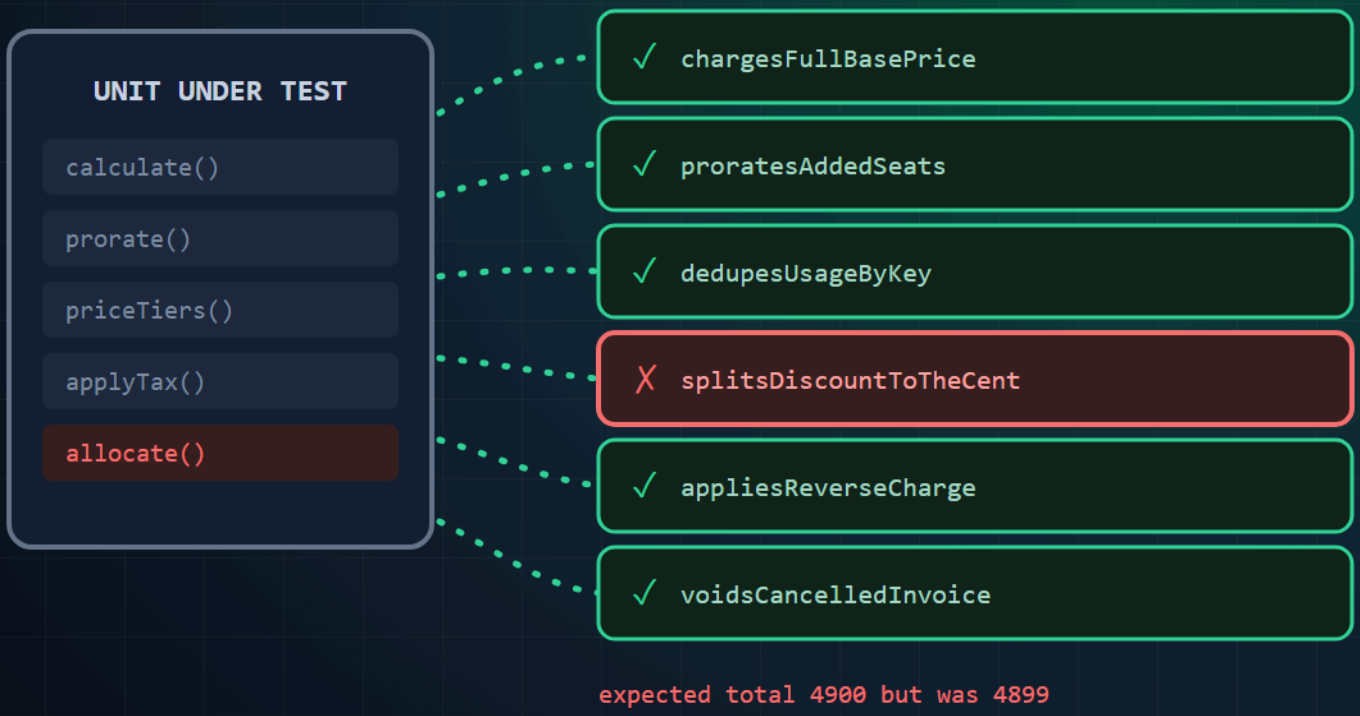
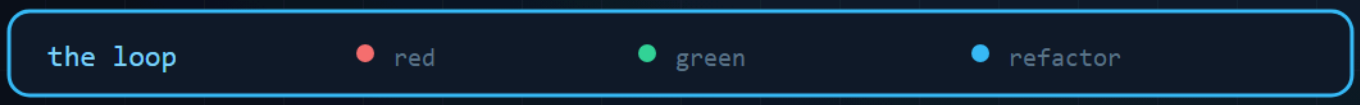




ARRANGE · ACT · ASSERT · ONE REASON TO FAIL

WRITING TESTS THAT CATCH REAL BUGS

Unit Testing for Developers

Tests that fail for the right reason — and keep failing until it is fixed

28 chapters · 435 pages · one billing system taken from no tests at all to a suite the team deploys on

DOUBLES · SEAMS · COVERAGE · MUTATION · TDD · LEGACY CODE

THENEXBOOKS

Unit Testing for Developers

Writing Tests That Catch Real Bugs
From first assertion to a suite your team trusts

First Edition

THENEXBOOKS

Unit Testing for Developers
Writing Tests That Catch Real Bugs

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. No test suite proves the absence of bugs; verify every technique against your own codebase before depending on it.

This book contains no benchmarks, defect-rate statistics, productivity figures, or industry survey results, because such numbers depend entirely on codebase, team, and measurement method, and are routinely quoted out of context. Where a number appears, it is either a budget chosen for the fictional system used as an example, or a blank for you to fill in from your own measurements.

Java, JUnit, AssertJ, Mockito, PostgreSQL, Docker, Git, and other product and project names mentioned in this book are trademarks of their respective owners. Their use is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by any of these projects, foundations, or companies.

Coppervane, the running example in this book, is fictional, as are its customers, payment provider, and every other organization named in its examples. Any resemblance to real companies or products is coincidental.

Preface

Almost every team says it values unit testing. Far fewer have a suite they would bet a Friday deploy on. The gap between those two states is not knowledge of a testing framework — frameworks are small and their documentation is good. The gap is judgment: knowing what deserves a test, what a test should assert, where to put the boundary between the code under test and everything else, and how to tell a suite that protects you from a suite that merely runs green.

This book is about that judgment. It assumes you can already write a test that passes. What it teaches is how to write tests that fail for the right reason, that keep failing until the bug is fixed, that survive a refactor of the code they cover, and that a colleague can read in a year and understand without asking you.

It is also honest about the costs. Tests are code you have to maintain, and a badly designed suite is worse than no suite: it slows every change, breaks on refactors that changed no behavior, and trains the team to ignore red. Several chapters here are about deleting tests, merging them, or deciding not to write one at all.

Who This Book Is For

- **Developers** who write tests but suspect they are testing the wrong things.
- **Senior engineers and tech leads** who review tests, set conventions, and have to defend what the suite costs.
- **Engineers inheriting a legacy codebase** with no seams and no tests, who need a way in.
- **Teams fighting a slow or flaky suite** that has quietly stopped being trusted.
- **Developers moving into a codebase** where other people depend on every change they make.

You should be comfortable in one object-oriented or statically typed language and know what an interface is. No prior experience with mocking frameworks, test-driven development, or coverage tooling is assumed.

What You Will Learn

- A precise definition of the unit under test, and why the popular definitions produce the worst suites.
- How to choose what to test, so effort goes where the bugs actually live.
- Names, structure, and assertions that turn a failure message into a diagnosis.
- Parameterized cases and property-based testing for an input space you cannot enumerate.
- Test data that stays readable when an object needs twelve fields just to exist.
- The five test doubles, what each is for, and the design changes that make most of them unnecessary.

- Controlling time, randomness, identifiers, and everything else that makes a test nondeterministic.
- Clear, practical positions on testing database code and code that calls other services.
- What coverage measures, what it hides, and how mutation testing grades the tests themselves.
- Test-driven development applied to real tickets rather than katas — including where it does not fit.
- Making a slow suite fast and a flaky suite deterministic, with a diagnosis procedure for both.
- Getting legacy code under test: seams, characterization tests, and refactoring without a safety net.
- Running tests in CI, reviewing them properly, and the anti-patterns worth naming and removing.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) defines the unit test, diagnoses why suites fail, and sets up the codebase and tooling. **Part II — Writing Tests Worth Keeping** (Chapters 6–11) is the craft of the individual test. **Part III — Dependencies and Boundaries** (Chapters 12–17) covers test doubles, testable design, and the boundaries that cause the most argument. **Part IV — Coverage, Quality, and Feedback** (Chapters 18–22) judges the suite rather than the code. **Part V — Legacy Code and Refactoring** (Chapters 23–25) is for the codebase you inherited. **Part VI — Team, Pipeline, and Practice** (Chapters 26–28) covers CI, team conventions, and a closing case study. Four appendices give a pattern cheat sheet, templates and the same tests in five languages, a suite health checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Coppervane

Throughout the book we work on **Coppervane**, a fictional subscription billing and usage-metering platform used by software companies. Customers subscribe to plans, add and remove seats in the middle of a month, send metered usage events, and receive an invoice at the end of each billing period. Payments succeed, fail, and get retried.

We chose billing because it is the ideal subject for unit testing and a merciless one. The rules are pure logic with brutal edge cases: proration across a month with 28 days, graduated pricing tiers, rounding a percentage discount across invoice lines so the total still adds up, a trial that ends on the same day the plan changes. Mistakes are visible to customers and expensive to correct. And the system around those rules has every boundary the book needs to discuss — a database, a payment provider, a clock, a stream of usage events, and a 1,900-line legacy engine that Chapters 23 to 28 rescue.

Conventions and a Note on Languages

Code examples are **Java 21 with JUnit 5**, using AssertJ for assertions and Mockito where a mocking framework is warranted, because that stack makes every concept in the book expressible and its idioms are widely imitated elsewhere. The techniques are not specific to it. **Appendix B writes the**

book's core tests four more times — in pytest, Vitest, Go's `testing` package, and xUnit.net — so you can carry each pattern into the language you actually use, and the text calls out the places where a language's idiom differs materially.

You will find no defect statistics, productivity figures, or survey results in this book. Those numbers depend on codebase and measurement method to the point where quoting them is misleading. Where a number appears, it is Coppervane's own budget or a blank for you to fill in from your own measurements.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, a clarification, or a useful aside.

> **WARNING**
> A common mistake, or a habit that produces a suite nobody trusts.

Code appears in `monospace`, diagrams are drawn as ASCII art so they survive any reader, and long lines are wrapped to fit the page. Money is always an integer in minor units with a currency code — never a floating-point number, in the production code or in the tests.

Contents

Preface

PART I — FOUNDATIONS

1 What a Unit Test Is — and What It Is Not

2 Why Test Suites Fail: Brittleness, Flakiness, and False Confidence

3 The Anatomy of a Good Test

4 Meet Coppervane: The System We Will Test

5 Setting Up: Runner, Assertions, Doubles, and a Fast Feedback Loop

PART II — WRITING TESTS WORTH KEEPING

6 Naming and Structure: Tests as Executable Documentation

7 Choosing What to Test: Behavior, Not Implementation

8 Assertions: Precision, Failure Messages, and Custom Matchers

9 Parameterized Tests and Property-Based Testing

10 Test Data: Builders, Object Mothers, and Fixtures

11 Errors, Edge Cases, and the Boundaries Nobody Tests

PART III — DEPENDENCIES AND BOUNDARIES

12 Test Doubles: Stubs, Mocks, Fakes, and Spies

13 Designing for Testability: Seams, Injection, and a Pure Core

14 Mocking Frameworks — and When to Put Them Down

15 Time, Randomness, and Everything Else That Is Not Deterministic

16 Testing Code That Owns a Database

17 Testing Code That Calls Someone Else's Service

PART IV — COVERAGE, QUALITY, AND FEEDBACK

18 Coverage: What It Measures and What It Hides

19 Mutation Testing: Grading the Tests Themselves

20 Test-Driven Development on Real Work

21 Keeping the Suite Fast

22 Flaky Tests: Diagnosis, Cure, and Prevention

PART V — LEGACY CODE AND REFACTORING

23 Getting Legacy Code Under Test

24 Characterization Tests and Safe Refactoring

25 Refactoring the Tests: Removing Duplication Without Hiding Intent

PART VI — TEAM, PIPELINE, AND PRACTICE

26 Tests in the Pipeline: CI, Reports, and Required Checks

27 Team Practice: Review, Ownership, Conventions, and Anti-Patterns

28 Case Study: Rewriting Coppervane's Billing Engine

APPENDICES

A Testing Pattern Cheat Sheet

B Templates and the Same Tests in Five Languages

C Test Suite Health Checklist

D Glossary

PART I

Foundations

What a unit test actually is, why suites that look healthy still let bugs through, the anatomy of a test worth keeping, a tour of the Coppervane billing platform we test throughout the book, and a setup that gives you an answer in seconds rather than minutes.

1 · What a Unit Test Is — and Is Not

2 · Why Test Suites Fail

3 · The Anatomy of a Good Test

4 · Meet Coppervane

5 · Setting Up a Fast Feedback Loop

CHAPTER 1 · PART I — FOUNDATIONS

What a Unit Test Is — and What It Is Not

The word "unit" has never had an agreed definition, and the one you pick quietly decides what your whole suite will look like in three years. This chapter picks one on purpose, and explains what it costs.

Three answers to "what is a unit?"

Ask five developers on the same team what a unit test is and you will get three definitions and two arguments. That is not a sign of sloppiness. The term arrived without a specification, and every tool that claims to support unit testing supports all three readings equally well. JUnit does not care. Your build does not care. The only place the definition shows up is in the shape of the suite you end up maintaining, which is to say: it shows up eighteen months later, when someone needs to change the code and discovers that four hundred tests have opinions about it.

The three answers are worth naming precisely, because each one produces a different suite.

Answer A: a unit is a class

This is the most common answer and the most damaging. It leads to a rule that feels tidy — one test class per production class, named after it — and to a second rule that follows from the first: if the class under test has collaborators, replace them with test doubles, because otherwise you are testing more than one class and the test is no longer a unit test.

Follow those two rules honestly and you get a suite that is a scale model of your current class diagram. Every constructor parameter becomes a mock. Every extracted helper becomes a new test file. The suite's structure is welded to a set of design decisions that were, at the time, provisional.

Here is what that looks like in Coppervane, the subscription billing platform we will be testing for the rest of the book. Early on, one class did the arithmetic for an invoice: it walked the subscription's seats, priced metered usage against graduated tiers, applied a coupon, and added tax. Later someone split it into `TierPricer`, `TaxCalculator`, `ProrationCalculator` and a slimmer `InvoiceCalculator` that coordinates them. No behavior changed. The invoices came out byte-identical.

Under "a unit is a class", the test for the coordinator now has to look like this:

billing/src/test/java/com/coppervane/billing/InvoiceCalculatorTest.java

```
// BAD EXAMPLE - do not copy. This is what "a unit is a class" forces.
@ExtendWith(MockitoExtension.class)
class InvoiceCalculatorTest {

    @Mock private TierPricer tierPricer;
    @Mock private TaxCalculator taxCalculator;
```

```
    @Mock private ProrationCalculator prorationCalculator;

    @InjectMocks private InvoiceCalculator calculator;

    @Test
    void appliesTaxToTheSubtotal() {
        when(tierPricer.price(any(), anyLong())).thenReturn(Money.zero(Currency.USD));
        when(taxCalculator.tax(any(), any())).thenReturn(Money.of(931, Currency.USD));

        Invoice invoice = calculator.calculate(
                subscription, plan, MARCH_2026, List.of(), null);

        assertThat(invoice.tax()).isEqualTo(Money.of(931, Currency.USD));
        verify(taxCalculator).tax(any(), any());
    }
}
```

Read that assertion again. It says: the tax on the invoice equals the number the stub was told to return. The test cannot fail unless the coordinator drops the value on the floor. It proves that a field is copied. It says nothing at all about whether Coppervane charges German customers 19% of the right number — and the "right number" is exactly the thing that matters, because the coupon has to be applied before tax and this test would pass whichever order the production code used.

That is the characteristic failure of Answer A. It produces tests that are simultaneously expensive to maintain and empty of information. They are expensive because they encode the collaborator list, so they break whenever the collaborator list changes. They are empty because every interesting value in them was supplied by the test itself.

There is a second, subtler cost. Because the class-per-class rule makes each test file the sole owner of one class, the arithmetic that spans classes ends up owned by nobody. Coppervane's real rule — a percent coupon applies to the subtotal, and tax applies to the discounted amount — lives in the seam between `InvoiceCalculator`, the coupon logic and `TaxCalculator`. Under Answer A that seam has no home: the coordinator's test stubs both sides, and each collaborator's test knows only its own fragment. The most valuable test in the whole suite is the one the convention forbids you to write.

Answer B: a unit is a method

Answer B is less popular and slightly worse in a different direction. Methods are not behaviors. A behavior — "usage records that share an idempotency key are counted once" — may live in one method, or be split across three, or be an emergent property of a loop and a guard clause. Aiming a test at a method produces two reliable symptoms. The first is tests for accessors, because a method exists and therefore feels like it needs coverage. The second is pressure to make private methods package-private or public so they can be reached, which turns an internal step into part of the class's contract.

The tell is in the names. A suite built on Answer B contains methods called `testCalculateSeatLine()` and `testApplyCoupon()`. A suite built on behavior contains `chargesForSeatsAboveThePlanAllowance()` and `appliesThePercentCouponBeforeTax()`. The second set can be read by a finance analyst. The first set can only be read by someone holding the source.

Answer C: a unit is a behavior

The definition this book uses is that a unit is a *behavior*: one observable rule about what the system does, exercised through whatever collection of classes currently implements it. Usually that is one class and a couple of value objects. Sometimes it is four classes that always travel together. The count is not the point; the point is that the test names a rule the business would recognize, and the classes behind it are free to move.

This definition has a property the other two lack: it survives refactoring. Split `InvoiceCalculator` into five classes, merge it back into one, rename all of them — a test that says "an invoice for twelve seats on `team-monthly` charges seven extra seats at nine dollars" keeps passing, because that sentence was never about classes.

> **NOTE**
>
> Answer C does not mean "test through the HTTP layer". The test still calls code directly, still runs in milliseconds, still touches nothing outside the process. What changes is which *question* the test asks. See the section on boundaries below for how to draw the line without counting classes.

The properties that actually matter

Arguing about the definition is less useful than agreeing on the properties, because the properties are what you feel every day. A test that has all five of these is worth keeping almost regardless of how many classes it loads. A test missing any one of them will eventually cost the team more than it returns.

Fast. Not "fast" as a virtue, but fast enough that running the relevant tests is cheaper than reasoning about whether to run them. The moment a developer starts choosing which tests to run, the suite has stopped being a feedback loop and become a gate at the end. Speed is not about a number you can put in a book; it is about whether the run finishes before your attention moves.

Deterministic. Same input, same result, every time, on every machine, in any order, forever. This is the property that sounds obvious and is violated most often — by `Instant.now()`, by `Math.random()`, by iterating a `HashMap`, by a default locale, by a test that passes only because another test ran first and left a row behind.

Isolated from other tests. Each test sets up everything it needs and shares no mutable state with its neighbors. The practical test for this property is to run the suite in a random order and in parallel. If that changes the result, the suite has hidden dependencies between tests, and the failures it reports

will point at the wrong test. Coppervane's suite had a static `TestData.NEXT_INVOICE_NUMBER` counter for exactly one week; two tests asserted on `INV-2026-0003` and each was correct only when it ran second.

Failing with a readable diagnosis. A test's real output is not "passed". A passing test prints nothing you read. The output you read is the failure message, and it is the only part of the test that has to communicate under pressure — at 5pm, to someone who did not write it, about code they have never seen. A test that fails with `expected: true but was: false` has technically done its job and practically wasted an hour.

Independent of environment. No database, no network, no filesystem, no container, no environment variable, no ambient system clock. Not because those things are dirty, but because each one is a way for the test to fail for reasons that have nothing to do with the code, and every such failure teaches the team to distrust red builds.

Property	What it buys you	How it is usually lost
Fast	Tests run as you type	Spring context, DB, sleeps
Deterministic	Red means broken	Real clock, random, hash order
Isolated	Failure names the cause	Static state, shared fixture
Readable failure	Fast diagnosis	Boolean asserts, no message
Environment-free	Same result anywhere	Config files, env vars, ports

The archetypal unit: Coppervane's ProrationCalculator

Coppervane charges monthly. Customers change seat counts mid-month, upgrade plans on a Tuesday, and cancel on the nineteenth. Every one of those events produces a partial charge, and partial charges are where billing systems go to die: off-by-one on the day count, a cent that vanishes, a month that has 28 days, a change effective before the period even started.

All of that logic lives in one small class with no dependencies at all.

billing/src/main/java/com/coppervane/billing/pricing/ProrationCalculator.java

```
package com.coppervane.billing.pricing;

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public final class ProrationCalculator {

    /**
     * The portion of {@code full} that covers effectiveOn (inclusive) to the
     * end of the period (exclusive).
     */
    public Money prorate(Money full, LocalDate effectiveOn, BillingPeriod period) {
        if (!effectiveOn.isAfter(period.start())) {
            return full;
```

```
        }
        if (!effectiveOn.isBefore(period.endExclusive())) {
            return Money.zero(full.currency());
        }
        int totalDays = period.days();
        int coveredDays =
                (int) ChronoUnit.DAYS.between(effectiveOn, period.endExclusive());

        return full.allocate(coveredDays, totalDays - coveredDays).get(0);
    }
}
```

This is the archetype. It takes values, returns a value, and reaches for nothing. There is no clock inside it — the effective date is a parameter, which is the whole reason the class is testable. There is no rounding by hand either: `Money.allocate` performs a largest-remainder split, so the two halves of a prorated month always add up to the full price. Chapter 4 takes `allocate` apart in detail; for now, notice only that the interesting decisions have been pushed into values the test can hand over.

A test for it looks like this, and this is roughly the standard of test the rest of the book is aiming at:

billing/src/test/java/com/coppervane/billing/pricing/ProrationCalculatorTest.java

```
class ProrationCalculatorTest {

    private static final BillingPeriod MARCH_2026 =
            new BillingPeriod(LocalDate.of(2026, 3, 1), LocalDate.of(2026, 4, 1));

    private final ProrationCalculator calculator = new ProrationCalculator();

    @Test
    void chargesTheFullPriceWhenTheChangeIsEffectiveOnTheFirstDay() {
        Money full = Money.of(4900, Currency.USD);

        Money charged = calculator.prorate(full, LocalDate.of(2026, 3, 1), MARCH_2026);

        assertThat(charged).isEqualTo(Money.of(4900, Currency.USD));
    }

    @Test
    void chargesSixteenOfThirtyOneDaysForAChangeOnTheSixteenth() {
        Money full = Money.of(4900, Currency.USD);

        Money charged = calculator.prorate(full, LocalDate.of(2026, 3, 16), MARCH_2026);

        assertThat(charged).isEqualTo(Money.of(2529, Currency.USD));
    }

    @Test
    void chargesNothingWhenTheChangeIsEffectiveAfterThePeriodEnds() {
        Money full = Money.of(4900, Currency.USD);

        Money charged = calculator.prorate(full, LocalDate.of(2026, 4, 1), MARCH_2026);
```

```
        assertThat(charged).isEqualTo(Money.zero(Currency.USD));
    }
}
```

Three tests, three sentences, three fixed numbers. Nothing is stubbed, because there is nothing to stub. Each one fails for exactly one reason, and the failure names a date and an amount — which is what the person diagnosing it needs.

> **TIP**
>
> The expected value `2529` is written as a literal on purpose. It is 16/31 of $49.00, rounded. Do not compute it in the test with `full.times(16).dividedBy(31)` — that re-implements the production formula, and a test that re-implements the formula agrees with the bug. Chapter 3 treats this as a rule.

Bad test, then the rewrite

Coppervane's repository history contains the following. It was written by someone who believed they were writing a unit test, and by the class-name definition they were: there is a class named `ProrationCalculator` and a test class named after it.

billing/src/test/java/com/coppervane/billing/pricing/ProrationCalculatorTest.java

```
// BAD EXAMPLE - do not copy.
@SpringBootTest
@Transactional
class ProrationCalculatorTest {

    @Autowired private ProrationCalculator calculator;
    @Autowired private SubscriptionRepository subscriptions;

    @Test
    void testProrate() {
        Subscription sub = subscriptions
                .findById(new SubscriptionId("sub_9001"))
                .orElseThrow();

        Money result = calculator.prorate(
                Money.of(4900, Currency.USD), LocalDate.now(), MARCH_2026);

        assertTrue(result.minorUnits() > 0);
    }
}
```

Count the defects. It boots the entire Spring context and opens a transaction to test a class with no dependencies. It requires a row with the id `sub_9001` to exist, so it fails on a clean database and on every developer machine seeded differently. It loads `sub` and never uses it. It uses `LocalDate.now()`,

which means the result changes every day — and because the result changes every day, the assertion had to be weakened to "greater than zero", which is the only claim that holds all month. The name `testProrate` describes the method called, not the rule checked.

Then April arrived. `LocalDate.now()` returned 2026-04-01, which is not before `endExclusive`, so `prorate` correctly returned zero, and the build went red:

```
ProrationCalculatorTest > testProrate() FAILED
    org.opentest4j.AssertionFailedError: expected: <true> but was: <false>
        at ProrationCalculatorTest.testProrate(ProrationCalculatorTest.java:24)
```

That message contains no date, no amount, and no hint that the calendar was involved. The on-call engineer looked at the diff for the day, found nothing related, re-ran the build, watched it fail again, and eventually added `@Disabled`. The suite lost a test and gained a piece of folklore about April.

The rewrite is the three-test class from the previous section. It is longer in lines and far cheaper in every other respect: it needs no context, no database row and no calendar; each test pins one date and one amount; and when the sixteenth-day case breaks, the failure reads:

```
ProrationCalculatorTest > chargesSixteenOfThirtyOneDaysForAChangeOnTheSixteenth() FAILED
    java.lang.AssertionError:
    expected: Money[minorUnits=2529, currency=USD]
     but was: Money[minorUnits=2450, currency=USD]
```

A reviewer who has never seen the class can tell from that message alone that the day count is off, because 2450 is exactly half of 4900 and half a month is what you get when you divide days by two instead of counting them. The failure message did the diagnosis. That is the property the original test had traded away to keep its `@SpringBootTest`.

Drawing the boundary by what the test controls

Most attempts to distinguish unit from integration tests count classes, and counting classes does not work: a test that exercises `InvoiceCalculator` plus `Money` plus `TierPricer` plus four records touches seven types and is still the fastest, most deterministic test in the suite. A test that exercises one class and a JDBC connection touches two and is neither.

The useful question is not *how much code runs* but *how much the test controls*. Everything inside the boundary is supplied by the test, deterministically, from literals it wrote itself. Everything outside the boundary is something the test hopes will behave.

```
UNIT: the boundary is the process, and the test controls everything inside it

  +-------------------------------------------------------------+
  |  test JVM                                                   |
  |                                                             |
  |   [ @Test method ] -- calls --> [ ProrationCalculator ]      |
```

```
  |         |                              |                    |
  |         | supplies: Money, LocalDate,  | returns: Money      |
  |         |           BillingPeriod      |                    |
  +-------------------------------------------------------------+
        X  PostgreSQL      X  ORCHARD-PAY      X  system clock
        X  message queue   X  LARKMAIL         X  filesystem

INTEGRATION: the boundary moves out; one real thing is no longer controlled

  +-------------------------------------------------------------+
  |  test JVM                                                   |
  |   [ @Test ] --> [ InvoiceRepositoryJdbc ] --> JDBC ---+     |
  +-------------------------------------------------------|-----+
                                                          v
                                          [ real PostgreSQL ]
                                     schema, types, constraints,
                                     transactions: NOT controlled

END-TO-END: almost nothing is controlled

  [ HTTP client ] --> admin-console --> billing service --> DB
                                              |
                                              +--> ORCHARD-PAY sandbox
```

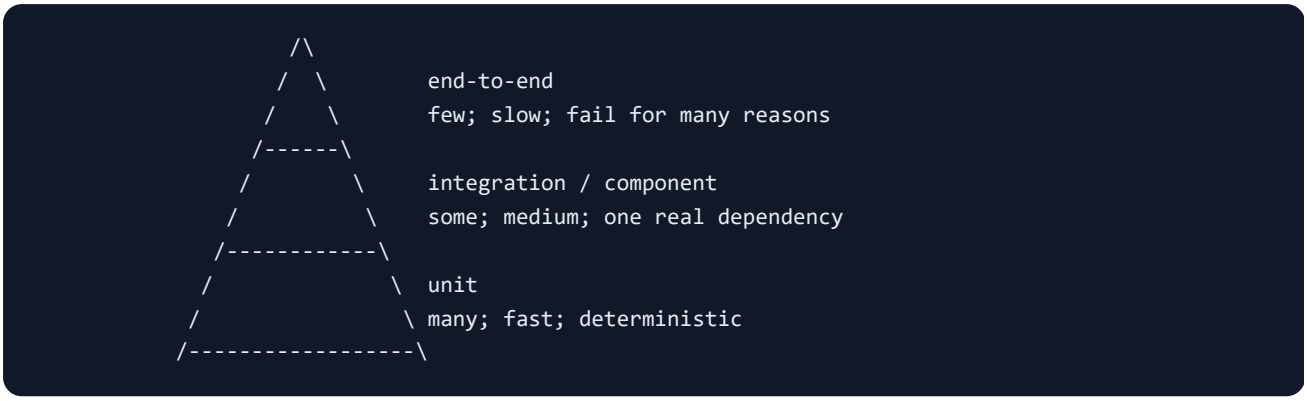
This reframing dissolves most of the arguments. "Is a test that uses an in-memory fake repository a unit test?" Yes — the fake is controlled by the test; it is deterministic, it lives in the process, and it has no schema to drift. "Is a test that starts H2 a unit test?" No, and the distinction matters, because H2 has its own SQL dialect and its own opinion about types, so the test can now fail for reasons that are about H2 and pass for reasons that are about H2.

The term **component test** fills the middle: several classes that ship together, wired for real, with everything outside the component replaced by controlled doubles. Coppervane's billing run is tested this way in Chapter 12 — real `BillingRunService`, real calculators, real invoice assembly, with `InMemorySubscriptionRepository` and `FakePaymentGateway` underneath. It is slower than a proration test and still deterministic. Whether your team files those under "unit" or "component" matters far less than that everyone knows what is controlled.

Kind	Controlled by the test	Not controlled
Unit	All inputs, all doubles	Nothing
Component	Inputs, outer doubles	Internal wiring
Integration	Inputs, some doubles	One real dependency
End-to-end	The request only	Almost everything

The pyramid, and the honest objections to it

The test pyramid is the oldest picture in this field: many fast tests at the bottom, fewer medium-sized ones above, a thin layer of whole-system tests at the top.



The pyramid earns its place for one reason: the cost of diagnosing a failure grows as you go up. A red unit test names a rule. A red end-to-end test says "checkout is broken", and finding out why means reading logs from four processes. Pushing detail down the pyramid is a statement about diagnosis cost, not about virtue.

The objections are real and you should hold them at the same time.

- **It is a shape, not a budget.** The picture implies you should count tests per layer. Nobody has ever improved a suite by hitting a ratio. A hundred class-level tests full of mocks make a wide base and a weak one.
- **It assumes your risk lives in the logic.** Coppervane's risk does — proration, tiers, rounding, tax. A service whose job is mostly to move JSON from a queue into a table has almost no logic to unit test, and a pyramid-shaped suite there is largely ceremony. Some teams draw a "trophy" instead, with the integration layer widest, and for that kind of system they are right.
- **It says nothing about which tests.** The pyramid is silent on the only question that matters: does this test correspond to a rule that could plausibly be got wrong? Chapter 7 is about that question.
- **The layers leak.** Many real suites contain a large middle layer of tests that boot a framework context to exercise pure logic — slow like an integration test, blind like a unit test, and counted as neither.

Use the pyramid as a reminder that diagnosis cost rises with scope. Do not use it as a target.

What unit tests structurally cannot catch

This is the part most testing material skips, and it is the part that keeps teams honest. Some defects are invisible to unit tests not because your unit tests are bad but because the category excludes them. If you believe a green unit suite means the release is safe, these are the bugs that will teach you otherwise.

- **Wiring and configuration.** Every unit is correct; the container injects the wrong implementation, or the profile that sets the tax service URL is not active in production. Nothing inside the boundary can see this, because the test does the wiring itself.
- **Schema drift.** The repository builds perfect SQL for a column that was renamed in a migration last Thursday. A unit test with an in-memory fake has no schema and therefore no opinion.
- **Serialization and contracts.** A field renamed in a record changes the JSON the admin console consumes, or the queue payload `coppervane-meter` expects. Both sides pass their own unit tests while disagreeing about the wire format.
- **Concurrency at scale.** A unit test can pin down a single-threaded algorithm. It cannot reproduce two billing runs starting three milliseconds apart and both issuing an invoice for the same subscription. That needs a real lock, a real database and a real race.
- **Performance and resource limits.** A correct method that loads every usage record for the month into a list passes every assertion and exhausts the heap in production.
- **Third-party reality.** A stubbed `PaymentGateway` returns what you guessed ORCHARD-PAY returns. If the provider sends `insufficient_funds` where you assumed `card_declined`, your stub is wrong and your tests are green. Chapter 17 is entirely about this trap.
- **The requirement was wrong.** The most expensive category. A test encodes an intention; if the intention was mistaken, the test locks the mistake in and defends it during review. No amount of coverage detects a correctly implemented wrong rule.

> **WARNING**
>
> A green unit suite is evidence about the cases you thought of. It is not evidence about the cases you did not, and it is never evidence that the deployment works. Treat "all tests pass" as one input to a release decision, alongside integration tests, a canary, and the ability to roll back.

A test is an asset with a maintenance liability

Every test you write is two things at once. It is an asset — it may catch a defect, it documents an intention, it lets someone refactor without fear. It is also a liability — it is code that must compile, be read, be understood when it fails, and be changed every time the thing it touches changes. Both sides are real, and the useful judgement is the difference between them:

```
value of a test  =  (bugs it would plausibly catch)
                  - (maintenance it demands)
                  - (false confidence it creates)
                  - (time it costs on every run)
```

Nothing in that expression is measurable in a spreadsheet, and you should be suspicious of anyone who offers you a number for it. But the terms are individually judgeable, and they explain decisions that otherwise look like heresy.

Test	Asset side	Liability side
Proration on day 16	Catches day-count bugs	Almost none
Getter returns field	None	Noise in every rename
Verifies 6 mock calls	Near zero	Breaks on every refactor
Tax at 190 permille	Catches rounding bugs	Low; rule is stable

Two consequences follow, and both are uncomfortable the first time.

First: deleting a test can be an improvement. A test whose asset side is empty — it asserts that a stub returned what the stub was told to return, or that a field it does not care about is still there — is pure liability. Removing it makes the suite smaller, faster and more honest. Coverage will drop. That is fine; Chapter 18 explains why coverage was never the thing you were protecting.

Second: not writing a test can be the right call. If the code has no branch, no arithmetic and no rule — a record with three components, a method that forwards one call — the plausible-bug term is close to zero and the maintenance term is not. Write the test for the tier boundary at 100,000 API calls instead. Testing is an engineering trade-off, and pretending otherwise produces suites nobody trusts and everybody maintains.

> **NOTE**
> "Delete it" and "skip it" are different decisions. A deleted test is an honest statement that the rule is not worth protecting. A skipped or retried test is a test that still looks present, still costs review time, and protects nothing. Chapter 22 shows what a retry button does to a team's instincts.

The definition this book uses

Everything above collapses into one paragraph, and the rest of the book depends on it.

> **TIP**
> **A unit test is an automated test that exercises one behavior of your own code, controls every input it depends on, runs entirely inside one process, and fails with a message that names the rule that broke.** How many classes it loads is not part of the definition.

Examine the clauses. *One behavior* gives you one reason to fail, and therefore a readable failure. *Your own code* excludes tests whose real subject is the framework, the driver or the payment provider — those are valuable and belong in Chapters 16 and 17. *Controls every input* delivers determinism, order-independence and machine-independence in one clause, and it is the clause that rules out the real clock, the real database and the real network. *One process* is what makes it fast enough to run without deciding to. *Names the rule that broke* is the clause nobody writes down and everybody needs at 5pm on a Friday.