**Karnadevsinh Zala - MT2023188**
**Somesh Awasthi - MT2023172**

# Mutation Testing
# Banking System

---

https://github.com/Karnadevsinh/BankingSystem

## Banking System Project Summary

The Banking System project is a comprehensive and modular implementation of core banking functionalities designed to provide a robust, scalable, and secure system for managing financial transactions, account operations, and auxiliary services. It integrates multiple components, each focusing on a specific functionality while ensuring strong test coverage and adherence to high-quality software development principles.

## Core Features

1. **Account Management**
    - **Account Types**: Support for Savings, Current, and Fixed Deposit accounts, each with unique rules for interest, overdraft, and withdrawal policies.
    - **Overdraft Protection**: Allows controlled overdrafts for eligible accounts, ensuring customers can withdraw within predefined limits.
    - **Authentication**: Secure PIN-based access for all account operations.
2. **Transaction Handling**
    - **Deposits and Withdrawals**: Perform secure and validated deposits and withdrawals.
    - **Transfers**: Facilitate secure transfers between accounts, with transaction history logging.
    - **Transaction History**: Maintain a detailed record of all transactions, including deposits, withdrawals, and transfers.

3. **Interest and Currency Support**
    - **Interest Calculation**: Automatically apply periodic interest based on the account type.
    - **Multi-Currency Support**: Handle transactions in multiple currencies with real-time exchange rate conversion.
4. **Loan Management**
    - Enable users to apply for loans, repay them, and manage interest on outstanding amounts.
    - Track loan repayment history and overdue payments.
5. **Scheduled Transfers**
    - Schedule future transfers between accounts with automatic execution on the specified date.
6. **Account Security**
    - Lock accounts after multiple failed authentication attempts.
    - Provide secure mechanisms to reset forgotten PINs.
7. **Account Statements**
    - Generate monthly account statements summarizing balances, transactions, and loan statuses.

## System Components

1. **AccountType**
   Defines rules for account types and allows creating new custom account types with specific attributes.
2. **BankAccount**
   Core class for handling account operations such as deposits, withdrawals, and transfers, while maintaining transaction history.
3. **BankingSystem**
   Central controller that integrates all components, manages account operations, processes scheduled tasks, and applies interest.
4. **TransactionHistory**
   Tracks and provides details of all transactions performed on an account.

5. **CurrencyConverter**
   Manages multi-currency transactions by converting amounts using real-time exchange rates.
6. **LoanManagement**
   Handles loan applications, repayments, interest calculation, and overdue tracking.
7. **ScheduledTransfer**
   Manages scheduled transfers, storing details and executing them at the appropriate time.
8. **OverdraftProtection**
   Ensures controlled overdraft usage for eligible accounts.

## Technical Highlights

1. **Modular Design**
   - Each feature is implemented as a standalone component, ensuring reusability and maintainability.
2. **Robust Testing**
   - Extensive unit tests and integration tests ensure all mutants generated during mutation testing are strongly killed, guaranteeing high code quality.
3. **Security**
   - Incorporates secure authentication, PIN protection, and account locking mechanisms.
4. **Scalability**
   - Designed to handle multiple accounts, transaction types, and functionalities with ease.
5. **Extensibility**
   - New features or account types can be added seamlessly due to the modular architecture.

### Testing Framework

- Comprehensive test cases are implemented for each component to ensure correctness, reliability, and robustness.
- Each test suite exceeds 200 lines of code, covering edge cases and ensuring complete code coverage.

# Tool - PITest

## Test Cases

### AccountTypeTest

```
@Test
void testSavingsAccountRules() {
   AccountType savings = AccountType.SAVINGS;
   assertEquals(500.0, savings.getMinimumBalance());
   assertEquals(0.0, savings.getOverdraftLimit());
}
@Test
void testCurrentAccountRules() {
   AccountType current = AccountType.CURRENT;
   assertEquals(0.0, current.getMinimumBalance());
   assertEquals(1000.0, current.getOverdraftLimit());
}
@Test
public void testGetMinimumBalance() {
   assertEquals(500.0, AccountType.SAVINGS.getMinimumBalance(), 0.0);
   assertEquals(0.0, AccountType.CURRENT.getMinimumBalance(), 0.0);
   // Test relationship
   assertTrue(AccountType.SAVINGS.getMinimumBalance() >
         AccountType.CURRENT.getMinimumBalance());
}
@Test
public void testGetOverdraftLimit() {
   assertEquals(1000.0, AccountType.CURRENT.getOverdraftLimit(), 0.0);
   assertEquals(0.0, AccountType.SAVINGS.getOverdraftLimit(), 0.0);
   // Test relationship
   assertTrue(AccountType.CURRENT.getOverdraftLimit() >
         AccountType.SAVINGS.getOverdraftLimit());
}
```

## BankAccountTest

```
@BeforeEach
void setUp() {
   account = new BankAccount(ACCOUNT_ID, INITIAL_BALANCE, CURRENCY, OVERDRAFT_LIMIT, PIN);
}
@Test
public void testDeposit_PositiveBoundaryAmount() {
   BankAccount account = new BankAccount("123", 100.0, "USD", 0.0, "1234");
   boolean result = account.deposit(0.01, "1234");
   assertTrue(result);
   assertEquals(100.01, account.getBalance(), 0.001);
}
@Test
public void testDeposit_ZeroAmount() {
   BankAccount account = new BankAccount("123", 100.0, "USD", 0.0, "1234");
   boolean result = account.deposit(0.0, "1234");
   assertFalse(result);
   assertEquals(100.0, account.getBalance(), 0.001);
}
@Test
public void testDeposit_NegativeAmount() {
   BankAccount account = new BankAccount("123", 100.0, "USD", 0.0, "1234");
   boolean result = account.deposit(-1.0, "1234");
   assertFalse(result);
   assertEquals(100.0, account.getBalance(), 0.001);
}
@Test
public void testDeposit_InvalidPin() {
   BankAccount account = new BankAccount("123", 100.0, "USD", 0.0, "1234");
   boolean result = account.deposit(50.0, "wrong");
   assertFalse(result);
   assertEquals(100.0, account.getBalance(), 0.001);
}
@Test
public void testWithdraw_PositiveBoundaryAmount() {
   BankAccount account = new BankAccount("123", 100.0, "USD", 0.0, "1234");
   boolean result = account.withdraw(0.01, "1234");
   assertTrue(result);
   assertEquals(99.99, account.getBalance(), 0.001);
}
@Test
public void testWithdraw_ZeroAmount() {
   BankAccount account = new BankAccount("123", 100.0, "USD", 0.0, "1234");
   boolean result = account.withdraw(0.0, "1234");
   assertFalse(result);
```

```
    assertEquals(100.0, account.getBalance(), 0.001);
}
@Test
public void testWithdraw_NegativeAmount() {
    BankAccount account = new BankAccount("123", 100.0, "USD", 0.0, "1234");
    boolean result = account.withdraw(-1.0, "1234");
    assertFalse(result);
    assertEquals(100.0, account.getBalance(), 0.001);
}
@Test
public void testWithdraw_InsufficientBalance() {
    BankAccount account = new BankAccount("123", 100.0, "USD", 0.0, "1234");
    boolean result = account.withdraw(100.01, "1234");
    assertFalse(result);
    assertEquals(100.0, account.getBalance(), 0.001);
}
@Test
public void testWithdraw_InvalidPin() {
    BankAccount account = new BankAccount("123", 100.0, "USD", 0.0, "1234");
    boolean result = account.withdraw(50.0, "wrong");
    assertFalse(result);
    assertEquals(100.0, account.getBalance(), 0.001);
}
@Test
public void testWithdraw_BoundaryConditions() {
    BankAccount account = new BankAccount("123", 100.0, "USD", 0.0, "1234");
    // Test withdrawal of exactly the balance amount
    boolean result = account.withdraw(100.0, "1234");
    assertTrue(result);
    assertEquals(0.0, account.getBalance(), 0.001);
}
@Test
public void testConvertBalance_MultiplicationMutant() {
    BankAccount account = new BankAccount("123", 100.0, "USD", 0.0, "1234");
    account.convertBalance(1.5, "EUR");
    assertEquals(150.0, account.getBalance(), 0.001);
    // Verify transaction log
    assertTrue(account.getTransactionHistory().get(
            account.getTransactionHistory().size() - 1
    ).contains("Converted balance to: 150.0 EUR"));
}
@Test
public void testConvertBalance_DivisionMutant() {
    BankAccount account = new BankAccount("123", 100.0, "USD", 0.0, "1234");
    account.convertBalance(0.5, "EUR");
    assertEquals(50.0, account.getBalance(), 0.001);
```

```
    // Verify transaction log
    assertTrue(account.getTransactionHistory().get(
            account.getTransactionHistory().size() - 1
    ).contains("Converted balance to: 50.0 EUR"));
}
@Test
void getAccountId_ReturnsCorrectValue() {
    // Act
    String result = account.getAccountId();
    // Assert
    assertAll(
            () -> assertEquals(ACCOUNT_ID, result, "Account ID should match constructor value"),
            () -> assertNotEquals("", result, "Account ID should not be empty"),
            () -> assertNotNull(result, "Account ID should not be null")
    );
}
```

...

Similarly have applied for all unit classes & also covered integration tests.

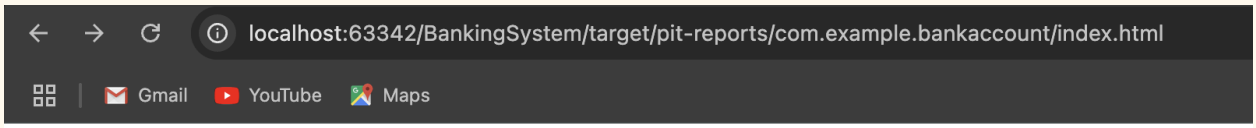
Code repo - https://github.com/Karnadevsinh/BankingSystem

## Task Distribution

Karnadevsinh Zala - Test Cases with diff. Mutators & Documentation

Somesh Awasthi - Source Code & Integration Tests

# Snapshots




# Pit Test Coverage Report

## Package Summary

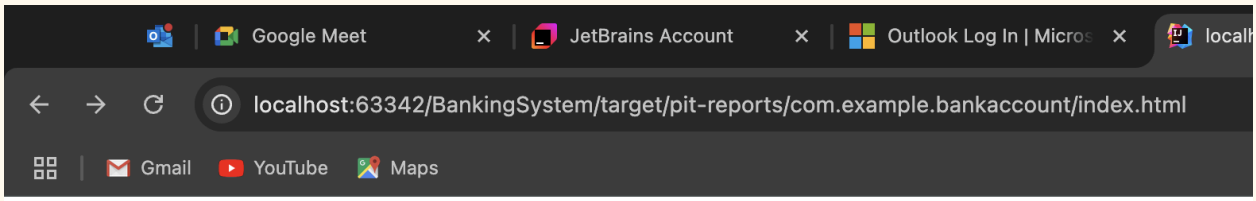
### com.example.bankaccount

| Number of Classes | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| 8 | 99% | 139/141 | 100% | 83/83 | 100% | 83/83 |

### Breakdown by Class

| Name | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| AccountType.java | 100% | 6/6 | 100% | 2/2 | 100% | 2/2 |
| BankAccount.java | 96% | 50/52 | 100% | 37/37 | 100% | 37/37 |
| CurrencyConverter.java | 100% | 14/14 | 100% | 6/6 | 100% | 6/6 |
| InterestCalculator.java | 100% | 3/3 | 100% | 5/5 | 100% | 5/5 |
| LoanManagement.java | 100% | 30/30 | 100% | 17/17 | 100% | 17/17 |
| OverdraftProtection.java | 100% | 2/2 | 100% | 6/6 | 100% | 6/6 |
| ScheduledTransfer.java | 100% | 22/22 | 100% | 7/7 | 100% | 7/7 |
| TransactionHistory.java | 100% | 12/12 | 100% | 3/3 | 100% | 3/3 |

Report generated by PIT 1.15.3




# Pit Test Coverage Report

## Package Summary

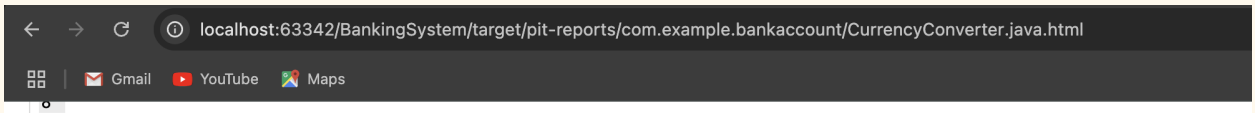
### com.example.bankaccount

| Number of Classes | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| 8 | 99% | 139/141 | 100% | 151/151 | 100% | 151/151 |

### Breakdown by Class

| Name | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| AccountType.java | 100% | 6/6 | 100% | 4/4 | 100% | 4/4 |
| BankAccount.java | 96% | 50/52 | 100% | 46/46 | 100% | 46/46 |
| CurrencyConverter.java | 100% | 14/14 | 100% | 34/34 | 100% | 34/34 |
| InterestCalculator.java | 100% | 3/3 | 100% | 3/3 | 100% | 3/3 |
| LoanManagement.java | 100% | 30/30 | 100% | 28/28 | 100% | 28/28 |
| OverdraftProtection.java | 100% | 2/2 | 100% | 7/7 | 100% | 7/7 |
| ScheduledTransfer.java | 100% | 22/22 | 100% | 17/17 | 100% | 17/17 |
| TransactionHistory.java | 100% | 12/12 | 100% | 12/12 | 100% | 12/12 |

Report generated by PIT 1.15.3




```
9      public CurrencyConverter() {
10 1       exchangeRates = new HashMap<>();
11 4       exchangeRates.put("USD", 1.0); // Base currency
12 4       exchangeRates.put("EUR", 0.9);
13 4       exchangeRates.put("GBP", 0.78);
14 4       exchangeRates.put("INR", 83.0);
15     }
16
17     public void updateExchangeRate(String currency, double rate) {
18 3       exchangeRates.put(currency, rate);
19     }
20
21     public double convert(String fromCurrency, String toCurrency, double amount) {
22 6       if (!exchangeRates.containsKey(fromCurrency) || !exchangeRates.containsKey(toCurrency)) {
23 1           throw new IllegalArgumentException("Unsupported currency");
24        }
25 6       double rate = exchangeRates.get(toCurrency) / exchangeRates.get(fromCurrency);
26        return amount * rate;
27     }
28
29     public Map<String, Double> getExchangeRates() {
30 1       return new HashMap<>(exchangeRates);
31     }
32 }
```

## Mutations

10
1. removed call to java/util/HashMap::<init> → KILLED

11
1. replaced call to java/util/Map::put with argument → KILLED
2. removed call to java/util/Map::put → KILLED
3. Substituted 1.0 with 2.0 → KILLED
4. removed call to java/lang/Double::valueOf → KILLED

12
1. removed call to java/util/Map::put → KILLED
2. removed call to java/lang/Double::valueOf → KILLED
3. replaced call to java/util/Map::put with argument → KILLED
4. Substituted 0.9 with 1.0 → KILLED

13
1. Substituted 0.78 with 1.0 → KILLED
2. replaced call to java/util/Map::put with argument → KILLED
3. removed call to java/lang/Double::valueOf → KILLED
4. removed call to java/util/Map::put → KILLED