

CS 4530: Fundamentals of Software Engineering

Module 03: Test Adequacy

Jonathan Bell, Adeel Bhutta, Mitch Wand
Khoury College of Computer Sciences

When have I written
enough tests?

Module Outline

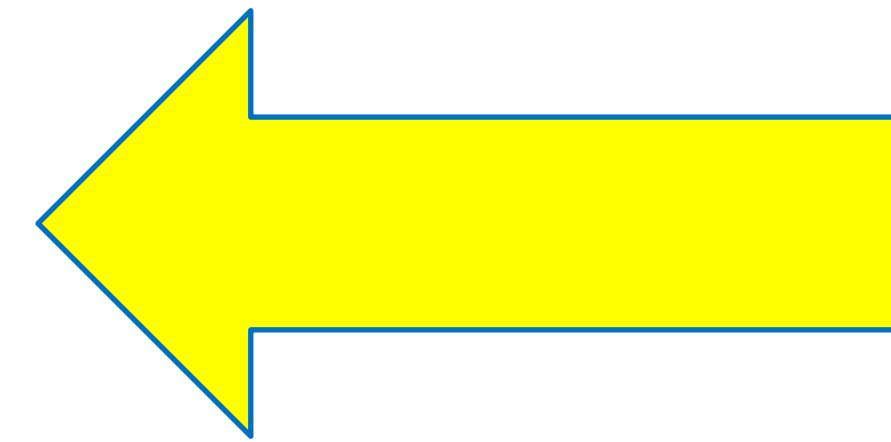
- Lesson 3.1 Writing tests for TDD
- Lesson 3.2 Assessing Test Coverage
- Lesson 3.3 Adversarial Coverage Testing

Learning Objectives for this Module

- By the end of this lesson, you should be able to:
 - Explain different reasons why you might want to test
 - Design a TDD test suite by identifying equivalence classes of inputs
 - Explain the following measures of code coverage, and how they differ:
 - Statement or line coverage
 - Branch coverage
 - Path 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 ?



CS 4530: Fundamentals of Software Engineering

Lesson 3.1 Writing tests for TDD

Jonathan Bell, Adeel Bhutta, Mitch Wand
Khoury College of Computer Sciences

What makes for a good test (suite)?

- Desirable properties of test suites:
 - Find bugs
 - Run automatically
 - Are relatively cheap to run
 - Don't depend on the order of tests.
- Desirable properties of individual tests:
 - Understandable and debuggable
 - No false alarms (not “flaky”)

Related Terminology:
“test smells”

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
- Beyoncé rule: “If you liked it you should have put a ~~ring~~ **test** on it” (SoftEng @ Google)

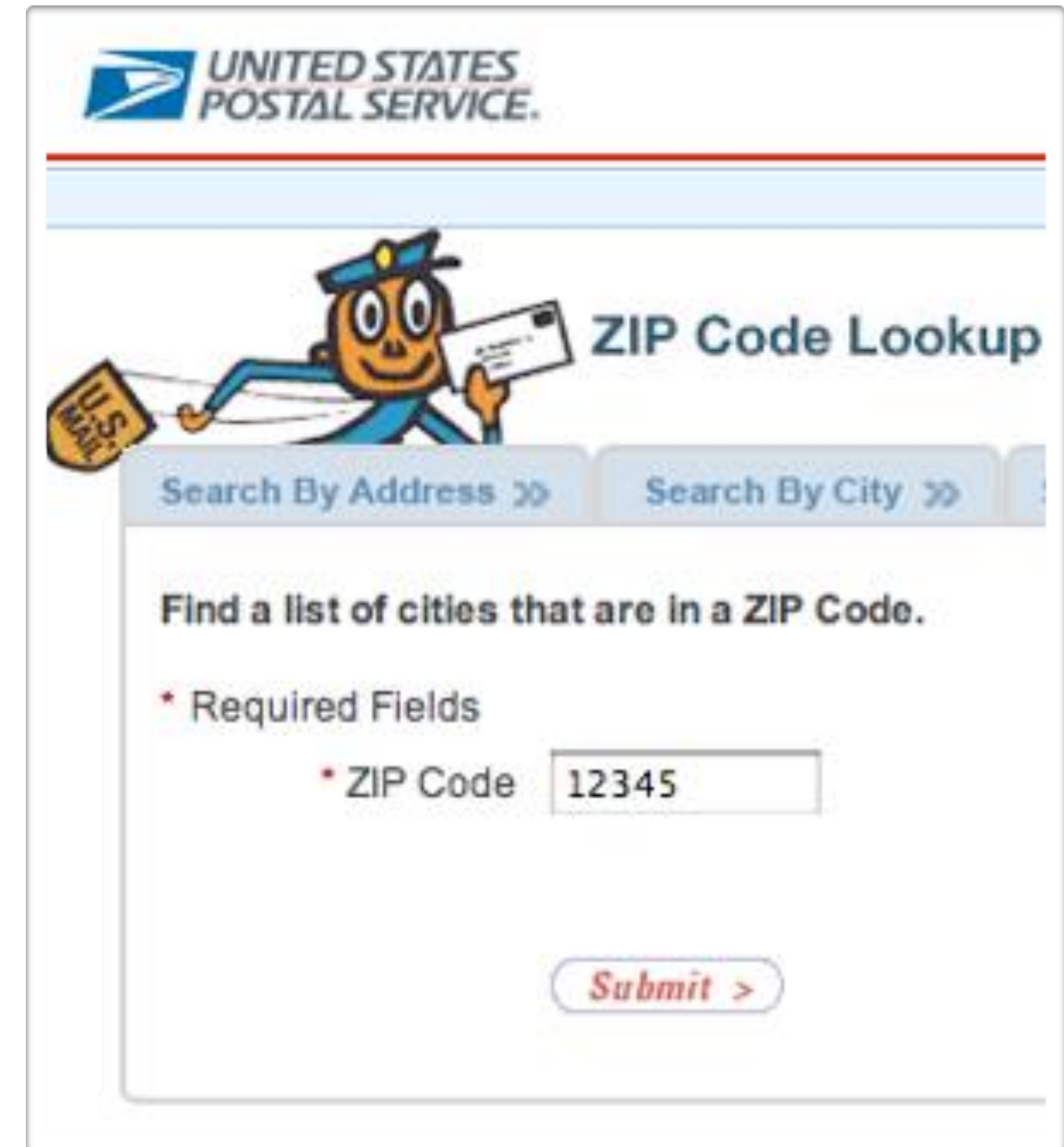
Building Test Suites From Specifications (TDD)

- 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.

These cases are sometimes called "equivalence classes" of inputs.

Example: Zip Code Lookup (1)

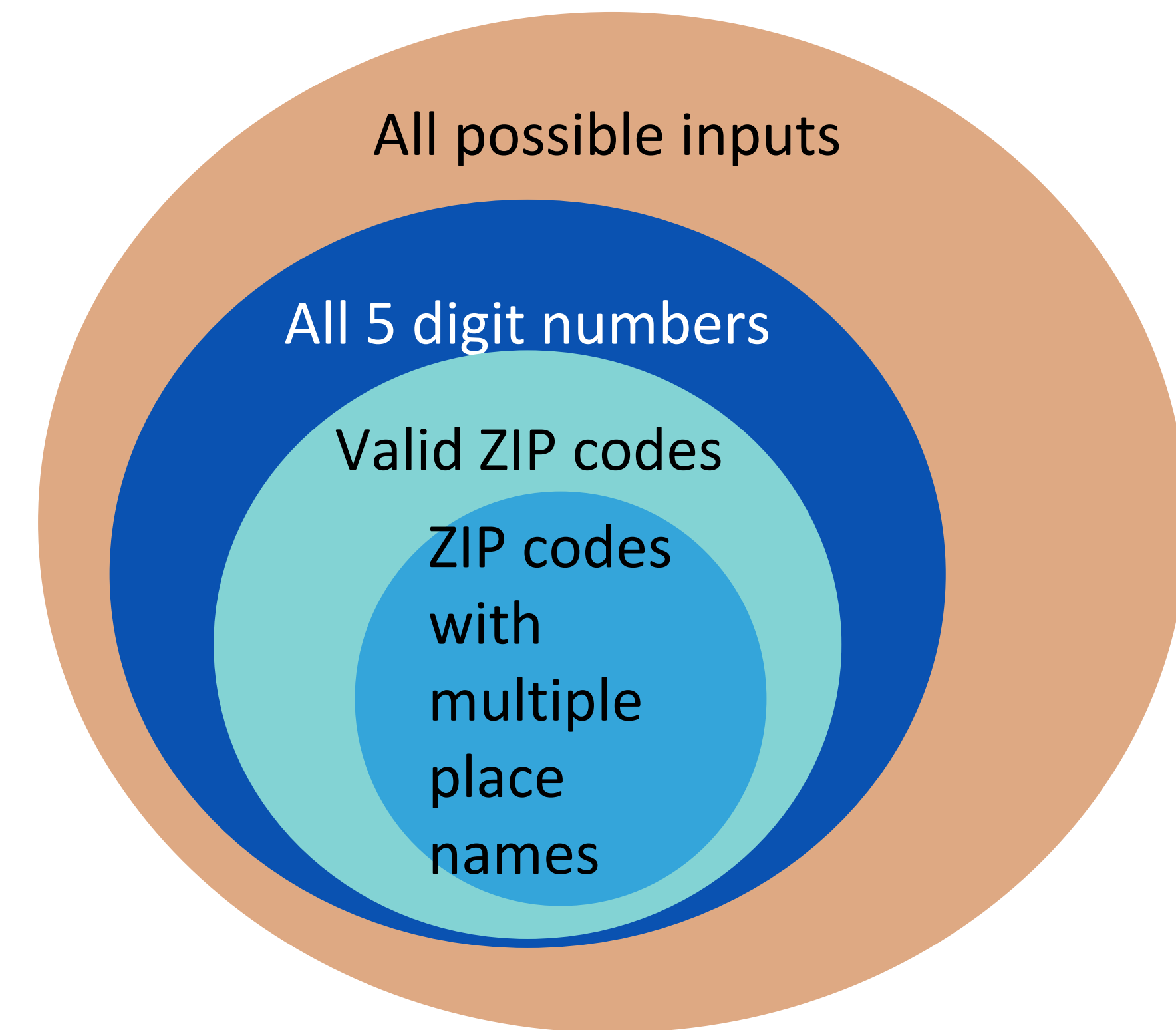
- USPS ZIP code lookup tool accepts a zip code as input, and outputs:
 - The “place names” that correspond to that ZIP code, or
 - “Invalid zip code”
- Strategy:
 - Determine the input equivalence classes, boundary conditions
 - Write tests for those inputs



The screenshot shows the USPS ZIP Code Lookup tool interface. At the top is the United States Postal Service logo. Below it is a cartoon mail carrier holding a letter. The title "ZIP Code Lookup" is prominently displayed. There are two search buttons: "Search By Address >>" and "Search By City >>". Below these buttons, the instruction "Find a list of cities that are in a ZIP Code." is shown. Under the heading "Required Fields", there is a label "ZIP Code" followed by a text input field containing the value "12345". At the bottom right of the form is a "Submit >" button.

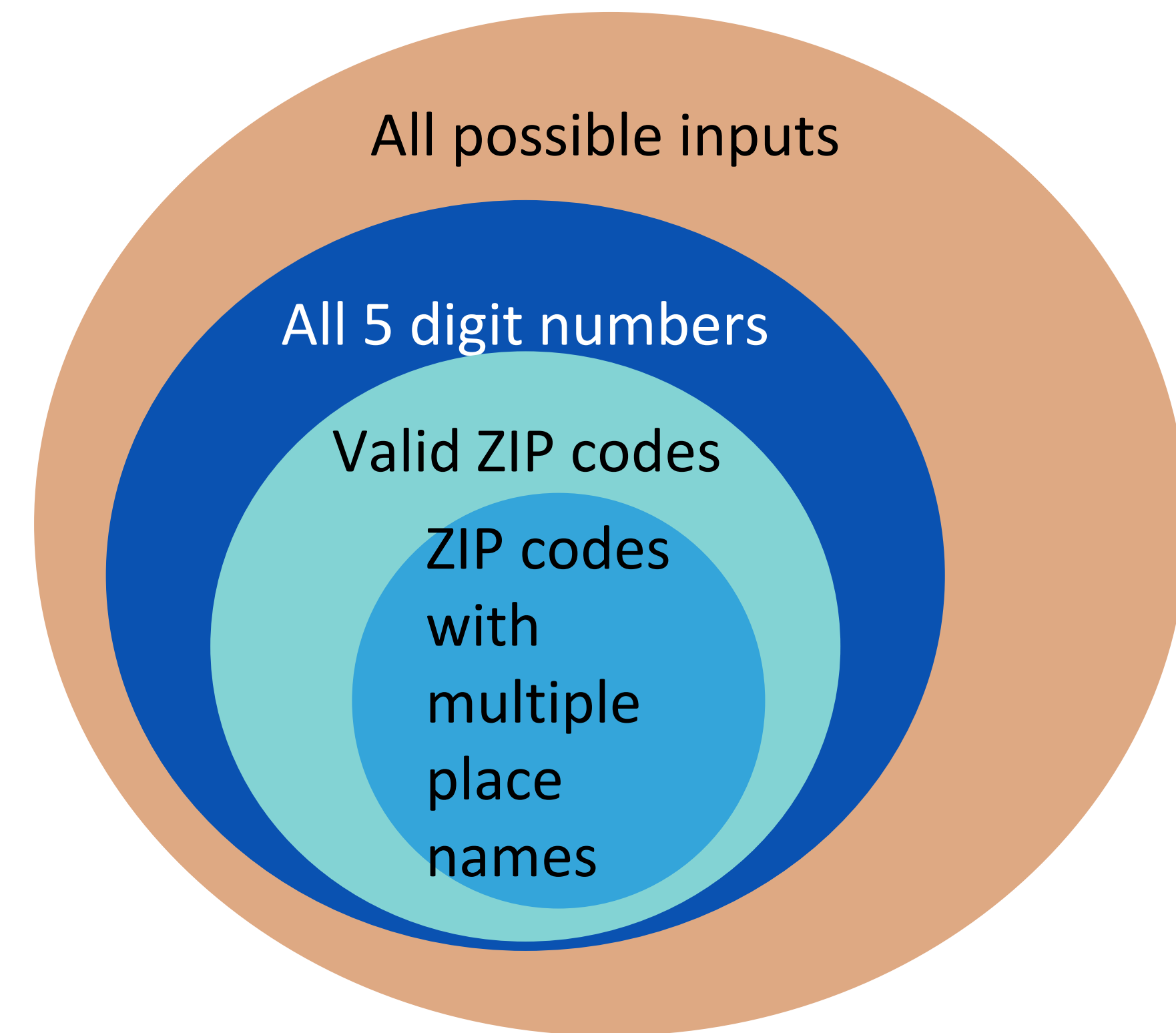
Zip Code Lookup (2): Need to test all feasible inputs

- 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



What does "all possible inputs" mean?

- Should we also test with non-numeric inputs? With an empty input? With an input that isn't even a string?
- Do we have to worry about the database going down?
- All this depends on what we can assume about the system in which the lookup tool is embedded.



Cases for looking something up in a list

1. The list is empty
2. The thing you want is not in the list
3. The thing you want is the first thing in the list
4. The thing you want is the last thing in the list
5. The thing you want is in the middle of the list

Example:

```
// find the first item in the list that is
// greater than or equal to the target.
// throw an error if none.
export default function search(list:number[], target:number) {
  // NEED TO TEST WHAT GOES HERE
}
```

1. The list is empty
2. The thing you want is not in the list
3. The thing you want is the first thing in the list
4. The thing you want is the last thing in the list
5. The thing you want is in the middle of the list

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, your turn
 - Board not full, the other person's turn
- What are the possible inputs to the tictactoe game?
 - You move
 - The other player moves
 - Someone else tries to move
 - One of the players leaves the game

Make sure you've covered the edge cases

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



But don't make unwarranted assumptions about the specification

- Specifications often leave room for undefined behaviors: details that are subject to change
- *Brittle tests* are tests that will fail unexpectedly if that undefined behavior changes

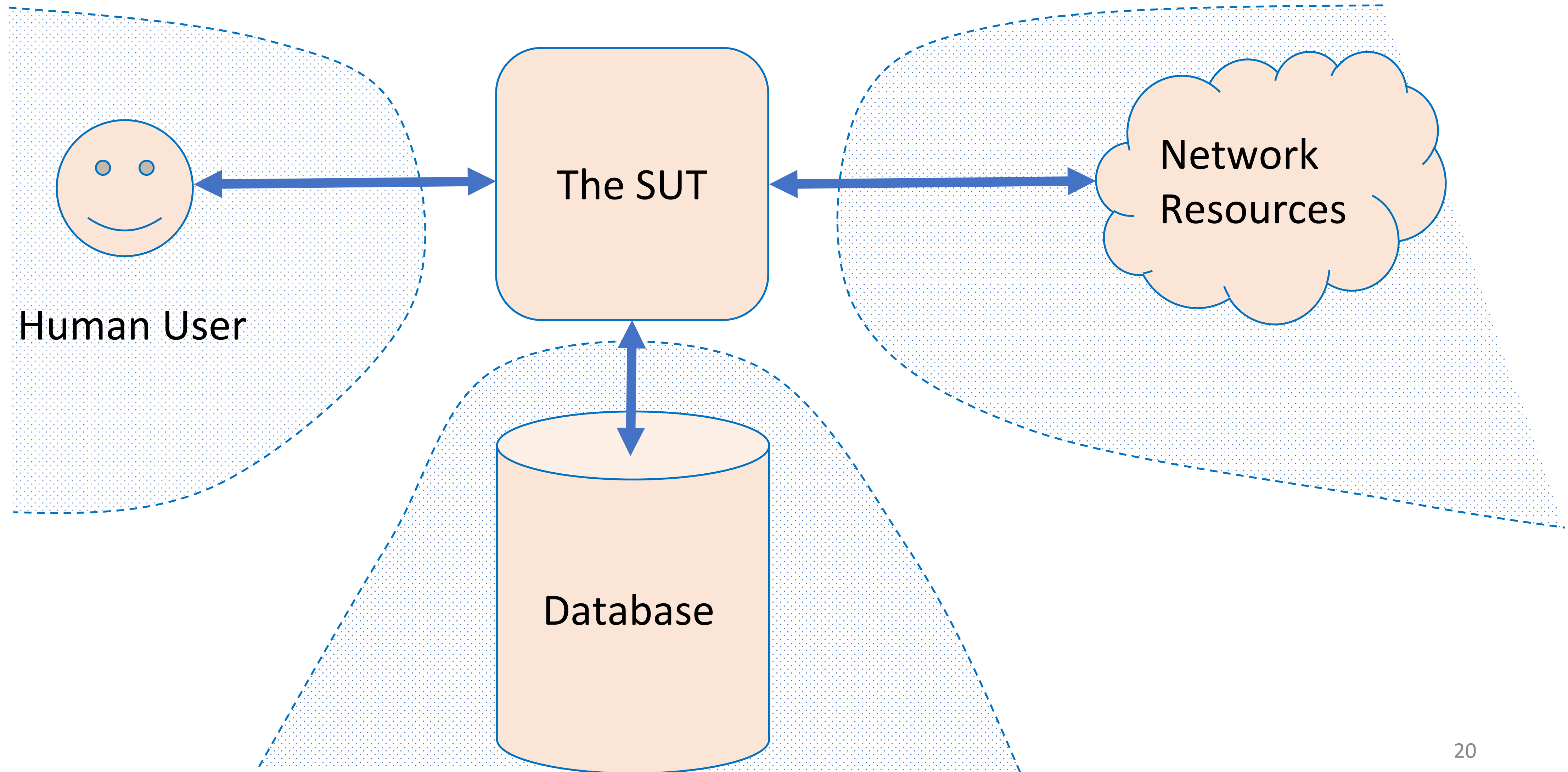
Example: Is this particular error message required, or is it incidental?

```
test("should throw an error if no such item", () => {  
  const list = [1, 2, 3];  
  const target = 4;  
  expect(search(list, target)).toThrowError("No such item");  
});
```

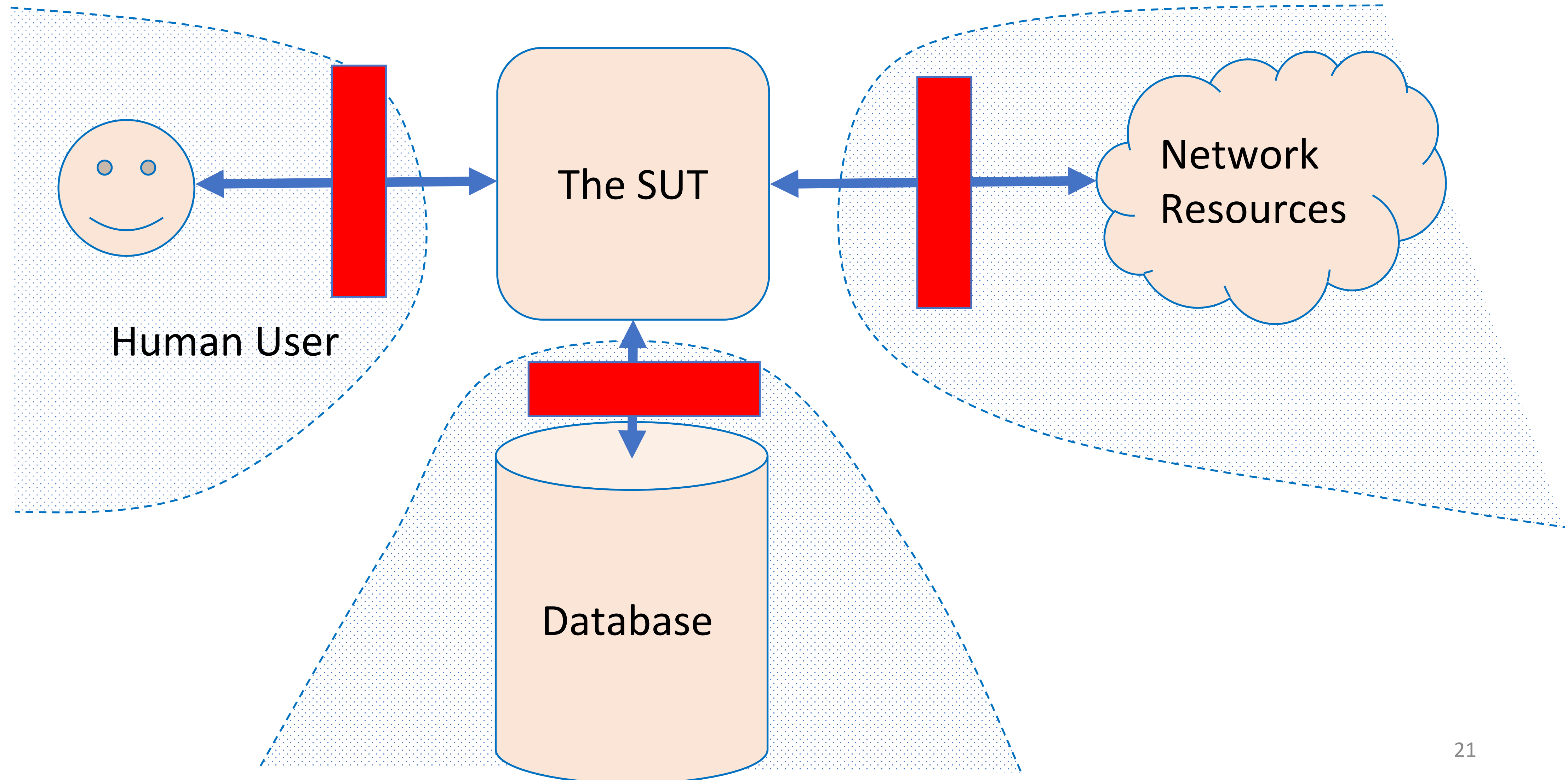
What does it mean for a test to succeed?

- *Test Oracles* define the criteria for a test to succeedPossible kinds of test oracles
 - 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

Your module may interact with uncontrollable things in the environment



Test doubles replace uncontrollable things with things that you do control



Test Doubles Intercept Calls to Methods

- Testing frameworks provide two common abstractions for doubles.
- In Jest, these are called **mocks** and **spies**.
- Other frameworks use terms like "fake" and "stub" for variants of these.
- You'll find more detail in the tutorial on Unit Testing.
- We'll discuss these in more detail in a later module.

CS 4530: Fundamentals of Software Engineering

Module 03.2 Measures of Test Coverage

Jonathan Bell, Adeel Bhutta, Mitch Wand
Khoury College of Computer Sciences

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 easier 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*

Statement 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 | |

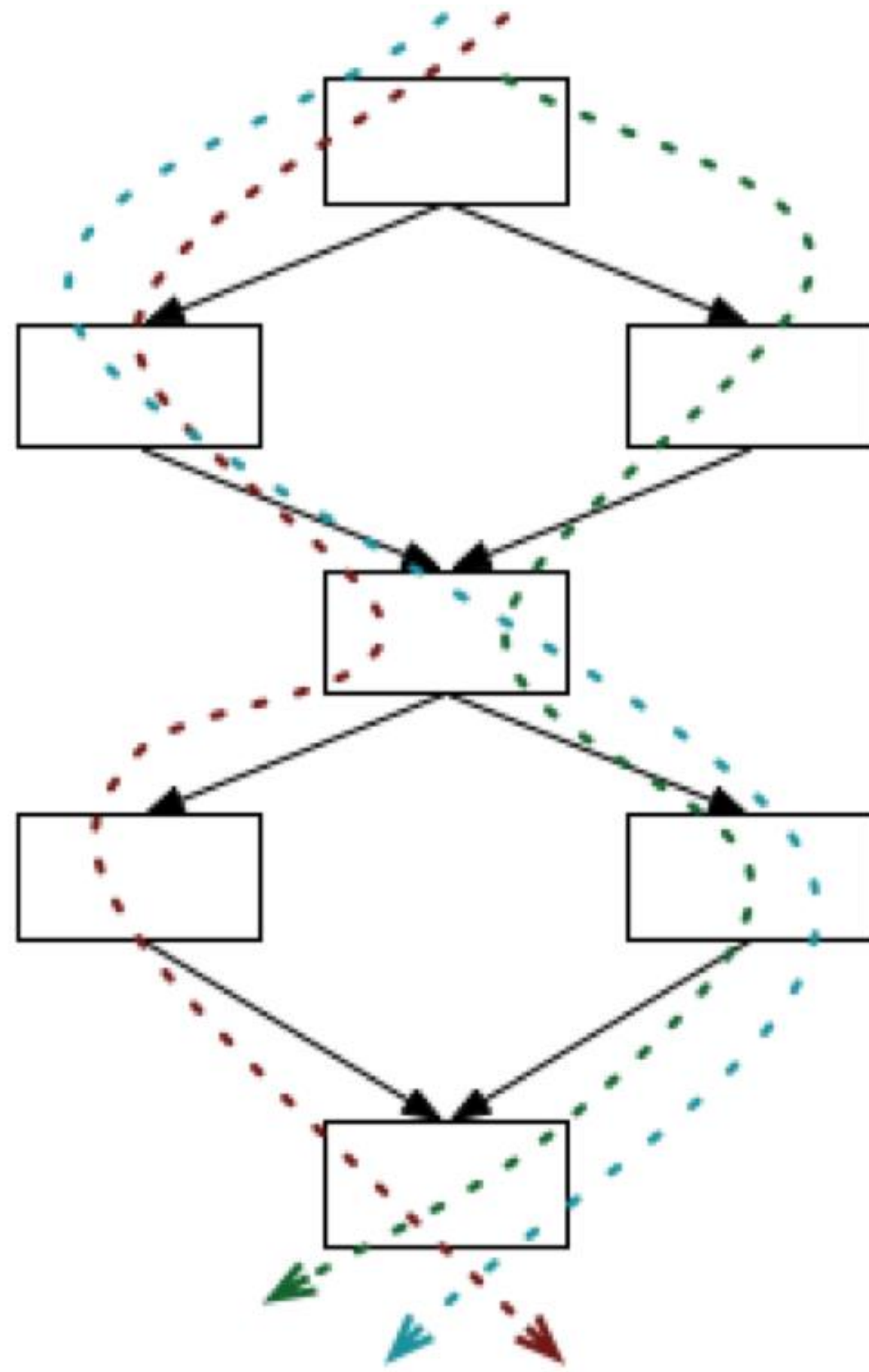
*see example at <https://github.com/philipbeel/example-typescript-nyc-mocha-coverage>

Executing every branch doesn't mean that you've executed every behavior

- In this example, all branches are covered by the test
- (1,22) covers the true branches
- (0,-10) covers the false branches
- BUT: (0,1) makes this function crash

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

Code like this will make path coverage hard to achieve



- n tests might lead to 2^n paths
- Sometimes a fault is only manifest on a particular path, as we saw in the preceding example.
- Worse, the number of paths can be infinite
 - E.g., if there is a loop.
- What to do?

Smarter tools can rule out unreachable paths

```
if (E1()) {A()}  
else {B()};  
if (E2()) {C()}  
else {D()}
```

- Looks like there might be 4 paths: AC AD BC BD
- But maybe not all of these are feasible.
- Depends on the details of what's in E1 and E2.
- Let's say that the path AD leads to an error.
- Crude analysis considers all possibilities.
- Better idea: Is it possible for E1() to be true and E2() to be false?
- Automatic theorem-proving can often show that this is impossible.

The Blue Screen of Death

Eliminated by using SLAM tool (2001-2011)



```
A problem has been detected and windows has been shut down to prevent damage
to your computer.

The problem seems to be caused by the following file: SPCMDCON.SYS

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this stop error screen,
restart your computer. If the screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any windows updates you might need.

If problems continue, disable all recently installed hardware
or software. Disable BIOS memory as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select startup options, and then
select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0x00000000, 0x00000000, 0x00000000, 0x00000000)

*** SPCMDCON.SYS - Address 7617 base at 5000, DateStamp 3d6dd67c
```

<https://cacm.acm.org/magazines/2011/7/109893-a-decade-of-software-model-checking-with-slam/fulltext>

The PL

BUT: The CrowdStrike bug revealed a new path, which went through the kernel, below where previous tools were looking!



<https://cacm.acm.org/magazines/2011/7/109893-a-decade-of-software-model-checking-with-slam/fulltext>

CS 4530: Fundamentals of Software Engineering

Lesson 3.3 Adversarial Testing

Jonathan Bell, Adeel Bhutta, Mitch Wand
Khoury College of Computer Sciences

Adversarial testing is a way of judging whether you have written enough tests.

- It is helpful to think of adversarial testing as a game in which you play against an adversary
- In adversarial 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.

One could, in principle, generate the mutants by hand

- 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 defects you thought to introduce.
- But:
 - Did we introduce realistic defects?
 - Clearly doesn't scale!

In mutation testing, the adversary generates buggy code by making simple changes

```
// 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);  
}
```

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);  
}
```

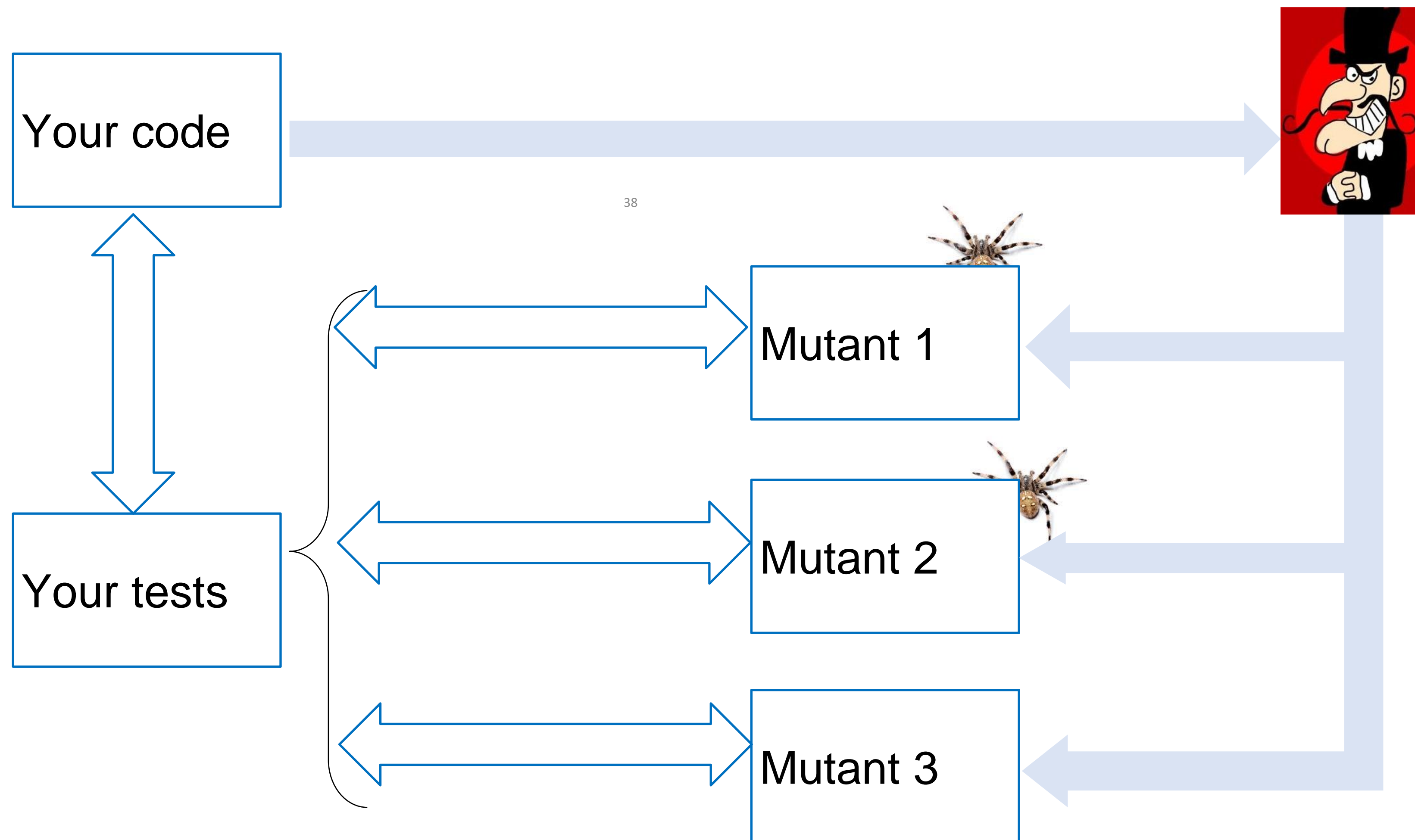
Mutated code (buggy)



The Stryker Game: The Opening

Player (You)

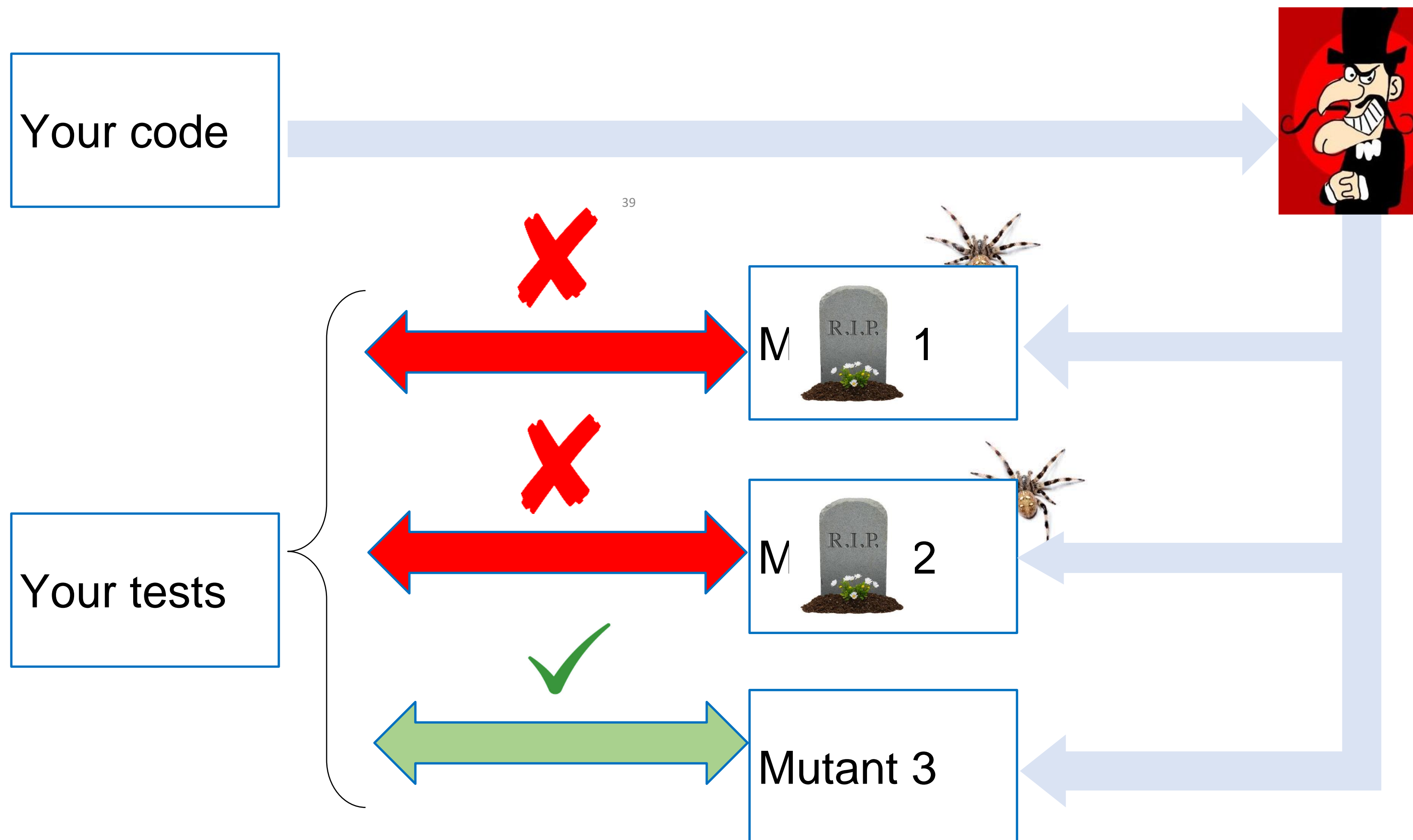
Opponent (Them)



The Stryker Game: Result of one round of play

Player (You)

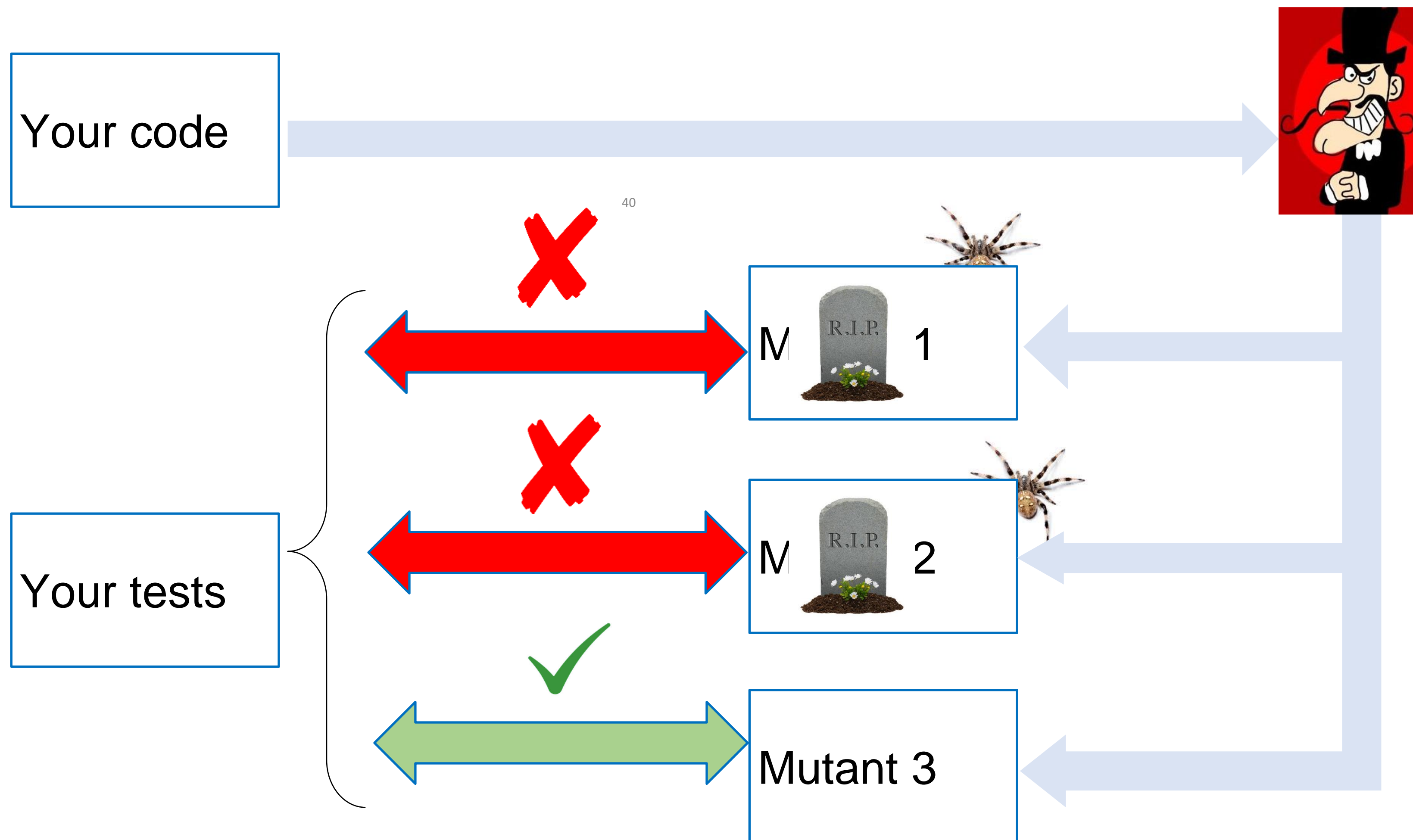
Opponent (Them)



The Stryker Game: Result of one round of play

Player (You)

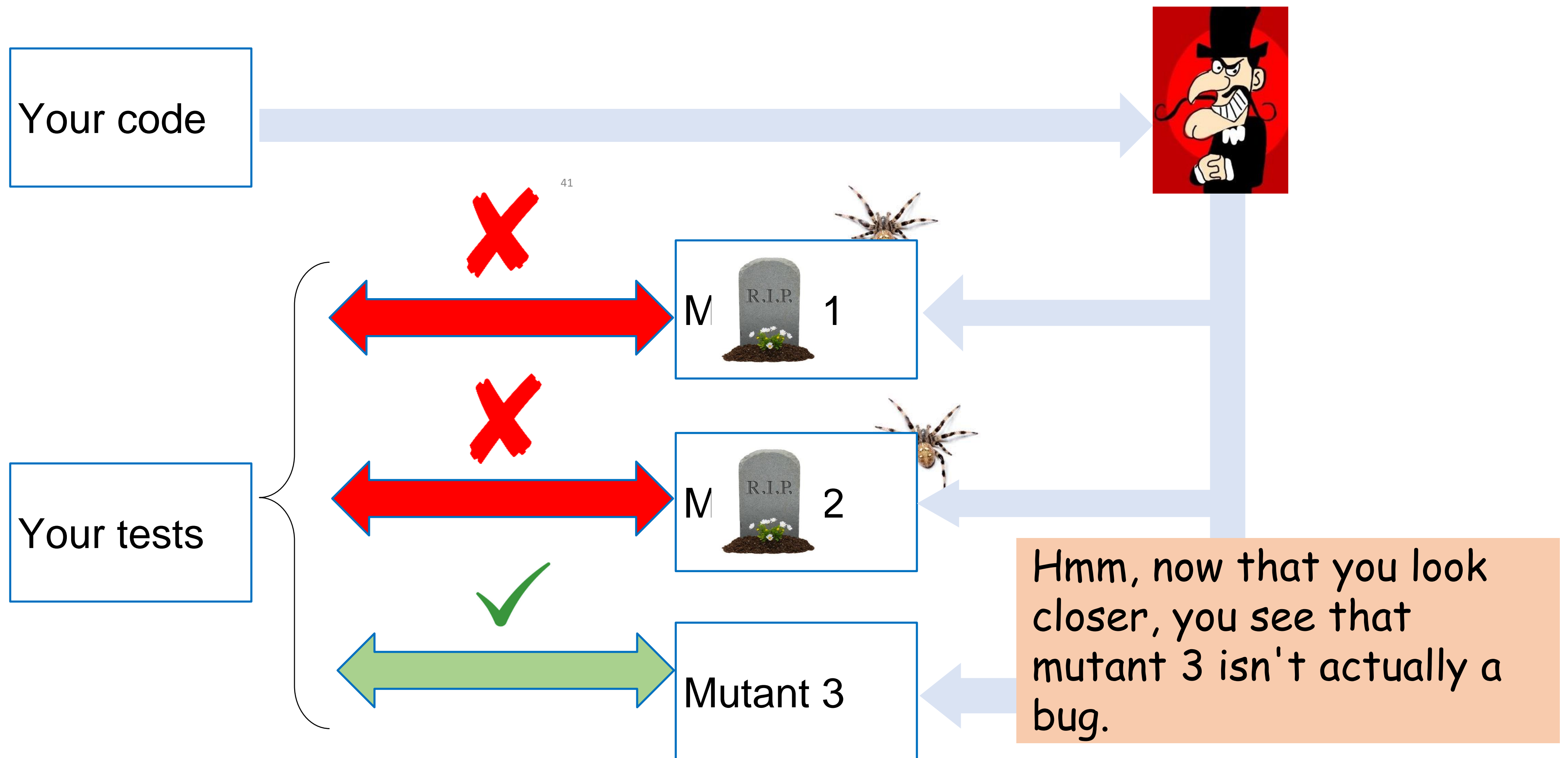
Opponent (Them)



The Stryker Game: a winning position

Player (You)

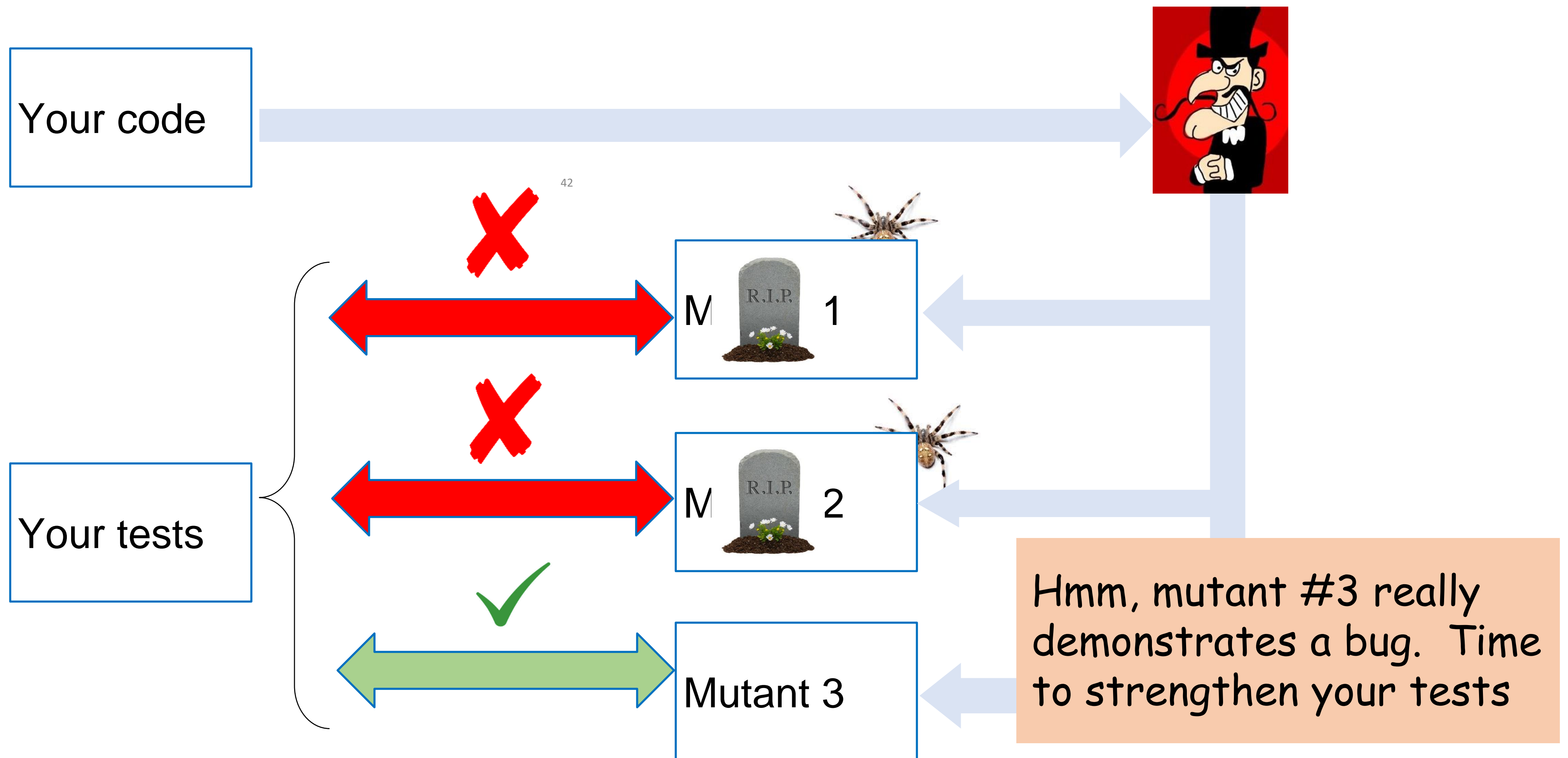
Opponent (Them)



The Stryker Game: a losing position

Player (You)

Opponent (Them)



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

```
// 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

[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);
```

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

```
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

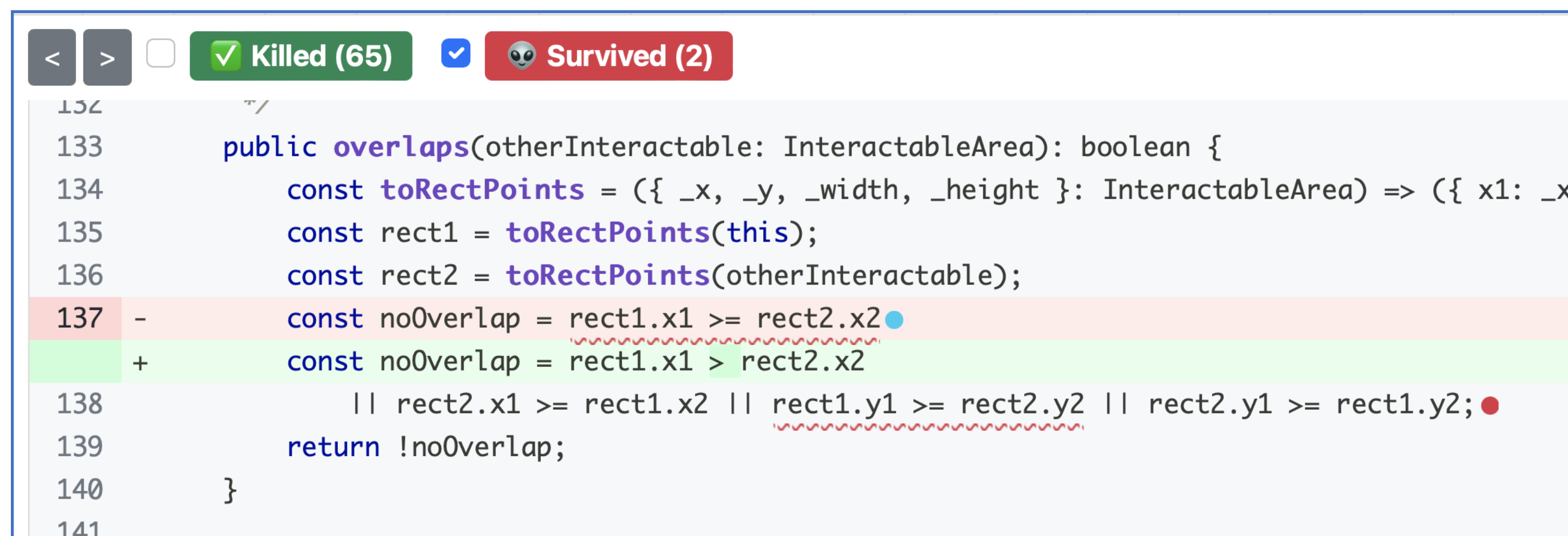
```
-      return list.find((item) => item >= target);  
+      return list.find((item) => item > target);
```


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.



```
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 }
```

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

Undetected Mutants May Not Be Bugs

```
62     public static fromMapObject(mapObject: ITiledMapObject, broadcast: boolean) {
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, width, height };
69     return new ConversationArea({ id: name, occupantsByID: {} }, rect);
70 }
```

- Unfortunately, we can't automatically tell if an undetected mutant is a bug or not
- This mutant is benign: the specification didn't require this particular error message to be generated.
- Testing for this message would be brittle

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
{linozem, 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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

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>

Activity: improving a test suite

- Enhance the test suite of the transcript server to improve line coverage and mutation coverage
- Download from Module 03 webpage

Review

- Now that you have come to the end of this lesson, you should be able to:
 - Explain different reasons why you might want to test
 - Design a TDD test suite by identifying equivalence classes of inputs
 - Explain the following measures of code coverage, and how they differ:
 - Statement or line coverage
 - Branch coverage
 - Path coverage
 - Use mutation testing to judge the completeness of a test suite