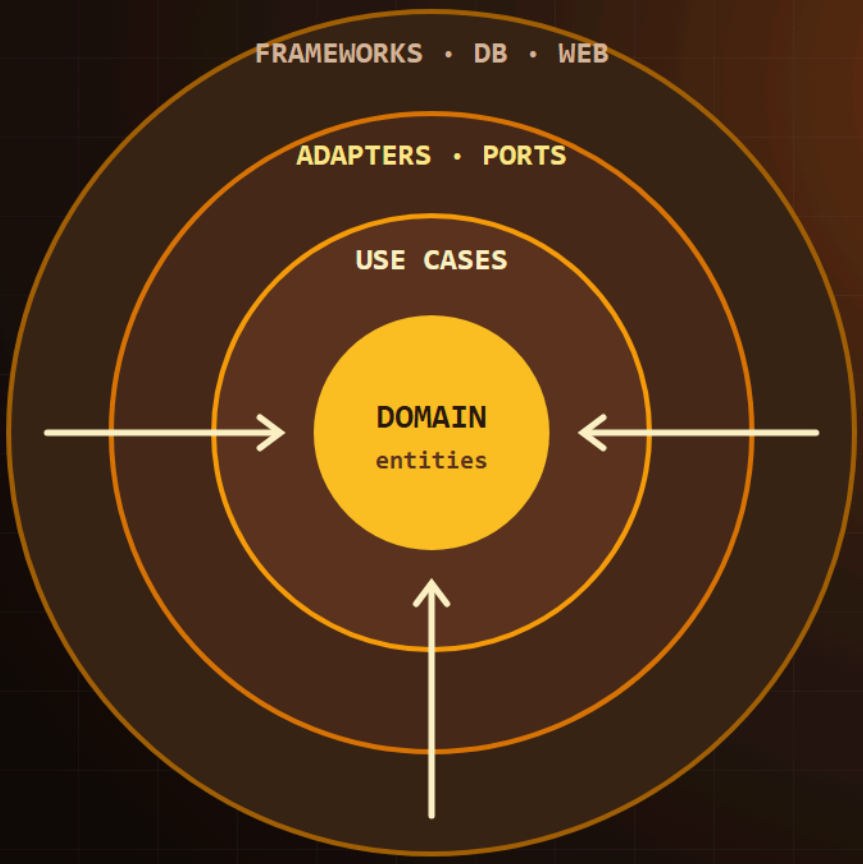


Strategy «pricing»

Adapter «port»

State «lifecycle»

Decorator

Specification

Repository

A PRACTICAL GUIDE

Clean Architecture and Design Patterns in Practice

Writing Maintainable Code

SOLID · 20+ design patterns · Hexagonal & Onion
Domain-Driven Design · Refactoring legacy code

PRINCIPLES · PATTERNS · ARCHITECTURE · DDD

THENEXBOOKS

Clean Architecture and Design Patterns in Practice

Writing Maintainable Code
SOLID, the classic design patterns, Clean and Hexagonal Architecture, and Domain-Driven Design, applied to one real codebase

First Edition

THENEXBOOKS

Clean Architecture and Design Patterns in Practice
Writing Maintainable Code

First Edition, 2026

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

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

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This book discusses ideas that originated in the work of many practitioners, including the classic design patterns catalog, the SOLID principles, Clean, Hexagonal, and Onion Architecture, Domain-Driven Design, and refactoring. Their originators are credited where the ideas are introduced. This book is an independent work; it is not affiliated with, sponsored by, or endorsed by any of them or their publishers. TypeScript is a trademark of Microsoft Corporation. Node.js is a trademark of the OpenJS Foundation. Other product names are trademarks of their respective owners and are used for identification only.

Brewline, the running example in this book, is fictional, as are all customers, cafés, carriers, payment providers, and systems mentioned in its examples. Any resemblance to real companies or products is coincidental.

Preface

Almost every codebase starts clean. A few files, a handful of features, a team that knows every corner. Then the product succeeds. Features pile up, deadlines tighten, people join and leave, and one day a change that should take an afternoon takes two weeks — and breaks something unrelated on the way out. Nobody decided to write unmaintainable code. It accumulated, one reasonable shortcut at a time.

The good news is that the forces that make code hard to change are well understood. Decades of practice have given us principles for managing dependencies, a shared vocabulary of design patterns, architectural styles that keep business rules independent of frameworks and databases, and techniques for improving existing code safely. The bad news is that these ideas are often taught in isolation, with toy examples, or as rules to follow rather than trade-offs to weigh. Teams either ignore them or apply them everywhere, and both paths lead back to code that is hard to change.

This book takes a practical path. We take one realistic system — a coffee roastery's online platform that grew into a tangled monolith — and improve it step by step. Every principle and pattern appears where it solves a real problem in that codebase, with before-and-after code, the costs it introduces, and the situations where you should not use it at all.

Who This Book Is For

- **Developers** who can ship features but want their code to stay easy to change as the system grows.
- **Backend and full-stack engineers** who want to separate business rules from frameworks, databases, and external services.
- **Engineers inheriting legacy code** who need safe, incremental ways to improve it without a rewrite.
- **Tech leads and architects** who set design conventions, review code, and make architectural decisions with a team.

What You Will Learn

- How coupling, cohesion, and the cost of change drive every design decision.
- Clean code habits, the SOLID principles, composition, and dependency injection — without dogma.
- How to refactor safely with code smells, characterization tests, and small steps.
- The creational, structural, and behavioral patterns you will actually use, in modern TypeScript, and when to avoid them.
- Value objects, state machines, specifications, and domain events.
- Clean, Hexagonal, and Onion Architecture: the dependency rule, use cases, ports and adapters, and enforcing boundaries.

- Domain-Driven Design essentials: bounded contexts, aggregates, repositories, CQRS, and the outbox pattern.
- Testing an architecture, migrating legacy code with the Strangler Fig, and handling cross-cutting concerns cleanly.

How This Book Is Organized

Part I — Foundations of Maintainable Code (Chapters 1–5) explains why code becomes hard to change and covers clean code, SOLID, composition, dependency injection, and safe refactoring. **Part II — Creational and Structural Patterns** (Chapters 6–10) introduces patterns as a tool, not a goal, and covers the creational and structural patterns plus value objects. **Part III — Behavioral Patterns** (Chapters 11–15) covers strategies, events, commands, pipelines, state machines, and specifications. **Part IV — Clean Architecture** (Chapters 16–20) organizes the whole application around its business rules. **Part V — Domain-Driven Design and Data** (Chapters 21–24) models the domain in depth and keeps persistence and integration from polluting it. **Part VI — Sustaining Maintainability** (Chapters 25–28) covers testing, legacy migration, cross-cutting concerns, and a complete case study. The appendices provide a cheat sheet, templates, a review checklist, and a glossary.

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

The Running Example: Brewline

Throughout the book we work on **Brewline**, a fictional specialty-coffee roastery that sells single bags and subscriptions online, supplies cafés wholesale, and ships worldwide. Brewline's first version was built fast, and it shows: a giant order service, SQL inside HTTP handlers, pricing rules scattered across the code, and a fragile integration with an old ERP system. Over the book we untangle it — extracting value objects, introducing patterns where they pay off, separating use cases from frameworks, splitting the system into bounded contexts, and migrating it gradually while it keeps running.

Code and Conventions

Examples are written in TypeScript with `strict` type checking, because it reads naturally for developers coming from Java, C#, Kotlin, Go, or JavaScript and supports both object-oriented and functional styles. The ideas are language-independent; where an idea looks notably different in another language, a Note says so. Frameworks and libraries appear only in the outer layers, which is exactly the point.

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

> **NOTE**
> Background information, history, or how an idea looks in another language.

> **WARNING**
> A common mistake, an over-engineering trap, or a design choice that can make code harder to change.

Code appears in `monospace`. Long lines are wrapped to fit the page. Money is always represented in integer minor units (cents), never as floating-point numbers.

Contents

Preface

PART I — FOUNDATIONS OF MAINTAINABLE CODE

1 Why Maintainability Matters: Cost of Change, Coupling, and Cohesion

2 Clean Code Essentials: Naming, Functions, and Readable Structure

3 SOLID Principles in Practice

4 Composition, Interfaces, and Dependency Injection

5 Refactoring Safely: Code Smells, Tests as a Safety Net, and Small Steps

PART II — CREATIONAL AND STRUCTURAL PATTERNS

6 Thinking in Patterns: When Patterns Help and When They Hurt

7 Creational Patterns: Factory Method, Abstract Factory, Builder, and Singleton

8 Structural Patterns I: Adapter, Facade, and Decorator

9 Structural Patterns II: Composite, Proxy, and Bridge

10 Value Objects, Immutability, and the Null Object Pattern

PART III — BEHAVIORAL PATTERNS

11 Strategy, Template Method, and Policy Objects

12 Observer, Domain Events, and Publish-Subscribe

13 Command, Chain of Responsibility, and Middleware Pipelines

14 The State Pattern and Explicit State Machines

15 Specification, Visitor, and Iterator

PART IV — CLEAN ARCHITECTURE

16 Architecture Fundamentals: Layers, Boundaries, and the Dependency Rule

17 Clean, Hexagonal, and Onion Architecture Compared

18 Entities and Use Cases: Modeling the Core of the Application

19 Ports and Adapters: Controllers, Presenters, Gateways, and Repositories

20 Organizing Code: Package by Feature, Modules, and Enforcing Boundaries

PART V — DOMAIN-DRIVEN DESIGN AND DATA

21 Domain-Driven Design Essentials: Ubiquitous Language and Bounded Contexts

22 Aggregates, Invariants, and Domain Events

23 Persistence Without Pollution: Repositories, Unit of Work, and Mapping

24 CQRS, Event-Driven Integration, and the Outbox Pattern

PART VI — SUSTAINING MAINTAINABILITY

25 Testing an Architecture: Unit, Integration, Contract, and Architecture Tests

26 Refactoring Legacy Code Toward Clean Architecture: Seams and the Strangler Fig

27 Cross-Cutting Concerns: Errors, Validation, Logging, and Observability in a Clean Design

28 Case Study: Evolving Brewline from a Big Ball of Mud to a Clean Architecture

APPENDICES

A Patterns and Principles Cheat Sheet

B Templates and Reference

C Maintainability Review Checklist

D Glossary

PART I

Foundations of Maintainable Code

Why code becomes hard to change, and the everyday habits and principles that keep it easy: clean code, SOLID, composition, dependency injection, and refactoring in small, safe steps.

1 · Why Maintainability Matters
2 · Clean Code Essentials
3 · SOLID Principles in Practice
4 · Composition and Dependency Injection
5 · Refactoring Safely

CHAPTER 1 · PART I — FOUNDATIONS OF MAINTAINABLE CODE

Why Maintainability Matters: Cost of Change, Coupling, and Cohesion

Most of the life of a codebase is spent being changed, not being written. This chapter explains what makes code easy or painful to change, introduces the two forces that decide it — coupling and cohesion — and takes a first honest look at Brewline, the tangled monolith we will clean up over the rest of the book.

Code Is Read and Changed Far More Than It Is Written

When you build a feature for the first time, the problem feels like a writing problem: you need to get the behavior into the code. But the moment that feature ships, the problem changes. Somebody will need to fix a bug in it, extend it for a new market, speed it up, make it work with a new payment provider, or delete half of it because the business changed its mind. The code you wrote in an afternoon may be read, debugged, and modified by a dozen people over several years. Every one of those later visits costs time, attention, and risk.

That is the practical reason this book exists. Design principles, patterns, and architecture styles are not decorations for impressing reviewers. They are tools for keeping the **cost of change** low — for making sure that the hundredth change to a system is not dramatically harder than the tenth. When teams say a codebase has become "hard to work in," they are usually describing the same experience: every small request touches many files, every fix seems to break something unrelated, nobody is sure what a function really does, and releases feel scary.

We will not claim a precise percentage of software cost that goes to maintenance; estimates vary widely and depend on how you define the word. What is uncontroversial is the qualitative picture that every experienced developer recognizes: successful software lives a long time, and the work of keeping it useful dwarfs the work of creating its first version.

What "Maintainable" Actually Means

"Maintainable" is one of those words everyone agrees with and nobody defines. To make it useful, we break it into four properties you can actually observe in a codebase. Throughout the book, when we say a change makes code "more maintainable," we mean it improves at least one of these without seriously harming the others.

Property	Question to ask	Warning sign
Understandable	Can a new teammate explain what this module does after reading it once?	Tribal knowledge; "ask Dana, she wrote it"

Changeable	Does a typical feature touch one area or ten?	Shotgun edits across many files
Testable	Can we check the business rule without a database or network?	Tests need a full environment to run
Deployable	Can we release a small change safely and quickly?	Big-bang releases; manual checklists

These properties reinforce one another. Code that is easy to understand is easier to change correctly. Code that is easy to test can be changed with confidence, because the tests tell you quickly when you broke something. Code that is split into well-defined pieces is easier to deploy in small increments. And the reverse is also true: a module that nobody understands tends to be untested, because nobody knows what to assert; untested code tends to be changed timidly, by adding special cases rather than restructuring; and a growing pile of special cases makes the code even harder to understand. That feedback loop is how a codebase slowly turns into what developers often call a *big ball of mud*.

Maintainability is relative to the changes you actually make

No design is maintainable in the abstract. A structure that makes adding new payment providers trivial may make changing the order total calculation awkward. Good design is about making the **likely** changes cheap. That means you need some understanding of the business: which parts are stable, which parts churn every sprint, and which parts are likely to vary in the future. A roastery will almost certainly add new coffees, new discount rules, and new shipping options. It is much less likely to stop needing orders. We will return to this idea constantly: design effort should follow the direction of change.

> **NOTE**
>
> Maintainability is not the only quality that matters. Performance, security, correctness, and time to market are real concerns too. A throwaway prototype for a trade show does not need layered architecture. The goal of this book is not to maximize structure everywhere but to give you the judgment to know where structure pays for itself.

Meet Brewline: A Monolith That Grew Up Too Fast

Our running example is Brewline, a fictional specialty-coffee roastery. It sells single bags of coffee online, runs weekly, biweekly, and monthly subscriptions, supplies wholesale customers such as cafés, and ships worldwide. It started, like many real businesses, as a quick project: one developer, one Express server, one database, and a deadline. It worked, and the business grew.

Then the requests started. Wholesale customers needed different prices. Subscriptions needed pause and resume. A second shipping carrier, SwiftPost, was added next to the original carrier, Parcelo. Finance insisted that every paid order be pushed into the company's accounting system, the aging

OldBooks ERP. Marketing wanted promotions. Each request was handled in the fastest way available at the time — usually by adding another branch to an existing function. No single decision was unreasonable, yet together they produced a codebase the team now dreads.

Here are the pain points the Brewline team lists when asked what slows them down:

- **A 600-line `OrderService`.** It validates input, calculates prices, applies discounts, charges cards through the Paybrook payment API, books shipments, sends emails, and writes to OldBooks. Everyone edits it; merge conflicts are routine.
- **SQL in controllers.** Several HTTP route handlers build SQL strings directly, so the shape of database tables leaks into request handling and response formatting.
- **Customer-type checks everywhere.** Expressions like `if (customer.type === "wholesale")` appear in pricing, shipping, invoicing, email templates, and reports. Adding a new tier would mean finding and changing all of them.
- **No fast tests.** The few tests that exist spin up a real database and call a sandbox payment API, so they are slow, flaky, and rarely run locally.
- **Fear of deploys.** Because nobody can predict the blast radius of a change, releases are batched and done late on Friday by the one person who "knows the system."

Let us look at a representative slice of that `OrderService`. It is shortened to fit the page, but every line reflects the style of the real thing.

src/services/order-service.ts (before)

```
import { Pool } from "pg";
import nodemailer from "nodemailer";

const db = new Pool({ connectionString: process.env.DATABASE_URL });

export class OrderService {
  async placeOrder(customerId: string, items: { sku: string; qty: number }[],
                   cardToken: string, promo?: string): Promise<string> {
    const c = await db.query("SELECT * FROM customers WHERE id = $1", [customerId]);
    const customer = c.rows[0];
    let total = 0;
    for (const it of items) {
      const p = await db.query("SELECT price FROM skus WHERE sku = $1", [it.sku]);
      let price: number = p.rows[0].price; // dollars, as a float
      if (customer.type === "wholesale") price = price * 0.8;
      if (customer.type === "wholesale" && it.qty >= 10) price = price * 0.95;
      total += price * it.qty;
    }
    if (promo === "WELCOME10" && customer.type !== "wholesale") total = total * 0.9;
    const shipping = customer.country === "US" ? 5.99 : 14.99;
    total += customer.type === "wholesale" && total > 200 ? 0 : shipping;

    const res = await fetch("https://api.paybrook.example/charges", {
      method: "POST",
      headers: { Authorization: `Bearer ${process.env.PAYBROOK_KEY}` },
      body: JSON.stringify({ amount: Math.round(total * 100), token: cardToken }),
```

```
    });
    if (!res.ok) throw new Error("payment failed");

    const o = await db.query(
      "INSERT INTO orders (customer_id, total, status) VALUES ($1, $2, 'paid') " +
        "RETURNING id",
      [customerId, total],
    );
    const mailer = nodemailer.createTransport({ host: "smtp.brewline.example" });
    await mailer.sendMail({
      to: customer.email,
      subject: customer.type === "wholesale" ? "Wholesale order" : "Thanks!",
      text: `Order ${o.rows[0].id} total ${total.toFixed(2)}`,
    });
    return o.rows[0].id;
  }
  // ...about 550 more lines: cancelOrder, refund, shipOrder, syncToOldBooks...
}
```

Nothing here is exotic. Code like this ships every day, and for a small shop in its first months it is arguably fine. But count the reasons this one method might have to change: a new discount rule, a new customer tier, a new shipping rate, a new payment provider, a change to the database schema, a new email template, a new currency, a decision to charge the card later in the flow. Each of those reasons belongs to a different person or team, and every one of them lands in the same function. That is the heart of the maintainability problem, and the rest of this chapter gives us the vocabulary to describe it precisely.

> **WARNING**
>
> Notice that prices are stored and computed as floating-point dollars. Binary floating-point cannot represent most decimal fractions exactly, so repeated multiplications and additions accumulate tiny rounding errors that eventually show up as off-by-one-cent totals. From Chapter 10 on, Brewline uses a `Money` value object that stores integer cents and a currency. Never use floats for money in production code.

The Cost of Change

Think about what it takes to make one change to the method above — say, "wholesale customers in the EU get free shipping above 150 EUR." To make that change safely, a developer must:

1. Find every place that computes shipping. In Brewline there are at least three: here, in the quote endpoint, and in the subscription renewal job.
2. Understand how each place interacts with discounts, currency, and customer type, because they are tangled in the same expressions.
3. Modify each place consistently, without breaking the retail path or the promo code.

4. Verify the change — which, without fast tests, means clicking through a staging site and hoping the sandbox payment API is up.
5. Deploy, and watch the logs nervously.

The actual business rule is one line. The cost is everything around it: finding, understanding, coordinating, verifying, and releasing. That surrounding cost is what design controls. Good design does not make the rule itself simpler; it makes the *path to the rule* short and safe.

A useful way to think about the cost of change is to split it into three parts:

- **Discovery cost** — how long it takes to find where the change belongs and to understand the code there.
- **Ripple cost** — how many other places must change as a consequence, and how many people must coordinate.
- **Verification cost** — how long it takes to become confident the change is correct and has not broken anything else.

Clear naming and small functions (Chapter 2) reduce discovery cost. Low coupling and high cohesion (this chapter and Chapter 3) reduce ripple cost. Testable design and dependency injection (Chapter 4) reduce verification cost. Architecture (Part IV) organizes all three at the scale of a whole application.

Coupling: How Much Parts Depend on Each Other

Coupling is the degree to which one part of a system depends on another. If changing module A forces you to change module B, they are coupled. Some coupling is unavoidable — a program whose parts never depend on each other would do nothing useful. The goal is not zero coupling but coupling that is *deliberate, narrow, and pointed in the right direction*.

Coupling comes in several flavors. Recognizing them helps you see problems in code review before they become expensive.

Kind of coupling	What it looks like	Brewline example
Content	Reaching into another module's internals	Reports reading the orders table columns directly
Shared mutable state	Many modules read and write the same global	A module-level `db` pool and config object
Control	Passing flags that tell a callee what to do	`sendEmail(order, true, false)`
Data	Passing only the values needed	`shippingFor(country, subtotal)`
Temporal	Things must happen in a hidden order	Call `initPaybrook()` before charging
Implementation	Depending on a concrete class or vendor API	Business code calling the Paybrook URL

Roughly speaking, the kinds near the top of the table are the most harmful and the kinds like data coupling are the most benign. The `placeOrder` method above exhibits almost all of them. It depends on the exact columns of three tables, on a global connection pool, on an SMTP host, on the Paybrook HTTP contract, and on environment variable names. Any change to any of those things is a reason to open this file.

Coupling to volatile things is what hurts

Not all dependencies are equally dangerous. Depending on the TypeScript standard library, on `Array.prototype.map`, or on a stable internal `Money` type is almost free, because those things rarely change. Depending on a vendor API, a database schema, or a UI framework is expensive, because those things change for reasons outside your business logic. A key idea that later chapters develop into the Dependency Inversion Principle (Chapter 3) and the Dependency Rule (Chapter 16) is simple: **let volatile details depend on stable policy, not the other way around.** The rule "wholesale customers get 20% off" is Brewline's policy. Whether prices live in PostgreSQL or a spreadsheet is a detail.

Hidden coupling

The worst coupling is the kind you cannot see in the code. If the subscription job and the checkout flow both compute shipping using copy-pasted logic, they are coupled — changing one without the other produces inconsistent charges — but no import statement reveals the link. Duplicated business rules, shared database tables written by multiple modules, and implicit ordering assumptions are all forms of hidden coupling. Making coupling *explicit*, even when you cannot remove it, is already a big improvement: an explicit dependency shows up in imports, in type errors, and in search results.

> **TIP**
>
> A quick way to estimate coupling in an unfamiliar codebase is to look at version history. Files that are almost always changed together in the same commits are coupled, whether or not they import each other. Many version-control tools and code-analysis products can report this "change coupling"; even a simple script over `git log --name-only` will reveal the pairs that move together.

Cohesion: How Well a Part Holds Together

Cohesion is the flip side of coupling. It describes how closely the elements inside one module belong together. A highly cohesive module has one clear purpose, and everything in it serves that purpose. A module with low cohesion is a grab bag: a `utils.ts` file with date formatting, retry logic, price rounding, and a Slack webhook helper; or an `OrderService` that also sends emails and syncs accounting data.

Why does cohesion matter for maintainability? Because related things change together. If everything about shipping rates lives in one place, then a change to shipping rates is a change to one place. If shipping logic is scattered across the order service, the quote controller, and the subscription job, each shipping change becomes a scavenger hunt. High cohesion concentrates the effect of a change; low cohesion spreads it.

Cohesion and coupling are two views of the same design decision. When you group code that changes together, you simultaneously raise cohesion inside the group and lower coupling between groups. That is why the most practical design heuristic in this book can be stated in one sentence: **put things that change together in the same place, and keep things that change for different reasons apart.** Chapter 3 formalizes the second half of that sentence as the Single Responsibility Principle.

Levels of cohesion, informally

- **Coincidental** — things are together by accident (the `utils.ts` junk drawer).
- **Logical** — things are together because they are the same "kind" of thing (all validators in one file, all SQL in one folder), even if they serve unrelated features.
- **Temporal** — things are together because they run at the same time (an `onStartup` function that initializes ten unrelated subsystems).
- **Functional** — things are together because they jointly accomplish one well-defined task (a `ShippingRateCalculator` with its rules and data).

Functional cohesion is what we aim for. Notice that "logical" cohesion is surprisingly common in codebases that look tidy: a folder called `controllers/`, another called `services/`, another called `models/`. Each folder holds the same kind of thing, yet a single feature is split across all of them. We will discuss the trade-offs of packaging by layer versus by feature in Chapter 20.

A First Refactoring Glimpse

We are not going to fix Brewline in this chapter — that is the job of the whole book. But it helps to see where we are heading. Here is the same "place order" behavior, reshaped so that each part has one reason to change and the business rules no longer know about SQL, HTTP, or SMTP. Do not worry about every detail; later chapters introduce each piece properly.

src/ordering/application/place-order.ts (after, simplified)

```
import type { Money } from "../../shared/money.js";
import type { CustomerRepository } from "./ports/customer-repository.js";
import type { OrderRepository } from "./ports/order-repository.js";
import type { PaymentGateway } from "./ports/payment-gateway.js";
import type { Notifier } from "./ports/notifier.js";
import type { PricingPolicy } from "../domain/pricing-policy.js";
import type { ShippingPolicy } from "../domain/shipping-policy.js";
import { Order, type OrderLineInput } from "../domain/order.js";

export interface PlaceOrderInput {
  readonly customerId: string;
```

```
  readonly lines: readonly OrderLineInput[];
  readonly paymentToken: string;
  readonly promoCode?: string;
}

export class PlaceOrder {
  constructor(
    private readonly customers: CustomerRepository,
    private readonly orders: OrderRepository,
    private readonly pricing: PricingPolicy,
    private readonly shipping: ShippingPolicy,
    private readonly payments: PaymentGateway,
    private readonly notifier: Notifier,
  ) {}

  async execute(input: PlaceOrderInput): Promise<string> {
    const customer = await this.customers.getById(input.customerId);
    const order = Order.draft(customer, input.lines);

    const subtotal: Money = this.pricing.subtotalFor(order, customer, input.promoCode);
    const shippingCost: Money = this.shipping.costFor(customer.address, subtotal);
    order.place(subtotal.add(shippingCost));

    await this.payments.charge(order.total, input.paymentToken);
    order.markPaid();

    await this.orders.save(order);
    await this.notifier.orderConfirmed(customer, order);
    return order.id;
  }
}
```

The method is still one flow you can read top to bottom, but look at what it no longer contains. There are no table names, no URLs, no environment variables, no discount percentages, and no `customer.type` checks. Pricing rules live in a `PricingPolicy`, where adding a tier is a local change. Shipping rules live in a `ShippingPolicy`. The Paybrook specifics live behind a `PaymentGateway` interface, implemented by an adapter in the infrastructure layer. Emails hide behind a `Notifier`. And because every dependency is passed in through the constructor, a unit test can hand this class in-memory fakes and check the whole flow in milliseconds.

This is also more code, spread across more files. That is a genuine cost, and it is worth stating plainly. For a two-week prototype, the "before" version might be the better choice. The "after" version pays off when the rules keep changing, when several people work on the code at once, and when mistakes are expensive — which describes Brewline today.

Essential and Accidental Complexity

Some complexity in Brewline is unavoidable. Wholesale pricing really does differ from retail. Subscriptions really can be paused, resumed, and cancelled, and each state really does allow different actions. International shipping really does have different rules per destination. This is **essential**

complexity: it comes from the problem domain itself, and no design can remove it. The best a design can do is express it clearly.

Other complexity comes from our choices rather than from the problem. Having to remember to call `initPaybrook()` first, reading raw database rows with loosely typed fields, repeating the wholesale discount in five places, needing a real SMTP server to run a test — none of that is required by coffee retail. This is **accidental complexity**: complexity introduced by tools, structure, or history. It is the complexity we can and should attack.

The distinction, which goes back to classic writing on software engineering, is valuable because it tells you where to spend effort. When a piece of code feels hard, ask: is this hard because the business is genuinely complicated here, or because of how we built it? If it is essential, invest in clear modeling — good names, explicit states, value objects, and tests that document the rules. If it is accidental, invest in removing it — consolidating duplicates, hiding vendor details behind interfaces, and cutting unnecessary layers.

> **NOTE**
> Design patterns and architecture can themselves become accidental complexity. An abstract factory for a single implementation or a five-layer architecture for a CRUD form adds indirection without solving a real problem. Chapter 6 discusses this "pattern fever" in detail. A good rule of thumb: every abstraction should be able to name the change it makes cheaper.

Technical Debt as a Metaphor

The phrase **technical debt**, coined by Ward Cunningham, compares shortcuts in code to borrowing money. Taking a shortcut lets you ship sooner — you receive the "loan" immediately. But every future change in that area costs a little extra, like interest, until you pay back the principal by cleaning up the design. Unmanaged debt, however, compounds. Eventually most of the team's effort goes into paying interest, and new features slow to a crawl.

The metaphor is useful because it translates engineering concerns into terms that product managers and business owners already understand. "This change will take three days because the shipping logic is duplicated in three places" is a statement about interest payments. "If we spend two days consolidating shipping rules, future shipping changes will take hours instead of days" is a proposal to pay down principal.

It helps to distinguish a few kinds of debt:

- **Deliberate and prudent** — "We know this is rough; we will ship now and refactor next sprint." This is a real loan with a repayment plan.
- **Deliberate and reckless** — "We do not have time for tests." Usually this costs more time within weeks, not years.

- **Inadvertent** — "Now that we understand the domain better, we see the model was wrong." This is not a failure; it is learning. The code must evolve with understanding.
- **Bit rot** — the code was fine, but the world changed around it: new requirements, new libraries, new team members with different assumptions.

The Brewline `OrderService` contains all four. Some shortcuts were conscious bets that paid off. Some were rushed. Some reflect a time when there was only one customer type. What matters now is that the team can see the debt and pay it down in small, safe steps.

> **WARNING**
> Do not use "technical debt" as a label for any code you dislike. If a module is ugly but stable and rarely changed, it charges almost no interest. Refactoring it may feel satisfying but delivers little value. Prioritize debt where change is frequent and painful — the "hotspots" we will learn to identify in Chapter 26.

How Design Principles Map to These Forces

Everything else in this book can be read as a set of tools for managing coupling, cohesion, and complexity. Here is a map you can come back to as you read.

Tool	Main effect	Where
Clear names, small functions	Lower discovery cost	Chapter 2
SOLID principles	Raise cohesion, direct coupling	Chapter 3
Interfaces and dependency injection	Replace concrete coupling with contracts	Chapter 4
Refactoring in small steps	Pay down debt safely	Chapter 5
Design patterns	Reusable shapes for recurring change	Parts II and III
Clean and Hexagonal Architecture	Protect policy from details	Part IV
Domain-Driven Design	Model essential complexity well	Part V
Tests and seams	Lower verification cost	Part VI

Measuring Without Inventing Numbers

Teams often ask how to know whether their codebase is getting better or worse. There is no single metric for maintainability, and any number can be gamed. But several signals are genuinely useful when watched as trends rather than targets:

- **Lead time for small changes.** How long does a one-line business rule change take from ticket to production? If that grows over time, coupling or verification cost is rising.

- **Files touched per change.** If typical feature commits touch many unrelated files, cohesion is low.
- **Change coupling.** Pairs of files that always change together but live far apart suggest a missing abstraction or a misplaced boundary.
- **Test feedback time.** How long do you wait to learn whether a change broke something? Minutes are fine; hours change behavior.
- **Defects that escape to production.** Especially regressions in areas nobody meant to touch.

Static metrics such as cyclomatic complexity, function length, or dependency counts can support these signals, especially for spotting outliers. Just remember that metrics describe code; they do not tell you whether the code models the business well. Use them to decide where to look, not to decide what "good" is.

> **TIP**
>
> Before starting a refactoring effort, write down two or three concrete pain points in plain language — for example, "adding a customer tier requires edits in seven files" or "checkout tests take several minutes and fail randomly." After the work, check the same statements. This keeps the effort anchored to business value instead of taste.

A Small Coupling Exercise

Let us make coupling concrete with a tiny example. The pricing snippet below is lifted from the Brewline monolith. Try listing every reason it could need to change before reading on.

src/services/pricing.ts (before)

```
export function unitPrice(basePrice: number, customer: { type: string },
                          qty: number): number {
  let price = basePrice;
  if (customer.type === "wholesale") price *= 0.8;
  if (customer.type === "wholesale" && qty >= 10) price *= 0.95;
  if (customer.type === "staff") price *= 0.5;
  return price;
}
```

The reasons include: a new customer tier appears; wholesale discount levels change; volume breaks become tiered instead of a single threshold; staff pricing is limited to certain coffees; currencies are introduced; the business decides discounts should apply to the order total rather than to each unit. Each reason pulls this function in a different direction, and because the tier is a free-form `string`, the compiler cannot help when someone misspells `"wholesale"`.

A small step toward better design does not require any pattern yet. We can make the tier explicit and give each rule a name: