Mutation Testing Using JUnit and MuClipse

Mutation Testing

- Mutation testing is a software testing technique that involves **modifying code** to create small defects or mutations, called **mutants**, to test the effectiveness of test cases.

- The goal of mutation testing is to determine the quality of the test suite by measuring its ability to detect these mutations.

- The main purpose of mutation testing is to increase the quality and effectiveness of software testing.

- It helps identify gaps in test coverage and highlights areas where test cases may not be detecting defects in the code.

➢ By creating mutants, mutation testing simulates real-world bugs and ensures that test cases can identify these bugs, thus improving the reliability of the software.

➢ - Eg:

Original Java code:

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Mutant:

```
public class Calculator {
    public int add(int a, int b) {
        return a - b;
    }
}
```

Mutation testing in Java

- ➢ **JUnit** and a mutation testing tool that works with JUnit eg: **PIT**(PItest), **Jumble**, etc.
- ➢ JUnit: popular open-source testing framework for Java.
- ➢ It helps to create unit tests.
- ➢ JUnit provides a range of assertion methods for verifying the results of tests.
- ➢ JUnit can be run using command-line tools or integrated into an IDE such as Eclipse or IntelliJ IDEA.

- **Eg:**

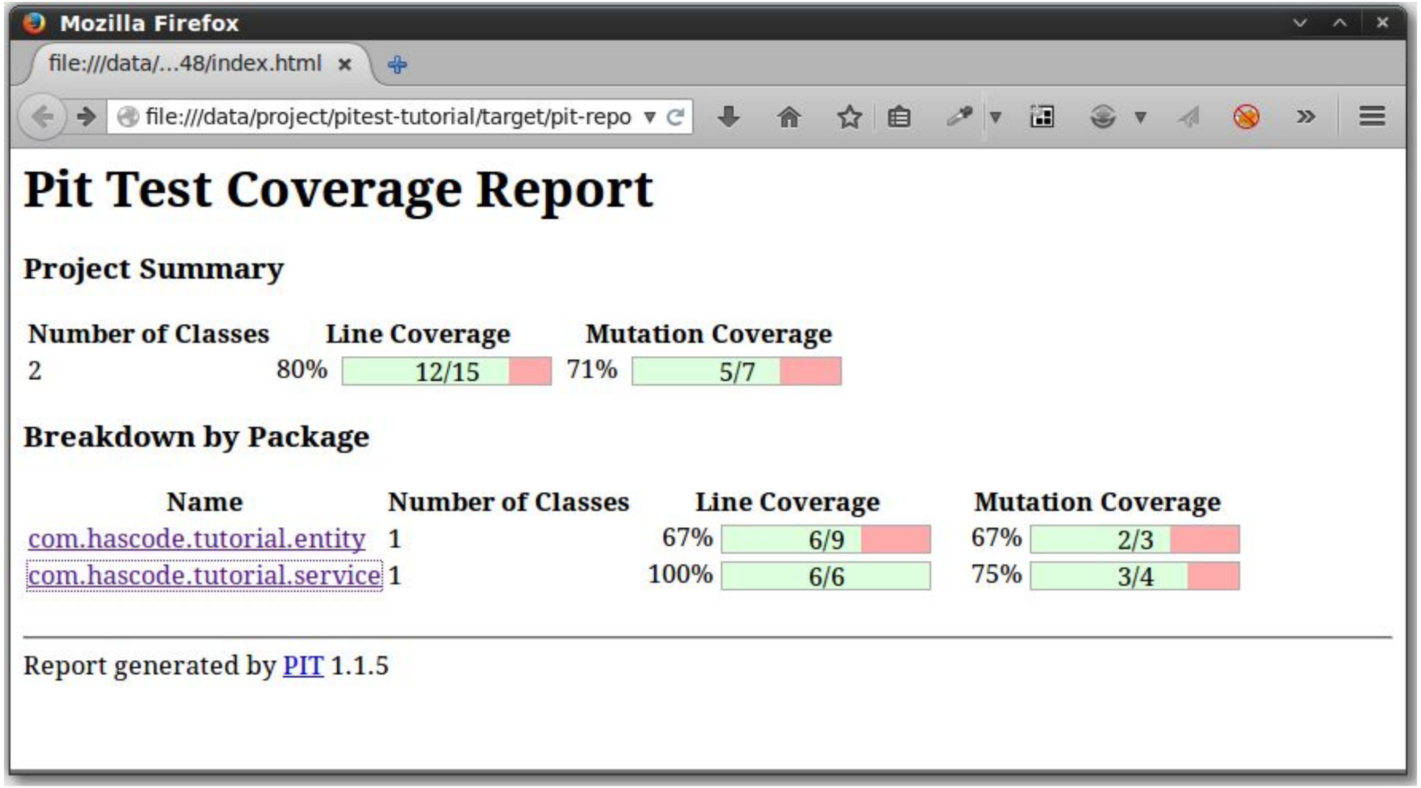
```
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest {
    @Test
    public void testAdd() {
        Calculator calc = new Calculator();
        int result = calc.add(2, 3);
        assertEquals(5, result);
    }
}
```

- **PIT**(Pitest): open-source mutation testing tool for Java that supports JUnit, TestNG, and other test frameworks.
- It helps to create the Mutants automatically.
- Used by many programmers
- PIT provides detailed reports on the mutation testing results, including the number of mutants generated, the percentage of mutants ‘killed’ by the test suite, and the mutation score.
- Result as an HTML file. Contains details like Line coverage, Mutation coverage, etc.
- PIT is actively maintained and has a large community of users and contributors.

Steps in Mutation Testing using JUnit and PIT:

1. Include the dependencies for Junit in configuration file.
2. Include the plugin for PIT in configuration file.
3. Prepare test cases using JUnit.
4. Apply test cases.
5. If successfully executed, generate mutants automatically using PIT.
6. Code coverage and Mutation coverage is performed.
7. Result analysis of ‘Killed’ and ‘survived’ mutants.

Output of Mutation Testing using JUnit and PIT:



Mozilla Firefox
file:///data/...48/index.html
file:///data/project/pitest-tutorial/target/pit-repo

Pit Test Coverage Report

Project Summary

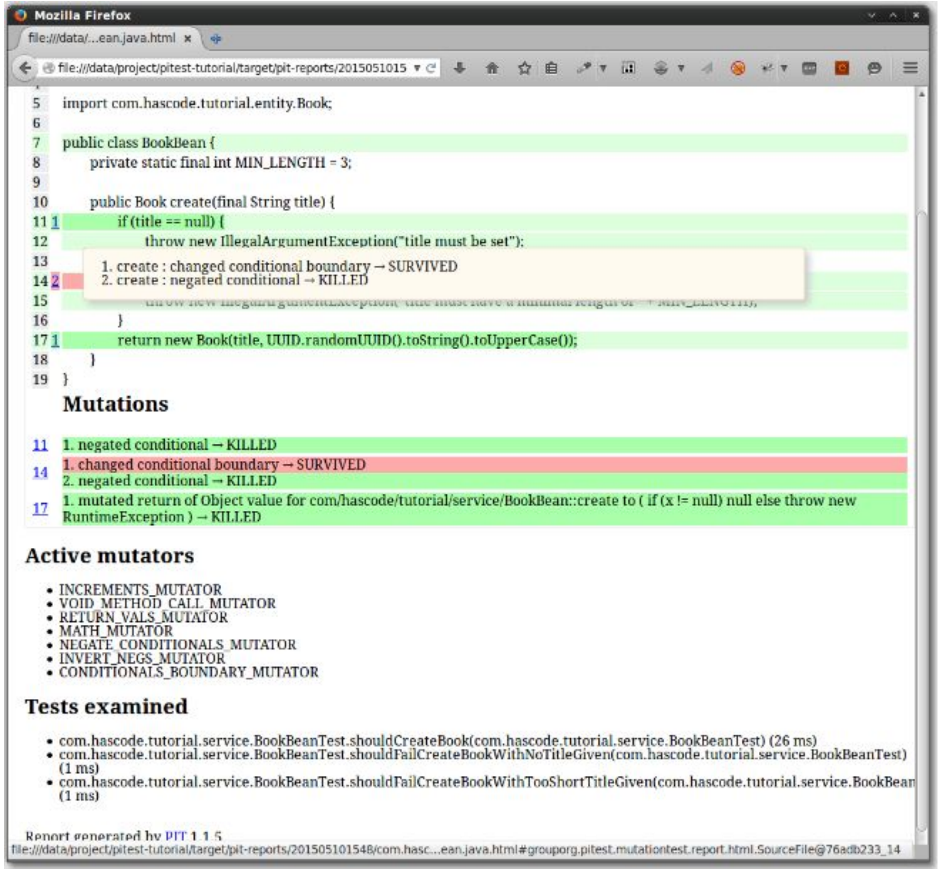
Number of Classes	Line Coverage		Mutation Coverage	
2	80%	12/15	71%	5/7

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
com.hascode.tutorial.entity	1	67%	6/9	67%	2/3
com.hascode.tutorial.service	1	100%	6/6	75%	3/4

Report generated by PIT 1.1.5

Output of Mutation Testing using JUnit and PIT:


Mozilla Firefox
file:///data/...ean.java.html
file:///data/project/pitest-tutorial/target/pit-reports/2015051015
file:///data/project/pitest-tutorial/target/pit-reports/201505101548/com.hasc...ean.java.html#grouporg.pitest.mutationtest.report.html.SourceFile@76adb233_14

```
5      import com.hascode.tutorial.entity.Book;
6
7      public class BookBean {
8          private static final int MIN_LENGTH = 3;
9
10         public Book create(final String title) {
11 1          if (title == null) {
12                 throw new IllegalArgumentException("title must be set");
13
14 2
15                 throw new IllegalArgumentException("title must have a minimal length of " + MIN_LENGTH);
16             }
17 1          return new Book(title, UUID.randomUUID().toString().toUpperCase());
18         }
19     }
```

```
1. create : changed conditional boundary → SURVIVED
2. create : negated conditional → KILLED
```

Mutations

11 1. negated conditional → KILLED

14 1. changed conditional boundary → SURVIVED
2. negated conditional → KILLED

17 1. mutated return of Object value for com/hascode/tutorial/service/BookBean::create to (if (x != null) null else throw new RuntimeException) → KILLED

Active mutators

- INCREMENTS_MUTATOR
- VOID_METHOD_CALL_MUTATOR
- RETURN_VALS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- INVERT_NEGS_MUTATOR
- CONDITIONALS_BOUNDARY_MUTATOR

Tests examined

- com.hascode.tutorial.service.BookBeanTest.shouldCreateBook(com.hascode.tutorial.service.BookBeanTest) (26 ms)
- com.hascode.tutorial.service.BookBeanTest.shouldFailCreateBookWithNoTitleGiven(com.hascode.tutorial.service.BookBeanTest) (1 ms)
- com.hascode.tutorial.service.BookBeanTest.shouldFailCreateBookWithTooShortTitleGiven(com.hascode.tutorial.service.BookBean (1 ms)

Report generated by PIT 1.1.5

Mutation Testing using JUnit and MuClipse:

- ➢ MuClipse is designed to integrate seamlessly with the Eclipse IDE, providing a user-friendly interface for configuring and running mutation tests.
- ➢ MuClipse provides detailed reports on the mutation testing results, including the number of mutants generated, the percentage of mutants killed by the test suite, and the mutation score.
- ➢ It can be used to run mutation tests on individual classes or entire projects, and supports both JUnit and TestNG test frameworks.
- ➢ It provides a range of configuration options, allowing users to customize the mutation testing process to suit their needs.
- ➢ MuClipse is no longer under active development.

- MuClipse does not perform mutation testing on its own.
- It is a plugin for Eclipse that integrates with the PIT (Pitest) mutation testing tool to provide a user-friendly interface for configuring and running mutation tests.
- PIT is responsible for generating the mutated versions of the code and running the test suite against each mutant to determine which tests pass and which fail.
- MuClipse provides a convenient way to configure and run PIT from within the Eclipse IDE, as well as a range of other features such as real-time reporting and test result visualization.
- MuClipse simplifies the process of performing mutation testing

Steps in mutation test using JUnit and MuClipse

1. Install MuClipse plugin in Eclipse from Eclipse Marketplace or by downloading from the MuClipse website.
2. Create a JUnit test suite for the code you want to test.
3. Configure the MuClipse plugin to use the PIT mutation testing tool.
4. Run the mutation tests.
5. Analyze the results.
6. Refine the test suite: adding new test cases or modifying existing ones to improve coverage and detect more faults.
7. Repeat the process

Advantages of using JUnit and MuClipse

- ➢ Easy integration: JUnit and MuClipse are both open-source tools that are easy to integrate into the software development process.
- ➢ Automated testing: JUnit and MuClipse automate the process of testing, making it faster and more reliable.
- ➢ Effective fault detection: Mutation testing using JUnit and MuClipse can effectively detect faults in the code that might be missed by other testing techniques.
- ➢ Test suite refinement: The mutation testing results generated by MuClipse can be used to refine the test suite and improve coverage.

References

- ➢ JUnit installation and setup in intellij idea:

  https://www.youtube.com/watch?v=cTEtSmNOtlE&ab_channel=Randomcode
- ➢ Pitest installation and setup in intellij idea:

  https://www.youtube.com/watch?v=3zbgOSEY2mU&ab_channel=Codemanship
- ➢ Mutation testing using Muclipse:

  https://www.youtube.com/watch?v=xiAGuydHiLE&ab_channel=PatelRaj
- ➢ MuClipse: https://muclipse.sourceforge.net/
- ➢ PIT: https://pitest.org/