String varName, int upperBound){  
 if(string == null || string.length() > upperBound || string.length() < 1){  
 throw new IllegalArgumentException("Invalid " + varName);  
 }  
}

This function enables me to check each string more efficiently than if I made separate functions for checking strings for every variable, or checked them within other functions. If it were allowed in the assignment, I would have made a superclass for Task, Contact, and Appointment, and put the implementation for checkString there, with Appointment having a method overwrite. Unfortunately, the assignments asked for specific files, not a zipped project, so I was not able to implement that.

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

I used Unit Testing. With every class, I was unit testing: writing tests specific to the unit, and trying to cover as much functionality as possible. For example, for the Contact, Task, and Appointment classes, I tested to make sure that every required variable was contained in the object. While the Service classes’ functions do all interact with their base Service, I don’t consider the tests of the Service classes integration tests, because I was still isolating every individual function in the Service, and testing it on its own.

An example of one of my unit tests:

@Test  
public void req3ContainsDescriptionSuccess(){  
 //Does appointment contain a description?  
 Date currentDate = new Date(System.*currentTimeMillis*());  
 Appointment appointment = new Appointment("appoint",new Date(currentDate.getTime() + 10),"description");  
 *assertEquals*("description",appointment.getDescription());  
}

I would argue that all the tests counted as acceptance testing, as I was writing them to test specific assignment requirements, like Appointments not being able to be scheduled in the past (GeeksforGeeks, 2025).

@Test  
public void req2PastDateModifyFails(){  
 //Can the date be in the past?  
 Date currentDate = new Date(System.*currentTimeMillis*());  
 Date date = new Date(currentDate.getTime() - 1000);  
 Appointment appointment = new Appointment("appoint", new Date(currentDate.getTime() + 10), "description");  
 Exception exception = *assertThrows*(IllegalArgumentException.class, () -> {  
 appointment.setDate(date);  
 });  
 *assertEquals*(expectedDate, exception.getMessage());  
}

I used regression testing when making changes to the code by rerunning my tests and modifying the tests to reflect some changes (for example, parameters for a function changing). I reran my tests after I made any change to the code. After I removed the error handling for Task, I had to write new tests that were meant to catch errors:

//Can description be null?  
@Test  
public void req3DescriptionNullCreateFails(){  
 //Does task's name need to be not null?  
 Exception exception = *assertThrows*(IllegalArgumentException.class, () -> {  
 Task task = new Task("task",  
 "name",null);  
 });  
  
 *assertEquals*(expectedDescription, exception.getMessage());  
}

I used equivalence partitioning when writing the tests, because covering every single possible input would be ridiculously time consuming and unnecessary, because there were only so many branches.

* + 1. What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details.

There are many other software testing techniques, so I will focus some of the ones I would normally use that I wasn’t able to use for the project.

I like manual testing, especially since I work front-end. It is often faster than writing a test if you’re working with a UI. For the milestones, I was *not* working with a UI, or with any user interaction available at all. Therefore, manual testing wasn’t possible. UAT and Usability testing would be irrelevant at this stage for the same reason: there is no user experience yet.

Boundary testing is where the boundaries for each use case are tested (GeeksforGeeks, 2025). I did not use this in the project, but boundary testing could be used, as many variables have cutoff values for their size. I could test just below and exactly at the cutoff value (for example, testing 50 versus 51 characters for the description of one of the units).

The next step after unit testing would be integration testing, after I would code some main function that tied all of the class I have written together. I would want to test how using Contacts, Tasks, and Appointments (plus their respective services) together works.

I have not done any non-functional testing (such as performance testing) yet, because that was not required for code coverage.

* + 1. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.

Unit testing- Practical for testing individual units in the code. Testing interactions between different units would be Integration testing. Without unit testing, bugs could be discovered in integration testing that would be more difficult to identify.

Integration testing- Practical for testing interactions between units in the code. In this code, the units don’t interact much yet outside of interacting with their Service classes, so it’s not at a state where it’s ready for integration testing.

Acceptance testing- Practical for testing for specific business requirements. This should be performed after every user story’s completion professionally.

Regression testing- Necessary when updating code- you need to match sure that your code still works.

Equivalence partitioning- Cuts down on testing time by having one value for each possible scenario instead of testing every possible value.

Manual testing- Practical for testing when there is a UI to work with, as it’s fast, so you can go through many scenarios quickly.

Boundary testing- Practical for testing when there are specific, relevant boundary values.

Non-functional testing- Very necessary testing for things such as security and performance, as those may not have been tested upon completion of the user story.

* 1. Mindset
     1. 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 went for 100% coverage to make sure that I was covering every line and every possible scenario. That to me is cautious, when going only for 80% would be reckless. I used equivalence partitioning to cover every scenario, and ran regression tests every time I modified the code. I wanted to always have good code coverage.

One example of testing every branch is when I was testing the checkString() function. I had to check every single condition in the for loop to get complete coverage on that line.

If one doesn’t understand their code’s complexity and interrelationships, they won’t write good tests, and they won’t be able to cover everything. Without getting the complexity, they wouldn’t get every branch, and without getting the interrelationships, they would struggle, because they wouldn’t know what objects would need to be made before running a test.

* + 1. 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.

Every developer would like to think that their code is perfect. I think there will always be an element of unconscious bias in review of your own code. Thankfully, there are tools that help to mitigate it, like testing for code coverage. My concern was less with bias, and more with my blind spots. Code coverage testing helped me spot my blind spots so I could patch them. For example, it highlighted exception handling I needed to remove, reminded me that I needed to test if phone numbers were integers, and reminded me that I needed to test for both non-empty and non-null strings in Task.

As a software developer, bias is a concern when testing your own code, but in addition to code coverage, there should also be QA/QE and UAT to catch anything you didn’t spot. When I worked as a web dev, I would regularly get things flagged by QA to fix before pushing my code to stage, like formatting being off.

* + 1. 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.

The code you don’t cover could always be the code you miss. Letting code accumulate without tests creates an absurd amount of technical debt over time, because depending on the complexity of the code, testing could take twice as long as writing the code did in the first place. When I worked at Optum, only about 30% of the code was covered, so we were relying entirely on QA/QE to find bugs. This added an enormous amount of time to each ticket, as QA/QE always found multiple defects we needed to patch that could’ve been avoided.

When I get back into development work, I plan to test as I code to avoid accumulating technical debt, and to ensure that my code is far more up to standards before it makes it to QA.

References

GeeksforGeeks. (2025, July 26). *Software testing techniques*. https://www.geeksforgeeks.org/software-testing/software-testing-techniques/