Software Testing
Course's Code: CSE 453
Test Design Techniques – Structure-based (White-Box) Technique – Mutation Technique
(Chapter 4)

What we will learn?

- ➢ How to measure the quality of our tests by using a very powerful tool like mutation testing?
  - ❑ Quality of tests means fault detection capability of the test suite

White-Box Testing Strategies

- Coverage-based:
  - Design test cases to cover certain program elements.
- Fault-based:
  - Design test cases to expose some category of faults

White-Box Testing

- Several white-box testing strategies have become very popular :
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Condition coverage
  - MC/DC coverage
  - Mutation testing
  - Data flow-based testing

- ➢ Mutation testing is an example of Fault-based testing

How good are the tests?

- One way to assess the test code quality consists in using some adequacy criteria

Adequacy criteria measure how thoroughly our test suite exercises the program under analysis

- Most common adequacy criteria are the coverage criteria for white-box testing,
    - like for example statement coverage, branch coverage, condition coverage, mc/dc coverage or path coverage

- We can also think of using black-box criteria, like for example state transition coverage

- However, we have to remember that test or code coverage is a necessity but not a sufficient condition to write proper test cases.

Test Coverage

Test coverage refers to how well the number of tests executed cover the functionality of an application

Black-Box Testing Mentality

Code Coverage

Code coverage refers to which application code is exercised when the application is running

White-Box Testing Mentality

Example

- To understand this concept let us consider an example

```java
public class Division {
    public static int[] getValues(int a, int b){
        if (b == 0){
            return null;
        }
        int quotient = a / b;
        int remainder = a % b;

        return new int[] {quotient, remainder};
    }
}
```

- This example has a small class with one single method.
- Such a method takes as input two integers and computes the quotient and the remainder of the division.
- Furthermore, the division cannot be computed when the denominator is equal to 0.
    - This case is also checked in the code by the if condition

How good are the tests?

- Now, let's assume that we choose branch coverage as the driving adequacy criterion

- Based on this choice, we wrote these two test cases.
    - The first one exercises the case where the denominator is not equal to 0;
    - The second one exercises the case where the denominator is 0.
- Based on coverage, we would conclude that this two tests cases are adequate because we reach 100% of branch and statement coverage

```java
@Test
public void testGetValues(){
    int[] values = Division.getValues(1, 1);
    assertEquals(1, values[0]);
    assertEquals(0, values[1]);
}
```

```java
@Test
public void testZero(){
    int[] values = Division.getValues(1, 0);
    assertNull(values);
}
```

100% Branch Cov.
100% Statement Cov.

How good are the tests?

- Now, let us consider two test cases that exercise the same scenario where the denominator is not equal to 0.

```java
@Test
public void testGetValues_V1(){
    int[] values = Division.getValues(1, 1);
    assertEquals(1, values[0]);
    assertEquals(0, values[1]);
}
```

```java
@Test
public void testGetValues_V2(){
    int[] values = Division.getValues(3, 2);
    assertEquals(1, values[0]);
    assertEquals(1, values[1]);
}
```

- These two tests are very similar to each other:
- They have the same method sequence and the same type of assertions.
- However, they differ on the input values
- Now the question is:
  - which of these two test cases has the best fault detection capability?
- To answer this question, we have to look back at the code

Fault Detection Capability

- Let us assume that we made one mistake when coding this program:
    - instead of computing the division, we use the multiplication operator as highlighted in red color.

```
public class Division {
    public static int[] getValues(int a, int b){
        if (b == 0){
            return null;
        }
        int quotient = a * b; // correct = a / b;
        int remainder = a % b;

        return new int[] {quotient, remainder};
    }
}
```

Fault Detection Capability

- If we run our two alternative test cases, we will discover that the first test case will pass while the second one fails.

- Why do we have such a difference in the test results?
- The two tests have the same code coverage and the same type of assertions
- The second test case has better input parameters and oracle (assertions) that allows to detect the injected fault

```java
@Test
public void testGetValues_V1(){
    int[] values = Division.getValues(1, 1);
    assertEquals(1, values[0]);
    assertEquals(0, values[1]);
}
```

```java
@Test
public void testGetValues_V2(){
    int[] values = Division.getValues(3, 2);
    assertEquals(1, values[0]);
    assertEquals(1, values[1]);
}
```

Mutation Testing

- This is the simple yet fundamental idea behind mutation testing

Mutation Testing

- Idea: Inserting artificial defects (mutants) in the production code to assess the quality of the test code
- Effective test suite: at least one of its test cases fails when executing the test suite against the mutants
- Mutation testing is like testing the tests

Mutation Testing

- Mutation Testing is a fault based testing technique
- It is one kind of white-box testing

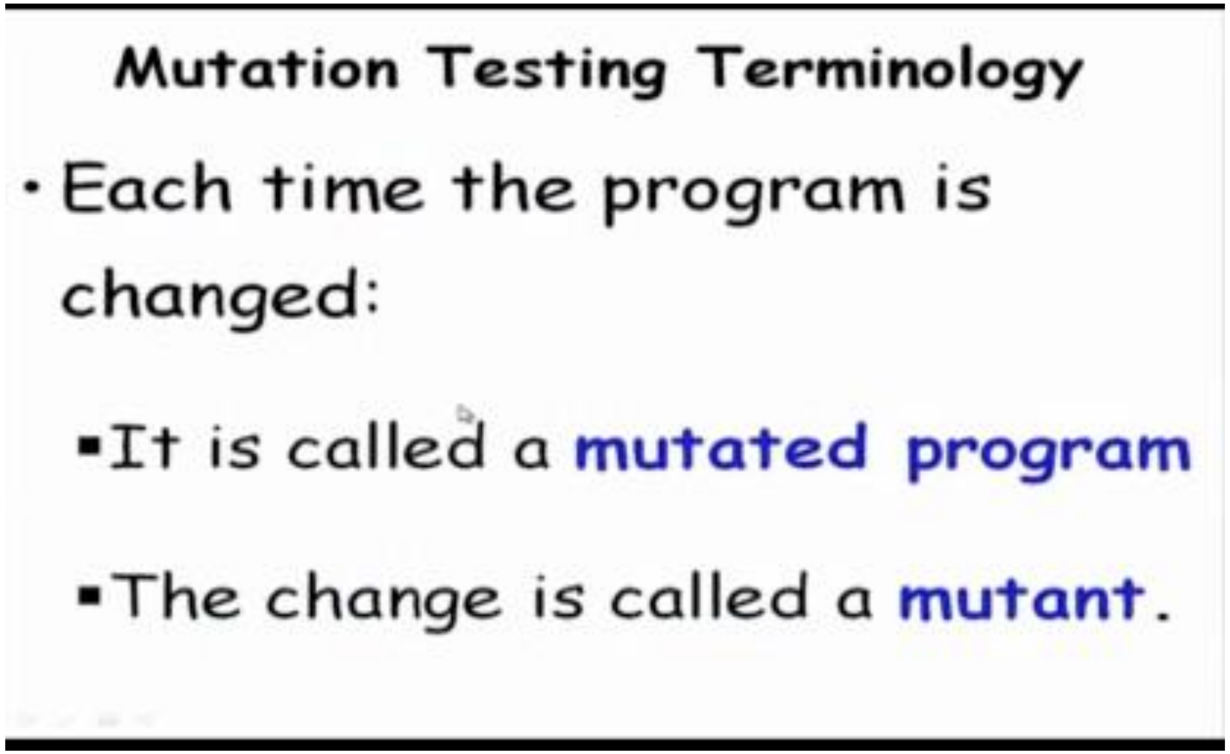
- In this, software is first tested:
  - Using an initial test suite designed using white-box strategies we already discussed.
- After the initial testing is complete,
  - Mutation testing is taken up.
- The idea behind mutation testing:
  - **Make a few arbitrary small changes to a program at a time.**

- A specific type of fault is introduced into the program and then, check whether the test cases are effective against that type of fault.

  - If not, the test case will be augmented with additional test cases to strengthen the test suite, so that the specific type of fault will be detected

Mutation Testing

- **Mutant**: Given a program P, a mutant P’ is obtained by introducing a simple syntactic change to P
- **Syntactic change**: small changes that make the mutated code valid (i.e., it can be compiled)
- **Change**: alterations to the production code that mimic typical human mistakes (glitches)



Mutation Testing

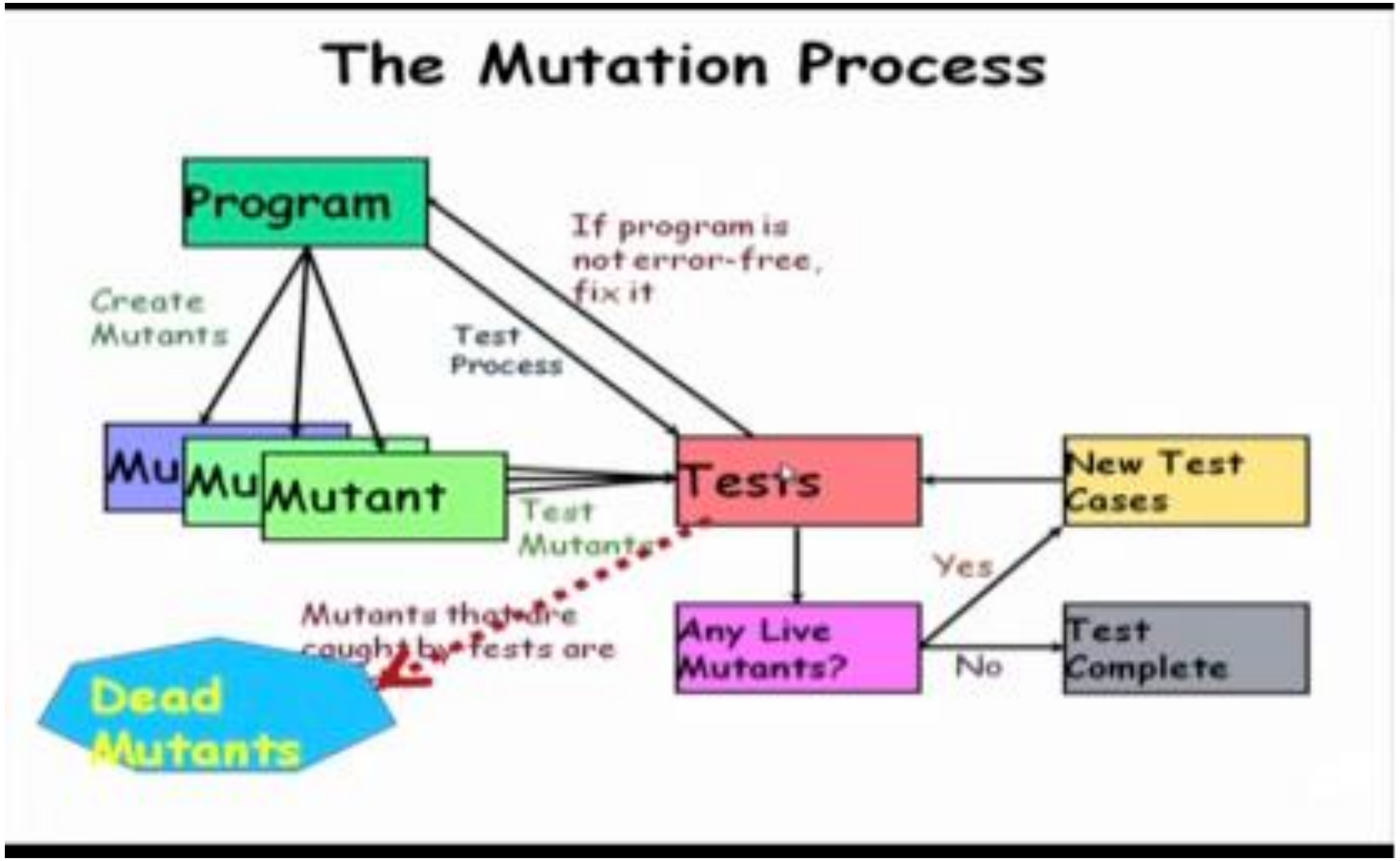
- A mutated program:
  - Tested against the full test suite of the program.
- If there exists at least one test case in the test suite for which:
  - A mutant gives an incorrect result,
  - Then the mutant is said to be dead.

Mutation Testing

- If the test cases pass for a mutated program, we say that the mutant is alive

- If a mutant remains alive:
  - Even after all test cases have been exhausted,
  - **The test suite is enhanced to kill the mutant.**
- The process of generation and killing of mutants:
  - **Can be automated by predefining a set of primitive changes that can be applied to the program.**

Mutation Testing



- ➢ Mutation testing just helps us to strengthen the test cases and ensure higher reliability of the programs than
  - ❑ the coverage based test techniques

Mutation Testing – Example

- The method invert returns a fraction where the numerator and the denominator are inverted.
- Besides, there are three conditions to check some corner cases
- For example when the numerator is zero.
  - In the case, the fraction cannot be inverted.

```java
public class Fraction {
    int numerator;
    int denominator;
    …
    public Fraction invert() {
        if (numerator == 0) {
            throw new ArithmeticException("...");
        }
        if (numerator==Integer.MIN_VALUE) {
            throw new ArithmeticException("...");
        }
        if (numerator<0) {
            return new Fraction(-denominator, -numerator);
        }
        return new Fraction(denominator, numerator);
    }
}
```

Mutation Testing – Example

- Now, let us consider a coverage adequate test suite.

- This suite reaches 100% of branch and statement coverage, and all test cases have assertions and don't fail.

```java
@Test
public void testInvert(){
    Fraction f = new Fraction(1, 2);
    Fraction result = f.invert();
    assertEquals(2, result.getFloat(), 0.00001);
}

@Test
public void testInvert_negative(){
    Fraction f = new Fraction(-1, 2);
    Fraction result = f.invert();
    assertEquals(-2, result.getFloat(), 0.00001);
}

@Test
public void testInvert_zero(){
    Fraction f = new Fraction(0, 2);
    assertThrows(ArithmeticException.class, () -> {f.invert();});
}

@Test
public void testInvert_minValue(){
    int n = Integer.MIN_VALUE;
    Fraction f = new Fraction(n, 2);
    assertThrows(ArithmeticException.class, () -> {f.invert();});
}
```

Mutation Testing – Example

- Now, the developer made a mistake by remove the sign (-) for the numerator

- This small mutant is highlighted in red color, and it is tiny syntactic change to the original code.

- Does our test suite detect the change?

Mutant

```
public class Fraction {
   int numerator;
   int denominator;
   …
   public Fraction invert() {
      if (numerator == 0) {
         throw new ArithmeticException("...");
      }
      if (numerator==Integer.MIN_VALUE) {
         throw new ArithmeticException("...");
      }
      if (numerator<0) {
         return new Fraction(-denominator, numerator);
      }
      return new Fraction(denominator, numerator);
   }
}
```

Mutation Testing – Example

- If we rerun our tests, we discover that the second test case fails.
- Therefore, we would say that the mutant is killed.

```java
@Test
public void testInvert(){
    Fraction f = new Fraction(1, 2);
    Fraction result = f.invert();
    assertEquals(2, result.getFloat(), 0.00001);
}

@Test
public void testInvert_negative(){
    Fraction f = new Fraction(-1, 2);
    Fraction result = f.invert();
    assertEquals(-2, result.getFloat(), 0.00001);
}

@Test
public void testInvert_zero(){
    Fraction f = new Fraction(0, 2);
    assertThrows(ArithmeticException.class, () -> {f.invert();});
}

@Test
public void testInvert_minValue(){
    int n = Integer.MIN_VALUE;
    Fraction f = new Fraction(n, 2);
    assertThrows(ArithmeticException.class, () -> {f.invert();});
}
```

Mutation Testing – Example

- The constant in the first if is changed to 1 from 0, as highlighted in red color.

- Does our test suite detect the mutant?

Mutant 2

```java
public class Fraction {
    int numerator;
    int denominator;
    ...
    public Fraction invert() {
        if (numerator == 1) {
            throw new ArithmeticException("...");
        }
        if (numerator==Integer.MIN_VALUE) {
            throw new ArithmeticException("...");
        }
        if (numerator<0) {
            return new Fraction(-denominator, -numerator);
        }
        return new Fraction(denominator, numerator);
    }
}
```

Mutation Testing – Example

- As you can notice, the first test case fails when executed against the mutant
- Once again, the test suite seems to be adequate.

```java
@Test
public void testInvert(){
    Fraction f = new Fraction(1, 2);
    Fraction result = f.invert();
    assertEquals(2, result.getFloat(), 0.00001);
}

@Test
public void testInvert_negative(){
    Fraction f = new Fraction(-1, 2);
    Fraction result = f.invert();
    assertEquals(-2, result.getFloat(), 0.00001);
}

@Test
public void testInvert_zero(){
    Fraction f = new Fraction(0, 2);
    assertThrows(ArithmeticException.class, () -> {f.invert();});
}

@Test
public void testInvert_minValue(){
    int n = Integer.MIN_VALUE;
    Fraction f = new Fraction(n, 2);
    assertThrows(ArithmeticException.class, () -> {f.invert();});
}
```

Generating Mutants Automatically

- Up to now, we have seen how to create mutant by manually mutating the production code.
- Clearly, this methodology is not practical nor efficient.
- For this reason, it is a common practice to rely on tools that generate mutants automatically.
- The mutants are created using the mutation operators.

- **Mutation operators**: rules to apply syntactic changes to the code under tests
- **Real fault based operators**: operators that apply changes very similar to defects seen in the past for the same code
- **Language-specific operators**: mutations for the inheritance in Java, mutations for pointers in C, etc.

Mutation Operators

- Nowadays, most of the existing mutation tools provide basic mutation operators
    - That alter parts of the code that common among most programming languages
    - Such as arithmetic or relational statement in the code.
- These mutation operators can be of different types

Basic Operators:

- Arithmetic Operator Replacement (AOR)
- Relational Operator Replacement (ROR)
- Conditional Operator Replacement (COR)
- Assignment Operator Replacement (AOR)
- Scalar Variable Replacement (SVR)

Mutation Operators

Arithmetic Operator Replacement (AOR)

This operator replaces an arithmetic operation (+, -,*,/,%) in the production code with an alternative operator

- We can create 4 mutants for each / and % operator
- Total 12 mutants can be created using AOR

```
public class Division {
   public static int[] getValues(int a, int b){
      if (b == 0){
         return null;
      }                                       +, - , * , %
      int quotient = a / b;
      int remainder = a % b;                  +, - , * , /

      return new int[] {quotient, remainder};
   }
}
```

- The arithmetic operator replacement also includes the case
- Where arithmetic operation is dropped and replaced with one of the two operands.

```
public class Division {
   public static int[] getValues(int a, int b){
      if (b == 0){
         return null;
      }                              quotient = a
      int quotient = a / b;
      int remainder = a % b;         remainder= b

      return new int[] {quotient, remainder};
   }
}
```

Mutation Operators

Relational Operator Replacement (ROR)

This operator replaces relational operators (<,>,<=,>=,==,!=) in the production code with an alternative operator

```
public class Division {
    public static int[] getValues(int a, int b){
        if (b == 0){ ——> <=, >=, !=, >, <
            return null;
        }
        int quotient = a / b;
        int remainder = a % b;

        return new int[] {quotient, remainder};
    }
}
```

- In this example, there is only one relational operator (==), that can be replaced with five different operations.
- Therefore, using ROR mutation operators five mutants will be produced for this small method.

Mutation Operators

Conditional Operator Replacement (COR)

This operator replaces conditional operators (&&, ||, &, |, !, ^) in the production code with an alternative operator

```
public class Division {
    public static int[] getValues(int a, int b){
        if (b == 0 && b == Integer.MIN_VALUE){
            return null;
        }
        int quotient = a / b;
        int remainder = a % b;

        return new int[] {quotient, remainder};
    }
}
```

&, ||, |

Mutation Operators

Assignment Operator Replacement (AOR)

This operator replaces assignment operators (=,+=,-=,*=,...) in the production code with an alternative operator

```
public class Division {
    public static int[] getValues(int a, int b){
        if (b == 0 && b == Integer.MIN_VALUE){
            return null;
        }
        int quotient = a / b;
        int remainder = a % b;

        return new int[] {quotient, remainder};
    }
}
```

+=, -=, /=,...

+=, -=, /=,...

Mutation Operators

Scalar Variable Replacement (SVR)

Each variable reference is replaced with another variable reference of the same type and that is already declared in the code

```
public class Division {
   public static int[] getValues(int a, int b){
      if (b == 0 && b == Integer.MIN_VALUE){
         return null;
      }
      int quotient = a / b;
      int remainder = a % b;

      return new int[] {quotient, remainder};
   }
}
```

- To make the code meaningful, the new variable reference must be declared before the location of the mutation
- In this example, we have 8 locations that can be mutated
- In the if condition, we can replace the variable b with the variable a
- In the last line, the variable 'quotient' can be replaced by all variables already defined in the code, like a, b, and Remainder.

Mutation Operators

- In Java, there are many other language--specific operators

Object-Oriented Operators

- Access Modifier Change
- Hiding Variable Deletion
- Hiding Variable Insertion
- Overriding Method Deletion
- Parent Constructor Deletion
- Declaration Type Change
- …

Different Ways of Mutation Testing

- So, there are different ways to change the program
  - Value Mutations – Values are changed
  - Decision Mutations – Logical or arithmetic operators are changed in decisions
  - Statement Mutations – Statements are deleted or replaced

- Example of Value Mutations:

Original Code:

```
int mod = 1000000007;
int a = 12345678;
int b = 98765432;
int c = (a + b) % mod;
```

Changed Code:

```
int mod = 1007;
int a = 12345678;
int b = 98765432;
int c = (a + b) % mod;
```

Different Ways of Mutation Testing

- Example of Decisions Mutations:

Original Code:
```
if(a < b)
 c = 10;
else
 c = 20;
```

Changed Code:
```
if(a > b)
 c = 10;
else
 c = 20;
```

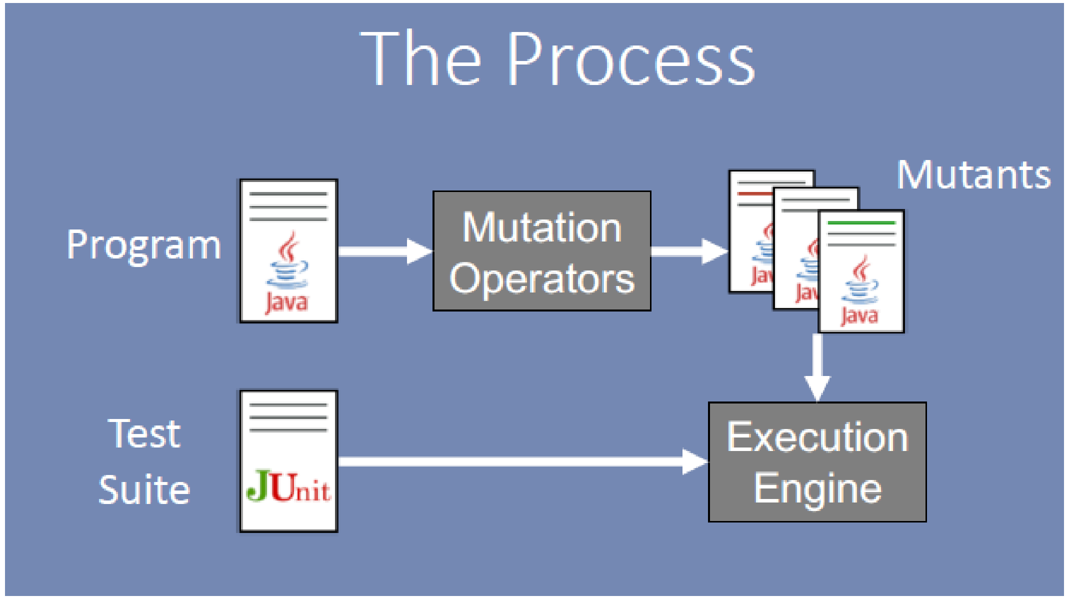
- Example of Statement Mutations:

Original Code:
```
if(a < b)
 c = 10;
else
 c = 20;
```

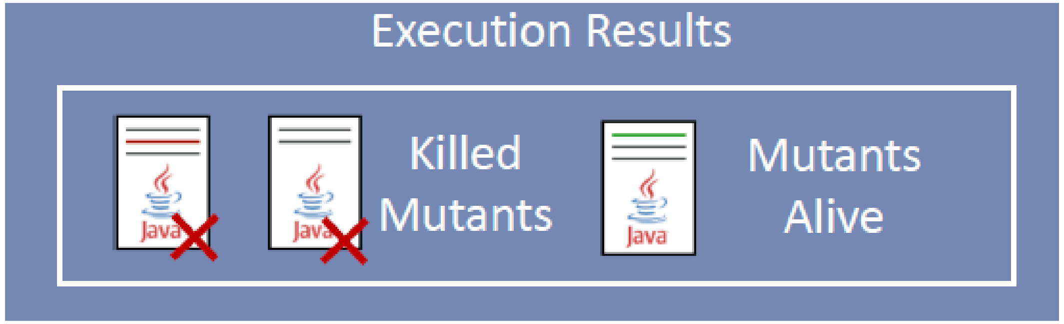
Changed Code:
```
if(a < b)
 d = 10;
else
 d = 20;
```

Mutation Testing in Practice

- There are three main steps in mutation analysis.
- The first step is generating mutants using the mutation operators.
- Then, the test code is executed against each mutant using an execution engine.
- In the final stage, we have to analyze the test results.



- Upon test execution, the mutants are classified into two categories:
- A mutant is killed if the test suite fails when executed against the mutant while it passes on the original program;
- Instead, a mutant is alive when the test suite passes on both the mutant and the original program.



Mutation Score

- Based on the ratio of killed and alive mutants, the quality of the test suite can be measured using the mutation score.

- If some mutants survive, it means that likely we need to improve the test suite by adding new tests or changing the existing ones.

- However, it is indeed possible that some mutants cannot be killed at all, whatever test case we use.

- This is the case of the equivalent mutants.

$$\text{Mutation score} = \frac{\#\text{ Killed Mutants}}{\#\text{ Mutants}}$$

Equivalent Mutants

Equivalent Mutants

- **Equivalent mutant:** a mutant M that is functionally equivalent to the original program P

Original Program

```
public void method(int a){
   int index = 10;
   while (...){
      ...
      index--;
      if (index == 0)
         break;
   }
}
```

Equivalent Mutant

```
public void method(int a){
   int index = 10;
   while (...){
      ...
      index--;
      if (index <= 0)
         break;
   }
}
```

- ➢ These two programs are functionally equivalent: whatever input value we consider, the while-loop will always end when the index is equal to 0.
- ➢ This is an elementary example of equivalent mutant.

Equivalent Mutants

- While trivial cases of equivalent mutant can be automatically discovered, it is impossible to detect all equivalent mutants in an automated fashion.

- Therefore, when you use mutation testing, you should always check the mutants that are alive because some of them can not be killed

- In general, mutants could provide a good indication of fall detection ability of a test suite when the mutation operators are carefully selected, and equivalent mutants are removed.

The cost of Mutation Testing

Let’s assume we have:

- A code base with 300 Java classes
- 10 test cases for each class
- On average, each test case requires 0.2 seconds for its execution
- The total test suite execution costs 300 * 10 * 0.2 = 600s (10 minutes)

Let’s assume we have, on average, 20 mutants per each class. The total cost of mutation analysis is:

300 * 10 * 0.2 * 20 = 12000s = 3h 20 minutes

Advantage and Disadvantage of Mutation Testing

Adv:

- Can be automated
- Helps effectively strengthen black box and coverage-based test suite

Disadvantages of Mutation Testing

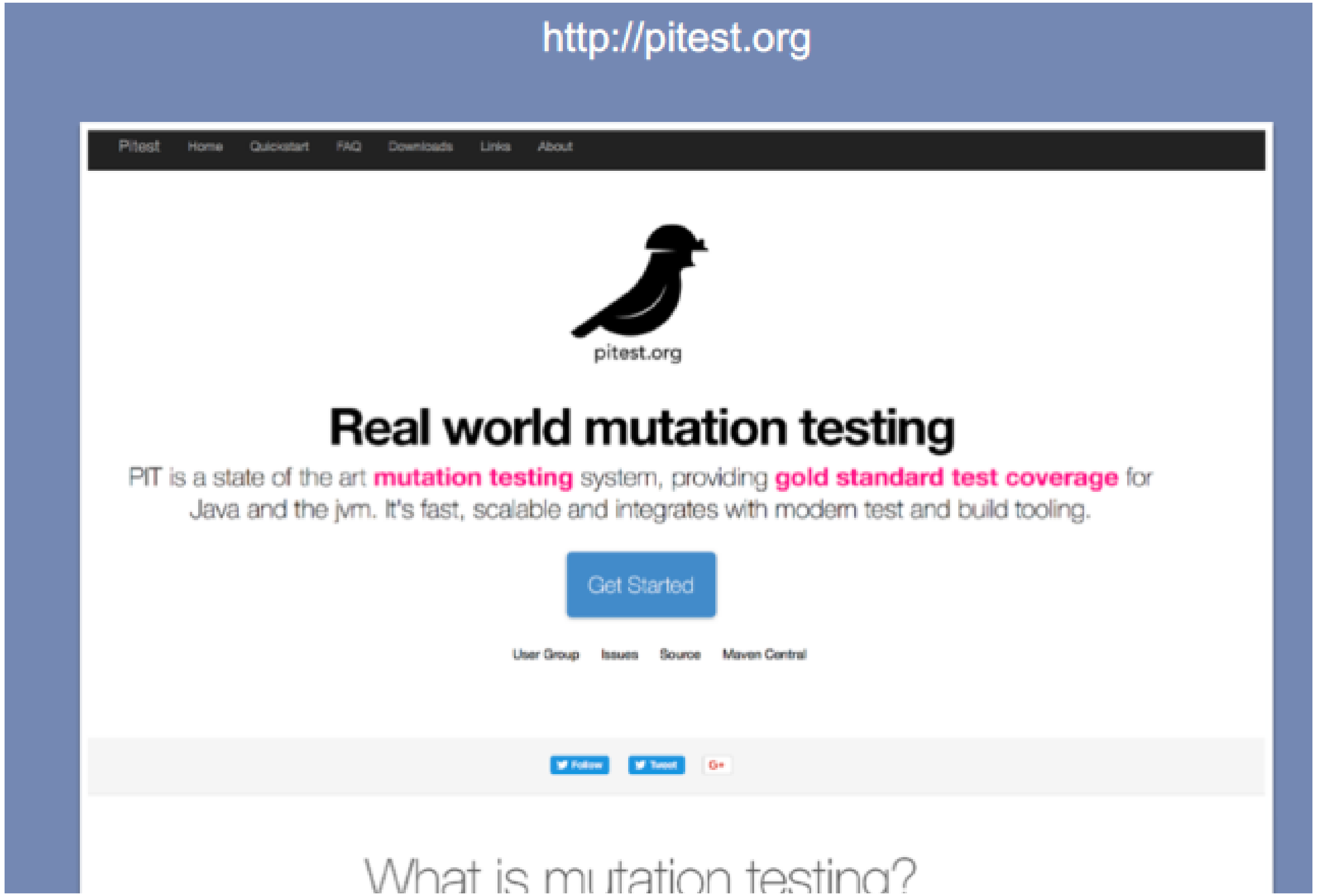
- Equivalent mutants
- Computationally very expensive.
  - A large number of possible mutants can be generated.
- Certain types of faults are very difficult to inject.
  - Only simple syntactic faults introduced.

Mutation Testing Tools

- For Java:
  - PIT
  - MuJava
  - Bacterio
  - Javalanche
  - Major
  - Descardes
- For PHP:
  - Humbug
  - Infection PHP
- For JavaScript:
  - Stryker
- For C#:
  - Nester
  - VisualMutator
- For C/C++
  - Dextool Mutate
  - Mutate.py

PIT Testing Tool

- One of the most mature tools for Java is PIT Testing, which is a publicly available tool for Java code.



PIT Testing Tool

- PIT testing can be launched via command line

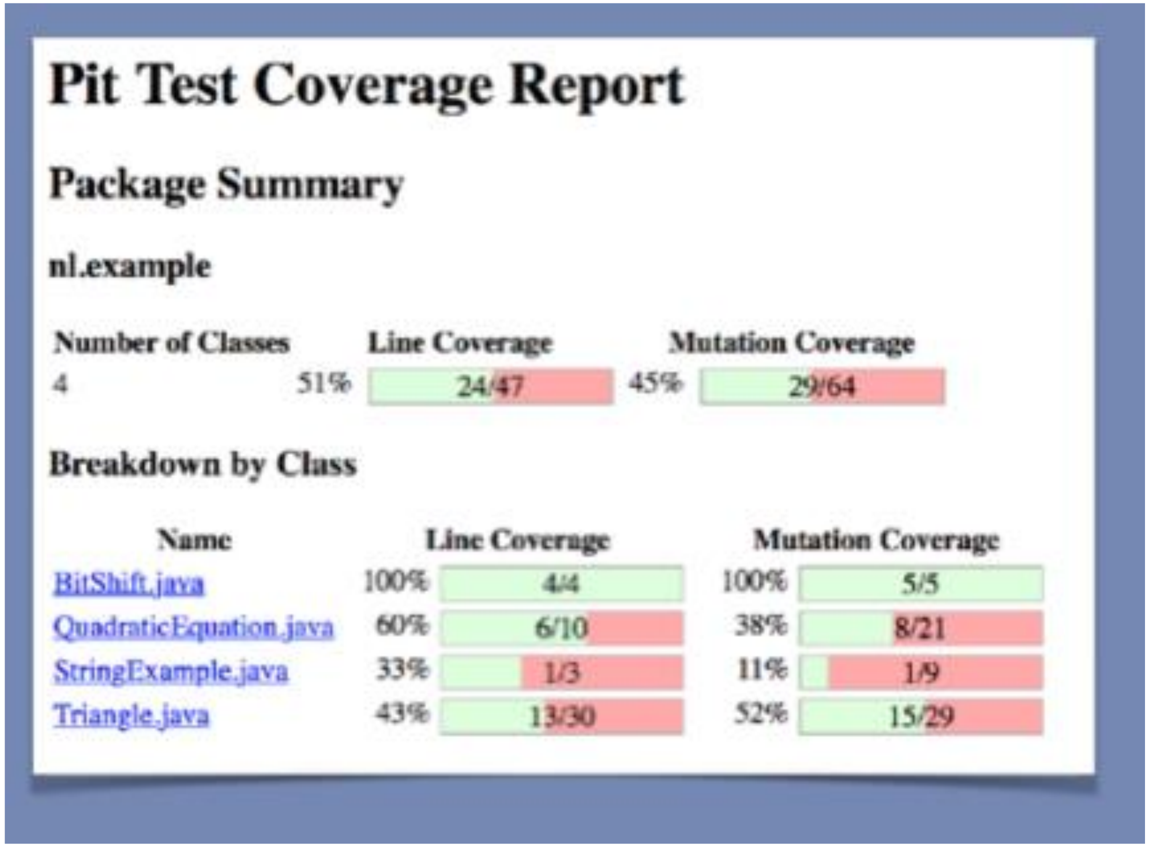
Command Line

```
java -cp <jar and dependencies> \
    org.pitest.mutationtest.commandline.MutationCoverageReport\
    --reportDir <outputdir> \
    --targetClasses com.your.package.tobemutated* \
    --targetTests com.your.package.*
    --sourceDirs <pathtosource>
```

- It is also integrated into most popular IDEs (like Eclipse and IntelliJ), in Maven and Gradle.

https://pitest.org/quickstart/mutators/

PIT Testing Tool



- PIT Testing provides the test execution reports with the mutation score for each class.
- It also allows you to inspect the source code and the mutants.

Installation of PIT Testing Tool in Eclipse

- Need to install **Pitclipse** plug-in in Eclipse.
- Go to Help -> Eclipse Marketplace -> Write Pitclipse in the search bar and install the plugin
- To run this plugin
    - Right click on the class of the test cases for which you want to do perform Mutation Testing
    - Select PIT Mutation Test and run
    - Two windows will be appeared – PIT Summary, and PIT Mutations
    - PIT Summary gives the report of Mutation Test