Java Testing

Table of contents

Levels of testing 1
Test frameworks 1
Regression testing 2
Black box vs white/glass box testing 2
Test Driven Development (TDD) 8

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?

Test frameworks

Libraries that make writing and running tests easier. **xUnit** is the prevalent family — JUnit, NUnit, CPPUnit, PyUnit, … — and this course uses **JUnit 4** (http://junit.org/junit4); see java-junit[2].

Depending on how the software is structured, a unit-testing framework can also be driven for other kinds of automated testing. A caveat: *"unit testing" is sometimes incorrectly used to mean any automated test.*

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

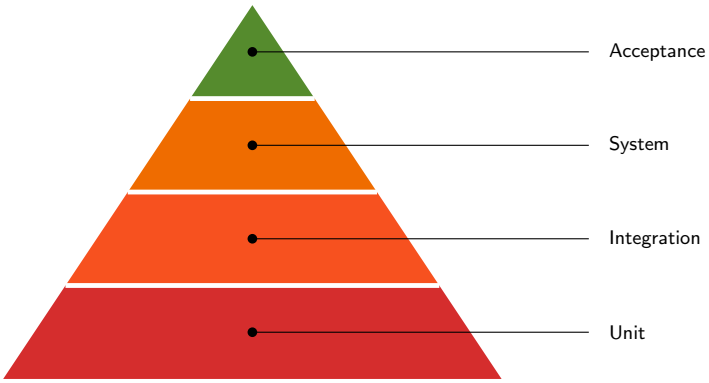

Figure 1: The four levels of testing as a pyramid. Unit tests form the wide base, acceptance tests the narrow apex. In the usual test-pyramid reading, the width of a band stands for how many tests of that kind a project has.

Regression testing

During development, testing helps answer (at least) two questions:

- Does the new stuff work?
- Have we broken things which used to work — have we **regressed**?

Regression testing ensures that old features are still working as they should when new features are introduced.

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[3]) — 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

Black box testing works from the Javadoc alone. The running example is a bus, and the *specification is all you get*:

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

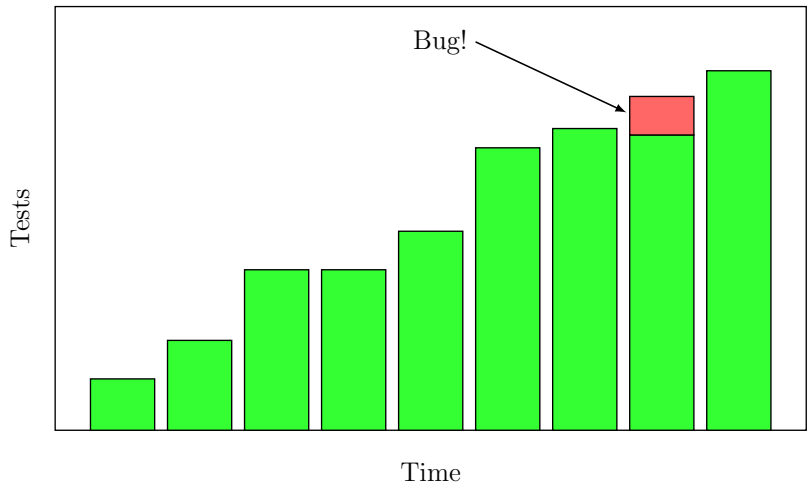

Figure 2: A regression (chart after browserstack.com/guide/regression-testing). Nine bars on *Tests*/*Time* axes: read each bar as a build, and its height as the size of the test suite growing as features are added. The red top slice on the second-to-last bar is the regression, and the arrow labelled Bug! points at it.

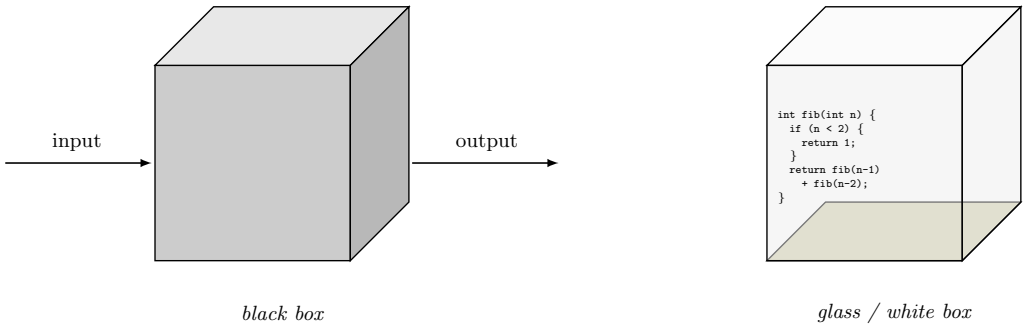

Figure 3: The two testing stances. Both are the same grey cube. The black box is opaque and shows only its input and output arrows. The glass box is translucent, so the implementation inside, a recursive `fib`, can be seen through its walls.

```java
/**
 * Calculate how many passengers are currently on the bus.
 *
 * @returns The number of passengers on the bus.
 */
public int getCurrentPassengers() {...}

/**
 * Calculate the average number of passengers on the bus between stops.
 *
 * @returns The average passengers on the bus.
 */
public int getAverageCount() {...}
```

> **Note**
>
> The tag is written `@returns` here and `@return` in the specification tag table from the exceptions lecture. The course uses both spellings — see csse2002-diagram-notation[4], "Specification tag spelling".

Smoke tests — test the common-case functionality. Usually the *first* test to be performed ("can it run?"), used to determine whether further, more rigorous testing is required. The name comes from electronic hardware testing: plug in a new board, turn on the power, and if you see smoke coming from the board, turn the power off — you don't have to do any more testing. (From *Lessons Learned in Software Testing*.)

Boundary tests — investigate extremes and corner cases, i.e. 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, etc.)	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. For `getCurrentPassengers()` on a bus whose maximum capacity is 80, build the table in two passes: first the classes, then the boundary tests for each class.

[4] courses/csse2002/csse2002-diagram-notation.pdf

Class	Values	Boundary tests
Valid – Empty	0	-1, 0, 1
Valid – Normal	1 – 79 (any number within the range)	Covered by above and below
Valid – Full capacity	80	79, 80, 81
Invalid	–	-1, 81

Minimal boundary set: `-1, 0, 1, 79, 80, 81`. The "Normal" class contributes no probes of its own because its edges are already the outer probes of the classes on either side.

The black-box method

The method in six points:

1. Black box testing: no knowledge of implementation, only specification.
2. JUnit is a test framework.
3. Tests must not violate preconditions.
4. Start with smoke tests.
5. Expand test suite with boundary tests.
6. Stop when confidence is high.

Point 3: a test that breaks `@requires` proves nothing, because the method is free to do anything at all once its precondition is violated. Point 6: testing is never exhaustive, so you stop on judgement rather than on a completeness proof. The loop argument below makes the same admission concrete.

White box technique: code coverage

With tests *and* an implementation, code coverage measures how much of the code is actually tested: of all the different ways your program could run, how many are covered by your tests. There are three levels, in this order:

1. **Statement coverage** — each statement is executed.
2. **Branch coverage** — each branch is tested for the `true` case and the `false` case.
3. **Path coverage** — every 'path' through the code is traversed.

A worked example:

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

- **Statement coverage:** any odd positive value, e.g. `x = 1`, gives a partial run: `if (x > 0)`, `positives += 1` and the `if (x % 2 == 0)` test are executed while `evens += 1` is missed. `x = 2` alone executes every statement, since it is positive *and* even so both `if` bodies run.
- **Branch coverage:** `x = 2` makes both conditions `true` and `x = -1` makes both `false`, so `x = 2` and `x = -1` together give each condition both outcomes, `true, false`. Two tests.
- **Path coverage:** all four combinations of the two conditions, so four tests. The four inputs are in the table below.

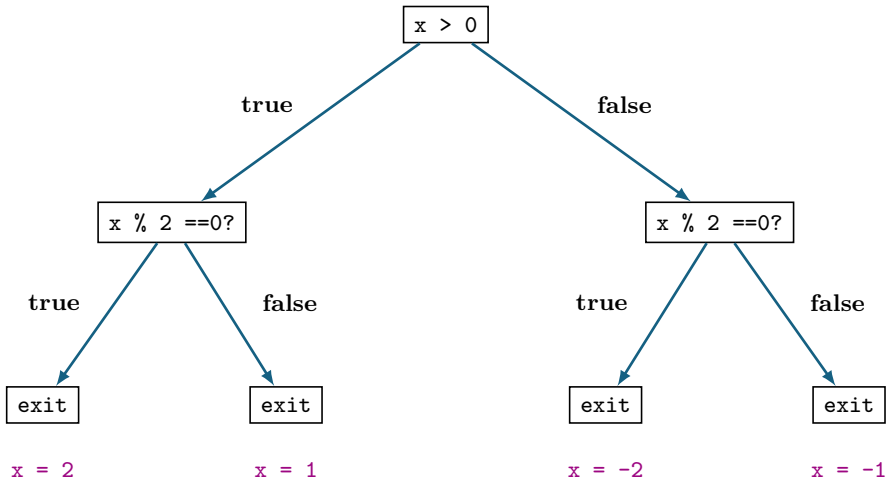

Figure 4: Path coverage for `register`. Each internal box is a condition, each teal arrow an outcome labelled true/false, each leaf one complete path to `exit` — four leaves, so four tests. The magenta input under each leaf is a value of `x` that drives that path.

Written out, the four paths and their inputs are:

Path	x
`x > 0 && x % 2 == 0`	2
`x > 0 && !(x % 2 == 0)`	1

Path	x
`!(x > 0) && x % 2 == 0`	-2
`!(x > 0) && !(x % 2 == 0)`	-1

The three levels run from weakest to strongest, and this exercise makes that concrete: the same `x = 2` that gives full *statement* coverage misses three of the four paths. That is why "100% statement coverage" is a weak claim.

Loops make exhaustive path coverage infeasible

```
for (int i = 0; i < 100; i++) {
    if (f(i, k)) {
        j++;
    }
}
```

There are 2^{100} paths through this program — 1,267,650,600,228,229,401,496,703,205,376 of them. At a billion tests per second that is 40.2 trillion years, roughly 2900 times the age of the universe.

So path coverage is **approximated**. The common approximation is that **2 or more iterations through a loop, or recursive calls, are equivalent**. This is *engineers' induction*: "One, two, three — that's good enough for me." For `for (init; test; iterate) { body }` the three cases to test are:

- **0 iterations:** `init`; `test` (**fails**); `nextStatement`;
- **1 iteration:** `init`; `test`; `body`; `iterate`; `test` (**fails**); `nextStatement`;
- **2 iterations:** `init`; `test`; `body`; `iterate`; `test`; `body`; `iterate`; `test` (**fails**); `nextStatement`;

Using JUnit for white-box testing

Design tests based on internal logic. You try to cover **branches**, **loops** and **edge cases**, and you may aim for a code coverage target (e.g. 100%). A worked white-box example — `DiscountCalculator`, the five tests that cover it, and its branch tree — is in java-junit[5].

[5] courses/csse2002/java-junit.pdf

Test Driven Development (TDD)

The TDD approach originates from the Agile manifesto principles and Extreme Programming. In TDD, developers create small test cases for every feature based on their initial understanding. The primary intention is to **modify or write new code only if the tests fail** — this prevents duplication of test scripts. Use the tests to determine when code is "ready".

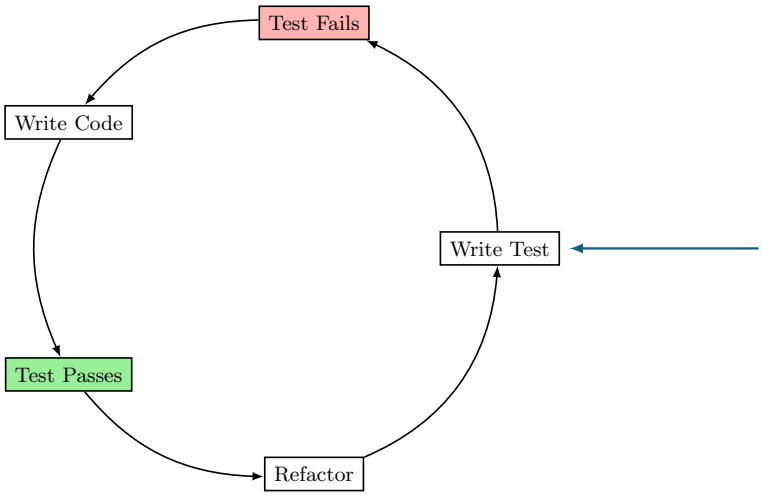

Figure 5: The TDD cycle. Read it counter-clockwise: write a test, watch it fail (pink), write code, watch it pass (green), refactor, and go round again. The teal arrow on the right points at *Write Test* and marks the entry point of the loop: the test comes before the code.

The *Refactor* step connects to java-refactoring[6]: the green test suite is what makes it safe to restructure code without changing its behaviour.

[6] courses/csse2002/java-refactoring.pdf