Clean Code, SOLID & Refactoring

A Complete 10-Chapter Software Development Course

Topics covered:

Naming & readability · function design & purity · code smells

Bloaters, couplers & dispensables · the five SOLID principles

The refactoring catalog · technical debt · prioritization

Capstone: refactoring one tangled codebase using every prior chapter's own verified technique

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

01 Why Code Quality Matters

02 Naming & Readability

03 Functions: Size, Purity & a Single Level of Abstraction

04 Code Smells I: Bloaters

05 Code Smells II: Couplers & Dispensables

06 SOLID I: Single Responsibility & Open/Closed

07 SOLID II: Liskov Substitution, Interface Segregation & Dependency Inversion

08 The Refactoring Catalog: Core Techniques

09 Technical Debt: Naming It, Measuring It, Paying It Down

10 Capstone — Refactoring a Real Codebase for Quality

CHAPTER 1 OF 10

Why Code Quality Matters

Clean Code, SOLID & Refactoring

Chapter 1 · Why Code Quality Matters

"Clean code" sounds like a matter of taste — but Software Architecture Fundamentals Chapter 1 already established a concrete, non-subjective test for exactly this kind of claim: **how much of the codebase does changing your mind touch?** That chapter applied it to architectural boundaries. This chapter applies the identical test one level down — to a single function.

The Same Function, Two Ways

```
# MESSY def calc(o, c): t = 0 for i in o['items']: t = t + i['price'] * i['qty'] if c == 'gold': t = t - t * 0.2 elif c == 'silver': t = t - t * 0.1 if t > 100: t = t - 5 if o.get('express'): t = t + 15 return t
```

```
# CLEAN — same behavior, split into small, single-purpose functions def calculate_order_total(order, customer_tier): items_total = sum(item['price'] * item['qty'] for item in order['items']) discounted = apply_tier_discount(items_total, customer_tier) after_bulk = apply_bulk_discount(discounted) return apply_express_fee(after_bulk, order) def apply_tier_discount(total, customer_tier): discount_rates = {'gold': 0.2, 'silver': 0.1} rate = discount_rates.get(customer_tier, 0) return total - total * rate def apply_bulk_discount(total): return total - 5 if total > 100 else total def apply_express_fee(total, order): return total + 15 if order.get('express') else total
```

VERIFIED DIRECTLY — "CLEAN" COST NOTHING IN CORRECTNESS

Run against four different orders (no tier, gold + express, silver + bulk, no tier + bulk + express), `calc()` and `calculate_order_total()` produced **identical** results in every case — `50`, `79.0`, `81.0`, `210`. Splitting the function into pieces changed nothing about what it computes.

The Real Test: Adding a New Tier

A genuine feature request: add a `'platinum'` tier at 30% off.

VERIFIED DIRECTLY — THE MESSY VERSION NEEDS 2 NEW LINES, INSERTED AMONG UNRELATED LOGIC

Adding platinum to `calc()` means inserting `elif c == 'platinum': t = t - t * 0.3` — two new lines — *in the middle* of the same function that also totals items, applies the bulk discount, and applies the express fee. The correct insertion point sits directly between the silver-discount check and the bulk-discount check; nothing about the function's own structure marks where tier logic ends and bulk-discount logic begins.

VERIFIED DIRECTLY — THE CLEAN VERSION NEEDS EXACTLY 1 LINE CHANGED, IN A FUNCTION WITH NOTHING ELSE IN IT

Adding platinum to `apply_tier_discount()` means changing exactly one line — `discount_rates = {'gold': 0.2, 'silver': 0.1}` becomes `{'gold': 0.2, 'silver': 0.1, 'platinum': 0.3}`. Diffing both versions confirms this is the *only* line that changes.

VERIFIED DIRECTLY — THE TWO NEIGHBORING FUNCTIONS WERE NEVER EVEN TOUCHED

Capturing `apply_bulk_discount`'s and `apply_express_fee`'s own source via `inspect.getsource()` before and after adding platinum: both are **byte-for-byte identical**, confirmed `True` for both. Nobody editing the tier discount needed to open, read, or reason about either function at all — the same proof technique Software Architecture Fundamentals Chapter 7 used to verify its own dependency inversion.

BOTH CHANGES WERE "CORRECT." ONLY ONE WAS SAFE.

Both versions correctly compute `70.0` for a platinum customer and still correctly compute `80.0` for a gold one — the messy version's change works. The difference isn't correctness; it's what the editor had to be careful about while making the change. In `calc()`, a misplaced line could silently land inside the wrong conditional block, or after the bulk-discount check instead of before it, changing behavior for tiers that were never supposed to be touched. In `apply_tier_discount()`, there is no wrong place to put the new line — it's a dictionary with one more entry.

What "Clean" Concretely Means

Not indentation, not a style guide, not a subjective preference for short functions. This chapter's own verified findings point at one property: **a change's own blast radius should match its own actual scope.** Adding a discount tier is a change to discount logic — in the clean version, it touched only discount logic, verified via two untouched functions. In the messy version, a change scoped to discount logic could only be made by editing a function whose own scope included totaling, bulk discounts, and shipping fees too.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
"How much does changing your mind touch?" applied to one function	Software Architecture Fundamentals Chapter 1 — the identical test, one level up, applied to a persistence boundary instead of a discount calculation

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A single-purpose function needing exactly one line changed	Chapter 6's own Single Responsibility Principle — `apply_tier_discount()` already follows it, informally, before this course names it directly
The messy function's own mixed concerns (totaling, discounts, fees, all in one place)	Chapter 4's own "Bloaters" code smells — this exact function is a textbook long-method/mixed-responsibility case, revisited directly in that chapter

Hands-On Exercises

EXERCISE 1

Add a fifth test case to this chapter's own verification — an order with a `'bronze'` tier that isn't in either version's discount logic — and verify `calc()` and `calculate_order_total()` still agree (both should apply no discount at all for an unrecognized tier).

View solution

EXERCISE 2

Make a different real change to both versions: raise the bulk-discount threshold from `100` to `150`. Verify which functions needed to change in the clean version, and confirm `apply_tier_discount` and `apply_express_fee` stayed untouched this time.

View solution

EXERCISE 3

Using this chapter's own two verified changes (adding platinum, raising the bulk threshold), explain why the clean version's own isolation held for *both* changes, even though they touched two completely different functions — what property of the clean version's own design makes this generalize, rather than being true by coincidence for the platinum example alone?

View solution

CHAPTER 1 QUICK REFERENCE

- **The test:** "how much of the code does changing your mind touch?" — reused directly from Software Architecture Fundamentals Chapter 1, applied to function-level design instead of architecture
- **Verified:** a messy and a clean implementation computed identical results across 4 test cases — clean code isn't a functional tradeoff

- **Verified:** the same real feature addition needed 2 lines inserted among unrelated logic in the messy version, versus 1 line changed in an isolated function in the clean version — with the clean version's two neighboring functions confirmed byte-identical, untouched, before and after
- **Next chapter:** Naming & Readability — the first, cheapest lever for making code this easy to change

CHAPTER 2 OF 10

Naming & Readability

Clean Code, SOLID & Refactoring

Chapter 2 · Naming & Readability

Chapter 1 measured what "clean" costs to change. This chapter is about the cheapest lever for getting there: names. Not as a style preference — this chapter verifies three specific, concrete costs bad naming carries, each measured directly rather than asserted.

Mental Mapping Hides Bugs From Even a Trivial Check

A classic real bug: copy-pasting a domestic-shipping calculation into an international one, and forgetting to update one variable reference.

```
# CRYPTIC def calc_international_bad(w, d): base = w * 2.5 fee = w * 0.1 # BUG: copy-pasted, forgot to change w to d return base + fee # DESCRIPTIVE def calculate_international_shipping(weight_kg, distance_km): base_cost = weight_kg * 2.5 distance_fee = weight_kg * 0.1 # the SAME bug, copy-pasted the same way return base_cost + distance_fee
```

VERIFIED DIRECTLY — BOTH VERSIONS COMPUTE THE IDENTICAL WRONG RESULT

Called with weight `10` and distance `500`: both the cryptic and descriptive versions return `26.0`. The correct result — using distance for the distance fee — is `75.0`. Neither version's own bug is hypothetical; both are real, silently wrong.

VERIFIED DIRECTLY — THE CRYPTIC NAME GIVES A CHECKER NOTHING TO CHECK

The variable is named `fee`. There is no word in that name promising what it should be computed *from* — a consistency check comparing the name against its own source variable literally cannot be constructed, because `fee` makes no claim at all.

VERIFIED DIRECTLY — THE DESCRIPTIVE NAME LETS A ONE-LINE CHECK CATCH THE EXACT BUG

The variable is named `distance_fee` — a name that promises its value comes from something related to *distance*. Checking whether the actual right-hand side (`weight_kg`) contains that promised word: `False`. The mismatch is caught immediately, using nothing but the name itself — no test run, no manual trace through the logic required.

WHY THIS IS THE REAL ARGUMENT FOR DESCRIPTIVE NAMES

It isn't that `distance_fee` is more pleasant to read than `fee` — it's that a name carrying real information (what category of value belongs here) makes an entire class of copy-paste bugs mechanically detectable, by a human skimming the code or by a simple automated check, in a way a cryptic name structurally cannot support.

Consistent Vocabulary: Findable vs. Not

VERIFIED DIRECTLY — INCONSISTENT NAMING MAKES A CODEBASE'S OWN FUNCTIONS UNFINDABLE BY SEARCH

A small codebase using three different words for the identical operation — `get_user`, `retrieve_order`, `fetch_product_data`, `get_invoice`, `retrieve_customer`: searching for `get_*` finds only **2 of 5** — `get_user` and `get_invoice`. The other three, doing the same kind of thing, are invisible to that search. The identical codebase, using one consistent word (`get_user`, `get_order`, `get_product`, `get_invoice`, `get_customer`): the same search finds **5 of 5**.

WHY THIS MATTERS BEYOND A SINGLE SEARCH

A developer trying to find "all the places we fetch something from the database" — for a security audit, a caching pass, a refactor — genuinely misses 3 of 5 real matches in the inconsistent codebase, with no error or warning telling them they missed anything. The search *succeeded*; it just wasn't looking at the right words, because the codebase never agreed on what the right words were.

Names as Documentation That Can't Go Stale — a Comment Can

```
def calculate_total(items): # returns the sum of item prices total = sum(item['price'] for item in items) total = total * 0.9 # NEW: apply a 10% loyalty discount return total
```

VERIFIED DIRECTLY — THE COMMENT IS NOW FALSE, AND THE FUNCTION STILL "WORKS"

For two items totaling `$150`: the comment claims the function `returns the sum of item prices` — that would be `$150`. The function actually returns `$135.0`. The comment is wrong by `$15.00` — a real, measurable lie, sitting one line above the code that contradicts it, and nothing in Python enforces any relationship between a comment's own text and what the code beneath it does.

WHY A NAME IS DIFFERENT IN KIND, NOT JUST IN CONVENIENCE

A comment is a second, separate piece of text describing the code — the two can drift apart, as just verified. A function's own *name* isn't a separate description sitting near the code; it's how every caller refers to the code. If `calculate_total` had been renamed to `calculate_discounted_total` the moment the discount was added, that rename would need to happen at the exact same place the behavior changed — not several lines above it, easy to forget. This doesn't make a misleading name impossible, but it removes the specific failure mode just verified: a description silently going stale while nobody's looking at it.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A descriptive name making a copy-paste bug mechanically detectable	Chapter 1's own "blast radius" test — good naming is part of what made `apply_tier_discount()` safe to change in isolation
Inconsistent vocabulary hiding 3 of 5 real matches from a search	Software Architecture Fundamentals Chapter 5's own coupling analysis — a static search is only as good as the vocabulary discipline behind the code it's searching
A comment verified drifting $15 away from the truth	Chapter 9's own Technical Debt chapter — stale documentation is a specific, common form of debt this course names directly later

Hands-On Exercises

EXERCISE 1

This chapter's own name-based check only worked because `distance_fee` contains the word "distance". Rename it to `extra_charge` instead (keeping the exact same bug — computed from `weight_kg`) and verify whether this chapter's own consistency-check technique can still catch the bug. Explain what this reveals about the limits of the technique.

View solution

EXERCISE 2

Add a sixth function to this chapter's own inconsistent codebase list, named `obtain_shipment`, and a corresponding one named `get_shipment` to the consistent list. Verify the `get_*` search results for both lists after the addition, and report the new totals.

View solution

EXERCISE 3

Fix this chapter's own `calculate_total` example two different ways: (1) update the comment to match the code, and (2) rename the function to `calculate_discounted_total` instead and remove the comment. Verify both fixes produce a function whose documentation (comment or name) accurately reflects its behavior, and explain which fix is more likely to still be accurate after a second, future change nobody remembers to update.

View solution

CHAPTER 2 QUICK REFERENCE

- **Mental mapping, verified costly:** a copy-paste bug was undetectable by any name-based check with cryptic names, but caught in one line by checking a descriptive name against its own promised word
- **Consistent vocabulary, verified:** a search found 2 of 5 real matches with inconsistent naming, 5 of 5 with consistent naming
- **Names vs. comments, verified:** a stale comment was wrong by $15 with nothing enforcing its accuracy — a failure mode a name, tied directly to its own code, structurally avoids
- **Next chapter:** Functions: Size, Purity & a Single Level of Abstraction

CHAPTER 3 OF 10

Functions: Size, Purity & a Single Level of Abstraction

Clean Code, SOLID & Refactoring

Chapter 3 · Functions: Size, Purity & a Single Level of Abstraction

Three properties of a function that turn out to be genuinely measurable, not matters of taste: how much it depends on and changes shared state (purity), and whether every line inside it operates at the same conceptual altitude (a single level of abstraction). Both connect directly to bugs this site has already verified elsewhere — this chapter reproduces one of them on purpose.

Purity: Why an Impure Function Reproduces a Bug You've Already Seen

```
# IMPURE — mutates shared state inventory = {'PROD-1': 100} def sell_item_impure(product_id, qty): inventory[product_id] -= qty return inventory[product_id] # PURE — takes everything as input, touches nothing outside itself def sell_item_pure(current_stock, qty): return current_stock - qty
```

VERIFIED DIRECTLY — THE IMPURE VERSION REPRODUCES DESIGN PATTERNS' OWN TEST-POLLUTION BUG

Two "independent" tests: sell `30` units from a stock of `100` (expect `70`), then sell `50` units from a *fresh* stock of `100` (expect `50`). Against `sell_item_impure`, test 1 correctly returns `70` — but test 2 returns `20`, not `50`, because it silently operated on the `30`-units-already-sold state test 1 left behind. This is exactly Design Patterns Chapter 2's own `SingletonCounter` finding, reproduced here at the plain-function level: shared mutable state makes correctness depend on call order.

VERIFIED DIRECTLY — THE PURE VERSION IS IMMUNE, BY CONSTRUCTION

The identical two calls against `sell_item_pure(100, 30)` and `sell_item_pure(100, 50)`, in the identical order: both correctly return `70` and `50`. Neither call could see anything left behind by the other, because neither call reads or writes anything outside its own parameters and return value.

PURITY AS A TESTABILITY PROPERTY, NOT A PURITY CONTEST

A pure function's own result is fully determined by its own arguments — nothing else needs to be set up, reset, or reasoned about between calls. This is the same property Software Architecture Fundamentals Chapter 7 verified paying off at a larger scale (~18,111× faster testing, because a fake adapter could stand in cleanly): purity is what makes standing in for something else — or simply calling a function twice — safe.

A Single Level of Abstraction, Measured

A function mixing "what should happen" (business steps) with "how exactly it happens" (raw arithmetic, string formatting, file I/O) forces a reader to switch mental altitude line by line.

```
# MIXED — business logic and low-level detail tangled together def process_order_mixed(order):
total = 0 for item in order['items']: total += item['price'] * item['qty'] if
order['customer_tier'] == 'gold': total *= 0.8 log_entry = f"Order {order['id']}:
total=${total:.2f}" with open('orders.log', 'a') as f: f.write(log_entry + '\n')
send_confirmation_email(order['customer_email'], total) return total # SINGLE LEVEL — the top
function reads as one consistent list of business steps def process_order_clean(order): total =
calculate_order_total(order) log_order(order, total)
send_confirmation_email(order['customer_email'], total) return total
```

VERIFIED DIRECTLY — THE MIXED VERSION TANGLES 5 LOW-LEVEL OPERATIONS WITH 1 HIGH-LEVEL CALL

Scanning `process_order_mixed`'s own body for raw operators, string formatting, and file I/O versus named business-function calls: **5** low-level operations (arithmetic, an f-string, a file write) sit alongside just **1** high-level call. A reader has to switch between "what does this business rule mean" and "how exactly does string formatting work" within the same six lines.

VERIFIED DIRECTLY — THE SINGLE-LEVEL VERSION HAS ZERO LOW-LEVEL OPERATIONS AT THE TOP

`process_order_clean`'s own body: **3** high-level calls, `0` low-level operations. Every line reads at the identical altitude — "calculate the total, log it, email it" — with all the raw detail pushed down into the three functions that own it individually.

VERIFIED DIRECTLY — BOTH VERSIONS COMPUTE THE IDENTICAL RESULT

For a gold-tier order totaling `$100`: both `process_order_mixed` and `process_order_clean` return `80.0`. Separating the levels of abstraction changed nothing about what the function computes.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Impurity reproducing a real test-order-dependent bug	Design Patterns Chapter 2's own `SingletonCounter` — the identical failure mode, verified twice in two different courses
A single-level function's own testability payoff	Software Architecture Fundamentals Chapter 7's own ~18,111× fake-adapter speedup — purity is the property that makes swapping in a fake safe in the first place
`process_order_mixed`'s own tangled responsibilities	Chapter 4's own "Bloaters" code smells — a function this tangled is a direct instance of the long-method smell that chapter names and catalogs

Hands-On Exercises

EXERCISE 1

Write a third "test" against this chapter's own `sell_item_impure` — selling `10` units from a fresh stock of `50` (expect `40`) — run immediately after this chapter's own two tests, with no reset in between. Verify what it actually returns and explain why, tracing the state left behind by each prior call.

View solution

EXERCISE 2

Add a fourth business step to `process_order_clean` — a call to a new `award_loyalty_points(order, total)` function — following the same pattern as the existing three calls. Verify the top-level function still has 0 low-level operations after the addition.

View solution

EXERCISE 3

Using this chapter's own two verified findings, explain why `calculate_order_total`, `log_order`, and `send_confirmation_email` (the three functions `process_order_clean` delegates to) are each individually easier to write as *pure* functions than `process_order_mixed` ever could be — what property of mixing abstraction levels makes purity harder to achieve?

View solution

CHAPTER 3 QUICK REFERENCE

- **Purity:** a function whose result depends only on its own arguments — verified: an impure function reproduced a real test-pollution bug (`20` instead of `50`); the pure equivalent was immune by construction
- **Single level of abstraction:** every line in a function operating at the same conceptual altitude — verified: a mixed function tangled `5` low-level operations with `1` high-level call; a single-level version had `3` high-level calls and `0` low-level operations, computing the identical result
- **Next chapter:** Code Smells I: Bloaters — naming and catalogging exactly the kind of tangled function this chapter's own `process_order_mixed` already demonstrated

CHAPTER 4 OF 10

Code Smells I: Bloaters

Clean Code, SOLID & Refactoring

Chapter 4 · Code Smells I: Bloaters

"Bloaters" are the code smells that grow slowly, one reasonable-looking addition at a time, until a function, class, or parameter list is carrying more than any one thing should. This chapter names and verifies four of the classic five — the fifth, Long Method, was already measured directly in Chapter 3.

Long Method — Already Measured

Chapter 3's own `process_order_mixed` is a textbook Long Method: **5** low-level operations tangled with **1** high-level call, all inside one function body. That chapter's own single-level-of-abstraction split (`calculate_order_total`, `log_order`, `send_confirmation_email`) is the standard fix — nothing new to verify here; the finding already stands.

Long Parameter List & Data Clumps: a Swap Bug, Visible or Not

```
# BAD — 9 positional parameters, 5 of them a "data clump" (an address) traveling separately def create_user_bad(first_name, last_name, email, phone, street, city, state, zip_code, country): return {'name': f'{first_name} {last_name}', 'address': f'{street}, {city}, {state} {zip_code}, {country}'} # GOOD — the address clump bundled into one object, everything else keyword-only def create_user_good(*, first_name, last_name, email, phone, address): return {'name': f'{first_name} {last_name}', 'address': f"{address['street']}, {address['city']}..."}
```

VERIFIED DIRECTLY — THE IDENTICAL MISTAKE RUNS SILENTLY WRONG IN BOTH VERSIONS

Swapping `city` and `state` at the call site: `create_user_bad('Jane', 'Doe', ..., '123 Main St', 'TX', 'Austin', '78701', 'USA')` runs without error and produces the address `"123 Main St, TX, Austin 78701, USA"` — genuinely wrong, silently. The identical mistake in the bundled version, `address={'city': 'TX', 'state': 'Austin', ...}`, produces the *exact same* wrong address string. Neither version's own type system catches this — Python has no way to know a two-letter string "should" be a state and not a city.

VERIFIED DIRECTLY — THE MISTAKE IS ONLY VISUALLY CATCHABLE IN ONE OF THE TWO VERSIONS

In `create_user_bad`'s own call, `'TX', 'Austin'` gives a reader nothing to go on — two adjacent strings, no labels, no way to tell which was meant for which without counting positions against the function's own signature. In

`address={'city': 'TX', 'state': 'Austin', ...}`, the mistake is *labeled directly at the point of error* — `'city': 'TX'` reads as wrong on sight to anyone who knows TX is a state abbreviation, with zero need to consult the function's own definition.

VERIFIED DIRECTLY — GENUINE TYPOS PRODUCE CATEGORICALLY BETTER ERRORS WITH KEYWORD ARGUMENTS

Calling `create_user_bad` with one positional argument missing raises `TypeError: create_user_bad() missing 1 required positional argument: 'country'` — a real error, but one that gives no information about *which* of the 9 positions the caller actually got wrong if the count happened to still be correct. Calling `create_user_good` with a typo'd keyword (`first_nam` instead of `first_name`) raises `TypeError: create_user_good() got an unexpected keyword argument 'first_nam'. Did you mean 'first_name'?` — Python itself suggests the fix.

Primitive Obsession: When a Raw Number Isn't Enough

```
class Money: def __init__(self, amount, currency): self.amount = amount; self.currency = currency
def __add__(self, other): if self.currency != other.currency: raise ValueError(f'Cannot add
{self.currency} and {other.currency}') return Money(self.amount + other.amount, self.currency)
```

VERIFIED DIRECTLY — RAW FLOATS SILENTLY COMBINE TWO DIFFERENT CURRENCIES INTO A MEANINGLESS NUMBER

`sum([10.0, 10.0])`, where one `10.0` represents `$10 USD` and the other `€10 EUR`: returns `20.0`, without error — a number with no honest meaning, since dollars and euros were never the same unit.

VERIFIED DIRECTLY — A MONEY TYPE CATCHES THE IDENTICAL MISTAKE, CORRECTLY

`Money(10.0, 'USD') + Money(10.0, 'EUR')` correctly raises `ValueError: Cannot add USD and EUR`. Same-currency addition still works correctly — `Money(10.0, 'USD') + Money(15.0, 'USD')` returns `25.0 USD`. The type itself now enforces a rule raw floats structurally cannot express.

Large Class: Counting Concerns, Not Just Lines

VERIFIED DIRECTLY — ONE CLASS MIXING THREE UNRELATED CONCERNS, SPLIT INTO THREE CARRYING EXACTLY ONE EACH

A single `UserManagerBad` class with 6 methods, classified by which concern each name touches (authentication, email, reporting): **3** distinct concerns present in *one* class. Splitting into `Authenticator`, `EmailService`, and `ReportGenerator` — each keeping only the methods matching its own name — leaves each new class with exactly `1` concern: `Authenticator` → `{'auth'}`, `EmailService` → `{'email'}`, `ReportGenerator` → `{'report'}`.

THE FORWARD REFERENCE

> "One class, one reason to change" is this chapter's own informal preview of Chapter 6's Single Responsibility Principle — the concern-counting technique used here is a concrete, mechanical proxy for exactly that principle, before this course names it formally.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A bundled parameter object making a swap mistake visually catchable	Chapter 2's own naming findings — labeling data at the point of use is another instance of a name doing real diagnostic work
A Money type rejecting a mismatched-currency addition	Chapter 6's Single Responsibility Principle — a type that enforces its own invariant is a small, concrete instance of "one reason to change"
A large class's own concerns, mechanically counted and split	Chapter 5's Couplers & Dispensables — a God object is frequently also where Feature Envy and Inappropriate Intimacy show up, covered directly next chapter

Hands-On Exercises

EXERCISE 1

Using this chapter's own `create_user_good`, construct a call where the *bundled* address dict itself is missing the `'zip_code'` key entirely. Verify what actually happens when the function tries to format it, and compare that failure mode to what a missing positional argument does in `create_user_bad`.

View solution

EXERCISE 2

Add a `__sub__` method to this chapter's own `Money` class, with the identical currency check `__add__` uses. Verify subtracting two same-currency amounts works correctly, and verify subtracting two different currencies is correctly rejected.

View solution

EXERCISE 3

Add a seventh method, `log_login_attempt`, to this chapter's own `UserManagerBad`, and run this chapter's own concern classifier against it. Verify whether the classifier recognizes it as belonging to any existing concern, or reveals a fourth, unaccounted-for one — and explain what this means for where the new method should actually live.

View solution

CHAPTER 4 QUICK REFERENCE

- **Long Method:** already measured in Chapter 3 — 5 low-level operations tangled with 1 high-level call
- **Long Parameter List / Data Clumps, verified:** a swap mistake ran silently wrong in both a 9-parameter and a bundled version, but was only visually catchable once labeled by keyword; typo'd keywords produced far better errors than a missing positional argument
- **Primitive Obsession, verified:** raw floats silently combined `$10 USD` and `€10 EUR` into a meaningless `20.0`; a `Money` type correctly rejected the identical mistake
- **Large Class, verified:** one class mixing 3 concerns, split into three classes each carrying exactly 1
- **Next chapter:** Code Smells II: Couplers & Dispensables — Feature Envy, Inappropriate Intimacy, dead code, and duplicated code

CHAPTER 5 OF 10

Code Smells II: Couplers & Dispensables

Clean Code, SOLID & Refactoring

Chapter 5 · Code Smells II: Couplers & Dispensables

Where Chapter 4's Bloaters were about things growing too large, this chapter's five smells are about the wrong things being *connected* — coupling that shouldn't exist, and code that shouldn't still exist. Each one is verified directly, tying back to Software Architecture Fundamentals Chapter 5's own coupling and cohesion criteria.

Feature Envy: a Method More Interested in Someone Else's Data

```
class InvoicePrinter: def print_invoice(self, order): total = 0 for item in order.items: total += item['price'] * item['qty'] return f"Total: ${total:.2f}"
```

VERIFIED DIRECTLY — THE METHOD TOUCHES ZERO OF ITS OWN DATA

Scanning `print_invoice`'s own source: `0` references to `self.`, at least `1` reference to `order.` — `InvoicePrinter` has no state of its own, and this method does nothing but reach into `Order`'s own data to do its job. The fix moves the calculation onto `Order` itself; the corrected `print_invoice` makes exactly one call, `order.calculate_total()`, with `0` direct references to `order.items` — both versions compute the identical `"Total: $35.00"`.

Inappropriate Intimacy: Reaching Into What's Supposed to Be Private

```
class AccountAuditorBad: def audit(self, account): return account._balance > 10000 # reaches directly into a 'private' field class AccountAuditorGood: def audit(self, account): return account.get_balance() > 10000 # uses the public interface only
```

VERIFIED DIRECTLY — A LEGITIMATE INTERNAL RENAME BREAKS ONE VERSION AND NOT THE OTHER

Both audits correctly return `True` for a `$15,000` balance *before* any change. `BankAccount` then undergoes a real, self-contained internal rename — `_balance` becomes `_current_balance`, with `get_balance()` updated to match. Afterward: `AccountAuditorBad.audit()` **breaks** — `AttributeError: 'BankAccountBadRenamed' object has no`

`attribute '_balance'`. `AccountAuditorGood.audit()` still correctly returns `True` — completely unaffected, because it never depended on the private field's own name.

Duplicated Code: Two Copies, One Fix

VERIFIED DIRECTLY — A REAL BUG FIX APPLIED TO ONLY ONE OF TWO IDENTICAL COPIES

`validate_email_v1` and `validate_email_v2` start byte-identical (copy-pasted), and agree — both incorrectly accept `"@b.com"` (no local part before the `@`). A real bug fix is applied to `v1` only. Afterward: `validate_email_v1_fixed("@b.com")` correctly returns `False`; `validate_email_v2("@b.com")` — the un-updated copy — still incorrectly returns `True`. Nothing broke, no error was raised; the two copies simply, silently disagree now.

THIS IS THE SAME BUG SOFTWARE ARCHITECTURE FUNDAMENTALS ALREADY VERIFIED, ONE COURSE APART

That course's own Chapter 8 measured two independently-implemented pricing functions diverging by a real $0.50 for the identical order. This chapter's finding is the same failure mode, caught earlier — while the two copies are still *identical*, before real business logic has had a chance to diverge on its own.

Dead Code: Verified, Not Assumed

VERIFIED DIRECTLY — ONE FUNCTION IS GENUINELY REFERENCED; THE OTHER GENUINELY ISN'T

Searching every other function's own source in a small codebase for a call to `calculate_shipping(`: found, `True`. Searching the identical codebase for a call to `calculate_shipping_legacy(` — a function that exists, has a plausible name, and could easily be mistaken for still in use — finds nothing: `False`. It's not that this function is *hard to find a use for*; a real, mechanical search across the actual codebase confirms there isn't one.

Speculative Generality: A Hook Nobody Ever Pulls

```
def calculate_price(base_price, discount_strategy=None): # 'in case we need custom strategies later' if discount_strategy is None: return base_price return discount_strategy(base_price)
```

VERIFIED DIRECTLY — EVERY REAL CALL SITE IGNORES THE FLEXIBILITY THAT WAS BUILT FOR IT

Checking all **4** real call sites in a small codebase for whether any of them actually pass a custom `discount_strategy`: `0` do. The parameter exists, adds a branch to read and reason about, and has never once been exercised with anything other than its own default.

WHY "MIGHT NEED IT LATER" ISN'T EVIDENCE ON ITS OWN

Design Patterns Chapter 10's own capstone named this exact risk directly: a pattern (or, here, a parameter) applied for a hypothetical future need is over-engineering unless that need is real. This chapter's own finding makes the test concrete — if zero real call sites use the flexibility after real code has been written, the generality was speculative, not forward-thinking.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Feature Envy's own zero-self-reference measurement	Software Architecture Fundamentals Chapter 5's own coupling/cohesion criteria — the identical underlying question, applied to methods instead of services
A private-field rename breaking one audit and not the other	Software Architecture Fundamentals Chapter 7's own dependency inversion finding — depending on a public interface instead of a private implementation detail is the same discipline, one level down
Duplicated code diverging silently after one fix	Software Architecture Fundamentals Chapter 8's own $0.50 thick-client discrepancy — the same risk, caught here before real logic had drifted apart

Hands-On Exercises

EXERCISE 1

Write a second method on `InvoicePrinter`, `print_item_count(order)`, that returns `len(order.items)`. Verify whether this new method also shows Feature Envy by this chapter's own self-vs-order reference count, and explain whether moving it onto `Order` would be as clearly justified as moving `print_invoice`'s own logic was.

View solution

EXERCISE 2

Fix this chapter's own duplicated-code problem properly: delete `validate_email_v2` entirely, and have every caller use `validate_email_v1_fixed` instead. Verify there is now only one implementation to keep correct, and confirm both original callers still get the correctly-fixed behavior.

View solution

EXERCISE 3

Add a fifth call site to this chapter's own `calculate_price` codebase that *does* pass a real `discount_strategy` function. Re-run this chapter's own speculative-generality check across all 5 call sites, and explain whether the

parameter is still speculative generality once it has a genuine use.

View solution

CHAPTER 5 QUICK REFERENCE

- **Feature Envy, verified:** a method with 0 references to its own class's data and at least 1 to another class's — fixed by moving the logic to the class it actually depends on
- **Inappropriate Intimacy, verified:** a private-field rename broke a class reaching directly into it; a class using only the public interface survived unaffected
- **Duplicated Code, verified:** two identical copies silently diverged after only one received a real bug fix — the same failure mode Software Architecture Fundamentals verified at a larger scale
- **Dead Code, verified:** a real codebase-wide search confirmed one function referenced, one genuinely not
- **Speculative Generality, verified:** a flexibility hook never once used with a non-default value across 4 real call sites
- **Next chapter:** SOLID I — Single Responsibility & Open/Closed, formalizing several of these smells' own fixes

CHAPTER 6 OF 10

SOLID I: Single Responsibility & Open/Closed

Clean Code, SOLID & Refactoring

Chapter 6 · SOLID I: Single Responsibility & Open/Closed

SOLID gives names to disciplines this course has already been practicing informally. Chapter 4's Large Class and Chapter 5's Feature Envy were both early instances of the first principle here; Chapter 1's platinum-tier example was an early instance of the second. This chapter verifies both principles directly, at their sharpest.

Single Responsibility: "One Reason to Change," Tested Concretely

```
# BAD — one class, two responsibilities: calculating pay AND formatting a report class
PayrollReportBad: def __init__(self, employees): self.employees = employees def
calculate_total_pay(self): return sum(e['hours'] * e['rate'] for e in self.employees) def
print_report(self): total = self.calculate_total_pay() # the formatter depends directly on the
calculator return f"Total: ${total:.2f}"
```

VERIFIED DIRECTLY — THE REPORT FORMATTER GENUINELY CANNOT BE TESTED WHILE THE CALCULATION IS BROKEN

Simulating `calculate_total_pay` temporarily broken (a real, common mid-refactor state): calling `print_report()` — which is supposed to be testing *formatting*, nothing else — correctly fails with `RuntimeError: pay calculation logic is broken mid-refactor`. There is no way to verify the report's own formatting is correct while this class's *other* responsibility is broken, because the two are bundled into one object with no boundary between them.

```
# GOOD — split by responsibility class PayCalculator: def calculate_total_pay(self): # ... class
PayrollReportFormatter: def format_report(self, total): return f"Total: ${total:.2f}"
```

VERIFIED DIRECTLY — THE FORMATTER NOW TESTS CORRECTLY WITH ZERO DEPENDENCY ON THE CALCULATOR

Calling `PayrollReportFormatter().format_report(9999.99)` — a fake total, with `PayCalculator` never even constructed — correctly returns `"Total: $9999.99"`. And `PayCalculator`'s own source, captured via `inspect.getsource()` before and after using the formatter: byte-identical, `True`. The two responsibilities can now genuinely be verified, changed, and reasoned about independently.

THIS IS THE SAME SHAPE AS SOFTWARE ARCHITECTURE FUNDAMENTALS CHAPTER 7'S OWN FAKE ADAPTERS

A formatter that can be tested against a fake total, with no working calculator required, is the exact same testability property that chapter measured as an ~18,111× speedup at a larger scale. SRP is what makes a component small and self-contained enough to fake in the first place.

Open/Closed: Extended Without Touching a Single Existing Line

```
class DiscountStrategy: def apply(self, price): raise NotImplementedError class
NoDiscount(DiscountStrategy): def apply(self, price): return price class
GoldDiscount(DiscountStrategy): def apply(self, price): return price * 0.8 class PricingEngine:
def __init__(self, strategy): self.strategy = strategy def calculate(self, price): return
self.strategy.apply(price)
```

VERIFIED DIRECTLY — EVERY EXISTING CLASS STAYED BYTE-IDENTICAL AFTER A REAL EXTENSION

Adding a new `PlatinumDiscount(DiscountStrategy)` class — a genuine new discount type — and re-capturing every existing class's own source afterward: `PricingEngine` unchanged (`True`), `DiscountStrategy` unchanged (`True`), `NoDiscount` unchanged (`True`), `GoldDiscount` unchanged (`True`). Nothing that already existed was opened, edited, or even needed to be re-read.

VERIFIED DIRECTLY — THE COMPLETELY UNMODIFIED PRICINGENGINE CORRECTLY RUNS THE NEW DISCOUNT

`PricingEngine(PlatinumDiscount()).calculate(100)` correctly returns `70.0` — `PricingEngine` 's own code never changed, yet it correctly applies a discount type that didn't exist when it was written. This is "closed for modification" verified literally, not just "unlikely to need a change."

THIS IS DESIGN PATTERNS' OWN STRATEGY, ONE COURSE LATER, FORMALLY NAMED

Design Patterns Chapter 7 built this exact shape and verified swapping strategies at runtime. This chapter's own finding — that `PricingEngine` stays genuinely untouched by new discount types — is what Open/Closed *means*: Strategy (and, for a different kind of extension, Decorator, Design Patterns Chapter 5) are concrete design patterns that exist specifically to satisfy this principle. SOLID names the goal; the patterns are working implementations of it.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A bundled class's own formatter untestable while the calculator is broken	Chapter 4's own Large Class finding (3 concerns in 1 class) — the exact same problem, now measured by testability instead of concern-counting
4 existing classes verified byte-identical after a real extension	Chapter 1's own platinum-tier example — the identical principle, verified more completely (whole classes, not one dict line)

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Strategy named directly as an OCP implementation	Design Patterns Chapter 5's Decorator — a second, equally valid way to satisfy Open/Closed, covered next in Chapter 8's refactoring catalog

Hands-On Exercises

EXERCISE 1

Add a second method to `PayrollReportFormatter`, `format_summary_line(total, employee_count)`, following the same pattern as `format_report`. Verify it can be tested with two fake values, with zero dependency on `PayCalculator`, exactly like this chapter's own `format_report`.

View solution

EXERCISE 2

Add a second new discount type, `SilverDiscount` (15% off), to this chapter's own `DiscountStrategy` hierarchy. Verify every one of the now-five existing classes (including `PlatinumDiscount` from this chapter) stays byte-identical, and verify `PricingEngine` correctly applies it.

View solution

EXERCISE 3

Using this chapter's own two verified findings, explain why satisfying Open/Closed for `DiscountStrategy` required Single Responsibility to already be true for `PricingEngine` — what would have gone wrong extending `PricingEngine` with a new discount type if it had been bundled with unrelated responsibilities the way `PayrollReportBad` was?

View solution

CHAPTER 6 QUICK REFERENCE

- **Single Responsibility, verified:** a bundled formatter genuinely couldn't be tested while its class's other responsibility was broken; a split-out formatter tested correctly against a fake value with zero dependency
- **Open/Closed, verified:** all 4 existing classes stayed byte-identical after a real extension, with the completely unmodified `PricingEngine` correctly applying the new discount
- **The connection to Design Patterns:** Strategy and Decorator are concrete, working implementations of Open/Closed — this chapter measured the principle; that course built the mechanism

- **Next chapter:** SOLID II — Liskov Substitution, Interface Segregation & Dependency Inversion

CHAPTER 7 OF 10

SOLID II: Liskov Substitution, Interface Segregation & Dependency Inversion

Clean Code, SOLID & Refactoring

Chapter 7 · SOLID II: Liskov Substitution, Interface Segregation & Dependency Inversion

The remaining three SOLID principles, each verified with a genuine, reproduced failure — a wrong number, a crash, and a hardcoded dependency that blocks extension. The last one connects directly to a course you've already completed.

Liskov Substitution: a Wrong Answer, Not Just an Awkward Design

```
class Rectangle: def set_width(self, width): self.width = width def set_height(self, height): self.height = height def area(self): return self.width * self.height class Square(Rectangle): # "is-a" Rectangle, in the type-hierarchy sense def set_width(self, width): self.width = width; self.height = width def set_height(self, height): self.width = height; self.height = height
```

VERIFIED DIRECTLY — A FUNCTION WRITTEN CORRECTLY FOR RECTANGLE PRODUCES A GENUINELY WRONG ANSWER FOR SQUARE

`resize_and_check(rect)` calls `set_width(5)` then `set_height(10)` and checks the resulting area against `5 × 10 = 50`. For a real `Rectangle(2, 2)`: expected `50`, actual `50` — correct. For a `Square(2, 2)`, substituted in exactly where a `Rectangle` was expected: expected `50`, actual `100` — `set_height(10)` silently forced the width to `10` too, so the area is `10 × 10`, not `5 × 10`. The function did nothing wrong; the substitution itself broke correctness.

THE LESSON ISN'T "DON'T MODEL SQUARES AS RECTANGLES"

It's that inheritance claims a subtype can stand in for its parent *anywhere* the parent is expected — and `Square`'s own override silently changes what "setting the width" means. Liskov Substitution is violated the moment a subtype's own behavior surprises code that only knows about the base type.

Interface Segregation: a Forced Method That Crashes Generic Code

```
class Worker: def work(self): raise NotImplementedError def eat(self): raise NotImplementedError
class RobotWorker(Worker): def work(self): return "Robot working" def eat(self): raise
NotImplementedError("Robots don't eat") # forced by the interface, but nonsensical
```

VERIFIED DIRECTLY — GENERIC CODE TRUSTING THE FAT INTERFACE GENUINELY CRASHES

`lunch_break(workers)` calls `.eat()` on every `Worker` in a list, trusting that every `Worker` honors the full interface. With `[HumanWorker(), RobotWorker()]`: it correctly **crashes** — `NotImplementedError: Robots don't eat` — the instant it reaches the robot.

VERIFIED DIRECTLY — SEGREGATED INTERFACES LET GENERIC CODE STAY CORRECT BY CONSTRUCTION

Splitting `Worker` into `Workable` and `Eatable`, with `RobotWorker` implementing only `Workable`: `lunch_break_fixed(workers)`, filtering with `isinstance(w, Eatable)`, correctly returns `['Human eating lunch']` — the robot is never even asked to eat, because it was never claimed to be able to.

Dependency Inversion: the Same Principle, One Course Later

```
# BAD — depends directly on a concrete class class NotificationServiceBad: def __init__(self):
self.sender = EmailSenderConcrete() # GOOD — depends only on an abstraction class
NotificationSender: def send(self, message): raise NotImplementedError class
NotificationServiceGood: def __init__(self, sender): self.sender = sender def notify(self,
message): return self.sender.send(message)
```

VERIFIED DIRECTLY — THE SOURCE STAYS BYTE-IDENTICAL REGARDLESS OF WHICH CONCRETE SENDER IS INJECTED

`NotificationServiceGood(EmailSender())` correctly returns `"Emailing: hello"`; `NotificationServiceGood(SmsSender())` correctly returns `"Texting: hello"`. `NotificationServiceGood`'s own source, captured before and after using both: byte-identical, `True`. `NotificationServiceBad`, by contrast, can only ever email — adding SMS support would require editing it directly.

THIS IS SOFTWARE ARCHITECTURE FUNDAMENTALS CHAPTER 7, VERIFIED AGAIN

That chapter's own `PricingEngine` depended only on `InventoryPort` / `NotificationPort` — never on `RealInventoryAdapter` or `FakeInventoryAdapter` by name — and measured an ~18,111× testability payoff from it. This chapter's `NotificationServiceGood` is the identical pattern, one level smaller: depend on `NotificationSender`, never on `EmailSender` or `SmsSender` directly. Dependency Inversion is the SOLID principle; Hexagonal Architecture is what it looks like applied to a whole application.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Square's own override silently breaking a caller's assumption	Design Patterns Chapter 8's State pattern — a subtype changing behavior in a way callers don't expect is exactly the risk State's own explicit transitions guard against
`lunch_break` crashing on a method the interface never should have forced	Chapter 4's Long Parameter List — both smells share the same root cause: a contract asking for more than every real user of it can honestly provide
A byte-identical service regardless of injected dependency	Software Architecture Fundamentals Chapter 7's own ~18,111× testability finding — the same principle, verified at two different scales, one course apart

Hands-On Exercises

EXERCISE 1

Write a second test against this chapter's own `resize_and_check`, calling `set_height(10)` *before* `set_width(5)` instead of after, against a fresh `Square(2, 2)`. Verify whether reversing the call order changes the outcome, and explain what this reveals about the nature of the LSP violation.

View solution

EXERCISE 2

Add a third worker type to this chapter's own segregated hierarchy, `VendingMachineWorker`, implementing neither `Workable` nor `Eatable` (it just dispenses snacks). Verify `lunch_break_fixed` correctly excludes it from the list, and verify a similarly-named `shift_schedule(workers)` function filtering by `Workable` also correctly excludes it.

View solution

EXERCISE 3

Add a third sender type to this chapter's own `NotificationSender` hierarchy, `PushNotificationSender`. Verify `NotificationServiceGood`'s own source is still byte-identical after adding it, and explain — using this chapter's own explicit connection to Software Architecture Fundamentals Chapter 7 — which specific verified finding from that chapter this result reproduces.

View solution

CHAPTER 7 QUICK REFERENCE

- **Liskov Substitution, verified:** substituting a `Square` for a `Rectangle` produced a real wrong area (`100` instead of `50`) in code written correctly for the base type
- **Interface Segregation, verified:** a fat interface forced `RobotWorker` to implement `eat()` nonsensically, crashing generic code; segregated interfaces filtered by `isinstance` excluded it correctly instead
- **Dependency Inversion, verified:** a service depending on an abstraction stayed byte-identical regardless of which concrete sender was injected — the same principle Software Architecture Fundamentals Chapter 7 measured at ~18,111×
- **Next chapter:** The Refactoring Catalog: Core Techniques — the concrete moves that turn a violation into compliance

CHAPTER 8 OF 10

The Refactoring Catalog: Core Techniques

Clean Code, SOLID & Refactoring

Chapter 8 · The Refactoring Catalog: Core Techniques

Every prior chapter identified a problem. This chapter names the concrete, mechanical moves that fix them — some of which you've already performed without the name attached. Two of the five get their strongest verification yet here; the other three are callbacks to work already done.

Extract Method: One Fix, Every Caller

```
# BAD — the same calculation duplicated across two functions def print_invoice_bad(order): total = 0 for item in order['items']: total += item['price'] * item['qty'] tax = total * 0.08 return f"Total: ${total:.2f}, Tax: ${tax:.2f}" def print_receipt_bad(order): total = 0 for item in order['items']: total += item['price'] * item['qty'] # duplicated return f"You paid: ${total:.2f}"
```

VERIFIED DIRECTLY — A REAL FIX TO ONE DUPLICATED COPY LEAVES THE OTHER GENUINELY WRONG

A real business change — a 10% loyalty discount — is applied to *only* `print_invoice_bad`. Afterward: `print_invoice_bad` correctly returns `"Total: $90.00, Tax: $7.20"`; `print_receipt_bad` still returns `"You paid: $100.00"` — the un-fixed copy, silently out of sync with the fixed one.

VERIFIED DIRECTLY — EXTRACTING THE CALCULATION MAKES ONE FIX REACH BOTH CALLERS AUTOMATICALLY

Extracting the shared logic into `calculate_order_total(order)`, called from both `print_invoice_good` and `print_receipt_good`: applying the identical discount to the *one* shared function makes both correctly return `$90.00` — `print_invoice_good` and `print_receipt_good` both reflect the fix, with neither one edited directly.

Extract Variable: Naming an Expression Makes It Debuggable

```
# BAD def is_eligible_bad(user): if user['age'] >= 18 and user['country'] == 'US' and user['verified']: return True return False # GOOD def is_eligible_good(user): is_adult = user['age'] >= 18 is_us_resident = user['country'] == 'US' is_verified = user['verified'] return is_adult and is_us_resident and is_verified
```

VERIFIED DIRECTLY — EXTRACTING VARIABLES CHANGES NOTHING ABOUT CORRECTNESS

Across four test users (all-pass, too young, wrong country, unverified): both versions agree on every case, confirmed directly. Extracting the sub-expressions changed nothing about what the function computes.

VERIFIED DIRECTLY — EXTRACTION MAKES THE EXACT CAUSE OF A REJECTION IMMEDIATELY VISIBLE

For a rejected 16-year-old, verified US, verified identity: `is_eligible_bad` reports only `False` — no way to tell which condition failed without re-evaluating the whole expression by hand. The extracted version shows `is_adult = False`, `is_us_resident = True`, `is_verified = True` — immediately, unambiguously identifying *age* as the specific cause.

Move Method & Rename — Already Fully Verified

Move Method/Field: Chapter 5's own Feature Envy fix — moving `calculate_total` off `InvoicePrinter` and onto `Order`, verified reducing direct data access to zero — *is* Move Method, performed before this chapter named it. **Rename**: Chapter 2's entire verified case for descriptive naming — the `distance_fee`/`fee` comparison, the consistent-vocabulary search results — is the case for this technique specifically. Nothing new to verify; the findings already stand.

Replace Conditional with Polymorphism

```
# BAD — a type-switch that must be edited for every new type def
calculate_shipping_bad(shipping_type, weight): if shipping_type == 'standard': return weight * 0.5
elif shipping_type == 'express': return weight * 1.2 + 5 elif shipping_type == 'overnight': return
weight * 2.5 + 15 # GOOD — reusing Design Patterns Chapter 7's own Strategy shape directly class
ShippingStrategy: def calculate(self, weight): raise NotImplementedError class
StandardShipping(ShippingStrategy): def calculate(self, weight): return weight * 0.5
```

VERIFIED DIRECTLY — EXTENDING THE TYPE-SWITCH REQUIRES EDITING THE SAME EXISTING FUNCTION; EXTENDING THE POLYMORPHIC VERSION DOESN'T

Adding a new shipping type, `'international'`, to `calculate_shipping_bad` requires rewriting the whole function, inserting a new `elif` among the existing branches. Adding the equivalent `InternationalShipping(ShippingStrategy)` class: `StandardShipping`, `ExpressShipping`, and `OvernightShipping` — captured via `inspect.getsource()` before and after — all confirmed byte-identical, `True` for all three.

VERIFIED DIRECTLY — BOTH VERSIONS AGREE ON COST FOR EVERY SHIPPING TYPE

Standard: `5.0` vs `5.0`. Express: `17.0` vs `17.0`. Overnight: `40.0` vs `40.0`. International: `55.0` vs `55.0`. All four match — the refactor changed how the code is organized, not what it computes.

THIS IS CHAPTER 6'S OPEN/CLOSED, PERFORMED AS AN EXPLICIT REFACTORING STEP

Chapter 6 measured Open/Closed as a static property of already-designed code. This technique is *how you get there* from a type-switch that violates it — and it's the exact same shape as Design Patterns Chapter 7's own `ShippingStrategy` example, reused directly rather than reinvented.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
One shared function fixing both callers automatically	Chapter 5's own duplicated-code finding — Extract Method is the concrete fix for exactly that smell
Extracted variables making a rejection's own cause immediately visible	Chapter 2's own naming findings — a named variable is documentation that stays attached to the exact value it describes
Replace Conditional with Polymorphism reproducing Design Patterns' own Strategy	Design Patterns Chapter 7 — the pattern was built there; this chapter shows the mechanical steps that arrive at it from tangled conditional code

Hands-On Exercises

EXERCISE 1

Add a third function, `print_summary_email(order)`, that also needs the order total, calling this chapter's own shared `calculate_order_total`. Apply a different real change (a flat $2 handling fee) to the shared function and verify all three callers reflect it correctly.

View solution

EXERCISE 2

Add a fourth extracted condition to this chapter's own `is_eligible_good` — `has_valid_payment_method` — and a matching check in `is_eligible_bad`'s own compound expression. Verify a user failing *only* this new condition is immediately identifiable in the extracted version.

View solution

EXERCISE 3

Add a second new shipping type to this chapter's own polymorphic hierarchy, `EconomyShipping`, alongside the existing `InternationalShipping` from this chapter. Verify all four now-existing classes (including `InternationalShipping`) stay byte-identical, and verify the new type's own cost matches a hand calculation.

📄 View solution

CHAPTER 8 QUICK REFERENCE

- **Extract Method, verified:** a duplicated calculation left one caller un-fixed after a real change; the extracted version fixed both callers from one shared function
- **Extract Variable, verified:** both versions computed identically, but only the extracted one made a rejected user's exact failing condition immediately visible
- **Move Method & Rename:** already fully verified in Chapters 5 and 2 — no new findings needed
- **Replace Conditional with Polymorphism, verified:** 3 existing classes stayed byte-identical after a real extension; all 4 shipping types agreed on cost between old and new implementations
- **Next chapter:** Technical Debt — naming it, measuring it, and paying it down deliberately

CHAPTER 9 OF 10

Technical Debt: Naming It, Measuring It, Paying It Down

Clean Code, SOLID & Refactoring

Chapter 9 · Technical Debt: Naming It, Measuring It, Paying It Down

Every prior chapter treated a code smell as something to fix. This chapter asks a harder question: when is fixing it not worth the effort? "Technical debt" is the standard name for that judgment call — but the term is used loosely enough today that it's worth being precise about what it actually meant, and building a real way to decide which debt to pay down first.

The Metaphor's Real Origin — and How It Drifted

The term comes from Ward Cunningham, describing a specific, deliberate choice: shipping a first working version of code, built on the team's current — incomplete — understanding of the problem, in order to learn from real use faster. The debt was the gap between that first understanding and the fuller one the team would only gain afterward. As long as the team went back and rewrote the code once they understood the problem better, the debt was harmless. Left unpaid, the cost of every future change grew, the same way unpaid financial interest compounds.

In popular use today, "technical debt" has drifted into a catch-all for any code that's hard to work with — including code that was simply written carelessly, with no deliberate tradeoff behind it at all. That drift matters, because Cunningham's own version of the metaphor implies something the loose version doesn't: debt taken on *deliberately*, with a plan to repay it, is a normal, sometimes correct engineering decision. Debt that accumulates *by accident*, with no one aware it's there, is a different problem entirely — nobody chose it, and nobody is tracking whether it's being repaid.

Fowler's Technical Debt Quadrant

Martin Fowler's widely used refinement splits debt along exactly that axis — deliberate vs. inadvertent — crossed with a second axis, reckless vs. prudent:

	RECKLESS	PRUDENT
Deliberate	"We don't have time to design this properly."	"We must ship now and deal with the consequences." (Cunningham's original case)
Inadvertent	"What's a layered architecture?"	"Now we know how we should have done it."

ONLY ONE QUADRANT IS CUNNINGHAM'S ORIGINAL MEANING

Reckless-deliberate debt (skipping design because there's "no time") and reckless-inadvertent debt (not knowing better) are both accidental or negligent in the way Cunningham's metaphor was never meant to excuse. Prudent-deliberate debt — a real tradeoff, made knowingly, with a repayment plan — is the one quadrant this chapter treats as a legitimate engineering decision rather than a problem to eliminate.

Measuring Debt: Severity Isn't the Whole Story

Chapter 8 measured duplication directly — how many copies of the same logic exist. That's a severity measure. But severity alone doesn't tell you which debt is actually expensive, because expense depends on how often that code gets touched. A badly-duplicated function nobody ever changes again costs nothing further; a mildly-duplicated function at the center of every sprint's work costs a little, over and over, forever.

```
# three modules, each with real duplicated logic (severity = copy count) # and a change frequency simulating how often it was actually touched modules = { 'A (shipping calc)': {'copies': 2, 'changes': 10}, 'B (notify format)': {'copies': 6, 'changes': 1}, 'C (tax lookup)': {'copies': 4, 'changes': 5}, } # interest = extra edits required per change, times how often that change happened for name, m in modules.items(): interest = (m['copies'] - 1) * m['changes']
```

VERIFIED DIRECTLY — RANKING BY SEVERITY ALONE GIVES THE WRONG PRIORITY ORDER

By copy count (severity) alone: **B (6 copies)** looks worst, followed by C (4), then A (2). But by **total interest paid** — extra edits per change × how often that change happened — the order reverses almost entirely: **C: 15**, **A: 10**, **B: 5**. Module B, the most severely duplicated code, is verified to cost the *least* in aggregate, because nothing ever touches it. Module C, only moderately duplicated, costs the most, because it's touched often.

THE UGLIER CODE IS NOT AUTOMATICALLY THE HIGHER PRIORITY

A prioritization process that ranks by how bad code looks, rather than by how much it actually costs to keep working with, will spend real refactoring effort on the wrong module. Frequency of change is not optional information — it's half of the actual cost.

When Carrying Debt Is the Right Call

Chapter 1 introduced a test for architectural decisions: how much of the codebase does changing your mind touch? The same test applies to debt, with one addition — how much of the codebase's *future* does this code have left? Interest only accrues while the code keeps getting changed. Code that's deleted young stops accruing interest permanently, regardless of how bad its debt was.

```
# identical severity (3 copies), two different lifespans copies = 3 extra_edits_per_change =
copies - 1 # = 2 # scenario 1: short-lived A/B test flag, changed twice, then deleted
experiment_interest = extra_edits_per_change * 2 # = 4 # scenario 2: identical severity, kept as
permanent core logic, changed 50x core_interest = extra_edits_per_change * 50 # = 100
```

VERIFIED DIRECTLY — IDENTICAL DEBT SEVERITY, 25X DIFFERENT COST, PURELY FROM LIFESPAN

The deleted experiment paid **4** total interest before it stopped existing. The identical severity of debt, left in permanent core logic changed 50 times, paid **100** — **25x more**, for exactly the same quality of code. Refactoring the experiment before deletion would have been pure wasted effort; refactoring the permanent code early would have saved the entire difference.

"IS THIS STAYING?" IS A REAL QUESTION, NOT AN EXCUSE

This isn't a license to skip refactoring anything inconvenient — it's a genuine, verifiable factor. Code confirmed to be short-lived (an experiment, a prototype being validated, a flag scheduled for removal) carries real debt at a real severity, but the interest on it is capped by how little time remains for it to accrue. The same debt in code with no planned end date has no such cap.

A Practical Prioritization Formula

Putting both measurements together gives a concrete ranking rule: `interest = (severity − 1) × frequency`, where severity is however many extra places a change has to be made correctly (Chapter 8's own duplicate-copy count is one direct instance of this), and frequency is how often that code actually gets changed. Pay down the highest-interest debt first — not the worst-looking debt, and not the debt that happens to be freshest in memory.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Severity alone ranks the wrong module as highest priority	Chapter 8's own duplicate-copy severity measure — this chapter adds the missing second dimension, frequency
Interest only accrues while code keeps changing	Chapter 1's own "how much of the codebase does changing your mind touch" test, extended with a lifespan factor

Hands-On Exercises

EXERCISE 1

Add a fourth module, D (discount rules) — 5 copies, changed 8 times — to this chapter's own ranking table. Compute its interest and determine where it lands in the full ranking.

View solution

EXERCISE 2

Using this chapter's own lifespan logic, compute the interest for a severity-5 feature flag deleted after 3 weeks (changed once before deletion) versus the same severity kept permanent and changed 20 times. State the resulting cost ratio.

View solution

EXERCISE 3

Chapter 4's own `UserManagerBad` was found (via that chapter's own Exercise 3) to bundle 4 separate concerns. Treating "concerns bundled" as this chapter's own severity measure, and assuming that class is changed 6 times in the tracked period, compute its interest and rank it against this chapter's own modules A through D.

View solution

CHAPTER 9 QUICK REFERENCE

- **The real metaphor:** a deliberate, tracked tradeoff to ship on an incomplete understanding — not a synonym for careless code
- **Fowler's quadrant:** Deliberate/Inadvertent × Reckless/Prudent — only prudent-deliberate debt is a legitimate engineering call
- **Verified:** ranking by severity alone put the worst-looking module (B, 6 copies) at the top; ranking by interest (severity × frequency) put it last
- **Verified:** identical debt severity cost 25x more when the code stayed permanent (100 interest) than when it was deleted young (4 interest)
- **Prioritization formula:** `interest = (severity - 1) × frequency` — pay down the highest-interest debt first, not the worst-looking debt
- **Next chapter:** Capstone — refactoring a real, messy codebase for quality, applying this course's full catalog

CHAPTER 10 OF 10

Capstone — Refactoring a Real Codebase for Quality

Clean Code, SOLID & Refactoring

Chapter 10 · Capstone: Refactoring a Real Codebase for Quality

One continuous worked project: TangleMart's order-processing system, genuinely tangled, refactored end to end using every technique from Chapters 1 through 9 — in the order the code's own dependencies actually force, not an arbitrary walkthrough order. Design Patterns' own capstone refactored a tangled function by *applying patterns*; this one refactors the same kind of tangle by fixing the underlying *quality problems* patterns are built on top of — naming, purity, parameter design, duplication, SOLID, and the refactoring catalog itself.

BASELINE The Tangled Starting Point

```
inventory = {'widget': 50, 'gadget': 30} # global mutable state def po(oid, cid, its, ship, disc, pri, exp, addr): # 8 positional params, cryptic names t = 0 for i in its: t += i['price'] * i['qty'] inventory[i['name']] -= i['qty'] # impure mutation buried in the calc loop if disc == 'gold': t = t * 0.9 elif disc == 'platinum': t = t * 0.8 if ship == 'standard': t += 5 elif ship == 'express': t += 15 elif ship == 'overnight': t += 30 return t def print_summary_bad(oid, cid, its, ship, disc, pri, exp, addr): t = 0 for i in its: t += i['price'] * i['qty'] # duplicated - and missing the shipping add entirely if disc == 'gold': t = t * 0.9 elif disc == 'platinum': t = t * 0.8 return f"Order {oid} total: ${t:.2f}"
```

VERIFIED DIRECTLY — THE DUPLICATED COPY HAD ALREADY SILENTLY DIVERGED

Before any deliberate refactoring even began, running both functions on the same order revealed `po(...)` returning `$41.00` (subtotal, discounted, plus shipping) while `print_summary_bad(...)` returned `$36.00` — shipping was added to one duplicated copy and never to the other. This is Chapter 8's own duplicated-code finding, discovered in the wild rather than staged.

VERIFIED DIRECTLY — THE IMPURE GLOBAL MUTATION BREAKS ON A RETRIED CALL

Calling the equivalent of `po()` twice on the identical order (e.g. a retried request after a network blip) mutated inventory **twice** — `widget` dropped by 4 instead of 2. This is Chapter 3's exact impurity finding, reproduced.

STEP 1 Naming (Chapter 2)

`po`, `t`, `i`, `oid`, `its`, `disc`, `ship` are all renamed to `process_order`, `total`, `item`, `order_id`, `items`, `discount`, `shipping` before anything else happens — a pure rename, verified producing byte-identical output, exactly Chapter 2's own case for why naming is free to fix and costly to skip.

STEP 2 Extract Method, Purity & Move Method (Chapters 3, 5, 8)

```
def calculate_order_total(items, discount, shipping): # pure - no mutation, safe to call any number of times total = sum(i['price'] * i['qty'] for i in items) if discount == 'gold': total *= 0.9 elif discount == 'platinum': total *= 0.8 if shipping == 'standard': total += 5 elif shipping == 'express': total += 15 elif shipping == 'overnight': total += 30 return total def apply_inventory_changes(items, inventory): # the ONLY place that mutates - isolated on purpose for i in items: inventory[i['name']] -= i['qty'] def checkout(items, discount, shipping, inventory): total = calculate_order_total(items, discount, shipping) apply_inventory_changes(items, inventory) # side effect happens exactly once return total
```

VERIFIED DIRECTLY — THE SHARED FUNCTION FIXED THE DIVERGENCE AND THE ISOLATED SIDE EFFECT FIXED THE RETRY BUG

`checkout()` and a new `print_summary()` both call the identical `calculate_order_total()` and now agree exactly, **$41.00 both** — resolving the baseline's own $41/$36 divergence. Calling `print_summary()` three times in a row left inventory completely untouched; a real checkout followed by a safe query-only retry left inventory decremented exactly **once**.

STEP 3 Long Parameter List (Chapter 4)

VERIFIED DIRECTLY — A SWAPPED ARGUMENT ORDER WAS SILENTLY WRONG WITH POSITIONAL ARGS, LOUDLY WRONG WITH KEYWORD-ONLY ARGS

Calling the original 8-positional-argument signature with `shipping` and `discount` accidentally swapped returned `$40.00` instead of the correct `$41.00` — no error, no warning, a plausible-looking wrong number. Converting to keyword-only arguments (`def calculate_order_total(*, items, shipping, discount)`) made the identical mistake, attempted positionally, raise `TypeError` immediately, before any calculation ran — reproducing Chapter 4's own `create_user_bad` / `create_user_good` finding on this exact codebase's own function.

STEP 4 Replace Conditional with Polymorphism (Chapters 6, 8)

Both if/elif chains — discount and shipping — are replaced with `DiscountStrategy` and `ShippingStrategy` hierarchies, reusing the exact class shapes verified in Chapters 6 and 8.

VERIFIED DIRECTLY — 6 EXISTING STRATEGY CLASSES STAYED BYTE-IDENTICAL AFTER ADDING 2 NEW ONES

Adding `SilverDiscount` and `EconomyShipping` left `NoDiscount`, `GoldDiscount`, `PlatinumDiscount`, `StandardShipping`, `ExpressShipping`, and `OvernightShipping` all confirmed byte-identical via `inspect.getsource()` — **all 6: True**. `SilverDiscount` + `EconomyShipping` correctly computed `$40.00` (`40 × 0.95 + 2`), matching a hand calculation exactly.

STEP 5 SOLID: LSP, ISP & DIP (Chapter 7)

```
class Chargeable: def charge(self, amount): raise NotImplementedError class Refundable: def refund(self, amount): raise NotImplementedError class GiftCardPayment(Chargeable): # only implements what it can actually support def charge(self, amount): return f"Charged ${amount} to gift card" class OrderService: def __init__(self, payment_method: Chargeable): # injected, not hardcoded self.payment = payment_method
```

VERIFIED DIRECTLY — THE ORIGINAL INTERFACE FORCED A BROKEN REFUND; SEGREGATING IT REMOVED THE CRASH ENTIRELY

The original single `Payment` interface forced `GiftCardPayment` to implement `refund()`, which raised `NotImplementedError: Gift cards can't be refunded` the moment code written against the full `Payment` contract tried to use it — an LSP violation caught in the act. Segregating into `Chargeable` / `Refundable` means `GiftCardPayment` simply never claims to be `Refundable` (`isinstance(GiftCardPayment(), Refundable)` is correctly `False`) instead of lying about it and crashing later.

VERIFIED DIRECTLY — ORDERSERVICE'S OWN SOURCE STAYED UNCHANGED ACROSS THREE DIFFERENT INJECTED PAYMENT TYPES

`OrderService`'s source, captured before and after injecting a brand-new `CryptoPayment` class it had never seen, is confirmed **byte-identical** — the same dependency-inversion guarantee Software Architecture Fundamentals Chapter 7 verified for `PricingEngine`, reproduced here for payment methods specifically.

STEP 6 Debt Prioritization Retrospective (Chapter 9)

Applying Chapter 9's own `interest = (severity - 1) × frequency` formula to the three duplication/branching-shaped smells actually found in the baseline code, using each chain's own branch count (including the implicit "no match" path) as severity:

SMELL	SEVERITY	CHANGES	INTEREST
Shipping if/elif chain	4 branches	9×	**27**
Duplicated total calc	2 copies	12×	**12**
Discount if/elif chain	3 branches	4×	**8**

VERIFIED DIRECTLY — THE SINGLE STEP 4 REFACTOR RESOLVED NEARLY 3X MORE INTEREST THAN THE DUPLICATION FIX

The shipping chain alone (27) outranks the duplication fix's own interest (12), even though duplication was fixed first — a structural necessity, since Step 4's strategy classes needed Step 2's own extracted `calculate_order_total` to plug into. Combined, both conditional chains fixed together in Step 4 total **35** interest — **2.92×** the duplication fix's own 12. The highest-value single step wasn't the one performed first; it was the one the earlier steps had to unlock.

Final Integration Check

VERIFIED END TO END — EVERY FIXED COMPONENT WORKING TOGETHER ON ONE REAL ORDER

A full order (2× widget, gold discount, standard shipping) run through the fully assembled system: pre-checkout `print_summary()` reports `$41.00` with zero inventory mutation; `checkout()` via an injected `CreditCardPayment` charges the identical `$41.00` and decrements inventory exactly once; two further `print_summary()` calls leave inventory unchanged; and swapping in a brand-new `StoreCreditPayment` — never seen by `OrderService` before — correctly charges the same order with zero changes to `OrderService`'s own source. Every number matches the original baseline's own correct total, and every bug the baseline exhibited is now structurally impossible, not just fixed by coincidence.

What This Course Doesn't Cover

- Automated refactoring tooling (IDE-assisted extract/rename, static analysis linters) — this course covered the underlying judgment, not any specific tool
- Language-specific idioms beyond Python's own conventions used throughout
- The full Gang-of-Four pattern catalog — that's Design Patterns' own territory, reused here only where a pattern (Strategy) was the direct destination of a refactor
- Formal code review process, team conventions, or style-guide enforcement — reserved for the still-outstanding Software Development Lifecycle course
- Performance profiling or optimization — a clean design and a fast one are different concerns, covered by Technical Support's own diagnostic courses instead

Where This Connects

THIS CAPSTONE'S TECHNIQUE	DIRECT CONNECTION
Extract Method resolving a duplicated, diverged calculation	Chapter 8, and Design Patterns' own Strategy chapter — the same shape of fix, one course apart

THIS CAPSTONE'S TECHNIQUE	DIRECT CONNECTION
Injected payment dependency, OrderService source unchanged across 3 types	Software Architecture Fundamentals Chapter 7's own ports-and-adapters finding, reproduced at class scale
Debt-interest ranking justifying which refactor mattered most	Chapter 9's own formula, applied retrospectively to a real, not hypothetical, codebase
Pseudocode & Algorithmic Problem-Solving's own decomposition discipline	The six-step refactor sequence itself — breaking one tangled function into independently verifiable pieces

Hands-On Exercises

EXERCISE 1

Add a fourth smell to this chapter's own Step 6 ranking table: the original 8-positional-parameter signature from Step 3, treating "adjacent same-typed parameters that could be silently swapped" (2: `shipping` / `discount`) as severity, changed 6 times over the same period. Compute its interest and determine where it lands in the full four-item ranking.

View solution

EXERCISE 2

Add a `PlatinumDiscount`-equivalent tier to this chapter's own final integrated system — a `LoyaltyDiscount` class applying a 15% discount — and verify all pre-existing discount classes (`NoDiscount`, `GoldDiscount`) stay byte-identical, and that a full checkout using the new tier produces the correct total.

View solution

EXERCISE 3

Add a second concurrent order to this chapter's own final integration check — a different item set, different discount and shipping strategies, checked out through the *same* `OrderService` instance used for the first order. Verify both orders' totals are correct and independent, and that inventory reflects both orders' own deductions correctly.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE SUMMARY

- **Verified end to end:** a genuinely tangled order processor, refactored in 6 dependency-ordered steps, each directly reusing a specific prior chapter's own verified technique
- **Step 2 fixed a real divergence:** $41.00 vs $36.00 in the baseline, both agreeing at $41.00 after Extract Method

- **Step 3 turned a silent wrong answer into a loud error:** $40.00 silently vs. an immediate TypeError
- **Step 4 verified 6 strategy classes byte-identical** after 2 real extensions
- **Step 5 verified an LSP violation caught in the act**, then fixed via ISP segregation and DIP injection, OrderService staying byte-identical across 3 payment types
- **Step 6 verified the highest-interest fix (27) wasn't the one performed first** — it was the one the earlier steps had to unlock
- **Course complete:** naming, purity, function design, code smells, SOLID, the refactoring catalog, and technical debt — all ten chapters, verified throughout