International Institute of Information Technology Bangalore

CSE 731-Software Testing
Term I 2024-25: Project Report

Nishtha Paul	MT2023161
Nisha Rathod	MT2023195

Problem

Mutation testing: Projects that use mutation testing, based on mutation operators applied at the level of a statement within a method or a function and at the integration level. The mutated program needs to be strongly killed by the designed test cases. At least three different mutation operators should be used at the unit level and three different mutation operators at the integration level should be used.

Source Code Description

This project aims to enhance financial management by creating a comprehensive system for tracking transactions, managing budgets, generating reports, and achieving financial goals. The system is tested rigorously using mutation testing to ensure its robustness and reliability.

GitHub
https://github.com/nisharathod231/Mutation_Testing_Using_PIT.git

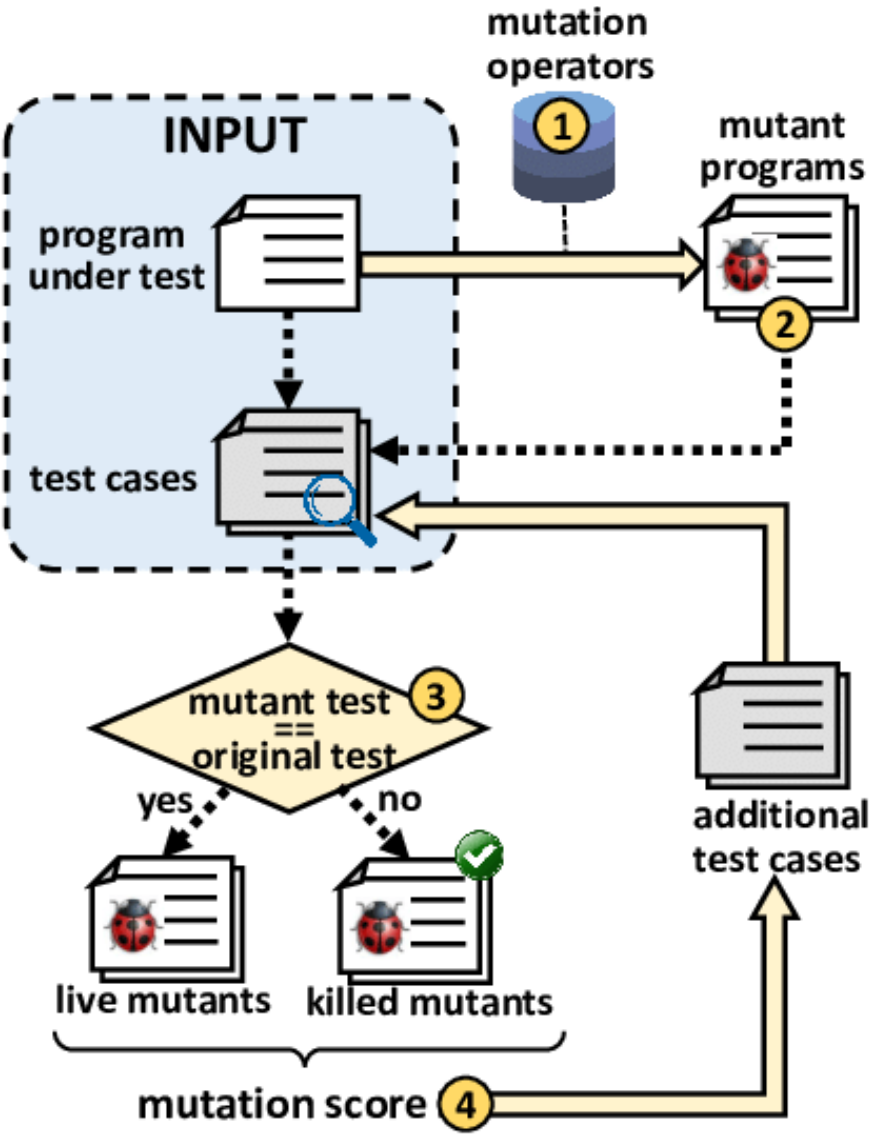
Introduction

Comprehensive testing plays a vital role in ensuring the reliability and robustness of software applications. This project emphasizes rigorous testing at **both unit and integration levels** using **mutation testing** with the **PIT tool**. The primary objective is to evaluate the effectiveness of the test suite, uncover weaknesses in the application's codebase, and ensure high-quality performance under diverse scenarios. The application being tested is designed to enhance healthcare delivery by empowering field health workers with a tablet-based solution.

Mutation Testing

Mutation testing is an advanced software testing technique used to measure the quality of a test suite by evaluating its ability to detect faults in the code. The core idea behind mutation testing is to introduce small, deliberate changes—known as *mutations*—to the program's source code to simulate potential errors that could occur during development. These changes typically involve modifying operators, variables, or conditions in the code. The altered versions of the program, referred to as *mutants*, are then tested against the existing test suite to determine if the tests can identify and fail the mutated code.

The primary goal of mutation testing is to assess the effectiveness of the test cases by determining how many of the introduced mutants are "killed" (detected and failed by the test cases). If a mutant is not detected and passes all the tests, it is considered a "surviving mutant," indicating a potential gap in the test coverage. This provides valuable insights into areas of the code that may not be adequately tested.



Key concepts:

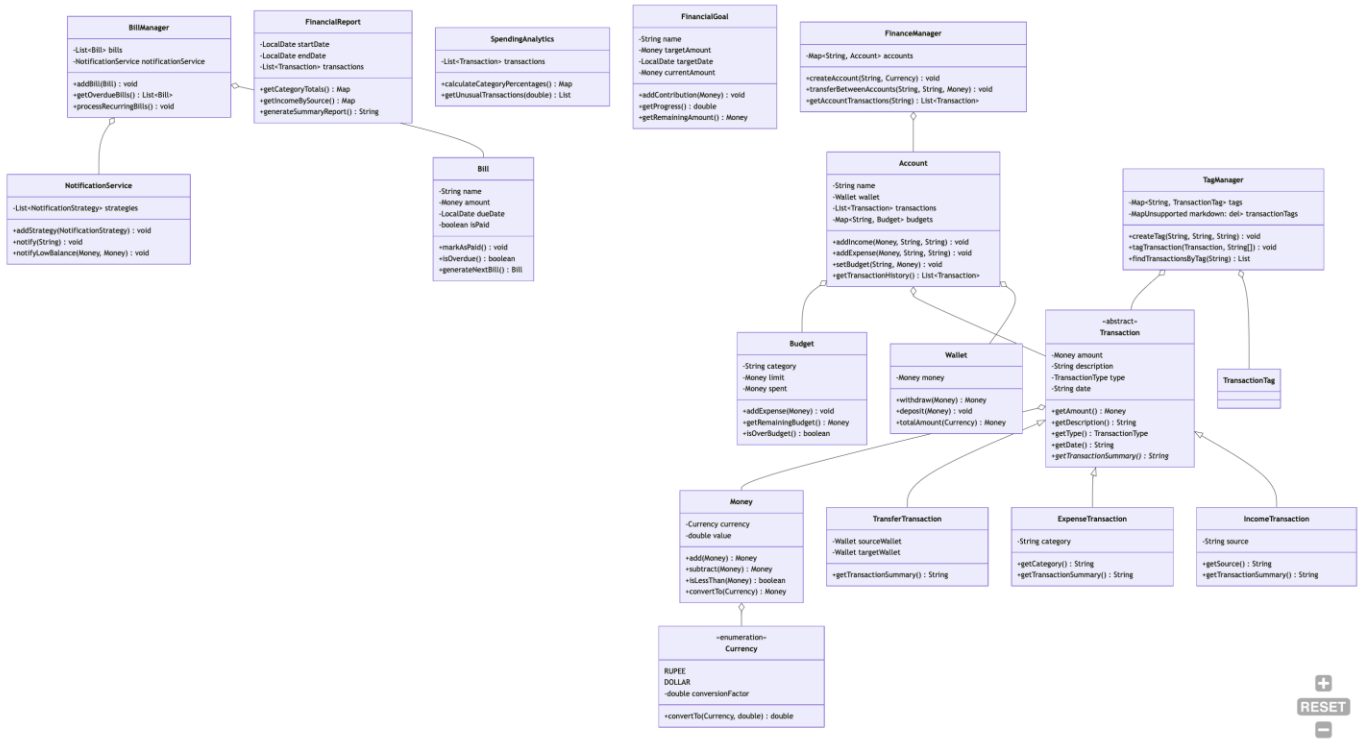
1. **Mutation Operators**: These are rules used to introduce changes in the code. Examples include replacing arithmetic operators (+ with -), altering logical conditions (> with <), or modifying return values.
2. **Mutant Killing**: When a test case fails due to the introduced mutation, the mutant is considered "killed," indicating the test case is effective.
3. **Surviving Mutants**: Mutants that pass all test cases suggest that the test suite may lack the necessary coverage to detect specific issues.

OPERATOR	EXAMPLE
AOR	`a + b` → {`a, b, a - b, a * b, a / b, a % b`}
LCR	`a && b` → {`a, b, a || b, true, false`}
ROR	`a > b` → {`a < b, a <= b, a >= b, true, false`}
UOI	`a` → {`a++, a--`} (also `a` → `!a`, etc.)
SBR	`stmt` → ∅

Mutation testing is considered more robust than traditional code coverage metrics (like line or branch coverage) because it evaluates the fault-detection ability of tests, not just whether a line of code was executed. However, it is computationally expensive since it requires generating and testing numerous mutants. Tools such as PIT (Pitest), Stryker, and MutPy help automate the process, making it more accessible for developers and testers.

By identifying weaknesses in a test suite, mutation testing ensures higher code quality and increases confidence in the reliability of the software, making it especially valuable for applications where robustness is critical. We will be using Pitest to test our application.

Class Diagram



Project Set Up

Before starting mutation testing, we need a properly structured project.

Codebase: We should have a Java-based application with logically divided packages and classes.

Unit Tests: Mutation testing depends on the presence of unit tests. These tests are small and isolated, focusing on testing specific methods or components. Without unit tests, PIT won't be able to evaluate how well our test suite detects errors.

Build Tool: Mutation testing with PIT works seamlessly with Maven or Gradle. Our project uses Maven.

Add PIT to Build Configuration

We configure PIT as a plugin in the pom.xml file, allowing us to automate the mutation testing process. This involves adding the following plugin configuration under the <build> tag:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.pitest</groupId>
            <artifactId>pitest-maven</artifactId>
            <version>1.17.1</version>
            <configuration>
                <mutators>
                    <mutator>ALL</mutator>
                </mutators>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Group ID and Artifact ID: These identifiers ensure Maven downloads the correct plugin.

Version: We specify the plugin version to avoid compatibility issues.

targetClasses: Specifies the classes in which mutations will be applied. For example, all classes under the com.example package.

targetTests: Identifies the test classes PIT will use to verify the mutants.

threads: Determines the number of parallel processes, which speeds up mutation testing on large codebases.

Run PIT for Mutation Testing

Once the configuration is complete, we run the PIT tool to perform mutation testing.

We execute the following command in the terminal:

```
mvn org.pitest:pitest-maven:mutationCoverage
```

This command tells Maven to invoke the PIT plugin, generate mutants, and evaluate our test suite.

Review the Mutation Testing Report

After execution, PIT generates a detailed HTML report. We can find this report in the following directories:

```
target/pit-reports
```

We open the report in a browser to analyze:

Mutation Score: The percentage of mutants killed by the test suite.

Mutant Details: Information about each mutant, including the type of mutation, location in the code, and whether it was killed.

Surviving Mutants: These are mutants that were not detected by the test cases, indicating potential gaps in the test suite.

For example, if a mutant modifies a condition like *if (x > 5) to if (x < 5)* and the test suite doesn't fail, it means our test cases don't cover scenarios where x is less than or equal to 5.

Analyze Surviving Mutants

Surviving mutants indicate areas in our code where the test suite is insufficient.

Root Cause Analysis: We examine why these mutants were not detected. Common reasons include:

- Missing test cases.
- Inadequate assertions in existing tests.
- Code that is difficult to test, such as edge cases or complex logic.

Enhancing Tests: Based on our analysis, we add or modify test cases to cover these scenarios

Optimize Configuration

To make the mutation testing process efficient, we fine-tune the PIT configuration.

Include Specific Classes: We can target only critical components for mutation testing.

Exclude Irrelevant Classes: We can skip classes like configuration files or boilerplate code.

```
<excludedTests>
    <param>com.example.integration.*</param>
    <param>com.example.config.*</param>
    <param>**/*IntegrationTest</param>
</excludedTests>
```

PIT Test Coverage Report

PIT (Pi test) generates a detailed mutation testing coverage report that provides insights into how well a test suite detects faults in the code. This report helps identify areas in the application where the test coverage is lacking, enabling us to enhance the overall quality of our tests. Let's break down the key components of the PIT coverage report:

1. **Mutation Coverage Score**
   The mutation coverage score is the primary metric in the report. It represents the percentage of mutants killed by the test suite.

$MS(P,T) = \frac{K}{(M-E)}$, where:	*P*: program under test *T*: test suite *K*: number of killed mutants *M*: number of generated mutants *E*: number of equivalent mutants

2. **Types of Mutations**
   The report categorizes mutants based on the type of change introduced in the code. Common mutation types include

Conditionals Boundary Mutator, Increments Mutator, Math Mutator, etc.

3. **Detailed HTML Report**
   PIT generates an interactive HTML report that visualizes the results in an easy-to-navigate format. This report includes:

- Dashboard Overview: A summary of overall mutation coverage, including graphs and percentages.
- Class-Level Views: Detailed coverage statistics for each class, along with hyperlinks to explore specific code sections.
- Color-Coded Indicators:

Green	Mutants killed by the tests.
Red	Surviving mutants.
Yellow	Mutants that were skipped or not executed

Project Implementation

1. PIT starts by analyzing the source code and identifying the parts where mutations can be made. They can be introduced for unit or integration test cases.

```
public class Money {
    12 usages
    private final Currency currency;
    12 usages
    private final double value;

    Nishtha Paul
    public Money(Currency currency, double value) {...}

    44 usages   Nishtha Paul
    public static Money createMoney(Currency currency, double value) throws InvalidMoneyException {
        if (value <= 0) {
            throw new InvalidMoneyException(value);
        }
        return new Money(currency, value);
    }
```

Initially, no functionality is tested, so we can see that coverage is 0.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
26	0%	0/393	0%	0/781	100%	0/0

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.example.account	1	0%	0/28	0%	0/36	100%	0/0
org.example.analytics	1	0%	0/44	0%	0/105	100%	0/0
org.example.bills	2	0%	0/48	0%	0/103	100%	0/0
org.example.budget	1	0%	0/9	0%	0/17	100%	0/0
org.example.exceptions	2	0%	0/6	0%	0/3	100%	0/0
org.example.goals	1	0%	0/17	0%	0/40	100%	0/0
org.example.manager	1	0%	0/19	0%	0/38	100%	0/0
org.example.models	3	0%	0/43	0%	0/97	100%	0/0
org.example.notifications	3	0%	0/22	0%	0/15	100%	0/0
org.example.recurring	2	0%	0/38	0%	0/82	100%	0/0
org.example.reports	1	0%	0/26	0%	0/71	100%	0/0
org.example.savings	2	0%	0/34	0%	0/62	100%	0/0
org.example.tags	2	0%	0/34	0%	0/72	100%	0/0
org.example.transactions	4	0%	0/25	0%	0/40	100%	0/0

Money.java

```
1    package org.example.models;
2
3    import org.example.exceptions.InsufficientMoneyException;
4    import org.example.exceptions.InvalidMoneyException;
5
6    public class Money {
7        private final Currency currency;
8        private final double value;
9
10       public Money(Currency currency, double value) {
11 1         this.currency = currency;
12 1         this.value = value;
13       }
14
15       public static Money createMoney(Currency currency, double value) throws InvalidMoneyException {
16 5         if (value <= 0) {
17 1             throw new InvalidMoneyException(value);
18          }
19 2         return new Money(currency, value);
20       }
```

2. PIT modifies the code to introduce “mutants”, which are variations of the original code.
   PIT has created 5 mutants for createMoney() like changing conditional boundaries (<= to <), negating conditions (if(x) to if (! x)) replacing constants, etc.

```
1. createMoney : changed conditional boundary → NO_COVERAGE
2. createMoney : Substituted 0.0 with 1.0 → NO_COVERAGE
3. createMoney : removed conditional - replaced comparison check with false →
NO_COVERAGE
4. createMoney : removed conditional - replaced comparison check with true →
NO_COVERAGE
5. createMoney : negated conditional → NO_COVERAGE
```

3. We will write test cases, but even if we include unnecessary code in them, it will still count towards line/code coverage. However, this only indicates that the code was executed, it doesn't guarantee correctness. Mutation testing is used to verify if we have written effective and accurate test cases to catch faults in the code.
4. We'll write a test case:

```java
@Test
public void shouldNotBeAbleToCreateNegativeValuedMoney() {
    assertThrows(InvalidMoneyException.class, () -> createMoney(Currency.RUPEE, value: -5));
}
```

5. PIT will run the test cases on original and mutated code to check if the mutants survived or were killed.

```
15      public static Money createMoney(Currency currency, double value) throws InvalidMoneyException {
16 5        if (value <= 0) {
17 1            throw new InvalidMoneyException(value);
18          }
19 2        return new Money(currency, value);
20      }
    1. changed conditional boundary → SURVIVED Covering tests
    Covered by tests:
      • org.example.models.MoneyTest.shouldNotBeAbleToCreateNegativeValuedMoney(org.example.models.MoneyTest)
    2. Substituted 0.0 with 1.0 → SURVIVED Covering tests
    Covered by tests:
16    • org.example.models.MoneyTest.shouldNotBeAbleToCreateNegativeValuedMoney(org.example.models.MoneyTest)
    3. removed conditional - replaced comparison check with false → KILLED
    4. removed conditional - replaced comparison check with true → SURVIVED Covering tests
    Covered by tests:
      • org.example.models.MoneyTest.shouldNotBeAbleToCreateNegativeValuedMoney(org.example.models.MoneyTest)
    5. negated conditional → KILLED
```

The "Covering tests" in the mutation testing report refers to the **test cases that executed the mutated code** but **failed to detect the**

mutation. This helps you identify which tests need additional assertions or logic to effectively verify the behavior of the code.

6. So, we added another test case

```
@Test
public void shouldNotBeAbleToCreateZeroValuedMoney() {
    assertThrows(InvalidMoneyException.class, () -> createMoney(Currency.RUPEE, value: 0));
}
```

```
    1. changed conditional boundary → KILLED
    2. Substituted 0.0 with 1.0 → SURVIVED Covering tests
16  3. removed conditional - replaced comparison check with false → KILLED
    4. removed conditional - replaced comparison check with true → SURVIVED Covering tests
    5. negated conditional → KILLED
17  1. removed call to org/example/exceptions/InvalidMoneyException::<init> → KILLED
```

We can see that the “changed conditional boundary mutant” has been killed now. It indicates our test suite covers this mutation.

Mutation Operators for Unit testing

Arithmetic Operator (MATH)

```
public class Money {
    new *
    public Money add(Money money) {
        Money convertedMoney = money.convertTo(this.currency);

        return new Money(this.currency, value: this.value + convertedMoney.value);
    }
```

```java
@Test
public void addRupeeToDollar() throws InvalidMoneyException {
    Money oneRupee = createMoney(Currency.RUPEE, value: 1);
    Money oneDollar = createMoney(Currency.DOLLAR, value: 1);

    Money sum = oneRupee.add(oneDollar);

    assertThat(sum, is(equalTo(createMoney(Currency.RUPEE, value: 75.85))));
}
```

```
22        public Money add(Money money) {
23 2          Money convertedMoney = money.convertTo(this.currency);
24
25 3          return new Money(this.currency, this.value + convertedMoney.value);
26        }
```

```
    1. Replaced double addition with subtraction → KILLED
25  2. removed call to org/example/models/Money::<init> → KILLED
    3. replaced return value with null for org/example/models/Money::add → KILLED
```

Mutant was created by replacing

return new Money (this.currency, this.value + convertedMoney.value);

by

return new Money (this.currency, this.value - convertedMoney.value);

Replace by trueops

```java
public class Wallet {

    9 usages
    private Money money;

    3 usages  Nishtha Paul
    public Money withdraw(Money money) throws InsufficientMoneyException {
        if (this.money == null) {
            throw new InsufficientMoneyException();
        }
        if (this.money.isLessThan(money)) {
            throw new InsufficientMoneyException();
        }
        this.money = this.money.subtract(money);
        return money;
    }
```

```java
@Test
public void shouldReflectChangeInWalletMoneyAfterWithdrawing() throws InsufficientMoneyException, InvalidMoneyException {
    Money fiveRupee = createMoney(Currency.RUPEE, value: 5);
    Wallet wallet = new Wallet();
    wallet.deposit(fiveRupee);

    Money twoRupee = createMoney(Currency.RUPEE, value: 2);
    wallet.withdraw(twoRupee);

    Money totalAmount = wallet.totalAmount(Currency.RUPEE);
    Money threeRupee = createMoney(Currency.RUPEE, value: 3);
    assertThat(totalAmount, is(equalTo(threeRupee)));
}
```

```
1. negated conditional → KILLED
2. removed call to org/example/models/Money::isLessThan → SURVIVED Covering tests
3. removed conditional - replaced equality check with true → KILLED
4. removed conditional - replaced equality check with false → SURVIVED Covering tests
```

Mutant was created by replacing

if (this.money.isLessThan(money)) by if (true)

This test case asserted the expected amount in wallet after withdrawing the money on the original code while in mutated code, it threw InsufficientMoneyException. Thus, the mutant was killed.

Relational Operator

```java
public class Money {
    12 usages
    private final Currency currency;
    12 usages
    private final double value;

    Nishtha Paul
    public Money(Currency currency, double value) {...}

    10 usages  Nishtha Paul
    public static Money createMoney(Currency currency, double value) throws InvalidMoneyException {
        if (value <= 0) {
            throw new InvalidMoneyException(value);
        }
        return new Money(currency, value);
    }
```

```java
@Test
public void shouldNotBeAbleToCreateNegativeValuedMoney() {
    assertThrows(InvalidMoneyException.class, () -> createMoney(Currency.RUPEE, value: -5))
}

Nishtha Paul
@Test
public void shouldNotBeAbleToCreateZeroValuedMoney() {
    assertThrows(InvalidMoneyException.class, () -> createMoney(Currency.RUPEE, value: 0));
}
```

```
1. changed conditional boundary → KILLED
2. Substituted 0.0 with 1.0 → KILLED
3. removed conditional - replaced comparison check with false → KILLED
4. removed conditional - replaced comparison check with true → KILLED
5. negated conditional → KILLED
```

Mutant was created by replacing

Changed Conditional Boundary - if (value <= 0) by if (value > 0)

Negated Conditional - if (value <= 0) by if (value < 0)

Mutation Operators for Integration testing

Integration Method Call Deletion

```java
public class Wallet {
    5 usages   new *
    public void deposit(Money money) {
        if (this.money == null) {
            this.money = money;
        } else {
            this.money = this.money.add(money);
        }
    }
```

```java
@Test
public void shouldReturnCorrectTotalAmountOfWalletMoneyInRupees() throws InvalidMoneyException {
    Money fiftyRupee = createMoney(Currency.RUPEE, value: 50);
    Money oneDollar = createMoney(Currency.DOLLAR, value: 1);
    Wallet wallet = new Wallet();
    wallet.deposit(fiftyRupee);
    wallet.deposit(oneDollar);

    Money totalAmount = wallet.totalAmount(Currency.RUPEE);

    assertThat(totalAmount, is(equalTo(createMoney(Currency.RUPEE, value: 124.85))));
}
```

```
1. removed call to org/example/models/Money::add → KILLED
2. Removed assignment to member variable money → KILLED
3. replaced call to org/example/models/Money::add with receiver → KILLED
4. replaced call to org/example/models/Money::add with argument → KILLED
```

Mutant was created by removing `this.money.add(money)` call. In the original code, this test case would add up all the money while in the mutant, it would remove the money.add call and there will be no change in the money variable. Thus, both give different results, and the mutant will be killed.

Replace by argument

```
class FinancialGoal {
    1 usage
    private final String name;
    3 usages
    private final Money targetAmount;
    2 usages
    private final LocalDate targetDate;
    5 usages
    private Money currentAmount;
    1 usage
    private final String category;

    1 usage   new *
    public Money getRemainingAmount() throws InsufficientMoneyException {
        return targetAmount.subtract(currentAmount);
    }
```

```
public class FinancialGoalTest {

    3 usages
    private FinancialGoal financialGoal;

    new *
    @Before
    public void setup() throws InvalidMoneyException {
        Money targetAmount = Money.createMoney(Currency.RUPEE, value: 10000);
        LocalDate targetDate = LocalDate.now().plusDays( daysToAdd: 60);
        financialGoal = new FinancialGoal( name: "Vacation", targetAmount, targetDate, category: "Leisure");
    }

    new *
    @Test
    public void testGetRemainingAmount() throws InsufficientMoneyException, InvalidMoneyException {
        Money contribution = Money.createMoney(Currency.RUPEE, value: 4000);
        financialGoal.addContribution(contribution);

        Money remainingAmount = financialGoal.getRemainingAmount();

        assertThat(remainingAmount.getValue(), is( value: 6000.0));
        assertThat(remainingAmount.getCurrency(), is(Currency.RUPEE));
    }
}
```

This mutant replaces the return statement `return targetAmount.subtract(currentAmount);` by `return currentAmount;`

`getRemainingAmount` method in the original code returns 10k - 4k = 6k, while the mutant will not subtract and will return 4k.

Replace return value with Collections.emptyMap

```java
public class Account {
    1 usage
    private final String name;
    5 usages
    private final Wallet wallet;
    5 usages
    private final List<Transaction> transactions;
    5 usages
    private final Map<String, Budget> budgets;

    2 usages  Nishtha Paul
    public Account(String name, Currency defaultCurrency) {
        this.name = name;
        this.wallet = new Wallet();
        this.transactions = new ArrayList<>();
        this.budgets = new HashMap<>();
    }

    1 usage  new *
    public void setBudget(String category, Money limit) { budgets.put(category, new Budget(category, limit)); }

    2 usages  new *
    public Map<String, Budget> getBudgets() { return new HashMap<>(budgets); }
```

```java
public class AccountTest {

    3 usages
    private Account account;
    Nishtha Paul
    @Before
    public void setup() {
        account = new Account( name: "Test Account", Currency.RUPEE);
    }
    new *
    @Test
    public void testGetBudgets() throws InvalidMoneyException {
        Money budgetLimit = Money.createMoney(Currency.RUPEE, value: 5000);
        String category = "Food";
        account.setBudget(category, budgetLimit);

        Map<String, Budget> budgets = account.getBudgets();

        assertThat(budgets, is(not(Collections.emptyMap())));
        assertThat(budgets.containsKey(category), is( value: true));
    }
```

```
1. replaced return value with Collections.emptyMap for org/example/account/Account::getBudgets → KILLED
2. removed call to java/util/HashMap::<init> → KILLED
```

The mutant has replaced the return statement

```
return new HashMap<>(budgets);
```

by

```
return Collections.emptyMap();
```

Removed assignment to member variable

```java
public class Wallet {
    9 usages
    private Money money;
    3 usages  Nishtha Paul
    public Wallet() {}
    3 usages  Nishtha Paul
    public Money withdraw(Money money) throws InsufficientMoneyException {
        if (this.money == null) {
            throw new InsufficientMoneyException();
        }
        if (this.money.isLessThan(money)) {
            throw new InsufficientMoneyException();
        }
        this.money = this.money.subtract(money);
        return money;
    }
```

```java
@Test
public void shouldReflectChangeInWalletMoneyAfterWithdrawing() throws InsufficientMoneyException, InvalidMoneyException {
    Money fiveRupee = createMoney(Currency.RUPEE, value: 5);
    Wallet wallet = new Wallet();
    wallet.deposit(fiveRupee);

    Money twoRupee = createMoney(Currency.RUPEE, value: 2);
    wallet.withdraw(twoRupee);

    Money totalAmount = wallet.totalAmount(Currency.RUPEE);
    Money threeRupee = createMoney(Currency.RUPEE, value: 3);
    assertThat(totalAmount, is(equalTo(threeRupee)));
}
```

```
1. Removed assignment to member variable money → KILLED
2. replaced call to org/example/models/Money::subtract with receiver → KILLED
3. removed call to org/example/models/Money::subtract → KILLED
4. replaced call to org/example/models/Money::subtract with argument → KILLED
```

This mutant replaces the highlighted line

`this.Money = this.money.subtract(money);`

by

`this.money.subtract(money)`

In the original code, this test case will withdraw the amount and update it in the wallet's money, while in the mutant, the test case will only withdraw the money, and it will not be updated in the wallet. Thus, they will give different outputs, and the mutant will be killed.

Conclusion

By leveraging mutation testing, we ensure higher code quality and increased confidence in the application’s reliability, especially for critical financial use cases.