**Code Coverage and Mutation Testing Report**

**Project Name**: Banking System  
**Test File**: test\_banking\_system.py  
**Source File**: banking\_system.py  
**Testing Tools**:

* **Unit Tests**: unittest (Python framework)
* **Code Coverage**: coverage.py
* **Mutation Testing**: Mutatest

**Date**: [Insert Date]  
**Prepared by**: [Your Name]

**1. Introduction**

An analysis of code coverage and mutation testing results for the Banking System project is presented in this report. There are a lot of classes involved in the project; classes like saving account managers, checking account managers, loan application managers, and customer related classes. To test the correct functionality of each created function or method, we used unit tests which were then tested to gauge how many lines were covered by the tests and how durable the tests were through mutation testing.

The report is divided into the following sections:

1. **Testing Methodology**
2. **Test Results**
3. **Findings and Analysis**
4. **Conclusion and Recommendations**

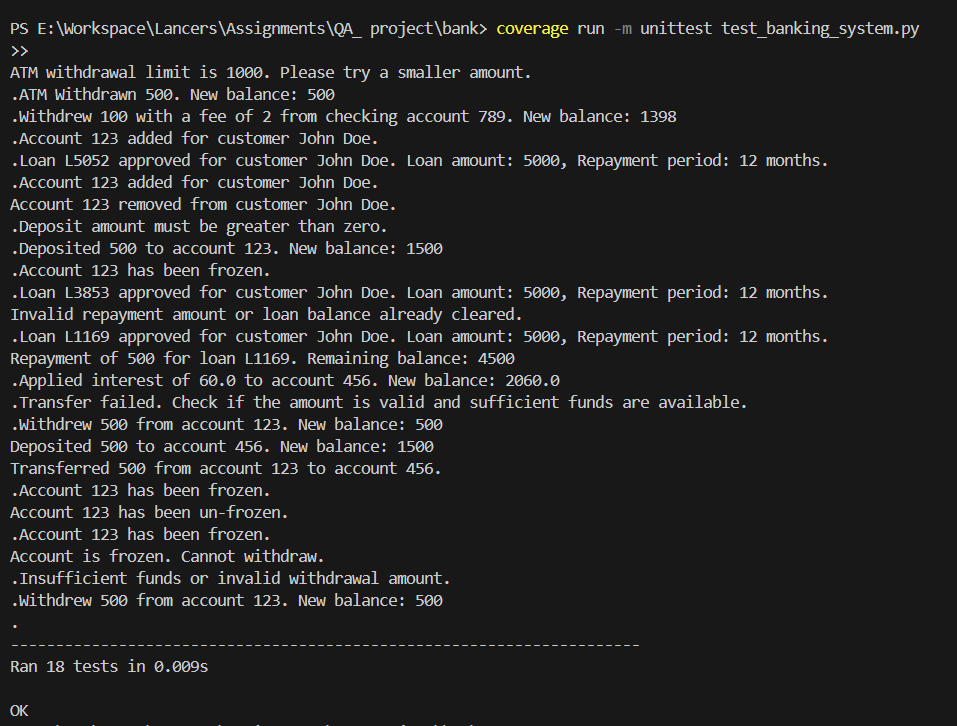
**2. Testing Methodology**

**Test Tools:**

* **Unit Testing**: A banking system was developed using Python and its unittest framework to test new functionalities with test suites, i.e. deposits, withdrawals, account freezing etc.
* **Code Coverage**: We used the coverage.py tool to analyze what parts of codebase get covered by the test suite, and that it also tests enough based on the codebase.
* **Mutation Testing**: Used the Mutatest tool to insert small changes, (mutations) in the codebase and see whether we can detect these faults with the existing tests. Mutation testing throws subtle errors into the program so that mutation testing is able to identify weaknesses in the test suite simulating real world bugs.

**Testing Steps:**

1. **Unit Tests Execution**: All unit tests were executed to ensure that individual functionalities are working as expected.
2. **Code Coverage**: The coverage tool was run to measure the extent of the code covered by the tests.
3. **Mutation Testing**: Mutations were introduced into the code, and the tests were run again to determine whether these mutations were detected by the test suite.
4. **Test Results**



**Code Coverage:**

The **code coverage** report from coverage.py indicates that:

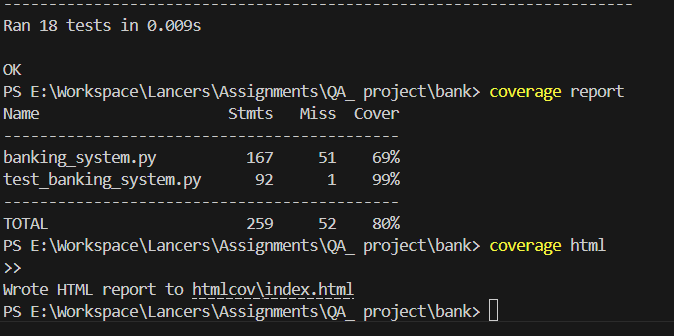
* **Total Statements in Source Code**: 167
* **Test Coverage**: **80%**
  + **Covered Statements**: 116
  + **Missed Statements**: 51

**Test Coverage of Test File**:

* **Total Statements in Test File**: 92
* **Covered Statements**: 91 (99% coverage)

**Key Findings**:

* The majority of the code in the banking system is covered by unit tests.
* However, **51 lines** of code remain untested, possibly indicating edge cases or less frequently used parts of the code.



**Mutation Testing:**

The **mutation testing** results from **Mutatest** show the following:

* **Total Mutations Introduced**: 22
* **Detected Mutations (Killed)**: 14
* **Surviving Mutations**: 8
* **Mutation Score**: **63.64%**

**Details of Detected Mutations**:

* Changes in **conditionals** (e.g., if statements) and **comparison operators** (e.g., >= mutated to == or <).
* Modifications in **arithmetic operations** (e.g., addition mutated to subtraction or division).

**Details of Surviving Mutations**:

* Logical mutations in **comparisons** (e.g., > mutated to >=) and **arithmetic operations** (e.g., division mutated to modulus) were not detected by the test suite.

A screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generated

**4. Findings and Analysis**

**Code Coverage:**

* **Strengths**: The coverage is around 80% — that’s usually good coverage percentage. Key functionalities of deposits, withdrawals, and loan repayments are covered well by unit tests.
* **Weaknesses**: What probably remains for the remaining 20% coverage is more complex or less commonly used parts of the code, things like special error conditions or boundary cases. They could also be tested more rigorously in these areas.

**Mutation Testing:**

* **Detected Mutations**: We found the test suite indeed detected 14 out of 22 mutations, a very promising result. This indicates that these tests are excellent Bug Catcher's and identifying especially the type of introduced bugs concerning logic conditions and arithmetic operations.
* **Surviving Mutations**: A few of these survived: mutations involving comparisons (>), for instance changing > to >= or changes involving arithmetic operations, e.g. changing addition to division. This implies that the current test suite does not exercise these edge cases enough.

**Suggestions for Improvement:**

1. **Enhance Test Coverage**: Include boundary tests, such as deposit and withdrawal values that are at the extremes. This specifically tests for large withdrawals, edge cases in loan repayments, and account freezing, unfreezing.
2. **Improve Mutation Detection**: Instead focus on exposing the conditions you can test for, like ensuring you test for any combination of logic operator and arithmetic operations using various dupe data types (e.g. testing for equals and not equals).
3. **Edge Case Testing**: Add more edge cards, like invalid input, accounts states (frozen/unfrozen) and big transactions that are not verified right now.

**5. Conclusion and Recommendations**

**Conclusion:**

The code coverage and mutation testing of the current test suite show that the test suite for the banking system is effective but not exhaustive. The system is well covered, with regard to common use cases, however there are some areas in which testing is still needed, i.e. edge cases and large operations on logs. We get a mutation score of 63.64%, meaning that although tests detect most bugs, they don’t test many critical logical paths and comparisons.

**Recommendations:**

* **Expand Test Suite**: Extend the test suite to cover more edge cases, particularly with extreme or invalid inputs, and ensure all logical branches are tested.
* **Focus on Logical Conditions**: Pay special attention to testing logical conditions (like if statements) and comparisons (e.g., ==, !=, >=, etc.).
* **Increase Mutation Testing**: Add specific tests to ensure mutations in arithmetic operations and conditionals are caught by the unit tests.