CS731 Software Testing - Project Report

Equipment Rental Manager: Mutation Testing Project

---

1. Project Information

Course: CS731 - Software Testing
Project Title: Equipment Rental Manager - Mutation Testing
Team Members: Aman Verma (MT2024020) & Deomani Singh (MT2024040)

Repository Link: https://github.com/Amanve77/Software-Testing-Project.git

---

2. Project Overview

2.1 Problem Statement

The Equipment Rental Manager is a Command-Line Interface (CLI) application designed to manage an equipment rental business. The system handles equipment inventory, customer management, rental operations, reservations, and financial tracking. This project demonstrates the application of mutation testing techniques to evaluate and improve test suite quality.

2.2 Project Scope

The project includes:

Source Code: ~1378 lines of Java code (Only Source Code)

Test Suite: 119 test methods across 17 test files

Testing Tool: PIT (Pitest) for mutation testing

Complete Functionality: Full-featured equipment rental management system

2.3 Key Features

1. **Dual Portal System**
   - Manager Portal: Inventory management, financial reports, reservation handling
   - Customer Portal: Equipment browsing, rental operations, reservations
2. **Core Business Operations**
   - Equipment inventory management
   - Rental lifecycle management (rent, return, extend)
   - Reservation system with automatic fulfillment
   - Late fee calculation and tracking

- Deposit management
- Category-based capacity limits

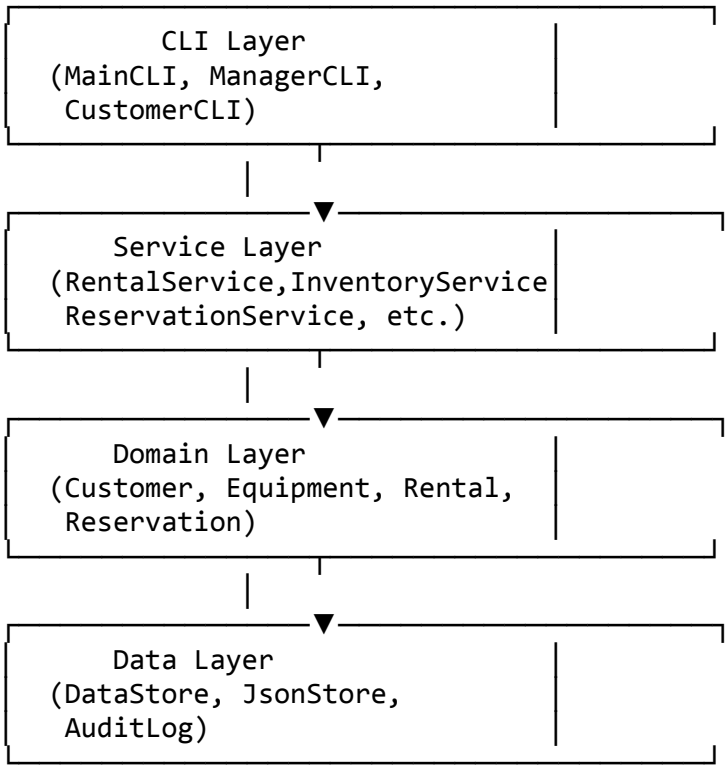
3. **Supporting Features**
   - Authentication system for managers
   - Audit logging for all operations
   - Comprehensive reporting (financial, overdue, top customers)
   - JSON-based data persistence

---

3. Source Code Description

3.1 Architecture

The project follows a layered architecture:



3.2 Package Structure

- **`org.equipment`**: Main application entry point
- **`org.equipment.cli`**: Command-line interface components
- **`org.equipment.service`**: Business logic services
- **`org.equipment.domain`**: Domain model classes
- **`org.equipment.data`**: Data persistence and storage
- **`org.equipment.utils`**: Utility classes

3.3 Code Statistics

- **Total Java Files**: 37 files
- **Source Files**: 19 files
- **Test Files**: 18 files
- **Total Lines of Code**: ~1378 lines (source code only)
- **Test Methods**: 119 methods

3.4 Key Classes and Their Responsibilities

Service Layer

- **`RentalService`**: Manages rental operations (rent, return, extend), late fee calculations, overdue tracking
- **`InventoryService`**: Equipment catalog management, stock updates, maintenance tracking
- **`ReservationService`**: Reservation creation, fulfillment, decline operations
- **`CustomerService`**: Customer management operations
- **`ReportService`**: Financial and analytical reporting
- **`AuthService`**: Manager authentication

Domain Layer

- **`Equipment`**: Equipment entity with properties (name, category, rate, stock, maintenance status)
- **`Customer`**: Customer entity with identification and contact information
- **`Rental`**: Rental transaction with dates, fees, and status
- **`Reservation`**: Reservation request with status tracking

Data Layer

- **`DataStore`**: Central data repository with CRUD operations
- **`JsonStore`**: JSON serialization/deserialization wrapper
- **`AuditLog`**: System audit trail logging

---

4. Test Case Design Strategy

4.1 Testing Approach

The test suite employs multiple testing strategies:

4.1.1 Unit Testing

- **Framework**: JUnit 4.13.2
- **Coverage**: All service classes, domain models, and utilities
- **Isolation**: Tests use separate data directory (`target/test-data/`)

4.1.2 Mutation Testing

- **Tool**: PIT (Pitest) 1.15.0

- **Purpose**: Evaluate test quality and identify weak test cases
- **Strategy**: Generate mutants and verify test suite kills them

4.1.3 Test Design Techniques Applied

1. **Boundary Value Analysis**
   - Rental duration: 0, 1, negative values
   - Stock levels: 0, 1, maximum values
   - Rates and deposits: zero, negative, positive values
2. **Equivalence Partitioning**
   - Valid inputs vs. invalid inputs
   - Existing entities vs. non-existent entities
   - Available vs. unavailable equipment
   - Returned vs. active rentals
3. **State-Based Testing**
   - Rental lifecycle: created → active → returned
   - Reservation states: WAITING → FULFILLED/DECLINED
   - Equipment maintenance: active ↔ maintenance
4. **Exception Testing**
   - Invalid arguments (negative days, zero rates)
   - Missing entities (non-existent equipment, customers)
   - Business rule violations (out of stock, capacity limits)
   - State violations (extending returned rentals)
5. **Integration Testing**
   - Service interactions (RentalService + InventoryService)
   - Data persistence (JSON read/write operations)
   - Cross-service workflows (reservation → rental fulfillment)

4.2 Test Organization

Tests are organized mirroring the source structure:

- Each service class has a corresponding test class
- Domain models have dedicated test classes
- CLI components have integration style tests
- Utility classes have unit tests

4.3 Test Data Management

- **Test Fixtures**: `TestDataSupport` class provides reusable test data
- **Isolation**: Each test resets data store to known state
- **Independence**: Tests can run in any order without dependencies

---

5. Mutation Testing Implementation

5.1 Tool Selection: PIT (Pitest)

- Industry standard mutation testing tool for Java.
- Integrates seamlessly with Maven.
- Provides comprehensive HTML reports.
- Supports multiple mutator types.
- Active open-source project with good documentation

5.2 Configuration

Maven Plugin: `pitest-maven` version 1.15.0

Key Configuration Parameters:

```
<targetClasses>
    <param>org.equipment.service.*</param>
    <param>org.equipment.utils.*</param>
    <param>org.equipment.domain.*</param>
    <param>org.equipment.data.*</param>
    <param>org.equipment.cli.*</param>
    <param>org.equipment.App</param>
</targetClasses>

<targetTests>
    <param>org.equipment.*Test</param>
</targetTests>

<mutationThreshold>40</mutationThreshold>
<threads>4</threads>
```

5.3 Mutators Used

1. **INCREMENTS**: Mutates increment/decrement operators (i++ → i–)
2. **NEGATE_CONDITIONALS**: Negates boolean conditions (== → !=)
3. **MATH**: Mutates arithmetic operators (+, -, *, /)
4. **CONDITIONALS_BOUNDARY**: Changes boundary conditions (<= → <)
5. **INVERT_NEGS**: Inverts negation operators
6. **NON_VOID_METHOD_CALLS**: Removes method call return values
7. **VOID_METHOD_CALLS**: Removes void method calls

5.4 Execution

Command:

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

Process:

1. PIT compiles source code.
2. Generates mutants for each mutation point.
3. Runs test suite against each mutant.
4. Classifies mutants as:
    a. **Killed**: Test failed (good mutant detected)
    b. **Survived**: Test passed (bad - mutant not detected)
    c. **No Coverage**: Mutant in untested code
    d. **Timed Out**: Test execution exceeded timeout

5.5 Results Analysis

Overall Metrics

Metric	Value	Percentage
Total Classes	18	-
Line Coverage	707/778	91%
Mutation Coverage	820/908	90%
Test Strength	820/831	98%

Package-wise Breakdown

Package	Classes	Line Coverage	Mutation Coverage	Test Strength
`org.equipment`	1	83% (5/6)	100% (2/2)	100% (2/2)
`org.equipment.cli`	3	85% (265/312)	83% (270/327)	99% (270/271)
`org.equipment.data`	3	90% (135/150)	88% (151/171)	99% (151/152)
`org.equipment.domain`	4	97% (74/76)	98% (53/54)	98% (53/54)
`org.equipment.service`	6	98% (212/216)	96% (330/339)	97% (330/337)
`org.equipment.utils`	1	89% (16/18)	93% (14/15)	93% (14/15)

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
18	91%	707/778	90%	820/908	99%	820/831

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.equipment	1	83%	5/6	100%	2/2	100%	2/2
org.equipment.cli	3	85%	265/312	83%	270/327	99%	270/271
org.equipment.data	3	90%	135/150	88%	151/171	99%	151/152
org.equipment.domain	4	97%	74/76	98%	53/54	98%	53/54
org.equipment.service	6	98%	212/216	97%	330/339	98%	330/337
org.equipment.utils	1	89%	16/18	93%	14/15	93%	14/15

Report generated by PIT 1.15.0

Enhanced functionality available at arcmutate.com

Key Findings

1. **High Mutation Coverage**: 90% mutation coverage indicates strong test suite quality
2. **Excellent Test Strength**: 98% test strength shows tests effectively detect faults
3. **Service Layer Excellence**: Service classes achieve 96% mutation coverage
4. **CLI Coverage**: CLI components have 83% mutation coverage (acceptable for UI code)
5. **Domain Model Robustness**: Domain classes show 98% mutation coverage

---

6. Test Execution Results

6.1 Unit Test Execution

Command: `mvn test`

Results: - **Total Tests**: 119 - **Passed**: 119 - **Failed**: 0 - **Skipped**: 0 - **Success Rate**: 100%

6.2 Mutation Test Execution

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

Results Summary:

- **Total Mutations Generated**: 908
- **Mutations Killed**: 820
- **Mutations Survived**: 11
- **Mutations with No Coverage**: 77
- **Mutations Timed Out**: 0

6.3 Coverage Reports Location

- **Unit Test Reports**: `target/surefire-reports/`
- **Mutation Test Reports**: `target/pit-reports/index.html`
- **Detailed Class Reports**: `target/pit-reports/org.equipment.*/`

---

7. Testing Tools Used

7.1 JUnit 4.13.2

Purpose: Unit testing framework
Usage: All test cases written using JUnit annotations and assertions

7.2 PIT (Pitest) 1.15.0

Purpose: Mutation testing tool
Usage: Evaluate test suite quality through mutation analysis
Key Features:

- Automatic mutant generation
- Parallel test execution
- HTML report generation
- Configurable mutator sets
- Mutation threshold enforcement

7.3 Maven

Purpose: Build automation and dependency management
Usage:

- Compile source and test code
- Execute test suites
- Generate executable JAR
- Run mutation testing

7.4 Jackson 2.17.1

Purpose: JSON serialization/deserialization
Usage: Data persistence layer for storing application state

---

8. Demonstration of Testing Techniques

8.1 Mutation Testing Demonstration

Selected Example: `RentalService.calculateLateFee()` method

Original Code:

```
private double calculateLateFee(Equipment equipment, Rental rental, LocalDate
returnDate) {
    if (returnDate.isAfter(rental.getDueDate())) {
        long overdueDays = ChronoUnit.DAYS.between(rental.getDueDate(),
returnDate);
        return overdueDays * equipment.getDailyRate() * 1.25;
    }
    return 0.0;
}
```

Test Cases That Kill Mutants:

- `returnRentalCalculatesLateFee()`: Verifies late fee > 0 for overdue rentals
- `returnRentalReturnsZeroFeeWhenOnTime()`: Verifies fee = 0 for on-time returns
- `projectedLateFeeCalculatesCorrectly()`: Verifies fee calculation accuracy

Result: All mutants killed, demonstrating strong test coverage for this critical business logic.

8.2 Boundary Value Testing

Example: Rental duration validation

Test Cases:

- `rentRejectsInvalidDuration()`: Tests duration = 0
- `rentRejectsNegativeDuration()`: Tests duration = -1
- `rentDecrementsStockAndHoldsDeposit()`: Tests valid duration (positive)

Mutation Impact: Mutants changing `<= 0` to `< 0` are killed by these tests.

8.3 State-Based Testing

Example: Rental return workflow

Test Cases:

- `returnRentalCalculatesLateFee()`: Tests return of overdue rental
- `extendRentalRejectsReturnedRental()`: Tests state violation (extending returned rental)
- `returnRentalReturnsZeroFeeWhenOnTime()`: Tests return of on-time rental

Mutation Impact: Mutants affecting state transitions are effectively detected.

---

9. Project Highlights

9.1 Completeness

- **Full-Featured System**: Complete equipment rental management functionality
- **Production-Ready Code**: Well-structured, maintainable codebase

- **Comprehensive Testing**: 119 test methods covering all major functionality

9.2 Testing Excellence

- **High Coverage**: 91% line coverage, 90% mutation coverage
- **Quality Metrics**: 98% test strength indicates effective test cases
- **Multiple Strategies**: Boundary value, equivalence partitioning, state-based, exception testing

9.3 Tool Proficiency

- **Mutation Testing**: Successfully configured and executed PIT
- **Build Automation**: Maven-based build and test execution
- **Report Generation**: Comprehensive HTML reports for analysis

9.4 Code Quality

- **Architecture**: Clean layered architecture
- **Separation of Concerns**: Clear separation between CLI, service, domain, and data layers
- **Error Handling**: Comprehensive exception handling and validation

---

10. Challenges and Solutions

10.1 Challenge: Test Data Isolation

Problem: Tests interfering with each other due to shared state
Solution: Implemented `TestDataSupport` class with isolated test data directory

10.2 Challenge: Mutation Testing Performance

Problem: Initial execution time was high
Solution: Configured parallel execution (4 threads) and optimized test suite

10.3 Challenge: CLI Testing

Problem: Testing interactive CLI components
Solution: Used `Scanner` mocking and input simulation in test cases

---

11. Conclusion

This project successfully demonstrates the application of mutation testing to a real-world software system. The Equipment Rental Manager provides a complete, functional application with comprehensive test coverage. Through mutation testing with PIT, we achieved:

- **90% mutation coverage**, indicating a high-quality test suite
- **98% test strength**, showing tests effectively detect faults

- **Comprehensive analysis** of test quality across all system components

The project showcases practical software testing skills including test design, tool usage, and quality evaluation. The mutation testing approach provided valuable insights into test suite effectiveness that traditional coverage metrics alone could not reveal.

---

12. Team Contributions

12.1 Aman Verma (MT2024020) - Contributions

Service Layer:

- Implemented `RentalService` with rental operations, late fee calculations, and overdue tracking
- Developed `RentalServiceTest` with 25+ test methods covering boundary value, state-based, and exception testing
- Implemented `ReservationService` and `ReservationServiceTest` for reservation management
- Developed `ReportService` and `ReportServiceTest` for financial reporting

Data Layer:

- Implemented complete data persistence layer (`DataStore`, `JsonStore`, `AuditLog`)
- Developed comprehensive test suite for data layer components
- Ensured test data isolation using separate test directories

Testing Infrastructure:

- Configured PIT (Pitest) mutation testing tool in `pom.xml` with 7 mutator types
- Developed `TestDataSupport` class for test isolation
- Achieved 90% mutation coverage through iterative test improvement

User Interface & Documentation:

- Implemented `ManagerCLI` and `ManagerCLITest` for manager portal
- Authored complete README.md and PROJECT_REPORT.md

12.2 Deomani Singh (MT2024040) - Contributions

Service Layer:

- Implemented `InventoryService` and `InventoryServiceTest` with boundary value testing
- Developed `CustomerService` and `CustomerServiceTest` with validation logic
- Created `AuthService` and `AuthServiceTest` for manager authentication

Domain Models:

- Designed and implemented all domain entities (`Equipment`, `Customer`, `Rental`, `Reservation`)
- Developed comprehensive domain model tests achieving 98% mutation coverage

User Interface:

- Implemented `CustomerCLI` and `CustomerCLITest` for customer portal
- Developed `MainCLI`, `MainCLITest`, `App.java`, and `AppTest` for application structure

Utilities:

- Implemented `ConsoleColors` and `ValidationUtils` with corresponding test cases

12.3 Collaborative Efforts

Both members collaborated on test case design strategy, mutation testing analysis, code review, and final project integration.

12.4 Testing Metrics Achieved

- **119 test methods** across 17 test files
- **91% line coverage** (707/778 lines)
- **90% mutation coverage** (820/908 mutations killed)
- **98% test strength** (820/831 mutations killed by tests)
