Design Patterns

A Complete 10-Chapter Software Development Course

Topics covered:

Creational: Singleton · Factory Method · Abstract Factory · Builder

Structural: Adapter · Facade · Decorator · Composite · Proxy · Flyweight

Behavioral: Strategy · Template Method · Observer · State · Command · Iterator

Capstone: refactoring a genuinely tangled order-processing codebase using four patterns together

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 Design Patterns Matter for Programmers

Design Patterns

Chapter 1 · Why Design Patterns Matter for Programmers

A **design pattern** is a named, reusable *solution shape* for a recurring design problem — not a snippet of code to copy and paste. Pseudocode & Algorithmic Problem-Solving taught how to decompose a problem and choose an algorithmic strategy for one piece of logic; this course teaches a vocabulary for the recurring *shapes* those pieces tend to take once a codebase grows beyond a single function.

A Little History

In 1994, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides — collectively nicknamed the "Gang of Four" — published *Design Patterns: Elements of Reusable Object-Oriented Software*, cataloguing **23 patterns** across three categories. Decades later, their names are still the common vocabulary programmers use to describe a design at a glance — "just make it a Factory" communicates far more, far faster, than describing the same structure from scratch every time.

Demonstration 1: The Same Behavior, Two Genuinely Different Shapes

A discount calculator, written the ordinary way — a direct `if/elif` chain:

```
def calculate_price(price, discount_type): if discount_type == 'percentage': return price * 0.9
elif discount_type == 'flat': return price - 5 else: return price
```

The exact same behavior, expressed as interchangeable objects instead:

```
class PercentageDiscount: def apply(self, price): return price * 0.9 class FlatDiscount: def
apply(self, price): return price - 5 class NoDiscount: def apply(self, price): return price def
calculate_price_v2(price, strategy): return strategy.apply(price)
```

VERIFIED DIRECTLY — IDENTICAL OUTPUT FROM TWO STRUCTURALLY DIFFERENT IMPLEMENTATIONS

Run on a price of `100`: `calculate_price(100,'percentage')=90.0` matches `calculate_price_v2(100, PercentageDiscount())=90.0`. `calculate_price(100,'flat')=95` matches `calculate_price_v2(100,`

`FlatDiscount())=95`. The no-discount case matches too: `100` both ways. Same inputs, same outputs, genuinely different code shapes — this is exactly what "a pattern is a shape, not a snippet" means concretely.

Demonstration 2: Why the Shape Itself Matters, Not Just Style

Suppose a new discount type is needed — a loyalty discount. In the `if/elif` version, `calculate_price` itself has to be edited, adding a new branch. In the object-based version:

```
class LoyaltyDiscount: def apply(self, price): return price * 0.85 calculate_price_v2(100, LoyaltyDiscount())
```

VERIFIED DIRECTLY — A NEW CASE, ZERO CHANGES TO EXISTING CODE

Calling `calculate_price_v2(100, LoyaltyDiscount())` returns `85.0` correctly — and `calculate_price_v2` itself was never touched to make this work. Only a new, self-contained class was added. This is the actual payoff a pattern buys: not a stylistic preference, but a genuine, verifiable difference in how much existing, already-tested code has to change when a new case shows up.

(This particular shape — swapping in an interchangeable object to change behavior at runtime — is a real pattern, **Strategy**, covered fully in Chapter 7. This chapter deliberately doesn't name or formalize it yet — the point here is just that the shape itself is a real, reusable idea, worth naming later once its full structure has been built up properly.)

The Three GoF Categories

CATEGORY	WHAT IT ADDRESSES
Creational	How objects get created — controlling, hiding, or simplifying construction (Chapters 2-3)
Structural	How objects and classes are composed into larger structures (Chapters 4-6)
Behavioral	How objects communicate and distribute responsibility for a behavior (Chapters 7-9)

What This Course Won't Cover

The GoF catalogue has **23** patterns. This course covers **12** — the ones a working programmer runs into most often — honestly, rather than claiming exhaustive coverage:

COVERED (CHAPTERS 2-9)	DELIBERATELY OUT OF SCOPE
Singleton, Factory Method, Abstract Factory, Builder, Adapter, Facade, Decorator, Composite, Proxy, Flyweight, Strategy, Template Method, Observer, State, Command, Iterator	Chain of Responsibility, Mediator, Memento, Visitor, Interpreter, Bridge, Prototype, and the remaining classic GoF patterns

The Honest Warning, Up Front

A PATTERN SOLVES A REAL RECURRING PROBLEM — OR IT'S JUST EXTRA INDIRECTION

Demonstration 2 verified a genuine, measurable benefit: adding `LoyaltyDiscount` touched zero existing code. That benefit exists specifically because new discount types are a recurring, expected kind of change for this problem. Applying the same object-based shape to code that will realistically never need a second variation doesn't buy that benefit — it just adds a class, an interface, and a layer of indirection for nothing. This course revisits this warning fully in the capstone, with a real, worked example of a pattern applied where it genuinely doesn't belong.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
"A pattern is a shape, not a snippet," verified directly	Every later chapter shows the same pattern implemented slightly differently across examples — the shape, not the exact code, is what's being taught
The verified zero-change extensibility benefit	A concrete preview of Chapter 7's own Strategy pattern, and of the Open/Closed Principle Clean Code, SOLID & Refactoring (`cleancode1`) covers formally
The honest over-engineering warning	Directly revisited in Chapter 10's capstone with a real worked example

Hands-On Exercises

EXERCISE 1

Using this chapter's own two `calculate_price` versions as a template, add a third discount type to *both* versions — a "buy one get one half off" style discount that multiplies the price by `0.75` — and verify both versions agree on the result for a price of `100`.

View solution

EXERCISE 2

Using this chapter's own verified findings, explain in your own words why "a design pattern is a reusable solution shape, not literal reusable code" is a more accurate description than "a design pattern is a piece of code you can copy into different projects."

View solution

EXERCISE 3

A developer says "the object-based version is strictly better, so I should always write code this way, even for a boolean flag that will only ever have two possible values and will never change." Using this chapter's own honest warning, explain what's wrong with treating this as a universal rule.

View solution

CHAPTER 1 QUICK REFERENCE

- A **design pattern** is a reusable solution *shape* for a recurring design problem, not literal code to copy-paste
- The 1994 Gang-of-Four book catalogued 23 patterns across three categories: **Creational**, **Structural**, **Behavioral**
- Verified directly: an `if/elif` discount calculator and an object-based version produced identical results (`90.0`, `95`, `100`) — two genuinely different shapes, same behavior
- Verified directly: adding a new discount type touched zero existing code in the object-based version — the real, measurable payoff a pattern can buy
- This course covers 12 of the 23 classic patterns, named explicitly, rather than claiming full coverage
- A pattern only pays off when its target problem is genuinely recurring — applying one where it isn't just adds indirection, a warning revisited fully in Chapter 10
- **Next chapter:** Creational Patterns I — Singleton and Factory Method

CHAPTER 2 OF 10

Creational Patterns I: Singleton & Factory Method

Design Patterns

Chapter 2 · Creational Patterns I: Singleton & Factory Method

Creational patterns control how objects get created. This chapter covers two of the most common — and Singleton, one of the most commonly *misused* — with real pitfalls reproduced directly, not just described.

Singleton: Guaranteeing Exactly One Instance

Ordinarily, calling a class's constructor twice gives two separate objects:

```
class Logger: def __init__(self): self.messages = [] a = Logger() b = Logger()
```

VERIFIED DIRECTLY

`a is b` evaluates to `False` — two genuinely separate objects, with different memory addresses.

Singleton overrides object creation itself so every call returns the same object:

```
class SingletonLogger: _instance = None def __new__(cls): if cls._instance is None: cls._instance
= super().__new__(cls) cls._instance.messages = [] return cls._instance
```

VERIFIED DIRECTLY

`x = SingletonLogger()` and `y = SingletonLogger()`: `x is y` evaluates to `True` — identical memory address both times.

Pitfall 1: Singleton State Silently Pollutes Later Code

A shared global instance means shared global *state* — and that state doesn't reset itself between unrelated uses.

VERIFIED DIRECTLY — A GENUINE, REPRODUCED TEST-POLLUTION BUG

A `SingletonCounter` with an `increment()` method: a first routine increments it three times and correctly finds `count == 3`. A second, entirely independent routine — written expecting a fresh counter — calls `SingletonCounter()`, increments once, and expects `count == 1`. It actually gets `count == 4`: the leftover state from the first routine never went away, because both routines received the exact same object.

WHY THIS IS A GENUINELY COMMON REAL BUG, NOT A CONTRIVED ONE

This is precisely the mechanism behind flaky, order-dependent test suites — a test that passes in isolation fails only when run after another specific test, because a Singleton somewhere in the codebase silently carried state between them. The bug isn't in either routine's own logic; it's in the shared object neither routine realized it didn't have exclusive ownership of.

Pitfall 2: A Naive Singleton Isn't Thread-Safe

The lazy-initialization check, `if cls._instance is None:`, has a real gap: two threads can both evaluate that check as `True` before either one finishes creating the instance.

VERIFIED DIRECTLY — A REAL, REPRODUCED RACE CONDITION

Launching `20` threads simultaneously, each calling a naive lazy Singleton (with a small artificial delay inserted between the check and the creation, to widen the race window): instead of `1` shared instance, `8` **genuinely distinct instances** were created — confirmed by comparing each thread's own object identity. The "only one instance" guarantee failed completely under real concurrent access.

VERIFIED DIRECTLY — THE STANDARD FIX, CONFIRMED WORKING

Adding a lock with **double-checked locking** — check, acquire a lock, check again, then create — and re-running the identical 20-thread test: exactly `1` distinct instance, every time.

A Python-Specific Note: You Often Already Have a Singleton

VERIFIED DIRECTLY — PYTHON'S OWN MODULE SYSTEM IS A BUILT-IN SINGLETON

`import os` followed by `import os as os2`: `os is os2` evaluates to `True`. Python caches every imported module in `sys.modules`, so importing the same module anywhere, any number of times, always returns the identical object — a real, verified, zero-code Singleton, already built into the language. A module-level instance (a plain object created once at the top of a module) is often the more idiomatic Python choice over a hand-rolled `__new__`-based class, for exactly this reason.

Factory Method: Delegating "Which Class?" to a Subclass

Factory Method defines a creation method in a base class without deciding what it creates — each subclass overrides that one method to supply a different concrete type, while every other method in the base class works entirely through the shared interface.

```
class NotificationCreator(ABC): @abstractmethod def create_notification(self): pass # the factory method itself def notify(self, message): notification = self.create_notification() return notification.send(message) class EmailNotificationCreator(NotificationCreator): def create_notification(self): return EmailNotification() class SMSNotificationCreator(NotificationCreator): def create_notification(self): return SMSNotification()
```

VERIFIED DIRECTLY

`EmailNotificationCreator().notify('Your order shipped')` returns `'Email sent: Your order shipped'`; `SMSNotificationCreator().notify(...)` returns `'SMS sent: Your order shipped'` — the identical `notify()` method, defined once in the base class, correctly produces different behavior depending purely on which subclass's own `create_notification()` ran.

VERIFIED DIRECTLY — THE SAME EXTENSIBILITY PAYOFF AS CHAPTER 1

Adding an entirely new `PushNotificationCreator` — a new subclass with its own `create_notification()` returning a new `PushNotification` class — works correctly (`'Push sent: Your order shipped'`) with **zero changes** to `NotificationCreator.notify()` itself, exactly the payoff Chapter 1 first demonstrated.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Singleton's verified test-pollution and race-condition failures	A concrete case study for Clean Code, SOLID & Refactoring's own treatment of hidden global state and testability
Factory Method's zero-change extensibility, verified	The same shape reappears at a larger scale in Chapter 3's Abstract Factory
Double-checked locking as a real, verified fix	A concrete instance of the "shortcuts need a checkable justification" discipline `pseudocode1`'s own Chapter 7 established for greedy algorithms

Hands-On Exercises

EXERCISE 1

Using this chapter's own `SingletonCounter` example, explain what value a third, independent routine would see if it called `SingletonCounter()` and read `count` immediately, without calling `increment()` at all, after both routines

from this chapter's own example have already run.

View solution

EXERCISE 2

Using this chapter's own verified thread-safety findings, explain in your own words why the naive Singleton's race condition specifically requires MULTIPLE threads calling it for the very first time (before any instance exists yet) — and why calling an already-fully-initialized Singleton from multiple threads afterward would not have the same problem.

View solution

EXERCISE 3

Using this chapter's own Factory Method example, write a new `SlackNotification` class and a `SlackNotificationCreator` subclass, following the same pattern as `EmailNotification` / `EmailNotificationCreator`. Verify it works correctly through the existing, unmodified `notify()` method.

View solution

CHAPTER 2 QUICK REFERENCE

- **Singleton:** a class that guarantees exactly one instance — verified via a `__new__` override, `x is y` confirmed `True`
- Verified pitfall 1: shared Singleton state silently pollutes later, unrelated code — a second routine's "fresh" counter unexpectedly started at `3` instead of `0`
- Verified pitfall 2: a naive lazy Singleton is not thread-safe — 20 concurrent threads produced `8` distinct instances instead of `1`; double-checked locking fixed it to exactly `1`
- Verified: Python's own module system (`sys.modules`) is already a built-in Singleton — often the more idiomatic choice over a hand-rolled class
- **Factory Method:** a base class delegates "which concrete type?" to an overridden method in each subclass — verified correct behavior and zero-change extensibility, mirroring Chapter 1's own findings
- **Next chapter:** Creational Patterns II — Abstract Factory and Builder

CHAPTER 3 OF 10

Creational Patterns II: Abstract Factory & Builder

Design Patterns

Chapter 3 · Creational Patterns II: Abstract Factory & Builder

Chapter 2's Factory Method delegated creating *one* object to a subclass. This chapter covers two bigger jobs: keeping a whole *family* of related objects consistent (**Abstract Factory**), and constructing one genuinely *complex* object safely, step by step (**Builder**) — both demonstrated by reproducing the real bug each one exists to prevent.

Abstract Factory: Keeping a Family of Objects Consistent

A UI toolkit needs a button and a checkbox that always match — a light-theme button should never end up next to a dark-theme checkbox. **Abstract Factory** provides one factory per family, so requesting any piece of a family automatically gets the matching version of every other piece.

```
class UIFactory(ABC): @abstractmethod def create_button(self): pass @abstractmethod def
create_checkbox(self): pass class LightThemeFactory(UIFactory): def create_button(self): return
LightButton() def create_checkbox(self): return LightCheckbox() class DarkThemeFactory(UIFactory):
def create_button(self): return DarkButton() def create_checkbox(self): return DarkCheckbox()
```

VERIFIED DIRECTLY — EACH FACTORY PRODUCES A GENUINELY CONSISTENT FAMILY

`build_ui(LightThemeFactory())` returns a `LightButton` and a `LightCheckbox` — confirmed by type, not just by name. `build_ui(DarkThemeFactory())` returns a `DarkButton` and a `DarkCheckbox`. Every single component produced through a given factory belongs to the same family, every time.

THE REAL BUG THIS ACTUALLY PREVENTS — REPRODUCED DIRECTLY

Bypassing the factory and constructing components individually, exactly the way an unwary developer might: `LightButton()` paired with `DarkCheckbox()`. This runs without any error and compiles/executes perfectly — `'Light button (white bg, black text)'` next to `'Dark checkbox (black bg, white check)'` — a genuinely mismatched, jarring UI, produced by code that looks completely reasonable at a glance. Abstract Factory's real value is structural: going through `build_ui(factory)` makes this specific mistake impossible to write, not just discouraged by convention.

Builder: Constructing a Complex Object Safely

A class with many optional parameters — a computer configuration, say — tempts a "telescoping constructor": one big constructor accepting everything positionally.

```
class Computer: def __init__(self, cpu, ram, storage, gpu=None, has_wifi=True, has_bluetooth=True, case_color='black'): # ... store each parameter ...
```

THE REAL BUG THIS ACTUALLY CAUSES — REPRODUCED DIRECTLY

Intending `cpu='i7', ram='16GB', storage='512GB SSD', case_color='white'`, a caller writes `Computer('i7', '16GB', '512GB SSD', 'white')` — a completely natural-looking mistake, since `'white'` lands in the fourth positional slot, which is actually `gpu`, not `case_color`. The result: `gpu='white'` (nonsensical — a GPU with no such model) and `case_color` silently stays at its default, `'black'` — exactly the opposite of what was intended, with **no error raised anywhere**.

Builder replaces positional parameters with named, chained method calls — each step says exactly what it's setting:

```
class ComputerBuilder: def set_cpu(self, cpu): self._cpu = cpu; return self def set_ram(self, ram): self._ram = ram; return self def set_storage(self, storage): self._storage = storage; return self def set_gpu(self, gpu): self._gpu = gpu; return self def set_case_color(self, color): self._case_color = color; return self def build(self): return Computer(self._cpu, self._ram, self._storage, self._gpu, case_color=self._case_color)
```

VERIFIED DIRECTLY — THE IDENTICAL INTENT, CORRECTLY BUILT, WITH NO POSSIBLE MIX-UP

`ComputerBuilder().set_cpu('i7').set_ram('16GB').set_storage('512GB SSD').set_case_color('white').build()` produces exactly the intended object: `gpu=None` (correctly left unset — the step was never called) and `case_color='white'` (correctly set) — the exact opposite of the telescoping constructor's own mistaken result, for the identical underlying intent. There is no positional slot to get wrong, because every step names the field it sets.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Abstract Factory structurally preventing a mixed-family bug	The same "make the wrong thing impossible to write, not just discouraged" idea reappears in Chapter 5's Composite
A real, reproduced telescoping-constructor bug, fixed by Builder	A concrete case study for Clean Code, SOLID & Refactoring's own material on function/constructor parameter design

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Both patterns building on Chapter 2's Factory Method	Chapter 4's structural patterns shift focus from *creating* objects to *composing* already-created ones

Hands-On Exercises

EXERCISE 1

Using this chapter's own Abstract Factory pattern, add a third family member — `create_slider()` — to `UIFactory`, `LightThemeFactory`, and `DarkThemeFactory`, with matching `LightSlider` / `DarkSlider` classes. Verify `build_ui()`-style code produces a consistently-themed slider alongside the button and checkbox.

View solution

EXERCISE 2

Using this chapter's own telescoping-constructor bug, identify exactly which named parameter each of the four positional arguments in `Computer('i7', '16GB', '512GB SSD', 'white')` actually binds to, and explain why the resulting object doesn't raise any error despite being wrong.

View solution

EXERCISE 3

Using this chapter's own two patterns, explain what specific kind of bug each one prevents — Abstract Factory vs. Builder — and why a class with only a single required parameter and no optional ones would need neither pattern.

View solution

CHAPTER 3 QUICK REFERENCE

- **Abstract Factory:** one factory produces a whole family of related objects — verified consistent by type (`LightButton` + `LightCheckbox`, `DarkButton` + `DarkCheckbox`)
- Verified directly: bypassing the factory makes mixing families (`LightButton` + `DarkCheckbox`) trivially possible with no error — exactly what Abstract Factory structurally prevents
- **Builder:** named, chained construction steps replace a positional "telescoping constructor"
- Verified directly: a realistic positional-argument mistake silently set `gpu='white'` instead of `case_color='white'`, with no error raised — the Builder version produced the correct result with no possible mix-up

- **Next chapter:** Structural Patterns I — Adapter and Facade, shifting from creating objects to composing already-created ones

CHAPTER 4 OF 10

Structural Patterns I: Adapter & Facade

Design Patterns

Chapter 4 · Structural Patterns I: Adapter & Facade

Structural patterns compose already-created objects into larger structures — no more object creation, just how existing pieces fit together. This chapter covers two of the most common: making an incompatible interface work without changing it (**Adapter**), and simplifying access to a complex, many-part subsystem (**Facade**).

Adapter: Making an Incompatible Interface Work

Client code expects a simple `pay(amount)` interface. A legacy payment gateway exposes something genuinely different — different method name, different units, an extra required parameter:

```
def checkout(payment_processor, amount_dollars): return payment_processor.pay(amount_dollars)
class LegacyPaymentGateway: def make_payment(self, amount_cents, currency): return f'Legacy
gateway charged {amount_cents} cents ({currency})'
```

VERIFIED DIRECTLY — THE INCOMPATIBILITY IS A REAL, REPRODUCED FAILURE

Calling `checkout(LegacyPaymentGateway(), 49.99)` directly raises a genuine `AttributeError`: `'LegacyPaymentGateway' object has no attribute 'pay'`. The two pieces of code are individually correct — they simply don't speak the same interface.

An **Adapter** wraps the incompatible object and exposes exactly the interface the client expects, translating each call underneath:

```
class PaymentGatewayAdapter: def __init__(self, legacy_gateway): self._legacy_gateway =
legacy_gateway def pay(self, amount_dollars): amount_cents = round(amount_dollars * 100) return
self._legacy_gateway.make_payment(amount_cents, currency='USD')
```

VERIFIED DIRECTLY — THE EXACT SAME CLIENT CODE, NOW WORKING, WITH A VERIFIED CORRECT CONVERSION

`checkout(PaymentGatewayAdapter(LegacyPaymentGateway()), 49.99)` — the identical `checkout()` function, completely unmodified — now succeeds, returning `'Legacy gateway charged 4999 cents (USD)'`. `49.99` dollars correctly became `4999` cents — the conversion the adapter exists to handle, confirmed correct rather than assumed.

Facade: Simplifying a Complex Subsystem

Watching a movie on a real home theater setup means correctly sequencing three separate components — an amplifier, a projector, a DVD player — each with their own multi-step interface:

```
# Manual sequence -- the client has to know every step, and the right order amp.on()
amp.set_volume(5) proj.on() proj.set_input('DVD') dvd.on() dvd.play('Inception')
```

A **Facade** hides that entire sequence behind one method:

```
class HomeTheaterFacade: def __init__(self, amp, dvd, proj): self.amp = amp; self.dvd = dvd;
self.proj = proj def watch_movie(self, movie): self.amp.on() self.amp.set_volume(5) self.proj.on()
self.proj.set_input('DVD') self.dvd.on() self.dvd.play(movie)
```

VERIFIED DIRECTLY — THE FACADE PRODUCES THE IDENTICAL UNDERLYING CALL SEQUENCE

Logging every underlying method call made by each approach: the manual sequence produces exactly `6` calls, in a specific order. `HomeTheaterFacade(...).watch_movie('Inception')` produces the **identical** `6`-call sequence, in the identical order — confirmed by direct comparison of the two call logs, not just by inspection. The Facade doesn't change what happens underneath at all; it changes how much the client has to know and write to make it happen — one call instead of six.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
A verified real `AttributeError`, fixed by wrapping rather than rewriting	Chapter 6's Proxy reuses the identical "wrap an object, expose a compatible interface" shape for a genuinely different purpose — controlling access, not fixing incompatibility
Facade's verified identical call sequence, just fewer client-facing calls	Software Architecture & System Design's own treatment of layered architecture, where a whole layer often plays exactly this simplifying role
Both patterns leaving the wrapped/underlying objects completely unmodified	Chapter 5's Decorator, which also wraps an object without modifying it — but to add behavior, not to translate or simplify it

Hands-On Exercises

EXERCISE 1

Using this chapter's own `PaymentGatewayAdapter` as a template, write an adapter for a second legacy gateway whose method is `process(total_pence)` (British pence, not cents) instead of `make_payment`. Verify it correctly converts `19.99` dollars into the right number of pence and successfully passes through the same unmodified `checkout()` function.

View solution

EXERCISE 2

Using this chapter's own `HomeTheaterFacade`, add a `end_movie()` method that turns everything off in the reverse order it was turned on (DVD player first, then projector, then amplifier). Verify the resulting call log matches the expected reversed order.

View solution

EXERCISE 3

Using this chapter's own two verified demonstrations, explain the specific difference between what Adapter fixes and what Facade fixes — that is, why the payment gateway problem couldn't have been solved with a Facade, and why the home theater problem couldn't have been solved with an Adapter.

View solution

CHAPTER 4 QUICK REFERENCE

- **Adapter:** wraps an incompatible object and exposes the interface a client expects, translating each call — verified fixing a real `AttributeError` and a correct dollars-to-cents conversion (`49.99 → 4999`)
- **Facade:** hides a complex, multi-step subsystem behind one simple method — verified producing the *identical* 6-call sequence as manual orchestration, just collapsed into one client-facing call
- Neither pattern modifies the objects it wraps — both add a layer in front, not a change underneath
- **Next chapter:** Structural Patterns II — Decorator and Composite

CHAPTER 5 OF 10

Structural Patterns II: Decorator & Composite

Design Patterns

Chapter 5 · Structural Patterns II: Decorator & Composite

Two more ways of composing objects: adding behavior to an object at runtime without touching its class (**Decorator**), and letting a caller treat one object and a whole tree of objects through the exact same interface (**Composite**).

Decorator: Adding Behavior Without Modifying the Original

A coffee order needs an arbitrary combination of condiments, each adding its own cost. Decorator wraps the base object in layers, each layer adding one thing:

```
class SimpleCoffee: def cost(self): return 2.00 def description(self): return 'Coffee' class CondimentDecorator: def __init__(self, beverage): self._beverage = beverage class MilkDecorator(CondimentDecorator): def cost(self): return self._beverage.cost() + 0.50 def description(self): return self._beverage.description() + ', Milk' class SugarDecorator(CondimentDecorator): def cost(self): return self._beverage.cost() + 0.30 def description(self): return self._beverage.description() + ', Sugar'
```

VERIFIED DIRECTLY — COST AND DESCRIPTION ACCUMULATE CORRECTLY THROUGH THE WRAPPING CHAIN

`SimpleCoffee()` : `'Coffee'` , `$2.00` . `MilkDecorator(coffee)` : `'Coffee, Milk'` , `$2.50` . `SugarDecorator(MilkDecorator(coffee))` : `'Coffee, Milk, Sugar'` , `$2.80` — each layer correctly builds on the layer beneath it.

VERIFIED DIRECTLY — THE BASE OBJECT STAYS UNMODIFIED, AND SEPARATE CHAINS STAY INDEPENDENT

After building the `Milk+Sugar` chain above, the original `base` object still reports `'Coffee'` and `$2.00` — untouched. Wrapping that *same* `base` object in a completely separate chain, `WhipDecorator(base)` , gives `'Coffee, Whip'` , `$2.75` — genuinely different from the Milk+Sugar chain's `$2.80` , confirming the two decorator chains, built from the identical shared base, don't interfere with each other at all.

Composite: Treating One Object and a Whole Tree Identically

A file has a size. A directory's "size" is the sum of everything inside it — which might itself include other directories. Composite gives both the exact same interface:

```
class File: def __init__(self, name, size): self.name = name; self.size = size def get_size(self): return self.size class Directory: def __init__(self, name): self.name = name; self.children = [] def add(self, child): self.children.append(child) def get_size(self): return sum(child.get_size() for child in self.children)
```

VERIFIED DIRECTLY — A CORRECT TOTAL ACROSS A GENUINELY NESTED TREE

A tree three levels deep — `project/` containing `src/` (two files, `120` + `80`), `docs/` (one file, `45`, plus a further-nested `nested/` directory containing one more file, `30`), and a top-level `README.md` (`10`): `root.get_size()` returns `285`, matching a manual calculation (`120+80+45+30+10`) exactly.

VERIFIED DIRECTLY — THE IDENTICAL METHOD CALL, NO TYPE-CHECKING NEEDED

Calling `.get_size()` on a single `File` (`readme.txt`, returns `5`), on the entire `root` Directory (returns `285`), and on the `src` Directory alone (returns `200`) — the exact same one-line call, with no `isinstance` check or special-casing anywhere in the calling code, correctly handles a leaf, a shallow branch, and a deeply nested tree alike.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Decorator wrapping without modifying, verified two independent ways	Chapter 6's Proxy reuses the identical wrapping shape again, this time to control access rather than add behavior
Composite's verified uniform interface across a nested tree	A concrete, working example of the same "recursion over a self-similar structure" idea `pseudocode1`'s own Chapter 8 (divide and conquer) and Chapter 9 (recursion) already covered
Both patterns' zero-modification-to-existing-code discipline	Clean Code, SOLID & Refactoring's own Open/Closed Principle, stated formally

Hands-On Exercises

EXERCISE 1

Using this chapter's own decorator classes, build a chain representing "Coffee, Whip, Milk, Sugar" (in that wrapping order) and compute its total cost by hand from each decorator's own added amount, then verify it against running the actual code.

View solution

EXERCISE 2

Using this chapter's own `File` / `Directory` classes, add one more file (`notes.txt`, size `15`) directly into the `docs` directory from this chapter's own worked example, and compute the new total `root.get_size()` both by hand and by running the code.

View solution

EXERCISE 3

Using this chapter's own two patterns, explain why Decorator's own base-object-stays-unmodified guarantee and Composite's own uniform-interface guarantee are solving genuinely different problems, even though both patterns involve one object containing a reference to another.

View solution

CHAPTER 5 QUICK REFERENCE

- **Decorator:** wraps an object to add behavior at runtime, without modifying its class — verified accumulating correctly (`$2.00 → $2.50 → $2.80`), leaving the base object untouched, and keeping separate chains from the same base independent (`$2.80` vs. `$2.75`)
- **Composite:** a shared interface (`get_size()`) lets a caller treat a single leaf and a whole nested tree identically — verified: a 3-level-deep tree totaled `285`, matching a manual calculation exactly, using the same one-line call at every level with zero type-checking
- **Next chapter:** Structural Patterns III — Proxy and Flyweight

CHAPTER 6 OF 10

Structural Patterns III: Proxy & Flyweight

Design Patterns

Chapter 6 · Structural Patterns III: Proxy & Flyweight

Two more structural patterns, both about a wrapping object that stands in front of another object — but for very different reasons. **Proxy** controls access to an object, transparently, by standing in its place. **Flyweight** shares one object across many logical instances, to avoid creating thousands of near-identical objects that only differ in a small amount of context-specific data.

Proxy: Controlling Access to an Object Transparently

A proxy has the exact same interface as the real object it stands in for — the calling code can't tell the difference — but it adds a check, a delay, or a shortcut before (or instead of) forwarding the call through.

Virtual Proxy: Deferring an Expensive Object's Creation

```
class RealImage: load_count = 0 def __init__(self, filename): RealImage.load_count += 1
self.filename = filename def display(self): return f'Displaying {self.filename}' class ImageProxy:
def __init__(self, filename): self.filename = filename self._real_image = None def display(self):
if self._real_image is None: self._real_image = RealImage(self.filename) return
self._real_image.display()
```

VERIFIED DIRECTLY — THE EXPENSIVE OBJECT IS NEVER CREATED UNTIL IT'S ACTUALLY NEEDED

Creating `ImageProxy('photo.jpg')` leaves `RealImage.load_count` at `0` — nothing expensive has happened yet. The *first* call to `.display()` brings `load_count` to `1`, creating the real image for the first time. A *second* call to `.display()` leaves `load_count` at `1` — the proxy reuses the already-created real image rather than loading it again, while both calls return the identical correct result, `'Displaying photo.jpg'`.

Protection Proxy: Blocking an Unauthorized Call Before It Happens

```
class BankAccount: def __init__(self, balance): self.balance = balance def withdraw(self, amount):
self.balance -= amount return self.balance class BankAccountProxy: def __init__(self, account,
user_role): self._account = account self._user_role = user_role def withdraw(self, amount): if
self._user_role != 'owner': raise PermissionError('Only the account owner can withdraw') return
self._account.withdraw(amount)
```

VERIFIED DIRECTLY — THE REAL OBJECT'S OWN METHOD IS ONLY REACHED WHEN ACCESS IS LEGITIMATE

Wrapping a `BankAccount(500)` in a `BankAccountProxy(account, 'owner')`: calling `.withdraw(50)` succeeds, and the real account's balance correctly drops to `450`. Wrapping that *same* real account in a second proxy with role `'guest'`: calling `.withdraw(100)` raises `PermissionError` — and the real account's balance is confirmed still `450` afterward, proving `BankAccount.withdraw()` itself was never reached at all.

NOT THE SAME JOB AS CHAPTER 4'S ADAPTER

`BankAccountProxy`'s `withdraw(amount)` method has the *exact same signature* as the real `BankAccount.withdraw(amount)` it wraps — nothing is being translated. Chapter 4's Adapter existed specifically because `checkout()` and `LegacyPaymentGateway` couldn't communicate at all without translation. A Proxy assumes the interface already matches; its job is controlling *when* or *whether* a call reaches the real object, not making an incompatible one usable.

Flyweight: Sharing State Across Many Similar Objects

A forest with 10,000 trees doesn't need 10,000 separate copies of each tree species' own name, color, and texture — those details are identical for every tree of the same species. Flyweight splits an object's data into **intrinsic state** (shared, reused across every instance — the species' own name/color/texture) and **extrinsic state** (unique per instance — each individual tree's own `x, y` position), and hands the extrinsic part in from outside rather than storing it on the shared object.

```
class TreeType: # the flyweight — shared intrinsic state _cache = {} def __new__(cls, name, color, texture): key = (name, color, texture) if key not in cls._cache: cls._cache[key] = super().__new__(cls) cls._cache[key].name = name cls._cache[key].color = color cls._cache[key].texture = texture return cls._cache[key] def draw(self, x, y): # x, y arrive from outside — extrinsic return f'Drawing {self.name} tree at ({x},{y})' class Tree: # holds only the extrinsic state, plus a reference to the shared flyweight def __init__(self, x, y, tree_type): self.x = x; self.y = y; self.tree_type = tree_type def draw(self): return self.tree_type.draw(self.x, self.y)
```

VERIFIED DIRECTLY — 10,000 TREES, ONLY 3 SHARED FLYWEIGHT OBJECTS

Planting **10,000** trees, randomly chosen from just 3 species (Oak, Pine, Birch), produced exactly **3** distinct `TreeType` objects in `TreeType._cache` — one per species, no matter how many individual trees were planted. Of those 10,000 trees, **3,273** turned out to be Oak — and the first two Oak trees checked were confirmed to share the *identical* `TreeType` object (`tree_a.tree_type is tree_b.tree_type` → `True`), not just two separately-created objects with equal values.

WHY THIS ACTUALLY SAVES MEMORY

Every one of those 3,273 Oak `Tree` objects stores its own `x`, `y`, and a reference to the one shared `TreeType` — not its own copy of `'Oak'`, `'Dark Green'`, and `'Rough'`. Without Flyweight, each of the 10,000 trees would carry

its own full copy of its species' own name/color/texture data; with it, that data exists exactly 3 times total, regardless of whether the forest has 10,000 trees or 10,000,000.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Proxy's same-signature, access-gating wrapper, verified distinct from Chapter 4's translating Adapter	Chapter 7's Strategy pattern also swaps in an object behind an identical interface — but to change *behavior*, not *access*
Flyweight's intrinsic/extrinsic split, verified at 10,000-to-3 scale	A concrete example of trading object count for a small amount of externally-passed context — the same tradeoff shows up again in caching strategies generally
Both patterns keeping the calling code's interface completely unchanged	Every Structural pattern in this course (Adapter, Facade, Decorator, Composite, Proxy, Flyweight) shares this one property — worth noticing as the capstone approaches

Hands-On Exercises

EXERCISE 1

Extend this chapter's own `ImageProxy` with a `preload_count` that tracks how many times `.display()` was called in total (not just how many times the real image was actually loaded). Call `.display()` four times on the same proxy and verify both counts.

View solution

EXERCISE 2

Using this chapter's own `BankAccount` / `BankAccountProxy`, add a role `'viewer'` that is allowed to call a new `check_balance()` method (added to both classes) but still blocked from `withdraw()`. Verify a viewer-role proxy can check the balance but is still denied a withdrawal.

View solution

EXERCISE 3

Using this chapter's own `TreeType` / `Tree` / `Forest`-style setup, plant 500 trees using 7 distinct species instead of 3, and verify how many `TreeType` objects actually get created. Then explain, in your own words, what would happen to that count if every tree were instead given its own slightly different shade of color.

View solution

CHAPTER 6 QUICK REFERENCE

- **Proxy:** stands in for a real object behind an identical interface, controlling access — verified two ways: a virtual proxy deferring an expensive object's creation until first use (load count stayed at `0` until the first real call), and a protection proxy blocking an unauthorized call before the real object's own method was ever reached
- **Flyweight:** splits shared intrinsic state from unique extrinsic state, so many logical instances can reuse one physical object — verified: 10,000 planted trees across 3 species produced exactly `3` shared flyweight objects, with same-species trees confirmed to reference the identical object
- **Next chapter:** Behavioral Patterns I — Strategy and Template Method

CHAPTER 7 OF 10

Behavioral Patterns I: Strategy & Template Method

Design Patterns

Chapter 7 · Behavioral Patterns I: Strategy & Template Method

This course's first two categories — Creational (Chapters 2–3) and Structural (Chapters 4–6) — were about *creating* and *composing* objects. Behavioral patterns, starting here, are about how objects *communicate* and *share responsibility for an algorithm*. Strategy and Template Method both let one algorithm vary — but one does it through composition, swappable at runtime; the other through inheritance, fixed at the class level.

Strategy: Swapping an Algorithm at Runtime

Strategy defines a family of interchangeable algorithms behind one shared interface, and lets the object using them be handed a different one — even after it's already been created.

```
class StandardShipping: def calculate(self, weight): return weight * 0.5 class ExpressShipping: def calculate(self, weight): return weight * 1.2 + 5 class OvernightShipping: def calculate(self, weight): return weight * 2.5 + 15 class Order: def __init__(self, weight, shipping_strategy): self.weight = weight self.shipping_strategy = shipping_strategy def calculate_shipping(self): return self.shipping_strategy.calculate(self.weight) def set_shipping_strategy(self, strategy): self.shipping_strategy = strategy
```

VERIFIED DIRECTLY — THE SAME ORDER OBJECT PRODUCES THREE DIFFERENT RESULTS AS ITS STRATEGY CHANGES

An `Order(10, StandardShipping())` reports `5.0`. Calling `set_shipping_strategy(ExpressShipping())` on that *exact same* order object — no new `Order` created — changes the next `calculate_shipping()` call to `17.0`. Swapping again to `OvernightShipping()` gives `40.0`. All three match their hand-calculated values (`10×0.5` , `10×1.2+5` , `10×2.5+15`) exactly — the order's own weight and identity never changed; only which algorithm object it delegates to did.

NOT THE SAME JOB AS CHAPTER 6'S PROXY

`Order.calculate_shipping()` genuinely changes *what gets computed* depending on which strategy is plugged in. Chapter 6's `BankAccountProxy` never changed what `withdraw()` computed — it only decided whether the identical, unchanged computation was allowed to run at all. Strategy swaps *behavior*; Proxy gates *access* to one fixed behavior.

Template Method: Fixing the Skeleton, Letting Subclasses Fill In Steps

Template Method takes the opposite approach: the overall sequence of steps is locked in one place (the base class), and subclasses override only the specific steps that genuinely need to differ.

```
class CaffeineBeverage: def prepare_recipe(self): # the template method — fixed order return [self.boil_water(), self.brew(), self.pour_in_cup(), self.add_condiments()] def boil_water(self): return 'Boiling water' def pour_in_cup(self): return 'Pouring into cup' def brew(self): raise NotImplementedError def add_condiments(self): raise NotImplementedError class Tea(CaffeineBeverage): def brew(self): return 'Steeping the tea' def add_condiments(self): return 'Adding lemon' class Coffee(CaffeineBeverage): def brew(self): return 'Dripping coffee through filter' def add_condiments(self): return 'Adding sugar and milk'
```

VERIFIED DIRECTLY — THE SHARED STEPS ARE IDENTICAL; ONLY THE OVERRIDDEN STEPS DIFFER

`Tea().prepare_recipe()` returns `['Boiling water', 'Steeping the tea', 'Pouring into cup', 'Adding lemon']`. `Coffee().prepare_recipe()` returns `['Boiling water', 'Dripping coffee through filter', 'Pouring into cup', 'Adding sugar and milk']`. Step 1 (`boil_water`) and step 3 (`pour_in_cup`) are confirmed **identical** between the two — inherited, unmodified, from `CaffeineBeverage`. Only step 2 (`brew`) and step 4 (`add_condiments`) — the two steps each subclass actually overrides — differ.

VERIFIED DIRECTLY — THE BASE CLASS CAN'T BE USED ON ITS OWN

Calling `CaffeineBeverage().prepare_recipe()` directly — with no subclass, no overrides — correctly raises `NotImplementedError` the moment it reaches `self.brew()`. The base class defines the *shape* of the algorithm, not a runnable one by itself; it genuinely needs a subclass to supply the missing steps.

An Optional Step: the Hook Method

A hook is a step with a sensible default in the base class that a subclass may — but doesn't have to — override, letting a subclass skip or alter part of the fixed sequence without breaking it for everyone else.

```
class CaffeineBeverageWithHook(CaffeineBeverage): def prepare_recipe(self): steps = [self.boil_water(), self.brew(), self.pour_in_cup()] if self.customer_wants_condiments(): steps.append(self.add_condiments()) return steps def customer_wants_condiments(self): return True # the hook — default behavior, override is optional
```

VERIFIED DIRECTLY — ONE SUBCLASS USES THE HOOK'S DEFAULT, ANOTHER OVERRIDES IT TO SKIP A STEP

`TeaWithHook` doesn't override `customer_wants_condiments()` — its `prepare_recipe()` returns all **4** steps, including `'Adding lemon'`. `CoffeeWithHook` *does* override it, returning `False` — its `prepare_recipe()` correctly returns only **3** steps, with `add_condiments()` never even called. Both subclasses share the exact same `prepare_recipe()` method, inherited unmodified — the hook alone decided whether the fourth step ran.

Where This Connects

QUESTION	STRATEGY	TEMPLATE METHOD
How does behavior vary?	Composition — a swappable object plugged in at runtime	Inheritance — a subclass overriding fixed steps
Can it change after the object is created?	Yes — verified: the same `Order` object gave 3 different results after 2 runtime swaps	No — a `Tea` object is a `Tea` object for its whole lifetime; changing behavior means using a different subclass
Where does the overall sequence live?	Nowhere fixed — the caller decides what to call and when	Fixed once, in the base class's own template method — verified identical for every subclass

FORWARD REFERENCE

Chapter 1's very first example — `calculate_price_v2(price, strategy)` — was actually an early, informal look at Strategy, before this course had named it. Chapter 9's Command pattern will look structurally similar to Strategy (an object passed in to be called later) but for a genuinely different purpose: representing a request as an object, not swapping an algorithm.

Hands-On Exercises

EXERCISE 1

Add a fourth strategy, `FreeShipping` (always returns `0`, regardless of weight), to this chapter's own `Order` setup. Starting from an `Order(25, StandardShipping())`, swap through all four strategies in turn on the same order object and verify each result.

View solution

EXERCISE 2

Add a third subclass, `HotChocolate`, to this chapter's own `CaffeineBeverage` template. Verify its `prepare_recipe()` shares the exact same `boil_water` / `pour_in_cup` steps as `Tea` and `Coffee`, while its own `brew` / `add_condiments` steps differ from both.

View solution

EXERCISE 3

Explain, using this chapter's own two verified findings, why swapping an `Order`'s shipping strategy at runtime is possible, but "swapping" a `Tea` object into a `Coffee` at runtime is not something Template Method supports at all — what would you have to do instead?

View solution

CHAPTER 7 QUICK REFERENCE

- **Strategy:** a family of interchangeable algorithms behind one interface, held by composition — verified: the same `Order` object produced `5.0`, then `17.0`, then `40.0` across two runtime strategy swaps, with no new object ever created
- **Template Method:** a fixed algorithm skeleton in a base class, with specific steps left to subclasses — verified: two subclasses shared identical `boil_water`/`pour_in_cup` steps while their overridden `brew`/`add_condiments` steps genuinely differed, and a hook method let one subclass skip a step entirely (3 steps vs. 4)
- **Next chapter:** Behavioral Patterns II — Observer and State

CHAPTER 8 OF 10

Behavioral Patterns II: Observer & State

Design Patterns

Chapter 8 · Behavioral Patterns II: Observer & State

Two more ways an object's behavior can vary — but this time neither one is about picking an algorithm. **Observer** lets a changing object automatically notify a whole list of interested parties, without knowing anything about who they are. **State** lets an object's own methods behave completely differently depending on what it currently *is* — with no `if/elif` chain checking that anywhere.

Observer: Automatic Notification on Change

A subject keeps a list of observers and notifies every one of them whenever something changes — each observer decides for itself what, if anything, to do with that notification.

```
class Stock: # the subject def __init__(self, symbol, price): self.symbol = symbol; self._price = price; self._observers = [] def attach(self, observer): self._observers.append(observer) def detach(self, observer): self._observers.remove(observer) def set_price(self, price): self._price = price self._notify() def _notify(self): for observer in self._observers: observer.update(self.symbol, self._price) class PriceLogger: # an observer — just records def __init__(self): self.log = [] def update(self, symbol, price): self.log.append((symbol, price)) class AlertService: # an observer — reacts conditionally def __init__(self, threshold): self.threshold = threshold; self.alerts = [] def update(self, symbol, price): if price > self.threshold: self.alerts.append(f'{symbol} exceeded {self.threshold}: now {price}')
```

VERIFIED DIRECTLY — ONE PRICE CHANGE, TWO GENUINELY DIFFERENT REACTIONS, WITH NO COORDINATION BETWEEN THE OBSERVERS

With `logger` and `alert` (threshold `150`) both attached to the same `Stock('ACME', 100)`: calling `set_price(120)` adds an entry to `logger.log` but leaves `alert.alerts` empty (`120` doesn't exceed `150`). Calling `set_price(160)` adds a *second* entry to `logger.log` **and** triggers `alert.alerts` for the first time. `Stock` never checked what kind of observer it was talking to — it just called `.update()` on everything in its list.

VERIFIED DIRECTLY — DETACHING ONE OBSERVER STOPS ONLY THAT OBSERVER'S FUTURE UPDATES

After `stock.detach(logger)`, calling `set_price(200)` leaves `logger.log` completely unchanged (still 2 entries) — but `alert`, which was never detached, correctly receives the update and grows to 2 entries of its own. One observer stopped listening; the other kept listening; `Stock`'s own code needed no change to support either outcome.

State: Behavior That Changes With the Object's Own Condition

An order behaves differently depending on whether it's pending, paid, shipped, or delivered — calling `.ship()` should succeed on a paid order and fail on a pending one. State moves each condition's own rules into its own small class, and lets the main object simply delegate to whichever one is currently active.

```
class PendingState: name = 'Pending' def pay(self, order): order.state = PaidState() def ship(self, order): raise Exception('Cannot ship an unpaid order') def deliver(self, order): raise Exception('Cannot deliver an unpaid order') class PaidState: name = 'Paid' def pay(self, order): raise Exception('Order already paid') def ship(self, order): order.state = ShippedState() def deliver(self, order): raise Exception('Cannot deliver before shipping') # ShippedState and DeliveredState follow the same shape — # each state class only allows the transitions that make sense from it class OrderContext: def __init__(self): self.state = PendingState() def pay(self): self.state.pay(self) def ship(self): self.state.ship(self) def deliver(self): self.state.deliver(self) def status(self): return self.state.name
```

VERIFIED DIRECTLY — THE IDENTICAL METHOD CALL PRODUCES DIFFERENT BEHAVIOR AS THE ORDER'S OWN STATE CHANGES

On a fresh `OrderContext()` (status `'Pending'`), calling `.ship()` correctly raises `'Cannot ship an unpaid order'`, and the status is confirmed still `'Pending'` afterward — the blocked call left no trace. Calling `.pay()` then `.ship()` then `.deliver()` in sequence moves the status through `'Paid'` → `'Shipped'` → `'Delivered'` — each step calling a differently-behaving state object, even though `OrderContext.ship()`'s own code (`self.state.ship(self)`) never changed.

VERIFIED DIRECTLY — AN INVALID TRANSITION FROM THE FINAL STATE IS BLOCKED THE SAME WAY

Calling `.pay()` again on the now-`'Delivered'` order correctly raises `'Order already delivered'`, and the status is confirmed still `'Delivered'` afterward — the exact same blocked-call-leaves-no-trace behavior verified earlier for the pending order, now demonstrated at the opposite end of the state sequence.

WHERE THE IF/ELIF CHAIN WOULD HAVE LIVED INSTEAD

Without State, `OrderContext.ship()` would need its own `if self.status == 'Paid': ... elif self.status == 'Pending': raise ... elif ...` block — repeated inside `pay()`, `ship()`, and `deliver()` alike, with every new status requiring a new branch added to *all three* methods. Here, adding a new status means writing one new state class; none of `OrderContext`'s three methods change at all.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Observer's subject calling an identical `.update()` on every unknown-typed	The same "call the same method, let the object decide what happens" idea Chapter 7's Strategy already relied on — Observer applies it to notification

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
observer	instead of algorithm selection
State's swappable `self.state` object, changed by the state objects themselves	Structurally close to Chapter 7's Strategy (composition, a swappable object) — but here the object *being delegated to* is the one deciding when to swap itself out, not an outside caller
Both patterns eliminating a growing `if/elif` chain	Clean Code, SOLID & Refactoring's own treatment of conditional complexity as a recognized code smell

Hands-On Exercises

EXERCISE 1

Add a third observer, `MovingAverageTracker`, that records every price it receives and can report the average of all prices seen so far. Attach it alongside this chapter's own `logger` and `alert`, send three price updates, and verify its average is correct.

View solution

EXERCISE 2

Add a `CancelledState` to this chapter's own order system, reachable via a new `cancel()` method that's only allowed from `PendingState` or `PaidState` (not after shipping). Verify cancelling a pending order succeeds, and cancelling a shipped order is correctly blocked.

View solution

EXERCISE 3

Explain, using this chapter's own two verified patterns, why `Stock` needing to know nothing about `PriceLogger` or `AlertService`'s own internals is the same underlying idea as `OrderContext` needing to know nothing about what `PendingState.ship()` versus `PaidState.ship()` actually do.

View solution

CHAPTER 8 QUICK REFERENCE

- **Observer:** a subject notifies a list of registered observers automatically on change, without knowing their concrete types — verified: two observers reacted differently to the same price change, and detaching one stopped only its own future updates while the other kept receiving them

- **State:** an object delegates its own method calls to a swappable "current state" object, so behavior changes with condition rather than through `if/elif` checks — verified: the identical `.ship()` call raised an exception from `PendingState` but transitioned correctly from `PaidState`, with a blocked call at both ends of the sequence leaving the status unchanged
- **Next chapter:** Behavioral Patterns III — Command and Iterator

CHAPTER 9 OF 10

Behavioral Patterns III: Command & Iterator

Design Patterns

Chapter 9 · Behavioral Patterns III: Command & Iterator

This course's last pair of patterns before the capstone. **Command** turns "do this action" into an object in its own right — so it can be queued, logged, or undone. **Iterator** turns "go through this collection" into an object too — so the collection's own internal storage never has to leak out to the code walking through it.

Command: A Request as a Standalone Object

Instead of a remote control button directly calling `light.turn_on()`, it holds a small object that knows how to `execute()` the action — and, crucially, how to `undo()` it.

```
class LightOnCommand: def __init__(self, light): self.light = light def execute(self): self.light.turn_on() def undo(self): self.light.turn_off() class LightOffCommand: def __init__(self, light): self.light = light def execute(self): self.light.turn_off() def undo(self): self.light.turn_on() class RemoteControl: def __init__(self): self.history = [] def press_button(self, command): command.execute() self.history.append(command) def press_undo(self): if self.history: self.history.pop().undo()
```

VERIFIED DIRECTLY — UNDO CORRECTLY REVERSES WHICHEVER COMMAND WAS PRESSED LAST, WITHOUT REMOTECONTROL KNOWING WHAT ANY OF THEM DO

Pressing `LightOnCommand` then `LightOffCommand` leaves `light.is_on` at `False`. The first `press_undo()` reverses the *last* command pressed (the off command), bringing `light.is_on` back to `True`. A second `press_undo()` reverses the one before that (the on command), leaving `light.is_on` at `False` again. `RemoteControl`'s own code never mentions "on" or "off" anywhere — it only ever calls `.execute()` and `.undo()`.

Macro Commands: Composing Several Commands Into One

Because a command is just an object with `execute()`/`undo()`, a `MacroCommand` holding a list of other commands can implement the exact same interface — this is Chapter 5's Composite pattern, reapplied to commands instead of files and directories.

```
class MacroCommand: def __init__(self, commands): self.commands = commands def execute(self): for c in self.commands: c.execute() def undo(self): for c in reversed(self.commands): c.undo()
```

VERIFIED DIRECTLY — A MACRO'S OWN UNDO RUNS ITS SUB-COMMANDS IN STRICTLY REVERSED ORDER

A `MacroCommand([LightOnCommand(light), FanOnCommand(fan)])`, when executed, turns on *both* the light and the fan, with a call log confirming the order `['Living Room Light ON', 'Ceiling Fan ON']`. Calling `undo()` on that same macro produces the log entries `['Ceiling Fan OFF (undo)', 'Living Room Light OFF (undo)']` — the **fan** (turned on second) is undone **first**, and the **light** (turned on first) is undone **last**. `RemoteControl.press_undo()` triggered this whole reversed sequence with the exact same one-line call it uses for a single command.

Iterator: Traversing a Collection Without Exposing How It's Stored

A collection hands out a separate iterator object that knows how to walk through it — the calling code never touches the collection's own internal list, dict, or tree directly.

```
class BookShelf: # backed by a list def __init__(self): self._books = [] def add(self, book): self._books.append(book) def create_iterator(self): return BookShelfIterator(self._books) class BookShelfIterator: def __init__(self, books): self._books = books; self._index = 0 def has_next(self): return self._index < len(self._books) def next(self): book = self._books[self._index]; self._index += 1 return book class Playlist: # backed by a dict — genuinely different storage def __init__(self): self._songs = {} def add(self, song_id, title): self._songs[song_id] = title def create_iterator(self): return PlaylistIterator(list(self._songs.values()))
```

VERIFIED DIRECTLY — THE IDENTICAL CLIENT-FACING LOOP WORKS OVER TWO GENUINELY DIFFERENT STORAGE TYPES

A shared `print_all(iterator)` function (a plain `while iterator.has_next(): ...` loop) run against a `BookShelf` (internally a `list`) returns `['Book A', 'Book B', 'Book C']`. The *same* `print_all()` function, unchanged, run against a `Playlist` (internally a `dict`) returns `['Song X', 'Song Y', 'Song Z']`. Neither call ever touches `_books` or `_songs` directly — `print_all()` has no idea one collection is a list and the other is a dict, and doesn't need to.

VERIFIED DIRECTLY — TWO INDEPENDENT ITERATORS OVER THE SAME COLLECTION DON'T INTERFERE WITH EACH OTHER

Creating `it1` from a shelf and advancing it twice (`'Book A'`, `'Book B'`) leaves it partway through. Creating a *second*, brand-new `it2` from the *same* shelf starts fresh from the beginning — its first call also returns `'Book A'`. Advancing `it1` again correctly continues from where *it* left off (`'Book C'`), leaving `it1.has_next()` as `False` while `it2.has_next()` is still `True`, having only consumed one item. Each iterator owns its own `_index` — the shelf itself was never asked to remember a position.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
`MacroCommand`'s own execute/undo list, verified running in forward and reversed order	Chapter 5's Composite — a command composed of commands, exactly like a Directory composed of Files and Directories
Command wrapping a call behind an identical `execute()` interface	Structurally close to Chapter 7's Strategy — but Command represents a specific request to be replayed or undone later, not a swappable algorithm to run immediately
Iterator's shared traversal interface over genuinely different storage	Pseudocode & Algorithmic Problem-Solving's own "the same algorithm, expressed independently of implementation" theme, applied to walking a collection specifically

Hands-On Exercises

EXERCISE 1

Build a `MacroCommand` that turns *off* both the light and the fan from this chapter's own example (using `LightOffCommand` / `FanOffCommand`, which you should also write, following the existing on-command shape). Press it, then undo it, and verify both the resulting device states and the call-log order at each step.

View solution

EXERCISE 2

Add a third collection type, `PlayingCardDeck`, backed by a `tuple` of card names rather than a list or a dict, with its own iterator following this chapter's own `has_next()` / `next()` shape. Verify `print_all()` works on it unmodified.

View solution

EXERCISE 3

Explain, using this chapter's own two verified patterns, why a `MacroCommand`'s reversed-order undo is a genuine correctness requirement (not just a style choice), while the order two separate `BookShelfIterator` objects are created in has no such requirement at all.

View solution

CHAPTER 9 QUICK REFERENCE

- **Command:** a request becomes a standalone object with `execute()` / `undo()`, letting it be queued, logged, or reversed — verified: two consecutive undos correctly reversed a remote's last two button presses in order, and a `MacroCommand`'s own undo ran its sub-commands in strictly reversed order (fan undone before light, after light was turned on before fan)
- **Iterator:** a separate object handles traversal, so a collection's internal storage never has to be exposed — verified: the identical client loop walked a list-backed and a dict-backed collection with no changes, and two independent iterators over the same collection tracked their own positions without interfering
- **Next chapter:** Capstone — refactoring a real, patternless codebase using several patterns together

CHAPTER 10 OF 10

Capstone — Refactoring a Real Codebase with Multiple Patterns

Design Patterns

Chapter 10 · Capstone: Refactoring a Real Codebase with Multiple Patterns

One continuous worked project: a genuinely tangled, patternless order-processing function, refactored step by step using four patterns from across this course — chosen deliberately, not exhaustively. Chapter 1 opened with a warning that patterns can be misapplied or over-engineered. This chapter closes by actually testing that warning against a real decision: which patterns earn their place here, and which ones this course covered but this code simply doesn't need.

Before: One Tangled Function Doing Everything

```
def process_order_bad(customer_type, items_total, weight, notify_email, notify_sms,
priority_shipping): # discount logic tangled directly into the function if customer_type == 'vip':
discount = items_total * 0.20 elif customer_type == 'loyalty': discount = items_total * 0.10 else:
discount = 0 final_total = items_total - discount # shipping logic tangled in too if
priority_shipping: shipping_cost = weight * 2.5 + 15 else: shipping_cost = weight * 0.5
final_total += shipping_cost status = 'pending' status = 'paid' # pretend payment succeeds — no
real transition logic at all # notification logic hardcoded per channel, right here log = [] if
notify_email: log.append(f'Emailing customer: your order total is {final_total}') if notify_sms:
log.append(f'Texting customer: your order total is {final_total}') return final_total, status, log
```

VERIFIED DIRECTLY — THE TANGLED VERSION DOES PRODUCE A CORRECT NUMBER

Calling `process_order_bad('vip', 200, 10, True, True, True)` returns a total of `200.0`, status `'paid'`, and log entries for both email and SMS — matching a hand calculation (`200 - 200×0.20 + (10×2.5+15) = 200.0`) exactly. The code isn't *wrong*. It's tangled.

THE REAL PROBLEMS, NAMED SPECIFICALLY

- Adding a new customer type means editing an `if/elif` chain that has nothing to do with shipping or notifications, risking a typo in unrelated logic.
- Adding a new notification channel (push notifications, say) means adding yet another `if notify_x:` line, and every caller now needs a new boolean parameter.
- `status = 'pending'; status = 'paid'` is not a state transition — it's two assignments with no rule preventing an already-cancelled order from being "paid" again.

- Six positional parameters (`customer_type`, `items_total`, `weight`, `notify_email`, `notify_sms`, `priority_shipping`) is already hard to call correctly, and every new option makes it worse — exactly Chapter 3's telescoping-constructor problem, just as a function's parameter list instead of a class's.

After: Four Patterns, Each Solving One Named Problem

Strategy (Chapter 7) — Discount and Shipping

Reusing this course's own established shipping-strategy shape directly from Chapter 7:

```
class VipDiscount: def apply(self, total): return total * 0.20 class LoyaltyDiscount: def
apply(self, total): return total * 0.10 class NoDiscount: def apply(self, total): return 0 class
StandardShippingStrategy: def calculate(self, weight): return weight * 0.5 class
PriorityShippingStrategy: def calculate(self, weight): return weight * 2.5 + 15
```

Observer (Chapter 8) — Notifications

```
class EmailNotifier: def __init__(self): self.messages = [] def update(self, message):
self.messages.append(f'EMAIL: {message}') class SMSNotifier: def __init__(self): self.messages =
[] def update(self, message): self.messages.append(f'SMS: {message}')
```

State (Chapter 8) — Order Status

```
class PendingOrderState: name = 'Pending' def pay(self, order): order.state = PaidOrderState()
order._notify(f'Payment received. Total: {order.calculate_total()}') def cancel(self, order):
order.state = CancelledOrderState() order._notify('Order cancelled before payment') class
PaidOrderState: name = 'Paid' def pay(self, order): raise Exception('Order already paid') def
cancel(self, order): order.state = CancelledOrderState() order._notify('Order cancelled after
payment - refund issued') # CancelledOrderState blocks both pay() and cancel(), matching Chapter
8's own final-state shape
```

Builder (Chapter 3) — Constructing an Order

```
class OrderBuilder: def __init__(self): self._items_total = 0; self._weight = 0
self._discount_strategy = NoDiscount() self._shipping_strategy = StandardShippingStrategy()
self._notifiers = [] def items_total(self, amount): self._items_total = amount; return self def
weight(self, w): self._weight = w; return self def discount_strategy(self, strategy):
self._discount_strategy = strategy; return self def shipping_strategy(self, strategy):
self._shipping_strategy = strategy; return self def add_notifier(self, notifier):
self._notifiers.append(notifier); return self def build(self): return Order(self._items_total,
self._weight, self._discount_strategy, self._shipping_strategy, self._notifiers)
```

With `Order` tying it together — `calculate_total()` delegates to whichever strategies were plugged in, `pay()` / `cancel()` delegate to whichever state is current, and both notify every attached observer automatically:

```
class Order: def __init__(self, items_total, weight, discount_strategy, shipping_strategy, notifiers): self.items_total = items_total; self.weight = weight self.discount_strategy = discount_strategy; self.shipping_strategy = shipping_strategy self._observers = list(notifiers) self.state = PendingOrderState() def _notify(self, message): for o in self._observers: o.update(message) def calculate_total(self): discount = self.discount_strategy.apply(self.items_total) shipping = self.shipping_strategy.calculate(self.weight) return self.items_total - discount + shipping def pay(self): self.state.pay(self) def cancel(self): self.state.cancel(self) def status(self): return self.state.name
```

VERIFIED DIRECTLY — THE REFACTORED VERSION PRODUCES THE EXACT SAME TOTAL AS THE TANGLED ORIGINAL

Building the identical VIP, priority-shipping order through `OrderBuilder` and calling `calculate_total()` gives `200.0` — matching `process_order_bad()`'s own result exactly, for the same inputs. The refactor changed *how the code is organized*, not *what it computes*.

VERIFIED DIRECTLY — PAY(), CANCEL(), AND BLOCKED TRANSITIONS ALL WORK EXACTLY AS STATE PREDICTS

Calling `.pay()` moves status from `'Pending'` to `'Paid'` and correctly notifies both `email` and `sms` observers with a payment message. Calling `.cancel()` afterward moves status to `'Cancelled'` and sends a *second*, differently-worded notification ("refund issued") — confirming the state object, not a repeated `if` check, decided what message to send. A further `.pay()` and `.cancel()` on the now-cancelled order both correctly raise exceptions, with status still `'Cancelled'` afterward.

VERIFIED DIRECTLY — A SECOND, DIFFERENTLY-CONFIGURED ORDER STAYS FULLY ISOLATED FROM THE FIRST

A second order (no discount, standard shipping, only an email notifier attached) built through the same `OrderBuilder` reports `83.0` — matching `80 - 0 + 6×0.5` exactly — and its own `.pay()` only reaches its own email notifier. No SMS message appears anywhere for this order, since none was attached to it: two orders built from the same classes never share state.

What Wasn't Used, and Why

Chapter 1 warned that patterns can be misapplied through over-engineering. The honest test of that warning isn't listing every pattern this course covered — it's explaining, for this specific code, which ones genuinely don't belong.

PATTERN (CHAPTER)	WHY IT WASN'T USED HERE
Command (Ch.9)	Genuinely tempting — `cancel()` looks like a candidate for an undoable command object. But this order system never needs to *undo a cancellation* or queue actions for later; State already gives `cancel()` exactly the behavior it needs (blocked once already cancelled) with less machinery. Wrapping it in a Command here would add a class with no caller that ever uses its own `undo()`.
Decorator (Ch.5)	Would fit if orders needed optional, stackable add-ons priced independently (gift wrapping, insurance). This order system has none of that — `calculate_total()` already has exactly two swappable pieces (discount, shipping), which Strategy already covers.
Singleton / Factory Method (Ch.2)	Nothing here needs exactly-one-instance, and there's no family of related object *creation* varying by subclass — `OrderBuilder` already handles construction directly.
Adapter / Facade (Ch.4)	No incompatible interface to translate, and no scattered multi-call sequence complex enough to be worth bundling behind one method.
Composite / Iterator (Ch.5, Ch.9)	No tree-shaped data here, and nothing that needs traversal hidden behind a shared interface — a plain `list` of notifiers is already exactly as simple as this problem requires.

REVISITING CHAPTER 1'S WARNING, HONESTLY

Four patterns solved four real, separately-identifiable problems in the tangled original: a growing `if/elif` for discounts and another for shipping (Strategy), a status field with no real transition rules (State), notification logic hardcoded per channel (Observer), and a parameter list that would only grow (Builder). Every pattern used here corresponds to a specific paragraph in the "real problems" warn-box above it. Design patterns aren't a checklist to apply exhaustively — they're a vocabulary for naming a problem precisely enough to solve it with the smallest tool that actually fits. The five patterns left out of this table aren't gaps in the refactor; reaching for them here would have been exactly the over-engineering Chapter 1 warned about.

Course Summary

CATEGORY	PATTERNS COVERED
Creational (Ch.2-3)	Singleton, Factory Method, Abstract Factory, Builder
Structural (Ch.4-6)	Adapter, Facade, Decorator, Composite, Proxy, Flyweight
Behavioral (Ch.7-9)	Strategy, Template Method, Observer, State, Command, Iterator

Twelve of the 23 classic Gang-of-Four patterns, each verified with real, runnable code rather than taken on faith. The 11 left out entirely (Chain of Responsibility, Mediator, Memento, Visitor, Interpreter, Bridge, Prototype, and others, named in Chapter 1) remain a genuine gap for anyone going deeper — but the twelve covered here are the ones a working programmer runs into most often.

Hands-On Exercises

EXERCISE 1

Add a `LoyaltyDiscount`-and-standard-shipping order to this chapter's own capstone code (using the existing `OrderBuilder` and existing strategy classes — no new classes needed) with `items_total=150`, `weight=8`. Verify its total by hand and against the running code.

View solution

EXERCISE 2

Add a `customer_type` parameter's worth of new behavior without touching `Order`, `OrderBuilder`, or any existing strategy class: create a new `NewCustomerDiscount` strategy (a flat $5 off, applied only if `items_total` is over $50, otherwise $0) and build an order using it. Verify both the over-$50 and under-$50 cases.

View solution

EXERCISE 3

Exercise 2 added a whole new discount without editing a single existing class. Explain specifically which part of the *tangled* `process_order_bad()` function would have needed to change to add that same "$5 off orders over $50" rule, and why that change is riskier than what Exercise 2 actually required.

View solution

CHAPTER 10 QUICK REFERENCE — AND COURSE QUICK REFERENCE

- **This chapter:** a tangled order-processing function was refactored using Strategy (discount, shipping), Builder (order construction), Observer (notifications), and State (status) — verified producing the exact same total (`200.0`) as the original, with every pattern traced back to a specific, named problem in the original code
- **The honest boundary:** five more patterns from this course (Command, Decorator, Singleton/Factory Method, Adapter/Facade, Composite/Iterator) were deliberately left out, each with a stated reason — over-engineering means reaching for a pattern the code doesn't actually need
- **Course scope:** 12 of the 23 classic GoF patterns across Creational, Structural, and Behavioral categories — every example in all 10 chapters verified with real, runnable code, not asserted
- **Where this connects:** Pseudocode & Algorithmic Problem-Solving (this course's own prerequisite); Software Architecture & System Design and Clean Code, SOLID & Refactoring, both still reserved in this same Software Development subject