1. Summary
2. My unit testing approach for each of the three features
3. I would say that my approach was very closely aligned to the software requirements for all of the three features (Contact Service, Task Service, and Appointment Service). I focused on making sure that I met all of the specified requirements for each feature which also included various constraints and functionalities. The following requirements, constraints, and functionalities are what I included, as well as thoroughly tested for in order to ensure that the software for all three features met the criteria they asked for:

* Contact Service

First off, the Contact class required a contact ID, first name, last name, phone number, and address. The constraints were not being null and only containing a certain character/digit length.

A screen shot of a computer program

Description automatically generated

While the ContactService required the addition, deletion, and updating of contacts.

A screen shot of a computer program

Description automatically generated

* Task Service

First off, the Task class required a task ID, name, and description, along with similar constraints as the Contact class as well.

A screen shot of a computer program

Description automatically generated

While the TaskService class just as the ContactService class, required the addition, deletion, and updating features but this time of tasks instead of contacts.

A screenshot of a computer program

Description automatically generated

* Appointment Service

First off, the Appointment class required an appointment ID, date, and description, along with similar constraints as both the Contact and Task classes as well.

A screen shot of a computer program

Description automatically generated

While the AppointmentService class required the addition and deletion of appointments, along with similar constraints as both the Contact and Task classes.

A screenshot of a computer program

Description automatically generated

1. The overall quality of my JUnit tests for all three features were pretty successful. I know and believe that my JUnit tests were effective based on the 79%+ coverage percentage for all three features. According to Salesforce, most successful projects should aim for a 75% statement coverage as opposed to a 100%. Obtaining a code coverage of 79%+ for all three features (Contact, Task, and Appointment Services) demonstrates that every possible code path and edge cases were tested. It also ensures that all features including the adding, deleting, updating, and retrieving of contacts, tasks, and appointments were validated as well as the date feature in the appointment service. Once again, this level of coverage (79%+) algins with the aim of creating robust and thorough tests. It also shows my strong commitment in delivering a reliable and well-tested application/software.
2. My experience writing the JUnit tests
3. I ensured that my code was technically sound by using and applying several strategies. This included making sure that rigorous validation checks were included and even comprehensively testing edge cases for all three features. An example that I can provide of this is:

A black background with white and red text

Description automatically generated

This validation check was included so I can ensure that the users input indeed met the features specified constraints. A second example I can provide is how I tested edge cases in order to endure that a contact, task, or appointment ID were not null or exceeded the maximum allowed length:

A black background with white text

Description automatically generated

1. I ensured that my code was efficient by mainly focusing on writing clear and concise methods from the very beginning. I also made sure to include comments throughout all of my code to thoroughly explain what is going on and what the code is doing. I also believe that I used the appropriate data structures for all three features, in this case a ‘HashMap’, so I can ensure that quick access and the manipulation of contacts, tasks, and appointments are seamless. An example of this:



According to both the research I made and previous knowledge from other classes, I decided to use a HashMap for the storage of contacts, tasks, and appointments because it has a O(1) time complexity. This O(1) time complexity makes it ideal for the adding, updating, and deletion operations that are both required and take place in all three features. I also tried to the best of my ability to ensure that my code had minimal to no redundancy. Below is an example of the ‘updateTask’ method which checks for null values before its fields are updated. This ensures that no unneeded operations take place and thus makes the operation perform much quicker.

A computer screen with white text

Description automatically generated

In conclusion, I feel that the following strategies that I implemented and followed is what made me ensure that my code was both technically sounds and efficient.

1. Reflection
2. Testing Techniques
3. The software testing techniques that I employed in this project for each of the milestones were:

* Contact Service

1. Unit Testing

Focused on small, isolated code sections which enabled the identification of any precise defects that could have been present. For example, the ‘testInvalidContactId’ method which is located in the ‘ContactTest.java’ file, helped ensure that the contact ID field was not null, and was composed of 10-characters.

1. Boundary Testing

Focused on ensuring that the contact fields constraints were correctly enforced by testing the edge cases. For example, I tested the contact IDs for all possible scenarios, such as having a contact ID that is exactly 10-characters long, shorter than 10-characters long, and longer than 10-characters long.

1. Exception Testing

Focused on handling invalid inputs correctly and properly by setting up the appropriate exceptions to be thrown. For example, an ‘IllegalArgumentException’ is thrown when a contact ID is longer than 10-characters long or is null.

* Task Service

1. Integration Testing

Focused on ensuring that the components not only worked as expected but also that an interaction between these multiple components indeed occurred. For example, I made sure through testing that tasks could be added, deleted, and updated from the Task Service.

1. Test-Driven Development (TDD)

Focused on writing the tests for TaskService.java before implementing any code in order to ensure that all requirements were thoroughly met. For example, writing the ‘addTask’ or ‘deleteTask’ methods before the actual code implementation took place.

* Appointment Service

1. Date Validation Testing

Focused on fulfilling the important requirement of ensuring that the appointment dates were not set in the past and that only future dates were accepted as valid. This was achieved by using ‘Date.before(new Date())’, as suggested by professor Tuft.

1. Coverage Analysis

Focused on measuring how much of the code was covered and tested by the unit tests in order to help determine what parts of the code remained untested.

1. The other software testing techniques that I did not use for this project were:

* Black-Box Testing

The functionality of an application is tested without having any insight of its internal structures or workings. As a result, the tester is only aware of what the inputs and outputs should be. These are its characteristics:

* Internal code structure is unknown
* User requirements and specifications
* Can be applied to all levels of testing such as unit, integration, system, and acceptance testing
* White-Box Testing

This testing technique on the contrary to Black-Box Testing, tests the internal structures or workings of the app and therefore the tester does have insight of its internal code structure. Its characteristics are the following:

* Knowledge of internal code structure is required
* Ensures that internal operations perform as expected and code is optimized
* Test covers code statements, branches, and paths
* System Testing

Tests that software is both complete and integrated. Here are its characteristics:

* Production environment is resembled when testing takes place
* Validates that the specified requirements are met by software
* Ensures that the whole system functions correctly
* Acceptance Testing

Tests if a system indeed satisfies its acceptance criteria. The following are its characteristics:

* Clients or end-users can perform this test
* Focuses on the required functionality of the business
* Determines the software’s readiness to be deployed

1. The practical uses and implications of each of the techniques that I discussed for different software development projects and situations are:

* Unit Testing

Helps in the early detection of bugs by verifying that things such as individual functions or methods are correct. This in the long run has the benefit of saving time, money, and resources. Two examples of its use case that can be provided are a banking and health portal app. The banking app would use unit tests to ensure that the user transaction calculations are accurate. While the health portal app would use unit tests to ensure that user input information such as insurance details, name, date of birth, etc., are not only accurate but also handled securely.

* Boundary Testing

Particularly useful in systems that contain critical input constraints. Again, the banking and health portal apps can serve as examples here. Boundary testing in the banking app would want to make sure that the user transactions taking place are valid if they are not then it is crucial for them to be flagged as invalid, not go through, and issue a warning. While the health portal app would use boundary testing for ensuring things such as phone number, insurance Id, and date of birth are all valid and lie within a reasonable range.

* Exception Testing

Crucial for robust error handling in systems that require it, as this not only enhances the user experience but also system reliability as well. Continuing with our banking and health portal apps as examples, ensuring that the correct information such as account numbers, funds, phone numbers, dates, etc., are crucial for a seamless and efficient experience.

* Integration Testing

Useful in a project that utilizes a microservice architecture, as it is crucial to ensure that the services within the microservice architecture communicate both properly and correctly. This as a result will help ensure that any issues are prevented from arising in a distributed system. Back to our banking and health portal app examples, a transfer of funds between accounts in a banking app, its account history, available funds, and restrictions, are all important and must all be working as a whole in order for a seamless transaction experience without any inconveniences or errors. Same for the health portal app, scheduling an appointment or the accessing of medical records should be both easy and seamless transactions.

* Test-Driven Development (TDD)

Valuable during an agile development environment. This is due to how rapidly requirements evolve. To address this, tests should be written before code in order to ensure that each developed feature indeed meets its specifications from the get-go. The banking app, account creation or closing for instance, can serve as examples. As for the health portal app, prescription refills can serve as an example since it can literally be a matter of life or death in some cases.

* Date validation Testing

Important and useful when an app uses scheduling or booking system features. Date validation testing ensures that the provided dates are valid and prevents any issues from taking place due to scheduling transactions in a past date (banking app) or appointment booking in past dates (health portal app).

* Black-Box Testing

Extensibly used in user acceptance testing (UAT). Mainly focuses on verifying the system meets user requirements without any prior knowledge of its internal code structure. User login and fund transfers can serve as examples of this in the banking app. While the health portal app examples include updating profile settings, and medical records.

* White-Box Testing

Useful for optimizing performance and ensuring the correctness of complex algorithms with prior knowledge of its internal code structure. In the banking app example, internal calculations for logging transactions or even interest calculations are important and must be precise. Data encryption and API security are also very crucial examples that apply to both the banking and health portal apps.

* System Testing

Validates that the entire system works as intended. An example of this in the baking app is testing that the transaction cycle indeed works as expected from logging in, fund transfers, transaction stamps, to logging off, are separate components that must all adequately work together. While testing in the health portal app might involve logging in, accessing medical records, viewing appointments, scheduling/cancelling appointments, prescriptions refills, etc., in order to ensure that the system is functioning seamlessly.

* Acceptance Testing

Validates that the software meets business requirements and is ready for deployment as well. The banking app example might implement this test by having actual users perform a transaction such as a fund transfer. While doing so they need to confirm that this feature meets their needs and works as expected. While in the health portal app, the same approach can be taken but this time it can test messaging between the patient and provider. Again, both the user and provider must confirm that this feature meets their requirements, needs, and works as expected in an intuitive matter.

1. Mindset
2. In acting as a software tester, I employed caution by thoroughly considering edge cases and potential failure points. An example of this can be seen during the Appointment Service, one of the edge cases I tested for was trying to set an appointment date that takes place in the past. This was important to test since allowing such condition would lead to incorrect data entry and thus potentially cause both confusion and errors in the system. It was important to appreciate the complexity and interrelationships of the code I was testing such as ensuring the immutability of all IDs because unintended side effects or bugs could arise if testing is not properly done and handled.
3. The ways that I tried to limit bias in my review of the code were by approaching testing with a curiosity mindset. I actively looked for potential weaknesses or any other areas of concern that could become or turn problematic. An example of this is that even though I felt confident in the logic that I implemented for the uniqueness of IDs, I still continued to write comprehensive tests in order to ensure that there were no duplicate IDs, nor could they be added. On the software developer side, yes, I believe that bias would be a concern if I were responsible for testing my own code to a certain extent. As with any other person, over confidence in the persons own code can lead to undertesting. I believe this bias can be mitigated by incorporating peer reviews and automated testing as this helps provide an external perspective and helps ensure that no assumptions are left unchecked.
4. I consider the importance of being disciplined in my commitment to quality as a software engineering professional is essential. Not only for this project but as a whole it is essential to keep this as a fundamental staple or building block in one’s arsenal. As this is what will keep our mindset, motivations, and guidelines on the right track. With that comes keeping code secure, making sure it remains ethic, and meets all requirements. It is important not to cut corners when it comes to writing or testing code because in the end it would only harm production as a whole when its already further down the line even if one doesn’t realize it. The reason being, that defects could be either introduced from the beginning or remain hidden in plain sight but only surface during production. As a result, causing not only a big headache but also far greater problems such as production delays, starting over, financial losses, and even exhaustion of resources. I plan to avoid technical debt as a practitioner in the field by prioritizing quality and thorough testing in order to ensure both a robust and maintainable codebase. An example of this is that I ensured that each service had a comprehensive coverage by incorporating the ‘assertAll’ function in order to verify multiple conditions in a single test, and thus, ensuring comprehensive coverage and preventing any potential technical debt from accumulating over time.

References

1. Gracia, B. (2017). *Mastering Software Testing with JUnit5*. Packt Publishing. Chapter 3:

JUnit 5 Standard.

1. Gracia, B. (2017). *Mastering Software Testing with JUnit5*. Packt Publishing. Chapter 6:

From Requirements to Test Cases.

1. Tutorialspoint. (n.d.). *JUnit Tutorial*. <https://www.tutorialspoint.com/junit/index.htm>