

CSE 403

Software Engineering

Mutation-based Testing

This week: more on test adequacy

- Mutation-based testing
- In-class exercise: hands-on mutation 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!

Mutation-based testing: the basics

Mutation testing

```
public class DrumborgKarting extends Game {
    /**
     * () add state variable here
     */
    private int condition;
    /**
     * () add initialization of state variable here
     */
    public void doonborghKarting() {
        condition = 0;
    }
    /**
     * () add lattice initialization here
     */
    public void initializeLattice() {
        ((DrumborgKarting?) getLattice()).condition = 0;
    }
    /**
     * () add drawing order definition here
     * something will mean the cell will not be drawn
     */
    public Color getColor() {
        return condition;
    }
    /**
     * () add state variable copy code here
     */
    public void copyState() {
        DrumborgKarting state = (DrumborgKarting)
            condition = condition;
    }
    /**
     * () add traverse function code here
     * describing the next state of the cell
     */
    public void traverseCell() {
        if (condition == 0)
            condition = condition + 1;
        else {
            state = state + 1;
            for (int i = 0; i < state.length; i++) {
                ((DrumborgKarting?) getLattice()).condition = i;
                break;
            }
        }
    }
}
```



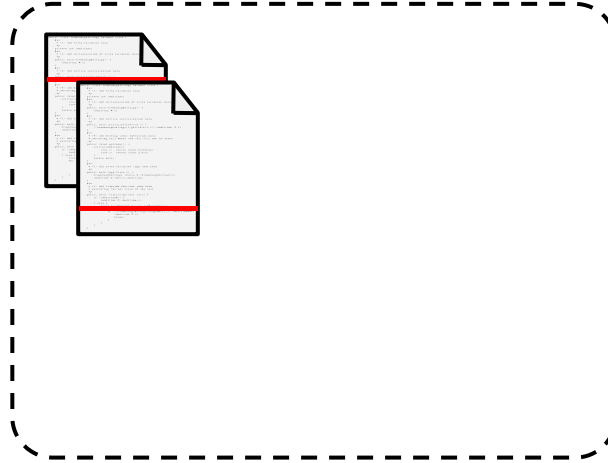
**Mutation
testing**

Program

Mutation operators

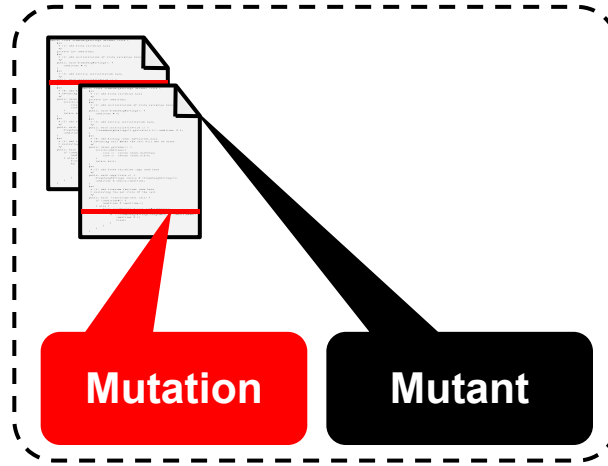
```

1 // set some things going, almost done
2
3 // add state variable here
4 //
5 private int condition;
6
7 // on initialization of state variable here
8 //
9 public void SomeOtherStarting() {
10     condition = 0;
11 }
12
13 // add lattice initialization here
14 //
15 public void InitializeLattice() { }
16
17
18 //
19 // on change, inner definitions here
20 //
21 // setting all these the cell will not be done.
22 //
23 // state generator (i)
24 //
25 void GenState() {
26     state = GenStateColor(black);
27     saw = GenStateColor(black);
28 }
29 //
30 //
31 //
32 //
33 //
34 //
35 //
36 //
37 //
38 //
39 //
40 //
41 //
42 //
43 //
44 //
45 //
46 //
47 //
48 //
49 //
50 //
51 //
52 //
53 //
54 //
55 //
56 //
57 //
58 //
59 //
60 //
61 //
62 //
63 //
64 //
65 //
66 //
67 //
68 //
69 //
70 //
71 //
72 //
73 //
74 //
75 //
76 //
77 //
78 //
79 //
80 //
81 //
82 //
83 //
84 //
85 //
86 //
87 //
88 //
89 //
90 //
91 //
92 //
93 //
94 //
95 //
96 //
97 //
98 //
99 //
100 //
101 //
102 //
103 //
104 //
105 //
106 //
107 //
108 //
109 //
110 //
111 //
112 //
113 //
114 //
115 //
116 //
117 //
118 //
119 //
120 //
121 //
122 //
123 //
124 //
125 //
126 //
127 //
128 //
129 //
130 //
131 //
132 //
133 //
134 //
135 //
136 //
137 //
138 //
139 //
140 //
141 //
142 //
143 //
144 //
145 //
146 //
147 //
148 //
149 //
150 //
151 //
152 //
153 //
154 //
155 //
156 //
157 //
158 //
159 //
160 //
161 //
162 //
163 //
164 //
165 //
166 //
167 //
168 //
169 //
170 //
171 //
172 //
173 //
174 //
175 //
176 //
177 //
178 //
179 //
180 //
181 //
182 //
183 //
184 //
185 //
186 //
187 //
188 //
189 //
190 //
191 //
192 //
193 //
194 //
195 //
196 //
197 //
198 //
199 //
200 //
201 //
202 //
203 //
204 //
205 //
206 //
207 //
208 //
209 //
210 //
211 //
212 //
213 //
214 //
215 //
216 //
217 //
218 //
219 //
220 //
221 //
222 //
223 //
224 //
225 //
226 //
227 //
228 //
229 //
230 //
231 //
232 //
233 //
234 //
235 //
236 //
237 //
238 //
239 //
240 //
241 //
242 //
243 //
244 //
245 //
246 //
247 //
248 //
249 //
250 //
251 //
252 //
253 //
254 //
255 //
256 //
257 //
258 //
259 //
260 //
261 //
262 //
263 //
264 //
265 //
266 //
267 //
268 //
269 //
270 //
271 //
272 //
273 //
274 //
275 //
276 //
277 //
278 //
279 //
280 //
281 //
282 //
283 //
284 //
285 //
286 //
287 //
288 //
289 //
290 //
291 //
292 //
293 //
294 //
295 //
296 //
297 //
298 //
299 //
300 //
301 //
302 //
303 //
304 //
305 //
306 //
307 //
308 //
309 //
310 //
311 //
312 //
313 //
314 //
315 //
316 //
317 //
318 //
319 //
320 //
321 //
322 //
323 //
324 //
325 //
326 //
327 //
328 //
329 //
330 //
331 //
332 //
333 //
334 //
335 //
336 //
337 //
338 //
339 //
340 //
341 //
342 //
343 //
344 //
345 //
346 //
347 //
348 //
349 //
350 //
351 //
352 //
353 //
354 //
355 //
356 //
357 //
358 //
359 //
360 //
361 //
362 //
363 //
364 //
365 //
366 //
367 //
368 //
369 //
370 //
371 //
372 //
373 //
374 //
375 //
376 //
377 //
378 //
379 //
380 //
381 //
382 //
383 //
384 //
385 //
386 //
387 //
388 //
389 //
390 //
391 //
392 //
393 //
394 //
395 //
396 //
397 //
398 //
399 //
400 //
401 //
402 //
403 //
404 //
405 //
406 //
407 //
408 //
409 //
410 //
411 //
412 //
413 //
414 //
415 //
416 //
417 //
418 //
419 //
420 //
421 //
422 //
423 //
424 //
425 //
426 //
427 //
428 //
429 //
430 //
431 //
432 //
433 //
434 //
435 //
436 //
437 //
438 //
439 //
440 //
441 //
442 //
443 //
444 //
445 //
446 //
447 //
448 //
449 //
450 //
451 //
452 //
453 //
454 //
455 //
456 //
457 //
458 //
459 //
460 //
461 //
462 //
463 //
464 //
465 //
466 //
467 //
468 //
469 //
470 //
471 //
472 //
473 //
474 //
475 //
476 //
477 //
478 //
479 //
480 //
481 //
482 //
483 //
484 //
485 //
486 //
487 //
488 //
489 //
490 //
491 //
492 //
493 //
494 //
495 //
496 //
497 //
498 //
499 //
500 //
501 //
502 //
503 //
504 //
505 //
506 //
507 //
508 //
509 //
510 //
511 //
512 //
513 //
514 //
515 //
516 //
517 //
518 //
519 //
520 //
521 //
522 //
523 //
524 //
525 //
526 //
527 //
528 //
529 //
530 //
531 //
532 //
533 //
534 //
535 //
536 //
537 //
538 //
539 //
540 //
541 //
542 //
543 //
544 //
545 //
546 //
547 //
548 //
549 //
550 //
551 //
552 //
553 //
554 //
555 //
556 //
557 //
558 //
559 //
560 //
561 //
562 //
563 //
564 //
565 //
566 //
567 //
568 //
569 //
570 //
571 //
572 //
573 //
574 //
575 //
576 //
577 //
578 //
579 //
580 //
581 //
582 //
583 //
584 //
585 //
586 //
587 //
588 //
589 //
590 //
591 //
592 //
593 //
594 //
595 //
596 //
597 //
598 //
599 //
600 //
601 //
602 //
603 //
604 //
605 //
606 //
607 //
608 //
609 //
610 //
611 //
612 //
613 //
614 //
615 //
616 //
617 //
618 //
619 //
620 //
621 //
622 //
623 //
624 //
625 //
626 //
627 //
628 //
629 //
630 //
631 //
632 //
633 //
634 //
635 //
636 //
637 //
638 //
639 //
640 //
641 //
642 //
643 //
644 //
645 //
646 //
647 //
648 //
649 //
650 //
651 //
652 //
653 //
654 //
655 //
656 //
657 //
658 //
659 //
660 //
661 //
662 //
663 //
664 //
665 //
666 //
667 //
668 //
669 //
670 //
671 //
672 //
673 //
674 //
675 //
676 //
677 //
678 //
679 //
680 //
681 //
682 //
683 //
684 //
685 //
686 //
687 //
688 //
689 //
690 //
691 //
692 //
693 //
694 //
695 //
696 //
697 //
698 //
699 //
700 //
701 //
702 //
703 //
704 //
705 //
706 //
707 //
708 //
709 //
710 //
711 //
712 //
713 //
714 //
715 //
716 //
717 //
718 //
719 //
720 //
721 //
722 //
723 //
724 //
725 //
726 //
727 //
728 //
729 //
730 //
731 //
732 //
733 //
734 //
735 //
736 //
737 //
738 //
739 //
740 //
741 //
742 //
743 //
744 //
745 //
746 //
747 //
748 //
749 //
750 //
751 //
752 //
753 //
754 //
755 //
756 //
757 //
758 //
759 //
760 //
761 //
762 //
763 //
764 //
765 //
766 //
767 //
768 //
769 //
770 //
771 //
772 //
773 //
774 //
775 //
776 //
777 //
778 //
779 //
780 //
781 //
782 //
783 //
784 //
785 //
786 //
787 //
788 //
789 //
790 //
791 //
792 //
793 //
794 //
795 //
796 //
797 //
798 //
799 //
800 //
801 //
802 //
803 //
804 //
805 //
806 //
807 //
808 //
809 //
810 //
811 //
812 //
813 //
814 //
815 //
816 //
817 //
818 //
819 //
820 //
821 //
822 //
823 //
824 //
825
```



Mutants

Mutation operators

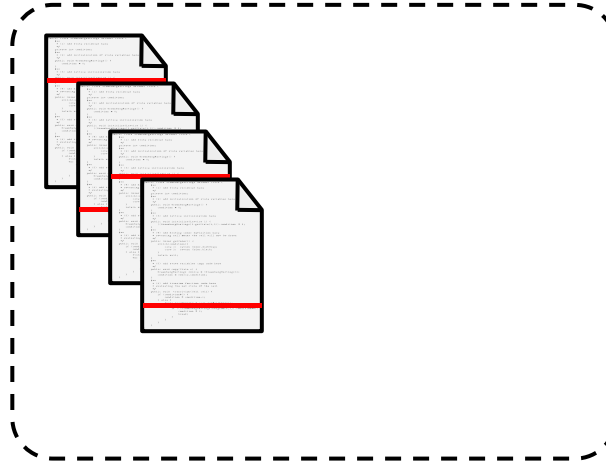
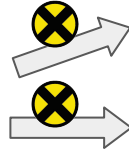
[illegible]

$Lhs < rhs \Rightarrow Lhs \leq rhs$
 $Lhs < rhs \Rightarrow Lhs \neq rhs$
 $stmt \Rightarrow no-op$

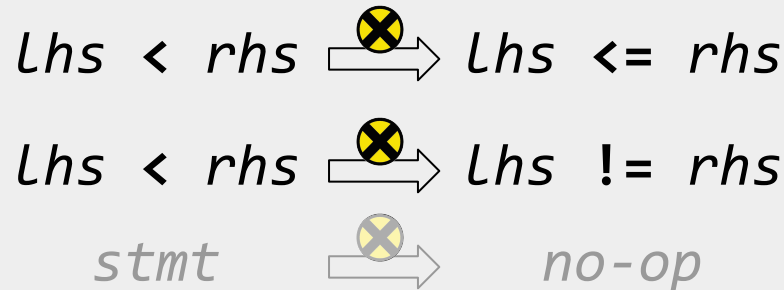
Mutation operators

```

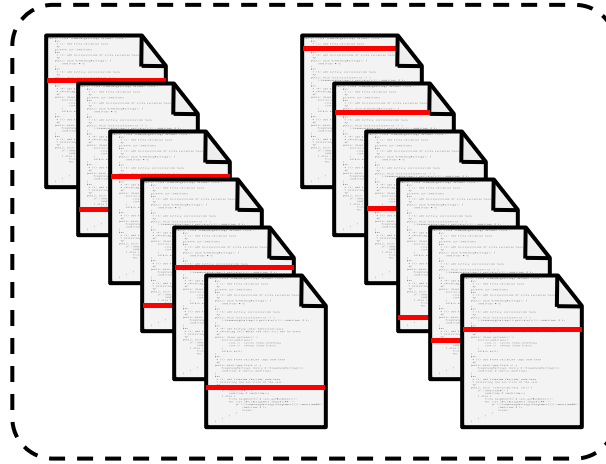
1 // set some things going, almost done
2
3 // add state variable here
4 //
5 private int condition;
6
7 // on initialization of state variable here
8 //
9 public void SomeOtherStarting() {
10     condition = 0;
11 }
12
13 // add lattice initialization here
14 //
15 public void InitializeLattice() { }
16
17
18 //
19 // on change, inner definitions here
20 //
21 // setting all these the cell will not be done.
22 //
23 // state generator (i)
24 //
25 void GenState() {
26     state = GenStateColor(black);
27     saw = GenStateColor(black);
28 }
29 //
30 //
31 //
32 //
33 //
34 //
35 //
36 //
37 //
38 //
39 //
40 //
41 //
42 //
43 //
44 //
45 //
46 //
47 //
48 //
49 //
50 //
51 //
52 //
53 //
54 //
55 //
56 //
57 //
58 //
59 //
60 //
61 //
62 //
63 //
64 //
65 //
66 //
67 //
68 //
69 //
70 //
71 //
72 //
73 //
74 //
75 //
76 //
77 //
78 //
79 //
80 //
81 //
82 //
83 //
84 //
85 //
86 //
87 //
88 //
89 //
90 //
91 //
92 //
93 //
94 //
95 //
96 //
97 //
98 //
99 //
100 //
101 //
102 //
103 //
104 //
105 //
106 //
107 //
108 //
109 //
110 //
111 //
112 //
113 //
114 //
115 //
116 //
117 //
118 //
119 //
120 //
121 //
122 //
123 //
124 //
125 //
126 //
127 //
128 //
129 //
130 //
131 //
132 //
133 //
134 //
135 //
136 //
137 //
138 //
139 //
140 //
141 //
142 //
143 //
144 //
145 //
146 //
147 //
148 //
149 //
150 //
151 //
152 //
153 //
154 //
155 //
156 //
157 //
158 //
159 //
160 //
161 //
162 //
163 //
164 //
165 //
166 //
167 //
168 //
169 //
170 //
171 //
172 //
173 //
174 //
175 //
176 //
177 //
178 //
179 //
180 //
181 //
182 //
183 //
184 //
185 //
186 //
187 //
188 //
189 //
190 //
191 //
192 //
193 //
194 //
195 //
196 //
197 //
198 //
199 //
200 //
201 //
202 //
203 //
204 //
205 //
206 //
207 //
208 //
209 //
210 //
211 //
212 //
213 //
214 //
215 //
216 //
217 //
218 //
219 //
220 //
221 //
222 //
223 //
224 //
225 //
226 //
227 //
228 //
229 //
230 //
231 //
232 //
233 //
234 //
235 //
236 //
237 //
238 //
239 //
240 //
241 //
242 //
243 //
244 //
245 //
246 //
247 //
248 //
249 //
250 //
251 //
252 //
253 //
254 //
255 //
256 //
257 //
258 //
259 //
260 //
261 //
262 //
263 //
264 //
265 //
266 //
267 //
268 //
269 //
270 //
271 //
272 //
273 //
274 //
275 //
276 //
277 //
278 //
279 //
280 //
281 //
282 //
283 //
284 //
285 //
286 //
287 //
288 //
289 //
290 //
291 //
292 //
293 //
294 //
295 //
296 //
297 //
298 //
299 //
300 //
301 //
302 //
303 //
304 //
305 //
306 //
307 //
308 //
309 //
310 //
311 //
312 //
313 //
314 //
315 //
316 //
317 //
318 //
319 //
320 //
321 //
322 //
323 //
324 //
325 //
326 //
327 //
328 //
329 //
330 //
331 //
332 //
333 //
334 //
335 //
336 //
337 //
338 //
339 //
340 //
341 //
342 //
343 //
344 //
345 //
346 //
347 //
348 //
349 //
350 //
351 //
352 //
353 //
354 //
355 //
356 //
357 //
358 //
359 //
360 //
361 //
362 //
363 //
364 //
365 //
366 //
367 //
368 //
369 //
370 //
371 //
372 //
373 //
374 //
375 //
376 //
377 //
378 //
379 //
380 //
381 //
382 //
383 //
384 //
385 //
386 //
387 //
388 //
389 //
390 //
391 //
392 //
393 //
394 //
395 //
396 //
397 //
398 //
399 //
400 //
401 //
402 //
403 //
404 //
405 //
406 //
407 //
408 //
409 //
410 //
411 //
412 //
413 //
414 //
415 //
416 //
417 //
418 //
419 //
420 //
421 //
422 //
423 //
424 //
425 //
426 //
427 //
428 //
429 //
430 //
431 //
432 //
433 //
434 //
435 //
436 //
437 //
438 //
439 //
440 //
441 //
442 //
443 //
444 //
445 //
446 //
447 //
448 //
449 //
450 //
451 //
452 //
453 //
454 //
455 //
456 //
457 //
458 //
459 //
460 //
461 //
462 //
463 //
464 //
465 //
466 //
467 //
468 //
469 //
470 //
471 //
472 //
473 //
474 //
475 //
476 //
477 //
478 //
479 //
480 //
481 //
482 //
483 //
484 //
485 //
486 //
487 //
488 //
489 //
490 //
491 //
492 //
493 //
494 //
495 //
496 //
497 //
498 //
499 //
500 //
501 //
502 //
503 //
504 //
505 //
506 //
507 //
508 //
509 //
510 //
511 //
512 //
513 //
514 //
515 //
516 //
517 //
518 //
519 //
520 //
521 //
522 //
523 //
524 //
525 //
526 //
527 //
528 //
529 //
530 //
531 //
532 //
533 //
534 //
535 //
536 //
537 //
538 //
539 //
540 //
541 //
542 //
543 //
544 //
545 //
546 //
547 //
548 //
549 //
550 //
551 //
552 //
553 //
554 //
555 //
556 //
557 //
558 //
559 //
560 //
561 //
562 //
563 //
564 //
565 //
566 //
567 //
568 //
569 //
570 //
571 //
572 //
573 //
574 //
575 //
576 //
577 //
578 //
579 //
580 //
581 //
582 //
583 //
584 //
585 //
586 //
587 //
588 //
589 //
590 //
591 //
592 //
593 //
594 //
595 //
596 //
597 //
598 //
599 //
600 //
601 //
602 //
603 //
604 //
605 //
606 //
607 //
608 //
609 //
610 //
611 //
612 //
613 //
614 //
615 //
616 //
617 //
618 //
619 //
620 //
621 //
622 //
623 //
624 //
625 //
626 //
627 //
628 //
629 //
630 //
631 //
632 //
633 //
634 //
635 //
636 //
637 //
638 //
639 //
640 //
641 //
642 //
643 //
644 //
645 //
646 //
647 //
648 //
649 //
650 //
651 //
652 //
653 //
654 //
655 //
656 //
657 //
658 //
659 //
660 //
661 //
662 //
663 //
664 //
665 //
666 //
667 //
668 //
669 //
670 //
671 //
672 //
673 //
674 //
675 //
676 //
677 //
678 //
679 //
680 //
681 //
682 //
683 //
684 //
685 //
686 //
687 //
688 //
689 //
690 //
691 //
692 //
693 //
694 //
695 //
696 //
697 //
698 //
699 //
700 //
701 //
702 //
703 //
704 //
705 //
706 //
707 //
708 //
709 //
710 //
711 //
712 //
713 //
714 //
715 //
716 //
717 //
718 //
719 //
720 //
721 //
722 //
723 //
724 //
725 //
726 //
727 //
728 //
729 //
730 //
731 //
732 //
733 //
734 //
735 //
736 //
737 //
738 //
739 //
740 //
741 //
742 //
743 //
744 //
745 //
746 //
747 //
748 //
749 //
750 //
751 //
752 //
753 //
754 //
755 //
756 //
757 //
758 //
759 //
760 //
761 //
762 //
763 //
764 //
765 //
766 //
767 //
768 //
769 //
770 //
771 //
772 //
773 //
774 //
775 //
776 //
777 //
778 //
779 //
780 //
781 //
782 //
783 //
784 //
785 //
786 //
787 //
788 //
789 //
790 //
791 //
792 //
793 //
794 //
795 //
796 //
797 //
798 //
799 //
800 //
801 //
802 //
803 //
804 //
805 //
806 //
807 //
808 //
809 //
810 //
811 //
812 //
813 //
814 //
815 //
816 //
817 //
818 //
819 //
820 //
821 //
822 //
823 //
824 //
825
```



Mutants



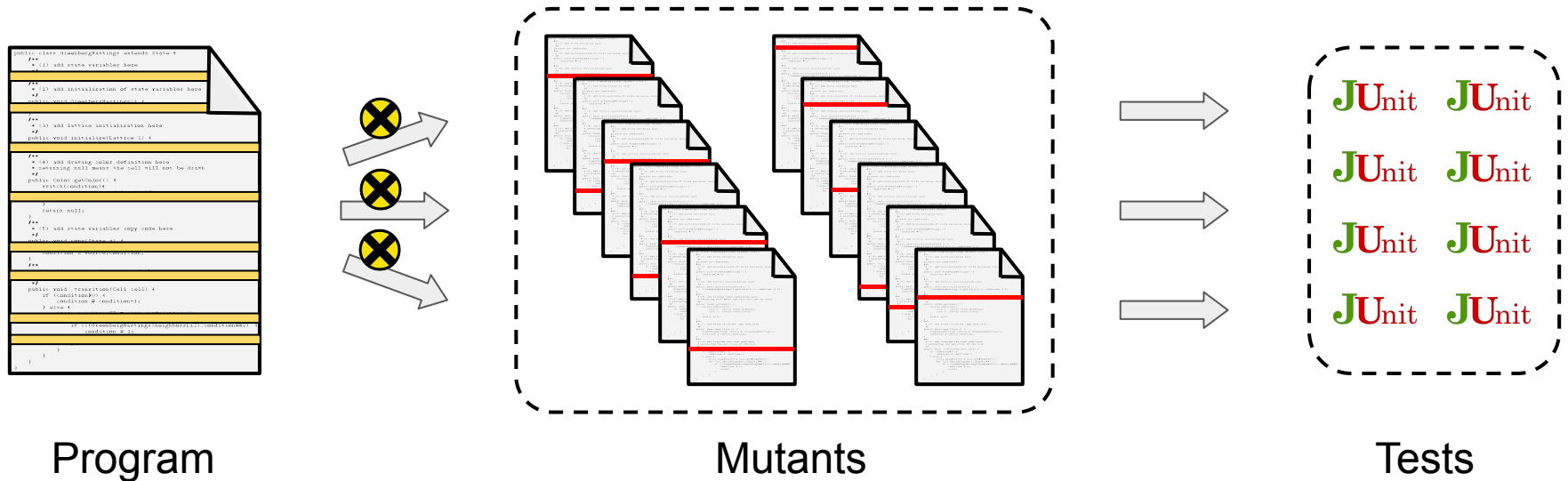
Mutation operators

[illegible]

$lhs < rhs \xrightarrow{\text{X}} lhs \leq rhs$
 $lhs < rhs \xrightarrow{\text{X}} lhs \neq rhs$
 $stmt \xrightarrow{\text{X}} no-op$

Mutation operators

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
(i.e., passes on the original program but fails on the mutant)

<https://forms.gle/ADTR1nDzewgqRXaD9>

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>	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?

<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: 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

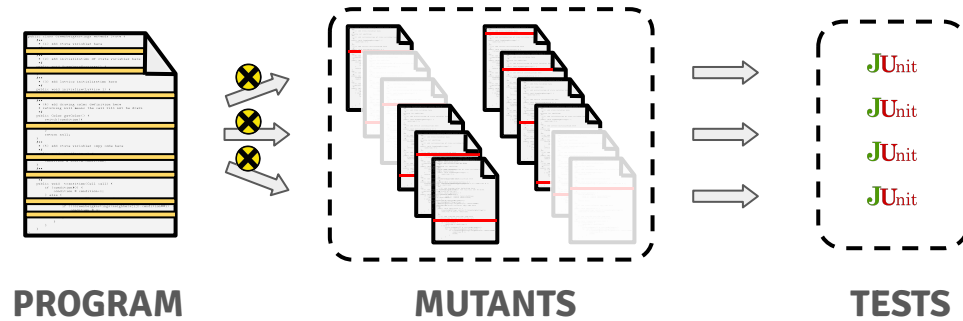
Mutation testing: challenges

- 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)

<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

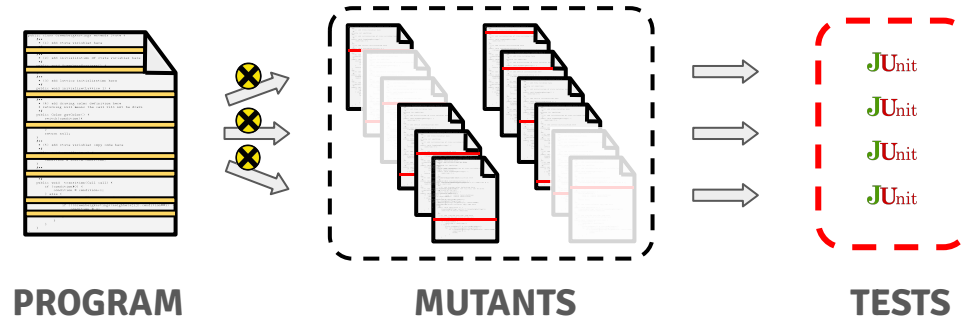
Mutation Testing vs. Mutation Analysis

Mutation Testing



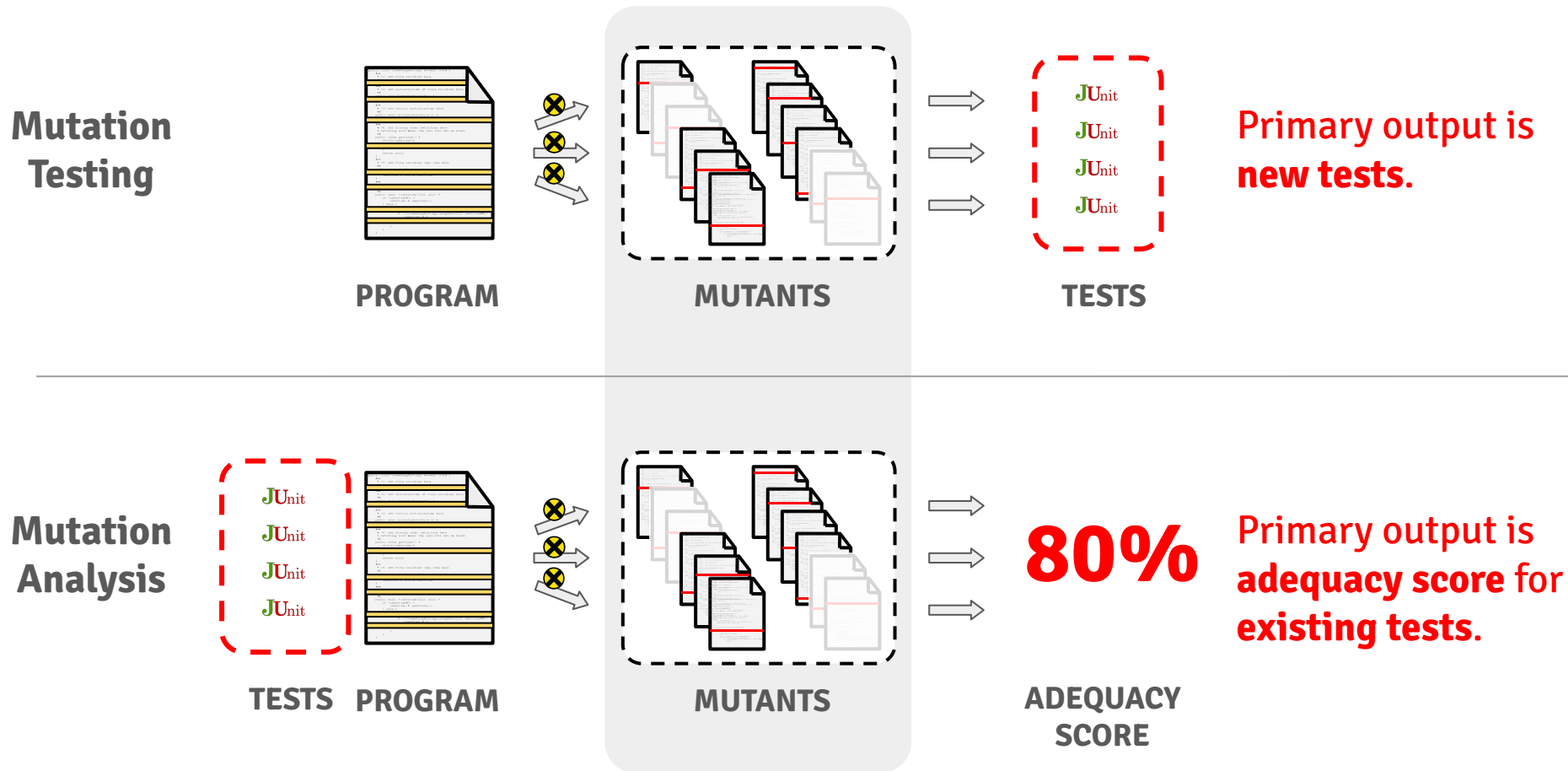
Mutation Testing vs. Mutation Analysis

Mutation Testing

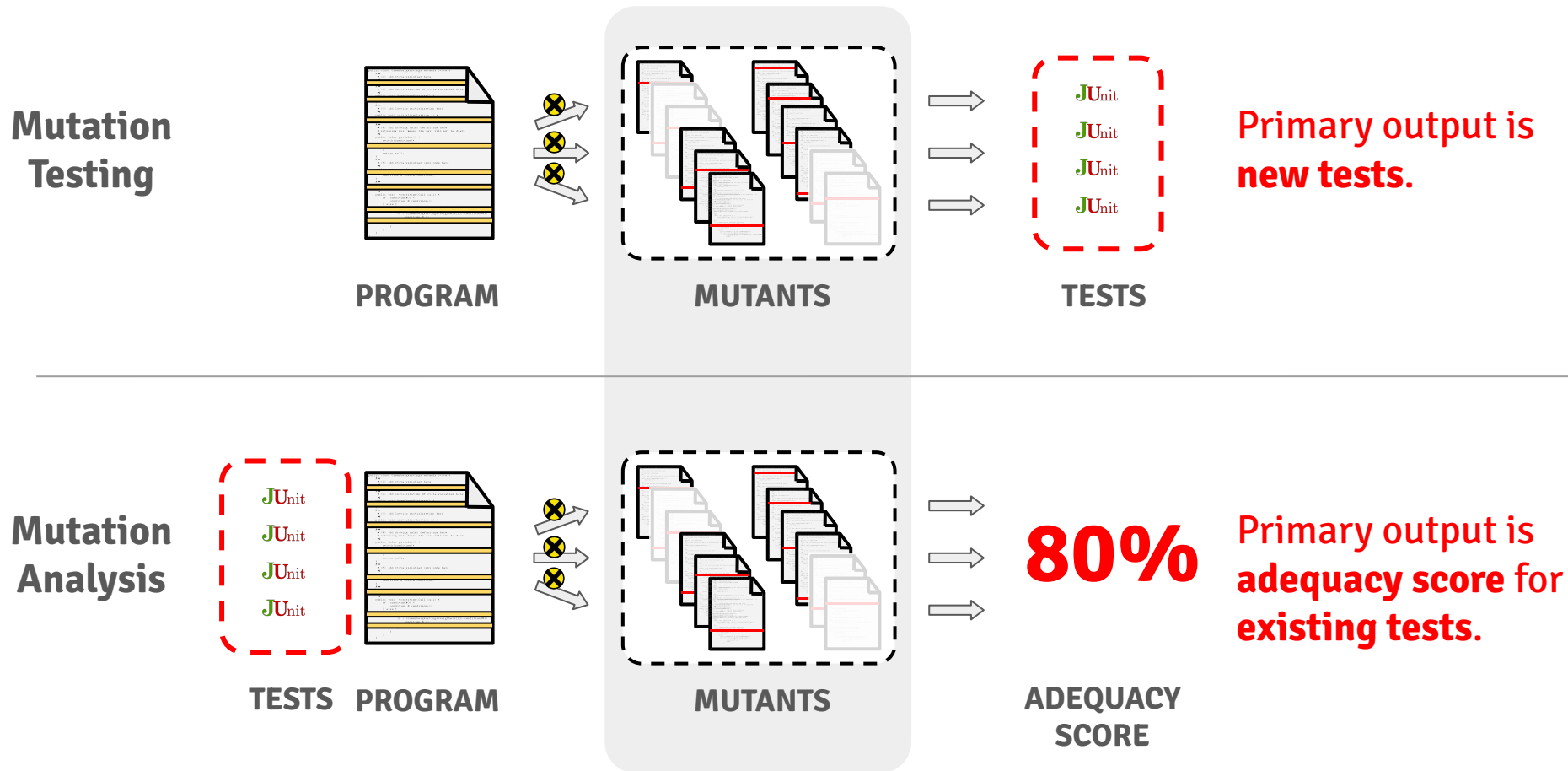


Primary output is
new tests.

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** $\Rightarrow$ **tests**
- **Equivalent** mutants are **bad** $\Rightarrow$ **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** $\Rightarrow$ **tests**
- **Equivalent** mutants are **bad** $\Rightarrow$ **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!**

Mutation-based testing: mutant subsumption

Mutant subsumption

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

Mutant detected (assertion)

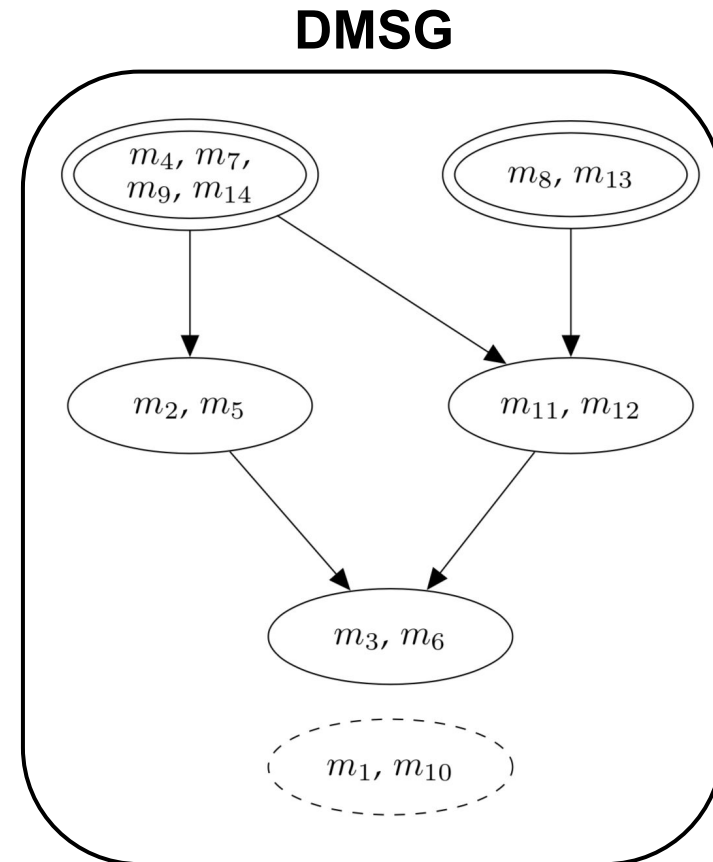
Mutant detected (exception)

Mutant not detected

Prioritizing Mutants to Guide Mutation Testing ([Reading](#))

DMSG: Dynamic Mutant Subsumption Graph

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



Prioritizing Mutants to Guide Mutation Testing ([Reading](#))