CSE 331
Software Design & Implementation

Kevin Zatloukal

Fall 2017

Testing

(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins, Zach Tatlock)

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

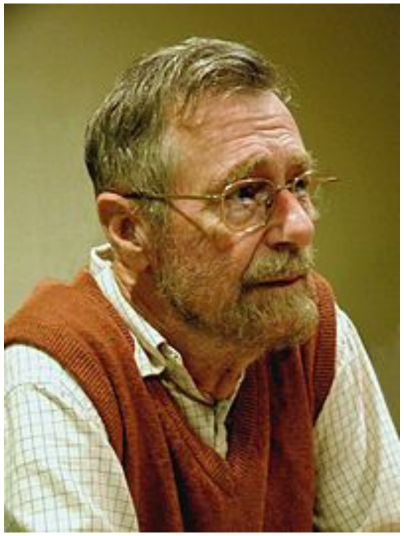
Each removes ~2/3 of bugs. Together >97%

What can you 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



Testing is essential but it is insufficient by itself

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

How do we ensure correctness?

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

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

Why you will care about testing

- Industry-wide trend toward developers doing more testing
  - 20 years ago we had large test teams
    - developers barely tested their code at all
  - now, test teams are small to nonexistent
    - e.g., Google may not have any

- Reasons for this change:
  1. easy to update products after shipping (users are testers)
  2. often lowered quality expectations (startups, games)
     - some larger companies want to be more like startups

- In all likelihood, you will be expected to test your own code
- This has positive and negative effects…

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

Outline

- Background
- Kinds of testing:
  - black-box testing
  - clear-box testing
  - regression testing
- Basic approach to testing
- Heuristics for good test suites
  - code coverage
- Tools

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 outcome

4) Compare *observed* outcome to *expected* outcome

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)

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 proc1(int x, int y, int z){…}
```

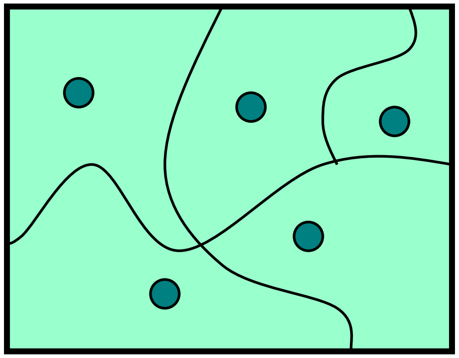
Exhaustive testing would require 1 trillion runs!

- 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.)

Approach: Partition the Input Space



Ideal test suite:

Identify sets with “same behavior”
(actual and expected)

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

Two problems:

1. Notion of same behavior is subtle
   - Naive approach: execution equivalence
   - Better approach: revealing subdomains

2. Discovering the sets requires perfect knowledge
   - If we had it, we wouldn’t need to test
   - Use heuristics to approximate cheaply

Naive Approach: Execution Equivalence

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

All x < 0 are execution equivalent:

- Program takes same sequence of steps for any x < 0

All x ≥ 0 are execution equivalent

Suggests that {-3, 3}, for example, is a good test suite

Execution Equivalence Can Be Wrong

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

{-3, 3} does not reveal the error!

Two possible executions: x < -2 and x >= -2

Three possible behaviors:

- x < -2 OK, x = -2 or x= -1 (BAD)
- x >= 0 OK

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 given subdomain
  - if subdomains cover the entire input space, we are *guaranteed* to detect the error if it is present
- 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

Example

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

- Value tested in comparison is a good (clear-box) hint

```
// 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, …}**

Heuristics for Designing Test Suites

A good heuristic gives:

- for all errors in some class of errors E: high probability that some subdomain is revealing for E and triggers E
- not an *absurdly* large number of subdomains

Different heuristics target different classes of errors

- in practice, combine multiple heuristics
  - (we will see several)
- a way to think about and communicate your test choices

Black-Box 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)*

Black Box 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)

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

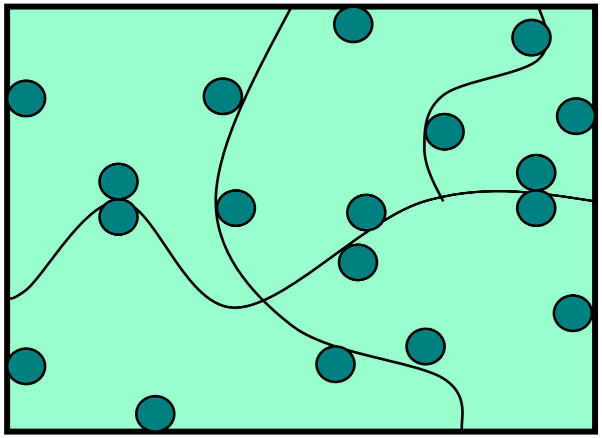
– Including scrutiny of effects and modifies

Heuristic: Boundary Testing

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

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*)

Other Boundary 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

Boundary Cases: Arithmetic Overflow

```
// 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* ≥ 0 (returns unchanged)
- Around *x* = 0 (boundary condition)
- *Specific tests: say x = -1, 0, 1*

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

Boundary 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

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

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

Clear-box Motivation

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];
    }
}
```

Clear Box Testing: [Dis]Advantages

- Finds an important class of boundaries
  - yields useful test cases

- Consider **`CACHE_SIZE`** in **`isPrime`** example
  - important tests **`CACHE_SIZE-1`**, **`CACHE_SIZE`**, **`CACHE_SIZE+1`**
  - if **`CACHE_SIZE`** is mutable, may need to test with different **`CACHE_SIZE`** values

Disadvantage:

- buggy code tricks you into thinking it's right once you look at it
  - (confirmation bias)
- can end up with tests having same bugs as implementation
- so also write tests **before** looking at the code

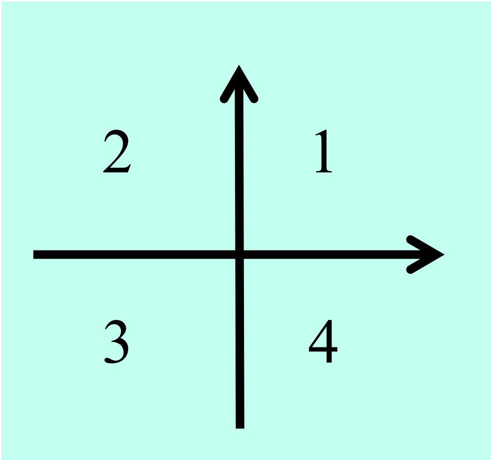
Code coverage: what is enough?

```
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: what is enough?

```
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: what is enough?

```
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):

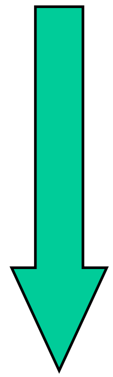
Statement coverage

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

Pragmatics: Regression Testing

- Whenever you find a bug
  - store the input that elicited that bug, plus the correct output
  - add these to the test suite
  - verify that the test suite **fails**
  - fix the bug
  - verify the fix
- Ensures that your fix solves the problem
  - don’t add a test that succeeded to begin with!
    - another reason to try to write tests before coding
- Protects against reversions that reintroduce bug
  - it happened at least once, and it might happen again (especially when trying to change the code in the future)

Summary of Heuristics

- Test boundaries appearing in the specification
- Test boundaries appearing in the implementation
- Test boundaries that commonly lead to errors
- 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 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
- You will see more sophisticated tools in industry
  - systems that ensure tests pass **before** code is submitted
  - libraries for creating fake 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
  - (clear-box tests 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