

Mutation-based testing

UW CSE P 504

Today

- Mutation-based testing
 - Fake bugs \approx 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”)

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

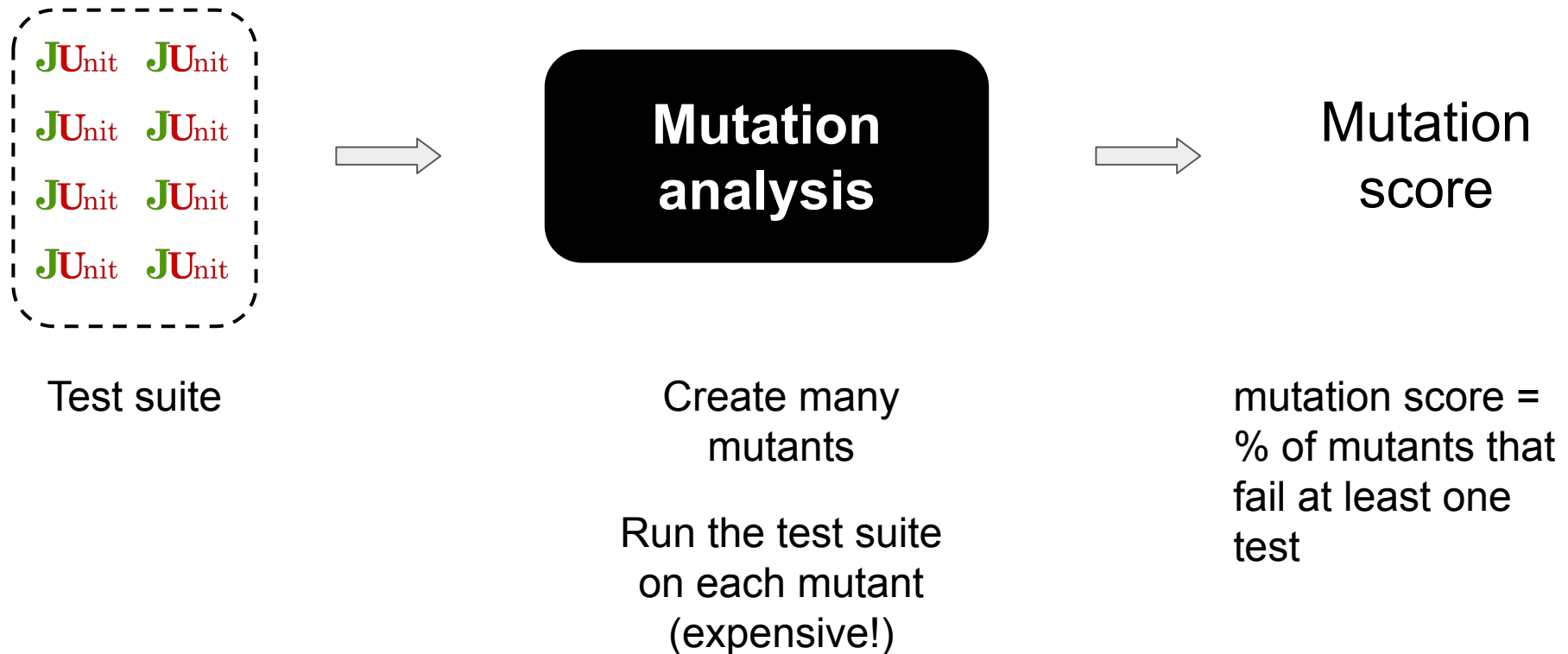
Terminology:

- 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)

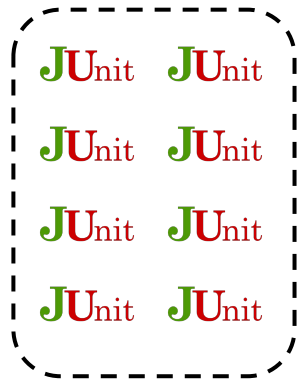
- Is test suite S good enough?*
- Which test suite is better, $S1$ or $S2$?*
- Prioritize tests within the suite.
- Should t be added to S ? Compare S to $S \cup \{t\}$.
- Should t be removed from S ? Compare S to $S \setminus \{t\}$.
- What tests should I write to improve S ?

Mutation analysis: computes an adequacy score



Mutation coverage: computes a quality score

~~Mutation analysis: computes an adequacy score~~



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

```
public class DrainingSettings extends Base {
    ... // add state variable here
}
private int condition;
... // add initialization of state variable here
public void drainingSettings() {
    condition = 0;
}
...
... // add initio initialization here
public void initializeInitio 2, {
    ... ((DrainingSettings) getInitio(1,3)).condition = 2;
}
...
... // add draining order definition here
... // setting null means the call will not be done
public Color getColor() {
    return condition;
}
... // add state variable copy code here
public void copyState() {
    ... ((DrainingSettings) getInitio(1,3)).condition = condition;
}
... // add reason function code here
... // deleting the state of the call
public void reasonFunctionCall() {
    if (condition == 0) {
        condition = condition-1;
    } else {
        stateNeighbor() = null;
        for (int i = 0; i < stateNeighbor().length; i++) {
            ... ((DrainingSettings) getInitio(1,3)).condition = 2;
            break;
        }
    }
}
}
```

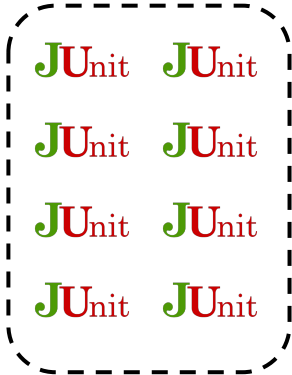
Program



mutant



test



Test suite

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~~

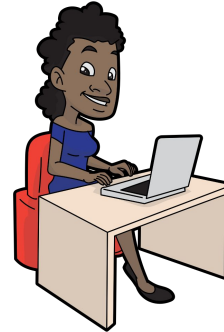
```
public class DrumsbergSettings {
    ...
    // add state variable here
    ...
    private int condition;
    ...
    // add initialization of state variable here
    ...
    public void setUpDrumsbergSettings() {
        condition = 0;
    }
    ...
    // add initio initialization here
    ...
    public void initializeInstance 2) {
        ... (DrumsbergSettings) getInstace(1); condition = 2);
    }
    ...
    // add during order definition here
    ...
    // setting null means the call will not be done
    ...
    public Color getColor() {
        return condition;
    }
    ...
    // add state variable copy code here
    ...
    public void copyState() {
        ... (DrumsbergSettings) state = (DrumsbergSettings);
        condition = state.condition;
    }
    ...
    // add reverse function code here
    ...
    // deleting the not state of the call
    ...
    public void reverseCall() {
        if (condition == 0) {
            condition = condition - 1;
        } else {
            state.neighbor() = null; getNeighbor();
            for (int i = 0; i < state.getNeighbor().length; i++) {
                if ((DrumsbergSettings) getInstace(1); condition == 0) {
                    state();
                }
            }
        }
    }
}
}
```

Program

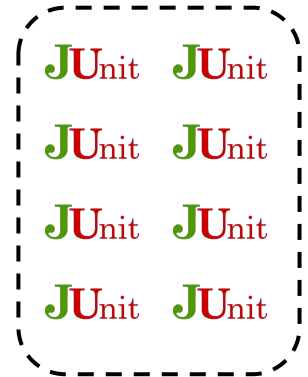


**Mutation
test
prompting**

mutant



test



Test suite

Tests can be created by a person or a tool

When to stop creating tests?

Mutation testing: mechanism

```
public class DrumborgSettings { private static *  
 * // add state variable here  
 *  
 private int condition;  
 *  
 * // add initialization of state variable here  
 *  
 public void DrumborgSettings() {  
     condition = 0;  
 }  
 *  
 * // add initioe initialization here  
 *  
 public void initializeInstance 21 {  
     ((DrumborgSettings) this).getInstanc(1,3); condition = 3;  
 }  
 *  
 * // add destroy order definition here  
 *  
 * // setting null means the call will not be done  
 *  
 public void getColors() {  
     return condition;  
     case 1: return Color.BLUE; // case 2: return Color.GREEN;  
 }  
 *  
 * // add state variable copy code here  
 *  
 public void copyState() {  
     DrumborgSettings dest = new DrumborgSettings();  
     condition = dest.condition;  
 }  
 *  
 * // add traverse function code here  
 *  
 * // skipping the rest state of the call  
 *  
 public void traverseCall (call) {  
     if (condition == 0) {  
         condition = condition-1;  
     } else {  
         this.neighbor() = null.getNeighbor();  
         for (int i = 0; i < neighbor.length; i++) {  
             ((DrumborgSettings) neighbor(i)).condition = 2;  
             break;  
         }  
     }  
 }  
 }  
 }
```

Program



**Mutation test
prompting**

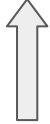
Mutation testing: mutant generation

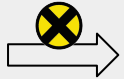
```
public class GreengrassSettings extends Base {
    ...
    // add state variable here
    ...
    private int condition;
    ...
    // add initialization of state variable here
    ...
    public void doSomething() {
        condition = 0;
    }
    ...
    // add initio initialization here
    ...
    public void initializeInitio 2, 4,
        ...((GreengrassSettings) getInitio(1,3)).condition = 3;
    }
    ...
    // add destroy order definition here
    ...
    // deleting null means the call will not be done
    ...
    public Color getColor() {
        ...
        case 1: return Color.BLUE;
        case 2: return Color.GREEN;
    }
    return null;
}
...
// add state variable copy code here
...
public void copyState() {
    GreengrassSettings dest = (GreengrassSettings)
        condition = source.condition;
}
...
// add traversal function code here
// deleting the last state of the call
public void traverseCall(int i) {
    if (condition == 0) {
        condition = condition - 1;
    } else {
        state.nextNode() = null;
        for (int i = 0; i < state.length(); i++) {
            if ((GreengrassSettings) getInitio(1,3).condition == 0) {
                break;
            }
        }
    }
}
}
```

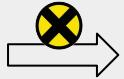
Program

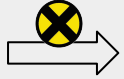


Mutation test prompting



Lhs < *rhs*  *Lhs* <= *rhs*

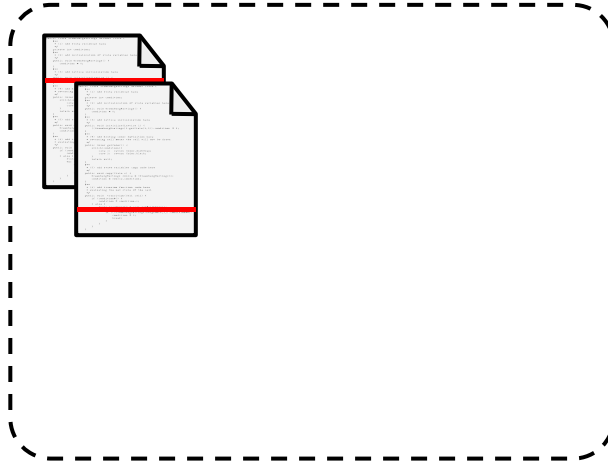
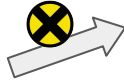
Lhs < *rhs*  *Lhs* != *rhs*

stmt  *no-op*

Mutation operators

Mutation testing: mutant generation

```
public class DrumburgSettings 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 drying state definition here  
    * (missing null means the call will not be done)  
    *  
    public Color getColor() {  
        return condition;  
        case 1: return Color.BLUE;  
        case 2: return Color.BLACK;  
    }  
    *  
    * () add state variable copy code here  
    *  
    public void copyState() {  
        doSomething();  
        condition = condition;  
    }  
    *  
    * () add reverse function code here  
    * (skipping the set state of the call)  
    public void reverseCall() {  
        if (condition() {  
            condition = condition-1;  
        } else {  
            condition = condition + 1;  
        }  
    }  
}
```



Program

Mutants

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

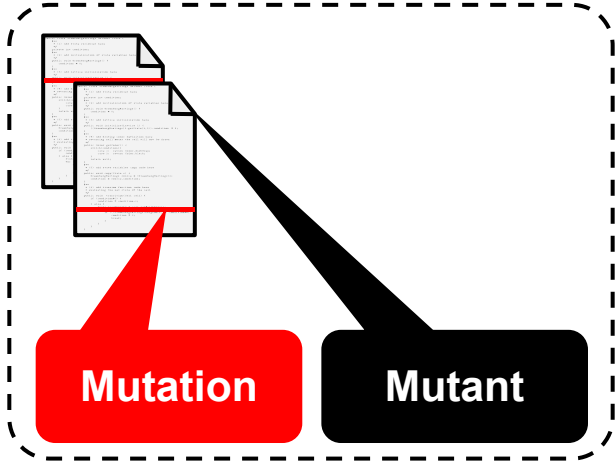
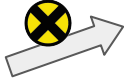
Mutation operators

Mutation testing: mutant generation

```

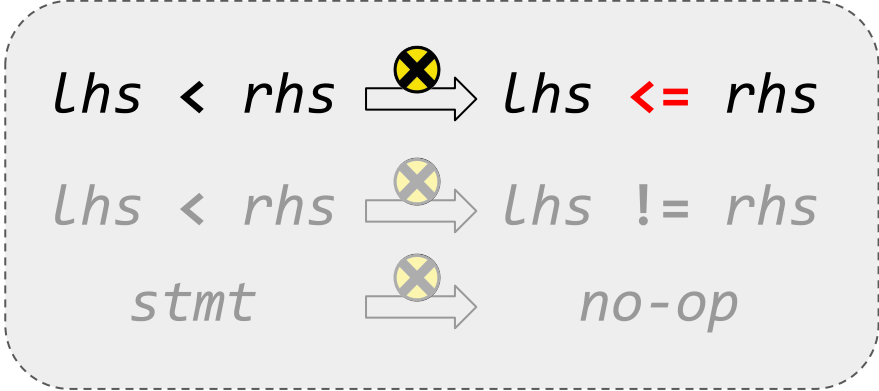
public class DrumburgSettings {
    /**
     * (1) add state variable here
     */
    private int condition;
    /**
     * (2) add initialization of state variable here
     */
    public void doSomething() {
        condition = 0;
    }
    /**
     * (3) add lattice initialization here
     */
    public void initializeLattice () {
    }

    /**
     * (4) add drawing state definition here
     * (5) adding null means the call will not be done
     */
    public Color getColor() {
        return condition;
    }
    /**
     * (6) add state variable copy code here
     */
    public void copyState () {
        doSomething();
    }
    /**
     * (7) add random function code here
     * (8) deleting the last state of the call
     */
    public void getRandomCall () {
        condition = condition - 1;
    }
}
    
```



Program

Mutants

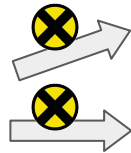


Mutation operators

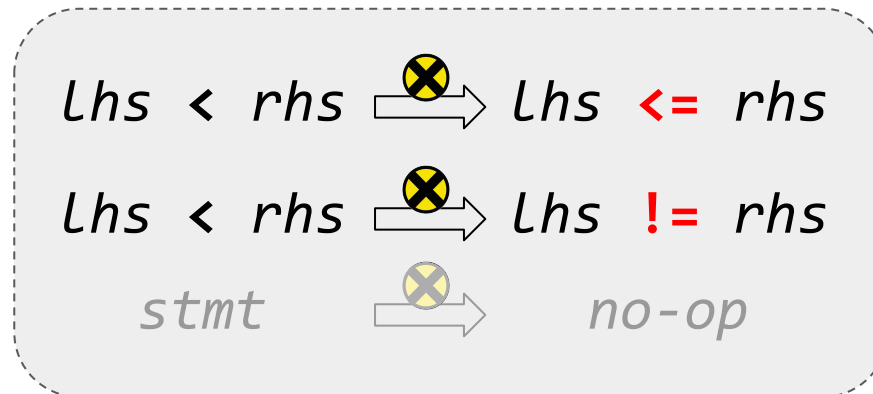
Mutation testing: mutant generation

```
public class GreengrassSettings extends Base {
    * // add state variable here
    private int condition;
    private int condition;
    * // add initialization of state variable here
    public void doSomething() {
        condition = 0;
    }
    * // add lattice initialization here
    * //
    public void initializeLattice () {
    }
    * // add lattice state definition here
    * // existing 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 () {
        GreengrassSettings settings = (GreengrassSettings)
            condition = condition;
    }
    * //
    * //
    * // add traversal function code here
    * //
    public void traverseFunction() {
        // do something
        if (condition) {
            condition = condition - 1;
        } else {
        }
    }
}
}
```

Program

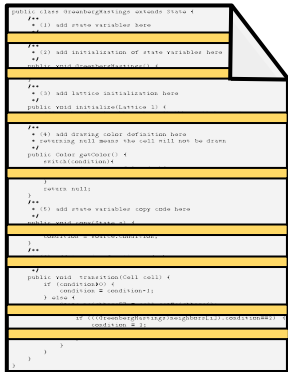


Mutants



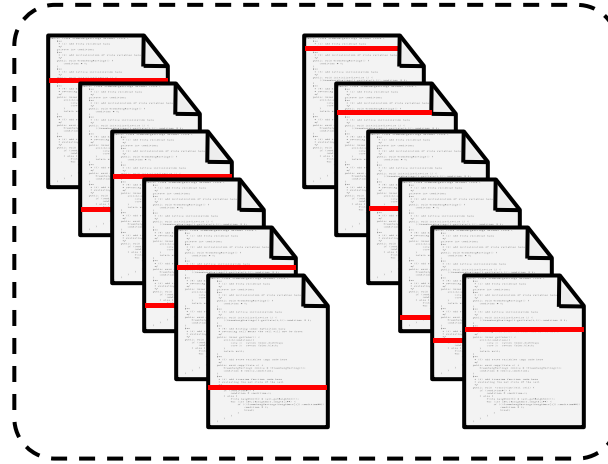
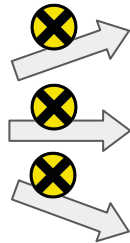
Mutation operators

Mutation testing: mutant generation

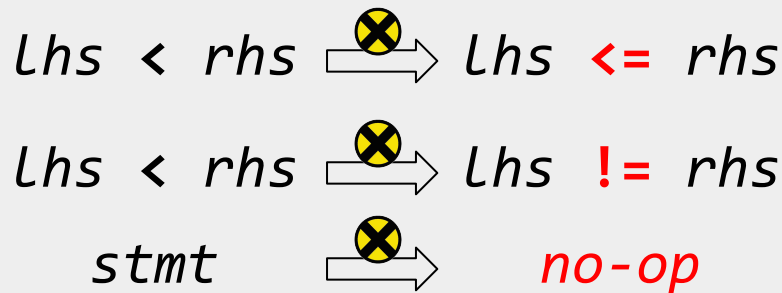


```
public class DriverManager { ... }
...
// (1) add private variable here
...
...
// (2) add private initialization here
public void initializeInstance () {
...
}
...
// (3) add private static definition here
// (WARNING: null means the call will not be done)
public static getDriver() {
...
}
...
// (4) add private variable copy code here
// (WARNING: null means the call will not be done)
...
...
public void transformToSql () {
...
}
...
}
}
```

Program



Mutants



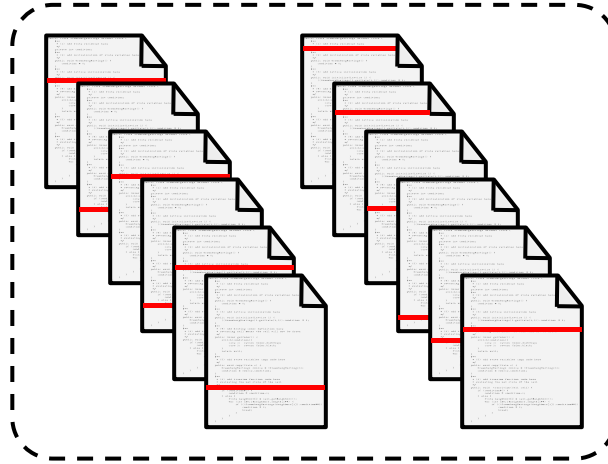
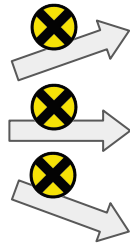
Mutation operators

Mutation testing: scoring

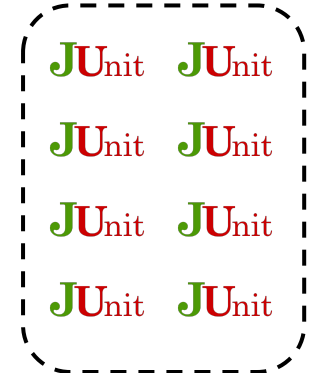
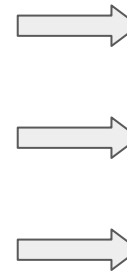


```
public class ExampleSettings { ... }
...
// (1) add private variable here
...
/**
 * (1) add instance initialization here
 * (2) add initialization of private variable here
 */
public void initializeInstance () {
...
}
/**
 * (1) add logging after definition here
 * (WARNING: full name: the call will not be done)
 */
public void getOutput () {
...
}
...
}
/**
 * (1) add private variable copy code here
 * (2) add initialization of private variable here
 */
...
}
public void transformToCall () {
...
}
}
}
```

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).

<i>a</i>	<i>b</i>	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?

<i>a</i>	<i>b</i>	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

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

What are analogous problems in statement coverage?

Detrimental mutants

- 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 \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

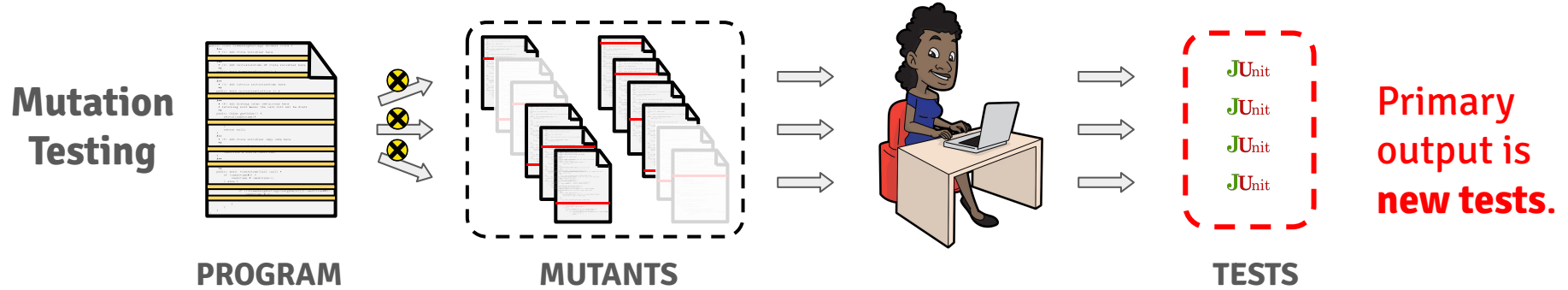
Mutation testing vs. mutation analysis

Mutation test prompting vs. mutation coverage

~~Mutation testing vs. mutation analysis~~

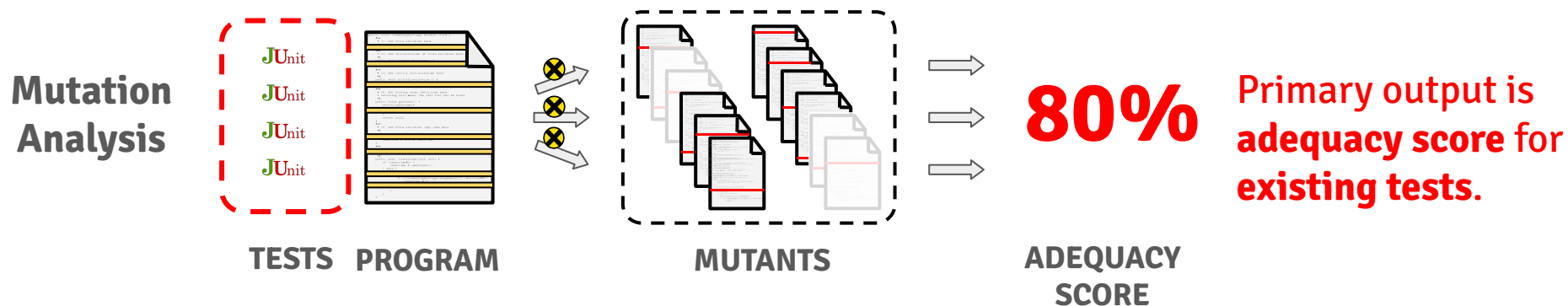
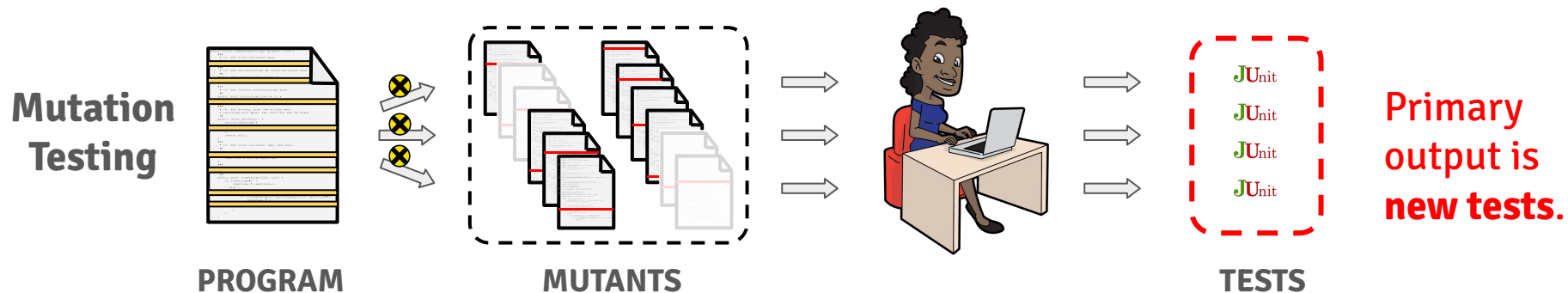
Mutation test prompting vs. mutation coverage

~~Mutation testing vs. mutation analysis~~



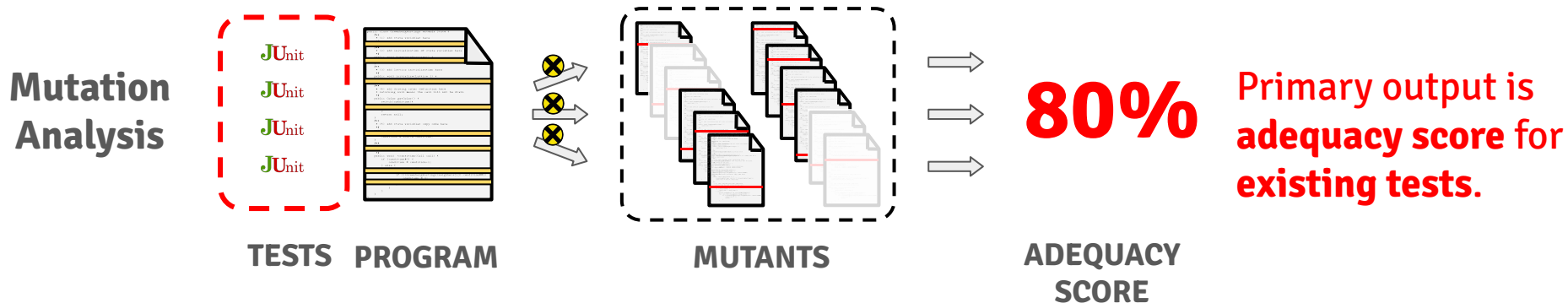
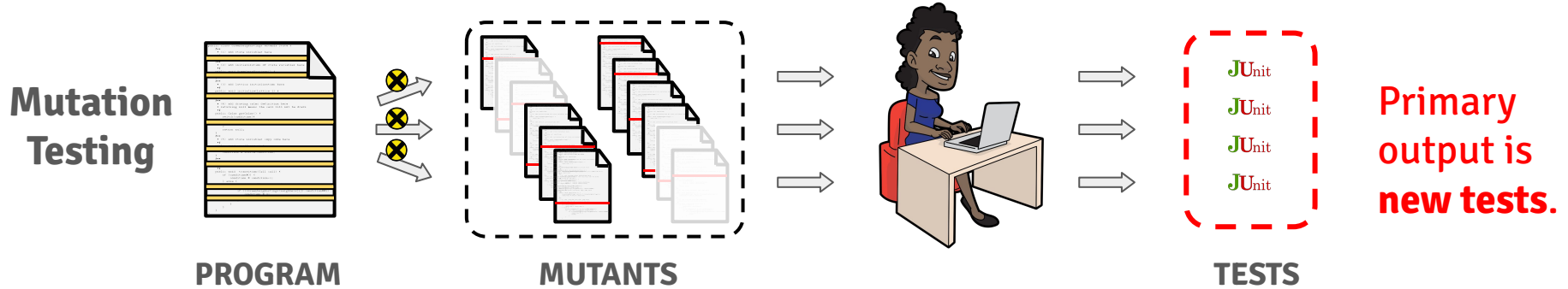
Mutation test prompting vs. mutation coverage

~~Mutation testing vs. mutation analysis~~



Mutation test prompting vs. mutation coverage

~~Mutation testing vs. mutation analysis~~



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

Mutation-based 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 **improves 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 3](#))

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 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

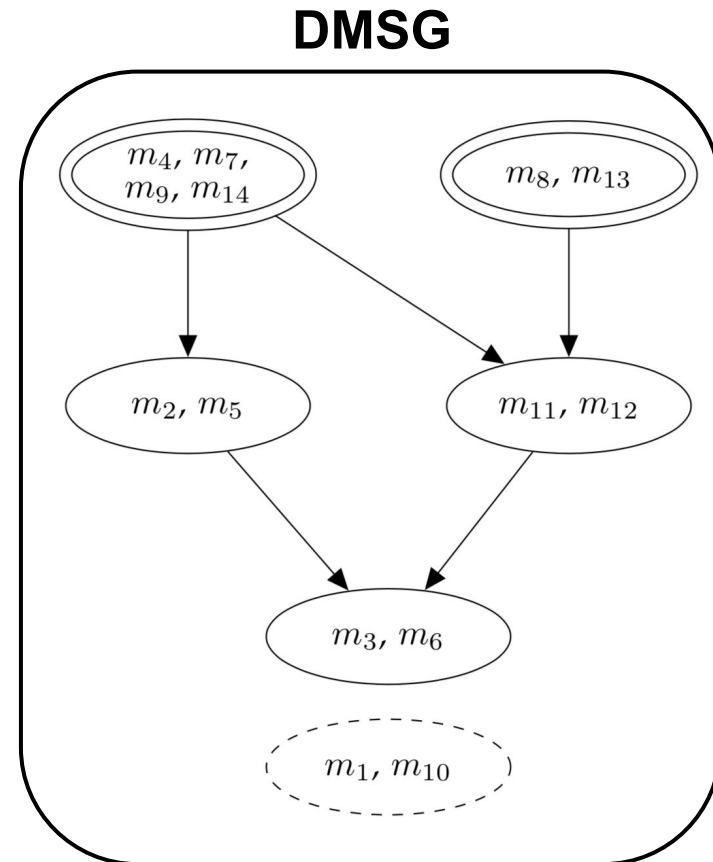
Mutant subsumption

Mutant	Tests					
	MutOp	t_1	t_2	t_3	t_4	
$m_1: < \mapsto !=$		○	○	○	○	Mutant detected (assertion)
$m_2: < \mapsto ==$		○	●	○	●	
$m_3: < \mapsto <=$		★	★	★	★	Mutant detected (exception)
$m_4: < \mapsto >$		○	●	○	○	
$m_5: < \mapsto >=$		○	●	○	●	
$m_6: < \mapsto \text{true}$		★	★	★	★	Mutant not detected
$m_7: < \mapsto \text{false}$		○	●	○	○	
$m_8: < \mapsto !=$		●	○	○	○	
$m_9: < \mapsto ==$		○	●	○	○	
$m_{10}: < \mapsto <=$		○	○	○	○	
$m_{11}: < \mapsto >$		●	●	○	○	
$m_{12}: < \mapsto >=$		●	●	○	○	
$m_{13}: < \mapsto \text{true}$		●	○	○	○	
$m_{14}: < \mapsto \text{false}$		○	●	○	○	

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 : $< \mapsto !=$		●	●	●	●
m_2 : $< \mapsto ==$		●	●	●	●
m_3 : $< \mapsto <=$		★	★	★	★
m_4 : $< \mapsto >$		●	●	●	●
m_5 : $< \mapsto >=$		●	●	●	●
m_6 : $< \mapsto \text{true}$		★	★	★	★
m_7 : $< \mapsto \text{false}$		●	●	●	●
m_8 : $< \mapsto !=$		●	●	●	●
m_9 : $< \mapsto ==$		●	●	●	●
m_{10} : $< \mapsto <=$		●	●	●	●
m_{11} : $< \mapsto >$		●	●	●	●
m_{12} : $< \mapsto >=$		●	●	●	●
m_{13} : $< \mapsto \text{true}$		●	●	●	●
m_{14} : $< \mapsto \text{false}$		●	●	●	●



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