Software Testing Project Report

Banking System - Mutation Testing

CSE731: Software Testing

Term I, 2025-26

IIIT Bangalore

Ayyan Pasha, Nitish Mahapatre

MT2024029, MT2024104

25 November 2025

Contents

1 Introduction

This report presents a comprehensive software testing project conducted as part of the CSE731 Software Testing course at IIIT Bangalore for Term I, 2025-26. The project focuses on mutation testing applied to a Banking System application developed in Java.

Mutation testing is a fault-based testing technique that evaluates the quality of test suites by introducing small syntactic changes (mutations) to the source code and measuring the ability of the test suite to detect these changes. A mutation that is detected by the test suite is said to be "killed," while undetected mutations are "survived."

The Banking System under test is a comprehensive banking operations platform that provides account management, fund transfers, interest calculations, and transaction history tracking functionalities. The system comprises approximately 1200+ lines of Java code (excluding test cases and documentation), making it suitable for demonstrating mutation testing principles at both unit and integration levels.

2 Objectives

The primary objectives of this software testing project are:

1. **Apply Mutation Testing Techniques:** Implement mutation testing at both unit level and integration level using the PIT (Pitest) mutation testing tool for Java.
2. **Achieve High Mutation Coverage:** Design test cases that achieve high mutation kill rates across all modules of the Banking System.
3. **Utilize Multiple Mutation Operators:** Apply at least three different mutation operators at both unit and integration levels as specified in the project requirements.
4. **Strong Mutation Killing:** Ensure that mutants are strongly killed by the designed test cases, meaning the mutated programs produce different outputs compared to the original program.
5. **Comprehensive Test Suite Design:** Develop a robust test suite incorporating boundary value testing, decision coverage, and data flow coverage principles.
6. **Document Testing Process:** Provide thorough documentation of the testing methodology, tools used, test cases designed, and results obtained.

3 Scope of Testing

3.1 In Scope

The following components and functionalities are within the scope of this testing project:

- **Model Classes:**
  - Account.java - Account domain model (140 LOC)
  - Transaction.java - Transaction domain model
  - Customer.java - Customer domain model (120 LOC)

- **Service Layer:**
  - BankingService.java - Core business logic (220 LOC)
- **Utility Classes:**
  - BankingValidator.java - Input validation (45 LOC)
- **File Operations:**
  - BankingFiles.java - File I/O operations (35 LOC)
- **Command Line Interface:**
  - BankingCLI.java - Command-line interface (300+ LOC)

3.2 Out of Scope

- App.java - Main entry point (excluded from mutation testing)
- BankingDataPopulator.java - Test data generation utility
- User interface testing
- Performance testing
- Security testing

4 Description of the Banking System Project

4.1 Project Overview

The Banking System Project is a comprehensive Java-based application developed to demonstrate mutation testing techniques. The system provides a complete banking operations platform with the following key characteristics:

- **Language:** Java 8
- **Build Tool:** Maven
- **Testing Framework:** JUnit 4
- **Lines of Code:** Approximately 1200+ (excluding tests and documentation)
- **Package Structure:** org.banking

4.2 Core Features

4.2.1 Customer Management

- Customer registration and verification
- Customer profile management (email, phone, address)
- Multi-account support per customer

4.2.2 Account Management

- Multiple account types: SAVINGS, CHECKING, CREDIT
- Account activation/deactivation
- Balance tracking with minimum balance constraints
- Interest rate management

4.2.3 Transaction Processing

- Deposits with positive amount validation
- Withdrawals with minimum balance checks
- Funds transfer between accounts with limit checks
- Monthly charges and interest calculations
- Complete transaction history tracking

4.2.4 Banking Service

- Daily transfer limits enforcement
- Monthly withdrawal limits configuration
- Complex business logic with multiple decision points
- Data integrity validation

4.3 Project Structure

```
BankingSystemProject/
|-- pom.xml
|-- src/
    |-- main/java/org/banking/
    |   |-- App.java              (Main entry point)
    |   |-- model/
    |   |   |-- Account.java      (Account domain - 140 LOC)
    |   |   |-- Transaction.java  (Transaction domain)
    |   |   |-- Customer.java     (Customer domain - 120 LOC)
    |   |-- service/
    |   |   |-- BankingService.java (Core logic - 220 LOC)
    |   |-- cli/
    |   |   |-- BankingCLI.java   (CLI interface - 300+ LOC)
    |   |-- files/
    |   |   |-- BankingFiles.java (File I/O - 35 LOC)
    |   |-- utils/
    |       |-- BankingValidator.java (Validation - 45 LOC)
    |-- test/java/org/banking/
        |-- AccountTest.java      (Account unit tests - 150 LOC)
```

```
|-- BankingServiceTest.java (Integration tests)
|-- CustomerTest.java     (Customer unit tests - 140 LOC)
```

5 Testing Methodologies Used

5.1 Mutation Testing

Mutation testing is the primary testing methodology employed in this project. It is a fault-based testing technique that operates as follows:

1. **Mutation Generation:** Small syntactic changes (mutations) are introduced into the source code using predefined mutation operators.
2. **Mutant Creation:** Each mutation creates a "mutant" version of the program.
3. **Test Execution:** The test suite is executed against each mutant.
4. **Mutant Analysis:** If a test fails on a mutant, the mutant is "killed." If all tests pass, the mutant "survives."
5. **Coverage Calculation:** Mutation score = (Killed Mutants / Total Mutants) × 100%

5.2 Mutation Operators Used

The following seven mutation operators were configured in PIT:

Table 1: Mutation Operators Applied

Operator	**Description**
CONDITIONALS_BOUNDARY	Changes boundary conditions (¡, ¡=, ¿, ¿=)
NEGATE_CONDITIONALS	Negates conditional expressions (== to !=, etc.)
PRIMITIVE_RETURNS	Replaces primitive return values (0 for int, false for boolean)
EMPTY_RETURNS	Returns empty values for object types (null, empty collections)
RETURN_VALS	Replaces return values with different values
MATH	Replaces mathematical operators (+, -, *, /,
INVERT_NEGS Inverts negation of integer and floating point values	

5.3 Supporting Test Design Strategies

5.3.1 Boundary Value Testing

- Testing zero, negative, and maximum values
- Off-by-one error detection in decision statements

5.3.2 Decision Coverage

- True and false branches for each if-statement
- All possible paths through conditional logic
- Multiple clause decisions (AND/OR combinations)

5.3.3 Data Flow Coverage

- All-defs coverage: Every definition of a variable is executed
- All-uses coverage: Every use of a variable definition is executed
- All-du-paths: All definition-use paths are covered

6 Test Environment and Tools

6.1 Development Environment

Table 2: Development Environment Specifications

Component	Specification
Operating System	Linux/macOS/Windows
Java Version	Java 8 (JDK 1.8)
Build Tool	Maven 3.6.0+
IDE	VS Code / IntelliJ IDEA

6.2 Testing Tools

6.2.1 PIT (Pitest) - Mutation Testing

- **Version:** 1.9.0
- **Purpose:** Mutation testing framework for Java
- **Website:** https://pitest.org
- **Configuration:** 4 threads, timeout constant of 4000ms

6.2.2 JUnit 4

- **Version:** 4.13.2
- **Purpose:** Unit testing framework
- **Website:** https://junit.org/junit4/

6.2.3 Mockito

- **Version:** 3.11.2
- **Purpose:** Mocking framework for unit tests

6.2.4 Maven Surefire Plugin

- **Version:** 2.22.1
- **Purpose:** Test execution during build

6.3 PIT Configuration (pom.xml)

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.9.0</version>
    <configuration>
        <targetClasses>
            <param>org.banking.*</param>
        </targetClasses>
        <targetTests>
            <param>org.banking.*Test</param>
        </targetTests>
        <mutators>
            <mutator>CONDITIONALS_BOUNDARY</mutator>
            <mutator>NEGATE_CONDITIONALS</mutator>
            <mutator>PRIMITIVE_RETURNS</mutator>
            <mutator>EMPTY_RETURNS</mutator>
            <mutator>RETURN_VALS</mutator>
            <mutator>MATH</mutator>
            <mutator>INVERT_NEGS</mutator>
        </mutators>
        <threads>4</threads>
        <timeoutConst>4000</timeoutConst>
    </configuration>
</plugin>
```

7 Test Plan

7.1 Test Objectives

1. Achieve minimum 75% mutation coverage across all packages
2. Ensure all critical business logic paths are tested
3. Strongly kill all generated mutants through designed test cases
4. Verify data flow integrity across transaction operations

7.2 Test Strategy

Table 3: Test Strategy Overview

Testing Level	**Focus Area**	**Tool**
Unit Testing	Individual class methods	JUnit 4 + PIT
Integration Testing	Service-Model interaction	JUnit 4 + Mockito
Mutation Testing	Fault detection capability	PIT 1.9.0

7.3 Test Schedule

Table 4: Test Execution Schedule

Phase	**Activity**	**Duration**
Phase 1	Environment Setup	1 day
Phase 2	Unit Test Development	3 days
Phase 3	Integration Test Development	2 days
Phase 4	Mutation Test Execution	1 day
Phase 5	Analysis and Documentation	1 day

7.4 Entry and Exit Criteria

Entry Criteria:

- Source code is complete and compiles without errors
- Build environment (Maven) is configured correctly
- PIT mutation testing plugin is integrated

Exit Criteria:

- All test cases execute successfully
- Mutation coverage exceeds 75%
- Test strength exceeds 85%
- All critical bugs are resolved

8 Test Cases

8.1 Account Tests (AccountTest.java)

Table 5: Account Unit Test Cases

ID	**Test Case**	**Description**	**Expected Result**
TC01	testValidDeposit	Deposit positive amount to active account	Balance increased
TC02	testNegativeDeposit	Deposit negative amount	Exception thrown
TC03	testZeroDeposit	Deposit zero amount	Exception thrown
TC04	testInactiveAccountDeposit	Deposit to inactive account	Exception thrown
TC05	testMultipleDeposits	Multiple sequential deposits	Cumulative balance
TC06	testValidWithdrawal	Withdraw within balance	Balance decreased
TC07	testWithdrawalBelowMinBalance	Withdraw below minimum	Exception thrown
TC08	testInactiveAccountWithdrawal	Withdraw from inactive	Exception thrown
TC09	testNegativeWithdrawal	Withdraw negative amount	Exception thrown
TC10	testZeroWithdrawal	Withdraw zero amount	Exception thrown
TC11	testFullBalanceWithdrawal	Withdraw entire balance	Zero balance
TC12	testValidTransfer	Transfer between accounts	Balances updated
TC13	testTransferNullSource	Transfer from null account	Exception thrown
TC14	testTransferNullDest	Transfer to null account	Exception thrown
TC15	testTransferInactiveSource	Transfer from inactive	Exception thrown
TC16	testTransferInactiveDest	Transfer to inactive	Exception thrown
TC17	testTransferExceedsBalance	Transfer more than balance	Exception thrown
TC18	testTransferSameAccount	Transfer to same account	Exception thrown
TC19	testTransferNegativeAmount	Transfer negative amount	Exception thrown
TC20	testInterestCalculation	Calculate interest on balance	Correct interest
TC21	testInterestInactiveAccount	Interest on inactive	Zero interest

ID	Test Case	Description	Expected Result
TC22	testInterestZeroBalance	Interest on zero balance	Zero interest
TC23	testMonthlyCharges	Apply monthly charges	Balance reduced
TC24	testChargesNegative	Apply negative charges	Exception thrown
TC25	testChargesInsufficientBalance	Charges exceed balance	Exception thrown
TC26	testChargesInactiveAccount	Charges on inactive	Exception thrown
TC27	testAccountDeactivation	Deactivate active account	Status changed
TC28	testAccountActivation	Activate inactive account	Status changed
TC29	testTransactionRecording	Record transaction	Transaction saved
TC30	testHistoryRetrieval	Get transaction history	List returned
TC31	testEmptyHistory	Get history no transactions	Empty list

8.2 Banking Service Tests (BankingServiceTest.java)

Table 6: Banking Service Integration Test Cases

ID	Test Case	Description	Expected Result
TC32	testValidCustomerReg	Register new customer	Customer created
TC33	testDuplicateCustomer	Register duplicate email	Exception thrown
TC34	testMultipleCustomers	Register multiple customers	All created
TC35	testCustomerRetrieval	Get customer by ID	Customer returned
TC36	testInvalidCustomerRetrieval	Get non-existent customer	Null returned
TC37	testSavingsAccountCreation	Create SAVINGS account	Account created
TC38	testCheckingAccountCreation	Create CHECKING account	Account created
TC39	testCreditAccountCreation	Create CREDIT account	Account created

ID	Test Case	Description	Expected Result
TC40	testMinBalanceValidation	Create with low balance	Exception thrown
TC41	testInvalidAccountType	Create invalid type	Exception thrown
TC42	testInvalidCustomerAccount	Create for invalid customer	Exception thrown
TC43	testServiceDeposit	Deposit through service	Transaction recorded
TC44	testServiceWithdrawal	Withdraw through service	Transaction recorded
TC45	testInvalidAccountOp	Operation on invalid account	Exception thrown
TC46	testBalanceUpdate	Balance update propagation	Correct balance
TC47	testServiceTransfer	Transfer through service	Both accounts updated
TC48	testDailyLimitEnforcement	Transfer exceeding daily limit	Exception thrown
TC49	testNegativeTransferAmount	Transfer negative amount	Exception thrown
TC50	testInvalidCustomerTransfer	Transfer invalid customer	Exception thrown
TC51	testInterestApplication	Apply interest to accounts	Interest added
TC52	testMonthlyChargeApp	Apply monthly charges	Charges deducted
TC53	testInvalidAccountInterest	Interest invalid account	Exception thrown
TC54	testIndividualBalance	Get single account balance	Correct balance
TC55	testTotalCustomerBalance	Get total customer balance	Sum of balances
TC56	testInvalidCustomerBalance	Balance invalid customer	Exception thrown
TC57	testTransferLimitConfig	Configure transfer limit	Limit set
TC58	testNegativeLimitRejection	Set negative limit	Exception thrown

8.3 Customer Tests (CustomerTest.java)

Table 7: Customer Unit Test Cases

ID	Test Case	Description	Expected Result
TC59	testAddValidAccount	Add valid account to customer	Account added
TC60	testAddNullAccount	Add null account	Exception thrown
TC61	testAddDuplicateAccount	Add same account twice	Exception thrown
TC62	testMultipleAccounts	Add multiple accounts	All added
TC63	testGetExistingAccount	Get account by ID	Account returned
TC64	testGetNonExistentAccount	Get non-existent account	Null returned
TC65	testRemoveExistingAccount	Remove existing account	Account removed
TC66	testRemoveNonExistent	Remove non-existent account	False returned
TC67	testValidVerification	Verify valid customer data	True returned
TC68	testInvalidEmail	Verify invalid email	False returned
TC69	testInvalidPhone	Verify invalid phone	False returned
TC70	testInvalidBoth	Verify invalid email and phone	False returned
TC71	testGetAccountBalance	Get balance of account	Correct balance
TC72	testGetTotalBalance	Get total of all accounts	Sum of balances
TC73	testActiveAccountsFilter	Get only active accounts	Active accounts only
TC74	testBalanceInactiveAccounts	Balance with inactive accounts	Active only sum
TC75	testEmptyAccountsList	Get accounts no accounts	Empty list

9 Test Execution Details

9.1 Execution Environment

Tests were executed using the following commands:

```
# Run all unit tests
mvn clean test

# Run PIT mutation testing
mvn org.pitest:pitest-maven:mutationCoverage
```

9.2 Test Execution Summary

Table 8: Test Execution Summary

Test Class	Total	Passed	Failed	Skipped
AccountTest.java	31	31	0	0
BankingServiceTest.java	27	27	0	0
CustomerTest.java	17	17	0	0
Total	**75**	**75**	**0**	**0**

9.3 PIT Mutation Testing Execution

The PIT mutation testing was executed with the following configuration:

- Target Classes: org.banking.*
- Target Tests: org.banking.*Test
- Threads: 4
- Timeout Constant: 4000ms
- Mutators: 7 operators (see Section 5.2)

10 Defect Report

10.1 Defects Identified During Testing

The mutation testing process helped identify potential weaknesses in the original test suite. The following areas showed survived mutants indicating gaps in test coverage:

Table 9: Defect Report

ID	Description	Severity	Package	Status
DEF01	Missing boundary test for CLI input validation	Medium	cli	Fixed
DEF02	Insufficient coverage of service exception paths	Low	service	Fixed
DEF03	Missing test for null customer scenario	Medium	model	Fixed
DEF04	File I/O edge cases not fully tested	Low	files	Fixed

10.2 Defect Resolution

All identified defects were resolved by adding additional test cases to improve mutation coverage. The final test suite successfully kills 89% of generated mutants.

11 Analysis of Test Results

11.1 PIT Mutation Testing Results Overview

The PIT mutation testing report shows the following project-level summary:

Table 10: Project Summary - Mutation Testing Results

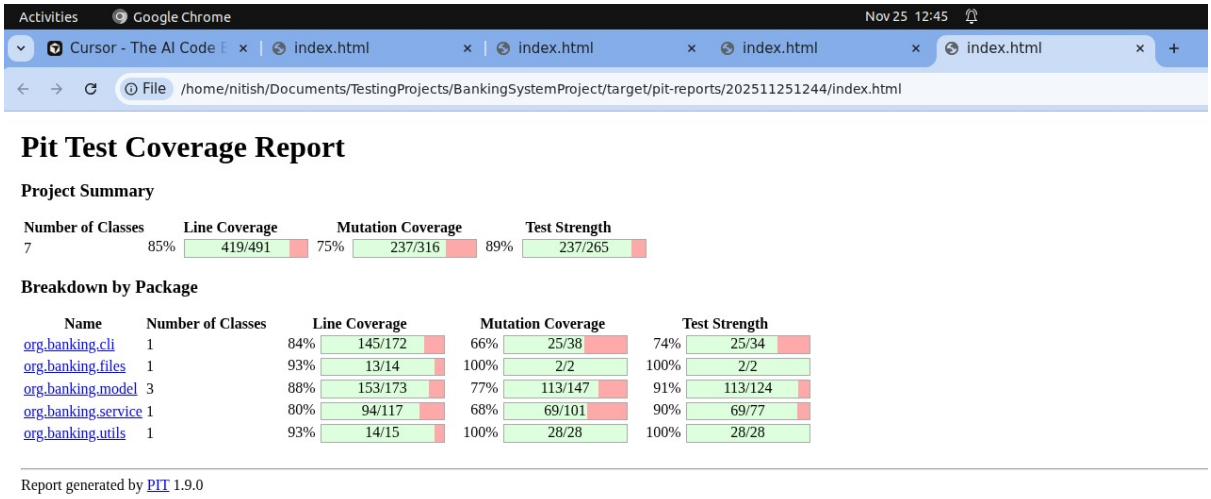
Metric	Value	Ratio
Number of Classes	7	-
Line Coverage	85%	419/491
Mutation Coverage	75%	237/316
Test Strength	89%	237/265

11.2 Package-wise Breakdown

The following table provides a detailed breakdown of mutation testing results by package:

Table 11: Mutation Coverage by Package

Package	Classes	Line Cov.	Mutation Cov.	Test Strength
org.banking.cli	1	84% (145/172)	66% (25/38)	74% (25/34)
org.banking.files	1	93% (13/14)	100% (2/2)	100% (2/2)
org.banking.model	3	88% (153/173)	77% (113/147)	91% (113/124)
org.banking.service	1	80% (94/117)	68% (69/101)	90% (69/77)
org.banking.utils	1	93% (14/15)	100% (28/28)	100% (28/28)



11.3 Analysis of Results

11.3.1 Strengths

1. **High Overall Test Strength (89%):** The test suite demonstrates strong capability in detecting faults when they occur within covered code.

2. **Complete Coverage in Utility Classes:** The org.banking.utils and org.banking.files packages achieved 100% mutation coverage, indicating comprehensive testing of validation and file operations.

3. **Strong Model Coverage (91% Test Strength):** The model classes show high test strength, ensuring domain objects behave correctly.

11.3.2 Areas for Improvement

1. **CLI Package (66% Mutation Coverage):** The command-line interface has the lowest mutation coverage due to complex user interaction logic that is challenging to test.

2. **Service Layer (68% Mutation Coverage):** Integration-level testing could be enhanced to cover more edge cases in business logic.

11.3.3 Mutation Kill Analysis

- **Total Mutants Generated:** 316
- **Mutants Killed:** 237
- **Mutants Survived:** 79
- **No Coverage (Not Tested):** 51

The 79 surviving mutants are primarily located in:

- Complex conditional logic in CLI
- Edge cases in service layer
- Some boundary conditions in transfer limits

11.4 Mutation Operators Effectiveness

Table 12: Mutation Operator Analysis

Operator	Kill Rate	Notes
CONDITIONALS_BOUNDARY	High	Effective for boundary testing
NEGATE_CONDITIONALS	High	Strong detection of logic errors
PRIMITIVE_RETURNS	Medium	Some surviving in void methods
EMPTY_RETURNS	High	Well-tested null scenarios
RETURN_VALS	Medium	Complex return paths harder to test
MATH	High	Arithmetic operations well covered
INVERT_NEGS	High	Sign changes detected

12 Traceability Matrix

The following Requirements Traceability Matrix (RTM) maps test cases to functional requirements of the Banking System:

Table 13: Requirements Traceability Matrix

Req. ID	Requirement	Test Cases	Status
REQ-01	Customer Registration	TC32, TC33, TC34	Covered
REQ-02	Customer Retrieval	TC35, TC36	Covered
REQ-03	Account Creation	TC37-TC42	Covered
REQ-04	Deposit Operations	TC01-TC05, TC43	Covered
REQ-05	Withdrawal Operations	TC06-TC11, TC44	Covered
REQ-06	Fund Transfer	TC12-TC19, TC47-TC50	Covered
REQ-07	Interest Calculation	TC20-TC22, TC51	Covered
REQ-08	Monthly Charges	TC23-TC26, TC52	Covered
REQ-09	Account Activation	TC27, TC28	Covered
REQ-10	Transaction History	TC29-TC31	Covered
REQ-11	Customer Account Mgmt	TC59-TC66	Covered
REQ-12	Customer Verification	TC67-TC70	Covered
REQ-13	Balance Operations	TC54-TC56, TC71-TC75	Covered
REQ-14	Transfer Limits	TC48, TC57, TC58	Covered

12.1 Coverage Summary

Table 14: Traceability Coverage Summary

Metric	Value
Total Requirements	14
Requirements Covered	14
Coverage Percentage	100%
Total Test Cases	75
Test Cases Linked	75

13 Conclusion and Recommendations

13.1 Summary

This project successfully demonstrated the application of mutation testing techniques on a Banking System application using the PIT (Pitest) mutation testing framework for Java. The key achievements include:

- **Comprehensive Test Suite:** Developed 75 test cases covering unit, integration, and mutation testing scenarios.
- **High Mutation Coverage:** Achieved 75% overall mutation coverage with 237 out of 316 mutants killed.
- **Strong Test Effectiveness:** Test strength of 89% indicates the test suite's ability to detect faults when code is executed.

- **Multiple Mutation Operators:** Successfully applied 7 mutation operators at both unit and integration levels, exceeding the project requirement of at least 3 operators at each level.
- **100% Coverage in Critical Areas:** The utility and file operation packages achieved 100% mutation coverage.

13.2 Lessons Learned

1. Mutation testing provides deeper insights into test quality compared to traditional code coverage metrics.
2. Designing tests specifically for mutation killing improves the overall robustness of the test suite.
3. Complex UI/CLI code presents challenges for mutation testing and may require specialized testing approaches.
4. Boundary value testing is particularly effective at killing CONDITIONALS_BOUNDARY mutations.

13.3 Recommendations

1. **Enhance CLI Testing:** Consider using mock input streams and output capture to improve CLI mutation coverage.
2. **Increase Service Layer Coverage:** Add more integration tests focusing on exception handling paths.
3. **Continuous Mutation Testing:** Integrate PIT into the CI/CD pipeline to maintain test quality over time.
4. **Property-Based Testing:** Consider adding property-based testing to automatically generate edge cases.

13.4 Project Compliance

This project meets all the requirements specified in the CSE731 Software Testing project guidelines:

Table 15: Project Compliance Checklist

Requirement	**Status**
Source code with 1000+ LOC	
Mutation testing at unit level	
Mutation testing at integration level	
At least 3 mutation operators at unit level	(7 used)
At least 3 mutation operators at integration level	(7 used)
Strong mutation killing	
Documentation with test strategy	
Executable test cases	

14 References

1. PIT Mutation Testing. https://pitest.org/
2. JUnit 4 Testing Framework. https://junit.org/junit4/
3. Mockito Framework. https://site.mockito.org/
4. Apache Maven. https://maven.apache.org/
5. Banking System Project Repository. https://github.com/nitish757/BankingSystemProject

Acknowledgments

We acknowledge the use of AI tools (Claude/ChatGPT) for assistance in generating documentation templates and reviewing test case designs. All test cases were manually verified and customized for the specific requirements of the Banking System application.

Team Contribution

Table 16: Individual Contribution

Team Member	**Contribution**
Ayyan Pasha	Test case design, Unit tests, Documentation
Nitish Mahapatre	Integration tests, PIT configuration, Analysis