CSE 331
Software Design & Implementation
Topic: More Testing

💬 Discussion: What is a piece of software that you find impressive?
Reminders

- After HW4, things are going to slow down a bit

Upcoming Deadlines

- Prep. Quiz: HW4 due Monday (7/11)
- HW4 due Thursday (7/14)
Last Time...

• Testing
  – unit vs. integration vs. system
  – clear-box vs. opaque-box
  – specification vs. implementation

• Testing Heuristics
  – specification
  – clear-box
  – boundary case

Today’s Agenda

• Recap: Testing
• More Testing Heuristics
• Code Coverage
• Discussion: HW4
Extra OH?

• Thinking about hosting Soham OH immediately after lecture
  – Monday/Wednesday/Friday?

• Benefits:
  – Students can work on homework immediately after the lecture
  – Ask questions about course material
  – Students can leave if they have no questions

• Drawbacks:
  – Many people lose focus when they are in the same room for 2 hours
  – Technically, I have another commitment at that time...
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 12 varieties total]:
  – *unit* testing versus *integration* versus *system / end-to-end* testing
    • ???
  – *clear-box* testing versus *opaque-box / black-box* testing
    • ???
  – *specification* testing versus *implementation* testing
    • ???
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 12 varieties total]:
  – *unit* testing versus *integration* versus *system / end-to-end* testing
    • one module’s functionality versus pieces fitting together
  – *clear-box* testing versus *opaque-box / black-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
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
Approach: Partition the Input Space

Ideal test suite:
Identify sets with “same behavior”
(actual and expected)
Test \textbf{at least} one input from each set
(we call this set a \textit{subdomain})

Two problems:

1. Notion of \textit{same behavior} is subtle
   • We want to find \textit{revealing subdomains}

2. Discovering the sets requires perfect knowledge
   • If we had it, we wouldn’t need to test
   • Use \textit{heuristics} to approximate cheaply
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
- 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
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 a black box: 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

1. (4, 3) => 4 (i.e. any input in the subdomain \( a > b \))
2. (3, 4) => 4 (i.e. any input in the subdomain \( a < b \))
3. (3, 3) => 3 (i.e. any input in the subdomain \( a = b \))
int find(int[] a, int value) throws MissingException

What tests might we want to consider for our test suite?

find([4, 5, 6], 5) => 1
find([4, 5, 6], 7) => throws MissingException
find([4, 5, 5], 5) => 1

In general, we should hunt for multiple cases (look at effects and modifies)
Heuristic: Clear-box testing

*Focus* on features not described by specification
- control-flow details (e.g., conditions of "if" statements in code)
- alternate algorithms for different cases
- behavior of the implementation not promised in the spec
  - e.g., spec doesn't promise smallest index, but implementation does produce that
Heuristic: Clear-box testing

*Focus* on features not described by specification
- control-flow details (e.g., conditions of “if” statements in code)
- alternate algorithms for different cases
- behavior of the implementation not promised in the spec
  - e.g., spec doesn’t promise smallest index, but implementation does produce that

```java
// returns: an index i such that a[i] == value
// throws: MissingException if value is not in a
int find(int[] a, int value) throws MissingException
```
// returns: x < 0 => returns -x
// otherwise => returns x
int abs(int x) {
    if (x < -2) return -x;
    else return x;
}

What subdomains might we want to consider for our test suite?

{..., -4, -3, -2, -1, 0, 1, 2, 3, ...}

is our entire input space.
Practice: Clear- and Black-Box

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

What **subdomains** might we want to consider for our test suite?

```plaintext
{..., -4, -3, -2, -1} {0, 1, 2, 3, ...}
```

after applying the specification heuristic.
// returns:  x < 0     => returns -x
//           otherwise => returns x

int abs(int x) {
    if (x < -2) return -x;
    else return x;
}

What subdomains might we want to consider for our test suite?


{..., -4, -3} {-2, -1} {0, 1, 2, 3, ...}

after applying the clear-box heuristic.
Practice: Clear- and Black-Box

Given the following partition

\{…, -4, -3\} \{ -2, -1 \} \{ 0, 1, 2, 3, …\}

what test cases should we consider for \texttt{abs}?

\[
\begin{align*}
\text{abs} (-4) & \Rightarrow 4 \\
\text{abs} (-2) & \Rightarrow 2 \\
\text{abs} ( 1) & \Rightarrow 1 
\end{align*}
\]
Heuristic: Boundary Cases

Create tests at the edges of subdomains

Why?
- off-by-one bugs
- smallest & largest numbers
- empty collection

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 1, ..., n versus n+1...

Example: $f(x)$ which requires $x \geq 0$
  - $x = 0$ is a boundary because $x < 0$ is not allowed
// returns:  x < 0  => returns -x
// otherwise => returns x
int abs(int x) {
    if (x < -2) return -x;
    else return x;
}

What subdomains might we want to consider for our test suite?

{... , -4} {-3} {-2} {-1} {0} {1, 2, 3, ...}

after applying the boundary case heuristic.
Practice: Clear- and Black-Box

Given the following partition

 {..., -4} {-3} {-2} {-1} {0} {1, 2, 3, ...}

what test cases should we consider for `abs`?

- `abs (-4)` => 4
- `abs (-3)` => 3 (boundary, clear-box)
- `abs (-2)` => 2 (boundary, clear-box)
- `abs (-1)` => 1 (boundary, specification)
- `abs ( 0)` => 0 (boundary, specification)
- `abs ( 1)` => 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
- No adjacent point in some direction

Example: f(x) which requires $x \geq 0$
- $x = 0$ is a boundary because $x < 0$ is not allowed
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
- no adjacent point in some direction

Example: list of integers
- basic operations: add, remove, set
- adjacent points: <[2,3],[2,3,3]>, <[2,3],[2]>, <[2,3],[4,3]>
- boundary point: [ ] (can't apply remove)
Heuristic: Special Cases

Arithmetic
- zero
- overflow errors in arithmetic

Objects
- null
- 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 ≥ 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)
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)
How many tests is enough?

Correct goal should use **revealing subdomains**:
- one from each subdomain
- 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 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  
- **very far** from perfect but widely used  
- low code coverage is certainly bad
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

```c
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

```c
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?

```c
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 examples on the boundaries
• Test special cases like nulls, 0, etc.
• Test any cases that caused bugs before (to avoid regression)
• Make sure tests exercise *at least* every branch & statement

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
More 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
HW4 – Part 1

• Reasoning worksheet
• Focuses on the union method in FiniteSet
HW4 – Part 2

- Writing unit tests for FiniteSet
- Testing Heuristics
  - Specification
  - Clear-box
  - Boundary
HW4 – Part 3

• We already chose this representation for you:
  – A FiniteSet of points
  – A boolean representing whether it is the complement

• Make sure you document the RI and AF
  – Will be *much* simpler than FiniteSet RI and AF
HW4 – Part 4

• If you were comfortable with the earlier parts, this should be straightforward.
• No new advice!
HW4 – Part 5

• Coding methods with many cases
• When union-ing two SimpleSets, how many cases are there?

• Homework Hack: Can you define some operations in terms of others?
HW4 – Part 6

- Start with the toString invariant
- Consider edge cases (e.g. the empty case)
1. Start on Prep. Quiz: HW4 as early as possible!
   - Reminds you about common set operations
     • E.g. union, intersection, complement
   - Think about some non-trivial cases needed for the homework