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

```
package avg;

public class Avg {

    /*
     * Compute the average of the absolute values of an array of doubles
     */
    public double avgAbs(double ... numbers) {
        // We expect the array to be non-null and non-empty
        if (numbers == null || numbers.length == 0) {
            throw new IllegalArgumentException("Array numbers must not be null or empty!");
        }

        double sum = 0;
        for (int i=0; i<numbers.length; ++i) {
            double d = numbers[i];
            if (d < 0) {
                sum -= d;
            } else {
                sum += d;
            }
        }
        return sum/numbers.length;
    }
}
```

- 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

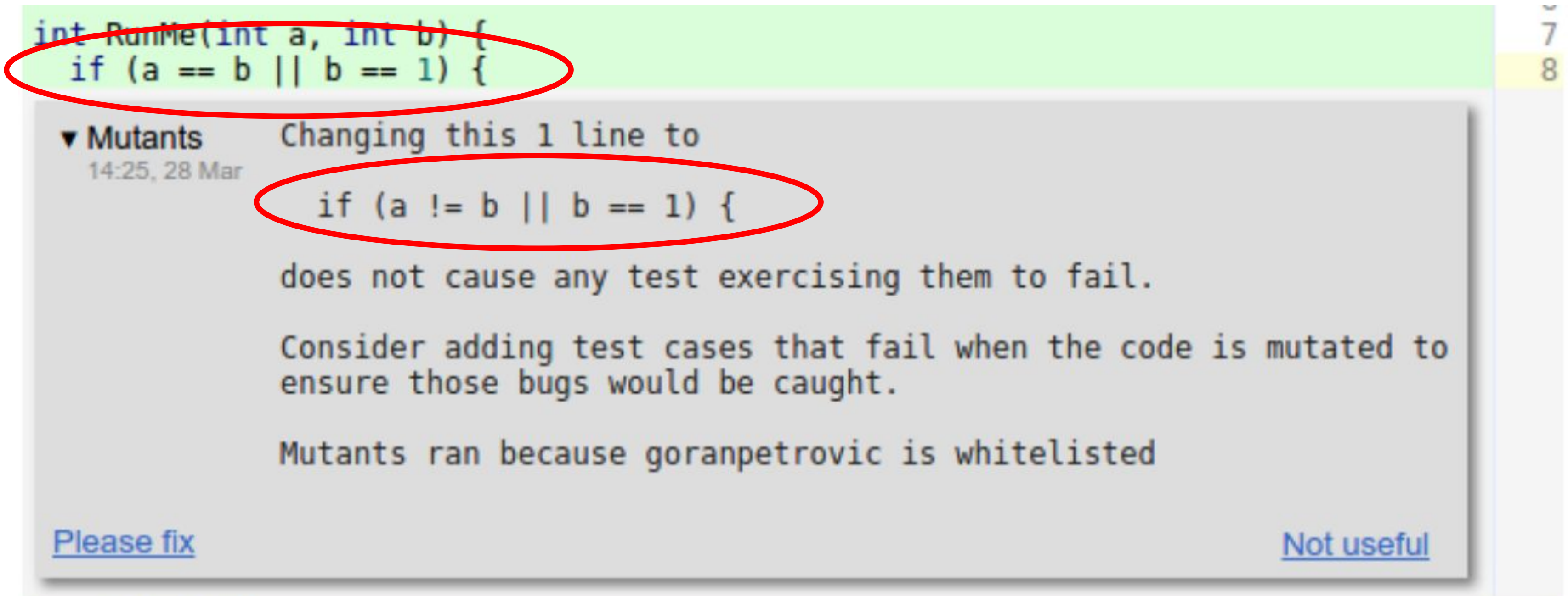
Classes in this File	Line Coverage	Branch Coverage	Complexity
Avg	100% 10/10	100% 8/8	6

```java
package avg;

public class Avg {

    /*
     * Compute the average of the absolute values of an array of doubles
     */
    public double avgAbs(double ... numbers) {
        // We expect the array to be non-null and non-empty
        if (numbers == null || numbers.length == 0) {
            throw new IllegalArgumentException("Array numbers must not be null or empty!");
        }

        double sum = 0;
        for (int i=0; i<numbers.length; ++i) {
            double d = numbers[i];
            if (d < 0) {
                sum -= d;
            } else {
                sum += d;
            }
        }
        return sum/numbers.length;
    }
}
```

- 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?**

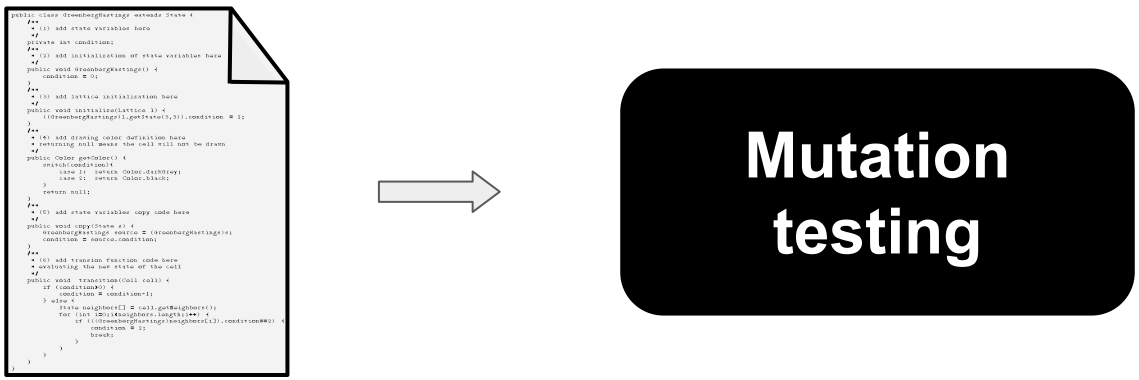
Mutation testing: the basics

Mutation testing: the high-level pitch



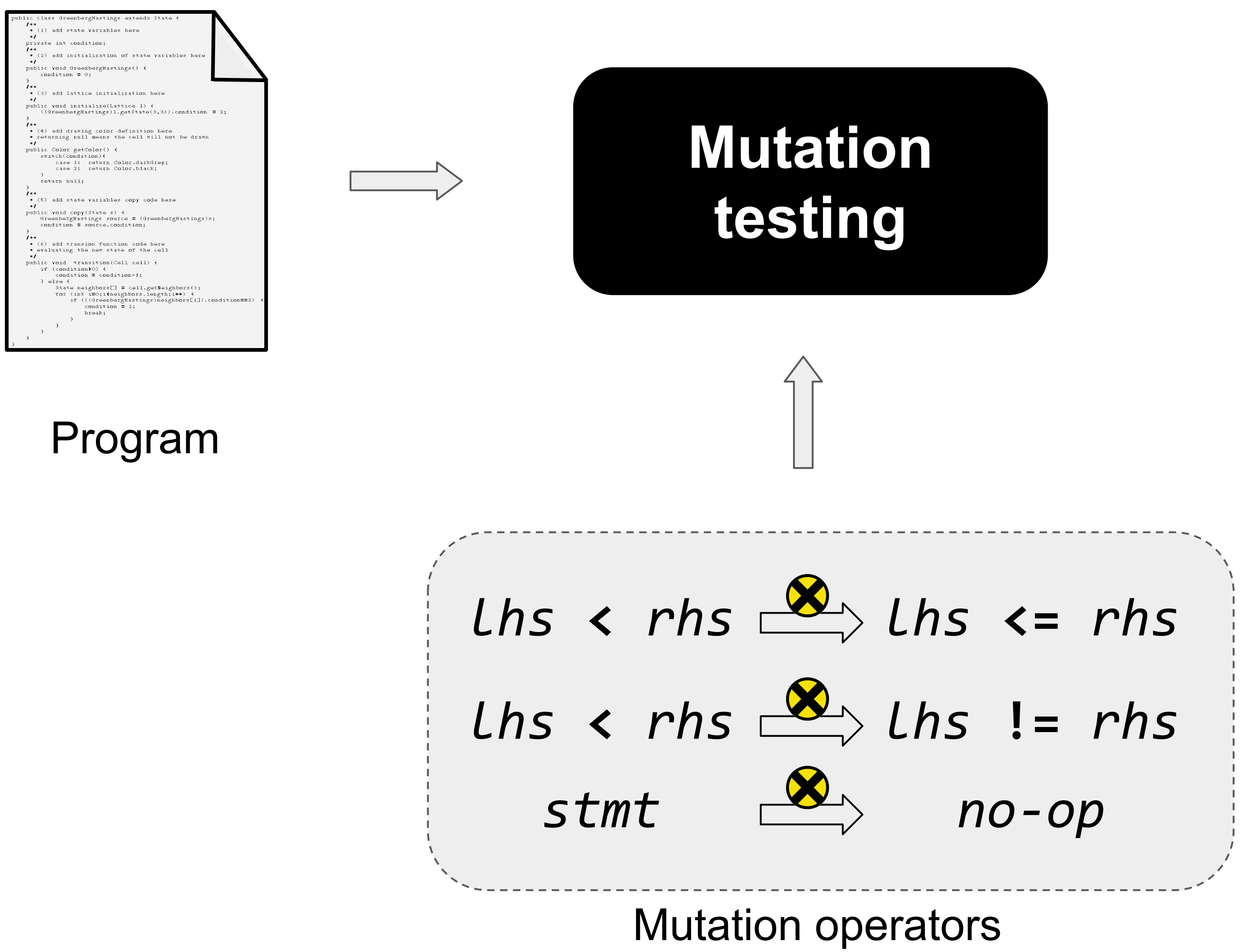
Practical Mutation Testing at Scale: A view from Google ([Reading](#))

Mutation testing: mutant generation

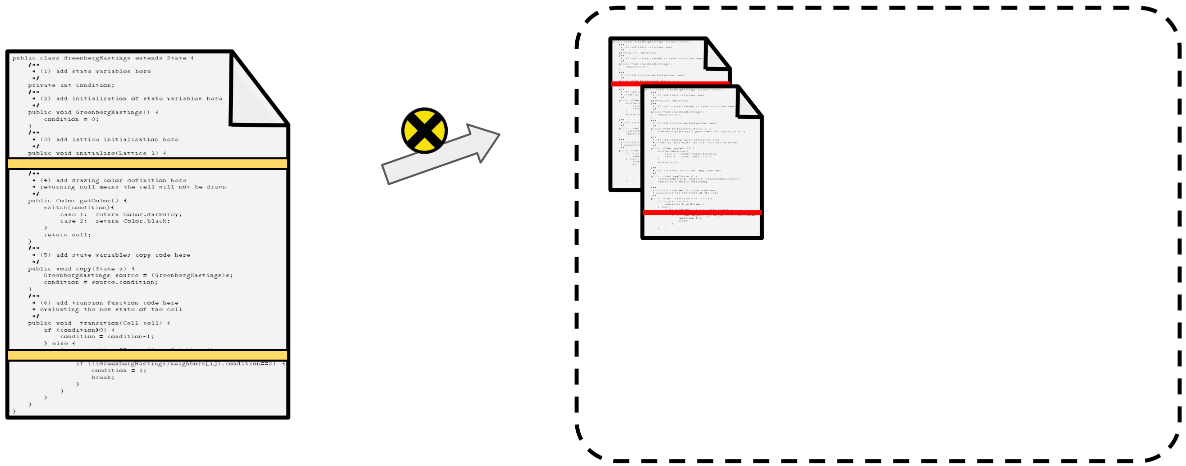


Program

Mutation testing: mutant generation



Mutation testing: mutant generation



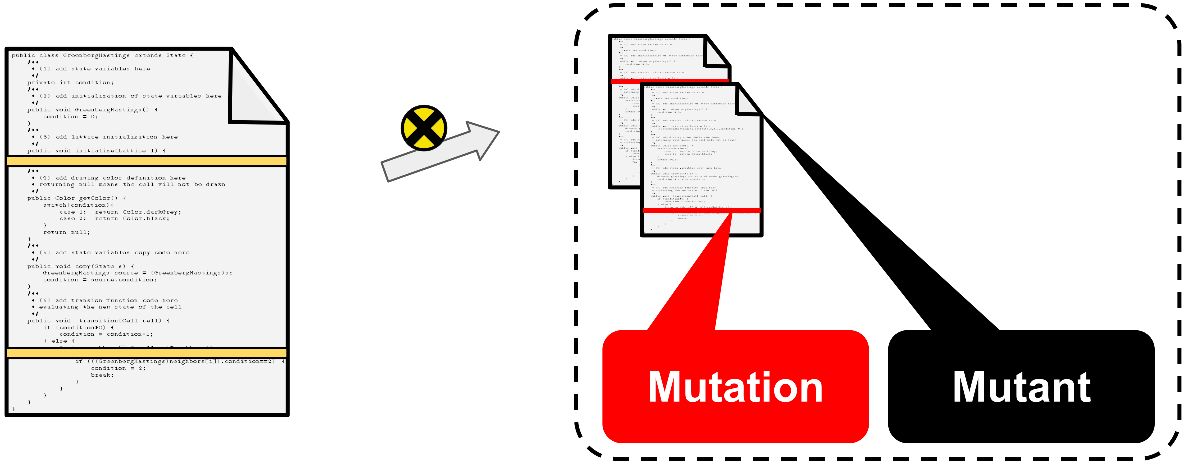
`lhs < rhs` → `lhs <= rhs`

`lhs < rhs` → `lhs != rhs`

`stmt` → `no-op`

Mutation operators

Mutation testing: mutant generation



Program

Mutants

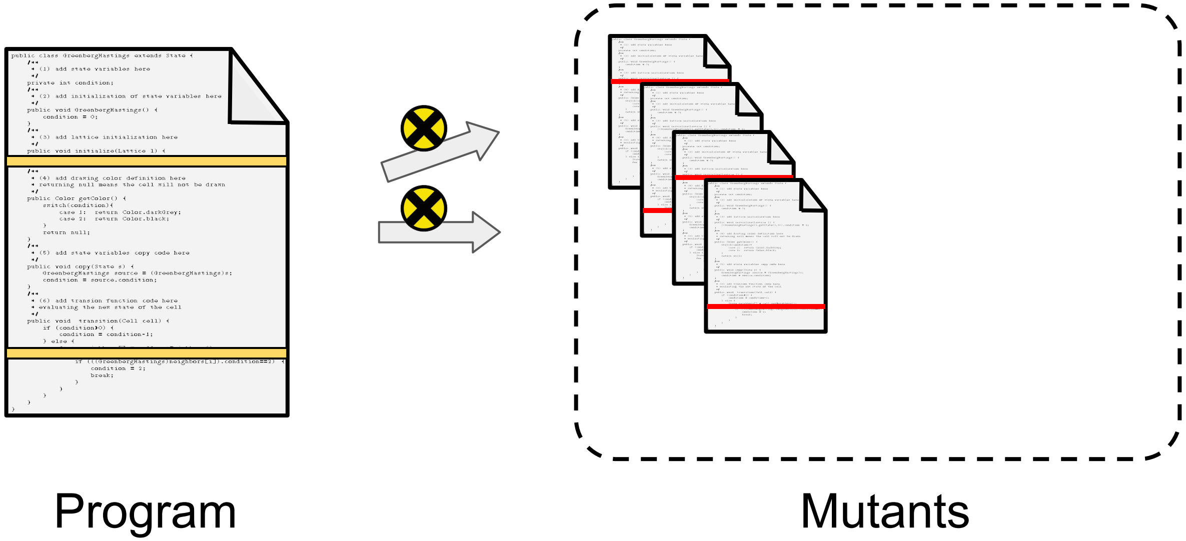
lhs < rhs → lhs <= rhs

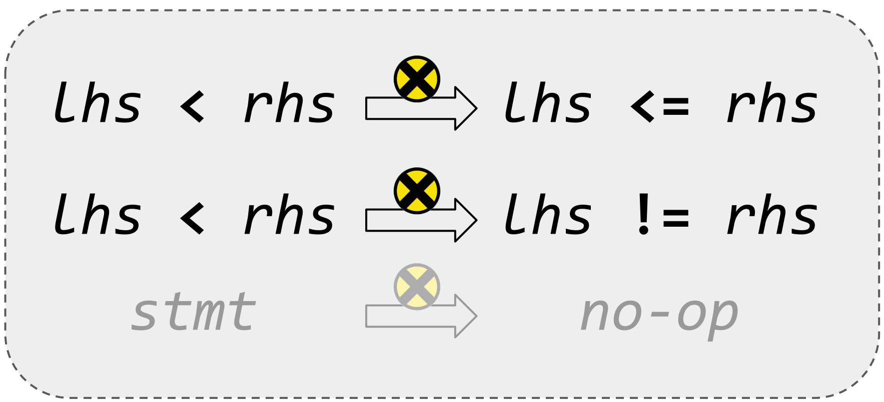
lhs < rhs → lhs != rhs

stmt → no-op

Mutation operators

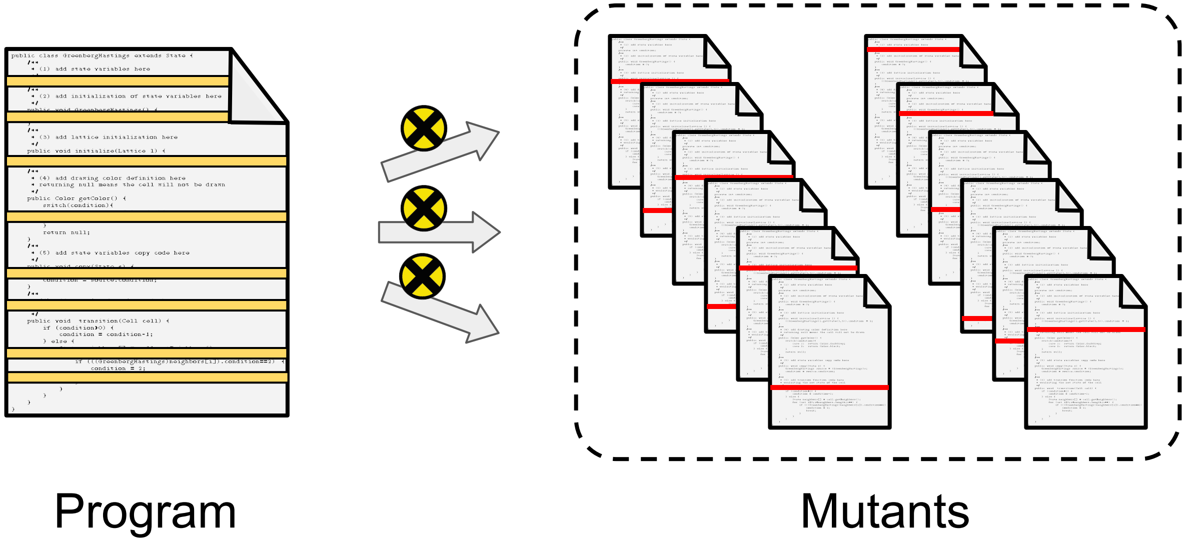
Mutation testing: mutant generation

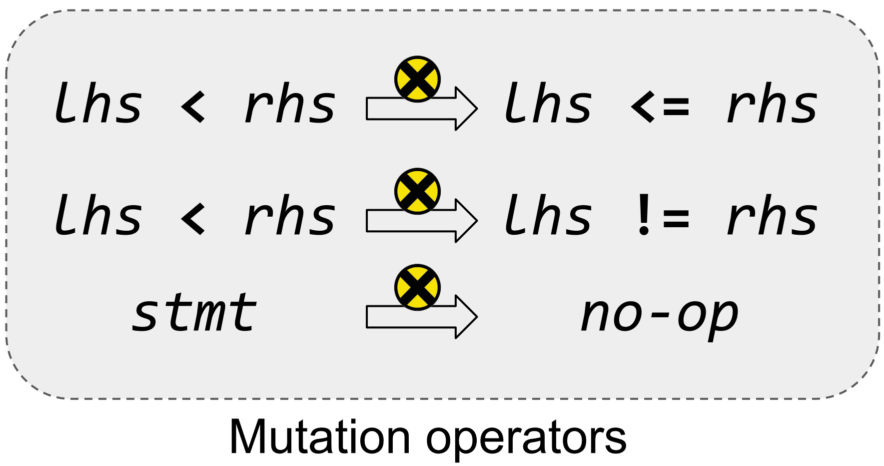




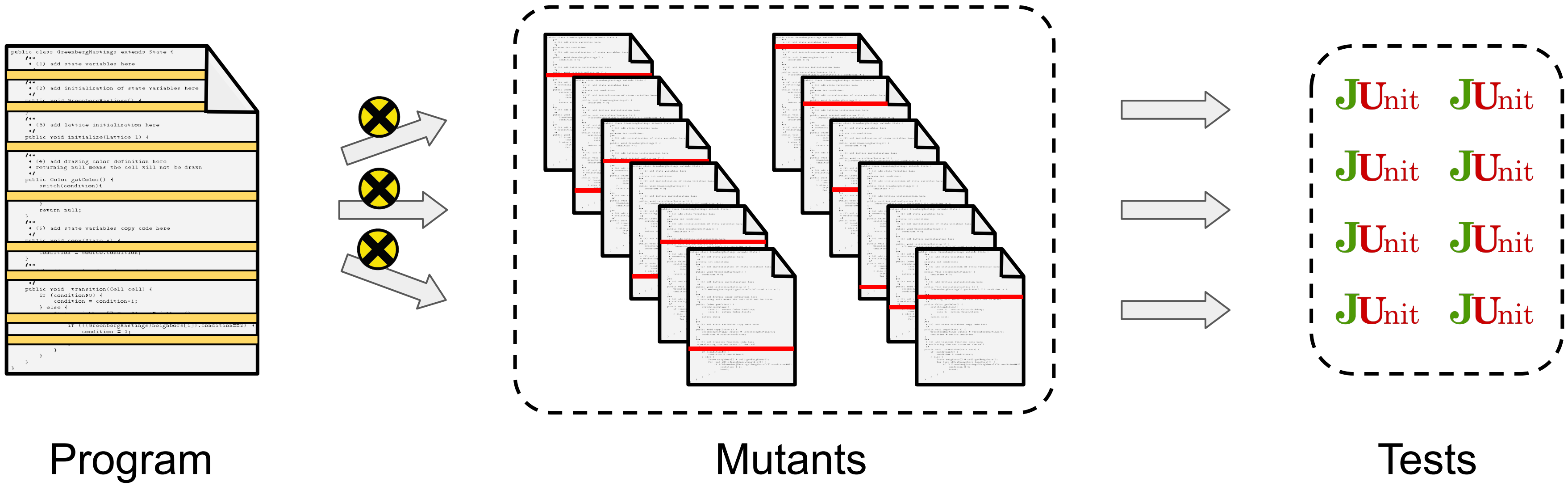
Mutation operators

Mutation testing: mutant generation





Mutation testing: test creation



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>, <b>) == <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 (M3)

Equivalent (M4)

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

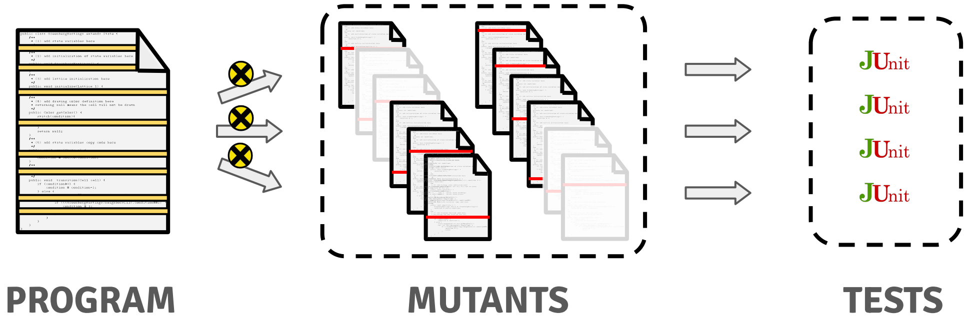
Mutation testing: challenges

- Redundant mutants
  - Inflate the mutant detection ratio
  - Hard to assess progress and remaining effort
- Equivalent mutants
  - Max mutant detection ratio != 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

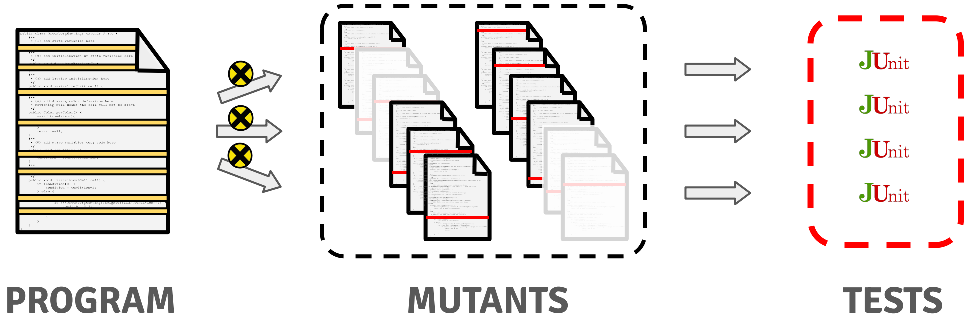
Mutation Testing vs. Mutation Analysis

Mutation Testing



Mutation Testing vs. Mutation Analysis

Mutation Testing

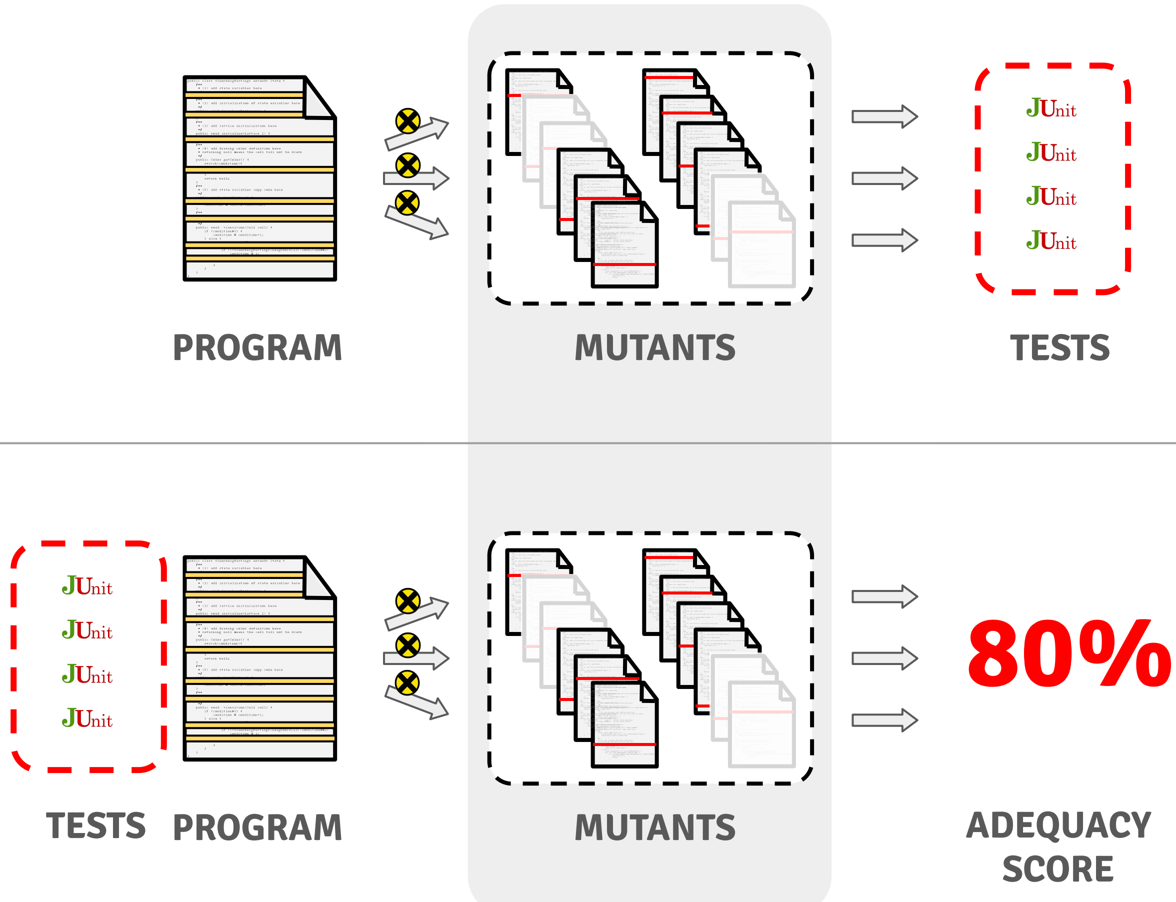


Primary output is **new tests**.

Mutation Testing vs. Mutation Analysis

Mutation Testing

Mutation Analysis



Primary output is **new tests.**

Primary output is **adequacy score** for **existing tests.**

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]

```java
package triangle;

/**
 * An implementation that classifies triangles.
 */
public class Triangle {

    /**
     * This enum gives the type of the triangle.
     */
    public static enum Type {
        INVALID, SCALENE, EQUILATERAL, ISOSCELES
    };

    /**
     * This static method does the actual classification of a triangle, given the lengths
     * of its three sides.
     */
    public static Type classify(int a, int b, int c) {
        if (a <= 0 || b <= 0 || c <= 0) {
            return Type.INVALID;
        }
        int trian = 0;
        if (a == b) {
            trian = trian + 1;
        }
        if (a == c) {
            trian = trian + 2;
        }
        if (b == c) {
            trian = trian + 3;
        }
        if (trian == 0) {
            if (a + b <= c || a + c <= b || b + c <= a) {
                return Type.INVALID;
            } else {
                return Type.SCALENE;
            }
        }
        if (trian > 3) {
            return Type.EQUILATERAL;
        }
        if (trian == 1 && a + b > c) {
            return Type.ISOSCELES;
        } else if (trian == 2 && a + c > b) {
            return Type.ISOSCELES;
        } else if (trian == 3 && b + c > a) {
            return Type.ISOSCELES;
        }
        return Type.INVALID;
    }
}
```

```java
package triangle;

/**
 * An implementation that classifies triangles.
 */
public class Triangle {

    /**
     * This enum gives the type of the triangle.
     */
    public static enum Type {
        INVALID, SCALENE, EQUILATERAL, ISOSCELES
    };

    /**
     * This static method does the actual classification of a triangle, given the lengths
     * of its three sides.
     */
    public static Type classify(int a, int b, int c) {
        if (a < 0 || b <= 0 || c <= 0) {
            return Type.INVALID;
        }
        int trian = 0;
        if (a == b) {
            trian = trian + 1;
        }
        if (a == c) {
            trian = trian + 2;
        }
        if (b == c) {
            trian = trian + 3;
        }
        if (trian == 0) {
            if (a + b <= c || a + c <= b || b + c <= a) {
                return Type.INVALID;
            } else {
                return Type.SCALENE;
            }
        }
        if (trian > 3) {
            return Type.EQUILATERAL;
        }
        if (trian == 1 && a + b > c) {
            return Type.ISOSCELES;
        } else if (trian == 2 && a + c > b) {
            return Type.ISOSCELES;
        } else if (trian == 3 && b + c > a) {
            return Type.ISOSCELES;
        }
        return Type.INVALID;
    }
}
```

Mutation testing: productive mutants

Detectable vs. productive mutants

Historically

- **Detectable** mutants are **good** ⇨ **tests**
- **Equivalent** mutants are **bad** ⇨ **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).

An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions (Reading)

Detectable vs. productive mutants

Historically

- **Detectable** mutants are **good** ⇨ **tests**
- **Equivalent** mutants are **bad** ⇨ **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!

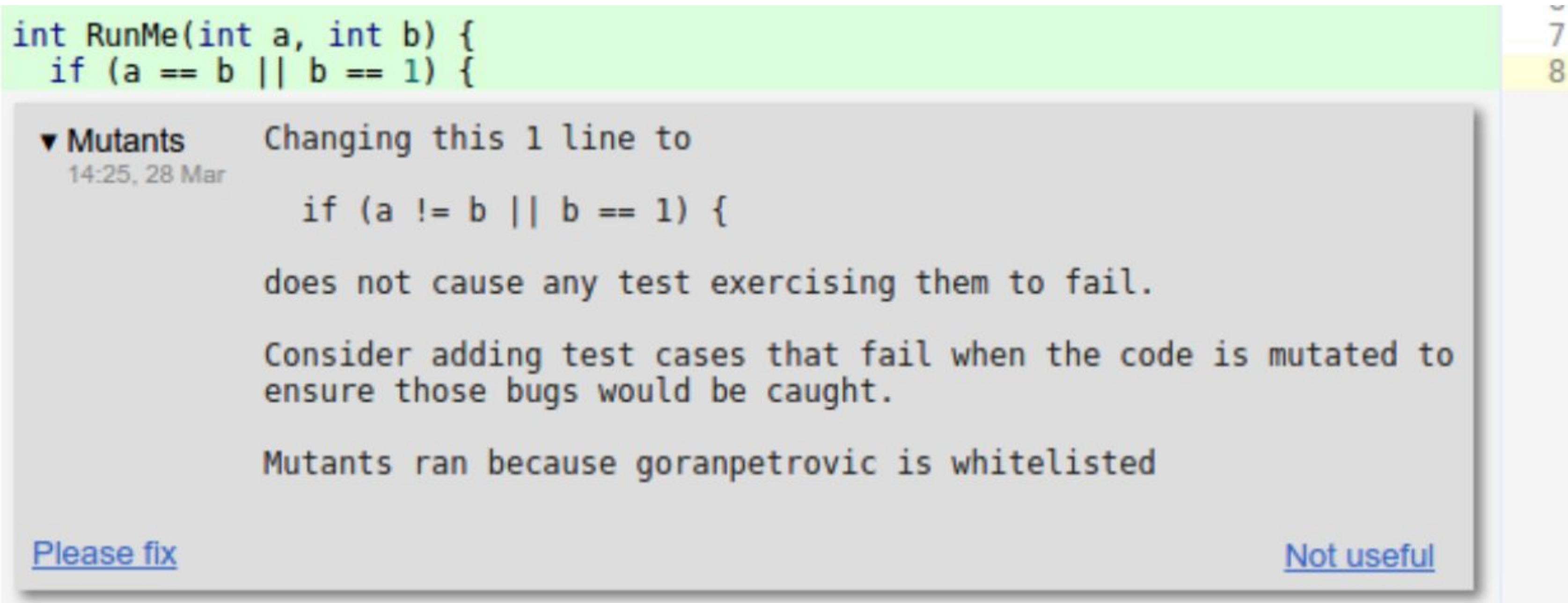
The notion of productive mutants is fuzzy!

A mutant is **productive** if it is

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

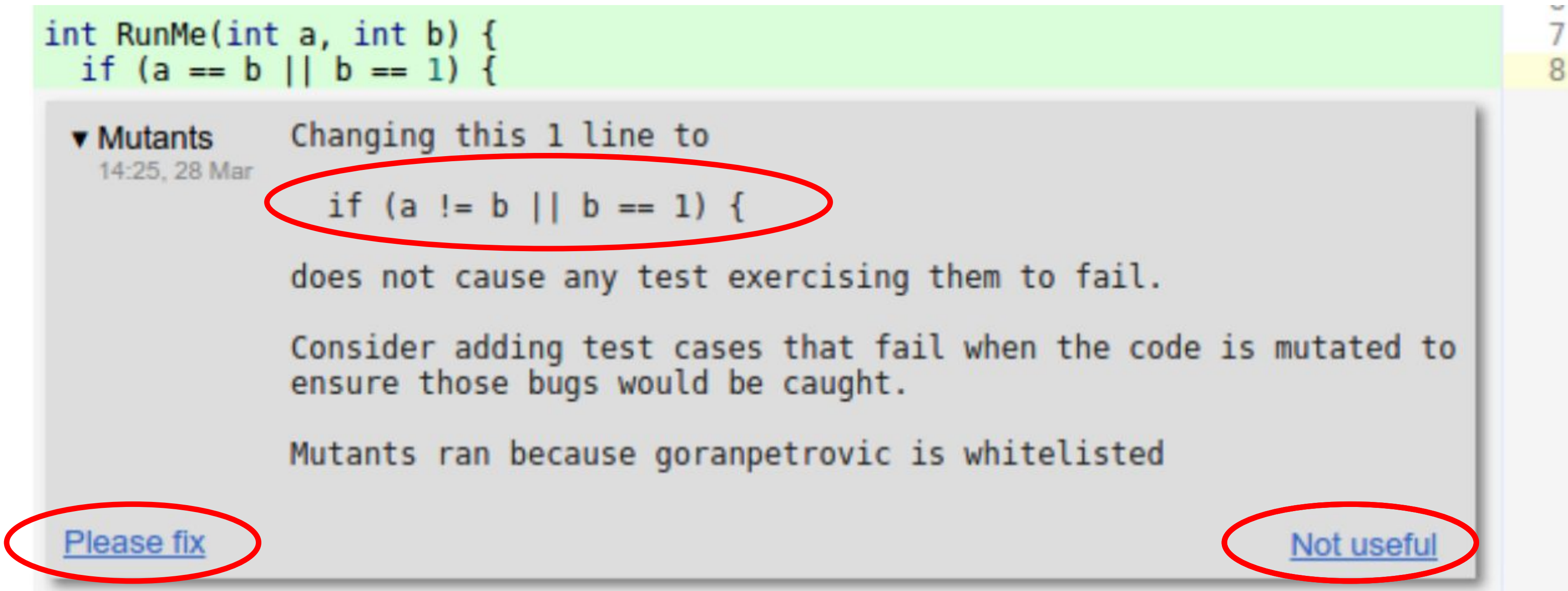
An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions (Reading)

Productive mutants: mutation testing at Google



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

Productive mutants: mutation testing at Google



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).