Java Testing

Table of contents

Introduced in 2026-03-26-testing[1] (Lecture, Week 5).

Levels of testing

1. **Unit testing** — check that each "unit" in the project behaves correctly.
2. **Integration testing** — check that components work together and that the interfaces between components work as expected.
3. **System testing** — does the system as a whole work correctly?
4. **(User) acceptance testing** — do users of the system agree that the system does what it's supposed to do?

These form a pyramid, from many low-level unit tests at the base up to a few acceptance tests at the top.

Regression testing

During development, testing helps answer two questions: does the new stuff work, and have we broken things that used to work (**regressed**)? Regression testing ensures old features keep working as new features are introduced.

[1] courses/csse2002/2026-03-26-testing.pdf

Black box vs white/glass box testing

- **Black box** — the software has inputs and outputs to test, but the implementation is unknown to you; you test according to the *specification* (see java-specification[2]) — what it's supposed to do.
- **White/glass box** — testing designed *with* knowledge of the internal implementation, so tests can pay special attention to cases where the implementation is complicated.

Black box techniques

Smoke tests — test the common-case functionality first ("can it run?"), to decide whether more rigorous testing is worthwhile. The name comes from electronics testing: plug in a board, turn on the power — if you see smoke, stop.

Boundary tests — investigate extremes and corner cases, since that's where bugs often occur:

Input type	Try
Whole number	0, -1, minimum value, maximum value
Floating point	0, -1, `NaN`, infinity
Collection (array, list, set…)	empty, one element, large
Reference type	`null`
Resource (file, link)	non-existent resource

Equivalence classes — groups of inputs that the system treats the same way. E.g. for `getCurrentPassengers()` on a bus with capacity 80: `Valid-Empty` (0), `Valid-Normal` (1-79), `Valid-Full` (80), `Invalid` (anything else). The corresponding *boundary* tests probe each class's edges: `-1, 0, 1` (around empty), `79, 80, 81` (around full capacity) — giving a minimal boundary set of `-1, 0, 1, 79, 80, 81`.

White box technique: code coverage

Of all the ways a program could run, how many are covered by the tests? Three levels, from weakest to strongest:

1. **Statement coverage** — every statement is executed at least once.
2. **Branch coverage** — every branch is tested for both the `true` and `false` case.
3. **Path coverage** — every distinct path through the code is traversed.

[2] courses/csse2002/java-specification.pdf

Example:

```
public void register(int x) {
    if (x > 0) {
        positives += 1;
    }
    if (x % 2 == 0) {
        evens += 1;
    }
}
```

- Statement coverage: `x = 2` alone covers every line (both `if` bodies run).
- Branch coverage: `x = 2` and `x = -1` together cover both branches of each `if` (true/false).
- Path coverage: needs all 4 combinations of the two conditions: `x=2` (true, true), `x=1` (true, false), `x=-2` (false, true), `x=-1` (false, false).

Loops make exhaustive path coverage infeasible — a loop like `for (int i = 0; i < 100; i++) { if (f(i, k)) { j++; } }` has 2^{100} paths (over a billion tests/second would still take ~40.2 trillion years, about 2900 times the age of the universe). Instead, path coverage for loops is **approximated**: treat 0, 1, and 2-or-more iterations (of a loop or recursive call) as equivalent ("engineers' induction: one, two, three — that's good enough for me").

Test Driven Development (TDD)

Originates from the Agile manifesto and Extreme Programming. Cycle: **write a (failing) test** → **write code** until the **test passes** → **refactor** → write the next test. Developers write small test cases for every feature based on their initial understanding, and only modify/write new code when a test fails (avoiding duplicated test scripts) — the tests determine when code is "ready".