# Mutation-based testing

**UW CSE P 504**

# Today

- Mutation-based testing
  - Fake bugs ≈ real bugs
  - Productive mutants
  - Mutant subsumption
- Coverage-based vs. mutation-based testing

# Mutation-based testing: the basics

# Mutation in brief

Coverage and mutation measure test suite quality (“adequacy”)

- coverage(S) = what % of the program is executed by S
- mutation_score(S) = what % of fake bugs are detected by S?
  - Which fake bugs are chosen?

Terminology:

- A mutation is a small program change that might be defective
- A mutant is a program with a fake bug
- A mutation operator creates mutations

# Uses for test suite quality metrics (e.g., coverage)

- Is test suite *S* good enough?*
- Which test suite is better, *S1* or *S2*?*
- Prioritize tests within the suite.
- Should *t* be added to *S*? Compare *S* to $S \cup \{t\}$.
- Should *t* be removed from *S*? Compare *S* to $S \setminus \{t\}$.
- What tests should I write to improve *S*?

# Mutation analysis:  computes an adequacy score

Test suite

**Mutation analysis**

Mutation score

Create many mutants

Run the test suite on each mutant (expensive!)

mutation score = % of mutants that fail at least one test

# Mutation coverage: computes a quality score

# ~~Mutation analysis: computes an adequacy score~~

JUnit JUnit
JUnit JUnit
JUnit JUnit
JUnit JUnit

Test suite

→

**Mutation coverage**

Create many mutants

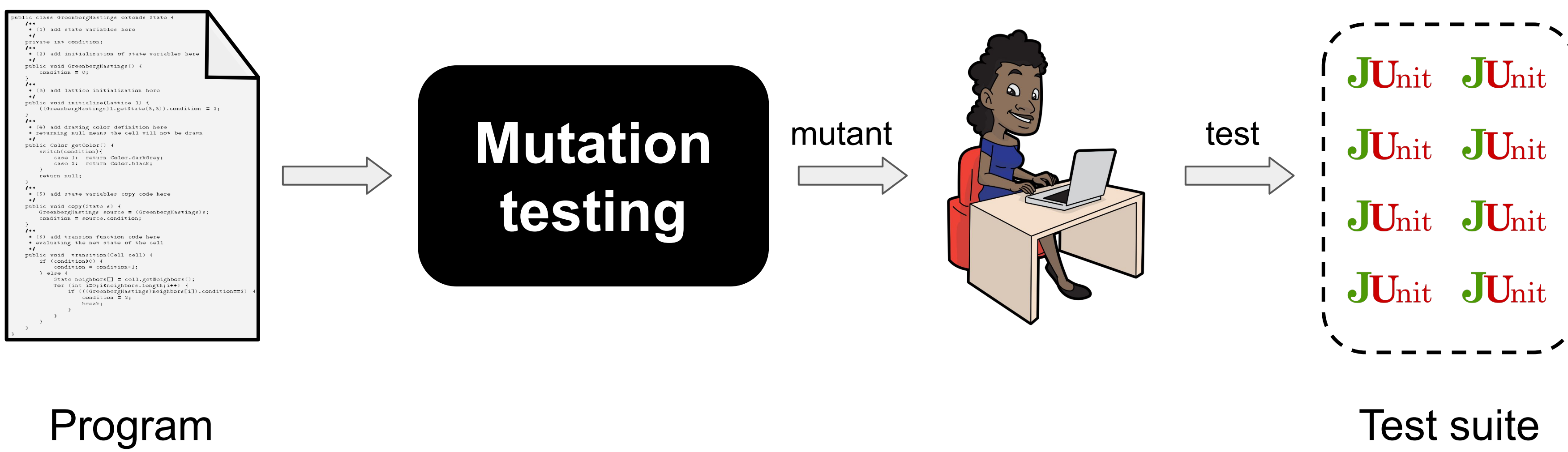
Run the test suite on each mutant (expensive!)

→

Mutation ~~score~~ coverage

mutation score = % of mutants that fail at least one test

# Mutation testing: guides the creation of tests

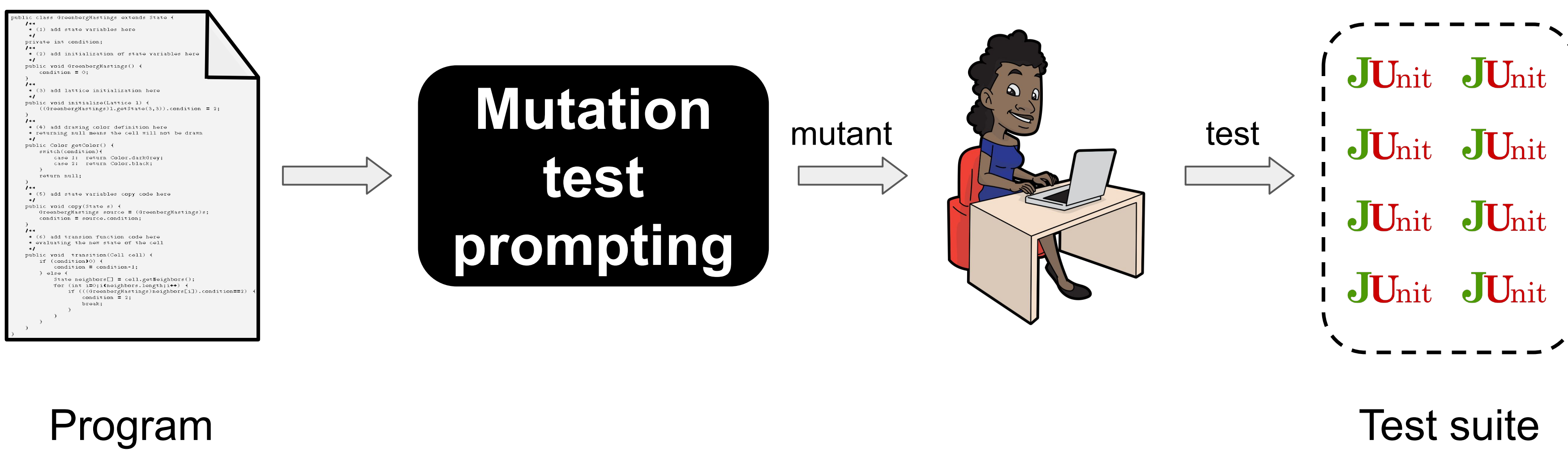



Tests can be created by a person or a tool

When to stop creating tests?

# Mutation-guided test prompting

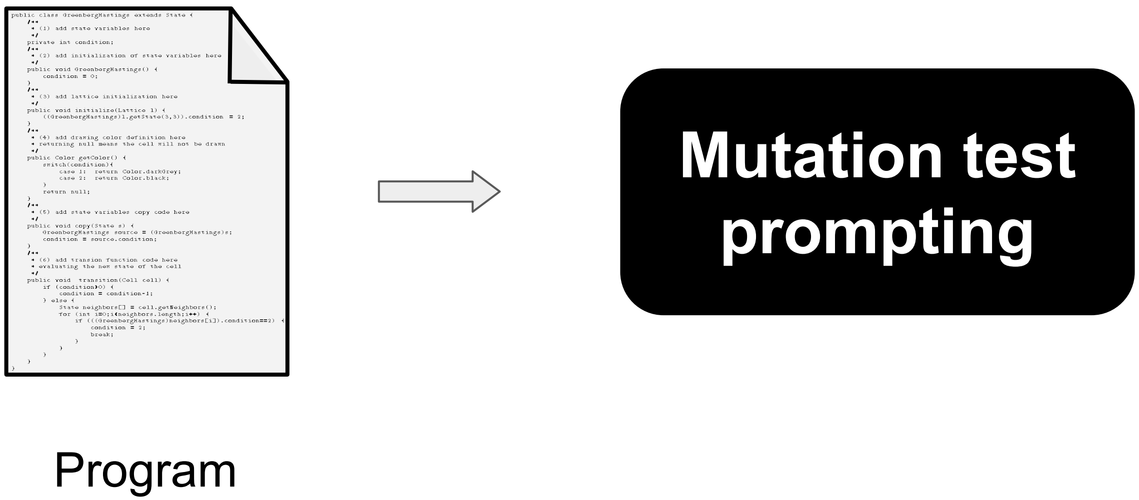
~~Mutation testing: guides the creation of tests~~



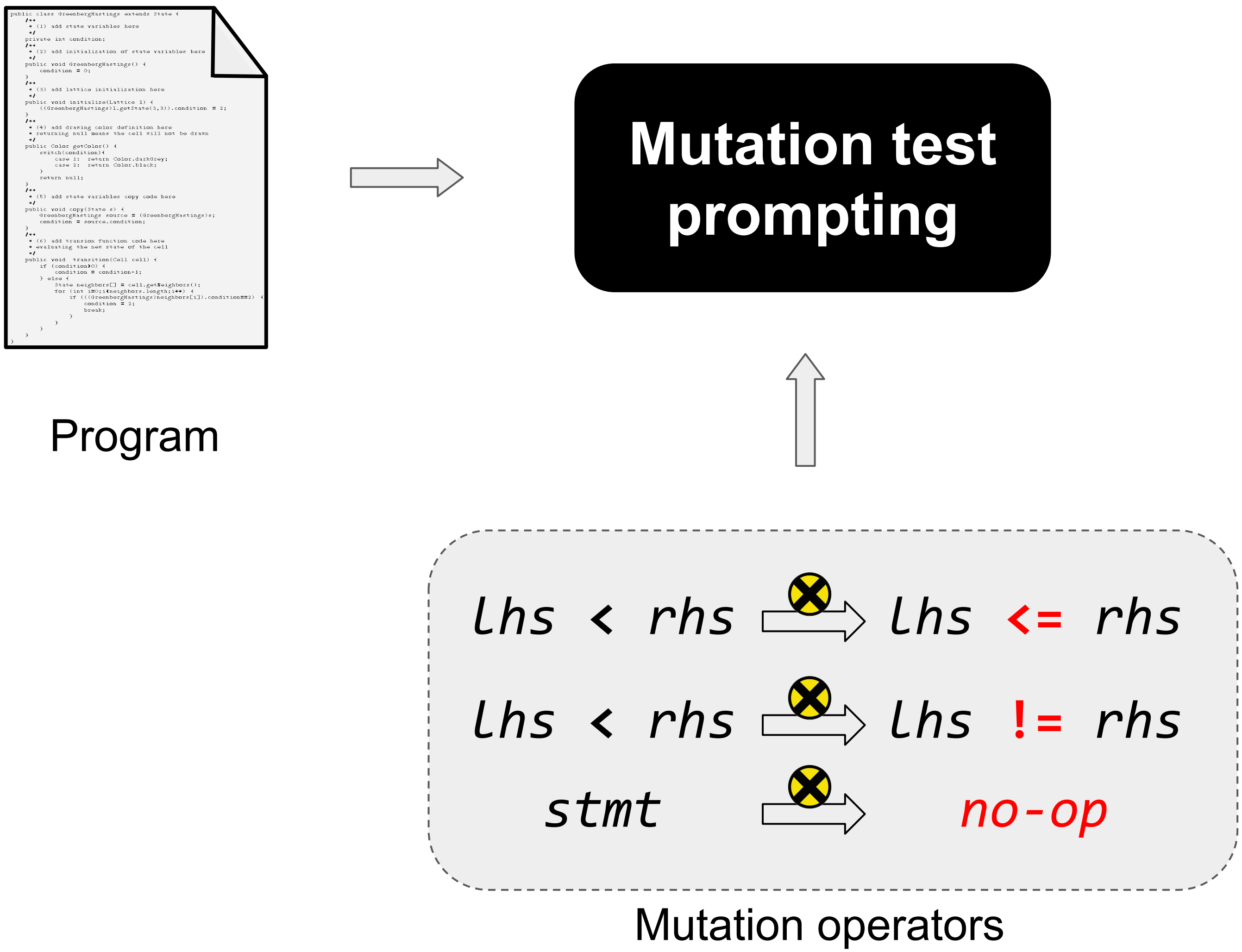

Tests can be created by a person or a tool

When to stop creating tests?

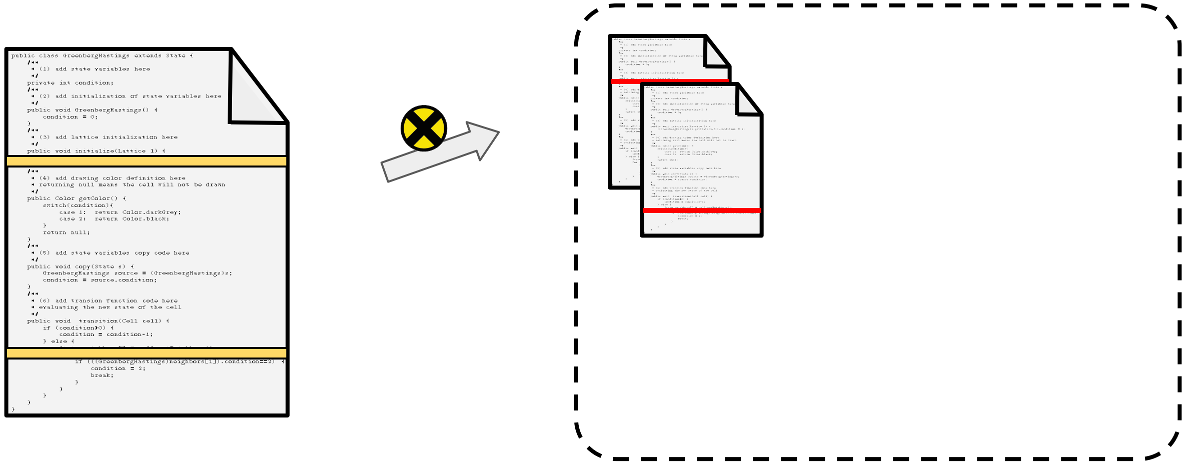
# Mutation testing: mechanism



# Mutation testing: mutant generation



# Mutation testing: mutant generation



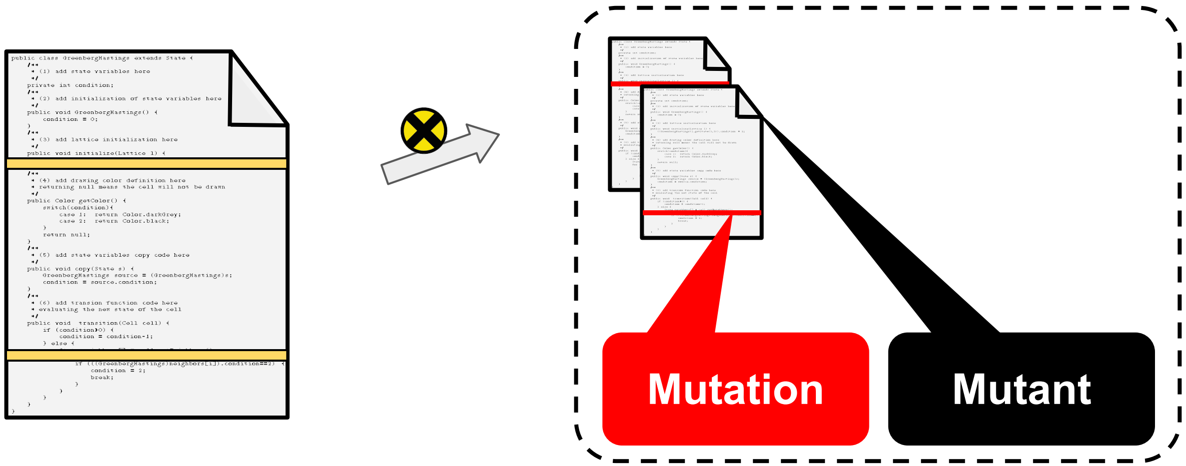
Program

Mutants

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

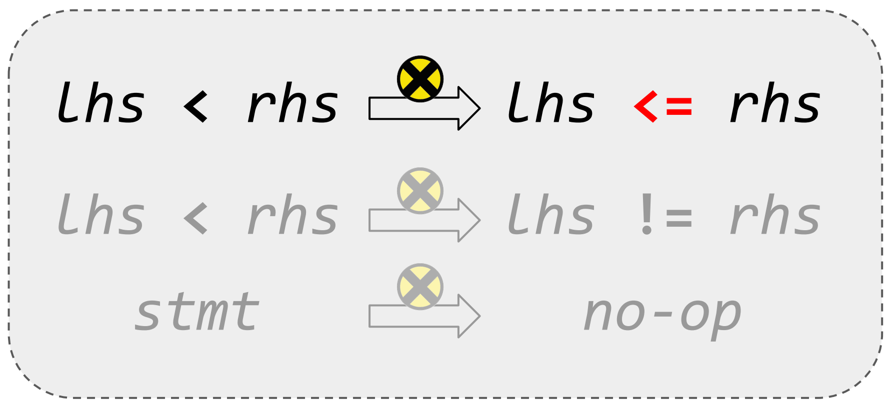
Mutation operators

# Mutation testing: mutant generation



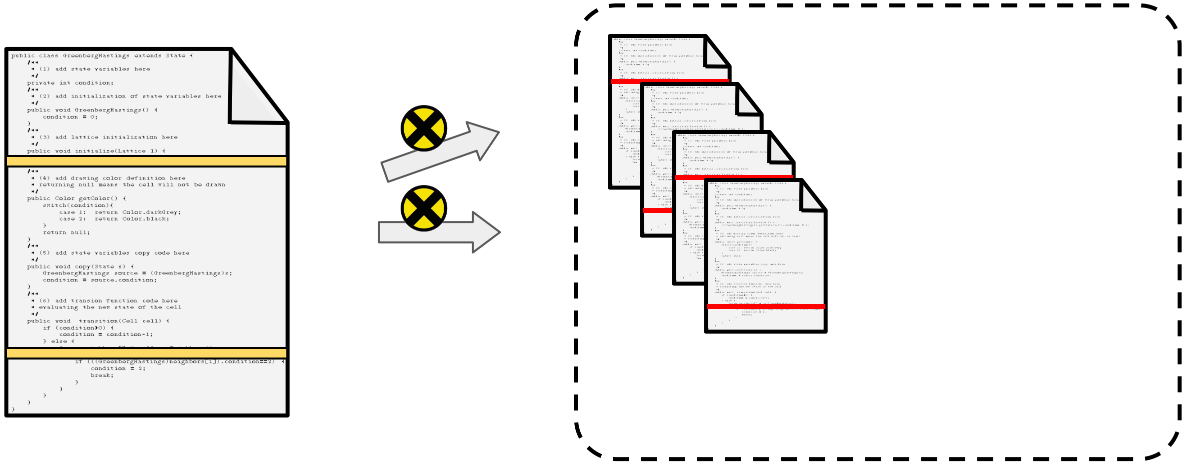

Program

Mutants




Mutation operators

# Mutation testing: mutant generation



Program

Mutants

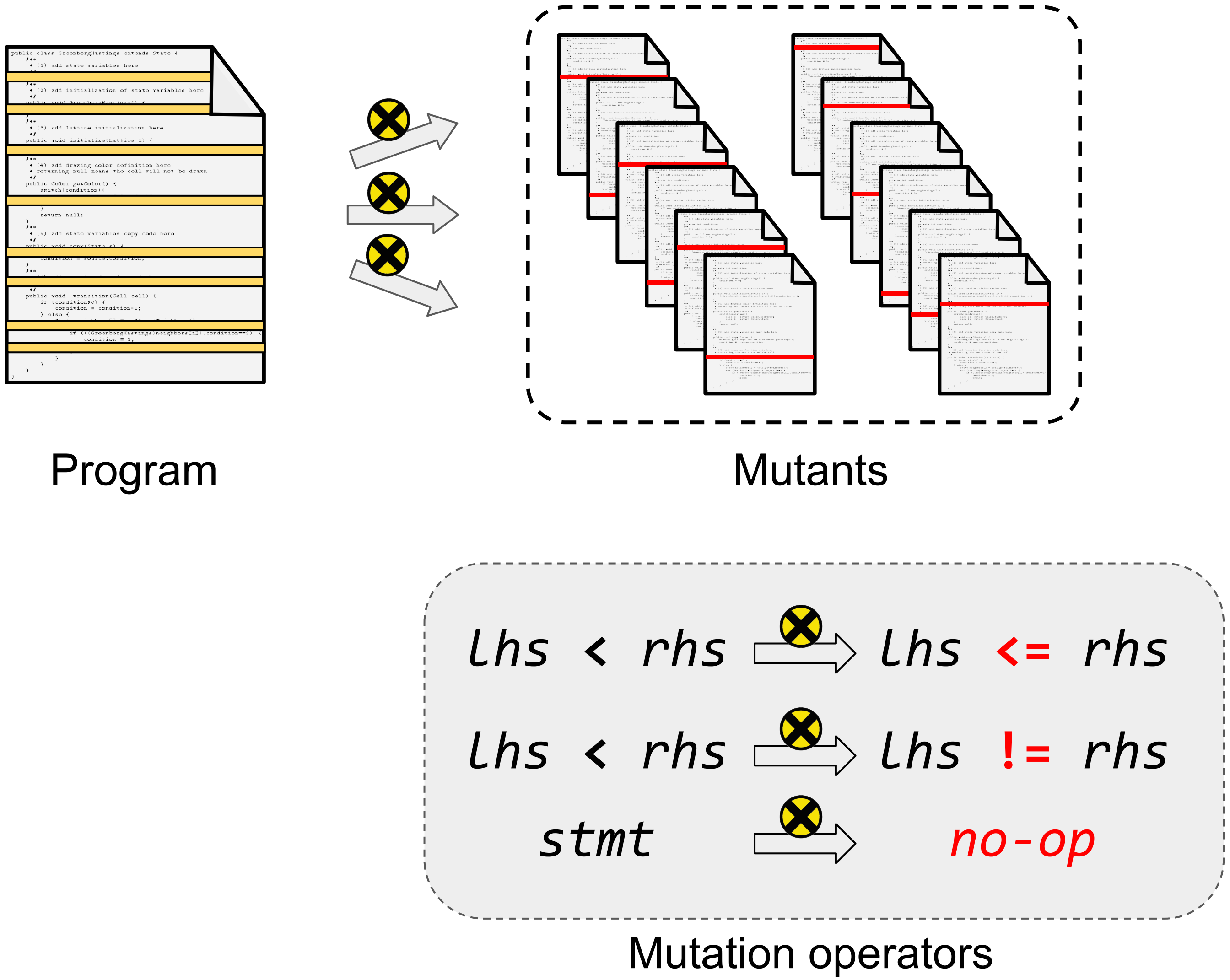
`lhs < rhs` → `lhs <= rhs`

`lhs < rhs` → `lhs != rhs`

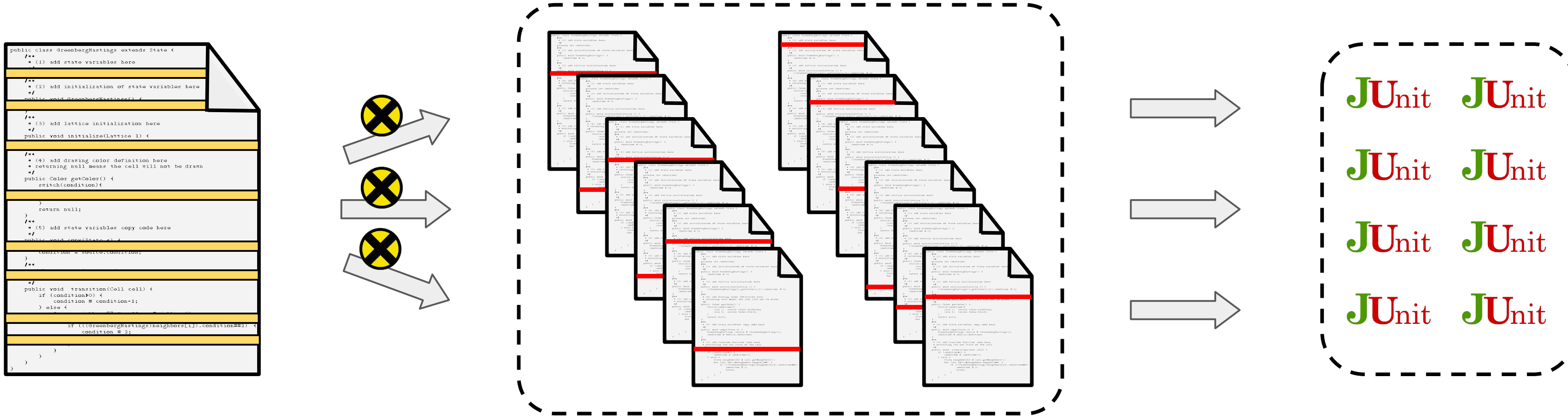
`stmt` → `no-op`

Mutation operators

# Mutation testing: mutant generation



# Mutation testing: scoring




Program

Mutants

Run the test suite on each mutant (expensive!)

## Assumptions

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

mutation score = % of mutants that fail at least one test

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

# Example mutant

**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;
}
```

# Another example mutant

**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;
}
```

# Yet another example mutant

**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;
}
```

# Last example mutant (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)

**In other words, create a test suite of maximal mutant score.**

# 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* | Asserted result | 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, to prompt the developer to write tests?**

| *a* | *b* | Asserted result | 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 |

# Detrimental mutants

What are analogous problems in statement coverage?

- Redundant mutant: is killed if the other mutants are killed
  - Inflates the mutant detection ratio
  - Hard to assess progress and remaining effort
- Equivalent mutant: behaves the same as the original program
  - 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 |

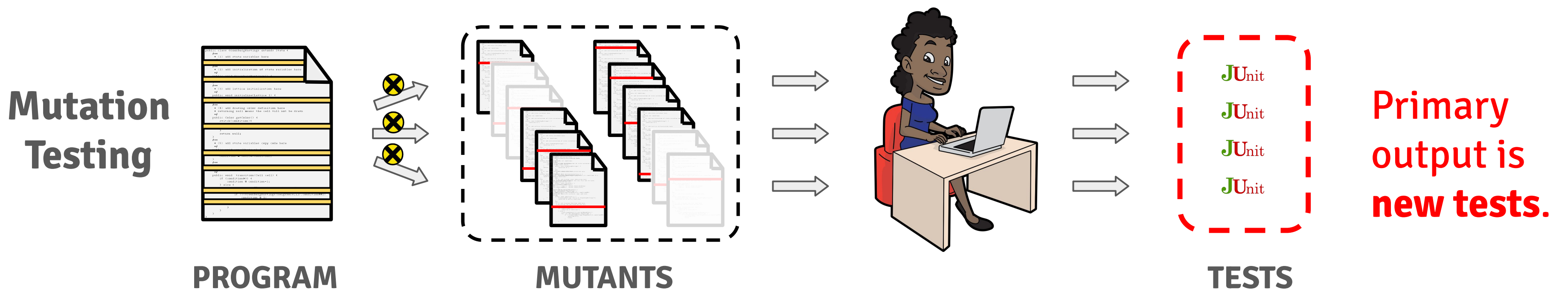
# Mutation testing vs. mutation analysis

Mutation test prompting vs. mutation coverage

~~Mutation testing vs. mutation analysis~~

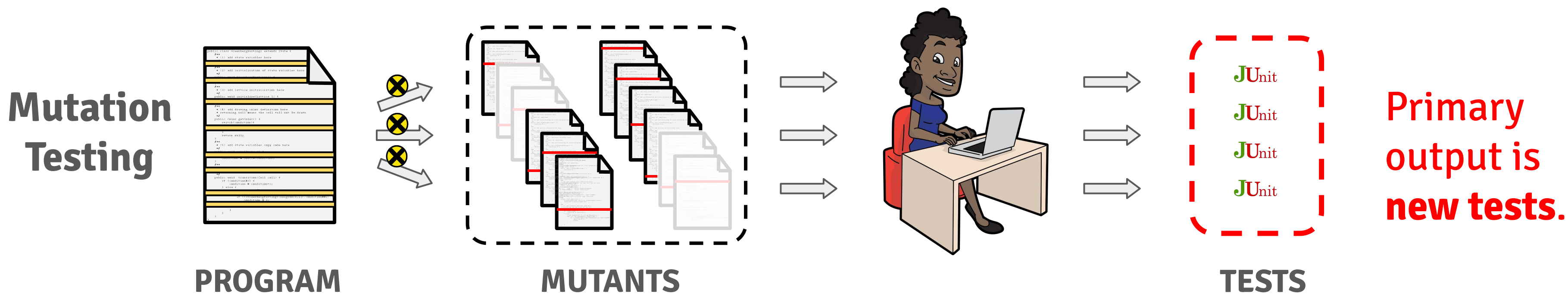
# Mutation test prompting vs. mutation coverage

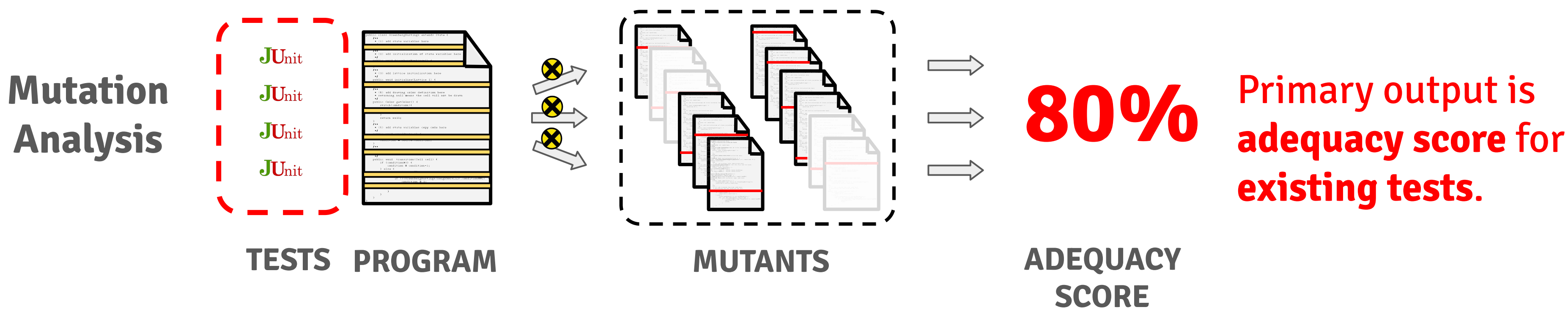
# ~~Mutation testing vs. mutation analysis~~



# Mutation test prompting vs. mutation coverage

~~Mutation testing vs. mutation analysis~~

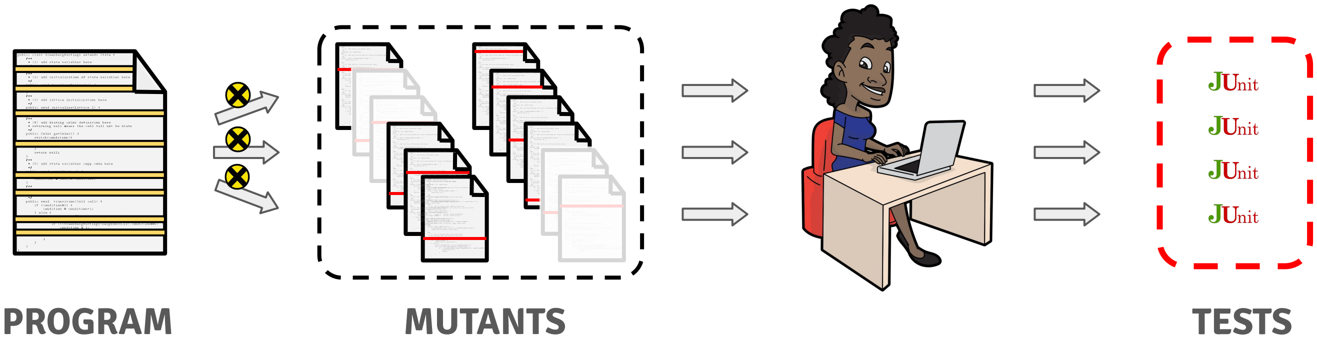


# Mutation test prompting vs. mutation coverage

# ~~Mutation testing vs. mutation analysis~~

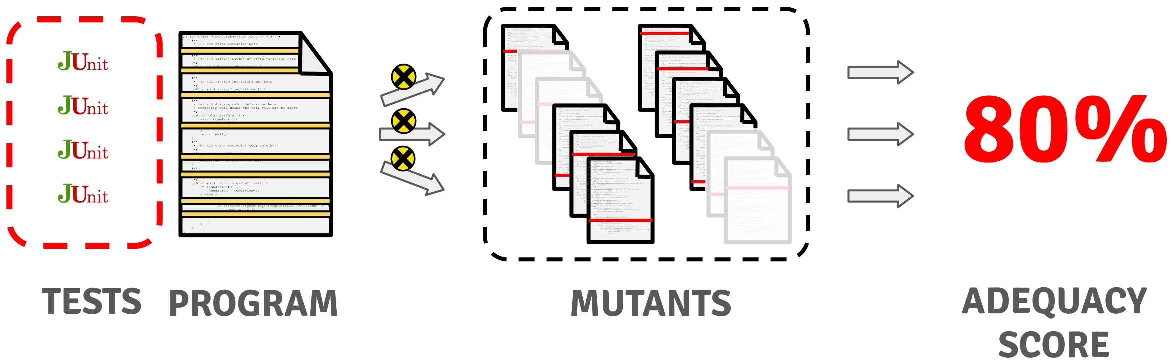
**Mutation Testing**




Primary output is **new tests.**

---

**Mutation Analysis**




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

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 1)*

# 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!

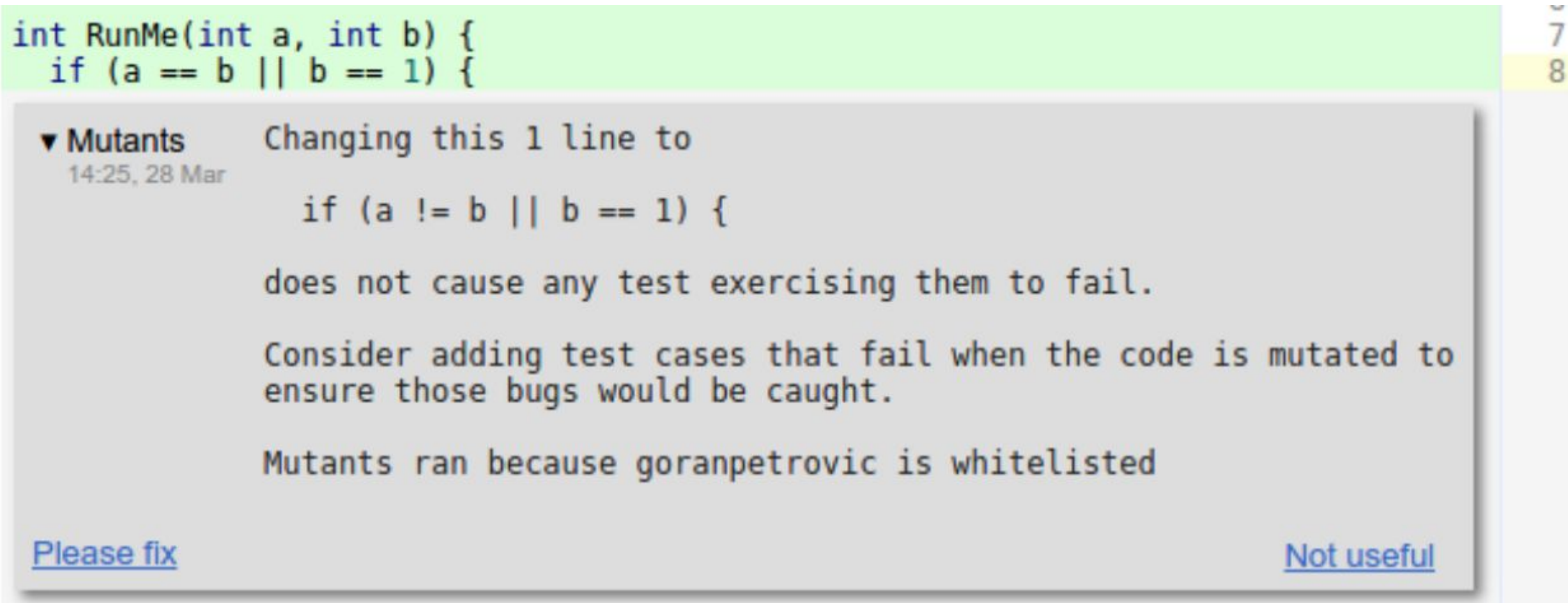
**The notion of productive mutants is fuzzy!**

A mutant is **productive** if it is

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

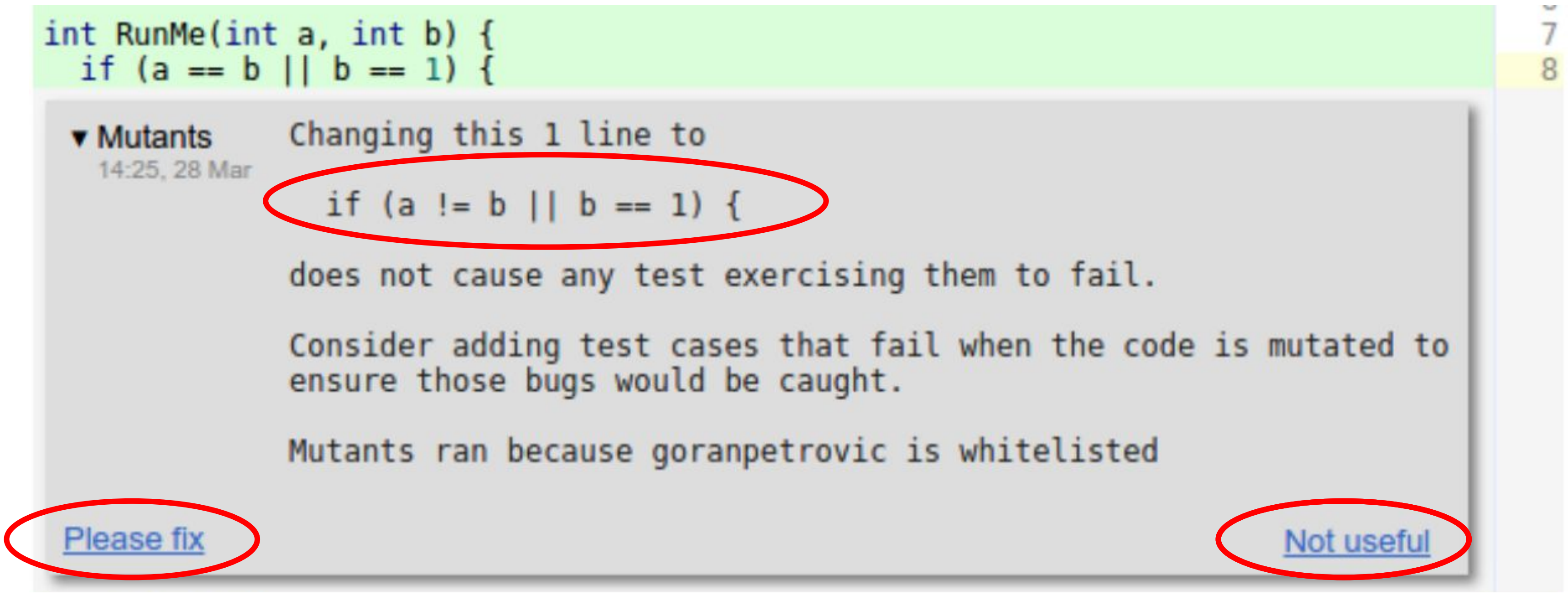
*An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions (Reading 1)*

# Productive mutants: mutation testing at Google




*Practical Mutation Testing at Scale: A view from Google (Reading 3)*

# Productive mutants: mutation testing at Google




*Practical Mutation Testing at Scale: A view from Google (Reading 3)*

# 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 **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);

...
```

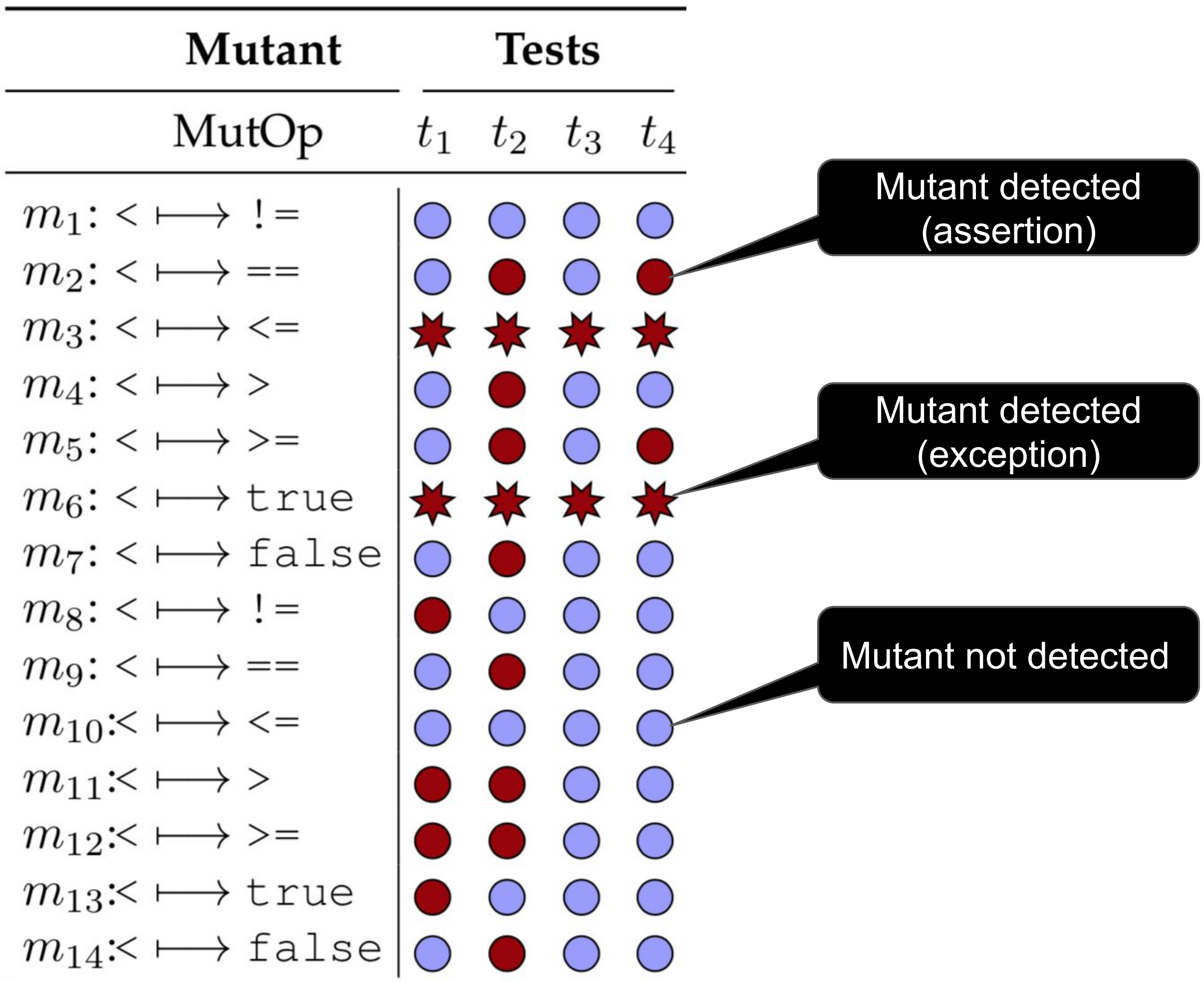
**Mutant**

```
...

Set cache = new HashSet(a + b);

...
```

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

# Mutation-based testing: mutant subsumption

# Mutant subsumption

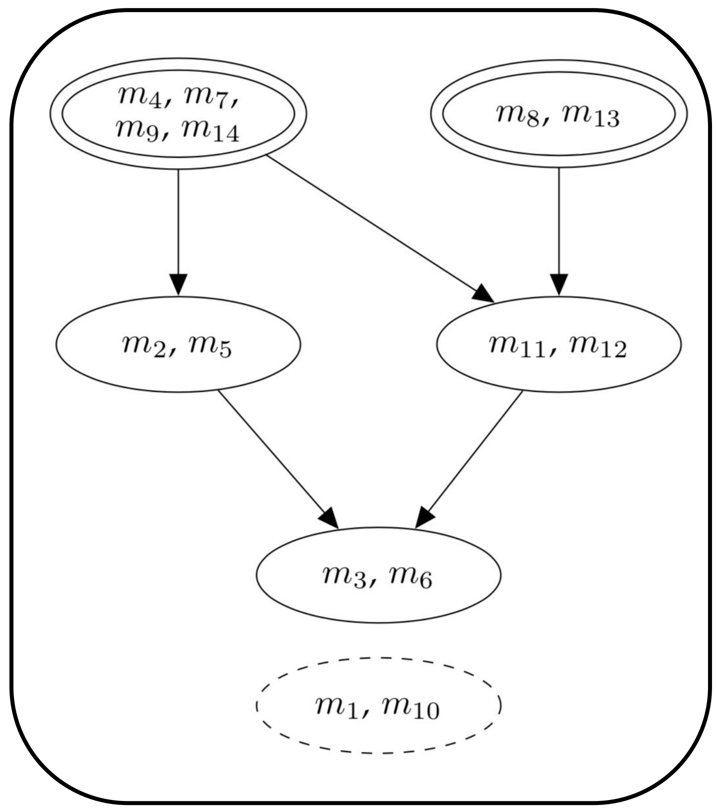



*Prioritizing Mutants to Guide Mutation Testing (Reading 2)*

# 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` | ○ | ● | ○ | ○ |

**DMSG**




*Prioritizing Mutants to Guide Mutation Testing ([Reading 2](Reading 2))*

# Coverage-based vs. mutation-based testing

*See dedicated Slides.*

# Teaser: delta debugging

From lecture 2: binary search is great. Example: git bisect.

What are the assumptions or limitations of binary search?

# Teaser: delta debugging

From lecture 2: binary search is great. Example: git bisect.

What are the assumptions or limitations of binary search?

- You are looking for one thing
- The search space is monotonic
- Every test yields “yes” or “no”

# Teaser: delta debugging

From lecture 2: binary search is great. Example: git bisect.

What are the assumptions or limitations of binary search?

- You are looking for one thing
- The search space is monotonic
- Every test yields “yes” or “no”

What can you do when these conditions are not met?

Examples: an image crashes a viewer

A program crashes a compiler

A webpage crashes a browser

How can you minimize these?

Searching for a *subset* of an input