7-2 Project Two: Summary and Reflections Report

Felie Marie Magbanua

Southern New Hampshire University

CS-320

Software Test, Automation QA

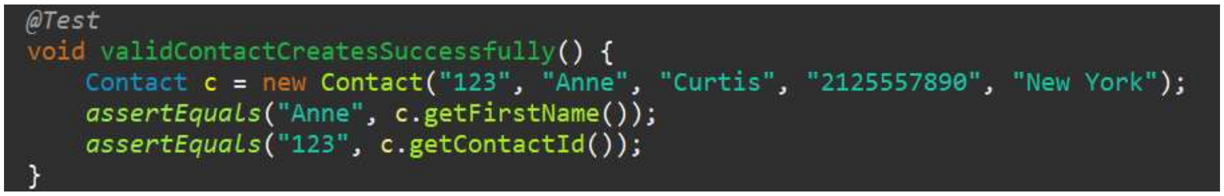
Sherif Antoun

October 19, 2025

This report summarizes my unit testing approach for the Contact, Task, and Appointment services developed in Project One, evaluates the testing techniques employed, and reflects on the mindset necessary for quality software engineering

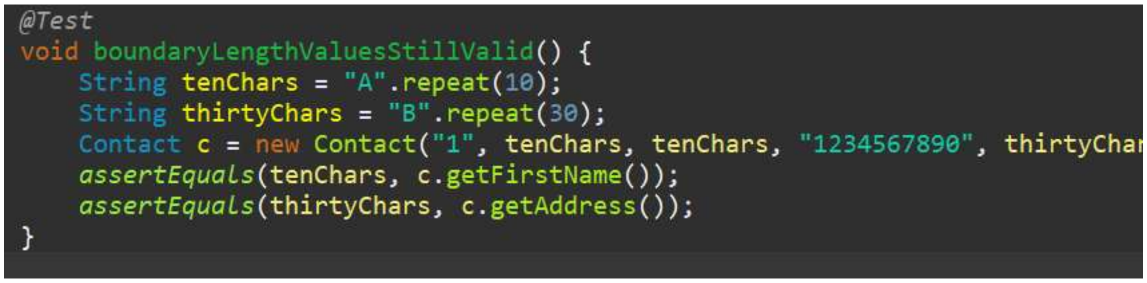
My unit testing approach directly validated each software requirement through targeted test cases. For Contact Service, requirements specified non-null, non-updatable IDs limited to 10 characters; names limited to 10 characters; phone numbers of exactly 10 digits; and addresses limited to 30 characters. I verified these constraints explicitly through tests confirming valid inputs were accepted and boundary values at maximum lengths were handled correctly.

File: ContactTest.java



```java
@Test
void validContactCreatesSuccessfully() {
    Contact c = new Contact("123", "Anne", "Curtis", "2125557890", "New York");
    assertEquals("Anne", c.getFirstName());
    assertEquals("123", c.getContactId());
}
```

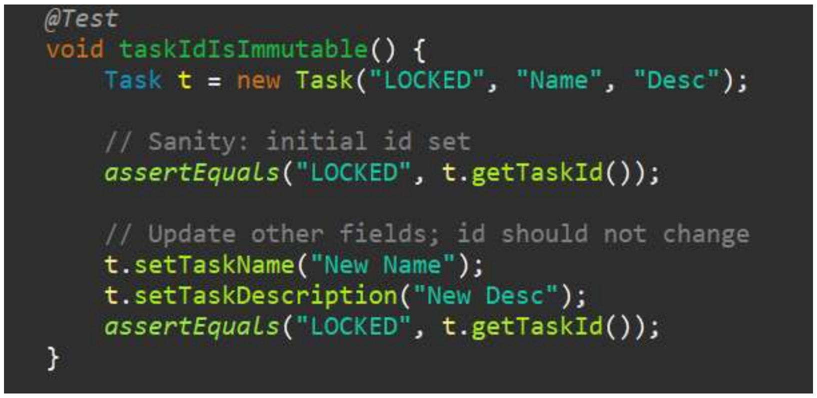
File: ContactTest.java



```java
@Test
void boundaryLengthValuesStillValid() {
    String tenChars = "A".repeat(10);
    String thirtyChars = "B".repeat(30);
    Contact c = new Contact("1", tenChars, tenChars, "1234567890", thirtyChar
    assertEquals(tenChars, c.getFirstName());
    assertEquals(thirtyChars, c.getAddress());
}
```

For Task Service, requirements mandated immutable IDs (10 characters max), names (20 characters max), and descriptions (50 characters max). My immutability test confirmed the ID remained constant despite other field updates. For Appointment Service, I verified dates cannot be in the past, and descriptions are limited to 50 characters using precise boundary validation testing dates one second in the past.

File: TaskTest.java

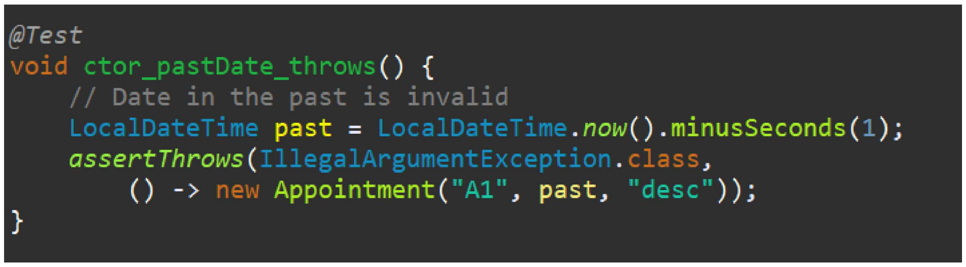


```java
@Test
void taskIdIsImmutable() {
    Task t = new Task("LOCKED", "Name", "Desc");

    // Sanity: initial id set
    assertEquals("LOCKED", t.getTaskId());

    // Update other fields; id should not change
    t.setTaskName("New Name");
    t.setTaskDescription("New Desc");
    assertEquals("LOCKED", t.getTaskId());
}
```

File: AppointmentTest.java

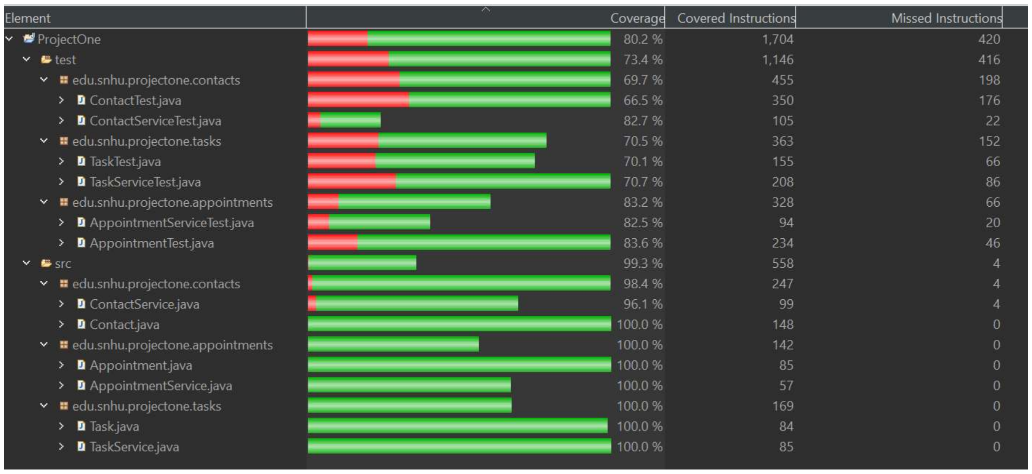


```java
@Test
void ctor_pastDate_throws() {
    // Date in the past is invalid
    LocalDateTime past = LocalDateTime.now().minusSeconds(1);
    assertThrows(IllegalArgumentException.class,
        () -> new Appointment("A1", past, "desc"));
}
```

This precision testing: rejecting dates even one second in the past, ensured the validation logic met the requirement at the finest granularity, not just at the day level.

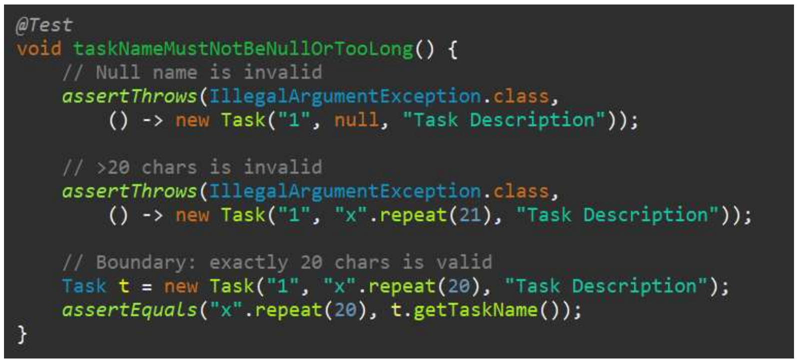
Each test was traceable back to a specific requirement, demonstrating complete alignment between specifications and test coverage. The service-layer tests extended this alignment by verifying operational requirements such as unique ID enforcement and proper exception handling for non-existent entities.

The test coverage report showed 80.2% overall project coverage with 100% coverage on all core domain classes. My tests explored boundary values, null inputs, and invalid formats systematically. I verified both acceptance of maximum-length values and rejection of excessive values through two-sided boundary testing (Garcia, 2017, pp. 283-285). Every required field was tested individually for null validation, and format validation confirmed rejection of non-numeric strings and wrong-length numbers. Test effectiveness was demonstrated when my tests caught defects during development, such as initial date validation issues leading to refactoring to LocalDateTime.



Element	Coverage	Covered Instructions	Missed Instructions
ProjectOne	80.2 %	1,704	420
test	73.4 %	1,146	416
edu.snhu.projectone.contacts	69.7 %	455	198
ContactTest.java	66.5 %	350	176
ContactServiceTest.java	82.7 %	105	22
edu.snhu.projectone.tasks	70.5 %	363	152
TaskTest.java	70.1 %	155	66
TaskServiceTest.java	70.7 %	208	86
edu.snhu.projectone.appointments	83.2 %	328	66
AppointmentServiceTest.java	82.5 %	94	20
AppointmentTest.java	83.6 %	234	46
src	99.3 %	558	4
edu.snhu.projectone.contacts	98.4 %	247	4
ContactService.java	96.1 %	99	4
Contact.java	100.0 %	148	0
edu.snhu.projectone.appointments	100.0 %	142	0
Appointment.java	100.0 %	85	0
AppointmentService.java	100.0 %	57	0
edu.snhu.projectone.tasks	100.0 %	169	0
Task.java	100.0 %	84	0
TaskService.java	100.0 %	85	0

File: TaskTest.java

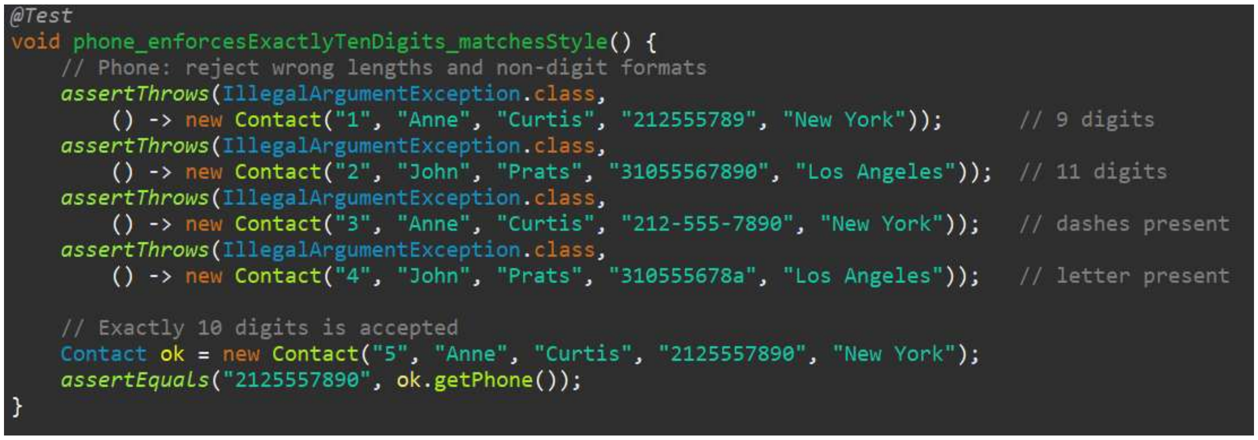


```java
@Test
void taskNameMustNotBeNullOrTooLong() {
    // Null name is invalid
    assertThrows(IllegalArgumentException.class,
        () -> new Task("1", null, "Task Description"));

    // >20 chars is invalid
    assertThrows(IllegalArgumentException.class,
        () -> new Task("1", "x".repeat(21), "Task Description"));

    // Boundary: exactly 20 chars is valid
    Task t = new Task("1", "x".repeat(20), "Task Description");
    assertEquals("x".repeat(20), t.getTaskName());
}
```

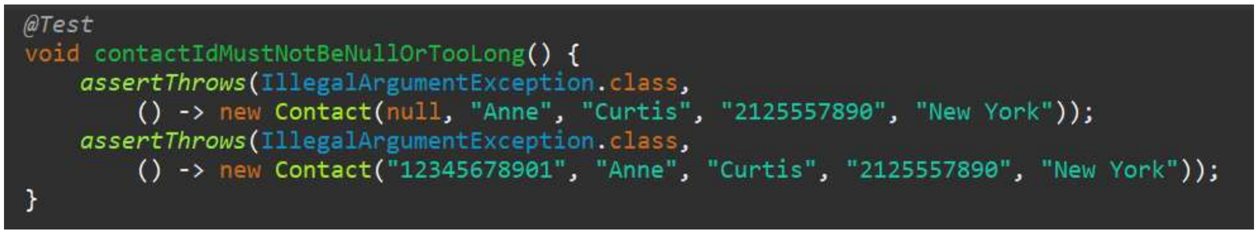
File: ContactTest.java



```java
@Test
void phone_enforcesExactlyTenDigits_matchesStyle() {
    // Phone: reject wrong lengths and non-digit formats
    assertThrows(IllegalArgumentException.class,
        () -> new Contact("1", "Anne", "Curtis", "212555789", "New York"));      // 9 digits
    assertThrows(IllegalArgumentException.class,
        () -> new Contact("2", "John", "Prats", "31055567890", "Los Angeles"));  // 11 digits
    assertThrows(IllegalArgumentException.class,
        () -> new Contact("3", "Anne", "Curtis", "212-555-7890", "New York"));   // dashes present
    assertThrows(IllegalArgumentException.class,
        () -> new Contact("4", "John", "Prats", "310555678a", "Los Angeles"));   // letter present

    // Exactly 10 digits is accepted
    Contact ok = new Contact("5", "Anne", "Curtis", "2125557890", "New York");
    assertEquals("2125557890", ok.getPhone());
}
```

File: ContactTest.java



```java
@Test
void contactIdMustNotBeNullOrTooLong() {
    assertThrows(IllegalArgumentException.class,
        () -> new Contact(null, "Anne", "Curtis", "2125557890", "New York"));
    assertThrows(IllegalArgumentException.class,
        () -> new Contact("12345678901", "Anne", "Curtis", "2125557890", "New York"));
}
```

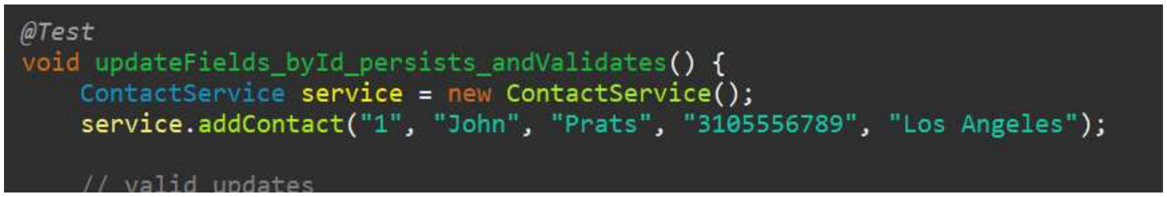
This test confirmed that passing null for the contact ID parameter caused the constructor to throw IllegalArgumentException rather than allowing a null ID to be stored or causing a NullPointerException later (Software Testing, 2019, pp. 22-53).

For the Appointment Service, I verified temporal validation with precision boundary testing. The use of minusSeconds(1) rather than minusDays(1) ensured the validation logic worked at the exact boundary, not just for obviously invalid dates (Garcia, 2017, pp. 283-285).

For the Task Service, I verified immutability by confirming that after modifying other fields, the task ID remained unchanged, proving the final keyword correctly prevented modification.

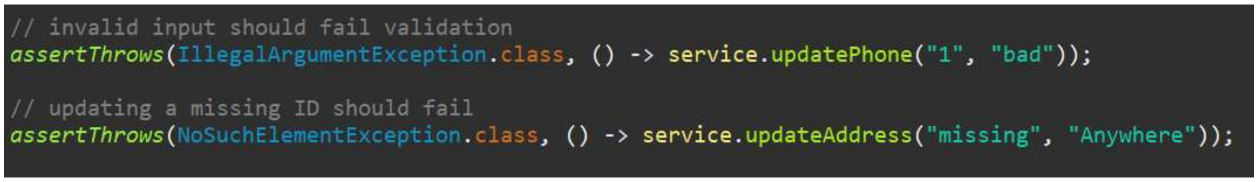
At the service layer, I verified defensive operations with:

File: ContactServiceTest.java



```java
@Test
void updateFields_byId_persists_andValidates() {
    ContactService service = new ContactService();
    service.addContact("1", "John", "Prats", "3105556789", "Los Angeles");

    // valid updates
```



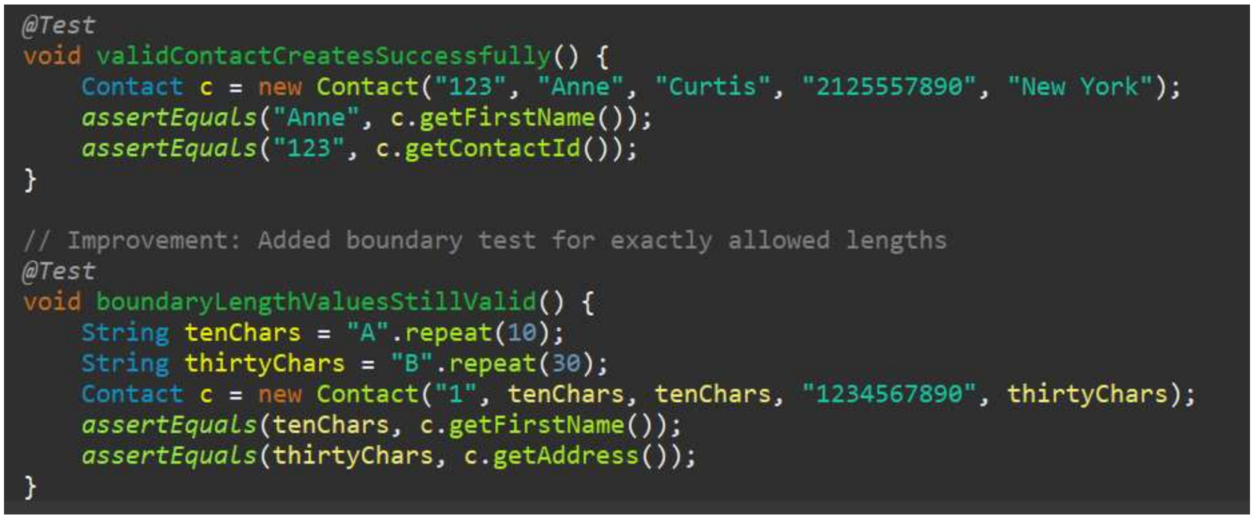
```java
// invalid input should fail validation
assertThrows(IllegalArgumentException.class, () -> service.updatePhone("1", "bad"));

// updating a missing ID should fail
assertThrows(NoSuchElementException.class, () -> service.updateAddress("missing", "Anywhere"));
```

I ensured technical soundness by explicitly verifying validation logic enforced all constraints and threw appropriate exceptions using JUnit 5's assertThrows() method (*JUnit - Test Framework*, n.d). Tests confirmed null IDs triggered IllegalArgumentException rather than causing NullPointerException later (Software Testing, 2019, pp. 22-53). At the service layer, I verified defensive operations ensuring invalid inputs failed validation and operations on non-existent entities threw meaningful exceptions rather than silently failing.

I ensured efficiency by keeping tests focused and isolated, with each test verifying a single behavior following the single responsibility principle (Garcia, 2017, pp. 286-287). Separating concerns made failures easier to diagnose. I used in-memory HashMap storage, eliminating external dependencies and allowing the entire suite to execute in milliseconds, enabling frequent testing without slowing productivity (Garcia, 2017, pp. 286-290). Descriptive test names made the suite self-documenting, reducing maintenance overhead.

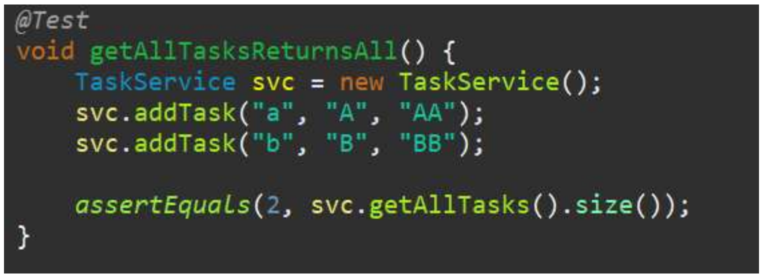
For example, rather than creating one large test that verified all Contact validation rules, I separated concerns:



```java
@Test
void validContactCreatesSuccessfully() {
    Contact c = new Contact("123", "Anne", "Curtis", "2125557890", "New York");
    assertEquals("Anne", c.getFirstName());
    assertEquals("123", c.getContactId());
}

// Improvement: Added boundary test for exactly allowed lengths
@Test
void boundaryLengthValuesStillValid() {
    String tenChars = "A".repeat(10);
    String thirtyChars = "B".repeat(30);
    Contact c = new Contact("1", tenChars, tenChars, "1234567890", thirtyChars);
    assertEquals(tenChars, c.getFirstName());
    assertEquals(thirtyChars, c.getAddress());
}
```

File: TaskServiceTest.java



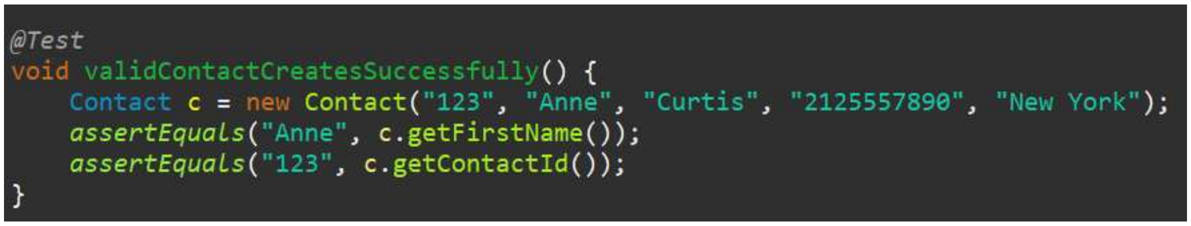
```java
@Test
void getAllTasksReturnsAll() {
    TaskService svc = new TaskService();
    svc.addTask("a", "A", "AA");
    svc.addTask("b", "B", "BB");

    assertEquals(2, svc.getAllTasks().size());
}
```

I employed three primary techniques: black-box testing, boundary value analysis, and equivalence partitioning.

Black-box testing validated behavior based solely on requirements without examining internal code (Hambling et al., 2019, pp. 87-121). I verified valid inputs produced correct outputs and invalid inputs threw exceptions.

File: ContactTest.java

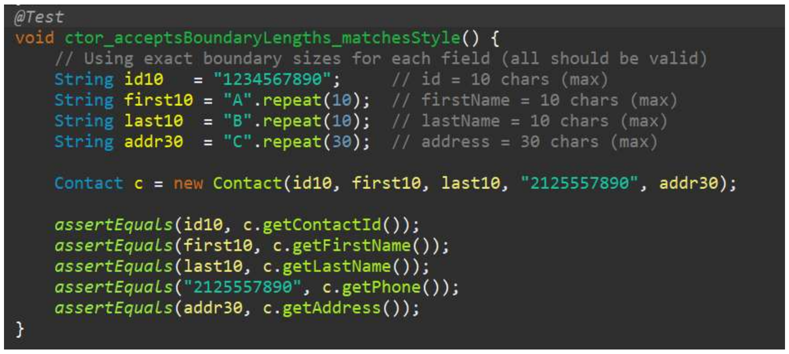


```java
@Test
void validContactCreatesSuccessfully() {
    Contact c = new Contact("123", "Anne", "Curtis", "2125557890", "New York");
    assertEquals("Anne", c.getFirstName());
    assertEquals("123", c.getContactId());
}
```

This approach ensured my tests remained focused on whether requirements were met rather than on implementation details.

Boundary value analysis examined behavior at input range edges (Garcia, 2017, pp. 283-285). I tested fields at maximum allowed lengths and values immediately beyond boundaries to catch off-by-one errors.

File: ContactTest.java



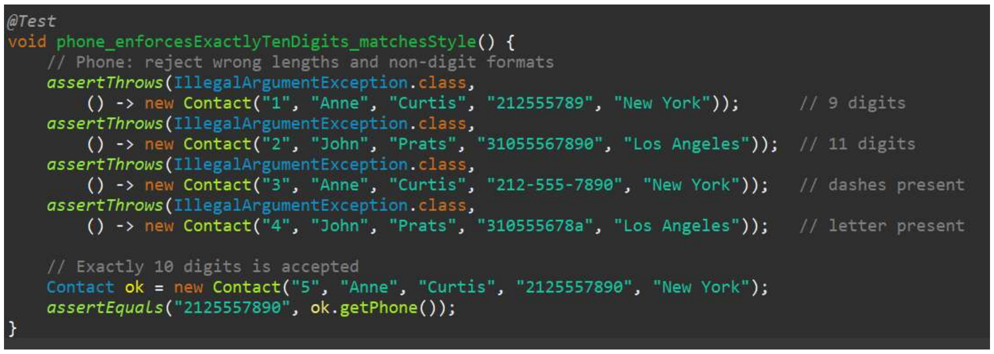
```java
@Test
void ctor_acceptsBoundaryLengths_matchesStyle() {
    // Using exact boundary sizes for each field (all should be valid)
    String id10   = "1234567890";      // id = 10 chars (max)
    String first10 = "A".repeat(10);  // firstName = 10 chars (max)
    String last10  = "B".repeat(10);  // lastName = 10 chars (max)
    String addr30  = "C".repeat(30);  // address = 30 chars (max)

    Contact c = new Contact(id10, first10, last10, "2125557890", addr30);

    assertEquals(id10, c.getContactId());
    assertEquals(first10, c.getFirstName());
    assertEquals(last10, c.getLastName());
    assertEquals("2125557890", c.getPhone());
    assertEquals(addr30, c.getAddress());
}
```

Equivalence partitioning divided inputs into classes where all members should be treated identically (Garcia, 2017, pp. 283-284). For phone validation, I identified three classes: valid 10-digit strings, non-numeric strings, and wrong-length strings, testing one representative from each class.

File: ContactTest.java



```java
@Test
void phone_enforcesExactlyTenDigits_matchesStyle() {
    // Phone: reject wrong lengths and non-digit formats
    assertThrows(IllegalArgumentException.class,
        () -> new Contact("1", "Anne", "Curtis", "212555789", "New York"));      // 9 digits
    assertThrows(IllegalArgumentException.class,
        () -> new Contact("2", "John", "Prats", "31055567890", "Los Angeles"));  // 11 digits
    assertThrows(IllegalArgumentException.class,
        () -> new Contact("3", "Anne", "Curtis", "212-555-7890", "New York"));   // dashes present
    assertThrows(IllegalArgumentException.class,
        () -> new Contact("4", "John", "Prats", "310555678a", "Los Angeles"));   // letter present

    // Exactly 10 digits is accepted
    Contact ok = new Contact("5", "Anne", "Curtis", "2125557890", "New York");
    assertEquals("2125557890", ok.getPhone());
}
```

There are other software testing techniques that I did not use for this project:

Integration testing examines interactions between multiple components (Software Testing, 2019, pp. 54-73). I did not employ this because each service operated independently with in-memory storage without external dependencies.

System testing evaluates the complete application against high-level requirements and user workflows (Software Testing, 2019, pp. 54-73). I did not perform this because the project scope was limited to individual service components without a user interface.

Regression testing verifies previously working functionality remains correct after code changes (Software Testing, 2019, pp. 54-73). While I re-ran tests after implementing feedback, I did not establish formal regression testing with version control and continuous integration.

Integration testing becomes essential for projects with multiple components, external dependencies, or distributed systems (Software Testing 2019, pp. 54-73). It often reveals issues invisible at unit level—race conditions, timeout handling, data serialization problems.

System testing is necessary for projects with end-users and complete workflows (Software Testing 2019, pp. 54-73). It evaluates non-functional requirements like performance, security, and usability.

Regression testing becomes mandatory in projects with frequent changes (Software Testing, pp. 54-73). It requires automation—manually re-running hundreds of tests is unsustainable.

As a software tester, I adopted a cautious, skeptical mindset, assuming code will fail unless proven otherwise (Software Testing 2019, pp. 22-53). I employed caution by systematically questioning every assumption and testing both success and failure paths equally. Rather than only verifying valid inputs were accepted, I confirmed invalid inputs were rejected, including past dates, null dates, and boundary cases.

I refused to assume similar code would behave identically without verification. Although Contact, Task, and Appointment followed similar validation patterns, I wrote explicit tests for each class. This proved valuable: Task descriptions allowed 50 characters while Contact addresses allowed 30, a difference that could have caused defects.

Appreciating code complexity and interrelationships was critical because simple requirements often have subtle interdependencies. The immutable ID requirement ripples through the system; service classes use IDs as HashMap keys, requiring immutability to maintain referential integrity. Domain classes enforce field-level validation while service classes enforce entity-level rules. Testing both validation layers separately ensured complete coverage.

Limiting bias required conscious effort against confirmation bias: the tendency to write tests confirming expectations rather than disproving assumptions (Software Testing, 2019, pp. 22-53). As both developer and tester, I was vulnerable because I knew implementation details and might unconsciously avoid areas where my code was weak.

I limited bias by writing tests expected to fail before implementing features, confirming tests actually detected the absence of validation. I systematically tested all input categories, covering every field individually for null validation even though I "knew" all fields had checks. This caught gaps during early development; I initially forgot null validation on one field, and comprehensive tests exposed this oversight.

I tested success and failure paths equally, balancing both to counteract bias toward proving code works rather than finding where it breaks. On the developer side, bias is significant when testing your own code because developers have emotional investment in implementation choices (Software Testing, 2019, pp. 22-53). When I initially implemented date handling with java.util.Date, my first instinct was questioning the tests when they revealed issues rather than accepting the implementation had problems.

Disciplined commitment to quality is essential because shortcuts compound into technical debt that becomes exponentially more expensive to address later (Garcia, 2017, pp. 286-290). Each shortcut creates potential failure modes that cascade through the system. If I had skipped boundary value testing on Contact address fields, defects might surface in production requiring database migration, data cleaning, and emergency deployment - far more expensive than writing the boundary test initially (Software Testing, 2019, pp. 22-53).

Cutting corners on test naming creates maintenance debt. Generic names mean future developers wouldn't understand what each test verified. My descriptive names mean failures immediately indicate which requirement is violated, drastically reducing debugging time.

To avoid technical debt in practice, I will maintain comprehensive test coverage from the start rather than promising to "add tests later." By writing tests alongside implementation, testing becomes part of Definition of Done. I will treat test code with the same quality standards as production code, ensuring tests remain readable and maintainable. I will advocate for automated testing in continuous integration - my fast-executing tests ran in milliseconds, making automated execution practical. I will resist pressure to skip testing to meet deadlines, recognizing thorough testing accelerates development long-term by catching errors when fixes are trivial (Software Testing, 2019, pp. 22-53). Finally, I will continuously refactor to maintain code quality, as incremental improvements supported by comprehensive tests prevent technical debt from becoming unmaintainable (Garcia, 2017, pp. 286-290).

References

Garcia, B. (2017). *Mastering Software Testing with JUnit 5 : Comprehensive Guide to Develop High Quality Java Applications*. https://research.ebsco.com/c/ix3dnl/search/details/hps3egvjeb?db=nlebk

JUnit - Test Framework. (n.d.). https://www.tutorialspoint.com/junit/junit_test_framework.htm

Software testing : An istqb-bcs certified tester foundation guide - 4th edition. (2019). BCS Learning & Development Limited.