

CSE 403

Software Engineering

Spring 2023

#17: Mutation-based Testing

Recap: structural code coverage

Classes in this File	Line Coverage	Branch Coverage	Complexity
Avg	100% 10/10	100% 8/8	6

```
1 package avg;
2
3 4 public class Avg {
4
5     /*
6     * Compute the average of the absolute values of an array of doubles
7     */
8     public double avgAbs(double ... numbers) {
9         // We expect the array to be non-null and non-empty
10 4         if (numbers == null || numbers.length == 0) {
11 2             throw new IllegalArgumentException("Array numbers must not be null or empty!");
12         }
13
14 2         double sum = 0;
15 8         for (int i=0; i<numbers.length; ++i) {
16 6             double d = numbers[i];
17 6             if (d < 0) {
18 2                 sum -= d;
19             } else {
20 4                 sum += d;
21             }
22         }
23 2         return sum/numbers.length;
24     }
25 }
```

- Code coverage is easy to compute.
- Code coverage has an intuitive interpretation.
- Code coverage in industry: [Code coverage at Google](#)
- Code coverage itself is not sufficient!

Recap: structural code coverage

Classes in this File	Line Coverage	Branch Coverage	Complexity
Avg	100% 10/10	100% 8/8	6

```
1 package avg;
2
3 4 public class Avg {
4
5     /*
6     * Compute the average of the absolute values of an array of doubles
7     */
8     public double avgAbs(double ... numbers) {
9         // We expect the array to be non-null and non-empty
10 4     if (numbers == null || numbers.length == 0) {
11 2         throw new IllegalArgumentException("Array numbers must not be null or empty!");
12     }
13
14 2     double sum = 0;
15 8     for (int i=0; i<numbers.length; ++i) {
16 6         double d = numbers[i];
17 6         if (d < 0) {
18 2             sum -= d;
19         } else {
20 4             sum += d;
21         }
22     }
23 2     return sum/numbers.length;
24 }
25 }
```

- Code coverage is easy to compute.
- Code coverage has an intuitive interpretation.
- Code coverage in industry: [Code coverage at Google](#)
- **Code coverage itself is not sufficient! Why?**

Mutation testing: the basics

Mutation testing: the high-level pitch

```
int RunMe(int a, int b) {  
  if (a == b || b == 1) {
```

▼ Mutants

14:25, 28 Mar

Changing this 1 line to

```
  if (a != b || b == 1) {
```

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted

[Please fix](#)

[Not useful](#)

Practical Mutation Testing at Scale: A view from Google ([Reading](#))

Mutation testing: mutant generation

```
public class GraphingSettings { private static *  
    * // add state variable here  
    *  
    private int condition;  
    *  
    * // add initialization of state variable here  
    *  
    public void doSomething() {  
        condition = 0;  
    }  
    *  
    * // add initioe initialization here  
    *  
    public void initializeInstance 21 {  
        ((GraphingSettings) getInstace(1)).condition = 2;  
    }  
    *  
    * // add destroy code defination here  
    * // making null means the call will not be done  
    *  
    public Color getColor() {  
        return condition *  
        case 1: return Color.BLUE; *  
        case 2: return Color.GREEN;  
    }  
    *  
    * // add state variable copy code here  
    *  
    public void copyState() {  
        ((GraphingSettings) clone()).condition = condition;  
    }  
    *  
    * // add transeon function code here  
    * // making the not state of the call  
    *  
    public void transformCall() {  
        if (condition == 0) *  
            condition = condition - 1;  
        } else { *  
            this.neighbor() = null; getNeighbor(); *  
            for (int i = 0; i < neighbor.length; i++) *  
                if ((GraphingSettings) neighbor(i).condition == 0) *  
                    break; *  
        } *  
    } *  
}
```

Program



**Mutation
testing**

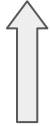
Mutation testing: mutant generation


```
public class DrumburgSettings { private static *  
 * // add state variable here  
 *  
 private int condition;  
 *  
 * // add initialization of state variable here  
 *  
 public void doSomething() {  
     condition = 0;  
 }  
 *  
 * // add initio initialization here  
 *  
 public void initializeInitio 21 {  
     ...((DrumburgSettings) getInitio(1)).condition = 2;  
 }  
 *  
 * // add drawing order definition here  
 *  
 * // adding null means the call will not be done  
 *  
 public Color getColor() {  
     return condition == 1  
         ? case 1: return Color.BLUE;  
         : case 2: return Color.RED;  
 }  
 *  
 * // add state variable copy code here  
 *  
 public void copyState() {  
     doSomething();  
     condition = condition;  
 }  
 *  
 * // add reverse function code here  
 *  
 * // deleting the last state of the call  
 *  
 public void reverseCall() {  
     if (condition == 1)  
         condition = condition - 1;  
     } else {  
         state neighbor() = null; getNeighbor();  
         for (int i = 0; i < neighbor.length; i++)  
             if ((DrumburgSettings) neighbor(i)).condition == 2)  
                 break;  
     }  
 }  
 *  
 }
```


Program

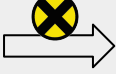


Mutation testing



Lhs < rhs  *Lhs <= rhs*

Lhs < rhs  *Lhs != rhs*

stmt  *no-op*

Mutation operators

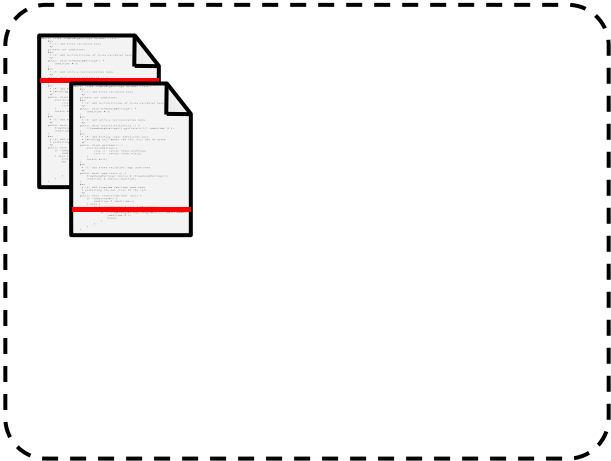
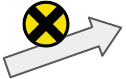
Mutation testing: mutant generation

```

public class GreengrassSettings extends State {
    // (1) add state variable here
    // (2) add constructor
    // (3) add initialization of state variable here
    public void doSomething() {
        condition # 0;
    }
    // (4) add lattice initialization here
    // (5) add initialization lattice 1;
}

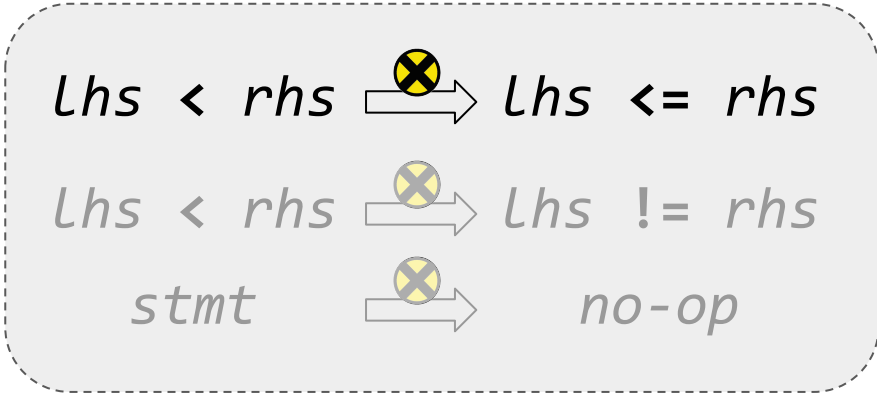
// (6) add starting state definition here
// (7) modify null check the call will not be done
// (8) add condition
public Color getColor() {
    return condition # 1;
    case 1: return Color.BLUE;
    case 2: return Color.RED;
}
// (9) add state variable copy code here
// (10) add constructor
public void copyState() {
    GreengrassSettings settings = new GreengrassSettings();
    condition # 9;
}
// (11) add traverse function code here
// (12) modify the way state of the call
public void traverseCall() {
    condition # 11;
}
}
}
}

```



Program

Mutants



Mutation operators

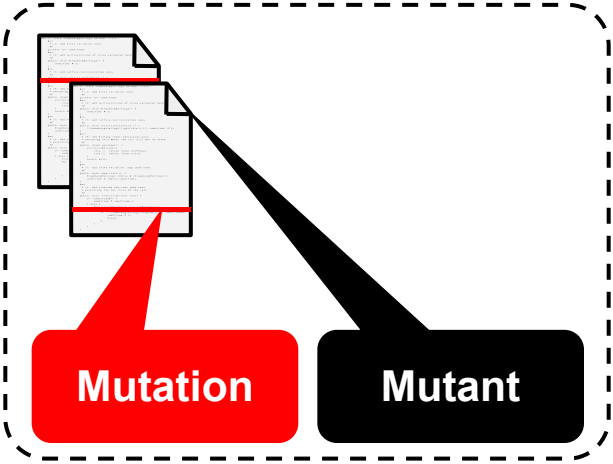
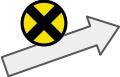
Mutation testing: mutant generation

```

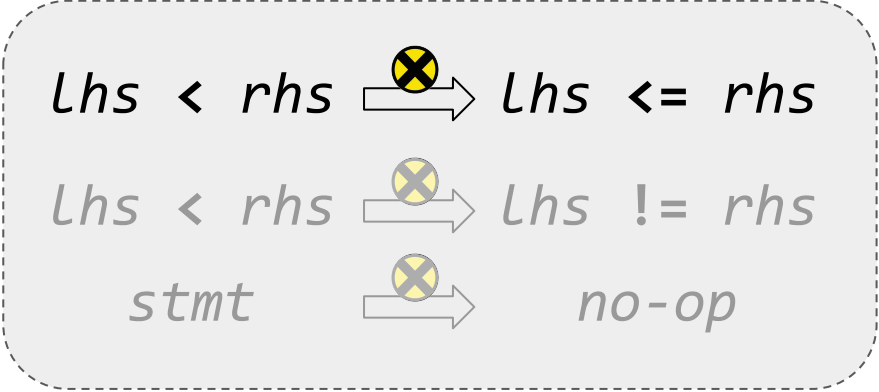
public class GreengrassSettings {
    // ...
    // (1) add state variable here
    private int condition;
    // ...
    // (2) add initialization of state variable here
    public void GreengrassSettings() {
        condition = 0;
    }
    // ...
    // (3) add lattice initialization here
    public void InitializeLattice() {
    }
    // ...
    // (4) add binary data definition here
    // ...
    // (5) add binary data definition here
    // ...
    public Color getColor() {
        return condition;
    }
    // ...
    // (6) add state variable copy code here
    // ...
    public void copyState() {
        GreengrassSettings settings = GreengrassSettings();
        condition = settings.condition;
    }
    // ...
    // (7) add reverse function code here
    // ...
    // (8) add reverse function code here
    // ...
    public void reverseCall() {
        condition = condition - 1;
    }
    // ...
    // (9) add reverse function code here
    // ...
    // (10) add reverse function code here
    // ...
}

```

Program



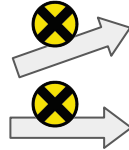
Mutants



Mutation operators

Mutation testing: mutant generation

```
public class GreengrassSettings extends Base {
    * // add state variable here
    private int condition;
    * // add initialization of state variable here
    public void doSomething() {
        condition = 0;
    }
    * // add lattice initialization here
    * // add state variable copy code here
    public void copyState() {
        condition = condition;
    }
    * // add transition function code here
    * // deleting the last state of the call
    public void transitionCall() {
        condition = condition-1;
    }
}
}
```



Program

Mutants

$Lhs < rhs \xrightarrow{\text{radiation}} Lhs \leq rhs$
 $Lhs < rhs \xrightarrow{\text{radiation}} Lhs \neq rhs$
 $stmt \xrightarrow{\text{radiation}} no-op$

Mutation operators

Mutation testing: test creation

```
public void setUp() throws Exception {
    // ...
}

public void testSomething() {
    // ...
}

public void tearDown() {
    // ...
}

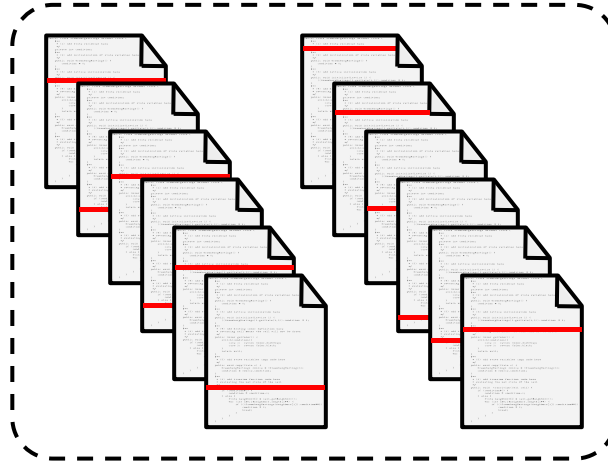
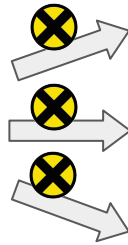
public void testAnother() {
    // ...
}

public void testYetAnother() {
    // ...
}

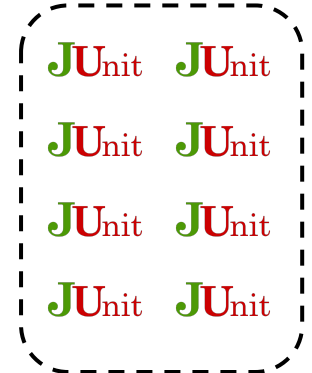
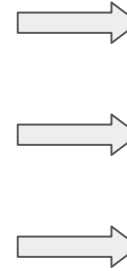
public void testOneMore() {
    // ...
}

public void testFinally() {
    // ...
}
}
```

Program



Mutants



Tests

Assumptions

- Mutants are coupled to real faults
- Mutant detection is correlated with real-fault detection

https://homes.cs.washington.edu/~rjust/publ/mutants_real_faults_fse_2014.pdf,

https://homes.cs.washington.edu/~rjust/publ/mutation_testing_practices_icse_2021.pdf

Mutation testing: a concrete example

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutant 1:

```
public int min(int a, int b) {  
    return a;  
}
```

Mutation testing: another example

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutant 2:

```
public int min(int a, int b) {  
    return b;  
}
```

Mutation testing: yet another example

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutant 3:

```
public int min(int a, int b) {  
    return a >= b ? a : b;  
}
```

Mutation testing: last example (I promise)

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutant 4:

```
public int min(int a, int b) {  
    return a <= b ? a : b;  
}
```


Mutation testing: exercise



Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutants:

M1: return a;

M2: return b;

M3: return a >= b ? a : b;

M4: return a <= b ? a : b;

For each mutant, provide a test case that detects it
(e.g., `min(<a>,) == <expected outcome>`)
the test must pass on the original program but fail on the mutant

<https://tinyurl.com/cse403-mut>

Mutation testing: exercise

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutants:

M1: return a;

M2: return b;

M3: return a >= b ? a : b;

M4: return a <= b ? a : b;

M4 cannot be detected (equivalent mutant).

a	b	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

Mutation testing: exercise

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutants:

M1: return a;

M2: return b;

M3: return a >= b ? a : b;

M4: return a <= b ? a : b;

Which mutant(s) should we show to a developer?

a	b	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

Mutation testing: summary

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutants:

M1: return a;

M2: return b;

M3: return a >= b ? a : b;

M4: return a <= b ? a : b;

Redundant

Equivalent

<i>a</i>	<i>b</i>	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

Mutation testing: challenges

- Redundant mutants
 - Inflate the mutant detection ratio
 - Hard to assess progress and remaining effort
- Equivalent mutants
 - Max mutant detection ratio \neq 100%
 - Waste resources (CPU and human time)

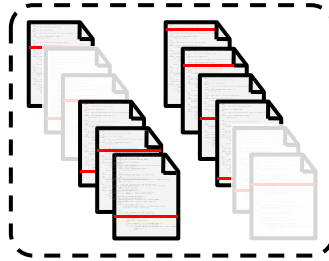
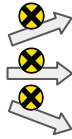
a	b	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

Mutation Testing vs. Mutation Analysis

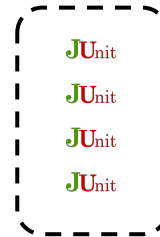
Mutation Testing



PROGRAM



MUTANTS



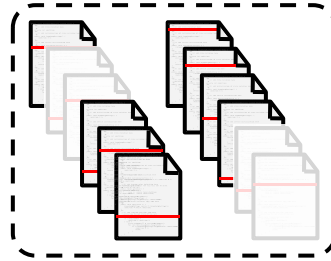
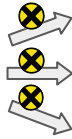
TESTS

Mutation Testing vs. Mutation Analysis

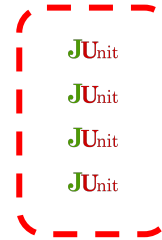
**Mutation
Testing**



PROGRAM



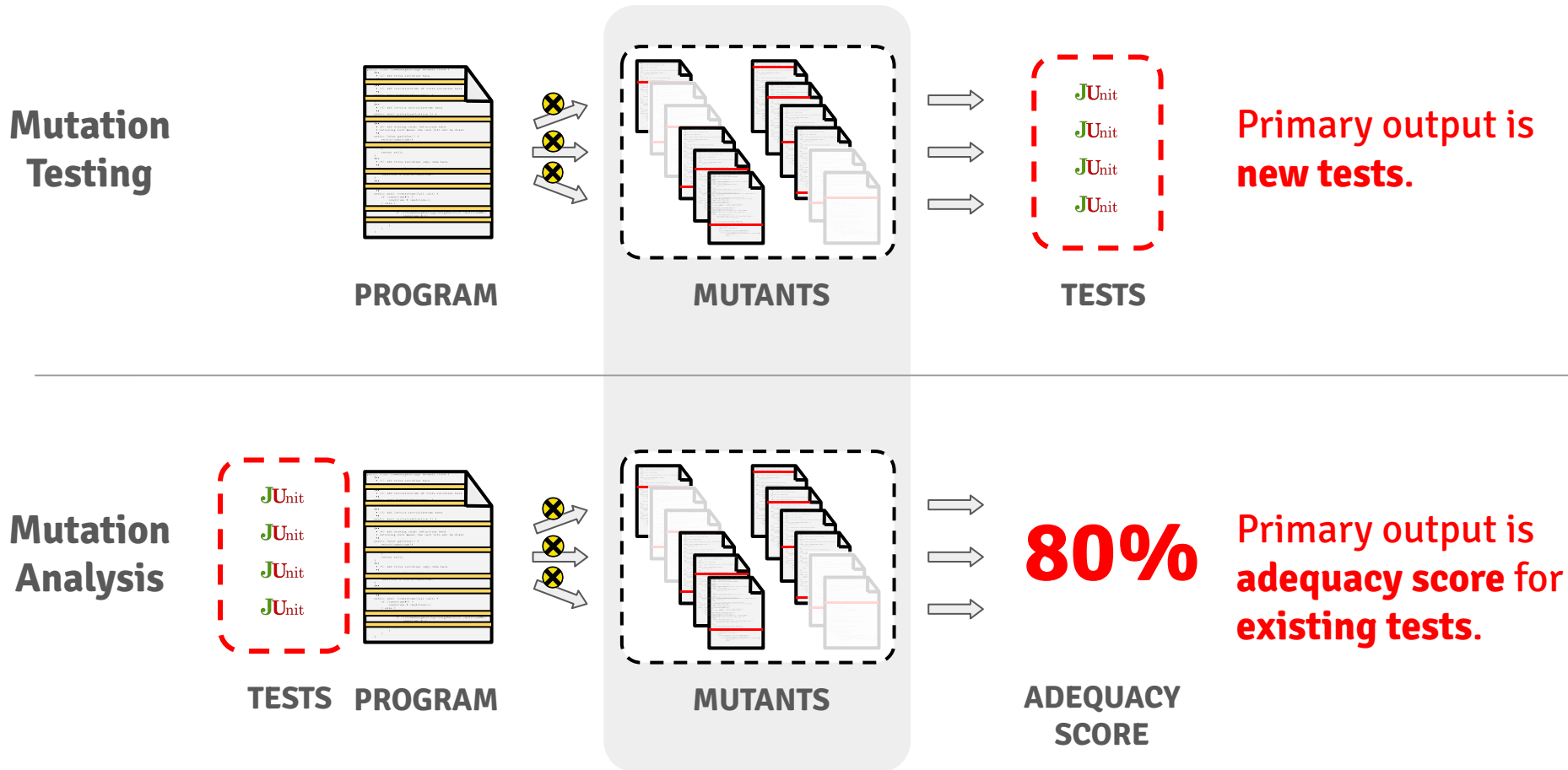
MUTANTS



TESTS

**Primary output is
new tests.**

Mutation Testing vs. Mutation Analysis



How expensive is mutation testing?
Is the mutation score meaningful?

Mutation testing: example

Test Information

Tests that covered the mutant:

- testTriangle[0: (0 1 2)->INVALID]

```
1 package triangle;
2
3 /**
4  * An implementation that classifies triangles.
5  */
6 public class Triangle {
7
8     /**
9      * This enum gives the type of the triangle.
10    */
11    public static enum Type {
12        INVALID, SCALENE, EQUILATERAL, ISOSCELES
13    };
14
15    /**
16     * This static method does the actual classification of a triangle, given the lengths
17     * of its three sides.
18    */
19    public static Type classify(int a, int b, int c) {
20        if (a <= 0 || b <= 0 || c <= 0) {
21            return Type.INVALID;
22        }
23        int trian = 0;
24        if (a == b) {
25            trian = trian + 1;
26        }
27        if (a == c) {
28            trian = trian + 2;
29        }
30        if (b == c) {
31            trian = trian + 3;
32        }
33        if (trian == 0) {
34            if (a + b <= c || a + c <= b || b + c <= a) {
35                return Type.INVALID;
36            } else {
37                return Type.SCALENE;
38            }
39        }
40        if (trian > 3) {
41            return Type.EQUILATERAL;
42        }
43        if (trian == 1 && a + b > c) {
44            return Type.ISOSCELES;
45        } else if (trian == 2 && a + c > b) {
46            return Type.ISOSCELES;
47        } else if (trian == 3 && b + c > a) {
48            return Type.ISOSCELES;
49        }
50        return Type.INVALID;
51    }
52 }
```

```
1 package triangle;
2
3 /**
4  * An implementation that classifies triangles.
5  */
6 public class Triangle {
7
8     /**
9      * This enum gives the type of the triangle.
10    */
11    public static enum Type {
12        INVALID, SCALENE, EQUILATERAL, ISOSCELES
13    };
14
15    /**
16     * This static method does the actual classification of a triangle, given the lengths
17     * of its three sides.
18    */
19    public static Type classify(int a, int b, int c) {
20        if (a < 0 || b <= 0 || c <= 0) {
21            return Type.INVALID;
22        }
23        int trian = 0;
24        if (a == b) {
25            trian = trian + 1;
26        }
27        if (a == c) {
28            trian = trian + 2;
29        }
30        if (b == c) {
31            trian = trian + 3;
32        }
33        if (trian == 0) {
34            if (a + b <= c || a + c <= b || b + c <= a) {
35                return Type.INVALID;
36            } else {
37                return Type.SCALENE;
38            }
39        }
40        if (trian > 3) {
41            return Type.EQUILATERAL;
42        }
43        if (trian == 1 && a + b > c) {
44            return Type.ISOSCELES;
45        } else if (trian == 2 && a + c > b) {
46            return Type.ISOSCELES;
47        } else if (trian == 3 && b + c > a) {
48            return Type.ISOSCELES;
49        }
50        return Type.INVALID;
51    }
52 }
```

Mutation testing: productive mutants

Detectable vs. productive mutants

Historically

- **Detectable** mutants are **good** \implies **tests**
- **Equivalent** mutants are **bad** \implies **no tests**

A more nuanced view

- **Detectable vs. equivalent** is **too simplistic**
- **Productive mutants** elicit effective tests, but
 - detectable mutants can be useless, and
 - equivalent mutants can be useful!

**The core question here concerns test-goal utility
(applies to any adequacy criterion).**

Detectable vs. productive mutants

Historically

- **Detectable** mutants are **good** \implies **tests**
- **Equivalent** mutants are **bad** \implies **no tests**

A more nuanced view

- **Detectable vs. equivalent** is **too simplistic**
- **Productive mutants** elicit effective tests, but
 - detectable mutants can be useless, and
 - equivalent mutants can be useful!

The notion of productive mutants is fuzzy!

A mutant is **productive** if it is

1. **detectable** and **elicits an effective test** or
2. **equivalent** and **advances code quality or knowledge**

Productive mutants: mutation testing at Google

```
int RunMe(int a, int b) {  
  if (a == b || b == 1) {
```

▼ Mutants

14:25, 28 Mar

Changing this 1 line to

```
  if (a != b || b == 1) {
```

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted

[Please fix](#)

[Not useful](#)

Practical Mutation Testing at Scale: A view from Google ([Reading](#))

Productive mutants: mutation testing at Google

```
int RunMe(int a, int b) {  
  if (a == b || b == 1) {
```

▼ Mutants

14:25, 28 Mar

Changing this 1 line to

```
if (a != b || b == 1) {
```

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted

[Please fix](#)

[Not useful](#)

Practical Mutation Testing at Scale: A view from Google ([Reading](#))

Detectable vs. productive mutants (1)

Original program

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum * nums[i];  
    }  
  
    return sum / len;  
}
```

Is the mutant is **detectable**?

Detectable vs. productive mutants (1)

Original program

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum * nums[i];  
    }  
  
    return sum / len;  
}
```

The mutant is **detectable**, but is it **productive**?

Detectable vs. productive mutants (1)

Original program

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum * nums[i];  
    }  
  
    return sum / len;  
}
```

The mutant is **detectable**, but is it **productive**? **Yes!**

Detectable vs. productive mutants (2)

Original program

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg + (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg * (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Is the mutant **detectable**?

Detectable vs. productive mutants (2)

Original program

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg + (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg * (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

The mutant is **not detectable**, but is it **unproductive**?

Detectable vs. productive mutants (2)

Original program

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg + (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg * (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

The mutant is not detectable, but is it unproductive? No!

Detectable vs. productive mutants (3)

Original program

```
...  
Set cache = new HashSet(a * b);  
...
```

Mutant

```
...  
Set cache = new HashSet(a + b);  
...
```

Is the mutant **detectable**?

Detectable vs. productive mutants (3)

Original program

```
...  
Set cache = new HashSet(a * b);  
...
```

Mutant

```
...  
Set cache = new HashSet(a + b);  
...
```

The mutant is **detectable**, but is it **productive**?

Detectable vs. productive mutants (3)

Original program

```
...  
Set cache = new HashSet(a * b);  
...
```

Mutant

```
...  
Set cache = new HashSet(a + b);  
...
```

The mutant is **detectable**, but is it **productive**? **No!**

Coverage-based vs. mutation-based testing

See dedicated [Slides](#) ([4 pages](#)).