CSE 331
Software Design & Implementation

Hal Perkins

Winter 2022

Testing

We’re back!

- Classes resume on campus on Monday 1/31
  – See longer message posted to ed last night
  – No live-streaming, but lecture recordings will appear on Panopto (not zoom) shortly after each class. Sections continue in-person only.

- Updated office hours for rest of the quarter posted shortly
  – We’ll see if we can arrange additional zoom hours

- Midterm exam Tuesday 2/8, 5-6 pm
  – Review session Sunday afternoon 2-3(?) pm (tent.)

Administrivia

- HW4 due Thursday night
  - Cannot change specs or tests
  - AF/RI and loop invariants are your friends… (really ☺)
  - Think of properly tagging the final commit as releasing the software to the customer (the course staff?) – so be sure to tag the right commit!

- Next set of lectures after this are about design, classes & modules, and general style issues
  - Leadup to hw5 which is the start of a long project
  - *Lots* of related readings. Please dive in – they will be very helpful on hw5 and later.

Project Grading

- Idea: provide meaningful feedback faster & avoid phony “precision” of complex, many-point grading rubrics
- Plan: project grades will be mostly holistic:
  - 4 major categories, graded independently:
    - Design (organization of classes/methods/etc.)
    - Documentation (quality of specs; javadoc; etc.)
    - Implementation/code quality (including RI/AF, other internal comments, naming, layout etc.)
    - Testing (design & quality of tests and coverage)
  - Some of these don’t apply until hw5 or later

Basic grading scale

- For each major category, a single 0-3 score (much like a work or code review, not like “x out of y points”)
  - 3 = very good / superior, no major issues, easy / pleasant to read, probably a few fairly minor things to improve / fix up
  - 2 = generally good but some non-trivial major, or too many minor, problems
  - 1 = significant problems, needs major work
  - 0 = not credible, cannot grade, etc.
- Expect scores to be a mix of 2’s and 3’s, with more 2’s earlier in the quarter and more 3’s as we improve with practice

Additional project feedback

- Most projects have these other scores:
  - Staff tests – automated tests run on tagged "hw*n*-final" versions of code. Max score varies depending on assignment but exact max doesn't matter – scores are normalized when computing course grades
    - If these fail because of tagging or other errors, can fix and resubmit for 80% max of original possible score
  - Answers – written answers to questions – again, exact max can vary but scores are normalized
  - Mechanics – 0-3 score for whether correct files were pushed and tagged properly, code compiles, javadoc generates, staff test scripts run even if some tests fail, etc. Should always be 3. If not, may seriously affect other scores.
- All scores kept as separate info in gradebook and combined at end of quarter to get an overall assessment
  - i.e., do not add up the various numbers and expect that total to have any significance

Administrivia (added Wed.)

- HW5 posted by late today; HW6 writeup by early next week
  - HW5: design/implement/test a Graph ADT
    - 2 parts: design & write tests (1 week); implement (2nd week)
  - More in section this week (*don't* miss)
  - Do initial design yourself (for sure have a first design by end of *this* weekend) then discuss ideas & tradeoffs with others (use whiteboards, etc.)
  - HW6: social network. Might provide some more insight for what the graph ADT created in hw5 needs to support.
- Section worksheet posted now – suggest downloading the code questions before section to save time

Outline

- Testing principles and strategies
  - Purpose of testing
  - Kinds of testing
  - Heuristics for good test suites
  - Black-box testing
  - Clear-box testing and coverage metrics
  - Regression testing

Non-outline

- Modern development ecosystems have much built-in support for testing
  - Unit-testing frameworks like JUnit
  - Regression-testing frameworks connected to builds and version control
  - Continuous testing
  - …
- No tool details covered here
  - See homework, section, internships, …

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

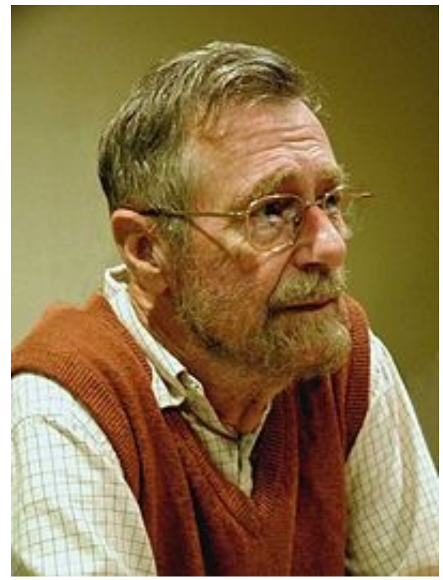
Each removes ~2/3 of bugs. Together >97%

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

Only **reasoning** can prove there are no bugs. Yet...

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

Why you will care about testing

In all likelihood, you will be expected to **test your own code**

- Industry-wide trend toward developers doing more testing
  - 20 years ago we had large test teams
  - now, test teams are small to nonexistent

- 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

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

An approach to testing

Validation = reasoning + testing

- Make sure module does what it is specified to do
- Uncover problems, increase confidence

Two rules:

1. Do it early and often
   - Catch bugs quickly, before they have a chance to hide
   - Automate the process wherever feasible

2. Be systematic
   - Have a strategy, and test everything eventually
   - If you thrash about randomly, the bugs will hide in the corner until you're gone

Kinds of testing

- Testing is so important the field has terminology for different kinds of tests
  - Won't discuss all possible 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 and system testing

- A unit test focuses on one method, class, interface, or module
- Test a single unit in isolation from all others
  - If it fails, defect is localized
  - Complications: if unit uses other libraries; if unit does mutations
- Typically done earlier in software life-cycle
  - As soon as implementation exists
  - Whenever it changes
- System testing = integration testing = end-to-end testing
  - Run whole system, ensure pieces work together

Black-box and clear-box tests

- Black-box testing
  - Tests designed using only information in the specification

- Clear-box (= white-box = glass-box) testing
  - Implementation influences test design

- But both types of tests pass for *any* implementation. Clear-box may be checking for specific edge cases and have different choices of inputs based on additional knowledge of implementation (more later)

Specification vs implementation tests

- A specification test verifies behavior guaranteed by the specification (only) and any implementation of that spec should pass these tests
- An implementation test verifies behavior of a particular implementation
  - Different implementations of a particular specification may have additional implementation-specific behaviors and properties that need to be checked
    - Including testing specific interfaces, methods or other things that can differ among implementations of the same specification
- Orthogonal to black- vs clear-box choice

How is testing done?

Write the test

1) Choose input data/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* ≥ 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*

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!

- Sounds totally impractical – and this is a trivially small problem

Key problem: choosing test suite (partitioning inputs)

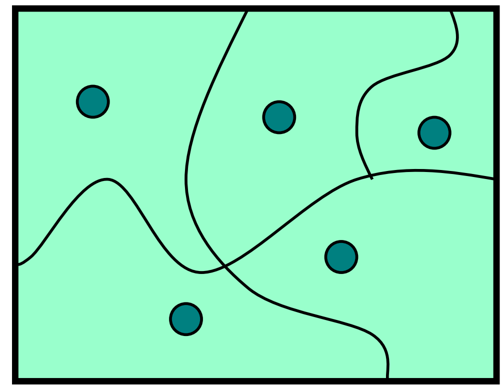
- Small enough to finish in a useful amount of time
- Large enough to provide a useful amount of validation

Approach: Partition the Input Space

Ideal test suite:

Identify sets with same behavior

Try 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;
}
```

Two execution behaviors: x < -2 and x >= -2

Three possible behaviors:

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

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

Heuristic: 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 only 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* these revealing subdomains
  - make educated guesses about where the bugs might be
  - then pick one example to test from each subdomain

Example

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, …}**

Heuristics for Designing Test Suites

A good heuristic gives:

- Few subdomains
- For all errors in some class of errors *E*: high probability that some subdomain is revealing for *E* (i.e., triggers *E*)

Different heuristics target different classes of errors

- In practice, combine multiple heuristics
- Really a way to think about and communicate your test choices

Heuristic: Black-Box Testing

Explore alternate cases in the specification

Procedure is a black box: interface visible, internals hidden, but you can use the spec to figure out things to test

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

- Assumptions embodied in code not propagated to test data
- Avoids “group-think” of making the same mistake

Robust with respect to changes in implementation

- Test data need not be changed when code is changed

Allows for independent testing (less common nowadays)

- Testers need not be familiar with code
- Tests can be developed before the code
  - Very helpful, *especially* if you are also the implementor

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?

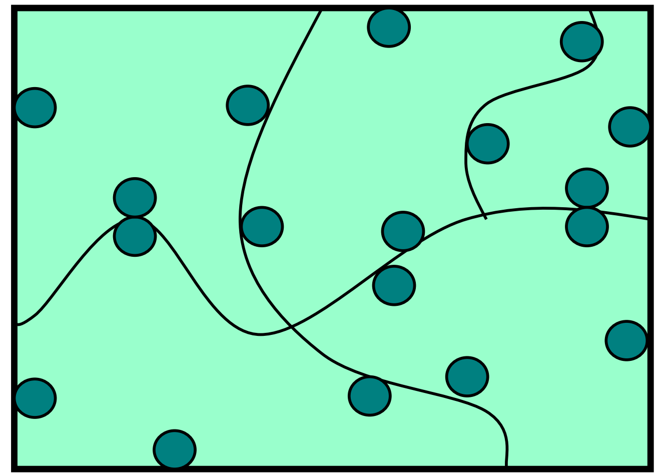
([4, 5, 5], 5) => 1

Must hunt for multiple cases

- Including scrutiny of effects and modifies

Heuristic: Boundary Testing & 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

To define the boundary, need a notion of adjacent inputs

One approach:

- Identify basic operations on input values
- Two values 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*, *remove*
- Adjacent points: <[2,3],[2,3,3]>, <[2,3],[2]>
- Boundary point: [ ] (can't apply *remove*)

Other Boundary Cases

Arithmetic

- Smallest/largest values
- Zero

Objects

- null
- Circular list
- Same object passed as multiple arguments (aliasing)

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, …)-box testing

Focus: Testing behavior required by the specification, but spec doesn't say how to do it, and want to verify that the chosen method works properly (i.e., spec tests informed by implementation details)

- Control-flow details
- Performance optimizations
- Alternate algorithms for different cases

Common *goal*:

- Ensure test suite covers (executes) all of the program
- Measure quality of test suite with % *coverage*

Assumption implicit in goal:

- High coverage → good test suite → most mistakes discovered

Clear-box Testing Example

There are some subdomains that are not evident from the specification, so black-box testing might not 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];
    }
}
```

Another Clear-box Testing Example

From the Java collections library:

```
class ArrayList<E> { ...

boolean add(E e) {
     if (currentsize == capacity) {
        increase list capacity
     }
     add e to list
}
```

Black-box testing might not add enough items to the list to require increasing the size of the underlying array. If we look at the implementation we can discover how much to add to be sure the resize happens and create a clear-box test that checks that.

It's still a specification (not implementation) test because it is only testing behaviour that is part of the spec for ArrayList.add

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:

- Tests may have same bugs as implementation
- Buggy code tricks you into complacency once you look at it (confirmation bias)
- Another good reason to write the tests *before* the code

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 can measure this for you

- Assumption is implicit in the goal
  - High coverage means most mistakes discovered
  - Far from perfect, but widely used
  - Low coverage definitely indicates problems

Code coverage: statement coverage

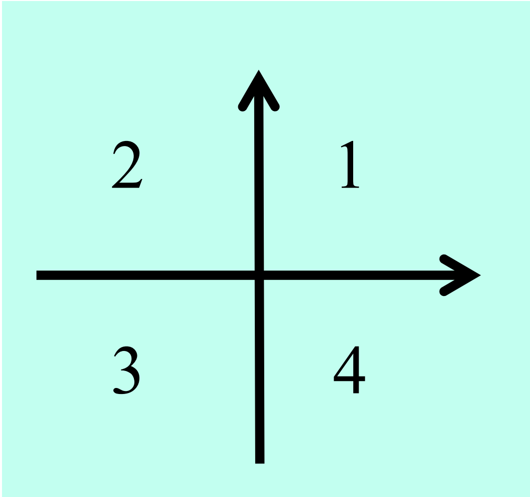
```
int min(int a, int b) {
    int r = a;
    if (a <= b) {
        r = a;
    }
    return r;
}
```

- Consider any test with $a \leq b$ (e.g., **`min(1,2)`**)
  - Executes every instruction
  - Misses the bug

- *Statement* coverage (% statements executed) 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 num_pos(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
- But here *path coverage* (% all possible control flow paths) is enough, but *no bound* on path-count

Code coverage: what is enough?

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

- *Path coverage* is not enough
  - Consider test suites where **`c`** is always 0

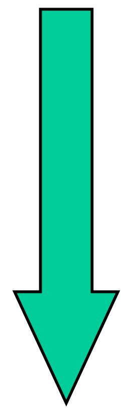
- Typically a moot point since full path coverage is unattainable for realistic programs
  - But do not assume a tested path is correct
  - Even though it is more likely correct than an untested path

- 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
   100% may be unattainable (dead code)
   High cost to approach the limit
2. Code is not necessarily correct even if executed (see buggy **`abs`** above)
3. Coverage is *just a heuristic*
   We really want the revealing subdomains

Pragmatics: How Many/What Tests?

- Ideal: each test checks one specific thing (method,…)
  - And checks only one specific behavior/aspect
  - Failure points to responsible component
- Reality: can't always test in complete isolation
  - Example: need to use observer(s) to see if creator, mutator, or producer yields correct result(s)
    - And if constructor test fails, defect could be in observer or creator
- Reality: try to structure test suites so each test checks one new thing and has minimal dependence on others
  - Failure more likely to point to a single component
- Reality: time is limited
  - Goal is to increase confidence to level needed

Pragmatics: Regression Testing

- Whenever you find a bug
  - Save 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!
- Helps to populate test suite with good tests
- Protects against regressions that reintroduce bug
  - It happened at least once, and it might happen again

Rules of Testing

First rule of testing: ***Do it early and do it often***

- Best to catch bugs soon, before they have a chance to hide
- Automate the process if you can
- Regression testing will save time

Second rule of testing: ***Be systematic***

- If you randomly thrash, bugs will hide in the corner until later
- Writing tests is a good way to understand the spec
  - Think about revealing domains and boundary cases
  - If the spec is confusing, fix it and/or write more tests
- Spec can be buggy too
  - Incorrect, incomplete, ambiguous, missing corner cases
- When you find a bug, write a test for it first and then fix it

Testing Tips

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

- Be systematic (revealing subdomains, …)

- Test your tests
  - Put a bug in the code and be sure a test catches it

- Test code is different from regular code
  - Changeability is less important; correctness is essential
  - Do not write any test code that is not *obviously* correct
    - Otherwise you need to test that code too!
    - Unlike with regular code, it's ok to repeat code in tests when needed

Closing thoughts on testing

Testing matters

- You need to convince others that the module works

Catch problems earlier

- Bugs become obscure beyond the unit they occur in

Don't confuse *volume* with *quality* of test data

- Can lose relevant cases in mass of irrelevant ones
- Look for revealing subdomains

Choose test data to cover:

- Specification (black box testing)
- Code (clear (glass, white) box testing)

Testing can't generally prove absence of bugs

- But it can greatly increase quality and confidence