

CS 4530: Fundamentals of Software Engineering

Module 2, Lesson 4

When Have I Written Enough Tests?

Rob Simmons

Khoury College of Computer Sciences

Learning Goals for this Lesson

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

- Explain how TypeScript types and documented preconditions influence what tests you need to write
- Explain what code coverage is, and how different measures differ, including statements, branches, functions, and lines
- Explain the benefits of mutation testing

Testing and preconditions

What input values do I need to test this function on?

```
/**
 * Prints "hello" repeatedly
 *
 * @param numHellos - number of times to apply fn to base,
 * must be an integer >= 0
 */
function iterateNTimes(numHellos: number) {
  for (let i = numHellos; i !== 0; i--) {
    console.log('hello');
  }
}
```

Testing and preconditions

What input values do I need to test this function on?

- Edge cases (definitely 0)
- Probably 1 and some larger number?
But most numbers > 1 are kind of interchangeable.
 - If we want to sound fancy we'll call these “equivalence classes of inputs.”
- What about -3? What about 1.4? What about `null` or `{ lol: 'owned' }`?

```
/**  
 * Prints "hello" repeatedly  
 *  
 * @param numHellos - number of times to apply fn to base,  
 * must be an integer  $\geq 0$   
 */  
function iterateNTimes(numHellos: number)
```

Testing and TypeScript

- TypeScript types are, at the end of the day, no better than precondition comments.

```
iterateNTimes({ lol: 'owned ' } as unknown as number)
```

- They do at least make it less likely you'll screw up *accidentally...*
- It makes sense *sometimes* to treat your precondition comments as not-needing-to-be tested
- It makes sense *often* to treat your TypeScript types as not-needing-to-be-tested
- Extra defensive checks have their own costs!

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 definitely not being tested!

Code Coverage

- *Line* and *Statement* coverage: coarsest measure.
- Testing $x = 0$ exercises lines 1 and 2
- Testing $x = 10$ exercises lines 1, 4, 5, and 6.

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

Code Coverage

- *Branch* coverage: most widely used in industry.
- Testing with $x > 4$ and $x \leq 4$ necessary to handle both branches on line 4.
- Testing with odd and even numbers necessary to handle both branches on line 5.
- The values -2, 0, 1, and 10 get full branch coverage.

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


Code Coverage

- The values -2, 0, 1, and 10 get full branch coverage...
- ...but 5 causes line 6 to divide by zero!
 - In JavaScript/TypeScript, this doesn't cause an exception, there's a number called "NaN" for "not a number"
- *Path* coverage covers all combinations of branches, but is infeasible in practice.

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

Code Coverage

- Total code coverage — by any metric — does not mean no bugs
 - Running code doesn't mean checking that it's doing the right thing!
- 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 is 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.
- You 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);  
}
```

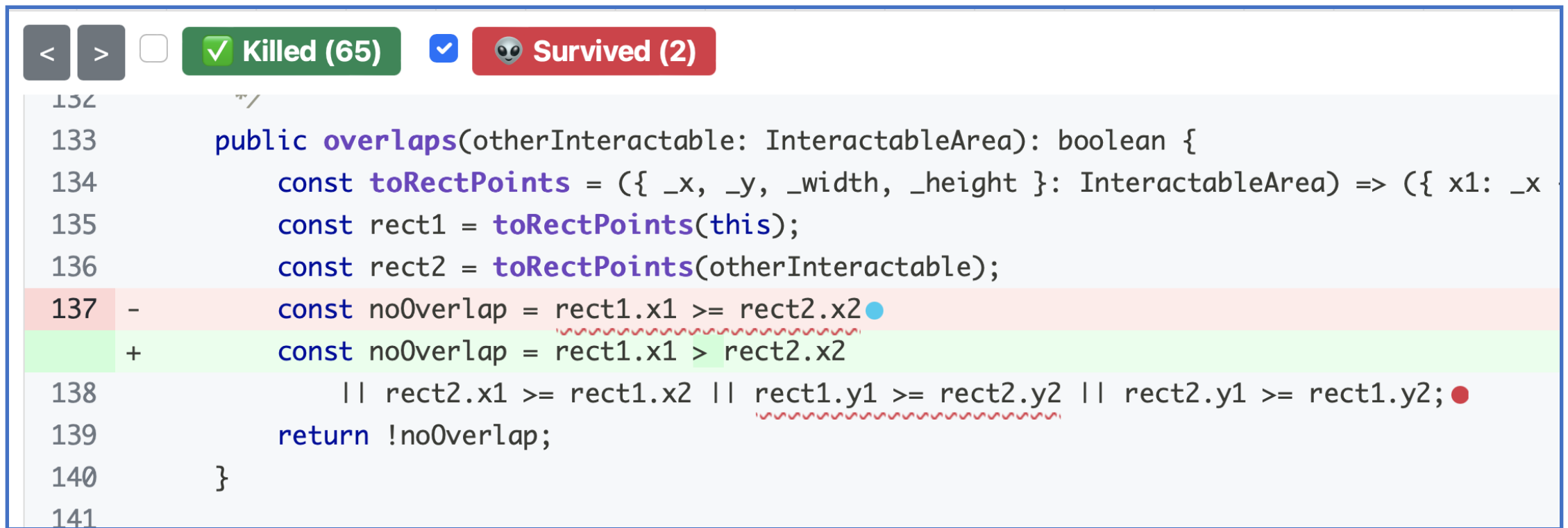
Mutated code (buggy)

```
// 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!



The screenshot shows the Stryker.js interface for a JavaScript file. At the top, there are navigation buttons (< >) and a summary bar showing 'Killed (65)' in green and 'Survived (2)' in red. The code is displayed with line numbers 132 to 141. A mutation on line 137 is highlighted, showing the original code with a blue dot and the mutated code with a green background. The original code is `const noOverlap = rect1.x1 >= rect2.x2` and the mutated code is `const noOverlap = rect1.x1 > rect2.x2`. The rest of the code is as follows:

```
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
138  +   const noOverlap = rect1.x1 > rect2.x2
139      || rect2.x1 >= rect1.x2 || rect1.y1 >= rect2.y2 || rect2.y1 >= rect1.y2;
140      return !noOverlap;
141  }
```

Adversarial Testing

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, broadcast
63         const { name, width, height } = mapObject;
64         if (!width || !height) {●
65 -         throw new Error(`Malformed viewing area ${name}`);●
66 +         throw new Error(``);
67     }
68     const rect: BoundingBox = { x: mapObject.x, y: mapObject.y, w
69     return new ConversationArea({ id: name, occupantsByID: [] },
70 }
```

Review

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

- Explain how TypeScript types and documented preconditions influence what tests you need to write
- Explain what code coverage is, and how different measures differ, including statements, branches, functions, and lines
- Explain the benefits of mutation testing