CSE 403

Software Engineering

Winter 2023

Mutation-based Testing

Recap: structural code coverage

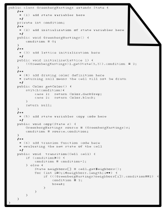
Classes in this File	Line Coverage		Branch Coverage		Complexity
Avg	100%	10/10	100%	8/8	6

```
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!

Mutation-based testing: the basics

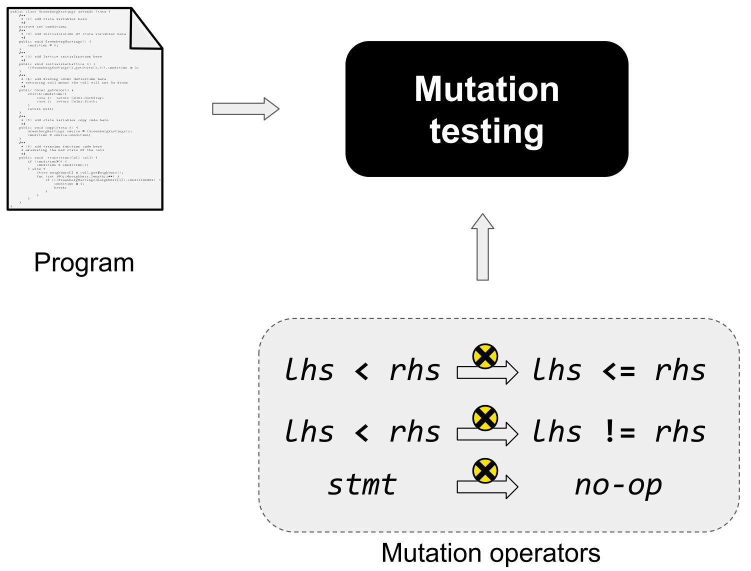
Mutation testing



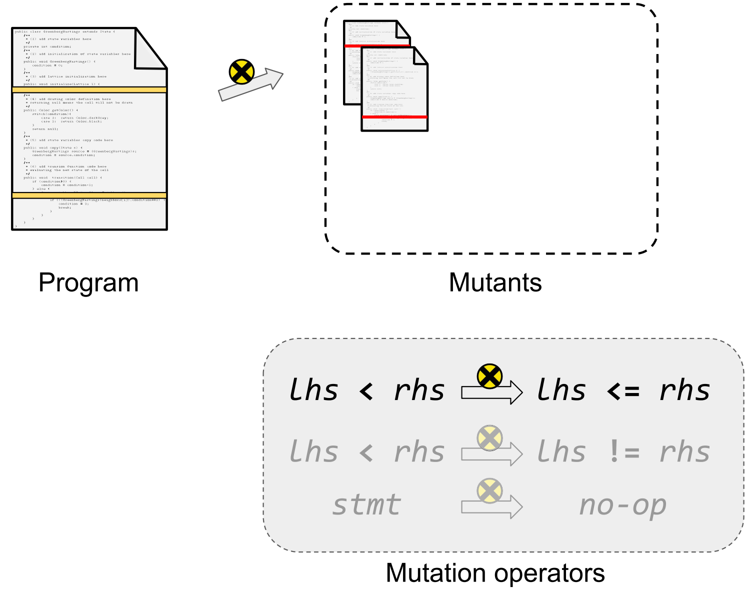
Program

Mutation testing

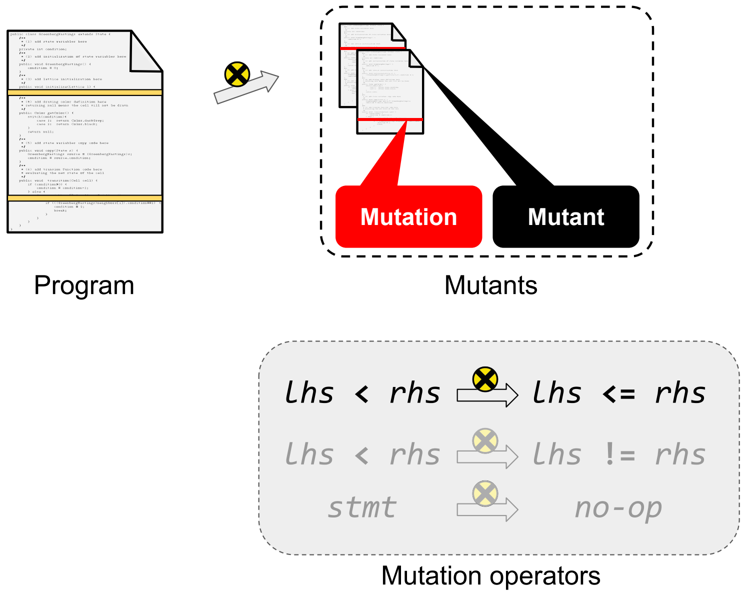
Mutation testing: mutant generation



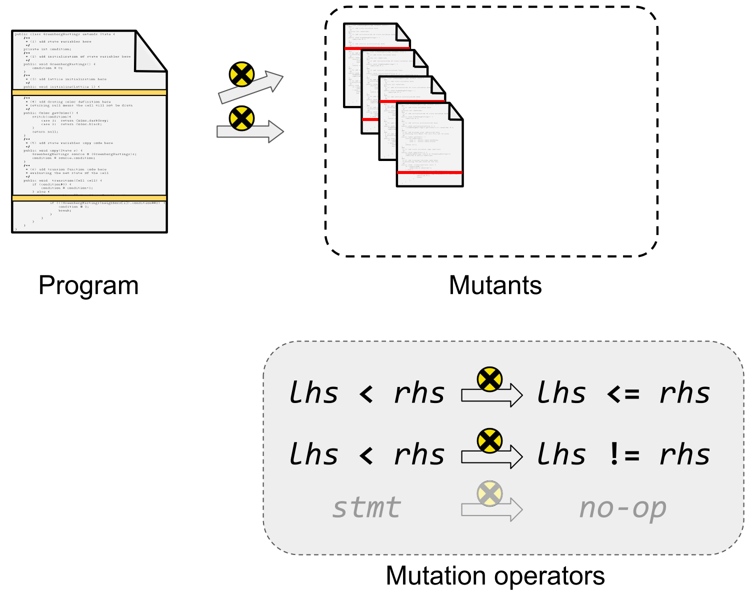
Mutation testing: mutant generation



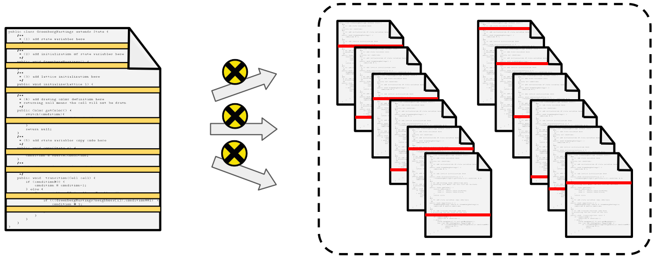
Mutation testing: mutant generation



Mutation testing: mutant generation



Mutation testing: mutant generation

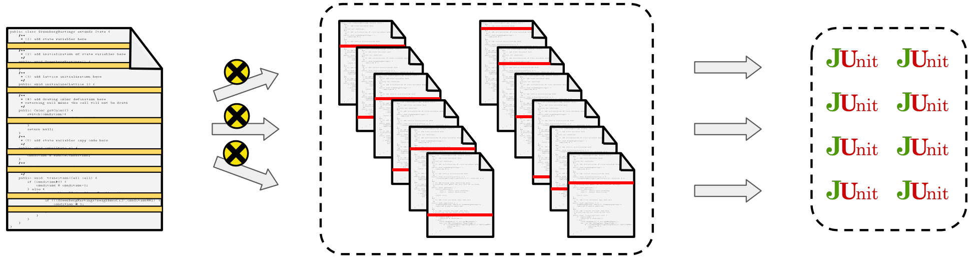


Program   Mutants

```
lhs < rhs  →  lhs <= rhs
lhs < rhs  →  lhs != rhs
stmt       →  no-op
```

Mutation operators

Mutation testing: test creation



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:

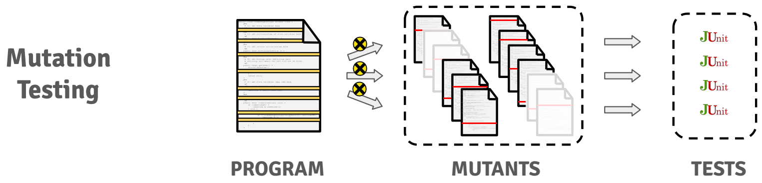
```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutants:

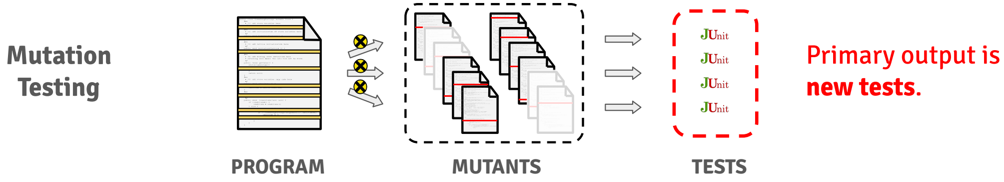
```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

For each mutant, provide a test case that detects it
(i.e., passes on the original program but fails on the mutant)

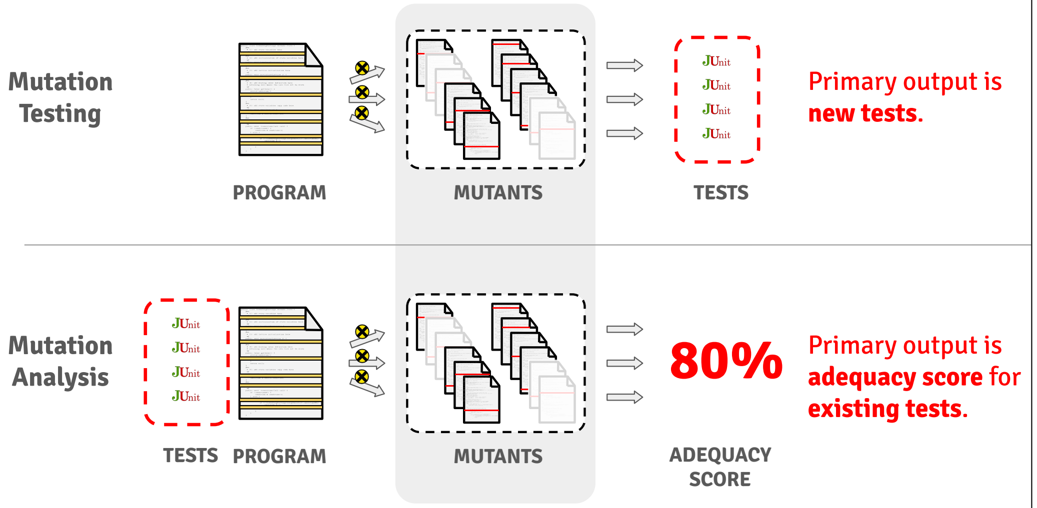
Mutation Testing vs. Mutation Analysis



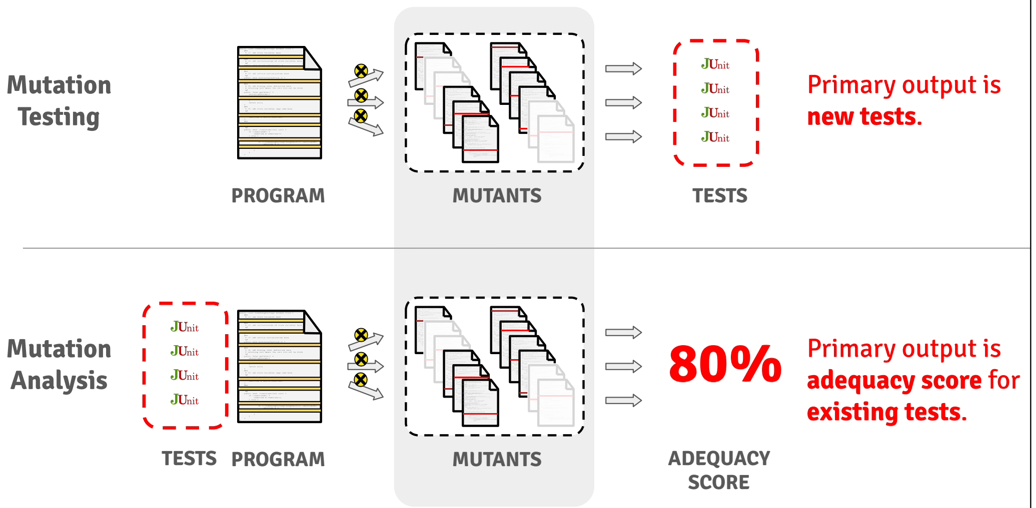
Mutation Testing vs. Mutation Analysis



Mutation Testing vs. Mutation Analysis



Mutation Testing vs. Mutation Analysis



How expensive is mutation testing?

Is the mutation score meaningful?

Mutation-based 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!

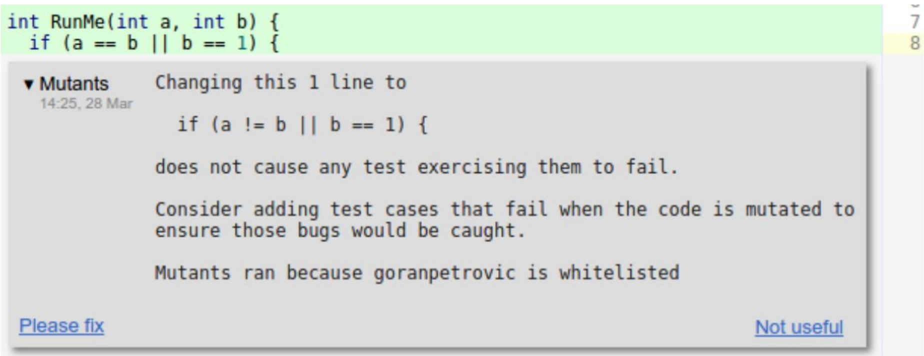
The notion of productive mutants is fuzzy!

A mutant is **productive** if it is

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

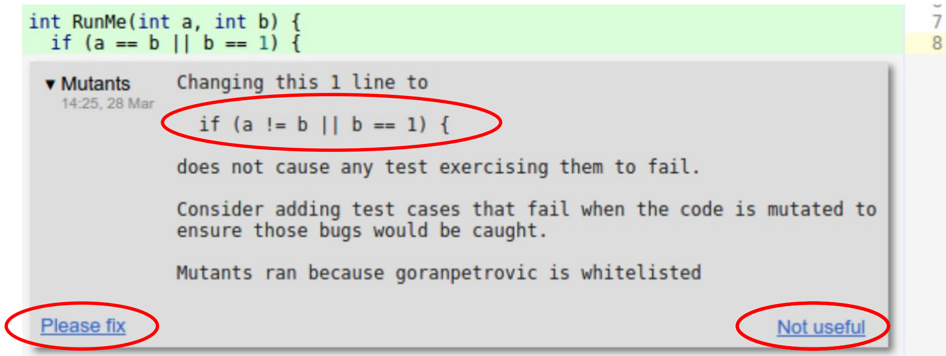
An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions (Reading)

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

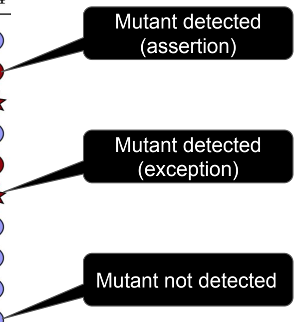
```
...
Set cache = new HashSet(a + b);
...
```

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

Mutation-based testing: mutant subsumption

Mutant subsumption

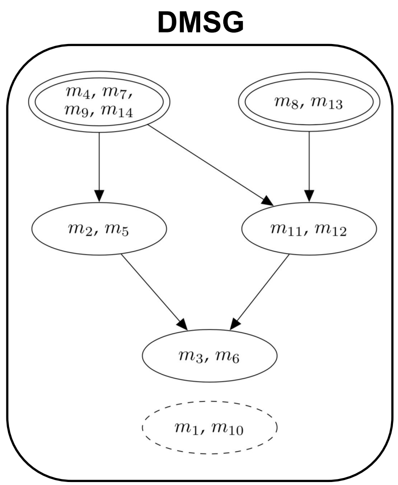
Mutant	Tests			
MutOp	t_1	t_2	t_3	t_4
m_1: $< \longmapsto$!=	○	○	○	○
m_2: $< \longmapsto$ ==	○	●	○	●
m_3: $< \longmapsto$ <=	✸	✸	✸	✸
m_4: $< \longmapsto$ >	○	●	○	○
m_5: $< \longmapsto$ >=	○	●	○	●
m_6: $< \longmapsto$ true	✸	✸	✸	✸
m_7: $< \longmapsto$ false	○	●	○	○
m_8: $< \longmapsto$!=	●	○	○	○
m_9: $< \longmapsto$ ==	○	●	○	○
m_{10}: $< \longmapsto$ <=	○	○	○	○
m_{11}: $< \longmapsto$ >	●	●	○	○
m_{12}: $< \longmapsto$ >=	●	●	○	○
m_{13}: $< \longmapsto$ true	●	○	○	○
m_{14}: $< \longmapsto$ false	○	●	○	○



Prioritizing Mutants to Guide Mutation Testing (*Reading*)

DMSG: Dynamic Mutant Subsumption Graph

Mutant	Tests			
MutOp	t_1	t_2	t_3	t_4
m_1: $< \longmapsto$!=	○	○	○	○
m_2: $< \longmapsto$ ==	○	●	○	●
m_3: $< \longmapsto$ <=	✸	✸	✸	✸
m_4: $< \longmapsto$ >	○	●	○	○
m_5: $< \longmapsto$ >=	○	●	○	●
m_6: $< \longmapsto$ true	✸	✸	✸	✸
m_7: $< \longmapsto$ false	○	●	○	○
m_8: $< \longmapsto$!=	●	○	○	○
m_9: $< \longmapsto$ ==	○	●	○	○
m_{10}: $< \longmapsto$ <=	○	○	○	○
m_{11}: $< \longmapsto$ >	●	●	○	○
m_{12}: $< \longmapsto$ >=	●	●	○	○
m_{13}: $< \longmapsto$ true	●	○	○	○
m_{14}: $< \longmapsto$ false	○	●	○	○



Prioritizing Mutants to Guide Mutation Testing (*Reading*)

Coverage-based vs. mutation-based testing

See dedicated Slides (4 pages).