To what extent was your approach **aligned to the software requirements**?

* The first feature, creating contact objects, had five fields: ID, firstName, lastName, (Phone)Number, Address. So, there needed to, at minimum, be null, max length, and exceeding length tests for each field. The phone number field was an exception though because it had to be exactly 10 characters long, so it also needed a test to check for less than 10 characters as well. Updatable fields needed their own section for testing all fields except ID. Each update field test would have to check it updated correctly and that it followed the rules already previously established for the fields. There was also a section of testing required for deleting objects through their ID and to verify their non-existence.
* The second feature had three fields: ID, Name, and Description. Most of what was tested in the first feature applied to the second feature as well except having an exact character length in a field. “Name” and “Description” were the only updatable fields.
* The third feature had three fields as well: ID, Date, and Description. This feature was different from the others in that there were no updatable fields and that testing for the “Date” field required the use of “new Date()” to check if the date is in the past. All of this is the bare minimum for all three of the features and align to the provided requirements.
* I am confident that the Junit tests go beyond the minimum requirements and provide an effective amount of coverage when testing the code. Setting aside having done at least the minimum requirements, Included also is Boundary Value Analysis (BVA) testing (i.e., null, 10 characters, 11 characters, empty string, single character, current date, 1 millisecond in the past, and 1 millisecond in the future), Equivalence Partitioning in conjunction with BVA (i.e., Valid: 1 to 10 characters, Invalid: null, empty, more than 10 characters), Exception Testing to verify that the code is throwing the exception correctly, and even testing that there are no duplicate IDs due to the creation or deletion of objects.

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

* Technically sound code is reliable and maintainable. The code should meet the requirements provided and it should be organized and set up to handle code added or removed without error by the one who created it as well as those entrusted to maintain it. I followed the rules of being reliable and maintainable by, to name a few things, having functioning and error free code, following the Java naming conventions as best as possible (i.e., setAppointmentDate), allowed scalability by using HashMap data structures for fast O(1) searches, followed requirements exactly as asked (i.e., Phone number is exactly 10 characters, no more no less), test code checked not only what was required for user input but also tested different parts of the code (i.e., setters, constructors), made sure IDs couldn’t be updated (i.e., private final String appointmentId; final makes sure the appointment ID cannot be updated/changed), and separated validations to easily test and maintain (i.e., private void validateAppointmentId(String appointmentId), private void validateAppointmentDate(Date appointmentDate)).

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

* Efficient code is readable, consistent and modular. Because the people that write code are not always the ones maintaining it, it makes sense to name variables and functions as descriptively as possible so that others can understand that this function specifically does as it’s named. I made this code as efficiently as I could by, to name a few things, using descriptive variables and functions, using a scalable HashMap database (i.e., put, get, and containsKey operations), simple direct access to getters (i.e., public String getAppointmentId() { return appointmentId; }), used logical operators while validating (i.e., if (appointmentId == null || appointmentId.isEmpty() || appointmentId.length() > 10), if first check is true the rest of the checks are skipped), and utilized “assertAll” to bulk test even if one part fails (i.e., *assertAll*( () -> *assertThrows*(IllegalArgumentException.class, () -> appointment.setAppointmentDate(null)), () -> *assertThrows*(IllegalArgumentException.class, () -> appointment.setAppointmentDate(pastDate)),)

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

* Assertion Grouping – Using “assertAll” allowed grouping of different tests into different specific areas. The following techniques were used in multiple “assertAll” sections:
  + Boundary value analysis (BVA) – Testing minimum and maximum values as well beyond minimum and maximum values including checking if exceptions were thrown for invalid inputs.
  + Equivalence partitioning – Within the “assertAll” lines input tests were organized into valid and invalid classes (i.g., if “appointmentId” has to be between 2 and 10 characters, testing multiple valid number of characters is redundant as testing 3 characters is the same as testing 7 characters)
* Exception Testing – Exceptions, usually, give users a chance to correct their invalid inputs and continue the tasks they started. Nobody wants to restart a process they are in the middle of so testing that exceptions work is a must. Exceptions in turn should be descriptive and specific. When testing exceptions we shouldn’t test more than one method at a time (i.e.,   
  “throw new IllegalArgumentException("Invalid appointment date");” ,   
  *“assertThrows*(IllegalArgumentException.class,   
  () ->appointment.setAppointmentDate(null)))  
  () -> assertThrows(IllegalArgumentException.class,   
  () -> appointment.setAppointmentDate(pastDate))” )
* State-Based Testing – Testing whether a change in an objects state was accepted. This means testing that methods for adding and deleting objects work. (i.e., checking here that app1, app3, and app4 have existing fields but app2 doesn’t because it’s deleted, see below)  
    
  A screen shot of a computer program

  AI-generated content may be incorrect.

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

* Integration Testing – Integration testing checks that multiple units interact and communicate with each other as intended. We used in-memory data structures so there was no set external database to test from, but if had some shared database we could’ve written tests to verify items like appointments.
* Security Testing – Testing security means identifying weaknesses and reducing the chances of bad actors taking advantage. Our project is really simple without data being transported anywhere so there wasn’t any need.
* Acceptance Testing – Formal quality test to verify specified requirements are met. This test is usually done when there are items to check against like user stories but again ours is a simple code with simple requirements that isn’t ready for the end-user.

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

* Assertion Grouping:
  + Practical Uses:
    - Organized multiple assertions in one test, that tests all assertions even if one fails
    - Example: Testing an exception multiple times in one method
  + Implications:
    - It is useful when diagnosing when tests are failing multiple times in one test but only if testing names are descriptive enough
* Boundary value analysis (BVA):
  + Practical Uses:
    - Helps create systems with defined input ranges but requires input validation
    - It can help prevent errors and vulnerabilities (i.g., security vulnerabilities like SQL injection and cross-site scripting (XSS))
  + Implications:
    - CI/CD pipeline integration is supported
    - Tests for values that are just below, on, and just above the boundaries because errors near value boundaries are common
* Equivalence partitioning:
  + Practical Uses:
    - Streamlines the amount of testing by using the minimum amount valid and invalid inputs
    - Especially useful for input fields with multiple valid and invalid classes (e.g., age groups, 0-18, 19-21, 22-40, etc.)
  + Implications:
    - Saves time and resources
    - Needs understanding of input requirements or risk missing errors
* Exception Testing:
  + Practical Uses:
    - Verifies errors throw exceptions correctly
  + Implications:
    - Helps with debugging or correcting users
    - Ensures exceptions throw acceptable descriptive messages
* State-Based Testing:
  + Practical Uses:
    - Used for complex state transitions, such as online shopping, user authentication, and inventory management
  + Implications:
    - Ensures the system states are functioning as intended
    - Modeling is required because of the complexity to out undesirable state changes, bugs, and errors
* Integration Testing:
  + Practical Uses:
    - Necessity when multiple components or systems are involved
    - Validates that the now combined component or systems function together
  + Implications:
    - Used to catch data issues and interface mismatches
    - Needs an environment set up to test multiple systems (e.g., database to test)
* Security Testing:
  + Practical Uses:
    - Mandatory for any sensitive or personal data
    - Testing makes it that much harder for bad actors to exploit systems
  + Implications:
    - Need a security expert or third-party security tools
    - Not a “one and done” test, security should be checked often as part of maintaining code
* Acceptance Testing:
  + Practical Uses:
    - Validates requirements set by a business or user stories
    - This is something tested near the end of a project because it involves end-user testing
  + Implications:
    - Sometimes there are acceptance criteria beyond business and user stories (e.g., regulated industries such as healthcare and finance)

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 was the most cautious at the beginning of the project. I worried about meeting the requirements provided at the time and was trying to ensure the created code didn’t error out or do something unintended. I was also still establishing how I wanted to organize the tests and by worrying about all that I let some important tests fall through the crack. Afterwards I was less cautious and tried to understand how to get a high code coverage for testing.

After studying up and researching about testing, I found that I needed to create a plan for testing each method, so I didn’t miss a test or create copies of the same test. Creating a list of each test type and including the requirements let me organize the tests into sections uncomplicating what I thought before then to be really difficult. Writing out the tests also helped with creating modular code because if it was too interconnected and complex the tests in turn would be extreme.

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.

* I tried to limit bias in my code by trying to follow the process known as Test-Driven Development (TDD). Writing out the types of tests needed based on the requirements provided I could work backwards, sort of. My idea involves making a test plan then creating the code and finishing up with the test code after. If the code passed, I’d keep it or fix it if it didn’t. I didn’t have any problems scrapping entire methods and trying something else. I’m not skilled enough yet to fully apply the TDD process and work backwards from test code.

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.

* It is important not to cut corners in writing and testing code because it has been proven, using a cost escalation model, that the further in the software lifecycle you are, the more time and resources/money are required to fix it. Technical debt is the implied cost of not applying the best solution. The cost escalation model can apply to technical debt as well, showing that the further in the lifecycle you are the more costly it will be to rework. Time and money aren’t the only cost when doing rework. The reputation as you, an individual, or the company you work for may be at risk too.  
  This is why I know it is important to be committed to quality because it should be done right the first time to save time, resources, and reputation. To accomplish this, I would speak up about problems and offer solutions and ideas. If I cannot get through to my superiors or coworkers, I know to document my actions and the decisions of those around me and create paper trail to potentially prevent such failures in the future.