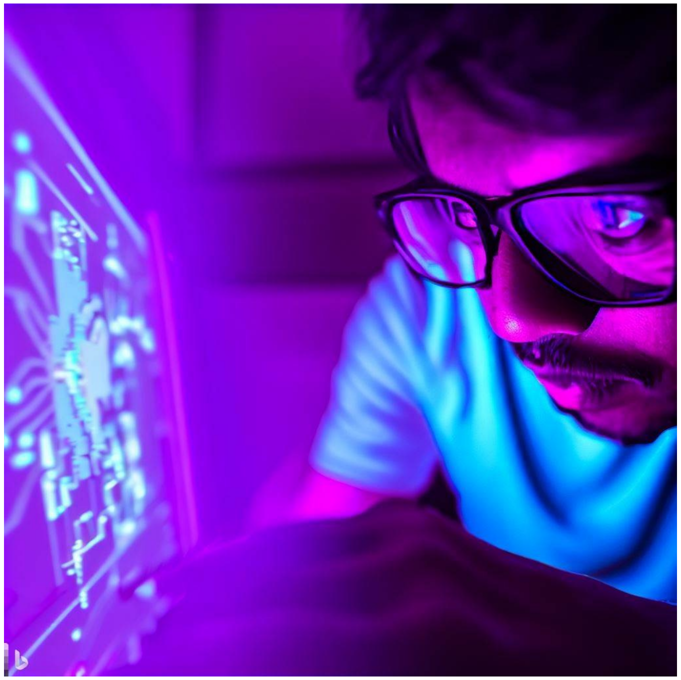


CSE 331

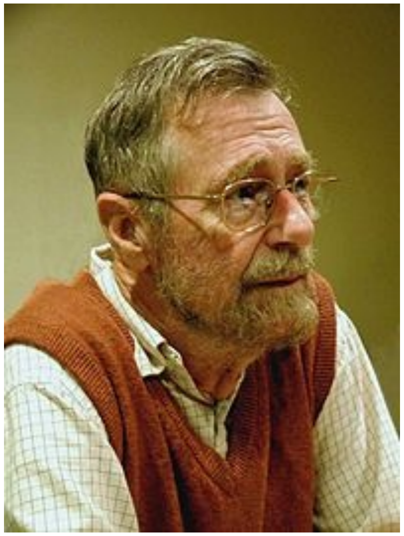
Testing

Kevin Zatloukal

What Can We Learn From Testing?

“Program testing can be used to show the presence of bugs, but never to show their absence!”

Edsgar Dijkstra
Notes on Structured Programming, 1970



“Beware of bugs in the above code;
I have only proved it correct, not tried it.”

Donald Knuth, 1977

Unit vs Integration Tests

- A unit test checks one component
  - ideally, without testing anything else (not always possible)

- You will be expected to write unit tests in industry

- There are also integration tests and end-to-end tests
  - someone will write them, but maybe not you

- We will focus on unit testing in this course

“Manual” vs Programmatic Tests

- Usually possible to run the code by hand (“manually”)
  - open it in node and execute it
  - open it in the browser and look at it (UI)
- No downside… unless the code changes
  - then, you need to do the tests again
- For some code (UI especially), manual is still easier
  - if written tests are 3x as hard to create,
then you’re better off unless you change it 3+ times
  - for UI, written tests aren’t perfect anyway

    need to see it in the browser to be sure that it looks right

Writing a Test

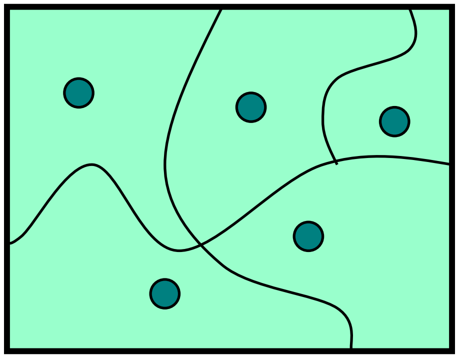
1. Choose an input / configuration
   - description of the inputs / configuration is the “test case”

2. **Think** through what the answer should be
   - if you run the code to get the answer, you’re not really testing

3. Write code that
   - calls the function that input
   - compares the actual answer to the expected one
   - useful libraries that do this

     we will use “mocha” in JS / TS

Key Problem



- Key question is what cases to test
  - at level -1, we can test all of them
  - at level 0+, we cannot
- Split the allowed inputs into **subdomains**
  - for inputs in one subdomain, code “does the same thing”
- Hope: code is entirely right or wrong for subdomain
  - one example in the subdomain will tell us if there is a bug
  - (note: this is not *always* true… see sec02 and HW2)
- Plan: Look at the code. See when it “does the same thing”

Need to Look At the Code

```
// Returns true iff n is a prime number
const isPrime = (n: number): boolean => { … }
```

- How about if we test 2, 3, 4, … 12?
  - seems okay?

Need to Look At the Code

```
// Returns true iff n is a prime number
const isPrime = (n: number): boolean => {
  if (n < 100) {
    return PRIME_CACHE[n];  // precomputed answers
  } else {
    for (let k = 2; k*k <= n; k++) {
      if (n % k === 0)
        return false;
    }
    return true;
  }
};
```

Need to Look At the Code

```
// Returns true iff n is a prime number
const isPrime = (n: number): boolean => {
  if (n < 100) {
    return PRIME_CACHE[n];
  } else {
    …
  }
}
```

- Cases 2, 3, 4, … 12 are just table lookups!

Primary Heuristic: Clear-Box Testing

- We need to look at the code to know what to test
  - this will be our **primary** heuristic
- In this class, I want a clear rule for how many tests
  - want homework and tests to have clear right/wrong answers
- Outside of class, these tests are also good
  - but other programmers may not use the same rules

Testing Straight-Line Code

Straight-line Code looks like

```
return 2 * (n-1) + 1;
```

Or, more generally, like this

```
const m = n - 1;
return 2 * m + 1;
```

- Any number of constant values allowed
  - often makes the code easier to read, but no different

- Inputs where it executes the same straight-line code are "***doing the same thing***"

Testing Straight-Line Code

Rule: same straight-line code is one subdomain

Straight-line Code looks like

```
return 2 * (n-1) + 1;
```

Or, more generally, like this

```
const m = n - 1;
return 2 * m + 1;
```

Testing Subdomains

Rule: at least two test cases per subdomain

(assuming subdomain contains at least two inputs)

- My main worry is copy-and-paste issues
  - copy “return 1;” and forget to change it later
  - if the test we pick happens to want 1, we’ll never notice
- Still doesn’t guarantee the code is right! (see HW2)
- More is obviously also okay
  - not a contest to write the fewest tests

Testing Function Calls

In general, function calls are still straight-line code

```
const m = n - 1;
return Math.sin(2 * m + 1);
```

- All inputs are still are “the same”
  - two cases is still enough
- Exception: recursive calls
  - we will test these differently (more later)
- (Unusual cases can require multiple subdomains
  - shouldn’t arise in this class)

Testing Conditionals

Conditionals look like this

```
if (n > 0) {
  return 2 * (n-1) + 1;
} else {
  return 0;
}
```

Two branches (“then” and “else”)

- in this case, both branches are straight-line code

Testing Conditionals

Rule: branches are in separate subdomains

- Would be **negligent** not to test both branches
- If both are straight-line code, then 4 tests
- With if/else if/else, we'd need 6 tests
  - 3 branches x 2 per straight-line block = 6 cases

Other Heuristics

Some other heuristics are also useful

- **<u>Boundary Cases</u>**: if n and n+1 are separated, then make sure you test n and n+1
  - easy to have "off by one" bugs
  - happens if you use "< n" instead of "<= n"

    behavior changes between n-1 and n instead

    (see John Carmack's tweet!)

- Often doesn't require any more tests
  - can be one of two cases for straight-line code

Testing Conditionals

Conditionals look like this (with `n` an integer)

```
if (n > 0) {
  return 2 * (n-1) + 1;
} else {
  return 0;
}
```

- Boundary cases are 0 and 1
  - cases for “then” block could be 1 and 10 (say)
  - cases for “else” block could be 0 and -1 (say)

Testing Subdomains

Another rule for subdomains

Rule: test each boundary case and
at least one non-boundary case

- If there are no boundaries, test two non-boundary
- If there is one boundary, then test it and one non-boundary
- If there are two boundaries, then test both and one non-boundary
  - e.g., if branch is executed for x between 3 and 10
  - 3 tests are now necessary (e.g., 3, 6, and 10)

Testing Recursion

Recursive calls are more complicated

```
const f = (n: number): number => { // n must be int
  if (n >= 2) {
    const m = Math.floor(n / 2);   // int division
    return 2 * f(m) + 1;
  } else {
    return 0;
  }
}
```

- Heuristics thus far would allow 0, 1, 2, 3
  - only tests 0 or 1 recursive calls
  - not enough! (see sec02)

Testing Recursion

Clear-box Testing for recursive calls:

Rule: inputs that cause 0, 1, and 2+ recursive calls are in separate subdomains

- Call this the “0–1–many” heuristic
- Split into 3 subdomains, then apply other rules
  - if subdomains run the same straight-line code, then 6 tests
  - if “0 recursive calls” has two branches, then 8 tests
  - if a subdomain has only one input, then just one test

    e.g., “0” is in its own subdomain, that’s just one test

Summary of Heuristics

- Split into subdomains where code is different
  - branches of conditionals
  - 0, 1, many recursive calls
- At least two tests per subdomain

  (unless subdomain is only 1 input)
  - include all boundaries and a non-boundary
- Not a contest to write the fewest tests!

Summary of Heuristics

- Continue splitting until no more splits needed
  - e.g., two inputs that both make 0 recursive calls BUT are in separate branches… are in separate subdomains
- For “2+ recursive calls”, look at first two calls
  - different paths are split into separate subdomains
  - e.g., same branch on first call but different on second
- Complete summary in the notes on website

Other Heuristics

Not required for 331 but useful in practice:

- Make sure every argument value is changed
- Look at special values
  - null, undefined, NaN, empty array, etc. often have bugs
- Look at the specification for branches
  - maybe the code doesn't split inputs where it should!
  - e.g., spec splits into "if $x \geq 0$" but code is "`if (x > 0)`"

Example 1

```
// n must be a non-negative integer
const f = (n: number): number => {
   if (n === 0) {
     return 0;
   } else {
     return Math.sin(Math.PI * (n + 0.5));
   }
 }
```

How many tests? Which ones?

- 0 (top branch) and 1, 5 (bottom branch)

Example 2

```
// n must be a non-negative integer
const f = (n: number): number => {
   if (n < 3) {
     return 0;
   } else if (n < 10) {
     return (n - 3) / 10;
   } else {
     return 1;
   }
 }
```

How many tests? Which ones?

- – 0, 1, 2 (top) and 3, 6, 9 (middle) and 10, 12 (bottom)
- – note that 0 is also a boundary case

Example 3

```
// n must be a positive integer
const f = (n: number): number => {
   if (n === 1) {
     return 0;
   } else {
     return 1 + f(1 + Math.floor((n - 2) / 2));
   }
 }
```

How many tests? Which ones?

- 1 (0 recursive calls)
- 2, 3 (1 recursive call)
- 4, 10 (many recursive calls)

Example 4

```
// n must be an integer between 1 and 10
const f = (n: number): number => {
   if (n === 1) {
     return 0;
   } else {
     return 1 + 2 * f(n - 1);
   }
 }
```

How many tests? Which ones?

- This is Level -1, so all of them

What Else?

- We only have rules for:
  - straight-line code
  - conditionals (“if” statements)
  - recursion

- What about everything else?

- Without mutation, this is all we need
  - loops require mutation