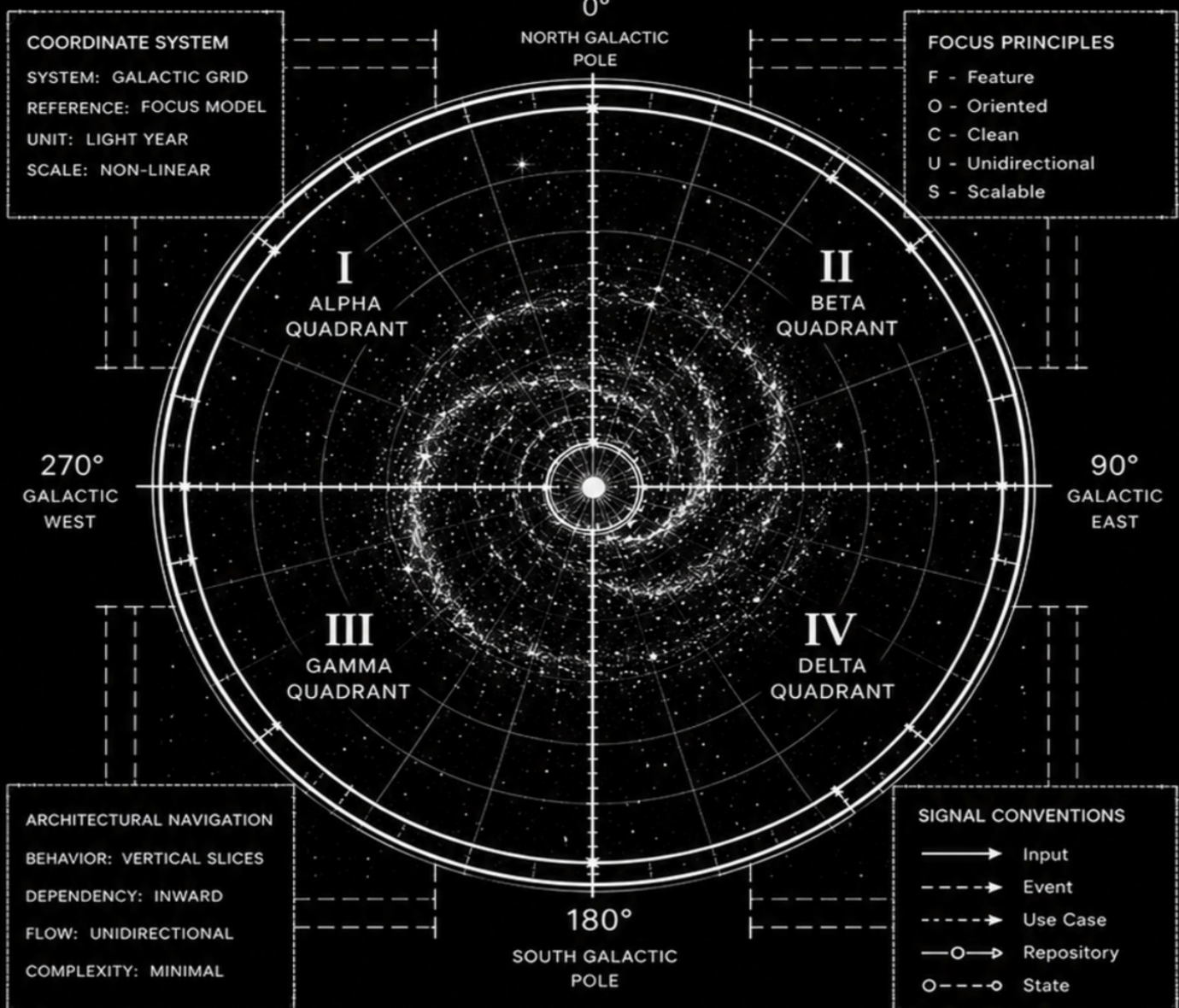


FOCUS

ARCHITECTURE

Feature-Oriented, Clean, Unidirectional & Scalable



J.C.Ködel



FOCUS: Architecture for People Who Ship Software

J.C. Ködel

FOCUS: Architecture for People Who Ship Software

1. [About the Author](#)
 1. [The F12 test](#)
 2. [The price of too many layers](#)
 3. [The price of too few layers](#)
 4. [Four pieces](#)
 5. [Why now](#)
 6. [Who looks for the rule now](#)
 7. [Rosie's Coffee Shop](#)
 8. [Quick reference](#)
2. [Map of the trilogy](#)
 1. [What each volume answers](#)
 2. [Where this book fits](#)
3. [The Day One Line Change Broke Three Screens](#)
 1. [The screen that started out reasonable](#)
 2. [Coupling](#)
 3. [Cohesion, the other side of the coin](#)
 4. [The axis of change](#)
 5. [Pitfalls](#)
 6. [Quick reference](#)
 7. [Exercises](#)

4. [AI Writes Fast. So What?](#)
 1. [The twelve-minute coupon](#)
 2. [The GitClear yardstick](#)
 3. [Why the generator fails this way](#)
 4. [Three guardrails against the same bug](#)
 5. [So is AI the problem?](#)
 6. [Pitfalls](#)
 7. [Quick reference](#)
 8. [Exercises](#)
5. [Simplicity Is a Decision: KISS and YAGNI](#)
 1. [The engine nobody asked for](#)
 2. [The four costs](#)
 3. [Where the acronyms came from](#)
 4. [The six lines the menu asks for](#)
 5. [What YAGNI doesn't cut](#)
 6. [And when the need finally arrives?](#)
 7. [Pitfalls](#)
 8. [Quick reference](#)
 9. [Exercises](#)
6. [DRY Isn't About Code](#)
 1. [Extraction by reflex](#)
 2. [What Hunt and Thomas actually wrote](#)
 3. [The inverse case: the card reader fee](#)
 4. [Timing tools](#)
 5. [The same knowledge outside the code](#)
 6. [The critique: DRY as a coupling factory](#)
 7. [Pitfalls](#)

8. [Quick reference](#)
9. [Exercises](#)
7. [SOLID Without Dogma](#)
 1. [One tab, four bosses](#)
 2. [Slice by actor, not by verb](#)
 3. [Read the code through the OCP and LSP lenses](#)
 4. [Narrow the contract and flip the arrow](#)
 5. [What each principle charges whoever is looking](#)
 6. [The cost of carrying what doesn't matter](#)
 7. [How many things you hold at once](#)
 8. [The critique SOLID earned](#)
 9. [Pitfalls](#)
 10. [Quick reference](#)
 11. [Exercises](#)
8. [Pure Functions and Immutability](#)
 1. [Two totals for the same tab](#)
 2. [Purity is what a function doesn't do](#)
 3. [Swap the call for the returned value](#)
 4. [Freeze the data: immutability across ten languages](#)
 5. [Functional core, imperative shell](#)
 6. [Pitfalls](#)
 7. [Quick reference](#)
 8. [Exercises](#)
9. [Errors Are Values](#)
 1. [The payment that only said "Something went wrong"](#)
 2. [Expected error is not a defect](#)

3. [The failure becomes part of the return type](#)
4. [Three steps, two rails](#)
5. [Go's counterpoint](#)
6. [Exceptions only at the boundary](#)
7. [Pitfalls](#)
8. [Quick reference](#)
9. [Exercises](#)
0. [Explicit Dependencies: DI and the Composition Root](#)
 1. [The payment that fetched its own dependencies](#)
 2. [Dependencies move up to the constructor](#)
 3. [If nobody calls the locator, who builds the graph?](#)
 4. [Pure DI before any container](#)
 5. [The Python counterpoint: discipline instead of syntax](#)
 6. [DI for the boundary, data for the rest](#)
 7. [Pitfalls](#)
 8. [Quick reference](#)
 9. [Exercises](#)
1. [FOCUS in One Page](#)
 1. [The handler that did everything](#)
 2. [The path of a tap](#)
 3. [The four pieces in the same gesture](#)
 4. [The same slice in your language](#)
 5. [Why just four](#)
 6. [The recipe the orchestrator follows](#)
 7. [Pitfalls](#)
 8. [Quick reference](#)
 9. [Exercises](#)

2. [Features, Not Layers](#)

1. [The change that touched four folders](#)
2. [The axis of change](#)
3. [The coffee shop in slices](#)
4. [Why it isn't four folders](#)
5. [The whole slice at once](#)
6. [What the imports give away](#)
7. [The same slice in ten languages](#)
8. [shared/ is born empty](#)
9. [Pitfalls](#)
10. [Quick reference](#)
11. [Exercises](#)

3. [The View: Dumb by Design](#)

1. [The screen that calculates](#)
2. [The state that arrives ready](#)
3. [The same screen, now dumb](#)
4. [Does this belong in the View?](#)
5. [The critique: bloated state](#)
6. [Pitfalls](#)
7. [Quick reference](#)
8. [Exercises](#)

4. [The Orchestrator: Event In, State Out](#)

1. [Where the one-way flow came from](#)
2. [The anti-solution: the orchestrator that decides](#)
3. [Events and states as sealed classes](#)
4. [The complete orchestrator](#)
5. [Go's counterpoint: no unions, no billing](#)

6. [The flow test: event on top, states below](#)
7. [Transient context and persistent context](#)
8. [The criticism: boilerplate and rules in the reducer](#)
9. [Pitfalls](#)
10. [Quick reference](#)
11. [Exercises](#)
5. [Use Cases: Where the Rules Live](#)
 1. [The anti-solution: the same rule in three places](#)
 2. [The rule as a pure function: the signature first](#)
 3. [The same rule, ten languages](#)
 4. [Orchestrator fetches, use case decides](#)
 5. [The use case as a retrieval unit](#)
 6. [Testing without a single test double](#)
 7. [The critique: “where’s the use case’s interface?”](#)
 8. [Pitfalls](#)
 9. [Quick reference](#)
 10. [Exercises](#)
6. [Repositories: The Exception Boundary](#)
 1. [The anti-solution: the copied catch in every screen](#)
 2. [The repository’s contract comes before its body](#)
 3. [The single translation: one catch, and only one](#)
 4. [The lookup feeding the orchestrator](#)
 5. [The same boundary in ten languages](#)
 6. [Where reading can stop](#)
 7. [Contracts that age slowly](#)
 8. [The critique: the generic repository and “the ORM already does this”](#)

9. [Preview of the fake: the interface you'll thank in chapter 17](#)
 10. [Pitfalls](#)
 11. [Quick reference](#)
 12. [Exercises](#)
7. [Commands and Queries: CQS Without Ceremony](#)
 1. [The anti-solution: paying and asking in the same gesture](#)
 2. [CQS: Meyer's rule](#)
 3. [The split, in all ten languages](#)
 4. [The canonical table's two tracks](#)
 5. [From CQS to CQRS, and where FOCUS stops](#)
 6. [Classify the eight operations](#)
 7. [Pitfalls](#)
 8. [Quick reference](#)
 9. [Exercises](#)
8. [Test Each Piece the Way It Asks to Be Tested](#)
 1. [The anti-solution: the suite that asserts the how](#)
 2. [Use case: a pure test, no double at all](#)
 3. [Repository: the fake you already have](#)
 4. [Orchestrator: flow test](#)
 5. [View and integration: where each one pays its own cost](#)
 6. [The two critiques](#)
 7. [Pitfalls](#)
 8. [Quick reference](#)
 9. [Exercises](#)
9. [What to Do When the Language Doesn't Help](#)
 1. [The slice on the board](#)

2. [Types first: four families and one warning](#)
3. [The three anchors, family by family](#)
4. [When the language doesn't help](#)
5. [The critiques, with a ruler instead of rhetoric](#)
6. [The equivalence table: porting to the 11th language](#)
7. [Pitfalls](#)
8. [Quick reference](#)
9. [Exercises](#)
0. [Anti-Patterns: How to Wreck FOCUS](#)
 1. [The shape of the card](#)
 2. [Card 1: business rule in the orchestrator](#)
 3. [Card 2: use case that hits the database](#)
 4. [Card 3: generic repository](#)
 5. [Card 4: layer by ceremony](#)
 6. [Card 5: domain try/catch](#)
 7. [Card 6: premature shared/](#)
 8. [What Go won't let you do](#)
 9. [Pitfalls](#)
 10. [Quick reference](#)
 11. [Exercises](#)
1. [Migrate Legacy Code Without Stopping the Factory](#)
 1. [The fig that strangles](#)
 2. [Choose the first slice: frequency times pain](#)
 3. [Step 1: fence the behavior with a characterization test](#)
 4. [Step 2: extract the rule into a pure use case](#)
 5. [Step 3: wrap the legacy in an adapter](#)
 6. [Step 4: wire the new view to the orchestrator](#)

7. [When NOT to migrate](#)
 8. [Both worlds on the same counter](#)
 9. [Pitfalls](#)
 10. [Quick reference](#)
 11. [Exercises](#)
2. [FOCUS + AI: The Duo That Scales](#)
 1. [The one-sentence request](#)
 2. [Each piece against a number](#)
 3. [The spec says what, the architecture says where](#)
 4. [Anatomy of the architectural prompt](#)
 5. [The same feature under the prompt](#)
 6. [Slice-guided review](#)
 7. [Two criticisms I take seriously](#)
 8. [Pitfalls](#)
 9. [Quick reference](#)
 10. [Exercises](#)
3. [Architecture for humans and for models](#)
 1. [The pattern nobody designed on purpose](#)
 2. [The name for this](#)
 3. [The question left over](#)
4. [Build Rosie's App](#)
 1. [The focus-coffee repository](#)
 2. [Menu: the read-only slice](#)
 3. [Tab: the slice with a lifecycle](#)
 4. [Payment: the slice of errors](#)
 5. [Loyalty: the rule, and the first shared/](#)

6. [Porting roadmap](#)
 7. [The critique: “book examples always work”](#)
 8. [Pitfalls](#)
 9. [Quick reference](#)
 10. [Exercises](#)
5. [Ship Increments Without Chaos](#)
 1. [FOCUS in a hallway conversation](#)
 2. [Who benefits \(and it’s a single reason\)](#)
 3. [The ruler against drift](#)
 4. [Quick reference](#)
 5. [Exercises](#)

About the Author

In this chapter, you'll:

- Recognize the cost of a project with too many layers
- Recognize the opposite cost: a project with no layers at all
- State this book's thesis in a single sentence

Have you ever opened a project and spent a week hunting for where a business rule lived? I have. This chapter tells the scars that led me to one conviction: building software shouldn't be hard, and four pieces are enough.

The F12 test

I have a favorite test for measuring a project's health. It's fast and it doesn't forgive. I open the IDE (Integrated Development Environment), rest the cursor on some call, and press F12. If I land straight on code that does something, the project passes. If I land on an interface that points to an abstraction that delegates to another abstraction, and ten jumps later I still haven't found a line that produces an effect in the real world, the project fails. And I already know how deep the trouble runs.

I learned this test the hard way. I consulted for a multinational insurance company, on a project that flew the DDD (Domain-Driven Design) flag and had read Eric Evans's book as a catalog of mandatory layers. The problem wasn't DDD, which was born to

bring code closer to the language of the business; it was the reading that turned every suggestion in it into law. In practice that was a stack of dozens of layers, where every implementation, however small, meant creating or changing several files. A new rule? Half a dozen files. The process was tedious and, worse, error-prone: the rule was spread across so much ceremony that nobody, not even whoever had written it, could see the whole thing at once.

That project was traumatic. It wasn't supposed to be complicated: customer records, policies, calculations, reports, a system like countless others, the kind any small, disciplined team would have shipped without drama. The complexity didn't come from the business. It came from the architecture choice. Code should be simple, direct, and to the point, and in decades of my career I rarely saw that. This book exists to make "rarely" less rare.

The price of too many layers

I've been programming professionally since 1995, and I've watched that insurance company's scene repeat in projects of every size: simple systems drowned in ceremony, boilerplate (the repeated ceremonial code you type the same way every time and that decides nothing), and verbosity, until productivity dies. Every layer is born from a promise: "this will give us flexibility." The promise almost never delivers. What every layer delivers for certain is cost: one more file to create, one more contract to maintain, one more place where someone will paste a business rule by mistake. When the core of the system swells, the architecture turns into bureaucracy. And bureaucracy, in code, gets paid for with every change, every day, for the rest of the project's life.

You've probably seen a project like this. Maybe you're stuck in one right now. The classic sign: the task looked like an hour of work and ate three days, because the small change crossed seven files and broke tests that had nothing to do with it. Nobody designed it that way out of malice. Layer by layer, each decision seemed sensible. The cost only shows up later, added up.

The price of too few layers

I met the opposite pain much earlier, inside my own code, in a system I had written alone that ran real customers' business. It was 1998, and the system was an ERP (Enterprise Resource Planning) built in Visual Basic 6. At first it was beautiful: no layers at all, the screen talked straight to the database, and every customer request turned into a feature the same day, sometimes with the customer still on the phone. Then the requests didn't stop. Every new feature made the system more fragile; any change spawned bugs in spots nobody had touched in months, until maintenance became impossible.

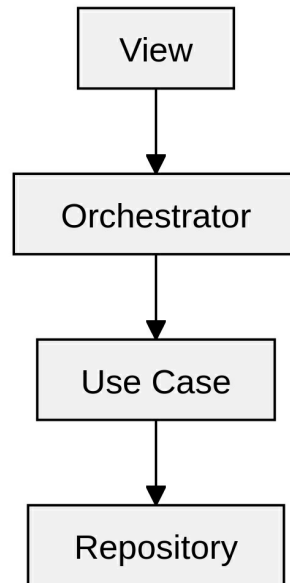
The 1998 ERP and the multinational insurer had opposite diagnoses and the same disease: the cost of change exploded. In one case, because the business rule was scattered across too many layers; in the other, because it was mixed in with screen and database, with no place to call its own. Keep that measure in mind. It's the thread running through this book.

Four pieces

The answer wasn't my invention, and it didn't fall from the sky in a flash of epiphany. It came from digging: decades gathering techniques from books, blogs, and people better than me, each one tested and proven by millions of developers around the

world. FOCUS is what survived that filter. Nothing here is new. The filter ran for real on one of my own products: Meu Cronograma Capilar (“My Hair Care Schedule”), a hair-care routine app I built in 2017, in Xamarin, kept deliberately simple. I rewrote it in Ionic, and JavaScript couldn’t keep up with my audience’s weak, outdated phones. I rewrote it again in Flutter. I was still learning the technology, and I leaned on what I already knew from Vue.js and MobX. The app wasn’t born with the architecture in place. It got distilled version after version: I cut what didn’t earn its own cost and reinforced what held changes together. Today the app carries more than 10 million downloads, a 4.8 rating on the Play Store, more than 300,000 active users, and 99.5% crash-free sessions, with sporadic updates, and I know exactly where everything lives. I never have to guess.

What survived the distillation were four pieces. I never had to question the two ends: a View shows things on screen and a Repository stores and fetches data; every system in the world has both. The middle was the only open question. Too much in the middle turns into the insurer’s bureaucracy. Too little turns into the 1998 ERP: repeated rules, an orchestrator calling another orchestrator, and changes that break ends nobody saw coming. The middle ground that survived every one of those versions was an Orchestrator that only translates events into state, and Use Cases that hold every business rule in functions you can test without booting a screen or a database.



Think of this diagram as the trailer for Part III. Each of these pieces gets its own chapters, with code and with the criticism it deserves.

Why now

I wrote *Spec Driven Development* (2026, <https://books.kodel.com.br/en/books/sdd>), where I argue that describing precisely what you want doesn't compete with AI, it multiplies what you get from it; you do not need to have read that book to follow this one. This book exists to pull one loose thread, a sentence I repeated more often than I liked: doing SDD (Spec Driven Development) without structure is vibe coding with extra ceremony. A model synthesizes code fast, and that's where structure decides the outcome: with nothing firm underneath, what comes out is the same old tangle, only quicker; GitClear's *AI Copilot Code Quality* reports (2024-2026) already measure that damage, and the numbers wait for chapter 3. With the right

structure, the opposite happens: the model recovers the context that matters before synthesizing, the new code doesn't break its neighbor, and whoever reads it later understands what was done.

Who looks for the rule now

In 1998, the only reader of my ERP was me. I'd open Visual Basic with last week's work still fresh in my head, and the whole project fit in there. No living project has a single reader today. The code of one single day passes through the hands of whoever joined the team last month, whoever reviews the pull request late in the afternoon, and a language model handed the task with nothing but what was open in the editor.

The model works in three movements: it recovers the context it can see, infers the intent from it, and synthesizes code that fits there. Getting the first movement wrong ruins the other two, and the first one depends entirely on how the project is organized. It's the same dependency as the person who joined last month, only measured in seconds instead of weeks.

Add the two up and you reach the arithmetic that changed. Implementing a rule got cheap: describing Rosie's loyalty discount and getting the function back takes less time than opening the right file. Locating where that discount lives still costs what it always cost. When one side of the arithmetic collapses and the other doesn't, finding becomes the expensive part, not implementing.

Hence the thesis of this book, in the single sentence it fits into: architecture lowers the cost of change because it makes the intent of the system recoverable, navigable and predictable for

humans and for models. The four pieces from the previous section are the means. That sentence is the end, and the rest of the book is the tally of what each piece charges to deliver it.

Rosie's Coffee Shop

Every architecture book trips on the same spot: each chapter invents a new domain, and you spend more energy understanding the example than the concept. Not here. This entire book uses a single domain: the ordering app for Rosie's Coffee Shop, with a menu, tabs, inventory, payment, and a loyalty program. Rosie doesn't exist; she's a character in this book. I picked this domain on purpose, because a coffee shop is a business everyone understands and that leaves no room to overcomplicate: if the architecture looks heavy for Rosie's Coffee Shop, it's heavy for real.

Quick reference

Situation	Fix
Evaluating an unfamiliar project	F12 test: count the jumps to the code
A simple task touches half a dozen files	Too many layers: cut
A change breaks code nobody touched	Too few layers: give the rule an address
Choosing between two architectures	Measure the cost of change for each

Looking for a business rule	It lives in the Use Case; the View just renders
-----------------------------	---

Tip 1

Architecture is measured in cost of change, not number of layers.

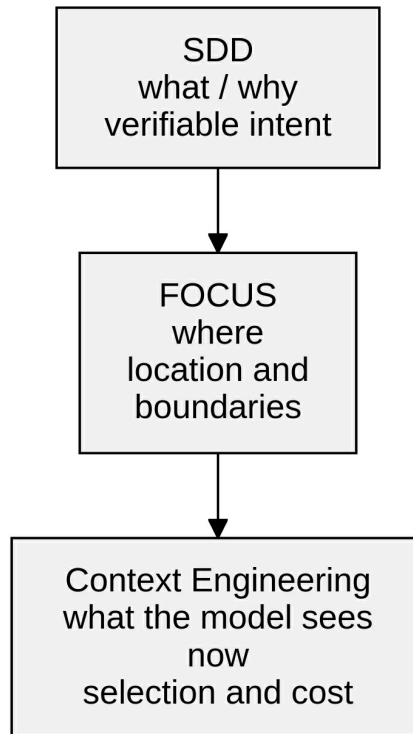
Next chapter: the day changing one line broke three screens, and what that accident teaches about where business rules should live.

Map of the trilogy

This is the second book in a trilogy, and you don't need to have read the first one: each volume stands on its own. This interlude exists so you know what lives in each of them when a bridge shows up in the middle of a chapter, and so you can ignore it with a clear conscience.

What each volume answers

Spec Driven Development (2026, <https://books.kodel.com.br/en/books/sdd>) answers what and why: it's about describing precisely what you want before asking for the code, so the work has a target to be checked against; you don't need it to follow this book. *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus>) is the one in your hands, and it answers where: how to organize code into slices with declared boundaries, so every change has an address. *Context Engineering* (2026, <https://books.kodel.com.br/en/books/context-engineering>) answers the question left over once the other two are standing: what the model sees right now, in this call's window, and at what cost; it also reads on its own, and this book doesn't depend on it on any page.



The order of the arrows is the order of the information, not a required reading order: the spec says what to do, the architecture says where what it asked for will live, and context carries both, in the right dose, to the model's window. You can come in through any door.

Where this book fits

The middle volume is the one that talks about code on disk. The discussion here is the folder, the file, the function's signature, and what each of those choices charges whoever has to find a rule months later. A reader who has never heard of an executable specification can apply everything that follows; a reader who already uses the first volume's flow will recognize the bridges and pick up two or three sentences of context when they appear.

That's this book's commitment to the other two ends: when one of them gets cited, the citation comes by name, with the address, and with whatever is needed summarized on the spot, precisely so you never have to interrupt your reading. No page here assumes you own the other two books.

Map in hand, on to the problem. Chapter 2 opens on a Monday when changing one line broke three screens.

The Day One Line Change Broke Three Screens

In this chapter, you'll:

- define coupling and cohesion in your own words;
- spot, inside forty-odd lines, the three reasons for change tangled together in them;
- predict which parts of a system break when a business rule changes.

Rosie is about to ask for the smallest change in the world: 15% off on rainy days. You're going to make the right change in the wrong place, and three screens you never opened are going to break. This chapter exists so you can name what broke and, next time, predict the break before you touch the code.

At the end of chapter 1 I promised an accident. Here it is. A drizzly Thursday, business is slow, and Rosie looks out the window of her coffee shop with the expression of someone who just had an idea that's going to cost somebody money. "When it rains, nobody comes in. Put a 15% discount on rainy days in the app, should be quick." She's right that it's quick: the rule fits on one line. The problem isn't the line. The problem is where lines just like it ended up.

The screen that started out reasonable

Rosie's Coffee Shop app has a menu screen. It wasn't born tangled: it grew out of three reasonable pull requests, the requests to merge a set of changes into the main codebase, each reviewed before it landed. In the first, the screen just listed items and prices. In the second, the loyalty program arrived, and the fastest way to give 10% off to anyone with ten past orders was to calculate it right there, where the price gets displayed. In the third, the team needed to log every order a customer tapped, and the fastest way was to write straight to the database, in the same file. Each step was defensible. Here's the result:

Dart

```
import "package:flutter/material.dart";

import "database.dart";

class MenuScreen extends StatefulWidget {

  const MenuScreen({super.key});

  @override

  State<MenuScreen> createState() => _MenuScreenState();
```

```
}
```

```
class _MenuScreenState extends State<MenuScreen> {
```

```
  final database = Database();
```

```
  final customerOrderCount = 12;
```

```
  final items = const [
```

```
    ("House coffee", 8.0),
```

```
    ("Cappuccino", 11.95),
```

```
    ("Cheese bread", 6.0),
```

```
  ];
```

```
  // calculate discount
```

```
  double priceWithDiscount(double price, int customerOrderCount) {
```

```
    var discount = 0.0;
```

```
    if (customerOrderCount >= 10) {  
  
        discount = 0.10;  
  
    }  
  
    return price * (1 - discount);  
  
}  
  
@override  
  
Widget build(BuildContext context) {  
  
    return ListView(  
  
        children: [  
  
            for (final (name, price) in items)  
  
                ListTile(  
  
                    title: Text(name),  
  
                    // format  
  
                    subtitle: Text(  

```

```

        "\${priceWithDiscount(price, customerOrderCount).toStringAsFixed(2)}",

    ),

    onTap: () {

        final value = priceWithDiscount(price, customerOrderCount);

        // write

        database.insert("orders", {"item": name, "value": value});

    },

),

],

);

}

}

```

Read the screen through its three comments, because they mark three different jobs disguised as one. `// calculate discount` is business logic: Rosie's loyalty policy, ten orders or more earn 10%, written as a screen method. `// format` is presentation: the price becomes text with a “\$” in front and two decimal places, the

way the designer asked for it. `// write` is persistence: tapping the item becomes a row in the `orders` table, with the value already calculated. If Python is the only language you know, don't get stuck on the Flutter syntax; keep the three labels in mind, because you're about to meet the exact same three jobs again in an eighteen-line Flask route.

Forty-four lines, none of them dumb. And yet this screen already costs money, and to measure that you need the metric that Tip 1 in chapter 1 announced without defining. **Cost of change** is the total effort needed to make a new decision hold true across the whole system: how many files you need to touch, how many spots you need to check, and how many breaks you need to fix before the system tells one consistent story. Good architecture is the kind that keeps this number small for the changes the business actually asks for. I don't know of a more direct measure than that. Layers, patterns and diagrams are means; the bill that arrives at the end of the month is the cost of change.

Let's pay that bill now, with Rosie's request. The new rule fits on one line inside `priceWithDiscount`. Except I already made this change in this code, counted the spots it touched, and the exact number is **four screen files**:

- `menu_screen.dart` : the screen you just read, where each item's price shows up;
- `tab_screen.dart` : the screen that totals a table's consumption, with the discount applied item by item;
- `payment_screen.dart` : the screen that closes the tab and charges the customer the final amount;
- `report_screen.dart` : Rosie's monthly report, which sums revenue with the discounts already deducted.

Each of these four files redoes the loyalty calculation on its own, and you'll see the other three copies a few pages from now, with the differences each copy picked up along the way. Worse: after you edit the menu, nothing warns you about the other three. The app compiles. The screen's tests pass. The tab, the payment and the report simply keep charging the old price, each in its own way, until someone notices the mismatch at the register.

Try it: open <https://focus.kodel.com.br/en/dart/02-01> and add the rainy-day discount yourself, right inside `priceWithDiscount`, on the menu tab. The whole app, all four screens, runs in the browser; if you prefer TypeScript, the same coffee shop is at <https://focus.kodel.com.br/en/ts/02-01>. Predicted result: the menu shows the new price right away, and the other three tabs don't change. If nothing breaks on screen, you didn't do anything wrong; that calm is exactly the problem the rest of this chapter dissects.

Where do the other three files come from? From the natural history of every calculation that lives inside a screen: when the tab screen needed to sum orders with loyalty applied, the method was sitting right there, private, locked inside the menu's `State`, and copying it was the path of least resistance. Here's the original source:

 **Dart** ·  **TypeScript**

```
// calculate discount

double priceWithDiscount(double price, int customerOrderCount) {

    var discount = 0.0;
```

```
if (customerOrderCount >= 10) {  
  
    discount = 0.10;  
  
}  
  
return price * (1 - discount);  
  
}
```

TS In TypeScript, only the signature changes: `function priceWithDiscount(price: number, customerOrderCount: number): number`. The body is identical, token for token, and that's why the two symbols share a single listing: copying this calculation is cheap in any language, and cheap is the danger.

And here are the three copies, one per screen. Notice that none of them matches the source, and none of them matches each other:

On the tab screen:

Dart

```
double tabTotal(List<double> prices, int orderCount) {  
  
    var total = 0.0;  
  
    for (final price in prices) {
```

```
total += price - price * (orderCount >= 10 ? 0.1 : 0);

}

return total;

}
```

On the payment screen:



```
double amountDue(double total, int customerOrderCount) {

    var factor = 1.0;

    if (customerOrderCount >= 10) {

        factor = 0.9;

    }

    return (total * factor * 100).roundToDouble() / 100;

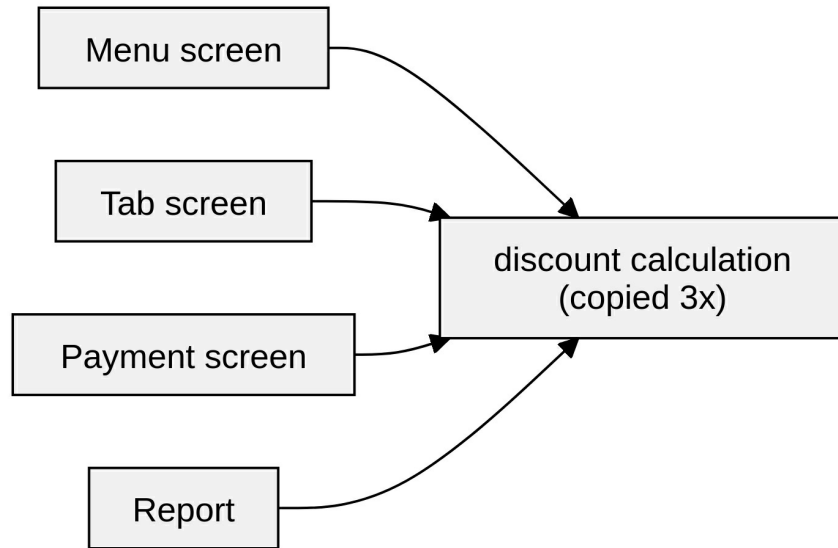
}
```


On the report:



```
double revenueWithDiscount(double gross, int monthlyOrderCount) {  
  
    final disc = monthlyOrderCount < 10 ? 0.0 : 0.10;  
  
    return gross - gross * disc;  
  
}
```

The tab screen turned the `if` into a ternary and changed the order of the math. The payment screen flipped the logic into a multiplying factor and rounded to cents, something no one else does. The report negated the condition and renamed the parameter to `monthlyOrderCount`, which doesn't even describe the same thing anymore. A copy never sits still: each screen pulled the calculation an inch toward its own side, and today the four versions agree by luck, not by design. That's why the break is silent. There isn't one place where the loyalty rule *lives*; there are four places where it *got pasted*.



The diagram is the map of the accident: four screens hanging off the same rule, and the rule with no fixed address. You felt the pain. Now let's name it, because pain with a name is a diagnosis.

Coupling

Coupling (from the Latin *copulare*, to join) is the degree to which one part of a system needs to change when another part changes. The word describes a chain, not a defect: coupled means it moves together. So far, no crime. In the coffee shop's code, the four screens are coupled to the loyalty rule, and the diagram shows the whole chain: pull the node in the middle and the four nodes above it move. Except the chain is invisible to the compiler, because the link isn't a function call, it's a resemblance between pasted text. Change `priceWithDiscount` and the compiler doesn't pull `tabTotal` along with it; the customer who got overcharged does.

Robert C. Martin, in *Design Principles and Design Patterns* (2000), named the two symptoms you just saw. **Rigidity**: a simple change forces a cascade of changes in modules that depend on it; the rainy-day discount was one line and became four files.

Fragility: a change breaks places with no apparent conceptual

relationship to it; whoever edits the menu has no reason to suspect the monthly report. Martin diagnosed this in enterprise systems twenty-six years ago. His code was different. The chain was this one.

I don't trust the eye to spot coupling, not even mine. After thirty years at this, my heuristic is still mechanical: pick a change the business would genuinely ask for, and count, in the code, how many files it touches. A number is a fact. "This screen is badly coupled" is an opinion people argue about in meetings; "this one-line change touches four files" ends the argument.

Cohesion, the other side of the coin

If coupling measures what changes *together across* parts, **cohesion** measures how much the things *inside* one part belong to each other. A cohesive screen contains only what shares the same fate; a low-cohesion screen is a house of tenants who don't know each other. The menu screen is the second case, and its three comments are the proof: loyalty policy, price formatting and database writes share one file without sharing a single reason to live there together.

That pair moves like a seesaw. When a screen's tenants don't belong to each other, some other part of the system needs them; the discount rule trapped inside the menu forced the tab screen to copy it, and every copy is a new link in the coupling chain. Low cohesion here manufactures coupling there. It isn't a coincidence, it's mechanics.

And it isn't a Flutter disease. The same screen, written as a Flask route by someone who came from the world of scripts, has the same three tenants in eighteen lines:

 Python

```
import sqlite3

from flask import Flask

app = Flask(__name__)

@app.route("/menu/<int:customer_order_count>")

def menu(customer_order_count):

    price = 11.95

    # calculate discount

    discount = 0.10 if customer_order_count >= 10 else 0.0

    value = price * (1 - discount)

    # format

    text = f"${value:.2f}"
```

```
# write

con = sqlite3.connect("orders.db")

con.execute("CREATE TABLE IF NOT EXISTS orders (item TEXT, value REAL)")

con.execute("INSERT INTO orders VALUES (?, ?)", ("Cappuccino", value))

con.commit()

con.close()


return text
```

Eighteen lines, the same three labels, the same future. The day the second endpoint needs the loyalty discount, someone is going to copy the `if` from this route, and the chain in the diagram starts growing in Python too. The language changes, the framework changes; the seesaw between cohesion and coupling doesn't.

The axis of change

There's one question missing that organizes all of this: who asks for each change? A piece of code's **axis of change** is the actor that triggers its edits, the person or role the requests come from. A healthy piece of code has a single reason to change because it answers to a single actor. The idea of reading code through the lens of who asks for it appears in Jimmy Bogard, "Vertical Slice

Architecture” (2018); here it only enters as a diagnostic lens, what to do with that lens is left for Part II. Point the lens at the menu screen and its three tenants get a face:

Snippet (label)	Who asks for the change	Example request
calculate discount	Rosie, the business owner	“15% off on rainy days”
format	The app’s designer	“show cents even when they’re zero”
write	The DBA / data owner	“new column on the table”

The third actor is the DBA (Database Administrator), the person who owns the shape of the tables. Three actors, three agendas, three different change calendars, all holding a key to the same file. When Rosie asks for the rainy-day discount, the risk doesn’t stay confined to her snippet: the change happens inches away from the formatting and the writing, inside the same `State`, and any slip splashes onto code that belongs to another actor. Now turn the lens on the three copies and the diagnosis closes: the loyalty rule has a single axis (Rosie), but its code is scattered across four files owned by other people. One actor, four addresses. That’s the full anatomy of the drizzle accident, and it’s everything this chapter promises: the exact name of the problem. The fix has a whole part of the book reserved for it.

Pitfalls

“I’ll just get rid of coupling.” Zero coupling doesn’t exist; the goal is to couple along the axis of change. A system whose parts don’t depend on anything doesn’t do anything. The tab screen

needs the discount calculation; the defect was never the dependency, it was copying as a way of depending.

“I already know the fix, just extract the calculation.” If you’re a senior developer, your hand has been itching since the first copy. Hold it back. Extracting now, without the criteria from Part II, tends to just relocate the coupling: the function lands in a `utils.dart` that three features fight over tomorrow, and the chain in the diagram is still there, with a different name on the middle node. The diagnosis came first in this book precisely because rushing to fix it is the most expensive trap.

“Nobody writes code like this.” Reread the origin story: three reasonable pull requests, approved one at a time. Nobody *decides* to write the tangled screen; it’s the natural state of any screen that takes reasonable shortcuts for six months straight. If your own repository doesn’t have one, look harder.

Q&A

Isn’t high cohesion just the same as a small class? No. Size is a symptom, not a criterion. An eight-line class that mixes business logic and formatting is less cohesive than a sixty-line one where everything serves the same actor. Measure belonging (do these snippets change for the same reason?), never line count.

Wouldn’t a stricter code review have caught the copies? It would help in the individual case and fail the pattern. Human reviewers get tired, lose context, and approve the fourth copy at 6pm on a Friday. A structure that makes copying unnecessary beats discipline that tries to forbid it; which structure that is, Part II answers.

Doesn't my framework already solve this? No framework decides where your business rules live; that decision is yours in Flutter, in React and in Flask, and the Flutter screen and the Flask route in this chapter show the same tangle in two unrelated ecosystems. The framework changes the frame. The picture is still yours.

Quick tip

Before you change a rule, search the whole repository for its constant. Search for the variants, not just one spelling: the four copies in this chapter write the same 10% as `0.10`, `0.1` and `0.9`, and a single `git grep -n "0.10"` only finds one of them. Prefer `git grep -nE "0\.10?|0\.9"` (or whatever magic number your rule uses, in every form it can take). Every hit is a potential spot you need to touch, and the ready-made list becomes your change checklist. Thirty seconds of grep save you an afternoon spent hunting the copy you forgot, after the register comes up short.

Quick reference

Situation	Fix
Measuring whether the architecture is good	Count the files a real change touches
One line turns into several files	Rigidity: map the chain before you edit
Changed here, broke over there	Fragility: hunt down the diverging copies

A file mixes jobs that don't belong together	Low cohesion: label each snippet
Not sure who owns a snippet	The actor who requests changes is its axis

Exercises

1. Rosie changed her mind: loyalty now kicks in at **five** orders, not ten. Before you open the editor, write down how many files you're going to touch and which ones. Then make the change across all four screens and check: if your list matched this chapter's count, you can already predict breakage better than whoever wrote the original screen.
2. The designer asked for prices to always show two decimal places, even for round numbers ("-\$7.00", not "\$7"). Could you predict the spots this change touches before you go looking in the code? Use the axis-of-change lens: the actor is different this time, and the answer isn't the same as in exercise 1.

Tip 2

Ask who's requesting the change before you ask where to put the code.

Next chapter: enter the world's fastest intern, the model that synthesizes code from whatever context it manages to retrieve, and you'll see what happens when it runs into a screen like the menu: the tangle a human takes six months to build up, it hands you today.

AI Writes Fast. So What?

In this chapter, you'll:

- quote from memory the three GitClear numbers that measure code degradation in the AI era: duplication +81%, error masking +47%, and refactoring dropping from ~25% to under 10% of changed lines;
- define *vibe coding* with author and year, and separate it from “using AI”;
- point out, in Rosie’s coupon bug, where each of FOCUS’s three guardrails would have caught the defect.

You’re going to ask an AI for a feature and get it back, done, in twelve minutes. You’ll test it on screen, watch it work, and ship before lunch. Three weeks later Rosie’s register will charge the wrong amount, and no log will explain why. This chapter shows where the defect hid, how much of it is already measured at scale, and which three structures would have stopped it at the door.

Chapter 2 ended with a promise: the fastest intern in the world was about to find the menu screen. She did. That tangle of business rule, formatting, and persistence crammed into one file cost a human team six months of shortcuts; I’ll guess, with the bluntness of someone exaggerating on purpose, that an AI delivers the same tangle 40 times faster. The 40 is my hyperbole,

not a measurement. The question it carries is serious: what happens to the cost of change when tangled code stops taking months to exist and starts existing in minutes?

The twelve-minute coupon

Wednesday, late afternoon. Rosie saw the competitor's coffee shop on her phone and showed up with the request ready: "I want a discount coupon button, the kind where you type WELCOME10 and get 10% off." You already have four tasks in the queue. So you paste the request into an AI, along with the order-flow file, and twelve minutes later there's a new handler, `applyCoupon`, with code validation, total calculation, and even a friendly message for an expired coupon. You type WELCOME10, the price drops 10%, and Rosie applauds from the counter. Deploy done, queue resumed.

Three weeks later, the customer at table four complains. She has seven orders in the loyalty program and used the new campaign's coupon; the app charged only the coupon discount, without adding the loyalty discount. The cashier checks, and the complaint holds up. You open the day's log: nothing. No exception, no warning, no line out of place. The app recorded no defect because, as far as it was concerned, no defect happened.

The archaeology takes half an hour and turns up two discoveries. First: the handler that came out of the model never calls the loyalty calculation that already existed in the order flow; it recomputes the total from scratch and carries its own copy of the rule, rewritten from whatever the model saw in the file. Two handlers now hold the same business rule, each on its own, and the new copy was born out of date: it uses the old threshold of ten orders, not the five Rosie adopted months earlier. With seven orders, the customer clears today's threshold and misses the one

the copy kept. This has a name. Knowledge duplication is the same business decision written in two places that don't know about each other; when the decision changes, someone has to remember every address, and chapter 2 already showed how that recall fails. The second discovery is worse. At the end of the new handler sat a consistency check that compared the coupon total against the order-flow total and threw an exception on any mismatch. The AI wrapped that check in an empty try/catch, commented "avoids blocking checkout." The alarm existed. Someone switched it off during installation. Error masking is exactly this: code that catches or swallows a failure without handling it, and hands the user the appearance of success in place of the problem. The coupon bug wasn't silent by accident; it was silenced by design.

The GitClear yardstick

One coffee shop case doesn't prove a trend. Numbers do, and someone counted them: GitClear, a company that analyzes code quality, examined hundreds of millions of real changes across repositories for its 2024, 2025, and 2026 reports. Hold on to three of those numbers; they're the spine of this entire book. Code duplication rose 81% relative to the pre-AI era (GitClear, 2024-2026). Error masking rose 47% (GitClear, 2026): the generator favors a silent catch, safe navigation, and stubs (facade implementations that return some value without doing the work) that hide defects over ones that handle them. And refactoring dropped from about 25% of changed lines in 2021 to under 10% in 2024 (GitClear, 2024): new code piles on top of new code, and almost nobody tidies up.

The rest of the numbers in the same set of reports back up those three. Duplicated blocks of five or more lines grew roughly 8x in 2024 (GitClear, 2024). The same report notes that 2024 was the

first year copy-paste outpaced moved code: more copying happened than reuse (GitClear, 2024). Cross-file calls, reuse between files, fell 35%, and legacy code maintenance dropped 74% (GitClear, 2024-2026). Outside GitClear the direction repeats: the arXiv 2409.19182 study (2024) and the 2024 DORA (DevOps Research and Assessment) research found delivery speed climbing while stability and maintainability fall when nothing constrains the generator. Notice the framing. The measured problem is never “used AI”; it’s what gets generated when nothing limits the shape of the output.

This way of generating has its own name. Vibe coding is the term Andrej Karpathy coined in February 2025 for the practice of accepting code from an LLM (Large Language Model) without reading it, guided only by the surface result: it ran, it looked fine on screen, move on. Notice the gap between that term and “using AI.” Someone who uses AI with review and a boundary stays in command of the code; someone who does vibe coding delegated the reading too. The coupon handler was classic vibe coding, and I was the first to do exactly the same: the code looked right, the screen worked, and twelve minutes is too tempting to resist.

Why the generator fails this way

An LLM doesn’t optimize for your system to last; it optimizes for the next answer to look correct to whoever reads it. Those are different goals. Inside a single file, “looking correct” and “being correct in the system” nearly line up, which is why the coupon handler worked so well in the demo. Across the whole system the two goals split apart: the loyalty rule that already existed sat outside the context the model could see, so recreating it inside the new handler was the path of least resistance. The same logic applies to failure. Handling an exception means deciding what the business wants in every bad case; swallowing the exception

makes today's demo pass. Without a boundary that forces the handling, the empty catch is the exit the generator has learned to prefer, and GitClear's +47% (2026) shows that preference at industrial scale.

The contrast fits in one listing. Rosie's inventory lookup, first as the AI delivered it, then as it looks once the failure becomes a return value; chapter 8 builds that Result piece by piece, so don't worry about the sealed syntax yet:

Dart

```
// what the AI delivered: looks like it works

Future<int> availableUnits(String item) async {

    var units = 0;

    try {

        units = await inventory.check(item);

    } catch (e) {}

    return units;

}
```

```
// the same lookup with Result: failure becomes a value the caller handles
```

```
sealed class InventoryQuery {}
```

```
class Available extends InventoryQuery {
```

```
    Available(this.units);
```

```
    final int units;
```

```
}
```

```
class InventoryDown extends InventoryQuery {}
```

```
Future<InventoryQuery> checkInventory(String item) async {
```

```
    try {
```

```
        return Available(await inventory.check(item));
```

```
    } on Exception {
```

```
        return InventoryDown();
```

```
}  
  
}
```

TypeScript

```
// what the AI delivered: looks like it works  
  
async function availableUnits(item: string): Promise<number> {  
  
    let units = 0;  
  
    try {  
  
        units = await inventory.check(item);  
  
    } catch (e) {}  
  
    return units;  
  
}  
  
// the same lookup with Result: failure becomes a value the caller handles  
  
type InventoryQuery =
```



```

    | { kind: "available"; units: number }

    | { kind: "inventoryDown" };

async function checkInventory(item: string): Promise<InventoryQuery> {

    try {

        return { kind: "available", units: await inventory.check(item) };

    } catch (e) {

        return { kind: "inventoryDown" };

    }

}

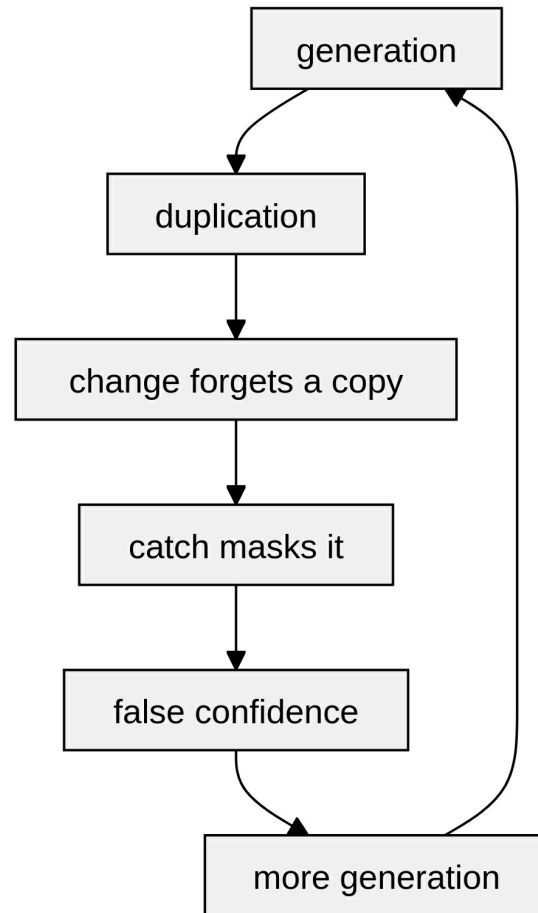
```

Read the two halves as two answers to the same question: “what happens when the inventory service goes down?” The top half answers with a polite lie. The offline inventory becomes 0, the menu shows “out of stock” for an item that exists, and no log reports anything; it’s the twin of the coupon’s empty catch. The bottom half changes the return type: whoever calls `checkInventory` gets `Available` or `InventoryDown` back and has to decide what the screen does in each case. The lie is no longer an option on the table.

Who enforces that decision depends on the language, and the difference matters. In Dart, a switch over a sealed class is exhaustive by construction: miss a case and the program doesn't compile. In TypeScript, exhaustiveness is optional; the compiler only complains if you ask it to, by assigning the unhandled case to a variable of type never in the default branch. Without that explicit request, an incomplete switch slips right through.

Try it: open <https://focus.kodel.com.br/en/dart/03-01> and run the file a few times; the example inventory fails at random. The naive version prints 0 as if the item had run out, and the Result version prints the inventory-down warning. Then delete the InventoryDown case from the switch and watch the compiler refuse the program. The same comparison in TypeScript is at <https://focus.kodel.com.br/en/ts/03-01>.

Put the chapter's pieces together and the coupon case stops looking like an isolated accident. It's one full turn of a cycle that feeds itself:



Every generation accepted without reading adds a copy; the next rule change forgets one of them; the catch masks the mismatch; the silence in the logs turns into confidence that everything is fine; that confidence authorizes generating even more. The cycle doesn't stop on its own. It stops when some structure breaks one of the links, and that's what the rest of this book is about.

Three guardrails against the same bug

Call the central idea a narrow search space: give the generator (and the reviewer) contracts so tight that only one correct way exists to complete the code. A model that can return anything will, sooner or later, return the wrong thing wearing the face of the right one; a model squeezed by types, tests, and a boundary

errs less and, when it does err, errs loud. FOCUS narrows that space with three guardrails, the same barriers that keep a car from leaving the road without taking the wheel out of your hands. Each one would have caught the coupon bug at a different point.

The first is the compiler armed with exhaustive types. If the coupon's consistency check returned a Result like the one in the listing, instead of throwing an exception, the empty try/catch wouldn't even be possible: the handler would be forced to declare, in visible code, what to do with a total mismatch. "Ignore the failure" would still exist as a decision, but a written, reviewable one, never an invisible omission. Chapter 8 builds this guardrail.

The second is the pure-function test. If the loyalty rule lived in a single function with no screen, no database, and no network nearby, it would have one address and a full-name test suite; the outdated copy inside the coupon handler would have nowhere to come from, and a threshold change from ten orders to five would break a test in the same second. Chapters 14 and 17 build this guardrail.

The third is the slice. With the coupon flow isolated in its own slice, with an explicit contract for talking to the rest of the system, the possible damage from a bad handler stays confined to the size of the slice: the blast radius of a twelve-minute generation becomes a directory, not the whole system. Chapter 11 builds this guardrail. Notice that none of the three demands a better AI or a superhuman reviewer; all three change the ground where any generator, human or not, is capable of erring.

So is AI the problem?

No, and I wouldn't have written this book if I thought so. I use AI every day (I used it to write code for this book too), and the speed gain from shipping that coupon is real; the same evidence that shows the degradation shows the gain. This book covers the code structure that makes generation fit inside a boundary. Steering that power with specifications is the subject of *Spec Driven Development* (2026, <https://books.kodel.com.br/en/books/sdd>), and you do not need to have read it to follow from here. The problem was never the fastest intern in the world. The problem is handing her a system where the business rule lives on four screens, no type forces failure handling, and any file can reach any other; on that ground, speed only amplifies the tangle from chapter 2. Structure first, generation second, and the pair scales; chapter 21 closes this argument with all of FOCUS on the table.

Pitfalls

“The tests pass, so it's correct.” Watch out for who wrote the tests. When the AI generates the code and the tests in the same breath, the test tends to lock in the generated behavior, not the business rule: the coupon handler's test asserted that a total mismatch returns the coupon total without complaint: the error masking had a test guaranteeing the lie stayed alive. A test is worth what it demands, not whether it passes; the rule for who demands what arrives in chapters 14 and 17.

“I review every diff myself, this won't happen to me.”

Rereading chapter 2 helps here: the human reviewer approves the fourth copy at 6pm on a Friday. AI multiplies diff volume by a factor no review discipline keeps up with; trusting your system's safety to a tired human's infinite attention is betting against the odds. Structure that makes the error impossible to compile doesn't get tired. “To err is human” is real: sooner or later, someone errs, always.

“So I’ll ban AI on the team.” A ban throws away the real speed gain and doesn’t remove the cause: the ground without a boundary is still there, and hurried humans produce the same tangle in slow motion, as three reasonable pull requests already proved in chapter 2. The right target is the ground, not the tool.

Q&A

Won’t these GitClear numbers age badly? They will, which is why each one carries its report year right next to it. What this chapter asks you to keep is the mechanism, ownerless copy plus swallowed failure, which stays explainable even after the percentages change.

Don’t newer models write better code and retire this whole discussion? They write better code, faster, and that cuts both ways: they also fail faster. GitClear’s 2026 report measured error masking rising in exactly the most capable generation to date. As long as the generator’s goal is to look correct to whoever reads it, the shape of the output stays the responsibility of whoever sets the boundary: you.

Is error masking an invention of the AI era? No; the empty catch has existed as long as exceptions have, written by people. The new part is scale: what used to be an occasional slip by a rushed developer became the statistical preference of a tool that writes a large share of the world’s new code, with a 47% rise measured by GitClear in 2026.

Quick tip

Before accepting any AI-generated diff, hunt it for masking patterns: `git diff | grep -nE "catch\s*(\s*(.|\s))*\s*\{\s*\s*$"` catches an empty-body catch on its first line, and it’s worth repeating the search for `?.` and suspicious default values like `?? 0`.

Better still: turn on your language's lint rule (`empty_catches` in Dart, `no-empty` in ESLint) as a CI (Continuous Integration) failure, the battery that runs on every push before code merges, and the most common masking dies before the merge, at zero cost in human attention.

Quick reference

Situation	What to do
Citing the numbers	+81% duplication, +47% error masking (GitClear 2024–2026)
Citing the refactoring drop	~25% to under 10% of lines (GitClear, 2024)
AI feature passed the demo	Hunt for the recreated rule and the silenced catch
Log too clean after a bug	Suspect error masking at the source
Team debates “AI, yes or no”	Debate the boundary instead: chs. 8, 14/17, and 11
Telling vibe coding from using AI	Accepting without reading (Karpathy, Feb 2025) is vibe

Exercises

1. Open <https://focus.kodel.com.br/en/dart/03-01> and add a third case to the sealed class: `ItemOutOfStock`, for when the

service reports zero units. Watch the order of events: the compiler flags the incomplete switch before you run anything at all. That's the feeling of working inside a narrow search space, and it's the one you'll build in chapter 8.

2. A coworker asked an AI for Rosie's coupon-application screen and got back the code at <https://focus.kodel.com.br/en/dart/03-02> (React version at <https://focus.kodel.com.br/en/ts/03-02>). The code compiles and the demo passes. Can you find the single spot of error masking hidden in it, name the information the user loses there, and propose what the function should return instead?

Tip 3

Never accept an error from an AI that the compiler wouldn't have caught.

Next chapter: before you learn to structure what you build, you'll learn the cheapest trick of the trade: deciding what not to build. With those scissors in hand, Part II begins: the foundations that turn the diagnosis of these three chapters into daily practice.

Simplicity Is a Decision: KISS and YAGNI

In this chapter, you'll:

- list Fowler's four costs of speculative functionality from memory and point to where each one shows up in real code;
- tell the origin of KISS and YAGNI with a name, a place, and a year;
- apply the “do I need this now?” test to a pull request and separate what YAGNI cuts from what YAGNI never cuts.

You open a file to add a price field and find a pricing engine with support for three currencies, five tax regions, and scheduled promotions, all written by someone who swore they were helping. Nobody asked for any of it, and you're still going to pay for every line. This chapter hands you the cheapest tool in the trade: the test for deciding what not to build.

The first three chapters made the diagnosis. Chapter 2 measured the cost of change: the price of touching one line is set by the tangle it crosses, not by the size of the edit. Chapter 3 showed AI multiplying the speed at which that tangle grows and laid out the guardrails that limit the damage. Part II starts here, and it starts with the scissors. Before you learn to structure what you build, you learn to refuse what doesn't need to exist, because the easiest line to maintain is still the one nobody wrote.

The engine nobody asked for

Rosie asked for one thing: the menu in the app needs to show the price of each item. Cappuccino at \$11.95, cheese bread at \$6.00, house coffee at \$8.00. She changes those numbers by hand, two or three times a year, whenever milk gets more expensive.

Rosie's Coffee Shop takes one currency, runs out of one address, and schedules exactly zero promotions.

The teammate who picked up the ticket handed back the file below. Read it slowly; the whole chapter's pain lives inside it.

Dart

```
// multi-currency support (the coffee shop has one)

enum Currency { usd, eur, brl }

const exchangeRates = <Currency, double>{

  Currency.usd: 1.0,

  Currency.eur: 1.09,

  Currency.brl: 0.19,

};

// tax by region (the coffee shop has one address)
```

```
enum Region { southeast, south, northeast, north, midwest }
```

```
class TaxRule {
```

```
    const TaxRule(this.region, this.rate);
```

```
    final Region region;
```

```
    final double rate;
```

```
}
```

```
const taxRules = <TaxRule>[
```

```
    TaxRule(Region.southeast, 0.12),
```

```
    TaxRule(Region.south, 0.11),
```

```
    TaxRule(Region.northeast, 0.09),
```

```
    TaxRule(Region.north, 0.08),
```

```
    TaxRule(Region.midwest, 0.10),
```

```
];
```

```
// scheduled promotions (Rosie changes prices by hand, whenever she wants)
```

```
class ScheduledPromotion {
```

```
    const ScheduledPromotion({
```

```
        required this.item,
```

```
        required this.discount,
```

```
        required this.start,
```

```
        required this.end,
```

```
    });
```

```
    final String item;
```

```
    final double discount;
```

```
    final DateTime start;
```

```
    final DateTime end;
```

```
    bool activeAt(DateTime instant) =>
```

```
        !instant.isBefore(start) && !instant.isAfter(end);
```

```
}
```

```
// base price per item, in dollars, before any adjustment
```

```
class BasePrice {
```

```
    const BasePrice(this.item, this.priceInDollars);
```

```
    final String item;
```

```
    final double priceInDollars;
```

```
}
```

```
class PricingEngine {
```

```
    PricingEngine({
```

```
        required this.basePrices,
```

```
        required this.defaultCurrency,
```

```
        required this.region,
```

```
        this.promotions = const [],
```

```
});

final List<BasePrice> basePrices;

final Currency defaultCurrency;

final Region region;

final List<ScheduledPromotion> promotions;

// looks up the item's base price

double _basePriceOf(String item) {

    for (final price in basePrices) {

        if (price.item == item) {

            return price.priceInDollars;

        }

    }

    throw ArgumentError("Item not on the menu: $item");
}
```

```
}

// applies the most aggressive scheduled promotion active at the instant

double _withPromotion(String item, double value, DateTime instant) {

    var bestDiscount = 0.0;

    for (final promotion in promotions) {

        if (promotion.item == item &&

            promotion.activeAt(instant) &&

            promotion.discount > bestDiscount) {

            bestDiscount = promotion.discount;

        }

    }

    return value * (1 - bestDiscount);

}
```

```
// adds the tax for the configured region

double _withTax(double value) {

    for (final rule in taxRules) {

        if (rule.region == region) {

            return value * (1 + rule.rate);

        }

    }

    return value;

}

// converts from the base currency (dollar) to the requested currency

double _inCurrency(double valueInDollars, Currency currency) =>

    valueInDollars / exchangeRates[currency]!;

// the method the menu calls to display a price
```



```

double priceOf(String item, {Currency? currency, DateTime? instant}) {

    final now = instant ?? DateTime.now();

    final base = _basePriceOf(item);

    final promotional = _withPromotion(item, base, now);

    final taxed = _withTax(promotional);

    return _inCurrency(taxed, now == instant ? currency! : defaultCurrency);

}

}

```

That's 95 lines to answer "how much is the cappuccino?" The code works, it compiles without a single warning, and every block carries a polite comment explaining its own sub-goal. That's exactly why it's dangerous: nothing in it looks wrong. The right question isn't "is this well written?" It's "who asked for it?" Nobody asked for currency conversion. Nobody asked for a regional tax rate. Nobody asked for a promotion calendar. Each of those three axes is **speculative functionality**: code written for a need nobody has today, a bet on a future imagined by whoever wrote it.

Ask the author and the defense comes pre-loaded: "what if Rosie opens a location in São Paulo? What if the next location is in a different region instead? What if she wants to run a winter

promotion?” None of those questions is absurd, and that’s exactly what makes speculation so seductive. Real coworkers write code like this, with good intentions and a future in mind. The problem isn’t how plausible the guess is. The problem is the price, and the next section measures that price in four installments.

Meanwhile, the task Rosie actually asked for, the price on the menu screen, got pushed two days further away: that’s how long the engine took to build.

The four costs

I’ve written this engine before. In my case it was a plugin system: a dynamic loader, an extension registry, contract versioning, all “for the future,” because the product would supposedly become a platform someday. The future arrived and asked for none of it. Years later I deleted the whole system myself, and no plugin beyond my own two examples had ever existed; the only lesson left standing is the subject of this chapter. It wasn’t an execution mistake; the code was good. It was a decision mistake.

Martin Fowler breaks that mistake into four costs, in the Yagni entry (You Ain’t Gonna Need It) of his *bliki*, the blog-wiki hybrid he’s kept since 2003, where each entry gets revised in place instead of turning into a new post (martinfowler.com/bliki/Yagni.html). Let’s measure each cost against the `PricingEngine` you just read.

The first is the **cost of build**: the hours spent analyzing, coding, and testing a feature nobody uses. On the pricing engine, that was two days of work for 95 lines, of which the menu exercises half a dozen. Everything else is effort paid for a hypothesis.

The second is the **cost of carry**: the tax that speculative functionality charges everyone who reads, edits, or debugs the code from then on, even without ever using it. It's the easiest cost to underestimate, because it never shows up on any invoice. Whoever opens the file to fix a price has to understand `Currency`, `Region`, `TaxRule`, and `ScheduledPromotion` before finding the line that matters. Add to that the extra surface for defects. Look at the end of `priceOf`: the expression `now == instant ? currency! : defaultCurrency` looks like the finesse of someone who handled every case, and it actually blows up at runtime if someone passes `instant` without passing `currency`. The bug lives in a parameter no real call ever uses. No multi-currency, no bug.

The third is the **cost of delay**: the value the requested feature failed to generate while the speculative one was being built. The price on the screen was worth money on Wednesday; it shipped on Friday. Fowler insists this is the decisive cost, because it delays exactly the thing somebody is waiting to use.

The fourth is the **cost of repair**: when the future finally shows up, it almost never has the shape the guess predicted, and the structure built ahead of time has to be twisted to fit. If Rosie opens that next location in a different region, her real tax bill won't be a single rate tied to a spot on the map; it'll be a mix of state, county, city, and product category that this five-line table can't represent. The engine didn't get any work done early. It created the work of tearing itself back down.

Keep the order in mind: build, carry, delay, repair. The reference table at the end of the chapter lists all four again, each one pointing back to the line of `PricingEngine` where you saw it.

Where the acronyms came from

KISS is the older of the two acronyms. “Keep It Simple, Stupid” is credited to Kelly Johnson, chief engineer at Lockheed’s Skunk Works in the 1960s, and the original context explains the meaning better than any definition could: Johnson’s airplanes had to be repairable by an average mechanic, in the field, with the tools that mechanic already had. Simple there wasn’t an aesthetic compliment. It was an operating requirement: a design that needs a genius to maintain has failed, no matter how gracefully it flies. The maxim carried that same sense into software, and that’s the sense this book uses.

YAGNI was born inside a project you can date exactly: Chrysler’s C3, the payroll system that served as the cradle of Extreme Programming in the late 1990s. Whenever someone argued for a hypothetical capability (“we’re going to need this when...”), Kent Beck gave back the same answer: “you aren’t gonna need it.” The answer turned into an acronym on the team, and the acronym turned into a published practice in *Extreme Programming Installed* (Ron Jeffries, Ann Anderson, and Chet Hendrickson, 2001). Jeffries’s own wording is the working definition this chapter applies:

“Always implement things when you actually need them, never when you just foresee that you need them.”

The word carrying the whole sentence is *foresee*. YAGNI doesn’t forbid building; it forbids building on a forecast. The only legitimate trigger is a present need, with the name of whoever asked for it attached.

The six lines the menu asks for

Apply Jeffries's sentence to the engine: what's left once you cut everything that exists on a forecast? This is what's left.

Dart

```
const pricesInCents = <String, int>{

  "cappuccino": 1195,

  "cheese bread": 600,

  "house coffee": 800,

};

int priceOf(String item) => pricesInCents[item]!;
```

Six lines of code. Every cut has a name. Cutting multi-currency erased `Currency`, `exchangeRates`, `_inCurrency`, and the optional-parameter bug from the previous section. Cutting the regional tax erased `Region`, `TaxRule`, the rate table, and `_withTax`. Cutting the promotion calendar erased `ScheduledPromotion`, `_withPromotion`, and the dependency on `DateTime.now()`, which had turned the price into a function of the clock. And one cut came free: the price became an integer in cents, because `double` only existed to accommodate exchange rates and tax rates, and floating-point money is a pain you don't need to buy today.

Try it: open <https://focus.kodel.com.br/en/dart/04-01> and run it. Then try adding back just the dollar-to-euro conversion, without touching tax or promotions. Count how many lines came back and how many of them today's menu actually calls. That answer is the cost of carry, measured by you.

The same solution in Go earns a cultural aside worth the detour.



Go

```
var pricesInCents = map[string]int{

    "cappuccino": 1195,

    "cheese bread": 600,

    "house coffee": 800,

}

func priceOf(item string) int {

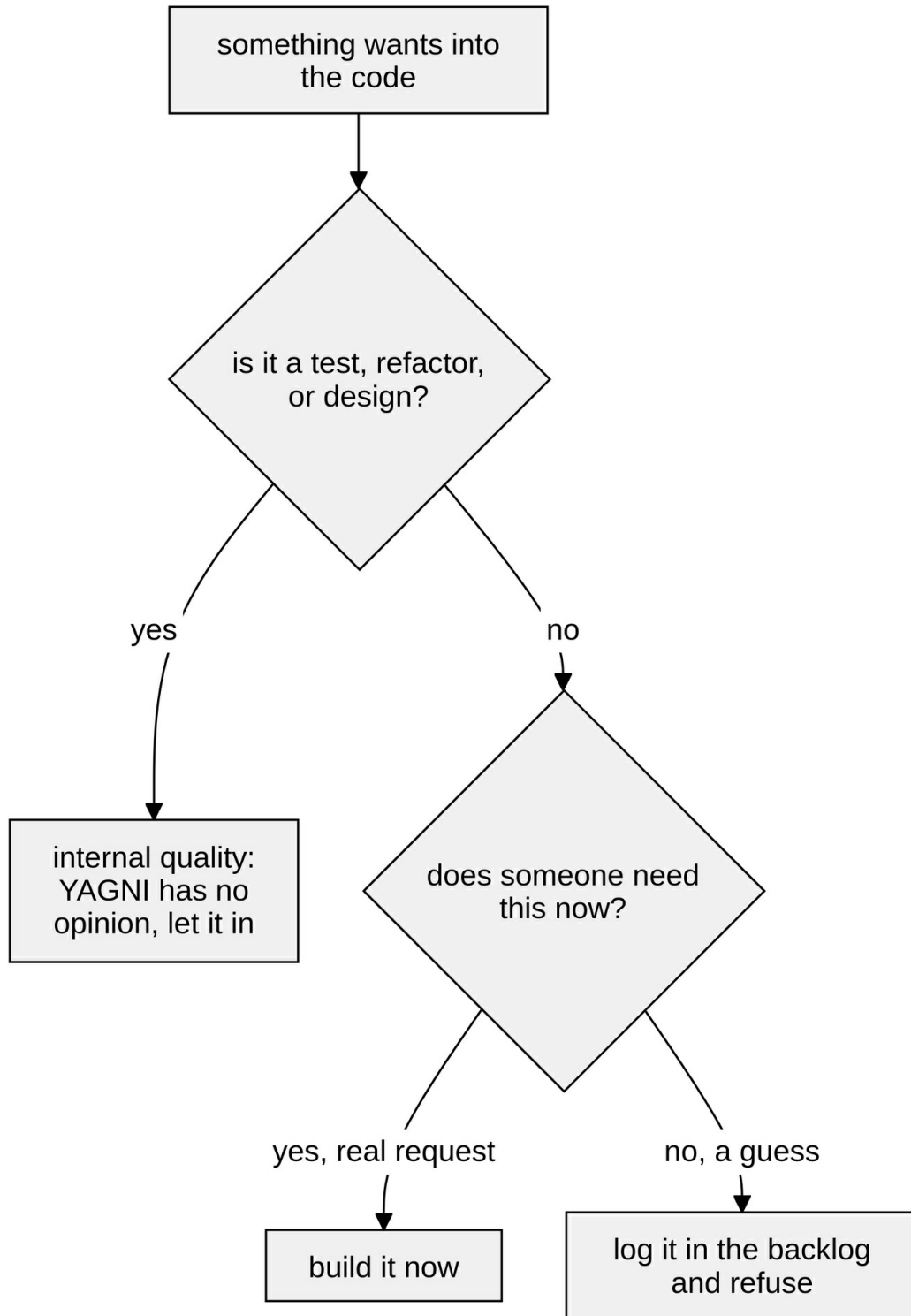
    return pricesInCents[item]

}
```

Try it: the Go version runs at
<https://focus.kodel.com.br/en/go/04-01>.

The code is almost the same, and the almost is the point. Go is the language that turned refusing features into a design philosophy: released in 2009, it spent 13 years saying no to generics, until Go 1.18 arrived in 2022 with a minimal design, only after real use cases had piled up. Rob Pike gave a whole talk about that stance, “Simplicity is Complicated” (dotGo, 2015): every refused feature is a deliberate decision, and the language’s simplicity is the accumulated result of those refusals. You don’t need to adopt Go to take the lesson. You only need to notice that the same test that shrank `PricingEngine` works at the scale of a programming language: the question is never “would this be useful?” because almost anything would be. The question is “does anyone need this now?”

That test deserves to leave the prose and become a flowchart, because it has three exits, not two:



You already know the bottom two exits from this chapter. The top exit is the safety catch of the next section, and it's the one that separates someone who understood YAGNI from someone

who just memorized the acronym.

What YAGNI doesn't cut

Every sharp argument cuts both ways, and YAGNI has been used to justify every untested hack in existence. Fowler closes that door in the same entry that defines the four costs, with a distinction this book treats as law. YAGNI applies to **presumptive capability**: system-visible functionality nobody has asked for yet, like multi-currency, regional tax, and the promotion calendar. YAGNI doesn't apply to **internal quality**: effort that adds no functionality at all but keeps the software easy to change, like tests, refactoring, and clean design. The test for the price calculation isn't a bet on an imagined future; it's what guarantees the present. Cutting it in YAGNI's name is quoting Fowler to disobey Fowler.

The ruler that tells the two cases apart fits into one question: if the future never arrives, does this turn into waste? Multi-currency will happen someday: without that São Paulo location, every line of it is dead weight. The price test will never need it: it pays for itself the first time a menu price changes, new location or not. The same logic undoes the "KISS equals simplistic" reading. The six-line menu isn't the lazy version of the engine; it's the version without the complexity that never paid off. Simplistic is code that cuts what pays off, the test, the precise name, the boundary, just to look short. Johnson wasn't asking for crude airplanes; he was asking for airplanes the field mechanic could actually fix.

And when the need finally arrives?

The classic critique of YAGNI deserves its own section: “refusing today creates rework tomorrow, once the need arrives; building it alongside everything else would have been cheaper.” The hidden premise is that changing the system costs a lot, and that’s where this book’s answer leans on chapter 2, which measured the cost of change as a function of the tangle, not the size of the edit. If adding multi-currency a year from now requires rewriting half the system, the critique holds, and the real problem is that cost of change, not the refusal. In a system where each feature lives in its own independent slice, the cost of adding multi-currency once the São Paulo location actually happens is close to what it would cost today, with one advantage a guess never has: the real shape of the requirement, in hand. Chapter 11 builds those slices, and that’s where this answer settles the bill.

Notice how the pieces fit together, because that fit is the central argument of Part II. YAGNI without cheap-to-change architecture really is a risky bet. Cheap-to-change architecture without YAGNI drowns in speculative code. The two practices pay for each other: you refuse the guess because you know the late addition is cheap, and the late addition is cheap because the system isn’t buried under guesses.

Pitfalls

“YAGNI, so I’m not writing a test.” This is the previous section turned upside down, and it’s the most expensive pitfall in this chapter. Tests and refactoring are internal quality; Fowler’s rule explicitly excludes them from the cut. The way out is mechanical: before you invoke YAGNI, run the diagram’s flow. If what you want to cut is a test, a refactor, or clean design, YAGNI doesn’t even have an opinion.

“I’m never abstracting anything again.” YAGNI refuses presumptive capability, not structure. The function `priceOf` is an abstraction, tiny and justified by today’s use. Once three screens need the price, extracting a module answers a present need, and YAGNI approves. What it blocks is the plugin engine before the second plugin exists.

“The client asked for it, but they won’t actually need it.” YAGNI looks inward, at the capabilities the team presumes it needs; it isn’t a veto over what the user requests. If Rosie asks for scheduled promotions on Thursday, promotions stop being speculation on Thursday. Refusing a real request while quoting the acronym is just stubbornness wearing a principle’s name.

Q&A

What if I’m almost certain I’ll need it? “Almost certain” is exactly the *foresee* in Jeffries’s sentence, and the answer is still no. Write the guess down in the backlog in one line; if it comes true, you build it with the real requirement in hand, without paying the repair cost. What you lose by waiting is almost always smaller than the four costs added up from building it now.

Doesn’t YAGNI conflict with designing architecture? No, thanks to the distinction from the previous section: architecture that makes change cheap is internal quality, the investment that makes refusal safe. The conflict is with speculative architecture, the plugin engine before the first plugin. Independent slices cost about the same whether you have one feature or twenty; the engine that carried multi-currency support cost 95 lines before it ever ran a single conversion, and the menu needed six.

Doesn't deleting a finished `PricingEngine` waste what's already been paid for? The cost of build is already spent; deleting the code doesn't refund it. What deletion refunds is the cost of carry on every future reading of the file, and that's the only cost still open. Code you already paid for isn't a reason to keep code that's expensive to keep; economists have a name for the opposite instinct: the sunk cost fallacy.

Quick tip

Speculation leaves a trace in the history: a file that was born big and never changed again. Run `git log --oneline --path/to/file | wc -l` on the suspects; a result of `1` means nobody has needed to touch that file since it was born, and it's worth asking whether anyone ever needed it at all.

Quick reference

Cost	What it charges	Where it showed up in <code>PricingEngine</code>
Build	Hours of analysis, code, and testing nobody uses	2 days for 95 lines
Carry	Reading, debugging, and defects for everyone	<code>currency!</code> in <code>priceOf</code>
Delay	Value of the requested feature	Requested Wednesday,

	stuck in the queue	shipped Friday
Repair	The guess gets the shape wrong; redo it later	Regional tax rate vs. flat rate

Situation	Fix
A feature proposal shows up	Run the diagram: request, guess, or quality
“What if I need it someday?”	One-line backlog entry; build it when it arrives
Cutting a test or a refactor	Refuse: internal quality, YAGNI has no opinion
Fear of future rework	Make change cheap (chapter 11), don’t guess ahead

Exercises

- Five proposals came in for the menu code, and each hunk below starts from the original file, independent of the others. For each one, decide: real request, speculation, or internal quality? What do you approve, and what do you send back with “you aren’t gonna need it”? The answer key comes right after; resist looking at it.

```

--- a/menu/menu_price.dart

+++ b/menu/menu_price.dart

```

@@ hunk 1: Rosie added a tea to the winter menu @@

```
const pricesInCents = <String, int>{  
  
    "cappuccino": 1195,  
  
    "cheese bread": 600,  
  
    "house coffee": 800,  
  
+   "hibiscus tea": 700,  
  
};
```

@@ hunk 2: lay the groundwork for foreign currencies @@

```
-int priceOf(String item) => pricesInCents[item]!;  
  
+int priceOf(String item, {String currency = "USD"}) =>  
  
+   pricesInCents[item]!;
```

@@ hunk 3: cover the price calculation with a test @@

```
+void main() {  
  
+   assert(priceOf("cappuccino") == 1195);  
  
+   print("price calculation ok");
```

```

+}

@@ hunk 4: extension point for future tax rules @@

+int withFutureTaxes(int valueInCents) => valueInCents;

@@ hunk 5: a name that states the unit of the return value @@

-int priceOf(String item) => pricesInCents[item]!;

+int priceInCentsOf(String item) => pricesInCents[item]!;

```

Answer key, hunk by hunk. Hunk 1 is a real request: Rosie added the item, approve it. Hunk 2 is classic speculation, a parameter no call uses that exists purely on a forecast; send it back, and notice how it echoes the `PricingEngine` bug. Hunk 3 is internal quality: a test for what exists today, approve it without ever invoking YAGNI. Hunk 4 is speculation in its purest form, a function that returns its own argument while waiting for a future; send it back. Hunk 5 is internal quality: renaming the function to tell the truth about cents improves every future reading without adding any capability at all, approve it.

2. Could you run this chapter's autopsy on code of your own? Pick a repository you maintain, find the file that looks the most like `PricingEngine`, the one born ready for a future that still hasn't shown up, and measure the four costs on it: build hours you remember, concepts a new reader has to cross, what sat in the queue at the time, and how much of the guess still matches today's actual need.

Tip 4

A feature nobody asked for is debt everybody pays.

Next chapter: this chapter's scissors meet their first hard case, and it looks harmless: when the same code shows up twice, is deleting it always the answer?

DRY Isn't About Code

In this chapter, you'll:

- decide, faced with two duplicated snippets, whether they carry the same knowledge or just the same text, and unify or keep them apart with a business justification;
- apply the rule of three when the honest answer is “I don't know”;
- explain why undoing the wrong abstraction costs more than deleting a duplication.

Two identical functions sit side by side, staring back at you. Every trained instinct you own says delete one and extract the other into a shared spot, because duplication is a sin and you learned that before you learned to code properly. This chapter tells the story of the day that instinct cost Rosie money, and hands you the question that separates cleanup from a trap: do these two snippets change for the same reason?

Chapter 4 ended on a taunt: when the same code shows up twice, is deleting it always the answer? That chapter's scissors decided what not to build. This chapter's decide what not to unify, and both cut along the same edge: real need, never reflex. There's even a name for the family resemblance: AHA (Avoid Hasty Abstractions), coined by Kent C. Dodds, is YAGNI applied to abstractions. Build only once the need actually arrives, never because you predict it's coming. That holds for an entire pricing engine, and it holds for a five-line extracted function.

Extraction by reflex

Rosie runs two promotions in the app. The loyalty program gives 10% off to whoever stamps their tenth purchase; it exists to reward the customer who comes back every week. The daily combo gives 10% off the cappuccino-and-cheese-bread pair; it exists to move stock that's at risk of sitting unsold in the display case. Two different business decisions, made on different days, for different reasons. In the code, they were born like this:

Dart

```
// Loyalty program: rewards customers who come back every week.

int loyaltyDiscount(int totalInCents) {

    return (totalInCents * 10) ~/ 100;

}

// Daily combo: moves stock that's at risk of sitting unsold.

int dailyComboDiscount(int totalInCents) {

    return (totalInCents * 10) ~/ 100;

}
```

The bodies are identical, character for character. A coworker opens the file for a different task, spots the repetition, and feels the itch you know well. Two identical snippets, one obvious refactor, thirty seconds of work. He extracts the shared function, deletes both copies, and pushes the commit with the message “remove duplication.” Nobody objects in review; the diff even shrank the file.



```
// The function that unified the two "identical" rules.

// Three months later loyalty jumped to 15%, and the fix landed here.

int calculateDiscount(int totalInCents) {

    return (totalInCents * 15) ~/ 100;

}

int loyaltyDiscount(int totalInCents) =>

    calculateDiscount(totalInCents);

int dailyComboDiscount(int totalInCents) =>

    calculateDiscount(totalInCents);
```

TypeScript

```
function calculateDiscount(totalInCents: number): number {  
  
    return Math.floor((totalInCents * 15) / 100);  
  
}
```

```
function loyaltyDiscount(totalInCents: number): number {  
  
    return calculateDiscount(totalInCents);  
  
}
```

```
function dailyComboDiscount(totalInCents: number): number {  
  
    return calculateDiscount(totalInCents);  
  
}
```

What actually changes is the math. Dart has `~/`, an operator that divides and truncates in one step, so the result already comes out whole. TypeScript only knows floating-point division, and `/` returns a fraction the moment the total isn't round: `Math.floor` sits there to cut that fraction off before half a cent sneaks onto Rosie's tab. Go is the cheap counterpoint, where neither gesture

is needed, because `/` between two `int` truncates by the language's own definition. Same business rule, three spellings. That's why the listings from here on run in Dart alone.

Three months later Rosie decides to boost loyalty: 15% starting Monday. The task lands on someone who's never seen this file. That person looks for the rule, finds `calculateDiscount`, swaps 10 for 15, tests the loyalty flow, it works. On Monday the daily combo wakes up giving 15% too. On a \$20 tab, the combo's discount used to be \$2 and becomes \$3; the difference comes straight out of Rosie's margin on every cappuccino-and-cheese-bread pair sold, and nobody notices until the month closes out. The loyalty test passed. There was no test saying the combo should stay at 10%, because in everyone's head it was "the" discount.

Try it: run the bug's two stages at <https://focus.kodel.com.br/en/dart/05-01> or <https://focus.kodel.com.br/en/ts/05-01>. Stage 1 runs the unified code with loyalty at 15% and shows the combo discount jumping right along with it; stage 2 undoes the extraction and prints the correct values. Notice the size of the fix before you keep reading.

Where was the bug born? The reflex answer is "the combo was missing a test," and it's true and insufficient. The bug was born in the extraction. To see that precisely, you need the original definition of DRY, the one almost nobody quotes in full.

What Hunt and Thomas actually wrote

DRY (Don't Repeat Yourself) appeared in *The Pragmatic Programmer* (Andy Hunt and Dave Thomas, 1999), and the original wording doesn't mention code at all:

“Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”

The word carrying the whole sentence is *knowledge*. **Knowledge**, here, is a decision about the business or about the system: loyalty pays 10%, the card reader fee is 3.49%, a canceled order never reaches the kitchen. **Text** is the shape that decision takes in a file. DRY forbids duplicating knowledge. About *text*, it says nothing at all.

In the twentieth-anniversary edition (2019), Hunt and Thomas spent a whole section undoing the misunderstanding this chapter is attacking. Two clarifications matter here. First: DRY is wider than code and covers database schemas, documentation, and build scripts; if the same decision lives in the schema and in a validation, it's duplicated, even without a single repeated line of code. Second, and this is the sentence that takes the coworker's commit apart: two pieces of code that are textually identical but represent different decisions do NOT violate DRY. Identical code with distinct meanings even has a name: **accidental duplication**, the coincidence of two independent decisions producing, for now, the same text. And notice *accidental*: it isn't something bad, catastrophic. Accidental just means something that happens by chance, without intent.

Now the bug's rediagnosis is short. `loyaltyDiscount` and `dailyComboDiscount` were accidental duplication: two pieces of knowledge, the reward for repeat visits and the push on inventory, that happened to be worth 10% in the same quarter. The text was one; the reasons to change it were two. Whoever extracted `calculateDiscount` didn't remove a duplication, because there never was one. They removed the boundary between two of

Rosie's decisions, and from then on any change to one would drag the other along. The DRY violation was the extraction, not the duplication. The commit message had it backwards.

Try it: before you turn the page, judge another case from the same app. The number 0.0349, the fee the card processor charges per card sale, shows up in three files: recording the sale, closing out the day, and simulating a price. Same knowledge, or coincidence? Unify, or keep separate? Decide now; the answer comes in the next section.

The inverse case: the card reader fee

If you answered “unify,” you got it right, and the reason matters more than the verdict. Look at the current state:

Dart

```
// payment.dart: deducts the fee when recording a card sale

int netSaleAmount(int amountInCents) =>

    amountInCents - (amountInCents * 0.0349).round();

// closing.dart: projects what the card reader pays out at day's end

int dailyPayout(int cardTotalInCents) =>

    cardTotalInCents - (cardTotalInCents * 0.0349).round();
```

```
// pricing.dart: shows how much of an item's margin the fee eats

int feeOnPrice(int priceInCents) =>

    (priceInCents * 0.0349).round();
```

The three snippets aren't even alike; each function carries a different name, parameter, and slightly different math. And yet this is exactly the duplication DRY forbids. The three 0.0349s are the same knowledge: the fee written into Rosie's contract with the card processor. There's a single document in the real world that defines that number. When the processor adjusts it to 3.79%, all three spots need to change together, at the same moment, for the same reason; whoever forgets the third one creates a day's closing that never matches the statement. The test was never "is the text the same?" The test is "is there a single decision behind it?" Here there is, so the representation has to be single too:

Dart

```
// fees.dart: the one line that changes when the processor adjusts its rate

const cardReaderFee = 0.0349;

int feeOn(int amountInCents) =>

    (amountInCents * cardReaderFee).round();
```



```
// payment.dart

int netSaleAmount(int amountInCents) =>

    amountInCents - feeOn(amountInCents);

// closing.dart

int dailyPayout(int cardTotalInCents) =>

    cardTotalInCents - feeOn(cardTotalInCents);

// pricing.dart

int feeOnPrice(int priceInCents) => feeOn(priceInCents);
```

Put the two verdicts side by side, because together they're the chapter's lesson. The discounts had identical text and different knowledge: keep them separate. The fee had different text and a single piece of knowledge: unify right away, without waiting for a third occurrence. The eye that compares characters gets both cases wrong. The question that gets both right doesn't look at the code; it looks at Rosie's business.

Timing tools

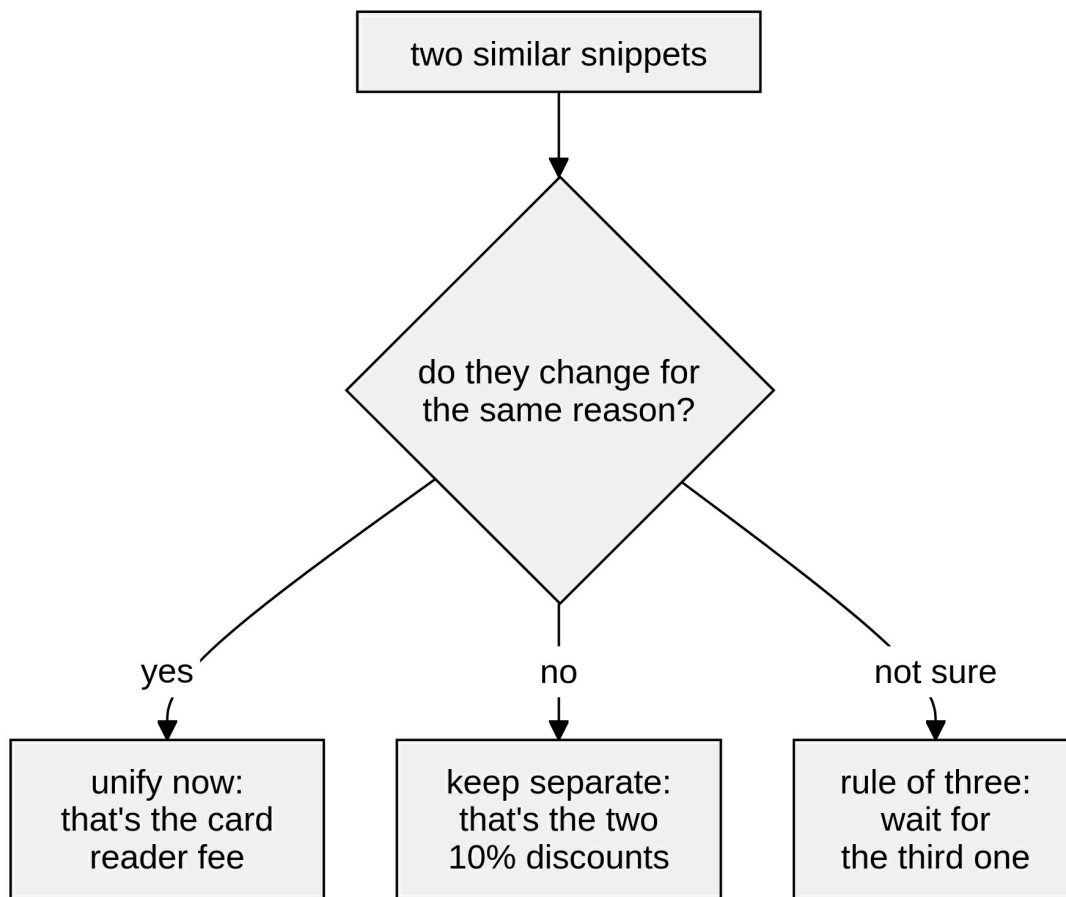
Knowing the right question doesn't erase the hard case: what about when you can't answer it? A small toolbox has built up around exactly that impasse, and each piece has an owner and an address.

The first is from Sandi Metz, in "The Wrong Abstraction" (2016): "prefer duplication over the wrong abstraction." The **wrong abstraction** is the function or class that unifies snippets that never carried the same knowledge, exactly what `calculateDiscount` became. Metz's argument is about interest. The wrong abstraction doesn't sit still waiting for you to undo it: the next almost-matching case shows up, someone adds a parameter to accommodate it, then a conditional, and every patch raises the price of taking the whole thing apart. Duplication just sits there instead, repeated and harmless, until someone understands it.

The second you already met in the opening: Kent C. Dodds's AHA (kentcdodds.com, "AHA Programming," 2019). Dodds doesn't ask you to duplicate forever; he asks you to wait for the abstraction to reveal itself, instead of forcing it at the first resemblance. It's the same muscle from chapter 4: you don't build capacity on a forecast, and you don't abstract on one either. Abstracting at the first coincidence is betting that two snippets will evolve together before you have any evidence of it.

The third tool answers "wait until when?" The **rule of three**, which Martin Fowler records in *Refactoring* (1999), says: the first time you write it, the second time you duplicate with your eyes open, the third time you extract. The third occurrence is the missing evidence; with three uses in hand, the abstraction's real shape shows up, and you build it knowing exactly what it needs to cover. Two occurrences are still too small a sample to guess the right boundary.

The fourth is less a rule and more a mnemonic reminder. Conlin Durbin coined WET (Write Everything Twice) in “What is WET code?” (dev.to, 2018): tolerate the second copy and only abstract on the third. It’s the rule of three dressed up as a pun on DRY, and it works as a short answer for the coworker who flags any second occurrence as debt. None of these four pieces contradicts Hunt and Thomas. Metz, Dodds, Fowler, and Durbin regulate the timing of abstracting similar-looking text; DRY demands a single representation for a single piece of knowledge. The whole flow fits in one diagram:



Run the chapter’s two cases through it. The discounts enter the question node and exit through “no”: loyalty changes when Rosie wants to reward more, the combo changes when inventory gets tight, independent reasons. The fee enters and exits through

“yes”: one contract, one number, three points of use. You’ll exercise the “not sure” branch in the exercises, with a pair that has no obvious answer on purpose.

The same knowledge outside the code

The 1999 sentence talks about a system, not a file, and that’s why it aged well. Rosie’s decision about the loyalty discount has more places to settle into today than it had back then, and three of them aren’t code.

Semantic duplication is the same decision written twice in different words. The use case requires the tenth purchase to unlock the 10%; the customer screen works out how many stamps are missing and prints “two to go”, with the arithmetic redone right there. No text search finds that pair, because there’s no repeated text: what repeats is the rule. When Rosie starts requiring twelve purchases, the use case changes and the screen keeps counting to ten. The criterion is the one it always was: both change for the same reason, so the decision needs a single representation, and the screen asks instead of recomputing.

Prompt duplication is the rule that comes to live in the request you write to ask for code as well. You paste into the request that the loyalty discount is 10% from the tenth purchase on, you get the function back and, from then on, the rule lives in two places: in the file and in the text of the request, which usually sits saved in some project instructions file. When the discount goes up to 15%, the file changes and the request doesn’t, and the next answer comes back with the old version, now carrying the authority of something fresh off the machine. The way out is the card reader fee’s: the request cites the file instead of repeating the rule, and what it carries is the address, not the number.

Context duplication is the copy someone makes to spare the reader from leaving the file. The comment that re-explains the discount rule at the top of the repository, the README that reproduces the use case's signature, the snippet pasted into the team's documentation. Each copy is a photograph of the day it was taken, and none of them breaks when the original changes: they just go quietly wrong, which is the worst way to be wrong.

All three go through the same question, and that's what keeps the extension from turning into another crusade against repeated text. The example code that shows up three times in this chapter isn't duplication: it exists to teach, and it ages along with the page. The discount rule copied into the request is: it exists to decide, and it decides wrong the moment it ages.

The critique: DRY as a coupling factory

Every popular principle collects criticism, and the most serious one against DRY is this: DRY breeds premature abstraction that couples what evolves separately. Whole teams, trained to hunt duplication, produce layers of generic helpers that nobody can change without breaking three screens. The critique describes real damage; you saw a miniature of it in Rosie's combo. Except its target is the misunderstanding, not the principle. Whoever extracted `calculateDiscount` was violating DRY, not applying it: they unified two pieces of knowledge into one representation, the literal opposite of the 1999 sentence. DRY correctly read and AHA don't compete for territory. DRY governs knowledge: a single representation for a single decision. AHA governs timing: without evidence it's the same decision, wait. The dispute between the two only exists once DRY turns into "delete all repeated text," and that version isn't on a single page Hunt and Thomas wrote.

Here's my position, so you can calibrate your own: between duplicating and risking the wrong abstraction, I duplicate and sleep fine. Undoing a duplication that turned out to be a single piece of knowledge is search and replace, ten minutes with the editor and the tests. Undoing the wrong abstraction is surgery: every caller depends on it in its own way, the accommodation parameters already created combinations nobody tested, and removing it means understanding every use case at once. Both mistakes are possible; the prices aren't in the same league.

Pitfalls

The `shared/` utility born on the second occurrence. You write a function, notice another feature has something similar, and create `shared/Utils.dart` for both right away. It's reflex extraction with a fancier address: with two occurrences you rarely know whether there's one piece of knowledge or two, and the shared directory invites the rest of the team to hang parameters off it. The way out is the diagram from the previous section: ask the knowledge question; when unsure, rule of three, and the utility only gets born on the third occurrence, shaped by whatever the three uses actually need.

"So I'm never extracting anything again." That's the mirror-image conclusion, and it costs just as much as the original. When the knowledge is genuinely one thing, like the card reader fee, unifying isn't optional and doesn't need a third occurrence: leaving it scattered is betting that three spots will change together by hand, forever, without a single slip. The rule of three is the way out of "I don't know," never a veto over "yes."

Unifying the text to "stay ready." The coworker argues that if the two rules ever truly converge, the code will already be prepared. You know this argument from chapter 4: it's presumed

capability, now wearing a function's shape. If the rules do converge, that day's extraction will be cheap and informed. Today's is a guess with the power to spread bugs.

Q&A

How do I find out if two snippets are the same knowledge?

Look for the decision's source outside the code. The fee has a signed contract; the discounts have two business motivations with different owners. If the question "who's in charge of this number?" points to two places, it's two pieces of knowledge, whether the text matches or not.

Wouldn't the extraction be defensible if `calculateDiscount` had a good test?

A test on the combo would have turned the silent bug into a visible failure, and that alone would be worth a lot. But the coupling would still be there: every change to loyalty would still run into the combo, now with a red test in the way. A good test exposes the wrong abstraction; only undoing it fixes it.

Does DRY apply outside code? It does, and that's the part the 2019 edition makes a point of underlining: database schema, documentation, and build also carry knowledge. If the menu says the cappuccino costs \$11.95 and a constant in the app says 1195 cents, those are two representations of the same decision, and one of them is going to rot.

Why 1195 cents instead of 11.95? Because money in floating point is a risk, as chapter 4 already flagged: float represents 0.10 as a binary approximation, and cents vanish in long sums. An integer is always worth the same thing, no matter how the language represents it (float, double, decimal) or how it travels (JSON, ProtoBuf); JSON parsers, in particular, decode a broken number into a float, so what leaves one side as an integer arrives whole on the other. The 0.0349 fee in

the listings can stay a float because it's a multiplier, not stored money; the result lands back in whole cents inside that same line's `round()`.

Quick tip

Before you extract a function to kill a duplication, run `git log -p` on the snippets involved. If they changed in separate commits, for separate reasons, that's strong evidence of two pieces of knowledge; if every change to one always came bundled with the other, unifying is probably overdue.

Tip 5

Before you unify two identical snippets, ask whether they change for the same reason.

Quick reference

Situation	Fix
Same text, same knowledge	Unify now, don't wait for a third time
Same text, different knowledge	Keep separate: it's accidental duplication
Not sure if it's the same knowledge	Rule of three: wait for the third time
Wrong abstraction already in place	Undo it and duplicate back; then reassess

Exercises

1. Four pairs of snippets from Rosie's Coffee Shop app. For each one, give your verdict: unify now, keep separate, or rule of three. The answer key comes right after; decide before you read it.

There are four pairs. Pair 1 is the **minimum delivery fee**, which shows up in the order calculation and again on the receipt screen. Pair 2 is two values of **10 minutes**: the prep time for the cornmeal cake and the validity window for the pickup code at the counter. Pair 3 is **price formatting**, which turns cents into "\$11.95" on two different screens. Pair 4 is two **roundings** that share the same rule today, one on loyalty points and the other on change owed, with no business decision recorded about either.

Answer key, pair by pair. Pair 1 is one piece of knowledge: one business value, two points of use; unify now. Pair 2 is numeric coincidence: prep time changes if the oven changes, the code changes if the line gets long; keep separate. Pair 3 is one piece of knowledge, and the kind the 2019 edition widened its scope to cover: how money gets represented in the system is a single decision, even though it's formatting and not a value; unify now. Pair 4 is the legitimate "I don't know": nobody decided that points and change round together, and nobody decided they don't; rule of three, wait for the third occurrence or the first change that pulls the two apart.

2. Could you repeat this judgment call on your own code? Pick a repository you maintain, find the number or string that repeats the most, and ask this chapter's question: who's in charge of this value? If the answer is a single decision,

measure how many spots you'd have to edit today if it changed; that number is your risk of a closing that never matches.

Next chapter: you learned to sniff out duplicated knowledge in functions and constants, but what about when the duplication lives in the shape of your classes, and “do they change for the same reason” becomes the question that decides an entire system's design?

SOLID Without Dogma

In this chapter, you'll:

- name, in front of a change that hurt, which principle the pain violates, using one test question per principle;
- slice Rosie's `Tab` into per-actor responsibilities, without falling into the tiny-class factory;
- justify a decision to NOT apply a principle, and say out loud when applying it would be pure ceremony.

Rosie's `Tab` calculates the total, applies the loyalty discount, formats the receipt, and writes everything to the database: one class serving four different bosses. In chapter 2 you learned to measure the cost of a piece of code by the reach of the changes it drags along. This chapter turns that measure into a five-question filter that, in front of a change that hurt, tells you which coupling force you stepped on. The filter is called SOLID, and it arrives here without the commandment weight people usually hang on it.

Recap of chapter 2 in one line: coupling is the reach of a change, cohesion is how much of a file changes together, and axis of change is the reason someone opens the code. That chapter diagnosed the disease. This one delivers the vocabulary for the diagnosis. SOLID (Single responsibility, Open-closed principle, Liskov substitution, Interface segregation, Dependency inversion) bundles five principles that Robert C. Martin compiled in *Design Principles and Design Patterns* (2000). The acronym

came later: around 2004, Michael Feathers noticed that the rearranged initials spelled “solid,” and the name stuck. The order of the letters is marketing, not hierarchy. Two of the principles are considerably older than the compilation, as you’ll see, and none of them was born as law.

One tab, four bosses

Before any principle, the code that motivates all of them. The Tab below is in production in Rosie’s Coffee Shop app, and it does everything the word “tab” suggests:

Dart

```
// Anti-solution: the Tab that serves four actors in a single file.
```

```
class Tab {
```

```
  Tab(this.items, {required this.isTenthPurchase});
```

```
  final List<({String name, int priceInCents})> items;
```

```
  final bool isTenthPurchase;
```

```
  // Calculates the total: finance's rule.
```

```
  int calculateTotal() {
```

```
var sum = 0;

for (final item in items) {

    sum += item.priceInCents;

}

return applyLoyalty(sum);

}

// Applies loyalty: marketing's rule (went from 10% to 15%).

int applyLoyalty(int sum) {

    if (!isTenthPurchase) {

        return sum;

    }

    return sum - (sum * 15) ~/ 100;
```

```
}

// Formats the receipt: the printer's layout.

String formatReceipt() {

    final lines = [

        for (final item in items) "${item.name}: ${item.priceInCents}",

        "TOTAL: ${calculateTotal()}",

    ];

    return lines.join("\n");

}

// Persists the tab: the DBA's schema.

Map<String, Object> toDatabaseRow() {

    return {"items": items.length, "total_in_cents": calculateTotal()};

}
```

}

Read the subgoal comments and notice who's in charge of each chunk. The total rule belongs to Rosie's finance side. The loyalty rule belongs to marketing, which adjusts the percentage whenever it wants to drive traffic. The receipt layout belongs to the printer and to the accountant's requirements. The persisted row's schema belongs to whoever owns the database. Four groups of people, each with its own agenda, and each requests changes in the same file. Keep that count in mind.

Now the scene. Marketing just bumped loyalty from 10% to 15%, a one-character edit in `applyLoyalty`, and that's the version you just read. The following week, a customer completes her tenth purchase, pays for an \$11.95 cappuccino with a \$6.00 cheese bread, and asks for the receipt to get reimbursed by the company she works for. Her company's accounting turns it down: the printed items add up to 1795 cents, the TOTAL says 1526, and no line on the paper explains where the 269 went. The receipt has been wrong since the 10% days; marketing's change only widened the hole until someone noticed.

The fix the accountant is asking for looks trivial: one "LOYALTY: -269" line before the TOTAL. Try implementing it inside this class. The discount doesn't exist as a number anywhere: it gets swallowed inside `applyLoyalty`, which hands back the already-reduced sum to `calculateTotal`. To give the receipt the line the accountant demands, you touch marketing's function and finance's function, and `toDatabaseRow` changes behavior right along with them, because it also calls `calculateTotal`. One boss's requirement forces you to edit two other people's code and changes a third person's output. That cascade has a name, and the name is the subject of the next section.

Try it: run the cascade at <https://focus.kodel.com.br/en/dart/06-01> or <https://focus.kodel.com.br/en/ts/06-01>. Step 1 prints the monolithic `Tab`'s receipt, with the items adding up to 1795 and the TOTAL saying 1526, no discount line. Step 2 prints the same order in the sliced version you're about to build next, with the LOYALTY line closing out the math. Compare the two receipts before you go on.

Slice by actor, not by verb

The first principle in the acronym is the most quoted and the worst read of all five. SRP (Single Responsibility Principle) says, in Martin's own formulation (2000): a class should have one, and only one, reason to change. Notice what the sentence doesn't say. It doesn't say "a class does one thing." Doing is about code; reason to change is about people. In later writing Martin himself tied off the loose end: reason to change is synonymous with **actor**, the group of people who ask for that kind of change. Finance is one actor. Marketing is another. The SRP question was never "how many things does this class do?" It's "how many bosses does this file have?"

The distinction matters because the two readings slice the `Tab` in different places. "A class does one thing" slices by verb: sum, discount, format, save, each verb in its own class, with no stopping rule. One reason per class slices by actor, and the `Tab` has exactly four:

Dart

```
// Finance: how the sum becomes a total.
```



```
class TotalCalculator {

    TotalCalculator(this.policy);

    final LoyaltyPolicy policy;

    int calculate(List<Item> items, {required bool isTenthPurchase}) {

        var sum = 0;

        for (final item in items) {

            sum += item.priceInCents;

        }

        return sum - policy.discount(sum, isTenthPurchase: isTenthPurchase);

    }

}

// Marketing: how much discount loyalty gives.
```

```
class LoyaltyPolicy {

    int discount(int sum, {required bool isTenthPurchase}) {

        if (!isTenthPurchase) {

            return 0;

        }

        return (sum * 15) ~/ 100;

    }

}

// Receipt printer: the receipt layout, with the discount on its own

// line.

class ReceiptFormatter {

    String format(

        List<Item> items, {

            required int discountInCents,
```

```

        required int totalInCents,

    }) {

        final lines = [

            for (final item in items) "${item.name}: ${item.priceInCents}",

            if (discountInCents > 0) "LOYALTY: -$discountInCents",

            "TOTAL: $totalInCents",

        ];

        return lines.join("\n");

    }

}

// DBA: the persisted row's schema.

class TabRepository {

    final _rows = <Map<String, Object>>[];

```

```

void save(List<Item> items, int totalInCents) {

    _rows.add({

        "items": items.length,

        "total_in_cents": totalInCents,

    });

}

}

```

It's the same `Tab` from the previous section, business line by business line (sum, discount, receipt, database), now with one file per boss. The accountant's requirement turned trivial. The discount is now a number with its own name, one that leaves `LoyaltyPolicy` and enters the formatter as a parameter; the LOYALTY line cost one `if` in the layout, without touching finance's rule or marketing's. When marketing tweaks the percentage again, the edit happens in a class whose only boss is marketing. The receipt still adds up, because whoever prints it receives the discount ready-made instead of having to guess it.

The same slice in TypeScript shows where the translation hurts and where it doesn't:

TS TypeScript

```

// Marketing: how much discount loyalty gives.

```

```

class LoyaltyPolicy {

  discount(sum: number, isTenthPurchase: boolean): number {

    if (!isTenthPurchase) {

      return 0;

    }

    return Math.floor((sum * 15) / 100);

  }

}

```

Two differences, and neither is about design. The first is division: Dart's `~/` truncates on its own, and TypeScript needs `Math.floor` so it doesn't hand back 269.25 cents. The second is the named parameter: in Dart, `{required bool isTenthPurchase}` forces the caller to write `isTenthPurchase: true` at the call site, and TypeScript has no such feature, so the boolean goes in by position and readability drops a notch. The actor is still just one, and that's what the SRP measures. The rest of this chapter's listings run in Dart, with the same correspondence holding.

Slicing works, and it's exactly because it works that it turns into a habit hard to break. Look at what happens when the knife keeps going after the actors run out:



```
// The overdone slice: nine lines no actor asked for.
```

```
class SubtotalCalculator {  
  
    int calculate(List<Item> items) {  
  
        var sum = 0;  
  
        for (final item in items) {  
  
            sum += item.priceInCents;  
  
        }  
  
        return sum;  
  
    }  
  
}
```

It looks professional. “Summing items is one responsibility, discounting is another,” says the colleague in the review, and the sum gets its own file. Ask the actor question before you approve it: who asks for a change to the subtotal? Finance. Who asks for a change to the total? Finance. Same boss, same reason, same class. The split doesn’t eliminate a single reason to change; it just spreads the same reason across two files that now need to change together. That’s new coupling dressed up as organization. The slice goes back inside `TotalCalculator` in the next commit, and chapter 4 already gave you the name for the rule that justifies

reverting it: YAGNI (You Aren't Gonna Need It). Slicing without an actor asking for it is building presumed capacity, this time in the shape of a class.

First question in the filter, then: **how many actors ask for changes in this file?** More than one, and the SRP is violated; the cascade pain is a matter of time. Exactly one, stop slicing, even if the class “does two things.”

Read the code through the OCP and LSP lenses

The next two principles are older than Martin's compilation, and in this section you won't write new code for them: you'll reread the code you just sliced. OCP (Open-Closed Principle) comes from Bertrand Meyer, in *Object-Oriented Software Construction* (1988): software entities should be open for extension and closed for modification. LSP (Liskov Substitution Principle) comes from Barbara Liskov's talk “Data Abstraction and Hierarchy” (1987): if one type substitutes another, the program can't tell the difference.

Reread `TotalCalculator` with these two lenses. It receives the loyalty policy ready-made instead of knowing the percentage itself. The day marketing invents a new policy, the calculator doesn't get edited: it receives a different policy. Extension without modification, the OCP in one sentence. And the swap only works if every policy behaves the way the original one promised: if one of them returns a negative discount, or one larger than the sum, the total breaks and the caller notices. Substitutability, the LSP in one sentence. No new hierarchy was created to satisfy either principle. They don't ask for structure; they ask that the existing structure respect two forces, the direction of who knows whom, and the confidence that the swap is safe.

If you still suspect these principles are language syntax tied to inheritance, Go takes that suspicion apart:

Go

```
// The calculator depends on an implicit interface: any type that has  
  
// Discount qualifies, without declaring that it implements anything.  
  
type DiscountPolicy interface {  
  
    Discount(sumInCents int) int  
  
}  
  
type Loyalty struct{}  
  
func (Loyalty) Discount(sumInCents int) int {  
  
    return sumInCents * 10 / 100  
  
}  
  
type NoDiscount struct{}
```



```
func (NoDiscount) Discount(sumInCents int) int {  
  
    return 0  
  
}  
  
// Composition instead of inheritance: the calculator carries the  
  
// policy. Swapping the policy doesn't edit a single line here.  
  
type TotalCalculator struct {  
  
    Policy DiscountPolicy  
  
}  
  
func (c TotalCalculator) Calculate(pricesInCents []int) int {  
  
    sum := 0  
  
    for _, price := range pricesInCents {  
  
        sum += price  
  
    }  
  
    return sum - c.Policy.Discount(sum)
```

```
}
```

🐹 Go has no inheritance, and `Loyalty` never declares anywhere that it implements `DiscountPolicy`: it just needs the `Discount` method with the right signature, and the interface is satisfied implicitly. Even so, both forces are fully present in the code above. The dependency direction points from the calculator to the interface, never to a concrete policy, and that's what keeps the calculator closed for modification when a new policy shows up. Substitutability is the behavior contract between `Loyalty` and `NoDiscount`: either one drops into the other's place without `Calculate` noticing. The syntax changes from one language to the next; the forces OCP and LSP name stay the same. Anyone who concludes that "Go doesn't need SOLID" is looking at the absence of `extends`, when they should be looking at the dependency arrow the code draws.

Two more questions for the filter. OCP: **does extending require editing what already works?** LSP: **can I swap the implementation without the caller noticing?**

Narrow the contract and flip the arrow

What's left is the pair FOCUS leans on at full strength, and it lives at the app's most unstable boundary: persistence. ISP (Interface Segregation Principle) says no client should depend on methods it doesn't use. DIP (Dependency Inversion Principle) says business rules shouldn't depend on infrastructure detail; both should depend on an **abstraction**. And abstraction, in the DIP sense, means depending on the contract that declares the behavior, never on the implementation that fulfills it. The term doesn't require the `abstract` keyword: a three-line interface is abstraction enough.

In practice the two principles arrive together, because whoever defines the contract is whoever consumes it. The `CloseTab` use case needs a single persistence operation, so it declares a contract that size:

Dart

```
// The narrow contract: only what the use case demands (ISP).

abstract interface class TabRepository {

    void save(List<Item> items, int totalInCents);

}

// The use case depends on the abstraction, not the implementation
// (DIP).

class CloseTab {

    CloseTab(this.repository);

    final TabRepository repository;

    void execute(List<Item> items, int totalInCents) {
```

```
        repository.save(items, totalInCents);

    }

}

// The implementation knows the contract; the reverse never happens.

class SqlTabRepository implements TabRepository {

    final _rows = <Map<String, Object>>[];

    @override

    void save(List<Item> items, int totalInCents) {

        _rows.add({

            "items": items.length,

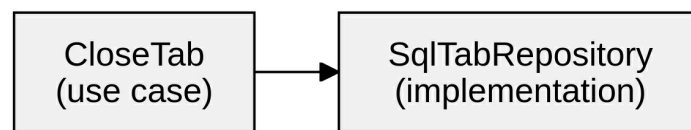
            "total_in_cents": totalInCents,

        });

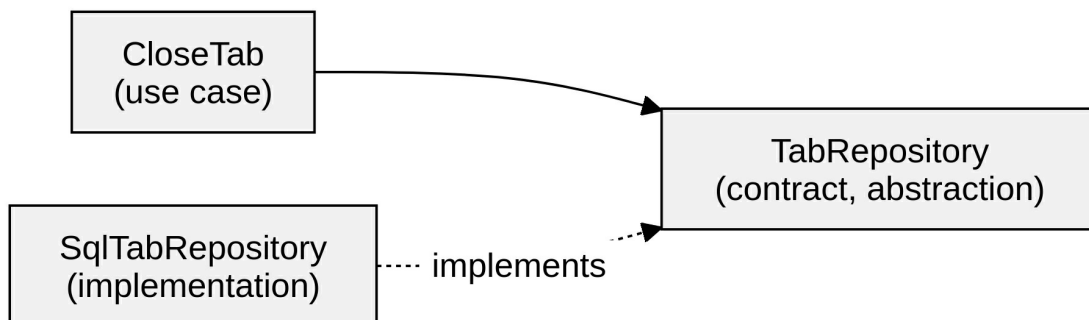
    }

}
```

The contract has one method because the use case uses one method. If the concrete repository offers twenty operations, the ISP tells the contract to ignore nineteen of them; a fat contract forces every consumer to know about methods it never asked for, and any change to them propagates to callers who never invoked them. The DIP lives in the direction of the arrows, and a diagram shows the inversion better than any prose. Before, the use case knows the implementation:



After, both point at the contract:



The implementation's arrow flipped direction: instead of being known by the use case, it now knows the contract. That inversion is what gives the principle its name. Swapping the SQL database for an in-memory implementation in tests, or for a different database in production, becomes a decision the use case never finds out about. Who instantiates `SqlTabRepository` and hands it to `CloseTab`'s constructor? Chapter 9 answers with the **Composition Root**: the single point in the program, usually startup, where concrete implementations get created and wired to whoever depends on them. And why is the repository the only place in the

app allowed to throw and catch infrastructure exceptions?
Chapter 15 closes that boundary. This chapter plants the seed
both of those chapters harvest.

The filter's last two questions. ISP: **does everyone who depends on this contract use all of it?** DIP: **does the use case know the implementation?**

What each principle charges whoever is looking

The filter's five questions measure coupling. There's a sixth lens, which replaces none of them and answers the question that opened the book: what does each principle charge whoever needs to find where a rule lives?

SRP charges the least of all, and that's why it came first. Slicing by actor turns "where's the discount rule?" into "who asked for that rule?", and the second question has an answer outside the code: it was Rosie, in the conversation about loyalty. OCP charges according to whether the extension is real or presumed. When it's real, the new policy is born in a file with a name of its own, and whoever is looking opens that file; when it's presumed, the answer is split between a factory, an interface with one implementer, and an extension point nobody used, and the search goes through all of them. LSP charges on the reading of implementations: a substitute that lies forces whoever is looking to check them one by one, because the contract stopped being a reliable summary of what happens. Where LSP holds, reading the contract is enough.

ISP and DIP charge in the opposite direction, and they charge little. A narrow contract is a short list of the questions that consumer asks, and the method list becomes an index instead of an inventory. A flipped arrow is the guarantee that the rule can be

read without opening the database: whoever looks for the discount calculation finds the use case, and `SqlTabRepository` stays out of the way until the day the question is about writing.

The cost of carrying what doesn't matter

The ISP argument has a second half, which in 2002 wasn't urgent. A fat contract charges the compiler, which propagates changes to whoever didn't ask for them, and it charges whoever reads: twenty methods on screen to find out which of the twenty answers today's question. For a person that's time. For a language model it's a literal budget, because everything it considers at once is measured in **tokens**, the pieces text is cut into before it enters the count, and the budget is finite.

Hence the name the architecture literature settled on: **token efficiency**, the share of what you read that is actually about the question you're answering. A one-method contract about tab persistence scores high for whoever wants to know how the tab gets written, and the same holds for the developer who opened the file at eleven at night. It's the same economy ISP always charged for, now with a unit of measure you can check.

How many things you hold at once

The previous section's arithmetic is about volume. There's another one, about simultaneity, and it has had a name since 1988: **cognitive load**, the number of things someone has to keep in mind at the same time to finish a task. John Sweller showed, studying how people learn, that this capacity is small and that badly organized material spends it before the person even reaches the problem.

Programming is the extreme case. To answer “why did the combo come out at 15%?”, someone has to hold the tab, the loyalty policy, the point where the two meet, and what they’ve already ruled out along the way. Every jump the architecture forces adds an item to that stack, and the stack overflows silently: the person doesn’t announce that they forgot, they conclude wrongly. It’s the same metric that has run through the book since the F12 test, and it’s why “how many jumps to the code that does something?” is a serious question and not nitpicking.

The critique SOLID earned

A principle announced as law accumulates enemies, and SOLID accumulated an entire article’s worth. In 2022, Dan North published the CUPID proposal (Composable, Unix philosophy, Predictable, Idiomatic, Domain-based), and along the way called the SRP a “pointlessly vague principle.” His central argument deserves attention: principles are binary rules, ones you either meet or violate, and North prefers **properties**: gradable qualities that code can have more or less of. That’s the distinction between principle and property running through the whole debate, a binary rule on one side, a continuous scale on the other. Robert Martin answered in “Solid Relevance” (2020, on his blog), where he argues the principles remain valid because the forces they name, coupling and dependency, haven’t aged. Both pieces are published and worth reading: North’s at dannorth.net/blog/cupid-for-joyful-coding, Martin’s at blog.cleancoder.com.

You’ve already seen this book’s position in action throughout the chapter, and now it gets a name: the principles work as a coupling heuristic, and CUPID’s properties work as a success ruler. The filter’s five questions are SOLID in heuristic form: none of them says “violate this and get punished”; all of them say “if

the answer is this, the pain comes from here.” And the result of a good slicing gets measured with North’s ruler: the sliced `Tab` is more predictable, more idiomatic, and more oriented toward the coffee shop’s domain than the monolithic one. The two schools measure different things. Pitting one against the other wastes both.

Here’s my scar from this debate. I inherited a project where the SRP had been read as “a class does one thing” and applied with zeal: more than sixty classes under ten lines each, every `Calculator` paired with a `Validator`, a `Normalizer`, and a `Formatter`, and not one business rule readable start to finish, because every rule crossed six files. It was this page’s `SubtotalCalculator` multiplied by sixty. None of those files had an actor; they had verbs. Undoing it cost weeks. Since then, when someone shows me a slicing, I don’t ask what each class does; I ask who asked for it.

Pitfalls

OCP read as “never edit existing code.” This is the chapter’s most expensive trap. That reading spawns speculative extension hierarchies: interfaces with one implementation, factories for one product, extension points nobody extends, all to avoid touching a file that has tests and would take minutes to edit safely. Chapter 4 already delivered the verdict on presumed capacity: YAGNI. Editing code covered by tests is cheap; maintaining an unused abstraction is expensive and permanent. The OCP pays off when the extension is real and recurring, like the loyalty policy marketing swaps every month, not as insurance against any future edit.

SRP by verb. The nine-line class factory from the previous section. The symptom is slicing without an actor: if you can’t say WHO asks for a change in a freshly created class, it shouldn’t

exist. The test question defuses the trap before the commit.

Contract as ceremony. After seeing the DIP work, the temptation is to create an interface for every class in the app, “for consistency.” An interface with a single consumer and a single implementation that never swaps is the overdone slice, contract edition. In repositories, FOCUS requires the abstraction, because infrastructure changes for its own reasons and tests need the swap; everywhere else in the app, wait for the second implementation to ask for a seat.

Q&A

Isn't the SRP just chapter 2's cohesion with a different name? It's cohesion with an operational test. Cohesion says a file's parts should change together; the SRP says how to test that: count the actors. The ruler is the same, but “how many bosses?” gets answered in a minute, while “is this cohesive?” turns into a meeting.

Should I create an interface for every repository from day one? For repositories, yes, and the reason is concrete: database, network, and filesystem change for their own reasons, and your tests will need an in-memory implementation by the first week; the swap isn't a hypothesis, it's routine. Outside the infrastructure boundary, the third pitfall's rule applies: with no second implementation in sight, the contract can wait.

If my language doesn't even have inheritance, does the LSP tell me anything? It tells you everything. The LSP talks about promises, and promises exist everywhere. Wherever there's a contract and two implementations, there's the question “does the swap surprise the caller?”, and you just watched Go answer it without a single `extends`.

Quick tip

To count a file's actors without guessing, ask the history: `git log --format="%s" -- path/to/file.dart` lists the commit messages that touched the file. If they alternate between “adjust promo discount,” “change receipt layout,” and “migrate database column,” you’ve got three bosses in one file, and the next chapter of pain is already on the calendar.

Tip 6

A good principle is one where you know when NOT to apply it.

Quick reference

Principle	Test question	Use in FOCUS
SRP	How many actors ask for changes in this file?	Use cases per actor
OCP	Does extending require editing what already works?	Lens; extend only when it hurts
LSP	Can I swap the implementation without the caller noticing?	Honor the contract
ISP	Does everyone who	Use case's narrow

	depends on this contract use all of it?	contract
DIP	Does the use case know the implementation?	Dependency injection

Exercises

1. The `OrderRecorder` below is in production at Rosie's counter. It contains TWO violations of principles from this chapter. Name both, fix ONLY the one that causes concrete change pain, and write one sentence justifying why the other one stays as is.

Dart

```
// Stores the logged orders.

class SqlOrderRepository {

  final _orders = <Order>[];

  void save(Order order) {

    _orders.add(order);

  }
}
```

```
}

// Two violations live in this class. Which ones?

class OrderRecorder {

    final _repository = SqlOrderRepository();

    void record(Order order) {

        _repository.save(order);

    }

    String formatPromoCoupon(Order order) {

        return "Come back tomorrow, ${order.customer}: "

            "10% off your next order!";

    }

}
```

Commented answer: the first violation is SRP: recording the order belongs to the counter staff, and the coupon text belongs to marketing, two actors in the same class; marketing changes that text with every campaign, so the pain is concrete and the fix pays off: extract a `CouponFormatter` whose only boss is marketing. The second violation is DIP: the class instantiates `SqlOrderRepository` directly, with no contract in between. It stays: the recorder is the only caller, no second implementation exists or is planned, and no actor has asked for the swap. Fixing it now would be contract as ceremony; the DIP's test question flags the violation, and chapter 4's YAGNI says shelve it until it hurts.

2. Could you create a second policy for the SRP section's calculator (a birthday discount, say) and swap it in for loyalty without editing `TotalCalculator` or `ReceiptFormatter`? If either class needs an edit, one of the OCP and LSP lenses will flag exactly where the design leaked.

Next chapter: you'll write business rules as pure functions, and find out that once a rule depends only on its input, the SRP stops being discipline and becomes a consequence.

Pure Functions and Immutability

In this chapter, you'll:

- classify any function as pure or impure with a one-line test, and justify the call out loud;
- refactor the tab-total calculation by extracting the pure core `calculateTotal(items, customer)`, with a three-line test and no mock;
- write the idiomatic immutable `Item` in your own language, with the copy-with-change move for each of the ten.

The same tab went through Rosie's register twice and printed two totals: \$41.80 on the first call, \$39.52 on the second. No item was added, no item was removed; the code just ran again. You're going to find the culprit, pull a pure function out of it, and leave this chapter with the cheapest, highest-return fix in the whole book.

Chapter 6 closed on a promise: write business rules as pure functions and SRP stops being discipline and starts being a consequence. This chapter pays that promise, in the currency chapter 4 minted: there, simplicity meant deciding what the code does NOT do; here, purity is that same decision applied one function at a time. A pure function doesn't read a singleton, doesn't write to a database, doesn't log, doesn't check the clock. What's left is little. And that little is exactly where the business rule lives, clean enough to test in three lines.

Two totals for the same tab

Friday night at the coffee shop. The clerk rings up table 4's tab: two cappuccinos at \$11.95, a ham and cheese toast at \$8.45, a brownie at \$9.45. The screen shows \$41.80, and the customer asks to split the check. The clerk taps "recalculate," and the same tab, with no other change, answers \$39.52. Rosie charges the lower amount, to be safe, and forwards you the screenshot. Here's the code that answered both calls.

One word before you read on: a **singleton** is the class that exists in exactly one instance in the whole program, reachable from anywhere through a static field. Hold onto the term, because it's the first of three guests.

Dart

```
// The singleton marketing changes whenever it wants.

class DiscountConfig {

    static final instance = DiscountConfig();

    int percentage = 0;

}

class Item {
```



```
Item(this.name, this.priceInCents, this.quantity);

final String name;

int priceInCents;

final int quantity;

}

int calculateTabTotal(List<Item> items) {

    // Sums the tab's items.

    var sum = 0;

    for (final item in items) {

        sum += item.priceInCents * item.quantity;

    }

    // Reads the percentage from the singleton: anyone could have changed it.
```

```

final percentage = DiscountConfig.instance.percentage;

final total = sum - (sum * percentage) ~/ 100;

// Rounds prices for the receipt and MUTATES the list it received.

for (final item in items) {

    item.priceInCents -= item.priceInCents % 10;

}

// Writes the register's log.

print("[register] total calculated: $total");

return total;

}

```

Prices are in cents, following the convention chapter 5 set, and the function has three guests who don't pay rent. It reads the percentage from a singleton that any part of the app can change. It rounds every price down to the nearest multiple of 10 by

writing INTO the list it received, because someone once decided the receipt looked nicer with round prices. And it logs a line. Each guest charges a different kind of pain.

The first pain: the result can't be reproduced. Between the first call and the second, marketing turned on the loyalty promotion and the singleton went from 0% to 5%. Same list, different total, and nothing in the function's signature warns you.

The second pain: the function can't be tested without setup. To write a test you have to prime the singleton with the right value beforehand and, to check the log, capture console output. The test ends up testing the function plus the whole global scenario surrounding it.

The third pain is the sneakiest: call order matters. The first call rounded the prices inside the very list it received. The second call got handed a tab that no longer matches the menu: cappuccino at \$11.90, ham and cheese toast at \$8.40, brownie at \$9.40.

Now the math checks out. On the first call, the faithful sum is 4180 cents and the discount is zero: \$41.80. On the way out, the mutation shaves the prices down and the tab loses 20 cents nobody asked to give up. On the second call, the sum of the already-mutilated list is 4160, the singleton's 5% takes off another 208, and the screen prints \$39.52. Of the \$2.28 difference, \$0.20 came from the mutation and \$2.08 from the singleton. Two independent bugs, invisible to each other, inside the same function body.

Purity is what a function doesn't do

Time to name the test. A **pure function** obeys two clauses: the same input always produces the same output, and nothing happens besides the output. Everything the function does beyond

returning a result is a **side effect**, a term borrowed from pharmacology: the pill treats the headache and, off the label, makes you drowsy. Reading the singleton breaks the first clause, because the output now depends on something that never came in through the front door. Mutating (changing something in place) the list and writing the log break the second, because the world is different after the call than it was before.

Applying the test to the register's method hands you the refactor's whole script for free: the singleton becomes a parameter, the mutation becomes a read, the log becomes a returned value. Here's the result:

 Dart ·  TypeScript ·  Kotlin

```
class Customer {  
  
    const Customer(this.name, {required this.loyaltyActive});  
  
    final String name;  
  
    final bool loyaltyActive;  
  
}  
  
class Total {  
  
    const Total(  

```

```
        this.subtotalInCents,  
  
        this.discountInCents,  
  
        this.totalInCents,  
  
    );  
  
    final int subtotalInCents;  
  
    final int discountInCents;  
  
    final int totalInCents;  
  
}  
  
// The pure core: same input, same output, zero side effects.  
  
Total calculateTotal(List<Item> items, Customer customer) {  
  
    // Sums the items without touching the list it received.  
  
    var subtotal = 0;  
  
    for (final item in items) {
```

```

        subtotal += item.priceInCents * item.quantity;

    }

    // The customer arrives as a parameter; the singleton is dead.

    final discount = customer.loyaltyActive ? (subtotal * 5) ~/ 100 : 0;

    // Returns the result instead of logging it.

    return Total(subtotal, discount, subtotal - discount);

}

```

A word about the three symbols stacked over a single listing: it's written in Dart, and in this chapter's TypeScript and Kotlin sources the function is the same line by line, just spelled differently (the `Customer` and `Total` classes become `type` in TypeScript and `data class` in Kotlin; the integer division `~/` becomes `Math.trunc` and `Int`'s `/`). When the translation is that direct, the book prints a single listing with the symbols stacked, the way chapter 2 set up.

Read the signature first, because now it tells the whole truth: `calculateTotal` takes items and a customer, returns a `Total`, done. The summing loop only reads `item.priceInCents`; no line writes to the list, so the tab that comes in is the tab that goes out. The configuration singleton became the `customer` parameter, and eligibility for the discount travels inside it as `loyaltyActive`;

whoever wants a different percentage tomorrow edits an explicit business rule, not a piece of global state. The log disappeared from the body: instead of printing, the function returns a `Total` with subtotal, discount, and total broken out, and whoever called it decides what to display. Receipt rounding is gone for good, because a receipt is a formatting concern, and chapter 6 already gave you the name of whoever handles that.

Call this function two hundred times with the same tab and the same customer, and it returns the same `Total` two hundred times. The two-totals bug wasn't fixed. It became impossible to write.

Try it: run <https://focus.kodel.com.br/en/dart/07-01> or <https://focus.kodel.com.br/en/ts/07-01>. The snippet calls the impure method twice and then calls `calculateTotal` twice, always with the same tab. Before you run it, write down your prediction for the four numbers. Ours: 4180 and 3952 for the impure pair, 4180 and 4180 for the pure pair.

Swap the call for the returned value

Purity buys you a property with a fancy name and a practical consequence. An expression has **referential transparency** when the call can be swapped for its returned value without changing the program's behavior. The name comes straight from logic: the reference (the call) is transparent because all that sits behind it is the referent (the value). For table 4's tab with loyalty active, writing `calculateTotal(items, ana)` or writing `Total(4180, 209, 3971)` is the exact same thing, at any point in the program, in any order, however many times you like. Try that with `calculateTabTotal` and the program changes: the fixed value doesn't mutate any list and doesn't print any log.

Substituting a call for a value is exactly what a test does: it asserts that the call on the left equals the result on the right. With a pure core, the test shrinks down to three lines:

 Dart ·  TypeScript ·  Kotlin

```
// The test: just data, a call, and a check. No mock, no setup.

void testCalculateTotal() {

    // Arrange: build the input.

    const customer = Customer("Ana", loyaltyActive: true);

    // Act: call the pure core.

    final total = calculateTotal([Item("Cappuccino", 1195, 2)], customer);

    // Assert: check the output.

    assert(total.totalInCents == 2271);

}
```

The comments name the classic test rhythm, *arrange*, *act*, *assert*, and each step fit on one line: build the customer, call the function, check the total. The math checks out: two cappuccinos add up to 2390, the 5% takes off 119, and 2271 is what's left.

There's no **mock**, a stand-in object that takes the place of a real dependency during a test, because there's no dependency to fake. There's no setup or teardown, because there's no state to prepare or clean up. There's no waiting (`async/await`), because there's no I/O (input/output: the program's conversation with disk, network, and screen). Compare that with the impure method's test, which needed to prime the singleton and capture the console. A pure function is the ideal unit of test, and chapter 17 builds the whole pyramid on top of this foundation.

Freeze the data: immutability across ten languages

The pure core promises not to mutate the list it receives, but so far that promise is just good manners: the list is still mutable, and so is the `priceInCents` field on the anti-pattern's `Item`. What's missing is closing the door from the data side. A value has **immutability** when, once built, it never changes again; to “change” an immutable value, you produce a copy with the new field, and that move has a name: **copy-with-change**. The two ideas are the other side of purity (a reminder, in one sentence: a pure function always returns the same output for the same input, with no side effect). Data that can't change turns the function's promise into something the compiler can verify, and that's exactly where the ten languages part ways: each one enforces the promise with a different amount of force.

The far north of that scale is Rust, where immutability isn't an option, it's the default:

Rust

```
#[derive(Clone)]
```

```
struct Item {  
  
    name: String,  
  
    price_in_cents: u32,  
  
    quantity: u32,  
  
}  
  
fn main() {  
  
    let cappuccino = Item {  
  
        name: String::from("Cappuccino"),  
  
        price_in_cents: 1195,  
  
        quantity: 2,  
  
    };  
  
    // cappuccino.price_in_cents = 1095;  
  
    // compiler error: `cappuccino` wasn't declared with `mut`
```

```

// Copy-with-change: struct update syntax.

let promotional = Item {

    price_in_cents: 1095,

    ..cappuccino.clone()

};

println!("{}", cappuccino.price_in_cents);

println!("{}", promotional.price_in_cents);

}

```

Ⓡ Every `let` is immutable until you write `mut`, and copy-with-change is baked into the language's syntax: `..cappuccino.clone()` fills in every field you didn't change. Whoever wants to mutate has to ask for it in writing. This is the one scenario where the language itself guarantees it, and the other nine get measured by how far they land from here.

The next family down enforces it at the field level. Dart, Kotlin, and Swift share the same construct under three different names: the keyword that declares a field which only accepts a value at construction is `final` in Dart, `val` in Kotlin, and `let` in Swift. Once the fields are locked, each one offers its own idiomatic move for

copying. From here on, this chapter's `Item` is the one below, with `priceInCents` frozen; the mutable-field version from the opening was the anti-pattern, and it dies right here:

 Dart ·  Kotlin ·  Swift

```
// The same Item from the opening, rewritten: all three fields lock at
// construction, and the only way to "change" the price is to produce
// another Item.

class Item {

  const Item({

    required this.name,

    required this.priceInCents,

    required this.quantity,

  });

  final String name;

  final int priceInCents;

  final int quantity;
```

```

Item copyWith({int? priceInCents}) {

    return Item(

        name: name,

        priceInCents: priceInCents ?? this.priceInCents,

        quantity: quantity,

    );

}

}

```

 `copyWith` is written by hand (or generated by a package), and the call reads `cappuccino.copyWith(priceInCents: 1095)`: the original stays intact.  A `data class` hands you the same move for free: `val` on the fields and `cappuccino.copy(priceInCents = 1095)`, without writing a single method.  The path is different and the effect is the same: a `struct` copies by value on assignment, so `var promotional = cappuccino` is already the copy, and changing `promotional.priceInCents` never touches the `cappuccino` declared with `let`.

Java and C# solved it with the same keyword, `record`, and only diverge on the copy:

 Java ·  C#

```

record Item(String name, int priceInCents, int quantity) {

```

```

    Item withPrice(int newPriceInCents) {

        return new Item(name, newPriceInCents, quantity);

    }

}

```

☕ Java 21's `record` freezes every field, but copy-with-change is a method you write yourself, like `withPrice` above; the language doesn't generate that move for you. 🐞 C# generates it: the `with` operator produces the copy with the fields swapped, `cappuccino with { PriceInCents = 1095 }`, no hand-written method needed. Two records that look identical, one with a native copy gesture and one with a manual one: that's the only difference that matters between them.

TypeScript and Python make their promise on the type checker's paper, and you need to know that BEFORE you trust the register to either of them:

TS TypeScript · 🐍 Python

```

type Item = {

    readonly name: string;

    readonly priceInCents: number;

    readonly quantity: number;

};

```

```
const cappuccino: Item = {

  name: "Cappuccino",

  priceInCents: 1195,

  quantity: 2,

};

// cappuccino.priceInCents = 1095;

// COMPILER error: Cannot assign to 'priceInCents'

// because it is a read-only property

// Copy-with-change: spread with the new field on top.

const promotional: Item = { ...cappuccino, priceInCents: 1095 };

// The guarantee gets erased along with the types: nothing protects

// you at runtime.

(cappuccino as { priceInCents: number }).priceInCents = 999;
```

```
console.log(`the runtime let it through: ${cappuccino.priceInCents}`);
```

TS `readonly` blocks the assignment at the compiler level, and the spread `{ ...cappuccino, priceInCents: 1095 }` is the idiomatic copy-with-change. Except the types get erased at compile time: the JavaScript that runs in production accepts the write the editor refused, as the last three lines prove. One misplaced `as`, or one piece of data that arrived from outside the type checker, and “immutable” changes.  `@dataclass(frozen=True)` stands one step higher: assigning to `cappuccino.price_in_cents` raises a `FrozenInstanceError` in plain runtime, and the copy comes out through `replace(cappuccino, price_in_cents=1095)`. That extra step, though, isn’t a vault: `object.__setattr__(cappuccino, "price_in_cents", 999)` breaks the freeze with one line. In both languages, `readonly` and `frozen` are contracts between people, enforced by static analysis while you develop (`tsc` in one case, `mypy` in the other), not by the machine that runs the code in production.

Go has no `final`, `readonly`, `frozen`, or `record`. What it has is pass-by-value, and the team’s discipline does the rest:

Go

```
type Item struct {  
  
    Name      string  
  
    PriceInCents int  
  
    Quantity   int  
  
}
```



```
// Receives by value: works on a copy, the original stays intact.
```

```
func withPromotionalPrice(item Item) Item {
```

```
    item.PriceInCents = 1095
```

```
    return item
```

```
}
```

```
// Receives a pointer: touches the original. The signature gives
```

```
// away the intent.
```

```
func lowerPrice(item *Item) {
```

```
    item.PriceInCents = 999
```

```
}
```

 When `Item` travels by value, every function gets its own copy and the caller is protected by construction. When it travels by pointer (`*Item`), or when the field is a slice (Go's version of a "list": a window into an array that lives somewhere else, so copying the struct copies the window, not the data behind it), the protection ends: whoever receives it writes to the original. The compiler doesn't weigh in; the signature is the only warning you get. So here's what the team promises instead: domain types travel by value, a pointer only shows up where mutation is the

declared goal, and a slice inside a domain struct gets copied before it's stored. That's a code-review promise, not a compiler one.

PHP closes out the list with a similar promise and one extra tool:

PHP

```
final class Item

{

    public function __construct(

        public readonly string $name,

        public readonly int $priceInCents,

        public readonly int $quantity,

    ) {

    }

    public function withPrice(int $newPriceInCents): Item

    {

        return new Item($this->name, $newPriceInCents, $this->quantity);

    }

}
```

```
}  
  
}
```

php PHP 8.2's `readonly` locks the property after the constructor runs; any write attempt after that raises a runtime error, and `copy-with-change` is a method that builds a fresh instance, like `withPrice`. The team's promise here is one of coverage: `readonly` on EVERY property of every domain class, no exceptions, because one forgotten mutable property reopens the door the others closed.

Hold onto that contrast, first name and last name: in Rust, the language guarantees it; in Go and PHP, the team promises it. The other six languages live somewhere between those two extremes, and knowing which rung your language stands on decides how much code review immutability is going to cost you. This `Item` and this `Tab`, by the way, aren't disposable: chapter 8 returns errors on top of them, and chapter 14 uses them as input for the use cases.

Try it: run <https://focus.kodel.com.br/en/go/07-02>. The snippet passes the same `Item` to both functions in the listing above. Prediction: the by-value version prints 1195 for the original and 1095 for the copy; the by-pointer version prints 999 for the “original,” because the “copy” never existed.

Functional core, imperative shell

One bill from the refactor is still open: the log and the configuration existed for a reason. The register really does need to record the total in the shop's console, and the loyalty promotion really does need to come from some directory. If purity bans both from living inside the function, where do they go? Into a thin shell wrapped around it:

 Dart ·  TypeScript ·  Kotlin

```
// The imperative shell: reads the world, calls the core, writes the log.

Total closeTab(List<Item> items) {

    final customer = lookupCustomerAtRegister();

    final total = calculateTotal(items, customer);

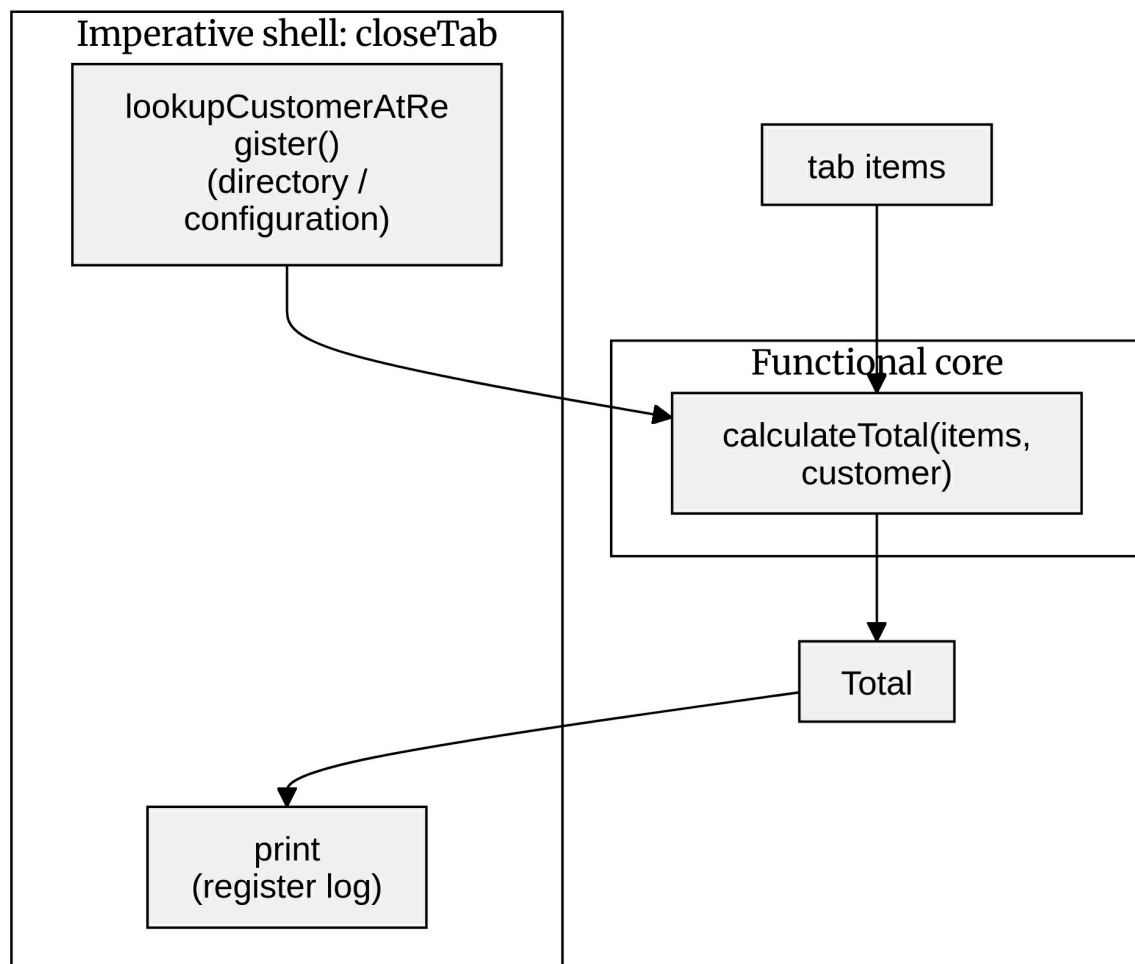
    print("[register] total: ${total.totalInCents}");

    return total;

}
```

Four lines of body, each with one job. The first reads the world: it looks up the customer (and her eligibility) in the register's directory. The second hands everything to the functional core and gets back the `Total`. The third writes the log the impure method used to write, only now from outside the business rule. The fourth returns the `Total` to whoever called it, untouched. Nothing was lost in the refactor; the effects just changed address.

This arrangement has a name and a source. Gary Bernhardt, in the talk “Boundaries” (SCNA, 2012) and the screencast “Functional Core, Imperative Shell” (Destroy All Software, 2012), named the pattern **functional core, imperative shell**: the program’s decisions become pure functions at the center, and a thin, dumb shell of effects wraps around that center to read inputs and dispatch outputs. The tab’s flow looks like this:



The arrows spell out the rule: side effects are born and die only in the shell, and the core just takes values and returns values. Notice the shell doesn’t decide anything; it just ferries things back and forth. If the log turns into a database write tomorrow, or the directory turns into a network call, `calculateTotal` never finds out, and the three-line test keeps passing without touching a mock.

The pattern draws two objections, and both deserve an answer instead of silence.

The first: “a useful program writes to a database and logs; total purity is a fantasy.” The objection gets the fact right and misses the target, because nobody asked for total purity. Bernhardt (2012) asks for something else: that DECISIONS live in pure functions, and that effects live in a shell with no decisions of its own. Rosie’s app keeps writing logs, reading the directory, and charging cards; it just stops mixing those jobs with the total’s arithmetic. A fantasy would be a program with no effects at all; functional core with imperative shell is Friday night with the register closing out right.

The second: “immutability costs performance, copying an object on every change is wasteful.” It does cost something, and I’ll stake out the position: in ten years of business apps, I have never once seen a three-field object’s `copyWith` show up in a **profiler**, the tool that measures where a program actually spends time and memory, instead of where you think it does. I make an exception in exactly two spots: byte buffers (the raw slice of memory where images, audio, and network packets travel) and **hot loops** (the loop that runs millions of times and dominates total runtime, flagged by the profiler). In those two, I mutate inside a function that never lets the mutation leak out. Everywhere else, trading the safety of a frozen value for microseconds nobody ever measured is selling lunch to buy dessert.

Pitfalls

The `sort` that mutates in place. Mainstream languages’ list methods love to edit the very list you called them on and hand back the result as a courtesy. You sort “a copy” and discover the original changed right along with it:

Dart

```
// Pitfall 1: sort orders the list IN PLACE.

final menu = ["Cappuccino", "Brownie", "Cheese bread"];

final sorted = menu..sort();

print(sorted);

print(menu); // the "original" changed too: it's the SAME list
```

The output prints `[Brownie, Cappuccino, Cheese bread]` TWICE: `sorted` and `menu` are the same list wearing two name tags. In Dart, the version that returns a new list is `[...menu]..sort()`, copy first, sort second; in JavaScript, `toSorted()` instead of `sort()`; in Python, `sorted(menu)` instead of `menu.sort()`. When the list really is immutable, the trap at least screams: try `prices.sort()` on a Dart `const` list and the runtime answers `Unsupported operation: Cannot modify an unmodifiable list`.

Two variables, one list. Assigning a collection to another variable doesn't copy anything; it copies the reference:

Dart

```
// Pitfall 2: two variables, one single list.

final table2Tab = [1195, 845];
```

```
final table3Tab = table2Tab;

table3Tab.add(945);

print(table2Tab); // table 3's brownie landed on table 2's bill
```

The output is `[1195, 845, 945]`: table 2 is about to pay for a brownie it never ordered, and `final` didn't stop any of it, because `final` locks the variable, not the contents. The fix is copying at the boundary (`[...table2Tab]`), or, in this chapter's spirit, modeling `Tab` as an immutable type and producing table 3's tab through copy-with-change.

Q&A

Is a function that reads a global constant impure? If the value is truly constant (`const SERVICE_FEE = 10`), the function is still pure: the constant is part of the code, like a literal. Impurity starts when the global can CHANGE between two calls, like the register's singleton.

What about a function that uses the clock or a random draw? `now()` and `random()` break the first clause: same input, different outputs. The fix is the same as the singleton's: the instant and the seed come in as parameters, and whoever checks the clock is the shell.

Where do I find Bernhardt's material? The talk is at destroyallsoftware.com/talks/boundaries and the screencast at destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell. The two together cost under an hour, and the talk alone is worth the price of admission.

Quick tip

Let the analyzer watch immutability for you: in Dart, turn on the `prefer_final_locals` and `prefer_final_fields` lints; in TypeScript, the `prefer-readonly` rule from `typescript-eslint`; in Python, run `mypy`, the tool that actually enforces `frozen`. A promise a machine collects on is a promise the team keeps.

Tip 7

If a function needs setup to be tested, it isn't a function yet: it's a method in disguise.

Quick reference

Language	Who enforces it	Copy-with-change
Rust	the language (default)	<code>Item { price_in_cents: 1095, ..item.clone() }</code>
Dart	compiler: <code>final / const</code>	<code>item.copyWith(priceInCents: 1095)</code>
Kotlin	compiler: <code>val</code>	<code>item.copy(priceInCents = 1095)</code>
Swift	compiler: <code>struct</code> by value	<code>var c = item; c.priceInCents = 1095</code>
Java	compiler: <code>record</code> (Java 21)	your own method: <code>item.withPrice(1095)</code>
C#	compiler: <code>record</code> +	<code>item with { PriceInCents = 1095 }</code>

	<code>init</code>	
TypeScript	type checker: <code>readonly</code>	<code>{ ...item, priceInCents: 1095 }</code>
Python	<code>frozen=True</code> (breakable)	<code>replace(item, price_in_cents=1095)</code>
Go	the team: structs by value	pass by value, return the copy
PHP	runtime: <code>readonly</code> (PHP 8.2)	constructor: <code>\$item->withPrice(1095)</code>

Exercises

1. Classify each of the five functions below as pure or impure, and justify your call by the chapter's test (same input, same output, zero side effects). Watch for the two traps: size isn't the test.

Dart

```
// Function 1

String formatPrice(int cents) {

    return "\${cents ~/ 100}.${(cents % 100).toString().padLeft(2, "0")}";

}
```

```
// Function 2
```

```
int nextTabNumber() => ++_tabCounter;
```

```
// Function 3
```

```
int calculateLoyaltyDiscount(int subtotalInCents, Customer customer) {
```

```
    if (customer.purchasesThisMonth < 5) {
```

```
        return 0;
```

```
    }
```

```
    final int percentage;
```

```
    if (customer.purchasesThisMonth >= 20) {
```

```
        percentage = 15;
```

```
    } else if (customer.purchasesThisMonth >= 10) {
```

```
        percentage = 10;
```

```
    } else {
```

```
        percentage = 5;

    }

    final discount = (subtotalInCents * percentage) ~/ 100;

    const capInCents = 2000;

    return discount > capInCents ? capInCents : discount;

}

// Function 4

void recordSale(Tab tab) {

    _database.add(tab);

}

// Function 5

List<Item> sortByPrice(List<Item> items) {
```

```
items.sort((a, b) => a.priceInCents.compareTo(b.priceInCents));

return items;

}
```

Answer key: function 1 is pure; formatting 4180 returns “\$41.80” today, tomorrow, and in the test, without touching anything. Function 2 is impure even at one line: every call returns a different number and also writes to the global counter, which breaks both clauses at once. Function 3 is pure even at twenty lines: tiers, a cap, and branches, all computed from the parameters alone, same input and same output every time. Function 4 is impure the classic way: it writes to the database (a side effect) and returns void; it exists solely for the effect. Function 5 is the chapter’s trap: it looks like a query, but `sort` reorders the received list in place, and the caller gets back the very same list, now scrambled; impure by argument mutation. If you called function 2 pure for being short, or function 3 impure for being long, reread the test: it never mentions size.

2. Could you swap `calculateTotal`’s flat 5% discount for function 3’s tiered discount, without breaking purity and without changing the `calculateTotal(items, customer)` signature? `Customer` will need to carry `purchasesThisMonth`, and your language’s copy-with-change move settles the model migration in one line.

Next chapter: `calculateTotal` returns a tidy `Total` when everything goes right, but Rosie’s business runs on cases that go wrong: so what does your pure function return when the customer doesn’t exist, the loyalty card has expired, or the tab arrives empty, given that throwing an exception is itself a side effect?

Errors Are Values

In this chapter, you'll:

- model payment failures as the sealed type `PaymentResult` and write the exhaustive switch that consumes each case on screen;
- classify any new failure as an expected error or a programmer defect, with a single yes-or-no question;
- state and apply the FOCUS rule: exceptions belong only at the infrastructure boundary, translated once into a typed `Failure`.

It's Friday night, the customer at table 4 tries to pay \$41.80, and the card gets declined for insufficient funds. Swapping cards would have fixed it; the screen says "Something went wrong" and she leaves without paying for the brownie. You're going to find out the blame doesn't sit with a badly written message: it sits with a failure that crossed the whole program without showing up in a single signature. By the end of this chapter, the failure will be a typed value, and the compiler will bill you for every case you forget to handle.

Chapter 7 ended on an uncomfortable question: a pure function returns a value, always the same one for the same input, but what does it return when the calculation can fail? Throwing an exception is a side effect, and side effects are exactly what we tore out of `calculateTotal`. The answer fits in one sentence: the failure becomes a return value too. Said like that, it sounds like a syntax

trick; what follows is the demonstration that it's the natural way to write the part of the code that makes the most money, the part that goes wrong.

The payment that only said “Something went wrong”

Before the technique, the pain. The tab payment was implemented like this, and code just like it is serving customers right now at thousands of registers out there:



```
class CardDeclinedException implements Exception {  
  
    CardDeclinedException(this.reason);  
  
    final String reason;  
  
}  
  
// The simulated processor: declines any card ending in 7.  
  
String chargeProcessor(int totalInCents, String cardNumber) {  
  
    if (cardNumber.endsWith("7")) {
```

```

        throw CardDeclinedException("insufficient funds");

    }

    return "COMP-4211";

}

void showOnScreen(String message) => print("[screen] $message");

void showError(String message) => print("[screen] $message");

// The signature promises a receipt. Nothing in it mentions failure.

String payTab(int totalInCents, String cardNumber) {

    return chargeProcessor(totalInCents, cardNumber);

}

void main() {

    // The customer at table 4 pays $41.80 with a card ending in 7.

```



```
try {  
  
    final receipt = payTab(4180, "5090 1117");  
  
    showOnScreen("Paid. Receipt $receipt.");  
  
} catch (e) {  
  
    // Any failure lands here and becomes the same message.  
  
    showError("Something went wrong");  
  
}  
  
}
```

Two words in this code need a definition before the critique. An **exception** is an object that interrupts execution at the point of the `throw` and climbs the call stack until something catches it; **try/catch** is the construct that catches it: the `try` fences off the watched block, and the `catch` receives any exception that blows up inside it. The mechanism creates what this book calls **invisible control flow**: an execution path no signature declares. `payTab` promises to return a `String`, and the return type is the only promise the compiler reads; the `throw` three calls down leaves no trace in it.

Now the scene. The customer at table 4 taps her card, the processor answers “insufficient funds,” and that valuable piece of information dies inside the `catch`. The screen shows “Something

went wrong.” She tries again, same message, and walks out thinking Rosie’s Coffee Shop app is broken. In her bag was a second card with plenty of room left. The sale wasn’t lost for lack of money; it was lost for lack of a type.

This failure has three layers, and separating them matters because a different piece of the chapter fixes each one. First: the signature lies. `String payTab(...)` claims that paying always produces a receipt, and whoever reads that signature, human or compiler, has no way to know that a declined card, a dropped connection, and an insufficient loyalty balance all live behind that `String`. Second: the compiler doesn’t collect. Delete the entire `try/catch` and the code still compiles without a single warning; failure handling is optional, and under deadline, everything optional disappears. Third: the message doesn’t help someone who could have helped themselves. The `catch (e)` received an object carrying the exact reason for the decline and flattened it into the most useless sentence in the history of interfaces. Someone who just needed to swap cards got the same warning as someone who’d hit a bug.

Expected error is not a defect

Before fixing the payment, you need a criterion, because not every failure deserves the same fate. The distinction that governs the rest of the book separates **expected error** from **programmer defect**. An expected error is part of the business flow: a declined card, a dropped connection, a loyalty balance that doesn’t cover the redemption. Rosie knows these things happen every single night; the program should know too, and the place where a program keeps knowledge is its type system. A programmer defect is the violation of a premise that should always hold: an index past the end of a list, a null value where null was supposed to be impossible. No business rule at the coffee shop mentions

those cases, because they don't belong to the business; they belong in the *bug tracker*, the system where the team logs and tracks defects.

The criterion fits in one question: is this failure part of the business flow? If it is, it becomes a return value, with its own type and its own data. If it isn't, it's a bug, and a bug should stay an exception: the program crashes early, with a **stack trace**: the list of calls that led up to the offending line, aimed straight at the guilty spot. At the coffee shop, the ruler reads like this:

Failure	Classification	Why
Card declined by the processor	value	staff can offer another card
Index out of range on the item list	exception (bug)	no flow produces this
No connection to the processor	value	happens weekly and the screen reacts
Null field where null was impossible	exception (bug)	violated premise

The question you should be asking: why not catch the bug too and show a friendly screen? Because catching the bug hides the defect. An out-of-range index signals that some earlier calculation is wrong, and the only useful reaction is to crash right there, with the whole stack pointing at the guilty line, ideally in your test environment. A generic `catch` wrapped around the bug trades today's stack trace for corrupted data next week. Crashing early is mercy; the friendly screen is what's actually cruel.

The failure becomes part of the return type

Criterion in hand, fixing the payment starts with the type. The idea has a family name: a **Result** (also called an **Either** in some languages) is a return type that carries either success or failure, one of the two, never both. Instead of returning `String` and throwing the rest out the back door, the function now returns a type that enumerates every possible outcome. Each language offers its own tool for enumerating outcomes, and two of them show up here; the chapter uses four languages in total, and chapter 18 consolidates the same recipe across all ten in the book.

The first tool is a **sealed class**: a closed hierarchy where the declaration itself guarantees the list of subtypes is complete, and the compiler knows the whole list. It's the opposite of ordinary inheritance, open to any subclass in any file. The same concept shows up in other languages as a **union type**: a type declared as the union of named alternatives. Sealed class and union type are two names for the same idea, a closed list of cases, and that closed list is what buys this chapter's central property: **exhaustiveness**, the compiler-backed guarantee that a `switch` over the type handled every case. In Dart, `PaymentResult` looks like this:

Dart

```
sealed class PaymentResult {}

final class PaymentApproved extends PaymentResult {

  PaymentApproved(this.receipt);
```

```
    final String receipt;

}

final class CardDeclined extends PaymentResult {

    CardDeclined(this.reason);

    final String reason;

}

final class NoConnection extends PaymentResult {}

// The signature now tells the truth: paying can approve,

// decline the card, or lose the connection. Nothing else.

PaymentResult payTab(Tab tab, String cardNumber) {

    // Simulated processor: declines cards ending in 7, drops the

    // connection on 9.
```

```
if (cardNumber.endsWith("7")) {

    return CardDeclined("insufficient funds");

}

if (cardNumber.endsWith("9")) {

    return NoConnection();

}

return PaymentApproved("COMP-4211");

}

// The screen consumes every case with a distinct action. No default.

String paymentScreen(PaymentResult result) {

    return switch (result) {

        PaymentApproved(:final receipt) =>

            "Paid. Receipt $receipt.",
```

```

CardDeclined(:final reason) =>

    "Card declined: $reason. Want to try another card?",

NoConnection() => "No connection to the processor. Try again?",

};

}

```

The `Tab` coming in is the list of immutable `Item` values from chapter 7, prices in cents, wrapped in its own type: that's the modeling chapter 7's second pitfall asked for, and whatever version you built works here too. Read the type top to bottom:

`sealed class` declares the closed hierarchy, and the three `final class` declarations are the complete list of outcomes. Notice that each case carries exactly the data the screen is going to need.

`PaymentApproved` carries the receipt, `CardDeclined` carries the reason the processor gave, and `NoConnection` carries nothing, because the only useful response is offering another attempt. No case carries a generic error string; if a case has no useful data, it carries none.

Compare the two signatures, because the whole chapter's difference lives in them. Before: `String payTab(...)`, a false promise. Now: `PaymentResult payTab(...)`, the whole truth. Whoever calls the second version receives a value that's useless until it's opened in a `switch`, and Dart's switch expression demands exhaustiveness: all three cases handled, no `default`, or a compile error. The payoff shows up the next day. Add a new result and every switch that consumes that type breaks right away, and you handle the new case because the compiler won't let it through. The signature stopped lying, and the compiler became the inspector of failure handling. The first two pains from the anti-solution died in this

block. The third died with them: with the decline reason typed and in hand, the screen offers “Want to try another card?” instead of “Something went wrong.”

In TypeScript, the closed list is written as a union (`|`), and exhaustiveness is bought with a three-line function:

TS TypeScript

```
type PaymentApproved = { type: "approved"; receipt: string };

type CardDeclined = { type: "cardDeclined"; reason: string };

type NoConnection = { type: "noConnection" };

type PaymentResult = PaymentApproved | CardDeclined | NoConnection;

// If every case was handled, this default is unreachable and the
// argument arrives with type never. Missed a case? tsc flags it now.

function assertNever(value: never): never {

    throw new Error(`Unhandled case: ${JSON.stringify(value)}`);

}
```



```

// The screen consumes each case; assertNever closes the door.

function paymentScreen(result: PaymentResult): string {

  switch (result.type) {

    case "approved":

      return `Paid. Receipt ${result.receipt}.`;

    case "cardDeclined":

      return `Card declined: ${result.reason}. Want another card?`;

    case "noConnection":

      return "No connection to the processor. Try again?";

    default:

      return assertNever(result);

  }

}

```

TS The `|` in the `PaymentResult` declaration is the union type in its most literal form, and the fixed-value `type` field is what TypeScript calls a discriminated union: the compiler reads `case "approved"` and narrows `result` to the right type inside that branch.

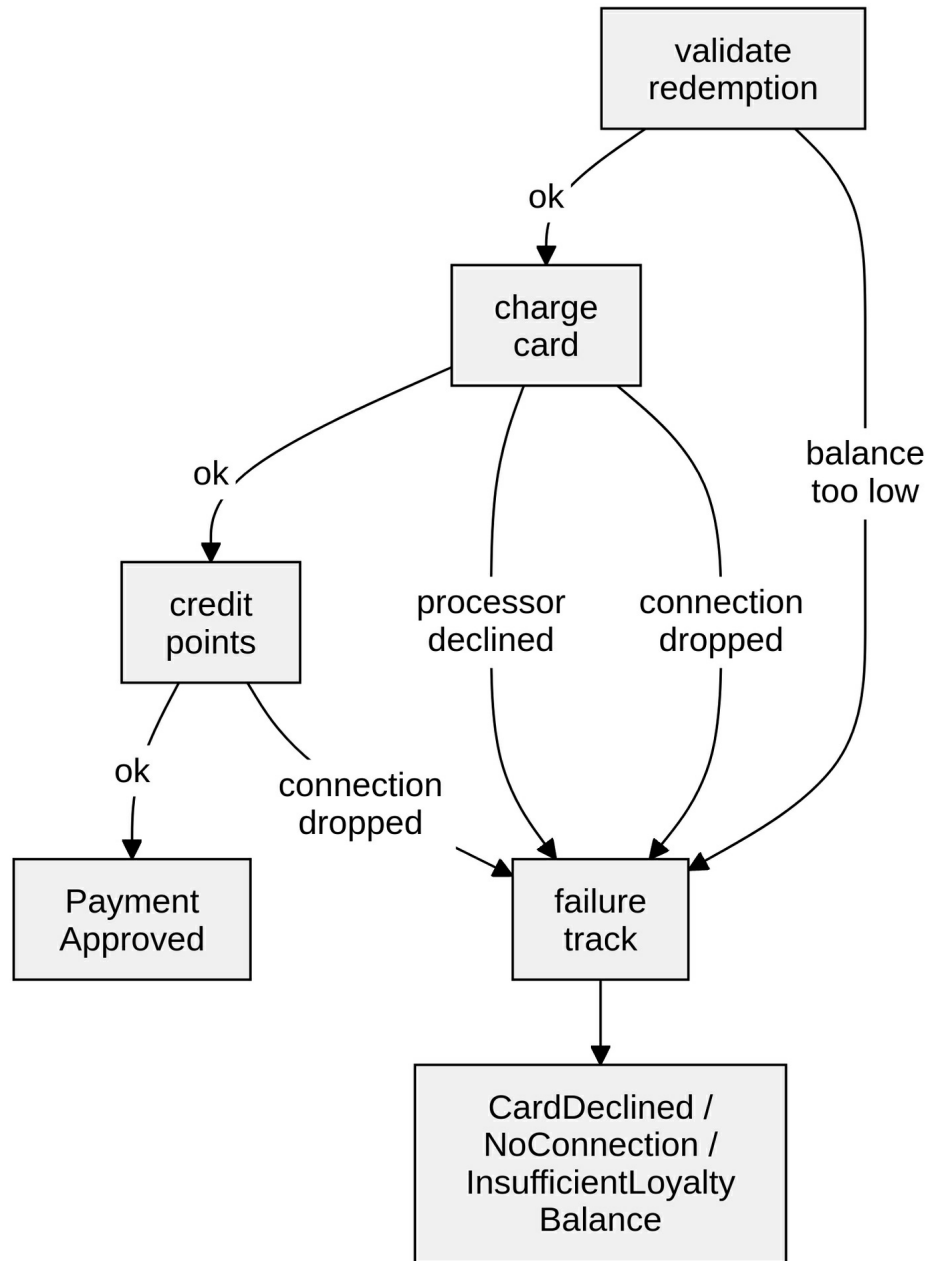
The difference from Dart is in the billing. TypeScript's switch accepts missing cases without complaint, because exhaustiveness here is *opt-in*: the feature only kicks in if you ask for it. The `assertNever` in the `default` is that request. The `never` type is the type with no possible values; if the three `case` branches cover every member of the union, what's left for the `default` is nothing, and `result` arrives there typed as `never`, which matches the parameter. Drop a case, and what's left over stops being nothing: `result` becomes `NoConnection`, the argument no longer fits `never`, and `tsc` flags the line. Three lines of function turn a loose switch into an exhaustive one.

Try it: run <https://focus.kodel.com.br/en/dart/08-01> or <https://focus.kodel.com.br/en/ts/08-01> and delete the no-connection case from the switch (in Dart, the `NoConnection()` line; in TypeScript, the `case "noConnection"` and its `return`). Predicted result: Dart answers `Error: The type 'PaymentResult' is not exhaustively matched by the switch cases since it doesn't match 'NoConnection()'.`, and TypeScript answers `error TS2345: Argument of type 'NoConnection' is not assignable to parameter of type 'never'.` The program doesn't even get to run: the forgotten failure became a compile error.

If your everyday language is C#, Java, PHP, or another of the ten, chapter 18 shows the equivalent of `PaymentResult` in each one, with the exhaustiveness caveats of every compiler. This chapter's idea doesn't depend on syntax: it depends on a closed list of cases the compiler knows about.

Three steps, two rails

Paying a real tab isn't one operation, it's three. The customer wants to redeem 100 loyalty points as a discount, pay the rest by card, and earn points on the new purchase. Each step can fail for its own reason: the point balance might not cover the redemption, the card might get declined, the connection might drop halfway through. Scott Wlaschin named this way of seeing composition **Railway-Oriented Programming**, in a series of articles and talks on fsharpforfunandprofit.com between 2013 and 2014. The image: the flow is a railway with two parallel tracks. The train starts on the success track, and every step is a potential switch; fail, and the train switches to the failure track and rides it straight to the end, and every step still ahead gets skipped.



The diagram calls for a new case in the type. `PaymentResult` was born with three outcomes and now gains a fourth, `InsufficientLoyaltyBalance`, which carries the data the screen needs: how many points were missing. And here's where exhaustiveness sends its first invoice in your favor: the instant that fourth `final class` lands in the file, the screen's switch breaks the build with the same message from the "Try it" above, now

pointing at `InsufficientLoyaltyBalance()`. You don't go hunting for the spots that need to handle the new case; the compiler hands you the list. The composed flow looks like this:

Dart

```
final class InsufficientLoyaltyBalance extends PaymentResult {

  InsufficientLoyaltyBalance(this.missingPoints);

  final int missingPoints;

}

// Paying the tab is three steps, and each one can switch
// to the failure track. The switch is explicit: a return.

PaymentResult payTab(

  Tab tab,

  Customer customer,

  String cardNumber, {

    required int pointsToRedeem,
```

```
}) {  
  
    // Step 1: validate the loyalty point redemption.  
  
    if (customer.loyaltyPoints < pointsToRedeem) {  
  
        return InsufficientLoyaltyBalance(  
  
            pointsToRedeem - customer.loyaltyPoints,  
  
        );  
  
    }  
  
    // Step 2: charge the card through the processor.  
  
    final charge = chargeCard(cardNumber);  
  
    if (charge is! PaymentApproved) {  
  
        return charge;  
  
    }  
  
    // Step 3: credit the points this purchase earned.
```

```

final credit = creditPoints(customer);

if (credit is! PaymentApproved) {

    return credit;

}

return charge;
}

```

Each step produces a `PaymentResult`, and the early `return` is the track switch: if the charge didn't come back approved, that same value (decline or dropped connection) gets returned upward, and step 3 never runs. No pyramid of nested `if`, no stack of `try`; the failure travels through the same channel as success, the return value. One caveat for page length: real payment is asynchronous, and nothing changes when these functions return a `Future` or a `Promise` of the same type; chapters 13 through 15 make that transition at a comfortable pace.

Rust was born with this railway built in, and the leanest version of the same flow shows how much of the ceremony was just the language missing support for it:

Rust

```

enum PaymentFailure {

```

```

    CardDeclined { reason: String },

    NoConnection,

    InsufficientLoyaltyBalance { missing_points: u32 },

}

struct Customer {

    loyalty_points: u32,

}

// Three steps, three ?. Each ? is the switch to the failure track.

fn pay_tab(

    customer: &Customer,

    card_number: &str,

    points_to_redeem: u32,

) -> Result<String, PaymentFailure> {

    validate_redemption(customer, points_to_redeem)?;

```



```

let receipt = charge_card(card_number)?;

credit_points(customer)?;

Ok(receipt)

}

```

 `Result<String, PaymentFailure>` is `Result` as a standard-library citizen: `Ok` carries success, `Err` carries failure, and the failure `enum` is the same closed list Dart wrote with `sealed class`. The novelty is the `?` at the end of every step. It does exactly what the two `if` with `return` did in the Dart version: if the step returned `Err`, return that `Err` to the caller right away; if it returned `Ok`, unwrap the value and keep going. The track switch that's explicit in Dart, three lines per step, is one character in Rust. On the consuming side, Rust's `match` is exhaustive by obligation: a missing arm produces `error[E0004]: non-exhaustive patterns`, a direct cousin of the message Dart showed in the “Try it” above.

Try it: run <https://focus.kodel.com.br/en/rust/08-02> and remove the `Err(PaymentFailure::NoConnection)` arm from the final `match`. Prediction: `error[E0004]: non-exhaustive patterns: `Err(PaymentFailure::NoConnection)` not covered`, before a single line executes.

Go's counterpoint

One of the chapter's four languages has treated errors as values since day one, no sealed class, no union type, no `?`. Go returns errors the most literal way there is, a second return value, and has done that since 2009. Rob Pike made the case in “Errors are values” (the official Go blog, 2015), and the phrase became the name of the principle: **errors-as-values** is the decision to design a language, or a codebase, so that failure is ordinary data, handled like any other, instead of an invisible jump up the stack. The payment flow in Go:

Go

```
// Three steps, three if err != nil. The error is ordinary data,  
  
// visible in the flow, but the compiler doesn't know which errors exist.  
  
func payTab(  
  
    customer Customer,  
  
    cardNumber string,  
  
    pointsToRedeem int,  
  
    ) (string, error) {  
  
    // Step 1: validate the loyalty point redemption.  
  
    if err := validateRedemption(customer, pointsToRedeem); err != nil {  
  
        return "", err
```

```
}
```

```
// Step 2: charge the card through the processor.
```

```
receipt, err := chargeCard(cardNumber)
```

```
if err != nil {
```

```
    return "", err
```

```
}
```

```
// Step 3: credit the points this purchase earned.
```

```
if err := creditPoints(customer); err != nil {
```

```
    return "", err
```

```
}
```

```
return receipt, nil
```

```
}
```

 Credit first, and it's substantial. The `(string, error)` signature doesn't lie: the caller gets the failure right in front of them, on the same `=` that receives the receipt, and the famous `if err != nil` is the track switch written out by hand. No invisible flow; an entire generation of Go programmers never watched an exception cross ten stack frames in silence, and the language's `panic` stayed reserved for what this chapter calls a programmer defect. Go got the principle right before most mainstream languages took Result seriously. Now, the loss: `error` is an open interface, not a closed list. The compiler doesn't know that `chargeCard` can return a decline or a dropped connection, doesn't force anyone to tell the two apart, and quietly accepts `_` in place of `err`. The error is visible, but exhaustiveness doesn't exist: you get the failure track and lose the inspector who checks whether every switch got handled.

Try it: run <https://focus.kodel.com.br/en/go/08-03> and swap one of the error receivers in `main` for `_`. Uncomfortable prediction: it compiles and runs without a single warning, and the ignored failure simply vanishes. It's the "Something went wrong" from the opening section, now without even the message.

Exceptions only at the boundary

One question is still left over from the anti-solution: if an expected error becomes a value, who's still allowed to throw and catch an exception? The FOCUS rule fits in one line: an infrastructure exception exists only at the infrastructure boundary, where it gets translated exactly once into a typed `Failure`; from there inward, only Results circulate through the app. The **infrastructure boundary** is the layer that talks to the

world outside your process: network, database, disk, card processor. It's the only territory where someone else's exceptions are unavoidable, because the libraries living there are the ones throwing them. The `Failure` family is the closed list of domain failures that boundary produces by translating each low-level exception into the vocabulary of the business. The translation looks like this:

Dart

```
try {  
  
    return TabFound(queryServer(table));  
  
} on SocketException {  
  
    return InfraFailure(Failure.noConnection);  
  
}
```

Five lines of translation: the networking library's `SocketException` dies right here, and what goes out to the rest of the app is `Failure.noConnection`, a domain value. This is the only legitimate `try/catch` in Rosie's Coffee Shop app, and it lives in the repository; chapter 15 builds this boundary in detail, with the complete `Failure` family, and chapter 14 shows use cases returning a `Result` from there on up. A programmer defect stays outside this rule: a bug throws an exception, a bug's exception doesn't get caught, and crashing early with a stack trace remains the right answer.

Every pattern that solves a real problem opens the door to two new ways of overdoing it, and this chapter doesn't end without laying out the criticism and the FOCUS answer. The first comes from Wlaschin himself, in the same 2013-14 material that named the rails: "don't take it to extremes." Not every failure in the universe should become a Result case; panics, bugs, and unrecoverable conditions stay out. FOCUS agrees by construction: the classification section exists exactly for that, and a programmer defect is still an exception. The second criticism goes by the nickname *Result hell*: wrapping and unwrapping Results layer after layer, each function converting the error from the layer below into the type of the layer above, until the useful code disappears under the bureaucracy. The FOCUS answer is this chapter's rule: the translation happens exactly once, at the boundary. The repository converts `SocketException` into a `Failure`, and that same `Failure` travels from the use case to the screen without changing clothes on every floor. Anyone re-wrapping the error on every layer isn't following the pattern; they're paying twice for the same insurance. If your code still starts looking like a Result notary office, chapter 19 dissects that overreach as an anti-pattern, with its warning signs.

There's a reason this discipline has gotten more urgent. GitClear, in the 2026 edition of the code-quality survey it publishes at gitclear.com, analyzed hundreds of millions of changed lines and measured a 47% rise in *error masking* (the `catch` that swallows a failure and moves on as if nothing happened) in AI-generated code. The tool synthesizing half your code learned, from our own repositories, that failure hides behind an empty `catch`. Result is the structural antidote: there's no silent `catch` where there's no `catch` at all, and the exhaustive switch won't compile with a swallowed case.

A first-person opinion to close out the section, because this subject leaves scars. Java tried failure exhaustiveness back in 1995 with *checked exceptions*, the ones a method must declare and the caller must handle: `throws` in the signature was mandatory and the compiler collected on it, the same promise this chapter makes. I wrote Java for years and watched that promise rot: since the failure was a jump, not a piece of data, the path of least resistance was an empty `catch (Exception e) {}` just to silence the compiler, and the whole mechanism fell into enough disrepute that Kotlin and C# dropped it on purpose. In my reading, Java's mistake wasn't demanding handling; it was demanding handling for a stack jump. Sealed class with an exhaustive switch gets right what `throws` got wrong, because the failure arrives as a value: you can stash it in a variable, return it, pass it along, and the switch arm that handles it returns something useful to the screen instead of existing only to quiet the inspector.

Pitfalls

The `default` that rebuilds “Something went wrong” with types. The first temptation for anyone coming from try/catch is to close the switch with a catch-all case:



```
// Pitfall: the default swallows the cases you haven't handled yet.
```

```
String paymentScreen(PaymentResult result) {
```

```
    return switch (result) {
```

```
        PaymentApproved(:final receipt) => "Paid. $receipt.",
```

```
        _ => "Something went wrong",  
  
    };  
  
}
```

It compiles, it runs, and it drags the chapter back to square one: the decline reason dies again before it reaches the customer.

Worse, the `_` kills exhaustiveness for good. When

`InsufficientLoyaltyBalance` joins the type, the compiler won't flag this switch, because the wildcard already "handles" the new case. The fix is having no wildcard: every case gets its own arm, and the day the type grows, the compiler hands you the list of screens to update.

The Result thrown back into an exception halfway through.

The second temptation is to receive the `PaymentResult` and throw:

`case CardDeclined: throw PaymentException(...)`. That sends the failure back into invisible flow at exactly the layer where it had just become data, and some screen three frames up is going to need a try/catch to recover what was already typed and in its hand. The boundary rule runs both ways: an exception becomes a value on the way into the domain, and it doesn't turn back into an exception while it's still inside.

Q&A

So I never use try/catch again? You use it, in exactly one place: the infrastructure boundary, where the network, database, and disk libraries throw their own exceptions and your repository translates each one into a typed `Failure`,

exactly once. Outside the boundary, a try/catch around business logic is a sign that an expected error got modeled as an exception.

My language doesn't have a sealed class or a union. Now what? The recipe survives: a base class with known subtypes and a switch covering all of them work in any language with polymorphism; what changes is how much exhaustiveness the compiler checks for you. Chapter 18 shows `PaymentResult` in all ten languages in the book, with each one's degree of enforcement.

Shouldn't `NoConnection` be an exception? The network actually dropped. The `SocketException` exists, but it dies at the boundary. A dropped connection is expected by the business (the coffee shop sits in a basement with basement Wi-Fi), the screen has a useful response for it, and whatever is expected and has a response is a value. Exceptions stay reserved for what should never happen at all.

Quick tip

Let the linter collect on the exhaustiveness your team promises: in TypeScript, turn on typescript-eslint's `switch-exhaustiveness-check` rule and `assertNever` becomes a welcome redundancy; in Dart, the `exhaustive_cases` lint extends the same enforcement to old-style enums. Zero cost, and the inspector starts working in the editor, before the compiler even runs.

Tip 8

If the failure shows up in the signature, it gets handled; if it doesn't, it gets forgotten.

Quick reference

Situation at the coffee shop	Value or exception?	Destination
Processor declined the card	value	<code>CardDeclined(reason)</code> case in the Result
Connection dropped mid-payment	value	<code>NoConnection</code> case in the Result
Point balance doesn't cover the redemption	value	<code>InsufficientLoyaltyBalance</code> case
Index past the end of the list	exception	crash early with a stack trace; it's a bug
Null where null was impossible	exception	crash early; violated premise is a bug
<code>SocketException</code> in the repository	exception	becomes a <code>Failure</code> at the boundary

Exercises

1. Classify each failure below as a return value or an exception, with the justification following the chapter's criterion (is it part of the business flow?). The label alone doesn't count; the justification is the exercise.
 1. The processor declined the customer's card.
 2. `items[5]` on a tab that has 3 items.

3. The app lost its connection while charging the card.
4. The customer asked to redeem 200 points and has 120.

Answer key: failure 1 is a value, because a declined card is register routine and the screen has a useful response (swap cards); it's the `CardDeclined(reason)` case. Failure 2 is an exception: no rule at the coffee shop produces index 5 on a list of 3, so some earlier calculation is wrong, and the program should crash early pointing at the line. Failure 3 is a value: a dropped connection is expected, and a useful response exists (try again); the corresponding `SocketException` dies at the boundary, translated into the `NoConnection` case. Failure 4 is a value, and it's the case the type gained in the rails section:

`InsufficientLoyaltyBalance(missingPoints: 80)`, which carries the data the screen needs to suggest a smaller redemption. If you classified failure 2 as a value "so the app doesn't crash," reread the classification section: catching a bug doesn't fix the bug, it only hides it.

2. Rosie wants to split a tab between two customers. Could you model the `SplitResult` before writing a single line of logic? Enumerate the outcomes (think: the split closes the tab evenly, a rounding cent is left over, one of the two shares comes out to zero) and decide what data each case carries for the screen to act on. If your first draft has an `Error(message: String)` case, it's still "Something went wrong" wearing a new badge.

Next chapter: today's `payTab` was born with the processor baked right into the function body, which is why the listing had to fake the decline with a magic card number; through which door does the real processor, the server that goes down, and every other failing dependency actually enter the function? Chapter 9 answers that question with dependency injection.

Explicit Dependencies: DI and the Composition Root

In this chapter, you'll:

- refactor the payment orchestrator so every dependency it has shows up in the constructor, visible in any signature you read;
- write the Composition Root in main and turn a forgotten dependency, the kind that takes down production today, into a compile error;
- decide, piece by piece, who gets an injected dependency and who gets plain data, following the asymmetry FOCUS embraces on purpose.

Friday, 7pm, and the phone rings: no card goes through at Rosie's Coffee Shop. The afternoon deploy shipped with every test green, the payment code didn't change a single line, and yet the first charge of the night dies with an exception no signature announced. You're about to find the hidden dependency that set this trap, drag it out of hiding, and put the compiler on guard in its place.

Chapter 8 ended with a question hanging in the air: `payTab` was born with the card processor wired straight into the function body, so which door do the real processor, the server that goes down, and the rest of the failing dependencies come through? Failure already turned into a value; what's missing is deciding

who hands the orchestrator the repository and the gateway capable of producing those failures. The answer has two parts, and each fits in one sentence. First: every dependency comes in through the constructor, where any reader can see it. Second: the whole object graph is born in a single place, next to the program's entry point. The rest of this chapter exists so those two sentences stop sounding like bureaucracy and start being the reason you sleep better on a Friday.

Four languages carry this chapter, each with its own job. Dart stays the running example's mother tongue. Kotlin appears stacked alongside it, because a constructor with a field typed by contract is the identical gesture in both, and that equivalence is part of the argument. C# steps in because this chapter's vocabulary (Composition Root, Pure DI) was born in the .NET community, and that's where the difference between the pattern and the tool shows up most sharply. Python closes the chapter as a counterpoint: with no nominal interface, it tests whether the rule survives when the compiler doesn't help. If your language is none of these, follow the Dart track; chapter 10 redoes the recipe in all ten of the book's languages.

The payment that fetched its own dependencies

The pain comes before the technique, as always. Two terms need a definition before the criticism starts. **Dependency Injection**, or DI (Dependency Injection), is the name Martin Fowler coined in 2004, in the article "Inversion of Control Containers and the Dependency Injection pattern" (martinfowler.com), for a specific move: instead of a class fetching its own dependencies, someone outside hands them over ready-made. Fowler coined the term precisely to disambiguate the generic IoC (Inversion of Control), which back then named almost anything. A **Service Locator** is

the alternative he describes in the same article: a global registry, usually a map from type to instance, that any class can ask “give me service X” the moment it needs it.

Rosie’s Coffee Shop’s payment orchestrator was written with the second option, and code just like it runs in thousands of apps right now:

Dart

```
// The locator: a global map from type to instance.

class ServiceLocator {

    static final _instances = <Type, Object>{};

    static void register<T extends Object>(T instance) =>

        _instances[T] = instance;

    static T get<T extends Object>() {

        final instance = _instances[T];

        if (instance == null) {

            throw StateError("no instance registered for $T");
        }
    }
}
```

```
}
```

```
return instance as T;
```

```
}
```

```
}
```

```
class PaymentOrchestrator {
```

```
// The constructor doesn't mention any dependency.
```

```
PaymentOrchestrator();
```

```
PaymentResult pay(int table, String cardNumber) {
```

```
// The dependency shows up here, in the middle of the method.
```

```
final gateway = ServiceLocator.get<PaymentGateway>();
```

```
return gateway.charge(cardNumber);
```

```
}
```

```
}
```

Read the constructor. It says: “I need nothing.” A lie. The `pay` method depends on a `PaymentGateway`, but that information only exists buried in the body, on the `ServiceLocator.get` line. Whoever creates a `PaymentOrchestrator()` has no way of knowing it needs a gateway registered beforehand; the signature, the class’s public contract, hides the requirement.

Now, Friday. Rosie’s Coffee Shop switched card processors, and someone wrote a `NewProcessorGateway`. The class was ready, its test passed, the deploy shipped. Except nobody called `ServiceLocator.register` with the new gateway in the right spot, and no tool caught it, because registration is a line of runtime code the compiler can’t tell apart from any other. The program compiled. The tests passed, and here’s the twisted part: they passed because every test registers its own fake in the global map before it runs, so the orchestrator’s test never exercises the production registration. At 7pm, the first customer of the night tries to pay, `get` can’t find the type, and it blows up:

```
Friday 7pm: Bad state: no instance registered for PaymentGateway
```

That line came from a real run of this chapter’s source, not from imagination. Three pains, then, each with a name. The signature lies: the constructor promises independence and the method demands a global registration. The composition error is a runtime error: wiring the program wrong only shows up once the program is running, at the worst possible hour. And the test requires global registration: every suite has to populate the map before it runs, which couples the tests to each other and, worse, hides the exact oversight that took down production.

Dependencies move up to the constructor

The fix is short and has a christened name: **Constructor Injection**, the form of dependency injection where the class declares everything it needs as constructor parameters and stores the dependencies in immutable fields. No global map. The orchestrator now asks for the two contracts the DIP from chapter 6 required you to create, and notice that it asks for the contract, never the implementation:

 Dart ·  Kotlin

```
// The contracts the DIP from chapter 6 required: one method each.
```

```
abstract interface class PaymentGateway {
```

```
    PaymentResult charge(String cardNumber);
```

```
}
```

```
abstract interface class TabRepository {
```

```
    LookupResult findTab(int table);
```

```
}
```

```
class PaymentOrchestrator {
```

```
// The constructor declares everything the class needs.

PaymentOrchestrator(this.repository, this.gateway);

final TabRepository repository;

final PaymentGateway gateway;

PaymentResult pay(int table, String cardNumber) {

    // Step 1: find the tab; infra failure is already a value (ch. 8).

    final lookup = repository.findTab(table);

    if (lookup is InfraFailure) {

        return NoConnection();

    }

    // Step 2: charge the card at the processor.

    return gateway.charge(cardNumber);
}
```

```
}
```

```
}
```

Line by line. Both contracts have one method each, and that's deliberate: `charge` takes the card number and returns the sealed `PaymentResult` from chapter 8; `findTab` returns the `LookupResult` from the boundary that same chapter built, with the infrastructure failure already translated into a value. A smaller contract doesn't exist. The constructor receives both dependencies and stores them in `final` fields; in Kotlin the gesture is identical, `val` parameters in the primary constructor, which is why the two symbols share a single listing. The body of `pay` is the flow you already know: find the tab, bail out if the infrastructure failed, charge the card. The logic didn't change one bit. Only the origin of the dependencies changed.

Compare the signatures of the two states, because the whole chapter lives in that comparison. Before: `PaymentOrchestrator()`, a promise of independence, a hidden requirement. After: `PaymentOrchestrator(repository, gateway)`. Whoever reads the constructor knows everything the class needs, without opening a single method body. The signature stopped lying. That's what explicit dependency means in this book: not a moral judgment about the code, but the concrete property that the signature declares what the body consumes.

If nobody calls the locator, who builds the graph?

It's a fair question. The locator, for all its flaws, solved a real problem: somewhere, someone has to create the concrete `ProcessorGateway` and hand it to whatever depends on `PaymentGateway`. That “somewhere” now has a name and a fixed address. The **Composition Root** is the one place in the program where concrete classes get instantiated and wired to each other; the term comes from Mark Seemann, in the book *Dependency Injection in .NET* (Manning, 2011; second edition with Steven van Deursen, *Dependency Injection Principles, Practices, and Patterns*, 2019). The set of objects created and wired there is the **object graph**: every object is a node, every dependency is an edge. And the right place for that root is next to the **entry point**, the spot the platform calls to start the program. In Dart, Kotlin, and Python, that's `main`. Three names, one idea: near the entry, and only there, the whole program gets assembled.

Rosie's Coffee Shop's `main` ends up like this:

 **Dart** ·  **Kotlin**

```
void main() {  
  
    // Compose the IO boundary: the concretes are born here, and only  
  
    // here.  
  
    final repository = ServerRepository();  
  
    final gateway = ProcessorGateway();  
  
  
    // Wire the orchestrator to the gateway and the repository.
```

```
final orchestrator = PaymentOrchestrator(repository, gateway);

// The rest of the program just uses the ready graph.

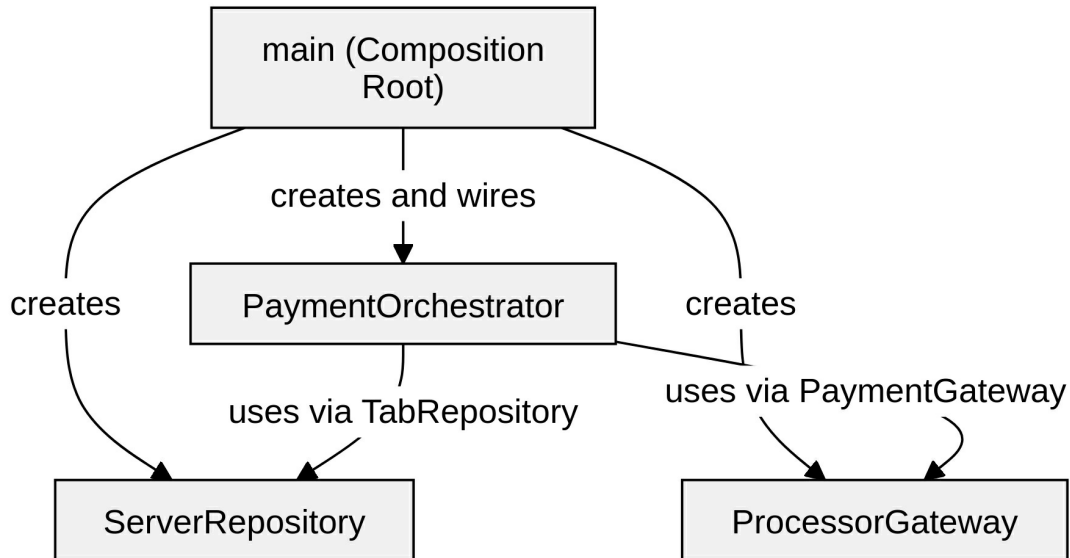
print(paymentScreen(orchestrator.pay(4, "5090 1112")));

print(paymentScreen(orchestrator.pay(4, "5090 1117")));

print(paymentScreen(orchestrator.pay(4, "5090 1119")));

}
```

Six lines of composition. The first two create the concretes at the IO boundary: the simulated repository and the processor that declines any card ending in 7. The third wires everything together: the orchestrator is born already holding both dependencies, complete from its first instant. From there on, the program just uses the graph; no class below `main` creates a dependency, none asks a global registry for anything. Here's the graph, with the creation arrows kept apart from the usage arrows:



There's a gain here that never shows up in the compiler and only gets charged during maintenance. The question “which concrete implementations does this program use?” has, with the composition root, one file for an answer, and it fits on one screen. With the locator, the same question is answered by reading every file that calls the registry, because each of them decides on its own what to fetch, and there is no place where the whole wiring is written down. Whoever joins the team on Monday pays that difference once per dependency; whoever gets the task without ever having opened the project pays it every time, because there is no way to know the other files exist.

Now, time to cash in the lead's promise. Repeat Friday's mistake in this new code: pretend the new gateway arrived and that you, in the rush, forgot to hand it to the orchestrator. Delete the gateway line and its argument in the constructor call. The program doesn't even get to run. Dart's answer, a literal transcript:

```
Error: Too few positional arguments: 2 required, 1 given.  
final orchestrator = PaymentOrchestrator(repository);
```

Kotlin gives the same refusal in a different accent: `error: no value passed for parameter 'gateway'`. The exact same oversight that used to sail through compiler, tests, and deploy to blow up at 7pm on a Friday now dies on your screen, in seconds, and the error points straight at the line. No new test was written for this; the constructor's signature became the spec, and the compiler became the one on call.

Try it: run <https://focus.kodel.com.br/en/dart/09-01> (or your language: <https://focus.kodel.com.br/en/kotlin/09-01>, <https://focus.kodel.com.br/en/csharp/09-01>, <https://focus.kodel.com.br/en/python/09-01>) and delete, inside `main`, the line that creates the `gateway`, along with its argument in the constructor call. Prediction: in Dart, `Error: Too few positional arguments: 2 required, 1 given.`; in Kotlin, `error: no value passed for parameter 'gateway'`; in C#, error `CS7036`. In Python the program does start and stops on the `TypeError` from composition's first line, and the counterpoint section explains why that difference matters.

Pure DI before any container

 Time for C#, and the choice is historical: this chapter's vocabulary was born in the .NET community, in Seemann's book, and .NET is where the confusion between the pattern and the tool shows up the most. There, “doing DI” became synonymous with “using Microsoft's container,” and this chapter exists to undo that fusion. First, the same Rosie's Coffee Shop graph, composed by hand in `Main`, with no library at all; Seemann named this form **Pure DI**, dependency injection in its purest state, just constructors and the composition root:



```
// Pure DI: the whole graph composed by hand, right here in Main.

var repository = new ServerRepository();

var gateway = new ProcessorGateway();

var orchestrator = new PaymentOrchestrator(repository, gateway);

Console.WriteLine(PaymentScreen(orchestrator.Pay(4, "5090 1112")));

Console.WriteLine(PaymentScreen(orchestrator.Pay(4, "5090 1117")));
```

It's Dart's `main` with semicolons in the right accent. Nothing new, and that's the thesis: DI is this, dependencies in the constructor plus a single place of assembly. If you forget the gateway here, the platform's compiler answers with error `CS7036`, "There is no argument given that corresponds to the required parameter 'gateway'", transcribed from a real compile of this chapter's source. Now, and only now, the tool. A **DI container** is a library that assembles the graph for you: you register which concretes implement which contracts, and the container resolves the chain of constructors on its own. The SAME graph, registered in the platform's official container, `Microsoft.Extensions.DependencyInjection`:




```
// The SAME graph, now registered in a DI container.

var services = new ServiceCollection();

services.AddSingleton<ITabRepository, ServerRepository>();

services.AddSingleton<IPaymentGateway, ProcessorGateway>();

services.AddSingleton<PaymentOrchestrator>();

var provider = services.BuildServiceProvider();

var fromContainer = provider.GetRequiredService<PaymentOrchestrator>();

Console.WriteLine(PaymentScreen(fromContainer.Pay(4, "5090 1119")));
```

The three `AddSingleton` lines tell the container what `Main` told it with `new` in the Pure DI version: which concrete serves each contract, and that the orchestrator exists. `BuildServiceProvider` freezes the registration, and `GetRequiredService` asks for the finished orchestrator; the container looks at the constructor, sees the two contracts, finds the registered concretes, and assembles everything. No business class changed. The orchestrator doesn't know whether it came from a `new` or from a container, and that's exactly how it should be: the container lives in the Composition Root and never leaks past it.

My position, so you can calibrate your own: I don't use a container in an app whose graph fits inside a 30-line `main`, and most of the apps that have passed through my hands fit that description. The cost is real (a forgotten registration turns back into a runtime error, as `GetRequiredService` for an unregistered type proves in two seconds) and the benefit, at that size, is zero. A container starts paying for itself once the graph has dozens of nodes and distinct scopes, one instance per request on the server, one per screen in the app; that's when the chain of constructors you'd wire by hand turns into expensive upkeep, and the tool takes over. Even then, notice: the pattern stays identical, dependencies in the constructor, composition at the root. A container is assembly convenience, never a requirement of the pattern.

The Python counterpoint: discipline instead of syntax

 Python takes apart a common excuse: “my language doesn't have interfaces, so DI doesn't apply.” The same graph, without a single nominal contract:

Python

```
class PaymentOrchestrator:

    # The constructor declares everything; there's no nominal

    # interface. Any object with find_tab and charge works

    # (duck typing).
```

```
def __init__(self, repository, gateway):
```

```
    self.repository = repository
```

```
    self.gateway = gateway
```

```
def main():
```

```
    # Compose the IO boundary: the concretes are born here, and only
```

```
    # here.
```

```
    repository = ServerRepository()
```

```
    gateway = ProcessorGateway()
```

```
    # Wire the orchestrator to the gateway and the repository.
```

```
    orchestrator = PaymentOrchestrator(repository, gateway)
```

 This part of the text is language-specific, and three differences deserve attention. The first: there's no `PaymentGateway` declared anywhere. The constructor accepts any object that has `find_tab` and `charge` with the right shapes; it's duck typing as usual, the contract exists, just in the team's heads instead of in a file. If you ever want that contract checkable by a tool, the standard library's `typing.Protocol` declares the same shape without coupling to any implementation, and one sentence about it is

enough for now. The second difference is a trap built into the language itself: every importable Python module is an accidental singleton, because every import of the same module hands back the same module object, with the same state. The temptation to write `gateway = ProcessorGateway()` at the top of a module and import it everywhere is the Service Locator back in slippers. The rule that holds the door is discipline, not syntax: compose everything inside the entry module's `main`, never in the body of an importable module.

The third difference is the one that justifies the discipline. With no compiler, a `main` missing the gateway doesn't turn into a compile error; this chapter's broken source fails like this, a literal transcript:

```
TypeError: PaymentOrchestrator.__init__() missing 1 required
positional argument: 'gateway'
```

Notice when this blows up: at instant zero of the program, on composition's first line, before any customer's request. That's the best a dynamic language can offer, and it beats the alternative by a mile: with a locator, that same oversight would wait for the first charge of the night. Constructor Injection plus Composition Root pull the error toward the cheapest moment the platform allows; with a compiler, before the program runs; without one, at second one of execution.

DI for the boundary, data for the rest

If dependency injection is so good, why didn't `calculateTotal(items, customer)` from chapter 7 get an injected repository? Look at it: the signature is still the same as in that chapter, it takes the list of items and the customer, and returns the total. It didn't change in this chapter, didn't get an interface, didn't enter `main`'s graph. And that wasn't an oversight.

This is the **FOCUS asymmetry**: dependency injection is for the IO boundary, and only for it. Repositories and gateways talk to the server, the database, and the card processor; they fail, they carry latency, they cost money to hit in a test, and that's where an injectable contract pays for itself, because the fake that replaces the processor in tests comes in through the same constructor (chapters 15 and 17 explore both pieces in detail). Use cases are the opposite: pure functions that take data and return data, the way chapter 7 built them. Injecting a repository into a use case would make it impure and tie it to infrastructure; FOCUS prefers the orchestrator to fetch the data at the boundary and hand the function ready-made values. Dependency for whoever touches the world. Data for whoever calculates.

Two serious critiques deserve an answer before the close, because both have serious authors. The first attacks this chapter's villain for going too far. Seemann has argued since 2010 that Service Locator is an anti-pattern, for the three reasons you lived through in the opening: hidden dependency, a lying API, error deferred to runtime. Jimmy Bogard, the creator of MediatR, answered in "Service Locator is not an Anti-Pattern" (jimmybogard.com, 2022) that the conviction generalizes too far: inside composition infrastructure, where a framework needs to resolve types it only learns about at runtime, calling the resolver is legitimate and unavoidable, and the C# listing's own `provider.GetRequiredService` is exactly that. Both are right on their own turf, and the synthesis is operational: resolving a service inside the composition root is part of assembling the graph; outside the

root, never. What Friday condemned wasn't the map's existence, it was the BUSINESS orchestrator asking the map for a dependency in the middle of a method.

The second critique attacks the remedy for ceremony: “you create an interface for every single class just so you can test it, and that’s noise.” I agree with the general diagnosis; the CUPID from chapter 6 already flagged that a single-implementation interface, created by reflex, is dead weight, and Seemann reached the same conclusion by a functional path: he showed that pure dependencies don’t need a contract to be swapped out. FOCUS’s answer is the asymmetry you just saw: an interface only where a real IO boundary exists (`PaymentGateway` , `TabRepository` , and each one earns its rent on the first test with a fake); no interface for use cases, which are pure functions and get tested by calling them with data. No `IDiscountCalculator` . The critique is right about the reflex and wrong about the target: the problem is an interface with no boundary, not the boundary’s interface.

Pitfalls

The first pitfall is the locator with a new badge. It rarely introduces itself as `ServiceLocator` ; it shows up as a `context` , `appState` , or `services` object that the whole codebase receives and that every method reaches into for whatever it wants (`context.gateway` , `context.repository`). The signature says “I take the context,” which is the same as saying nothing, and all three pains from the opening come back intact. The fix is the one this chapter taught: every class declares in its constructor exactly the dependencies it uses, and the grab-bag object dies.

The second is setter injection: build the object empty and hang the dependencies on it later, `orchestrator.gateway = ProcessorGateway()` . Between the constructor and the setter there’s a half-built object

that compiles, gets passed around, and blows up with a null the moment someone uses it too soon; the constructor's signature went back to lying, only now with a window of time attached. If a dependency is mandatory, its place is the constructor, no exceptions.

The third is the dissolving root. The project starts with the graph in `main`, and six months later there's a `new ProcessorGateway()` inside a screen, another in a helper, a third in a test that turned into production code; each one of those `new` calls is a clandestine piece of Composition Root, and swapping the processor now takes a hunt through five files. The symptom is easy to measure: if you need more than one place to swap an implementation, the root has dissolved. Pull the creations back together, next to the entry point.

Q&A

What about get_it, Hilt, or Koin? Isn't the book going to teach me how to use them? No, on purpose. All three are DI containers, and the criterion from the Pure DI section decides for you: small graph, compose it by hand in `main`; graph with dozens of nodes and distinct scopes, adopt whichever container your platform has already blessed. The pattern this chapter taught doesn't change in either case, and learning a container's API takes an afternoon once the dependencies already live in the constructors.

Doesn't a constructor with lots of dependencies turn into a monster? It does, and that's a feature. A constructor asking for eight dependencies is a class confessing it does too much; the locator hid that confession, the constructor prints it. The fix isn't going back to hiding it, it's splitting the class, and the SRP from chapter 6 tells you where to cut.

Do only classes get injection? What about functions? Same idea, different clothes: a function that takes the dependency as a parameter (or a closure that captures it at creation) is Constructor Injection without the word `class`. What matters is the property, visible in the signature, assembled at the root; syntax is just a detail of the language.

Quick tip

Run `grep -rn "ServiceLocator.get\|GetIt.I\|getIt<" lib/` (adjust the names to your project) and look at every result that isn't inside `main`: that list is the map of your code's hidden dependencies, in order of on-call risk.

Tip 9

Explicit dependencies show up in the constructor; hidden dependencies show up on call.

Quick reference

FOCUS piece	Gets DI?	What it gets
View	no	the ready state; no business dependency
Orchestrator	yes	the IO boundary's contracts, through the constructor
Use Case	no	data as

		parameters; stays a pure function
Repository/Gateway	is the endpoint	implements the contract; the concrete is born at the root

Exercises

1. The cash closing below hides the locator in exactly two methods. Migrate it to Constructor Injection and write the Composition Root in `main`; the exercise is in Dart, and it's worth translating to your language of choice before you solve it.

Dart

```
class CashClosing {

  CashClosing();

  String dailySummary(int totalInCents) {

    // First hideout: the clock comes from the global map.

    final clock = ServiceLocator.get<ShopClock>();

    final day = clock.now();
```

```

        return "Closing for ${day.day}/${day.month}: "

            "\${totalInCents / 100}";

    }

    void closeTheDay(int totalInCents) {

        // Second hideout: the printer, too.

        final printer = ServiceLocator.get<Printer>();

        printer.print(dailySummary(totalInCents));

    }

}

```

Check your result against the orchestrator listing: the constructor should declare `ShopClock` and `Printer`, both `get` calls should disappear, and `main` should create and wire all three pieces.

2. Four classes from Rosie's Coffee Shop each need a decision: interface and injection, yes or no? Justify each answer using the FOCUS asymmetry before you look at the answer key. The

classes: `PaymentGateway` , `TabRepository` , the discount use case from chapter 7, and a `ReceiptFormatter` that takes a `PaymentResult` and returns the receipt's text.

Answer key: `PaymentGateway` and `TabRepository` get an interface and enter the graph through injection, because both sit at the IO boundary, fail for real, and earn a fake on the first test. The discount use case is a pure function, it takes `items` and `customer` as data and gets no interface at all; injecting it would be ceremony. `ReceiptFormatter` is the trick question: it looks like a “service,” but it takes a value and returns a value, without touching the world; it's pure logic, tested by calling it, and it also goes without an interface. If you gave it an interface “just in case,” reread the second critique from the asymmetry section.

Next chapter: you're holding the pieces forged since chapter 4: simplicity, contracts, pure functions, failure as a value, and now dependencies composed at the root. Chapter 10 opens Part III by snapping all of them into a design that fits on a single page, the whole FOCUS architecture at once.

FOCUS in One Page

In this chapter, you'll:

- follow the path from a tap on the screen to the updated total, crossing FOCUS's four pieces in the order they talk to each other;
- fill in the canonical responsibility table, which states what each layer does and, above all, what each one forbids;
- justify why the architecture stops at four layers, and reject the fifth by the same criterion that approves the other four.

In chapter 2, you changed one line in the discount calculation and broke three screens: the tab, the register, and the report. The blame wasn't the language's, and it wasn't your carelessness either. It was an address problem: that rule lived in three screens at once, and nothing in the project said where it should live. What was missing has a name, and it fits on one page: a four-piece map that answers, for any line of code you write, which one of them it lives in.

Chapters 4 through 9 delivered loose pieces. The right not to build what nobody asked for (ch. 4). Duplication that's about knowledge, not text (ch. 5). Small contracts and dependencies pointing at abstractions (ch. 6). Pure functions and immutable models (ch. 7). Failure as a value, translated exactly once at the boundary (ch. 8). Constructor injection and an object graph

assembled in a single place (ch. 9). Each one solves a real problem on its own, and none of them answers the question left standing: where do the others go?

Architecture is the agreement that answers that question before you need it. Without the agreement, every developer decides in the heat of the sprint, and the discount rule ends up in three screens again. With it, “where does this go?” has one answer, and it’s the same answer on Monday and two years from now. The four pieces below are FOCUS in full. Each chapter in Part III takes one of them apart; this one shows all four together, working, in the same gesture as always: adding a cappuccino to table 4’s tab.

The handler that did everything

Before the map, the pain. The “add to order” button in Rosie’s Coffee Shop app started small, grew with every order from the counter, and today looks like this, packed into a single method:



```
void onTapAdd(int table, Item item, int points) {  
  
    // Validates the loyalty discount: business rule, right in here.  
  
    var discountPercentage = 0;  
  
    if (points >= 100) {  
  
        discountPercentage = 10;  
    }  
}
```

```
}

// Builds and fires the network call, in here too.

List<Item> items;

try {

    items = api.saveItem(table, item);

} on Exception catch (error) {

    // Handles the infrastructure failure in the middle of the logic.

    totalText = "Failed to add item: $error";

    return;

}

// Recalculates the total and applies the discount to it.

var totalInCents = 0;
```

```

    for (final current in items) {

        totalInCents += current.priceInCents;

    }

    totalInCents -= totalInCents * discountPercentage ~/ 100;

    // Formats the screen text, in here for the fourth time.

    final dollars = (totalInCents / 100).toStringAsFixed(2);

    totalText = "Table $table: \$$dollars";

}

```

Line by line, and notice how each comment marks a different responsibility. The first block, four lines of code, decides how much the customer's loyalty is worth: a hundred points buy ten percent off, and that's a business rule from the coffee shop, written inside a screen method. The second calls the network and catches the exception right there: inside the `catch`, an infrastructure failure turns into interface text. The third recalculates the total, walks the items, and applies the percentage decided above. The last two lines format dollars and cents for the screen label. Four jobs, one method, thirty-four lines.

That code runs. The problem isn't that it's wrong today; it's what it stops you from doing tomorrow. Three concrete roadblocks follow.

The first: you can't test the discount rule. To check that a hundred points buy ten percent off, the test has to build the screen, inject an API, and read the `totalText` field back, then compare a formatted string instead of a number. The rule sits in the `if (points >= 100)` lines, and it has no door of its own.

The second: you can't reuse the rule. The register closes the bill and needs the same discount; the monthly report needs it again. Since the calculation lives inside `onTapAdd`, the fastest way out is copying those lines into the other two spots, and that exact copy is what broke three screens in chapter 2.

The third: the exception leaks into the logic. The `catch` sits in the middle of the method, so a dropped connection turns into a business decision in the same scope where the discount gets computed. Chapter 8 showed that an infrastructure failure should become a value at a single boundary; here it turns into screen text wherever the author was in a hurry.

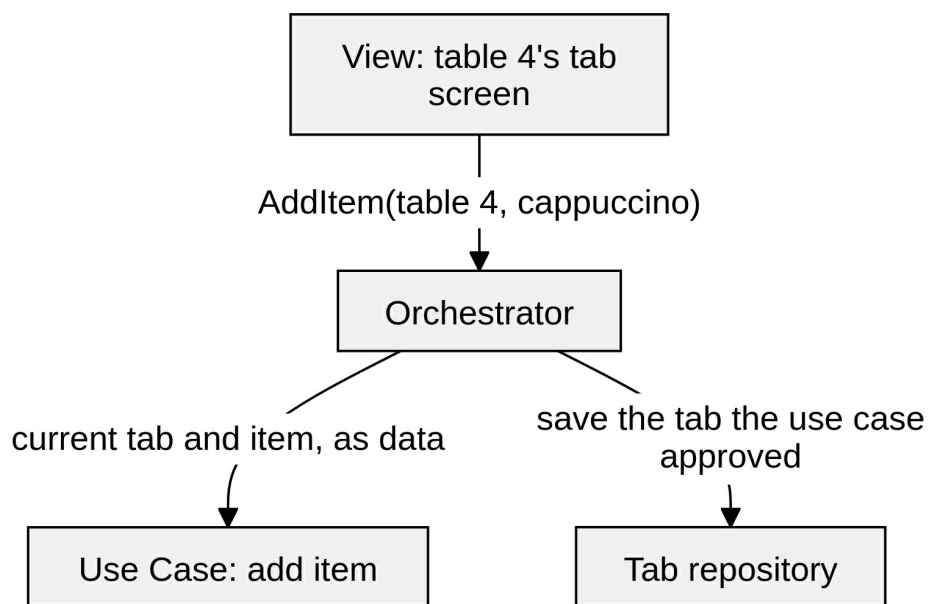
Try it: run <https://focus.kodel.com.br/en/dart/10-02> (or <https://focus.kodel.com.br/en/ts/10-02>) and try exactly one thing: write a test that proves 120 points earn a ten percent discount, without instantiating `TabScreen` and without `CoffeeShopApi`. Prediction: you can't, and that impossibility is this whole chapter's argument.

The path of a tap

Before the new code, the design. FOCUS organizes the program around a fixed direction, and that direction has a name.

Unidirectional flow is the rule that an event travels up one path and state travels down another, with nobody calling back whoever called them: the View announces that something happened, the request crosses the pieces in one direction, and the answer comes back as a data emission, never as a nested callback. Chapter 13 goes deeper into this once it builds the real orchestrator.

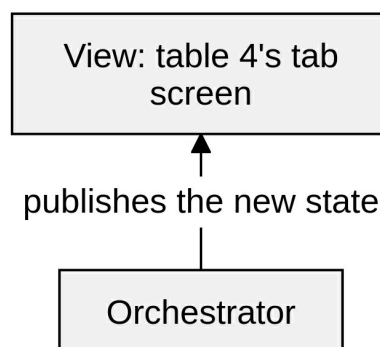
The gesture is the usual one. The customer at table 4 orders a cappuccino, the clerk taps the button, and the event leaves the screen:



Two things in this drawing tend to surprise anyone coming from another architecture. The first is that the use case never talks to the repository: it receives the tab already fetched, as data, and hands back another piece of data. The second is that the orchestrator sits between two arrows instead of one, because it's the one that fetches, calls, and writes.

Calls to the repository are ordinary conversations: you ask for table 4's tab, and it arrives; you send it off to be saved, and you get a confirmation or a failure back. Nothing sits there listening. That matters, because most systems talk to an API or a database built by another team, where nothing like "let me know when it changes" exists, and the architecture can't depend on a feature only some databases offer.

The path back to the screen runs on a different mechanism:



That arrow is an emission, not a call. The orchestrator has no idea the screen exists: it publishes a state, and whoever wants to listen does. That's why the View never has to ask anything, and that's where an app's reactivity, when it has any, actually lives. It belongs to the orchestrator, not to the repository.

The tap becomes an event, the event reaches the orchestrator, the orchestrator asks the repository for the current tab and hands the tab and the item to the use case, the use case returns a new tab with the total recalculated, the orchestrator sends it off to be saved and publishes the state the screen redraws. One full loop, and the screen never called anyone back.

The four pieces in the same gesture

Now the thirty-four-line handler comes apart. No line disappears and no example starts over: each job it does today moves to the piece that has the right to do it.

The View sends the event and draws what comes back

The View is the piece the user touches. It has two verbs and no more: fire the event when something happens, and render the state when it arrives. Nothing migrates here from the naive handler. Not the total calculation, not the dollars-and-cents formatting: the text arrives ready-made, because formatting is deciding how information looks, and no decision belongs to the screen. The orchestrator assembles that text when it publishes the state, and chapter 12 shows the screen that only receives.

What the View sends is an event, and an event carries the bare minimum:



```
// The event that rises from the View. Carries the table and the  
  
// item, and no total: total is the result of a rule, and rules  
  
// belong to the use case.  
  
sealed class TabEvent {}  
  
final class AddItem extends TabEvent {  
  
    AddItem(this.table, this.item);
```

```
final int table;

final Item item;

}
```

TypeScript

```
// The event that rises from the View. Carries the table and the

// item, and no total: total is the result of a rule, and rules

// belong to the use case.

type TabEvent = {

  readonly type: "addItem";

  readonly table: number;

  readonly item: Item;

};
```

Both listings say the same thing by different roads, and the difference is the one chapter 8 already covered. Dart declares a closed family with `sealed class`, and the compiler knows every member of it. TypeScript has no sealed inheritance, so it tags each member with a literal field (`type: "addItem"`) and lets the

compiler narrow the type on that field, a feature called a **discriminated union**. With one lone event the two look like overkill; by the screen's fifth event, that same frame is what stops a `switch` from forgetting a case. Notice what the event doesn't carry: no total, no discount, no formatted text. The View doesn't know how much the cappuccino costs after the discount, and that's exactly why it will never be the reason that calculation changes. Chapter 12 fills this box in with Flutter, React, or whatever your platform uses.

The orchestrator converts event into state

The **orchestrator** is the piece that receives the event, fetches whatever's needed, calls whoever decides, and publishes the resulting state. If the name sounds new, the role doesn't: it's the BLoC or the Cubit in Flutter, the store in Redux or Zustand, the ViewModel on Android. From here on, this book calls the piece the orchestrator and nothing else, because your ecosystem's name changes and the role doesn't; the full mapping lives in chapter 13.

It's worth telling the orchestrator apart from the MVC (Model-View-Controller) controller, the habit you most likely bring with you. The classic controller tends to decide: it validates, applies a rule, picks a path. The orchestrator decides nothing. It sequences. What migrates here from the naive handler is exactly the sequence, that chain of fetching the tab, applying the change, and publishing the result, with none of the decisions that used to sit in the middle.

What it publishes is a state, and the state is a closed value:



```
// The state that flows down to the View. Three cases, one
```

```
// exhaustive switch, the infrastructure failure already translated
```

```
// into a value (ch. 8).
```

```
sealed class TabResult {}
```

```
final class TabUpdated extends TabResult {
```

```
    TabUpdated(this.tab);
```

```
    final Tab tab;
```

```
}
```

```
final class InvalidItem extends TabResult {
```

```
    InvalidItem(this.reason);
```

```
    final String reason;
```

```
}
```

```
final class InfraFailure extends TabResult {
```

```

    InfraFailure(this.reason);

    final String reason;

}

```

Line by line. The first declaration opens the sealed family and has no body, because it exists only to name the set. `TabUpdated` carries the new tab, total already recalculated, and it's the happy case. `InvalidItem` carries the reason as text and covers a rule rejecting something, an item with no price on the menu. `InfraFailure` carries the reason for the technical failure and exists because the network drops. Three cases, and the screen needs to know how to draw all three; the compiler collects on that in the `switch`.

TS TypeScript

```

// The state that flows down to the View. Three cases, one

// exhaustive switch, the infrastructure failure already translated

// into a value (ch. 8).

type TabResult =

  | { readonly type: "tabUpdated"; readonly tab: Tab }

  | { readonly type: "invalidItem"; readonly reason: string }

```

```
| { readonly type: "infraFailure"; readonly reason: string };
```

The TypeScript version fits in four lines because the union is written in one shot, vertical bars separating the cases, instead of one class per case. The difference is syntax and origin: Dart models variants through sealed inheritance, TypeScript through a union of literal types. The practical effect is identical, and that's what matters here. The tab's orchestrator receives the repository dependency through the constructor, the way chapter 9 required, and chapter 13 fills this box in.

The use case is the only one that decides

The **use case** is where the business rule lives, and it's a pure function in chapter 7's sense: same input, same output, no touching network, database, clock, or screen. What migrates here from the naive handler are the four lines of the loyalty discount and the loop that recalculates the total. This is the migration that pays for the whole chapter: those lines were untestable inside the screen, and now they're a function that takes data and returns data.

The signature tells the whole story:

 **Dart** ·  **TypeScript**

```
// The use case's signature. Chapter 14 fills in the body; notice it

// takes data and returns a Result, with no repository along the way.

typedef AddItemToTab = TabResult Function(

  Tab tab,
```



```
Item item,  
  
int loyaltyPoints,  
  
);
```

The three parameters are data, not collaborators: the current tab, the item coming in, and the customer's loyalty points. No repository, so this function can't reach the database even if the author wanted it to. The return is the `TabResult` you just saw, which means a rejected rule is a return value, not a thrown exception. In TypeScript the same contract is a three-argument function returning the same type, an identical gesture, and that's why the two symbols share a single listing. Testing this function means calling it: build a tab, pass 120 points, check the total. No screen, no network, no fake. Chapter 14 fills in the body.

The repository is the only boundary with the world

The **repository** stores and returns data, and it's the only piece that knows what's out there. What migrates here from the naive handler are the two loose responsibilities left over: the network call and the `try/catch` that rides along with it. The exception doesn't stop existing; it stops circulating. It's born here, it dies here, and it leaves converted into a value, exactly as chapter 8 established.

 Dart ·  TypeScript

```
// The repository's contract. Chapter 15 fills this box in with a  
  
// real database and network; here there are only two operations,
```

```

// on demand: find a table's tab and save the whole tab. Neither

// one observes anything; whoever needs the data asks for it on the

// spot and gets the answer back. Both return a Result because the

// exception dies right here, at the boundary, and leaves as a

// value (ch. 8).

abstract interface class TabRepository {

    Future<TabResult> findTab(int table);

    Future<TabResult> save(Tab tab);

}

```

Two methods, and both are ordinary conversations: question and answer. Finding and saving are two of the four operations in **CRUD** (Create, Read, Update, Delete), the basic set of things you do with stored data, and that whole set belongs to the repository. You ask for table 4's tab and it arrives; you send it off to be saved and get a confirmation or a failure back. Neither one sits there listening for a change, and that choice is deliberate. Continuous observation exists in some databases and is great where it exists, but it's a special case: when the data lives in another team's API,

or in a database that only answers queries, there's nothing to observe. The architecture needs to hold up in both scenarios, so the contract stays at the common denominator.

Notice both operations return `TabResult`, and that's on purpose. Reading fails just as much as saving does: the server drops mid-query, the database refuses the connection, the table doesn't exist. If `findTab` returned the raw tab, reading would be the one spot in the program where an infrastructure failure had no way to become a value, and this section's opening sentence would stop being true. The price is one extra check in the orchestrator, which needs to look at what the fetch brought back before calling the use case. It's a fair price for a promise that holds all the way through.

In TypeScript the contract is an `interface` with the same two methods; only `Future` becomes `Promise`. The equivalence is trivial and earns the shared listing. Notice what the contract doesn't have: no method called `applyDiscount`. Chapter 15 fills this box in.

If your screen needs to refresh on its own, that job belongs to the orchestrator, which already publishes state to whoever's listening. That's where a timer, a manual refresh, a WebSocket, or a database's `watch` (where the feature exists) comes in. The repository keeps answering questions, and the View still never knows where the data came from.

Try it: run <https://focus.kodel.com.br/en/dart/10-01> (or <https://focus.kodel.com.br/en/ts/10-01>) and change the discount from ten to twenty percent. Prediction: you'll find the number in exactly one place, inside the pure function, and neither the screen nor the repository needs to be opened. Then run 10-02 and look for the same number there.

The same slice in your language

You just watched the four pieces in Dart and in TypeScript. Nothing they do depends on those two languages, and the proof is the other eight, below. In every listing, look for the same sequence: the event that rises from the View, the state that flows down to it, the use case's signature, and the repository's contract. The names don't change. What changes is how each language writes a closed set of cases, and what it charges you when a case gets forgotten.

All ten implementations are published, they compile, and they print the exact same line:

```
Table 4: $17.06
```

Kotlin writes the slice almost the way Dart does: `sealed class` for both families, `data class` for each case. The difference that matters shows up on the consuming side. A `when` over a sealed class is exhaustive by the compiler's own obligation, so forgetting a case doesn't compile.

Kotlin

```
sealed class TabEvent

data class AddItem(

    val table: Int,
```

```
        val item: Item,

    ) : TabEvent()

sealed class TabResult

data class TabUpdated(val tab: Tab) : TabResult()

data class InvalidItem(val reason: String) : TabResult()

data class InfraFailure(

    val reason: String,

) : TabResult()

typealias AddItemToTab =

    (tab: Tab, item: Item, loyaltyPoints: Int) ->

    TabResult
```

```
interface TabRepository {  
  
    fun findTab(table: Int): TabResult  
  
    fun save(tab: Tab): TabResult  
  
}
```

Swift swaps sealed inheritance for an `enum` with associated values, and the state's three cases fit in three lines inside one single type. The `switch` is exhaustive by obligation too. Notice `findTab`: it returns the standard library's `Result`, because in Swift the generic result type already comes built in and needs no inventing.

Swift

```
struct AddItem {  
  
    let table: Int  
  
    let item: Item  
  
}  
  
enum TabResult {  
  
    case tabUpdated(Tab)
```

```

    case invalidItem(reason: String)

    case infraFailure(reason: String)

}

typealias AddItemToTab =

    (Tab, Item, Int) -> TabResult

protocol TabRepository {

    func findTab(table: Int) async -> Result<Tab, Error>

    func save(tab: Tab) async -> TabResult

}

```

C# writes everything with `record`, and `abstract record` plays the family's role. This is where the first real gap shows up: the hierarchy is closed by convention, not by the compiler, so a `switch` over `TabResult` emits a warning instead of an error when a case is missing, and the code needs a dead arm that never runs.



```
public abstract record TabEvent;
```

```
public sealed record AddItem(int Table, Item Item) : TabEvent;
```

```
public abstract record TabResult;
```

```
public sealed record TabUpdated(Tab Tab)
```

```
    : TabResult;
```

```
public sealed record InvalidItem(string Reason) : TabResult;
```

```
public sealed record InfraFailure(string Reason)
```

```
    : TabResult;
```

```
public delegate TabResult AddItemToTab(
```

```
    Tab tab,
```

```
    Item item,
```



```

        int loyaltyPoints);

    public interface ITabRepository

    {

        Task<TabResult> FindTab(int table);

        Task<TabResult> Save(Tab tab);

    }

```

Java 21 has a genuinely closed set: `sealed interface` for the family, `record` for each case, and a `switch` with patterns the compiler collects on. The use case's signature is the one spot that's a nuisance, because Java has no type alias for a function: it becomes a one-method interface, tagged `@FunctionalInterface`.



```

sealed interface TabEvent {}

record AddItem(int table, Item item) implements TabEvent {}

sealed interface TabResult {}

```

```
record TabUpdated(Tab tab) implements TabResult {}
```

```
record InvalidItem(String reason) implements TabResult {}
```

```
record InfraFailure(String reason)
```

```
    implements TabResult {}
```

```
@FunctionalInterface
```

```
interface AddItemToTab {
```

```
    TabResult apply(
```

```
        Tab tab, Item item, int loyaltyPoints);
```

```
}
```

```
interface TabRepository {
```

```
    TabResult findTab(int table);
```

```
    TabResult save(Tab tab);
```

```
}
```

PHP has no sealed class. The closed set gets written by hand, in the union type of every signature, which is why

`TabUpdated|InvalidItem|InfraFailure` reappears in full on every return type. It works. The price is the usual one: the day a fourth case is born, nobody gets warned, and you go hunting for the unions one by one.

PHP

```
final readonly class AddItem

{

    public function __construct(

        public int $table,

        public Item $item,

    ) {}

}

final readonly class TabUpdated

{
```

```
    public function __construct(public Tab $tab) {}

}

final readonly class InvalidItem

{

    public function __construct(public string $reason) {}

}

final readonly class InfraFailure

{

    public function __construct(public string $reason) {}

}

interface AddItemToTab

{

    public function __invoke(
```

```
        Tab $tab,  
  
        Item $item,  
  
        int $loyaltyPoints,  
  
    ): TabUpdated|InvalidItem|InfraFailure;  
  
}
```

```
interface TabRepository
```

```
{  
  
    public function findTab(  
  
        int $table,  
  
    ): TabUpdated|InvalidItem|InfraFailure;  
  
    public function save(  
  
        Tab $tab,  
  
    ): TabUpdated|InvalidItem|InfraFailure;  
  
}
```

Python closes the set in a single line, the named union `TabResult`, and every `match` over it gets checked against that line by the type checker, never by the interpreter. The models are frozen `dataclass` instances, chapter 7's immutability enforced at runtime. The repository is a `Protocol`: the concrete class inherits nothing, it just needs the methods.

Python

```
@dataclass(frozen=True, slots=True)
```

```
class AddItem:
```

```
    table: int
```

```
    item: Item
```

```
@dataclass(frozen=True, slots=True)
```

```
class TabUpdated:
```

```
    tab: Tab
```

```
@dataclass(frozen=True, slots=True)
```

```
class InvalidItem:

    reason: str


@dataclass(frozen=True, slots=True)

class InfraFailure:

    reason: str


TabResult = TabUpdated | InvalidItem | InfraFailure


AddItemToTab = Callable[[Tab, Item, int], TabResult]


class TabRepository(Protocol):

    async def find_tab(self, table: int) -> TabResult: ...

    async def save(self, tab: Tab) -> TabResult: ...
```

Go is the book's counterpoint, and it's where the design truly changes. There's no union type: the state stops being a single value and becomes the pair `(Tab, error)`, the two failure cases become distinct error types, and the View swaps the exhaustive `switch` for a chain of `errors.As`. Forgetting a case still compiles. Notice `TabData` at the end of the listing: with no union, the value-plus-error pair needs to travel together in a struct to reach the screen intact. The architecture survives; the compiler's safety net doesn't.



Go

```
type AddItem struct {  
  
    Table int  
  
    Item Item  
  
}  
  
type InvalidItem struct{ Reason string }  
  
func (e InvalidItem) Error() string { return e.Reason }  
  
type InfraFailure struct{ Reason string }
```



```
func (e InfraFailure) Error() string { return e.Reason }
```

```
type AddItemToTab func(
```

```
    tab Tab,
```

```
    item Item,
```

```
    loyaltyPoints int,
```

```
) (Tab, error)
```

```
type TabRepository interface {
```

```
    FindTab(table int) (Tab, error)
```

```
    Save(tab Tab) (Tab, error)
```

```
}
```

```
type TabData struct {
```

```
    Tab    Tab
```

```
    Error error
```

```
}
```

Rust sits at the opposite extreme from Go. The algebraic `enum` declares the three cases as a single type, `match` is exhaustive by obligation, and the repository's `Result` is the same `Result` the whole language uses. The immutability chapter 7 asked for is the default here, so there's nothing left to lock down.

Rust

```
struct AddItem {  
  
    table: u32,  
  
    item: Item,  
  
}  
  
enum TabResult {  
  
    TabUpdated(Tab),  
  
    InvalidItem(String),  
  
    InfraFailure(String),  
  
}
```

```

type AddItemToTab = fn(&Tab, &Item, u32) -> TabResult;

trait TabRepository {

    fn find_tab(&self, table: u32) -> Result<Tab, String>;

    fn save(&mut self, tab: Tab) -> Result<Tab, String>;

}

```

None of this makes Go or C# bad languages for FOCUS. It makes them languages where exhaustiveness gets paid for with discipline and review, instead of billed by the compiler. Chapter 18 works through that bill gap by gap, with what to do in each case.

All ten complete slices, orchestrator, use case, and repository filled in, run at the routes below, all under <https://focus.kodel.com.br>:

Language	Route
 Dart	/en/dart/10-01
 TypeScript	/en/ts/10-01
 Java	/en/java/10-01
 C#	/en/csharp/10-01
 Go	/en/go/10-01

 PHP	<code>/en/php/10-01</code>
 Python	<code>/en/python/10-01</code>
 Kotlin	<code>/en/kotlin/10-01</code>
 Swift	<code>/en/swift/10-01</code>
 Rust	<code>/en/rust/10-01</code>

Try it: open your language's route and Go's side by side. Look, in both, for the spot where the loyalty discount gets calculated. Prediction: you'll find it in both in under thirty seconds, and in both it sits inside the use case, alone, with no network nearby.

Why just four

Anyone who's already taken a beating from layered architecture has an objection ready at this point, and it's a fair one. Mozaic Works published the argument in full, in the article "Is Hexagonal Architecture Overengineering?"

(<https://mozaicworks.com/blog/is-hexagonal-architecture-overengineering>): layers turn into folders, folders turn into interfaces with a single implementation, interfaces turn into indirection, and the team ends up writing five files to add one field, with nobody able to point at what got gained. The critique isn't against separating responsibilities. It's against separating for ceremony's sake.

The lineage the critique targets is well known. Alistair Cockburn described Ports and Adapters in 2005: the application talks to the world through ports, and the world adapts to them. Robert C.

Martin distilled the idea in the post “The Clean Architecture,” in 2012, and later in the book *Clean Architecture*, in 2017: concentric rings and the **Dependency Rule**, which states that the code’s dependencies always point inward, toward the business rule, and never outward, toward infrastructure and the screen. That’s the spine of the design you just saw, and chapter 6 already showed its technical half, dependencies pointing at abstractions. Both ideas are good, and neither one closes the subject, because neither says how many layers your coffee shop app needs. They say which direction the dependencies run.

FOCUS’s answer to the critique fits in one sentence: what gets preserved is the dependency rule and the single boundary where an exception becomes a Result, not the ring count in the drawing. If your code respects both with four boxes, four boxes are enough. If someone adds a ring just to look like the picture in the article, that ring is exactly the ceremony Mozaic Works is calling out, and FOCUS agrees with the complaint.

The criterion that settles this is simple to state and uncomfortable to apply: **every layer pays its own way**. A layer only earns its place if it can point at a verifiable gain that would vanish without it, and “organization” and “best practices” aren’t verifiable gains. Apply it to the four. The View pays because, in isolation, it can be swapped out whole (from Flutter to React, from mobile to web) without one rule line changing. The orchestrator pays because it creates a single spot where the screen’s state is born, and that’s what lets you reproduce a screen bug with no network involved. The use case pays the steepest price and gives back the biggest refund: it’s testable with zero infrastructure, and it’s where the calculation that broke three screens in chapter 2 lives. The repository pays because it concentrates in one file the only place in the program where an exception can be born.

Now the fifth layer, the one almost every project ends up proposing: a DTO (Data Transfer Object) mapper between the use case and the repository, meant to translate the domain model into the persistence model. The translation is necessary; nobody disputes that. The question is who owns it.

It belongs to the repository. Look at what the repository knows that nobody else does: the database table's column names, the date format some API returns, the field that came back as a string when it should have been a number, the foreign key the server demands. That's someone else's rule. The database wasn't designed for your tab, and the tax system's API even less so. What comes out of the repository is what the orchestrator and the use case asked for, in the shape they asked for it, because the contract is theirs. Turning one thing into the other is the job of whoever signed both contracts, and only the repository signed the third party's.

This isn't a layer, it's a data transformation. An adapter that converts someone else's rule into ours, and it lives inside the box that already exists. The practical difference shows up the day the server renames a field: with the translation inside the repository, one file changes; with a separate mapping layer, the mapper changes, whoever calls the mapper changes, and both tests change.

That's where the rule behind the table's "forbids" column comes from: the layers above never know the layer below's model. The persistence model belongs to the repository and dies inside it. If your use case imports the class that represents the table row, it just inherited the database's migration calendar, and the pure function you wrote in chapter 7 now depends on an `ALTER TABLE`. That's why the prohibition is written down instead of assumed:

this is the boundary that leaks first, and it leaks with the best of intentions, to “avoid duplication” between two models that only look alike.

At a coffee shop where the tab model and the tab’s database table share the same fields, a separate mapper costs one extra file and a field-by-field copy someone will forget to update. When the two models really do diverge, and in some systems they diverge a lot, the translation grows and earns its own name, file, and test. It stays inside the repository. What changes is the box’s size, not the number of boxes.

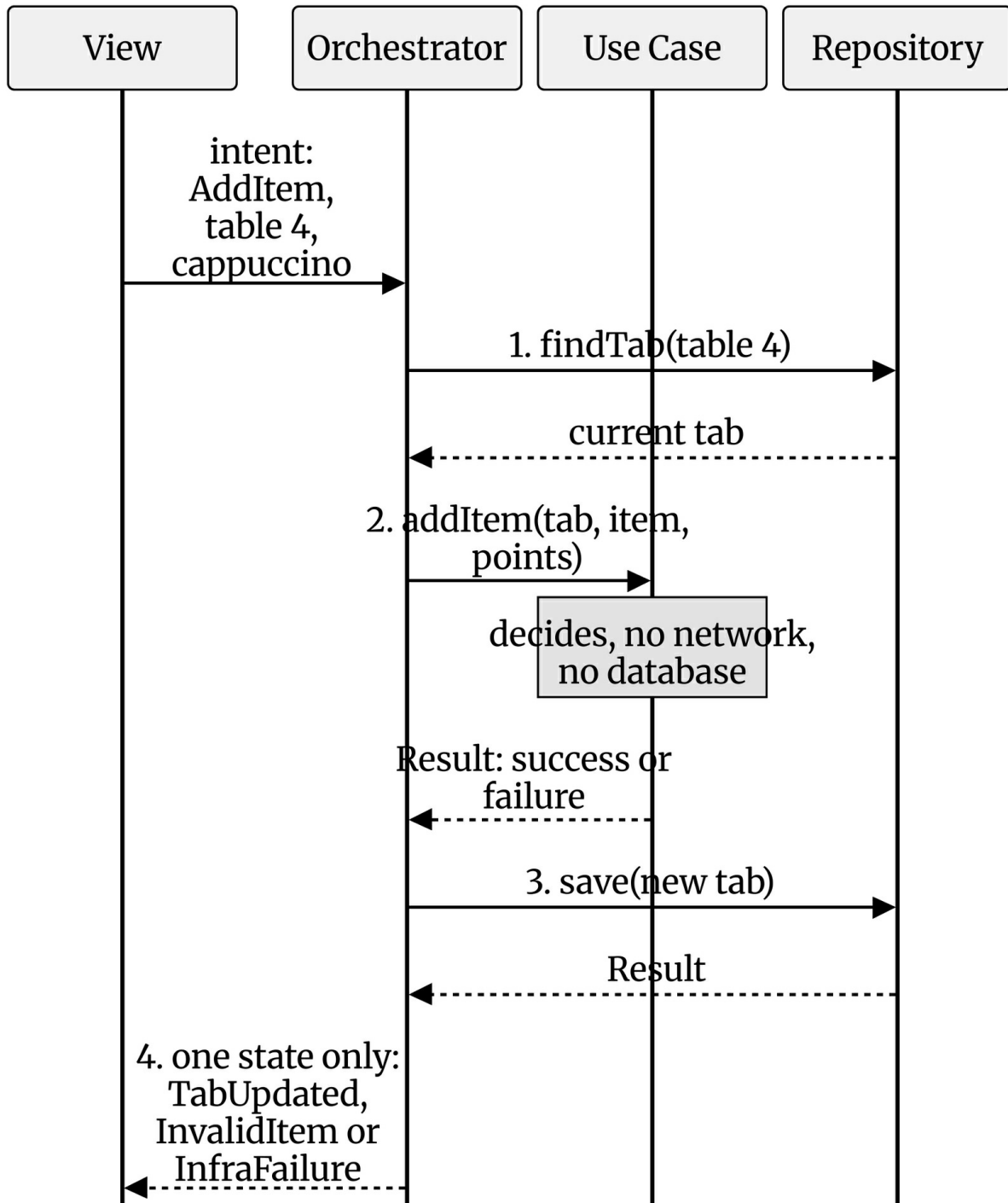
There’s another reason, and it’s the most common one of all: what the screen needs is almost never what the database has to offer. The tab screen wants the item’s name, its price, and the total. The table also stores the date it was added, who rang it up, the shift ID, and a field left over from a 2019 migration. What the view needs is almost always less, and in a different shape: a trimmed-down entity, not everything available. Who defines that contract are the orchestrator and the use case, because they’re the ones consuming what got asked of the repository. The repository fills the order it received; it doesn’t hand over everything the database has and leave the checking to whoever called.

I once worked on a system with seven carefully christened layers. I spent two weeks tracing why a new field never reached the screen and found that five of those seven did nothing beyond receiving an object, building another one with the same values, and passing it along. Nobody had the nerve to remove any of them, because each one had a respectable name and showed up in the diagram the consultancy had delivered. I didn’t remove any either, and that’s exactly why I’m writing this: my rule ever since

is that a layer that can't say what it pays for is a layer that goes, and FOCUS has four because that's as far as I've managed to answer that question.

The recipe the orchestrator follows

The two diagrams from the start showed who talks to whom. What's left is showing the order, which is what you'll reproduce every time you write a new orchestrator. It receives an intent from the View, and from there it follows a four-step recipe, numbered in the diagram: gathers the ingredients the use case is going to need, hands everything over at once, persists only what held up, and publishes one state, always just one. It never tastes the batter along the way.



Notice what leaves the use case and what reaches the View: a single value. Either it worked or it didn't, and both cases travel back through the same path, in the same type. It's chapter 8's Result doing its job right here: the View doesn't ask "did it fail?" before it draws; it draws whatever case arrived. There's no

intermediate state sneaking out the side, no exception climbing outside the flow, and no second channel where the failure travels. One intent goes in, one Result comes out.

That's why the orchestrator is the only piece that talks to two others. It collects from the repository because the use case has no right to, and it calls the use case because the decision isn't its own. The recipe stays the same every time, which is why chapter 13 can turn it into code you copy from feature to feature.

Want to test business rules? Test the use cases. They're the *units* of unit testing. Want to test integration? Test the orchestrator. It's the piece of code that defines one action, from intent to the database or the API. Testing the View gets a lot simpler too, because all you need is firing the intent (the event) and checking how the result gets drawn.

Pitfalls

The first shows up the following Monday, and almost always with the same line: "it was just an `if`." A last-minute request comes in, the tab needs to reject an item once the table has already closed out, and the closest spot to the keyboard is the orchestrator, which is already sitting there sequencing things. The symptom is an orchestrator that grows while the use case stays small; the rule's test goes back to needing a repository fake. The way out is mechanical: if the line decides something about the business, it goes down to the use case, even if it's three lines and even if the deadline is real.

The second is the View that reads the repository directly, "just to show a counter." It looks harmless, because it's reading, not writing. The symptom shows up weeks later, when the counter shows a different number from the rest of the screen, because

now there are two sources of state and nobody keeps them in sync. The way out is having the counter born from the same state as the rest of the screen, published by the orchestrator, even if that costs one extra field in the state.

The third is the hardest to spot, because the code looks clean: the use case that takes the repository instead of data. The signature turns into `addItem(TabRepository repo, int table, Item item)` and everything looks fine, except now the function fetches, decides, and saves. The symptom is the test that goes back to needing a fake and the function that can now fail from a network error. It's an orchestrator wearing a use-case costume, and the way out is handing the fetch back to whoever holds that right: the use case always receives the tab already ready.

Q&A

What about when the use case has no rule at all? On a plain CRUD screen, it sits empty. It sits nearly empty, yes, and the layer stays. The real cost is one signature and one line that returns the validated data, and the payoff is that the day the first rule shows up (and on a real system, it does), there's an obvious place for it, instead of a debate. I won't pretend that cost is zero: on a screen that just manages menu categories, this layer is bureaucracy for a few weeks. The bet is that the software's lifespan runs longer than a few weeks.

What's the practical difference between the orchestrator and my controller? The word "decides." Most frameworks' controllers validate, apply a rule, and pick a path, all inside themselves. The orchestrator only sequences: fetch, call, publish. If you open your orchestrator and find an `if` that talks about the business, it just turned into a controller.

Can a screen have more than one use case? It can, and it will. The tab screen adds an item, removes an item, applies a discount, and closes the bill, and each one of those is a use case with its own signature. The screen's orchestrator knows all four; none of the four knows the others.

Quick tip

Before you write the line, say its verb out loud. "Draws" goes to the View, "sequences" and "formats" go to the orchestrator, "decides" goes to the use case, "stores" goes to the repository. A verb that doesn't fit any of the four is usually two lines wearing one line's clothes.

Quick reference

The table below has two columns instead of one because a dependency rule is easy to promise on paper and hard to collect on in code review. The "does" column is the promise; the "forbids" column is what turns the promise into something a reviewer can point at on screen, with no debate about style. "This `if` decides whether the discount applies, and it sits in the orchestrator" is a checkable sentence. "This code seems a bit coupled" isn't.

Layer	Does	Forbids
View	fires events	business rules
	renders state	data access
Orchestrator	converts event to state	deciding

		rules
	fetches data from the repository	persisting
	calls use cases	
	publishes state	
Use Case	the only place for business rules	IO
	is a pure function	framework
	takes data, returns a Result	domain exception
Repository	CRUD (fetch and save)	business rules
	the only place an infra exception becomes a Result	

This table is the contract for the rest of the book, and chapters 11 through 17 cite its lines the way someone cites a statute. No line from the naive handler got thrown out along the way here: thirty-four lines in one method became four pieces with four responsibilities, and the output stays the same, [Table 4: \\$17.06](#) .

Notice what that half page does to reading. It answers where a rule may live and where it may not, without opening a single file. The property has a name: **semantic compression**, a design's capacity to fit into a short description that still serves for deciding, and not only for describing. For a newcomer, the table's eleven lines stand in for reading the naive handler's thirty-four, and they keep standing when that code changes, because what they record is each piece's intent.

It's this book's thesis in action, and this is the chapter where it can be said with all four pieces already on the table: architecture lowers the cost of change because it makes the intent of the system recoverable, navigable and predictable for humans and for models. Recoverable is finding the discount rule from the table, knowing nothing about the project. Navigable is landing on it in one F12 jump. Predictable is knowing, before opening the file, that it isn't in the View.

Exercises

1. Fill in the table below unaided, without looking back at "Quick reference." The eight cells are the contract the next seven chapters cite, and it's worth rebuilding them wrong now and checking your answer, rather than just recognizing them when they show up later.

Layer	Does	Forbids
View		
Orchestrator		
Use Case		
Repository		

2. Take the naive handler's first three lines (the ones that decide the discount percentage) and say which layer each one lives in. Then do the same with the line `totalText = "Table $table: \\\$dollars";` and with the `catch` line. Answer key: the first three are use case, because they decide a rule; the text line is orchestrator, because assembling the state's text is a decision,

and the View gets the text ready-made; the `catch` line is repository, because that's the boundary where the exception is born.

3. Could you place, across the four layers, the flow for closing table 4's tab and splitting it three ways between customers? Start with the event the screen fires, decide what it carries, and write the use case's signature before anything else. If the signature needs the repository, go back and reread the third pitfall.

Tip 10

If you don't know which layer the code belongs in, it isn't ready to be written yet.

Next chapter: you'll find out that the folder called `views/`, with every screen in the app inside it, is usually the first place where this map gets betrayed.

Features, Not Layers

In this chapter, you'll:

- sketch Rosie's Coffee Shop's folder tree from memory, with the five features and each slice's four pieces in place;
- point, for any requested change, to the exact folder where the diff lands, before you open the editor;
- decide with an explicit criterion whether code should move up to `shared/`, and refuse the promotion when reuse is still a bet.

Chapter 10 handed you a four-piece map with a signed contract, and left you with a loaded question: which folder does each piece live in? You might think the answer is cosmetic, the kind of thing people argue about in a meeting over folder names. It's the decision that sets the blast radius of every request Rosie makes from here to the end of the project. This chapter shows the organization that looks natural and charges dearly for it, the principle that replaces it, and the directory tree the rest of the book lives in.

Chapter 10's canonical table says what each piece does and forbids, but not where it lives. The path to that answer has three stops. First, the coffee shop organized the way most projects are born, which takes a real request and spreads the damage. Then the principle that explains why it hurt, with a name, a source, and

a diagram. Finally, the same coffee shop refactored, folder by folder, with chapter 10's table getting a disk address and one deliberate absence you'll notice before I explain it.

The change that touched four folders

Rosie's Coffee Shop has five features: menu, tab, inventory, payment, and loyalty. Organized the way almost every project starts out, the folder criterion is the file's technical type:

```
views/          menu/  tab/  inventory/  payment/  loyalty/
controllers/    menu_controller  tab_controller  inventory_controller  ...
services/       menu_service  tab_service  inventory_service  ...
models/         menu  tab  inventory  payment  loyalty
```

Each folder holds one kind of file, and each kind holds all five features mixed together. It looks organized, and it is: by the wrong criterion, as Rosie is about to demonstrate without meaning to.

Her request lands on a Tuesday: “the customers at the table want to split the tab.” One feature, one sentence. The diff (the line-by-line difference between two versions of a file) that fills the request:

```
--- a/models/tab.dart

+++ b/models/tab.dart

@@ class Tab
```

```
+ // The "split tab" change starts here: the tab starts tracking
```

```
+ final int people;
```

```
--- a/services/tab_service.dart
```

```
+++ b/services/tab_service.dart
```

```
@@ class TabService
```

```
+ int valuePerPerson(Tab tab, int people) {
```

```
--- a/controllers/tab_controller.dart
```

```
+++ b/controllers/tab_controller.dart
```

```
@@ class TabController
```

```
+ int splitTab(int table, int people) {
```

```
--- a/views/tab/tab_view.dart
```

```
+++ b/views/tab/tab_view.dart
```

```
@@ class TabView
```

```
+ String renderSplit(Tab tab, int people) {
```

Four folders for one sentence from Rosie. The model gained a field, the service gained the rule, the controller gained the pass-through, and the view gained the button. None of these edits is large; the problem is where they landed.

The first pain shows up in review. Whoever reviews this pull request navigates four folders to understand a single intent, and between the `+ final int people;` of the model and the `renderSplit` of the view sit dozens of menu, inventory, and loyalty files that have nothing to do with the change, but live along the way.

The second pain shows up in the merge (folding two lines of work into the same file). While you were editing `services/tab_service.dart`, your teammate was adding this week's promo to `services/menu_service.dart`, in the same folder. Both pull requests touch `services/`, both compete for the same neighborhood, and the merge conflict is born between two features that don't know each other.

The third pain doesn't show up in any tool, and it's the worst one: no folder has an owner. Who's responsible for `services/`? Everyone, because every feature has a file in there. A folder that belongs to everyone belongs to no one, and the question "who owns the tab?" has no answer on disk.

The axis of change

What hurt on Tuesday has a name. **Axis of change** is the criterion stating that things that change together should live together: you organize code by what changes in the same request, not by what looks alike technically. The tab model looks like the inventory

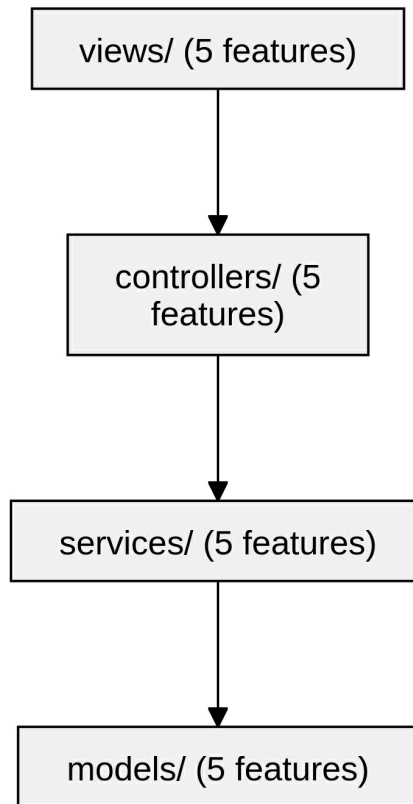
model, both are data classes; but the tab model changes together with the tab view, and never together with inventory. The resemblance is in shape; the change is in business.

Applied to directories, the axis of change produces the **vertical slice**: the system's cut that contains everything a feature needs, from view to repository. The cut crosses the layers top to bottom instead of lying flat over one of them. Jimmy Bogard gave this organization a name and an argument in "Vertical Slice Architecture" (2018): minimize coupling between slices, maximize cohesion within each one. The acronym VSA you'll run into elsewhere is exactly this: Vertical Slice Architecture.

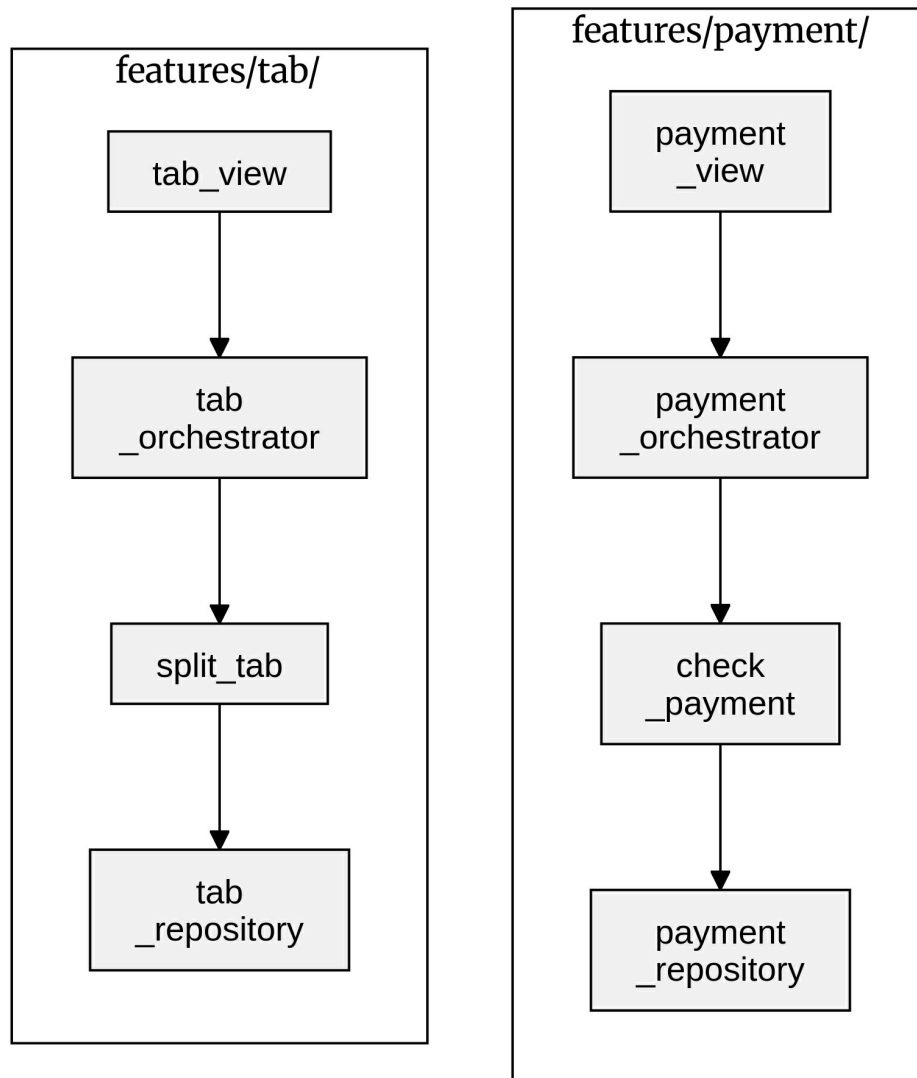
The third name is the visible symptom of the other two. **Feature-folder** is the per-feature folder: `features/tab/` with everything about the tab inside it. Keep the hierarchy among the three names in mind: it settles a critique further ahead, the principle is the axis of change, the cut is the vertical slice, and the folder is only the symptom. Whoever copies the folder without the principle carries the name and drops the benefit.

The two diagrams below compare the two worlds. The first one is the layered organization: the tab change crosses all four folders, and every folder it crosses is shared with the other features.

Layered: the change crosses everyone's folders



The second one is the same change in slices: every arrow is born and dies inside its own folder.



The first one has four stacked boxes, one per layer, and each box announces it serves all five features at once; the arrow running from `views/` down to `models/` is the “split tab” change crossing shared territory. The second one has one box per feature, and the arrows link neighboring files in the same folder, without leaving it. It’s the same change; what changes is how many fences it jumps.

The coffee shop in slices

No new project here: the tree below is the previous section's coffee shop, refactored. The same files, relocated by the axis of change:

```
features/  
├── menu/  
│   ├── check_menu  menu_orchestrator  
│   └── menu_repository  menu_view  
├── tab/  
│   ├── split_tab  
│   ├── tab  
│   ├── tab_orchestrator  
│   ├── tab_repository  
│   └── tab_view  
├── inventory/  
│   ├── check_inventory  inventory_orchestrator  
│   └── inventory_repository  inventory_view  
├── payment/  
│   ├── check_payment  payment_orchestrator  
│   └── payment_repository  payment_view  
└── loyalty/  
    ├── check_loyalty  loyalty_orchestrator  
    └── loyalty_repository  loyalty_view
```

Two levels, and the second one is the file list in alphabetical order, the way your terminal prints it. The tab slice is open in full; in the other four the files sit side by side just to save lines.

The tree is deliberately neutral: with no file extension, it holds equally for Dart, TypeScript, Java, or any of the book's ten languages, and the names are about business and role, never about framework. What varies from one language to the next is the identifier's case, and each one follows what it already uses in file names: `tab_orchestrator` in Dart, TypeScript, Go, Rust, and Python; `TabOrchestrator` in C#, Java, Kotlin, Swift, and PHP. It's the same tree with the local convention.

Notice, too, what the tree doesn't have: no `shared/` folder, no `utils/`, no place "for whatever's left over." The absence is a decision, not an oversight, and two sections from now you'll see the criterion that decides when that folder earns the right to exist.

Notice also what it doesn't have on the inside: no subfolder. The four pieces of chapter 10's canonical table are all in there, but as a filename suffix, not as a folder. The table never described folders; it describes the four pieces that exist inside each slice, repeated in every feature:

Layer	Does	Forbids
View	fires events	business rules
	renders state	data access
Orchestrator	converts event to state	deciding rules
	fetches data from the repository	persisting
	calls use cases	
	publishes state	
Use Case	the only place for business rules	IO
	is a pure function	framework
	takes data, returns a Result	domain exception
Repository	CRUD (fetch and save)	business rules

	the only place an infra exception becomes a Result	

Inside `features/tab/`, each row has an address, and each address has a chapter that fills it in:

Table row	Path in the slice	Who fills it in
View	<code>features/tab/tab_view</code>	chapter 12
Orchestrator	<code>features/tab/tab_orchestrator</code>	chapter 13
Use Case	<code>features/tab/split_tab</code>	chapter 14
Repository	<code>features/tab/tab_repository</code>	chapter 15

Three of the four names follow the same formula, entity first and role as a suffix, and that formula is what replaces the folder: `tab_view` says what `view/tab_view` said, with one segment less and the same information. The fourth breaks the formula on purpose. The use case is called `split_tab`, a verb, not `tab_usecase`: every business rule turns into a file named after what it does, and the slice grows one file per rule. This chapter shows only signatures and names; the inside of each piece is chapters 12 through 15's business, one per row, in the table's order.

There's a fifth file missing, one that isn't a table piece at all: the domain model. `Tab` and `Item` are the data the four pieces trade with each other, so they live in the slice that defines them, in `features/tab/tab`, next to the repository that reads and writes them. That's the only file in the slice with no role suffix, and the absence is the mark: the bare name is the data. Don't confuse this model with the persistence model, the one that mirrors the database table's columns: chapter 10 already sent the persistence

model to be born and die inside the repository, and that ban still holds. What `tab_repository` hands to the rest of the slice is the domain model; the database's shape stays in the box.

There's still the proof missing. The same Tuesday request, "split the tab," redone on the new tree. Compare this diff with the previous section's, file by file: it's the same four edits, one per piece.

```
--- a/features/tab/tab.dart

+++ b/features/tab/tab.dart

@@ class Tab

+   final int people;

--- /dev/null

+++ b/features/tab/split_tab.dart

+// The "split tab" change lives entirely inside this slice.

+int splitTab(Tab tab, int people) {

--- a/features/tab/tab_orchestrator.dart

+++ b/features/tab/tab_orchestrator.dart
```

```
@@ class TabOrchestrator

+ int onSplitTab(int table, int people) {

--- a/features/tab/tab_view.dart

+++ b/features/tab/tab_view.dart

@@ class TabView

+ String renderSplit(int table, int people) {
```

The diff didn't shrink: it's still four edits, because the change still needs a field, a rule, a sequence, and a button. What shrank was the blast radius. Every edit lives under `features/tab/`, the reviewer opens one folder and sees the whole intent, the merge only conflicts with whoever else touched the tab, and the question "who owns the tab?" points to a folder with an owner's name on it.

Try it: run <https://focus.kodel.com.br/en/dart/11-01> (or <https://focus.kodel.com.br/en/ts/11-01>): it's the entire tab slice in a single file, with each piece's path preserved in the comments. Delete the use case section and run it again. Predicted result: only the split breaks; menu, inventory, payment, and loyalty don't even exist in the file, because none of them takes part in this change. The other eight languages are at the routes <https://focus.kodel.com.br/en/java/11-01>, <https://focus.kodel.com.br/en/csharp/11-01>,

<https://focus.kodel.com.br/en/go/11-01>,
<https://focus.kodel.com.br/en/php/11-01>,
<https://focus.kodel.com.br/en/python/11-01>,
<https://focus.kodel.com.br/en/kotlin/11-01>,
<https://focus.kodel.com.br/en/swift/11-01>, and
<https://focus.kodel.com.br/en/rust/11-01>.

Why it isn't four folders

By now the question has already formed: why doesn't the tab slice have a `view/` folder inside it, plus an `orchestrator/` and the other two? The table has four rows, the slice would have four folders, each file would fall into its own, and the arrangement looks more serious than five loose files.

The answer is that a folder is a promise of ownership, and a technical role has no owner. Apply two questions to any folder you think of creating, and both have to answer yes at the same time:

1. **Who named it?** The cut is named by the business owner, in business words.
2. **What's left if I delete it?** Deleting the whole folder removes the cut from the system without touching a single file left in the slice.

`tab` passes both. Rosie says "the tab" without anyone teaching her the word, and deleting `features/tab/` takes the tab out of the system without menu, inventory, payment, and loyalty losing a line. `view` fails both. No business owner ever asked for "a view", and deleting `view/` would mutilate all five slices at once: each one loses its screen, none loses a whole capability. What cuts like that isn't a business cut, it's a file drawer.

The criterion has a practical consequence, and it is the slice's growth mechanism. A slice that puts on weight doesn't gain a drawer: it gains a **sub-feature**, a business cut with a folder of its own inside the slice, which holds every file that concerns only it and is flat on the inside the same way. If the inner cut grows, the criterion applies again, one level down.

Suppose Rosie's loyalty grows into two programs she names herself: the coffee stamp she punches on a paper card today and a monthly subscription club. Each one has its own screen, rule, and storage, each one is opened by the app's route table, which lives outside the slice, and what's left at the root is the customer both programs read. The tree looks like this:

```
features/loyalty/  
├── coffee_stamp/  
│   ├── coffee_stamp_orchestrator  
│   ├── coffee_stamp_repository  
│   ├── coffee_stamp_view  
│   └── punch_stamp  
├── subscription_club/  
│   ├── charge_club_membership  
│   ├── subscription_club_orchestrator  
│   ├── subscription_club_repository  
│   └── subscription_club_view  
└── loyal_customer
```

Hypothetical is the word: the coffee shop that actually runs, the one in chapter 22, has its four slices flat, none of them with a sub-feature, because none reached that size. Two things to notice in the drawing. Both new folders are flat on the inside, with the same role suffixes in the names, and that is what it means to say the mechanism is recursive. And `loyal_customer` stayed at the root

because it belongs to neither program: deleting `coffee_stamp/` takes the stamp out of the system and doesn't touch `loyal_customer` or the club.

The counterexample is more common than the example, and it is the cut that passes one condition and fails the other. Splitting the tab among the people at the table is named by Rosie in those words, so the first condition passes. The second one fails: the payment screen declares the screen of the split shares and instantiates it, so deleting a `split_tab/` folder would leave `payment_view` pointing at a file that no longer exists. One condition alone isn't enough, and the split stays flat in the payment slice, next to the rest. There's a mechanical symptom of the same verdict: with the extra folder in the path, that file's import runs past 85 columns, the limit this book imposes on its own listings, and it fits without it.

The whole slice at once

The folder criterion answers where each file lives. What's missing is the measure of size, and it comes from a limit that didn't exist when the architecture vocabulary was written.

Whoever works with a language model works inside the **context window**, the total text the model considers at once, request and response together, counted in chapter 6's tokens. Blow past the limit and something is left out, and what gets left out isn't chosen by importance. The practical question that creates for a project's design is a blunt one: which unit of reading answers a whole task without blowing the window?

The flat slice is this chapter's answer. To touch the tab's discount, what has to come in are the five files in `features/tab/` and the contracts it uses, and nothing else in the system has to come

along. In the layered organization the same task means opening five distant folders and carrying, in each one, the files of the other four features that live there by accident of technical role. The cost isn't aesthetic: it's the number of irrelevant things taking up the window before the question gets answered.

The property has a name: **context locality**, what changes together being close enough to be read together. It isn't a new concept in this chapter, it's the axis of change measured with another ruler. The axis of change asks what changes for the same reason; context locality asks how much you read to answer that change. When the cut gets the first one right, the second comes along, and the same slice that fits in the window is the one that fits in the head of whoever joined the team yesterday.

What the imports give away

You don't have to take a diagram's word for it: the compiler records coupling in text, right in every file's header. The listing below compares the tab view's imports under the two organizations. Both languages write identical paths; the only mechanical difference in TypeScript is that the path drops the `.dart` and the line gains braces around the imported name, as in

```
import { TabService } from "../../services/tab_service".
```

 Dart ·  TypeScript

```
// Group 1: layered, the header of views/tab/tab_view.dart. The view

// climbs two levels and crosses the project to find what it uses.

import '../..models/tab.dart';
```

```
import '../services/tab_service.dart';

// Group 2: sliced, the header of features/tab/tab_view.dart. The view

// imports one neighbor only: the orchestrator that publishes the state

// it draws.

import 'tab_orchestrator.dart';
```

Read the paths as arrows from the axis-of-change section's diagram. Every `../..` in the first group is an arrow crossing the diagram end to end: the view lives in one folder, climbs to the root, and dives into another folder shared by all five features. The second group's import has no path at all, just a filename: what it uses sits in the same folder, inside the slice's fence.

Notice what the second group doesn't import, and notice that distance has nothing to do with it anymore. `tab_repository` and `split_tab` sit in the same folder, one name away, and they stay out of the view's reach: the View row forbids data access and forbids business rules, and the ban belongs to the contract, not to the geography. In the tree with drawers the two were easy to confuse, because writing `../data/` looked expensive. In the flat folder the shortcut is cheap, and the only thing barring it is chapter 10's table. Chapter 12 devotes a whole pitfall to the day this import shows up.

The imports' distance measures coupling between folders, and that gives you an audit trick that works on any project, yours or someone else's: open half a dozen files and look only at the headers. Short, neighboring imports say that what changes

together lives together. Headers full of `../../` crossing the project say the axis of change and the folder tree disagree, and every simple request is about to cost a crossing.

The same slice in ten languages

The slice isn't a Dart idea, or a TypeScript one. The ten implementations of snippet 11-01 carry the same tree in their path comments, and the spot where it becomes visible on a single screen is the orchestrator: it calls `split_tab` and asks `tab_repository` for the tab, two files that sit right next to it. Three names, two lines of code, not one step outside `features/tab/`.

Start with the two languages that opened the chapter. Notice that the use case file has a verb for a name and stands alone, with no class wrapped around it:

Dart

```
// features/tab/split_tab.dart

int splitTab(Tab tab, int people) {

  var total = 0;

  for (final item in tab.items) {

    total += item.priceInCents;

  }
```

```

    return total ~/ people;

}

// features/tab/tab_orchestrator.dart

class TabOrchestrator {

    TabOrchestrator(this.repository);

    final TabRepository repository;

    int onSplitTab(int table, int people) {

        return splitTab(repository.fetchTab(table), people);

    }

}

```

TypeScript writes the same neighborhood with two mechanical swaps: integer division becomes `Math.trunc`, and the dependency enters through a field assigned in the constructor.

TS TypeScript

```
// features/tab/split_tab.ts
```

```
function splitTab(tab: Tab, people: number): number {
```

```
    let total = 0;
```

```
    for (const item of tab.items) {
```

```
        total += item.priceInCents;
```

```
    }
```

```
    return Math.trunc(total / people);
```

```
}
```

```
// features/tab/tab_orchestrator.ts
```

```
class TabOrchestrator {
```

```
    private readonly repository: TabRepository;
```

```
    constructor(repository: TabRepository) {
```

```

        this.repository = repository;

    }

    onSplitTab(table: number, people: number): number {

        return splitTab(this.repository.fetchTab(table), people);

    }

}

```

The other eight write the same neighborhood, and repeating the whole slice ten times would just repeat the same tree in different syntax. Below is only each one's orchestrator, which is where the three names meet. In every one, look for the same two things: the path in the comment and the use case's name called with no folder qualification at all.

Kotlin shrinks the orchestrator down to a primary constructor, and the dependency is declared on the class's own line:

Kotlin

```

// features/tab/TabOrchestrator.kt

class TabOrchestrator(private val repository: TabRepository) {

    fun onSplitTab(table: Int, people: Int): Int {

```

```
        return splitTab(repository.fetchTab(table), people)

    }

}
```

Swift uses a struct with `let`: the dependency is an immutable property, and the initializer comes for free, with no line written for it.

Swift

```
// features/tab/TabOrchestrator.swift

struct TabOrchestrator {

    let repository: TabRepository

    func onSplitTab(table: Int, people: Int) -> Int {

        splitTab(repository.fetchTab(table: table), people: people)

    }

}
```

C# spells out the constructor in full, and the use case needs a class wrapped around it, because C# has no standalone functions. The file still has a verb for a name, and the call `SplitTab.Execute` shows the neighboring folder right in its own name.



```
// features/tab/TabOrchestrator.cs

class TabOrchestrator

{

    private readonly TabRepository _repository;

    public TabOrchestrator(TabRepository repository)

    {

        _repository = repository;

    }

    public int OnSplitTab(int table, int people)

    {

        return SplitTab.Execute(
```

```
        _repository.FetchTab(table),  
  
        people  
  
    );  
  
}  
  
}
```

Java has the same restriction as C# and solves it the same way: the rule is a static method inside a class with a verb for a name.

Java

```
// features/tab/TabOrchestrator.java  
  
class TabOrchestrator {  
  
    private final TabRepository repository;  
  
    TabOrchestrator(TabRepository repository) {  
  
        this.repository = repository;  
  
    }  
  
}
```

```

int onSplitTab(int table, int people) {

    return SplitTab.splitTab(

        repository.fetchTab(table),

        people

    );

}

}

```

PHP goes back to a standalone function for the rule, and PHP 8's constructor promotion declares the dependency right in the signature.

PHP

```

// features/tab/TabOrchestrator.php

final class TabOrchestrator

{

    public function __construct(

        private readonly TabRepository $repository,

```



```

    ) {

    }

    public function onSplitTab(int $table, int $people): int

    {

        return splitTab(

            $this->repository->fetchTab($table),

            $people,

        );

    }

}

```

Python is the only one that splits the fetch from the decision into two named lines, and that leaves the orchestrator's sequence literal: the data first, the rule after.

Python

```

# features/tab/tab_orchestrator.py

class TabOrchestrator:

```

```

def __init__(self, repository: TabRepository) -> None:

    self._repository = repository

def on_split_tab(self, table: int, people: int) -> int:

    tab = self._repository.fetch_tab(table)

    return split_tab(tab, people)

```

Go has no classes. The orchestrator is a struct with a receiver method, and the dependency is the `Repository` field, filled in by whoever assembles the graph. The folder neighborhood is identical.



Go

```

// features/tab/tab_orchestrator.go

type TabOrchestrator struct {

    Repository TabRepository

}

func (o TabOrchestrator) OnSplitTab(table, people int) int {

```

```
        return SplitTab(o.Repository.FetchTab(table), people)

    }
```

Rust splits data from behavior into two blocks, `struct` and `impl`, and `&self.repository` makes the borrow explicit. The path in the comment stays the same.

Rust

```
// features/tab/tab_orchestrator.rs

struct TabOrchestrator {

    repository: TabRepository,

}

impl TabOrchestrator {

    fn on_split_tab(&self, table: i64, people: i64) -> i64 {

        split_tab(&self.repository.fetch_tab(table), people)

    }

}
```

Ten syntaxes, one tree. None of the ten needed a common folder to work, and that absence is what the next section is about.

shared/ is born empty

Time to pay off the tree's promise: where's `shared/`? Nowhere, and the rule governing that is the chapter's fourth and last concept. The late `shared/` rule says the `shared/` folder is born empty and absent, and that code only moves up to it once reuse has proven itself in at least two real slices, written and working. Two, not one and a half: as long as the second slice that needs the code doesn't exist on disk, the code stays in the only slice that uses it. The trigger is chapter 5's rule of three: wait for evidence of repetition before you abstract. The foundation is chapter 4's YAGNI: don't build for the need you imagine, build for the one that showed up.

I carry a scar that backs up this rule, and I'd rather tell it than fake neutrality. On a project that passed through my hands, the `shared/` folder was created on day one, before the first feature, "because we're going to need it." Two years later it was the system's biggest source of coupling: forty-something files that every feature imported, where any change demanded testing the whole app, and where every ownerless piece of code got pushed, because the common folder is the path of least resistance. No slice had a fence, because all of them had a tunnel to the same basement. The folder born to avoid duplication turned into the place every change leaked through.

Two things the word "folder" runs together are worth pulling apart, because this chapter's rule looks like it bans both and bans only one. The drawer it bans is a technical-role folder repeated inside every slice, and its damage is cutting one business capability into pieces: with it, a change to the tab jumps four fences inside the tab itself. `shared/` and the root infrastructure

folder don't do that. Both sit outside `features/`, both exist once in the whole project instead of once per slice, and neither cuts any capability: `shared/` holds what two slices proved they have in common, and infrastructure holds the wiring nobody asks for in business words, the route table, the dependency graph, the entry point. The axis of change backs them up, because what lives in them changes for its own reasons and not alongside a feature. What doesn't change is the timing: `shared/` is still born late, with reuse proven in two real slices.

By now you've probably heard the two classic critiques of this way of organizing, and both deserve an answer:

The first: "organizing by feature is just reorganizing folders, and the same layers stay inside every folder." Oskar Dudycz published that objection in "My thoughts on Vertical Slice Architecture" (<https://www.architecture-weekly.com/p/my-thoughts-on-vertical-slices-cqrs>): for him, this just relocates the layered structure into each feature folder: the same over-engineering, rearranged into new drawers.

He's right, and the tree you read in this chapter is what that critique produced. The design I used to defend before it put `view/`, `orchestrator/`, `usecases/`, and `data/` inside every slice, and I called that a vertical slice. It was the objection's exact target: the four layers were still there, with the same mandatory crossing, only multiplied by five, one copy per feature. Read that way, it isn't an objection to the vertical slice; it's an objection to whoever copied the folder and left the principle behind. The four folders became four name suffixes, each file's role is still declared, and the crossing is gone.

What the objection doesn't reach is the rest. If the tree were the principle, there'd be nothing left to answer; but the tree is the symptom, and the principle is the axis of change, which also sizes the slice from the inside. A CRUD with no business rule

gains no use case file at all: the slice keeps the view, the orchestrator, and the repository, and chapter 10's table still stands as the contract for what the use case would do if it existed, ready for the day the first rule shows up. A slice isn't a four-piece mold; it's the cut of what changes together, whatever size the feature calls for.

The second critique: slices duplicate code and fragment the system, because each one rewrites what could be shared. The answer is the rule that opens this section. Once reuse proves itself in two slices, the code moves up to `shared/` with a business name and the duplication dies; until it proves itself, temporary duplication is cheaper than the wrong abstraction, as chapter 5 argued with Sandi Metz. Bogard himself, in "Vertical Slice Architecture" (2018), treats coupling between slices as the cost to minimize; a premature `shared/` is exactly that cost, installed wholesale on day one.

Pitfalls

The first pitfall is a premature `shared/`, the same one from the scar. What goes wrong: every slice ends up depending on the common folder, and any change to it ripples through the whole system. Why: reuse was guessed instead of proven, and guessing at reuse errs on the expensive side. How to get out: return each piece of code in `shared/` to the one slice that actually uses it; whatever's left, used by two or more, has earned the right to stay.

The second is the `utils/` folder, which starts with two date functions and turns into the project's junk drawer. What goes wrong: `utils/` has no axis of change at all, so everything fits in it, and what fits everywhere belongs nowhere. Why: the name is technical and empty, it says nothing about which business the code is for. How to get out: every function in `utils/` either belongs

to a slice and goes back to it, or has proven reuse in two slices and moves up to `shared/` with a business name, like `price_formatting`, never `helpers`.

The third is slicing by screen instead of by feature. It looks the same, until the day `features/tab_screen/` and `features/split_tab_screen/` change together on every single request, because they're the same feature cut in two. That's the symptom: two slices that always show up in the same diff. The fix is to merge them and give the slice the business name, `tab`, with as many screen files as it needs: `tab_view`, `tab_list_view`, one per screen.

Q&A

What if one slice needs another? Payment checks loyalty to apply the discount. The case exists and has an address: the collaboration happens in the payment orchestrator, which asks for the points through the loyalty slice's public contract, never importing one of its internal files. The fine-grained design of that conversation belongs to chapters 13 and 14; for now, keep the pocket rule: a slice talks to a slice through the front door.

My project is small, three screens. Do I need this? You need the criterion, not the ceremony. Three screens in three slices cost three folders, the same number of folders `views/`, `controllers/`, and `models/` would cost, and they save you a change of address the day the project grows. No subfolder comes along for the ride: a small slice stays small.

Does a plain CRUD need all four pieces? No. With no business rule, there's nothing to write in a use case file, and the slice keeps the view, the orchestrator, and the repository. Chapter 10's table keeps being the contract: when the first rule arrives, you'll know exactly which file to create and what it can and can't do.

Quick tip

In your next code review (another person's review of the code before it merges), ignore the body of the files for one minute and read only the list of paths touched in the diff. If that list doesn't fit inside one feature folder, you just found the project's real axis of change, and it disagrees with the tree.

Quick reference

Situation	Fix
Screen or text change	<code>features/<slice>/<slice>_view</code>
New or changed business rule	<code>features/<slice>/<verb>_<entity></code> , one per verb
New field from the API or database	<code>features/<slice>/<entity></code> and its repository
New sequence from event to state	<code>features/<slice>/<slice>_orchestrator</code>
Large business cut inside the slice	a folder with the name the owner uses; reread the two conditions
Same code in two real slices	<code>shared/</code> candidate, with a business name
Code that “might” get reused	stays in the slice that uses it; YAGNI (chapter 4)

Exercises

1. Close the book and sketch Rosie's Coffee Shop's tree: the five slices and the five files of `features/tab/`, with the name this chapter gave each one. Check it against the coffee-shop-in-slices section; any folder you invented inside the slice is the drawer coming back, any file you left out is the table row you haven't linked to a name yet.
2. Place three of Rosie's requests, one file each: "two-for-one coffee promo on Thursdays," "a new digital wallet payment method," and "low-stock alert." Answer key: the promo is a menu pricing rule and turns into a verb file in `features/menu/`; the digital wallet is a new way to pay and lives in `features/payment/` (the rule in a verb file, the integration in `payment_repository`); the alert belongs to inventory and lives in `features/inventory/`.
3. Price formatting in dollars today exists only in the tab's view. Should it move up to `shared/`? Decide and say the criterion out loud before you check: it doesn't move up, because reuse hasn't proven itself in two real slices yet; the day the menu view needs the same formatting, the two slices prove the reuse and the function moves up with a business name.

Tip 11

Organize by what changes together, not by what looks alike.

Next chapter: chapter 12 lifts the table's first row off the page: the View, the piece the user touches, built inside `features/tab/tab_view` without carrying a single rule.

The View: Dumb by Design

In this chapter, you'll:

- classify any screen-code snippet as “belongs in the View” or “leaked from another layer,” with a test that fits in one sentence;
- refactor a screen that adds up totals, decides discounts, and formats currency, until only the event/state pair is left;
- define renderable state and use the wrong-layer test in your next code review.

You’ve debugged a wrong total at eleven at night and found out the math lived inside a widget. The screen looked like the natural place: the value shows up there, so it gets calculated there. This chapter shows why that “natural” spot is the most expensive place in the system for a rule to live, and hands you the alternative: the screen that decides nothing. A screen with no decisions has no rule bugs; at worst, it has a pixel bug.

Chapter 11 left you standing at the door of `features/tab/tab_view`, the first of the slice’s four pieces. That door’s contract has been signed since chapter 10, in the View row of the canonical table: it “fires events” and “renders state,” and it forbids “business rules” and “data access.” Two short cells, and every word in them carries a decision. “Fires events” means the screen’s output is a business-named notice, never a call to a service. “Renders state” means the input arrives ready, with nothing left to calculate. And

the two forbidding cells shut the back doors: no rule inside the screen, no direct data access. This chapter's path is watching that row turn into code: first the screen that ignores the contract and does everything, then the state that arrives ready, then that same screen shrinking until it obeys.

The screen that calculates

Rosie's Coffee Shop's tab screen, the way almost every screen is born. It receives the raw items, price in cents, and settles the rest on its own:



```
// The screen that does everything for the tab. It works, and that's
// the problem.

class NaiveTabView extends StatelessWidget {

  const NaiveTabView({

    required this.items,

    required this.loyaltyPoints,

    super.key,

  });
```

```
// Each item arrives raw: name and price in cents, no decision
```

```
// at all.
```

```
final List<{String name, int priceInCents}> items;
```

```
// And the customer's points arrive raw too, for the screen to
```

```
// decide.
```

```
final int loyaltyPoints;
```

```
@override
```

```
Widget build(BuildContext context) {
```

```
    // Adds up the items in a loop: business rule inside build.
```

```
    var totalInCents = 0;
```

```
    for (final item in items) {
```

```
        totalInCents += item.priceInCents;
```

```
    }
```

```
// Applies the loyalty discount in an inline if: more rule.

if (loyaltyPoints >= 100) {

    totalInCents = (totalInCents * 90) ~/ 100;

}

// Formats the currency inside build: a local presentation

// decision.

final total = "\${(totalInCents / 100).toStringAsFixed(2)}";

// Builds the list and decides, at the button, whether it can pay.

return Column(

    children: [

        for (final item in items)

            Text(

                "${item.name}: "

                "\${(item.priceInCents / 100).toStringAsFixed(2)}",
```

```

    ),

    Text("Total: $total"),

    ElevatedButton(

        onPressed: totalInCents > 0 ? () {} : null,

        child: const Text("Pay"),

    ),

],

);

}

}

```

A bit over forty lines, and all of them work. The barista sees the items, the total comes out right, the 10% loyalty discount kicks in the moment the customer hits 100 points (the `if` compares with `>=`, so an even hundred already qualifies). No manual test fails this screen. The problem isn't what it shows; it's what it knows.

The first pain shows up when you try to test the sum. The `totalInCents` loop lives inside `build`, so checking that $700 + 1195$ equals 1895 requires you to instantiate a widget, build a render

tree, and inspect a `Text` by its string. You pay the price of a UI test to check the addition of two integers.

The second pain shows up on the screen next door. The coffee shop has the barista's screen and the register screen, and both show the same tab's total. If the math lives inside `build`, each screen carries its own copy of the loop and the discount `if`, because `build` doesn't care about exports and doesn't export itself. The tab's most important business rule now exists in two places that don't know about each other.

The third pain is the second pain's bill, and it arrives with a date. The day Rosie changed the discount from 10% to 15%, the developer edited the `if` on the barista's screen and forgot the one on the register screen; QA (Quality Assurance) opened the app and found three screens with three different totals for the same table, because the end-of-day report carried a third copy of the rule. None of the three was "wrong" in its own code. What was wrong were the copies, and copies are exactly what a screen that calculates manufactures.

The state that arrives ready

The way out of the three pains is an inversion: instead of the screen receiving raw data and deciding, it receives everything already decided. **Renderable state** is the data structure the View receives ready to display: the total's text already formatted, the button enabled or disabled as a boolean, the list already sorted. Nothing to calculate. If it arrived, render it.

Notice the hidden test buried in that definition: the field's type gives away who's deciding. A `double total` invites the screen to format it; a `String formattedTotal` was already formatted by someone else, and the screen just passes it along. Between the two,

renderable state always picks the second, because formatting is deciding how a piece of information looks, and no decision belongs to the screen.

State is the input. The output follows the same discipline, and chapter 10 already introduced it from a distance: **event as the View's only output** means the screen doesn't call a service, doesn't query a repository, and doesn't decide where to go next; it emits an event with a business name and waits for the next state to arrive. Input and output are the two ends of the same contract, and the tab's contract fits in one listing:

 **Dart** ·  **TypeScript**

```
// The tab's renderable data: everything the screen shows arrives

// ready.

class TabData {

  const TabData({

    required this.items,

    required this.formattedTotal,

    required this.canPay,

  });
```



```
// Arrive ALREADY sorted: the screen doesn't sort.

final List<TabItem> items;

// Ready text (e.g. "$27.50"): the screen doesn't format.

final String formattedTotal;

// Decision made elsewhere: the screen doesn't compare.

final bool canPay;

}

class TabItem {

    const TabItem(this.name, this.formattedPrice);

    final String name;

    final String formattedPrice;

}
```

```
// The View's only output: business-named events, chapter 10's
```

```
// spelling.
```

```
sealed class TabEvent {}
```

```
final class AddItem extends TabEvent {
```

```
    AddItem(this.table, this.item);
```

```
    final int table;
```

```
    final String item;
```

```
}
```

```
final class RemoveItem extends TabEvent {
```

```
    RemoveItem(this.table, this.item);
```

```
    final int table;
```

```
    final String item;
```

```

    }

    final class PayTab extends TabEvent {

        PayTab(this.table);

        final int table;

    }

```

TypeScript writes the same contract with a `type` and a discriminated union, like chapter 10: `TabData` becomes a `type` with `readonly` fields, and each event becomes a member with a literal field `type: "addItem"`. The names don't change from one language to the other, and that's deliberate: `AddItem` and `TabEvent` are the spelling chapter 10 published, and `RemoveItem` and `PayTab` follow the same verb-plus-object convention. Notice, too, what the state doesn't carry: the table. The screen knows which table it is from the navigation context, and it carries the number inside the event; the state only carries what gets drawn.

`TabItem` deserves a second look, because it's the definition of renderable state in miniature. Chapter 10's menu item had `priceInCents`, an integer for a rule to do math with. This one has `formattedPrice`, a text for the screen to display. Same business item, two different types, because each layer receives the data in the shape its role consumes.

Try it: open <https://focus.kodel.com.br/en/dart/12-01> (or <https://focus.kodel.com.br/en/ts/12-01>): the whole contract in a single file, with a stub that hands back ready states. Delete the `formattedTotal` field from the state and run it. Prediction: the compiler points straight at the line that renders the total, for free, on the spot; without typed state, that same mistake would be a UI test breaking at 3am, or a customer complaining about a blank total. The other eight languages live at <https://focus.kodel.com.br/en/java/12-01>, <https://focus.kodel.com.br/en/csharp/12-01>, <https://focus.kodel.com.br/en/go/12-01>, <https://focus.kodel.com.br/en/php/12-01>, <https://focus.kodel.com.br/en/python/12-01>, <https://focus.kodel.com.br/en/kotlin/12-01>, <https://focus.kodel.com.br/en/swift/12-01>, and <https://focus.kodel.com.br/en/rust/12-01>.

The same screen, now dumb

No new screen: the refactor takes `NaiveTabView` from the first section apart, decision by decision, and every decision it loses gets a named destination.

The sum leaves first. The `totalInCents` loop stops existing in the screen, because the total arrives inside the state, in `formattedTotal`; who builds that state is the orchestrator's job, and how it builds it is chapter 13's business. The discount `if` leaves next, by the same road: if the total that arrives already carries the discount applied, the screen has no reason to know the 100-point cutoff or the 10% cut. Formatting leaves last, `toStringAsFixed` and the currency prefix, because ready text travels better than a raw number: `"$18.95"` displays the same on any screen that receives it, and the

display rule ends up with a single home. Even the button's decision goes away: `canPay` arrives as a boolean, compared by no one here.

What's left is this:

Dart

```
// The SAME screen, now dumb: renders state and fires events. That's
// it.

class TabView extends StatelessWidget {

  const TabView({

    required this.table,

    required this.data,

    required this.onEmit,

    super.key,

  });

  final int table;

  final TabData data;
```

```
final void Function(TabEvent event) onEmit;

@override

Widget build(BuildContext context) {

    // Builds the list: the items arrive ready and already sorted.

    return Column(

        children: [

            for (final item in data.items)

                ListTile(

                    title: Text(item.name),

                    trailing: Text(item.formattedPrice),

                    onLongPress: () => onEmit(RemoveItem(table, item.name)),

                ),

            // Passes the total's text along: it arrived ready, it

            // leaves ready.
```

```

        Text("Total: ${data.formattedTotal}"),

        // Fires the event: the screen's only output.

        ElevatedButton(

            onPressed: () => onEmit(AddItem(table, "Cappuccino")),

            child: const Text("Add cappuccino"),

        ),

        ElevatedButton(

            // Enablement already decided elsewhere: the screen

            // doesn't compare.

            onPressed: data.canPay ? () => onEmit(PayTab(table)) : null,

            child: const Text("Pay"),

        ),

    ],

);

}

```

```
}
```

Look for a calculation in that listing. There isn't one. No sum, no comparison against a business value, no `toStringAsFixed : build` turned into a direct translation from `TabData` to widgets, plus three spots where a barista's gesture becomes a `TabEvent`. The screen does exactly the two things the table's row grants it, and not one more.

The same screen in React proves the pattern isn't Flutter's alone:

TS TypeScript

```
// The SAME screen, now dumb: renders state and fires events. That's
```

```
// it.
```

```
type Props = {
```

```
  readonly table: number;
```

```
  readonly data: TabData;
```

```
  readonly onEmit: (event: TabEvent) => void;
```

```
};
```

```
export function TabView({ table, data, onEmit }: Props) {
```

```
  return (
```



```
<div>

  { /* Builds the list: the items arrive ready and already
    sorted. */}

  <ul>

    {data.items.map((item) => (

      <li

        key={item.name}

        onDoubleClick={() =>

          onEmit({ type: "removeItem", table, item: item.name })

        }

      >

        {item.name}: {item.formattedPrice}

      </li>

    )}}

  </ul>
```

```

    { /* Passes the total's text along: arrived ready, leaves
      ready. */ }

    <p>Total: {data.formattedTotal}</p>

    { /* Fires the event: the screen's only output. */ }

    <button

      onClick={() =>

        emit({ type: "addItem", table, item: "Cappuccino" })

      }

    >

      Add cappuccino

    </button>

    <button

      // Enablement already decided elsewhere: no comparison here.

      disabled={!data.canPay}

      onClick={() => emit({ type: "payTab", table })}

```

```

    >

    Pay

  </button>

</div>

);

}

```

The two listings differ on the skin and agree on the skeleton. Flutter's `build` returns a widget tree built in plain Dart; React returns elements written in JSX (JavaScript XML), the syntax that mixes markup and expression. Flutter receives its three dependencies through the constructor and emits through `onEmit`, a handler; React receives the same three through `props` (short for *properties*, the values a component receives from outside) and emits through the same kind of callback. Swap the names around and the design is one and the same: `TabData` comes in, `AddItem`, `RemoveItem`, and `PayTab` go out.

If you're coming from React, one absence should have jumped out at you: there's no `useState` in this component. That's not an oversight. Business state (items, total, can-pay) lives outside the screen and arrives through renderable state; `useState` stays legitimate for local, pure-UI state, the kind no other layer has any reason to know about: a field's focus, the scroll position, an animation's progress. The test is asking whether Rosie cares. She cares about the total; she doesn't care where the scroll stopped.

Two frameworks don't make a proof yet; they make a coincidence. The proof is the book's other eight languages, each in the real framework its readers use at work, all of them coming up next. In each listing, look first for where the state comes in ready. In the first six, look also for where the event goes out named, whether through a callback or a form's `action`. The last two, Go and Rust, print only the drawing half: in them the event is born outside the listing, and the text says where. Everything that changes from one to the next is the syntax in the middle.

Kotlin, in Jetpack Compose, is Flutter's next-door neighbor: a function annotated `@Composable` instead of a class with `build`, and the same three parameters arriving from outside:

Kotlin

```
// The dumb screen: renders state and fires events. That's it.

@Composable

fun TabView(

    table: Int,

    data: TabData,

    onEmit: (TabEvent) -> Unit,

) {

    Column {
```

```
// Builds the list: the items arrive ready and already sorted.
```

```
data.items.forEach { item ->
```

```
    Text("${item.name}  ${item.formattedPrice}")
```

```
}
```

```
// Passes the total's text along: it arrived ready, it
```

```
// leaves ready.
```

```
Text("Total: ${data.formattedTotal}")
```

```
// Fires the event: the screen's only output.
```

```
Button(onClick = { onEmit(AddItem(table, "Cappuccino")) }) {
```

```
    Text("Add cappuccino")
```

```
}
```

```
Button(
```

```
    onClick = { onEmit(PayTab(table)) },
```

```

        // Enablement already decided elsewhere: no comparison

        // here.

        enabled = data.canPay,

    ) {

        Text("Pay")

    }

}

}

```

Swift, in SwiftUI, writes the same function as a `struct` that declares a `body`: the View is literally a function of the state it receives, and events go out through the same callback. The visible difference is the events `enum` with associated values, which chapter 10 introduced: the dot before `.removeItem` is Swift shortening `TabEvent.removeItem`.

Swift

```

// The dumb screen: renders state and fires events. That's it.

struct TabView: View {

    let table: Int

```

```
let data: TabData

let onEmit: (TabEvent) -> Void

var body: some View {

    VStack {

        // Builds the list: items arrive ready and already

        // sorted.

        List(data.items) { item in

            HStack {

                Text(item.name)

                Spacer()

                Text(item.formattedPrice)

            }

            .onLongPressGesture {

                onEmit(.removeItem(table: table, item: item.name))

            }

        }

    }

}
```

```
}

// Passes the total's text along: arrived ready, leaves

// ready.

Text("Total: \$(data.formattedTotal)")

// Fires the event: the screen's only output.

Button("Add cappuccino") {

    onEmit(.addItem(table: table, item: "Cappuccino"))

}

Button("Pay") {

    onEmit(.payTab(table: table))

}

// Enablement already decided elsewhere: no comparison

// here.
```



```
        .disabled(!data.canPay)

    }

}

}
```

C#, in Blazor, closes out the component-framework group: the markup lives in a `.razor` file, and the three dependencies arrive through `[Parameter]`, Blazor's equivalent of React's `props`:



```
@* The dumb screen: renders state and fires events. That's it. *@

<ul>

    @foreach (var item in Data.Items)

    {

        @* Builds the list: items arrive ready and already sorted. *@

        <li>@item.Name: @item.FormattedPrice</li>

    }

</ul>
```

```
@* Passes the total's text along: arrived ready, leaves ready. *@
```

```
<p>Total: @Data.FormattedTotal</p>
```

```
@* Fires the event: the screen's only output. *@
```

```
<button @onclick='() => OnEmit(new AddItem(Table, "Cappuccino"))'>
```

```
    Add cappuccino
```

```
</button>
```

```
@* Enablement already decided elsewhere: no comparison here. *@
```

```
<button disabled="@(!Data.CanPay)"
```

```
    @onclick="() => OnEmit(new PayTab(Table))">
```

```
    Pay
```

```
</button>
```

```
@code {
```

```
    [Parameter] public int Table { get; set; }
```

```

[Parameter] public TabData Data { get; set; } = default!;

[Parameter] public Action<TabEvent> OnEmit { get; set; }

    = default!;

}

```

The next three languages live on the server, and in them the View changes body without changing contract. In Spring MVC, in Laravel, and in Django, the “screen” is a pair: a controller that converts the HTTP gesture into an event, and a template that repeats the state. The template is dumb by construction, because a template language barely knows how to do math; the controller is the part discipline keeps dumb: it receives the request, assembles the event, and passes it on to the orchestrator, with no rule along the way. The event doesn’t go out through a callback: it goes out through the form’s `action`, which names the gesture’s route. In Java, with Thymeleaf:

Java

```

<!-- The dumb view: it just repeats what arrived ready in the state. -->

<ul>

    <li th:each="item : ${data.items}">

        <span th:text="${item.name}"></span>

        <span th:text="${item.formattedPrice}"></span>

```

```

</li>

</ul>

<p th:text="'Total: ' + {{data.formattedTotal}}"></p>

<form th:action="@{/tab/{t}/add(t={{table}})}" method="post">

    <button name="item" value="Cappuccino">Add cappuccino</button>

</form>

<form th:action="@{/tab/{t}/pay(t={{table}})}" method="post">

    <!-- Enablement already decided elsewhere: the template doesn't
        compare a business value, it just reads the ready boolean.
    -->

    <button th:disabled="{{!data.canPay}}">Pay</button>

</form>

```

PHP, with Blade, writes the same template in Laravel's syntax, and the `@disabled` directive reads the same ready boolean `th:disabled` read:

 **PHP**

```

<ul>

```

```
@foreach ($data->items as $item)

    {{-- Builds the list: items arrive ready and already sorted. --}}

    <li>{{ $item->name }}: {{ $item->formattedPrice }}</li>

@endforeach

</ul>

{{-- Passes the total's text along: arrived ready, leaves ready. --}}

<p>Total: {{ $data->formattedTotal }}</p>

{{-- Fires the event: the screen's only output. --}}

<form method="post" action="/tab/{{ $table }}/add">

    @csrf

    <button name="item" value="Cappuccino">Add cappuccino</button>

</form>

<form method="post" action="/tab/{{ $table }}/pay">
```

```

@csrf

{{-- Enablement already decided elsewhere: the template doesn't
      compare a business value, it just reads the ready boolean.
--}}

<button @disabled(!$data->canPay)>Pay</button>
>

</form>

```

Python, with the Django template, closes out the server-side trio. Notice that all three template languages forbid almost everything on purpose: none of them can add numbers, which is exactly why a business rule in a template isn't even a temptation; the temptation lives in the controller, and that's where the dumb View's discipline does its work.

Python

```

<ul>
  {% for item in data.items %}
    {# Builds the list: items arrive ready and already sorted. #}
    <li>{{ item.name }}: {{ item.formatted_price }}</li>
  {% endfor %}
</ul>

{# Passes the total's text along: arrived ready, leaves ready. #}
<p>Total: {{ data.formatted_total }}</p>

{# Fires the event: the screen's only output. #}
<form method="post" action="/tab/{{ table }}/add">
  {% csrf_token %}
  <button name="item" value="Cappuccino">Add cappuccino</button>
</form>

```

```
<form method="post" action="/tab/{{ table }}/pay">
  {% csrf_token %}
  {# Enablement already decided elsewhere: the template doesn't
    compare a business value, it just reads the ready boolean. #}
  <button {% if not data.can_pay %}disabled{% endif %}>Pay</button>
</form>
```

Go skips the framework: `html/template` ships in the standard library, and the dumb template fits in a constant. The `if not .CanPay` doesn't violate the wrong-layer test, and it's worth understanding why: it doesn't compare a business value, it just reads a boolean that arrived already decided, exactly like React's `disabled={!data.canPay}`. The listing shows only half the design, and there's no form and no `action`: Go's template limits itself to repeating the state. What turns the click into `AddItem` is the route's `net/http` handler, written in plain Go, outside the template.



Go

```
// features/tab/tab_view.go

// The dumb template lives in an html/template const: it just repeats

// what arrived ready in the state. The pay button reads the ready

// boolean; no business comparison here.

const tabTemplate = `<ul>

{{- range .Items}}
```

```

    <li>{{.Name}}: {{.FormattedPrice}}</li>
  >

  {{- end}}

  {{- if not .Items}}

    <li>(empty tab)</li>
  >

  {{- end}}

</ul>

<p>Total: {{.FormattedTotal}}</p>

<button{{if not .CanPay}} disabled{{end}}>Pay</button>

`

```

Last, Rust, which pushes the proof to the extreme: the screen doesn't even need to be graphical. The tab's View becomes a terminal render loop, a function that draws the state with `println!`, and the pattern survives intact because it never depended on a widget, on HTML, or on a screen; it only ever depended on the contract: state in, event out. The listing is half the design: the function only prints. The other half is the loop that reads the keystroke and turns it into a `TabEvent` before calling `draw` again.

 **Rust**


```
// The terminal's dumb screen: draws what arrived ready, and nothing

// else.

fn draw(table: u32, data: &TabData) {

    println!("+--- Tab: table {table} ---+");

    for item in &data.items {

        println!("| {} {}", item.name, item.formatted_price);

    }

    if data.items.is_empty() {

        println!("| (empty tab)");

    }

    println!("| Total: {}", data.formatted_total);

    // Enablement already decided elsewhere: no comparison here.
```

```

let pay = if data.can_pay {

    "[P] Pay"

} else {

    "[ ] Pay (unavailable)"

};

println!("| {}", pay);

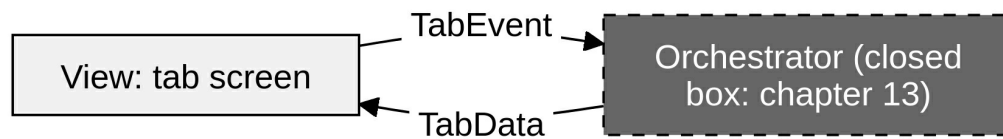
println!("+-----+");

}

```

Ten languages, and not one syntax repeated: a widget in Dart, JSX in TypeScript, `@Composable` in Kotlin, `body` in Swift, `.razor` in C#, three server templates, a standard-library constant in Go, and a `println!` in Rust. What repeated was the skeleton: `TabData` comes in ready in all ten, the event goes out named in the eight that have somewhere to emit it, and not one listing did any math. `RemoveItem` only shows up in Dart, React, and Swift, because only in those three did the remove gesture fit inside the listing; in the other seven it stays in the contract and waits for the gesture that fires it. That invariance is what chapter 10's table called framework as a replaceable detail.

What's left unanswered is where the state comes from. The full answer is chapter 13; for now, the whole cycle fits in a diagram with a closed box in the middle:



The diagram has two boxes and two arrows. The screen sends `TabEvent` to the orchestrator; the orchestrator, drawn as a closed box with a dashed border, returns `TabData` to the screen. What happens inside (where the items come from, who adds them up, who formats them) the diagram hides on purpose, because chapter 13 opens that box unhurried.

Until that box opens, this chapter's code uses a stand-in with an honest name: `OrchestratorStub` takes any `TabEvent` and returns the next `TabData` from a pre-built sequence, written by hand, with no calculation at all. It exists only so the screen has something to render in the playgrounds; the real orchestrator is born in chapter 13.

Try it: open <https://focus.kodel.com.br/en/dart/12-02> (or <https://focus.kodel.com.br/en/ts/12-02>) and tap “Add cappuccino.” Prediction: the list gains the cappuccino and the total becomes \$18.95 without the screen adding anything up, because the stub handed back the sequence's second state; tap again and the tab empties out, the total zeroes, and the “Pay” button disables itself, because `canPay` arrived `false` and the screen never compared anything. The same dumb screen exists in the real framework of each of the other eight languages, at <https://focus.kodel.com.br/en/java/12-02>, <https://focus.kodel.com.br/en/csharp/12-02>,

<https://focus.kodel.com.br/en/go/12-02>,
<https://focus.kodel.com.br/en/php/12-02>,
<https://focus.kodel.com.br/en/python/12-02>,
<https://focus.kodel.com.br/en/kotlin/12-02>,
<https://focus.kodel.com.br/en/swift/12-02>, and
<https://focus.kodel.com.br/en/rust/12-02>.

Does this belong in the View?

The refactor gave you the instinct; what's missing is the test you can say out loud. The **wrong-layer test** is this: if the screen compares business values in an `if`, that `if` is in the wrong layer. Comparison is the cheapest symptom to catch, and wherever there's a business comparison there's a decision, and the table from chapter 10 forbids decisions in the View. Apply the test to five snippets that show up in every codebase, all of them from the coffee shop.

Formatting the pickup date inside the screen, a `DateFormat("MM/dd").format(pickup)` in the middle of `build` ? Leaked. Date formatting is a presentation decision with a rule hiding inside it (time zone, locale, "today" versus "07/18"), and a duplicated decision diverges the same way the three totals diverged. The date arrives in the state as ready text.

Sorting the item list inside the screen, an `items.sort()` before the loop? Leaked. The sorting criterion (alphabetical? most recent first? drinks before food?) is the tab's business rule, and the definition of renderable state already said it: the list arrives sorted.

Deciding the inventory alert's color, a `stock < minimum ? red : green` ? Leaked, and this is the case that fools people most, because color looks like the screen's business. Comparing against the

minimum is the rule; what the screen can do is map an enum that arrived in the state (`alert: critical`) to the platform's color. Mapping the appearance of a ready value is the View's job; comparing to produce that value isn't.

Passing the total's text to the widget, the listing's `Text("Total: ${data.formattedTotal}")` ? View. There's no decision at all: the text came in ready and went out ready.

Firing `PayTab` when the barista taps the button? View, and it's its heart: turning a gesture into a business-named event is exactly the "fires events" from the table.

Five verdicts, one pattern: ask who's deciding. If the answer is "the screen," the code changes address.

The critique: bloated state

The most common objection to all this deserves to be stated in full before it gets answered: "A dumb View bloats the state and moves formatting to a place it doesn't belong. Formatting currency is presentation; presentation is the screen's job; a `TabData` full of ready-made strings is a model polluted with visual detail."

The answer starts by taking the premise apart. Formatting looks cosmetic and is a decision: choosing a comma or a period, two decimal places or none, "\$" before or after the number, is choosing how the business presents itself, and the coffee shop already paid to find out what a duplicated decision does. When formatting lived in the screens, it was three copies and three totals in QA. Moving it to whoever builds the state costs once: a `String` field instead of a `double`, one formatting function with a single owner. Leaving it in the screens costs every release, in every new screen that copies the rule, in every UI test that climbs

a widget tree to check a comma. Who formats is the orchestrator, while building the state, with a helper function it calls and that any test calls too, without ever booting a screen.

This trade isn't this book's invention. Michael Feathers documented it in 2002, in the paper "The Humble Dialog Box": keep the dialog humble, with no intelligence of its own, and move the decisions into a class you can test without a UI. Martin Fowler cataloged the same idea in 2006 under the name "Passive View": the screen reduced to a passive relay, updated from outside, with almost nothing left to get wrong. That's more than two decades of people pulling code out of the most expensive place to test, and the most expensive place to test is still the screen.

Here's my position, no fence-sitting: a widget test is no place for a business rule. I don't write a test that climbs a render tree to check whether a 10% discount came out right, and I get suspicious of any suite where the UI tests are the ones that break most, because that's a rule living in the screen giving itself away. With a dumb View, the rule gets a pure function test, and the View is left with so little inside it that there's almost nothing left to test in it: what's left is checking that state turns into widgets and gestures turn into events, and the compiler plus half a dozen thin tests cover that.

Pitfalls

The first pitfall is a business `if` disguised as "just a bit of formatting." The usual disguise: `Text(total, style: points >= 100 ? green : black)`. What goes wrong: the 100-point cutoff just got a second home, and the day it becomes 120 points someone will update the rule and forget the color, and the screen will paint green on a total that no longer earns a discount. Why: the comparison looks

like style because the result is a color, but the operand is a business value, and the wrong-layer test fails on the operand, not the result. How to get out: the state hands over the decision already made (`highlightTotal: true` , or an alert enum), and the screen only maps the value to the platform's style.

The second pitfall is reaching straight into the repository from the screen, “just this once,” a `TabRepository().fetchTab(table)` inside `initState` because the deadline is tight. What goes wrong: the screen becomes the owner of both the rule and the IO at once; it decides when to fetch, what to do with a failure, and how to store the result, and every one of those decisions turns untestable without booting a UI and faking a network. Why: the shortcut punches through both of the View row's prohibitions at once, and sets the precedent the next screen copies; chapter 15 hasn't even arrived and the repository already has a client it shouldn't have. How to get out: the screen fires a load event (or the orchestrator listens to navigation, chapter 13 shows both ways) and renders whatever state comes back, including the error one, which also arrives ready.

Q&A

If the screen can't calculate, who does? Chapter 13's orchestrator, which receives the event, triggers whoever knows the rule, and publishes the new state. This chapter kept it as a closed box on purpose: to the View, it's just “the place the state comes from,” and that ignorance is exactly what keeps the screen replaceable.

What about form validation? Can the email field turn red while I type? Immediate typing feedback can live in the screen as long as it's a shape check (“does this look like an email?”), with no business value involved. Validation that decides (“does this coupon exist? is this email already

registered?”) is a rule: it fires an event and comes back in the state. When in doubt, apply the test: what value is the `if` comparing against?

Can local focus and scroll state stay put? Yes, and it should live in the screen: focus, scroll, and animation are details no other layer has any reason to know about. The test is the same one from the refactor section: if Rosie doesn’t care, it’s the screen’s.

Quick tip

Open your most complex screen’s file and run your editor’s search three times: `if (`, `+`, and `format`. Every hit that compares, adds, or formats a business value is a candidate to change address, and the search costs less than a whole code review.

Quick reference

Situation	Fix
Formatting currency, date, or text	leaked: arrives ready (<code>formattedTotal</code>)
Sorting or filtering the list	leaked: the list arrives sorted in the state
Comparing business values to decide a color	leaked: the state hands over the enum
Enabling or disabling a button	ready boolean in the state (<code>canPay</code>)

Passing ready text to the widget	View
A tap turning into an event (<code>PayTab</code>)	View: “fires events”
Field focus, scroll position, animation	View: local, pure-UI state
Fetching data “just this once” in the screen	leaked: reread the second Pitfall

Exercises

- Classify each snippet below as “belongs in the View” or “leaked from another layer,” using the wrong-layer test; the answer key is in exercise 3. (a) `Text(data.customerName)` ; (b) `if (tab.items.length > 10) showFullTableWarning()` ; (c) `onPressed: () => onEmit(RemoveItem(table, item.name))` ; (d) `priceInCents / 100` inside a price widget; (e) an `AnimationController` driving the item panel’s opening.
- The coffee shop’s pickup counter screen receives the raw order list and does three things: sorts by promised time, marks the late ones red by comparing against the clock, and formats the time as “HH:mm”. Refactor it on paper: sketch the `PickupCounterState` (which fields? which types?) and the events the screen fires when the attendant taps an order. When you’re done, check: did any `if` with a business value survive in the screen?
- Answer key for exercise 1: (a) View, ready text passed along; (b) leaked, it compares a business quantity to decide something; (c) View, a gesture turning into an event; (d) leaked, money arithmetic is both math and formatting; (e)

View, animation is local UI state. Now the open challenge: grab a real screen from one of your own projects, run the Quick tip on it, and count how many business decisions you find; then write the renderable state that would leave that screen dumb.

Tip 12

If the screen decides, you don't have a View: you have a rule hiding where it's most expensive to test.

Next chapter: the closed box opens. Chapter 13 builds the orchestrator that receives `AddItem` and returns ready `TabData`: event in, state out, and you'll see what happens in between.

The Orchestrator: Event In, State Out

In this chapter, you'll:

- write the tab's orchestrator that receives `AddItem` and publishes the right sequence of states, with no business rule tucked inside;
- test the event-to-states flow without booting a single screen, with a test that breaks the moment the publish order flips;
- point, inside someone else's orchestrator, at the exact line where a rule snuck in where only a connection belonged.

You've chased a total that flickered wrong on one screen and right on another, unable to say who changed the value. The problem wasn't the value: it was the number of paths leading to it. This chapter builds the piece that narrows those paths down to one: the Orchestrator, the box chapter 12 left closed. An event goes in at the top, a state comes out at the bottom, and nothing else happens.

Chapter 12 ended with the dumb screen firing `TabEvent` into a dashed box and getting `TabData` back. That box's contract has been signed since chapter 10, on the Orchestrator's line of the canonical table:

Orchestrator	Contract
--------------	----------

Does	converts event to state
Does	fetches data from the repository
Does	calls use cases
Does	publishes state
Forbids	deciding rules
Forbids	persisting

Four verbs and two bans. Each one deserves its own paragraph, because every word in that line was paid for with somebody's bug.

“Converts event to state” is the job description in one line. The Orchestrator is a translation function with effects in the middle: a gesture named after the business goes in (`AddItem`), and something the screen draws without thinking comes out. It never returns raw data, never returns an exception, never returns “void and good luck.”

“Fetches data from the repository” says where the raw material comes from. In this chapter's tab, the conversation is a question and an answer: the Orchestrator asks for table 4's tab, the repository hands back table 4's tab, done. That's not a ban on streams. When the data changes by someone else's hand (another waiter writing to the same tab on a local database, a GraphQL subscription), the repository is free to expose a stream, and the Orchestrator is the one who subscribes to it and translates every emission into state. What decides the contract is the source: a local database and GraphQL know how to announce that something changed; a REST or RPC API has no way to, and faking reactivity on top of either turns into polling wearing a disguise. Chapter 15 shows when each contract fits.

“Calls use cases” points at whoever knows the rules. The Orchestrator hands over the tab it fetched and the event’s item to the use case and waits for a verdict. It never second-guesses the verdict. If the loyalty discount changed, if the item dropped off the menu, if the tab hit some cap: none of that is the Orchestrator’s problem.

“Publishes state” closes the cycle, and hides the line’s most important word: publishes. The screen doesn’t ask; it subscribes. Wherever reactivity exists in the system, this is where it lives: the Orchestrator is the sole owner of the channel that carries state down to the View, and that’s exactly why chapter 12’s screen could stay so dumb.

The bans close the back doors. “Deciding rules” is forbidden because a rule already has its own house (the use case, chapter 14), and a rule outside that house turns into a copy, the same way chapter 12’s three screens grew three totals. “Persisting” is forbidden because saving data is the repository’s trade (chapter 15), and an orchestrator that saves on its own opens a second write path nobody audits. The Orchestrator knows *when* and *where to*. It never knows *what*.

Where the one-way flow came from

This discipline has a birth certificate, and it helps to remember why each ban exists. In 2014, Facebook lived with a bug every user of the era saw: the notification counter at the top of the page announced a new message, you opened the chat, there was no message at all, and a few minutes later the counter accused a phantom again. Engineering fixed it, and the bug came back in the next release. The cause, told by Facebook’s own team at the Flux talk (F8 conference, 2014) and retold on the “Prior Art” page of the Redux documentation: models and views updating each

other, every screen holding write permission on the state the others read. Nobody could answer “who changed this value?”, because the answer was “anyone.”

Facebook’s fix was radical: ban scattered writes and force every state change through a single path, always running the same direction. That design has a name: **unidirectional data flow** is the architecture where state changes through only one path, always in the same direction: the interface fires an intent, a single responsible party processes it and publishes the new state, the interface renders whatever arrived. No shortcuts, no side writes, no screen talking to screen.

The idea wasn’t born at Facebook, and it didn’t die there either. It got rediscovered so many times, by people solving the same problem, that each generation gave it a new name. The five stops are worth knowing, because you’ll meet all five names in job listings, and they’re all the same design.

Elm came first. Around 2013, Evan Czaplicki structured the Elm Architecture: every program is a `Model`, an `update` function that takes a message and returns the new model, and a `view` that draws the model. The official Elm guide (guide.elm-lang.org/architecture) still presents it today as the pattern the others derived from. The problem it solved: in Elm there’s no mutation, so “who changes the state?” has to have a single answer by construction.

Flux came in 2014, out of the counter bug: Facebook needed an answer to “who changed this value?” across a giant JavaScript codebase. Actions, a dispatcher, and stores that are the sole owners of state. Flux was the one that popularized the slogan this chapter keeps repeating: data flows in one direction.

Redux, in 2015, is Dan Abramov and Andrew Clark simplifying Flux “following Elm’s lead,” in the documentation’s own words (redux.js.org, “Prior Art”): one store, pure reducers, immutable state. The problem it solved was ergonomics: the original Flux had too many moving parts, and Redux proved three were enough.

MVI (Model-View-Intent) formalized the same cycle as an observable flow. André Staltz described the whole family in “Unidirectional User Interface Architectures” (2015, staltz.com), showing that Elm, Flux, Redux, and his own Cycle.js were variations on one cycle; the Android community adopted the name MVI for that variation. The problem it solved: giving the cycle a reactive shape, where even the user’s intent is a stream.

BLoC closes the lineage in 2018. Paolo Soares and Cong Hui, from Google, presented the **Business Logic Component** at DartConf 2018, in a talk about sharing code between Flutter and AngularDart: events go in through one stream, states come out through another, and nothing else crosses the boundary. The problem it solved: the same logic serving two different interfaces without a rewrite.

Five names, one design: event goes in, state comes out, and state changes through one path only. FOCUS’s Orchestrator is that design’s seat in the canonical table, with one detail the rest of this chapter explores: in FOCUS, the “single responsible party” connects, but the use case decides.

The anti-solution: the orchestrator that decides

Like every chapter in this part, the right path starts with the wrong one. Two pains, in the table line’s two bans.

The first pain is a rule sneaking in. The orchestrator below receives the event, fetches, calls the use case, and publishes: all correct, until the middle of the switch:

Dart

```
// The anti-solution: the right cycle, with a rule smuggled into the
```

```
// middle.
```

```
class TabOrchestrator extends Bloc<TabEvent, TabState> {
```

```
  TabOrchestrator(this._repository, this._loyaltyPoints)
```

```
    : super>Loading() {
```

```
    on<AddItem>(_onAddItem);
```

```
}
```

```
final TabRepository _repository;
```

```
final int _loyaltyPoints;
```

```
Future<void> _onAddItem(
```

```
  AddItem event,
```



```
Emitter<TabState> emit,  
  
) async {  
  
    emit>Loading());  
  
    final lookup = _repository.findTab(event.table);  
  
    switch (lookup) {  
  
        case InfraFailure():  
  
            emit(Failed("no connection"));  
  
        case TabFound(:final tab):  
  
            final result = addItemToTab(tab, event.item);  
  
            switch (result) {  
  
                case Success(:final value):  
  
                    // The smuggled rule: the loyalty discount computed right  
  
                    // here, comparing business data in the middle of a wire.
```

```
var total = value.totalInCents;

if (_loyaltyPoints >= 100) {

    total = total * 90 ~/ 100;

}

emit(

    Ready(

        toData(

            Tab(value.table, value.items, total, value.canPay),

        ),

    ),

);

case RuleViolated(:final reason):

    emit(Failed(reason));

}
```

```
}  
  
}  
  
}
```

The `if` works, and that's the danger. The 100-point cutoff and the 10% markdown now live in two places: in the use case chapter 14 is going to write, and in this switch. The day Rosie changes the discount, one of the two falls behind, and the app starts showing one total at the register and another in the kitchen. You already watched this movie in chapter 12; the difference is that the copy isn't sitting on a screen this time, it's sitting in the middle of the wire, which is harder to catch in code review.

The second pain is the local edition of Facebook's bug. Two screens show the total of the same tab, and someone solved the sharing problem the fast way: a global mutable object.

Dart

```
// Pain (b): the mutable state two screens share. Each screen writes  
  
// straight into the total; neither knows what the other did.  
  
class SharedState {  
  
  int totalInCents = 0;  
  
  int loyaltyPoints = 0;
```

```
}
```

```
final globalState = SharedState();
```

```
// The waiter's screen adds the item straight into the global state...
```

```
void waiterScreenAddsItem(int priceInCents) {
```

```
    globalState.totalInCents += priceInCents;
```

```
}
```

```
// ...and the cashier's screen applies the discount on its own. Run it
```

```
// twice and the discount lands twice: the total flickers wrong on the
```

```
// other screen.
```

```
void cashierScreenAppliesDiscount() {
```

```
    if (globalState.loyaltyPoints >= 100) {
```

```
        globalState.totalInCents = globalState.totalInCents * 90 ~/ 100;
```

```
}
```

}

Trace the flow slowly. The waiter adds an item: their screen writes to the total. The cashier opens the payment screen: it applies the discount and writes to the same total. The waiter adds another item: their screen sums on top of a value that's already discounted. The cashier refreshes: discount again, now doubled. The total flickers wrong, right, wrong, and neither snippet has an isolated bug; the bug is the scattered write permission. It's Facebook's notification counter at coffee-shop scale, and the fix is the same one from 2014: cut every write path but one.

Both pains share a diagnosis, and it deserves a name. **Connect vs. decide** is the test that separates what the Orchestrator does from what it hands off: connecting means knowing *when* to act and *where* to send it (`AddItem` arrived, fetch the tab, call the use case, publish the result); deciding means knowing *what* the rule says (100 points, 10%, an item off the menu). Every line inside an orchestrator answers one of those two questions. If it answers “what,” it's in the wrong place.

Events and states as sealed classes

The refactor starts with the types, and the types are the half you already have. `TabEvent` came ready-made from chapter 12, and its spelling doesn't change again. Chapter 13's new piece is the type the Orchestrator publishes: a sealed hierarchy of states.

 Dart ·  Kotlin

```
// The View's only output: business-named events, spelled the way  
  
// chapter 12 fixed them.
```

```
sealed class TabEvent {}
```

```
final class AddItem extends TabEvent {
```

```
    AddItem(this.table, this.item);
```

```
    final int table;
```

```
    final String item;
```

```
}
```

```
final class RemoveItem extends TabEvent {
```

```
    RemoveItem(this.table, this.item);
```

```
    final int table;
```

```
    final String item;
```

```
}
```

```
final class PayTab extends TabEvent {
```

```
PayTab(this.table);
```

```
final int table;
```

```
}
```

```
// What the Orchestrator publishes: the cycle's state, new in chapter 13.
```

```
sealed class TabState {}
```

```
final class Loading extends TabState {}
```

```
final class Ready extends TabState {
```

```
    Ready(this.data);
```

```
    final TabData data;
```

```
}
```

```
final class Failed extends TabState {
```

```
Failed(this.message);

final String message;

}
```

Kotlin writes the same contract with `sealed interface` and `data class`, one line per type: `data class Ready(val data: TabData) : TabState`. Same names, same shape, half the lines.

Notice that `TabState` doesn't replace `TabData`; it wraps it. Chapter 12's renderable data survives intact as `Ready`'s payload, and the other two states exist because the real cycle has time and failure: between the waiter's tap and the finished data there's `Loading`, and when the use case says no there's `Failed` with a message ready to show. The screen renders the state with a three-armed switch, and the compiler guarantees no arm is missing.

This type already showed up once, under another name and a smaller frame. Chapter 10 called the value the Orchestrator published `TabResult`, and reused that same type in two other spots: the use case's return and the repository's return. That fit on one page because it was a sketch, and from here on the three things split apart, each one wearing its own trade's name. What the screen subscribes to is this `TabState`. The use case's verdict is a `Result` (chapter 14). The repository's answer is a `LookupResult` (chapter 15). The sketch also had no `Loading`, because it had no time: nobody was waiting on anything, and now somebody is.

Keep this listing's size in mind, because it's an answer. The most repeated criticism against Redux, MVI, and their relatives is boilerplate: "a class for every event, one for every state, too much ceremony." The entire tab's set of events and states just fit on

half a page, and every one of those classes buys something no annotation can: the compiler now knows every possible case. The criticism section returns to this point with sources; for now, the Try It below is worth more than the argument.

Try it: open <https://focus.kodel.com.br/en/dart/13-01> (or <https://focus.kodel.com.br/en/kotlin/13-01>) and add a new state to the hierarchy: `final class Cancelled extends TabState {}`. Prediction: the code doesn't even run; the compiler flags the switch in `renderState` as non-exhaustive, naming the missing variant. In Kotlin, do the same with a new event in the orchestrator's `when`: the error is identical. That free warning, at every point in the system that consumes the type, is what the “extra” classes buy. The other eight languages live at `/en/ts/13-01`, `/en/java/13-01`, `/en/csharp/13-01`, `/en/go/13-01`, `/en/php/13-01`, `/en/python/13-01`, `/en/swift/13-01`, and `/en/rust/13-01`.

The complete orchestrator

With the types in place, the orchestrator that only connects fits in one class. In Dart, the implementation uses `package:bloc`, the pattern's dominant library in the Flutter ecosystem; the book's prose keeps saying “orchestrator,” because chapter 10 already showed that BLoC, ViewModel, and Store are the same job under different badges.

Dart

```
// The orchestrator only connects: receives the event, fetches, calls  
  
// the use case, publishes the state. No rule, no persistence.
```

```
class TabOrchestrator extends Bloc<TabEvent, TabState> {

  TabOrchestrator(this._repository) : super>Loading() {

    on<AddItem>(_onAddItem);

    on<RemoveItem>(_onRemoveItem);

  }

  final TabRepository _repository;

  Future<void> _onAddItem(

    AddItem event,

    Emitter<TabState> emit,

  ) async {

    // Publishes the waiting state: the screen reacts before the data

    // arrives.

    emit>Loading();
```

```
// Fetches the tab: the read can now fail over the network, so the

// repository returns a LookupResult (chapter 15), with two

// variants.

final lookup = _repository.findTab(event.table);

switch (lookup) {

    case TabFound(:final tab):

        // Calls whoever knows the rule: the use case (chapter 14).

        final result = addItemToTab(tab, event.item);

        // Translates the Result into state: exhaustive switch, as in

        // chapter 8.

        switch (result) {

            case Success(:final value):

                emit(Ready(toData(value)));

            case RuleViolated(:final reason):
```

```

        emit(Failed(reason));

    }

    case InfraFailure():

        emit(Failed("no connection"));

    }

}

Future<void> _onRemoveItem(

    RemoveItem event,

    Emitter<TabState> emit,

) async {

    // Same cycle, only the use case changes; the fetch can fail over

    // the network here too.

    emit>Loading());

    final lookup = _repository.findTab(event.table);

```

```
switch (lookup) {

    case TabFound(:final tab):

        final result = removeItemFromTab(tab, event.item);

        switch (result) {

            case Success(:final value):

                emit(Ready(toData(value)));

            case RuleViolated(:final reason):

                emit(Failed(reason));

        }

    case InfraFailure():

        emit(Failed("no connection"));

    }

}
```

Read `_onAddItem` as a checklist against the table's line. It publishes `Loading`: the screen shows a waiting indicator without knowing what's being waited on. It fetches the tab: one question to the repository, one answer, the contract this stub offers. It calls `addItemToTab`: the use case receives the tab and the item, and whatever it decides comes back as a `Result`. It runs chapter 8's exhaustive switch: `Success` becomes `Ready`, `RuleViolated` becomes `Failed`, and the compiler confirms no variant went untranslated. Four steps, zero decisions. The whole method survives any change to the coffee shop's rules without a single edited line.

The two functions the Orchestrator calls are closed boxes, and they stay closed until the chapters ahead. From the use case, chapter 13 knows only the signature, and that signature is a contract: chapter 14 implements exactly this function, with the loyalty discount living inside it.

Dart

```
Result<Tab> addItemToTab(Tab tab, String item) {
```

From the repository, the same story, with one difference chapter 15's revision brought: `findTab(int table)` no longer returns the raw tab. Since the read can now fail over the network, the return grew from `Future<Tab>` into `LookupResult`, a synchronous `Result` with two variants, `TabFound` and `InfraFailure`. The closed box's contract now declares those types:

Dart

```
enum Failure { noConnection }
```

```
sealed class LookupResult {}

final class TabFound extends LookupResult {

    TabFound(this.tab);

    final Tab tab;

}

final class InfraFailure extends LookupResult {

    InfraFailure(this.failure);

    final Failure failure;

}

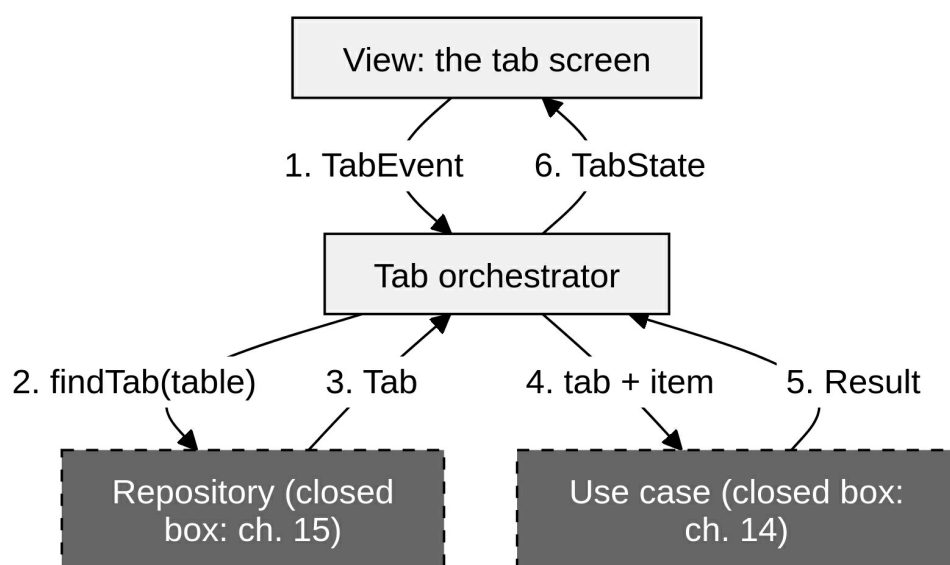
abstract interface class TabRepository {

    LookupResult findTab(int table);

}
```

The full definition of these types, along with the single exception translation that produces them, is chapter 15's business. In this chapter's code, both have provisional bodies (the use case adds up prices from a fixed table; the in-memory repository returns `TabFound` with a seed tab), enough for the whole cycle to compile and run today.

The complete cycle, with the closed boxes marked:



Six numbered arrows, and every state change in the system travels all six, in order, every time. When table 4's total shows up wrong, you know where to look, because only one path could have carried it there. Compare that with pain (b): there, “who changed this value?” answered “any screen”; here, the answer is a four-step method.

The design is the same across the book's ten languages; what changes is the mechanism that publishes the state, and each platform has its own established one. Kotlin, in the Android world, publishes with `StateFlow`, the mechanism the official documentation recommends for UI state; the orchestrator turns into a plain class with an exhaustive `when` over the event:



```
class TabOrchestrator(  
  
    private val repository: TabRepository,  
  
) {  
  
    private val mutableState = MutableStateFlow<TabState>(Loading)  
  
    val state: StateFlow<TabState> = mutableState  
  
    suspend fun onReceive(event: TabEvent) {  
  
        when (event) {  
  
            is AddItem -> onAddItem(event)  
  
            is RemoveItem -> onRemoveItem(event)  
  
            is PayTab -> Unit  
  
        }  
  
    }  
}
```

```
private suspend fun onAddItem(event: AddItem) {

    // Publishes the waiting state: the screen reacts before the

    // data arrives.

    mutableState.value = Loading

    // Fetches the tab: the read can now fail over the network,

    // and the repository returns LookupResult (chapter 15).

    when (val lookup = repository.findTab(event.tab)) {

        is TabFound -> {

            // Calls whoever knows the rule: the use case (ch. 14).

            val result =

                addItemToTab(lookup.tab, event.item)

            // Translates the Result into state: exhaustive when,

            // chapter 8.

            mutableState.value = when (result) {
```

```

        is Success -> Ready(toData(result.value))

        is RuleViolated -> Failed(result.reason)

    }

}

is InfraFailure -> mutableState.value = Failed("no connection")

}

}

}

```

The visible difference from Dart is where the dispatch lives: `package:bloc` registers one handler per event with `on<AddItem>`, while Kotlin receives everything in `onReceive` and routes it through a `when`. The `when` has a teaching advantage: when you create a new event, it's the one the compiler flags as incomplete.

TypeScript has the most famous mechanism of all: Redux Toolkit, the official successor of the original Redux. The state is the store's state, `createSlice` generates the publishing actions, and the Orchestrator is the async function that connects:

TS TypeScript

```

// The state is the slice's state; the reducer just swaps the

```

```
// published state.

const stateSlice = createSlice({

  name: "state",

  initialState: { type: "loading" } as TabState,

  reducers: {

    loading(): TabState {

      return { type: "loading" };

    },

    ready(_, action: PayloadAction<TabData>): TabState {

      return { type: "ready", data: action.payload };

    },

    failed(_, action: PayloadAction<string>): TabState {

      return { type: "failed", message: action.payload };

    },

  },

});
```

```
});
```

Notice what the reducers do: they swap the state, and nothing else. The fetch, the use case call, and the switch on the Result live in an `orchestrate` function outside the slice, which ends with `store.dispatch(ready(...))` or `store.dispatch(failed(...))`. That split is a stance on where rules belong, and the criticism section returns to it, because the Redux documentation recommends the opposite.

Java doesn't have a dominant reactive mechanism outside front-end frameworks, so the port uses the design the market calls MVVM (Model-View-ViewModel) with the standard library's own property notification, `PropertyChangeSupport` from `java.beans`: the orchestrator keeps the state and fires the listeners on every publish:

Java

```
// Publishes the state: the "state" property notifies listeners.

// The previous value goes in as null: equal states published back

// to back still notify, because firePropertyChange normally

// silences equal values, and null defeats that check on purpose.

private void publish(TabState next) {

    state = next;

    support.firePropertyChange("state", null, next);
```

```
}
```

The rest of the class is the same four-step cycle, with pattern matching over sealed records (Java 21), exhaustive just like Dart's.

C# is MVVM's home turf: the notification interface ships with the platform (`INotifyPropertyChanged` , from `System.ComponentModel`), and the orchestrator is a `ViewModel` whose `State` property notifies from its setter:



C#

```
public TabState State

{

    get => _state;

    private set

    {

        _state = value;

        PropertyChanged?.Invoke(

            this, new PropertyChangedEventArgs(nameof(State)));

    }

}
```

```
}
```

C#'s caveat is the same one from chapter 8: record hierarchies carry no guaranteed exhaustiveness, so the switch over the Result carries a `_` arm that throws. The compiler doesn't bill you for the missing case; the extra arm bills you at runtime.

PHP and Python don't have an established BLoC-style mechanism, and the book makes the same choice for both: a hand-rolled ViewModel, with registered observers and notification at a single publish point. In PHP:

PHP

```
private function publish(TabState $state): void
{
    $this->state = $state;

    foreach ($this->observers as $observer) {
        $observer($this->state);
    }
}
```

In Python, the same design with callbacks, plus 3.10's structural `match` playing the switch's part:



```
# Translates the Result into state: structural match, chapter 8.

match result:

    case Success(value):

        self._publish(Ready(to_data(value)))

    case RuleViolated(reason):

        self._publish(Failed(reason))

    case _:

        impossible_variant(result)
```

In both, exhaustiveness is the type checker's job (Psalm or PHPStan on one side, mypy or pyright on the other), not the interpreter's; the `case _` that throws exists for the day someone runs the code without checking.

Swift publishes with SwiftUI's standard mechanism: `ObservableObject` and `@Published`, from Combine. The whole orchestrator ends up looking like an iOS ViewModel, and the enums with associated values give the family's cleanest exhaustive switch:




```
final class TabOrchestrator: ObservableObject {

    @Published var state: TabState = .loading

    private let repository: TabRepository

    init(repository: TabRepository) {

        self.repository = repository

    }

    func send(_ event: TabEvent) {

        switch event {

        case .addItem(let table, let item):

            runCycle(table: table, item: item, useCase: addItemToTab)

        case .removeItem(let table, let item):

            runCycle(table: table, item: item, useCase: removeItemFromTab)

        case .payTab:
```

```

        state = .failed("paying the tab arrives in chapter 14")

    }

}

```

Rust skips the framework, just like chapter 12: the state goes out through a standard-library channel (`std::sync::mpsc`), and `main` is a render loop consuming that channel. The port's nicest detail is that chapter 8's `Result` never needed a declaration, because in Rust it's the language's own `Result<T, E>` type: `Ok` plays `Success`'s part and `Err` carries `RuleViolated`'s reason:

Rust

```

// Translates the Result into state: exhaustive match, chapter 8.

match result {

    Ok(tab) => {

        publisher

            .send(TabState::Ready(to_data(&tab)))

            .unwrap();

    }

    Err(reason) => {

```

```
publisher.send(TabState::Failed(reason)).unwrap();
```

```
}
```

```
}
```

Nine languages so far, and the same inventory as always: the publishing mechanism changes name (bloc's stream, `StateFlow`, Redux's store, `PropertyChangeSupport`, `INotifyPropertyChanged`, a hand-rolled observer, `@Published`, a channel), and the cycle's four steps never change. The tenth language earned its own section, because it's missing a piece.

Try it: open <https://focus.kodel.com.br/en/dart/13-02> (or <https://focus.kodel.com.br/en/kotlin/13-02>) and run it. Prediction: the console prints “loading...” and then the tab with Espresso, Cappuccino, and the \$18.95 total, without a single line of View code adding anything up; after that, a second event with an item that's not on the menu prints the failure state with the ready-made message. Swap “Cappuccino” for “cheese bread” and predict the total before you run it. The other eight languages live at </en/ts/13-02>, </en/java/13-02>, </en/csharp/13-02>, </en/go/13-02>, </en/php/13-02>, </en/python/13-02>, </en/swift/13-02>, and </en/rust/13-02>.

Go's counterpoint: no unions, no billing

Go can write the same orchestrator, and the listing proves it; what Go can't do is force you to keep it complete. The language has no union types and no sealed classes: the idiom for “one of N

types” is a marker interface with a private method, which keeps outside packages from implementing it but never tells the compiler how many implementations exist.



Go

```
// Events: no sealed class, so the idiom is a marker interface with a
// private method. Only types in this package can be an Event.

type Event interface {

    event()

}

type AddItem struct {

    Table int

    Item  string

}

type RemoveItem struct {

    Table int
```

```

    Item string

}

type PayTab struct {

    Table int

}

func (AddItem) event() {}

func (RemoveItem) event() {}

func (PayTab) event() {}

```

The cycle is the usual one, and publishing goes out through a channel, Go's native reactive mechanism:



Go

```

func Orchestrate(event Event, states chan<- State) {

    switch e := event.(type) {

    case AddItem:

```

```
states <- Loading{}

// The read can now fail over the network: FindTab returns

// LookupResult (chapter 15), with two variants.

switch lookup := FindTab(e.Table).(type) {

case TabFound:

    result, err := AddItemToTab(lookup.Tab, e.Item)

    if err != nil {

        states <- Failed{err.Error()}

        return

    }

    states <- Ready{ToData(result)}

case InfraFailure:
```

```
states <- Failed{"no connection"}
```

```
}
```

```
}
```

```
}
```

Now repeat the Try It from the sealed-classes section, here. In Dart and Kotlin, the new variant broke the build at every incomplete switch. In Go, look at the type switch above: `RemoveItem` and `PayTab` exist in the package and have no arm, and `go vet` on this exact file comes back clean. The event goes in, lands in no arm, and the function returns in silence: no state published, no error, a screen waiting forever for a state that never comes.

The reason for the difference is a language design choice. Go treats “one of N outcomes” with the pair `(T, error)`, which this port’s own use case relies on and which chapter 8 introduced as Go’s native Result; for “one of N types,” the official answer is the type switch, and exhaustiveness never made it into the contract. What the Go team does about it is what the community recommends: convention (every type switch over `Event` covers every event, and code review enforces it) and testing (one flow test per event; the next section’s test is the template). The pattern’s cost didn’t drop in Go; it just left the compiler and landed on your process. That’s going to be a central argument two sections from now.

The flow test: event on top, states below

The whole chapter's promise fits in one test. If every state change travels a single path, then declaring “this event goes in, this sequence of states comes out” has to be enough to test the Orchestrator, with no screen involved. In Dart, `bloc_test` (the testing companion of `package:bloc`) writes exactly that sentence:

Dart

```
blocTest<TabOrchestrator, TabState>(  
  
  "AddItem publishes Loading then Ready, in that order",  
  
  build: () => TabOrchestrator(InMemoryTabRepository()),  
  
  act: (orchestrator) => orchestrator.add(AddItem(4, "Cappuccino")),  
  
  expect: () => [  
  
    isA<Loading>(),  
  
    isA<Ready>().having(  
  
      (state) => state.data.formattedTotal,  
  
      "formattedTotal",  
  
      "\$18.95",  
  
    ),  
  
  ],
```



```
);
```

Read the three parts. `build` sets up the orchestrator with the in-memory stub: no database, no framework mock. `act` delivers the event, the same gesture the screen would make. `expect` declares the sequence: `Loading` first, then `Ready` with the formatted total the screen is going to show. No `pumpWidget`, no render tree, no animation wait; the whole test file doesn't import a single UI concern.

This test is only possible because the Orchestrator is dumb. An orchestrator with a rule inside would force you to test the rule right here, one case per discount tier; an orchestrator with shared state would force you to set up both screens to reproduce the flicker. The orchestrator that only connects narrows the test down to the contract: event goes in, states come out, in this order. And the order carries weight: flip the publishes inside `_onAddItem` (emit the ready state before `Loading`) and this test fails on the spot: it points at the sequence it actually received. The Pitfalls section shows how this exact bug is born on its own inside async code, with nobody flipping anything on purpose.

The flow test above has a deliberate limit: it exercises the **happy path**, the run where everything works, the item exists, the repository answers, the total comes out. A test that stops there is half a test. Code spends its life on the other paths: the invalid input, the rule that says no, the value sitting right on the limit; a good test set builds every possible scenario, failures included, because the scenario nobody visits is where a bug ages in peace.

Writing a whole test per scenario gets tiring, and test frameworks fix that with parametrization: you declare a list of cases, each with its input and expected verdict, and the framework turns every row into a test. JUnit does it with

`@ParameterizedTest`, pytest with `parametrize`; Dart's `package:test` needs no annotation at all, because `test` is an ordinary function and a `for` over the list is enough. The tab's use case, tested this way:

Dart

```
typedef Case = ({  
  
  String description,  
  
  List<OrderItem> items,  
  
  String item,  
  
  int? expectedTotal,  
  
  String? expectedReason,  
  
});  
  
void main() {  
  
  const cases = <Case>[  
  
    (  
  
      description: "happy path: a menu item adds to the total",  
  
      items: [(name: "Espresso", priceInCents: 700)],  

```

```
    item: "Cappuccino",

    expectedTotal: 1895,

    expectedReason: null,

),

(

    description: "empty tab: the first item enables paying",

    items: [],

    item: "cheese bread",

    expectedTotal: 600,

    expectedReason: null,

),

(

    description: "repeated item: adds again, no deduplication",

    items: [(name: "Cappuccino", priceInCents: 1195)],

    item: "Cappuccino",
```

```
    expectedTotal: 2390,

    expectedReason: null,

),

(

    description: "item off the menu: rule violated, with a reason",

    items: [(name: "Espresso", priceInCents: 700)],

    item: "Feijoadá",

    expectedTotal: null,

    expectedReason: "Feijoadá is not on the menu",

),

(

    description: "empty text: also an item off the menu",

    items: [],

    item: "",

    expectedTotal: null,
```

```
        expectedReason: " is not on the menu",

    ),

];

for (final testCase in cases) {

    test(testCase.description, () {

        final total =

            testCase.items.fold(0, (s, i) => s + i.priceInCents);

        final tab = Tab(4, testCase.items, total, testCase.items.isNotEmpty);

        final result = addItemToTab(tab, testCase.item);

        switch (result) {

            case Success(:final value):

                expect(value.totalInCents, testCase.expectedTotal);

                expect(value.canPay, isTrue);

            case RuleViolated(:final reason):
```

```
expect(reason, testCase.expectedReason);
```

```
}
```

```
});
```

```
}
```

```
}
```

Five rows in the table, five generated tests, and only the first one is the happy path. The other four visit the empty tab, the repeated item, the rejected rule, and the degenerate input (the empty string), and each one costs a single row of data. Notice too the division of labor between this chapter's two test files: the flow test asks the Orchestrator "do the states come out in the right order?", and the parametrized test asks the use case "is the verdict right for every input?" Each one interrogates its own layer, neither boots a UI, and both run in milliseconds. When chapter 14 writes the loyalty discount inside this same use case, this table is going to gain the boundary rows every discount deserves: 99 points with no discount, 100 points with one, and the order that only goes through when no item on the tab is out of stock.

Transient context and persistent context

This chapter's flow moves two kinds of information that are easy to confuse, and the confusion is expensive in both directions.

Transient context is what holds while someone is working the screen: `Loading`, the item that was just refused, the tab with the seven items being assembled right now. It is born when the screen opens and dies when the app closes, and that is exactly what the orchestrator publishes. **Persistent context** is what has to still be true tomorrow morning: the saved tab, the customer's point balance, the confirmed payment. That one isn't published, it's stored, and what stores it is the repository.

Swapping one for the other produces bugs from two different families. Treating the persistent as transient is the item Rosie added at three in the afternoon that lived only in the orchestrator's state: the phone ran out of battery and the order vanished, with no error on screen. Treating the transient as persistent is writing on every keystroke, with the database receiving half a tab and the screen waiting on the confirmation of a write nobody asked for.

The ruler is the previous paragraph's question, and it fits on one line: does this have to survive the app closing? If it does, it crosses the use case and goes to the repository before it becomes state. If it doesn't, it's state and that's it, and the orchestrator owns it. Whoever reads the orchestrator's file finds the answer without opening anything else, because the list of states is the list of what's ephemeral in that slice.

The criticism: boilerplate and rules in the reducer

A pattern with this many names collects criticism, and the two strongest ones deserve a sourced answer.

The first is boilerplate. Redux's own documentation keeps an FAQ page about code structure (redux.js.org/faq/code-structure) that exists because the question "why does one interaction need so many files?" never stopped arriving; back in classic Redux, every interaction demanded an action constant, an action creator, a reducer, and a selector, scattered across four folders. The criticism was fair, and its answer sits in the sealed-classes section: the tab's complete set of events and states fit on half a page of sealed classes, and every one of those classes buys exhaustiveness checking at every switch in the system. The 2016 Redux boilerplate came from the language: JavaScript with no unions, no pattern matching, and no compiler to bill you for missed cases. The pattern was never at fault. The proof is the language that fixed the problem: Redux Toolkit, from the same maintainers, eliminated the four folders, and what's left in modern TypeScript is close to the Dart listing wearing different syntax.

Here's my scar, so you know where I'm speaking from: I wrote Redux in 2016, with actions named by string, a 400-line switch in a reducer three teams edited, and a typo in `"ADD_ITEM_SUCESS"` that spent six hours pretending to be a network bug, because a misnamed action breaks nothing, it just never fires. I'll take three extra sealed classes over that any day of the week. The compiler that flags my incomplete switch works for free; the intern hunting a string typo doesn't.

The second criticism attacks from the other direction: "if the Orchestrator decides nothing, it's a bureaucratic layer; the logic could just live there." That's not a strawman, it's Redux's official recommendation: the documentation's style guide (redux.js.org/style-guide) carries the rule "Put as Much Logic as Possible in Reducers." FOCUS disagrees on purpose, and the reason starts with the Orchestrator's job description: BLoC, ViewModel, and Store are support code for the View, born tied to

the screen they serve. A business rule written there inherits that leash, and the leash charges you twice. When a second screen needs the same rule, either the rule gets copied into the second orchestrator (chapter 12's three screens with three totals, one layer down), or one orchestrator starts calling the other, and then a piece that holds a rule depends on another piece that holds a rule: coupling between siblings in the same layer, which the bloc library's own documentation says to avoid at any cost ("no bloc should know about any other bloc," bloclibrary.dev, architecture page).

The way out of both dead ends is the same one, and it isn't exclusive to FOCUS: Flutter's official architecture guide (docs.flutter.dev/app-architecture) recommends extracting use cases exactly when logic "will be reused by different view models" or needs to combine data from more than one repository. That's the table's line said with different words: the rule leaves the Orchestrator and becomes a reusable verb, a pure function, testable with no framework, that any orchestrator can call without knowing about the others (the waiter's screen and the cashier's report both call the same `addItemToTab`). The smell that gives away the divergence in practice, you already saw in the anti-solution: an `if` comparing a business value inside the Orchestrator. The fix is always the same change of address, and chapter 14 is entirely about the right address.

Pitfalls

The first pitfall is a rule disguised as "just one if." It never introduces itself as a business rule; it arrives as a "tiny validation": `if (event.item.isEmpty) return;` at the top of the handler, "not even worth calling the use case for." What goes wrong: the criterion for a valid item just gained a second home, and when the rule grows (a duplicate item? an item from another shift?)

the `if` grows with it, in the wrong place. How to get out: run the connect vs. decide test on the operand. An `if` that compares a business value (item, price, quantity) is a decision, and decisions belong in the use case, which returns `RuleViolated` with a ready message.

The second pitfall is persistence disguised as “just a cache.” The Orchestrator just received the updated tab from the use case, and stashing it in a map “so the next fetch is instant” looks like harmless optimization. What goes wrong: a second place where the tab lives was just born, and it doesn’t talk to the repository; the next screen that fetches straight from the repository gets the stale version, and the total flickers again. Why: a cache is persistence with an expiration date, and the table’s line forbids persisting, deadline or not. How to get out: the tab’s keeper is chapter 15’s repository, which is free to keep its own internal cache, in one place, invisible to whoever fetches.

The third pitfall is state emitted out of order inside async code, and this one shows up with nobody writing wrong code on purpose. The waiter taps twice: two events go in, two async cycles start, and if the first cycle is waiting on a slow repository while the second one hits a cached value, the second one’s state can land before the first one’s. What you see on screen: the right total shows up, and half a second later a late `Loading` runs it over. How to get out: first, have the flow test, because it’s the one that turns “the total sometimes flickers” into a declared sequence that fails in CI; second, process events serially inside the Orchestrator (`package:bloc` does this by default inside every `on<E>`), and it’s one of the reasons it exists; inside a hand-rolled ViewModel, a plain queue solves it). Sophisticated event concurrency is a subject the book returns to later; this chapter’s rule is simpler: never emit state from a cycle that’s already been overtaken.

Q&A

The Orchestrator builds `TabData`, formats currency, and picks the error message. Isn't that deciding? It's translating, and the distinction sits in the operand: formatting `1895` as "\$18.95" doesn't compare a business value against anything, it just dresses up a value someone already decided. It turns into a decision the moment a business `if` shows up in the middle: showing red when the total crosses X is a rule, and it goes down to the use case to hand back already decided. When building the state grows, extract pure presentation functions (a `formatCurrency`), called by the Orchestrator. The use case keeps working on the other side of that line, in whole cents, never reading or writing text.

One orchestrator per screen, or per feature? Per feature, the way chapter 11's tree suggests: one `tab_orchestrator` in the slice serves the waiter's screen and the cashier's, and that exact sharing is what kills pain (b) without any global state: both screens subscribe to the same source, and writing only happens through an event.

`PayTab` has existed since chapter 12 and the Orchestrator still doesn't handle it. Shouldn't the compiler complain? In Kotlin's `when` it does, and the `is PayTab -> Unit` arm sits there, explicit, waiting for chapter 14. `Unit` is Kotlin's "nothing": the counterpart to `void`, except it's an actual value, the object that means "there's nothing useful to return." A `-> Unit` arm is therefore a written-down "do nothing": the compiler demands the variant get handled, and you record that you handled it by choosing to ignore it, which is a very different thing from forgetting it. In Dart with `package:bloc` there's no

switch over events: an event with no registered handler blows up at runtime. It's a real trade-off between the two APIs, and exercise 1 puts it in your hands.

Do I need `package:bloc` ? Can't I write the Orchestrator by hand? You can, and this chapter's Java, PHP, and Python ports are exactly that: a class with observers and a single publish point. The library buys you serial event processing, `bloc_test`, and the conventions another developer in the ecosystem already recognizes.

Quick tip

Open the fattest orchestrator in your current project and run the search three times: `if (`, `>=`, and `[`. Every `if` whose operand is a business value, every quantity comparison, and every write into a collection that survives past the method is a candidate to change address: the first two go to a use case, the third to the repository. Same exercise as chapter 12's Quick tip, one layer down.

Quick reference

Situation	Fix
Receive event, choose which use case to call	orchestrator: the “when”
Fetch data from the repository before the rule	orchestrator: the “where to”
Compare total, quantity, or item in an <code>if</code>	leaked: the decision belongs in the use case

Compute a discount, fee, or price	leaked: the rule belongs in the use case
Stash a result in a “cache” map	leaked: persisting belongs to the repository
Translate <code>Result</code> into state in a switch	orchestrator: translates, doesn’t decide
Format an already-decided value for display	orchestrator (or a pure function)
Publish the state for the screen	orchestrator: owns the channel
Emit state from a cycle already overtaken	bug: reread the third Pitfall

Exercises

1. Add the event `ClearTab(int table)` to the `TabEvent` hierarchy from snippet 13-02 in your language and follow the compiler errors all the way through the cycle: in Kotlin and Swift, the orchestrator’s `when` and `switch` break right away and name the missing arm; in Dart, the compiler stays quiet (`bloc` registers handlers, it doesn’t switch) and it’s the flow test that catches it; in Go, nothing complains, exactly as the counterpoint section predicted. Finish by publishing the sequence `Loading` and then `Ready` with an empty tab.
2. The orchestrator below connects and it runs, and exactly two business rules snuck into it. Find both before reading the answer key in exercise 3; apply the Quick reference’s question to every line:



```
Future<void> _onAddItem(

  AddItem event,

  Emitter<TabState> emit,

) async {

  emit>Loading());

  final lookup = _repository.findTab(event.table);

  if (lookup is! TabFound) {

    emit(Failed("no connection"));

    return;

  }

  final tab = lookup.tab;
```

```
// "Just a safety cap, not really a rule": the line compares a
// business quantity and decides right here.

if (tab.items.length >= 20) {

    emit(Failed("table ${event.table} passed the item limit"));

    return;

}

final result = addItemToTab(tab, event.item);

switch (result) {

    case Success(:final value):

        // "Just a cache for the next fetch": the tab saved right
        // here, persistence wearing another name.

        _cacheByTable[event.table] = value;
```

```

        emit(Ready(toData(value)));

    case RuleViolated(:final reason):

        emit(Failed(reason));

    }

}

```

3. Answer key for exercise 2. First rule: the `if (tab.items.length >= 20)` right after the fetch; the operand is a business quantity (the table's item limit is Rosie's policy), so the line violates the table's "forbids deciding rules," and the cap moves address into `addItemToTab`, which already knows how to return `RuleViolated` with a ready message. Second rule: the `_cacheByTable[event.table] = value` inside the `Success` case; it's persistence disguised as a cache (second Pitfall), it violates "forbids persisting," and its destination is chapter 15's repository, the only place a tab gets saved. Open challenge: run the Quick tip on the biggest orchestrator in one of your own projects and count how many lines answer "what" instead of "when" or "where to."

Tip 13

The Orchestrator knows when and where to; never what. The moment it knows what, you just found a rule hiding in plain sight.

Next chapter: `addItemToTab` stops being a signature: chapter 14 opens the use case and writes, as a pure function, the discount rule that spent this whole chapter locked in the box.

Use Cases: Where the Rules Live

In this chapter, you'll:

- write Rosie's Coffee Shop's loyalty discount as a pure function, `applyLoyaltyDiscount`, that takes data and returns a `Result` with the typed refusal, without touching a database, a screen, or a framework;
- port that same rule to the book's ten languages and see what each one gains or loses expressing it;
- test that rule without a single test double, and answer why the single-implementation interface so many people ask for here is dead weight.

In chapter 13 the orchestrator fetched the tab, called a closed box named `addItemToTab`, and published whatever came back, without deciding a thing. This chapter opens that box. Inside it lives the rule the whole book promised would have a single address: the discount Rosie gives customers who rack up points. You're going to write it in a way that can't go wrong in two different places, because it only exists in one.

Picture the gesture that opens the box: table 4's tab is ready, the customer holds out her loyalty card, and someone asks "how much with the discount?". In chapter 13 that question reached the orchestrator, which passed it along without an opinion. Now it reaches its destination. That destination's contract has been signed since chapter 10, on the third line of the canonical responsibility table:

Use Case	Contract
Does	the only place for business rules
Does	is a pure function
Does	takes data, returns a Result
Forbids	IO
Forbids	framework
Forbids	domain exception

Each piece of that line deserves unpacking, because the chapter implements it to the letter. Start with the word that names the piece. A **use case** is a business rule isolated as a named operation: a domain verb that takes the data it needs, applies the house policy, and returns the verdict. “Apply the loyalty discount” is a use case; “add two integers” isn’t, because it carries no business policy at all.

“The only place for business rules” is the central promise, and it points back to chapter 5: knowledge that repeats itself diverges. If the discount lives in one place, changing it means editing a function; if it lives in three, changing it means a manhunt.

“Pure function” you already mastered in chapter 7: same input, same output, no side effect. The use case doesn’t read a clock, doesn’t roll dice, doesn’t write to disk; it calculates. “Takes data” closes the loop with chapter 9: everything the rule needs arrives ready, as an argument, materialized by whoever called it. And “returns a Result” is the idiom from chapter 8: the refusal doesn’t fly off as an exception, it comes back as a typed value the caller is forced to handle.

The prohibitions are the negative of the same photo. “IO, framework, and domain exception” are out: no `SELECT`, no widget, no `throw` to signal that the customer has no points. Keep that list in mind, because the chapter’s first pitfall is violating the first one of them with the best of intentions.

The anti-solution: the same rule in three places

Like every chapter in this part, the right path starts at the wrong one. Rosie’s loyalty rule is simple: a hundred points or more earn a 10% discount on the total. The problem isn’t the rule, it’s where it ended up. In a codebase that grew without a use case, it leaks into every spot that needs the total already discounted.

The first leak is in the View, to “show the right total on screen right away”. During a promotion, someone bumped the discount to 15% here and forgot the rest:

Dart

```
// Copy 1, in the View: to show the total right away, the rule got
// written here. During the promotion, it became 15%, and only here.

String totalOnScreen(int totalInCents, int points) {

    final percentage = points >= 100 ? 15 : 0;

    final discounted = totalInCents - totalInCents * percentage ~/ 100;

    final dollars = (discounted / 100).toStringAsFixed(2);
```

```
return "Total with discount: \$$dollars";  
  
}
```

The second leak is in the HTTP handler, which revalidates “for safety” with the rule handwritten again, still at the old 10%. The third is a database trigger, a third truth about the same discount, now in SQL, far from the eyes of whoever reads the Dart. All three calculate the same discount, and nothing guarantees they agree:

Dart

```
int totalInHandler(int totalInCents, int points) {  
  
    final percentage = points >= 100 ? 10 : 0;  
  
    return totalInCents - totalInCents * percentage ~/ 100;  
  
}
```

Run a \$40.00 tab for a customer with 120 points through all three paths. The screen promises \$34.00, the handler charges \$36.00, the database records \$36.00. The customer sees one number and pays another, and none of the three snippets has an isolated bug: each one is internally correct. The bug is the existence of three copies with the right to disagree. You already watched this movie in chapter 12 with three screens and three totals; here it’s the same disease one layer down, and the cure is the same too: one truth, and only one.

The rule as a pure function: the signature first

Before writing a single line of the body, write the signature, because the signature is what defines the piece. **Signature as contract** is the idea that a function's input and output types say everything it can and can't do, before any body exists at all:

Dart

```
DiscountResult applyLoyaltyDiscount(  
  
    Tab tab,  
  
    int loyaltyPoints,  
  
);
```

Read what that line already promises. It takes a `Tab` and an `int`, and nothing else: no repository, no HTTP client, no clock along the way, so there's no way to query the database in here, not even when temptation strikes. It returns `DiscountResult`, a type of its own, not an `int` or a `bool`: the refusal is going to have a name. The piece's entire architecture is already in those three elements; the body just fills in the promise.

One detail of the input deserves a name, because it's the chapter's thesis in miniature. Notice that the inventory information (whether "Cheese bread" is out of stock today) isn't a separate parameter or a query: it arrives inside the `Tab` itself, in an `outOfStock` field on each item. That's **materialized data**: everything the rule needs gets assembled BEFORE the call, by whoever called it, and handed over ready. Missing a piece of data? The input grows to carry it. The use case never goes looking for it.

With the signature standing, the body is almost a deduction.
First, the Result: three result variants, each refusal carrying its typed reason:

 Dart ·  Kotlin ·  Swift

```
sealed class DiscountResult {}

final class DiscountApplied extends DiscountResult {

    DiscountApplied(this.tab);

    final Tab tab;

}

final class NotEligibleForDiscount extends DiscountResult {

    NotEligibleForDiscount(this.points, this.pointsNeeded);

    final int points;

    final int pointsNeeded;

}
```

```
final class ItemOutOfStock extends DiscountResult {

    ItemOutOfStock(this.name);

    final String name;

}
```

`DiscountApplied` carries the tab with the total already reduced; `NotEligibleForDiscount` carries the points the customer has and the points they were missing, so the screen can explain the refusal without consulting any rule; `ItemOutOfStock` carries the name of the item that canceled the order. None of this is a loose string: every refusal is a type, and chapter 8 already showed why that matters.

Kotlin writes the same contract with `sealed class` and `data class`, one line per variant: `data class DiscountApplied(val tab: Tab) : DiscountResult()`. Swift uses an `enum` with associated values, the leanest spelling in the family: `case discountApplied(Tab)`. Three spellings for the same design, and the rule's body is trivially the same across all three. Here it is, and the order of the checks is the rule:

 Dart ·  Kotlin ·  Swift

```
DiscountResult applyLoyaltyDiscount(

    Tab tab,

    int loyaltyPoints,
```



```
) {  
  
    // First subgoal: an out-of-stock item cancels the whole order.  
  
    for (final item in tab.items) {  
  
        if (item.outOfStock) {  
  
            return ItemOutOfStock(item.name);  
  
        }  
  
    }  
  
    // Second: fewer than a hundred points, no discount.  
  
    if (loyaltyPoints < 100) {  
  
        return NotEligibleForDiscount(loyaltyPoints, 100);  
  
    }  
  
    // Third: ten percent off the total, truncated integer division.  
  
    final total = tab.totalInCents;  
  
    final discounted = total - total * 10 ~/ 100;
```

```
return DiscountApplied(Tab(tab.table, tab.items, discounted));  
  
}
```

Three subgoals, three `return` s, no side effect. The `~/` is Dart's integer division, the same choice snippet 10-01 made: working in cents and truncating, the total is an exact `int`, and the chapter's ten versions all land on the same number with no floating-point argument. A customer with 120 points on a \$40.00 tab gets back `DiscountApplied` with 3600 cents, \$36.00. The same customer, if the tab has an out-of-stock item, gets back `ItemOutOfStock("Cheese bread")` before any math runs.

Notice what's NOT here. No `try`, no `throw`, no `await`, no database `import`. The domain's most dramatic refusal (the item ran out) is a one-line `return`. That's the difference between a rule that lives in its own house and a rule squatting in the middle of a handler.

A quick aside about `addItemToTab`, which chapter 13 left as a closed signature: it lives in this same layer, and now its body can be opened too. The signature doesn't change a comma, `Result<Tab> addItemToTab( Tab tab, String item)`, and the body is another pure function: it looks the item up in the menu, returns `RuleViolated` if it can't find it, `Success` with the grown tab if it can. Same layer, same rules, each operation with its own `Result`.

One last note on semantics, so it doesn't get confused with chapter 10. The rule is exactly the one from snippet 10-01: threshold 100, 10%, an out-of-stock item cancels. What changes is the gesture. In 10-01, the discount showed up embedded in `addItemPure`, and a customer under 100 points still succeeded with a 0% reduction, because adding an item always works. Here the gesture is different: the customer ASKS for the discount when

closing the tab, and asking without having earned it is a real refusal, `NotEligibleForDiscount`, not a silent success. Same rule, different gesture, different Result.

The same rule, ten languages

Now the piece travels. The first three languages already came stacked together because the code was the same; the next seven each get their own listing. The rule is identical across all ten. What changes is how each language says “one of three results”, and that’s what’s worth comparing. In none of them is the rule rewritten: it’s the same function, ported.

Java 21 has the right tool: `sealed interface` plus `record` plus pattern-matching `switch`. The sealed interface lists who’s allowed to implement it, the record gives you a data carrier with no ceremony, and the expression `switch` demands exhaustiveness just like Dart:

Java

```
sealed interface DiscountResult {}

record DiscountApplied(Tab tab) implements DiscountResult {}

record NotEligibleForDiscount(int points, int pointsNeeded)

    implements DiscountResult {}
```

```
record ItemOutOfStock(String name) implements DiscountResult {}

static DiscountResult applyLoyaltyDiscount(

    Tab tab,

    int loyaltyPoints

) {

    for (Item item : tab.items()) {

        if (item.outOfStock()) return new ItemOutOfStock(item.name());

    }

    if (loyaltyPoints < 100) {

        return new NotEligibleForDiscount(loyaltyPoints, 100);

    }

    int total = tab.totalInCents();

    int discounted = total - total * 10 / 100;
```

```
    return new DiscountApplied(  
  
        new Tab(tab.table(), tab.items(), discounted));  
  
    }
```

TypeScript has no sealed class, and the most didactic entry point is the **discriminated union**: a type that's the sum of several objects, each with a literal field (`kind`) that says which one it is. The compiler narrows the type by that field's value, and an `assertNever` in the final branch guarantees that if you add a fourth case and forget to handle it, `tsc --strict` complains:

TypeScript

```
type DiscountResult =  
  
    | { readonly kind: "discountApplied"; readonly tab: Tab }  
  
    | { readonly kind: "notEligibleForDiscount"; readonly points: number;  
  
        readonly pointsNeeded: number }  
  
    | { readonly kind: "itemOutOfStock"; readonly name: string };  
  
function applyLoyaltyDiscount(  
  
    tab: Tab,
```

```
    loyaltyPoints: number,

): DiscountResult {

    for (const it of tab.items) {

        if (it.outOfStock) return { kind: "itemOutOfStock", name: it.name };

    }

    if (loyaltyPoints < 100) {

        return { kind: "notEligibleForDiscount", points: loyaltyPoints,

            pointsNeeded: 100 };

    }

    const total = tab.totalInCents;

    const discounted = total - Math.trunc((total * 10) / 100);

    return { kind: "discountApplied",

        tab: { ...tab, totalInCents: discounted } };

}
```

```
}
```

Rust is the conceptual north of the whole thing: the algebraic `enum` is made exactly for this, and the exhaustive `match` is the law of the language, no way around it. The `NotEligibleForDiscount` variant uses named fields, and `DiscountApplied` wraps the tab; no case can go unhandled and still compile:

Rust

```
enum DiscountResult {  
  
    DiscountApplied(Tab),  
  
    NotEligibleForDiscount { points: i64, points_needed: i64 },  
  
    ItemOutOfStock(String),  
  
}  
  
fn apply_loyalty_discount(  
  
    tab: Tab,  
  
    loyalty_points: i64,  
  
    ) -> DiscountResult {  
  
    for item in &tab.items {
```

```
    if item.out_of_stock {

        return DiscountResult::ItemOutOfStock(item.name.clone());

    }

}

if loyalty_points < 100 {

    return DiscountResult::NotEligibleForDiscount {

        points: loyalty_points,

        points_needed: 100,

    };

}

let total = tab.total_in_cents;

let discounted = total - total * 10 / 100;

DiscountResult::DiscountApplied(Tab {
```



```

        total_in_cents: discounted,

        ..tab

    })

}

```

C# has `record` and `switch`, but with a caveat chapter 8 already flagged: exhaustiveness over a record hierarchy is only a warning, not an error. That's why this port uses the `OneOf` library, which swaps the hierarchy for a closed sum type and forces a `Match` with one arm per case. That arm-per-case demand is exactly the exhaustiveness the language doesn't enforce on its own:



```

public static OneOf<DiscountApplied, NotEligibleForDiscount,

    ItemOutOfStock> ApplyLoyaltyDiscount(Tab tab, int loyaltyPoints)

{

    foreach (var item in tab.Items)

    {

        if (item.OutOfStock) return new ItemOutOfStock(item.Name);

    }

}

```

```

        if (loyaltyPoints < 100)

        {

            return new NotEligibleForDiscount(loyaltyPoints, 100);

        }

        int total = tab.TotalInCents;

        int discounted = total - total * 10 / 100;

        return new DiscountApplied(tab with { TotalInCents = discounted });

    }

```

PHP 8.2 has no real sealed union, and this port's idiom is an abstract class with `final` subclasses (sealed by convention) plus a `match` over `instanceof`. The important difference lives in the `match`: without a `default` arm, an unforeseen case becomes an `UnhandledMatchError` at runtime, not at compile time. The enforcement exists, but it arrives late. Here's the port:

PHP

```

function applyLoyaltyDiscount(

    Tab $tab,

```

```
int $loyaltyPoints,  
  
): DiscountResult {  
  
    foreach ($tab->items as $item) {  
  
        if ($item->outOfStock) {  
  
            return new ItemOutOfStock($item->name);  
  
        }  
  
    }  
  
    if ($loyaltyPoints < 100) {  
  
        return new NotEligibleForDiscount($loyaltyPoints, 100);  
  
    }  
  
    $total = $tab->totalInCents;  
  
    $discounted = $total - intdiv($total * 10, 100);  
  
    return new DiscountApplied(
```

```
new Tab($tab->table, $tab->items, $discounted));  
  
}
```

Go is the pedagogical counterpoint, and it's worth reading closely, because it shows what gets lost without unions. Go has no sealed class and no algebraic enum; its idiom for "one of N results" is the `(T, error)` pair. Both refusals become typed errors (`NotEligibleForDiscount` and `ItemOutOfStock` implement `error`), and success comes back as the tab plus a `nil`. It works, and the listing proves it; what's lost is compiler enforcement, because nothing forces the caller to tell the two errors apart. The idiomatic mitigation is `errors.As` in the caller plus a test per flow:



Go

```
type NotEligibleForDiscount struct {  
  
    Points      int  
  
    PointsNeeded int  
  
}  
  
func (e NotEligibleForDiscount) Error() string {  
  
    return fmt.Sprintf("not eligible: %d of %d points",  
  
        e.Points, e.PointsNeeded)
```

```
}
```

```
type ItemOutOfStock struct{ Name string }
```

```
func (e ItemOutOfStock) Error() string {
```

```
    return fmt.Sprintf("%s is out of stock", e.Name)
```

```
}
```

```
func applyLoyaltyDiscount(
```

```
    tab Tab,
```

```
    loyaltyPoints int,
```

```
) (Tab, error) {
```

```
    for _, item := range tab.Items {
```

```
        if item.OutOfStock {
```

```
            return Tab{}, ItemOutOfStock{item.Name}
```

```
        }
```

```

    }

    if loyaltyPoints < 100 {

        return Tab{}, NotEligibleForDiscount{loyaltyPoints, 100}

    }

    total := tab.TotalInCents

    discounted := total - total*10/100

    return Tab{tab.Table, tab.Items, discounted}, nil

}

```

Python closes the list with a `frozen dataclass` (the immutability from chapter 7), a `|` union, structural `match` from 3.10, and `assert_never` for the type checker to close the exhaustiveness. The rule is the same, with integer division as `//`:

Python

```

@dataclass(frozen=True)

class DiscountApplied:

```

```
tab: Tab
```

```
@dataclass(frozen=True)
```

```
class NotEligibleForDiscount:
```

```
    points: int
```

```
    points_needed: int
```

```
@dataclass(frozen=True)
```

```
class ItemOutOfStock:
```

```
    name: str
```

```
DiscountResult = DiscountApplied | NotEligibleForDiscount | ItemOutOfStock
```

```
def apply_loyalty_discount(
```

```

    tab: Tab,

    loyalty_points: int,

) -> DiscountResult:

    for item in tab.items:

        if item.out_of_stock:

            return ItemOutOfStock(item.name)

    if loyalty_points < 100:

        return NotEligibleForDiscount(loyalty_points, 100)

    total = tab.total_in_cents

    discounted = total - total * 10 // 100

    return DiscountApplied(replace(tab, total_in_cents=discounted))

```

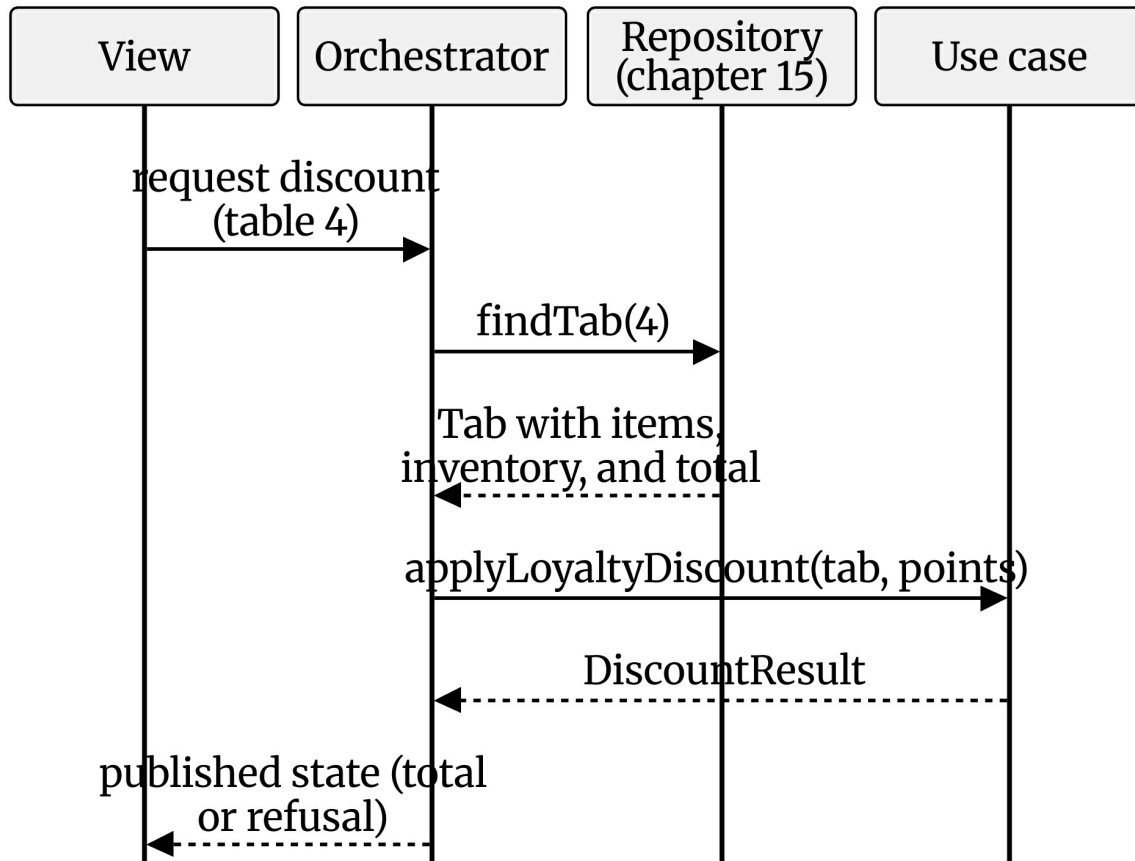
Ten languages, one rule, and the same inventory as always: what changes is how each one says “one of three results” and how early it enforces the cases you’re missing (Dart, Rust, Swift, and Kotlin at compile time; Java and TypeScript too, opt-in; C# with a

library; PHP and Python only at runtime or in the type checker; Go doesn't enforce it at all). What doesn't change is the rule, or the total: 3600 cents, across all ten. The ten complete versions, with a `main` running both canonical cases, are at the short routes in the Try it box at the end of the next section.

Orchestrator fetches, use case decides

Now it's possible to name the division of labor the whole chapter has been building. **Orchestrator fetches, use case decides** is the pair of responsibilities that makes the discount work without leaking: the orchestrator from chapter 13 knows WHEN to act and WHERE to find the data, materializes the tab with inventory and points, and hands it over ready; this chapter's use case takes that data and says WHAT the rule decided. One knows the address, the other knows the policy, and they never swap roles.

The full cycle, with the repository still the closed box chapter 15 is going to open:



Follow the arrows. The View asks; the orchestrator fetches from the repository, which returns the tab already materialized, with each item's `outOfStock` filled in from inventory; the orchestrator calls the use case with the tab and the points; the use case decides and returns the Result; the orchestrator translates it into state and publishes it. The use case shows up at the end of the chain: it takes data and hands back a verdict, with no idea the repository even exists. It's the table's row turning into a sequence.

Notice the repository as a closed box: the use case never talks to it. When "Cheese bread" is out of stock, the inventory is what knows that, and that information enters the tab BEFORE the use case gets called. The use case just finds an `item.outOfStock == true` already sitting in the data it received. That's why the signature has two parameters and not a database client: a piece of data is missing, the input grows, never a query.

Try it: open <https://focus.kodel.com.br/en/dart/14-01> (or <https://focus.kodel.com.br/en/kotlin/14-01>) and run it.

Prediction: the console prints “Table 4: \$36.00” and, on the next line, “No go: Cheese bread is out of stock”, with no screen and no database. Now swap the 120 points for 90 and predict the output before running it. The other eight languages are at /en/ts/14-01, /en/java/14-01, /en/csharp/14-01, /en/go/14-01, /en/php/14-01, /en/python/14-01, /en/swift/14-01, and /en/rust/14-01.

The use case as a retrieval unit

It pays to look at this division of labor through the question asked by whoever shows up later. “What’s the loyalty discount rule?” has, in this design, a one-file answer. That’s no accident: the signature declares everything the rule consumes, the body queries nothing on the outside, and the outcome is a value.

Whoever opens `applyLoyaltyDiscount` finishes the reading knowing the entire policy, including what it refuses and why.

Compare that with the version this chapter opened with, where the 10% lived in the View, in the HTTP handler, and in a SQL trigger. There the same question forces you to find three files in three languages, confirm those are all three, and decide which one wins when they disagree. The cost isn’t in reading the rule, which is short in both versions; it’s in building the certainty that no fourth copy is left over.

The property the pure version has is the one the architecture literature calls **retrieval-oriented architecture**: a design judged by what has to be retrieved to answer a question about the system. FOCUS’s retrieval unit is the use case, and it works because of chapter 7’s purity. A hidden dependency is precisely

what ruins retrieval: if the body went looking for inventory, the complete answer would start demanding the repository, its configuration, and the decision about which environment was running.

Testing without a single test double

Here purity delivers on its promise. A function that only takes data and returns a value tests in the simplest way there is: you arrange the data, call it, and compare the Result. There's no repository to simulate, no clock to freeze, no screen to assemble, so there's not a single test double (the mocks, stubs, and fakes from chapter 9). There's nothing to fake, because there's no dependency at all:

Dart

```
test("120 points earn 10%: 4000 becomes 3600", () {

    final tab = Tab(4, items, 4000);

    final result = applyLoyaltyDiscount(tab, 120);

    check(result).isA<DiscountApplied>()

        .has((r) => r.tab.totalInCents, "total")

        .equals(3600);

});
```

```

test("an out-of-stock item cancels the order", () {

    final withOutOfStock = [

        const Item("Espresso", 700),

        const Item("Cheese bread", 600, outOfStock: true),

    ];

    final tab = Tab(4, withOutOfStock, 1300);

    final result = applyLoyaltyDiscount(tab, 120);

    check(result).isA<ItemOutOfStock>()

        .has((r) => r.name, "name")

        .equals("Cheese bread");

});

```

Read the arrangement, the call, and the comparison. Each test builds whichever `Tab` it wants, calls `applyLoyaltyDiscount` with that case's points, and checks the Result's variant and what it carries, with `package:checks` from chapter 9 giving the expressive assertion.

The happy path (120 points, \$36.00) and the refusal (out-of-stock item) cost a handful of lines each, and the whole file imports nothing from UI or from a database.

A test that stops at the happy path is half a test, so the real file still covers the boundary every threshold demands: 100 points win, 99 don't. It's the kind of case snippet 10-01 had no way to isolate (the rule was embedded) and that here costs one line, because the rule has an entry point of its own. And there's the edge case that fools a lot of people: an empty tab, with a zero total, is `DiscountApplied` with a zero total, not a refusal, because 10% of zero is zero and success doesn't depend on there being anything to discount.

Keep one last proof for the next section: change the `< 100` to `< 90` in the use case, or the `10` to `15`, and run the test. It breaks right away and points at the wrong number. The rule has exactly one place where sabotage lands, and the test stands guard over it.

Try it: open <https://focus.kodel.com.br/en/dart/14-02> and run the test. Prediction: four cases pass, including the 100/99 threshold and the empty tab. Now do the sabotage from the paragraph above (`< 100` becomes `< 90`) and run it again: the case “100 points win; 99 don't” breaks, because with the threshold at 90 a 99-point customer starts earning the discount. Revert it and watch everything go green again. The other nine languages are at /en/kotlin/14-02, /en/ts/14-02, /en/java/14-02, /en/csharp/14-02, /en/go/14-02, /en/php/14-02, /en/python/14-02, /en/swift/14-02, and /en/rust/14-02.

The critique: “where's the use case's interface?”

Anyone coming from an enterprise codebase is going to miss one thing in this chapter, and the omission is deliberate. Where's the `IDiscountUseCase`, the interface the use case implements "so it can be mocked in the test"? The question is legitimate and has serious defenders, so it deserves an answer backed by a source, not by taste.

The direct critique comes from Dan North, the same creator of BDD (Behavior Driven Development), in the essay "CUPID, for joyful coding" (2022, dannorth.net): interfaces with a single implementation, created only to satisfy a mocking tool, are ceremony that gets in the way of reading, not abstraction that helps the project. And the positive formulation comes from Mark Seemann, whom nobody accuses of going easy on dependency injection: in the post "Dependency rejection" (2017, blog.ploeh.dk) and in the book *Dependency Injection Principles, Practices, and Patterns* (Manning, 2019, with Steven van Deursen), he argues that a pure function that takes data has no dependency to inject, and therefore nothing to abstract behind an interface.

That's the core of FOCUS's answer, and it's structural, not a matter of preference. A test interface exists so you can swap an implementation for a double. But `applyLoyaltyDiscount` receives no collaborator to swap: it takes a `Tab` and an `int`, inert data. There's no hidden database, no service to replace. The interface isn't skipped to save effort; it simply has nothing to abstract. Real dependency injection still exists in the project, but at the address from chapter 9: it lives in the repositories, where there's an actual IO boundary to swap between production and test. The use case stays pure, and pure doesn't need a double.

Here's my position, so you know where I'm speaking from. I've inherited more than one project with a single-method `IDiscountUseCase`, its single-method `DiscountUseCaseImpl`, a registration in the injection container, and a factory, all of it to wrap a

function that adds up a discount. Five files of scaffolding around ten lines of rule, and not one of them ever got a second implementation. It looked like abstraction and worked like dead weight: the cost of an indirection that never bought a single ounce of flexibility. I prefer the pure function, called by name, tested with data. If a real second implementation ever shows up, that's when the interface earns its keep, and extracting an interface out of a pure function is a two-minute refactor. Before that, it's just one more file for the next developer to open expecting logic and finding a `return impl.apply(x)`.

Pitfalls

The first pitfall is the most tempting one: the “just one quick `SELECT`” inside the use case. The rule needs to know whether the item is out of stock, and inventory is one query away, so why not query it right here? Because the instant the use case talks to the database, it stops being a pure function, goes back to depending on IO, and the test from the previous section needs a repository double to run. What goes wrong is exactly what the whole chapter avoided. How to get out of it: the missing data enters through the signature, materialized by whoever calls it. Missing the inventory? The `Tab` grows an `outOfStock` field, and the orchestrator fills it in before calling. The input grows; the use case never leaves home.

The second pitfall is returning the rule as an exception. It's tempting to write `throw NoPointsException()` when the customer isn't eligible, because “it's an exceptional case”. It isn't: a customer without points is routine, not exceptional, and the table's row forbids domain exceptions in the use case. What goes wrong: the refusal turns into invisible control flow the caller can forget to

catch, and chapter 8 showed the damage that does. How to get out of it: a refusal is a `return`, not a `throw`; a typed Result variant that the caller's `switch` is forced to handle.

The third pitfall is solving a missing piece of data with a query instead of letting the signature grow. A new requirement lands (the discount now depends on the time of day, or on the customer's spending that month), and the reflex is to inject a clock or a repository into the use case. What goes wrong: every injected dependency is one less unit of purity and one more double the test needs. How to get out of it: if the rule needs a new piece of data, it becomes a parameter, and whoever calls the function materializes it. A good use case's signature tells the whole story of what the rule consumes; a use case that hides queries lies about what it needs.

Q&A

What if the rule needs a piece of data that didn't come with the tab? The signature grows. Needed the month's total spend for a progressive discount? It becomes a parameter, `int monthlySpendInCents`, and the orchestrator fetches it from the repository and hands it over ready. The wrong reflex is injecting a repository into the use case so it can query on its own; that makes it impure and brings back the problem the chapter solved. The question that separates the two paths: "is this data an input to the rule, or is the rule going out looking for it?" Input grows the signature; going looking breaks purity.

Isn't it wasteful to materialize everything up front, even when the rule refuses right away over an out-of-stock item? In practice, the orchestrator was already going to fetch the tab anyway to show it on screen, so the data is already materialized by the time the use case gets called;

there's no extra fetch. If some day there's an expensive piece of data that only a fraction of calls actually use, the answer still isn't to query inside the use case: it's for the orchestrator to decide whether to materialize it. The rule stays pure.

One use case per operation, or one service with several methods? One per operation, one domain verb per function, like in this chapter. A “service” with ten methods turns into the junk drawer where ownerless rules pile up, and you're back to the cohesion problem from chapter 6. Pure functions named inside the feature folder (chapter 11) age better than a Swiss-army-knife class.

Quick tip

Open any use case or “service” in your project and read only the signature, no body. If it takes a repository, an HTTP client, a clock, or a logger, the body has hidden IO and the test is going to ask for a double. Write down those parameters: each one is a candidate to become an input materialized by whoever calls it. The impurity moves to the boundary, and the rule stays testable with plain values.

Quick reference

Situation	Fix
It's a business rule (discount, limit, policy)?	use case: pure function
The rule needs a piece of data that didn't come?	the signature grows

I want to refuse the customer's request	<code>return</code> a Result variant
I need to know if the item is out of stock	it comes ready in the input data
Fetch the tab before deciding	orchestrator (chapter 13), not the use case
Save the tab already discounted	repository (chapter 15), not the use case
Test the rule	arrange data, call it, compare the Result; no double
I created a single-method interface to mock	dead weight; the pure function is enough

Exercises

1. On birthdays, Rosie gives an extra 5% discount. Extend `applyLoyaltyDiscount` so the birthday customer earns 15% instead of 10%. Treat the birthday as DATA, not as a hidden rule: let the signature grow with what it needs to know. Start by breaking it on purpose and follow the errors from the exhaustive `switch` of whoever consumes the Result. Prove the original cases still hold (\$40.00 with 120 points still gives \$36.00 for a non-birthday customer) and that the birthday customer with 120 points and a \$40.00 tab gets back \$34.00.
2. Could you write `splitTabBetween(Tab tab, int people)` as a use case from the same family? Think about what it takes (just data?), what it returns (which typed refusals: an empty table? a split that doesn't land on round cents?), and how you'd test it

without booting up anything. You don't have to get the rounding rule right on the first try; you have to keep the piece pure and the refusal a value.

Tip 14

If a rule needs a mock to be tested, it's in the wrong layer. A pure rule tests with data; the mock is the smell of a dependency that should be at the boundary, not inside the rule.

Next chapter: the use case received the materialized tab and never asked where it came from. So where did it come from? Who filled in the `outOfStock` field, who turned the server's "no connection" into a value, who keeps the tab between one tap and the next? Chapter 15 opens the last closed box, the repository's, and answers where the data comes from.

Repositories: The Exception Boundary

In this chapter, you'll:

- write the tab repository that fetches and writes data, translating a network drop into `Failure.noConnection` in one single place;
- return one typed Result from the read and another from the write, with no infrastructure `try/catch` leaking into the orchestrator or the screen;
- spot, in someone else's repository, the business rule that hid inside a “clever” query where only a connection to the world should be.

Chapter 14's use case receives a ready tab and decides the rule on it. But someone has to go fetch that tab from the real world, where Rosie's Coffee Shop's Wi-Fi drops in the middle of the lunch rush and the card processor takes five seconds to answer. This chapter builds the piece that faces that dirty world and hands the rest of the app nothing but clean values: the repository, the last box chapters 13 and 14 left sealed.

Chapter 13 closed the orchestrator's cycle with two boxes still sealed: the use case, which chapter 14 opened, and the repository, which opens now. The contract's line has been signed since chapter 10, in the canonical table's fourth row:

--	--	--

Layer	Does	Forbids
Repository	CRUD (fetch and save)	business rules
	the only place an infra exception becomes a Result	

It's worth reading slowly, because every piece of that row got written after a bug. **CRUD** (Create, Read, Update, Delete) is the four data operations any app runs against storage. The one asking is chapter 13's orchestrator: it wants table 4's tab, and the repository hands back table 4's tab, whether it comes from an API, a local database, or a file.

"The only place an infra exception becomes a Result" is the hard thesis, and it comes from chapter 8. The **infrastructure boundary** is the layer that talks to the world outside your process: network, database, disk, card processor. It's the only territory where someone else's exceptions are unavoidable, because the libraries living there are the ones throwing them. FOCUS's rule fits in one line: an infrastructure exception gets caught once, right there, and translated into a typed `Failure`; from there inward, only Results circulate through the app. Chapter 8 showed the gesture in five lines and promised that "chapter 15 builds this boundary in detail." That promise comes due now.

The ban closes the back door. "Business rules" are forbidden in the repository because a rule has its own home, chapter 14's use case, and a scattered rule turns into a copy, the way you already saw in chapter 12 with the three screens and in chapter 13 with the discount stuck in the middle of the switch. The repository knows *where* the data lives and *how* to talk to it. It never knows *whether* the customer earned the discount.

The piece's name is old. **Repository** is the pattern Martin Fowler cataloged in *Patterns of Enterprise Application Architecture* (2002) as the layer that “mediates between the domain and data mapping layers, acting like an in-memory collection of domain objects.” Eric Evans, in *Domain-Driven Design* (2003), backs the same design: one repository per domain aggregate, with the interface speaking the business's vocabulary, not the table's. Notice that both define a repository by domain *feature*, never as a generic per-table CRUD. That distinction comes back in the critique section, and it's the difference between a pattern and a decoration.

One common misunderstanding needs clearing up before a single line gets written: the screen doesn't subscribe to the repository. The one publishing state is chapter 13's orchestrator, the owner of the channel that carries state down to the View. When the data changes in another waiter's hands and the screen needs to know, it's the orchestrator that listens to the source and translates the change into state. The repository hands over the data and gets out of the way.

The anti-solution: the copied catch in every screen

Like every chapter in this part, the right path starts with the wrong one. Rosie hired a developer who solved the network outage the most direct way possible: wherever the code needed the data, it fetched straight from the server and wrapped the call in a `try/catch`. It worked on the first screen.

Dart

```
// Menu screen: fetches straight from the server and handles the
```

```
// failure by hand.

String menuScreen(int table) {

    try {

        final tab = queryServer(table);

        return "table ${tab.table}'s tab";

    } on SocketException {

        return "no connection";

    }

}
```

It worked on the second screen too, and the third, and the fourth. The problem isn't any one of them alone; it's that there are now four. The menu screen, the tab screen, the history screen, and the register report each carry a copy of the same `try/catch`, and each developer who wrote theirs invented a different message: "no connection," "network failed, try again," "couldn't load," "communication error." Three pains come out of that, and they're worth naming.

The first is **duplication**: the same `SocketException` translation in four places. The day the network failure needs to become a metric, or a single message, or a retry, you edit four files and

hope you didn't miss one. The second is **invisible forgetting**, and it's worse, because it doesn't show up in a single-file code review. Look at the fifth screen, the payment one, written last week by someone in a hurry:

Dart

```
// Payment screen, just created. It forgot the try/catch. With the  
  
// network down, the SocketException rises bare and the app crashes on  
  
// the most expensive gesture: getting paid.  
  
void paymentScreen(int table) {  
  
    final tab = queryServer(table);  
  
    saveToServer(tab);  
  
}
```

There's no `catch` here. While the network stays up, nobody notices. The day the Wi-Fi drops in the middle of the lunch rush, this screen crashes in the customer's hand, card reader out, right at the moment of payment. The crash isn't a logic bug; it's the absence of a line the other four screens had and this one doesn't. Nothing in the compiler charged for the gap, because `SocketException` is an ordinary exception, and an ordinary exception doesn't show up in anyone's signature.

The third pain is the **blind caller**. Even in the four screens that remembered the `catch`, whoever calls `menuScreen` receives a `String`, and that string can just as easily be a tab as it can be “no connection.” The type doesn’t tell success from failure apart. Whoever consumes it has to compare text, or worse, trust that the string looks like a tab. The failure turned into a second-class value, disguised as a success.

All three pains share one root: the infrastructure exception is being handled everywhere, when it should be translated in one place. Take `SocketException` back to its own territory, and all three disappear together.

The repository’s contract comes before its body

Write the interface first. The contract is what chapter 13’s orchestrator sees; the body, nobody outside gets to see. At Rosie’s Coffee Shop, the tab repository does three things: finds a tab, saves a tab, and marks a tab as paid. The lookup already existed as a signature in chapter 13; the two writes come in now, with the payment feature.

Dart

```
// features/tab/tab_repository.dart

// The contract: fetch and write on demand, request/response. No

// generic method, no watch/Stream (reactivity lives in the

// orchestrator, chapter 13).
```

```
abstract interface class TabRepository {

    LookupResult findTab(int table);

    SaveResult save(Tab tab);

    SaveResult markAsPaid(Tab tab);

}
```

Read it method by method. `findTab(int table)` arrives exactly as chapter 13 declared it, with the `LookupResult` return that chapter anticipated; now you see where that type comes from, because the lookup can fail over the network and needs to say so in its type. `save` and `markAsPaid` are new, and they return `SaveResult`, a Result of the write's own. `markAsPaid` is the gesture where the Wi-Fi drops in the anti-solution; it's the one driving the rest of the chapter.

Kotlin writes the same contract with `interface` and plain functions, one line per method, the same names: `fun findTab(table: Int): LookupResult`. Same shape, half the ceremony. Notice what the interface *doesn't* have: no generic `save<T>(T entity)`, no `getAll()`. Just the three verbs the tab feature asks for. That restriction is a design decision, and the critique section explains why it's the pattern's whole point.

Now the return types. The read and the write have distinct Results, and that's on purpose.

Dart

```
// features/tab/failure.dart
```

```
// The sealed family of INFRASTRUCTURE failures, with the same

// identifiers chapter 8 printed. It only grows with infra variants.

enum Failure { noConnection }


// features/tab/lookup_result.dart

// Result of the READ (chapter 8) and Result of the WRITE (new). Two

// distinct sealed types plant chapter 16's command/query distinction.

sealed class LookupResult {}


sealed class SaveResult {}


final class TabFound extends LookupResult {

  TabFound(this.tab);


  final Tab tab;

}
```

```

final class TabSaved extends SaveResult {

    TabSaved(this.tab);

    final Tab tab;

}

// A class can't extend two sealed types; it implements both instead.

// The infra failure is the same one, whether you're reading or

// writing.

final class InfraFailure implements LookupResult, SaveResult {

    InfraFailure(this.failure);

    final Failure failure;

}

```

`TabFound` and `InfraFailure` are spelled exactly as chapter 8 spelled them, character for character; chapter 8 showed the usage, this chapter gives the full definition. `Failure` is the sealed family of infrastructure failures, with the same identifier chapter 8 already

printed (`noConnection`). Only an infra variant belongs there: the repository doesn't produce a domain failure, because a domain failure is a rule's verdict, and rules don't live here.

Two things deserve attention. The first: `LookupResult` and `SaveResult` are two distinct sealed types, with `TabFound` on one side and `TabSaved` on the other. Asking and changing are gestures of a different nature, and naming them with different types plants a distinction chapter 16 will harvest. The second: `InfraFailure` belongs to both. The network failure is the same one whether you're reading or writing, so it `implements` both sealed types instead of duplicating the variant. In Dart, a class extends at most one sealed type and implements as many as it needs; that language restriction is the technical reason `InfraFailure` uses `implements` while `TabFound` uses `extends`. Kotlin solves the same restriction with `sealed interface`, which a class can implement more than once.

The single translation: one catch, and only one

With the contract signed, the body fits in a small class, and it's the opposite of the anti-solution. Where there were five scattered `try/catch` blocks (four copied and one forgotten), now there's one.



```
// features/tab/tab_repository_impl.dart

// The single translation: the SocketException try/catch lives HERE,

// in one place, on both the read AND the write. From there inward,

// only Results circulate.
```

```
class TabRepositoryImpl implements TabRepository {
```

```
    TabRepositoryImpl(this._driver);
```

```
    final ServerDriver _driver;
```

```
    @override
```

```
    LookupResult findTab(int table) {
```

```
        try {
```

```
            return TabFound(_driver.query(table));
```

```
        } on SocketException {
```

```
            return InfraFailure(Failure.noConnection);
```

```
        }
```

```
    }
```

```
    @override
```

```
    SaveResult save(Tab tab) {
```

```

try {

    _driver.write(tab);

    return TabSaved(tab);

} on SocketException {

    return InfraFailure(Failure.noConnection);

}

}

@Override

SaveResult markAsPaid(Tab tab) => save(tab);

}

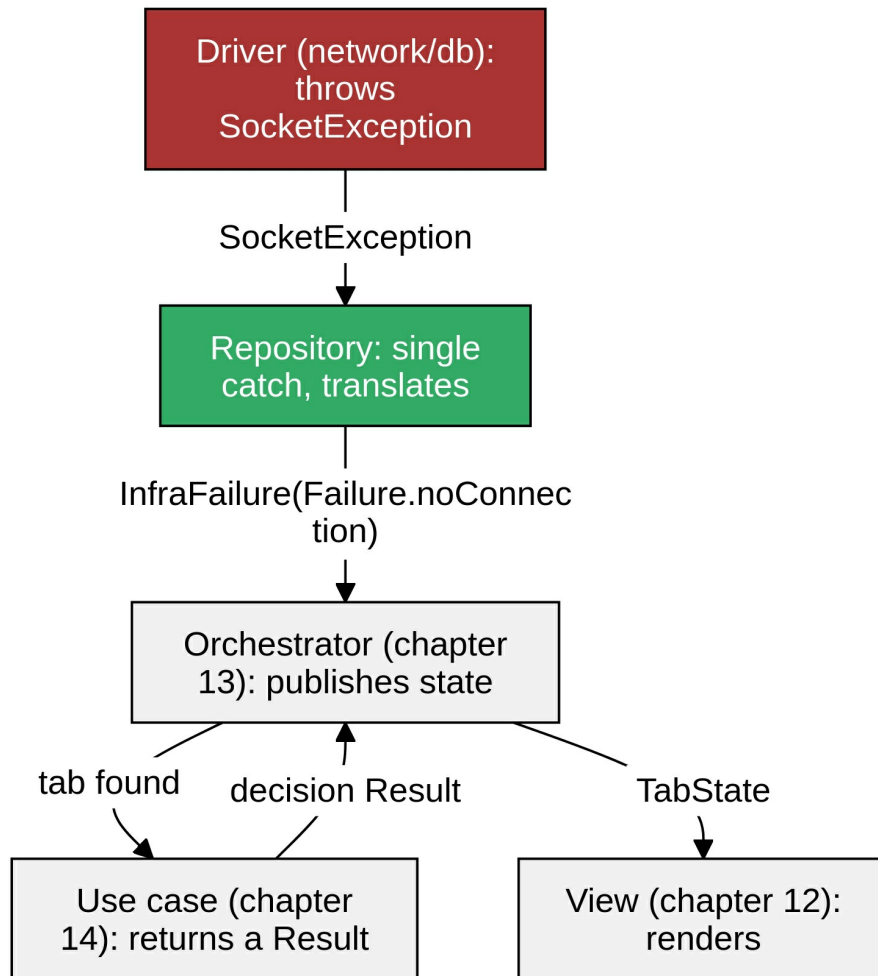
```

This is where the definition that titles the chapter is born. **Single exception translation** is catching the infrastructure exception at the boundary, exactly once, with immediate conversion into a typed `Failure`; no other layer ever touches that exception again. `_driver` is an opaque box that talks to the dirty world (the database, the HTTP server) and can throw `SocketException`; no line here writes SQL, because the driver is a detail behind the contract. What the repository does with the exception is exactly

one thing: turn it into `InfraFailure(Failure.noConnection)` and hand that back as a value. From the `return` on, `SocketException` has stopped existing for the rest of the app.

Notice the translation happens on both the read and the write, with the same `on SocketException`. It's the two-handed funnel: every piece of data going in and every piece coming out passes through this one point, and it's only here that a network exception becomes a value. The anti-solution's four screens now call `findTab` and get back a `LookupResult`; the fifth, the payment one, calls `markAsPaid` and gets back a `SaveResult`, and the compiler no longer lets it ignore the failure, because the failure is now a case of the type it has to handle.

The exception's path fits in one diagram, and it's the map for the whole chapter:



The exception is born red in the driver, dies green in the repository, and what climbs through every layer after that is a typed value, never an exception. Compare that with the anti-solution: there, `SocketException` could climb out of any of the five screens, and one of them let it climb all the way to a crash. Here there's one point, and only one, where an infra failure has permission to exist.

A `main` proves the two-handed funnel, with the scenario that runs the same way across the book's ten languages:

Dart

```
// Canonical scenario: two verbs. A live lookup returns the tab; a
```

```
// write with the network down returns a typed noConnection, no crash
```

```
// and no try/catch in the consumer.
```

```
void main() {
```

```
    final online = TabRepositoryImpl(ServerDriver());
```

```
    final lookup = online.findTab(4);
```

```
    switch (lookup) {
```

```
        case TabFound(:final tab):
```

```
            print(
```

```
                "findTab(4)      success      → "
```

```
                "TabFound (table ${tab.table}, "
```

```
                "${tab.totalInCents} cents)",
```

```
            );
```

```
        case InfraFailure(:final failure):
```

```
            print("findTab(4)      → InfraFailure($failure)");
```

```

    }

    final offline = TabRepositoryImpl(ServerDriver(networkDown: true));

    final saved = offline.markAsPaid(Tab(4, const [], 1895, true));

    switch (saved) {

        case TabSaved(:final tab):

            print("markAsPaid(t) → TabSaved(${tab.table})");

        case InfraFailure(:final failure):

            print(

                "markAsPaid(t) network down → "

                "InfraFailure(Failure.${failure.name})",

            );

    }

}

```

Run it and you get the two lines every version of this chapter prints:

```
findTab(4)      success      → TabFound (table 4, 1895 cents)
markAsPaid(t)   network down → InfraFailure(Failure.noConnection)
```

The consumer handles both Results with an exhaustive `switch`, the same one from chapter 8, and nowhere in it does a `try/catch` exist. The network drop arrived as a value, typed, and the compiler guaranteed the failure case got handled.

Try it: open <https://focus.kodel.com.br/en/dart/15-01> and delete the `case InfraFailure(...)` arm from one of `main`'s switches. Prediction: the code doesn't even run; the compiler flags the switch as non-exhaustive and lists the missing variant. That's the warning the anti-solution never had: there, the payment screen forgot the `catch` and nobody warned it. The other nine languages are at `/en/kotlin/15-01`, `/en/ts/15-01`, `/en/java/15-01`, `/en/csharp/15-01`, `/en/go/15-01`, `/en/php/15-01`, `/en/python/15-01`, `/en/swift/15-01`, and `/en/rust/15-01`.

The lookup feeding the orchestrator

The repository's contract closes the loop with chapter 13. The orchestrator used to fetch the tab and hand it to the use case; now the lookup can return `InfraFailure`, and it's the orchestrator that translates that into state, because it, not the repository, is the one publishing state to the screen.



```
// In the orchestrator (chapter 13): the lookup can now fail over the  
  
// network.  
  
final lookup = _repository.findTab(event.table);  
  
switch (lookup) {  
  
  case TabFound(:final tab):  
  
    emit(Ready(toData(tab)));  
  
  case InfraFailure():  
  
    emit(Failed("no connection")); // the failure state, not the exception  
  
}
```

The failure arm doesn't even open the variant: the orchestrator just needs to know it failed, and the unbound `case InfraFailure()` says exactly that. The publishing mechanism is chapter 13's (the bloc's stream, `StateFlow`, the Redux store), and nothing about it changes here; what changes is that the failure state can now be born from a translated network drop, on top of the use case's rule violation. The repository doesn't know a screen is waiting; it returned a value, and the orchestrator decided which state that value becomes.

The same boundary in ten languages

The single translation isn't a Dart trick. The nine listings below are the same `TabRepositoryImpl` you just read, each one in its own language's idiom. In every one, look for the same three things: the point where the driver's exception gets caught, the line that converts it into `InfraFailure`, and the absence of any business rule in between. What changes from one to the next is only how the language writes "one of two outcomes."

TypeScript has no `sealed class` and solves the Result with a **discriminated union**, a `type` whose `kind` field tells the cases apart. The parameterless `catch` catches any error the driver throws, and the return comes back already labeled:

TS TypeScript

```
type LookupResult =  
  
  | { readonly kind: "tabFound"; readonly tab: Tab }  
  
  | { readonly kind: "infraFailure"; readonly failure: Failure };  
  
type SaveResult =  
  
  | { readonly kind: "tabSaved"; readonly tab: Tab }  
  
  | { readonly kind: "infraFailure"; readonly failure: Failure };
```

```
class TabRepositoryImpl implements TabRepository {

    constructor(private readonly driver: ServerDriver) {}

    findTab(table: number): LookupResult {

        try {

            return { kind: "tabFound", tab: this.driver.query(table) };

        } catch {

            return { kind: "infraFailure", failure: "noConnection" };

        }

    }

    save(tab: Tab): SaveResult {

        try {

            this.driver.write(tab);

            return { kind: "tabSaved", tab };

        }

    }

}
```



```

    } catch {

        return { kind: "infraFailure", failure: "noConnection" };

    }

}

markAsPaid(tab: Tab): SaveResult {

    return this.save(tab);

}

}

```

Exhaustiveness doesn't come free: the consumer switches on `result.kind` with a `default` that calls `assertNever`. Add a third case to the union and forget to handle it, and `never` complains at compile time. It's chapter 8's protection, opt-in.

Kotlin is Dart's next-door neighbor, with one advantage: `sealed interface` accepts multiple implementation, so `InfraFailure` belongs to both Results without the `extends / implements` pair Dart needed. And since `try` is an expression in Kotlin, each method fits in a single `=`, with no `return`:

Kotlin

```
class TabRepositoryImpl(
```

```
private val driver: ServerDriver,

) : TabRepository {

    override fun findTab(table: Int): LookupResult =

        try {

            TabFound(driver.query(table))

        } catch (e: IOException) {

            InfraFailure(Failure.noConnection)

        }

    override fun save(tab: Tab): SaveResult =

        try {

            driver.write(tab)

            TabSaved(tab)

        } catch (e: IOException) {

            InfraFailure(Failure.noConnection)

        }
}
```

```
override fun markAsPaid(tab: Tab): SaveResult =  
  
    save(tab)  
  
}
```

Swift models both Results as an `enum` with associated values, and the translation turns into `do/catch`. Notice the loose dot in `.infraFailure` and `.noConnection`: Swift infers the type from the declared return, so the enum's name disappears from the line. Whoever consumes it gets an exhaustive `switch` by the language's own requirement.

Swift

```
struct TabRepositoryImpl: TabRepository {  
  
    let driver: ServerDriver  
  
    func findTab(_ table: Int) -> LookupResult {  
  
        do {  
  
            return .tabFound(try driver.query(table))  
  
        } catch {  
  
            return .infraFailure(.noConnection)  
  
        }  
    }  
}
```

```
}  
  
}  
  
func save(_ tab: Tab) -> SaveResult {  
  
    do {  
  
        try driver.write(tab)  
  
        return .tabSaved(tab)  
  
    } catch {  
  
        return .infraFailure(.noConnection)  
  
    }  
  
}  
  
func markAsPaid(_ tab: Tab) -> SaveResult {  
  
    return save(tab)  
  
}
```

```
}
```

C# carries chapter 8's caveat: a `record` hierarchy doesn't guarantee exhaustiveness, and the compiler only emits a warning. There's a second consequence here, and it's about modeling: a `record` in C# is a class, and a class inherits from only one, so `LookupResult` and `SaveResult` need to be interfaces for `InfraFailure` to belong to both.



```
class TabRepositoryImpl : ITabRepository
{
    private readonly ServerDriver _driver;

    public TabRepositoryImpl(ServerDriver driver) =>
    {
        _driver = driver;
    }

    public LookupResult FindTab(int table)
    {
        try
        {

```

```
        return new TabFound(_driver.Query(table));

    }

    catch (System.IO.IOException)

    {

        return new InfraFailure(Failure.NoConnection);

    }

}

public SaveResult Save(Tab tab)

{

    try

    {

        _driver.Write(tab);

        return new TabSaved(tab);

    }

}
```

```

        catch (System.IO.IOException)

        {

            return new InfraFailure(Failure.NoConnection);

        }

    }

    public SaveResult MarkAsPaid(Tab tab) =>

        Save(tab);

    }

```

Java 21 writes the same design with `sealed interface` and `record`, and the consumer's pattern `switch` is exhaustive just like Dart's. The difference that jumps out is the `throws IOException` on the driver's signature: in Java, an IO exception is *checked*, and the compiler demands that someone handle it. That someone is the repository. The language pushes the `catch` exactly where FOCUS wants it.

Java

```

static final class TabRepositoryImpl implements TabRepository {

    private final ServerDriver driver;

```

```
TabRepositoryImpl(ServerDriver driver) {

    this.driver = driver;

}

@Override

public LookupResult findTab(int table) {

    try {

        return new TabFound(driver.query(table));

    } catch (IOException e) {

        return new InfraFailure(Failure.noConnection);

    }

}

@Override

public SaveResult save(Tab tab) {

    try {
```



```

        driver.write(tab);

        return new TabSaved(tab);

    } catch (IOException e) {

        return new InfraFailure(Failure.noConnection);

    }

}

@Override

public SaveResult markAsPaid(Tab tab) {

    return save(tab);

}

}

```

PHP has no real sealed types, and the snippet solves it with marker interfaces: `LookupResult` and `SaveResult` are empty interfaces, and the variants are `final readonly` classes that implement them. Sealing turns into convention, enforced in review, and the

consumer's `match` fails at runtime if a case is missing. Notice the `catch (RuntimeException)` with no variable: PHP 8 lets you omit `$e` when nobody's going to use it.

PHP

```
final class TabRepositoryImpl implements TabRepository
{

    public function __construct(private readonly ServerDriver $driver)
    {

    }

    public function findTab(int $table): LookupResult
    {

        try {

            return new TabFound($this->driver->query($table));

        } catch (RuntimeException) {

            return new InfraFailure(Failure::NoConnection);

        }

    }

}
```

```
}
```

```
public function save(Tab $tab): SaveResult
```

```
{
```

```
    try {
```

```
        $this->driver->write($tab);
```

```
        return new TabSaved($tab);
```

```
    } catch (RuntimeException) {
```

```
        return new InfraFailure(Failure::NoConnection);
```

```
    }
```

```
}
```

```
public function markAsPaid(Tab $tab): SaveResult
```

```
{
```

```
    return $this->save($tab);
```

```
}  
  
}
```

Python builds the Results with a `frozen dataclass` and the `|` union operator, and catches `ConnectionError`, the standard library's network exception. Exhaustiveness is left to the type checker: in the consumer, the `match` ends in a `case _: assert_never(...)`, which mypy enforces during static analysis and the interpreter ignores.

Python

```
class TabRepositoryImpl:  
  
    def __init__(self, driver: ServerDriver) -> None:  
  
        self._driver = driver  
  
    def find_tab(self, table: int) -> LookupResult:  
  
        try:  
  
            return TabFound(self._driver.query(table))  
  
        except ConnectionError:  
  
            return InfraFailure(Failure.NO_CONNECTION)
```

```

def save(self, tab: Tab) -> SaveResult:

    try:

        self._driver.write(tab)

    return TabSaved(tab)

    except ConnectionError:

        return InfraFailure(Failure.NO_CONNECTION)


def mark_as_paid(self, tab: Tab) -> SaveResult:

    return self.save(tab)

```

Go pulls in the opposite direction, and it's the chapter's counterpoint. The language has no union and no sealed type; its idiom for “success or failure” is the native `(T, error)` pair, which chapter 8 already introduced. Here's the point: translating an infrastructure error into a domain error inside the repository isn't an imported pattern in Go, it's the idiomatic way to do it. The translation even earns a method with its own name, `translate`, and `errors.As` is what feeds it:



Go

```

func (r TabRepositoryImpl) translate(err error) error {

```

```
var opErr *net.OpError
```

```
if errors.As(err, &opErr) {
```

```
    return ErrNoConnection
```

```
}
```

```
return err
```

```
}
```

```
func (r TabRepositoryImpl) FindTab(table int) (Tab, error) {
```

```
    tab, err := r.Driver.Query(table)
```

```
    if err != nil {
```

```
        return Tab{}, r.translate(err)
```

```
    }
```

```
    return tab, nil
```

```

}

func (r TabRepositoryImpl) Save(tab Tab) error {

    return r.translate(r.Driver.Write(tab))

}

func (r TabRepositoryImpl) MarkAsPaid(tab Tab) error {

    return r.Save(tab)

}

```

Without `TabFound` and `InfraFailure` types, Go carries the same information in the `(Tab, error)` pair, and the caller compares `err` against `ErrNoConnection` through `errors.Is`. What Go doesn't do is force you to handle the error: a `_` swallows `err` and the compiler stays quiet. Exhaustiveness left the compiler and turned into review discipline, exactly as in chapter 13. The single translation, though, still holds: the boundary's `ErrNoConnection` is the other languages' `InfraFailure` wearing different clothes.

Rust closes the loop at Go's opposite extreme: here there's no exception at all to catch. The driver already returns `Result<Tab, NetworkError>`, so the single translation turns into a `match` that swaps the infra error for the domain `Failure`. It's the same gesture as Dart's, with the compiler billing you for every arm.

Rust

```
impl TabRepository for TabRepositoryImpl {

    fn find_tab(&self, table: i64) -> LookupResult {

        match self.driver.query(table) {

            Ok(tab) => LookupResult::TabFound(tab),

            Err(NetworkError) => {

                LookupResult::InfraFailure(Failure::NoConnection)

            }

        }

    }

}

fn save(&self, tab: Tab) -> SaveResult {

    match self.driver.write(&tab) {

        Ok(()) => SaveResult::TabSaved(tab),

        Err(NetworkError) => {
```



```

        SaveResult::InfraFailure(Failure::NoConnection)

    }

}

}

fn mark_as_paid(&self, tab: Tab) -> SaveResult {

    self.save(tab)

}

}

```

Ten languages, ten syntaxes, one structure. Sealed types in three of them, a marker interface in two, a type union in two, an algebraic enum in two, and the `(T, error)` pair in one; `try/catch` in six, `do/catch` in one, `try/except` in one, `match` over a `Result` in one, and `errors.As` in one. And in every one of them, without exception, the same single point: the driver's exception goes in, the typed `Failure` comes out, and nothing infrastructural crosses the door. Each version runs on its own short route and prints the same two lines.

Where reading can stop

Go back to the three-method contract and ask a reader's question rather than an author's: to change the payment rule, how much of the repository do you have to read? The answer is the interface, and nothing below it. The three signatures say what goes in, what comes out, and how failure shows up; the driver, the SQL, the retry, and the table's schema stay on the other side. It isn't that they're irrelevant: it's that they're irrelevant to that change.

A point in the system where reading can stop without owing anything has a name in the architecture literature for language models: **context boundaries**, the limits that say how far context has to be loaded to understand a decision. Chapter 11 dealt with the slice that fits whole in the window; here comes what the slice can legitimately leave out. The two hold each other up: the slice only fits because a boundary says the driver doesn't have to come along.

The proof is the reverse exercise. In the anti-solution that opened the chapter, the `SocketException` catch sat inside every screen, and every screen was itself a piece of driver. Reading couldn't stop anywhere there: to know why the menu screen returned "no connection" and the history screen returned "couldn't load," you had to read both screens and the network library both of them used. The boundary isn't a folder or a package; it's the point where the infrastructure type stops existing.

Contracts that age slowly

The boundary in the section above holds for a snapshot of the code. The other half is missing, which is what happens to it over time.

Swap `TabRepositoryImpl`'s `ServerDriver` for a client of a different database. Then add an in-memory cache for the lookup. Then a retry with backoff for the write. All three touch the body, and none touches the three signatures: `findTab` still takes a table and returns a `LookupResult`, `markAsPaid` still returns a `SaveResult`. Whoever read the tab feature before the three changes reads the same thing after them.

That property is what the same literature calls **stable contracts**: interfaces that survive swapping out what sits behind them. It isn't automatic, and the ruler for checking is a harsh one. If swapping the implementation forces a change to the signature, the contract was leaking an implementation detail and you've just found out where. A `findBySql(String sql)` fails on sight. Returning the driver's exception inside the Result fails too, and that's why `Failure` has `noConnection`, which is a fact about the tab that never arrived, and not `SocketException`, which is a fact about the library of the month.

The promise is worth measuring. A stable contract doesn't mean a frozen contract: when the coffee shop starts accepting partial payment, a new method comes in, and it should. What the stable contract promises is something else: that the signature changes when the rule changes, not when the database changes.

The critique: the generic repository and “the ORM already does this”

A pattern with a design-pattern name draws two strong critiques, and both deserve a sourced answer.

The first attacks the generic form. There's a temptation, popular in enterprise code, to write an `IRepository<T>` with `Add(T)`, `GetById(id)`, `GetAll()`, and `Delete(T)`, and derive every repository

from it. Ben Morris takes that apart in the article “Why the generic repository is just a lazy anti-pattern” (ben-morris.com), and the argument is direct: the generic repository isolates nothing. To serve any real query it ends up exposing an `IQueryable`, and the underlying technology (Entity Framework, SQL) leaks through the interface that was supposed to hide it. You gained a layer of indirection and didn’t gain the isolation it promised. It’s the same reading Fowler and Evans gave, defining the repository by domain aggregate, with methods the feature asks for, not as a per-table CRUD.

Here’s my position, no middle ground: I wouldn’t write a generic `save<T>(T entity)` under torture, and the tab’s case says why. A generic `save` doesn’t know how to translate `SocketException` into the tab’s vocabulary; it would hand back the raw exception or a `bool`, and the typed `InfraFailure` would die at the door. A generic `getAll()` invites fetching the whole tab table when the screen only needs table 4’s total. FOCUS’s repository contract is the opposite of generic: the three methods the tab feature asks for, named in the business’s own words, and nothing more. That restriction is the pattern’s main feature.

The second critique comes from ORM users: “Entity Framework, Prisma, Eloquent are already repositories; FOCUS’s layer is redundant.” The **ORM** (Object-Relational Mapping) is the library that translates table rows into objects and back. It solves the mapping, and solves it well. What it doesn’t do are the two roles the canonical table’s row demands. The first is the per-feature contract: EF’s `DbContext` exposes the whole database, every table, every possible query; the tab repository exposes three verbs. The second is the exception funnel: the ORM throws its own connection exception (`DbUpdateException`, `SocketException` underneath) and doesn’t translate it into your vocabulary. The repository is

the house where that translation happens exactly once. The ORM lives *inside* the repository, as one more driver: it's one of the things the boundary wraps.

Preview of the fake: the interface you'll thank in chapter 17

All this interface discipline pays off somewhere that only shows up in testing. Since the repository is the one piece that touches network, database, and disk, it's also the one piece you don't want to call for real in a unit test: a test that opens a socket is slow, flaky, and depends on the server being up. The way out is swapping the real implementation for a double, and the interface from the start of the chapter is what makes the swap trivial.

A **repository fake** is the same interface implemented in memory, one that hands back canned data or a simulated failure, without touching any IO at all.

Dart

```
// THE FAKE: same interface, in memory. The network outage is a flag,  
  
// not a socket. The test controls the outcome without touching any IO  
  
// at all.  
  
class TabRepositoryFake implements TabRepository {  
  
  TabRepositoryFake({this.simulateNetworkOutage = false});  
  
  final bool simulateNetworkOutage;
```

```
@override

LookupResult findTab(int table) {

    if (simulateNetworkOutage) {

        return InfraFailure(Failure.noConnection);

    }

    return TabFound(

        Tab(

            table,

            const [

                (name: "Espresso", priceInCents: 700),

                (name: "Cappuccino", priceInCents: 1195),

            ],

            1895,

            true,

        ),
```

```
);  
  
}  
  
@override  
  
SaveResult save(Tab tab) {  
  
    if (simulateNetworkOutage) {  
  
        return InfraFailure(Failure.noConnection);  
  
    }  
  
    return TabSaved(tab);  
  
}  
  
@override  
  
SaveResult markAsPaid(Tab tab) => save(tab);  
  
}
```


A single `bool` in the constructor plays the role of all the network's instability. The test flips the flag, calls `markAsPaid`, and checks that it gets back a typed `InfraFailure(Failure.noConnection)`, with no server and no framework double. The `confirm` in the snippet is the snippet's own assertion, which prints "ok" when the condition holds and blows up when it doesn't; in your project, swap it for `expect` from `package:test`:

Dart

```
// Simulated network outage: the write returns a typed noConnection.

final TabRepository offline =

    TabRepositoryFake(simulateNetworkOutage: true);

final saved = offline.markAsPaid(Tab(4, const [], 1895, true));

switch (saved) {

    case TabSaved():

        confirm(

            "with the network down, the write should not go through",

            false,
```

```

    );

    case InfraFailure(:final failure):

        confirm(

            "markAsPaid with the network down returns a typed InfraFailure",

            failure == Failure.noConnection,

        );

    }

```

The fake steps into the real one's place through dependency injection (chapter 9), the same technique already holding up the rest of the app: whoever builds the orchestrator receives a `TabRepository`, and in the test that parameter is the fake. Notice where injection pays off and where it doesn't. In chapter 14's use case, which is a pure function, you inject nothing; you call the function with data and compare the Result, no double needed. It's at the IO boundary that injection earns its keep, because that's where an external-world dependency exists to swap out. Injecting an interface into a pure use case would be ceremony with no payoff; injecting one into the repository is what makes the test possible at all. Chapter 17 is entirely about this, and it reuses exactly this fake.

Try it: open <https://focus.kodel.com.br/en/dart/15-02> and flip the `simulateNetworkOutage` flag in the happy-path case. Prediction: the lookup's assertion fails on the spot, because

the fake started returning `InfraFailure` where the test expected the tab. That's the test proving the network drop arrives typed, with no socket ever opened. The other nine languages are at `/en/kotlin/15-02`, `/en/ts/15-02`, `/en/java/15-02`, `/en/csharp/15-02`, `/en/go/15-02`, `/en/php/15-02`, `/en/python/15-02`, `/en/swift/15-02`, and `/en/rust/15-02`.

Pitfalls

The first pitfall is a business rule hiding inside a “clever” query. The repository fetches the tab, and someone figures it's elegant to filter right there: fetch only the tabs belonging to customers with more than 100 loyalty points. It looks like optimization, saves a round trip. What goes wrong: the “earned the discount” criterion just gained a second home, inside a `WHERE`, far from the use case that decides it, and the day the rule changes, the repository falls behind. How to get out: the repository fetches the whole tab, and the use case decides who earns what. The query brings data; it doesn't judge.

Watch out for the inverted reading of that pitfall: it doesn't ban `WHERE`, `ORDER BY`, or pagination from the repository. Filtering is the query's job, and the database does it better than any layer of yours: fetching a thousand tabs to throw away nine hundred and ninety-eight in the application trades an index for wasted network traffic and memory. I call that waste the `findAll` trap, because it's always the same gesture: pull the whole table to filter it in memory. “Table 4's open tabs, newest first” is a legitimate method on the contract, with the filter and the ordering inside the query. The line separating the two cases is the *meaning* of the criterion: table, status, date, and page size describe *what data* you

want, and they go into the query; “earned the discount” is a *rule’s verdict*, and verdicts live in the use case. The repository selects; it doesn’t decide.

The second pitfall is an IO `catch` leaking into another layer. The orchestrator or the use case receives the repository’s Result and, “just to be safe,” wraps the call in a `try/catch` too. What goes wrong: you just recreated the anti-solution’s duplication, one layer up, and now there are two places translating the exception, with the risk that they disagree. The numbered tip at the end of the chapter is the test: if an infra exception showed up outside the repository, you have a leak, and the fix is removing the `catch` above, not adding one more.

Q&A

The repository is asynchronous in real life (the network takes time). Why is this chapter’s code synchronous? So the canonical scenario runs the same, short way across all ten languages, without dragging each one’s concurrency model into the example. The mechanism (`async/await` in Dart and C#, `suspend` in Kotlin, a goroutine in Go) is orthogonal to the thesis: the single translation happens in the same place, synchronous or not. Swap the return for a `Future` / `Promise` and the `catch` stays exactly where it is.

Where does caching fit in, then? Shouldn’t the lookup keep the last result around? It can, and the place for it is inside the repository, invisible to the caller. Caching is a detail of how the repository fulfills the contract, not an item on the interface. What chapter 13’s third pitfall bans is caching *in the orchestrator*, which creates a second owner of the data; inside the repository, in one single place, it’s legitimate.

So are `WHERE` , `ORDER BY` , and pagination banned from the repository? No; they're its job. Filtering, ordering, and paginating at the source is what keeps a query cheap, and pulling everything to sift through it in the application is the earlier section's `findAll` trap. What the first pitfall bans is something else: a criterion carrying a business decision ("who earned the discount") hidden inside the query. Selecting which data to bring back is the repository's job; deciding what the data means is the use case's.

Why two Results, `LookupResult` and `SaveResult` , if the failure is the same? Because the success isn't the same: reading returns a found tab, writing confirms a saved tab, and those are gestures of a different nature. The infra failure, that one really is shared, which is why `InfraFailure` implements both. That asymmetry (different success, same failure) is the seed of chapter 16's distinction.

Quick tip

Open your project's fattest repository and search it for `if` and for `where` / `filter` . Every condition whose operand is a business value (loyalty points, a price range, "earned it") is a rule that leaked into the query, and its destination is a use case. The selection conditions (table 4's tab, today's open ones, ordered by date, twenty per page) stay right where they are: that `WHERE` is the repository doing its job, in the cheapest place to do it. The repository filters by selection; never by policy.

Quick reference

The question “does this belong in the repository?” answers itself by nature:

Situation	Fix
Infra exception (network, database, disk)	repository: translates it into a <code>Failure</code>
IO <code>try/catch</code> in another layer	leaked: remove it, the repository already translated
Deciding who earned the discount	leaked: the decision goes to the use case (chapter 14)
Fetching table 4's tab	repository: it owns access to the data
Screen updates itself when the data changes	orchestrator (chapter 13)
Generic method <code>save<T></code> or <code>getAll()</code>	out: only the feature's own verbs
<code>WHERE</code> / <code>ORDER BY</code> /page by table, date	repository: selection is the query's job
<code>WHERE</code> that decides discount policy	leaked: the rule goes to the use case
Caching the lookup's last result	repository, in one place, invisible

Exercises

1. Add the variant `Failure.timedOut`, translated from `TimeoutException`, to snippet 15-01 in your language. Start with the repository: add a second `catch` (or error arm) that turns `TimeoutException` into `InfraFailure(Failure.timedOut)`, next to the `noConnection` that's already there. Now follow the compiler's errors: in Dart, Kotlin, Swift, and Rust, every consumer `switch / match` that doesn't handle `timedOut` becomes non-exhaustive and the compiler points at the exact spot; in C#, PHP, and Python, it's the `_` arm or the type checker that charges you. Finish with the base rule intact: `noConnection` keeps working, and the output gains a third line with the new timed-out case, without you touching the repository in more than one spot.
2. Rosie wants the app to work offline: if the network drops during a lookup, the screen should show the last known tab instead of an error. Could you add a cache inside the repository, invisible to the orchestrator, that returns `TabFound` with the stale data when the network drops, without moving a single line of code outside the boundary? Think about where the `SocketException` `catch` decides between the cache and the `Failure`.

Tip 15

Every infrastructure exception dies in the repository; if it showed up in another layer, you have a leak.

Next chapter: the repository has two verbs of a different nature, finding and saving, and this chapter treated them almost the same, with the same `InfraFailure` on both sides. Do asking and changing really deserve the same treatment, or is there an asymmetry this chapter hasn't charged for yet?

Commands and Queries: CQS Without Ceremony

In this chapter, you'll:

- classify any operation in Rosie's Coffee Shop app as a command or a query using Meyer's rule, without hesitating over the eight real operations;
- refactor the hybrid method `payAndGetTab()` into a command that returns a `Result` and a query the orchestrator publishes as state;
- explain why FOCUS stops at CQRS-lite, owning Fowler's criticism instead of arguing it away.

You've been separating writes from reads since chapter 13 without knowing the name for it. The orchestrator that fetches and publishes, the use case that returns a `Result`, the two distinct sealed types from chapter 15: all of it is already a discipline with a name, a surname, and a birth year. This chapter gives it the name, shows the rule behind it, and draws the exact line where FOCUS stops following it.

Chapter 15 closed with a question: do asking and changing deserve the same treatment? The answer was already planted in that chapter. Notice that the repository returns two distinct sealed types, `LookupResult` for reads and `SaveResult` for writes, with `TabFound` on one side and `TabSaved` on the other. That split wasn't a typing whim. It's half of a rule from 1988 that this chapter

presents in full, and you built the other half back in chapter 13, when the orchestrator fetched the tab from the repository and published the state to the screen. One thing is still missing: the code that breaks the rule, so the pain shows up before the fix, the way every chapter in this part does it.

The anti-solution: paying and asking in the same gesture

Rosie's Coffee Shop's payment feature needs two things: charge the customer and show the updated tab on screen. A developer in a hurry solves both at once, in a single method, and the signature even looks convenient: you call it, the customer gets charged, and the tab comes back ready to render.

Dart

```
// The hybrid: looks up, charges, and returns the tab, all in one gesture.
```

```
Tab payAndGetTab(int table) {
```

```
    final lookup = repository.findTab(table);
```

```
    final tab = switch (lookup) {
```

```
        TabFound(:final tab) => tab,
```

```
        InfraFailure() => const Tab(0, [], 0, false),
```

```

};

// Charges the customer. The write's outcome gets thrown away: if

// the network drops right here, nobody finds out.

repository.markAsPaid(tab);

// And it returns the "updated" tab, as if the charge had gone

// through. The signature promises data; the mutation shipped with

// no receipt.

return tab;

}

```

Read the signature before the body. `Tab payAndGetTab(int table)` promises read data, and that's all the caller sees. The body, though, does a second thing: it calls `repository.markAsPaid(tab)`, a write from chapter 15, and throws away the `SaveResult` it returns. The type that carried the charge's outcome died on a line with no assignment.

While Rosie's Coffee Shop's network stays up, this method works, and that's what makes it dangerous. The bug doesn't show up the day you write it; it shows up on a busy Saturday

night, when the Wi-Fi drops between the lookup and the write. The lookup went fine. The write returned

`InfraFailure(Failure.noConnection)`, and nobody read it. The method returns the tab as usual, the screen shows “paid,” and now nobody can answer the question that matters: did the customer get charged? The server never recorded the payment, the screen says it did, and Rosie finds the gap when she closes the register. A returned value is an answer; this method answered the wrong question.

Try it: run <https://focus.kodel.com.br/en/dart/16-01>. The output shows table 4’s tab coming back whole, 1895 cents, with the network down partway through. Find the line in the code that discards the `SaveResult`: it’s a call with no `final` in front, and the compiler doesn’t complain about a thing. The same hybrid runs in the other nine languages, at <https://focus.kodel.com.br/en/ts/16-01>, <https://focus.kodel.com.br/en/kotlin/16-01>, <https://focus.kodel.com.br/en/java/16-01>, <https://focus.kodel.com.br/en/csharp/16-01>, <https://focus.kodel.com.br/en/go/16-01>, <https://focus.kodel.com.br/en/php/16-01>, <https://focus.kodel.com.br/en/python/16-01>, <https://focus.kodel.com.br/en/swift/16-01>, and <https://focus.kodel.com.br/en/rust/16-01>.

CQS: Meyer’s rule

The pain has had a name and a diagnosis since 1988. CQS (Command-Query Separation) is the rule Bertrand Meyer wrote down in *Object-Oriented Software Construction* (1988): a method changes state or returns data, never both. The etymology helps it

stick. A **command** is an order: “pay the tab” changes the world and earns a receipt saying whether the order went through. A **query** is a question: “what’s table 4’s total?” changes nothing and earns an answer. The hybrid from the last section gave an order and returned the answer to a different question; the order’s receipt went in the trash.

The refactor splits the two gestures, and its best part is what you won’t write: no new type. The command returns the `SaveResult` chapter 15 already defined; the repository does the writing, and the orchestrator (chapter 13) tells it to, because a use case doesn’t do IO. This particular command has no rule to decide, so it doesn’t even need a use case; the day it does, the rule decides first, in a pure function that returns a `Result`, the shape chapter 14 built. The query is `findTab`, spelled exactly as chapter 15 spelled it; it returns the read `Result` chapter 8 introduced.

Dart

```
// features/tab/pay_tab.dart

// THE COMMAND: changes state and returns the order's outcome, nothing

// else. The repository from chapter 15 does the writing, on the

// orchestrator's orders; a business rule, once one exists, decides

// first in the use case (chapter 14).

SaveResult payTab(Tab tab) =>

    repository.markAsPaid(tab);
```

```
// THE QUERY: answers the question, exactly as chapter 15 published it.
```

```
// The orchestrator (chapter 13) publishes this result as state.
```

```
LookupResult findTab(int table) => repository.findTab(table);
```

Two lines of body, and the whole pain is gone. `payTab` takes the tab and returns `SaveResult`: either `TabSaved` or `InfraFailure`, and the exhaustive `switch` from chapter 8 forces the caller to face both cases. There's no more way for the charge to fail in silence, because the outcome now IS the return value, and a sealed type's return doesn't get discarded without the compiler flagging the untreated variant in whoever consumes it. `findTab` stayed untouched: it's the same signature the orchestrator from chapter 13 already called to publish state. The screen that used to show “the tab the payment returned” now shows “the state the orchestrator published after looking up again,” and those two sentences describe different worlds: in the second one, the screen never lies.

The split, in all ten languages

Two functions and no new type: that's the shape of the refactor, and it crosses all ten languages in the book without losing anything along the way. In the nine listings below, look for the same pair every time: one function whose return is the order's receipt, and one function whose return is the question's answer. What changes from language to language is where the pair lives and how it reaches the repository.

TypeScript writes the same split with the discriminated unions from chapter 15, plus one extra detail: both functions return a `Promise`, because in the JS ecosystem the real repository is always asynchronous. The `async` wrapper doesn't change the rule; it only changes how the same `Result` gets delivered.

TS TypeScript

```
// THE COMMAND: changes state and returns the order's outcome, nothing else.
```

```
async function payTab(tab: Tab): Promise<SaveResult> {
```

```
    return repository.markAsPaid(tab);
```

```
}
```

```
// THE QUERY: answers the question, exactly as chapter 15 published it.
```

```
async function findTab(table: number): Promise<LookupResult> {
```

```
    return repository.findTab(table);
```

```
}
```

Kotlin, C#, and Java form the next family: in all three, the pair lives inside a class that takes the repository through its constructor, `PaymentService`. It's not a new layer; it's the same pair

with an address, and chapter 9 already justified the injection. Kotlin opens the family, and its single-expression `=` fits each operation into one line:

Kotlin

```
// features/tab/PaymentService.kt (CQS version)

// Command and query, split apart: whoever calls payTab RECEIVES the
// write's outcome and the exhaustive when forces them to handle it.

class PaymentService(private val repository: TabRepository) {

    // Command: changes the world and returns the outcome, no read data.

    fun payTab(tab: Tab): SaveResult =

        repository.markAsPaid(tab)

    // Query: only reads, no hidden side effect.

    fun findTab(table: Int): LookupResult =

        repository.findTab(table)

}
```

C# writes the same class with expression-bodied members, the `=>` that's Kotlin's `=` cousin. The difference that matters shows up in the consumer: as chapter 8 warned, C#'s exhaustiveness is weak, so the `switch` over `SaveResult` needs a `_` arm that throws at runtime.



```
class PaymentService
{

    private readonly ITabRepository _repository;

    public PaymentService(ITabRepository repository) =>

        _repository = repository;

    // Command: changes the world and returns the outcome, no read data.

    public SaveResult PayTab(Tab tab) =>

        _repository.MarkAsPaid(tab);

    // Query: only reads, no hidden side effect.
```



```

public LookupResult FindTab(int table) =>

    _repository.FindTab(table);

}

```

Java closes the family with more ceremony and the same anatomy: a final field, an explicit constructor, two one-line methods. In exchange, Java 21's pattern `switch` over the `sealed interface` from chapter 15 is genuinely exhaustive, and the compiler bills you for the failure case the anti-solution used to swallow.

Java

```

static final class PaymentService {

    private final TabRepository repository;

    PaymentService(TabRepository repository) {

        this.repository = repository;

    }

    // Command: changes the world and returns the outcome, no read data.

    SaveResult payTab(Tab tab) {

```

```

        return repository.markAsPaid(tab);

    }

    // Query: only reads, no hidden side effect.

    LookupResult findTab(int table) {

        return repository.findTab(table);

    }

}

```

Swift, PHP, Python, and Rust take a different road, and it's just as valid: free functions that take the repository as their first parameter. No class, no field, no constructor. It's the usual trade between constructor injection and parameter injection, and CQS doesn't care which one you pick, because its rule lives in each function's signature. Swift shows the shape:

Swift

```

// Command: changes the world and returns the change's outcome, nothing more.

func payTab(

    _ repository: TabRepository, _ tab: Tab

```

```

) -> SaveResult {

    return repository.markAsPaid(tab)

}

// Query: answers a question without changing anything.

func findTab(

    _ repository: TabRepository, _ table: Int

) -> LookupResult {

    return repository.findTab(table)

}

```

PHP writes the same two functions with declared return types, and those types carry the contract: `SaveResult` on the order, `LookupResult` on the question. Without those two types in the signature, PHP would let the hybrid through without a complaint.

 **PHP**

```

// Command: changes the world and returns the change's outcome, nothing more.

function payTab(

```

```

    TabRepository $repository,

    Tab $tab,

): SaveResult {

    return $repository->markAsPaid($tab);

}

// Query: answers a question without changing anything.

function findTab(

    TabRepository $repository,

    int $table,

): LookupResult {

    return $repository->findTab($table);

}

```

Python uses type annotations for the same reason, with one difference that matters: they're worth nothing at runtime. What enforces the split is the type checker, and it's the type checker that flags a command returning `LookupResult`. Without mypy in your pipeline, CQS in Python turns into a code-review discipline.



```
# Command: changes the world and returns the change's outcome, nothing more.
```

```
def pay_tab(  
  
    repository: FakeTabRepository, tab: Tab  
  
    ) -> SaveResult:  
  
    return repository.mark_as_paid(tab)
```

```
# Query: answers a question without changing anything.
```

```
def find_tab(  
  
    repository: FakeTabRepository, table: int  
  
    ) -> LookupResult:  
  
    return repository.find_tab(table)
```

Go tells the whole chapter's story on its own, in the signature, and that's why it's worth reading slowly. Look at the pair of return types: the command returns `error` and nothing else; the query returns `(Tab, error)`.



```
// THE COMMAND: changes state and returns only the order's outcome. A
// clean signature: error, and nothing else.

func payTab(tab Tab) error {

    return repository.MarkAsPaid(tab)

}

// THE QUERY: answers the question, exactly as chapter 15 published it.

func findTab(table int) (Tab, error) {

    return repository.FindTab(table)

}
```

Now compare that to the hybrid from the first section, whose Go signature reads `func payAndGetTab(table int) (Tab, error)`. In Go there's no hiding a dual nature: the pair `(T, error)` is the only way to return both data and an outcome, so the hybrid confesses in its signature that it does both, and it reads ugly. That ugliness is a feature. In single-return languages, `Tab payAndGetTab(...)` looked innocent, because the discarded Result happened out of sight, in the body; in Go, the `(Tab, error)` signature on a method named

“pay” shouts that there’s too much going on there. If your Go method’s signature mixes both without being a query, CQS got violated, and you didn’t even have to open the body to know it.

Rust closes the loop. `&dyn TabRepository` is the repository arriving as a reference to a trait object, Rust’s way of accepting any implementation of chapter 15’s contract, real or fake. The rest is the same pair:

Rust

```
// Command: changes the world and returns the change's outcome, nothing more.

fn pay_tab(

    repository: &dyn TabRepository,

    tab: Tab,

) -> SaveResult {

    repository.mark_as_paid(tab)

}

// Query: answers a question without changing anything.

fn find_tab(

    repository: &dyn TabRepository,
```

```
table: i64,  
  
) -> LookupResult {  
  
    repository.find_tab(table)  
  
}
```

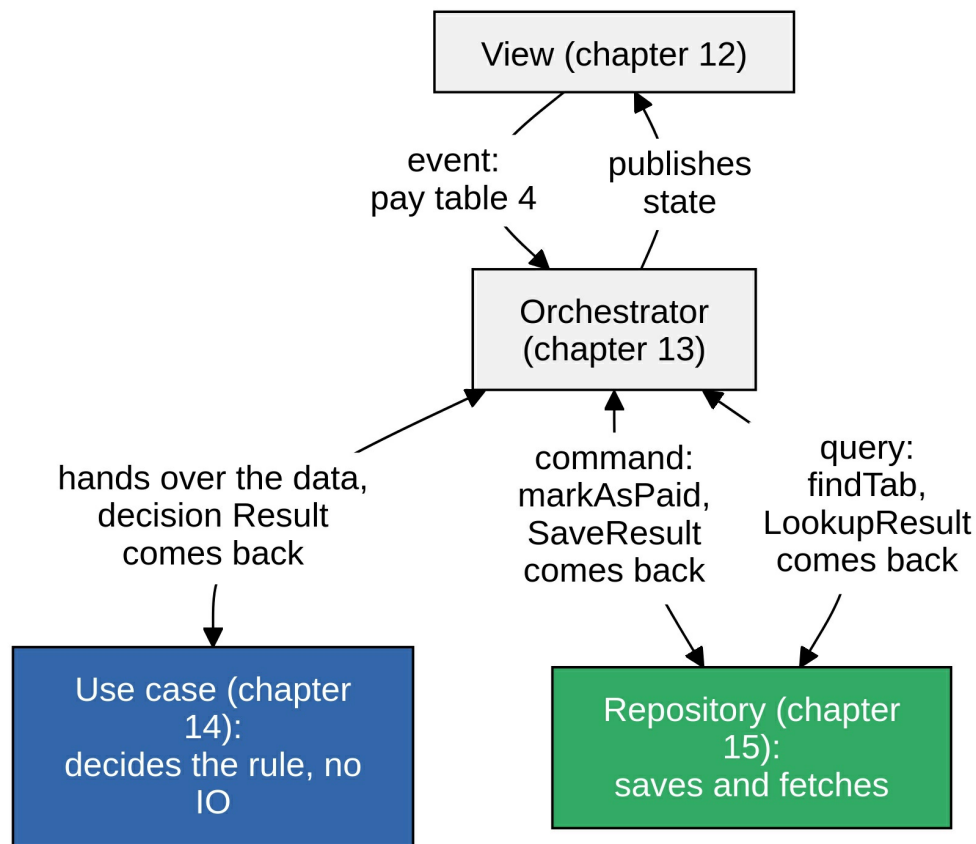
Ten languages, three ways to host the pair: a class method in Kotlin, C#, and Java; a free function with the repository as a parameter in Swift, PHP, Python, and Rust; a top-level function with the repository injected through a variable in Dart, TypeScript, and Go. None of them needed a new type, a library, an annotation, or a framework. CQS costs one signature.

Try it: run <https://focus.kodel.com.br/en/dart/16-02> and play out the scenario: paying with the network up prints `TabSaved(4)`, paying with the network down prints `InfraFailure(Failure.noConnection)`, and looking up prints table 4's `TabFound`. Try discarding `payTab`'s return the way the hybrid did: the code still compiles, but the outcome is now a value in your hands, and ignoring it becomes a visible decision in the diff, not an accident. The other nine languages are at <https://focus.kodel.com.br/en/ts/16-02>, <https://focus.kodel.com.br/en/kotlin/16-02>, <https://focus.kodel.com.br/en/java/16-02>, <https://focus.kodel.com.br/en/csharp/16-02>, <https://focus.kodel.com.br/en/go/16-02>, <https://focus.kodel.com.br/en/php/16-02>, <https://focus.kodel.com.br/en/python/16-02>,

<https://focus.kodel.com.br/en/swift/16-02>, and <https://focus.kodel.com.br/en/rust/16-02>, with the same three-line output.

The canonical table's two tracks

The refactor you just did uncovered an architecture that was already there. Look at the whole flow in a single diagram, with the command going down one track and the query coming back on the other:



The command track goes down in two steps, and each step has an owner. When the order involves a rule, the orchestrator hands the data to the use case, and that's the Use Case row from chapter 10's canonical table: "the only place for business rules, a pure

function, takes data and returns a Result,” with the ban on “IO, framework, and domain exception.” That ban is the detail the diagram has to respect: a use case decides and returns the decision’s Result, but it never writes. The write is the second step: the orchestrator tells the repository to write and gets back the `SaveResult`. Today’s `payTab` is only that second step, because it has no rule to decide yet.

There’s a subtlety worth facing head-on here, because it comes back in the table of eight operations. Chapter 14’s `applyLoyaltyDiscount` is the first step in action, and on its own, by Meyer’s ruler, it’s a query: a pure function, takes data, returns a verdict, and changes nothing in the world. What changes the world is the second step. A whole business gesture (“apply the discount and charge”) is usually a query followed by a command, and Meyer’s ruler applies to each method, never to the whole gesture. Mixing up the two levels is the most common mistake anyone classifying for the first time makes.

The query track comes back: the orchestrator fetches, the repository answers, the state reaches the screen. It’s the Orchestrator row from the same table: “converts event to state, fetches data from the repository, calls use cases, publishes state,” with the ban on “deciding rules and persisting.” The pair “fetches data from the repository” and “publishes state” is FOCUS’s definition of a query, and chapter 13 built it before you knew its name. Chapter 15’s repository answers on demand; the one who turns that answer into state for the View is the orchestrator, never the repository.

Two rows of the table, two tracks, two kinds of operation. The canonical table was already Meyer’s CQS, written in the vocabulary of layers.

From CQS to CQRS, and where FOCUS stops

Twenty years after Meyer, the method-level rule leveled up. **CQRS** (Command Query Responsibility Segregation) is the pattern Greg Young named in 2010: instead of separating methods, separate the models themselves, one object for writes and one for reads, each free to evolve on its own. The difference in level matters more than the similar-sounding name. CQS is a method rule: it fits in a signature and costs nothing. CQRS is an architecture decision: it splits the system into two paths and charges maintenance on both.

A mythology grew up around CQRS that Young himself spent years dismantling, in the article “CQRS, Task Based UIs, Event Sourcing agh!” (2010), and that Oskar Dudycz revisits on event-driven.io. Three myths fall at once. CQRS doesn’t require *Event Sourcing*, the technique of storing the sequence of events that happened instead of the final state: Young presented the two together and the market married them, but a CQRS system can write ordinary state just fine. CQRS also doesn’t need two databases, because the segregation is of the model, and both models can live in the same database. And CQRS doesn’t need eventual consistency: stale reads are an implementation choice, outside the definition. If you’ve ever turned down CQRS “because I don’t want two databases,” you turned down a myth.

Once the myths clear out, the real criticism remains, and it comes from Martin Fowler, in the “CQRS” bliki entry: “for most systems CQRS adds risky complexity.” Fowler is right, and FOCUS isn’t going to pretend otherwise. Two models is twice the code for the same feature, and the sync between them is a problem most apps never needed to have.

I carry a scar from that complexity. I watched a team adopt full CQRS, two databases and projections, for a 12-screen CRUD registration flow. Syncing the write database with the read database burned more development hours than all 12 screens combined, and the first question in every bug report became “are the databases in agreement?” I wouldn’t do it again even on a system ten times bigger; the pain bought no benefit, because no read in that system ever needed to diverge from the write.

FOCUS’s position distills that experience into one term: **CQRS-lite** is the logical split between commands and queries, with none of the distributed cost. Commands change state and return a Result, with the rule decided in the use case and the write done in the repository; queries are on-demand reads from the repository, published as state by the orchestrator. One database, no events, no eventual consistency. It’s everything chapters 13, 14, and 15 already built, plus the discipline of never mixing the tracks, and the extra price is zero, because the structure was already standing. CQS’s clarity, without CQRS’s bill.

Classify the eight operations

Meyer’s rule is only worth what you can apply to a menu of real operations. Take Rosie’s Coffee Shop app’s eight and ask, for each one: does it change state, or answer a question?

Operation	Classification	Why
Pay the tab	command	changes state; returns a Result
Show the tab total	query	answers a question; becomes state

Apply the loyalty discount	query	a pure function decides (chapter 14)
List the menu	query	on-demand read
Mark an item out of stock	command	changes state
Split the tab	command	changes state
Look up loyalty points	query	answers a question
Register an order	command	changes state

Four commands, four queries, no operation on both teams. The discount row is usually the most contested, and it's the one that teaches the most: chapter 14's use case takes the tab and the points, returns a Result with the amount discounted, and writes nowhere. Question asked, answer given. Whoever saves the discount afterward is the payment command, in a second method. The one that tricks people most is "register the order and show the total," which sounds like it wants a hybrid just like the anti-solution's. It's two operations: the command "register order" returns the write's Result, and the query "show total" fetches and publishes the new total. The screen's flow chains the two; the code doesn't fuse them. Every time a feature "needs" a method that changes and returns, redo this split; it's the same exercise as this chapter's refactor, with different names.

Pitfalls

The classic CQS pitfall is the query that “takes advantage” of the trip to update something. Rosie asks: “I want to know how many times the menu got viewed.” The developer figures it’s efficient to bump a counter inside `listMenu()`, since “the query’s already right there.” What goes wrong: the read turned into a write in disguise, and now showing the menu twice counts two visits, the screen’s automatic retry inflates the metric, the test that calls the query to set up a scenario changes the database, and the cache chapter 15 allowed inside the repository starts hiding writes. A side effect in a read is Meyer’s rule violation number one. How to get out: the counter is state, so changing it is an order. Create the command `recordMenuView()` and let the query only ask; the orchestrator decides when to fire the command, on the screen-opening event, once.

The second pitfall is the command that returns read data “for convenience”: `payTab` handing back the whole tab so the screen can skip a fetch. It’s the anti-solution’s hybrid coming back thin. A command’s Result carries the order’s outcome, and an outcome is a different thing than screen data; the day the screen needs more fields, the command swells right along with it, and the two tracks tangle up again.

The third is concluding that adopting the split forces you to adopt the infrastructure: “if it’s CQRS, I need two databases.” Reread Young’s and Dudycz’s myths from the earlier section. FOCUS stays at the lite version exactly so you collect the logical split while paying zero extra infrastructure.

Q&A

What about a command that needs to return the generated id, like “register order” creating a new tab? A command’s Result carries the order’s outcome, and the outcome can name the thing it created: `TabSaved` already carries the tab

inside, the way chapter 15 defined it. What a command doesn't return is read data for the screen to lay out; that's a question, and a question is a query. A receipt with a confirmation number, yes; a receipt with the full statement, no.

Can a query never have any effect at all? Not even a log?

Meyer's criterion is observable domain state. A log, an infrastructure metric, and the repository's internal cache (chapter 15) don't change the answer to any business question, so they don't violate the rule. The view counter from the pitfall above does: it's data Rosie wants to read, so it's domain state, so only a command may touch it.

`payTab` just delegates to `markAsPaid`. **Why the layer, for one line?** Today it's one line; the track is what matters. The day the rule "a split tab can't be closed out by a single waiter" shows up, it goes into the use case, the only place for business rules (chapter 14), and no caller changes. Without the track, the rule would be born in the orchestrator or the repository, the two homes the canonical table bans it from.

Quick tip

Open any repository or service file in your current project and search its returns: a method with a change verb in its name (`pay` , `save` , `apply` , `register`) returning the whole object is a hybrid candidate. In five minutes you'll have your codebase's list of `payAndGetTab` s; this chapter's refactor works the same way on every one of them.

Quick reference

Situation	Fix
Changes state (pay, register, mark)	command: returns a Result
Rule to decide before writing	pure use case (chapter 14), then the command
Answers a question (total, menu, points)	query: the repository fetches
Query answered for the screen	the orchestrator publishes it as state (chapter 13)
Method changes state AND returns data	split it: <code>payTab()</code> and <code>findTab()</code>
Command “needs” to return screen data	the Result is the outcome; use the query
Query “takes advantage” to save a counter	the counter is state: create the command
“Do I need two databases?”	no: CQRS-lite is logical, one database, zero events

Exercises

1. Classify the eight operations from this chapter’s table without looking at the answer column: pay the tab, show the tab total, apply the loyalty discount, list the menu, mark an item out of stock, split the tab, look up loyalty points, register an order. For each one, also write down what the return type would be

in your language: a save Result for the commands, data (or a lookup Result) for the queries. Check yourself against the rule: changes state, command; answers a question, query.

2. Open <https://focus.kodel.com.br/en/dart/16-01> (or your language's route) and refactor the hybrid yourself: delete `payAndGetTab()` and write `payTab()` and `findTab()` as two separate functions, reusing the types from chapter 15 that are already in the snippet. Can you get the output to tell the truth by printing the `InfraFailure(Failure.noConnection)` the hybrid used to swallow, without creating a single new type?

Tip 16

A method that changes state returns a Result; a method that answers a question returns the data. If it returns both, that's two methods.

Next chapter: the two tracks you just split ask for different kinds of proof, and that's exactly what chapter 17 builds: each track calls for a different kind of test.

Test Each Piece the Way It Asks to Be Tested

In this chapter, you'll:

- write the loyalty feature's suite for Rosie's Coffee Shop: chapter 14's use case test with no double at all, and chapter 13's orchestrator flow test with chapter 15's repository fake;
- justify in writing where integration testing pays for itself and where it doesn't, operation by operation, across the coffee shop;
- explain why a test that checks call order breaks on a refactor that doesn't change behavior.

Since chapter 10, FOCUS has promised that every layer is easy to test. That promise comes due today. You're going to watch a green suite turn red without the program changing behavior, and you're going to see why the other suite, written against that same refactor, stays green. The difference between the two isn't a matter of style: it's what each one chose to assert.

You arrive here with four pieces built and none of them tested. Chapter 12's dumb View fires an event and renders state. Chapter 13's orchestrator converts an event into state. Chapter 14's loyalty discount is a pure function. Chapter 15's repository translates an infrastructure exception into a Result, and it brought along an in-memory fake. Chapter 16 split the two tracks, command and

query. Each of these pieces asks for a different kind of proof, and this chapter's thesis is that you don't choose which: the architecture already chose for you. Whoever separated rule from IO earned a cheap testing base. Whoever didn't pays in doubles.

Before the technique, the pain.

The anti-solution: the suite that asserts the how

The operation that crosses all four layers is paying the tab. The orchestrator receives the event, looks up the tab in the repository, tells it to mark the tab paid, and publishes the state. A developer sits down to test this and does the thing that looks like the most rigorous move in the world: puts a double in place of the repository and checks whether the orchestrator called the right methods, in the right order, the right number of times.

Test double (Gerard Meszaros's term) is the name he gave, in *xUnit Test Patterns* (2007), to any object that stands in for a real collaborator during a test. Meszaros cataloged five kinds; this chapter uses two, and the difference between them is this whole chapter's axis. A **mock** is an interaction checker: it asserts which methods got called and in what order. Keep that definition in mind. In the "Repository: the fake you already have" section, it gets contrasted with the definition of a fake.

Notice it takes just ONE double for the pain to show up. The orchestrator only receives one injected collaborator, the repository, and the whole suite leans on it.

Dart

```
class TabRepositoryMock extends Mock implements TabRepository {}
```

```
// Builds the double already taught to respond, fires the event, and
```

```
// returns the double for the assertions.
```

```
Future<TabRepositoryMock> payTableFour() async {
```

```
    final repository = TabRepositoryMock();
```

```
    final tab = Tab(
```

```
        4,
```

```
        const [(name: "Espresso", priceInCents: 700)],
```

```
        700,
```

```
        true,
```

```
    );
```

```
    when(() => repository.findTab(any()))
```

```
        .thenReturn(TabFound(tab));
```

```
    when(() => repository.markAsPaid(any()))
```

```
        .thenReturn(TabSaved(tab));
```

```
final orchestrator = TabOrchestratorWithPayment(

    repository,

    refactored: refactored,

);

orchestrator.add(PayTab(4));

await Future<void>.delayed(const Duration(milliseconds: 50));

await orchestrator.close();

return repository;
}

test("looks up the tab before marking it paid", () async {

    // Arrange + Act

    final repository = await payTableFour();
```

```
// Assert: the ORDER of the calls. Not one line looks at the state

// that went out the door.

verifyInOrder([

    () => repository.findTab(4),

    () => repository.markAsPaid(any()),

]);

});

test("looks up the tab twice", () async {

    // Arrange + Act

    final repository = await payTableFour();

    // Assert: the CALL COUNT. This is the line the refactor knocks down,

    // without a single comma of observable behavior changing.

    verify(() => repository.findTab(4)).called(2);

});
```

TypeScript

```
it("looks up the tab before marking it paid", () => {

    // Arrange + Act

    const repository = payTableFour();

    // Assert: the ORDER of the calls.

    const find = vi.mocked(repository.findTab);

    const mark = vi.mocked(repository.markAsPaid);

    expect(find.mock.invocationCallOrder[0]).toBeLessThan(

        mark.mock.invocationCallOrder[0]!,

    );

});

it("looks up the tab twice", () => {
```

```
// Arrange + Act

const repository = payTableFour();

// Assert: the COUNT.

expect(vi.mocked(repository.findTab)).toHaveBeenCalledTimes(2);

});
```

Read the two assertions and ask what they know about paying a tab. The answer is nothing. They know `findTab` got called before `markAsPaid`, and that the first one got called twice. Not one line looks at the state the orchestrator published, which is the only thing the waiter's screen ever sees. The suite asserts the HOW, and the how is exactly the part you have the right to change.

Why `called(2)` and not `called(1)`? Because the handler is written in a silly way, on purpose: it looks up the tab once to validate, then looks it up again to pay. It's the kind of duplication nobody reread. The mockist suite, green, is photographing exactly that defect.

Try it: open <https://focus.kodel.com.br/en/dart/17-01> (or your language's route) and run it. Before you look at the output, answer this: if someone fixes the duplicated lookup, which of the two assertions falls?

Now someone comes along and fixes it. The refactor extracts a private method `_pay` and reuses the result of the first lookup instead of querying the repository a second time. One lookup, not two.



```
// AFTER version: _pay extracted, the result of the first lookup

// reused, a single call. Same states, same payloads.

Future<void> _onPayTabAfter(

    PayTab event,

    Emitter<TabState> emit,

) async {

    emit>Loading();

    final lookup = _repository.findTab(event.table);

    switch (lookup) {

        case InfraFailure():

            emit(Failed("no connection"));
```

```

case TabFound(:final tab):

    await _pay(tab, emit);

}

}

```

Before showing the break, prove the refactor changed nothing. This order isn't ceremony: if the behavior had changed, the mockist suite would be right to complain, and this chapter's whole argument would collapse. The proof is running the flow suite against both versions and comparing the output character by character.

```

$ diff <(grep -v '^[0-9:~]* ' /tmp/flow-before.txt) \

    <(grep -v '^[0-9:~]* ' /tmp/flow-after.txt)

$ diff /tmp/output-before.txt /tmp/output-after.txt

```

Both diffs come out empty. Same sequence of states, same payloads, same output. The program does exactly what it did before. Now run the mockist suite against the new version:

```

00:00 +0: loading test/mockist_test.dart
00:00 +0: (setUpAll)
00:00 +0: looks up the tab before marking it paid
00:00 +1: looks up the tab twice

```

```

00:00 +1 -1: looks up the tab twice [E]
  Expected: <2>
  Actual: <1>
  Unexpected number of calls

package:matcher                                expect
package:mocktail/src/mocktail.dart 595:5  VerificationResult.called
test/mockist_test.dart 73:48                main.<fn>

00:00 +1 -1: (tearDownAll)
00:00 +1 -1: Some tests failed.

Failing tests:
  test/mockist_test.dart: looks up the tab twice

```

And the flow suite, against that same new version:

```

00:00 +0: loading test/flow_test.dart
00:00 +0: paying a tab that's found publishes Loading then Ready
00:00 +1: network outage on the LOOKUP publishes Loading then Failed
00:00 +2: failure on the SAVE publishes Loading then Failed
00:00 +3: All tests passed!

```

Red on one side, green on the other. `Expected: <2> / Actual: <1>` is the test saying the program stopped doing something it never promised to do, against code whose observable behavior hasn't changed by a comma since the previous version. The developer who did the refactor now has two bad options: undo the improvement, or open the test and adjust the number. Everyone picks the second one, at three in the afternoon on a Friday. That's when the test turns into a stamp: it starts recording what the code does, and a test that records what the code does catches no defect at all.

Use case: a pure test, no double at all

Move up a layer and look at the loyalty discount from chapter 14. The table from chapter 10 says the Use Case forbids “IO, framework, and domain exceptions.” Read that ban as a testing promise: if the layer can’t touch IO or a framework, there’s no dependency to fake. Zero doubles, because there’s nothing to double.

This is where the investment in purity from chapters 7 and 14 gets paid back, with interest. `applyLoyaltyDiscount` takes a tab and a number of points, and returns one of the three variants of `DiscountResult`, without querying a database, without asking any framework’s permission, and without depending on anything you’d need to set up first. You build the input by hand, call the function, and compare the output. That’s it. A function is the easiest thing there is to test.

There are three cases, one per variant. The third is the most interesting, because it sends spare points alongside an out-of-stock item: it proves the ORDER of the rule, not just the result.

Dart

```
test("applies 10% when there are enough points", () {  
  
    // Arrange: the tab is a value. Nobody needs a database to build one.  
  
    const tab = Tab(4, [  
  
        Item("Espresso", 700),  
  
        Item("Cappuccino", 1195),
```

```

], 4000);

// Act: the function under test is pure. Calling it is the whole test.

final result = applyLoyaltyDiscount(tab, 120);

// Assert: the variant that came out, and the value it carries.

check(result).isA<DiscountApplied>().has(

    (r) => r.tab.totalInCents,

    "totalInCents",

).equals(3600);

});

test("rejects for insufficient points and states how many are needed", () {

    // Arrange

    const tab = Tab(4, [Item("Espresso", 700)], 700);

```

```

// Act

final result = applyLoyaltyDiscount(tab, 40);

// Assert

check(result)

    .isA<NotEligibleForDiscount>()

    .has((r) => r.pointsNeeded, "pointsNeeded")

    .equals(100);

});

test("rejects for an out-of-stock item, before checking points", () {

    // Arrange: spare points, but one item out of stock. The rule's order

    // is what decides the outcome.

    const tab = Tab(4, [

        Item("Espresso", 700),

        Item("cheese bread", 600, outOfStock: true),

```

```

], 1300);

// Act

final result = applyLoyaltyDiscount(tab, 500);

// Assert

check(result)

    .isA<ItemOutOfStock>()

    .has((r) => r.name, "name")

    .equals("cheese bread");

});

```

Line by line. The Arrange block builds the tab with `const`: no database, no factory, no builder. The Act block is one line, because the function needs nothing beyond its arguments. The Assert block uses `package:checks`, which chains `isA<DiscountApplied>()` (*is it?*) to assert the variant and `has(...)` (*does it have?*) to drill down into the field. The `// Arrange`, `// Act`, and `// Assert` comments are subgoal labels: each one names the goal of the block that follows, and they exist because the listing runs past fifteen lines.

Two things deserve an explanation. The 4000 isn't the sum of the two items, which comes to 1895, and the difference is on purpose: `applyLoyaltyDiscount` doesn't add up a single item; it takes 10% off the `totalInCents` that already arrived calculated, and if it ever starts recalculating from the items, this is the test that breaks. The second is the `Tab`. Here it carries `items`, because the out-of-stock rule scans the list. In the flow tests, which never look at a single item, the published snippet carries a `Tab` reduced to table, total, and `canPay`, which is why this chapter's listings build tabs of different shapes.

Now the other nine. What changes from one language to the next is how each one expresses "this is the variant that came out," and how much the compiler helps.

TS TypeScript

```
it("applies 10% when there are enough points", () => {

  // Arrange

  const tab: Tab = {

    table: 4,

    items: [item("Espresso", 700), item("Cappuccino", 1195)],

    totalInCents: 4000,

  };
```



```

// Act

const result = applyLoyaltyDiscount(tab, 120);

// Assert

expect(result.kind).toBe("discountApplied");

if (result.kind === "discountApplied") {

    expect(result.tab.totalInCents).toBe(3600);

}

});

```

The `if` after `expect` is uncomfortable, and it's telling you something true. The discriminated union only narrows the type inside a block that tests the discriminant, and `expect` doesn't narrow anything as far as the compiler is concerned. Where Dart writes `isA<DiscountApplied>().has(...)`, TypeScript has to assert twice: once for the runner, once for the type checker.

Kotlin · Swift

From here on, the listings call `confirm`, and there's no point searching for that function in any library: it's a three-line checker defined right inside the snippet, which prints “ok” or “FAILED” next to the case name and makes the program exit

with an error if any case failed. It exists because these languages' playgrounds run a `main`, not a test runner, and three lines are enough to do here what a runner would.

```
confirm(  
  
    "rejects for insufficient points and states how many are needed",  
  
    declined == DiscountResult.NotEligibleForDiscount(40, 100),  
  
)  
  
confirm(  
  
    "rejects for an out-of-stock item before checking points",  
  
    outOfStock == DiscountResult.ItemOutOfStock("cheese bread"),  
  
)
```

Kotlin and Swift ship stacked because the assertion is the same sentence in both: compare the whole result against the expected variant, by value equality. `data class` in Kotlin and `enum` with associated values in Swift give structural equality for free, so the test doesn't need to drill down field by field. In both, a `when` or `switch` that forgot a variant wouldn't even compile.

```

confirm("applies 10% with enough points",

    applied instanceof DiscountApplied d

        && d.tab().totalInCents() == 3600);

confirm("rejects for insufficient points and states how many are needed",

    declined.equals(new NotEligibleForDiscount(40, 100)));

```

Java and C# also ship together, for the same reason: `record` in both languages generates structural equality, and the pattern matching of `instanceof` (Java) and `is` (C#) ties the type test and the field extraction into a single expression. The difference between the two is which compiler complains about an incomplete `switch`, and it's Java: in C#, exhaustiveness over a sealed hierarchy earns a warning, not an error.

Rust

```

#[test]

fn applies_ten_percent_with_enough_points() {

    // Arrange

    let tab = Tab {

```

```
        table: 4,

        items: vec![Item::new("Espresso", 700), Item::new("Cappuccino", 1195)
    ],

    total_in_cents: 4000,

};

// Act

let result = apply_loyalty_discount(&tab, 120);

// Assert

match result {

    DiscountResult::DiscountApplied(t) => {

        assert_eq!(t.total_in_cents, 3600);

    }

    other => panic!("expected DiscountApplied, got {other:?}"),

}
```

```
}
```

Rust leaves the Kotlin-and-Swift group for a practical reason: it has a test runner built in. `cargo test` finds any function tagged with `#[test]` in the same file as the code, so the listing shows the language's own testing idiom instead of the hand-rolled checker. The `match` in the Assert block is exhaustive by the compiler's own demand, and the `other` arm exists to give a readable error message, not to paper over a typing gap.

PHP

```
confirm(  
  
    'applies 10% with enough points',  
  
    $applied instanceof DiscountApplied  
  
    && $applied->tab->totalInCents === 3600,  
  
);
```

PHP has no sealed union and no automatic structural equality, so the test combines `instanceof` with a field-by-field comparison and uses `===` to avoid the type coercion of `==`. It's more verbose than Java for the same reason Java is more verbose than Kotlin: every feature the language lacks reappears as a line of test.

Go

```
// Act
```

```

result := ApplyLoyaltyDiscount(tab, 120)

// Assert

applied, ok := result.(DiscountApplied)

if !ok {

    t.Fatalf("expected DiscountApplied, got %T", result)

}

if got, want := applied.Tab.TotalInCents, 3600; got != want {

    t.Errorf("total = %d, want %d", got, want)

}

```

Go is the counterpoint in form. There's no assertion library here, and not by oversight: `if got != want { t.Errorf }` is the language's idiom, and the test gets longer in exchange for having nothing to learn beyond `if`. Notice the `t.Fatalf` in the first block against the `t.Errorf` in the second. The first one aborts, because continuing without the tab makes no sense; the second one records and moves on.



```
# Act
```

```
applied = apply_loyalty_discount(with_points, 120)
```

```
declined = apply_loyalty_discount(without_points, 40)
```

```
out_of_stock = apply_loyalty_discount(with_shortage, 500)
```

```
# Assert
```

```
confirm(
```

```
    "applies 10% with enough points",
```

```
    isinstance(applied, DiscountApplied)
```

```
    and applied.tab.total_in_cents == 3600,
```

```
)
```

```
confirm(
```

```
    "no variant went unhandled",
```

```
    {type(applied), type(declined), type(out_of_stock)}
```

```
    == set(DiscountResult.__args__),
```

)

Python is the counterpoint in content. Its suite has four cases, not three. In Dart, Kotlin, Swift, Java, and Rust, a `switch` that forgot a variant of `DiscountResult` doesn't compile, and forgetting a case turns into a compile error instead of a missing test. In Python, exhaustiveness only exists if an external type checker runs with `assert_never`, and that checker doesn't run in the CI (Continuous Integration) of anyone who just runs `pytest`. The work the compiler did for free in the other five languages comes back to your own suite. The price is the fourth case: it gathers the types of the three already-computed results into a set and compares that set against what the union declares.

Try it: open <https://focus.kodel.com.br/en/dart/17-02> (or your language's route) and delete the out-of-stock item case. Which test breaks: the out-of-stock one, or the discount-applied one?

Repository: the fake you already have

Go back to chapter 15 and look at what it left ready. Alongside the real repository and the boundary's `try/catch`, that chapter published a second implementation of the same contract, in memory, called `TabRepositoryFake`. It's been there since snippet 15-02. This chapter doesn't invent the fake: it harvests what chapter 15 planted.

Dart

```
class TabRepositoryFake implements TabRepository {
```



```
TabRepositoryFake({this.simulateNetworkOutage = false});

// The network outage becomes a flag, not a socket: the test controls
// the outcome without touching any IO at all.

final bool simulateNetworkOutage;

@override

LookupResult findTab(int table) {

    if (simulateNetworkOutage) {

        return InfraFailure(Failure.noConnection);

    }

    return TabFound(Tab(table, 1895, true));

}

@override
```

```

SaveResult save(Tab tab) {

    if (simulateNetworkOutage) {

        return InfraFailure(Failure.noConnection);

    }

    return TabSaved(tab);

}

@Override

SaveResult markAsPaid(Tab tab) => save(tab);

}

```

Three things about this class matter. Its size isn't one of them. First: it implements the three methods of the contract, the same three chapter 15 declared, not one more. Second: none of them touch network, disk, or a database; the network outage is a boolean flag, not a socket. Third: it fits on one screen, and it fits because chapter 15's contract was designed small on purpose, with verbs from the feature instead of `save(T)` and `getAll()`.

A **fake** is exactly that: a simplified, working implementation of a contract, which you use to check STATE. Meszaros (2007) draws the line right here. The mock in the anti-solution checked interaction: which methods got called, and in what order. The whole suite depended on the orchestrator continuing to call the repository the same way it called it on the day the test was written. The fake checks nothing. It works, and the checking is done by your test's assertion, which looks at the result. A mock asserts the path; a fake lets you look at the destination.

Meszaros cataloged five kinds of test double, and this chapter uses two on purpose. Dummy, stub, and spy have their place, and I'm not going to teach them here: the distinction that changes an architecture decision is fake versus mock, and carrying the whole taxonomy would only make you memorize names.

Orchestrator: flow test

The orchestrator from chapter 13 is the middle piece, and the table from chapter 10 says it forbids “deciding rules and persisting.” Read that again as a testing promise: if it doesn't decide rules and doesn't persist, its only collaborator is the repository, and the repository already has a fake. What's left to test? The flow. An event goes in, a sequence of states comes out.

Before the test, the target. Chapter 13 declared the `PayTab` event and never registered a handler for it, so this chapter builds one, in a subclass called `TabOrchestratorWithPayment`. There's no new business rule in it: it's chapter 13's cycle stitched together with chapter 15's lookup and save, and the operation itself is a command, in the exact sense chapter 16 gave that word.



```

class TabOrchestratorWithPayment extends TabOrchestrator {

  TabOrchestratorWithPayment(this._repository, {bool refactored = false})

    : super(_repository) {

    on<PayTab>(

      refactored ? _onPayTabAfter : _onPayTabBefore,

    );

  }

  final TabRepository _repository;

}

```

Now the test. There are three cases, and the fun part is the distance between them.

Dart

```

class FakeThatDoesNotSave extends TabRepositoryFake {

  @override

  SaveResult markAsPaid(Tab tab) =>

```

```

        InfraFailure(Failure.noConnection);

    }

    blocTest<TabOrchestratorWithPayment, TabState>(

        "paying a tab that's found publishes Loading then Ready",

        // Arrange: chapter 15's fake in its default setting, which finds

        // and saves.

        build: () => TabOrchestratorWithPayment(

            TabRepositoryFake(),

            refactored: refactored,

        ),

        // Act: an event goes in.

        act: (orchestrator) => orchestrator.add(PayTab(4)),

        // Assert: the sequence of states that comes out, not a word about

        // calls.

        expect: () => [isA<Loading>(), isA<Ready>()],
    ),

```

```
);
```

```
blocTest<TabOrchestratorWithPayment, TabState>(
```

```
  "network outage on the LOOKUP publishes Loading then Failed",
```

```
  // Arrange: the same fake, a different argument.
```

```
  build: () => TabOrchestratorWithPayment(
```

```
    TabRepositoryFake(simulateNetworkOutage: true),
```

```
    refactored: refactored,
```

```
  ),
```

```
  act: (orchestrator) => orchestrator.add(PayTab(4)),
```

```
  expect: () => [isA<Loading>(), isA<Failed>()],
```

```
);
```

```
blocTest<TabOrchestratorWithPayment, TabState>(
```

```
  "failure on the SAVE publishes Loading then Failed",
```

```
  // Arrange: a fake that finds the tab and doesn't save it.
```

```

build: () => TabOrchestratorWithPayment(

    FakeThatDoesNotSave(),

    refactored: refactored,

),

act: (orchestrator) => orchestrator.add(PayTab(4)),

expect: () => [isA<Loading>(), isA<Failed>()],

);

```

TypeScript

```

class FakeThatDoesNotSave extends TabRepositoryFake {

    override markAsPaid(_tab: Tab): SaveResult {

        return { kind: "infraFailure", failure: "noConnection" };

    }

}

// Fires the event and returns just the names of the published states.

```

```
function statesFrom(repository: TabRepositoryFake): string[] {

    const orchestrator = new TabOrchestratorWithPayment(repository, refactored)
    ;

    orchestrator.payTab(4);

    return orchestrator.states.map((s) => s.kind);

}

it("publishes loading then ready when it finds the tab", () => {

    // Arrange: chapter 15's fake in its default setting.

    const repository = new TabRepositoryFake();

    // Act + Assert

    expect(statesFrom(repository)).toEqual(["loading", "ready"]);

});
```



```
it("publishes loading then failed on a network outage during LOOKUP", () => {

    // Arrange: the same fake, a different argument.

    const repository = new TabRepositoryFake(true);

    // Act + Assert

    expect(statesFrom(repository)).toEqual(["loading", "failed"]);

});

it("publishes loading then failed on a failure during SAVE", () => {

    // Arrange: a fake that finds the tab and doesn't save it.

    const repository = new FakeThatDoesNotSave();

    // Act + Assert

    expect(statesFrom(repository)).toEqual(["loading", "failed"]);

});
```

Compare the first case with the second. The only difference between them is one construction argument on the fake, `simulateNetworkOutage: true`. You didn't set up an expectation, didn't teach the double how to respond, didn't configure anything: it's a flag chapter 15 had already left ready.

The third case is this chapter's argument in miniature. The default fake fails at the lookup and never gets to save, so the SAVE error branch would go untested. To cover it, extend the fake and override one method. `FakeThatDoesNotSave` has a single line of body. With a mock, the same scenario would cost one more chained expectation, and one more expectation is one more assertion about the orchestrator's insides: one more spot that can break on the next refactor.

About `blocTest`: it's a convenience of the Dart ecosystem, not a requirement of the architecture. What matters is that the assertion is about the OBSERVABLE SEQUENCE OF STATES, and the TypeScript version right above does the same thing with an array and a `toEqual`. If you switch libraries tomorrow, these three tests stay valid, because what they assert is what comes out the door.

Try it: open <https://focus.kodel.com.br/en/dart/17-03> (or your language's route) and comment out the `markAsPaid` line in the handler. Which of the three tests turns red?

View and integration: where each one pays its own cost

Two ends are still missing. Chapter 12's View forbids "business rules and data access," and that ban answers the question on its own: there's nothing to fake in a dumb View, because it has no collaborator. What's left is rendering. If the screen just draws the state it received, field by field, a widget test would only confirm that Flutter knows how to draw text, and that isn't your problem. Don't write a single test. I'm explicitly authorizing the blank page here, because the alternative is people writing widget tests out of guilt or a sense of completeness that brings no value at all.

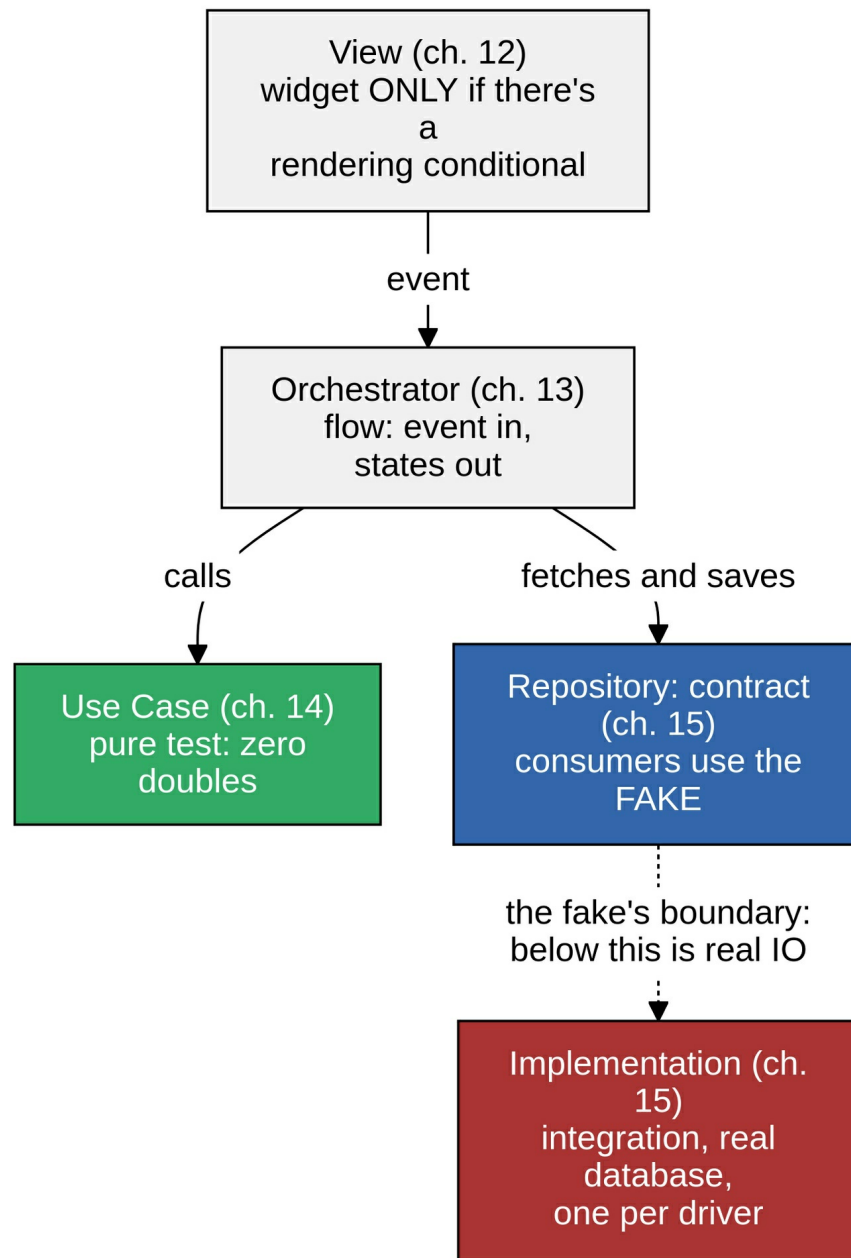
The trigger is a rendering conditional. The moment the screen chooses between two drawings, it earned a behavior of its own, and a behavior of its own deserves proof. Rosie's Coffee Shop's pay button is the case: it shows up enabled or disabled depending on `canPay`, and a widget test that mounts the screen with `canPay: false` and looks for the disabled button catches the inverted boolean, the most common defect there is.

On the other end, the real repository. It's the only layer that talks to a database, and that's why it's the only one where the fake isn't enough. The justification is concrete and singular: no fake ever catches a broken migration. You can have a hundred green tests against `TabRepositoryFake` and production still falls over because the `paid_at` column changed type in the database. The integration test runs against a real database, one per driver, and it exists to catch exactly what the fake can't know: whether the SQL is right, whether the migration ran, whether the mapping matches. There are few of them, and they're expensive. Few, because each one boots infrastructure; expensive, because each one takes time. One per driver is enough, because what you're testing is the translation, and it's the same for every query on that driver. This test can also be replaced by a SQL test, run with a tool or with a script executed by hand, to confirm the database really is what

the application expects. On PostgreSQL, the typical tool is **pgTAP**, a unit-testing framework written in SQL that runs inside the database itself.

Rosie's Coffee Shop, operation by operation:

Operation	Strategy	Why
Apply discount	pure test	pure function: input by hand, zero doubles
Flag item out of stock	pure test	the rule decides, and never touches IO
Pay the tab	flow with a fake	the sequence of states matters
Add item	flow with a fake	the orchestrator only connects the ends
Pay button	widget test	<code>canPay</code> chooses between two drawings
List the menu	no test	no conditional, nothing to fail
Save a paid tab	integration	a fake catches no migration or bad SQL
Look up a tab	integration	checks the column-to-field mapping



The dashed line is the fake's boundary. Everything above it runs in memory, in milliseconds, without booting anything. Below it lives the cost, and it stays confined to a single layer because chapter 15 put the `try/catch` in one place, and one place only.

The two critiques

The first critique is the most serious, and the anti-solution already proved it: mocks couple the test to the implementation. You watched a suite turn red without the program changing behavior. Shai Yallin, in “Fake, Don’t Mock” (2023), argues that a double checking interaction turns the test into a copy of the code, and Martin Fowler, in “Mocks Aren’t Stubs” (2007), named the two schools behind that split. The **classicist** tests by state: real objects or fakes stand in, and the result gets inspected. The **mockist** tests by interaction: collaborators get replaced, and calls get checked. FOCUS sides with the first, and not out of taste: with rules living in pure functions and IO sitting behind a small contract, the classical school comes cheap, and the mockist one gets expensive for nothing.

That doesn’t ban mocks. I use a mock when the dependency has no way of getting a fake that matches the real thing, typically a third-party SDK whose behavior I don’t control and can’t reproduce without guessing. Outside that, I write the fake and sleep better. If you disagree, the test is empirical: refactor the inside of one of your own services without changing behavior, and count how many tests break.

The second critique is the dispute between Mike Cohn’s **testing pyramid** (*Succeeding with Agile*, 2009), which calls for many unit tests at the base, integration in the middle, and few end-to-end tests at the top, and Kent C. Dodds’s **Testing Trophy** (2018), whose motto is “Write tests. Not too many. Mostly integration.” and which shifts the weight to the middle. Which one is right?

Neither, and the question is broken. Dodds is right about the base he saw: in an architecture where the business rule lives scattered across controllers and services stuffed with dependencies, a unit test only exists behind a wall of mocks, and a wall of mocks is

fragile and proves nothing. His answer was to move up a level. FOCUS's answer was a different one: move the rule into a pure function, which is chapter 14 in full. When the rule lives in a pure function, the base of the pyramid gets cheap again, genuinely cheap, because `applyLoyaltyDiscount` needs no double at all.

The shape of your suite is a consequence of your architecture, not a choice you make before you start coding. If your base is expensive, the pyramid isn't the problem.

Pitfalls

The fake that lies. It's this chapter's main pitfall, and it's silent. Your fake returns `TabFound` for any table; the real repository returns `InfraFailure` when the table doesn't exist. The tests stay green and production fails, and the worst part is the suite stays green the whole time the bug is happening.

The way out is a three-step discipline, and not one of the steps is writing more tests. First: the fake and the real one implement the SAME interface, and you never add a method to the fake that the contract doesn't have. If chapter 15 declared three methods, the fake has three. Second: whenever the real one gains a new observable behavior, such as a new failure or a new Result variant, the fake gains the matching one in the same commit. The sealed `Failure` family helps here: adding a variant breaks every non-exhaustive `switch` that's missing a `default`, and the compiler points you straight at the fake. Third: whenever doubt about divergence shows up, it's a question about the real implementation, and the integration test is what answers it. A fake that lies is a fake that aged alone.

Chasing 100% coverage. Coverage measures lines executed, not claims made. A suite that runs every line and asserts nothing scores 100% and catches no defect at all. This chapter doesn't promise full coverage, and it names what it deliberately skips: the View without a conditional, the formatter that only formats cents, and the real repository, left for the integration test.

Testing the View without a conditional. If you write a widget test for a screen that only draws the state it received, you'll be testing the framework, and you'll pay for it every time you change a padding.

Q&A

What if the refactor had changed behavior? Then the mockist suite would be right to break, and I'd have no argument at all. That's exactly why the preservation gets demonstrated first, with two empty diffs, before any mention of the break. A test that breaks when behavior changes is a good test. The mockist's problem is breaking when behavior does NOT change.

What does this chapter deliberately not test? The View without a rendering conditional, for having no behavior of its own. The real repository, which calls for integration and stays out of scope here. And the end-to-end path, the one that boots the whole app: it exists, it's expensive, and one per critical flow is enough.

If mocks are so bad, why does the library exist? Because it solves the case where you don't control the dependency and can't build a fake that matches the real thing. Mock is a tool of last resort, not first choice. The question I ask before using one is: could I write a fake faithful to this? When the answer is yes, the fake wins.

Do I need a fake per repository implementation? No. The fake belongs to the CONTRACT, not the implementation. One contract, one fake, and as many real repositories as the app needs.

Quick tip

Open your current project's suite and search for `verify`, `toHaveBeenCalled`, `assert_called_with`, or your tool's equivalent. Each hit is a claim about the INSIDE of something. Don't delete anything yet: just count, and compare that count against the number of assertions on return values. The ratio between the two numbers is how much your suite is going to hurt on the next refactor.

Quick reference

Layer	Test strategy	Chapter
View	widget test only where there's a conditional	12
Orchestrator	flow test with a fake repository	13 and 15
Use Case	pure test, no double at all	14
Repository	fake for consumers, integration for the real one	15

Each row is born from a ban in chapter 10's table. The View forbids "business rules and data access," so there's nothing to fake in it. The Orchestrator forbids "deciding rules and

persisting,” so its only collaborator is the repository, which already has a fake. The Use Case forbids “IO, framework, and domain exceptions,” so there’s no dependency to fake. The Repository does “CRUD (fetch and save), the only place an infra exception exists and becomes a Result,” and forbids “business rules”: it’s the only boundary that needs a fake, and the only one that pays for integration.

Exercises

1. Rosie decided to give a discount to customers celebrating a birthday. Add the case to the use case test at <https://focus.kodel.com.br/en/dart/17-02> (or your language’s route): build a tab, pass the date, and assert the variant that comes out. The completion criterion is the diff: it must contain only the use case’s test file, and you can’t create, touch, or configure a single test double. If you needed one, the rule leaked outside the pure function.
2. Open <https://focus.kodel.com.br/en/dart/17-03> (or your language’s route) and simulate a network outage in the happy path’s flow test: assert `Loading` followed by `Failed`. The criterion is the count: exactly one construction argument on the fake changes. Can you write a third case, the one where the tab is found and doesn’t save, without touching the `TabRepositoryFake` chapter 15 published?

Tip 17

Test what the piece promises, not how it delivers. A mock checks the how, and the how changes.

Next chapter: the architecture is complete and tested in one language, and chapter 18 opens Part IV by rebuilding the same slice of Rosie's Coffee Shop in all ten, so you can find out what in FOCUS is an idea and what was just Dart's accent.

What to Do When the Language Doesn't Help

In this chapter, you'll:

- locate FOCUS's three anchors (business rule, exception that becomes a Result, repository injection) in any of the ten implementations of chapter 10's slice;
- port the slice to an 11th language using the five-column equivalence table, without rereading chapters 10 through 17;
- say what YOUR language doesn't enforce for the team and which discipline foots that bill, with the cost measured in lines, not in opinion.

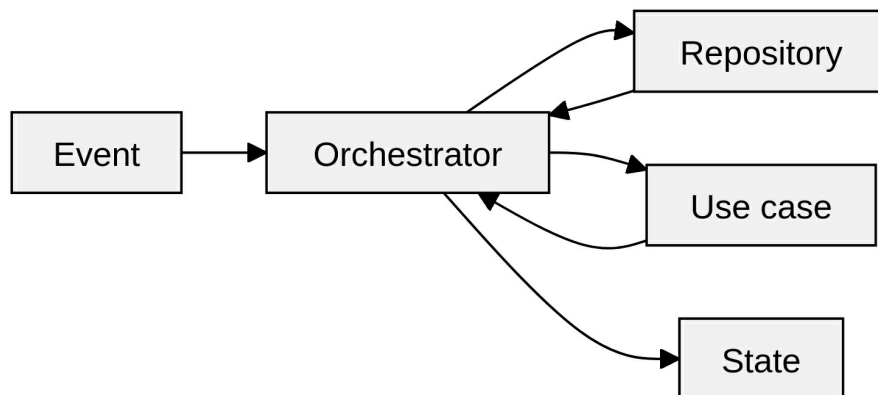
You finished Part III with the whole coffee shop slice working in one language. Now comes the question that decides whether FOCUS is an architecture or an accent: what survives when you switch languages? This chapter shows that the pieces and the arrows survive intact, that three language features change names without changing jobs, and that where a feature doesn't exist, a discipline with a known price exists instead.

The discount use case has existed since chapter 14: a tab with items, a customer with points, 10% off above 100 points, an out-of-stock item cancels the order. The full slice around that use case (View, orchestrator, repository, and composition root) was published in chapter 10, in the book's ten official languages, and

it's still available at <https://focus.kodel.com.br/en/dart/10-01> (swap `dart` for your language: `ts`, `java`, `csharp`, `go`, `php`, `python`, `kotlin`, `swift`, `rust`). This chapter doesn't reprint any of that. It does the work that's missing: hand you the map that turns ten similar- looking files into ONE drawing with ten spellings.

The slice on the board

Before the spellings, the drawing. Here it is, and it's the same across all ten of chapter 10's implementations, comment bytes aside:



The event enters through the orchestrator (chapter 13). The orchestrator fetches data from the repository (chapter 15), hands everything ready-made to the use case (chapter 14), saves whatever held up, and publishes the new state. The View draws the state and nothing else (chapter 12). Open any file from the `/en/<lang>/10-01` route, and that's the flow sitting there, in the same order, with the same business names. What changes from language to language fits in three spots, and this chapter calls those spots anchors: where the business rule lives, where the exception becomes a Result, and where the repository gets injected. Find those three, and you've found the architecture; the rest is syntax around it.

Types first: four families and one warning

The piece most sensitive to language choice isn't the orchestrator or the repository: it's the pair of types chapter 8 called errors as values. The use case's Result (`DiscountApplied` | `NotEligibleForDiscount` | `ItemOutOfStock`) and the tab's state (`Loading` | `Ready` | `Failed` , from chapter 13) need two guarantees: the list of variants is closed, and whoever consumes the list is forced to handle all of them. Seven of the ten languages write that closed list, and they group into four spelling families. In five of them (Kotlin, Dart, Swift, Rust, and Java) both guarantees come from the compiler, and forgetting a variant is a build error. TypeScript and Python close the list the same way, but coverage is only checked if an external type checker runs, and that difference decides who pays the bill when someone forgets a case. The other three, Go, C#, and PHP, don't close the list with any proof at all. PHP gets its own section right below; Go and C# wait for "When the language doesn't help," alongside Python, because in those three the guarantee turns into team discipline.

Kotlin

```
// The use case's Result: a sealed class, three variants, each  
  
// refusal with its reason typed.  
  
sealed class DiscountResult  
  
data class DiscountApplied(val tab: Tab) : DiscountResult()
```

```
data class NotEligibleForDiscount(

    val points: Int,

    val pointsNeeded: Int,

) : DiscountResult()


data class ItemOutOfStock(val name: String) : DiscountResult()


// The exhaustive when: no `else` branch, and if a new variant is
// born, this file STOPS compiling until you decide what it shows.


fun describe(result: DiscountResult): String =

    when (result) {

        is DiscountApplied ->

            "Total with discount: ${result.tab.totalInCents} cents"

        is NotEligibleForDiscount ->

            "Missing ${result.pointsNeeded - result.points} points"
```

```
is ItemOutOfStock -> "${result.name} is out of stock"

}
```

The **sealed class** family: the word `sealed` closes the list of subclasses in the same file (Kotlin) or the same library (Dart), and that closure is what lets the compiler prove the `when` above covered everything. Notice what's NOT there: no `else`. The `else` is exactly the hole a new variant would slip through unnoticed. Dart writes the same drawing with `sealed class` and a `switch` expression with destructuring patterns: `DiscountApplied(:final tab) => ...`. The spelling gap has a historical reason: Kotlin has had `sealed` since 2016 and `when` was always an expression; Dart only got `sealed` and `switch` as an expression in Dart 3 (2023), and used the chance to bring object patterns along, so the Dart version deconstructs right in the arm instead of doing a smart cast.

Try it: open <https://focus.kodel.com.br/en/kotlin/18-01> (or <https://focus.kodel.com.br/en/dart/18-01>) and delete the `ItemOutOfStock` arm from the `when`. The compiler points at the missing arm before you even run it. Now add an `OnTheHouse` variant to the sealed class: EVERY switch in the program breaks at once, and that cascading break is what you want.

Swift

```
// The use case's Result: an enum, each variant carries ITS OWN data.

enum DiscountResult {

    case discountApplied(Tab)
```



```
    case notEligibleForDiscount(points: Int, pointsNeeded: Int)

    case itemOutOfStock(name: String)

}

// The exhaustive switch: no `default`, and if a new variant is born,
// this file STOPS compiling until you decide what it shows.

func describe(_ result: DiscountResult) -> String {

    switch result {

    case .discountApplied(let tab):

        return "Total with discount: \$(tab.totalInCents) cents"

    case .notEligibleForDiscount(let points, let pointsNeeded):

        return "Missing \$(pointsNeeded - points) points"

    case .itemOutOfStock(let name):

        return "\$(name) is out of stock"

    }

}
```

The **algebraic enum** family: instead of a base class with subclasses, a single `enum` whose variants carry associated values. It's the same concept with a different genealogy: Swift and Rust inherited the design of algebraic data types from ML and Haskell, where the sum of variants is a type in itself, not a hierarchy. In practice the difference shows up in weight: the sealed-class version asks for one declaration per variant, the enum declares all three in three lines. Rust writes `enum DiscountResult` with variants `DiscountApplied(Tab)` and `NotEligibleForDiscount { points: u32, points_needed: u32 }`, and `match` plays the role of `switch` with the same proof of coverage, no `_` arm. Rust's `_` and Swift's `default` are the same hole `else` was in the previous family. It exists, sure. The team's agreement is to use neither of them on top of a `Result` or a state.

Try it: open <https://focus.kodel.com.br/en/swift/18-02> (or <https://focus.kodel.com.br/en/rust/18-02>) and swap the order of the `case` s. Nothing changes: exhaustiveness isn't order, it's coverage. Then delete `case .itemOutOfStock` and watch the error point at the variant by name.

TS TypeScript

```
// The use case's Result: a union discriminated by the `kind` field.

type DiscountResult =

  | { readonly kind: "discountApplied"; readonly tab: Tab }

  | {
```

```

        readonly kind: "notEligibleForDiscount";

        readonly points: number;

        readonly pointsNeeded: number;

    }

    | { readonly kind: "itemOutOfStock"; readonly name: string };

// The exhaustiveness bodyguard, and it's optional: it only exists if
// you write it. The `never` is only checked with the type checker on.

function assertNever(value: never): never {

    throw new Error(`Unhandled case: ${JSON.stringify(value)}`);

}

// The switch with assertNever in the default: forget a case and the
// TYPE CHECKER complains; without it, the miss only blows up at
// runtime.

function describeResult(result: DiscountResult): string {

```

```

switch (result.kind) {

  case "discountApplied":

    return `Total with discount: ${result.tab.totalInCents} cents`;

  case "notEligibleForDiscount":

    return `Missing ${result.pointsNeeded - result.points} points`;

  case "itemOutOfStock":

    return `${result.name} is out of stock`;

  default:

    return assertNever(result);

}

}

```

The **discriminated union** family: the closed type is a union of object shapes, and the `kind` field is the tag TypeScript's narrowing uses to know, inside each `case`, which shape is in hand. Here sits a distinction the two previous families don't ask for. TypeScript **ALLOWS** exhaustiveness, but doesn't **REQUIRE** it. A `switch` without the `default: assertNever(...)` compiles just the same, and the forgotten case quietly returns `undefined`. `assertNever` is optional: you write the function, you remember to use it in every

switch, and `tsc` only complains if it's running. It's real protection, but protection the team installs, not one the language ships out of the box. Python belongs in this family for the same reason: a `frozen dataclass` per variant, a union `DiscountApplied | NotEligibleForDiscount | ItemOutOfStock`, and `typing.assert_never` in the `match`'s `case _:`. Python's protection depends on mypy or pyright running in CI, and the Python workaround section, further down, shows what happens without them.

Try it: open <https://focus.kodel.com.br/en/ts/18-03> (or <https://focus.kodel.com.br/en/python/18-03>), delete a `case`, and run the type checker. Then delete the `assertNever` from the `default` too and run it again: the error is gone, and the forgotten case turned into `undefined` on the screen.

Java

```
// The use case's Result: a sealed interface plus one record per
// variant. The `permits` is the closed list the compiler now watches.

sealed interface DiscountResult

    permits DiscountApplied, NotEligibleForDiscount, ItemOutOfStock {}

record DiscountApplied(Tab tab)

    implements DiscountResult {}
```

```
record NotEligibleForDiscount(int points, int pointsNeeded)

    implements DiscountResult {}

record ItemOutOfStock(String name) implements DiscountResult {}

// The switch with patterns (Java 21): no `default`, and if a new
// record enters `permits`, this file STOPS compiling.

static String describe(DiscountResult result) {

    return switch (result) {

        case DiscountApplied(Tab tab) ->

            "Total with discount: "

                + tab.totalInCents() + " cents";

        case NotEligibleForDiscount(int points, int pointsNeeded) ->

            "Missing " + (pointsNeeded - points) + " points";

        case ItemOutOfStock(String name) -> name + " is out of stock";

    };
}
```

```
}
```

Java earns its own label because it's the rare case of a language that arrived LATE to the party and brought the whole package: `sealed interface` (Java 17, 2021), `record` (Java 16), and `switch` with record patterns and proof of exhaustiveness (Java 21, 2023). The explicit `permits` is the house style: where Kotlin infers the closed list from the file, Java asks for the list in writing, in the style of someone who's already been burned by 25 years of open hierarchies. If your production Java is still 11 or 17 without preview features, you don't have the exhaustive switch, and in that case the C# recipe coming up (a hand-rolled `Match` method) works exactly the same way in Java.

Try it: open <https://focus.kodel.com.br/en/java/18-04>, add a record `OnTheHouse` to `permits`, and recompile without touching the switch. The error that shows up names the missing pattern by name.

PHP

```
// PHP's match: it looks like Kotlin's when, but NOTHING here is

// checked at compile time. A forgotten `instanceof` only blows up at

// runtime.

function describeResult(DiscountResult $result): string

{
```

```

return match (true) {

    $result instanceof DiscountApplied =>

        "Total with discount: {$result->tab->totalInCents} cents",

    $result instanceof NotEligibleForDiscount =>

        'Missing ' . ($result->pointsNeeded - $result->points)

        . ' points',

    $result instanceof ItemOutOfStock =>

        "{$result->name} is out of stock",

};

}

```

PHP falls outside the four families for a reason its own `match` documentation states plainly: when no arm matches, `match` throws `\UnhandledMatchError` at RUNTIME (php.net, “match”). The spelling lies to you. PHP 8’s `match` looks like Kotlin’s `when`, returns a value, and compares with `===`, but the guarantee didn’t come along for the ride. PHP 8.1’s `enum` and 8.2’s `readonly` classes give you immutability and a list of cases, and even then no native tool proves coverage before deploy. Don’t treat this as a detail. It’s the difference between the customer at table 13 seeing “connection error” and seeing a fatal error’s blank white screen. The discipline that compensates is the same as Python’s: PHPStan or

Psalm in CI, at the level that flags a non-exhaustive `match`, and no merge with the analysis turned off. The full PHP slice lives at <https://focus.kodel.com.br/en/php/10-01>, and it carries this fragility in its body: it's the largest of the ten implementations, and the numbers section further down shows what that costs.

The three anchors, family by family

With the types in place, the rest of the translation is finding three lines. It doesn't matter which language: chapter 10's slice has ONE use case signature that takes ready data and returns a Result (anchor 1, chapter 14), ONE point where an infrastructure failure becomes a value (anchor 2, chapter 15), and ONE spot where the concrete repository gets built and injected (anchor 3, chapter 9). Below, the minimal excerpt from each family with all three anchors together, pulled literally from the files published at the `/en/<lang>/10-01` route. The type names are chapter 10's own (`TabResult`, with variants `TabUpdated`, `InvalidItem`, and `InfraFailure`): that chapter's gesture was adding an item, and each operation's Result belongs to that operation, as chapter 14 established.

 Kotlin ·  Dart

```
// Anchor 1: the use case's signature, data goes in, Result comes out.
```

```
typealias AddItemToTab =
```

```
(tab: Tab, item: Item, loyaltyPoints: Int) ->
```

```
TabResult
```

```
// Anchor 2: the repository's boundary, exceptions die here.

    return try {

        val saved = saveToServer(tab)

        tabs[saved.table] = saved

        TabUpdated(saved)

    } catch (error: IllegalStateException) {

        InfraFailure(error.message ?: "unknown failure")

    }

// Anchor 3: the composition root, the concretes are born here, and
// only here.

val repository = FakeTabRepository()

val orchestrator = TabOrchestrator(

    repository,
```

```
::addItemToTab,  
  
)
```

In the sealed family, the use case is a function `typealias`: the rule is a pure function and the orchestrator receives the function, not a class. Anchor 2's `catch` is the ONLY try-catch in the entire file, and it returns a Result variant instead of rethrowing. Anchor 3 lives in `main`: no other line in the file writes `FakeTabRepository()`.

Rust

```
// Anchor 1: the use case's signature, data goes in, Result comes out.  
  
type AddItemToTab = fn(&Tab, &Item, u32) -> TabResult;  
  
// Anchor 2: the repository's boundary. Rust has no exceptions; the  
  
// failure is BORN a value, and Err plays the role the catch did.  
  
fn save(&mut self, tab: Tab) -> Result<Tab, String> {  
  
    if tab.table == 13 {  
  
        return Err("no connection to the server".to_string());  
  
    }  
}
```

```
// Anchor 3: the composition root, the concretes are born here, and

// only here.

let mut orchestrator = TabOrchestrator {

    repository: FakeTabRepository { tabs },

    add_item: add_item_to_tab,

    state: Box::new(|state| println!("{}", tab_screen(&state))),

};
```

In the enum family, anchor 2 changes face without changing job: Rust has no exception to translate, so the boundary returns `Result<Tab, String>` directly, and `Err` is the same line `catch` was in Kotlin. Swift sits halfway: it has `throws` in the infrastructure, and the boundary uses `do/catch` to return `.infraFailure`, as you can check at <https://focus.kodel.com.br/en/swift/10-01>. Rust's anchor 3 builds the orchestrator's struct with named fields; dependency injection without a framework is just that, a struct literal in `main`.

TS TypeScript

```
// Anchor 1: the use case's signature, data goes in, Result comes out.

type AddItemToTab = (

    tab: Tab,
```

```
    item: Item,

    loyaltyPoints: number,

) => TabResult;

// Anchor 2: the repository's boundary, exceptions die here.

try {

    const saved = await this.saveToServer(tab);

    this.tabs.set(saved.table, saved);

    return { type: "tabUpdated", tab: saved };

} catch (error) {

    return { type: "infraFailure", reason: `${error}` };

}

// Anchor 3: the composition root, the concretes are born here, and
```

```
// only here.

const repository = new FakeTabRepository();

const orchestrator = new TabOrchestrator(

    repository,

    addItemToTab,

);
```

In the union family, the Result that comes out of `catch` is an object with a `type` field, because the discriminated union is the local shape of a sealed class. The `await` in anchor 2 recalls a point the other families hide: in TypeScript the repository is asynchronous by nature, and the boundary translates both the exception and the rejected promise. On the same route's Python version, anchor 2 is an `except` that returns the failure dataclass, and anchor 3 builds the orchestrator in `main` with the fake repository passed positionally.

Java

```
// Anchor 1: the use case's signature, data goes in, Result comes out.

@FunctionalInterface

interface AddItemToTab {

    TabResult apply(
```

```
    Tab tab, Item item, int loyaltyPoints);

}

// Anchor 2: the repository's boundary, exceptions die here.

try {

    Tab saved = saveToServer(tab);

    tabs.put(saved.table(), saved);

    return new TabUpdated(saved);

} catch (Exception error) {

    return new InfraFailure(error.getMessage());

}

// Anchor 3: the composition root, the concretes are born here, and

// only here.
```

```
TabRepository repository = new FakeTabRepository();

TabOrchestrator orchestrator =

    new TabOrchestrator(repository, UseCases::addItemToTab);
```

Java swaps the `typealias` for a `@FunctionalInterface`, which is the platform's way of naming a function type, and injects the rule with a method reference (`UseCases::addItemToTab`). Other than that, the three anchors are the same, in the same spots. That's this whole chapter's practical test: open the `/en/<lang>/10-01` route of a language you DON'T know and look for the three anchors. If you found the signature that returns a `Result`, the only try-catch (or the only `Err`), and the single spot that builds the concrete repository, you just read a FOCUS implementation in a language you never studied.

Try it: randomly pick one of chapter 10's ten routes, in a language you don't use day to day, and time how long it takes you to mark the three anchors. My guess: less time than it took you to find the discount rule in chapter 2's monolith.

When the language doesn't help

Five languages deliver the proof of coverage in the compiler. The other five don't, and they're among the most used in the world. PHP already got its section, alongside the types; that leaves Go, C#, and Python. Pretending that gap doesn't exist would treat those languages' readers as second-class citizens, so the three

sections below do the opposite: they show the naive code the gap allows, the concrete pain it causes, and the workaround with the price on the label.

Go: no union, no exhaustiveness, and a native Result

Go rejected unions and exhaustiveness by design decision, not out of lag: the language prefers a single mechanism (values) over a type system that proves coverage. The anti-solution shows up when you try to write the tab's state as if Go had a real enum:



Go

```
// ANTI-SOLUTION: a string "enum" and a switch the compiler doesn't
// watch. Forget the "failed" case and this file compiles without a
// peep.

func describeStateNaively(state string, tab Tab) string {

    switch state {

    case "loading":

        return "Loading..."

    case "ready":

        return fmt.Sprintf("Table %d ready", tab.Table)
```

```

    }

    // The "failed" case doesn't exist and nobody warned you: whoever

    // lands here takes an empty string to the screen.

    return ""

}

```

The pain: the `"failed"` case has no arm, the compiler says nothing, and Rosie's screen shows an empty string at the exact moment it should show "no connection to the server." No happy-path test catches this. The workaround has two parts. For the use case's Result, Go already has the right idiom out of the box, and Rob Pike named it on the official blog: "errors are values" (go.dev/blog/errors-are-values, 2015). Chapter 14's rule returns `(Tab, error)` with errors named per variant:

Go

```

// WORKAROUND, part 1: the use case's refusals become named errors,

// values the caller is forced to look at (or ignore, in writing).

var (

    ErrNotEligibleForDiscount = errors.New("fewer than 100 points")

    ErrItemOutOfStock        = errors.New("item out of stock")

```

```

)

// WORKAROUND, part 2: chapter 14's rule returns (T, error), Go's
// native Result. Above 100 points, 10% off; an out-of-stock item
// cancels the order.

func applyLoyaltyDiscount(
    tab Tab,

    loyaltyPoints int,
) (Tab, error) {

    for _, item := range tab.Items {

        if item.OutOfStock {

            return Tab{}, fmt.Errorf(

                "%w: %s", ErrItemOutOfStock, item.Name,

            )

        }

    }

}

```

```
if loyaltyPoints < 100 {

    return Tab{}, fmt.Errorf(

        "%w: customer has %d", ErrNotEligibleForDiscount,

        loyaltyPoints,

    )

}

total := tab.TotalInCents

discounted := total - total*10/100

return Tab{

    Table:      tab.Table,

    Items:      tab.Items,

    TotalInCents: discounted,

}, nil

}
```

For the tab's state, which needs variants with different data, the recipe is a struct closed BY CONVENTION: private fields, one named constructor per case (`Loading()` , `Ready(tab)` , `Failed(message)`), and the team-agreed rule that nobody builds the struct by hand. The price sits in that word, agreed. Go's compiler doesn't stop a zeroed `TabState{}` , and it doesn't charge for a new case in `renderState` ; whoever charges for it is code review, every time, forever. I'd rather pay that price than fake a hierarchy with empty interfaces and type switches scattered around, because the cost stays concentrated in two auditable spots: the state file and the team's review checklist.

Try it: open <https://focus.kodel.com.br/en/go/18-05> and run it. The output's first line is the anti-solution returning `""` for the forgotten state. Then add a `Cancelled()` constructor to the state and notice that NOTHING breaks: that's the difference from the four families, and the reminder to update `renderState` is yours, not the compiler's.

C#: exhaustiveness that's just a warning

C# has records, pattern matching, and a `switch` expression, and it still falls outside the four families over a detail many teams only find out about in production: switch coverage isn't an error, it's warning CS8509. The anti-solution compiles:



```
// ANTI-SOLUTION: switch expression missing the Failed arm. The  
  
// compiler emits warning CS8509 and moves on; the build passes.
```

```
static string DescribeNaively(TabState state) =>

    state switch

    {

        Loading => "Loading...",

        Ready ready => $"Table {ready.Tab.Table} ready",

    };
```

The pain arrives in two stages. At build time, Microsoft politely warns: “the switch expression does not handle all possible values of its input type” (warning CS8509, documented at learn.microsoft.com). Polite warnings drown among 200 others in a large project. Then, the day a `Failed` reaches that switch, the runtime throws `SwitchExpressionException`, a forgotten-case exception exploding in front of the customer, exactly what chapter 8 told you to prevent. The workaround turns the omission into a COMPILE error using what C# has that’s strongest, the method signature:



```
// WORKAROUND: the sealed hierarchy. `abstract record` plus a private

// constructor close the list of variants in the same file.

abstract record DiscountResult
```

```
{

    private DiscountResult() { }

    public sealed record DiscountApplied(Tab Tab)

        : DiscountResult;

    public sealed record NotEligibleForDiscount(

        int Points,

        int PointsNeeded

    ) : DiscountResult;

    public sealed record ItemOutOfStock(string Name) : DiscountResult;

    // Match is the exhaustiveness the language didn't give you: one

    // delegate per variant, all of them mandatory. New variant = new

    // parameter = every call site breaks at COMPILE time.
```

```

public T Match<T>(

    Func<DiscountApplied, T> onApplied,

    Func<NotEligibleForDiscount, T> onNotEligible,

    Func<ItemOutOfStock, T> onOutOfStock

) =>

    this switch

    {

        DiscountApplied applied => onApplied(applied),

        NotEligibleForDiscount notEligible => onNotEligible(notEligible),

        ItemOutOfStock outOfStock => onOutOfStock(outOfStock),

        _ => throw new InvalidOperationException(),

    };

}

```

The private constructor keeps heirs out of other files, so the list is truly closed. `Match` takes a delegate per variant, and delegates are parameters: miss one, and the code doesn't even compile. When someone adds the `OnTheHouse` variant, `Match` gains a fourth

parameter and EVERY call site breaks together, which is the four families' behavior, rebuilt by hand. If you'd rather not write that boilerplate, the OneOf library (github.com/mcintyre321/OneOf) delivers the same `Match` through metaprogramming; the choice between the two is taste, the design is the same. The workaround's cost: one `Match` method per closed type, maintained by whoever maintains the type, and a team rule that a direct `switch` on top of these hierarchies either handles ALL the cases or doesn't pass review. Promoting CS8509 to an error in the `.csproj` (`WarningsAsErrors`) closes the loop from above. I go further than that: I turn on every *warning as error* in every project of mine, from the first commit. A warning nobody reads isn't protection.

Try it: open <https://focus.kodel.com.br/en/csharp/18-06>, compile it, and watch warning CS8509 point at `DescribeNaively`. Then delete the `onOutOfStock:` argument from any `Match` call and compare: the warning turned into an error, and that promotion is what you bought.

Python: the habit the Result needs to unlearn

Python has everything the union family uses: `frozen dataclass`, `union`, `match`, and `typing.assert_never` (documented in the official `typing` module docs, Python 3.11+). Python's problem isn't a missing feature, it's a habit with its own name: EAFP (Easier to Ask Forgiveness than Permission), the style enshrined in the language's own official glossary, which says to try first and catch the exception after. For IO, EAFP works well. For business rules, it produces this:

 Python

```
# ANTI-SOLUTION: a business refusal as an exception, the EAFP way. The
```

```
# signature promises a Tab and hides the two refusals; the distracted
```

```
# caller only discovers ItemOutOfStockError in production.
```

```
class NotEligibleForDiscountError(Exception):
```

```
    pass
```

```
class ItemOutOfStockError(Exception):
```

```
    pass
```

```
def apply_discount_naively(
```

```
    tab: Tab,
```

```
    loyalty_points: int,
```

```
) -> Tab:
```

```
    for item in tab.items:
```

```
        if item.out_of_stock:
```

```

        raise ItemOutOfStockError(item.name)

    if loyalty_points < 100:

        raise NotEligibleForDiscountError(loyalty_points)

    total = tab.total_in_cents

    return replace(tab, total_in_cents=total - total * 10 // 100)

```

The pain: the signature says `-> Tab` and it's lying, because the function has three exits and two of them travel over a channel no type checker tracks. The caller who forgets the `try` gets no tooling warning at all; `ItemOutOfStockError` crosses the orchestrator, crosses the View, and blows up in the production log, far from the rule that raised it. The workaround is the union family's spelling, which you already saw in the types section: a frozen dataclass per variant, a union as the Result, and `assert_never` closing the `match`:

Python

```

# WORKAROUND, part 3: match + assert_never. Delete an arm and mypy

# flags it; WITHOUT a type checker, Python runs this file the same

# way, and the missing arm only blows up when that case reaches

# production.

```

```

def describe(result: DiscountResult) -> str:

    match result:

        case DiscountApplied(tab):

            return f"Total with discount: {tab.total_in_cents} cents"

        case NotEligibleForDiscount(points, points_needed):

            return f"Missing {points_needed - points} points"

        case ItemOutOfStock(name):

            return f"{name} is out of stock"

        case _:

            assert_never(result)

```

Before you test this and think the recipe is broken: running this file with one arm short does NOT throw anything in the interpreter, as long as the forgotten case never shows up in the data. `assert_never` isn't runtime magic; it's a function that only makes sense to the type checker. All the protection lives in `mypy --strict` (or `pyright`) running in CI as a merge gate, on the same step as the tests. Without that step, the workaround is decorative. The cost, then, is infrastructure and culture: the team accepts that domain Python code without a type checker is code with no

coverage check, and that a business refusal comes back as a value, while EAFP stays where it belongs, the IO boundary from chapter 15.

Try it: open <https://focus.kodel.com.br/en/python/18-07> and run it twice: `python3 18-07.py` exits clean even if you delete the `ItemOutOfStock` arm; `mypy --strict 18-07.py` flags the deleted arm on the `assert_never` line. The gap between the two runs is the exact size of your dependency on CI.

The critiques, with a ruler instead of rhetoric

Two objections come up every time FOCUS meets a language with no syntactic sugar, and both deserve a numbered answer. The measurement criterion, fixed before looking at any result: non-empty lines across the ten published files of chapter 10's slice, comments included, counted with `grep -cve '^\s*$'`.

The first objection is “Result hell”: returning a Result instead of throwing an exception would drown the code in unwrapping. The criticism has pedigree; Scott Wlaschin himself, author of Railway-Oriented Programming (fsharpforfunandprofit.com/rop/, 2013), warns on that same page not to take the pattern to extremes (“don’t take it to extremes,” in his words). The ruler says this about the coffee shop’s slice: the shortest implementation has 151 non-empty lines (Kotlin and Python tied) and the longest has 237 (PHP). The 86-line gap between the extremes, 57% more in PHP than in Kotlin, has a cause you can name line by line. Kotlin declares a Result variant in one line (`data class InvalidItem(...) : TabResult()`). PHP pays for constructors, assignments, and braces for every `readonly` class, plus convenience getters that `data class` throws in

for free. The Result isn't the villain here. All ten versions use Result the same number of times, and the gap between 151 and 237 comes from how much boilerplate each language charges for DECLARING types, not for using them. Rob Pike defends the same design with no sugar at all: "errors are values" (go.dev/blog/errors-are-values, 2015), and Go's slice lands at 162 lines, below Java, C#, and TypeScript.

The second objection says the unidirectional flow is bureaucracy: event, orchestrator, state, all that just to call a function. This criticism also has a respectable source. Redux's documentation lists boilerplate as the community's number-one complaint (redux.js.org, "Prior Art"), and the Elm guide, which inspired Redux according to that same page, answers that the repetitive structure is what makes the program predictable (guide.elm-lang.org/architecture/). The coffee shop's ruler gives the bureaucracy's size: the WHOLE slice, View, orchestrator, use case, repository, fake, and composition root, fits in 151 to 237 non-empty lines depending on the language. The unidirectional flow itself costs almost the same in all of them: what separates the extremes is the type-declaration section, as the previous paragraph measured. Real bureaucracy is what chapter 2 showed: one discount line scattered across three screens.

One measurement in this section surprised me, and it's worth recording against myself. I expected Rust, with `?`, `match`, and `derives`, to produce a visibly smaller slice than Java's. The grep says otherwise: Rust has 172 non-empty lines, Java has 165. The sugar exists where the rhetoric promises it (the use case's `match` is shorter than Java's `switch`), but `structs`, `impl` blocks, and closures with `Box` give the difference back in the rest of the file. The lesson is this chapter's own thesis, applied to me: per-piece expressiveness doesn't add up linearly across a file, and anyone who wants to compare languages has to run the grep, not quote reputation. Measured numbers age better than adjectives.

The equivalence table: porting to the 11th language

Your language isn't among the ten? The equivalence table is the whole chapter in five questions, split into three pieces to fit the page. Answer the five columns for your target language and you'll know what it gives you for free, what needs a library, and what needs discipline; the three anchors and the drawing on the board do the rest.

Language	sealed/union	Exhaustiveness
Dart	<code>sealed class</code>	<code>switch</code> expression, guaranteed
TypeScript	discriminated unions	<code>never</code> /assertNever (opt-in)
Java	<code>sealed interface</code> + <code>record</code>	<code>switch</code> patterns (21), guaranteed
C#	<code>abstract record</code> / <code>OneOf</code>	warning only
Go	doesn't have one; <code>(T, error)</code>	doesn't have it
PHP	<code>enum</code> + unions in signature	<code>match</code> fails at runtime
Python	<code>Union</code> + frozen dataclasses	<code>assert_never</code> (type checker)
Kotlin	<code>sealed class</code> / <code>interface</code>	<code>when</code> exhaustive, mandatory

Swift	<code>enum</code> with associated values	<code>switch</code> exhaustive, mandatory
Rust	algebraic <code>enum</code>	<code>match</code> exhaustive, mandatory

Language	Immutability	Idiomatic DI
Dart	<code>final</code> / <code>const</code> , <code>copyWith</code>	constructor; <code>get_it</code> /riverpod
TypeScript	<code>readonly</code> , <code>as const</code>	closures; NestJS DI in the enterprise
Java	<code>record</code> , <code>List.of()</code>	Spring/CDI/Guice
C#	<code>record</code> + <code>with</code> , <code>init</code>	MS.Extensions.DI, standard
Go	by value/discipline	explicit constructor
PHP	<code>readonly</code> (8.2)	PSR-11 container (Laravel/Symfony)
Python	<code>frozen dataclass</code>	constructor; FastAPI <code>Depends</code>
Kotlin	<code>val</code> , <code>data class.copy()</code>	constructor; Koin/Hilt
Swift	<code>let</code> + value semantics	initializer
Rust	immutable by default	generics/traits in the constructor

--	--

Language	Result vs exceptions
Dart	sealed Result at the boundary; exceptions in infra
TypeScript	exceptions by default; <code>neverthrow</code> / Either growing
Java	exceptions dominate; Result via sealed is growing
C#	exceptions; <code>OneOf</code> / <code>ErrorOr</code> in the domain
Go	native errors-as-values; panic = bug
PHP	exceptions dominate
Python	exceptions (EAFP); Result is a niche
Kotlin	<code>kotlin.Result</code> / Arrow; exceptions at the boundary
Swift	<code>Result</code> stdlib + typed throws
Rust	<code>Result</code> + <code>?</code> is THE idiom

The porting roadmap uses the columns in order. First, the closed sum: if the target language has a closed sum of types (Elixir with structs and pattern matching, Scala 3 with `enum`, F# with discriminated unions), you're in one of the four families and the port is spelling. Second, exhaustiveness: if the answer is "opt-in" or "doesn't have it," pick NOW which workaround section is yours, because that decision becomes a CI and review rule, not an invitation. Third, immutability: look for the equivalent of `copy` / `with` / `replace`, because chapter 14's use case returns a new tab, never edits the one it received. Fourth, DI: anchor 3 only needs a constructor; a DI framework is optional in EVERY language on the table, as chapter 9 established. Fifth, Result vs exceptions: decide where the exception dies, and FOCUS's answer

is always the same, at the repository's boundary, chapter 15. Two hours with this table and the `/en/<lang>/10-01` route open on the side beat any tutorial for the new language; that's how the slice reached all ten.

Pitfalls

The first pitfall is porting syntax instead of architecture: taking the Kotlin version and looking up “how do you write a sealed class in Go.” You don't. The right question is different: “where does Go guarantee that the use case's three outcomes are visible to the caller?” The answer (named errors in `(T, error)`) doesn't look anything like a sealed class, and it still plays exactly the same role: it keeps the rule's three outcomes in plain sight for whoever calls it. Translate word for word and you get a Frankenstein with an empty interface and a type switch; translate anchor by anchor and you get Go that looks like Go.

The second is trusting opt-in protection as if it were a guarantee. TypeScript's `assertNever`, mypy's `assert_never`, and CS8509 promoted to an error share the same fine print: someone can turn it off. If the merge passes with the type checker red, or the `.csproj` doesn't promote the warning, the coverage you think you have is the coverage some future intern removes on a Saturday. Opt-in protection only counts when CI makes it mandatory, which is why it's a question of process before it's a question of language. Process matters as much as product and people. A team with no defined process is a team where everyone does whatever they want.

The third is PHP's false friend: `match` returns a value and compares with `===`, so it looks like Kotlin's `when`, and it isn't. The difference only shows up with the arm missing and the rare case arriving, the worst possible moment. If your team is PHP, treat

`\UnhandledMatchError` in monitoring as a symptom of a forgotten arm and PHPStan in CI as the vaccine, and reread this chapter's types section before you promise exhaustiveness in code review.

The fourth is using this chapter's line count as a language ranking. The numbers measure ONE slice, in ONE formatting style, and prove only what I claimed: the flow costs about the same and the type declaration is what varies. If someone quotes "Rust is smaller than Java" or the opposite in an architecture discussion, you already know what to ask for: the grep, the criterion, and the corpus. The ruler exists to name the cause of a difference, never to crown a language.

Q&A

My language has guaranteed exhaustiveness. Can I skip the workaround sections? You can, until the day a Go backend joins the project or the Python data script turns into a service. The workaround sections are the map of your neighbors' languages, and the equivalence table works in both directions.

Why doesn't the chapter use `OneOf`, `Arrow`, `neverthrow`, and the like in the listings? Because the slice needs to show each language's REAL cost with no middleman. A library hides the price exactly where this chapter wants you to see it. In your own project, use them: `OneOf` in C# and `Arrow` in Kotlin are exactly this page's workaround, packaged and tested.

Is it worth adopting `Result` in Python code that's all EAFP? At the IO boundary, no; EAFP is the right idiom there, as chapter 15 showed with the repository's `except`. In business rules, yes, and the test is the signature: if the function has

more than one business outcome, the outcomes show up in the return type. Migrate function by function, starting with the use cases.

What if my 11th language has NONE of the five columns?

Then it's Go without the `(T, error)`, and the recipe is the oldest one in the book: documented convention, code review, and chapter 10's table taped to the wall. FOCUS degrades to pure discipline; the design stays the same.

Quick tip

Save the command `grep -cve '^\\s*$' file` (non-empty lines) in your shell. It's this chapter's ruler, and it works for any verbosity argument: instead of "I think it got big," you say "the new version has 23 more lines, and 19 are the type declarations." An argument about taste turns into an argument about cause.

Quick reference

Situation	Recipe
Reading FOCUS in a new language	find the three anchors: Result, boundary, root
Guaranteed sealed/enum/union	use the family from the types section
Any switch over a Result	ban <code>else</code> , <code>default</code> , and <code>_</code>
Go	named <code>(T, error)</code> ; struct closed by convention

C#	sealed records with <code>Match</code> , or <code>OneOf</code> ; CS8509 as an error
Python	<code>dataclass</code> + <code>union</code> + <code>assert_never</code> ; mypy on merge
PHP	<code>enum</code> + <code>match</code> with PHPStan in CI; monitor the error
Porting to the 11th	answer the five columns; translate anchor by anchor
Verbosity argument	measure with <code>grep -cve '^s*\$'</code> and name the cause

Exercises

1. Open <https://focus.kodel.com.br/en/csharp/18-06> and add the variant `OnTheHouse(string name)` to the sealed `DiscountResult`. The completion criterion is the trail of errors: note how many compilation errors `Match`'s fourth parameter generated, and confirm each one was a spot that HAD to decide what to do with the freebie. Then repeat the gesture on your family's language route and compare the trails.
2. Pick a language outside the ten (Elixir, Scala, F#, or whatever your team is dating) and fill in the equivalence table's five columns for it, with one official-documentation source per cell. Can you port chapter 14's discount use case using only your table and the three anchors, without opening any of the ten implementations?

Tip 18

Translate the architecture, not the syntax. Pieces and arrows travel between languages; spelling stays home.

Next chapter: with FOCUS standing in any language, it's time to catalog the ways to knock it down: the anti-patterns that stalk every piece, and how to spot them before the damage is done.

Anti-Patterns: How to Wreck FOCUS

In this chapter, you'll:

- identify, in a diff, which of the six anti-patterns is present, naming the symptom;
- name each one's damage in cost of change, what gets expensive six months later, not in an adjective;
- apply each card's fix, with a citation to the chapter that taught the rule it breaks.

The coffee shop's slice got built piece by piece across chapters 10 through 17: a dumb View, an orchestrator that connects, a pure use case, a repository at the boundary, and a test suite covering all of it. This chapter is the same design seen in negative: the six most common ways to tear that slice down without a single test screaming on the day it happens. No new concept shows up here; every wreck breaks a rule you already know, and every fix points back to the chapter that taught it.

Why a catalog of wrecks? Because code rots through shortcuts that look harmless in the diff. Nobody writes "coupling the slices" in a pull request description; they write "extracted a helper." This chapter's goal is immunization: after it, you look at a diff and name the wreck by its symptom, before the merge, while undoing it is still cheap. And so the catalog doesn't turn into a witch hunt, every card ends with the legitimate exception:

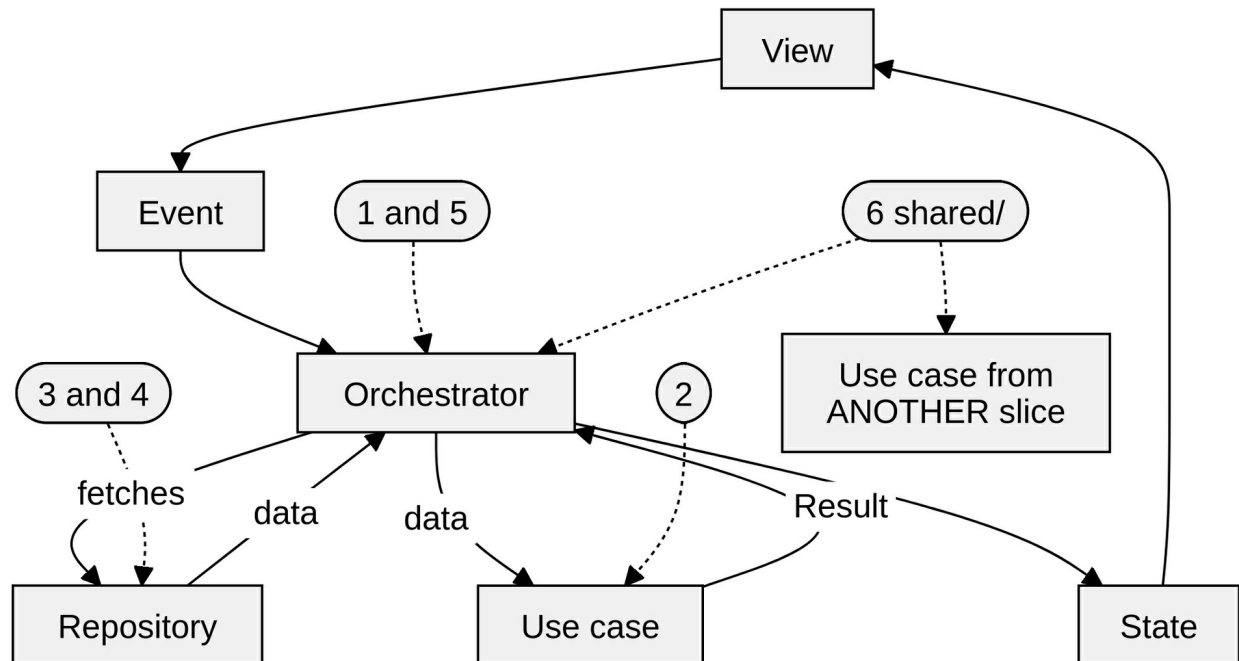
the case where that same code is NOT an anti-pattern. Keep that part. A reviewer who memorized the ban and forgot the exception is a wreck of a different kind.

The shape of the card

The six cards follow the same skeleton, in the order the book has always worked in (anti-solution before solution, since chapter 2):

- **Symptom:** what shows up in the diff or on screen, the phrase you use in code review.
- **Damage:** the concrete cost of change six months later, measured in files touched, tests rewritten, or silent defects.
- **Fix:** the refactor, with a citation to the chapter that taught the rule.
- **Legitimate exception:** when that same code isn't a wreck and the reviewer should let it pass.

The six wrecks attack different points of the flow chapter 10 drew. The diagram marks where each one punctures the arrow:



Markers 1 and 5 puncture the orchestrator (an inline business rule and a domain try/catch). Marker 2 punctures the use case (which starts fetching its own data). Markers 3 and 4 puncture the repository (too generic or too ceremonial). Marker 6 is the only one that crosses two slices at once: the premature shared/ couples use cases from different features through a common helper. Now, the cards.

Card 1: business rule in the orchestrator

Symptom: the orchestrator decides instead of connecting. In the diff, a business calculation (discount, eligibility, total) shows up inside the event handler, instead of a call to chapter 14's use case. The review comment is short: "that if is business, it doesn't live here."

Here's the wreck in code. Someone copied the discount rule into the orchestrator "because it was just an if," and the copy aged: the use case learned the `ItemOutOfStock` variant, and the copied

switch never found out.



```
class TabOrchestrator {  
  
  TabOrchestrator(this.tabs, this.loyaltyPointsByTable);  
  
  final Map<int, Tab> tabs;  
  
  final Map<int, int> loyaltyPointsByTable;  
  
  String on(PayTab event) {  
  
    final tab = tabs[event.table]!;  
  
    final points = loyaltyPointsByTable[event.table] ?? 0;  
  
    final DiscountResult result;  
  
    if (points < 100) {  
  
      result = NotEligibleForDiscount(points, 100);  
  
    } else {
```

```
final total = tab.totalInCents;

final discounted = total - total * 10 ~/ 100;

result = DiscountApplied(

    Tab(tab.table, tab.items, discounted),

);

}

switch (result) {

case DiscountApplied(:final tab):

    final dollars = (tab.totalInCents / 100).toStringAsFixed(2);

    return "Table ${tab.table}: \$$dollars";

case NotEligibleForDiscount(:final points, :final pointsNeeded):

    return "Not eligible: $points of $pointsNeeded points";
```

```
}  
  
}  
  
}
```

Here the compiler helps: since the Result is a sealed family (chapter 8), `dart analyze` catches the stale copy. The output below is real, not edited:

```
error - 19-card1-broken.dart:75:5 - The type  
'DiscountResult' isn't exhaustively matched by the switch cases  
since it doesn't match the pattern 'ItemOutOfStock()'. Try adding a  
default case or cases that match 'ItemOutOfStock()'. -  
non_exhaustive_switch_statement
```

You got lucky this time. The Kotlin version of the same wreck compiles without complaint, because whoever copied the rule used an `else` to silence the compiler:

Kotlin

```
return when {  
  
    points >= 100 && outOfStock == null -> {  
  
        val total = tab.totalInCents  
  
        val discounted = total - total * 10 / 100
```

```
    "Table ${tab.table}: \${discounted / 100}." +  
  
    "%02d".format(discounted % 100)  
  
  }  
  
  else -> "Not eligible: $points of 100 points"  
  
}
```

Run this code with an item out of stock and 120 points: the screen says “Not eligible: 120 of 100 points.” The customer is eligible; the item is what’s missing. The `else` turned a compile error into a lying message.

Damage: six months later, the discount rule exists in two places. Chapter 14’s use case evolves (birthday customers, happy hour, a discount cap) and the orchestrator’s copy doesn’t; which of the two versions Rosie’s register runs depends on which screen fired the event. The use case’s test passes, the defect stays in production, and the fix demands an archaeological diff to find out when the versions drifted apart.

Fix: the orchestrator goes back to only connecting. It fetches the data, hands it to the `applyLoyaltyDiscount` use case (chapter 14), and translates the Result into state, with the exhaustive switch chapter 13 demanded. The temptation has a source, and it isn’t the orchestrator’s lineage. The unidirectional architectures that inspired it never told anyone to concentrate rules there. The official Elm guide (guide.elm-lang.org) describes `update` as something that reacts to messages, the Redux documentation (redux.js.org, “Prior Art” section) inherits that design, and André

Staltz (staltz.com) describes the whole family as a data flow, not a decision warehouse. The reducer is the seat of the state TRANSITION; the business rule lives in the use case.

Dart

```
class TabOrchestrator {

  TabOrchestrator(this.tabs, this.loyaltyPointsByTable);

  final Map<int, Tab> tabs;

  final Map<int, int> loyaltyPointsByTable;

  String on(PayTab event) {

    final tab = tabs[event.table]!;

    final points = loyaltyPointsByTable[event.table] ?? 0;

    return switch (applyLoyaltyDiscount(tab, points)) {

      DiscountApplied(:final tab) =>

        "Table ${tab.table}: \${$_dollars(tab.totalInCents)}",
```

```

        NotEligibleForDiscount(:final points, :final pointsNeeded) =>

            "Not eligible: $points of $pointsNeeded points",

        ItemOutOfStock(:final name) => "No go: $name is out of stock",

    };

}

}

```

Kotlin spells the exhaustiveness check differently. A `when` used as an expression forces you to cover every variant of the sealed type, and the `else` is exactly what can't show up: with it, the compiler stops demanding the new variant. Notice also the `val r =` inside the `when`, which names the value so the `is` branches can read it:

Kotlin

```

class TabOrchestrator(

    private val tabs: Map<Int, Tab>,

    private val loyaltyPointsByTable: Map<Int, Int>,

) {

    fun on(event: PayTab): String {

```

```

    val tab = tabs.getValue(event.table)

    val points = loyaltyPointsByTable[event.table] ?: 0

    return when (val r = applyLoyaltyDiscount(tab, points)) {

        is DiscountApplied -> {

            val c = r.tab.totalInCents

            "Table ${r.tab.table}: \${c / 100}.\${"%02d".format(c % 100)}"

        }

        is NotEligibleForDiscount ->

            "Not eligible: ${r.points} of ${r.pointsNeeded} points"

        is ItemOutOfStock -> "No go: ${r.name} is out of stock"

    }

}

}

}

```


Legitimate exception: a trivial presentation `if` can stay in the orchestrator. Deciding whether the state becomes `Loading` or `Ready`, picking the generic error message, formatting cents as dollars: that's translating a `Result` into state, the orchestrator's actual job. The test is to ask "if Rosie changes the business rule, does this `if` change?" If the answer is no, it can stay.

Try it: open <https://focus.kodel.com.br/en/dart/19-01> (or swap `dart` for `kotlin`, `ts`, `java`, `csharp`, `go`, `php`, `python`, `swift`, `rust`) and run the fixed slice: an orchestrator that only connects and a use case that receives data. Delete one case from the translation switch and watch your language's compiler demand the missing variant.

Card 2: use case that hits the database

Symptom: the use case's signature picked up a dependency. In the diff, `applyLoyaltyDiscount(tab, points)` turned into `applyLoyaltyDiscount(repository, tab)`, almost always with the justification "just this once, it's a small lookup."

 Dart ·  Kotlin

```
int applyLoyaltyDiscount(  
  
    LoyaltyPointsRepository repository,  
  
    Tab tab,  
  
) {
```

```
final points = repository.pointsForCustomer(tab.table);

if (points < 100) {

    return tab.totalInCents;

}

final total = tab.totalInCents;

return total - total * 10 ~/ 100;

}
```

The signature lies. It claims to calculate a discount, but it also decides where the points come from. And the lie charges you at test time: what used to be `applyLoyaltyDiscount(tab, 120)` with literal values now demands building a repository double just to exercise a ten-line rule.

Damage: six months later, every new test of the rule pays the double's toll, and the use case stops composing. Chapter 16 chained use cases together because they all shared the same shape (data goes in, a Result comes out); a use case that fetches on its own breaks the chain, because nobody can call it without infrastructure wrapped around it. The coffee shop's most important rule becomes the most expensive one to test.

Fix: use cases receive DATA, not dependencies. The one who knows the repository is the orchestrator (chapter 13); it fetches the points and hands them over ready-made. It's the line chapters 7 and 9 drew: a pure function at the center, injection only at the edge, in the composition root Mark Seemann has described since 2011 (blog.ploeh.dk), on the foundation Martin Fowler laid in "Inversion of Control Containers and the Dependency Injection pattern" (2004).

 **Dart** ·  **Kotlin**

```
int applyLoyaltyDiscount(Tab tab, int loyaltyPoints) {  
  
    if (loyaltyPoints < 100) {  
  
        return tab.totalInCents;  
  
    }  
  
    final total = tab.totalInCents;  
  
    return total - total * 10 ~/ 100;  
  
}
```

Legitimate exception: when the rule demands multiple chained lookups (fetch, decide, fetch again based on the decision), pushing everything into the orchestrator turns it into a

procedural script. In that case a use case that orchestrates other pure use cases, a thin, documented command that receives the repository and delegates every decision to pure functions, is a solution, not a wreck. The sign of health: the RULES still live in functions that test with literals; only the choreography touches the repository.

Try it: the route <https://focus.kodel.com.br/en/dart/19-01> (and the other languages, same pattern) carries exactly this slice: card 1's fix and card 2's fix are the same code, because both wrecks are deviations from the same design.

Card 3: generic repository

Symptom: a single type serves the entire coffee shop. In the diff, `CoffeeShopRepository<T>` (or `Repository<T, TId>`) handles tabs, inventory, and payment, and its methods speak the database's language: a string filter, a string sort key, a string table name. The TypeScript ecosystem knows this plague well, because ORMs hand you the generic one ready-made, right out of the box.

TypeScript

```
class CoffeeShopRepository<T> {  
  
  constructor(  
  
    private readonly tableName: string,  
  
    private readonly rows: ReadonlyMap<string, T>,  

```

```

) {}

query(filter: string): T | undefined {

    return this.rows.get(`${this.tableName}:${filter}`);

}

list(sortBy: string): readonly T[] {

    return [...this.rows.values()].sort((a, b) =>

        String((a as Record<string, unknown>)[sortBy]).localeCompare(

            String((b as Record<string, unknown>)[sortBy]),

        ),

    );

}

}

```

Look at the callers: `tabs.query("table=4")`, `inventory.query("name=Espresso")`. Every call carries a query fragment. The repository existed to be the database's boundary (chapter 15); the generic version tore the

boundary open and let the database leak into every file that consumes it.

Damage: six months later, changing the database schema means hunting strings across the whole application, because every feature writes its own filters by hand. And the verbs the business actually needs don't exist: `markAsPaid` turns into a scattered `update("status=paid")`, with no single place for the write rule. Ben Morris called the generic repository a lazy anti-pattern in “Why the generic repository is just a lazy anti-pattern” (ben-morris.com), and the original definition of a repository, in Martin Fowler's *Patterns of Enterprise Application Architecture* (2002), already called for an interface speaking the domain's language.

Fix: one repository per feature, with that feature's verbs and nothing else, the way chapter 15 built it. The infrastructure exception gets translated exactly once, right here, and only Results circulate outward.

TypeScript

```
class TabRepository {  
  
  constructor(private readonly http: HttpClient) {}  
  
  findTab(table: number): LookupResult {  
  
    try {  
  
      return { kind: "tabFound", tab: this.http.query(table) };  
  
    } catch {
```

```
        return { kind: "infraFailure", failure: "noConnection" };  
  
    }  
  
}  
  
}
```

Legitimate exception: an INTERNAL generic is fine. If `TabRepository` and `InventoryRepository` share a private `QueryRunner<T>`, hidden behind the business verbs, nobody outside sees the generic and the boundary stays closed. The symptom was never the `<T>` itself; it's `<T>` in the PUBLIC signature, which forces the caller to speak the database's language.

Try it: open <https://focus.kodel.com.br/en/ts/19-02> (or swap `ts` for your language) and run the per-feature repository. Take down the network (`new HttpClient(true)`) and watch the outage turn into a typed `Failure.noConnection` instead of a loose exception.

Card 4: layer by ceremony

Symptom: files that only pass things along. In the diff, an `ITabRepository` interface with exactly one implementation, a `TabDto` identical to the model, and a mapper that copies field by field. No new behavior; just toll booths between the call and the data.

 Kotlin ·  Dart

```
data class TabDto(val table: Int, val totalInCents: Int)
```

```
data class Tab(val table: Int, val totalInCents: Int)
```

```
interface ITabRepository {
```

```
    fun findTab(table: Int): Tab
```

```
}
```

```
fun dtoToDomain(dto: TabDto) = Tab(dto.table, dto.totalInCents)
```

```
class TabRepositoryImpl(
```

```
    private val dtos: Map<Int, TabDto>,
```

```
) : ITabRepository {
```

```
    override fun findTab(table: Int) = dtoToDomain(dtos.getValue(table))
```

```
}
```


Damage: six months later, adding a field to the tab crosses four files (model, DTO, mapper, interface) to reach the same place, and any one of them can drift silently. The team starts “forgetting” the field in the DTO, the mapper zeroes the value out, and the defect shows up far from its cause. Alex Bolboacă ran these numbers in “Is Hexagonal Architecture Overengineering?” (mozaicworks.com, 2025): layers pay for themselves when they isolate change, and charge you when they just repeat types. Dan North proposed CUPID (dannorth.net, 2022) with the same target in mind: code that’s a joy to work with carries no ceremony, and ceremony that protects nothing is dead weight.

Fix: the concrete class, no ceremony interface and no twin DTO, the way chapters 6 and 9 argued. Seemann is explicit in the dependency injection book (*Dependency Injection in .NET*, 2011; 2nd edition, 2019): you extract an abstraction when the second REAL implementation shows up, not before, not “just in case.”

 **Kotlin** ·  **Dart**

```
data class Tab(val table: Int, val totalInCents: Int)

class TabRepository(private val tabs: Map<Int, Tab>) {

    fun findTab(table: Int) = tabs.getValue(table)

}
```

Legitimate exception: an interface with two or more real implementations is architecture, not ceremony. And chapter 17’s test double COUNTS as a real implementation: if the tests’ in–

memory repository implements the same contract as the HTTP repository, the interface is earning its own keep. The same holds for a DTO when the boundary genuinely diverges from the domain (the card processor's JSON isn't your tab). Ceremony is the layer that exists with no second form in sight.

Try it: the route <https://focus.kodel.com.br/en/kotlin/19-02> (and the other languages) shows the fix's concrete repository: card 3's fix and card 4's fix land in the same file, because the generic type and the single-implementation interface die together.

Card 5: domain try/catch

Symptom: a business case treated like an accident. In the diff, a `try { } catch (e) { log(e); }` wraps a call that returns a legitimate business refusal, and the flow moves on as if nothing happened.

 Dart ·  Kotlin

```
String pay(int table, int cents) {  
  
    try {  
  
        processor.charge(cents);  
  
    } catch (e) {  
  
        print("log: $e");  
  
    }  
}
```

```
}  
  
return "Table $table: payment approved";  
  
}
```

Run it: the card is over its limit, the processor declines it, the log records it, and the screen prints “Table 4: payment approved.” Rosie finds out at closing, when she counts money that never came in.

Damage: a silent failure is the most expensive defect to diagnose, because the symptom shows up far from the cause, days later. Chapter 3 introduced GitClear’s reports on AI-generated code; the 2026 edition measured *error masking*, exactly this wreck, with a 47% rise (gitclear.com). Code generators prefer the silent catch that makes today’s test pass and hides tomorrow’s refusal. This is the only anti-pattern on the list that the tools commit STRAIGHT OUT OF THE BOX: if you accept AI suggestions without reading the catch, you already have this wreck in your repository.

Fix: a business refusal is a value, not an exception (chapter 8). The payment Result gains a `CardDeclined` variant with a typed reason, and the only try/catch left standing lives at the repository’s boundary (chapter 15), which translates the processor library’s exception exactly once. It’s the design Scott Wlaschin called Railway-Oriented Programming (fsharpforfunandprofit.com, 2013): the error rail runs alongside the success rail all the way to the screen, with no invisible detours.

```
PaymentResult charge(int cents) {  
  
    try {  
  
        processor.charge(cents);  
  
        return PaymentApproved(cents);  
  
    } on StateError catch (e) {  
  
        return CardDeclined(e.message);  
  
    } catch (_) {  
  
        return InfraFailure();  
  
    }  
  
}
```

And the caller has no way to lie: the exhaustive switch over the sealed Result forces the screen to show the refusal.

```
String pay(int table, int cents) =>  
  
    switch (repository.charge(cents)) {
```

```
PaymentApproved() => "Table $table: payment approved",

CardDeclined(:final reason) =>

    "Table $table: payment declined ($reason)",

InfraFailure() => "Table $table: no connection, try again",

};
```

Legitimate exception: a try/catch AT the infrastructure boundary is exactly where it belongs, and it's one per slice, not one per call. The repository above uses a try/catch and isn't an anti-pattern: it translates, once, the outsider's exception into the insider's value. The card's symptom is a catch in DOMAIN code, the kind that swallows a business decision.

Try it: open <https://focus.kodel.com.br/en/dart/19-03> (or your language) and run both charges: the declined one shows up declined. Then swap `on StateError` for a generic catch that returns approved, and watch the lie come back.

Card 6: premature shared/

Symptom: this is the only one you recognize by the file's PATH, before reading a single line: `shared/helpers/` receiving code on the second occurrence of a similar-looking calculation. Two features had similar functions; someone unified them "to avoid duplication."

```
// shared/helpers/discount_calculator.dart

int calculateDiscount(

  int totalInCents,

  int points, {

    required bool isBirthday,

  }) {

  if (isBirthday) {

    return totalInCents - totalInCents * 15 ~/ 100;

  }

  if (points < 100) {

    return totalInCents;

  }

  return totalInCents - totalInCents * 10 ~/ 100;
```

}

Notice the `isBirthday` boolean. It's the scar left by the unification: the two features did NOT share the same rule, they had similar-looking rules, and the helper needed a parameter to break the tie. Every new divergence adds another parameter and another if.

Damage: here's my position, stated in the first person: I consider the premature shared/ the most expensive anti-pattern on this list, because it's the hardest to reverse. The other five undo themselves by editing one slice; this one demands touching EVERY consumer of the helper, working out which half of the if each one uses, and separating back out what never should have been glued together. The cost of reversing it grows with the number of consumers, and consumers only ever increase. Sandi Metz named the cause: "duplication is far cheaper than the wrong abstraction" (sandimetz.com, 2016). Kent C. Dodds turned the advice into an acronym, AHA (Avoid Hasty Abstractions, kentcdodds.com/blog/aha-programming), and the rule of three, which chapter 11 brought over from Fowler's *Refactoring* (1999), gives the number: extract on the third occurrence, never the second.

Fix: each feature keeps its own function, and the duplication gets accepted as the cost of independence between slices, the design Jimmy Bogard argues for in Vertical Slice Architecture (jimmybogard.com, 2018), the one chapter 11 adopted.

 Dart ·  Kotlin

```
// features/tab/loyalty_discount.dart

int loyaltyDiscount(int totalInCents, int points) {
```

```

if (points < 100) {

    return totalInCents;

}

return totalInCents - totalInCents * 10 ~/ 100;

}

// features/loyalty/birthday_bonus.dart

int birthdayBonus(int totalInCents) =>

    totalInCents - totalInCents * 15 ~/ 100;

```

Two functions, two slices, zero tie-breaking parameters. If tomorrow the birthday bonus becomes 20%, the tab feature doesn't even find out.

Legitimate exception: extracting to shared/ is legitimate once the reuse has PROVEN itself: a third occurrence, the same rule (not similar-looking rules), and the same reason to change in all three. Formatting cents as dollars is the classic example: every spot formats the same way and changes together. The test isn't "does the code look alike?"; it's "when one changes, DOES the other have to change too?"

Try it: open <https://focus.kodel.com.br/en/dart/19-04> (or your language) and run the two independent features. Then try reintroducing the single helper and count how many tie-breaking parameters you need to keep the same output.

What Go won't let you do

Two of the six cards aren't expressible in Go, and that earns a page of counterpoint instead of forced examples. There's no inheritance or type hierarchy to hide a `CoffeeShopRepository[T]` behind subclasses (card 3 is born crippled), and there's no exception for a generic catch to swallow (card 5 simply doesn't compile in spirit: there's no throw). The error in Go is a value returned, the way Rob Pike summed up in "Errors are values" (the official Go blog, 2015), and a returned value shows up in the signature:



Go

```
func Charge(cents int) (int, error) {  
  
    if cents > 3600 {  
  
        return 0, errDeclined  
  
    }  
  
    return cents, nil  
}
```

```
}
```

The caller pays the price in verbosity, and it's a price, not a detail: the `(T, error)` pair charges an `if err != nil` on every call, line after line, where Dart's sealed Result charges one switch per translation.

```
value, err := Charge(total)

if err != nil {

    return fmt.Sprintf("Table %d: payment declined (%v)", table, err)

}
```

In exchange, ignoring the refusal becomes a decision visible in the diff (an `_` where `err` should be), not a forgotten catch three layers up. The counterpoint's lesson holds for the other nine languages: the less your language prevents by construction, the more this chapter's cards are the discipline holding the roof up.

Pitfalls

A reader who finishes this chapter with a trained eye runs a new risk: rushing off to fix all six wrecks at once, in a single heroic-refactor pull request. That big bang is the seventh wreck. A diff that touches the orchestrator, the use case, the repository, and shared/ all at the same time is impossible to review, impossible to revert, and nearly guaranteed to break behavior no test was

covering. Fixing an anti-pattern follows the same rule as every change in FOCUS: one card per pull request, one slice at a time, with the slice's test green before and after. Chapter 20 shows the grown-up version of this discipline, strangling legacy code; save the impulse for there.

Q&A

I fixed card 1 and the orchestrator ended up three lines long. Isn't that layer by ceremony? No: ceremony is a layer that only passes things along WITHOUT protecting anything. The three-line orchestrator protects the View from knowing the use case and the use case from knowing the screen, and it's where the state is born. It's small because it's right.

And the duplication between features, nobody pays for that? Somebody does, and the book accepts the price with eyes open: duplicating a ten-line calculation costs less than coupling two slices through a helper with a tie-breaking boolean. The accepted cost has a clear limit (chapter 11): on the third occurrence of the SAME rule, changing for the same reason, extract it. Before that, duplication is cheap rent; coupling is a mortgage.

My language has no sealed types or exhaustive switch. Do cards 1 and 5 still apply to me? They apply harder: with no compiler demanding the missing variant, you're left with chapter 18's discipline (a single translation, Result by convention, a test for the translation). The Go counterpoint above is the same reasoning.

Quick tip

Card 2's wreck gets hunted with a grep. In a flat slice the only file allowed to know about infrastructure is the repository, so `grep -rn "import" features/ | grep -E "http|sql|dio|axios" | grep -v _repository` has to come back empty. Hang that grep on the project's lint step and card 2 never gets past a pull request again.

Quick reference

Symptom in the diff	Anti-pattern	Ch.
Business calculation in the handler	Rule in the orchestrator	13 and 14
Repository in the use case's signature	Use case that hits the database	7 and 9
Public <code>Repository<T></code> , string filter	Generic repository	15
Single-implementation interface, twin DTO	Layer by ceremony	6 and 9
Generic <code>catch</code> that logs and moves on	Domain try/catch	3 and 8
Helper in <code>shared/</code> on the 2nd occurrence	Premature shared/	11

Anti-pattern	Fix
Rule in the orchestrator	the rule moves back to the use case
Use case that hits the database	use case receives data; the orchestrator fetches
Generic repository	one repository per feature, business verbs
Layer by ceremony	concrete class until the 2nd real implementation
Domain try/catch	refusal becomes a Result variant; translate at the edge
Premature shared/	each feature keeps its own function; rule of three

Exercises

1. The diff below landed in a coffee shop pull request. It contains TWO anti-patterns from this chapter, one inside the lines and one outside them. Name both, citing each card's symptom. The answer key follows right after; try it before you read it.

```

--- /dev/null

+++ b/shared/helpers/payment_helper.dart

@@ -0,0 +1,13 @@

```

```

+String paySecurely(int table, int cents) {

+  try {

+    final result = repository.charge(cents);

+

+    if (result is CardDeclined) {

+      log("declined: ${result.reason}");

+    }

+  } catch (e) {

+    log(e);

+  }

+

+  return "Table $table: payment approved";

+}

```

2. Open question, no answer key: open your OWN current project's repository and walk through the quick reference table row by row. How many of the six wrecks exist in it

today? Which one has the most consumers, and therefore costs more with every week that passes?

Answer key for exercise 1: inside the lines, a domain try/catch (card 5): the payment's refusal is logged and swallowed, and the function returns "approved" unconditionally, the same design as the catch that logs and moves on. Outside the lines, in the file's path, a premature shared/ (card 6): `shared/helpers/payment_helper.dart` is a payment helper being born outside the payment slice. Two wrecks, and you identified the second one without reading a single line of code.

Tip 19

The cost of reversing a wreck grows with the number of consumers. That's why the premature shared/ is the most expensive one on the list, and why the time to name the anti-pattern is in the diff, while the consumer count is still one.

Next chapter: and what about when the code was already born with all six wrecks at once, in a ten-year-old legacy system holding up the company's register?

Migrate Legacy Code Without Stopping the Factory

In this chapter, you'll:

- pick the first slice to migrate with a criterion that has a number (frequency of change times pain), not gut feeling;
- run the four steps of strangling on a real slice: characterization, use cases, adapter, new view;
- write down, in the open, what you will NOT migrate, covering the four cases where migrating is waste.

Chapter 19 ended with a question: what happens when the code was born with all six damages at once, in a ten-year-old legacy that carries the company's cash register? This chapter answers it. You won't rewrite anything. You'll fence today's behavior with a test that freezes it, pull one slice at a time into FOCUS's shape, and let the rest of the old system keep running in peace, behind a boundary that lives in a single file.

Almost no reader of this book holds a greenfield project: a field built from scratch, with no inherited line of code to respect. The project that pays your salary is probably an old system, written by people who already left the company, with no tests, business rules scattered everywhere. If your case is the rare greenfield, read on anyway: this chapter is the vaccine against the decision that kills the most healthy systems, the full rewrite, and the

investment criterion you'll use once your greenfield ages. All code turns into legacy; the only open question is who's on call when it happens.

Our patient is Rosie's Coffee Shop's tab system, production version: eight-year-old procedural PHP that runs the counter every day. Here's its heart, the "controller" that closes a tab:

PHP

```
function calculate_total(array $items): float
{

    $total = 0.0;

    foreach ($items as [$name, $price]) {

        $total += $price;

    }

    // Closing rounds the float to 2 places and moves on with its life.

    return round($total, 2);

}
```

```
// The "controller": fetches data, applies the business rule, and  
  
// builds the response, all in the same place.  
  
function controller_pay(int $table): string  
{  
  
    $items = find_tab($table);  
  
    $total = calculate_total($items);  
  
    // The card machine speaks cents; the conversion truncates the float.  
  
    $cents = (int)($total * 100);  
  
    // Inline business rule: the card machine's limit, hardcoded.  
  
    if ($cents > 5000) {  
  
        // A business refusal disguised as an accident.  
  
        throw new DomainException("card limit");  
  
    }  
}
```

```
return "table $table: paid $cents cents";  
  
}
```

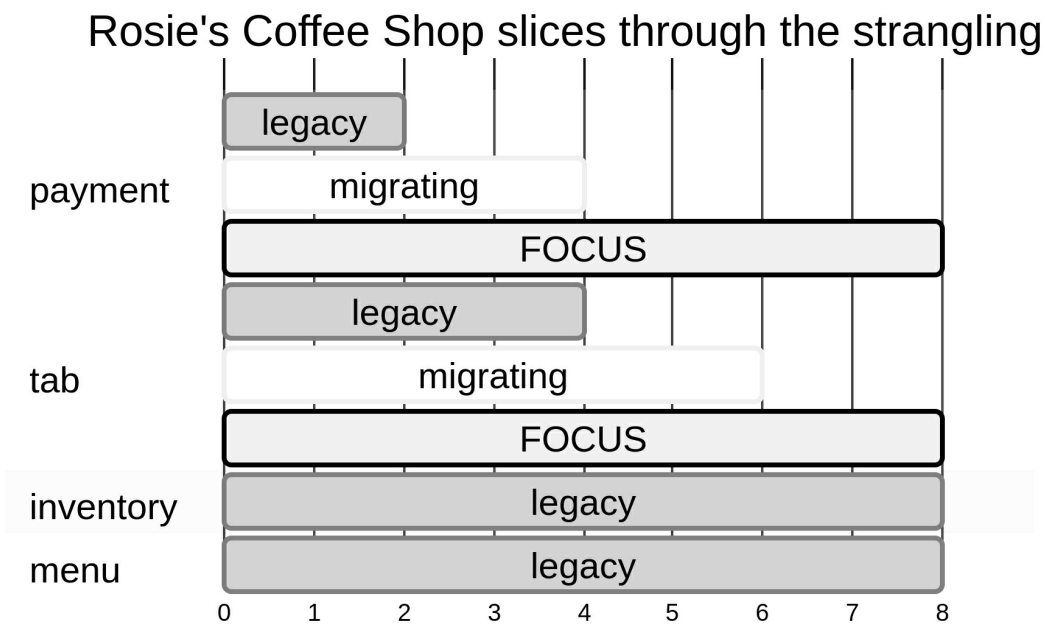
You just walked straight out of an anti-pattern catalog; use it. The limit rule inside the controller is entry 1 of chapter 19 (business rule in the orchestrator, except here there's no orchestrator at all). The payment refusal traveling as a `DomainException` up to a generic catch at the top is entry 5 (domain try/catch), made worse by the exception crossing every layer, because there are no layers. Prices live in a float, and the conversion to cents truncates whatever the float got wrong. And there isn't a single test. This code breaks half a dozen rules you know by name, and it still has one virtue your new diagram doesn't: it has been closing real tabs for eight years. Respect that. This chapter's goal isn't to erase this file; it's to retire it gradually, without Rosie ever noticing.

The fig that strangles

The pattern has a plant's name: strangler fig. Martin Fowler coined `StranglerFigApplication` on his bliki (martinfowler.com, 2004; the post was originally called "StranglerApplication" and got renamed in 2019): a new system grows at the edges of the old one, route by route, until the old one stops receiving calls and can be switched off with no funeral. The analogy comes from the strangler figs Fowler saw in Australia: the seed germinates high on a host tree, roots climb down the outside of the trunk to the ground, and years later the fig stands on its own, shaped like the tree that hosted it. At no point did the forest go without a tree in

that spot. That's the pattern's entire promise, and it's the yardstick for every decision in this chapter: at no point does the coffee shop go without a tab system.

FOCUS adapts the pattern with one directional decision: strangle by feature, never by layer. The layer-by-layer alternative looks organized on a slide ("first we migrate all of persistence, then all the services, then the screens"), and it's a big-bang rewrite wearing a new hat. While the whole layer isn't ready, nothing works end to end; the value only shows up at the end, which is exactly the flaw strangling exists to avoid. Migrating the whole payment slice (view, orchestrator, use case, repository, a vertical cut from chapter 11) delivers a feature running on the new shape in the first week already, while tab, inventory, and menu keep running on the old PHP, untouched.



Look at the diagram: inventory and menu might well die legacy, and that's fine. Strangling isn't a purity crusade; it's an investment, and investments get chosen.

Choose the first slice: frequency times pain

Which slice to migrate first? The answer isn't "the ugliest one." It's the one that combines two measures you already have in the repository: how often that piece changes (count commits per area over the last few quarters) and how much it hurts when it breaks. Ugly code nobody touches charges no rent; decent code that changes every week charges compound interest. Rosie's Coffee Shop board, last quarter's commits:

Feature	Commits this quarter	Pain when it breaks
payment	31	a declined payment brings down the whole tab
tab	12	a wrong order noted, customer waits
loyalty	9	wrong points, complaint at the counter
inventory	6	manual count at closing time
menu	2	stale price until someone edits it

Payment wins on both axes. That's 31 commits in a quarter (a change every two business days in code with zero tests), and the pain is the worst on the list: when payment fails, the whole tab jams and the line walks over to the competitor. The first slice is elected, with a number you can defend in a meeting. Run this count on your own system before you write a single line of code; if the commit champion barely hurts when it breaks, choose by pain instead, because commits measure activity and pain measures consequence.

Step 1: fence the behavior with a characterization test

Before you move a single line, freeze what exists. A characterization test is a test that documents what the code DOES today, not what it should do; the term comes from Michael Feathers, in *Working Effectively with Legacy Code* (2004), the same book that defines legacy in the most useful way I know: legacy is code with no tests. Not old code, not ugly code. Code whose behavior nobody can state with confidence, and characterization exists to turn that ignorance into a contract.

The detail that separates a characterization test from an ordinary one: if the legacy has a bug, the test expects the bug. Rosie's system has one, and it's a good one. Table 7's tab has a \$7.00 espresso and a \$9.90 slice of cake. Add it up in your head: \$16.90, or 1690 cents. The legacy charges 1689. The cause lives in binary representation: 16.90 doesn't exist as an exact double (the closest neighbor is `16.89999999999999858`), and the conversion `(int)($total * 100)` truncates `1689.9999999999998` down to 1689. The `round($total, 2)` that closing does doesn't save it, because the result of round is the *same* crooked double. One cent per tab, for eight years, on every sum that lands on a neighbor below.

The temptation to fix it right now is enormous. Resist it. The characterization test expects 1689 on purpose:

 **PHP**

```
// What each table produces today, exceptions normalized as text.  
  
// Table 7's 1689 is wrong in the arithmetic and correct in the  
  
// characterization.
```

```
$cases = [  
  
    [4, "table 4: paid 1300 cents"],  
  
    [7, "table 7: paid 1689 cents"],  
  
    [9, "table 9: declined (card limit)"],  
  
    [99, "table 99: tab not found"],  
  
];  
  
$matched = 0;  
  
foreach ($cases as [$table, $expected]) {  
  
    try {  
  
        $got = controller_pay($table);  
  
    } catch (DomainException $exception) {  
  
        $got = "table $table: declined ({ $exception->getMessage()})";  
  
    } catch (RuntimeException $exception) {  
  
        $got = "table $table: { $exception->getMessage() }";  
  
    }  
  
}
```

```

    }

    if ($got === $expected) {

        $matched++;

        echo "ok $got\n";

    } else {

        echo "FAILED expected [$expected], got [$got]\n";

    }

}

```

No test framework, on purpose: a table of cases, a loop, and a string comparison are enough, and they run wherever the legacy runs. Notice that the test fences the legacy *from the outside*, through the public interface (the controller function), without touching the fenced file. And notice what the table freezes: table 4's correct value, table 7's wrong value, table 9's refusal, and table 99's exception, all carrying the same weight. That's the contract. From here on, any change that alters one of these four lines is a behavior change, and a behavior change during a migration is a defect, even when the new value is the arithmetically correct one. Table 7's cent will get fixed, but after the strangling, as a separate change, with the table updated on purpose and Rosie warned. Migration changes structure; a fix changes behavior. Never in the same commit.

Try it: open <https://focus.kodel.com.br/en/php/20-01> and run the legacy; table 7 pays 1689 cents. Then open <https://focus.kodel.com.br/en/php/20-02> and run the characterization: 4 of 4 cases match, bug included. Now “fix” the truncation in the legacy and run the characterization again. The FAILED that shows up is the test doing its job: you changed behavior in the middle of a migration.

Step 2: extract the rule into a pure use case

With the fence in place, start moving the business rule to where it should have lived from the start: a pure function, in chapter 14’s shape. Rosie’s payment rule is the limit decision, buried today in the controller as a throw. Extracted, it turns into data going in and a result coming out:

 **Dart** ·  **TypeScript**

```
PaymentResult payTab(int totalInCents, int limitInCents) {  
  
    if (totalInCents > limitInCents) {  
  
        return PaymentDeclined("card limit");  
  
    }  
  
    return PaymentApproved(totalInCents);  
}
```

```
}
```

```
function payTab(  
  
    totalInCents: number,  
  
    limitInCents: number,  
  
): PaymentResult {  
  
    if (totalInCents > limitInCents) {  
  
        return { type: "paymentDeclined", reason: "card limit" };  
  
    }  
  
    return { type: "paymentApproved", totalInCents };  
  
}
```

Two languages, the same scene: the refusal stopped being an exception and became a Result variant, exactly as chapter 8 called for. The limit stopped being a magic number and became a parameter. And the function tests with two literals, no test double at all, because it depends on nothing. What the use case does *not* do matters as much as what it does: it doesn't recompute the tab's total. The total's arithmetic, lost cent included, stays the old code's responsibility, and the next step explains why.

Step 3: wrap the legacy in an adapter

Here comes the chapter's third new concept. A legacy adapter is the old code placed behind the slice's repository interface: to whoever consumes it, it's a repository like chapter 15's; on the inside, the work is done by eight-year-old PHP. One sentence to untangle the name collision: the Adapter from the GoF (Gang of Four, the nickname for the authors of *Design Patterns*, 1994) catalog converts one interface into another in the general case, and the legacy adapter is that same gesture with one fixed purpose: hide an entire system behind one slice's contract. It's the piece that makes migrating without rewriting possible: the new use case sees the legacy as replaceable infrastructure, the same way it would see a database or an API.

Dart

```
class LegacyAdapter implements PaymentRepository {

  @override

  LookupResult tabTotal(int table) {

    try {

      return TotalAvailable(_legacyCents(_legacyCalculateTotal(table)));

    } on StateError {

      // The legacy exception becomes a Failure HERE, in one place.

      return InfraFailure(Failure.tabNotFound);
    }
  }
}
```

```
}  
  
}  
  
}
```

Two decisions live in this small file. First: the adapter *delegates* the total's calculation to the old code, instead of reimplementing the sum. That's why table 7's lost cent crosses the adapter intact, and step 1's characterization keeps passing; if the adapter redid the math "the right way," table 7 would pay 1690, the test would break, and you'd have changed behavior by accident. Second: the exception the legacy throws gets translated into `Failure` exactly once, at this boundary, the same rule from chapters 8 and 15. The rest of the new slice never sees a throw from the old world. Whenever the legacy is finally switched off, this is the only file that dies with it.

In Go, the same step wears a different face, and the difference is worth learning from. There's no exception to translate: the legacy's error is already born a value, in the `(T, error)` pair. The Go adapter normalizes instead of translating: it takes the old code's open `error` and fits it into the slice's typed `Failure`.

Go

```
func (LegacyAdapter) TabTotal(table int) LookupResult {  
  
    total, err := legacyCalculateTotal(table)  
  
    if err != nil {
```

```
// The legacy error becomes a Failure HERE, in one place.

failure := FailureTabNotFound

return LookupResult{Failure: &failure}

}

return LookupResult{TotalInCents: legacyCents(total)}

}
```

The boundary stays a single place; what changes is the verb. In exception-based languages, the boundary translates; in error-as-value languages, it normalizes. If your legacy is Go or Rust, step 3 gets cheaper, and it's no less necessary for that: a raw `error` saying “sql: no rows” leaking into the use case couples the new slice to the old database the same way an exception would leak.

Step 4: wire the new view to the orchestrator

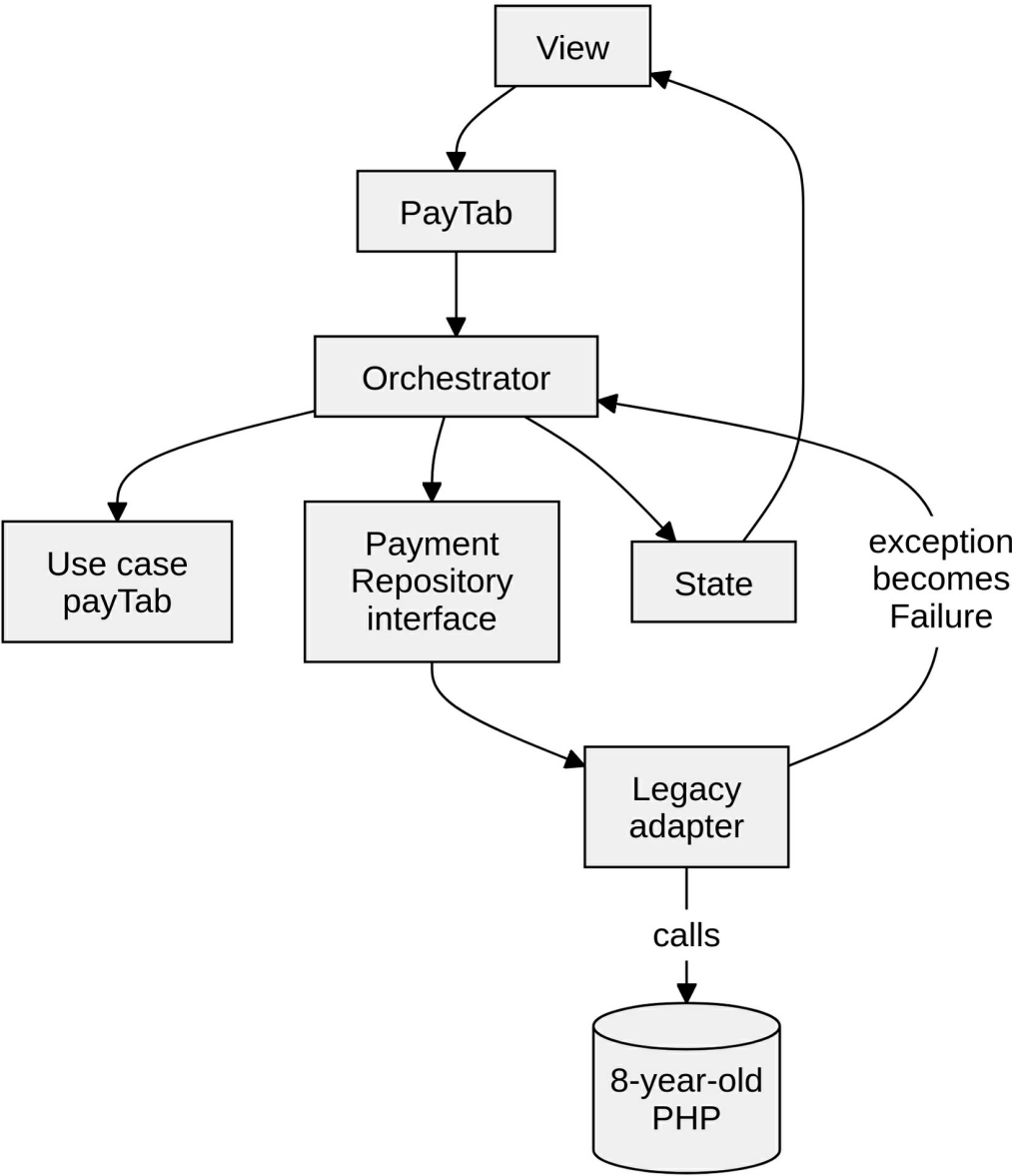
The last step has no new concept, and that's on purpose: dumb view and orchestrator are chapters 12 and 13, and they work here with zero adaptation. The orchestrator receives the `PayTab` event, asks the repository (which is the adapter, though it doesn't know that) for the total, hands the total to the use case, and translates the result into state for the view:



```
class PaymentOrchestrator {  
  
  PaymentOrchestrator(this._repository);  
  
  final PaymentRepository _repository;  
  
  TabState on(PayTab event) =>  
  
    switch (_repository.tabTotal(event.table)) {  
  
      TotalAvailable(:final totalInCents) => switch (  
  
        payTab(totalInCents, 5000)) {  
  
          PaymentApproved(:final totalInCents) =>  
  
            Ready("table ${event.table}: paid $totalInCents cents"),  
  
          PaymentDeclined(:final reason) =>  
  
            ErrorState("table ${event.table}: declined ($reason)",  
  
          },  
  
      InfraFailure() => ErrorState("table ${event.table}: tab not found"),  
    }  
}
```

```
};  
  
}
```

The slice is complete, and its drawing shows where the old world ended up:



The proof is still missing. The strangling's success criterion is objective: the *same* characterization from step 1, run against the new slice, has to produce the *same* output, byte for byte. Running the table against the legacy PHP and against the new orchestrator:

```
ok table 4: paid 1300 cents
ok table 7: paid 1689 cents
ok table 9: declined (card limit)
ok table 99: tab not found
characterization: 4 of 4 cases match
```

The two outputs are identical; the diff between them is empty. Table 7 keeps paying the wrong 1689 cents, and that's how you know the migration didn't change behavior: even the bug arrived alive on the other side. New structure, old behavior, contract fulfilled.

Try it: open <https://focus.kodel.com.br/en/dart/20-03> (or swap `dart` for `ts`, `go`, `kotlin`, `swift`, `csharp`, `python`, `java`, `php`, `rust`) and run the migrated slice: the output is identical to the legacy's characterization, lost cent included. Then change the limit from 5000 to 6000 in the use case and run it again. Table 9 gets approved, and the characterization's FAILED shows the test catching a rule change, now in a place where the rule has an owner.

When NOT to migrate

This section exists because the whole chapter is a hammer, and after learning the four steps every system starts looking like a nail. It isn't. I've seen more value destroyed by unnecessary migration than by poorly kept legacy, and I stand by the choosing section's criterion to the end: migrating code that doesn't change is paying interest on a debt nobody is collecting.

The first case is stable code. Rosie's menu module had 2 commits this quarter, both price adjustments. Is it ugly? Yes. Does it cost anything? No. The right answer for it is a thin characterization (step 1 alone, none of the other three) and nothing else: the fence guarantees nobody breaks it by accident, and the cost of carrying it ugly is zero as long as it doesn't change. The second case is the system with a marked end of life: if the current card machine gets discontinued by the vendor in eighteen months and the module dies with it, every hour spent migrating is an hour thrown in a bin with a date stamped on it. The third is the module about to be replaced by a purchase: if the coffee shop's accounting is about to become an off-the-shelf SaaS (Software as a Service) next year, characterize the data export and stop there.

The fourth case is the full rewrite, and it arrives disguised as virtue, with four different names in the mouth of whoever proposes it: modernization, standardization, deep refactor, version 2. Under all four it's always the same sentence: "since the legacy is bad, let's rewrite everything at once." Joel Spolsky called the full rewrite "the single worst strategic mistake that any software company can make" in "Things You Should Never Do, Part I" (joelonsoftware.com, 2000), written about Netscape 6, the rewrite that took three years, shipped nothing in between, and handed the market to the competitor. His argument aged well: old, ugly code carries decades of fixes nobody documented, and a rewrite throws those fixes out along with the ugliness. Strangling exists precisely to capture that knowledge (characterization freezes the fixes, including the ones that look

like bugs) instead of betting it on a rewrite. If someone at your company proposes the full rewrite, the counterproposal fits in one sentence: same budget, one slice at a time, value delivered every week.

Both worlds on the same counter

After the first slice, the coffee shop lives a coexistence that bothers tidy people: payment runs on the new shape, everything else runs on the old PHP, and both worlds share the same counter. The feature board gains a column and turns into the migration's progress panel:

Feature	Commits this quarter	Strangled?
payment	31	yes
tab	12	in progress
loyalty	9	no
inventory	6	no
menu	2	no (and maybe never)

This table costs five minutes a week and answers the question every boss asks (“how much is left?”) with data instead of a feeling. The real cost of the coexistence is temporary inconsistency: for a few months, a declined payment is a typed `PaymentDeclined` in the new slice and a `DomainException` in the rest of the system. Own that cost out loud, with a deadline: inconsistency is an acceptable intermediate state when it has an end date, and an unacceptable final state when it doesn't. The entire boundary

between the two worlds lives in one file, the adapter, and that's what keeps the cost low: nobody needs to remember where the old touches the new, because the spot has a name and an address.

There's still the usual criticism, the same one the vertical slice has heard since chapter 11: "now the limit rule exists twice, in the new use case and in the old controller." It does, and the answer is the defense Jimmy Bogard makes of Vertical Slice Architecture (jimmybogard.com, 2018): coupling slices to eliminate duplication trades a visible, cheap cost for an invisible, expensive one. Here the trade is even worse, because the "reuse" would couple the new code to the old code you're trying to retire; the duplication during strangling is scaffolding, not debt, and it dismantles itself the moment the last call to the old controller dies. Sandi Metz gave this instinct a ruler in "The Wrong Abstraction" (sandimetz.com, 2016): duplication is cheaper than the wrong abstraction, and a wrong abstraction over a dying legacy is the wrongest of all.

Pitfalls

The migration that turns into a rewrite. You're at step 2, in the middle of extracting the limit rule, and you notice the loyalty calculation is a disgrace too. "While we're at it..." is the sentence that turns a one-week migration into a three-month swamp; every "while we're at it" doubles the diff and the risk. The slice's scope is the boundary: loyalty has 9 commits on the board and will get its turn. Write it down, close the current slice's pull request, migrate the next one when its time comes.

The characterization that fixes the bug. You write table 7's test, see 1689, "know" the right answer is 1690, and write 1690 as the expected value. The test is born red, you "fix" the legacy so it passes, and there it goes: you destroyed the contract the test

existed to freeze. Now there's no way to tell whether the migrated slice behaves like the legacy, because the legacy changed in the middle of the measurement. Worse: Rosie's accounting has been closing the register with 1689 for eight years, and your "corrected" cent just created an accounting discrepancy nobody asked for. The characterization test documents what IS. The fix comes later, separate, announced.

Q&A

The payment slice needs the tab's data, and the tab is still legacy. Do I migrate both together? No; the adapter is the answer. The new slice sees the tab through the repository interface, and whoever implements that interface today is the old code wrapped up. When the tab slice gets migrated, you swap the implementation behind the interface and the payment use case doesn't even recompile differently.

My system is greenfield; do I throw this chapter away? Keep at least two pieces. The vaccine: when your system turns five and someone proposes a rewrite, you'll have Spolsky's argument and a concrete alternative. And the criterion: frequency times pain decides where to invest refactoring in any code, new or old.

Shouldn't the characterization use a real test framework? Inside the new slice, yes, and chapter 17 already did that. To fence the legacy from the outside, the table-plus-loop has an advantage no framework can match: it runs in the legacy's own environment, no matter how hostile. If your eight-year-old PHP runs on a server that won't accept Composer, the characterization test runs there just the same.

Quick tip: before writing the first characterization, run `git log --since="3 months ago" --name-only` and count commits per directory. Ten minutes of shell and you have the frequency column for your own system's slice board, with real numbers for the meeting where someone is about to propose the full rewrite.

Quick reference

Step	What it does
0. Choose the slice	commits this quarter times pain
1. Characterize	freezes what the legacy DOES, bugs included
2. Use cases	rule becomes a pure function: data in, Result out
3. Adapter	legacy behind the interface; exception becomes Failure
4. View + orchestrator	event in, state out
Don't migrate	stable, end of life, purchase, full rewrite

Step	Done criterion
0. Choose the slice	a number defends the choice
1. Characterize	characterization is green against the

	legacy
2. Use cases	rule tests with literals, no test double
3. Adapter	new slice never sees a throw from the legacy
4. View + orchestrator	characterization green, identical output
Don't migrate	decision written down with the reason

Exercises

1. This chapter's characterization fenced the tab's closing. Write the cases that fence the legacy's other public function, `calculate_total`, straight against the floats it returns. Watch table 7's case: your test's expected value is the double the function returns today, not the \$16.90 from bakery arithmetic. If your new case exposes one more lost cent on another tab, even better: freeze that one too.
2. Strangle the inventory slice on your own, with the four steps. Before you start, reread the choosing section's board: inventory has 6 commits this quarter and the pain is a manual count at closing time. Finish the exercise by writing down whether this migration should happen at all, and which of the four "when NOT to migrate" cases it touches. Doing the exercise and concluding it shouldn't have been done is the right answer; knowing how to run the migration and knowing how to refuse it are the same muscle.

Tip 20: migrate what changes, fence what doesn't. Characterization is the fence; strangling is the change; the frequency-times-pain board says which of the two each piece deserves.

Next chapter: the tab slice is next in the migration queue, and you already know its shape by heart. What if the one writing the next slice isn't you, but a language model?

FOCUS + AI: The Duo That Scales

In this chapter, you'll:

- write an architectural prompt for a feature of your own, with folder structure, pasted contracts, and explicit bans;
- review a generated diff with the slice checklist and name the chapter 19 card each violation breaks;
- say which FOCUS piece neutralizes which GitClear number, and by what mechanism.

Chapter 3 opened a debt. It measured code degrading in the AI era, pointed at the three guard-rails that would catch it, and promised that FOCUS as a whole would close the argument here. Time's up. You'll watch the same Rosie's Coffee Shop feature come out of a model four different ways, count the differences instead of reaching for adjectives, and leave with a prompt and a checklist ready to use tomorrow.

The idea holding up this chapter is the narrow search space, and it's already been yours since chapter 3: the fewer correct ways there are to complete the code, the fewer chances the generator has of picking the wrong one. What was missing was the other half. Narrowing the search space has two dials, not one. One tells the generator **WHAT** to build, and it's called the specification; the other tells it **WHERE** each decision lives, and it's this book's architecture. Turn only the first and you get the right thing in a

tangle. Turn only the second and you get the wrong thing, neatly arranged. And turn neither and you're doing vibe coding (Karpathy, February 2025), with the results you already know.

The one-sentence request

Before any technique, the disaster. I asked Claude Opus 4.8, in July 2026, to implement Rosie's Coffee Shop's tab split in Dart. The model generated the code; I asked and read what came back. Each run went in a clean context, with a disposable configuration profile, no file from this project nearby, and the word FOCUS never appearing anywhere.

One caveat before the numbers, and it holds for the whole chapter. One run per condition isn't a controlled study. A model's output isn't deterministic, and if you repeat the experiment you'll get a different result, probably better in one spot and worse in another. The studies that back the argument are chapter 3's: GitClear, DORA, and arXiv. What follows is my own observation, with the model and the date on its badge.

The vague request, in full: "People at Rosie's are always asking to split the table's bill. Can you build that in Dart for me?" One sentence. It's the request Rosie herself would make at the counter, and it's also the request plenty of people paste into the chat at eleven at night.

What came back was 530 lines of code, not counting comments and blank lines. An interactive terminal menu, three split modes (even, by consumption, by item), a configurable service charge, a discount field, and a settle-up algorithm for when one person pays for everyone and the rest square up later. None of it was asked for. The request had one idea; the output has seven.

A quick vocabulary note, the same diff chapter 11 already defined: from here on, the diff is always whatever the model handed back. Three excerpts from the output, copied as they came back, with the location comments added by me:

Dart

```
// splitter.dart: business refusal thrown as an exception

void addPerson(String name) {

    final n = name.trim();

    if (n.isEmpty) throw ArgumentError('Empty name.');
```

```
    if (people.any((p) => p.toLowerCase() == n.toLowerCase())) {

        throw ArgumentError('$n is already at the table.');
```

```
    }

    people.add(n);

}

// main.dart: the funnel that catches all three exceptions and moves on

void _attempt(void Function() action) {
```

```
try {  
  
    action();  
  
} on FormatException catch (e) {  
  
    print(' x ${e.message}');  
  
} on ArgumentError catch (e) {  
  
    print(' x ${e.message}');  
  
} on StateError catch (e) {  
  
    print(' x ${e.message}');  
  
}  
  
}  
  
  
// test.dart: the same five-line block, repeated three times  
  
check('rejects negative price', () {  
  
    try {  
  
        Item('x', -1);  
  
        return false;  
    }  
}
```

```
    } on ArgumentError {  
  
        return true;  
  
    }  
  
}();
```

Read the first excerpt by what kind of refusal it is. “This person is already at the table” is a business decision, exactly as legitimate as “this tab is already paid,” and the code turns it into an `ArgumentError`, the type Dart reserves for a programmer’s malformed argument. Empty name, negative price, a count under one, a tab with nobody on it: all of it becomes an exception. The second excerpt shows where those exceptions go to die. `_attempt` catches `FormatException`, `ArgumentError`, and `StateError` in the same funnel, prints the message, and hands control back to the menu. A typo and a business refusal leave through the same pipe, with the same face. That’s chapter 19’s Card 5, domain try/catch, and it’s the damage that chapter described as the one the tools commit straight out of the box.

The third excerpt is a different animal. That five-line block shows up three times in a row in the test file, body swapped and frame identical, and a second five-line block shows up twice more further down. GitClear, on the same definition chapter 3 already used, calls any five-or-more-line stretch that reappears after normalizing whitespace and stripping comments a duplicate. That’s two duplicated blocks in a single output. None of chapter 19’s six cards covers duplication, and I’d rather say so than force the fit: duplication isn’t a slice anti-pattern, it’s the metric the vertical slice neutralizes, and it comes back next section with the number attached.

One pain I expected never showed up, and it's worth recording rather than hiding. I expected files touched outside the feature, some utility being born in a shared directory, some global config changed. None of that happened, in any run. The model stayed inside the feature the whole time, and the measure of files touched outside the slice came out zero in every condition. Here, it didn't separate anything.

Of the three error cases the feature needs to handle, the vague request's output handles one. A people count under one has a path, via the exception you just saw. Tab not found and tab already paid don't exist in the code: with no word for "tab" in the request, the model never even invented the concept of a tab stored somewhere. You can't handle the error of a concept that never got born.

And then the experiment proved me wrong

I ran the same condition, still with no structure and no architecture, with a complete statement this time: the three refusal cases named, the integer-cents requirement, and the requirement that the parts sum to exactly the total. Still no folder structure, no prescribed return type, no ban of any kind, and no word FOCUS.

The model got it right. Exhaustive types for the three refusal cases, integer cents from start to finish, zero `try/catch` in the entire code, zero duplicated blocks, a passing test file, 200 lines. It's not the result I expected, and it's the result this chapter publishes. Hiding this, or rerunning the round until a disaster showed up, would be fabricating the same anti-solution chapter 3 accuses AI of fabricating.

What the comparison shows, then, is a single variable: the statement. With a one-sentence request, the generator invented six features nobody wanted and handled one error case out of three. With the request spelled out, it got there without receiving any architecture at all. That's exactly the "spec" half of this chapter's argument, demonstrated by accident and in full: the specification says what, and without it the generator decides what on its own.

That leaves the question that matters. If a good statement already produces good code, what does architecture add? The answer isn't in the first generation, and the round 2 section goes looking for it where it lives, which is the same feature's second change.

One more observation, from a run I threw out for a method flaw. This experiment's first attempt ran in an environment I thought was isolated and wasn't: the generator inherited my own coding instructions left over in the context, and handed back code with a sealed Result, comments in the format I use, and even a final report in my own working style, none of it asked for by the five-line statement. That run was voided for the count. But it's accidental evidence for the thesis: a model with architecture contracts in its context produces structured code without the request asking for it. I found that out by getting the isolation wrong.

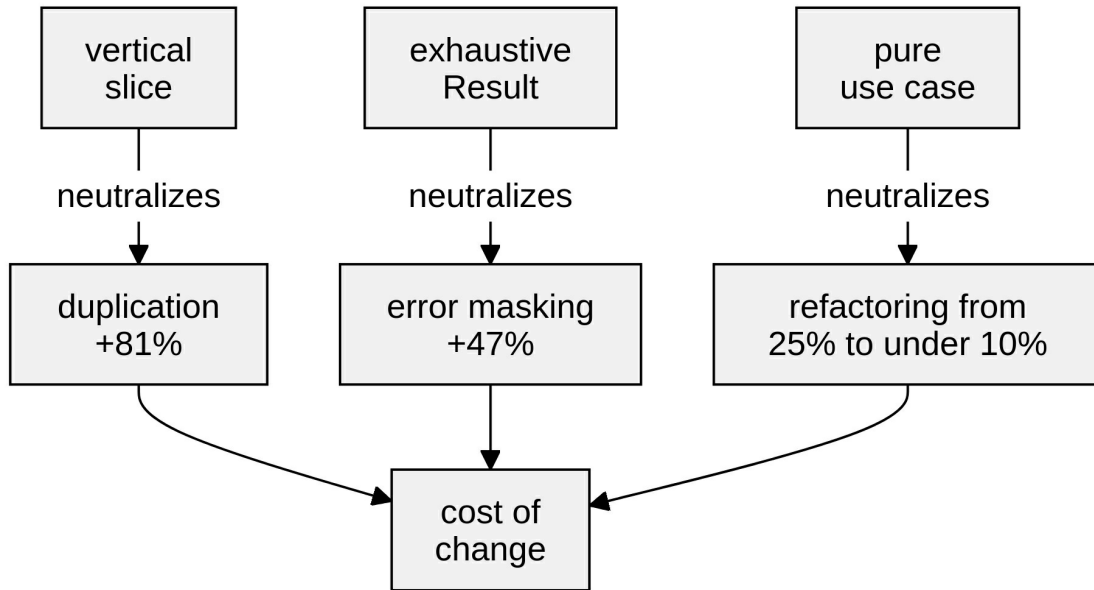
Each piece against a number

The three GitClear numbers chapter 3 asked you to hold on to aren't a portrait of tragedy. Each one has a FOCUS piece that neutralizes it, and the mechanism is concrete in every case, not a hope for discipline.

Code duplication rose 81% relative to the pre-AI era (GitClear, 2024–2026). The piece is chapter 11's vertical slice, and the mechanism is the generator's reach. A model recreates the utility it can't see; that's how chapter 3's loyalty rule got its second copy. When the whole feature fits in one directory, the context you paste is the slice, and whatever already exists inside it is in plain sight, so recreating it stops being the path of least resistance. Duplication between different slices still exists, and chapter 19 already signed that lease with its eyes open.

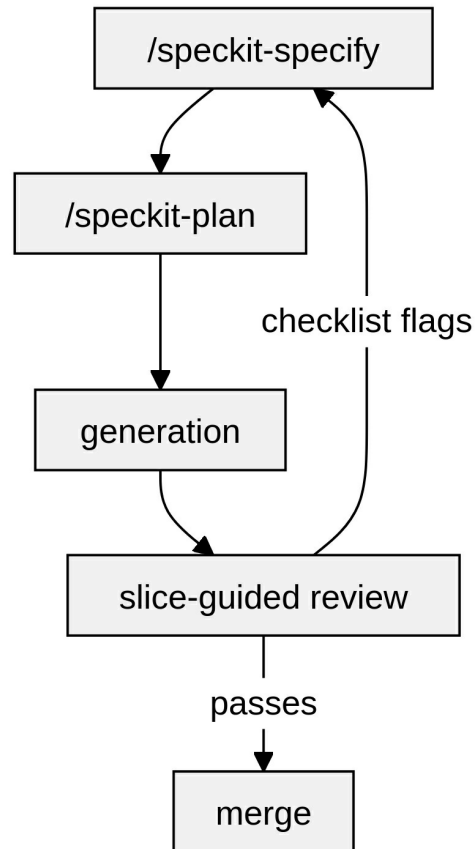
Error masking rose 47% (GitClear, 2026). The piece is the exhaustive Result from chapters 8 and 15, and the mechanism is the compiler. A `switch` over a sealed family doesn't compile with a case missing, and a generator wanting to swallow the failure would have to write, in visible code, the line that swallows it. Ignoring it is still possible; ignoring it silently isn't. That's the difference between a reviewable decision and an invisible omission.

Refactoring dropped from about 25% of changed lines in 2021 to under 10% in 2024 (GitClear, 2024). The piece is chapter 14's pure use case, and the mechanism is the test with no infrastructure. Nobody refactors what they can't tell is broken. A rule that lives in a pure function has a test that runs in milliseconds, no database, no network, and no screen, and the cost of touching it drops to the point where touching it is worth it. A rule scattered across four files with I/O in the middle has the opposite cost, and what happens to it is new code piling on top, which is precisely what the number measures.



Outside GitClear the direction repeats, and it's what convinces me the numbers aren't an artifact of one methodology alone: the 2024 DORA research and the arXiv 2409.19182 study measured delivery speed climbing while stability and maintainability fall, when nothing constrains the shape of what the generator produces. Three independent sources, same direction. Their target is never the tool; it's the terrain.

The spec says what, the architecture says where



The flow above is Spec Kit's, and it fits on half a page. `/speckit-specify` takes the request in plain language and produces a specification: what the feature does, for whom, with which error cases and which acceptance criteria. No line of code shows up at this stage, and that's on purpose: the specification is where questions are still cheap to answer. `/speckit-plan` takes the finished specification and decides how to implement it in this project, with this architecture, in these languages. Only then comes generation. And the arrow that matters most is the one going back: when review flags something, what changes first is the specification, not the generated code. Fixing only the code lets the next generation repeat the same mistake, because its source is still sitting there.

What you just read is the skeleton. Each of those stages has its own rules, its own pitfalls, and a way of going wrong that only shows up after the third feature; to follow this chapter, the skeleton is enough. That's what *Spec Driven Development* (2026, <https://books.kodel.com.br/en/books/sdd>) covers, where I teach you to run the flow I only use here, and you do not need to have read it to go on from here.

Now the argument itself, with both sides failing on their own. Spec without architecture you've already watched work, and it's the complete-statement case from the previous section: the right thing, with all three cases handled: everything inside one file that runs the calculation, holds the data, and prints the output. The program is correct. The question "where do I go when the rule changes" has exactly one answer, and it's "somewhere in that file." Architecture without spec is the reverse picture, and I've seen it happen more times than I'd like: the generator gets the four folders, the contracts, the bans, and hands back a textbook slice that implements a feature nobody asked for, with split-by-consumption and a service charge each neatly tucked into its own file. Every piece in the right place, and the problem it solves is the wrong one.

The pair works because the two dials constrain different things. The specification cuts down the space of WHAT can be built; the architecture cuts down the space of WHERE each decision can live. Turning only one leaves the other axis free, and the free axis is where the generator improvises.

Anatomy of the architectural prompt

An **architectural prompt** is the prompt that names your architecture's layers, types, and contracts, pastes those contracts into the request itself as code, and lists what's off-limits. It

doesn't describe the implementation. It describes the box the implementation has to fit inside.

This is the prompt that generated round 2, in full, unedited and with no extra context. It went to the model exactly as it appears here:

```
Implement Rosie's Coffee Shop's "split tab" feature, in Dart.
```

```
What the feature does: the cashier gives the table number for an open
tab and the number of people; the system returns how much each person
pays. Values are integers in cents, never floating point, and the
parts must sum to exactly the tab's total.
```

```
Refusal cases the business already knows about: a people count under
1, tab not found, tab already paid.
```

```
## Where each thing lives
```

```
The feature is a vertical slice. Create exactly these folders and
nothing outside them:
```

```
features/tab/
  view/          what the person sees; decides nothing
  orchestrator/  receives the event, calls the use case and
                  repository, publishes state
  usecases/      the business rule, as a pure function
  data/          data access; knows about the outside world
```

```
## Contracts that already exist, use these, don't invent others
```

```
// infrastructure failure, never a business refusal
enum Failure { noConnection, unavailable, notAuthorized }
```

```
// what the repository returns
sealed class LookupResult {}
class TabFound extends LookupResult {
  TabFound(this.tab);
  final Tab tab;
}
class TabNotFound extends LookupResult {}
class InfraFailure extends LookupResult {
  InfraFailure(this.failure);
  final Failure failure;
}
```

```

}

// the event the view fires
class SplitTab {
  const SplitTab({required this.table, required this.people});
  final int table;
  final int people;
}

// the state the view receives
sealed class TabState {}
class Loading extends TabState {}
class SplitReady extends TabState {
  SplitReady(this.data);
  final SplitData data;
}
class SplitRejected extends TabState {
  SplitRejected(this.reason);
  final String reason;
}

```

Bans

- Don't create or change any file outside features/tab/.
- Don't use try/catch for business refusal. Business refusal is a return value, with its own type, and the caller decides with an exhaustive switch.
- The use case doesn't receive a repository, doesn't do I/O, and doesn't import anything from data/. Data comes in as an argument, the result goes out as a return value.
- The orchestrator doesn't calculate any business logic. It calls the use case.
- The view doesn't decide anything. It fires the event and renders the state.
- Don't create an interface with a single implementation just for ceremony.

Definition of done

The code compiles with Dart 3 and runs. A demo main exercises all four paths: a successful split with a remainder (\$100.00 among 3 people should give 3334, 3333, 3333 cents), an invalid people count, tab not found, and tab already paid.

One caveat before walking through the parts, because it jumps out at anyone who read ch. 11. The prompt above is from July 2026 and asks for four technical-role folders inside the slice. FOCUS doesn't ask for that today: the slice is flat, and what grows inside it is a sub-feature, not a drawer. The prompt is printed as it was handed over that day because it's what produced the measurements in this section, and rewriting it here would mean showing an input nobody ran. If you're going to use this prompt, swap the second part for the flat tree from ch. 11; the rest stands as is, and what the section measures still holds, because what was being measured is the effect of constraining structure, not the specific folder layout.

Five parts, and it's worth walking through them one by one, because each one cuts a different axis of the search space.

The first part is the business goal, and it's the specification in miniature: what the feature does, with what input, with what result, and the integer-cents requirement chapter 20 taught me to never leave implicit. The three refusal cases come named. It's the same information that made the difference between the two no-architecture runs, and it's still just as necessary here: no structural ban fixes a statement that doesn't say what the business refuses.

The second part is the folder structure, written as a tree, with one sentence per folder stating its job. Notice the sentence describes responsibility, not content. "what the person sees; decides nothing" is more restrictive than a list of files, because it holds for files that don't exist yet.

The third part is the one most people forget: the contracts pasted as code, not described in prose. Describing a type in prose leaves the generator free to reinvent it under another name, another shape, and another semantics, and you get back a generic `Result<T, E>` where there should be a `LookupResult` with three variants. Pasted,

the type is a hard constraint: the model continues the text you started. It's the same reason `SplitData` shows up in the state contract without being defined in the prompt; the generator has to produce it under that name for the rest to fit.

The fourth part is the bans, and they're the direct translation of chapter 19's cards into the language of the request. No file outside the slice closes Card 6. No try/catch for business refusal closes Card 5. A use case that doesn't receive a repository closes Card 2, an orchestrator that doesn't calculate closes Card 1, and the ban on a single-implementation interface closes Card 4. The bans are negative on purpose. They say where you can't go, and leave the how up to whoever generates.

The fifth part is the definition of done, with the canonical output spelled out: \$100.00 among 3 people gives 3334, 3333, 3333. A verifiable criterion in the prompt is worth more than three paragraphs of desired quality, because the generator can check it on its own before handing you the result.

Try it: copy the prompt above, swap in your own feature and contracts, and send it to the model you use. Then count how many of the five parts you'd have written without this list. My bet is on the first and the fourth; the third is the one that usually gets left out, and it's the one that holds the result together the most.

The same feature under the prompt

The previous section's prompt went to a clean context byte for byte, in the same isolation as the other runs. It needed no revision: the first output already brought the slice's files in the

four folders and compiled, which was the agreed criterion. What came back was 198 lines of code across six files, and the two listings below are what the model handed back, uncurated by me.

The listings come out in Dart only. The prompt named Dart, so that's the run that existed; printing the same slice with the TypeScript symbol stacked next to it would claim the model produced this in both languages, and nobody ran the second one. The equivalent slice in all ten official languages is in the Try It at the end of the section.

The use case, in full:

Dart

```
import 'tab.dart';

/// Business refusal is a return value, not an exception. Whoever calls

/// decides with an exhaustive switch.

sealed class SplitResult {}

class SplitCalculated extends SplitResult {

  SplitCalculated(this.data);

  final SplitData data;

}
```

```
class InvalidPeopleCount extends SplitResult {
```

```
    InvalidPeopleCount(this.people);
```

```
    final int people;
```

```
}
```

```
class TabAlreadyPaid extends SplitResult {
```

```
    TabAlreadyPaid(this.table);
```

```
    final int table;
```

```
}
```

```
/// Pure function: data comes in as an argument, the result goes out as
```

```
/// a return value. No repository, no I/O, no import from data/.
```

```
///
```

```
/// The remainder is handed out one unit at a time to the first people
```

```
/// in line, so the parts sum to exactly the total.
```

```
SplitResult splitTab({
```



```
required Tab tab,

required int people,

}) {

    if (people < 1) {

        return InvalidPeopleCount(people);

    }

    if (tab.paid) {

        return TabAlreadyPaid(tab.table);

    }

    final base = tab.totalInCents ~/ people;

    final remainder = tab.totalInCents % people;

    final parts = List<int>.generate(

        people,
```

```

        (index) => index < remainder ? base + 1 : base,

    );

    return SplitCalculated(

        SplitData(

            table: tab.table,

            totalInCents: tab.totalInCents,

            partsInCents: parts,

        ),

    );

}

```

The file imports exactly one thing, `tab.dart`, the folder neighbor holding the data types. No import from `data/`, no repository in the signature, no I/O call. Both business refusals come out as variants of the return; each one carries the data the screen will need for its message: `InvalidPeopleCount` carries the number given, `TabAlreadyPaid` carries the table number. The doc comments came from the model too, and they repeat the prompt's bans almost

word for word, which is a cheap and useful signal: the generator recorded the constraint in the artifact, and whoever reads the file next gets the constraint along with it.

The orchestrator, without the event and state types the prompt already pasted:

Dart

```
/// Receives the event, calls the repository and the use case, publishes  
  
/// state. Calculates no business logic: translates results into state.  
  
class TabOrchestrator {  
  
  TabOrchestrator({  
  
    required TabRepository repository,  
  
    required void Function(TabState) publish,  
  
  }) : _repository = repository,  
  
       _publish = publish;  
  
  final TabRepository _repository;  
  
  final void Function(TabState) _publish;
```

```
Future<void> onSplitTab(SplitTab event) async {

    _publish>Loading());

    final lookup = await _repository.findTab(event.table);

    switch (lookup) {

        case TabNotFound():

            _publish(SplitRejected('Table ${event.table}: no tab found.'));

        case InfraFailure(:final failure):

            _publish(SplitRejected(_describeFailure(failure)));

        case TabFound(:final tab):

            _emitSplit(tab, event.people);

    }

}
```

```
void _emitSplit(Tab tab, int people) {

    final result = splitTab(tab: tab, people: people);

    switch (result) {

        case SplitCalculated(:final data):

            _publish(SplitReady(data));

        case InvalidPeopleCount(people: final given):

            _publish(SplitRejected(

                'Invalid people count: $given. Must be at least 1.',

            ));

        case TabAlreadyPaid(:final table):

            _publish(SplitRejected('Table $table: tab already paid.'));

    }

}
```

}

Two exhaustive switches, one over the repository's result and one over the use case's result, and no arithmetic in between. The three error cases show up in two different places, and that's design, not carelessness: invalid people count and tab already paid are refusals the business rule knows about, so they come out of the use case; tab not found is missing data, so it comes out of the repository as `TabNotFound`. The orchestrator is where the two families turn into the same thing for the screen, which is a message.

The generator also reported, unprompted, two decisions it had to make on its own. It put the demo `main` in `view/`, because the ban on creating files outside the four folders left no room for the program's composition point; that's my prompt's flaw, not the generator's. And it noted that the `InfraFailure` arm is handled but never exercised, because the demo repository is an in-memory map with no way to go down. It chose to say so rather than invent an artificial failure just to make the case count look tidy.

Now the three measures, counted with the same definition across all three outputs, before any prose:

Measure	Vague statement	Complete statement	Architectural prompt
files touched outside the slice	0	0	0
duplicated blocks of 5+ lines	2	0	0
error cases	1	3	3

handled, out of 3			
domain try/catch	1	0	0
lines of code	530	200	198

The file measure is files outside the slice, not the total file count, on purpose. The output under the architectural prompt has six files against three for the others, because the slice has four folders; counting the total would rank size and call the expected result a defect. Outside the slice, every file touched is coupling the boundary should have blocked, and there the number ranks quality in the same direction across all three outputs. This round it came out zero across the board, so it didn't separate anything.

Look at the table without playing favorites. Columns two and three match on the first four rows. Over a good statement, architecture didn't improve any of the three measures, for the simplest reason there is: they were already on target. Anyone trying to sell architecture with this table is selling what it doesn't show.

The fourth measure: the second change

This book's yardstick has always been a different one, and it's time to use it. Specification and architecture don't pay off on the first draft; they pay off on the same feature's second change, when someone needs to touch what already exists. So I handed all three outputs the same new business request, in the same isolation, with no mention of architecture in any of them: each person pays their own part separately, the cashier marks who's

paid, and the tab only closes once every part is paid. Each base became a repository with an initial commit, and the diff was measured against it.

Measure	Vague	Complete	Architectural
files touched	3 of 3	3 of 3	7 of 7, 1 new
lines added	396	370	330
lines removed	1	46	51
where the new rule lives	+147 in the usual file	+248 in the same file	new file, 61
previous behavior	broken	preserved	preserved

The row that separates the three outputs is the fourth. Under architecture, the new rule was born as its own 61-line file, one pure function next to the others, and the rest of the diff is wiring: the orchestrator gained an arm, the view gained a button. Without architecture, the same rule went in as 248 lines inside the file that already held everything, and that file went on to do one more thing. Both programs work. The difference isn't in working; it's in the answer to "where do I look for this next time," which is the question you'll ask six months from now, probably with Rosie waiting on the phone.

Now the row I won't use. The vague-statement base broke previous behavior, and the other two didn't. It's tempting to say architecture prevented the regression, and it would be false: the complete-statement base, which has no architecture at all, survived exactly as well as the one that does. I chased the hypothesis that the other two had the same defect hidden by a

missing test, gave all three the same isolated edge case (split 3 cents among 4 people and pay part by part), and the defect only exists in the vague base. The defect is real, and it belongs to the vague statement alone. But the variable that produced it is still the statement, not the architecture. This data supports the claim that architecture changes where the change lands; it doesn't support the claim that it prevents regression, and I'd rather publish the smaller, true claim.

One last thing the fourth measure showed, and one I hadn't predicted. The second change's request repeated no contract: it said nothing about Result, about refusal as a value, about pure functions. The output under architecture handed back the new rule with business refusal as a return value, created explicit states for an open tab and a closed tab, and refused to re-split a tab with a part already paid. The contracts pasted into the first prompt kept governing the second generation without anyone repeating them, because they were sitting in the code the generator read before it wrote anything. A contract that lives in the repository doesn't need to be pasted twice.

Try it: open <https://focus.kodel.com.br/en/dart/21-01> (or swap `dart` for `kotlin`, `ts`, `java`, `csharp`, `go`, `php`, `python`, `swift`, or `rust`) and run the full slice, with all four paths. Then delete one variant from the orchestrator's `switch` and watch your language's compiler demand the missing case. In languages with no exhaustiveness check, the same route shows what stands in for the compiler instead, which is chapter 18's discipline.

Slice-guided review

You received a whole slice at once. Six files, 198 lines, all of it plausible. Reading in the order the model wrote it is the worst option available: that order is generation's order, not the system's, and it leads you to judge each file by what it looks like instead of by the place it occupies.

Slice-guided review is walking the diff in FOCUS's order, view, event, orchestrator, repository, and use case, and asking the question that fits at each stop. You don't read files; you follow a piece of data's path from the customer's finger to the database and back. A file can get visited twice, and sometimes it does.

Architectural review checklist is the list of questions you ask at those stops. It isn't new: it's chapter 19's six cards, same names and same order, rewritten as a question you can answer by looking at the diff. Keep the difference between the two lists straight, because it's confusing on first read. The numbers are still the cards' numbers; the order you ask the questions in is the slice's order, and the two don't line up.

The first question you ask before opening a single file, just looking at the diff's list of paths: did a new file show up outside the slice? That's Card 6, premature `shared/`, and it's cheap because the answer is in the file names.

Then start walking. At the orchestrator stop, Card 1: is there business calculation inside the event handler? At the use case stop, Card 2: did the signature pick up a repository dependency? At the repository stop, Card 3: does some type serve the whole coffee shop and speak the database's language?

Card 5, domain try/catch, gets three stops instead of one. At the orchestrator and the use case the question is whether there's a `try/catch` around a legitimate business refusal, and the right

answer is none. At the repository boundary the `try/catch` is legitimate, and the question changes: does it translate the exception into a value, or does it swallow it and move on?

That leaves Card 4, layer for ceremony, which has no stop of its own because it has all of them. A file that only passes things through, an interface with one implementation, a DTO identical to the model, a field-by-field mapper: it's the question you repeat every time you open a new file in the diff, wherever it sits along the path.

Apply it to the vague statement's diff and see what happens. Card 6 answers before you read a single line: no file outside the slice, and that's the answer for every run in this chapter. Card 1 finds no orchestrator to flag, because that output doesn't have one; the rule and the menu live in the same place, which is worse than Card 1's damage and isn't Card 1's damage. Card 5 flags it, and flags it twice: `ArgumentError` for "this person is already at the table" is business refusal turning into an exception, and `_attempt` catching three exception types in the same funnel is Card 5's generic catch in the flesh.

Card 3 is a clean pass, and it matters as much as the flags do. There's no generic repository in that output because there's no repository at all; the data lives in lists inside the tab object. The checklist passes clean, and passing clean is the right answer. A checklist that flags every single item isn't reviewing, it's complaining, and you stop trusting it by the third time.

The case Card 5 catches in any language

Card 5's symptom shows up with a different accent in every language. In Python it takes an almost idiomatic shape, and it's the one that slips past review the most:

 Python

```
class TabRepository:

    def __init__(self, service: TabService) -> None:

        self._service = service

    def find_tab(self, table: int) -> Tab | None:

        tab = None

        try:

            tab = self._service.read(table)

        except Exception:

            pass

        return tab
```

Point this repository at a service that's up and ask for a tab that doesn't exist. Then point it at a service that's down and ask for a tab that does exist. Both calls return `None`. The cashier's screen will say the same thing in both cases, and they're opposite cases:

in the first, the tab doesn't exist and the cashier needs to double-check the number; in the second, the tab exists and it's the system that failed to read it.

Why does the generator prefer this shape? Because the apparent goal of whoever's asking is that the program doesn't crash, and swallowing the exception meets that goal in one line, with no need for the generator to know what the business wants when the service goes down. Handling it for real costs a decision the prompt never gave. And Python has no exhaustive `switch` demanding the missing variant, so nothing in the environment complains; the program runs, the tests pass, the rushed reviewer sees three harmless lines. It's the same mechanism as chapter 18: where the language doesn't stop you, convention has to.

The fix is the one from chapters 8 and 15. The `try/catch` stays where it's legitimate, at the repository boundary, and translates the library's exception exactly once, into a value the rest of the system understands:

Python

```
LookupResult = TabFound | TabNotFound | InfraFailure
```

```
class TabRepository:
```

```
    def __init__(self, service: TabService) -> None:
```

```
        self._service = service
```

```
def find_tab(self, table: int) -> LookupResult:

    try:

        tab = self._service.read(table)

    except ConnectionError:

        return InfraFailure(Failure.NO_CONNECTION)

    if tab is None:

        return TabNotFound(table)

    return TabFound(tab)
```

Three changes, all small. `except Exception` became `except ConnectionError`, because catching everything also catches the `AttributeError` from your own typo. `pass` became a named return, `InfraFailure`, the type from chapters 8 and 15. And missing data stopped being the same thing as a read failure: now they're two distinct variants, and whoever calls has to choose what to do with each one. The two calls from the previous paragraph now print different things, which is the least you'd expect from two different cases.

Two criticisms I take seriously

The first criticism is the strongest one this chapter faces, and it deserves the unvarnished version: models are going to keep improving, generation after generation; two years from now the generator will produce better-structured code than the average team produces today, and this whole apparatus of pasted contracts, bans, and checklists will be dead weight nobody maintains. It's happened before, to other defensive disciplines.

My answer is a dated fact. Between 2024 and 2026 the models improved a great deal, and GitClear's numbers got worse over the same period: the 47% rise in error masking was measured in 2026, over the most capable generation to date, not over the 2022 models. If generator quality solved the problem, the curve would have turned. It didn't turn because the bottleneck was never generation. It's review and maintenance, and both are still done by people, at the same old pace, over a volume of code that keeps growing. A better model produces more plausible code per hour, and plausible code is exactly what eats human review. I'd be glad to be wrong about this, and the test is public: when a GitClear report shows duplication and masking falling with nothing having changed in how repositories are structured, this section goes obsolete, and I'll say so.

The second criticism is more practical and almost always comes from someone who's already tried it: writing the specification costs more than writing the code. For the tab split, this section's architectural prompt has more lines than the use case it produced. For a two-screen feature, the time it takes to write the statement, paste the contracts, and list the bans outruns the time it takes to just write the thing. The criticism is right, and that's exactly why it doesn't get answered with "but it looks nicer."

It gets answered with the fourth measure instead. Specification and architecture don't pay off the first time; they pay off on the same feature's second change, which is when someone needs to

figure out where the rule lives. On the first draft you pay for the prompt and get the same result you'd have gotten writing it straight. On the second, you get a new 61-line file instead of 248 lines stacked into a file that already did something else, and you get the contracts governing the new generation without anyone repeating a thing. My position, spelled out in full: for code that's getting thrown away next week, don't write any specification, and skip this entire chapter with my blessing. For code Rosie is going to run her register on for the next ten years, the second change always shows up.

Pitfalls

Trusting the checklist and giving up on reading the code. The checklist narrows the search; it doesn't replace reading what the diff does. None of the six items asks whether the cents split is right, whether the remainder was distributed, or whether the total adds up. A slice can pass all six items and still charge table four the wrong amount. Use the checklist to clear the known error classes in two minutes, and spend what's left reading the business rule, the one part only you know how to check.

Pasting in too much context and blowing the window. Once chapter 11's context window is blown, something gets dropped, and what gets dropped first is usually the beginning, which is exactly where your contracts were sitting. The temptation is to paste the whole repository so the generator can "understand the system." The result is a prompt where the important ban ends up buried under thirty irrelevant files. The slice is the cut that makes the context fit: the feature's folder, the contracts it uses, and nothing more. If your prompt doesn't fit in a slice, the problem probably isn't the window's size.

The architectural prompt that turns into hand-written code in prose. There's a point where detailing the request stops constraining and starts dictating: when the prompt says which loop to use, how to name the index variable, and in what order to run the checks, you wrote the program in English and asked for a translation. At that point the earlier criticism about cost is dead right, and by a wide margin. The prompt names the contract and the boundary; the implementation is what you're delegating. If the generator solves the problem in a way you wouldn't have chosen, but it respects the contracts and passes the checklist, its way is fine.

Q&A

I ran the prompt here and the model handed back something else. Did I get it wrong? No. A model's output isn't deterministic, and mine wouldn't repeat identically if I ran it again today. The success criterion isn't matching this chapter's listings; it's whether what came back passes the previous section's checklist: files only inside the slice, business refusal as a value, a use case with no repository, an orchestrator with no arithmetic, all three error cases with a path. If it passes, it's good, even under different names.

I don't use AI to write code. Does this chapter apply to me? It does, and outside the experiment sections it barely mentions AI at all. The checklist asks about the code, not about where it came from: a pull request from a human teammate at six on a Friday evening gets reviewed by the same six questions, in the same order, with the same outcome. The architectural prompt becomes the issue's text, which is also a statement written before the implementation.

If the complete statement already produced good code, is architecture optional? For a small feature’s first generation, this chapter’s numbers say yes, and I’m not going to pretend otherwise. The math changes on the second change, and changes again on the same system’s fifth feature, once “where does this live” stops having an obvious answer. Architecture is what keeps the answer obvious after the system grows.

Does the prompt need to paste the contracts every time? On the slice’s first generation, yes. Once the slice exists in the repository, the contracts are in the code the generator reads before it writes, and this chapter’s second change showed they keep holding without being repeated. Paste them again when the slice is new or when the contract has changed.

Quick tip

Keep your architectural prompt in a versioned file next to the feature, not in the chat history. It’s the one artifact that goes stale in silence when the contracts change, and a file in the repository shows up in the diff when someone touches the type it pastes. Bonus: whoever joins the team reads the prompt and understands the slice faster than reading the code.

Quick reference

Symptom in the generated diff	FOCUS piece that blocks it upfront
Business refusal turning into an exception	Sealed Result, refusal is a variant (chapter 8)

Generic <code>catch</code> that logs and moves on	Single translation at the boundary (chapter 15)
Rule recreated, the generator never saw it	Vertical slice: the rule fits in the context
Business arithmetic in the handler	Pure use case, called by the orchestrator (chapter 14)
Use case with a repository in the signature	Data goes in, Result comes out (chapter 14)
New file outside the slice	Banned in the prompt, Card 6 checks it (chapter 11)
Inflated scope, feature nobody asked for	Statement first, with a definition of done

Exercises

1. Write the architectural prompt for the coffee shop's inventory feature (deduct items when the tab closes, warn when an item hits its minimum). Use this section's five parts: business goal with the refusal cases named, folder structure with one sentence per folder, the contracts pasted as code, the bans, and a definition of done with a checkable numeric result. Then count the refusal cases you named: if there are fewer than two, you probably still don't know what the feature does when something goes wrong.
2. Run the tab split on the model you use, with the first section's one-sentence request, and apply the checklist to the output. How many of the six items flagged something? Can you predict, before rerunning it with the complete statement,

which items will stop flagging just because of the statement, and which only stop once you paste the contracts?

Tip 21

Narrow the search space before you ask for the search. The statement cuts down what can be built; the architecture cuts down where each decision can live.

Next chapter: no more isolated slices, and no more one-feature examples. You'll build Rosie's Coffee Shop's entire app, one specification at a time, with everything the previous twenty-one chapters left on the table.

Architecture for humans and for models

Monday morning, somebody's first day on the team. They clone the repository, open the editor, and ask the most ordinary question there is: where does the discount rule live? What happens over the next twenty minutes says more about that project's architecture than any diagram hanging on the wall.

Hold on to that scene, because it repeats several times a day with a different reader. When you hand a task to an assistant and it starts working, the first thing that happens on the other side is the same question, with one difference that changes everything: it can't get up and walk over to the colleague at the next desk. Whatever it manages to retrieve on its own is all it gets.

The pattern nobody designed on purpose

The last few chapters answered that question five times over, each one without knowing about the others.

Chapter 11 said the slice has to fit whole in the context window: to change a business capability, one folder is enough. Chapter 14 put the rule in a single file, with the signature declaring everything it consumes: to know what the policy decides, the use case is enough. Chapter 15 drew the point where reading can stop without owing anything, and promised that point stays put after the database changes. Chapter 9 concentrated in one file the

answer to “which concrete implementations does this program use?” Chapter 10 compressed the whole book into a page that keeps what decides.

Five decisions, five retrieval questions, and not one of them was made with a language model in mind. The axis of change dates to the 1970s. The composition root predates any coding assistant. The single page exists because a tired reader doesn’t reread a chapter. What changed wasn’t the design: it was how many readers depend on it per day.

The name for this

AI-friendly architecture is the design that treats the cost of retrieving context as a design criterion, alongside the criteria the discipline already had. It isn’t a technique you install in a project. It’s the sum of the eight concepts that came into the previous chapters, each one in the place where the idea was already needed for another reason.

Here’s my position, with no middle ground. AI-friendly architecture is not architecture made for AI. Not one line of this book asks you to write worse for people on the generator’s behalf, and the day those two readings genuinely conflict, the human one wins, because that’s the reader who answers for the system at three in the morning. What happened is more modest and more useful: a second reason showed up, a measurable one, for the same choices we already defended on readability grounds. Whoever was measuring the cost of change now measures the cost of retrieval too, and both accounts point the same way.

It’s worth saying what this doesn’t promise. Good architecture doesn’t fix a bad prompt, doesn’t replace review, and doesn’t stop a model from inventing a rule nobody asked for. It does one

thing, and does it well: it shrinks what has to be loaded in order to decide, whoever is doing the deciding.

The question left over

Notice what stayed outside. This book organizes the repository, and organizing the repository determines what exists to be retrieved. The other half is left: given one specific task, who picks what goes into that call's window, in what order, and at what cost? A well-drawn slice makes the choice possible; it doesn't make the choice.

That's another book's question, and the book exists. *Context Engineering* (2026, <https://books.kodel.com.br/en/books/context-engineering>) is about deliberately assembling the information that reaches the model on each call, and you don't need it to finish this one, the same way you didn't need the first volume to get to this page. The bridge is on the record, and nothing more.

What's missing is the part where the arguing stops. Turn the page: chapter 22 builds the whole app, slice by slice, with everything Part III promised.

Build Rosie's App

In this chapter, you'll:

- clone the coffee shop app's repository and run all four slices together, with a file database and a simulated card reader, in a single command;
- point, for every architecture decision in the app, to the file where it lives and the chapter that taught it;
- port a slice to your language and prove the port against the repository's shared case suite.

You saw the four pieces one at a time. The view in chapter 12, the orchestrator in chapter 13, the use case in chapter 14, the repository in chapter 15, each in an example the size of one page. None of those examples had a database, a real screen, or a failure that happened on its own. Now all four sit inside the same app, one that opens, saves, charges, and fails.

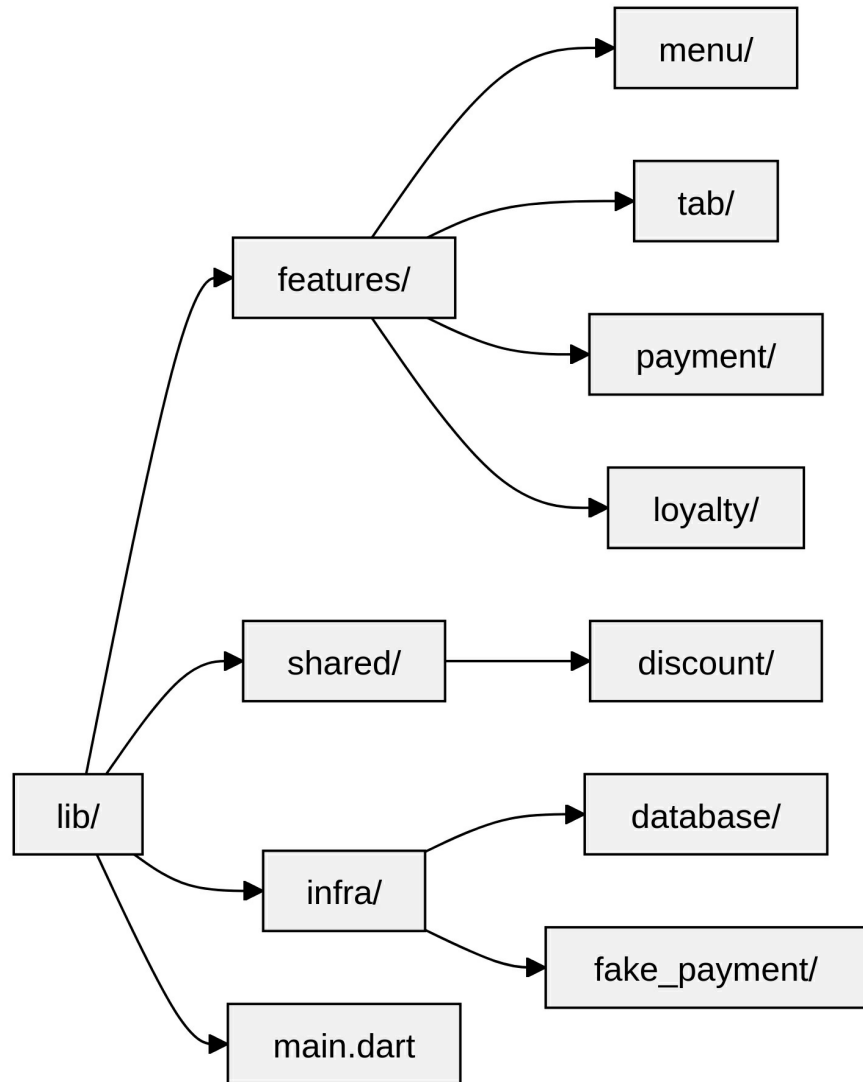
The scaffolding ends here. In the previous chapters, every listing came with a line-by-line explanation, because the concept was new. It no longer is. From this point on, the text gives three things per piece: what the spec asked for, which decision got made and why, and the snippet where the decision lives. The rest of the file sits in the repository, and the path printed before each snippet is the real path over there.

No new term shows up from this paragraph on. Everything the app uses was already defined in an earlier chapter, and every section points to the number. If a name looks new, it isn't: it's a business name from the coffee shop stuck onto a shape you already know.

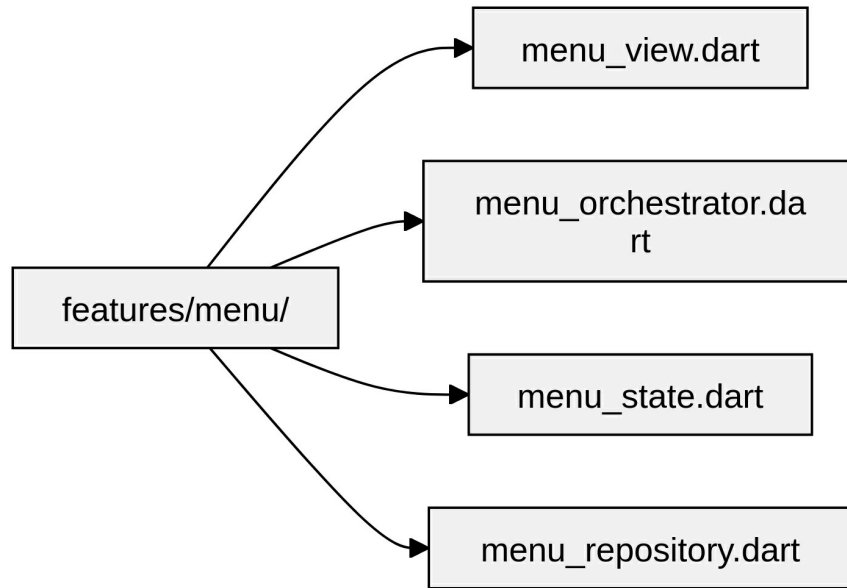
The focus-coffee repository

The app lives at github.com/JCKodel/focus-coffee, public, MIT (Massachusetts Institute of Technology) license, the permissive license named after the university that wrote it. It's chapter 22 in code: every file path printed from here on is a path over there, at tag `coffee-v2`.

The tree has fewer surprises than you'd expect from an app with four features:



Every slice holds the same four files. The menu one, opened up:



The files drawn under `menu/` repeat under the other three slices, with the slice's name up front. It's chapter 11's structure, unadapted: the folder is the feature, and each piece's role lives in the file name, not in a folder.

Folder	What it holds	Chapter
<code>lib/features/menu/</code>	the read-only slice	12, 13, 15
<code>lib/features/tab/</code>	the slice with a lifecycle	13
<code>lib/features/payment/</code>	the slice with the failures	8, 15, 16
<code>lib/features/loyalty/</code>	the slice with the rule	14, 18
<code><slice>/*_view.dart</code>	dispatches events, renders state	12

<code><slice>/*_orchestrator.dart</code>	turns events into state	13
<code><slice>/*_state.dart</code>	the state the view renders	13
<code><slice>/<verb>_<entity>.dart</code>	pure rule, no I/O	7, 14
<code><slice>/*_repository.dart</code>	CRUD and exception translation	15
<code>lib/shared/discount/</code>	the only <code>shared/</code> , by the rule of three	5, 11
<code>lib/infra/</code>	database, fake card reader, formatting	9
<code>lib/main.dart</code>	the composition root, and the only one	9
<code>cases/</code>	the shared test suite, in neutral JSON	22
<code>ports/</code>	ports to other languages	18, 22
<code>docs/</code>	each slice's spec and plan	21

This table is the repository's `README.md`, word for word, so that whoever arrives through the code and whoever arrives through the book read the same map.

The `README.md` also carries the commands, and there are four:

```
flutter pub get

dart run build_runner build --delete-conflicting-outputs

flutter test

flutter run -d linux
```

The first two are setup: the second generates the database code, and without it the project doesn't compile. The two you care about are the last two. `flutter test` runs 85 tests. `flutter run -d linux` opens the window with Rosie's menu loaded from the database.

The platform checked step by step, from a clean clone, is Linux desktop. On macOS or Windows, swap the `-d` target for `macos` or `windows`.

There's no account to create, no API key to get, and no service to stand up. The database is a local file, and the card reader is a simulation whose outcome you program.

Menu: the read-only slice

The spec in `docs/001-menu/` asks for little: the cashier opens the app and sees the menu with price and out-of-stock mark, without typing anything. An out-of-stock item shows up struck through

and can't be added. Nothing else.

A feature like this tends to turn into half a dozen lines thrown at the screen. Here it has three pieces, and the third is the one that usually disappears.

First decision: this slice has no use case. The whole slice is six files, and the listing is the proof:

```
lib/features/menu/  
  menu_item.dart  
  menu_list_view.dart  
  menu_orchestrator.dart  
  menu_repository.dart  
  menu_state.dart  
  menu_view.dart
```

There's no rule file because there's no rule to apply: the slice reads the menu and shows it. The flat slice makes the absence visible for free. In the tree organized by technical role it needed a note, because an empty folder doesn't survive git and the only way to show the gap was deliberate was a file inside it saying so. Here whoever looks for the rule walks six names and sees it isn't there.

Layer for ceremony's sake is chapter 19's entry for exactly this: the layer that exists because the diagram has four boxes, not because anyone needed it. A `FetchMenuUseCase` that just forwards the call to the repository protects nothing, decides nothing, and still costs one more file on every future change. I didn't write that file, and I wouldn't.

Second decision: the exception dies in the repository. The slice's boundary is an interface, and the implementation with the drift package (the Dart database-access generator, unrelated to

the “drift” between code and document from chapter 23) is the only piece that knows SQLite exists. It’s in

```
lib/features/menu/menu_repository.dart :
```



```
/// The slice's boundary. Whoever sits above here doesn't know drift,
```

```
/// doesn't know SQLite, and doesn't know exceptions.
```

```
abstract interface class MenuRepository {
```

```
    Future<MenuLookupResult> fetchMenu();
```

```
}
```

```
class MenuRepositoryImpl implements MenuRepository {
```

```
    const MenuRepositoryImpl(this._database);
```

```
    final CoffeeShopDatabase _database;
```

```
    @override
```

```
    Future<MenuLookupResult> fetchMenu() async {
```

```
try {

    final rows = await _database.select(_database.menuItems).get();

    final items = rows

        .map(

            (row) =>

                MenuItem(row.name, row.priceInCents, outOfStock: row.outOfStock),

        )

        .toList();

    return MenuFound(items);

} on Exception {

    return InfraFailure(Failure.databaseUnavailable);

}

}
```


It's chapter 15 applied at full price. One `try`, one `catch`, one return value. Whoever calls it gets back `MenuFound` or `InfraFailure`, and the compiler charges for both branches.

Third decision: the price turns into text in the orchestrator, never in the view. Chapter 12 calls this renderable state, and the app takes it seriously: the view receives `formattedPrice` as a ready string and doesn't know price is an integer count of cents. The conversion is in `lib/features/menu/menu_orchestrator.dart`:

Dart

```
Future<void> _onLoadMenu(LoadMenu event, Emitter<MenuState> emit) async {

    emit>Loading();

    final result = await _repository.fetchMenu();

    switch (result) {

        case MenuFound(:final items):

            emit(Ready(MenuData(items.map(_render).toList())));

        case InfraFailure(:final failure):
```

```
emit(Failed(_messageFor(failure)));

}

}
```

The formatter itself lives in `lib/infra/money_formatting.dart`, not in `lib/shared/`. The reason shows up in the loyalty section, along with the repository's only `shared/`.

With state ready, the view comes out the size chapter 12 promised. This is the whole file, at `lib/features/menu/menu_view.dart`:

Dart

```
/// Fires the event and draws the state. Doesn't format, doesn't

/// calculate, and doesn't decide anything.

///

/// `onTapItem` arrives from outside because the menu doesn't know

/// about the tab. What wires one slice to the other is the

/// composition root, and that's where the tap turns into `AddItem`.

class MenuView extends StatelessWidget {

  const MenuView({required this.onTapItem, super.key});
```

```
final void Function(String name) onTapItem;

@override

Widget build(BuildContext context) {

  return BlocBuilder<MenuOrchestrator, MenuState>(

    builder: (context, state) {

      return switch (state) {

        Loading() => const Center(child: CircularProgressIndicator()),

        Ready(:final data) => _MenuList(data: data, onTapItem: onTapItem),

        Failed(:final message) => Center(child: Text(message)),

      };

    },

  );

}

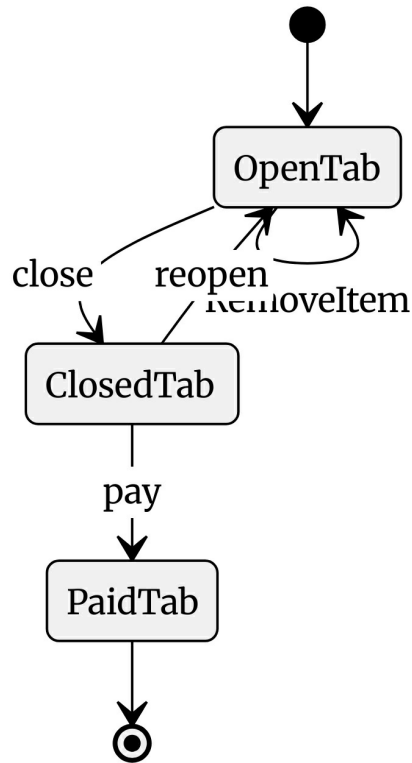
}
```

Three states, three branches, zero `if`. Add a fourth state to the sealed type and this file stops compiling, which is the whole point of chapter 13.

Notice `onTapItem`. Tapping a menu item adds it to the tab, and that's this slice's second decision point: the view could read the tab's orchestrator directly, with a `context.read`, and settle everything in one line. It doesn't, and the reason is chapter 11. The menu slice imports nothing from `features/tab/`, and stays that way even after this connection: the tap climbs up as a function, and whoever wires it to the `AddItem` event is the composition root, in `lib/main.dart`. Two slices that talk through the place where the concrete implementations already meet, which is chapter 9's subject. The day the menu becomes a screen on its own, with no tab beside it, nothing inside the slice changes.

Tab: the slice with a lifecycle

The spec in `docs/002-tab/` asks the cashier to open a table's tab, add menu items to it, remove what got added by mistake, close the bill, and mark it as paid. Three situations, in order. A closed tab still reopens, because the customer asks for one more coffee after asking for the bill; the only transition with no way back is paying.



First decision: the lifecycle is the hierarchy, not a field. The temptation is a `bool paid` or an `enum status` inside one `Tab` class. The app does the opposite, in `lib/features/tab/tab.dart`:

Dart

```
/// The tab's lifecycle is the hierarchy, not a field.

///

/// An open tab still changes and has no closed total. A closed tab and

/// a paid one do. With three types, the compiler stops anyone from

/// reading the total of a tab that's still changing.
```

```
sealed class Tab {

    const Tab(this.table, this.lineItems);

    final int table;

    final List<TabLineItem> lineItems;

}

final class OpenTab extends Tab {

    const OpenTab(super.table, super.lineItems);

    OpenTab withItem(TabLineItem item) {

        return OpenTab(table, [...lineItems, item]);

    }

    OpenTab withoutItem(int index) {

        final remaining = [...lineItems]..removeAt(index);
```

```
    return OpenTab(table, remaining);

}

ClosedTab close() {

    final total = lineItems.fold(0, (sum, item) => sum + item.priceInCents);

    return ClosedTab(table, lineItems, total);

}

}

final class ClosedTab extends Tab {

    const ClosedTab(super.table, super.lineItems, this.totalInCents);

    final int totalInCents;

    PaidTab pay() => PaidTab(table, lineItems, totalInCents);
```

```

    /// A closed tab reopens: the customer ordered one more coffee before

    /// paying. The only transition with no way back is `pay`.

    OpenTab reopen() => OpenTab(table, lineItems);

}

final class PaidTab extends Tab {

    const PaidTab(super.table, super.lineItems, this.totalInCents);

    final int totalInCents;

}

```

Look at where `totalInCents` shows up and where it doesn't. `OpenTab` has no such field, because a tab that still receives items has no closed total. With a `bool paid`, that total would exist the whole time and someone would read the wrong value. With three types, reading the total of an open tab doesn't compile. The transition is also a method, and only on the type that can make it: `close` exists on `OpenTab`, `reopen` and `pay` exist on `ClosedTab`, and `PaidTab` has no transition at all, because there's no leaving it.

Second decision: the use case receives the price, it doesn't fetch it. Here the app diverges from chapter 14, and the divergence has a lesson. Chapter 14 prints `addItemToTab(Tab tab, String item)` with a price table inside the use case file itself, which is perfect for a

self-contained example and impossible in a real app, where the menu lives in the database. The real signature is in

`lib/features/tab/add_item_to_tab.dart` :

Dart

```
/// The rule lives here, and nowhere else.

///

/// A pure function at the top of the file: takes data, returns a

/// result. It doesn't receive a repository, doesn't read the database,

/// doesn't throw an exception. The item's price arrives as an argument

/// because a pure function doesn't consult the menu; the orchestrator

/// does that, before calling it.

Result<Tab> addItemToTab(

  Tab tab,

  String item, {

    required int priceInCents,

    required bool outOfStock,
```

```

    }) {

        if (tab is! OpenTab) {

            return RuleViolated("table ${tab.table}'s tab isn't open");

        }

        if (outOfStock) {

            return RuleViolated("$item is out of stock");

        }

        return Success(tab.withItem(TabLineItem(item, priceInCents)));

    }

```

Duplicating the price table inside the use case would be duplicated knowledge in chapter 5's sense, and giving it a repository would stop it from being pure. What's left is the third way out: the data enters as a parameter. The one who queries the database is the orchestrator, before calling it. I'd rather admit the signature change out loud than pretend chapter 14's example would survive contact with a real database untouched.

Third decision: the orchestrator converts, and only converts. It fetches the menu, calls the use case, publishes the state, and decides nothing. The `Emitter` is called `emit`, as in chapter 13. The snippet is in `lib/features/tab/tab_orchestrator.dart`:

Dart

```
Future<void> _onCloseTab(CloseTab event, Emitter<TabState> emit) async {

    final current = _tab;

    if (current is! OpenTab) {

        emit(Failed("table $_table's tab isn't open"));

        return;

    }

    _tab = current.close();

    emit(Ready(_data()));

    final save = await _tabRepository.save(_tab);
```

```
if (save is InfraFailure) {  
  
    emit(Failed(_messageFor(save.failure)));  
  
}  
  
}
```

Every transition in the diagram is a one-line method call. No math, no `try`, no rule. The orchestrator is plumbing, and that's what it should look like.

Notice what this handler does **not** do: it doesn't mark the tab as paid. Closing and paying are two transitions in the diagram, and two events in the code, because a card reader that can decline sits between one and the other. The payment slice does the paying, and the tab only receives `ConfirmPayment` after the charge comes back approved. If the card reader declines, the tab stays closed and the bill is still open, which is what happens at the counter.

That separation wasn't there either. The first version closed and paid on the same line, and the app looked like it worked: the button disabled and the tab went to "paid" without a single cent getting charged. I'll come back to that defect in Pitfalls.

Fourth decision: the state carries the reason, not just the number. I learned this one with the app on screen, after everything was green. I added a \$6.00 cheese bread and the tab showed a total of \$5.40, with no word about why. The math was right, it was the 10% loyalty discount from chapter 14. What was wrong was what the orchestrator was publishing.

The use case returns the sealed `DiscountResult`, and each of its variants carries the reason: the discount applied, how many points are missing, which out-of-stock item canceled everything. The method that built the payload received that whole type and collapsed it into a single number. The reason went to the trash before reaching the screen. The view, dumb by design since chapter 12, had no way to explain what it never received.

The rule that stuck applies to any of your slices: if the view needs to explain something to the user, that explanation is part of the state, not a deduction the screen makes. `TabData` gained the subtotal and a ready-made sentence per variant, and the file is

`lib/features/tab/tab_state.dart`:

Dart

```
final List<TabItem> items;

/// The view shows the reopen button when the tab is closed and not

/// yet paid. The decision belongs to the orchestrator, never to the

/// screen.

final bool canReopen;

final String formattedSubtotal;
```

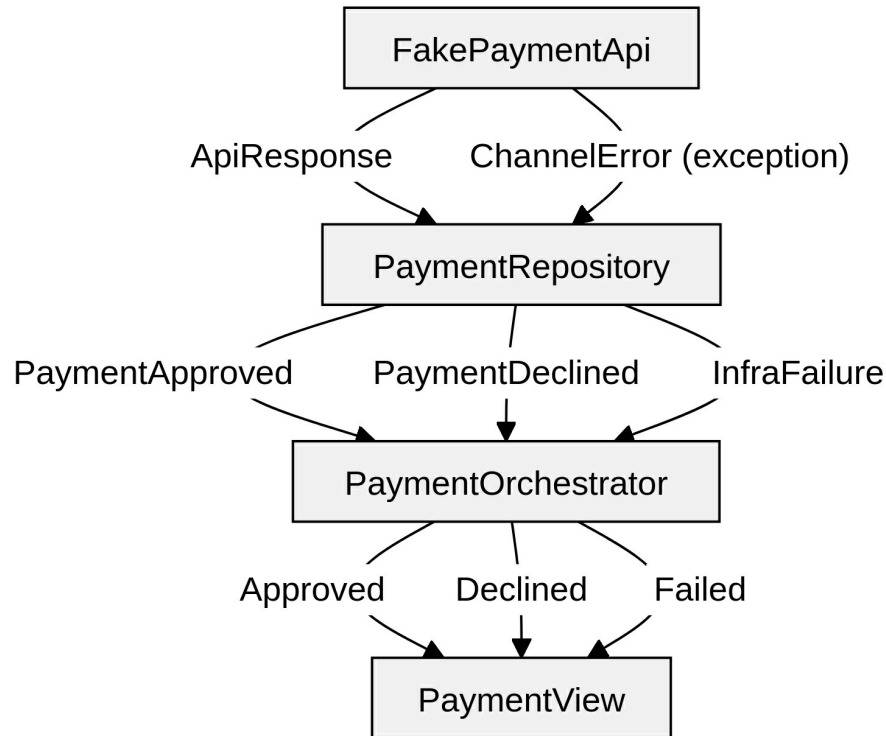
```
/// Why the total differs from the sum of the items, in a ready-made  
  
/// sentence. Empty when total and subtotal match and there's nothing  
  
/// to explain.  
  
final String discountExplanation;  
  
final String formattedTotal;  
  
final bool canPay;
```

Notice what the view does with this: nothing. It prints the sentence if there is one. Who decided the text was the orchestrator, from a type the use case returned, and that's why changing the discount rule doesn't touch a single widget.

Payment: the slice of errors

The spec in [docs/003-payment/](#) asks for three outcomes with three different treatments: approved, declined by the acquirer, and channel down. The same screen splits the bill among the people at the table before charging.

The path a payment takes to the screen is this:



Notice the exception only exists on the first arrow. From the second arrow on, everything is a value, and every translation is one sealed union turning into another.

First decision: the `Result` is born at the boundary, once. The payment's sealed type and the interface are in

`lib/features/payment/payment_repository.dart` :

 **Dart**

```
/// The payment's three outcomes. A declined card is a domain variant,  
  
/// not a `Failure`: the channel worked, the acquirer is the one who  
  
/// said no.  
  
sealed class PaymentResult {}
```

```
final class PaymentApproved extends PaymentResult {
```

```
    PaymentApproved(this.receipt);
```

```
    final String receipt;
```

```
}
```

```
final class PaymentDeclined extends PaymentResult {
```

```
    PaymentDeclined(this.reason);
```

```
    final String reason;
```

```
}
```

```
final class InfraFailure extends PaymentResult {
```

```
    InfraFailure(this.failure);
```

```
    final Failure failure;
```

```
}
```



```
abstract interface class PaymentRepository {  
  
    Future<PaymentResult> charge(int amountInCents);  
  
}
```

The translation happens right below, in the same `lib/features/payment/payment_repository.dart`, and fits in twenty-one lines:



```
/// The boundary. It's here, and only here, that the acquirer's  
  
/// exception turns into a value.  
  
class PaymentRepositoryImpl implements PaymentRepository {  
  
    const PaymentRepositoryImpl(this._api);  
  
    final FakePaymentApi _api;  
  
    @override  
  
    Future<PaymentResult> charge(int amountInCents) async {  
  
        try {
```

```

    final response = await _api.charge(amountInCents);

    if (response.authorized) {

        return PaymentApproved(response.receipt);

    }

    return PaymentDeclined(response.reason);

} on ChannelError catch (error) {

    return InfraFailure(error.failure);

}

}

}

```

Second decision: a declined card isn't a `Failure`. Insufficient funds is a business answer: the card reader worked, the network answered, the acquirer looked at it and said no. Nothing failed. Treating this as a channel failure sends the screen to ask “try again” to a customer who needs a different card. Chapter 8 already split the two in its Rust example, and chapter 15 spends pages taking the mix-up apart.

Third decision, and this is the one I got wrong first. Chapter 15 closes with `enum Failure { noConnection }` and proposes `timedOut` as an exercise. That's what the app was born with, those two variants. Both describe a remote channel, and the app has two channels: the acquirer, which is remote, and the embedded SQLite, which runs in the same process and has no connection to drop.

The menu and the tab read from SQLite. With only two variants available, their repositories returned `Failure.noConnection` whenever a local read failed, and the screen told the cashier there was no connection, on a machine that had never been connected to anything. The enum modeled one channel, and whoever used the other one had to lie.

The lie showed up in full in the payment repository. Since `ChannelError` carried the API's outcome, and the outcome has four values, the exhaustive `switch` demanded a branch for `approved` and another for `declined`, and both ended up at `Failure.noConnection`. Two lines claiming that approved means no connection. They were unreachable, because the fake API only raises `ChannelError` on the two channel outcomes, so nothing broke: 71 green tests and a clean `flutter analyze`. Neither tool catches a lying type, because the type compiles.

The fix has two parts, and the first is in `lib/infra/failure.dart`:

Dart

```
/// Infrastructure conditions, by channel.  
  
///  
  
/// `noConnection` and `timedOut` belong to the remote channel: they
```

```
/// apply to the acquirer, never to the embedded database, which runs

/// in the same process and has no connection to drop. A local read or

/// write that fails is `databaseUnavailable`.

///

/// A declined card does NOT belong here. Decline is the acquirer's

/// business answer, and lives in the payment slice's domain sealed

/// type.

enum Failure { noConnection, timedOut, databaseUnavailable }
```

The second part is in `lib/infra/fake_payment/fake_payment_api.dart`, and it's the one that erased the problem instead of patching it:

Dart

```
/// Channel error raised by the fake API. It's an exception on purpose:

/// the repository exists to translate it into a value, once, at the

/// boundary.

///

/// It carries the `Failure` itself, not the outcome, because only two
```

```

/// of the four outcomes are channel errors. A type that only

/// represents the possible cases spares the repository from deciding

/// what to do with `approved` in a place `approved` never reaches.

class ChannelError implements Exception {

  const ChannelError(this.failure);

  final Failure failure;

  @override

  String toString() => "ChannelError(${failure.name})";

}

```

With `ChannelError` carrying `Failure` instead of an API outcome, the translation method that used to live in the repository disappeared whole, and with it the two lying branches. It's chapter 8's lesson applied one level up: when a type represents only the possible cases, the code written to satisfy impossible ones vanishes.

What's left is the spec's third case, the split tab. It's a pure function called before charging, in

```
lib/features/payment/split_tab_between.dart :
```



```

/// Pure function: data comes in as an argument, a result goes out as
/// the return. No repository, no I/O, no exception.
///
/// The remainder of the division is handed out one unit at a time to
/// the FIRST people: the share at index `i` gets one extra unit if,
/// and only if, `i < remainder`. The sum of the shares closes exactly
/// on the total. Everything in an integer count of cents: floating
/// point doesn't enter here, which is why the same case gives the
/// same result in any language that ports this rule.

```

```
    return TabAlreadyPaid(tab.table);

}

final total = tabTotal(tab);

final base = total ~/ people;

final remainder = total % people;

final shares = List<int>.generate(

    people,

    (index) => index < remainder ? base + 1 : base,

);

return SplitCalculated(

    SplitData(table: tab.table, totalInCents: total, sharesInCents: shares),

);
```

}

Splitting 6615 cents between four people gives 1654, 1654, 1654, and 1653. The first three get the remainder's cent because their index is below 3. The sum closes on 6615. It's this arithmetic that's behind chapter 16's rule for a command with a typed return: the split can be refused for two different reasons, and each refusal has its own name in the return.

Loyalty: the rule, and the first shared/

The spec in `docs/004-loyalty/` brings chapter 14's rule unchanged: one hundred points earn a ten percent discount, an out-of-stock item cancels the whole discount, and the division is truncated integer division. This is the rule the book used to explain pure functions, and it's the same one the app runs in production.

First decision: this rule moved up to `lib/shared/`, and it's the only one that did. The file is `lib/shared/discount/loyalty_discount.dart`:

Dart

```
/// The rule lives here and nowhere else.

///

/// Order: an out-of-stock item cancels first, then eligibility, then

/// the discount itself. No IO, no framework, no exception.

///
```



```
/// The discount uses truncated integer division (~/), never  
  
/// rounding. Nine hundred and ninety-nine cents at ten percent is  
  
/// ninety-nine point nine, and the rule truncates to ninety-nine: the  
  
/// total lands at nine hundred, not eight hundred and ninety-nine.  
  
/// That case is what separates a correct port from one that reached  
  
/// for a floating-point shortcut.
```

```
DiscountResult applyLoyaltyDiscount(  
  
    Tab tab,  
  
    int loyaltyPoints,  
  
) {  
  
    for (final item in tab.lineItems) {  
  
        if (item.outOfStock) {  
  
            return ItemOutOfStock(item.name);  
  
        }  
  
    }  
  
}
```

```

if (loyaltyPoints < 100) {

    return NotEligibleForDiscount(loyaltyPoints, 100);

}

final total = tabTotal(tab);

final discounted = total - total * 10 ~/ 100;

return DiscountApplied(ClosedTab(tab.table, tab.lineItems, discounted));

}

```

Chapters 5 and 11 fix the rule of 3: promote when the third case shows up, not when the second one scares you. The three callers exist, and they have names. The first is the `tab`, which shows the total already discounted, in `lib/features/tab/tab_orchestrator.dart`:

Dart

```

/// The displayed total already carries the loyalty discount. It's
/// the first of the three callers of the rule that lives in
/// `lib/shared/discount/`.

```

```

TabData _data() {

    final gross = tabTotal(_tab);

    final discount = applyLoyaltyDiscount(_tab, _loyaltyPoints);

```

The second is the payment, which needs to charge the right amount, not the storefront value, in

`lib/features/payment/payment_orchestrator.dart` :



```

/// The amount charged is the discounted amount. It's the second of

/// the three callers of the rule that lives in

/// `lib/shared/discount/`.

int _amountToCharge(Tab toCharge) {

    final discount = applyLoyaltyDiscount(toCharge, _loyaltyPoints);

```

The third is the loyalty program screen, which shows the customer how much their points are worth today, in

`lib/features/loyalty/loyalty_orchestrator.dart` :



```

/// Third and last caller of the shared rule. This third caller is

```

```
/// what authorized the promotion to `lib/shared/`; with two callers,  
  
/// it would have stayed inside the tab slice.  
  
void _onOpenProgram(OpenProgram event, Emitter<LoyaltyState> emit) {  
  
    final gross = tabTotal(event.tab);  
  
  
    final result = applyLoyaltyDiscount(event.tab, _loyaltyPoints);  
  
}
```

Second decision: the cost of the promotion is real, and it's declared. Whoever touches the discount percentage touches three slices at once. They're the total the tab shows, the amount the payment charges, and the number the loyalty screen promises. That's `shared/`'s price, and it's expensive. That's why it only shows up after the third case, and never at the second one's scare. With two callers, the rule would have stayed inside the tab slice, and the payment would call it from there.

Third decision: `formatCurrency` didn't move up. It has four callers, more than the discount rule, and it still lives in `lib/infra/money_formatting.dart`. Currency formatting is a presentation convention, not a rule. There's no condition, no refusal, no change request Rosie could make about it. `lib/shared/`, in the sense of chapters 11 and 19, is domain promoted to shared. Mixing the two things in there would weaken the one legitimate `shared/` example the repository has.

The rule and the tab split, added together, fit in one self-contained file, with no database and no screen, and that's how it runs in the 10 official languages.

Try it: open <https://focus.kodel.com.br/en/dart/22-01> and swap `en/dart` for `en/go`, `en/rust`, `en/python`, or any of the book's 10 official languages, whose equivalence table sits in chapter 18. A tab worth 7350 cents with 120 loyalty points becomes 6615, split four ways as `1654, 1654, 1654, 1653` in every one of them. Now swap the integer division for regular division in the language you picked. If a decimal place shows up in the output, the port broke, and chapter 18 explains which column of the table it broke in.

Porting roadmap

This section reads on its own. If you skipped the rest of the chapter to port the app, start here: it says in what order, what changes in each language, and how to know you're done.

Port the four features in this order: **menu**, **tab**, **payment**, **loyalty**. The order isn't arbitrary, and it isn't a code dependency either: no feature imports another out of technical necessity. Each one depends on the previous ones in vocabulary. The menu introduces repository, sealed union, and renderable state with the smallest possible surface. The tab adds a lifecycle on top of that vocabulary. Payment adds error, which is a lifecycle with a bad outcome. Loyalty is the only one that requires the previous three finished, because the three callers of `shared/` live in them.

Chapter 18's equivalence table is the translation map. These are the rows that decide the port the most:

Language	sealed/union	Exhaustiveness
Dart	<code>sealed class</code>	<code>switch</code> expression, guaranteed

Go	none; <code>(T, error)</code>	none
TypeScript	discriminated union	<code>assertNever</code> , opt-in
Rust	algebraic <code>enum</code>	exhaustive <code>match</code> , mandatory

Language	Result versus exceptions
Dart	sealed Result at the boundary; exception in infra
Go	error as a value, native; <code>panic</code> is a bug
TypeScript	exception by default; <code>neverthrow</code> /Either growing
Rust	<code>Result</code> plus <code>?</code> is THE idiom

Three cut-down ports live in `ports/`, one per divergence. They don't reproduce the whole app: each one brings the piece where the language forces the architecture to change shape.

Go

The piece that changes shape is the **use case**. Go has no sealed union, and the return becomes `(T, error)`: exhaustiveness stops being a compiler guarantee and becomes review discipline. The snippet is in `ports/go/main.go`:

```
// splitTab returns (T, error) instead of a sealed union.

//

// The remainder goes to the FIRST shares, one unit at a time: the share
```

```
// at index i gets one extra unit if, and only if, i < remainder.

func splitTab(tab Tab, people int) (SplitTabData, error) {

    if people < 1 {

        return SplitTabData{}, ErrInvalidPeopleCount

    }

    if tab.Paid {

        return SplitTabData{}, ErrTabAlreadyPaid

    }
}
```

Nothing forces the caller to tell `ErrInvalidPeopleCount` apart from `ErrTabAlreadyPaid`, and nothing warns when a third error shows up. In Dart the compiler charges for the new branch. In Go the reviewer is the one who charges, and that's the trade chapter 18 calls disciplinary compensation.

TS TypeScript

The piece that changes shape is the **view**. It's the most replaceable of the four, and the port makes that explicit: the orchestrator publishes state to subscribers and doesn't know who draws it. The snippet is in `ports/typescript/menu.ts`:

```
// The orchestrator publishes state and doesn't know who draws it.
```

```
// Swapping React for Svelte, for Vue, or for nothing only swaps the
```

```
// subscriber.
```

```
export class MenuOrchestrator {
```

```
    private state: MenuState = { type: "loading" };
```

```
    private readonly subscribers: ((s: MenuState) => void)[] = [];
```

```
    private readonly fetchMenu: () => Promise<readonly MenuItem[]>;
```

```
    constructor(fetch: () => Promise<readonly MenuItem[]>) {
```

```
        this.fetchMenu = fetch;
```

```
    }
```

```
    subscribe(subscriber: (s: MenuState) => void): void {
```

```
        this.subscribers.push(subscriber);
```

```
        subscriber(this.state);
```

```
    }
```


Swapping React for Svelte swaps the subscriber and nothing else. That's why the repository's `CONTRIBUTING.md` doesn't require a port to reproduce the screen: demanding the same interface in ten languages would turn the acceptance criteria into a framework fight.

Rust

The piece that changes shape is the **repository**, or rather, the type it returns. Rust already ships `Result` and `enum` with data built into the language, so the manual construction from chapters 8 and 15, which in Dart costs a `sealed class` plus one `final class` per variant, disappears. The snippet is in `ports/rust/src/main.rs`:

```
/// The signature is the language's own Result<T, E>. No hand-written

/// sealed class: the '?' operator and the exhaustive 'match' come

/// built in.

fn split_tab(tab: &Tab, people: i64) -> Result<Vec<i64>, SplitRefusal> {

    if people < 1 {

        return Err(SplitRefusal::InvalidPeopleCount(people));

    }

    if tab.paid {
```

```
return Err(SplitRefusal::TabAlreadyPaid);  
  
}
```

The generic `Result<T, E>` that chapter 21 points to as a symptom of code generated with no architecture is, in Rust, the language's own idiom. The difference is who wrote the error variants: here `SplitRefusal` names the two business refusals, and that's what separates a `Result` with meaning from a `Result<T, String>`.

The port's acceptance criterion is a single one: **the case suite in `cases/` passes whole, value by value and in the printed order.**

The cases live in three JSON files, one per rule, so a partial port can run what it already implemented. JSON because all 10 languages read JSON from their standard library, and the suite exists for whoever ports, not for whoever wrote the original. Passing the suite is a necessary condition for a correct port. It isn't sufficient, and the next section explains why.

The critique: “book examples always work”

I used to be that skeptical reader, and the distrust is fair. A book example has no deadline, no customer complaining, and it usually gets born with the outcome agreed on in advance. Three answers, and none of them is “but this one is different.”

The first is that the repository is open to issues, and the invitation is literal. Found a spot where the architecture gets in the way instead of helping? Open an issue with the file path and the change request that made it awkward. I won’t answer that you didn’t understand FOCUS. The `CONTRIBUTING.md` says how a port gets in and what the suite proves; it doesn’t say the app is good.

The second is naming the weak points here, instead of waiting for you to find them. The app has a single table, hardcoded in `lib/main.dart`, because a table selector wouldn’t teach anything the rest doesn’t already teach. Loyalty points enter as a number in the composition root, with no customer record. The card reader is local and always answers instantly, so nothing in the app exercises a retry, a real timeout, or a pending payment. History is missing too. A paid tab goes to the database and nobody ever reads it back. Each of these absences is a new slice, and none of them would change the shape of the four that exist.

The third is the one that stings. Full FOCUS doesn’t pay off in a throwaway prototype. If what you have is a weekend and the question is whether the idea interests anyone, four layers, sealed types per state, and a case suite are pure cost. Write it all in one file, show it to three people, throw it away. This book’s architecture pays off when the code is going to be changed by someone else, or by you six months from now, who is practically the same person.

And there's an argument worth more than the three. This app didn't work on the first try. The payment section shows the exact spot where the model was wrong, an enum with a single channel, and the damage it caused in two slices that weren't even its own. The defect got past 71 green tests and a clean analyzer. It's printed in the chapter with the fix right beside it, because an example that never fails teaches less than one that fails and shows where.

Pitfalls

Running the app before reading the specs. The repository's four specs in `docs/` say what each slice is supposed to do, and they were committed before its code. Whoever reads only the code discovers the "how" and misses the "what," which is exactly what chapter 21 uses to restrain the code generator. Read `docs/001-menu/spec.md` before opening `lib/features/menu/`.

Testing the pieces and never assembling the toy. This is the most expensive defect this app had, and the most embarrassing to admit in a chapter that promises four slices working together. All four were written. All four had use case, orchestrator, and repository tests, all green. Two of them, payment and loyalty, **weren't on the screen**: the composition root only assembled menu and tab. The pay button wrote "paid" straight to the database, without ever triggering the card reader, so the app charged nothing, and the exhaustive `Result` the payment section describes never ran once while the app was running.

No test could catch this, because testing a piece is exactly testing a piece. `flutter test` claimed every slice worked on its own, and every slice really did work. Nothing claimed the app assembles the four. The fix was a new file, `test/app_test.dart`, whose first test is thirteen lines long: it boots the app and looks for the four views

in the widget tree. At tag `coffee-v2` the file has already grown, because a second test walks the whole counter, but what closed the hole were the first thirteen lines. It also caught, as a bonus, a width overflow on the payment buttons, which nobody would have seen without opening the window.

If your architecture separates the pieces well, and FOCUS does, you get a blind spot exactly where they meet. Write the test that boots the whole system and checks the parts are there. Just one, the dumbest possible one. It doesn't replace the piece tests: it covers the one spot the piece tests, by definition, don't look at.

Trusting a green suite to say the screen opens. When I finished the tab slice, the 71 tests passed and the analyzer didn't complain about anything. I opened the app and the menu showed up on the left, with the tab spinning a progress indicator forever on the right. The orchestrator was born in `Loading` and only registered handlers for the interaction events, so the load event and the line that fires it in `lib/main.dart` were both missing. The menu slice had both, and that's why it worked. No test caught it, and the reason matters: every `blocTest` in that file starts by firing an interaction, and nobody had written the case where the orchestrator just got born and the user hasn't touched anything yet, which is exactly the state it finds the app in when it opens. The rule that stuck: an orchestrator whose initial state is a loading state needs a load event, the matching `..add()` in the composition root, and a test that fires only that event. Without that test, your suite can't tell whether the app opens.

Porting syntax instead of architecture. Translating line by line produces Dart written with Go's words, and the result has the shape of the original with none of its guarantees. Chapter 18 closes with the tip to translate anchor by anchor, and there are three anchors. The signature that returns a Result, the single spot where the exception dies, and the place where the concretes

get built. Reproduce the three in your language, in whatever order your language allows. If your port has `try` in three different files, you ported syntax.

Copying the whole structure into a weekend prototype. Already in the previous section, and I repeat it here because it's the most common mistake for someone who just finished an architecture book. Four layers for a screen you're going to throw away is the same waste chapter 4 calls speculative functionality, just applied to the design.

A port that passes the suite with the layers scrambled. The suite proves behavior, not design. You can pass every case with business rules inside the view and `try / catch` scattered around, and the `CONTRIBUTING.md` says so out loud. Passing isn't the same as getting it right. The port's architecture review is chapter 21's checklist: the same questions you'd ask of AI-generated code apply, without adding or removing a single one, to code ported by hand.

Q&A

Why Flutter, if the book is multi-language? Because the app needed to run on a single machine, with an embedded database and no server, and because chapter 12 already fixed Flutter as Dart's canonical framework. The choice shows up in the `*_view.dart` files and nowhere else: the orchestrator knows `package:bloc`, the use case knows nothing, and the repository knows drift. Swapping Flutter for another screen touches the screen files, a handful per slice.

Isn't the app too small to prove architecture? It's small, and it proves little on its own. What it proves is what a small app can prove: that the four pieces fit together with no ceremony, that the slice with no rule ends up with no use

case, and that a modeling mistake in `lib/infra/failure.dart` leaks into two slices. Architecture proves itself in change, not in size, and that's what the exercises ask for.

Can I use a different database? You can, and the swap stays local. Only `lib/infra/database/` and the files in `<feature>/data/` import drift. Rewrite the implementations of `MenuRepository` and `TabRepository` with whatever you want, keep the interfaces, and nothing above `data/` finds out. If something above breaks, chapter 15's boundary was leaking before the swap.

Does the port need to pass every case, or just the ones for the feature I ported? Just its own. The cases live in three separate files exactly for that reason, and a partial port is welcome in `ports/`. What the `CONTRIBUTING.md` demands is that the file you declared passes whole, value by value and in order.

Quick tip

Before porting, run `flutter test` and read the test names with `flutter test --reporter expanded`. The list of names is the app's executable specification, feature by feature, and it's shorter than the four specs put together. Porting with that list open beside you turns the port into an exercise of making names turn green.

Quick reference

Feature	Concepts it uses
Menu	dumb view, exception-

	translating repository, slice with no use case
Tab	sealed lifecycle, pure use case, rule of 3 on consumption
Payment	Result at the boundary, decline versus channel failure, pure split
Loyalty	pure function, <code>shared/</code> by the rule of 3, equivalences

Feature	Chapters behind it
Menu	12, 13, 15, 19
Tab	8, 13, 14
Payment	7, 8, 15, 16
Loyalty	5, 7, 11, 14, 18

Exercises

1. Add “out of stock” to the menu through the full path: write the change to the `docs/001-menu/` spec first, then the test, then the code. The result precedent already exists: `ItemOutOfStock` has been a `DiscountResult` variant since chapter 14, and the column already sits in the database. Finish with `flutter test` green and the screen showing the item struck through and unclickable.
2. Port the menu slice to your language and make `cases/loyalty-discount.json` pass on it. No view required, if you’d rather skip it: a function that receives the state and returns lines of text is

enough, the way the TypeScript port does it. Open a pull request in `ports/` once it passes.

Tip 22: architecture proves itself on the second change request, not on the first commit.

The first commit is easy with any design. That's why it's no evidence. The second request is the one that reveals where the decisions live: if the change fits inside one slice, the design held; if it opens seven files across four folders, it didn't, and now you know which boundary leaked. You have a project of your own waiting for a request like that. Pick the feature that hurts the most to change, sketch the four pieces for it on paper, and see how many files the next request opens.

Next chapter: the app exists, and so do the specs, and that's where the problem begins. Who guarantees the two still say the same thing six months from now?

Ship Increments Without Chaos

In this chapter, you'll:

- state FOCUS in a hallway conversation, with the four pieces and the chapter number where each one got built;
- say who benefits from it and why, with one reason that covers both humans and AI;
- apply, to your own project, the ruler that keeps the drift between what the code does and what the project says it does in check.

Rosie's app has been running on your machine since chapter 22. You cloned the repository, generated the database, watched the 85 tests pass, and saw the window open with the menu loaded. What's missing isn't code. What's missing is the pocket-sized version of the book, the one that fits in a hallway conversation, and the ruler for tomorrow, when the next change request lands.

This chapter teaches nothing new, on purpose. It hands the book back in pocket size, and to do that it repeats definitions you already read: repeating the definition at the point of use has been this book's decision since the start, because the rule against repeating yourself governs code, not teaching. If a term sounds newly invented, it isn't. Each one carries the number of the chapter that paid for it with code, a test, and an answered critique.

FOCUS in a hallway conversation

Someone asks in the hallway what you've been reading lately. You have thirty seconds. The answer fits in them. FOCUS is a four-piece architecture with flow in one direction only, and every piece in this list carries the number of the chapter that built it, so you can check any sentence of mine against the original. The View fires events and draws the state it receives; it decides nothing (chapter 12). The Orchestrator receives the event, fetches the data, calls the rule, and publishes the next state, with no business rule inside it (chapter 13). Use Cases are the only place for business rules: pure functions that take data and return a Result (chapter 14). Repositories fetch and save, and they're the boundary where the exception exists and turns into a value exactly once (chapter 15). The direction of the flow never reverses, and the one-page table stating what each layer does and what each layer forbids lives in chapter 10; it's still the only page in the book worth memorizing.

Two terms from that conversation deserve a one-sentence definition, because whoever's listening may not have read the book, and because you might be reading this conclusion standing in a bookstore. A **slice** is a complete feature inside a single folder: the view, the orchestrator, the use cases, and the data for the same feature live together, and the folder sets the blast radius of any change (chapter 11). A **Result** is a return value that carries success and every rejection as types the compiler forces you to handle, in place of an exception that crosses layers without warning (chapter 8).

Under the four pieces sit three pillars. Business rules are pure functions: data goes in, a result comes out, no IO in between (chapter 7). Errors are values, and the branch you didn't handle breaks the build instead of breaking production (chapter 8). Code organizes by feature, not by layer, so every change request opens

one folder instead of seven (chapter 11). The practical payoff is cheap testing: each piece gets tested the way it asks to be tested, the use case with no test double at all, the orchestrator by event-to-state flow, the repository against a fake (chapter 17).

What's it for, then? For the business app that's going to be maintained: menu, tab, payment, and loyalty at Rosie's Coffee Shop; sign-ups, invoices, and reports in the system that pays your salary. It's the software that changes every week because the business changes, and whose rule needs a fixed address. And when should you skip it? In the weekend throwaway prototype, where four layers are pure cost: write it all in one file, show it to three people, and throw it away, as the book has already admitted twice (chapters 4 and 22).

Who benefits (and it's a single reason)

Four agents edit or judge the code of a living project: whoever maintains it alone, whoever joins the team mid-story, whoever reviews code they didn't write, and the language model asked to produce part of it. Their question is the same one. It isn't "how do I write this?"; it's "where does this live, and what breaks if I touch it?" That question costs you the size of the search space: how many files could hold the answer, how many places the change could reach.

FOCUS shrinks that space, and it shrinks equally for all four. The rule has a single address, the slice's use case. The side effect has a single boundary, the repository. The contract between the pieces is narrow, and the compiler collects on it: a new state in the sealed union breaks the build of every view that ignores it, a new Result variant breaks every caller that doesn't handle it. For the solo maintainer, that means coming back six months later and knowing where to touch without rereading the project. For

whoever joins mid-story, it means opening the tab folder and understanding the whole tab without opening the payment one. For whoever reviews, it means receiving a diff that fits inside one slice. For the model, it means recovering the context of a single folder and synthesizing inside a space where a good share of the wrong outputs don't even compile.

Notice what didn't change from one sentence to the next: the mechanism. Narrow contracts and isolated slices cut the cost of review and reasoning for any agent editing the code, whether it carries a keyboard or a context window. I stopped separating those two audiences in practice: I review a colleague's code and a model's code with the same slice checklist from chapter 21, and FOCUS is the reason that checklist is a single one. Architecture that's good for AI and architecture that's good for people were never two separate lists of requirements.

This is where the line that started in chapter 1 closes, and the sentence from back there closes whole: architecture lowers the cost of change because it makes the intent of the system recoverable, navigable and predictable for humans and for models. What's left of this chapter is the small version of that sentence, and it fits in a single gesture: if the next increment fits inside one slice, the intent is still in place; if it spreads across half a dozen folders, something stopped being predictable before it got expensive.

The ruler against drift

What's missing is a name for the enemy. The name is **drift**: the gradual gap that opens between what the code does and what the project says it does. It's a reused word, and a warning is due: chapter 22 uses "drift" as the proper name of the Dart package that generates database access. Unrelated. Here the word carries

its ordinary sense of drifting apart, and what drifts apart are the document and the code, one moving away from the other. The spec promises one rule, the code delivers a similar one, the screen explains a third version, and nobody decided that in any meeting. Drift never arrives as an accident. It arrives as a rush: one patch at a time, each one too small to deserve a discussion, until the day the document and the code describe different systems.

This book's anti-drift ruler fits in one sentence: in a FOCUS project with specs, every increment has one place to be born in, a contract the compiler collects on, and a pure test that rejects the wrong rule. When the increment is generated by a model, the sentence holds word for word: the single place becomes an instruction in the prompt, the contract becomes code pasted into the prompt, and the pure test becomes an acceptance criterion that runs in seconds. The evidence lives where it always has. Chapter 3 measures the damage of code generated with no structure: GitClear's reports and the vibe coding Karpathy named are the evidence, and chapter 21 shows the opposite movement: specs and FOCUS narrowing the generator's search space before the search even starts. No number got reprinted on this page, and that's deliberate. A number ages; a chapter with a named source doesn't.

The material proof is public. The repository github.com/JCKodel/focus-coffee is chapter 22 in code: four slices, each one born from a spec committed before its code, with the defects the app had printed in that chapter right next to the fixes. Clone it, run the tests, read a spec, then read the slice it describes. The distance between what the document promises and what the code delivers is the measure that matters, and you measure it yourself, with no need to take my word for it.

Q&A

Does FOCUS work for a small project? It works for a small, living project, and Rosie's Coffee Shop is the proof: four slices, a database in a single file, and a simulated card reader. It doesn't work for a small, dead project, the prototype that exists to answer one question and get thrown away; chapter 4 calls that cost speculative functionality, and I agree with it.

When AI gets better, will architecture still matter? Models got better year over year while this book was being written, and the degradation measured in chapter 3 grew over the same period. A better model searches a bigger space, and faster. What architecture does is shrink the space where the search happens, and that math doesn't change with the quality of the searcher. My bet: the better the generator, the more valuable the contract that decides what it's allowed to generate.

Do I need to adopt all four pieces at once? No. Extract one rule into a pure function that returns a Result (chapter 14). Push the exception to the boundary in the next repository you touch (chapter 15). Group by feature the next time a folder gets born (chapter 11). Each step pays for itself, with no need to wait for the others, and that's exactly how FOCUS was born in my own code: distilled, not decreed.

Quick reference

Situation	Fix
New business rule	pure function returning a Result (ch. 14)
A rejection the screen needs to explain	Result variant (ch. 8)

Database, network, or disk in play	repository; the exception dies there (ch. 15)
An event just left the screen, now what	orchestrator publishes the state (ch. 13)
The screen wants to decide something	it doesn't decide (ch. 12)
Not sure which folder this belongs in	the feature's slice (ch. 11)
The change opens three slices	stop; talk before you code (ch. 11)
Generating the slice with a model	paste the contract and spec into the prompt (ch. 21)
The code contradicts the spec	that's drift; fix the spec before the patch
Legacy code with no tests ahead	characterize, then strangle it (ch. 20)
A throwaway prototype	one file, no layers (chs. 4 and 22)

Exercises

1. Somewhere in your own backlog sits a deferred change request, the one you keep pushing back because you don't know what it breaks. Write its spec in five lines: what the change must do and what it must refuse. Then answer which

slice it belongs in. If the honest answer is “three,” you just found the boundary that leaked, and the exercise paid off more than it would have if the answer had been a single one.

2. Take the oldest slice in one of your own projects and read what its documentation promises, a README, a card, or a comment at the top of the file. Mark every sentence the code no longer keeps. The count is your drift measurement, and it tends to surprise you. Could you bring that count down to zero by touching only the document, without changing a single line of code?

Tip 23

Spec first, slice second, pure test in between: the increment born that way has an address, a contract, and a judge.

The tip describes tomorrow morning’s routine, not a new ceremony. Before you open the editor, write what the change must do and what it must refuse; that’s the spec, even at five lines. Decide which slice the change belongs in; if the answer is “three,” the design is asking for a conversation before the code. Write the rule’s test as a pure function, and only then write the rule, with your own hands or with a model in the editor. The judge is the same one in both cases, and that’s why the routine doesn’t change when the tool does.

This book started with a week spent hunting for a business rule with no address. It ends with the address. What it can’t hand you is the proof: that one is born in your own repository, on the day a change request you would have deferred opens a single folder and closes the same day. When that happens, you won’t need me to know it worked.
