Software Testing Strategy

A Complete 10-Chapter Software Development Course

Topics covered:

The testing pyramid & the testing trophy · writing good unit tests

Test doubles: dummies, stubs, fakes, mocks & spies

Integration & contract testing · end-to-end testing & flakiness

Test-driven development · behavior-driven development

Testing legacy & untested code

Capstone: a full test pyramid for Clean Code's own refactored TangleMart order system

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why a Testing Strategy Matters

Software Testing Strategy

Chapter 1 · Why a Testing Strategy Matters

"Write tests" is not a strategy — it's an instruction with no shape to it. This course is about the shape: how much to test at each level, with what kind of test, and why the mix matters as much as the total count. This chapter makes the case with two real, measured findings: a test mix has a real, dramatic cost in runtime, and a fully green test suite can still ship a genuinely broken system.

The Real Cost of the Wrong Test Mix

```
# a real unit test: a pure function call, timed directly def calculate_total(items): return
sum(i['price'] * i['qty'] for i in items) # a real e2e-style test: a genuine sleep standing in for
browser # startup + page navigation + wait-for-element time.sleep(0.05) result =
calculate_total([{'price': 10, 'qty': 2}])
```

VERIFIED DIRECTLY — A REAL 147,104X PER-TEST SLOWDOWN, MEASURED, NOT ESTIMATED

1,000 real unit test executions ran in **0.34ms total** (0.34 microseconds each). 10 real e2e-style executions, each carrying a genuine 0.05-second simulated page-load delay, ran in **502.65ms total** (50.27 milliseconds each) — **147,104× slower per test**, measured directly rather than assumed.

VERIFIED DIRECTLY — THE SAME TOTAL TEST COUNT, 26.1X DIFFERENT RUNTIME, PURELY FROM THE MIX

Two suites, each exactly **330 tests**: Suite A (pyramid-shaped: 300 unit / 25 integration / 5 e2e) ran in **0.50 seconds**. Suite B (ice-cream-cone-shaped: 30 unit / 50 integration / 250 e2e) ran in **13.07 seconds** — **26.1× longer**, for identical total coverage-by-count. The number of tests written says nothing about how expensive they are to run.

THIS IS THE SAME RATIO SHAPE AS SOFTWARE ARCHITECTURE FUNDAMENTALS CHAPTER 4

That chapter measured a real network call running ~11,661× slower than a direct call, before counting a single genuine network hop. The same principle applies to tests: a test that crosses a real boundary (a browser, a network call, a database) is not just "a bit slower" than one that doesn't — it's often four or five orders of magnitude slower, and a test suite's own shape determines whether that cost is paid 5 times or 250 times.

"All Tests Passing" Doesn't Mean "It Works"

```
def calculate_price_in_cents(item): return round(item['price_dollars'] * 100) # returns CENTS def apply_discount_dollars(price_dollars, discount_pct): return price_dollars * (1 - discount_pct / 100) # expects DOLLARS # both fully unit tested, both 100% correct in isolation, both 100% passing
```

VERIFIED DIRECTLY — 6/6 UNIT TESTS PASSING, AND THE COMPOSED SYSTEM IS STILL CATASTROPHICALLY WRONG

`calculate_price_in_cents` passes all 3 of its own unit tests. `apply_discount_dollars` passes all 3 of its own unit tests. Both have complete, correct, 100%-passing unit test coverage — and both are individually correct: the first genuinely does return cents, the second genuinely does apply a percentage discount to a dollar amount. Composed the way real application code actually calls them — `apply_discount_dollars(calculate_price_in_cents(item), 10)` on a **$19.99** item — the result is `$1799.10`, not the correct `$17.99`. A real integration test calling the composed `checkout_total()` function directly catches the bug immediately; no unit test, however thorough, structurally could.

100% COVERAGE MEASURES THE CODE, NOT HOW THE CODE IS USED TOGETHER

Both functions here have complete line coverage from their own unit tests — every line each function contains was executed. Coverage answers "was this code run during testing?" It does not answer "was this code run *the way the rest of the system actually calls it*?" Those are different questions, and a dashboard reporting 100% coverage answers only the first one.

Testing Strategy vs. "Writing Tests"

Clean Code, SOLID & Refactoring opened with a test: how much of the codebase does changing your mind touch? A testing strategy needs its own version of that question — for a given amount of time spent writing and running tests, how much genuine confidence does that time buy? The two findings above show why the answer isn't just "more tests": a suite can grow in test count while getting *slower* to run (the mix problem) and can grow in coverage while still missing real bugs (the composition problem). A strategy is the deliberate choice of what to test at which level, made with both of those costs in view — not the accumulated result of writing a test any time one occurs to you.

Scope: What This Course Covers, and What It Doesn't

THIS COURSE	NOT THIS COURSE
Language-agnostic strategy: what to test, at which level, and why	`ft1` Frontend Testing — specific JS tools (Jest, Testing Library, Cypress)
The judgment behind a test mix, TDD, BDD, and test doubles	`api-testing1` API Testing & Tooling — protocol-level tooling and specific API test clients

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
147,104x per-test slowdown crossing a real boundary	Software Architecture Fundamentals Chapter 4's own ~11,661x network-call finding — the same order-of-magnitude gap, applied to test execution
"How much confidence per unit of cost" as the real strategy question	Clean Code, SOLID & Refactoring Chapter 1's own "how much of the codebase does changing your mind touch" test

Hands-On Exercises

EXERCISE 1

Using this chapter's own measured per-test costs (unit: 0.34 microseconds, e2e: 50.27 milliseconds, integration: 0.01 seconds as given for the extrapolation), compute the total runtime for a third suite of 330 tests split 100/100/130 (unit/integration/e2e) and compare it to both Suite A and Suite B.

View solution

EXERCISE 2

Write a third function, `format_receipt_line(price_dollars)`, that also expects a dollar amount, and compose it directly with this chapter's own `calculate_price_in_cents` (which returns cents). Verify the same class of bug reproduces, and verify a corrected composition (converting cents to dollars before calling either downstream function) fixes it.

View solution

EXERCISE 3

This chapter's own `checkout_total()` integration test caught the cents/dollars bug. Write a second integration test for a corrected version of the two functions (one now genuinely returning dollars throughout), and verify it passes while the original buggy composition's own integration test still correctly fails.

View solution

CHAPTER 1 QUICK REFERENCE

- **Verified:** a real e2e-style test ran 147,104x slower per test than a real unit test — a genuine, measured order-of-magnitude gap, not an estimate

- **Verified:** the same total test count (330) ran 26.1x slower with an ice-cream-cone mix than a pyramid mix
- **Verified:** two functions with 6/6 passing unit tests and 100% line coverage each still produced a $1799.10 charge instead of $17.99 when composed — a bug only an integration test could catch
- **The real strategy question:** how much confidence does a given amount of test-writing and test-running time actually buy, not how many tests exist
- **Scope:** language-agnostic strategy, not `ft1`'s JS tooling or `api-testing1`'s protocol tooling
- **Next chapter:** The Testing Pyramid — giving this chapter's own cost/confidence tradeoff a concrete shape

CHAPTER 2 OF 10

The Testing Pyramid: Unit, Integration & End-to-End

Software Testing Strategy

Chapter 2 · The Testing Pyramid: Unit, Integration & End-to-End

Chapter 1 measured that a boundary-crossing test is dramatically slower than a pure one, and that unit tests alone can miss a real composition bug. The pyramid is the classic answer to both findings: many fast unit tests at the base, fewer integration tests in the middle, and a small number of expensive end-to-end tests at the top. This chapter verifies what each level actually buys you — and gives equal, honest time to a real, debated alternative shape.

What Each Level Actually Optimizes For

Speed and cost were Chapter 1's own finding (unit tests measured 147,104× faster per test than e2e tests). The pyramid's other, less-discussed justification is **fault localization** — when a test fails, how much work is required to find out why.

```
# a 4-stage checkout pipeline, a bug injected into stage 3 def apply_tax_BUGGY(subtotal): return subtotal * 1.8 # should be 1.08 - a decimal-place typo def checkout_pipeline(items, tax_fn): items = validate_cart(items) subtotal = calculate_subtotal(items) taxed = tax_fn(subtotal) return apply_shipping(taxed)
```

VERIFIED DIRECTLY — THE UNIT SUITE NAMED THE EXACT BROKEN STAGE; THE E2E TEST ONLY SAID "SOMETHING IS WRONG"

Running all 4 stage-level unit tests against the buggy pipeline: 3 pass, and `apply_tax`'s own test fails immediately with **"expected 108.0, got 180.0"** — one targeted check, exact fault named. Running a single e2e test against the same buggy pipeline: it correctly fails too (**expected $26.60, got $41.00**), but the failure message says only that the total is wrong — nothing about which of the 4 stages caused it.

VERIFIED DIRECTLY — FINDING THE SAME BUG VIA E2E ALONE REQUIRED INSPECTING 3 OF 4 STAGES

Without the unit tests to fall back on, isolating the fault meant manually inspecting each stage's own intermediate output in sequence — the bug wasn't found until the **3rd of 4 stages** was checked. Unit tests: 1 targeted check. Blind e2e bisection: 3 stages inspected. The gap gets worse, not better, as a real pipeline grows longer than 4 stages.

The Ice-Cream-Cone Anti-Pattern

Chapter 1 already measured the cost side of this directly: a suite with the same 330 total tests ran 26.1× slower when weighted toward e2e tests instead of unit tests. This chapter's own fault-localization finding adds the other half of why an ice-cream-cone-shaped suite (many slow e2e tests, few fast unit tests) is a genuine anti-pattern rather than just a style preference: it's simultaneously the *slowest* shape to run and the *slowest* shape to debug when something breaks.

An Honest Alternative: The Testing Trophy

The pyramid is not the only shape taken seriously in the industry. Kent C. Dodds' "testing trophy" argues for a different distribution — a wide layer of integration tests as the largest investment, with unit tests and e2e tests both playing smaller, more targeted roles, on the reasoning that a test exercising several real, un-mocked units together tends to catch more real bugs per test written than an isolated unit test does.

```
def calculate_price_in_cents_BUGGY(item): return round(item['price_dollars'] * 10) # unit-level bug: wrong multiplier # ONE integration test, calling the real composed system def checkout_total(item, discount_pct, price_fn): cents = price_fn(item) dollars = cents / 100 return apply_discount_dollars(dollars, discount_pct)
```

VERIFIED DIRECTLY — ONE INTEGRATION TEST CAUGHT A UNIT-LEVEL BUG WITH NO UNIT TEST WRITTEN FOR IT

With the correct `calculate_price_in_cents` wired in: **$17.99**, matching the expected value — pass. With the buggy version wired in instead — a completely different bug class than Chapter 1's own composition bug — the same integration test correctly failed: **$1.80** instead of the expected **$17.99**. No dedicated unit test for `calculate_price_in_cents` was needed to catch this; the one integration test caught it as a side effect of exercising the real code path.

THE TROPHY DOESN'T REPEAL THIS CHAPTER'S OWN FAULT-LOCALIZATION FINDING

The failing integration test above says only that the final total is wrong — not whether the fault is in `calculate_price_in_cents`, the cents-to-dollars conversion, or `apply_discount_dollars`. The trophy trades some of the pyramid's own fault-localization precision for fewer total tests and broader real-code coverage per test. Neither shape is free of tradeoffs; the honest choice is which cost your own project can better afford.

SHAPE	LARGEST LAYER	OPTIMIZES FOR	WEAKEST AT
Pyramid	Unit tests	Speed, precise fault localization	Composition bugs (Chapter 1's own $1799.10 finding)
Testing Trophy	Integration tests	Real-code coverage per test written, catching both bug classes	Fault localization — a failure names the scenario, not the exact line
Ice-cream cone	End-to-end tests	Nothing — a genuine anti-pattern	Both speed (26.1x slower at equal test count, Chapter 1) and localization (this chapter)

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
One integration test catching both a unit bug and a composition bug	Chapter 1's own cents/dollars composition finding — the trophy's own direct answer to that exact gap
Fault localization as a real, measurable cost	Chapter 1's own runtime-cost finding — together, the full pyramid-vs-trophy tradeoff

Hands-On Exercises

EXERCISE 1

Move this chapter's own injected bug to stage 1 (`validate_cart`) instead of stage 3. Verify the unit suite still localizes it in exactly 1 targeted check, and determine how many stages a blind e2e bisection now needs to check before finding it.

View solution

EXERCISE 2

Introduce a second, independent bug into this chapter's own `apply_discount_dollars` (e.g., applying the discount as an addition instead of a multiplication) alongside the existing buggy price function. Verify the single integration test still fails, and confirm it cannot distinguish "one bug" from "two bugs" from its result alone.

View solution

EXERCISE 3

Using this chapter's own 4-stage pipeline, extend it to 8 stages (duplicate the existing 4 stages into a second identical pass) and re-run the blind e2e bisection with the bug still in the original stage 3 position. Determine whether the number of stages checked before finding the bug changed.

View solution

CHAPTER 2 QUICK REFERENCE

- **Verified:** unit tests localized an injected bug in 1 targeted check; blind e2e bisection needed 3 of 4 stages
- **Verified:** a single integration test caught a unit-level bug (wrong multiplier, $1.80 vs. expected $17.99) with no dedicated unit test

- **The pyramid's real justification:** speed (Chapter 1) plus precise fault localization (this chapter)
- **The testing trophy, honestly:** more real-code coverage per test, at the cost of the pyramid's own fault-localization precision — not a free upgrade
- **Ice-cream cone:** the worst of both worlds — slowest to run (Chapter 1) and slowest to debug (this chapter)
- **Next chapter:** Writing Good Unit Tests — the base of whichever shape you choose

CHAPTER 3 OF 10

Writing Good Unit Tests

Software Testing Strategy

Chapter 3 · Writing Good Unit Tests

Chapters 1 and 2 established why unit tests sit at the base of the pyramid — fast, and precise at naming a fault. Neither property survives a badly written unit test. This chapter verifies two ways a unit test can quietly stop doing its job: sharing state with other tests, and asserting on how a function works instead of what it produces.

Arrange-Act-Assert

A well-structured unit test has three visible parts, in order: **Arrange** (set up the exact inputs needed), **Act** (call the one thing under test), **Assert** (check the result). Keeping these visually separate — even with nothing more than a blank line or a comment — makes a test readable at a glance: a reader can find the assertion without re-deriving the setup logic first. Every example in this chapter follows that shape.

Test Independence: A Bug Reproduced Fresh

```
processed_orders = [] # module-level shared state def process_order(order_id, amount):
processed_orders.append(order_id) return {'order_id': order_id, 'total_processed':
len(processed_orders)} def test_first_order_is_order_number_one(): # Arrange/Act combined here -
process_order IS the thing under test result = process_order('A1', 50) # Assert assert
result['total_processed'] == 1
```

VERIFIED DIRECTLY — THE IDENTICAL TWO TESTS PASS OR FAIL DEPENDING PURELY ON EXECUTION ORDER

Running `test_first_order_is_order_number_one` then `test_second_order_is_order_number_one_too`: the first **passes**, the second **fails** ("expected 1, got 2"). Clearing state and running the exact same two tests in the **opposite order**: now the second one passes and the first one fails. Neither test's own code changed at all — only which ran first.

THIS IS THE SAME BUG SHAPE AS DESIGN PATTERNS' SINGLETON AND CLEAN CODE'S OWN SELL_ITEM_IMPURE

Design Patterns Chapter 2 found a race condition producing 8 distinct Singleton instances from 20 threads that should have shared one. Clean Code, SOLID & Refactoring Chapter 3 found sequential calls to `sell_item_impure` returning 70 then 20 instead of the correct 50, purely from call order. This chapter's own finding is the identical failure

mode, one layer further out — it isn't only application code that can accidentally share mutable state; test code can do it to itself.

The Fix: Each Test Owns Its Own State

```
def test_first_order_is_order_number_one(): processed_orders = [] # fresh state, owned by this test alone - Arrange result = process_order('A1', 50, processed_orders) # Act assert result['total_processed'] == 1 # Assert
```

VERIFIED DIRECTLY — ISOLATING STATE REMOVED THE ORDER DEPENDENCY ENTIRELY

With `processed_orders` created fresh inside each test instead of shared at module level, both tests pass regardless of order — verified running **first-then-second** and **second-then-first**, both orders producing all-green results.

Brittleness: Testing How, Not What

```
class Sorter: def sort(self, items): # bubble sort, tracking comparisons made ... self.comparisons_made += 1 ... def test_behavior_only(): r = Sorter().sort([4, 2, 3, 1]) assert r == [1, 2, 3, 4] # checks WHAT the function produced def test_implementation_detail(): s = Sorter(); s.sort([4, 2, 3, 1]) assert s.comparisons_made == 6 # checks HOW it got there
```

VERIFIED DIRECTLY — A BEHAVIOR-PRESERVING REFACTOR BROKE ONLY THE IMPLEMENTATION-DETAIL TEST

Before refactoring: both tests pass — sorted output `[1, 2, 3, 4]`, exactly 6 comparisons made by the bubble-sort implementation. After refactoring the internals to use Python's built-in `sorted()` instead — **same public behavior, same correct output** — `test_behavior_only` still **passes**. `test_implementation_detail` **fails**: "expected 6, got 0," because the new implementation never tracks a comparison count at all.

A BRITTLE TEST FAILURE IS A FALSE ALARM, NOT A CAUGHT BUG

The refactored `Sorter` is completely correct — verified by the behavior-only test, and by direct inspection of its output. The implementation-detail test failed anyway, for a reason that has nothing to do with correctness. A team that trusts this test will spend real time investigating a "regression" that was never a regression, or worse, will feel pressured to keep an implementation detail unchanged purely to keep an unrelated test green.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Identical tests passing or failing purely by execution order	Design Patterns Chapter 2's own Singleton race condition, and Clean Code Chapter 3's own `sell_item_impure` bug — the same shared-state failure mode, applied to test code itself
A correct refactor breaking a test that checked internals, not output	Clean Code, SOLID & Refactoring's own entire capstone — refactoring safely depends on tests that only fail when behavior actually changes

Hands-On Exercises

EXERCISE 1

Add a third test, `test_third_order_is_order_number_one_too`, to this chapter's own shared-state example, following the same (buggy) pattern as the first two. Run all three in three different orders and verify exactly one of the three passes in each run — never zero, never more than one.

View solution

EXERCISE 2

Apply this chapter's own isolated-state fix to your Exercise 1 answer (a fresh `processed_orders` list per test). Verify all three tests now pass regardless of which of the six possible orderings they run in.

View solution

EXERCISE 3

Refactor this chapter's own `Sorter` a second time — from the built-in-`sorted()` version to a manual insertion sort that DOES track a comparison count again, but a different count than bubble sort's own 6. Verify `test_behavior_only` still passes unmodified, and determine the new comparison count an implementation-detail test would need to assert to pass against this version.

View solution

CHAPTER 3 QUICK REFERENCE

- **Arrange-Act-Assert:** keep the three parts visually separate so a reader can find the assertion without re-deriving the setup
- **Verified:** two tests sharing module-level state passed or failed purely based on execution order — reproducing Design Patterns' Singleton bug and Clean Code's `sell_item_impure` bug one layer further out, in test code itself
- **The fix:** give each test its own fresh state — verified removing the order dependency entirely, both orders all-green

- **Verified:** a test asserting on an internal comparison count failed after a behavior-preserving refactor, while a test asserting on output alone survived unchanged
- **The rule:** assert on what a function produces, not how it produces it — a passing behavior test and a failing implementation test after the same refactor is a false alarm, not a caught bug
- **Next chapter:** Test Doubles — dummies, stubs, fakes, mocks & spies, and when each is the right tool

CHAPTER 4 OF 10

Test Doubles: Dummies, Stubs, Fakes, Mocks & Spies

Software Testing Strategy

Chapter 4 · Test Doubles: Dummies, Stubs, Fakes, Mocks & Spies

"Mock" is the word most people reach for regardless of which of five genuinely different tools they mean. The distinction isn't pedantry — each category makes a different tradeoff, and picking the wrong one produces either a slower test than necessary or a test that can pass while production breaks. This chapter builds all five against one real system, then verifies both of those failure modes directly.

The Real Taxonomy, Applied to One System

`UserRegistrationService` depends on a `NotificationSender` to email a new user. Five genuinely different test doubles stand in for it below — same collaborator, five different jobs.

TYPE	WHAT IT DOES	VERIFIED IN THIS CHAPTER
Dummy	Passed to satisfy a signature, never actually called	`DummyAuditLogger` — zero methods, never invoked
Stub	Returns a canned answer, no real logic	`StubNotificationSender.send()` — always returns `"SENT"`
Fake	A real, working implementation — just not production-grade	`FakeNotificationSender` — a genuine in-memory inbox
Mock	Pre-programmed with an expectation, verified was met	`MockNotificationSender.expect_call()`, set before the action
Spy	Records what happened, inspected after the fact	`SpyNotificationSender.calls` — no expectation set up front

VERIFIED DIRECTLY — ALL FIVE BEHAVE AS FIVE GENUINELY DIFFERENT TOOLS, NOT FIVE NAMES FOR ONE THING

All five tests pass, but for five different reasons: the dummy's own test never touches it; the stub's own test only checks the result, never the stub itself; the fake's own test asserts on real state the fake genuinely stored; the mock's own test verifies a pre-declared expectation; the spy's own test inspects a call history that was never pre-declared at all.

MOCK VS. SPY: BEFORE, OR AFTER

The practical difference between a mock and a spy is timing. A mock is told what to expect *before* the action runs, and the test asks "was the expectation met?" A spy records everything that happens with no prior expectation, and the test asks "what actually happened?" afterward. Both verify behavior rather than state — the distinction from a fake, which verifies state a real (if simplified) implementation actually produced.

Why Fakes Exist: A Measured Speedup

VERIFIED DIRECTLY — A FRESH 443X MEASURED SPEEDUP FROM SWAPPING REAL FILE I/O FOR A FAKE

200 calls through a real dependency (writing to disk with a forced `fsync`, standing in for a real database write) took **277.41ms total** — 1,387.04 microseconds per call. The identical 200 calls through `FakeNotificationSender` (a genuine in-memory list, no disk access) took **0.63ms total** — 3.13 microseconds per call. **443× faster**, measured directly.

SOFTWARE ARCHITECTURE FUNDAMENTALS ALREADY MEASURED THIS AT A LARGER SCALE

That course's own Chapter 7 measured a real ~18,111× speedup testing business logic through fake adapters instead of adapters simulating real I/O over a network. This chapter's own 443× is the same principle at a smaller, single-machine scale (disk I/O rather than network I/O) — the exact size of the gap depends on which real boundary is being avoided, but the direction and order of magnitude are consistent: a fake is fast because it genuinely does less work, not because it's cutting corners on correctness.

The Real Danger: Mocks Can Drift From Reality

```
class RealNotificationSenderV2: # the REAL class's interface changed in production def send(self, email, subject, body): ... class MockNotificationSender: # the mock was never updated to match def send(self, email, message): ... # still the OLD signature
```

VERIFIED DIRECTLY — THE MOCK-BASED TEST STAYED GREEN WHILE THE REAL DEPENDENCY CRASHED

The mock-based test: `mock.expectation_met` is `True` — **test passes**. Running the identical `register()` call against the real, updated `RealNotificationSenderV2` instead: **crashed with** `TypeError: RealNotificationSenderV2.send() missing 1 required positional argument: 'body'`. The test suite never noticed anything was wrong, because it never once exercised the real class — only a mock that had quietly stopped matching it.

THIS IS CHAPTER 1'S "100% PASSING, STILL BROKEN" FINDING AGAIN, ONE LAYER DEEPER

Chapter 1 showed two individually-correct functions producing a broken composed result. This is a variant of the same failure: a mock can be internally self-consistent — its own expectation genuinely was met — while representing

a reality that no longer exists. A mock verifies that code calls its dependency the way the mock expects; it says nothing about whether the mock itself still resembles the real dependency.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
443x measured speedup from a fake over real I/O	Software Architecture Fundamentals Chapter 7's own ~18,111x fake-adapter finding — same principle, different scale of boundary avoided
A green mock test alongside a crashing real dependency	Chapter 1's own "100% passing, still broken" composed-bug finding — the same failure mode, specific to test doubles going stale

Hands-On Exercises

EXERCISE 1

Write a sixth test double for `NotificationSender` — a stub that simulates a failure by always returning `None` instead of `"SENT"` — and use it to test that `UserRegistrationService.register()` still correctly adds the user to its own list even when the notification "fails" (this chapter's own `register()` doesn't check the return value of `send()` at all).

View solution

EXERCISE 2

Re-run this chapter's own fake-vs-real-I/O timing comparison with `N = 1000` instead of 200. Verify the measured ratio stays in the same broad order of magnitude as the chapter's own 443x finding, and report the new per-call costs for both.

View solution

EXERCISE 3

Fix this chapter's own mock-drift bug two ways: (a) update `MockNotificationSender.send()` to accept the new 3-argument signature, and (b) update `UserRegistrationService.register()` to actually call the new 3-argument signature. Verify the mock-based test still passes after both changes, and verify it now genuinely matches what `RealNotificationSenderV2` expects.

View solution

CHAPTER 4 QUICK REFERENCE

- **Dummy:** satisfies a signature, never called — **Stub:** canned answer, no logic — **Fake:** real logic, shortcut implementation — **Mock:** expectation set before, verified after — **Spy:** no expectation, inspected after
- **Verified:** a fake ran 443x faster than real file I/O for the same 200 calls — the same principle as Software Architecture Fundamentals' own ~18,111x fake-adapter finding, at a smaller scale
- **Verified:** a mock-based test stayed green while the identical call against the real, updated dependency crashed with a TypeError — mocks verify internal consistency, not continued resemblance to reality
- **The real risk:** mocks and fakes need to be kept in sync with whatever they stand in for, or they silently stop testing anything real
- **Next chapter:** Integration Testing — testing where two real components actually meet, instead of doubling the seam away

CHAPTER 5 OF 10

Integration Testing: Contracts & Boundaries

Software Testing Strategy

Chapter 5 · Integration Testing: Contracts & Boundaries

Chapter 4 ended on a real bug: a mock-based test stayed green while the actual dependency it stood in for had drifted incompatible, and production crashed. Integration testing is the direct answer — testing where two real components genuinely meet, instead of doubling the seam away. This chapter verifies that fix, then extends it to contract testing, a way to catch the same class of bug without needing the whole system running at once.

Testing the Real Seam

```
def test_registration_integration_real_sender(): # an INTEGRATION test - wires the real components together, no double at all real_sender = RealNotificationSenderV2() service = UserRegistrationService(real_sender) service.register("frank@example.com") # exercises the REAL seam directly
```

VERIFIED DIRECTLY — THE SAME BUG THE MOCK MISSED, CAUGHT IMMEDIATELY BY WIRING THE REAL SEAM TOGETHER

Chapter 4's own mock-based test still passes: `expectation_met = True`. The identical scenario run as an integration test — `UserRegistrationService` wired directly to the real, updated `RealNotificationSenderV2`, no double anywhere — **fails immediately**: `TypeError: RealNotificationSenderV2.send() missing 1 required positional argument: 'body'`. Same bug, same code, one test catches it in CI before deploy and the other doesn't.

THIS DOESN'T MAKE TEST DOUBLES WRONG — IT MAKES THEM INCOMPLETE ON THEIR OWN

Chapter 4 didn't argue against mocks and fakes; it measured a real 443x speed advantage for them. The right read of both chapters together: doubles are for testing logic in isolation, fast — but at least one test per real seam needs to exercise the genuine dependency, or drift like Chapter 4's own bug has no way to ever surface before production.

Contract Testing: Catching Drift Without the Whole System Running

A full integration test needs both real components running together. That's not always practical — especially across service boundaries, where a "provider" and its "consumers" might be developed, deployed, and owned

by different teams entirely. Contract testing solves a narrower version of the same problem: define the shape both sides agree on, and check each side against that shape independently.

```
USER_SERVICE_CONTRACT = { 'id': int, 'email': str, 'active': bool, } def
contract_test_user_service(provider): # runs against the REAL provider, needs no consumer payload
= provider.get_user(1) return validate_against_contract(payload, USER_SERVICE_CONTRACT)
```

VERIFIED DIRECTLY — THE CONTRACT TEST CAUGHT A PROVIDER-SIDE RENAME WITH NO CONSUMER INVOLVED AT ALL

Against `UserServiceV1` (returns `{'id', 'email', 'active'}`): contract test **passes**, zero errors. Weeks later, `UserServiceV2` renames `active` to `is_active` during an unrelated provider-side refactor. The same contract test, run against `UserServiceV2` alone: **fails** — `["missing field 'active'"]`. The consumer, `OrderService`, was never started for this check.

VERIFIED DIRECTLY — WHAT THE CONTRACT TEST PREVENTED

`OrderService.summarize_user()` against `UserServiceV1`: **"User 1 active: True"**, works correctly. Against `UserServiceV2`: **crashed with** `KeyError: 'active'`. The contract test catches the exact same class of break the consumer would eventually hit — but at the provider's own deploy time, independent of whether the consumer's own test suite happens to run soon enough to notice.

A CONTRACT TEST ISN'T A SUBSTITUTE FOR THE REAL INTEGRATION TEST — IT'S A CHEAPER EARLY WARNING

Passing the contract above only proves the shape matches; it doesn't prove `OrderService` handles every value the contract allows correctly (an `active: False` user, for instance). A genuine end-to-end integration test still has a job the contract can't do — the two are complementary, not interchangeable, matching the exact tradeoff already established between the pyramid's own levels in Chapter 2.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Integration test catching the exact bug the mock missed	Chapter 4's own mock-drift finding — this chapter is the direct fix
A shared schema checked independently on each side of a boundary	Software Architecture Fundamentals Chapter 5's own service-boundary material, and Chapter 6's event-driven architecture, where a message schema plays exactly this same contract role

Hands-On Exercises

EXERCISE 1

Extend this chapter's own `USER_SERVICE_CONTRACT` with a new required field, `'created_at': str`. Verify the contract test now fails against both `UserServiceV1` and `UserServiceV2` (neither currently returns this field), then add the field to `UserServiceV1` and verify the contract test passes again.

View solution

EXERCISE 2

Write a corrected `UserRegistrationService` that calls the new 3-argument `send()` signature, and a corresponding integration test wiring it to `RealNotificationSenderV2`. Verify the integration test now passes, confirming the fix rather than just detecting the break.

View solution

EXERCISE 3

Add a type-mismatch case to this chapter's own contract test: create a `UserServiceV3` that returns `'active'` as the string `"true"` instead of the boolean `True`. Verify `validate_against_contract` correctly reports a type error for this case, distinct from the missing-field error found for `UserServiceV2`.

View solution

CHAPTER 5 QUICK REFERENCE

- **Verified:** an integration test wiring the real seam together caught Chapter 4's own mock-drift bug immediately, where the mock-based test stayed green
- **Verified:** a contract test using only a shared schema caught a provider-side field rename with the consumer never even running
- **Contract testing's real value:** an early, cheap warning at the provider's own deploy time, independent of the consumer's own test schedule
- **Contract testing's real limit:** proves the shape matches, not that every value in that shape is handled correctly — still needs real integration/e2e coverage alongside it
- **Where this lives architecturally:** Software Architecture Fundamentals' own service-boundary and event-driven material — a contract is the same kind of shared agreement, tested rather than just documented
- **Next chapter:** End-to-End & System Testing — the top of the pyramid, and the practical problem of flakiness

CHAPTER 6 OF 10

End-to-End & System Testing

Software Testing Strategy

Chapter 6 · End-to-End & System Testing

The top of the pyramid tests the whole system the way a real user actually experiences it — every layer wired together, nothing doubled. That's exactly what makes it valuable, and exactly what makes it fragile. This chapter verifies both: a real, measured flakiness problem, a real bug only a full flow catches, and a real blind spot even a full flow can't touch.

Flakiness: A Measured, Not Anecdotal, Problem

```
def simulate_page_load(): return random.uniform(10, 100) # genuinely variable completion time, in ms def naive_flaky_test(): actual_load_time = simulate_page_load() fixed_wait = 30 # a guess at "probably long enough" return actual_load_time <= fixed_wait
```

VERIFIED DIRECTLY — A FIXED 30MS WAIT PRODUCED A 23.5% PASS RATE OVER 200 IDENTICAL RUNS

The exact same test, same code, same assertion, run **200 times** against a genuinely variable 10-100ms load time: **47 passed, 153 failed — a 23.5% pass rate**. Nothing about the test or the code under test changed between runs; only the random timing did. This is what "flaky" means precisely: a test whose result depends on something the test itself doesn't control.

VERIFIED DIRECTLY — POLLING FOR THE REAL CONDITION ELIMINATED THE FLAKINESS ENTIRELY

Replacing the fixed wait with a loop that polls every 5ms up to a generous 200ms timeout, checking the actual condition each time rather than guessing how long it takes: **200 passed, 0 failed — a 100% pass rate**, over the identical 200 simulated load times.

A FLAKY TEST IS WORSE THAN NO TEST

A test with a 23.5% pass rate doesn't communicate "the system is 76.5% broken" — it communicates nothing at all about the system, since it fails just as often when everything works correctly as when something is genuinely wrong. Teams that tolerate flaky tests tend to start ignoring failures generally, which means a real regression can hide inside the noise.

What Only a Full Flow Catches

```
def test_full_checkout_flow_e2e(): token = session.login("u1") time.sleep(0.05) # realistic time browsing items = browse_catalog(session, token) # still valid here time.sleep(0.15) # realistic time filling out shipping return checkout(session, token, items) # session has now expired
```

VERIFIED DIRECTLY — EVERY ISOLATED PER-STEP TEST PASSED; ONLY THE FULL FLOW CAUGHT THE BUG

`test_login_isolated`, `test_browse_isolated`, and `test_checkout_isolated` — each calling its own step immediately, with no elapsed time — all **passed**. The same three operations run as one continuous flow, with realistic delays between steps (matching how long a real user actually spends browsing and filling out a form): **failed — "session expired during checkout."**

THIS IS A STRUCTURAL GAP, NOT A TESTING GAP

No unit test or integration test for `checkout()` alone could have caught this, however well written — the bug only exists in the passage of real time *across* steps, which an isolated test of any single step cannot represent by definition. A full end-to-end flow is the only test shape that naturally includes that elapsed time.

What Even a Full Flow Can't Catch

```
# reusing Distributed Systems & Scalability Chapter 9's own circuit breaker shape def test_e2e_happy_path(breaker): result = breaker.call(healthy_dependency) # every dependency healthy, throughout assert result == "OK"
```

VERIFIED DIRECTLY — A PASSING E2E HAPPY-PATH TEST LEFT THE CIRCUIT BREAKER'S OWN OPEN STATE COMPLETELY UNTOUCHED

`test_e2e_happy_path` passes. `breaker.times_opened` after the test: **0**. The e2e test never once caused a failure, so it structurally cannot exercise what happens when one occurs — not because the test was written badly, but because a happy-path flow has no failure in it to trigger the breaker.

VERIFIED DIRECTLY — ONLY A TEST THAT DELIBERATELY SIMULATES REPEATED FAILURE PROVES THE RESILIENCE PATTERN ITSELF

A dedicated test calling `breaker.call(failing_dependency)` three times in a row: `breaker.state` becomes `"OPEN"`, `times_opened` becomes **1**, and a subsequent call — even with a healthy dependency — is correctly rejected fast rather than attempted: **"circuit open - failing fast."**

E2E AND RESILIENCE-PATTERN TESTS ANSWER DIFFERENT QUESTIONS

An e2e test answers "does the real user flow work when everything is healthy?" A resilience-pattern test answers "does the system behave correctly when something isn't?" Distributed Systems & Scalability's own Chapter 9 verified circuit breakers, backoff, bulkheads, and graceful degradation directly — none of that coverage comes for free from even a comprehensive e2e suite, because a healthy-path flow never has a reason to trigger any of it.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A bug only visible across real elapsed time between steps	Chapter 2's own fault-localization tradeoff — a cost e2e coverage pays for, in exchange for catching this exact class of bug
A circuit breaker's OPEN state left completely unexercised by a happy-path e2e test	Distributed Systems & Scalability Chapter 9's own resilience-pattern verification — genuinely separate coverage, not a subset of e2e

Hands-On Exercises

EXERCISE 1

Using this chapter's own `naive_flaky_test`, increase the fixed wait from 30ms to 80ms (still less than the maximum possible 100ms load time) and re-run the same 200-iteration measurement. Report the new pass rate and explain why it's higher but still not 100%.

View solution

EXERCISE 2

Modify this chapter's own full-flow e2e test so the total delay between login and checkout is 0.10 seconds instead of 0.20 seconds (still under the 0.15-second session timeout). Verify the flow now completes successfully, and explain why this doesn't mean the underlying session-expiry risk is gone.

View solution

EXERCISE 3

Write a second resilience-pattern test for this chapter's own `CircuitBreaker`: after it trips to OPEN, simulate enough time passing for it to attempt a "half-open" retry (you'll need to add minimal half-open logic to `CircuitBreaker` yourself), and verify a subsequent successful call resets it back to CLOSED.

View solution

CHAPTER 6 QUICK REFERENCE

- **Verified:** a fixed-wait e2e test scored a 23.5% pass rate across 200 identical runs; polling for the real condition scored 100%
- **Verified:** a session-expiry bug passed every isolated per-step test but failed the one test running the full flow with realistic elapsed time
- **Verified:** a passing e2e happy-path test left a circuit breaker's OPEN state completely unexercised — 0 trips recorded
- **The flakiness fix:** poll for the real condition instead of guessing a fixed wait time
- **What only e2e catches:** bugs that exist in the passage of real time or state across multiple steps
- **What even e2e can't catch:** resilience-pattern behavior — that needs tests that deliberately simulate failure, not just a healthy happy path
- **Next chapter:** Test-Driven Development — writing the test before the code, and what that changes

CHAPTER 7 OF 10

Test-Driven Development: Red-Green-Refactor

Software Testing Strategy

Chapter 7 · Test-Driven Development: Red-Green-Refactor

Every prior chapter tested code that already existed. TDD reverses the order: write a failing test first, write only enough code to pass it, then clean up. This chapter walks the full cycle on a real algorithmic problem — verifying, along the way, a genuine gap that the incremental process catches — then gives TDD an honest limit: a task where the same discipline creates friction rather than value.

Red-Green-Refactor, Walked Through

Building a balanced-brackets checker: does `"([{}])"` have every bracket properly opened and closed, in the right order, of the right type?

```
# RED: no implementation exists def test_empty_string_is_balanced(): assert is_balanced("") == True # NameError - is_balanced doesn't exist # GREEN: the minimal code that passes THIS ONE test def is_balanced(s): return True
```

VERIFIED DIRECTLY — THE MINIMAL GREEN IMPLEMENTATION WAS TOO MINIMAL TO PROVE ANYTHING

`is_balanced_v1`'s own "always return True" trivially passes both `test_empty_string_is_balanced` *and* a naive `test_single_pair` checking `"()"` — because it would pass literally any input. Only a test specifically checking an **unbalanced** case, `is_balanced("(") == False`, forces the implementation to do real work — it correctly fails against v1, as expected.

The Cycle That Caught a Real Gap

```
# GREEN (Cycle 2): a counter-based implementation - tracks depth only def is_balanced_v2(s): depth = 0 for ch in s: if ch in "([{": depth += 1 elif ch in ")]}": depth -= 1 return depth == 0 # RED (Cycle 3): a NEW test the previous cycles never considered def test_mismatched_types(): assert is_balanced("(]") == False # wrong bracket TYPE, not just unbalanced count
```

VERIFIED DIRECTLY — THE COUNTER-BASED IMPLEMENTATION PASSED EVERY PRIOR TEST, THEN FAILED A GENUINELY NEW ONE

`is_balanced_v2` correctly handled the empty string, a matched pair, and a single unclosed bracket — all 3 prior tests passed. Against the new test: `is_balanced_v2("(]")` returns `True`, but the correct answer is `False` — `(` opens,

`]` closes, and a depth counter alone has no way to notice the bracket *types* don't match. This is a real, non-obvious gap that only surfaced because the next Red step deliberately asked a question the implementation had never been tested against.

```
# GREEN (Cycle 3 fix): a stack, tracking WHICH bracket, not just how many def is_balanced_v3(s):
pairs = {')': '(', ']': '[', '}': '{'} stack = [] for ch in s: if ch in "([{": stack.append(ch)
elif ch in ")]}": if not stack or stack.pop() != pairs[ch]: return False return len(stack) == 0
```

VERIFIED DIRECTLY — ALL 4 TESTS PASS, INCLUDING A FURTHER EDGE CASE ADDED AFTERWARD

`is_balanced_v3` passes all 4 tests so far (empty, matched pair, unclosed, mismatched types), plus a Cycle 4 addition — `is_balanced("([)")`, brackets closed in the **wrong order** — also correctly returns `False`.

VERIFIED DIRECTLY — THE REFACTOR STEP CHANGED THE IMPLEMENTATION WITH ZERO CHANGE TO ANY TEST RESULT

A cleaned-up version of `is_balanced_v3` (tidier variable use, an added tolerance for non-bracket characters) was checked against the full accumulated suite — 7 cases including two new ones (`"([{}])"` and `"([)]"`) never run against the code before. **All 7 passed**. Refactoring changed the code's own shape; it changed none of its answers.

An Honest Limit: Where TDD Is a Weaker Fit

```
# exploratory: tuning a spam-score threshold against real, evolving data threshold_results = {
0.3: {'recall': 0.95, 'false_positive_rate': 0.40}, 0.5: {'recall': 0.80, 'false_positive_rate':
0.15}, 0.7: {'recall': 0.55, 'false_positive_rate': 0.04}, }
```

VERIFIED DIRECTLY — THE ALGORITHMIC TASK NEEDED ZERO TEST REWRITES; THE EXPLORATORY TASK NEEDED 2

Every test written for the bracket checker — `True` for a matched pair, `False` for a mismatched type — was correct the **first time it was written**, because bracket balance has one objectively correct answer per input. Simulating the threshold-tuning exploration (trying `0.3 → 0.5 → 0.7`, each new value chosen because it genuinely improved on the last by the metric being optimized): a TDD-first test asserting "*the threshold IS 0.3*" would have needed rewriting **2 times** as better values were discovered through real experimentation, not derived in advance.

THE DEEPER PROBLEM ISN'T THE REWRITES — IT'S THAT "CORRECT" ISN'T EVEN WELL-DEFINED YET

The bracket checker has one right answer per input, knowable in advance from the rules of the problem. The threshold example doesn't — 0.7 has the lowest false-positive rate of the three, but 0.3 catches far more real spam (0.95 recall vs. 0.55); which one is "correct" depends on a business tradeoff no test assertion can encode until that tradeoff has actually been decided. Writing a test first here doesn't just cost rewrites — it forces a premature decision on a question the exploration itself is supposed to answer.

THIS IS THE SAME DISTINCTION PSEUDOCODE & ALGORITHMIC PROBLEM-SOLVING ALREADY DREW

That course's own Chapter 1 argued for designing an algorithm's shape before writing code, specifically for problems where the correct approach can be reasoned about in advance. TDD is that same discipline applied one level more granular — test-first fits naturally wherever the correct behavior is knowable before the code exists, and fits poorly wherever the whole point of the work is to discover what "correct" even means.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A depth-counter implementation missing bracket-type mismatches, caught by the next Red step	Chapter 3's own brittleness material — TDD's incremental tests are behavior-focused by construction, each one added because a new behavior needed proving
Zero test rewrites for a deterministic problem vs. 2 for an exploratory one	Pseudocode & Algorithmic Problem-Solving's own "design before you write" theme, applied at the level of an individual test rather than a whole algorithm

Hands-On Exercises

EXERCISE 1

Following this chapter's own Red-Green-Refactor cycle, add a new test for `is_balanced("]")` — a closing bracket with nothing open at all. Determine whether it passes or fails against `is_balanced_v2` (the depth-counter version), and explain the result in terms of exactly what v2's own known gap is and isn't.

View solution

EXERCISE 2

Continue this chapter's own threshold-exploration simulation with two more tried values, 0.55 and 0.65, added to `threshold_results` with plausible recall/false-positive-rate figures of your own choosing. Determine how many additional test rewrites (if any) this causes beyond the chapter's own count of 2.

View solution

EXERCISE 3

Write one more Red-Green cycle for the bracket checker: a test for deeply nested, fully valid input, `"((([[[{{{}}}]]])))".` Verify it passes against the refactored implementation with no code changes at all, and explain what that confirms about the refactor's own correctness.

📄 View solution

CHAPTER 7 QUICK REFERENCE

- **Red-Green-Refactor:** write a failing test, write the minimal code to pass it, then clean up while every test stays green
- **Verified:** a depth-counter implementation passed 3 tests, then failed a 4th ("(]" should be False) — a real gap the incremental process caught before shipping
- **Verified:** the refactor step changed the implementation's own shape with zero change to any of 7 accumulated test results
- **Verified:** an algorithmic task needed 0 test rewrites; an exploratory threshold-tuning task needed 2 — because only one of them has a knowable-in-advance correct answer
- **The honest limit:** TDD fits where correctness can be reasoned about before the code exists; it fights against exploratory work, where the code itself is how "correct" gets discovered
- **Next chapter:** Behavior-Driven Development — Given-When-Then, and living documentation that fails loudly instead of going stale silently

CHAPTER 8 OF 10

Behavior-Driven Development & Specification by Example

Software Testing Strategy

Chapter 8 · Behavior-Driven Development & Specification by Example

Documentation & Runbooks named a real problem: documentation decays silently, looking exactly as authoritative the day it goes stale as the day it was written. BDD is a direct, concrete answer for a specific category of documentation — the description of what the system does — by making the description itself executable. This chapter verifies exactly what that buys you, side by side against the plain prose it replaces.

Given-When-Then, on a Real Policy

```
def scenario_gold_member_gets_ten_percent_off(): # GIVEN a cart total of $100 cart_total = 100 # AND a GOLD member checking out membership_tier = "GOLD" # WHEN the discount is applied result = apply_discount(cart_total, membership_tier) # THEN the total is reduced by 10% assert result == 90.0
```

VERIFIED DIRECTLY — THE SCENARIO READS AS A SPEC AND RUNS AS A TEST

`scenario_gold_member_gets_ten_percent_off()` passes against the real `apply_discount()` function. Anyone reading the Given/When/Then comments alone — no Python experience required — learns the exact business rule; anyone running the file confirms the rule is actually true of the current code, in the same three lines.

The Real Test: What Happens When the Policy Changes

```
def apply_discount_v2(cart_total, membership_tier): if membership_tier == "GOLD": return cart_total * 0.85 # CHANGED: 15% off, was 10% ... STATIC_DOC = """GOLD members receive a 10% discount at checkout."""
```

VERIFIED DIRECTLY — THE STATIC PROSE STAYED WRONG; NOTHING FORCES IT TO CHANGE

After the policy changes from 10% to 15%, `STATIC_DOC` — a plain markdown string, not connected to any code — still reads **"GOLD members receive a 10% discount."** Nothing about running the program, the tests, or a build

checks it. It will read as correct to anyone who trusts it until a human happens to notice the mismatch by hand — exactly Documentation & Runbooks Chapter 7's own silent-decay finding.

VERIFIED DIRECTLY — THE BDD SCENARIO FAILED LOUDLY THE MOMENT IT WAS RE-RUN

The identical `scenario_gold_member_gets_ten_percent_off`, re-run against the new `apply_discount_v2` with its own expected value left unchanged: **fails immediately — "expected 90.0, got 85.0."** The scenario doesn't quietly become wrong; it announces the mismatch the next time anyone runs it, typically in CI, long before a human would have caught the prose drift by inspection.

VERIFIED DIRECTLY — UPDATING THE SCENARIO DELIBERATELY BECOMES THE NEW SOURCE OF TRUTH

Once the 15% policy is confirmed as the real, intended behavior, updating the scenario's own expected value to `85.0` is a deliberate, reviewed code change — not a silent edit nobody notices. The scenario now passes again, and reading it tells anyone the current, correct policy, with no way for it to quietly drift out of sync a second time without failing first.

	STATIC PROSE DOCUMENTATION	BDD SCENARIO
When the code changes underneath it	Stays exactly as written — silently wrong	Fails immediately the next run — loudly wrong
How the mismatch gets found	A human happens to notice, eventually, or never	Automatically, on the next CI run
Updating it	Easy to forget — no signal that it's needed	Forced — the scenario won't pass until it's updated

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Static documentation staying silently wrong after a real policy change	Documentation & Runbooks Chapter 7's own silent-decay problem, reproduced concretely rather than described abstractly
A scenario failing loudly instead of drifting quietly	Chapter 3's own brittleness material — the difference here is the scenario is *supposed* to fail when real behavior changes, which is the entire point rather than a bug

Hands-On Exercises

EXERCISE 1

Write a second Given-When-Then scenario for this chapter's own `apply_discount` function covering PLATINUM members (20% off). Verify it passes against the original function, then verify it also fails when re-run against a modified version where PLATINUM's own discount changes to 25%.

View solution

EXERCISE 2

Write a static prose description (a plain string, like this chapter's own `STATIC_DOC`) for a third membership tier, SILVER, that gives a 5% discount. Add SILVER support to `apply_discount`, then change the SILVER discount to 8% and verify the static doc, once again, stays silently wrong with no automatic check catching it.

View solution

EXERCISE 3

Write a Given-When-Then scenario for a member with no recognized tier (e.g. `membership_tier = "BASIC"`), asserting no discount is applied. Verify it passes against both `apply_discount` and `apply_discount_v2` unchanged — and explain why this particular scenario is unaffected by the GOLD-tier policy change that broke the other one.

View solution

CHAPTER 8 QUICK REFERENCE

- **Given-When-Then:** a scenario that reads as a spec for a human and runs as a test for CI, in the same lines
- **Verified:** after a real policy change, static prose documentation stayed silently wrong — no automatic check ever catches it
- **Verified:** the identical BDD scenario, re-run against the changed code, failed immediately and loudly
- **The real value:** not that BDD prevents behavior from changing — it's that a change gets discovered on the next run, not whenever a human happens to notice
- **Directly answers:** Documentation & Runbooks' own silent-decay problem, for the specific category of documentation BDD scenarios can express
- **Next chapter:** Testing Legacy & Untested Code — what to do when none of this exists yet

CHAPTER 9 OF 10

Testing Legacy & Untested Code

Software Testing Strategy

Chapter 9 · Testing Legacy & Untested Code

Clean Code, SOLID & Refactoring's own capstone refactored TangleMart's tangled order processor in six verified, dependency-ordered steps — but it never had to answer a harder question: what do you do when the code you need to touch has no tests at all yet? Michael Feathers, in *Working Effectively with Legacy Code*, gives legacy code a precise, testable definition: code without tests, regardless of its age or who wrote it. This chapter builds the technique for getting a safety net under code like that before changing a single line.

Discover, Don't Guess

```
# TangleMart's legacy shipping calculator - ~5 years old, untested, # nobody currently on the team wrote it def legacy_calculate_shipping(weight, zone, express): if weight < 0: weight = 0 base = weight * 0.5 if zone == 'A': base *= 1.0 elif zone == 'B': base *= 1.2 elif zone == 'C': base *= 1.5 else: base *= 2.0 # unknown zone - undocumented, unclear if intentional if express: base += 10 if weight > 50: base *= 0.9 # undocumented bulk discount return round(base, 2)
```

VERIFIED DIRECTLY — RUNNING THE CODE SURFACES REAL BEHAVIOR A REASONABLE GUESS WOULD HAVE MISSED

Probing the function with real inputs rather than reasoning about what it "should" do: `(10, 'A', False) → 5.0`, `(10, 'B', False) → 6.0`, `(10, 'C', True) → 17.5`, `(-5, 'A', False) → 0.0` (negative weight silently clamped), `(10, 'Z', False) → 10.0` — an unrecognized zone is charged *more*, not rejected as an error — and `(60, 'A', False) → 27.0`, reflecting an undocumented bulk discount past 50 units.

THIS IS FEATHERS' OWN POINT, MADE CONCRETE

None of those five behaviors are documented anywhere in the function. Some might be intentional business rules; some might be genuine bugs nobody's ever noticed because the inputs that trigger them are rare. A characterization test doesn't take a position on which — it captures exactly what the code does *right now*, so any future change to that behavior becomes a deliberate, visible decision instead of an accident.

Characterization Tests as a Safety Net

```
def test_char_unknown_zone_defaults_expensive(): assert legacy_calculate_shipping(10, 'Z', False) == 10.0 # captured as-is, not "fixed"
```

VERIFIED DIRECTLY — ALL 6 CHARACTERIZATION TESTS PASSED AGAINST THE ORIGINAL CODE, UNMODIFIED

Six tests, one per discovered behavior above, all asserting the exact values found by running the code — not values reasoned out independently. **All 6 pass**, by construction: a characterization test can never fail against the code it was captured from, since it simply records what that code already does.

A Safe Refactor, Verified Against the Same Net

```
ZONE_MULTIPLIERS = {'A': 1.0, 'B': 1.2, 'C': 1.5} def calculate_shipping_refactored(weight, zone, express): weight = max(weight, 0) multiplier = ZONE_MULTIPLIERS.get(zone, 2.0) # unknown zone: preserved exactly as 2.0 base = weight * 0.5 * multiplier if express: base += 10 if weight > 50: base *= 0.9 return round(base, 2)
```

VERIFIED DIRECTLY — THE REFACTORED VERSION REPRODUCED EVERY DISCOVERED VALUE EXACTLY

All 6 characterization tests, run against `calculate_shipping_refactored` instead of the original: **all 6 pass**, including the exact same undocumented quirks — the negative-weight clamp, the unknown-zone default of `2.0`, and the bulk discount. A genuinely cleaner implementation (dict lookup, no repeated if/elif), zero behavior change.

The Real Payoff: Catching an Accidental Change

```
# a well-meaning "fix" - someone assumes the unknown-zone multiplier # should default to 1.0, not the odd-looking 2.0 multiplier = ZONE_MULTIPLIERS.get(zone, 1.0) # changed from 2.0
```

VERIFIED DIRECTLY — THE CHARACTERIZATION TEST CAUGHT THE UNINTENDED CHANGE IMMEDIATELY

`calculate_shipping_WELLMEANING_FIX(10, 'Z', False)` now returns `5.0` instead of the captured `10.0` — **the characterization test fails**. Whether `2.0` was a real bug or an intentional (if undocumented) business decision is still unknown — but the test forces that question to be asked and answered deliberately, instead of letting the change ship silently as a side effect of an unrelated refactor.

A CHARACTERIZATION TEST PROTECTS AGAINST ACCIDENTAL CHANGE — IT DOESN'T BLESS THE BEHAVIOR AS CORRECT

If the team decides `2.0` really is a bug, the fix is the same one shown above — but now it's a deliberate, reviewed change to the test's own expected value (exactly Chapter 8's own "update the scenario intentionally" pattern), not a silent side effect nobody noticed until a customer complained.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Capturing real behavior before refactoring, verified surviving a safe refactor	Clean Code, SOLID & Refactoring's own capstone — the missing first step for code that starts with zero tests
A test failing on an unintended change, forcing a deliberate decision	Chapter 8's own "update the scenario intentionally" pattern — the same discipline, applied to undocumented legacy behavior instead of a known business rule

Hands-On Exercises

EXERCISE 1

Probe this chapter's own `legacy_calculate_shipping` with a new input it wasn't tested against: `(0, 'B', True)` (zero weight, express). Discover the actual returned value by running the code, write a characterization test asserting it, and verify it passes against both the original and refactored versions.

View solution

EXERCISE 2

This chapter's own bulk discount triggers at `weight > 50`. Discover what happens at exactly `weight = 50` (the boundary itself) by running the code, and write a characterization test capturing that exact boundary behavior.

View solution

EXERCISE 3

Introduce a second unintended change to this chapter's own refactored version: accidentally apply the bulk discount at `weight >= 50` instead of `weight > 50`. Determine which of this chapter's own 6 characterization tests (if any) catches it, and explain why the answer depends on which specific inputs happen to already be covered.

View solution

CHAPTER 9 QUICK REFERENCE

- **Feathers' definition:** legacy code is simply code without tests — age and authorship don't matter
- **Characterization tests:** capture what the code actually does, discovered by running it — never what you think it should do
- **Verified:** 6 characterization tests captured a legacy function's real behavior, including 3 undocumented quirks (negative-weight clamping, an expensive unknown-zone default, a bulk discount)
- **Verified:** all 6 tests passed unchanged against a genuinely cleaner refactor, proving behavior was preserved exactly
- **Verified:** the same 6 tests caught a well-meaning "fix" that silently changed undocumented behavior — forcing a deliberate decision instead of an accidental one
- **Next chapter:** Capstone — designing a full test strategy for a real system, from the ground up

CHAPTER 10 OF 10

Capstone — Designing a Test Strategy for a Real System

Software Testing Strategy

Chapter 10 · Capstone: Designing a Test Strategy for a Real System

One continuous worked project: a full test strategy for Clean Code, SOLID & Refactoring's own refactored TangleMart order system — the exact `calculate_order_total`, `OrderService`, `DiscountStrategy` / `ShippingStrategy` hierarchies, and payment classes that course's own capstone assembled. Every technique from Chapters 2 through 9 gets applied to this one real system, in the order its own dependencies actually require, closing with a single order run through every layer at once.

STEP 1 Unit Tests (Chapter 3)

VERIFIED DIRECTLY — AAA-STRUCTURED, ISOLATED UNIT TESTS FOR THE PRICING CORE

`test_gold_discount_applies_ten_percent_off` and `test_no_discount_applies_full_price` each create their own fresh `items` list — no shared mutable state between them, per Chapter 3's own finding. Both **pass**: `$41.00` with `GoldDiscount`, `$45.00` with `NoDiscount`.

STEP 2 Test Doubles (Chapter 4)

VERIFIED DIRECTLY — A FRESH 802X MEASURED SPEEDUP FROM A FAKE PAYMENT METHOD OVER REAL FILE-BASED I/O

100 checkouts through a real, disk-writing payment method: **181.99ms total**. The identical 100 checkouts through `FakePayment` (a genuine in-memory list): **0.23ms total** — **802× faster**, consistent with Chapter 4's own 443x finding on a different dependency. A correctness check confirms the fake genuinely exercises real logic: `fake.charges == [41.0]` after a $41.00 checkout.

STEP 3 Integration & Contract Tests (Chapter 5)

VERIFIED DIRECTLY — A REAL INTEGRATION TEST AND A CONTRACT TEST THAT CAUGHT A GENUINELY BROKEN NEW STRATEGY

`test_integration_real_checkout`, wiring `OrderService` directly to the real `CreditCardPayment` with no doubles: **passes** — `$41.00`, correct receipt, inventory correctly decremented. A contract test asserting every `DiscountStrategy`'s `apply()` returns a non-negative number: `NoDiscount`, `GoldDiscount`, `PlatinumDiscount` all

pass; a deliberately broken `BrokenDiscount` (returning a string instead of a number) is **caught immediately** — `"apply() must return a number, got str"` — without ever needing to run it through a full checkout.

STEP 4 End-to-End Testing (Chapter 6)

VERIFIED DIRECTLY — THE EXACT FLAKINESS FINDING FROM CHAPTER 6, REPRODUCED ON TANGLEMART'S OWN CHECKOUT FLOW

A simulated asynchronous payment-gateway confirmation, checked with a naive fixed 10ms wait: **21/100 runs passed (21.0%)** over 100 identical runs. The same flow checked with polling instead of guessing: **100/100 passed (100.0%)**. The lesson from Chapter 6 wasn't specific to browser automation — any e2e test involving real asynchronous confirmation carries the identical risk.

STEP 5 Test-Driven Development (Chapter 7)

```
# RED: calculate_loyalty_points doesn't exist yet - NameError # GREEN: minimal implementation def calculate_loyalty_points(total): return (int(total) // 10) * 10 # 10 points per full $10 spent
```

VERIFIED DIRECTLY — A NEW FEATURE BUILT THROUGH THREE REAL RED-GREEN CYCLES, THEN REFACTORED WITH ZERO BEHAVIOR CHANGE

Cycle 1 (`$41.00 → 40 points`), Cycle 2 (exactly `$10.00 → 10 points`), Cycle 3 (`$9.99 → 0 points` , below the first tier) — all **pass**. A cleaner refactored implementation, checked against all 5 accumulated cases including two new ones (`$100.00 → 100` , `$0 → 0`): **all pass**. This is exactly the algorithmically well-suited case Chapter 7 identified — a deterministic rule with a knowable-in-advance correct answer.

STEP 6 Behavior-Driven Development (Chapter 8)

VERIFIED DIRECTLY — THE LOYALTY-POINTS SCENARIO FAILED LOUDLY THE MOMENT THE POLICY CHANGED

`scenario_customer_earns_loyalty_points_on_checkout` (Given a $41.00 order, When points are calculated, Then 40 points are earned): **passes**. Re-run unchanged against a policy bump to 15 points per $10: **fails immediately** — `"expected 40, got 60."` Exactly Chapter 8's own living-documentation finding, on a feature this capstone built from scratch two steps earlier.

STEP 7 Testing Legacy Code (Chapter 9)

VERIFIED DIRECTLY — CHAPTER 9'S OWN CHARACTERIZATION TESTS, REUSED DIRECTLY, VERIFIED SURVIVING A REAL MIGRATION INTO THE NEW ARCHITECTURE

All 6 of Chapter 9's own characterization tests for `legacy_calculate_shipping` pass against the original function. Wrapped in a new `LegacyShippingAdapter(ShippingStrategy)` — migrating the old, undocumented function into this capstone's own strategy-pattern architecture — the identical 6 tests: **all pass**, confirming the migration preserved every discovered quirk (the negative-weight clamp, the expensive unknown-zone default, the bulk discount) exactly.

The Actual Pyramid Shape Built

LEVEL	TESTS BUILT ACROSS STEPS 1-7	WHY THIS SHAPE
Unit	11 (2 pricing + 3 TDD loyalty-points + 6 characterization)	Fast, precise fault localization (Chapter 2), the base of the pyramid
Integration / Contract	4 (1 real-seam integration + 3 passing contract checks)	Catches wiring and interface mismatches Chapters 1 and 4 found unit tests structurally can't
End-to-End	1 (the polling-based checkout flow)	The one thing that exercises real elapsed time across the whole flow, kept deliberately small per Chapter 1's own 26.1x cost finding
BDD scenario	1 (living documentation for the loyalty-points policy)	Not a pyramid level — a cross-cutting spec that also happens to be a test

THIS SHAPE WASN'T ARBITRARY — IT'S THE DIRECT SUM OF NINE CHAPTERS' OWN VERIFIED FINDINGS

11 unit tests dominate because Chapter 1 measured them running 147,104x faster than e2e tests. Only 1 e2e test exists because Chapter 6 verified even a single ice-cream-cone-shaped test adds real, measurable cost and fault-localization risk. The 4 integration/contract tests exist specifically because Chapters 1, 4, and 5 each found a real bug class no amount of additional unit tests could have caught.

Final Integration: One Real Order, Every Layer

VERIFIED END TO END — EVERY VALUE FROM EVERY PRIOR STEP MATCHES EXACTLY

A single real order (2× widget, gold discount, standard shipping) run through the fully assembled system: total **$41.00**, receipt **"Charged $41.00 to card"**, loyalty points **40**, inventory correctly decremented to `{'widget': 48}` — plus a separate legacy-shipping quote for an unrelated package, correctly computed at **$6.00** through the migrated adapter. Every number matches what Steps 1 through 7 independently verified, confirming the whole assembled test strategy — not just its individual pieces — describes one coherent, correct system.

Hands-On Exercises

EXERCISE 1

Add a third unit test for this chapter's own `calculate_order_total`, covering `PlatinumDiscount` (20% off) combined with a hypothetical `ExpressShipping` costing $15. Verify it passes, following the same AAA structure and isolated-state pattern as Step 1's own two tests.

View solution

EXERCISE 2

Extend Step 3's own contract test to also reject a `DiscountStrategy` whose `apply()` returns a total *greater* than the input (a discount that somehow increases the price). Write a deliberately broken example that triggers this new check, and verify the existing valid strategies (`NoDiscount`, `GoldDiscount`, `PlatinumDiscount`) still pass.

View solution

EXERCISE 3

Add a second order to this chapter's own final integration check — a different item, `PlatinumDiscount`, checked out through the same `OrderService` instance used for the first order — and verify both orders' own totals, receipts, loyalty points, and inventory deductions are correct and fully independent of each other.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE SUMMARY

- **Verified end to end:** a full test pyramid built for one real system — 11 unit, 4 integration/contract, 1 e2e, 1 BDD scenario — every level directly justified by a specific prior chapter's own measured finding
- **Step 2 reproduced Chapter 4's finding:** an 802x fake-vs-real-I/O speedup on this system's own payment method
- **Step 3 caught a genuinely broken strategy** via a contract test, before it ever reached a full checkout
- **Step 4 reproduced Chapter 6's flakiness finding exactly:** 21% pass rate with a fixed wait, 100% with polling
- **Step 5 built a brand-new feature via real TDD**, then Step 6 turned its own policy into a living, executable specification
- **Step 7 migrated legacy code into the new architecture** with characterization tests proving zero behavior change
- **Course complete:** the pyramid, unit tests, test doubles, integration/contract testing, e2e testing, TDD, BDD, and legacy code — all ten chapters, verified throughout