CSE 331
Software Design & Implementation

Kevin Zatloukal

Spring 2020

Testing

How do we ensure correctness?

Best practice: use three techniques

1. **Tools**
   – e.g., type checking, @Override, libraries, etc.
2. **Inspection**
   – think through your code carefully
   – have another person review your code
3. **Testing**
   – usually >50% of the work in building software

Each removes ~2/3 of bugs. Together >97%

How do we ensure correctness?

"Beware of bugs in the above code;
I have only proved it correct, not tried it."
-Donald Knuth, 1977

Trying it is a surprisingly useful way to find mistakes!

No **single activity** or approach can guarantee correctness

We need tools **and** inspection **and** testing to ensure correctness

It's hard to test your own code

Your **psychology** is fighting against you:

- confirmation bias
  - tendency to avoid evidence that you're wrong
- operant conditioning
  - programmers get cookies when the code works
  - testers get cookies when the code breaks

You can avoid some effects of confirmation bias by

writing most of your tests before the code

Not much you can do about operant conditioning

Testing Tips

- Write tests both **before** and **after** you write the code
  - (only clear-box tests need to come afterward)

- Be systematic: think through revealing subdomains & test **each one**

- ...

Kinds of testing

- Testing field has terminology for different kinds of tests
  - we won't discuss all the kinds and terms
- Here are three orthogonal dimensions [so 8 varieties total]:
  - *unit* testing versus *system/integration* testing
    - one module's functionality versus pieces fitting together
  - *black-box* testing versus *clear-box* testing
    - did you look at the code before writing the test?
  - *specification* testing versus *implementation* testing
    - test only behavior guaranteed by specification or other behavior expected for the implementation

Unit Testing

- A unit test focuses on one class / module (or even less)
  - could write a unit test for a single method
- Tests a single unit in isolation from all others
- Integration tests verify that the modules fit together properly
  - usually don’t want these until the units are well tested
    - i.e., unit tests come first

How is testing done?

Write the test

1) Choose input / configuration
2) Define the expected outcome

Run the test

3) Run with input and record the actual outcome
4) Compare *actual* outcome to *expected* outcome

What's So Hard About Testing?

"Just try it and see if it works..."

```
// requires: 1 ≤ x,y,z ≤ 10000
// returns:  computes some f(x,y,z)
int func1(int x, int y, int z){…}
```

Exhaustive testing would require 1 trillion runs!

- impractical even for this trivially small problem

Key problem: choosing test suite

- Large/diverse enough to provide a useful amount of validation
- (Small enough to write/run in reasonable amount of time.)
  - less important... very few projects have too many tests

Approach: Partition the Input Space

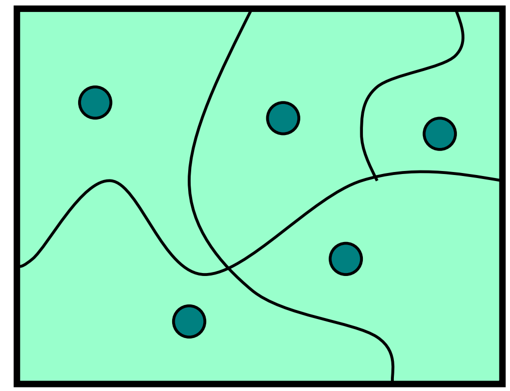
Ideal test suite:

Identify sets with "same behavior"

(actual and expected)

Test **at least** one input from each set

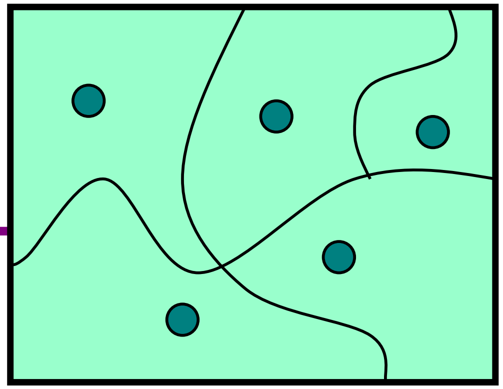
(we call this set a *subdomain*)



Problem: we don't know the behavior on all inputs

– if we did, we wouldn't need to test!

Revealing Subdomains



- A *subdomain* is a subset of possible inputs
- A subdomain is *revealing* for error *E* if either:
  - *every* input in that subdomain triggers error E, *or*
  - *no* input in that subdomain triggers error E
- Need test at least one input from a revealing subdomain to find bug
  - if you test one input from every revealing subdomain for E, you are guaranteed to find the bug
- The trick is to *guess* revealing subdomains for **the errors present**
  - even though your reasoning says your code is correct, make educated guesses where the bugs might be

Testing Heuristics

- Testing is *essential* but difficult
  - want set of tests likely to reveal the bugs present
  - but we don’t know where the bugs are
- Our approach:
  - split the input space into enough subsets (subdomains) such that inputs in each one are likely all correct or incorrect
  - can then take just one example from each subdomain
- Some heuristics are useful for choosing subdomains...

Specification Testing

Heuristic: Explore alternate cases in the specification

Procedure is a black box: specification visible, internals hidden

Example

```
// returns:  a > b => returns a
//           a < b => returns b
//           a = b => returns a
int max(int a, int b) {…}
```

3 cases lead to 3 tests

(4, 3) => 4 *(i.e. any input in the subdomain a > b)*
(3, 4) => 4 *(i.e. any input in the subdomain a < b)*
(3, 3) => 3 *(i.e. any input in the subdomain a = b)*

More Complex Example

Write tests based on cases in the specification

```
// returns: the smallest i such
//          that a[i] == value
// throws:  Missing if value is not in a
int find(int[] a, int value) throws Missing
```

Two obvious tests:

([4, 5, 6], 5) => 1
([4, 5, 6], 7) => throw Missing

Have we captured all the cases?

([4, 5, 5], 5) => 1

Must hunt for multiple cases

– Including scrutiny of effects and modifies

Specification Testing: Advantages

Process is not influenced by component being tested

- avoids psychological biases we discussed earlier
- can only do this for your own code if you **write tests first**

Robust with respect to changes in implementation

- test data need not be changed when code is changed

Allows others to test the code (rare nowadays)

Heuristic: Clear (glass, white)-box testing

Focus on features not described by specification

- control-flow details (e.g., conditions of “if” statements in code)
- performance optimizations
- alternate algorithms for different cases

Clear-box Example

There are some subdomains that black-box testing won't catch:

```
boolean[] primeTable = new boolean[CACHE_SIZE];

boolean isPrime(int x) {
    if (x > CACHE_SIZE) {
        for (int i=2; i <= x/2; i++) {
            if (x % i == 0)
             return false;
        }
        return true;
    } else {
        return primeTable[x];
    }
}
```

Combining Clear- and Black-Box

For buggy **`abs`**, what are revealing subdomains?

```
// returns:  x < 0      => returns -x
//           otherwise => returns x
int abs(int x) {
   if (x < -2) return -x;
   else        return x;
}
```

Example sets of subdomains:

- Which is best?

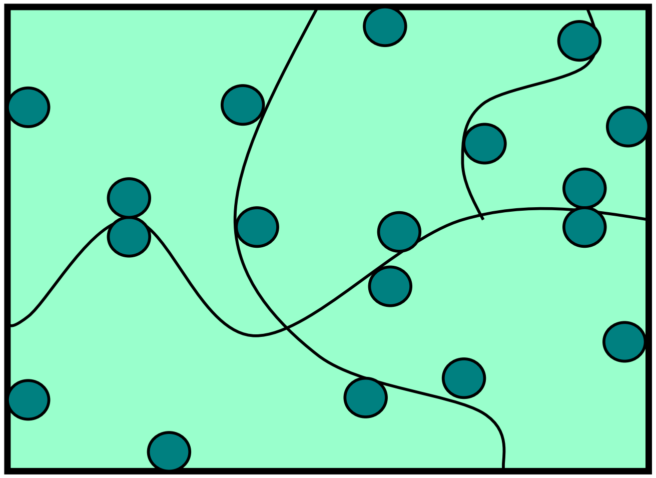
… {-2} {-1} {0} {1} …

{…, -4, -3} {-2, -1} {0, 1, …}

Why *not*: **{…,-6, -5, -4} {-3, -2, -1} {0, 1, 2, …}**

Heuristic: Boundary & Special Cases

Create tests at the edges of subdomains



Why?

- Off-by-one bugs
- "Empty" cases (0 elements, null, …)
- Overflow errors in arithmetic
- Object aliasing

Small subdomains at the edges of the "main" subdomains have a high probability of revealing many common errors

- also, you might have misdrawn the boundaries

Boundary Testing

Point is on a boundary if either:

- there exists an adjacent point in a different subdomain
- there is no point to one side

Example: function has different behavior on n and n+1

Boundary Cases: Integers

```
// returns: |x|
public int abs(int x) {…}
```

What are some values or ranges of *x* that might be worth probing?

- $x < 0$ (flips sign) or $x \geq 0$ (returns unchanged)
- Around $x = 0$ (boundary condition)
- *Specific tests: say x = -1, 0, 1*

Boundary Testing

To define the boundary, need a notion of adjacent inputs

Example approach:

- identify basic operations on input points
- two points are adjacent if one basic operation apart

Point is on a boundary if either:

- there exists an adjacent point in a different subdomain
- some basic operation cannot be applied to the point

Example: list of integers

- basic operations: *create*, *append*, *set*, *remove*
- adjacent points: <[2,3],[2,4]>, <[2,3],[2,3,3]>, <[2,3],[2]>
- boundary point: [ ] (can't apply *remove*)

Heuristic: Special Cases

Arithmetic

- smallest/largest values
- zero

Objects

- null
- list containing itself
  - maybe a bit too pathological
- same object passed as multiple arguments (aliasing)

All of these are common cases where bugs lurk

- you'll find more as you encounter more bugs

Special Cases: Arithmetic Overflow

```
// returns: |x|
public int abs(int x) {…}
```

How about…

```
int x = Integer.MIN_VALUE; // x=-2147483648
System.out.println(x<0);   // true
System.out.println(Math.abs(x)<0); // also true!
```

From Javadoc for **`Math.abs`**:

> Note that if the argument is equal to the value of **`Integer.MIN_VALUE`**, the most negative representable int value, the result is that same value, which is negative

Special Cases: Duplicates & Aliases

```
// modifies: src, dest
// effects:  removes all elements of src and
//           appends them in reverse order to
//           the end of dest
<E> void appendList(List<E> src, List<E> dest) {
  while (src.size() > 0) {
    E elt = src.remove(src.size() - 1);
    dest.add(elt);
  }
}
```

What happens if **`src`** and **`dest`** refer to the same object?

– this is *aliasing*
– it's easy to forget!
– watch out for shared references in inputs

sqrt example

```
// throws: IllegalArgumentException if x<0
// returns: approximation to square root of x
public double sqrt(double x){…}
```

What are some values or ranges of *x* that might be worth probing?

- $x < 0$ (exception thrown)
- $x \geq 0$ (returns normally)
- around $x = 0$ (boundary condition)
- perfect squares (sqrt(*x*) an integer), non-perfect squares
- *x*<sqrt(*x*) and *x*>sqrt(*x*) – that's *x*<1 and *x*>1 (and *x*=1)
- *Specific tests: say x = -1, 0, 0.5, 1, 4* (probably want more)

How many tests is enough?

Common *goal* is high **code coverage**:

- ensure test suite covers (executes) all of the program
- assess quality of test suite with % *coverage*
  - tools to measure this for you

Assumption implicit in goal:

- if high coverage, then most mistakes discovered
- **far** from perfect but widely used

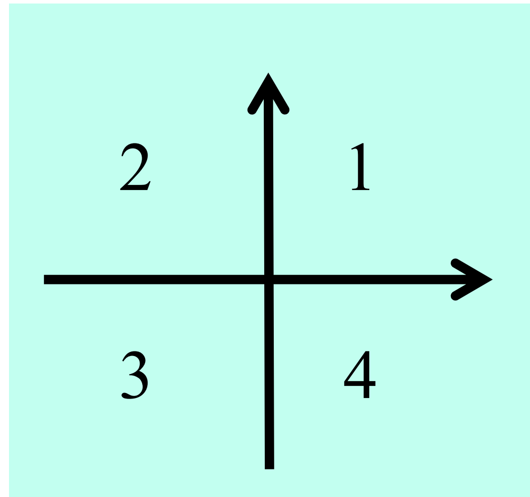
Code coverage: statement coverage

```
int min(int a, int b) {
    int r = a;
    if (a <= b) {
        r = a;
    }
    return r;
}
```

- Consider any test with $a \leq b$ (e.g., **`min(1,2)`**)
  - executes every instruction
  - misses the bug
- *Statement* coverage is not enough

Code coverage: branch coverage

```
int quadrant(int x, int y) {
  int ans;
  if (x >= 0)
    ans=1;
  else
    ans=2;
  if (y < 0)
    ans=4;
  return ans;
}
```



- Consider two-test suite: (2,-2) and (-2,2). Misses the bug.
- *Branch coverage* (all tests "go both ways") is not enough
  - here, *path coverage* is enough (there are 4 paths)

Code coverage: path coverage

```
int countPositive(int[] a) {
    int ans = 0;
    for (int x : a) {
      if (x > 0)
        ans = 1; // should be ans += 1;
    }
    return ans;
}
```

- Consider two-test suite: [0,0] and [1]. Misses the bug.
- Or consider one-test suite: [0,1,0]. Misses the bug.

- *Branch coverage* is not enough
  - here, *path coverage* is enough, but *no bound* on path-count!

Code coverage: what is enough?

```
int sumOfThree(int a, int b, int c) {
  return a+b;
}
```

- *Path coverage* is not enough
  - consider test suites where `c` is always 0
- Typically a “moot point” since path coverage is unattainable for realistic programs
  - but do not assume a tested path is correct
  - even though it is more likely correct than an untested path
- Another example: buggy `abs` method from earlier in lecture

Varieties of coverage

Various coverage metrics (there are more):

- Statement coverage
- Branch coverage
- *Loop coverage*
- *Condition/Decision coverage*
- Path coverage

increasing number of test cases required (generally)

Limitations of coverage:

1. 100% coverage is not always a reasonable target
   – may be *high cost* to approach 100%
2. Coverage is *just a heuristic*
   – we really want the revealing subdomains for the errors present

Summary of Heuristics

- Split subdomains on boundaries appearing in the specification
- Split subdomains on boundaries appearing in the implementation
- Test boundaries that commonly lead to errors
- Test special cases like nulls, empty arrays, 0, etc.
- Tests to exercise every branch of the code
  - all paths would be even nicer (but not always possible)
- Test any cases that caused bugs before (to avoid regression)

On the other hand, don't confuse *volume* with *quality* of tests

- look for revealing subdomains
- want tests in every revealing subdomain not **just** lots of tests

Testing Tools

- Modern development ecosystems have built-in support for testing
- Your homework introduces you to Junit
  - standard framework for testing in Java
- Continuous integration
  - ensure tests pass **before** code is submitted
- You will see more sophisticated tools in industry
  - libraries for creating mock implementations of other modules
  - automated tools to test on every platform
  - automated tools to find severe bugs (using AI)
  - …

Testing Tips

- Write tests both **before** and **after** you write the code
  - (only clear-box tests need to come afterward)

- Be systematic: think through revealing subdomains & test **each one**

- Test your tests
  - try putting a bug in to make sure the test catches it

- Test code is different from regular code
  - changeability is less important; **correctness** is more important
  - do not write **any test code** that is not obviously correct
    - otherwise, you need to test that code too!
    - unlike in regular code, it's *okay* to repeat yourself in tests