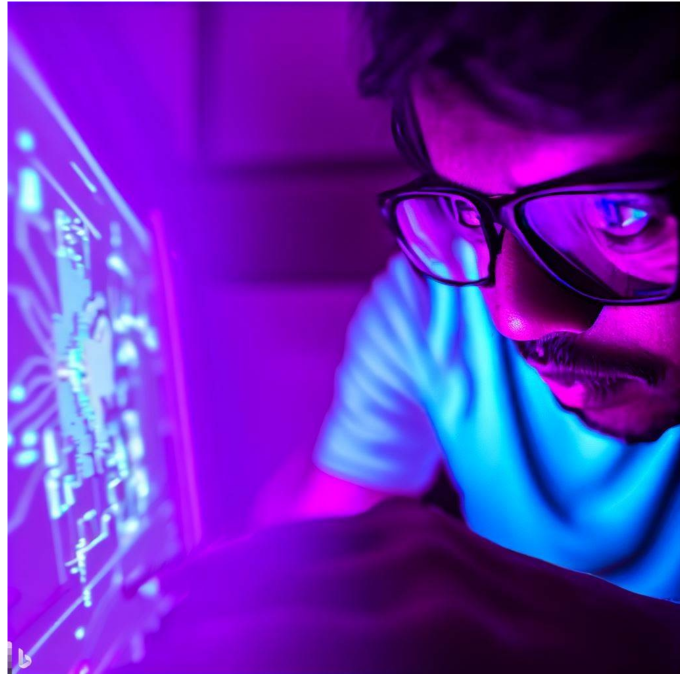


CSE 331

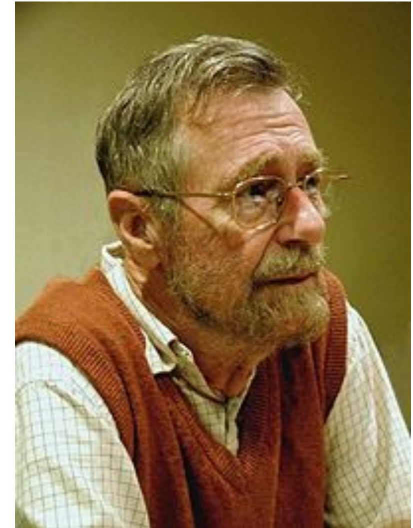
Tools & Testing

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 Written 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

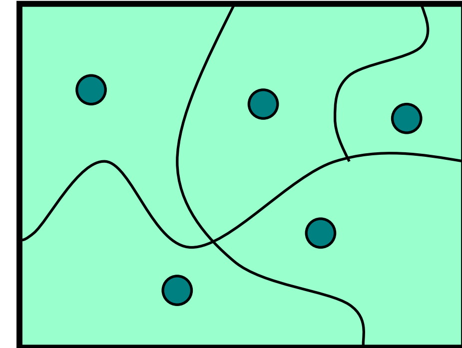
1. Choose an input / configuration
   - description of the inputs / configuration is the "test case"

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

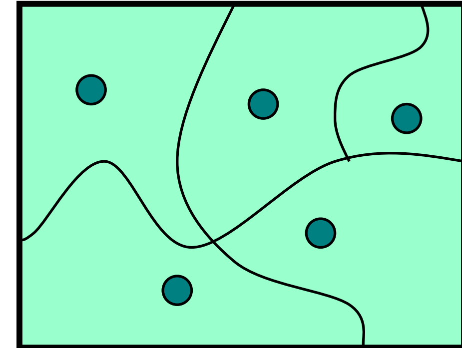
     we will use "mocha" in JS / TS

Key Problem



- Key question is what cases to test
  - at level -1, we can test all of them
  - at level 0+, we cannot
- Split up the set into **subdomains** that
  - intuitively: “are the same”
  - formally: if there is a bug, all of them reveal it
- In that case, we can pick one example per domain and know that they will catch any bug

Key Problem



- Key question is what cases to test
  - at level -1, we can test all of them
  - at level 0+, we cannot
- Split up the set into **subdomains** that
  - intuitively: “are the same”
  - formally: if there is a bug, all of them reveal it
- We don’t know where the bugs are, so we don’t know what subdomains would reveal them
  - instead, we will use some heuristics to choose subdomains
  - (another reminder: testing & tools are not enough)

Heuristic 1: Specification Testing

Idea: split cases described differently in the spec

```
// Returns a if a >= b and b otherwise.
function max(a: number, b: number): number { … }
```

- Spec describes $a \geq b$ and $a < b$ different, so we need at least two cases
  - would be negligent not to test both

Heuristic 1: Specification Testing

```
// Returns a if a >= b and b otherwise.
function max(a: number, b: number): number {
  return a;
}
```

Passes a test with a ≥ b but not a < b

```
// Returns a if a >= b and b otherwise.
function max(a: number, b: number): number {
  return b;
}
```

Passes a test with a < b but not a ≥ b

Heuristic 1: Specification Testing

Idea: split cases described differently in the spec

```
// Returns a if a >= b and b otherwise.
function max(a: number, b: number): number { … }
```

- Spec describes a ≥ b and a < b different, so we need at least two cases
  - would be negligent not to test both

- Testing both is necessary but not sufficient

Specification Testing is Not Enough

Idea: split cases described differently in the spec

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

- Spec only has one case
  - obviously, that's not good enough
- How about if we test 2, 3, 4, … 12?
  - seems okay?

Specification Testing is Not Enough

Idea: split cases described differently in the spec

```
// Returns true iff n is a prime number
function 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;
  }
}
```

Specification Testing is Not Enough

Idea: split cases described differently in the spec

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

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

Heuristic 2: Clear-Box Testing

- We need to look at the code to know what to test

- This will be our **primary** heuristic
  - usually gives finer-grained subdomains that spec testing
  - (but if not, we should use that heuristic too)

- 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 on the same straight-line code are "*the same*"

Testing Straight-Line Code

Clear-box Testing for straight-line code:

Rule: at least two test cases

- 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!
- More is obviously also okay
  - not a contest to write the fewest tests

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

Clear-box Testing for conditionals:

Rule: apply the other rules to **both** branches

- Would be **negligent** not to test one of them
- 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

- **Boundary Cases**: behavior is different on n vs n+1, then make sure you test those
  - easy to have "off by one" bugs

    e.g., behavior changes between n-1 and n instead

    happens if you use "< n" instead of "<= n"

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

Testing Conditionals

Conditionals look like this

```
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 Conditionals

Conditionals look like this

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

- **Make sure this matches spec!**
  - **does the spec say 0 and 1 are different**

    maybe the spec says the boundary is between -1 and 0
  - **don't let the code trick you into think it's right!**

Other Heuristics

Some other heuristics are also useful

- **Boundary Cases**: behavior is described different on n vs n+1, then make sure you test those
  - easy to have "off by one" bugs

    e.g., behavior changes between n-1 and n instead

- **Special Cases**: certain inputs often cause bugs
  - e.g., null, NaN, 0, empty list

    better to exclude null via the type system

    (if the input should be an integer, then clients should not pass NaN)

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
- Two important exceptions…

Testing Recursion

Recursive calls are more complicated

```
function 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 section / homework)

Testing Recursion

Clear-box Testing for recursive calls:

Rule: need to test 0, 1, and 2+ calls

- 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 a subdomain has only one input, then just one test

    e.g., "0" is in its own subdomain, that's just one test

Testing Conditionals (revisited)

Recall this conditional

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

Testing Function Calls (revisited)

Suppose we rewrite it like this

```
    return cond(n > 0, 2 * (n-1) + 1, 0);
```

where

```
  // Returns x if b else y
  function cond(b: boolean, x: number, y: number) {
    if (b) {
      return x;
    } else {
      return y;
    }
  }
```

Testing Function Calls (revisited)

Suppose we rewrite it like this

```
return cond(n > 0, 2 * (n-1) + 1, 0);
```

we have hidden the branch in a function call!

- Still need to test both branches
- Branch shows up in the spec of "cond"

```
// Returns x if b else y
function cond(b: boolean, x: number, y: number)
```

Testing Function Calls (Revisited)

Clear-box Testing for function calls:

Rule: branches in spec of the function we call are branches in our code

- Apply the other rules to all branches

Testing Recursion (Revisited)

- Recursive calls are often a special case of this:

```
// Returns 1 if n = 0 else n * fact(n-1)
function fact(n: number) {  // n >= 0
  if (n === 0) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
```

  – "else" case splits into n = 0 and n > 0

    1 recursive call and 2+ recursive calls

Summary of Heuristics

- Split inputs spec describes differently
- Split inputs where code is different
  - branches of conditionals
  - branches in specs of calls (e.g., recursion)

- Two tests per remaining subdomain
  (unless subdomain is only 1 input)
  - include boundary and special cases
  - make sure every argument changes value

- Not a contest to write the fewest tests!

Example 1

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

How many tests? Which ones?

- 0, 1, 2

Example 2

```
// n must be a positive integer
function 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, 2, 3, 4, 9 (say)

Example 3

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

How many tests? Which ones?

– 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

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