Faith Agepogu

CS 320

October 17, 2025

7-2 Project Two

For my project, I had to create three separate service components for a mobile app. They were called Contact Service, Task Service, and Appointment Service. For all three, my goal was to use JUnit to run tests to verify that the specific requirements were met. I believe the logic I used was thoroughly tested since all three projects ran with no errors, so 100% of the tests were correctly validated and functional. I made sure to add conditional features for how the app would behave if certain requirements were successful or invalid.

For Contact Service, I checked the code to make sure the contact could not be duplicated, so it could only be added once (and it returns false if the requirement is not met). I also made sure the required fields were valid, and those specific fields were: firstName, lastName, phone, and address. For the firstName and lastName fields, I set boundaries for the character limits. If firstName and lastName are too long, it would show an exception like this:

// Fail if the first name is too long

@Test(expected = IllegalArgumentException.class)

public void firstNameTooLongFail() {

new Contact("C002", "Somelongfirstnamegoeshere", "Agepogu", "4056174382", "630 NW 22nd St");

}

// Fail if the last name is too long

@Test(expected = IllegalArgumentException.class)

public void lastNameTooLongFail() {

new Contact("C003", "Faith", "Somelonglastnamegoeshere", "4056174382", "630 NW 22nd St");

}

I also made sure the phone number entry only contained valid numbers and characters, and that the character range was valid using these:

// Fail if the phone number includes characters that are not numbers

@Test(expected = IllegalArgumentException.class)

public void phoneNotDigitsFail() {

new Contact("C004", "Faith", "Agepogu", "abcdefghij", "630 NW 22nd St");

}

// Fail if the phone number is too short

@Test(expected = IllegalArgumentException.class)

public void phoneTooShortFail() {

new Contact("C005", "Faith", "Agepogu", "405617", "630 NW 22nd St");

}

For TaskService, I declared the character limits for the fields in public Task:

// Validate and create task

public Task(String id, String name, String desc) {

if (id == null || id.length() > 10) {

throw new IllegalArgumentException("Invalid id");

}

if (name == null || name.length() > 20) {

throw new IllegalArgumentException("Invalid name");

}

if (desc == null || desc.length() > 50) {

throw new IllegalArgumentException("Invalid description");

}

this.id = id;

this.name = name;

this.desc = desc;

}

That meant it would throw an error if my ID exceeded 10 characters. It would also throw an error if the name field exceeded 20 characters. Lastly it would throw an error if the desc field exceeded 50 characters. To further prove the requirements were met, I also added parts to add and delete tasks. I also made sure it could not duplicate tasks:

public class ContactServiceTest {

@Test

public void testAddContact() {

ContactService s = new ContactService();

Contact c = new Contact("C101", "Faith", "Agepogu", "4056174382", "630 NW 22nd St");

assertTrue(s.addContact(c));

}

@Test

public void testAddDuplicate() {

ContactService s = new ContactService();

Contact c1 = new Contact("C102", "Taylor", "Swift", "1234567890", "123 Oak Ave");

Contact c2 = new Contact("C102", "Miley", "Cyrus", "9876543210", "456 Pine St");

s.addContact(c1);

assertFalse(s.addContact(c2));

}

@Test

public void testDeleteContact() {

ContactService s = new ContactService();

Contact c = new Contact("C103", "Ariana", "Grande", "2223334444", "789 Maple Rd");

s.addContact(c);

assertTrue(s.deleteContact("C103"));

}

@Test

public void testUpdateFirstName() {

ContactService s = new ContactService();

Contact c = new Contact("C104", "Harry", "Styles", "5556667777", "321 Birch Blvd");

s.addContact(c);

s.updateFirstName("C104", "NewName");

assertEquals("NewName", s.getContact("C104").getFirstName());

}

}

For Appointment Service, I made sure it could not accept dates from the past:

// Fail if date is in the past

@Test

void testCreateAppointmentPastDateFail() {

// throw error if date is in the past

assertThrows(IllegalArgumentException.class, () ->

new Appointment("1", new Date(System.currentTime() - 1000), "desc"));

}

The code was also efficient because I used HashMaps for lookups and avoided nested loops. All the above is why I am confident the quality of my tests were good, and they were verified to have passed over 80% of the checks since there were no errors detected, so it was 100%. The above shows null checks and set limits to define conditions for failure and success, so only valid data would be recognized, and no duplicates would be registered.

Initially I struggled with setting up JUnit because I was unfamiliar with it before taking the course. Once I familiarized myself with it, I was able to practice to validate the functionality of the app I was creating tests for. I changed the names between my assignments since I didn’t like how vague they were initially, and I also took feedback to specify success/failure when naming, so it is clear what part of the code is meant to do what. I learned it is important to consider all scenarios when testing an app, like all possible ways it could be interacted with, so it lessens the chance of any security risks or poor design (where duplicates can be added). Also, I learned it is important to clarify failure/success in the code because if it is meant to be seen by others, it would be easier for them to read it. I also had to work on adding comments for readability, but for this essay, I removed most of the comments to keep it to the point. My submission files for Project One include all the comments, however.

For my testing techniques, I used unit testing. Unit testing was done by focusing on validating classes. I also used boundary-value testing when I set specific character limits, like the 10-digit phone number and 50-character description limit. I didn’t use integration or system testing since the project was mostly just JUnit testing and it wasn’t focused on making a fully functional app. In a larger system, however, I would use integration testing to make sure all 3 components (Contact, Task, Appointment) function well all together. It functions in isolated parts, but it might not function as intended if it is all combined, so for a full app, it would be important to test how they all interact simultaneously. One way I would test in a full app is to check if a new contact could be linked to a task or appointment without breaking the validation logic.

For my mindset during this project, I was cautious and considered a lot of what-if scenarios for how the code could be broken, which guided me to add the failure/success validations. It was easier to be thorough by assuming all parts of the code could potentially break, like for example random characters for a phone number input that wouldn’t make sense for a phone number, or what would happen if someone had a name that was too long. I tried to limit personal bias when code testing in this way. If I was making a code for myself, I could’ve been more relaxed about validation and consequently cause more room for errors since it would be easier to assume the code would just work as is. To be thorough, however, I used negative test cases and invalid data on purpose, which is how I was able to be unbiased about my methods. People may not interact with an app the same way I do, so it is important to try and test for multiple cases and scenarios. In a professional setting, rushing and leaving out validation checks in other projects could lead to time being wasted, and time wasted is also money wasted. It is good practice to think about all scenarios, so work is done in an efficient manner. Furthermore, this will make it easier to expand upon existing work.