CS 4530: Fundamentals of Software Engineering

Module 11: Test Adequacy

Jonathan Bell, Adeel Bhutta, Mitch Wand

Khoury College of Computer Sciences

Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
  - Give different reasons why you might want to test
  - List the properties of a good test
  - Use equivalence classes to design a TDD test suite
  - Explain 3 measures of code coverage
  - Use mutation testing to judge the completeness of a test suite

Why do we test?

- Test Driven Development
  - Does the SUT satisfy its specification?
  - “Good” test suite exercises the *entire* specification
- Regression Testing
  - Did something change since some previous version?
  - Prevent bugs from (re-)entering during maintenance.
  - “Good” test suite detects bugs that we introduce in code
- Acceptance Testing
  - Does the SUT satisfy the customer
  - “Good” test suite answers: Are we building the right system ?

What makes for a good test (suite)?

- Desirable properties of test suites:
  - Find bugs
  - Run automatically
  - Are relatively cheap to run
- Desirable properties of individual tests:
  - Understandable and debuggable
  - No false alarms (not “flaky”)

Related Terminology:
“test smells”

Good Tests have Strong Oracles

- Test oracle defines criteria for when test should fail
- What kind of oracle should we choose?
  - Function returns the exact “right” answer
  - Function returns an acceptable answer
  - Returns the same value as last time
  - Function returns without crashing
  - Function crashes (as expected)
  - Function has the right effects on its environment
    - And no others

A good test is self-contained

- Contain all information necessary to set up, execute, and tear down environment
- Leaves no trace of its execution
- So it doesn't matter in what order your tests run.

Jargon word: "hermetic"

```
// NOT HERMETIC
// assumes starting ID of 4, leaves an extra Avery in the application
describe('Create student', () => {
    it('should return an ID', async () => {
        const createdStudent = await client.addStudent('Avery');
        expect(createdStudent.studentID).toBeGreaterThan(4);
    });
})
```

"Software Engineering at Google: Lessons Learned from Programming Over Time," Wright, Winters and Manshreck, 2020 (O'Reilly)

Good Tests Aren’t *Brittle*

- Brittle tests make invalid assumptions about the specification
- Specifications often leave room for undefined behaviors: details that are subject to change
- Brittle tests will fail unexpectedly if that undefined behavior changes

```
// BRITTLE!
// Assumes the application shows this specific error message
// Does the specification require this?
it('Should an error if there is no layer called "objects"', async () => {
    expect(() => town.initializeFromMap(testingMaps.noObjects))
        .toThrowError('There is no layer called "objects"');
});
```

Good Tests are Clear

- Clear tests help ensure that the bug is in the SUT, not in the test itself.

```
// not clear: if this fails, is the bug in SUT or in the test itself?
it('remove() only removes one', () =>{
        const tree = makeBST();
        for (let i = 0; i < 1000; ++i) {
          tree.add(i);
        }
        for (let j = 0; j < 1000; ++j) {
          for (let i = 0; i < 1000; ++i) {
            if (i != j) tree.remove(i);
          }
          expect(tree.contains(j)).
            toBe(true);
        }
      });
```

“Software Engineering at Google: Lessons Learned from Programming Over Time,” Wright, Winters and Manshreck, 2020 (O’Reilly)

Good Tests produce informative data when they fail

- If this test fails, all you get is "expected: true; received: false"
- Not very helpful!

```
// not clear: if this fails, is the bug in SUT or in the test itself?
it('remove() only removes one', () =>{
        const tree = makeBST();
        for (let i = 0; i < 1000; ++i) {
          tree.add(i);
        }
        for (let j = 0; j < 1000; ++j) {
          for (let i = 0; i < 1000; ++i) {
            if (i != j) tree.remove(i);
          }
          expect(tree.contains(j)).
            toBe(true);
        }
      });
```

“Software Engineering at Google: Lessons Learned from Programming Over Time,” Wright, Winters and Manshreck, 2020 (O’Reilly)

Good Tests Invoke Public APIs Only

- Interact with SUT as a client of the SUT would:
  - Public methods of classes
  - Exported members of modules

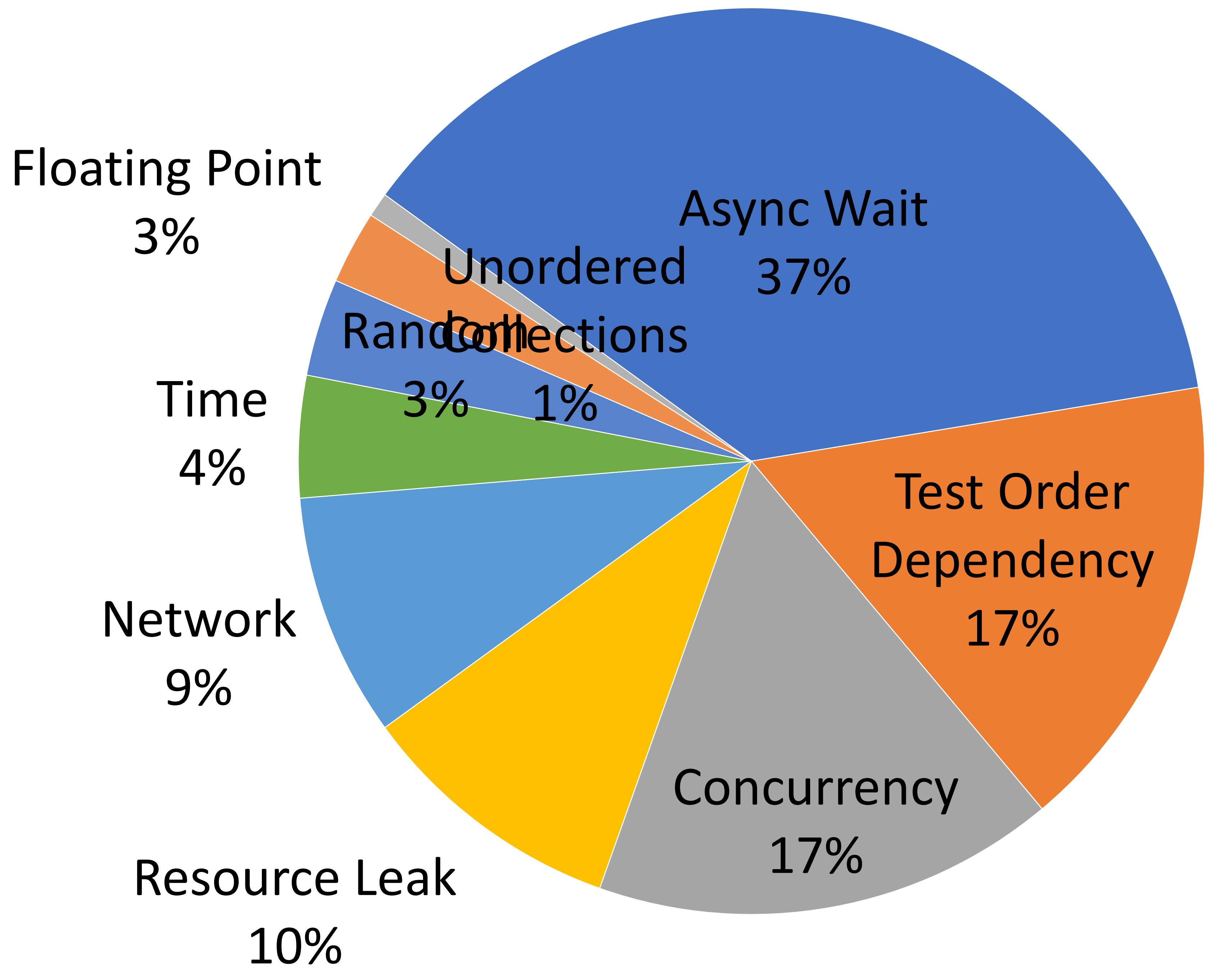
```
public initializeFromMap(map: ITiledMap) {
    ...
    this._validateInteractables();
}

// can't test this directly..
// instead, test via inititalizeFromMap
private _validateInteractables() {
    // Test Me!
}
```

It might be tempting to make _validateInteractables public and test it directly: but that's not how clients would call it!

Good Tests Aren't *Flaky*

- A *flaky test* is one that may fail unpredictably, or due to causes other than the SUT or the test itself.
- Flaky test failures are false alarms
- Most common cause of *flaky* test failures: "async wait" - tests that expect some asynchronous action to occur within a timeout



[Luo et al, FSE 2014 "An empirical analysis of flaky tests"]

Pattern for testing an async function

```
import axios from 'axios'

async function echo(str: string) : Promise<string> {
    const res =
        await axios.get(`https://httpbin.org/get?answer=${str}`)
    return res.data.args.answer
}

test('echo should return its argument', async () => {
    expect.assertions(1)
    await expect(echo("33")).resolves.toEqual("33")
})
```

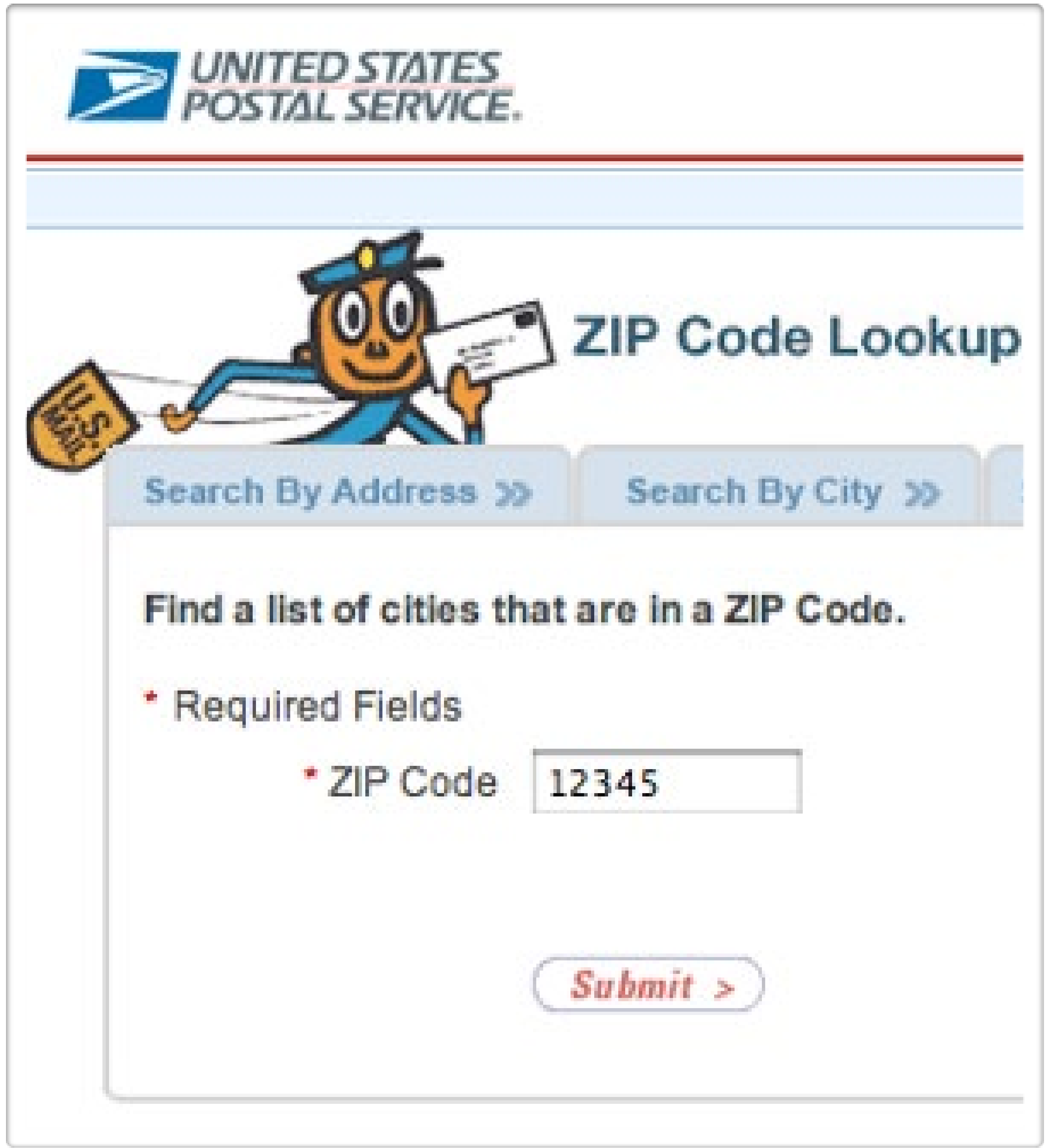
Building Test Suites From Specifications (TDD)

- First task is to enumerate the different classes of behaviors in the specification.
- Example:
  - Requesting the transcript for a student ID.
  - Two cases:
    - The ID belongs to a student
    - The ID is not the ID of any student
  - The SUT should work similarly for all inputs in each case.

Jargon: these are sometimes called "equivalence classes" of inputs.

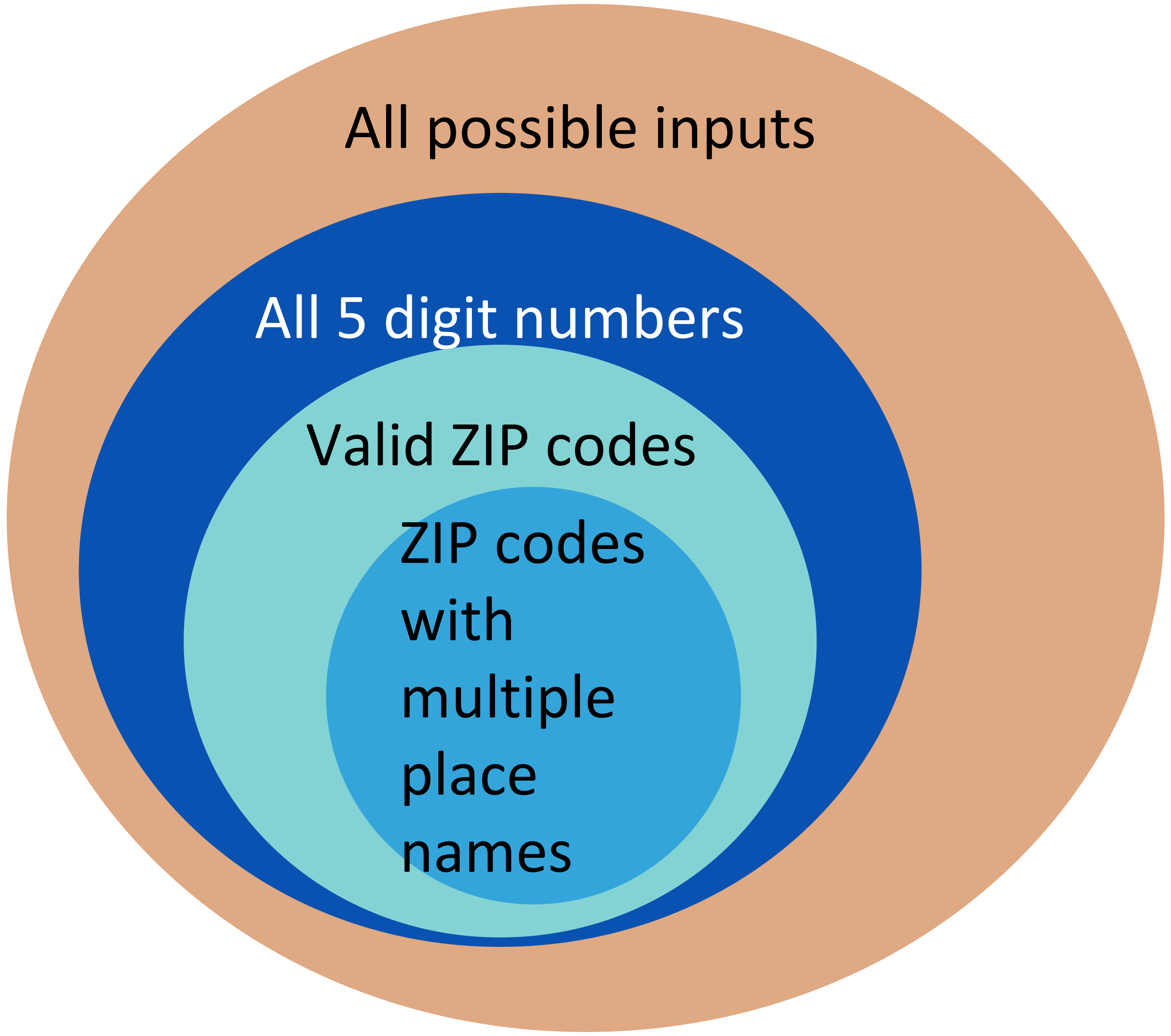
Building Test Suites From Specifications: Zip Code Lookup

- USPS ZIP code lookup tool accepts a zip code as input, and outputs:
  - The "place names" that correspond to that ZIP code, or
  - "Invalid zip code"
- Strategy:
  - Determine the input equivalence classes, boundary conditions
  - Write tests for those inputs



Building Test Suites From Specifications: Zip Code Lookup

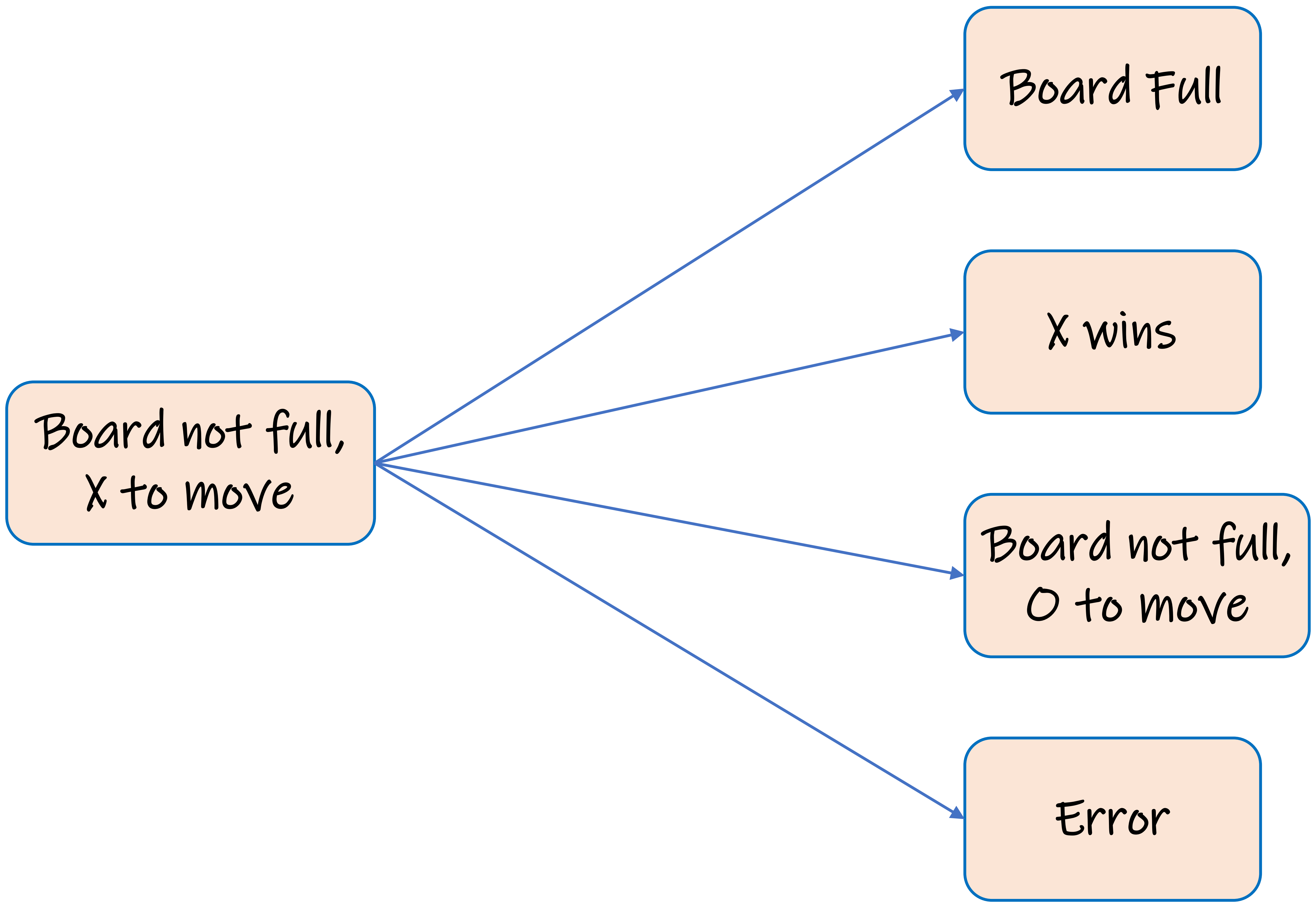
- Need to test behavior when the input is:
  - Not a 5 digit number
  - A 5 digit numbers
    - A valid ZIP code
      - With one place name
      - With multiple place names
    - Not a valid ZIP code
- Test at least one input from each class, plus boundaries (e.g. 4 digit numbers, 6 digit numbers, no numbers)
- Encode the expected output of the system for each test



Example: TicTacToe

- What are the possible states of a tictactoe game?
  - Board is full (draw)
  - Board is not full
    - Board not full, one player has won
    - Board not full, X to move
    - Board not full, O to move
- What are the possible inputs to the tictactoe game?
  - X moves
  - O moves
  - Someone else tries to move
  - X or O leaves the game
- Can make a graph out of these

A piece of the TicTacToe Graph



Your tests should exercise each of these arrows.

Make sure the regions have the right boundaries.

- Test at and near boundaries
  - Barely legal, barely illegal inputs
  - < vs <=
  - Empty inputs?
- Integer overflows / buffer overflows
  - ComAir crew scheduling
  - problem due to a list getting more than 32767 elems
- https://arstechnica.com/uncategorized/2004/12/4490-2/

Building Tests from Specifications (TDD)

- The real specification is often implicit.
- When delivering a feature, it is important to deliver tests to ensure that the feature keeps working this way *in the future*
- You may have specific domain knowledge that future developers who touch the code do not
- Specifications are hard to interpret and check, automated tests are easy (consider individual project…)
- Beyoncé rule: “If you liked it you should have put a ~~ring~~ **test** on it” (SoftEng @ Google)

When have I written enough tests?

- Hard to verify that your tests cover the whole specification
  - Especially if the specification is only in someone's head!
- But easy to verify that your tests cover all of your code.
- This is called "Code Coverage"
- Coverage gives a quantitative measure of how much of your code is exercised by your tests
- If the code isn't exercised, it's definitely not tested!

Measures of code coverage

- Statement or Block coverage
- Branch coverage
- Path coverage

Statement Coverage

- Each line (or part of) the code should be executed at least once in the test suite
- Adequacy criterion: *each statement must be executed at least once*

$$\text{Coverage: } \frac{\#\ \text{executed statements}}{\#\ \text{statements}}$$

Branch Coverage

- Adequacy criterion: *each branch in the control-flow graph must be executed at least once*

$$\text{coverage: } \frac{\#\ \text{executed branches}}{\#\ \text{branches}}$$

- Subsumes statement testing criterion because traversing all edges implies traversing all nodes

- Most widely used criterion in industry

Tools for measuring coverage

- Coverage is computed automatically while the tests execute
- jest --coverage
  - Makes it easy

```
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 |                   |
[-------------|----------|----------|----------|----------|-------------------|
```

*see example at https://github.com/philipbeel/example-typescript-nyc-mocha-coverage

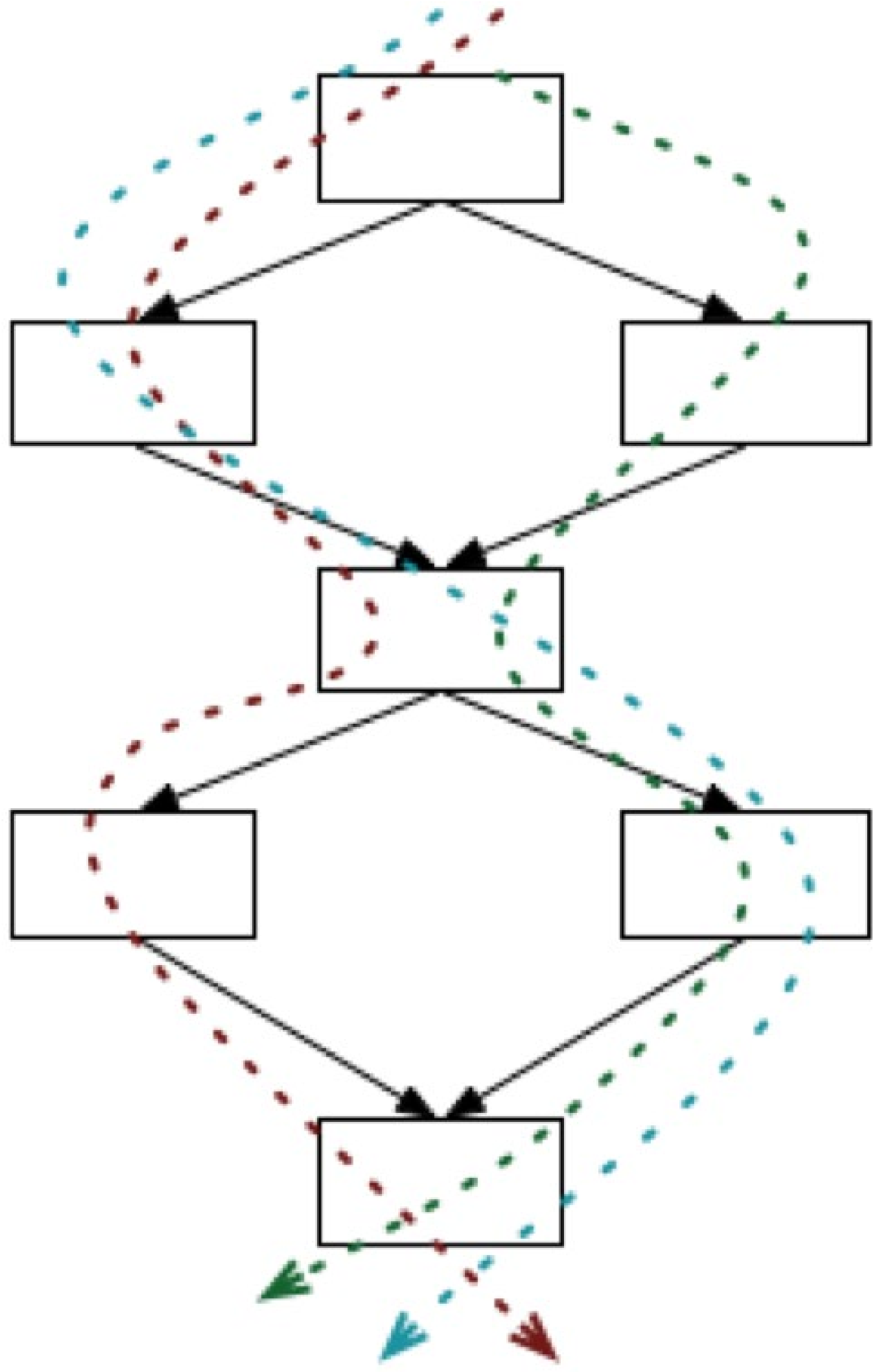
Every Branch Executed != Every Behavior Executed

- In this example, all branches are covered by the test
- However: magic will crash under certain inputs

```
function magic(x: number, y: number) {
  let z = 0;
  if (x !== 0) {   ✓ T1
    z = x + 10;
  } else {   ✓ T2
    z = 0;
  }
  if (y > 0) {   ✓ T1
    return y / z;
  } else {   ✓ T2
    return x;
  }
}
test("100% branch coverage", () => {
  expect(magic(1, 22)).toBe(2); //T1
  expect(magic(0, -10)).toBe(0); //T2
});
```

Path Coverage is Exhaustive

- Sometimes a fault is only manifest on a particular path
  - E.g., choosing the left branch and then choosing the right branch. (dashed blue path)
- But the number of paths can be infinite
  - E.g., if there is a loop.
- There are ways to bound the number of paths to cover.



100% Coverage may be Impossible

- Path coverage (even without loops)
  - Dependent conditions: if (x) A; B; if (x) C; D
    - A-B-D is a path in the flow graph, but will never happen because this code will always execute either A and C or neither.
- Branch coverage
  - Dead Branches e.g., if (x < 0) A; else if (x == 0) B; else if (x > 0) C;
  - (x > 0) test will always succeed
- Statement coverage
  - Dead code (e.g., defensive programming)

Another approach: Adversarial Testing

- Goal: “A good test suite finds all of the bugs”
- Problem: How to know the bugs that we might have made?
- Strawman - “Seeded Faults”:
  - Create N variations of the codebase, each with a single manually-written defect
  - Evaluate the number of defects detected by test suite
  - Test suite is “good” if it finds all of the bugs you can think of

Mutation Analysis tests the Tests

- Idea: What if many (real) bugs could be represented by a single, one-line “mutation” to the program?

```
public contains(location: PlayerLocation): boolean {
  return (
    location.x + PLAYER_SPRITE_WIDTH / 2 > this._x &&
    location.x - PLAYER_SPRITE_WIDTH / 2 < this._x + this._width &&
    location.y + PLAYER_SPRITE_HEIGHT / 2 > this._y &&
    location.y - PLAYER_SPRITE_HEIGHT / 2 < this._y + this._height
  );
}
```

Correct code for checking whether one sprite contains another

```
public contains(location: PlayerLocation): boolean {
  return (
    location.x + PLAYER_SPRITE_WIDTH / 2 < this._x &&
    location.x - PLAYER_SPRITE_WIDTH / 2 < this._x + this._width &&
    location.y + PLAYER_SPRITE_HEIGHT / 2 > this._y &&
    location.y - PLAYER_SPRITE_HEIGHT / 2 < this._y + this._height
  );
}
```

Mutated (and buggy) code for ‘Contains”

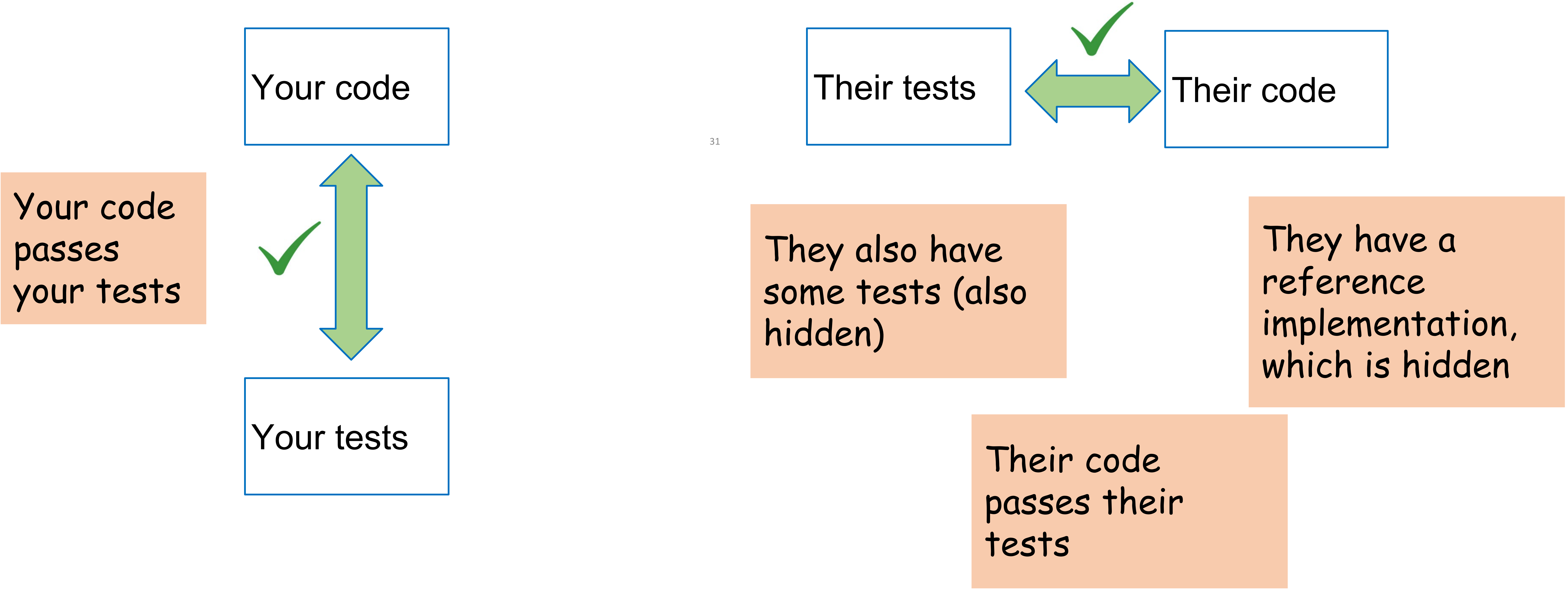
Mutation Testing Judges the tests

- Mutation testing is a way of judging whether you have written enough tests.
- It is helpful to think of mutation testing as a game in which you play against an adversary– in IP1, this was the autograder
- In mutation testing, the adversary generates a set of "mutants" – buggy versions of a reference solution.
- You win against the adversary if your tests reject all of the mutants.

The Autograder Game: Setup (Part 1)

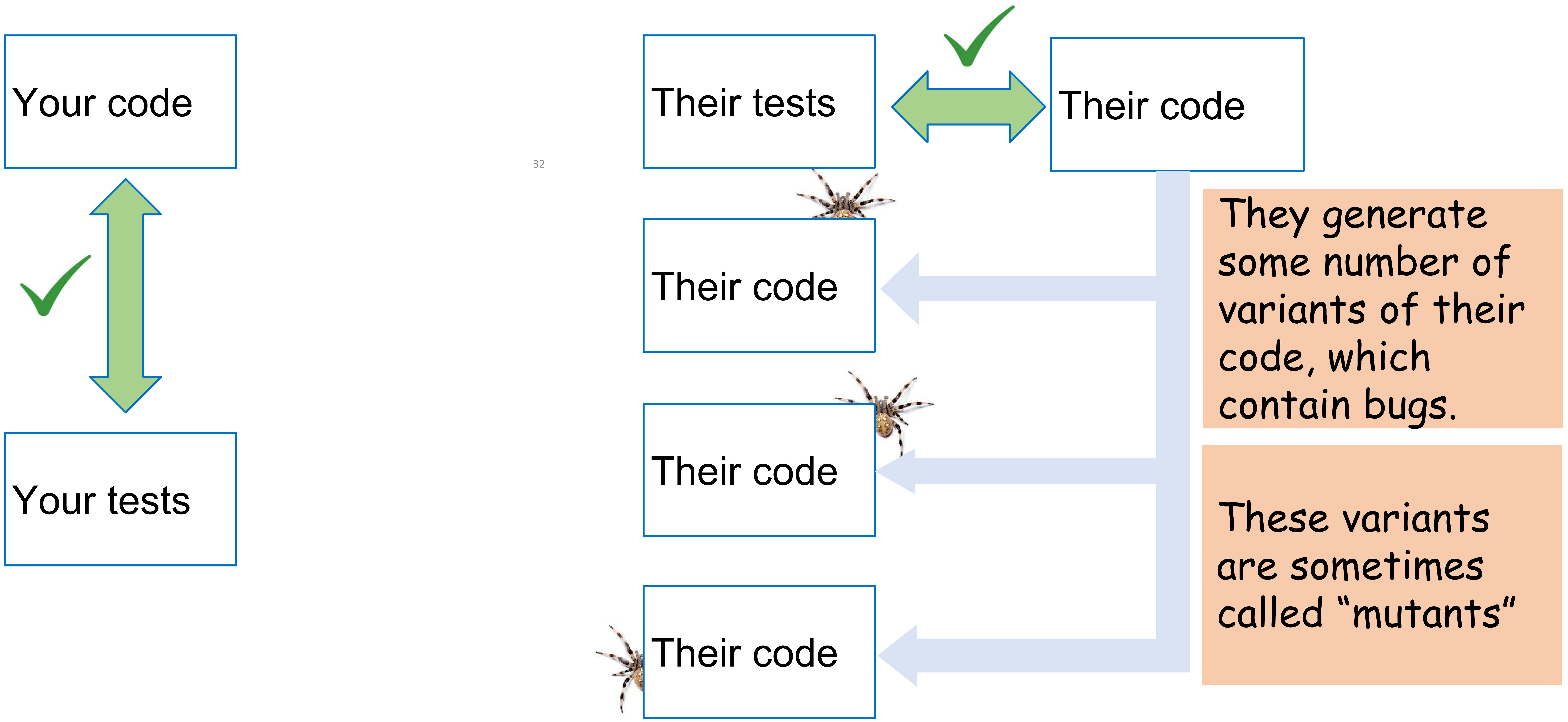
Player (You)

Opponent (Them)



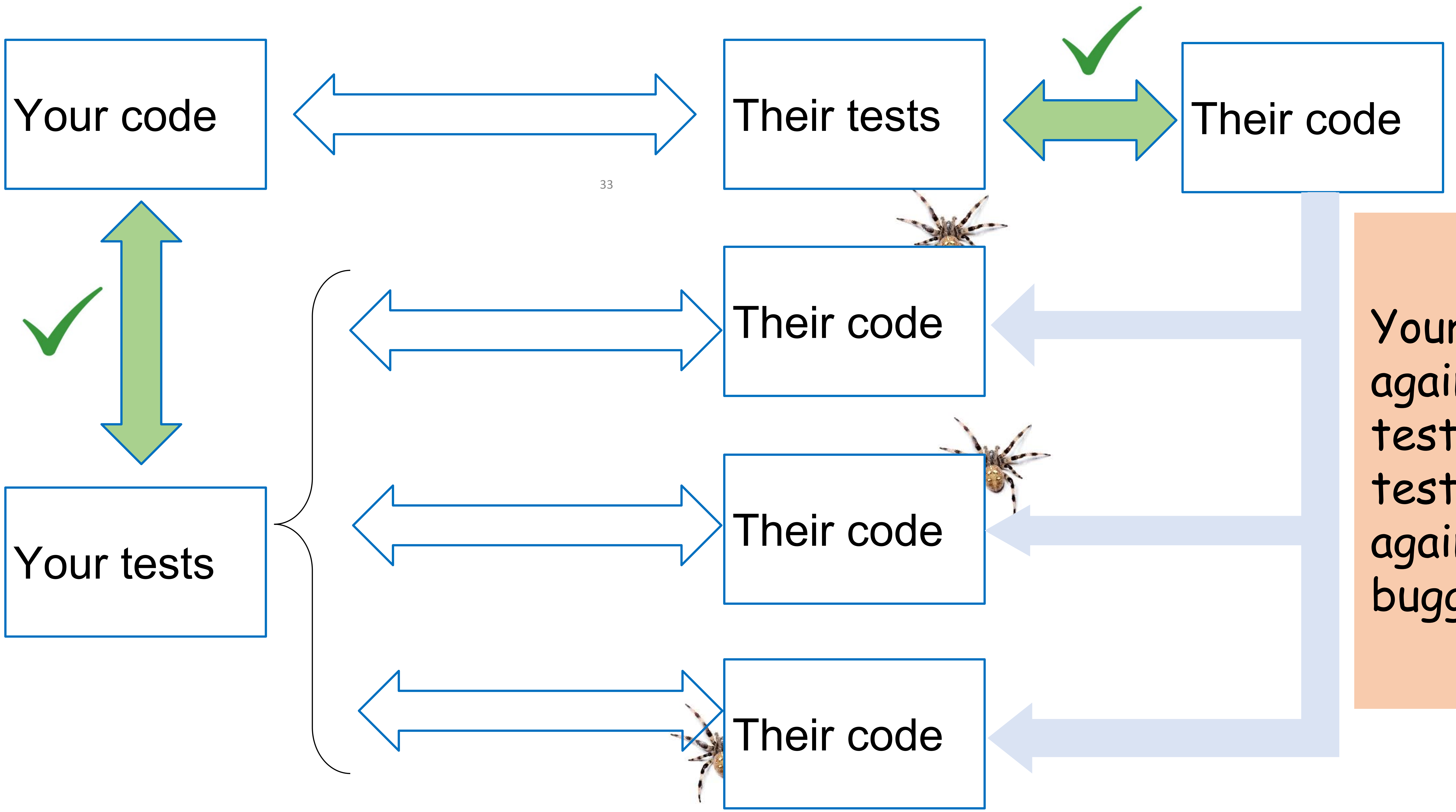
The Autograder Game:
Setup (Part 2)

Player (You)   Opponent (Them)



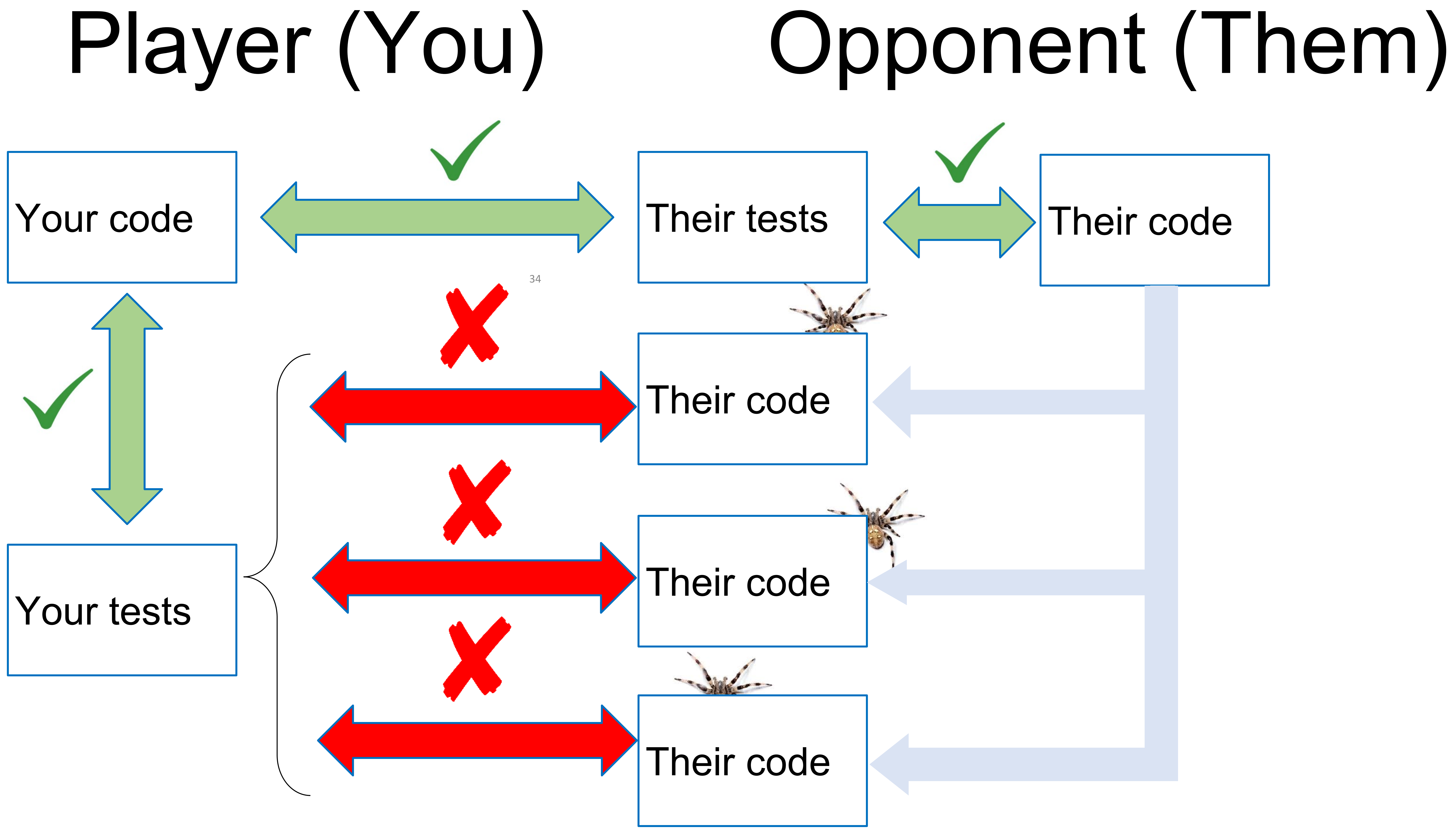
The Autograder Game: Play

Player (You) | Opponent (Them)

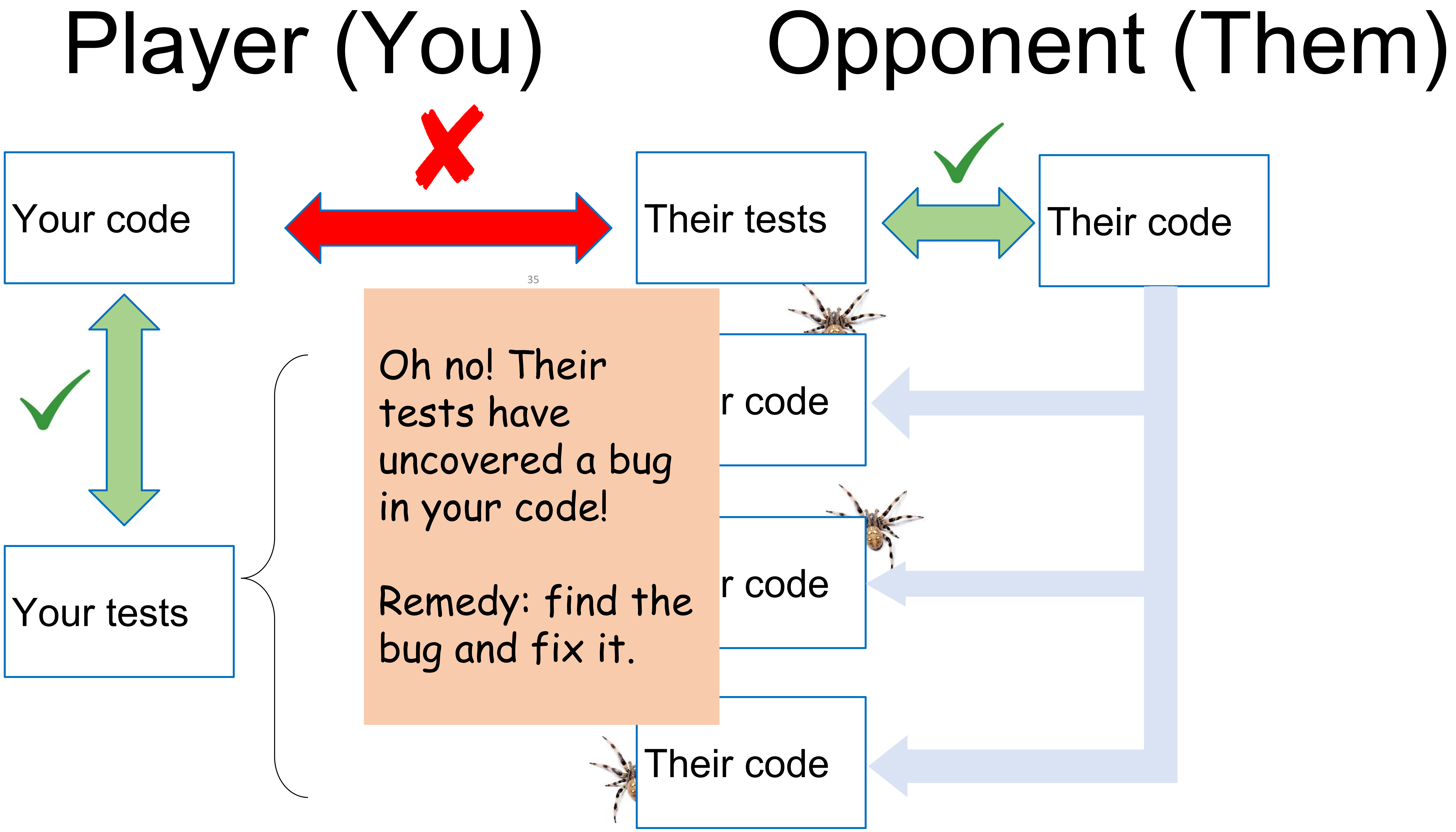


Your code is run against their tests, and your tests are run against their buggy mutants

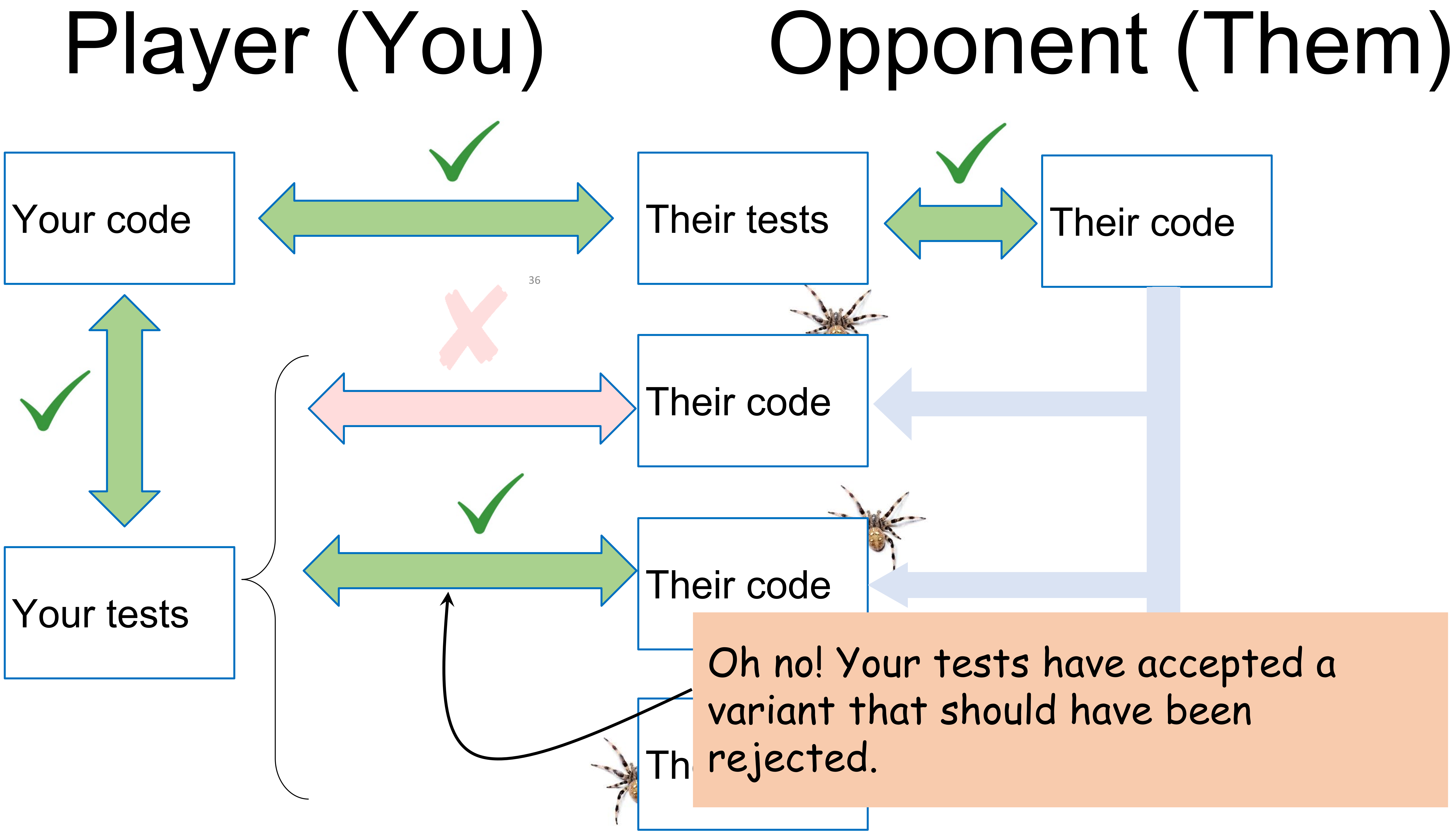
The Autograder Game: Your Winning Position



The Autograder Game: Losing Position #1



The Autograder Game: Losing Position #2



Remedy: strengthen your tests

- In this situation, you need to add some tests.
- Then you should check that your code passes your revised tests.

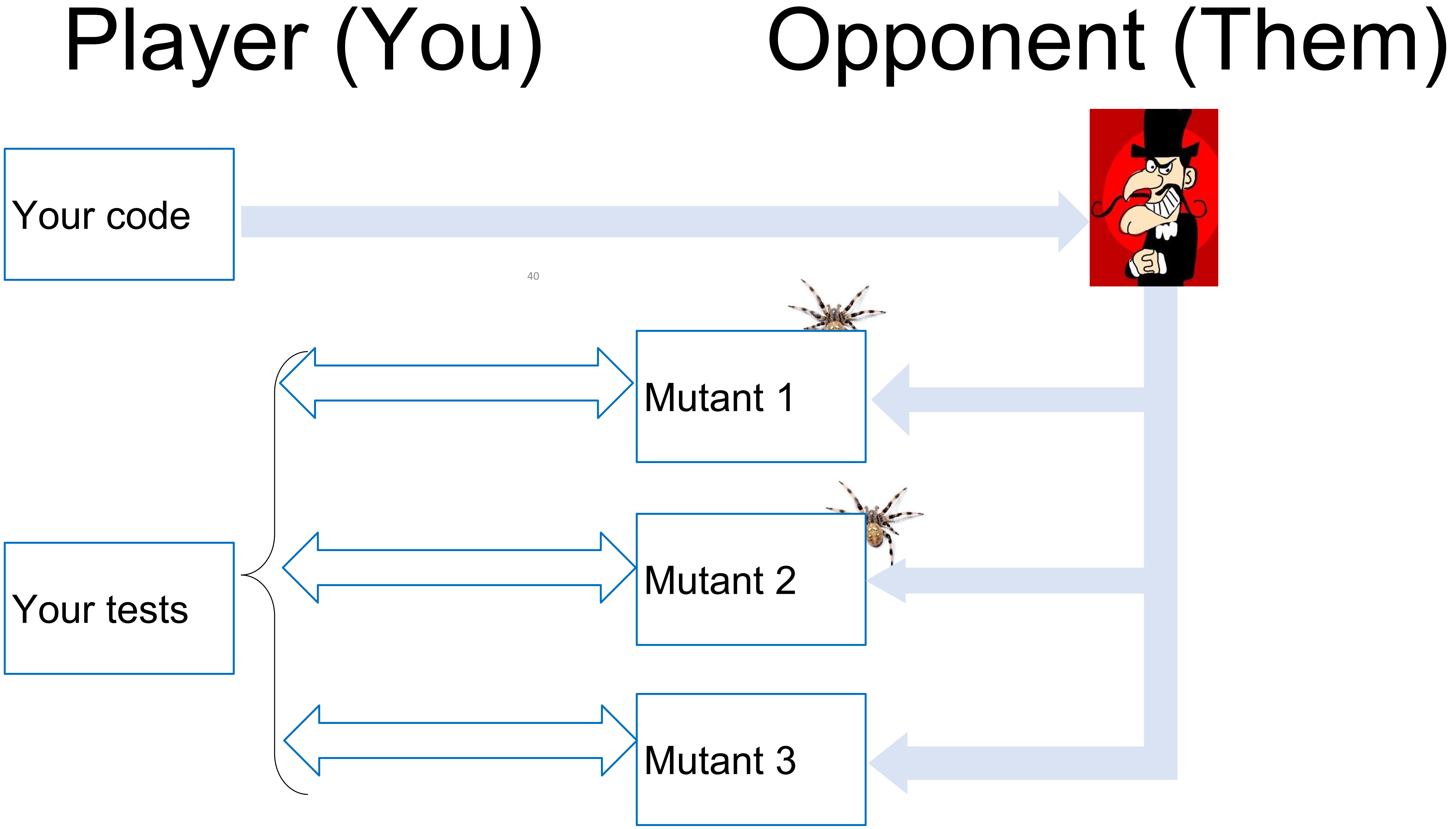
Hmm, what did I miss?

- Different versions of the game may give you different clues about what you missed.
- For IP1, we ran each variant against "their tests" and noted which tests failed
- The you got a "Clue", which consists of the titles of the tests that failed
- This was supposed give you an idea of what requirements you need to add tests for.
- Other systems do different things– we'll talk about this in Module 12.

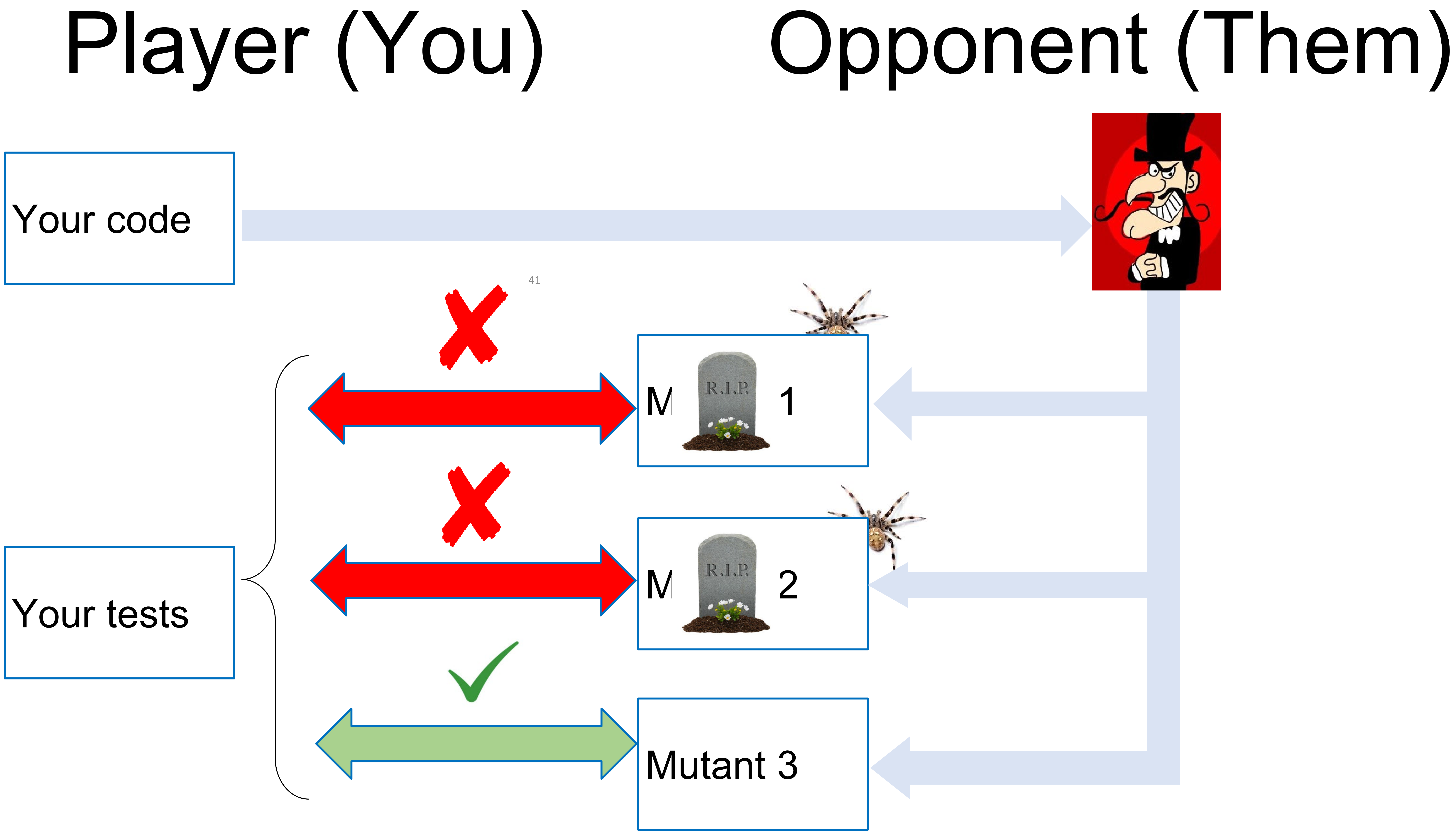
Mutation Analysis tests the Tests

- Automatically mutates SUT to create mutants, each a single change to the code
- Runs each test on each mutant, until finding that a mutant is detected by a test
- Can be a time-consuming process to run, but fully automated
- State-of-the-art mutation analysis tools:
  - Pit (JVM)
  - Stryker (JS/TS, C#, Scala)

The Stryker Game: The Opening

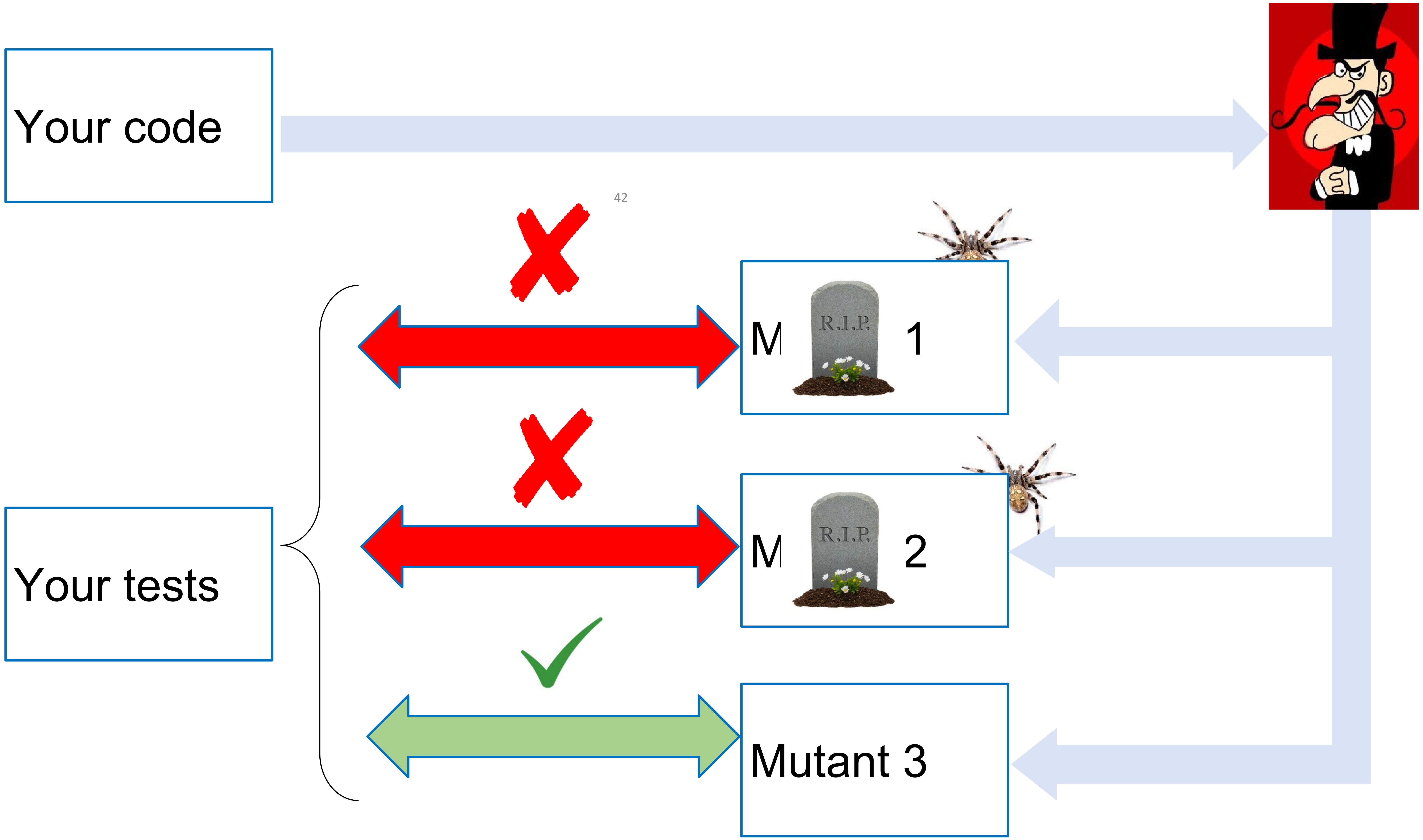


The Stryker Game: Result of one round of play

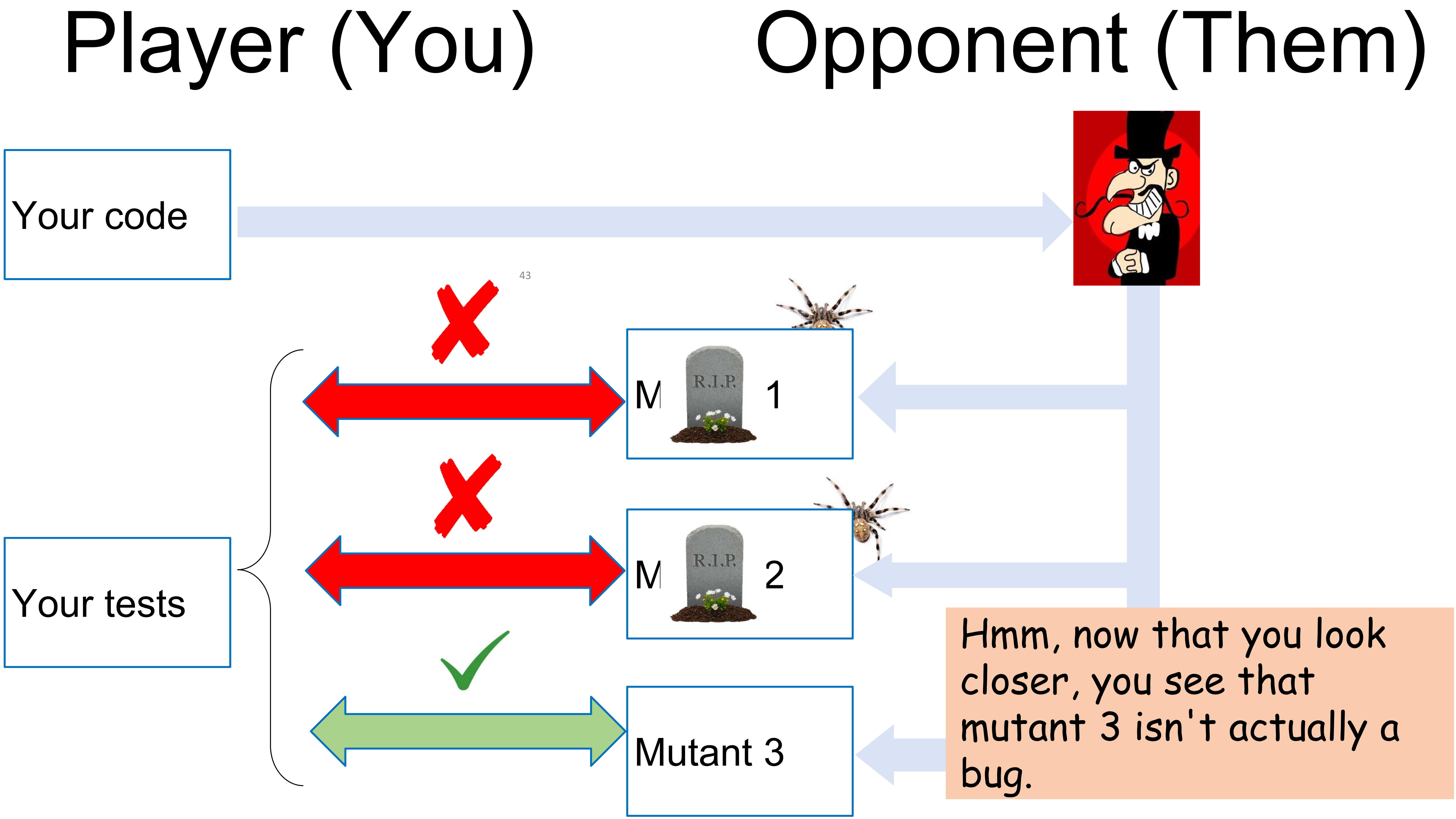


The Stryker Game: Result of one round of play

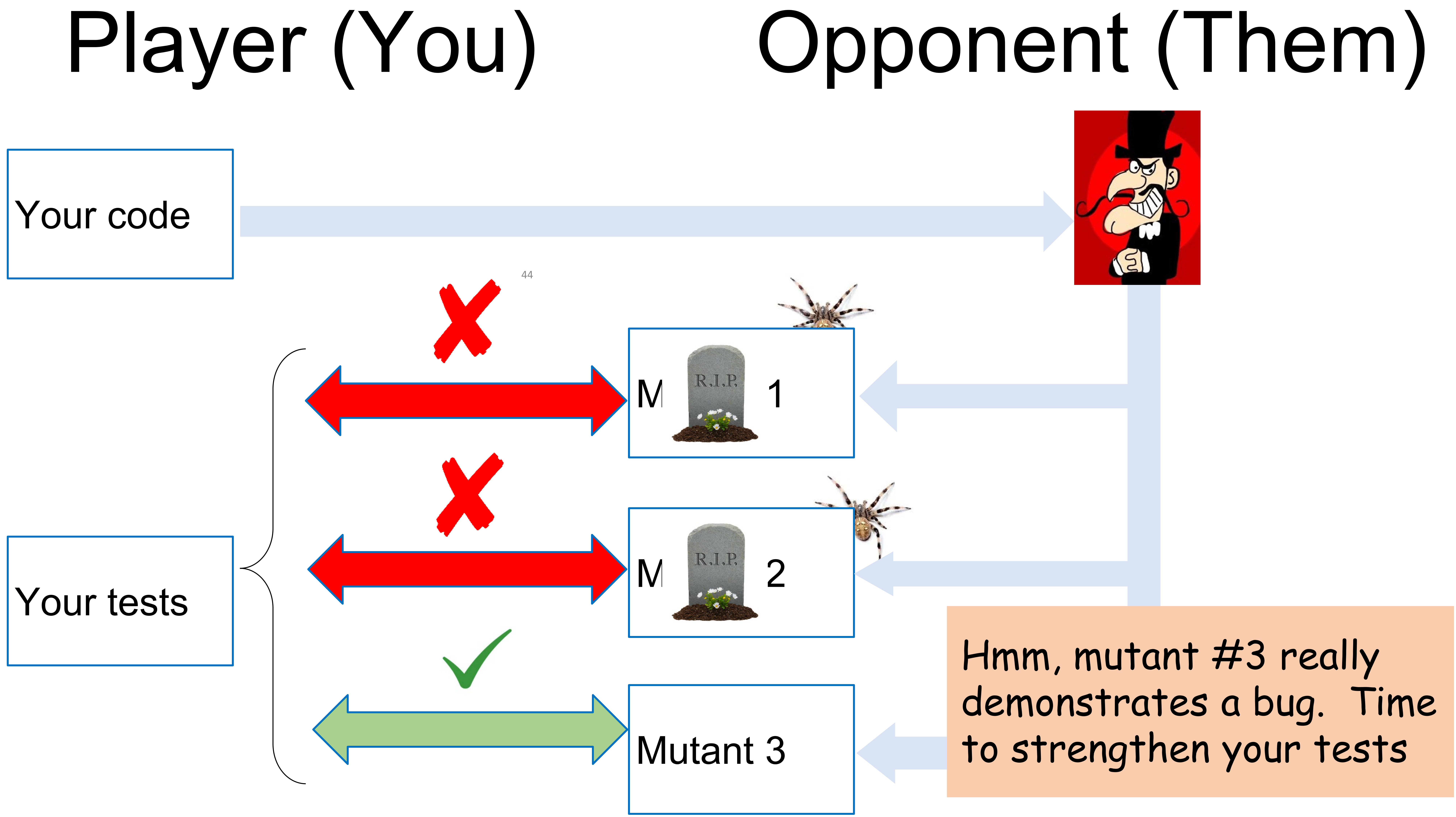
Player (You) Opponent (Them)



The Stryker Game: a winning position



The Stryker Game: a losing position



Mutation Report Shows Undetected Mutants

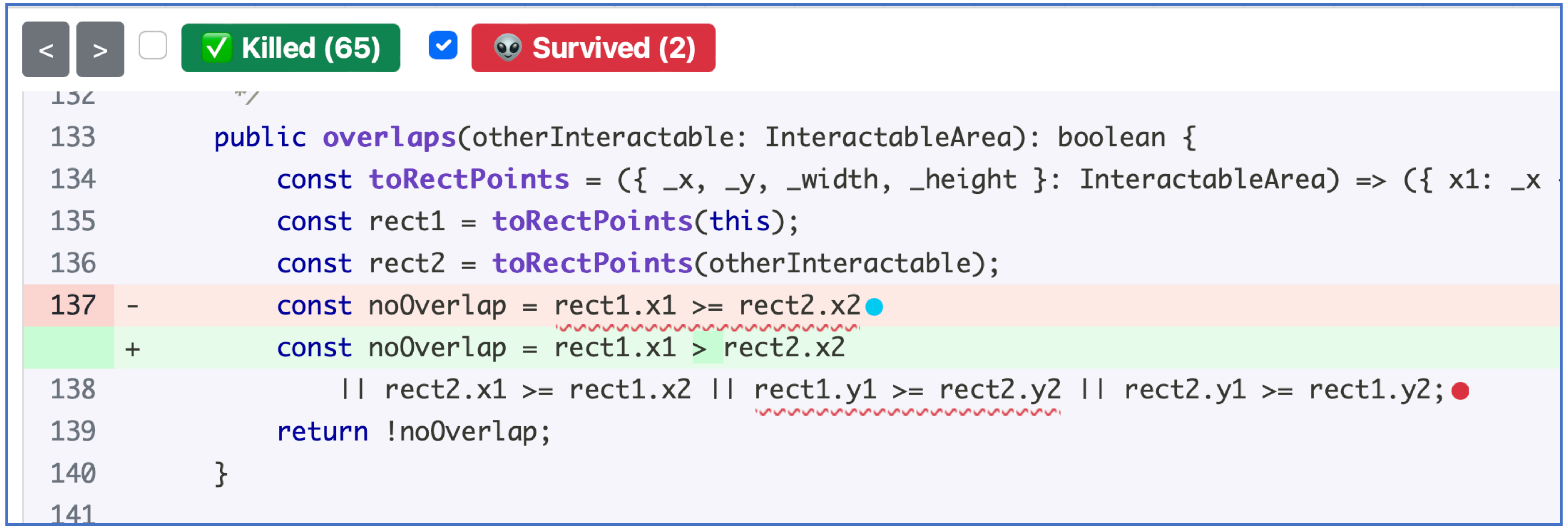
- Mutants "detected" are bugs that are found
- Mutants "undetected" might be bugs, or could be equivalent to original program (requires a human to tell)

File / Directory	Mutation score		# Killed	# Survived	# Timeout	# No coverage	# Ignored	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	90.30%	90.30	121	13	0	0	0	0	0	**121**	**13**	**134**
TS ConversationArea.ts	76.92%	76.92	10	3	0	0	0	0	0	**10**	**3**	**13**
TS InteractableArea.ts	97.01%	97.01	65	2	0	0	0	0	0	**65**	**2**	**67**
TS Town.ts	85.00%	85.00	34	6	0	0	0	0	0	**34**	**6**	**40**
TS ViewingArea.ts	85.71%	85.71	12	2	0	0	0	0	0	**12**	**2**	**14**

```
public overlaps(otherInteractable: InteractableArea): boolean {
    const toRectPoints = ({ _x, _y, _width, _height }: InteractableArea) => ({ x1: _x - PLAYER_SPRI
    const rect1 = toRectPoints(this);
    const rect2 = toRectPoints(otherInteractable);
    const noOverlap = rect1.x1 >= rect2.x2
        || rect2.x1 >= rect1.x2 || rect1.y1 >= rect2.y2 || rect2.y1 >= rect1.y2;
    return !noOverlap;
}
```

Use Mutation Analysis While Writing Tests

- When you feel “done” writing tests, run a mutation analysis
- Inspect undetected mutants, and try to strengthen tests to detect those mutants



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

Undetected Mutants May Not Be Bugs

- Unfortunately, we can not automatically tell if an undetected mutant is a bug or not

```
    public static fromMapObject(mapObject: ITiledMapObject, broadcas
        const { name, width, height } = mapObject;
        if (!width || !height) {
-           throw new Error(`Malformed viewing area ${name}`);
+           throw new Error(``);
        }
        const rect: BoundingBox = { x: mapObject.x, y: mapObject.y, w
        return new ConversationArea({ id: name, occupantsByID: [] },
    }
```

This mutant is *equivalent* to the original program: Even th
the error message changed, the specification doesn't indi
what error message should be thrown.

- Here the mutation was to change the error message (from something informative to an empty string)
- Clearly that doesn't change the behavior of the program, just the error message that is generated.
- We chose not to test for this because the text of this error message was not specified in the specification
- Testing for this particular error message would have been brittle

Undetected Mutants May Not Be Bugs

- Unfortunately, we can not automatically tell if an undetected mutant is a bug or not

```diff
   public initializeFromMap(map: ITiledMap) {
     const objectLayer = map.layers.find(eachLayer => eachLayer.nam
-    if (!objectLayer) {
-      throw new Error(`Unable to find objects layer in map`);
-    }
+    if (!objectLayer) {}
     const viewingAreas = objectLayer.objects
       .filter(eachObject => eachObject.type === 'ViewingArea')
       .map(eachViewingAreaObject => ViewingArea.fromMapObject(eac
```

- This mutant is equivalent to the original program. Even without the check for undefined, an error is still thrown when the undefined layer is dereferenced on the following line.

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

Are Mutants a Valid Substitute for Real Faults in Software Testing?

René Just[†], Darioush Jalali[†], Laura Inozemtseva[*], Michael D. Ernst[†], Reid Holmes[*], and Gordon Fraser[‡]

[†]University of Washington
Seattle, WA, USA
{rjust, darioush, mernst}
@cs.washington.edu

[*]University of Waterloo
Waterloo, ON, Canada
{lminozem, rtholmes}
@uwaterloo.ca

[‡]University of Sheffield
Sheffield, UK
gordon.fraser@sheffield.ac.uk

ABSTRACT

A good test suite is one that detects real faults. Because the set of faults in a program is usually unknowable, this definition is not useful to practitioners who are creating test suites, nor to researchers who are creating and evaluating tools that generate test suites. In place of real faults, testing research often uses mutants, which are artificial faults — each one a simple syntactic variation — that are systematically seeded throughout the program under test. Mutation analysis is appealing because large numbers of mutants can be automatically-generated and used to compensate for low quantities or the absence of known real faults.

Unfortunately, there is little experimental evidence to support the use of mutants as a replacement for real faults. This paper investigates whether mutants are indeed a valid substitute for real faults, i.e., whether a test suite's ability to detect mutants is correlated with its ability to detect real faults that developers have fixed. Unlike prior studies, these investigations also explicitly consider the conflating effects of code coverage on the mutant detection rate.

Our experiments used 357 real faults in 5 open-source applications that comprise a total of 321,000 lines of code. Furthermore, our experiments used both developer-written and automatically-generated test suites. The results show a statistically significant correlation between mutant detection and real fault detection, independently of code coverage. The results also give concrete suggestions on how to improve mutation analysis and reveal some inherent limitations.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Measurement

Keywords

Test effectiveness, real faults, mutation analysis, code coverage

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2635929

1. INTRODUCTION

Both industrial software developers and software engineering researchers are interested in measuring test suite effectiveness. While developers want to know whether their test suites have a good chance of detecting faults, researchers want to be able to compare different testing or debugging techniques. Ideally, one would directly measure the number of faults a test suite can detect in a program. Unfortunately, the faults in a program are unknown a priori, so a proxy measurement must be used instead.

A well-established proxy measurement for test suite effectiveness in testing research is the *mutation score*, which measures a test suite's ability to distinguish a program under test, the *original version*, from many small syntactic variations, called *mutants*. Specifically, the mutation score is the percentage of mutants that a test suite can distinguish from the original version. Mutants are created by systematically injecting small artificial faults into the program under test, using well-defined *mutation operators*. Examples of such mutation operators are replacing arithmetic or relational operators, modifying branch conditions, or deleting statements (cf. [18]).

Mutation analysis is often used in software testing and debugging research. More concretely, it is commonly used in the following use cases (e.g., [3, 13, 18, 19, 35, 37–39]):

Test suite evaluation The most common use of mutation analysis is to evaluate and compare (generated) test suites. Generally, a test suite that has a higher mutation score is assumed to detect more real faults than a test suite that has a lower mutation score.

Test suite selection Suppose two unrelated test suites T_1 and T_2 exist that have the same mutation score and $|T_1| < |T_2|$. In the context of test suite selection, T_1 is a preferable test suite as it has fewer tests than T_2 but the same mutation score.

Test suite minimization A mutation-based test suite minimization approach reduces a test suite T to $T \setminus \{t\}$ for every test $t \in T$ for which removing t does not decrease the mutation score of T.

Test suite generation A mutation-based test generation (or augmentation) approach aims at generating a test suite with a high mutation score. In this context, a test generation approach augments a test suite T with a test t only if t increases the mutation score of T.

Fault localization A fault localization technique that precisely identifies the root cause of an artificial fault, i.e., the mutated code location, is assumed to also be effective for real faults.

These uses of mutation analysis rely on the assumption that mutants are a valid substitute for real faults. Unfortunately, there is little experimental evidence supporting this assumption, as discussed in greater detail in Section 4. To the best of our knowledge, only three previous studies have explored the relationship between mutants and

654

Activity: strengthening a test suite

- Enhance the test suite of the transcript server to improve line coverage and mutation coverage
- Download on Module 11 webpage

Review

- Now that you have come to the end of this lesson, you should be able to:
  - Give different reasons why you might want to test
  - List the properties of a good test
  - Use equivalence classes to design a TDD test suite
  - Explain 3 measures of code coverage
  - Use mutation testing to judge the completeness of a test suite