VIVA PREPARATION GUIDE - Mutation Testing Project

Project Overview

Project Name: Banking System Mutation Testing Study
Team Members: Deepanshu Saini (MT2024039) & Nakul Siwach (MT2024096)
Mutation Score: 79% (408/518 mutations killed)
Line Coverage: 96% (373/387 lines)
Total Tests: 168 tests across 5 test classes

---

1. PROJECT STRUCTURE & COMPONENTS

1.1 What did we build?

We built a banking application with 5 main classes:

1. **Calculator.java** (58 lines)
    - Basic arithmetic operations: add, subtract, multiply, divide, abs, max
    - Foundation class used by other components
    - Handles division by zero validation
2. **LoanCalculator.java** (197 lines)
    - EMI calculation using compound interest formula
    - Loan eligibility checks based on income and EMI ratio
    - Prepayment penalty calculations
    - Refinancing logic
3. **InvestmentCalculator.java** (231 lines)
    - Compound interest calculations
    - SIP (Systematic Investment Plan) returns
    - ROI and CAGR calculations
    - Portfolio value computation
    - Risk assessment based on volatility
4. **AccountManager.java** (288 lines)
    - Deposit and withdrawal operations
    - Transfer between accounts
    - Overdraft handling with limits (Premium: 50k, Regular: 10k, Savings: 0)
    - Interest calculations for savings accounts
    - Maintenance charge deductions
    - Account upgrades
5. **TransactionValidator.java** (289 lines)

- Transaction amount validation (limit: 500,000)
- Account number format validation
- IFSC code validation
- PIN validation (checks for sequential/repeated digits)
- Description validation for suspicious patterns
- Risk scoring based on multiple factors

Total: 1063 lines of production code

---

2. TESTING STRATEGY

2.1 Test Classes Created

We wrote 168 unit tests across 5 test classes:

1. **CalculatorTest.java** - 7 tests
   - Tests for add, subtract, multiply, divide operations
   - Edge cases like division by zero
   - Negative number handling
2. **LoanCalculatorTest.java** - 30 tests
   - EMI calculation with various interest rates and tenures
   - Eligibility checks with different income/EMI ratios
   - Invalid input handling (negative values, zero values)
   - Prepayment scenarios
   - Refinancing logic
3. **InvestmentCalculatorTest.java** - 40 tests
   - Compound interest with different compounding frequencies
   - SIP calculations over various time periods
   - Portfolio value with multiple investments
   - CAGR calculations
   - Risk level assessments
4. **AccountManagerTest.java** - 38 tests
   - Valid/invalid deposits and withdrawals
   - Transfer operations between accounts
   - Overdraft scenarios for different account types
   - Interest calculations
   - Maintenance charge deductions
   - Account upgrade eligibility
5. **TransactionValidatorTest.java** - 53 tests
   - Transaction amount validation (boundary tests)
   - Account number format validation

- IFSC code validation with valid/invalid formats
- PIN validation (4 and 6 digit PINs)
- Sequential and repeated digit detection in PINs
- Description validation for suspicious content
- Risk score calculations

All 168 tests passed with 100% success rate

---

3. MUTATION TESTING EXPLAINED

3.1 What is Mutation Testing?

Mutation testing is a technique to evaluate test quality by:

1. Creating small changes (mutations) in the source code
2. Running the test suite against each mutation
3. Checking if tests detect the mutation (kill it)
4. If a mutation survives, it means tests are not thorough enough

3.2 Tool Used: PIT (PITest)

- **Version**: 1.15.3 with JUnit 5 plugin
- **Integration**: Maven plugin (pitest-maven)
- **Command**: `mvn pitest:mutationCoverage`
- **Report**: HTML report generated in `target/pit-reports/index.html`

3.3 Mutation Operators Used (9 total)

Unit-Level Operators (3):

1. **ConditionalsBoundaryMutator** (102 generated, 26 killed, 25%)
   - Changes `<` to `<=`, `>` to `>=`
   - Example: `if (value < 0)` becomes `if (value <= 0)`
   - Low kill rate because we didn't test exact boundary values

2. **MathMutator** (135 generated, 113 killed, 84%)
   - Changes `+` to `-`, `*` to `/`, etc.
   - Example: `a + b` becomes `a - b`
   - Good kill rate shows arithmetic operations are well tested

3. **NegateConditionalsMutator** (158 generated, 153 killed, 97%)
   - Changes `==` to `!=`, `<` to `>=`, etc.
   - Example: `if (x > 0)` becomes `if (x <= 0)`
   - Highest kill rate shows excellent conditional testing

Integration-Level Operators (3):

4. **PrimitiveReturnsMutator** (41 generated, 40 killed, 98%)

- Changes return values to 0 for numbers, false for booleans
- Example: `return a + b;` becomes `return 0;`
- Tests integration when one class uses another's return value

5. **BooleanReturnsMutators** (70 generated, 65 killed, 93%)
   - Changes `return true` to `return false` and vice versa
   - Tests method chaining (e.g., withdraw() called by transfer())
6. **MathMutator (Integration Context)**
   - Same as #2 but tests cross-class arithmetic
   - Example: Calculator.subtract() used by TransactionValidator

Additional Operators (3):

7. **IncrementsMutator** (6 generated, 5 killed, 83%)
8. **InvertNegsMutator** (1 generated, 1 killed, 100%)
9. **EmptyObjectReturnValsMutator** (5 generated, 5 killed, 100%)

---

4. INTEGRATION TESTING

4.1 What is Integration-Level Mutation Testing?

Unlike unit testing that tests a single method, integration testing verifies how classes work together.

4.2 Integration Points in Our Project

Integration Point 1: LoanCalculator → Calculator

- Location: `LoanCalculator.isEligible()` uses `Calculator.add()`
- Purpose: Calculate total EMI (existing + new)
- Mutation: When Calculator.add() returns 0, eligibility check fails
- Tests that kill it: `testIsEligibleWithHighEMI()`, `testIsEligibleWithValidIncome()`

Integration Point 2: TransactionValidator → Calculator

- Location: `TransactionValidator.hasSequentialDigits()` uses `Calculator.subtract()` and `Calculator.abs()`
- Purpose: Detect sequential digits in PIN (like 1234)
- Mutation: When subtract changes to add, detection logic inverts
- Tests that kill it: `testIsValidPINSequential()`, `testIsValidPINRepeated()`

Integration Point 3: AccountManager → Calculator

- Location: `AccountManager.calculateOverdraftFee()` uses `Calculator.abs()`
- Purpose: Calculate overdraft amount as positive value
- Mutation: When abs() is mutated, fee calculation breaks
- Tests that kill it: `testCalculateOverdraftFeeLargeOverdraft()`

Integration Point 4: AccountManager → AccountManager

- Location: `AccountManager.transfer()` calls `withdraw()` and `deposit()`
- Purpose: Transfer money between two accounts
- Mutation: If withdraw() returns false when it should return true, transfer fails
- Tests that kill it: `testTransferValid()`, `testTransferInsufficientFunds()`

Integration mutations: ~168 out of 518 (32% of total)

---

5. KEY RESULTS & ANALYSIS

5.1 Mutation Score Breakdown by Operator

Operator	Generated	Killed	Kill Rate	Interpretation
InvertNegsMutator	1	1	100%	Perfect
EmptyObjectReturnValsMutator	5	5	100%	Perfect
PrimitiveReturnsMutator	41	40	98%	Excellent
NegateConditionalsMutator	158	153	97%	Excellent
BooleanFalseReturnValsMutator	24	23	96%	Very Good
BooleanTrueReturnValsMutator	46	42	91%	Very Good
MathMutator	135	113	84%	Good
IncrementsMutator	6	5	83%	Good
ConditionalsBoundaryMutator	102	26	25%	Needs Improvement

5.2 Why Did Some Mutations Survive?

ConditionalsBoundaryMutator (75 survived):

- We didn't test exact boundary values
- Example: Tested `amount > 0` but not `amount == 0`
- Solution: Add more edge case tests at exact boundaries

MathMutator (22 survived):

- Complex formulas where mutated result is still "acceptable"
- Floating-point precision masks some mutations
- Example: In compound interest calculation, small arithmetic changes produce similar results
- Solution: Use stricter assertion tolerances

NegateConditionalsMutator (4 survived):

- Some nested conditional branches not fully tested
- Rare edge cases we didn't consider
- Solution: Achieve 100% branch coverage

5.3 Overall Statistics

- **Total Mutations**: 518
- **Killed**: 408 (79%)
- **Survived**: 102 (20%)
- **No Coverage**: 8 (1%)
- **Line Coverage**: 96%
- **Test Strength**: 80%
- **Test Executions**: 848 runs

79% mutation score exceeds industry standard of 70-75%

---

6. CHALLENGES FACED & SOLUTIONS

Challenge 1: Test Failures During Development

Problem: 6 tests initially failed after first implementation

Failures:

1. Portfolio value mismatch (expected 71000, got 70000)
2. Savings withdrawal expecting exception but got false
3. Overdraft fee returning 0 instead of 1000
4. PIN validation failing for "1234" (sequential)
5. PIN validation failing for "9876" (sequential descending)
6. Description validation case sensitivity issue

Solutions:

1. Fixed expected value to 70000 (calculation was correct)
2. Changed assertion to expect false instead of exception
3. Changed account type to Premium (50k limit) instead of Regular (10k limit)
4. Changed test PINs to non-sequential: "2580", "9753", "258036"
5. Same as #4
6. Fixed source code to use uppercase patterns: "