Lab VI: Mutation Testing

Software Testing and QA (COE891)

Week 11

1 Lab Objectives

In this lab, you will learn how to use an open-source mutation testing tool called *PITclipse* to execute mutation test for the following purposes:

- Evaluating the effectiveness of test cases by measuring their ability to detect faults (Test Adequacy).
- Uncovering weaknesses in the test suite by introducing mutations to the source code and observing detection capabilities (Fault Detection).
- Identifying areas of code not adequately covered by tests to refine testing strategies and enhance software reliability (Code Quality Improvement).

In order to review general concepts and read a brief tutorial on mutation testing, please see the Appendices at the end of this document.

2 PITclipse: A Mutation Testing Tool

Mutation testing tools can help speed up the mutant generation process. Tools can also create reports showing killed and live mutations and no coverage, timeouts, memory-related and run-time errors. PIT (Parallel Isolated Test) is a state-of-the-art mutation testing system, providing gold standard test coverage for Java and the JVM. It is fast, scalable and it integrates with modern test and build tooling. PIT runs your unit tests against automatically modified versions of your application code. When the application code changes, it should produce different results and cause the unit tests to fail. If a unit test does not fail with a mutated code, it can be an indicator of a fault.

There are several mutation testing tools but compared to PIT, they are slower and more difficult to use.

- PIT is fast and can analyse in minutes what would take earlier systems days.
- It is easy to use - works with `ant, maven, gradle` and others.
- It is actively developed and supported.

PIT produces reports in an easy-to-read format, combining line coverage and mutation coverage. The most effective way to use mutation testing is to run it **frequently** against only the code that has been changed.

2.1 PIT's Main Features

- Reliability: Relies on PIT (Pitest).
- Customization: Provides numerous preferences to tailor analysis.
- JUnit Support: Works with both JUnit 4 and JUnit 5 tests.

2.2 Installation

This plug-in is available in the Eclipse Marketplace. You can use this link to install the plug-in or manually:

1. Open Eclipse IDE.
2. Go to Help > Eclipse Marketplace...
3. Search for Pitclipse and click *install* button.

2.3 Usage

Once the plug-in is installed, you can run Pitest:

1. Right-click on a Java project defining unit tests (test class).
2. Run As > PIT Mutation Test.

Wait a few seconds, two views should open to show the results:

- **PIT Summary:** Shows the percentage of mutation coverage.
- **PIT Mutations:** Shows the detected mutations and their location in code.

It is also possible to run a single JUnit test class. Specific PIT options can be configured from the Launch Configuration window:

1. Run > Run Configurations...
2. Click on PIT Mutation Test.
3. Specify the options (e.g., Mutators) and press Apply and then Run.
4. Preferences also allow to change mutation settings (Window > Preferences > Pitest).

2.4 Example

Consider the following Java class `Calculation` with its associated test class `CalculationTest`:

Calculation Program:

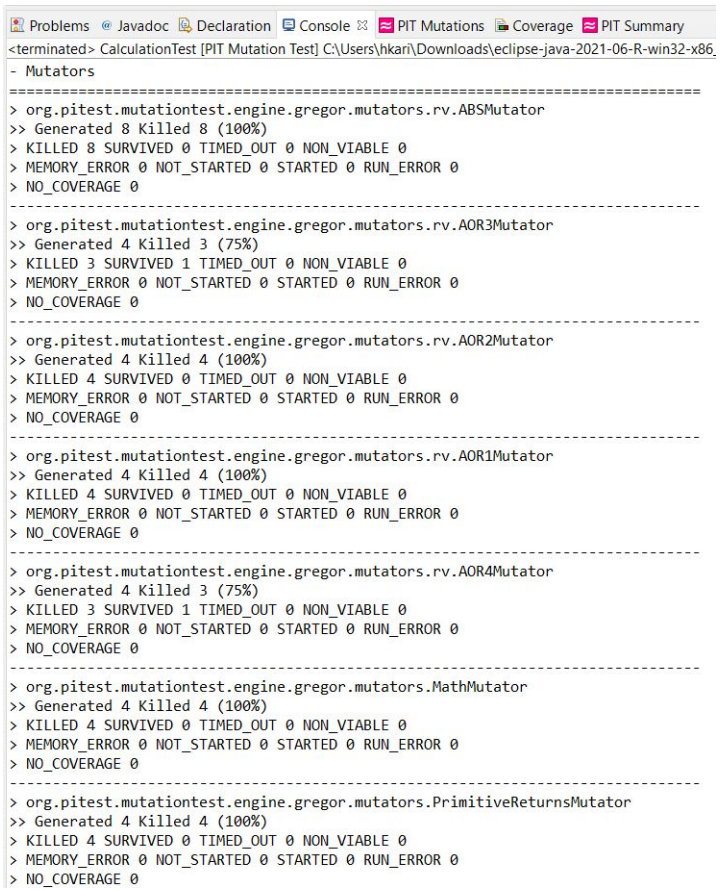
```
1  public class Calculation {
2   public static void main(String[] args) {
3    // TODO Auto-generated method stub
4   }
5   public double add(int a, int b) {
6    return a + b;
7   }
8   public double subtract(int a, int b) {
9    return a - b;
10  }
11  public double multiply(int a, int b) {
12   return a * b;
13  }
14  public double divide(int a, int b) {
15   return a / b;
16  }
17 }
```

CalculationTest Program:

```
1  import static org.junit.Assert.*;
2  import org.junit.*;
3
4  public class CalculationTest {
5        Calculation calculate = new Calculation();
6           @Before
```

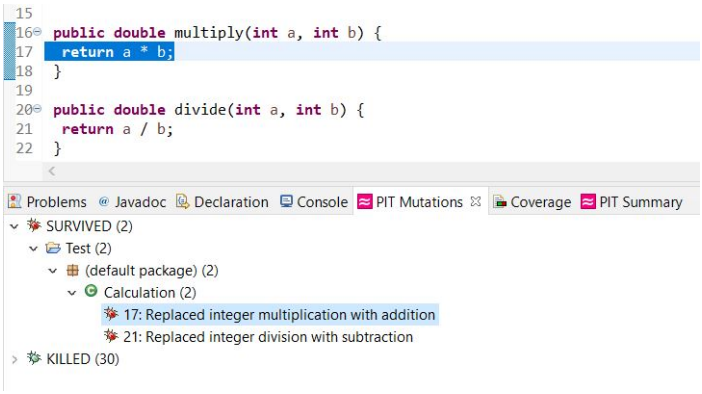
```
7           public void setUp() throws Exception {
8           }
9           @After
10          public void tearDown() throws Exception {
11          }
12          @Test
13          public void addTest() {
14           assertEquals(3, calculate.add(2,1),0.01);
15          }
16          @Test
17          public void subtractTest() {
18           assertEquals(1, calculate.subtract(2,1),0.01);
19          }
20          @Test
21          public void multiplyTest() {
22           assertEquals(4, calculate.multiply(2,2), 0.01);
23          }
24          @Test
25          public void divideTest() {
26           assertEquals(2, calculate.divide(4,2), 0.01);
27          }
28  }
```

Then, when you run the test class with PIT Mutation Test, you can see the Mutation Information, Mutators, Timings, and Statistics (including the Mutation Score) in the console of Eclipse IDE as shown in the following figures:

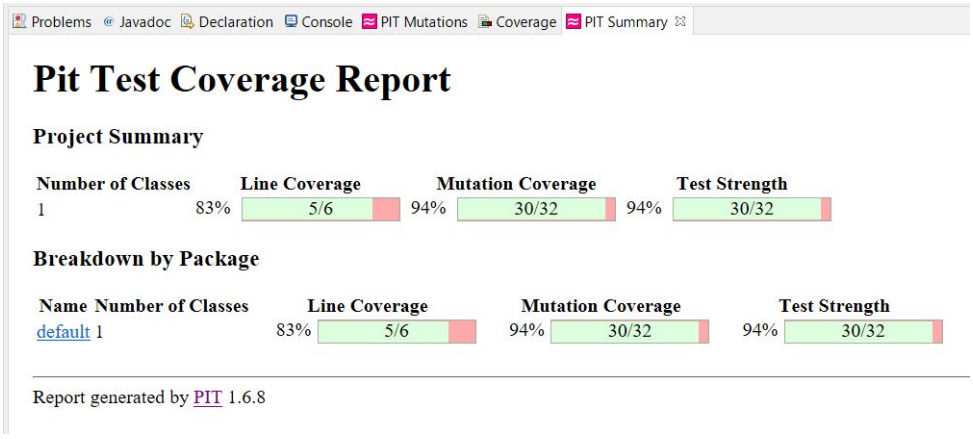


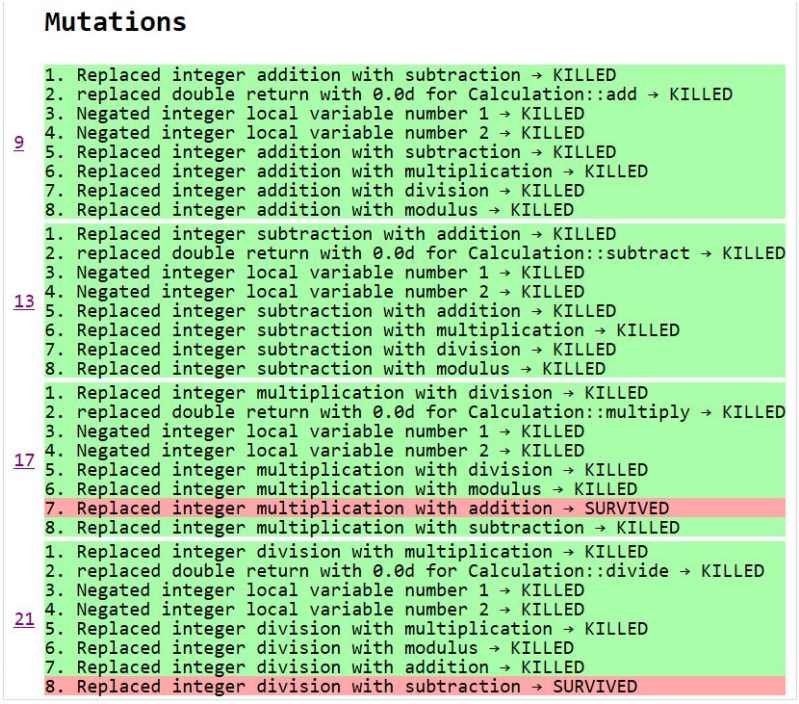
```
================================================================================
- Timings
================================================================================
> pre-scan for mutations : < 1 second
> scan classpath : < 1 second
> coverage and dependency analysis : < 1 second
> build mutation tests : < 1 second
> run mutation analysis : 1 seconds
--------------------------------------------------------------------------------
> Total  : 2 seconds
--------------------------------------------------------------------------------
================================================================================
- Statistics
================================================================================
>> Line Coverage: 5/6 (83%)
>> Generated 32 mutations Killed 30 (94%)
>> Mutations with no coverage 0. Test strength 94%
>> Ran 32 tests (1 tests per mutation)
```

In the PIT Mutations tab, you can see the type, location (double-click on the tests), and information of killed, survived, and uncovered tests:



In the PIT Summary tab, you can find a detailed report on your mutation testing coverage generated by PIT as shown in the following figures. This PIT Test report shows line coverage, mutation coverage, and test strength for each class. By clicking on each class name, you can see the detail of used mutations (including killed, survived, and uncovered mutants), active mutators (mutation types), and methods (units) that are tested by the mentioned mutants.





Mutations

9	1. Replaced integer addition with subtraction → KILLED 2. replaced double return with 0.0d for Calculation::add → KILLED 3. Negated integer local variable number 1 → KILLED 4. Negated integer local variable number 2 → KILLED 5. Replaced integer addition with subtraction → KILLED 6. Replaced integer addition with multiplication → KILLED 7. Replaced integer addition with division → KILLED 8. Replaced integer addition with modulus → KILLED
13	1. Replaced integer subtraction with addition → KILLED 2. replaced double return with 0.0d for Calculation::subtract → KILLED 3. Negated integer local variable number 1 → KILLED 4. Negated integer local variable number 2 → KILLED 5. Replaced integer subtraction with addition → KILLED 6. Replaced integer subtraction with multiplication → KILLED 7. Replaced integer subtraction with division → KILLED 8. Replaced integer subtraction with modulus → KILLED
17	1. Replaced integer multiplication with division → KILLED 2. replaced double return with 0.0d for Calculation::multiply → KILLED 3. Negated integer local variable number 1 → KILLED 4. Negated integer local variable number 2 → KILLED 5. Replaced integer multiplication with division → KILLED 6. Replaced integer multiplication with modulus → KILLED 7. Replaced integer multiplication with addition → SURVIVED 8. Replaced integer multiplication with subtraction → KILLED
21	1. Replaced integer division with multiplication → KILLED 2. replaced double return with 0.0d for Calculation::divide → KILLED 3. Negated integer local variable number 1 → KILLED 4. Negated integer local variable number 2 → KILLED 5. Replaced integer division with multiplication → KILLED 6. Replaced integer division with modulus → KILLED 7. Replaced integer division with addition → KILLED 8. Replaced integer division with subtraction → SURVIVED

Active mutators

- ABS_MUTATOR
- AOR_1_MUTATOR
- AOR_2_MUTATOR
- AOR_3_MUTATOR
- AOR_4_MUTATOR
- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INLINE_CONSTANT_MUTATOR
- MATH_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR

Tests examined

- CalculationTest.subtractTest(CalculationTest) (1 ms)
- CalculationTest.divideTest(CalculationTest) (1 ms)
- CalculationTest.multiplyTest(CalculationTest) (1 ms)
- CalculationTest.addTest(CalculationTest) (18 ms)

You can modify the types of mutations (active mutators) that you want to use in your mutation testing for the purpose of achieving the highest possible mutation coverage score and test strength. You can choose your required active mutators by following the steps below:

1. In your Eclipse IDE under Package Explorer, right-click on your test class > Run As > Run Configurations...
2. Under PIT Mutation Test > Under Mutators tab.

Then, you can check/choose every mutator type that you need to be applied to your mutation testing by checking the required checkboxes > Apply > Run. You can find the full details and required information on the list of available mutators and their associated groups using this List of Mutators. This documentation is highly crucial for choosing the proper and required types of active mutators. It is obvious that choosing *improper* or *excessive* mutators for your specific application can give rise to obtaining weaker test coverage with lower mutation score. Note that in order to increase your mutation coverage score and test strength, you first need to increase the line coverage (statement coverage) in your production code by adding or modifying the required test cases in your test classes to cover as many statements as possible in your production code

(refer to Lab 2 contents and instructions), then you are prepared to choose and include the proper active mutators to achieve the highest possible mutation score.

By default, PITclipse uses two types of mutators: *Math Mutators* (replacing math operators with each other) and *Primitive Returns Mutators* (returning 0 instead of the existing values). However, in this example, you can see that out of 32 mutant programs (as shown in *Mutators* section in Console and *Mutations* section in PIT report, we have 4 test methods and each method has 8 mutations based on different selected mutation types), 30 mutant programs are killed and 2 are survived which leads to 94% mutation coverage score. You may be able to obtain 100% mutation coverage score by using less mutators.

3 Assignments

Q1. For this task, first read Section 2 carefully and thoroughly. Import (automatically or manually) the previously-used (in lab 2) project file named *Money.zip* to your Eclipse IDE and then perform the following tasks:

1. Based on what you have already done in lab 2 and the associated feedback you received during your lab 2 demo, modify the code by adding the required pieces of code and/or test cases to adequately increase the statement/line coverage to near or equal to 100%. You may use or change your modified code from your lab 2 for this part.
2. Using PITclipse and its default mutators, run a PIT mutation test (initial PIT test) to check both test classes `MoneyTest` and `MoneyBagTest`. Report the line coverage and mutation coverage, number of killed, survived, and uncovered mutants, and submit the PIT testing information in PIT Summary section.
3. Now, using your initial PIT testing and detecting the survived and/or uncovered mutants, change the default mutators to add additional mutators for both classes and adequately increase the total mutation coverage score and test strength as much as possible.
   You need to select the proper mutators based on your code and existing test cases (the default and added test cases in Part 1). Note that excessive or improper selection of mutation types may give rise to lower mutation coverage score.
4. Provide a list or a screenshot (in your PDF report) of the set of your own selected mutation types that achieves the highest mutation score.
5. Report the mutation scores, active mutators, number of killed, survived, uncovered mutants, and any other information regarding your PIT test for both classes.
6. Compare the achieved PIT results with the initial PIT results and analyze the PIT results in terms of mutation coverage score, added mutators, and the number and type of killed, survived, and uncover mutants.
7. Report and describe which (and how many) initially survived and uncovered mutants are killed by your own selected set of mutations (mutation types) to increase the total mutation coverage score?
8. Moreover, describe why you could not handle (kill) the remaining survived or uncovered mutants if there is any (could not obtain 100% mutation score)?

Note that you may submit screenshots for your reports along with their corresponding descriptions.

4 Submission and Marking Instructions

4.1 Submissions

Once all required tasks are completed, you should submit your lab assignment. Follow the instructions below for a valid submission:

- After checking the accuracy and completeness of your assignment tasks, you should export and submit the FULL Eclipse project/folder as a ZIP file containing packages and source code files (for implementation and coding tasks/questions).

  - Individual files (e.g. java or class files) will NOT be accepted as a valid submission since your submitted package or Java project should be run completely and successfully by itself.

- You MUST submit a single PDF file containing required description or explanation for each of the assignment tasks/questions separately including any required justification, graphs, diagrams, test requirements/cases, calculation, pictures/snapshots from tools or IDE, test results, and so forth.

- The **submission deadline** for this lab (Lab VI) is the corresponding lab session in **Week 12**.

- The lab demo and questioning-answering will be held during the lab sessions of the corresponding submission weeks.

4.2 Marking Scheme

- This lab (Lab VI) constitutes 4% of your entire grade for this course.

- All assignment tasks/questions in each lab have the same grade.

- The grade for each lab is constituted from 50% for the lab submissions, 10% for the lab attendance, and 40% for demo and questioning-answering during the lab session.

- Note that all the labs constitute 25% of your entire grade for this course.

A What is Mutation Testing?

Mutation testing, also known as code mutation testing, is a form of white box testing in which testers change specific components of an application's source code to ensure a software test suite can detect the changes. Changes introduced to the software are intended to cause errors in the program. Mutation testing is designed to ensure the quality of a software testing tool, not the applications it analyzes. Mutation testing is typically used to conduct unit tests. The goal is to ensure a software test can detect code that isn't properly tested or hidden defects that other testing methods don't catch. Changes called mutations can be implemented by modifying an existing line of code. For example, a statement could be deleted or duplicated, true or false expressions can be changed or other variables can be altered. Code with the mutations is then tested and compared to the original code. If the tests with the mutants detect the same number of issues as the test with the original program, then either the code has failed to execute, or the software testing suite being used has failed to detect the mutations. If this happens, the software test is worked on to become more effective. A successful mutation test will have different test results from the mutant code. After this, the mutants are discarded. The software test tool can then be scored using the mutation score. The mutation score is the number of killed mutants divided by the total number of mutants, multiplied by 100.

$$\text{Mutation Score} = \frac{\text{Number of Killed Mutants}}{\text{Total Number of Mutants Killed or Surviving}} \times 100$$

A mutation score of 100% means the test was adequate.

B Mutation Testing Procedure

A mutation is a small syntactic change made to a program statement. Mutations typically contain one variable that causes a fault or bug. For example, a mutation could look like the statement (A<B) changed to (A>B). Testers intentionally introduce mutations to a program's code. Multiple versions of the original program are made, each with its own mutation, or mutants. The mutants are then tested along with the original application. After testing, testers compare the results to the original program test.

Once the testing software has been fixed, the mutants can be kept and reused in another code mutation test. If the test results from the mutant code to the original programs are different, then the mutants can be discarded, or killed. Mutants that are still alive after running the test are typically called live mutants, while those killed after mutation testing are called killed mutants. Equivalent mutants have the same meaning as the original source code even though they have different syntax. Equivalent mutants aren't calculated as part of the mutant score.

Following are the steps to execute mutation testing (mutation analysis):

1. Faults are introduced into the source code of the program by creating many versions called mutants. Each mutant should contain a single fault (one error), and the goal is to validate the test's efficiency and cause the mutant version to fail which demonstrates the effectiveness of the test cases.
2. Test cases are applied to the original program and also to the mutant program. A Test Case should be adequate, and it is tweaked to detect faults in a program.
3. Compare the results of an original and mutant program.
4. If the original program and mutant programs generate the different output, then the mutant is killed by the test case. Thus, the test case is good enough to detect the change between the original and the mutant program.
5. If the original program and mutant program generate the same output, Mutant is kept alive. In such cases, more effective test cases need to be created to kill all mutants.

B.1 Types of Mutation Testing

There are three main types of mutation testing:

- **Statement Mutation:** Statements are deleted or replaced with a different statement, i.e., developers cut and paste a part of a code of which the outcome may be a removal of some lines. For example, the statement "A=10 by B=5" is replaced with "A=5 by B=15."
- **Value Mutation:** Values are changed to find errors, i.e., values of primary parameters are modified. For example, "A= 15" is changed to "A=10" or "A=25."
- **Decision Mutation:** Arithmetic or logical operators are changed to detect errors, i.e., control statements are to be changed. For example, "(A<B)" is changed to "(A>B)."

B.1.1 An Example

Original Program:

```
if(x>y)
    print "Hello"
else
    print "Hi"
```

Mutant Program:

```
if(x<y)
    print "Hello"
else
    print "Hi"
```

B.2 How to Generate a Mutant Program?

There are several techniques that could be used to generate mutant programs:

B.2.1 Operand Replacement Operators

Replace the operand with another operand (x with y or y with x) or with the constant value. For example:

- In `if(x>y)`, replace x and y values.
- In `if(5>y)`, replace x by constant 5.

B.2.2 Expression Modification Operators

Replace an operator or insertion of new operators in a program statement. For example:
In `if(x==y)`, we can replace == into >= and have mutant program as `if(x>=y)` and/or inserting ++ in the statement as `if(x==++y)`.

B.2.3 Statement Modification Operators

Programmatic statements are modified to create mutant programs. For example:

- Delete the else part in an if-else statement.
- Delete the entire if-else statement to check how a program behaves.
- Some of sample mutation operators:
  - GOTO label replacement
  - Return statement replacement
  - Statement deletion
  - Unary operator insertion (Like – and ++)

- Logical connector replacement
- Comparable array name replacement
- Removing of else part in the if-else statement
- Adding or replacement of operators
- Statement replacement by changing the data
- Data modification for the variables
- Modification of data types in the program
- Automation of mutation testing

C Advantages and Disadvantages

Code mutation provides the following advantages:

- Helps to ensure the identification of weak tests or code.
- Offers a high level of error detection.
- Increases the use of object-oriented frameworks and unit tests if an organization uses them.
- Offers more mutation testing tools due to the increased frameworks and unit tests.
- Helps organizations determine the usefulness of their testing tool through the use of scoring.
- Uncovers ambiguities in the source code and has the capacity to detect all the faults in the program. Customers are benefited from this testing by getting a most reliable and stable system.

Disadvantages of code mutation include the following:

- Is extremely time-consuming, costly, and complicated to execute manually due to the large number of mutants being generated and tested. To speed up the process, it is advisable to go for automation tools. Automation tools reduce the cost of testing as well.
- As this method involves source code changes, it is not at all applicable for Black Box Testing.