CS 4530: Fundamentals of Software Engineering

Module 11: What makes a good test suite?

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:
  - Explain what makes a good test, and give examples and counter examples
  - Explain different things a test suite might accomplish, and sketch how one might judge how well a test suite accomplishes those goals

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 are Hermetic

- Contain all information necessary to set up, execute, and tear down environment
- Leaves no trace of its execution
- Important to reduce *flakiness* - test failures

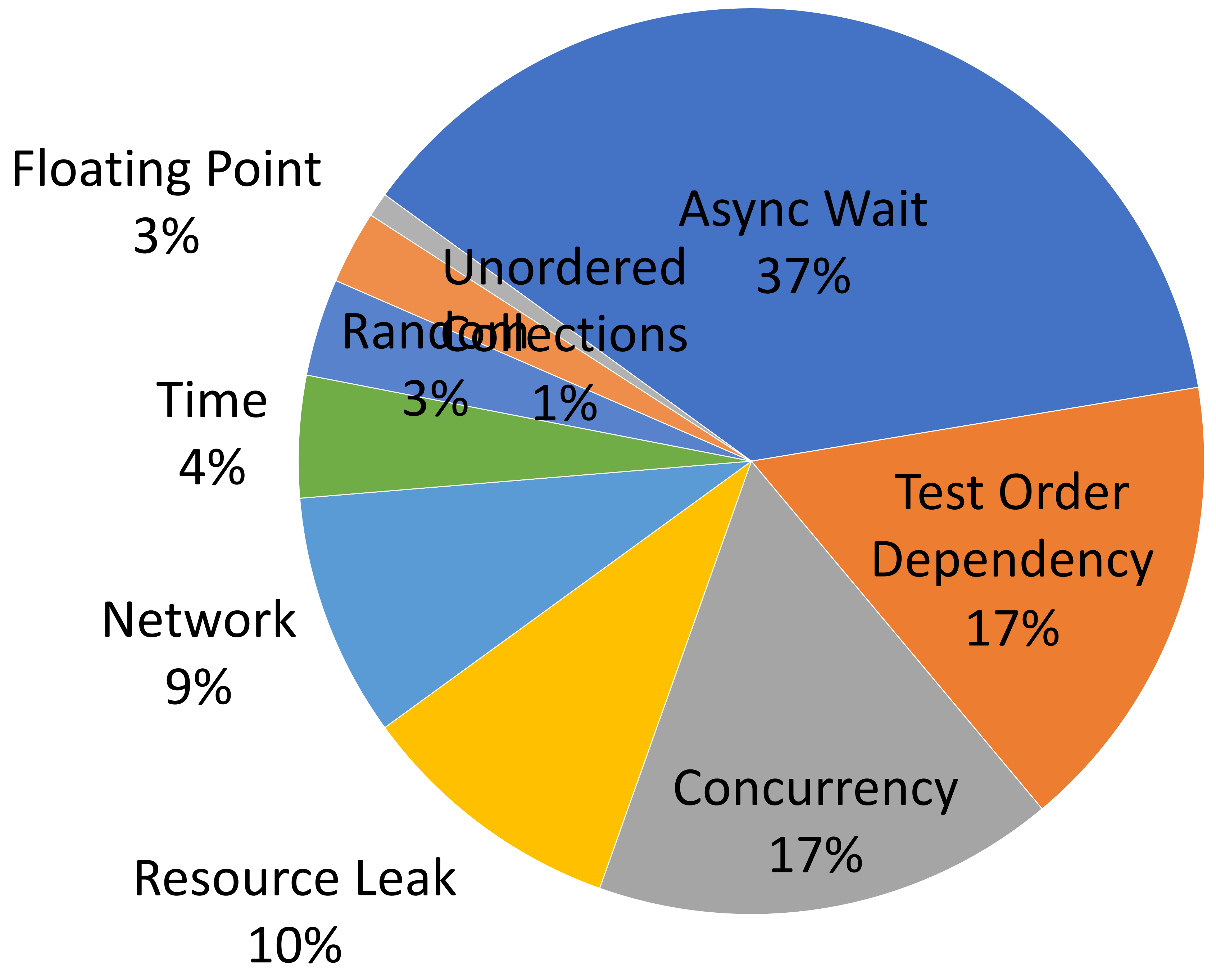
```
describe('Create student', () => {
  it('should return an ID', async () => {
    const createdStudent = await client.addStudent('Avery');
    expect(createdStudent.studentID).toBeGreaterThan(4);
  });
})
```

This test is not hermetic: assumes starting ID of 4, leaves an extra Avery in the application

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

Good Tests Aren't *Flaky*

- Flaky test failures are false alarms
- Tests that are *hermetic* defend against "test order dependency" - failures due to tests running in specific order
- Most common cause of *flaky* test failures: "async wait" - tests that expect some asynchronous action to occur within a timeout
- Good tests avoid relying on timing



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

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
- Example: Asserting that a specific error message is thrown (IP1)

```
it('Throws an error if there is no layer called "objects"', async () => {
  expect(() => town.initializeFromMap(testingMaps.noObjects))
          .toThrowError('There is no layer called "objects"');
});
```

Unless the specification states that this is the error message that should be thrown, this test is brittle

Good Tests are Clear

- Test failures indicate:
  - There is a bug in the system under test, and/or
  - There is a bug in the test
- Clear tests help engineers diagnose the actual problem

```
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);
    }
  }
```

This test is not clear: if it fails, why?

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

Good Tests Invoke Public APIs Only

- Tests should only invoke public methods of SUT (system under test)
- Interact with SUT as a client of the SUT would:
  - Public methods within classes
  - Exported members of modules

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

private _validateInteractables() {
  // Test Me!
}
```

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

What makes a Test Suite good?

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

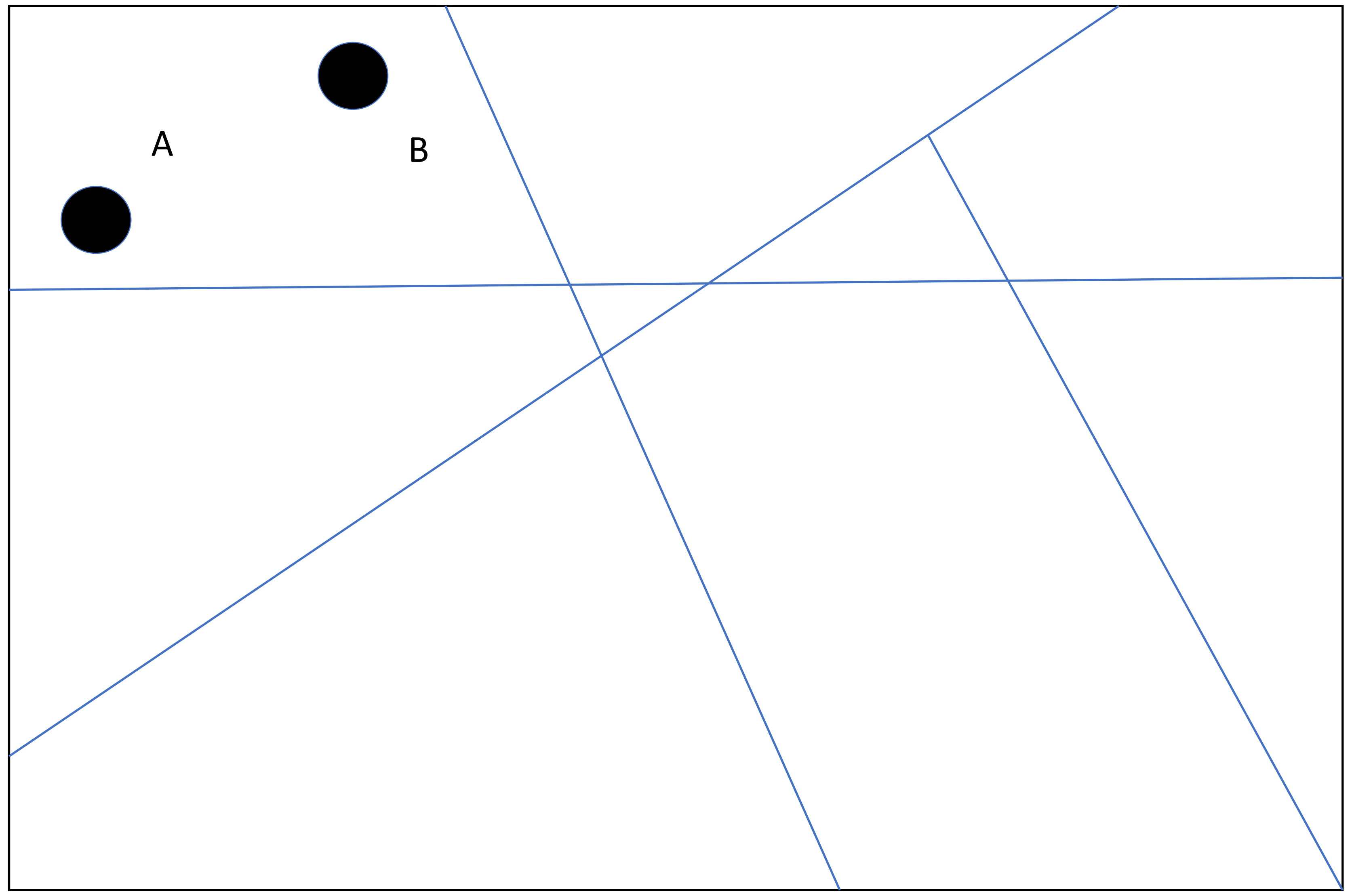
Does the SUT satisfy its specification?

- Test behavior without regard to the implementation ("black-box testing" or "functional testing").
- What's a specification?:
  - A precise definition of all acceptable behaviors of a SUT (outputs, state mutation, other effects) in all situations (state and inputs)
  - A specification may be formal (mathematical), informal (natural language) or implicit ("I know it when I see it").
- A test suite is an approximation to an unwritten specification
  - That's the "T" in TDD
  - Adequacy of test suite is likelihood that an implementation passing all the tests actually fulfills the (unwritten) specification.

Not often seen in the wild

Building Test Suites From Specifications (TDD)

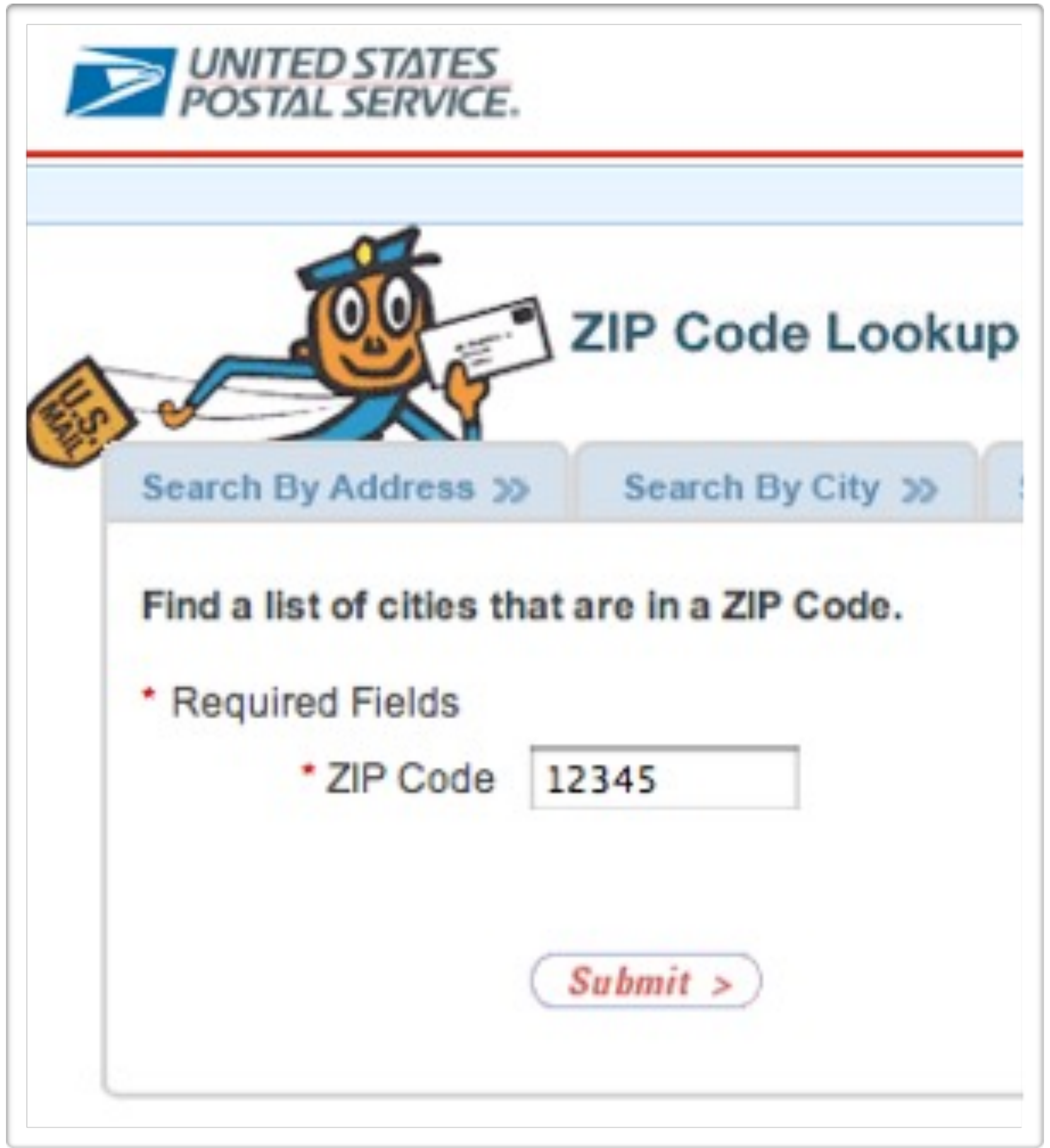
- Enumerate *equivalence* classes of inputs to the SUT, and the expected behavior of that class
- Identify *boundary cases* (near misses between input classes)
- Evaluate the adequacy of the test suite by comparing the tested behaviors with the specified behaviors
- Sometimes referred to as “black box” testing



If the program works for input A, it will probably work for input B

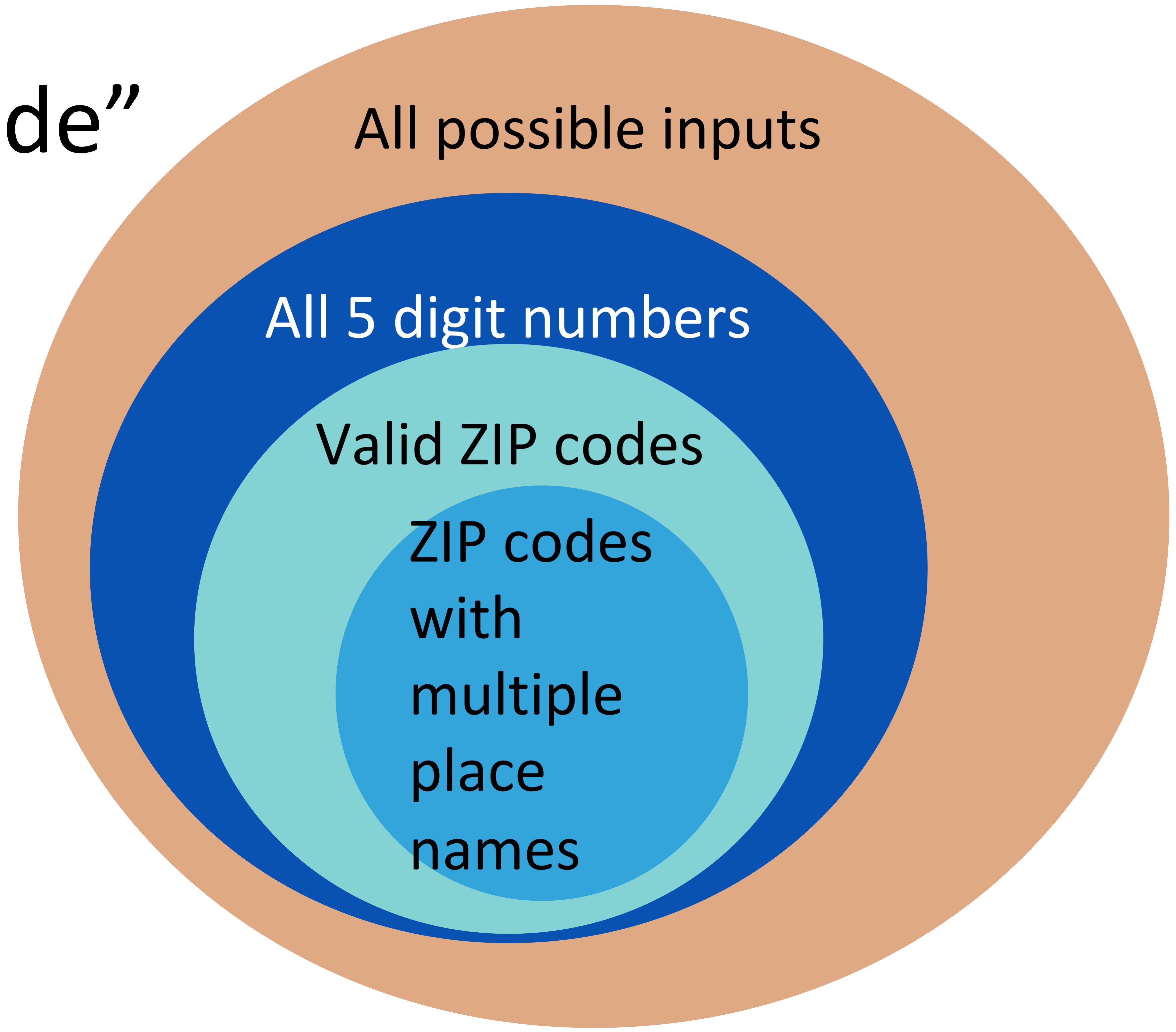
Building Test Suites From Specifications: Zip Code Lookup

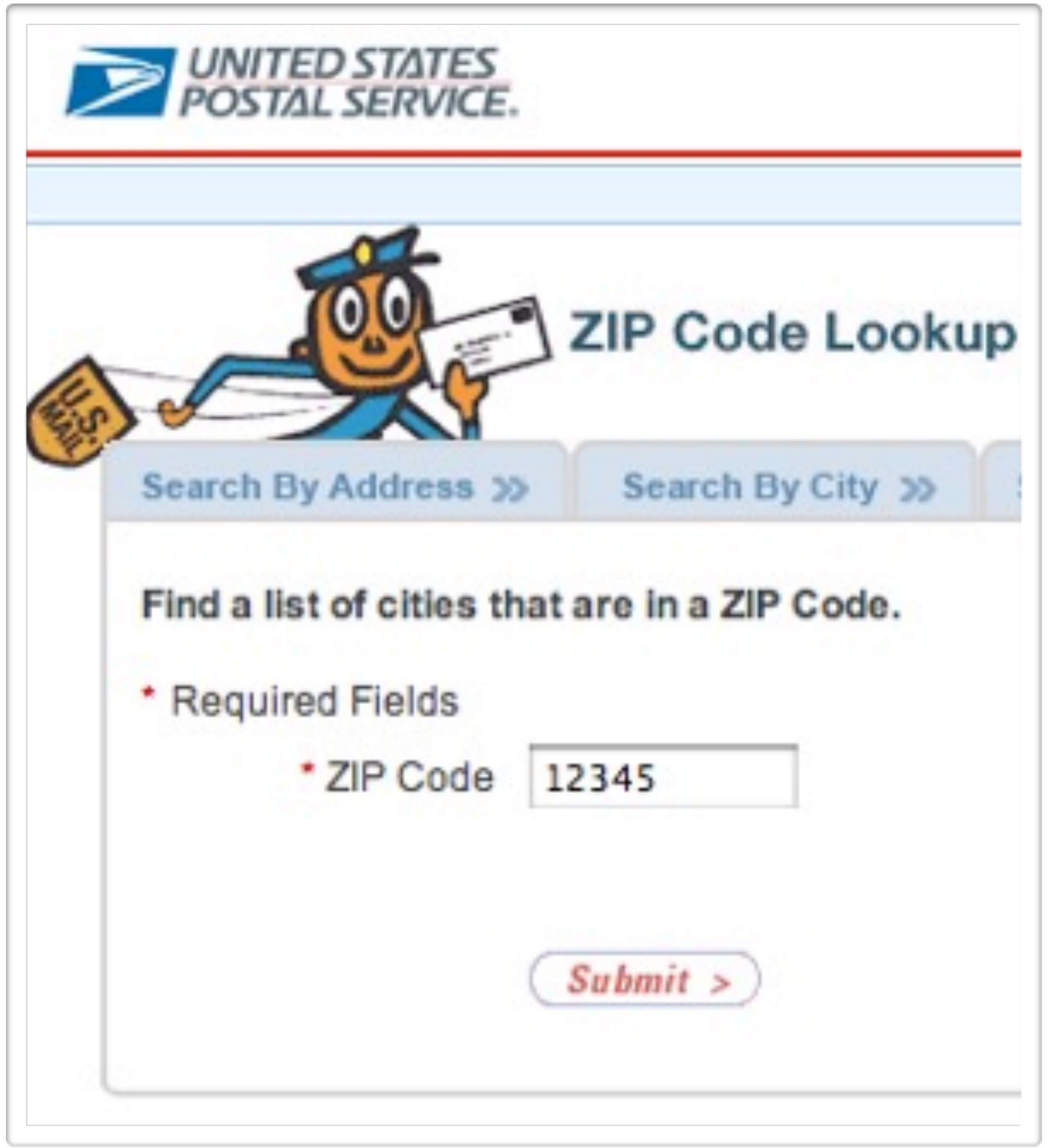
- USPS ZIP code lookup tool accepts a zip code as input, and outputs:
  - The "place names" that correspond to that ZIP code, or
  - "Invalid zip code"
- Strategy:
  - Determine the input equivalence classes, boundary conditions
  - Write tests for those inputs



Building Test Suites From Specifications: Zip Code Lookup

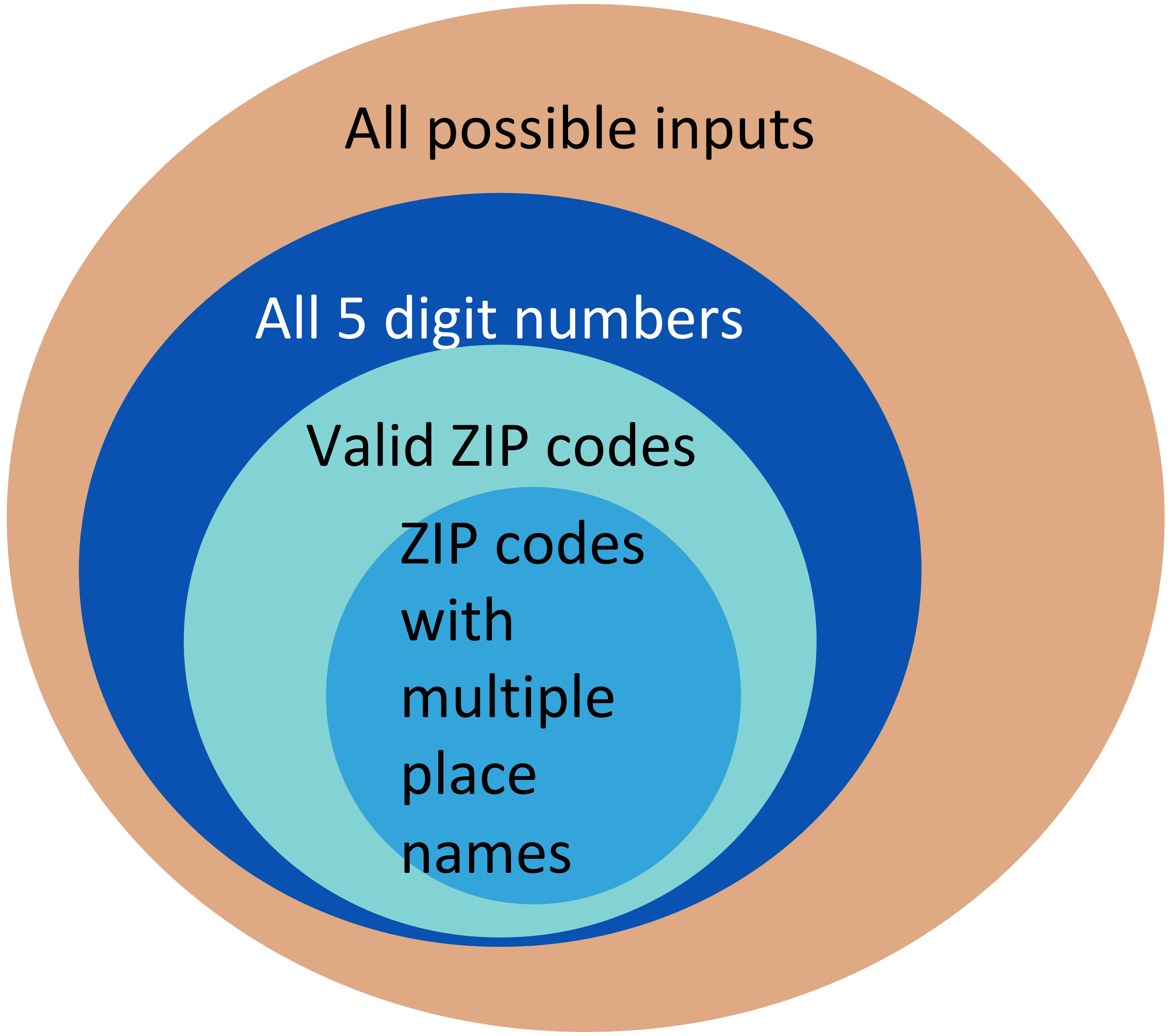
- USPS ZIP code lookup tool accepts a zip code as input, and outputs:
  - The "place names" that correspond to that ZIP code, or
  - "Invalid zip code"





Building Test Suites From Specifications: Zip Code Lookup

- Equivalence classes:
  - 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
- Generate 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



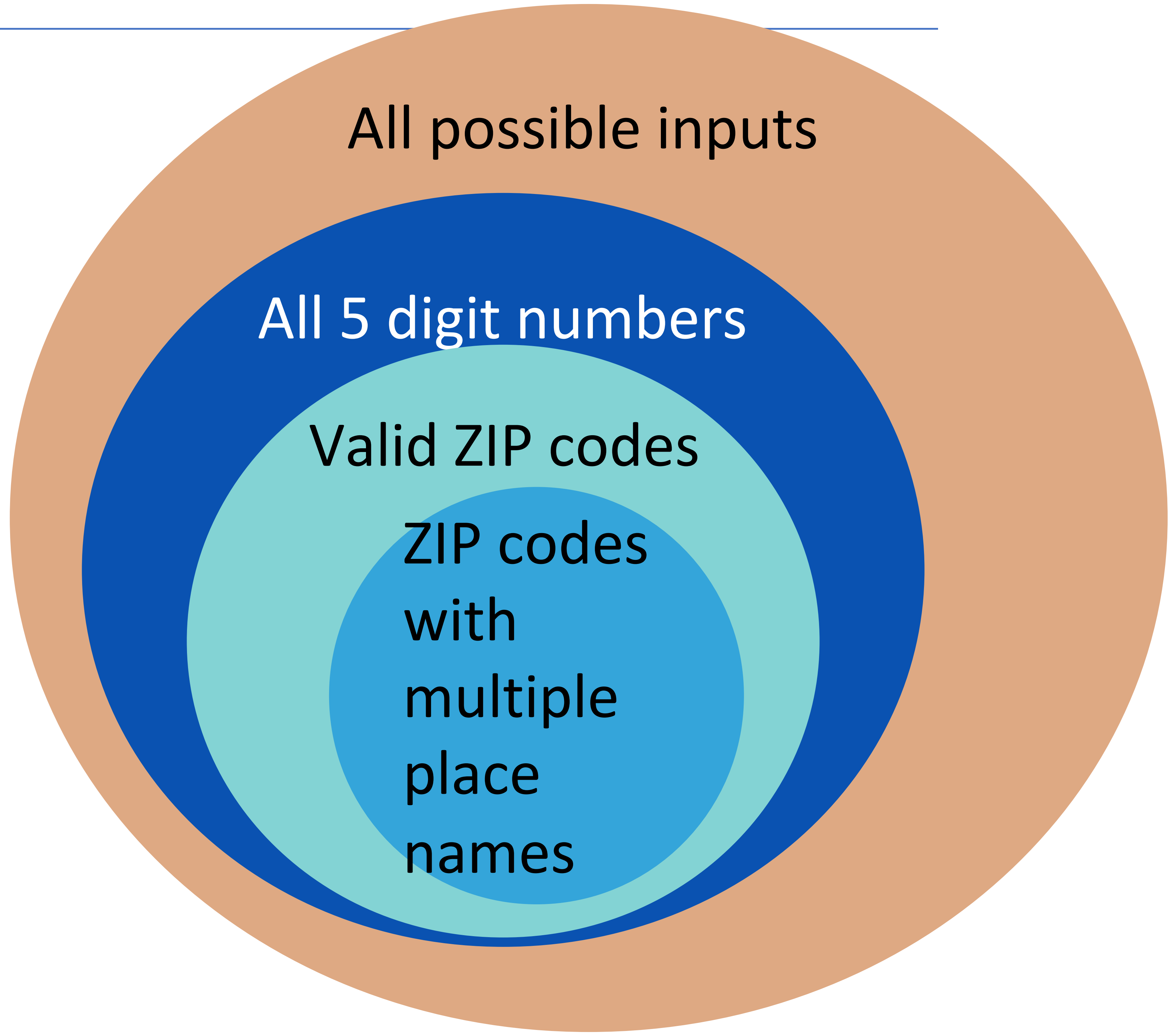
Make sure the regions have the right boundaries.

- Select “special” values of a range
  - Boundary values;
  - Barely legal, barely illegal inputs;

  => ***boundary testing***
- Integer overflow a serious problem: may be implicit
  - ComAir problem due to a list getting more than 32767 elems
- https://arstechnica.com/uncategorized/2004/12/4490-2/

Building Tests from Specifications (TDD)

- 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)



Building Test Suites for Code ("Whitebox" Testing)

- Examine the code of the system under test
- Enumerate all public methods and observable behaviors
- Write tests that execute those methods and check those behaviors
- A "good" test suite executes and checks all of the possible behaviors of our code

```
function getPlaceNames(input: string): string[] {
  try{
    if(input.length == 5) {
      const parsed = parseInt(input);
      if (isValidZipcode(parsed)) {
        const primaryPlaceName = getPrimaryPlaceName(parsed);
        if(hasOtherPlaceNames(parsed)){
          return
[primaryPlaceName].concat(otherPlaceNames(parsed))
        }
        return [primaryPlaceName];
      }
    }
    throw new Error("Invalid zip code")
  }catch(err){
    throw new Error("Invalid zip code")
  }
}
```

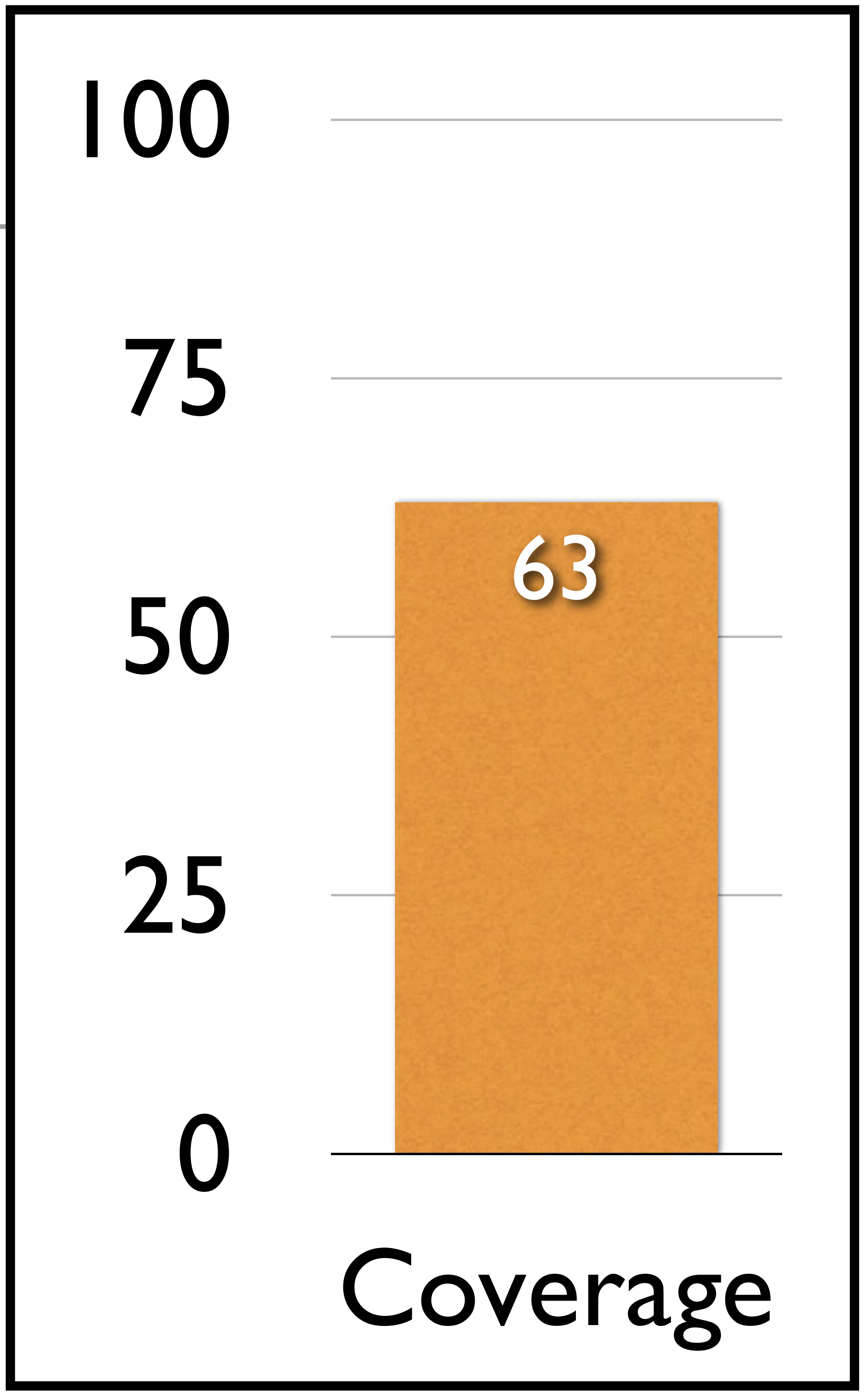
Do our tests execute all of the code?

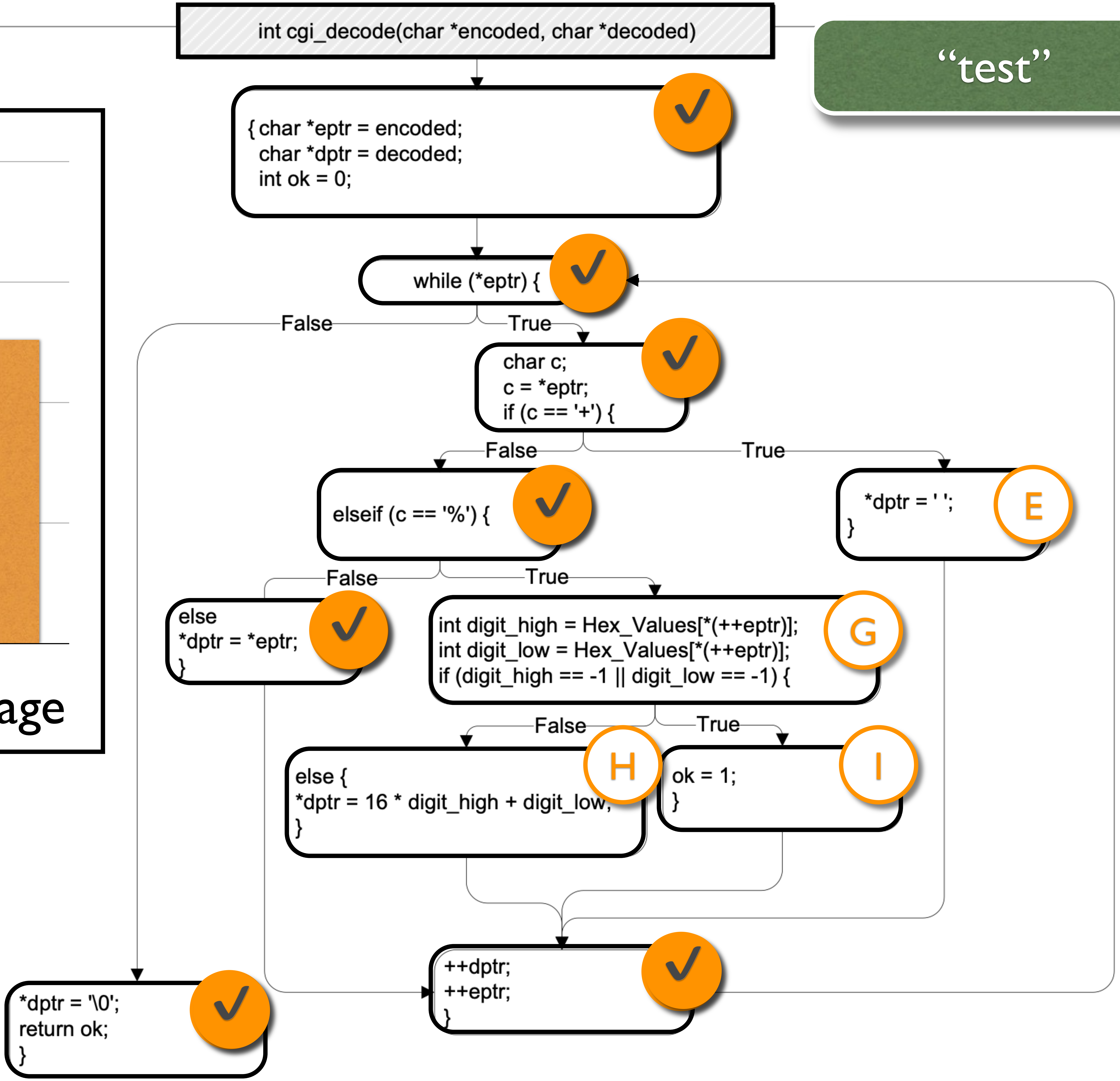
- Idea: Quantitative measure the portion of code executed by test suite. Write new test inputs to execute more code.
- This is the question of test coverage, examples:
  - Statement or Block coverage
  - Branch coverage
  - Path coverage
- If some (statement/branch/path) is not covered, it is definitely not tested
- If some (statement/branch path) is covered, it might be tested

Statement Coverage

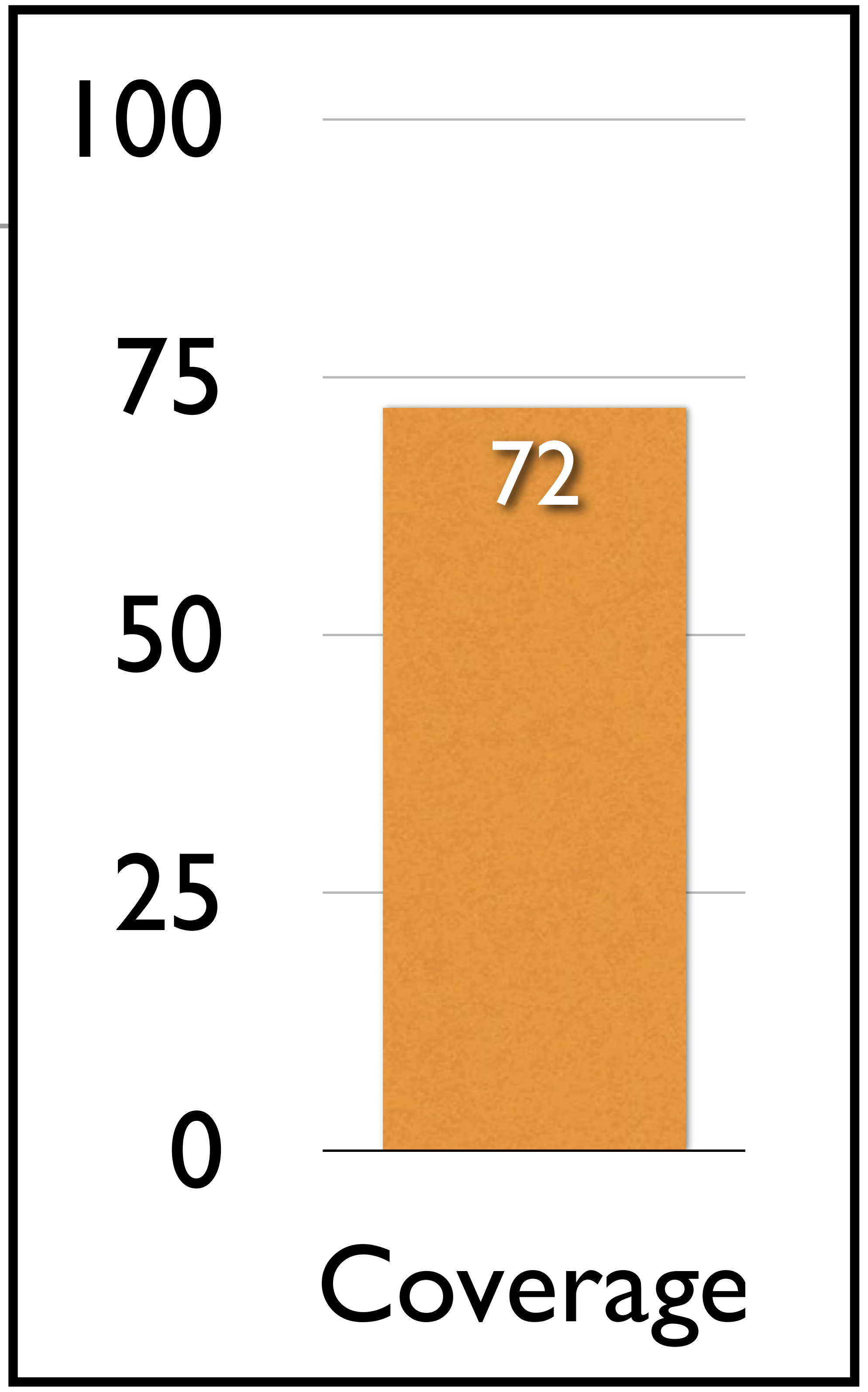
- Each line (or part of) the code should be executed at least once in the test suite
- There are good tools for measuring how many lines were executed or not executed
  - Jest -- coverage
- Adequacy criterion: *each statement must be executed at least once*

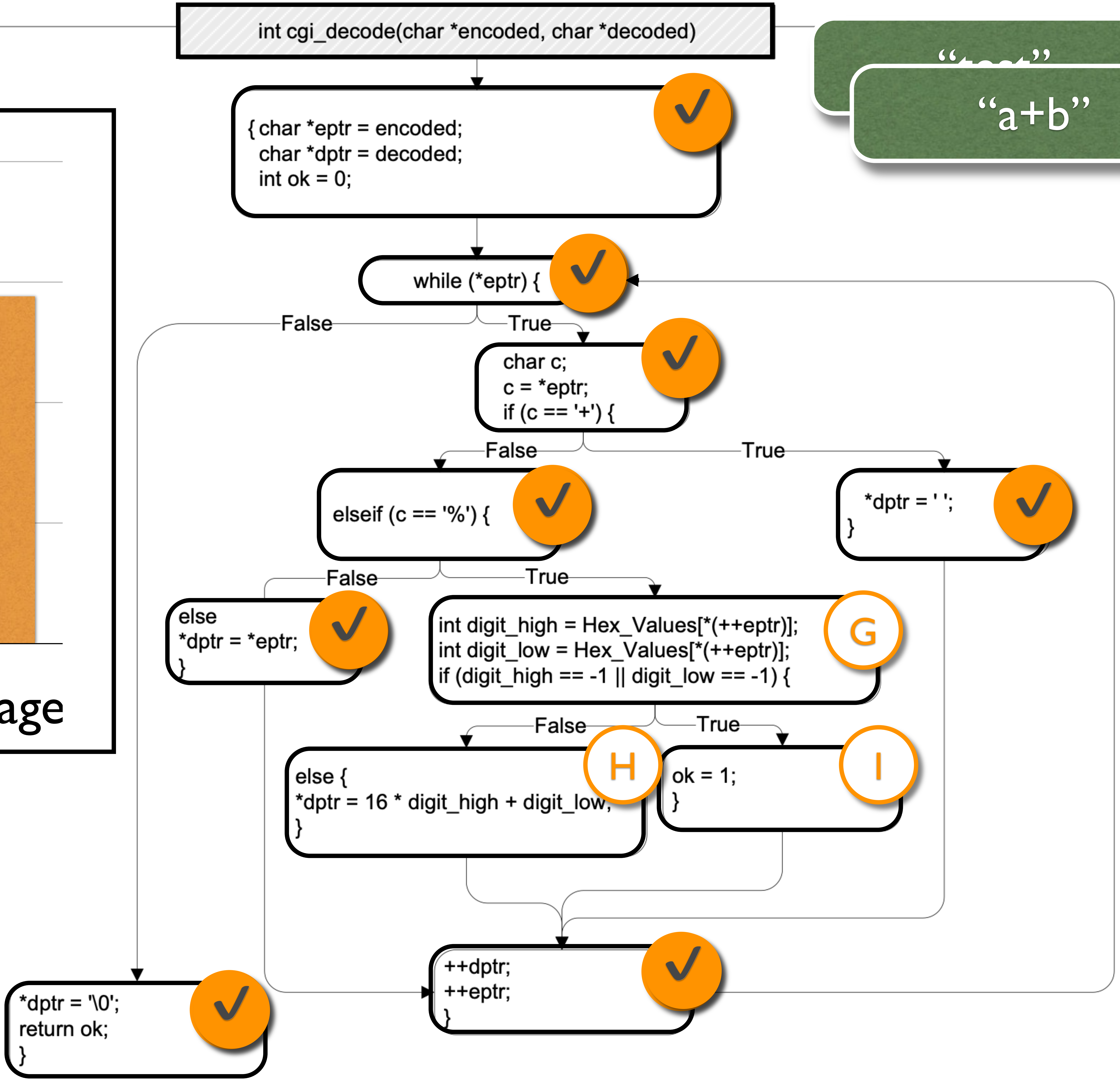
$$\textit{Coverage:} \quad \frac{\#\ \textit{executed statements}}{\#\ \textit{statements}}$$


100
75
50
25
0
63
Coverage

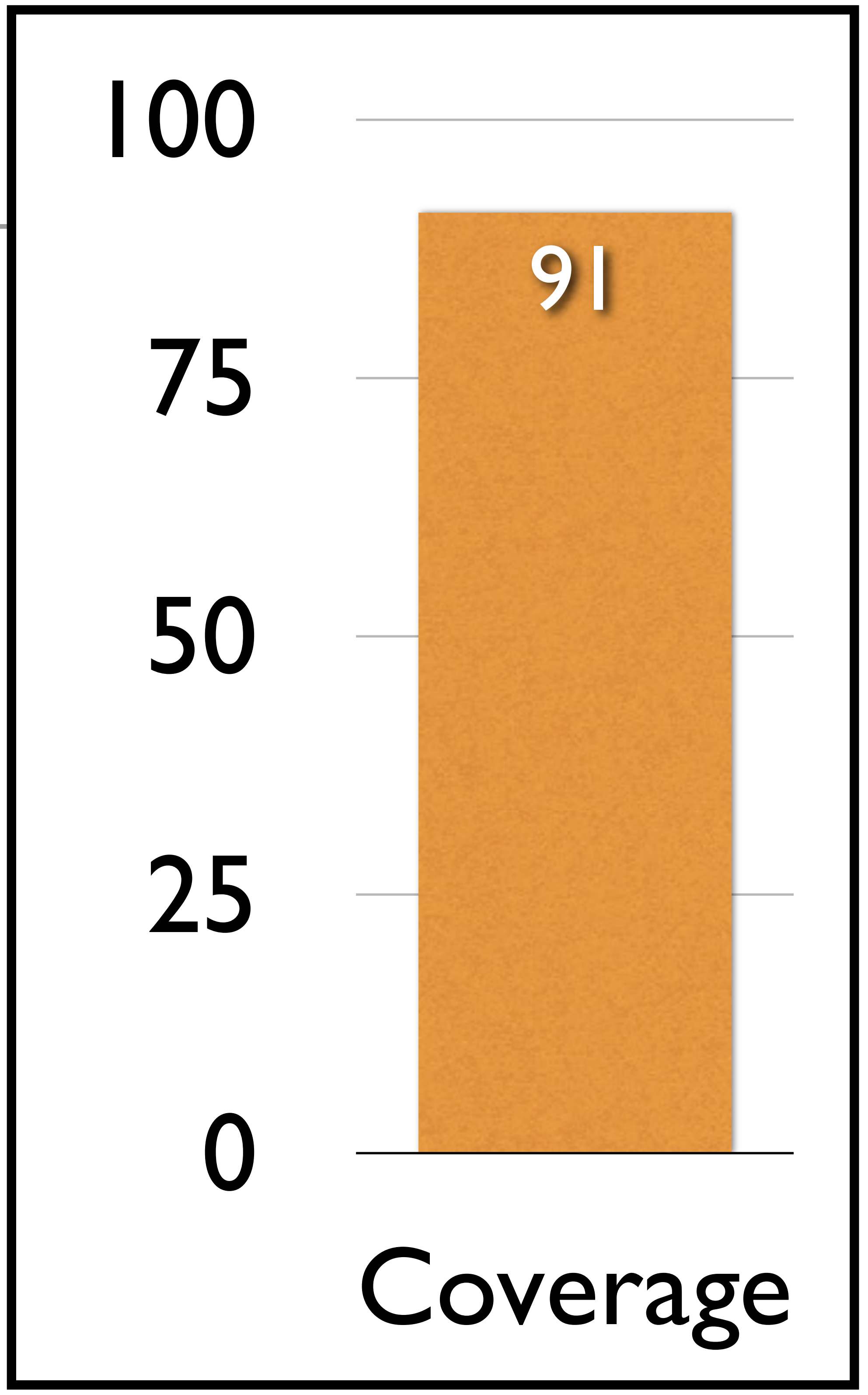

int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}

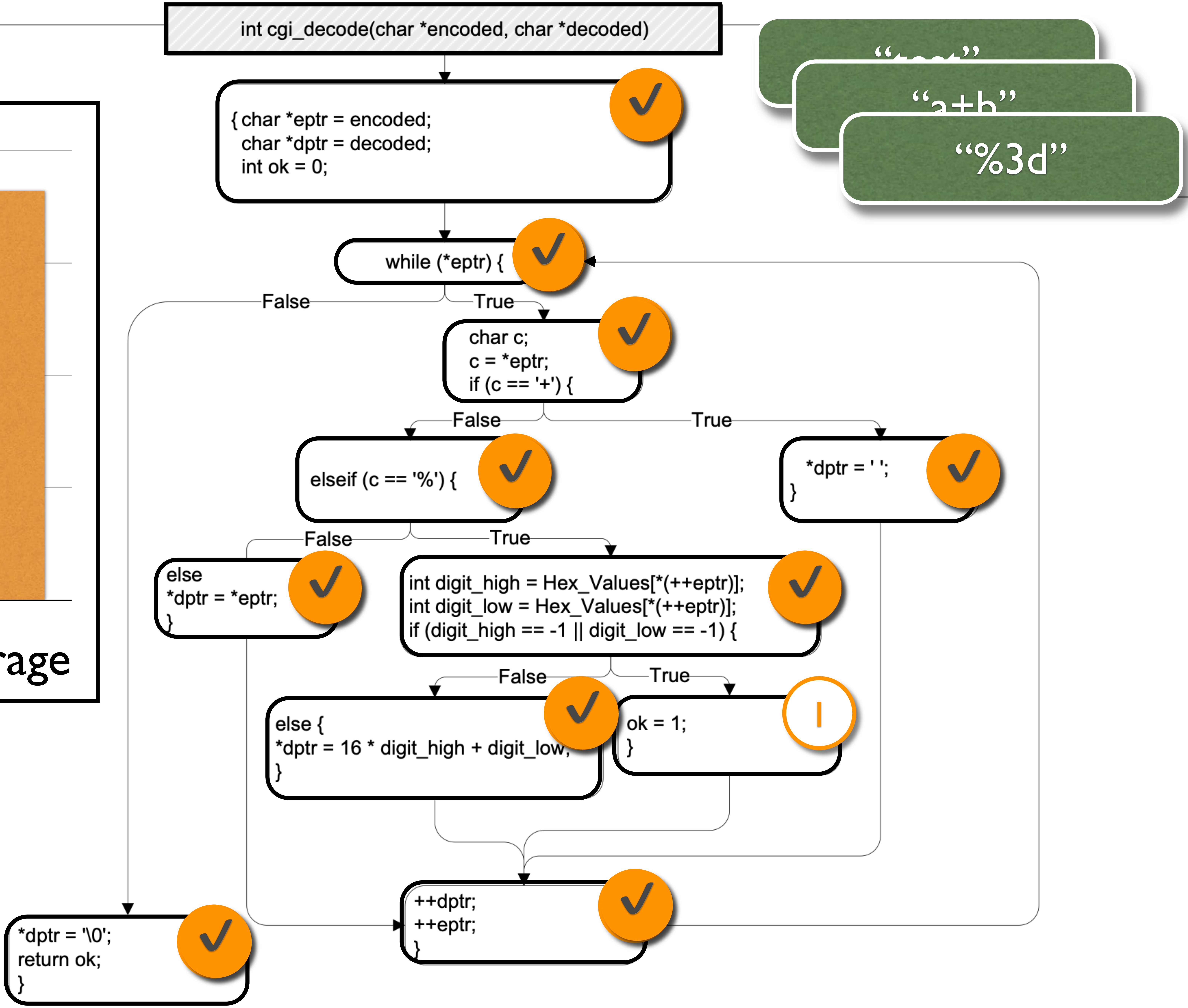
“test”

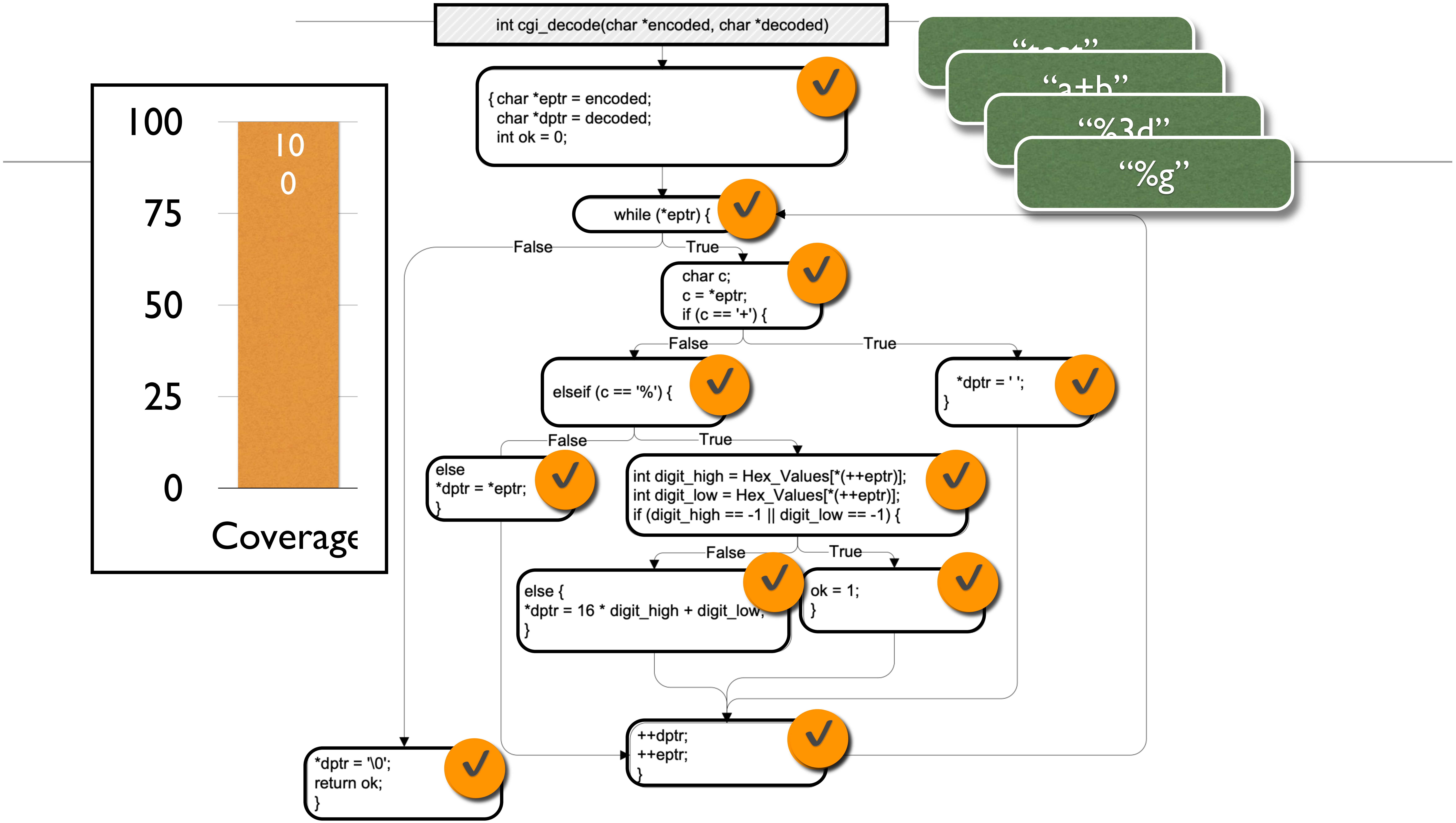

100
75
50
25
0
72
Coverage


int cgi_decode(char *encoded, char *decoded)
{char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}

“a+b”


100
75
50
25
0
91
Coverage


int cgi_decode(char *encoded, char *decoded)
{char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
1
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}
“%3d”



int cgi_decode(char *encoded, char *decoded)
{char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}
100
75
50
25
0
100
Coverage
"a+b"
"%g"

Branch Coverage

- Adequacy criterion: *each branch in the CFG must be executed at least once*

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

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

- Most widely used criterion in industry

Branch Coverage Measures

- Coverage is computed automatically while the tests execute
- jest --coverage
  - Does it all for you

```
  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

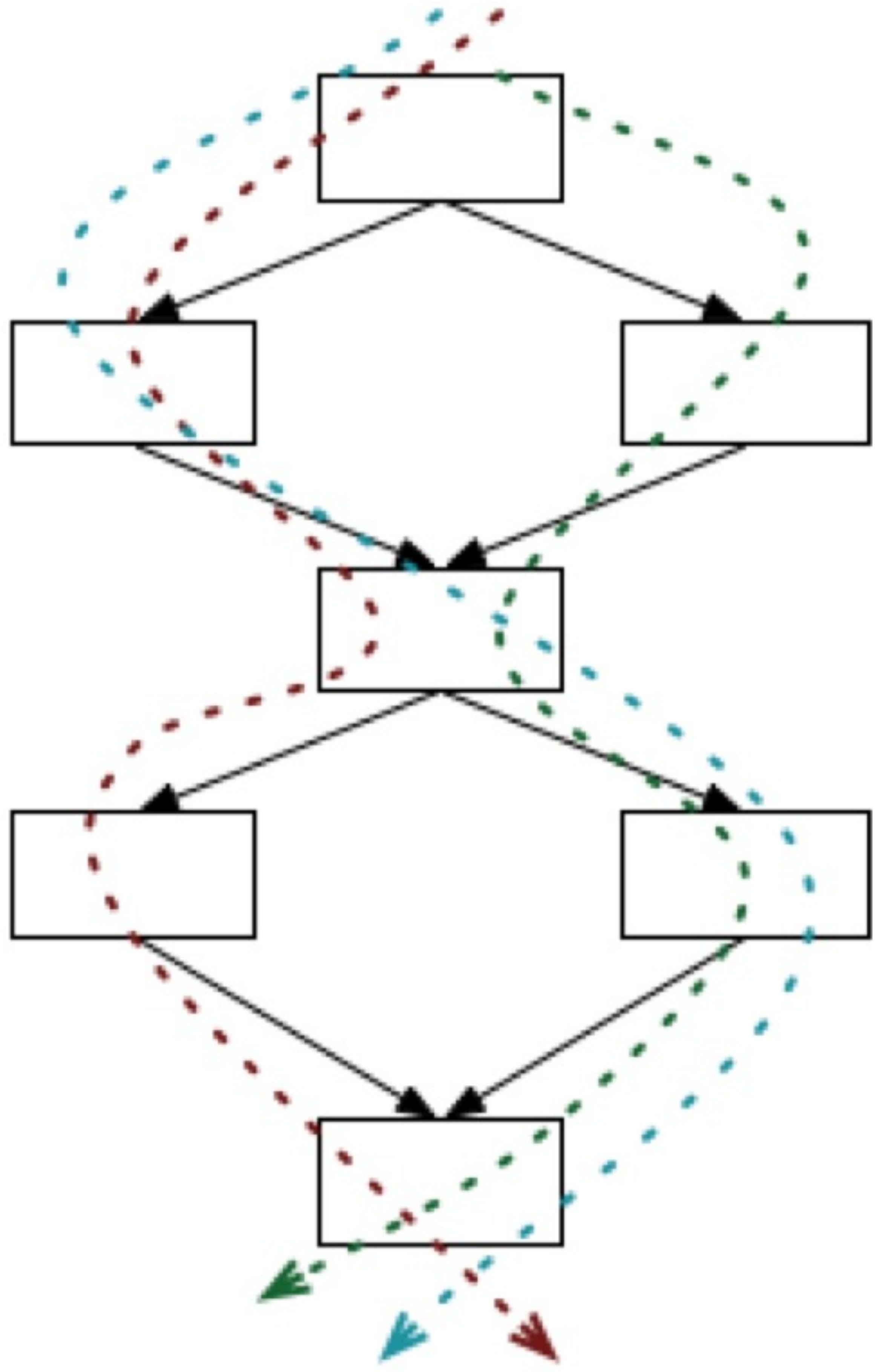
Every Branch Executed != Every Behavior Executed

- 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;`
- 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)

Pareto’s Law

Approximately 80% of defects come from 20% of modules

Good Tests have Strong Oracles

- Test oracle defines criteria for when test should fail
- Strong oracles check all observable behaviors and side-effects
- How to determine an oracle?
  - 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)

How to evaluate the strength of test oracles?

- Goal: “A good test suite finds all of the bugs”
- Problem: How to know the bugs that we could make?
- 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 ‘Contains” in IP1

```
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” in IP1

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)

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

< > ✅ Killed (65) 👽 Survived (2)

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

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

```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 this check for undefined, an error is still thrown when the undefined layer is dereferenced on the following line

```diff
  public static fromMapObject(mapObject: ITiledMapObject, broadca
      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 though the error message changed, the specification doesn't indicate what error message should be thrown.

Mutants are a Valid Substitute for Real Faults

- 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've studied this lesson, you should be able to:
  - Explain some properties of good tests.
  - Explain different things a test suite might accomplish, and sketch how one might judge how well a test suite accomplishes those goals