CSE 331
Software Design & Implementation

James Wilcox
Autumn 2021
Testing
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
  – think carefully through the subdomains you are using
  – 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 opaque: specification visible, internals hidden

Example

```c
// 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)
Specification Testing 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: Transparent 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
Combining Transparent and Opaque

For buggy `abs`, what are revealing subdomains?

```c
// 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?
  - `{... -6, -5, -4} {-3, -2, -1} {0, 1, 2, ...}
  - `{... -4, -3} {-2, -1} {0, 1, ...}
  - `{... -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 ≥ 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?

Correct goal should use **revealing subdomains**:
- one from the middle of each subdomain
- examples along the boundaries of each subdomain
How many tests is enough?

Common goal is to achieve 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
– low code coverage is definitely bad
int \textbf{min}(\text{int} \ a, \ \text{int} \ b) \ { \\
    \text{int} \ r = a; \\
    \text{if} \ (a \leq b) \ { \\
        r = a; \\
    } \\
    \text{return} \ r; \\
} \\

- Consider any test with \(a \leq b\) (e.g., \text{min}(1,2))
  - executes every instruction
  - misses the bug

- \textit{Statement coverage} is not 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)
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.
• *Path coverage* is enough, but *no bound* on path-count!
Code coverage: what is enough?

```java
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

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)
  – ...

CSE 331 Spring 2021 23
Testing Tips

• Write tests both before and after you write the code
  – (only transparent 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