

Coverage

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- [Homework 1](#): due tonight, Friday, October 9, 11:59pm
- [Reading 2](#): due Friday, October 16, 11:59pm

So far

- **Correct:** code that meets its specification
- **Three ways to know it:** testing, tools, reasoning
- **A test:** a chosen input, actual behavior against expected behavior
- **Four steps:** input, expected behavior from the spec, run, compare
- **Four tips** for the input, all read from the specification
- **Specification-based:** checks only what the specification promises

Lecture 4's four tips

1. Only test allowed inputs
2. Test each method separately
3. Test exhaustively when possible
4. Cover each case and each boundary

Tests from the specification alone

```
11  /**
12   * Decides whether a number is prime.
13   *
14   * @param n the number to decide
15   * @requires 0 < n
16   * @return true if n is prime, and false if it is not
17   */
18  static boolean isPrime(int n) {
```

- Inputs from the specification: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
 - Primes and composites, each with an answer anyone can work out

Now read the implementation

```
18  static boolean isPrime(int n) {  
19      if (n < 100) {  
20          return SMALL[n];  
21      }  
22      for (int k = 2; k * k < n; k++) {  
23          if (n % k == 0) {  
24              return false;  
25          }  
26      }  
27      return true;  
28  }
```

Below 100, a table: all eleven tests were lookups

The loop never ran. Is the method right?

Try 121

```
$ javac IsPrime.java
$ java IsPrime 121
true

$ echo '11 * 11' | bc
121
```

Line 22 stops one step early: $k * k < n$ where it should be $k * k \leq n$

Nothing in the specification says to look past 100. The code does.

Two questions

The specification is not the only thing to read.

Read the code too.

- How many tests are enough?
- Which ones?

The code gives a list: which test to write next, and when you may stop

How many tests are enough?

- Our eleven tests all ran lines 19 and 20, and never the loop
- Fifty below 100 would be no better: the number is not what matters
- Every allowed input: $2^{31} - 1$ for `isPrime`, each needing its answer
- Lecture 2's check: every input within a bound, none outside it

One test is never enough

- Suppose `isPrime` always returned `true`
 - The test on 7 expects `true`, and passes
 - The test on 8 expects `false`, and fails
- One test cannot catch it: the answer may be the one it expects
- So every method needs at least two tests

Coverage

Our eleven tests all ran the same part of the method.

What matters is which parts of the method the tests run.

- **Which parts has a set of tests run?** The code says, not the specification
- **Lecture 4** read the specification. **Now** read the code
- **Coverage:** how much of the code a set of tests has run
 - From the code: **clear-box** testing. From the specification: **black-box**
- Three measures, each catching what the last one missed

Statement coverage

- Our eleven tests ran lines 19 and 20, and no test ran the loop

Definition

A set of tests has **statement coverage** of a method if every statement that some allowed input reaches is run by some test.

- Not enough, by this measure: 2 to 12 never ran the loop. 121 did
- Enough, by this measure: 2, 100, and 101 pass, and the mistake stays

One test of min

```
14  static int min(int a, int b) {  
15      int r = a;  
16      if (a <= b) {  
17          r = a;  
18      }  
19      return r;  
20  }
```

- **The specification:** the smaller of *a* and *b*
- *firstIsSmaller*: `assertEquals(1, Min.min(1, 2))`

Every statement ran, and the test passes

A second test

- `secondIsSmaller: assertEquals(1, Min.min(2, 1))`

```
$ java -jar junit.jar execute --class-path . --select-class MinTest
:
|   └─ firstIsSmaller() ✓
|   └─ secondIsSmaller() ✗ expected: <1> but was: <2>
:
```

The first test never ran the if with its condition false

Branch coverage

- Every **if** has two branches: condition true, condition false
 - An **if** with no **else** still has a false branch, and it is empty

Definition

A set of tests has **branch coverage** of a method if every branch that some allowed input takes is run by some test.

- Our two tests of **min** have it; the second found the mistake

quadrant

```
10  *           y
11  *         2   |   1
12  *  -----+----- x
13  *         3   |   4
14  *
15  * @param x the point's x coordinate
16  * @param y the point's y coordinate
17  * @requires x != 0 and y != 0
18  * @return 1 if x > 0 and y > 0,
19  *         2 if x < 0 and y > 0,
20  *         3 if x < 0 and y < 0, and
21  *         4 if x > 0 and y < 0
```

quadrant, the implementation

```
23  static int quadrant(int x, int y) {  
24      int q = 1;  
25      if (x < 0) {  
26          q = 2;  
27      }  
28      if (y < 0) {  
29          q = q + 1;  
30      }  
31      return q;  
32  }
```

(1, 1) runs both false branches; (-1, -1) runs both true branches

Two tests, and both branches of both ifs have run

Two tests with branch coverage

```
6  @Test
7  void upperRight() {
8      assertEquals(1, Quadrant.quadrant(1, 1));
9  }
10
11 @Test
12 void lowerLeft() {
13     assertEquals(3, Quadrant.quadrant(-1, -1));
14 }
```

Both pass. Is the method right?

A third test

```
16  @Test
17  void lowerRight() {
18      assertEquals(4, Quadrant.quadrant(1, -1));
19  }
```

```
$ java -jar junit.jar execute --class-path . --select-class QuadrantTest
```

```
├─ upperRight() ✓
├─ lowerLeft() ✓
└─ lowerRight() ✗ expected: <4> but was: <2>
```

Paths

- The two passing tests ran every branch, but never this combination
- A path through the method is one combination of branches
 - `quadrant` has four, and testing every path finds the mistake
- We do not ask for paths: they multiply
 - Ten `if`s in a row: 2^{10} paths, over a thousand
 - A loop: a path for every number of times its body can run
- One more measure, for loops, which branches do not measure

Lecture 4's indexOf and its four tests

```
18  static int indexOf(int[] a, int lo, int hi, int x) {  
19      for (int i = lo; i < hi; i++) {  
20          if (a[i] == x) {  
21              return i;  
22          }  
23      }  
24      return -1;  
}
```

- **findsTheOnlyMatch**: {3, 0, 7}, 0 to 3, x = 7, expects 2
- **returnsMinusOneWhenAbsent**: {3, 0, 7}, 0 to 3, x = 42, expects -1
- **findsTheFirstMatch**: {5, 1, 5, 2, 5}, 0 to 5, x = 5, expects 0
- **findsSomeMatch**: the same input, expects any index holding 5

Measure IndexOfTest

- Statement and branch coverage: yes
- The loop's body: how many times does it run in each test?
- **findsTheOnlyMatch**: `lo` 0, `hi` 3, so `i` takes 0, 1, 2. Three times
- **returnsMinusOneWhenAbsent**: the same range, no 42. Three times
- **findsTheFirstMatch**, **findsSomeMatch**: 5 at index 0. Once

Never zero. The missing test: an empty range, `lo == hi`

Loop coverage

Definition

A set of tests has **loop coverage** of a method if, for every loop in it, some test runs the loop's body zero times, some test runs it once, and some test runs it more than once.

- **Zero times:** the code after the loop must be right when it did nothing
- **Once:** the body must be right on its own
- **More than once:** what the loop carries from one run to the next

pow

```
16  static int pow(int b, int n) {  
17      int r = 1;  
18      for (int i = 0; i < n; i++) {  
19          r = b;  
20      }  
21      return r;  
22  }
```

- **The specification:** b to the n, for $0 \leq n$, when it fits in an `int`
- **Two tests:** `pow(2, 0)` expects 1, and `pow(2, 1)` expects 2

Every statement, both branches, both pass. Is the method right?

Line 19 should read `r = r * b;`

A third test of pow

- `pow(2, 3)` expects 8: the loop runs three times

```
$ java -jar junit.jar execute --class-path . --select-class PowLoopTest
:
├─ loopOnce() ✓
├─ loopZeroTimes() ✓
└─ loopManyTimes() ✗ expected: <8> but was: <2>
:
```

Recursion

- The same rule, counting calls: zero, one, more than one

```
32  static int powRec(int b, int n) {  
33      if (n == 0) {  
34          return 1;  
35      }  
36      powRec(b, n - 1);  
37      return b;  
38  }
```

- Two tests: `powRec(2, 0)` expects 1, and `powRec(2, 1)` expects 2

Every statement, both branches, both pass. Is the method right?

Lines 36 and 37 should read `return b * powRec(b, n - 1);`

A third test of powRec

- `powRec(2, 3)` expects 8: three calls

```
$ java -jar junit.jar execute --class-path . --select-class PowRecTest
:
|  ┌ oneCall() ✓
|  ┌ zeroCalls() ✓
|  └ manyCalls() ✗ expected: <8> but was: <2>
:
```

When to stop

A method's tests are enough when there are at least two and, between them, they have run all an allowed input can reach:

- **Every statement**
- **Both branches of every `if`**
- **Every loop's body** zero times, once, and more than once
- **Every method that calls itself** with zero, one, and more than one call

Until then, the list says which test to write next. Then you may stop.

The second list of James Wilcox and Kevin Zatloukal's "Testing" (2025).

Three things about the list

- **Only allowed inputs count**, as tip 1 says
 - A statement only a disallowed input reaches is not on the list
- **The measures nest**: loops bring branches, branches bring statements
 - Plan for loops and branches, and get statements for free
- **In wide use**: coverage tools report statement and branch coverage
 - Loop coverage you count yourself, as we did for [IndexOfTest](#)
 - Homework holds a method's tests to the whole list

What each measure asks for

Every allowed input

Every path

Branch coverage

Statement coverage

every statement runs

Loop coverage

zero times, once, more than once
branch coverage asks for none

Coverage is not correctness

- **min**, **pow**, and **powRec**: found by a test the list asked for
- **quadrant**: two tests completed the list and passed
 - Only a third, which the list did not ask for, found the mistake
- **isPrime**: add 100 and 10,007, a prime, to our eleven



```
isPrime(100): returns false, loop runs 1  
isPrime(10007): returns true, loop runs 99
```

Every measure met, every test passes, and 121 still passes as prime

A boundary, read off the code

- The condition $k * k < n$ divides inputs at $k * k == n$
 - A test should sit on that line: a perfect square
- Lecture 4's boundaries: the specification. **This one:** the code
- The list does not ask for it; a careful tester writes it anyway

Outside the list

- **A missing case:** in the specification, and in no line of the code
 - No statement, branch, or loop to run, so the list asks for no test
 - Its test comes from reading the specification, as Lecture 4 did
- **The Therac-25:** six patients overdosed in the 1980s
 - An edit to the prescription within the machine's eight seconds
 - The task that set the magnets never noticed the edit
 - The mistake was in the timing of two tasks, outside the list

Testing refutes, never confirms

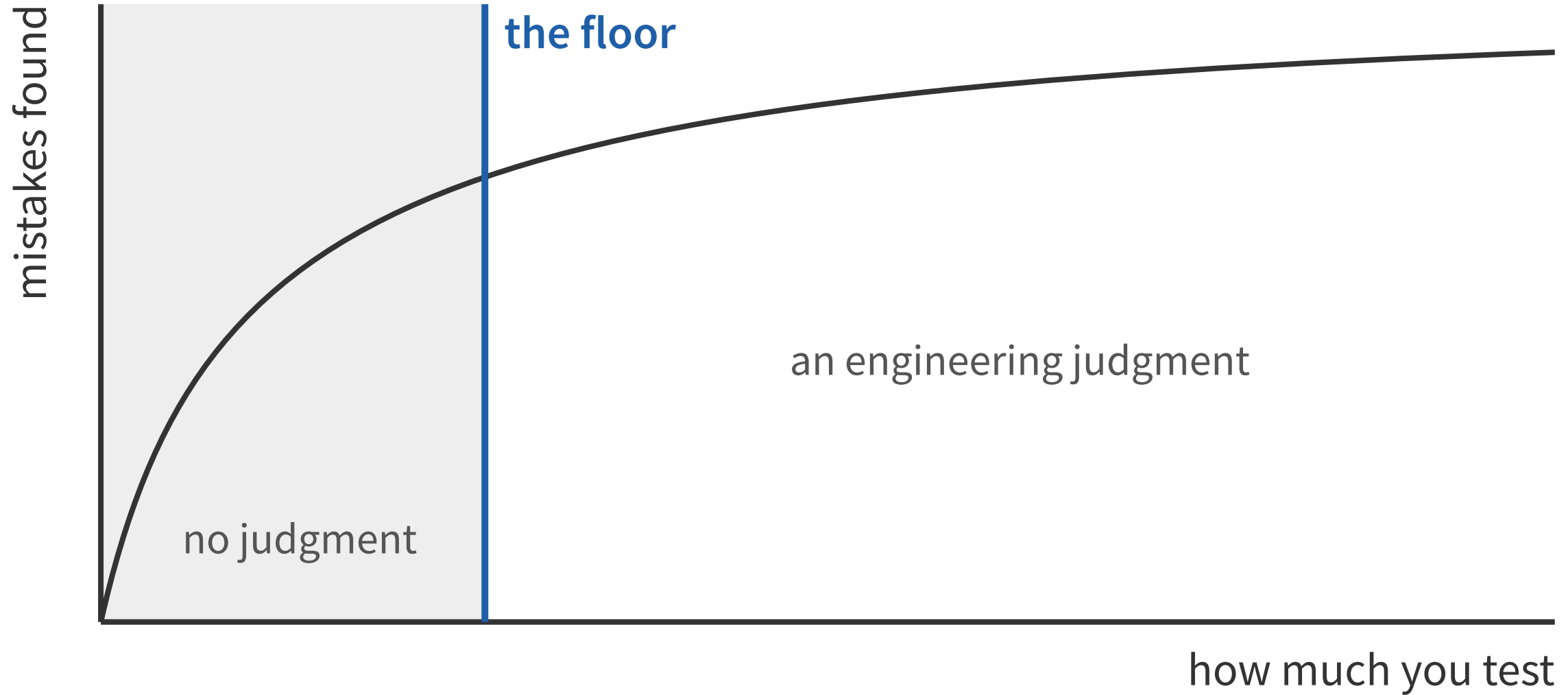
- **Lecture 1:** testing can refute a claim about every run, never confirm it
 - Short of testing every allowed input
- **Completing the list:** a good reason to believe, not knowledge
- **A tool** confirms one claim: the one it was built to check
- **Reasoning**, after Monday, confirms a claim about every run

The floor

- The list says when you may stop, not that you must
- **Above the list:** how much more depends on what a mistake would cost
 - A steering program: every path, every boundary read off the code
 - An engineering judgment, on every program you write
- **Below the list:** no judgment to make
 - The list, plus Lecture 4's cases and boundaries
 - The least any code you deliver gets, whatever the deadline

That least is the floor. The motto says whose job it is.

Above the floor



Below the floor

- Code tested less than the floor is not code you know works
- Delivering it anyway is not a trade-off an engineer may make
- It is a failure of due diligence

Half sheet (3 minutes)

- 1. `isPrime`'s tests meet the floor, and 121 still passes as prime
 - Was leaving 121 out a failure of due diligence? Say why
- 2. `min(1, 2)` expects 1 and `min(3, 3)` expects 3, and both pass
 - Branch coverage? If not, which test completes it, and what it expects
- Hand it in at the door as you leave

goto fail, 2014

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
...
```

Pasted once too often: the second `goto fail;` sits under no `if`

Every run jumps to the exit with `err` still success. The check below never ran

Statement coverage would have shown it: no test ran those lines, and none could

Adam Langley, "Apple's SSL/TLS bug" (2014), from Apple's published source.

Heartbleed, 2014

- **OpenSSL**: a library many web servers use to secure their connections
- A **heartbeat request** carries a length and asks for that many bytes
- **The code never checked** that length against the bytes that came
 - A missing check, the kind of mistake the last section named
- **Anyone could read** 64 kilobytes of a server's memory per request
 - Passwords and private keys included

Tip 4 would have shown it: a length larger than the payload is a boundary

OpenSSL Security Advisory, April 7, 2014 (CVE-2014-0160).

Test code is code

- **Lecture 2's check was a program:** a loop over every small array
 - A loop that skipped the empty range would have reported success
- **Keep tests simple:** one call and one `assertEquals`, checked by eye
 - A test with a loop in it needs its own tests
- **A clever test:** write two plain ones instead

Testable by design

- A method is hard to unit test when:
 - It is large, or does several jobs at once
 - It reads or writes outside its arguments and return: a file, the screen
- Lecture 2 split **Sum** into `readList` and `sumList`
 - A test of `sumList`: one call, one `assertEquals`, no keyboard
 - The split that lets you understand a program lets you test it

What tests tell us

- A **test** confirms a claim about its own run
- **Coverage** says how many such runs, and which: read off the code
- The **expected answers** still come from the specification

Your job is to deliver code that you know works.

- **To know** takes reasons: tests and tools now, arguments after Monday

What you owe

The floor is the price of admission to calling yourself an engineer.

You don't get to cost-benefit your way below the baseline.

The curve tells you how far up to go;

the floor tells you what you owe no matter what.

If you don't test your code at least this much,

you are failing due diligence as an engineer.

Coming up

- **Monday:** error handling, and interactive programs
- **Tuesday:** section 2, on testing
- **Homework 1:** due tonight, 11:59pm
- Hand in your half sheet at the door