CS 4530: Fundamentals of Software Engineering Module 3.2: Code Coverage & Mutation Testing

Adeel Bhutta, Rob Simmons, and Mitch Wand

Khoury College of Computer Sciences

Learning Goals for this Lesson

At the end of this lesson, you should be able to

- Explain what code coverage is, and how different measures differ, including statements, branches, functions, and lines

- Explain the benefits of mutation testing

When have I written enough tests?

Why do we test?

- Test Driven Development
  - If we start with tests, passing the tests lets us know we're done meeting the conditions of satisfaction
- Acceptance Testing
  - Does the customer agree we've met the conditions of satisfaction?
- Regression Testing
  - Did something change since some previous version?
  - Prevent bugs from (re-)entering during maintenance.
  - "Good" test suite detects bugs that (inevitably) get added as software is developed over time: tests are for *the future*

When Have I Written Enough Tests?

- When I've tested the valid inputs

- **When I've tested all the code**

- **When the tests will catch bugs**

Code Coverage

- The industry standard answer for "have I written enough tests"

- Measures "how much of your code" is exercised by your tests

- If none of your test even *execute* a piece of code, it's not being tested!

Code Coverage: Line or Statement Coverage

- *Line* and *Statement* coverage
  are a coarse measurement.

- Testing x = 0 exercises lines 2
  and 3

- Testing x = 10 exercises lines 2,
  5, 6, and 7.

```
1| export function f(x: number) {
2|    if (x === 0) {
3|      return 3;
4|    }
5|    const y = x > 4 ? 2 : 3;
6|    const z = x % 2 === 0 ? 1 : 2;
7|    return x / (y - z);
8| }
```

Code Coverage: Branch Coverage

- *Branch* coverage: widely used in industry.

- Testing with x > 4 and x <= 4 necessary to handle both branches on line 5.

- Testing with odd and even numbers necessary to handle both branches on line 6.

- The values -2, 0, 1, and 10 get full branch coverage.

```
1|  export function f(x: number) {
2|     if (x === 0) {
3|        return 3;
4|     }
5|     const y = x > 4 ? 2 : 3;
6|     const z = x % 2 === 0 ? 1 : 2;
7|     return x / (y - z);
8|  }
```

Code Coverage: Path Coverage

- The values -2, 0, 1, and 10 get full branch coverage…

- …but 5 causes line 7 to divide by zero!
  - In JavaScript/TypeScript, this doesn't cause an exception, there's a number called "Infinity" (which is *different* from "NaN," for "not a number")

- *Path* coverage covers all combinations of branches; it is infeasible in practice.

```
1| export function f(x: number) {
2|    if (x === 0) {
3|       return 3;
4|    }
5|    const y = x > 4 ? 2 : 3;
6|    const z = x % 2 === 0 ? 1 : 2;
7|    return x / (y - z);
8| }
```

Code Coverage: Example 1

What tests will get line/branch/path coverage here?

```
/** Returns an array that repeats "hello"
 * @param numHellos - number of "hello"s to return, must be an integer >= 0
 */
function helloNTimes(numHellos: number): string[] {
  const arr: string[] = [];
  for (let i = numHellos; i !== 0; i--) { arr.push("hello"); }
  return arr;
}
```

Code Coverage: Example 2

What tests will get line/branch/path coverage here?

```
const zHelloInput = z.int().gte(0);

/** Returns an array that repeats "hello"
 * @param numHellos - number of times to say "hello"
 * @throws if the input is not an integer >= 0
 */
function helloNTimes(numHellos: unknown): string[] {
  const parseResult = zHelloInput.safeParse(numHellos);
  if (!parseResult.success) throw new Error("Invalid input");
  const arr: string[] = [];
  for (let i = parseResult.data; i !== 0; i--) { arr.push("hello"); }
  return arr;
}
```
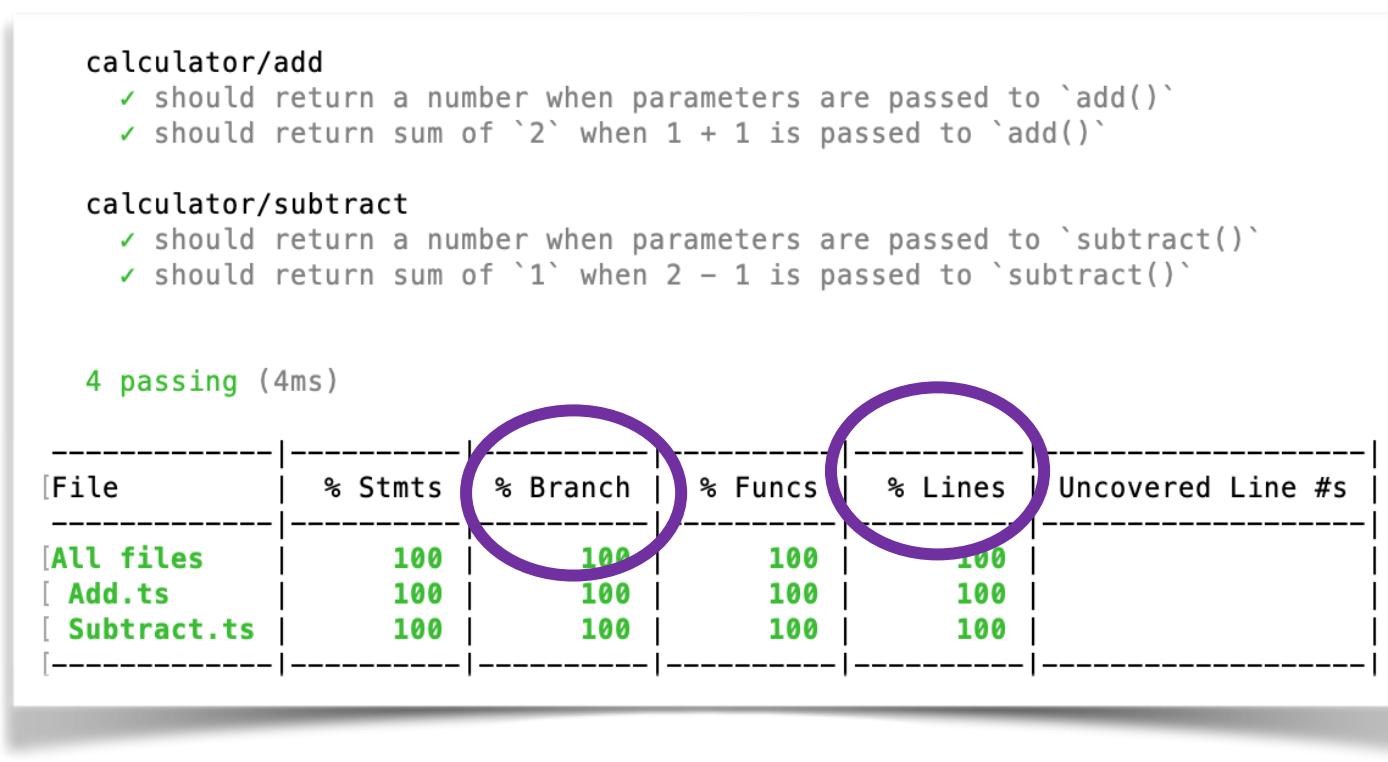
Code Coverage: More Examples

- Some of our intuition about "edge cases" can be understood in terms of trying to imagine inputs that will provide code coverage for an unknown function.
  - BUT: code coverage won't tell you to check 0 specifically if there's a branch that checks x >= 0
  - Thinking about the "edge case" of a branch is a good move
- What inputs should we start with if:
  - We're taking numbers that are supposed to represent a 5-digit zip code?
  - We're taking strings that are supposed to represent a 5-digit zip code?
  - We're taking an array of strings that represent people's names?

Code Coverage in Vitest

- Tools like vitest makes it easy to check code coverage

```
calculator/add
  ✓ should return a number when parameters are passed to `add()`
  ✓ should return sum of `2` when 1 + 1 is passed to `add()`

calculator/subtract
  ✓ should return a number when parameters are passed to `subtract()`
  ✓ should return sum of `1` when 2 – 1 is passed to `subtract()`


4 passing (4ms)


--------------|----------|----------|----------|----------|---------------------|
[File          | % Stmts  | % Branch | % Funcs  | % Lines  | Uncovered Line #s   |
--------------|----------|----------|----------|----------|---------------------|
[All files     |      100 |      100 |      100 |      100 |                     |
[ Add.ts       |      100 |      100 |      100 |      100 |                     |
[ Subtract.ts  |      100 |      100 |      100 |      100 |                     |
[--------------|----------|----------|----------|----------|---------------------|
```

Code Coverage: Review

- Total code coverage — by any metric — does not mean no bugs
    - Running code doesn't mean checking that it's doing the right thing! (important to Assemble/Act/<u>Assess</u>)
    - "Coverage is no substitute for thought" — Rob Pike
- Coverage checking can be invaluable at identifying when you *think* you're testing something but you're not, which is a real problem in practice.
    - Test-Driven Development also valuable for this problem: it's important that tests switch from failing to succeeding *when you expect them to*

Adversarial Testing

- It can be helpful to think of testing as a game in which you play against an adversary.

- Your adversary plays by producing multiple versions of code that you agree is buggy, and multiple versions of code you agree is correct.

- Your win if your tests catch all the buggy code, and pass all the correct code.

Adversarial Testing

Original code (correct)

```
// find the first item in the list that is
// greater than or equal to the target.
export default function search(list:number[], target:number) {
    return list.find((item) => item >= target);
}
```

Mutated code (buggy)

```
// find the first item in the list that is
// greater than or equal to the target.
export default function search(list:number[], target:number) {
    return list.find((item) => item > target);
}
```

Adversarial Testing

Stryker is a *mutation tester* for JavaScript — an automated adversary!


```
< >  ☐  ✅ Killed (65)    ☑  👽 Survived (2)

132                 */
133        public overlaps(otherInteractable: InteractableArea): boolean {
134            const toRectPoints = ({ _x, _y, _width, _height }: InteractableArea) => ({ x1: _x
135            const rect1 = toRectPoints(this);
136            const rect2 = toRectPoints(otherInteractable);
137  -         const noOverlap = rect1.x1 >= rect2.x2●
     +         const noOverlap = rect1.x1 > rect2.x2
138                || rect2.x1 >= rect1.x2 || rect1.y1 >= rect2.y2 || rect2.y1 >= rect1.y2;●
139            return !noOverlap;
140        }
141
```

Remedy: you need to devise tests that distinguish the original code from the mutants

- Devise a test that your original code will pass, but
the mutant will fail.

A tiny example

Imagine that this is the code to be tested

```typescript
// find the first item in the list that is
// greater than or equal to the target.
export default function search(list:number[], target:number)
{
    return list.find((item) => item >= target);
}
```

and we have written some tests.

Stryker report for this test

[Survived] EqualityOperator
src/for-midterm/adrian.ts:4:32
```
-        return list.find((item) => item >= target);
+        return list.find((item) => item > target);
```
Tests ran:
    search should return the first item in the list that is greater
than or equal to the target

The location of the mutation

The tests that were run on this piece of code

[Survived] ConditionalExpression
src/for-midterm/adrian.ts:4:32
```
-        return list.find((item) => item >= target);
+        return list.find((item) => true);
```
Tests ran:
    search should return the first item in the list that is greater
than or equal to the target

Let's look at the second one:

```
[Survived] ConditionalExpression
src/for-midterm/adrian.ts:4:32
-        return list.find((item) => item >= target);
+        return list.find((item) => true);
```

- How does this mutant behave differently from the original?

- Answer: This mutant always returns the first element of the list

- Remedy: Add a test that searches for something that is NOT the first element in the input?

Here's one test that will cause the mutant to be killed.

```javascript
test("should return the second element of the list", () => {
  expect(search([5, 7, 9], 6)).toBe(7);
});
```

What about the other mutant?

```
[Survived] EqualityOperator
src/for-midterm/adrian.ts:4:32
-        return list.find((item) => item >= target);
+        return list.find((item) => item > target);
```
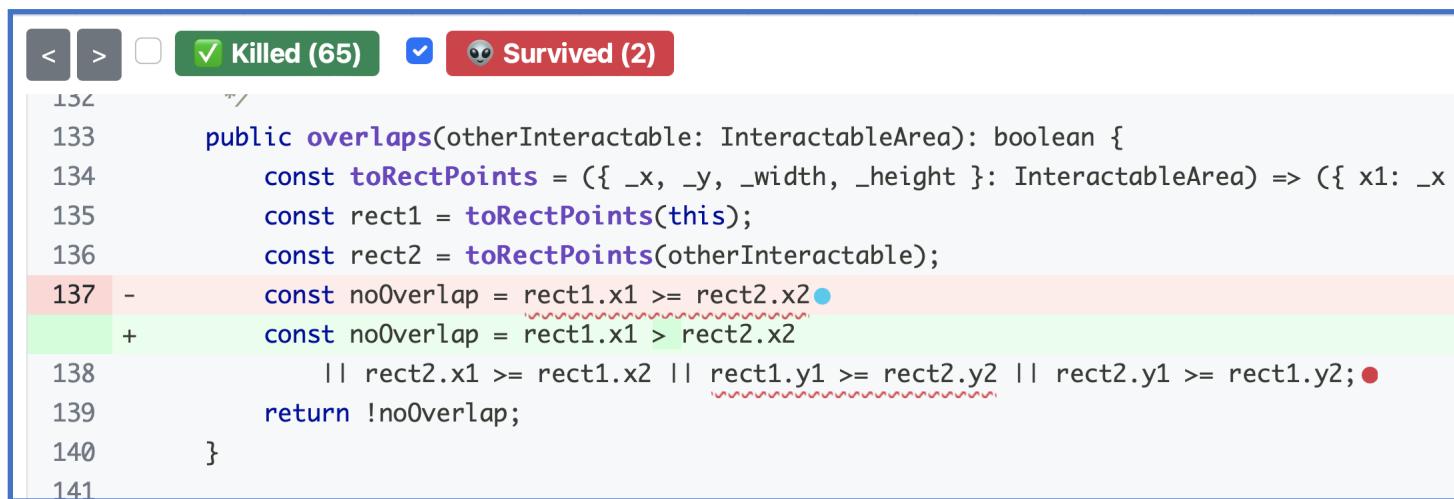
- How does this mutant behave differently from the original?

- Answer: This mutant returns the first element of the list that is larger than the original

- Remedy: Add a test where the input list included an "equal" item before a larger item

Here's one test that will catch that mutant

```
test("try target that is equal to some item in the list", () => {
    expect(search([5, 7, 9], 7)).toBe(7);
});
```

Use Mutation Analysis While Writing Tests

- When you feel "done" writing tests, run a mutation analysis

- Inspect undetected mutants, and try to write tests that will make those mutants fail.


```
        < > ☐  ✅ Killed (65)  ☑  👾 Survived (2)

132           */
133           public overlaps(otherInteractable: InteractableArea): boolean {
134               const toRectPoints = ({ _x, _y, _width, _height }: InteractableArea) => ({ x1: _x
135               const rect1 = toRectPoints(this);
136               const rect2 = toRectPoints(otherInteractable);
137  -           const noOverlap = rect1.x1 >= rect2.x2●
     +           const noOverlap = rect1.x1 > rect2.x2
138                   || rect2.x1 >= rect1.x2 || rect1.y1 >= rect2.y2 || rect2.y1 >= rect1.y2;●
139               return !noOverlap;
140           }
141
```

Detailed mutation report for "overlaps" method - two mutants were not detected!

Adversarial Testing and Overspecification

Stryker is a *mutation tester* for JavaScript — an automated adversary!

Sometimes it loses the game because mutants aren't bugs.

```
62          public static fromMapObject(mapObject: ITiledMapObject, broadca
63              const { name, width, height } = mapObject;
64              if (!width || !height) {●
65  -               throw new Error(`Malformed viewing area ${name}`);●
    +               throw new Error(``);
66              }
67              const rect: BoundingBox = { x: mapObject.x, y: mapObject.y, w
68              return new ConversationArea({ id: name, occupantsByID: [] },
69          }
```

Are mutants a Valid Substitute for Real Faults? Probably yes.

- Do mutants really represent real bugs?

- Researchers have studied the question of whether a test suite that finds more mutants also finds more real faults

- Conclusion: For the 357 real faults studied, yes

- This work has been replicated in many other contexts, including with real faults from student code

Review

It's the end of the lesson, so you should be able to

- Explain what code coverage is, and how different measures differ, including statements, branches, functions, and lines
- Explain the benefits of mutation testing

- When have you written enough tests?
  - When you've tested all valid inputs?
  - When your tests get full code coverage?
  - When your tests can catch any buggy mutant code?
  - The answer depends!