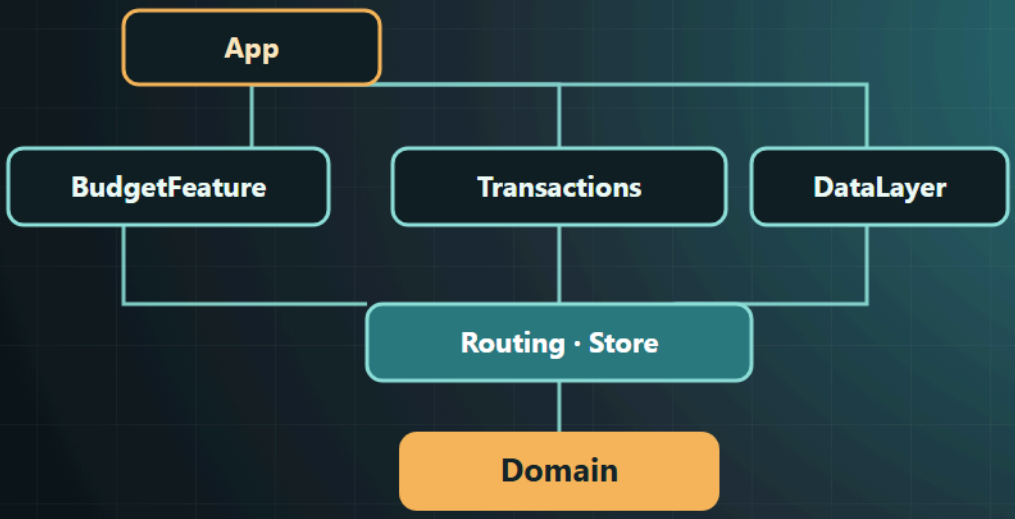
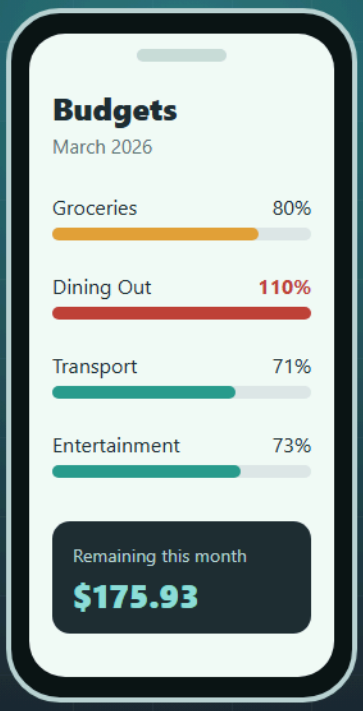





```
func reduce(into state: inout State,
            action: Action) -> Effect
  switch action {
  case let .loaded(rows):
    state.transactions = rows
    return .none
  }
```

```
AppContainer
.live()
.preview()
.test(overrides)
```

A PRACTICAL GUIDE FOR GROWING APPS

iOS App Architecture in Practice

MVVM, unidirectional data flow, modular Swift packages, and dependency injection

Observation · Reducers · Routers · Deep Links · Use Cases
Repositories · SwiftPM Modules · Feature Flags · Testing · Swift 6

STRUCTURE · TESTABILITY · MODULES · TEAMS

THENEXBOOKS

iOS App Architecture in Practice

MVVM, Unidirectional Data Flow, Modular Swift Packages, and Dependency Injection for Apps That Scale

From one tangled view controller to a modular, testable app a whole team can grow

First Edition

THENEXBOOKS

iOS App Architecture in Practice
MVVM, Unidirectional Data Flow, Modular Swift Packages, and Dependency Injection for Apps That Scale

First Edition, 2026

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

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

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Swift, Xcode, iOS, and their frameworks change from release to release; menu names, build settings, compiler diagnostics, and API availability can differ between versions. Always consult Apple's current documentation for the versions you use.

Apple, iOS, iPadOS, macOS, Xcode, Swift, SwiftUI, UIKit, SwiftData, App Store, and SF Symbols are trademarks of Apple Inc., registered in the U.S. and other countries and regions. Other product and library names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Apple Inc. The cover artwork, diagrams, and interface sketches are original and do not reproduce any Apple artwork.

Pennyhaven, the personal budgeting app used as the running example in this book, is fictional, as are Pennyhaven Classic, its team members, accounts, payees, transactions, budgets, web addresses, and all sample data. Domains use the reserved `.example` top-level domain. Nothing in this book is financial advice.

Preface

Most iOS apps do not fail because a single screen is hard to build. They fail slowly. The first version ships quickly, and then every new feature takes a little longer than the last. A change to the budget screen breaks the transaction list. Nobody dares to touch the class that "does the data". Tests are rare because nothing can be created without the whole app behind it. Two developers cannot work on two features without merge conflicts, and the new team member needs weeks before they can find where anything lives.

Architecture is the set of decisions that keeps that from happening — or that lets you climb back out once it has. It is not a framework you install or a diagram you draw once. It is a handful of ideas applied consistently: clear boundaries, one-way data flow, explicit dependencies, a domain layer that knows nothing about screens or databases, and modules that make the right thing easy and the wrong thing hard to compile.

This book teaches those ideas by building one app the way a real team would. We begin with a working but tangled UIKit app, refactor its first screen to MVVM, design a trustworthy domain model, build a small unidirectional store ourselves, add navigation that can be driven by deep links, inject every dependency explicitly, split the code into Swift packages, and finish with the practices that keep a codebase healthy for years: architecture tests, decision records, and incremental migration. At every step we explain the trade-off — what a pattern costs and when you should not use it.

Who This Book Is For

- **Mid-level iOS developers** who are comfortable with Swift and SwiftUI and want to design apps that stay pleasant to change as they grow.
- **Team leads and senior developers** who need shared conventions, a module structure several people can work in at once, and a vocabulary for architecture decisions.
- **Developers maintaining an older UIKit codebase** who want a realistic, step-by-step path to a modern architecture without a rewrite.
- **Developers coming from other platforms** who know architecture concepts and want to see how they map to Swift 6, SwiftUI, Observation, and Swift Package Manager.

We assume you can build a SwiftUI screen, understand optionals, protocols, generics, and closures, and have used `async` / `await`. We do not re-teach the basics; we focus on structure.

What You Will Learn

- The principles behind good app architecture: separation of concerns, the dependency rule, and testability as a design signal.
- How to move from MVC to MVVM with `@Observable` view models, and how to decide where state should live.

- How to model a domain with value types, typed identifiers, and a `Money` type that never loses a cent.
- How unidirectional data flow works by building a small reducer-based store yourself, including effects, cancellation, and composition.
- How to design navigation as state: typed routes, a router, coordinators for multi-step flows, deep links, universal links, and state restoration.
- How to inject dependencies with initializers, the SwiftUI environment, a composition root, and protocol witnesses — and when a container library is worth it.
- How to build a domain layer with use cases, a repository-based data layer over SwiftData and a REST API, and error handling that crosses layers cleanly.
- How to modularize with local Swift packages, design a module graph without cycles, reason about build times, and ship a design system module.
- How to handle configuration, environments, feature flags, logging, and analytics behind small abstractions.
- How to test every layer, enforce boundaries with architecture tests, and use Swift 6 concurrency as part of the design.
- How to refactor a legacy app incrementally, mix UIKit and SwiftUI, and document decisions with ADRs so a team stays aligned.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains why architecture matters, the principles we rely on, the move from MVC to MVVM, domain modeling, and state ownership in SwiftUI. **Part II — Flow and Navigation** (Chapters 6–10) builds a unidirectional store, grows it with effects and composition, and designs navigation with a router, coordinators, deep links, and state restoration. **Part III — Dependencies, Domain, and Data** (Chapters 11–15) covers dependency injection, the domain layer and use cases, repositories and the data layer, and error handling. **Part IV — Modular Apps** (Chapters 16–20) splits Pennyhaven into Swift packages, designs the module graph, examines build times, and adds a design system module, configuration, and feature flags. **Part V — Quality and Cross-Cutting Concerns** (Chapters 21–24) covers logging and analytics abstractions, testing every layer, architecture tests, and concurrency as architecture. **Part VI — Evolving Architecture** (Chapters 25–28) refactors the legacy app, mixes UIKit and SwiftUI, establishes team conventions and decision records, and closes with an end-to-end case study. The appendices provide a cheat sheet, templates, a production architecture checklist, and a glossary.

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

The Running Example: Pennyhaven

Throughout the book we build **Pennyhaven**, a fictional personal budgeting app. Pennyhaven tracks accounts (a checking account, a savings account, and a credit card), categorized transactions, and monthly budgets that turn amber when you are near the limit and red when you go over. Its small sample data set — eighteen March transactions — is small enough to check by hand, so every total in the book can be verified: Dining Out, for example, ends March at $165.00 against a $150.00 budget, $15.00 over. Next to it lives **Pennyhaven Classic**, the app's first UIKit version, built the way many real apps start: one large view controller, a shared singleton, and money stored as `Double`. Classic is our cautionary tale in the early chapters and our refactoring project in Part VI.

By the end, Pennyhaven is a modular app with a thin app target, a pure domain package, a data layer over SwiftData and a REST API, feature packages that never import each other, a hand-built unidirectional store, a router driven by deep links, explicit dependency injection, feature flags, logging and analytics abstractions, and tests at every layer.

Versions and Conventions

The code targets the **Swift 6 language mode** with complete data-race checking, **Xcode 26-era tooling**, and a **deployment target of iOS 18**. A few iOS 26 features appear where they matter, always behind availability checks and marked as such. Apple's tools change quickly; menu names, build settings, and diagnostics may differ slightly in the version you use, so when something is not exactly where the book says, look nearby or check Apple's current documentation. Third-party libraries are discussed as options only; all of Pennyhaven's code uses Apple frameworks and packages we write ourselves.

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

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can cause bugs, data loss, or maintenance problems.

Code appears in `monospace`, with a caption above each listing that shows the file it belongs to, such as `Packages/BudgetFeature/Sources/BudgetFeature/BudgetViewModel.swift`. Long lines are wrapped to fit the page. Results you would see at run time are introduced with "you should see something like", because exact console formatting varies between tool versions. Diagrams and screen sketches are drawn in text so they stay readable in any PDF viewer.

Contents

Preface

PART I — FOUNDATIONS

1 Why Architecture Matters: Goals, Trade-offs, and Meet Pennyhaven

2 Principles That Scale: Separation of Concerns, Boundaries, and Testability

3 From MVC to MVVM: Observable View Models with the Observation Framework

4 Modeling the Domain: Value Types, Typed IDs, and a Money Type You Can Trust

5 State Ownership in SwiftUI: Sources of Truth, View State, and Data Flow

PART II — FLOW AND NAVIGATION

6 Unidirectional Data Flow: Building a Small Store Yourself

7 Reducers, Effects, and Composition: Growing the Store

8 Navigation Architecture: NavigationStack, Typed Routes, and a Router

9 Coordinators and Multi-Step Flows

10 Deep Links, Universal Links, and State Restoration

PART III — DEPENDENCIES, DOMAIN, AND DATA

11 Dependency Injection Fundamentals: Initializers, the Environment, and the Composition Root

12 Advanced Dependency Injection: Protocol Witnesses, Containers, and Overrides

13 The Domain Layer: Use Cases and Business Rules

14 The Repository Pattern and the Data Layer

15 Error Handling Across Layers

PART IV — MODULAR APPS

16 Feature Modules with Swift Package Manager

17 Designing the Module Graph: Boundaries, Access Control, and Avoiding Cycles

18 Modularization and Build Times

19 A Design System Module

20 Configuration, Environments, and Feature Flags

PART V — QUALITY AND CROSS-CUTTING CONCERNS

21 Logging and Analytics Behind Abstractions

22 Testability by Design: Testing Every Layer

23 Architecture Tests and Fitness Functions

24 Concurrency as Architecture: Isolation, Sendable, and Swift 6

PART VI — EVOLVING ARCHITECTURE

25 Refactoring a Legacy App Incrementally

26 UIKit and SwiftUI Together in a Modular App

27 Team Conventions, Code Review, and Architecture Decision Records

28 Case Study: Pennyhaven End to End

APPENDICES

A Architecture Cheat Sheet

B Templates and Reference

C Production Architecture Checklist

D Glossary

PART I

Foundations

Why architecture matters and what it costs, the principles that keep code changeable, the move from a massive view controller to observable view models, a domain model built on value types and a trustworthy Money type, and deciding where state should live in SwiftUI.

1 · Why Architecture Matters
2 · Principles That Scale
3 · From MVC to MVVM
4 · Modeling the Domain
5 · State Ownership in SwiftUI

CHAPTER 1 · PART I — FOUNDATIONS

Why Architecture Matters: Goals, Trade-offs, and Meet Pennyhaven

Before we write a single view model, we need a shared answer to a simple question: what is architecture for? This chapter defines architecture as the set of decisions that are expensive to change, names the goals it serves and the costs it brings, and introduces Pennyhaven, the budgeting app we will grow, test, and modularize throughout the book.

What We Mean by Architecture

Ask ten iOS developers what "architecture" means and you will hear ten answers: MVVM, VIPER, Clean Architecture, "the folder structure", "whatever the lead decided", or "the diagram nobody updates". Each answer contains some truth, but none of them is useful when you are staring at a pull request and trying to decide whether it makes the codebase better or worse.

In this book we use a working definition: **architecture is the set of decisions in an app that are expensive to change later.** Choosing a name for a local variable is cheap to change; you rename it and move on. Choosing how money is represented is expensive: once amounts are stored as `Double` in a database, sent to a server, and formatted in forty places, changing the representation becomes a migration project. Choosing where screen state lives, how navigation is triggered, how a view gets the objects it depends on, and which code is allowed to import which other code are all decisions of the second kind.

That definition has a few consequences that shape the rest of the book:

- **Architecture is about decisions, not diagrams.** A diagram is a picture of some decisions. If the code does not follow the picture, the code is the architecture and the picture is fiction.
- **Patterns are vocabulary, not goals.** MVVM, unidirectional data flow, coordinators, and repositories are names for well-understood decisions. We use them because they let a team talk quickly, not because a pattern is good in itself.
- **Some decisions should be postponed.** If a decision is expensive to change, the best move is sometimes to arrange the code so that it becomes cheap to change, and decide later when you know more. A repository protocol, for example, lets us postpone how much of Pennyhaven's data lives on the server.
- **Architecture is never finished.** An app with three screens and one developer needs different decisions than the same app with thirty screens and eight developers. Good architecture evolves deliberately; bad architecture evolves by accident.

The Goals: What Good Architecture Buys

Because architecture costs effort, we should be precise about what we expect in return. Pennyhaven's architecture is designed to serve six goals. Every chapter refers back to them, and when a pattern does not serve at least one of them, we do not use it.

Goal	What it means in practice	How Pennyhaven shows it
Local change	A change to one feature touches a small, predictable set of files.	Budget rules live in one pure type; the Budgets screen changes without touching Transactions.
Cheap testing	Most logic can be tested in milliseconds without a simulator, network, or database.	Domain tests run against `SampleData`; view models run against in-memory repositories.
Parallel work	Several developers can build features at the same time with few merge conflicts.	One Swift package per feature; features never import each other.
Onboarding	A new developer can find where things live and add a screen in their first weeks.	Naming rules, a fixed folder layout, and decision records.
Replaceable infrastructure	Storage, networking, and analytics can change without rewriting screens.	SwiftData and the REST API live behind repository protocols.
Correctness	Important rules are encoded in types so mistakes do not compile.	`Money` in integer minor units; typed IDs that cannot be mixed up.

Two of these goals deserve a closer look because they are easy to misunderstand.

Local change does not mean "every change is small". Some product changes are genuinely large. It means that the size of a code change should be proportional to the size of the product change. When a designer asks to show the remaining budget in red instead of orange, one view should change. When the product manager asks to carry unspent budget into the next month, the budget rules and the budget screen should change, and nothing else. The failure mode we want to avoid is the small request that somehow requires edits in seven files across four features.

Cheap testing is not the same as "high test coverage". A codebase can have many tests that are slow, flaky, and painful to write because every test has to build half of the app. We want an architecture where the obvious way to write a test is also the fast way. Throughout the book we treat a hard-to-test type as a design smell: if a test needs a running server to check how much you spent on groceries, the grocery calculation is in the wrong place.

The Costs: What Architecture Takes Away

Every architectural pattern in this book has a price, and we will name it each time. In general terms, the costs fall into five groups.

Cost	What it feels like	How we keep it small

Indirection	Following a tap to the code that runs takes several jumps.	Consistent naming, so the next jump is predictable.
Ceremony	A trivial feature needs a protocol, a view model, and a factory.	We skip layers that add nothing (Chapter 13 shows when).
Learning curve	New team members must learn the house style first.	Short conventions, ADRs, and this book's templates.
Build configuration	More packages, manifests, and dependency declarations.	One package per folder and a checked module graph.
Premature abstraction	Flexibility nobody uses, paid for in every change.	Abstract only at real seams: I/O, time, IDs, features.

Indirection is the cost you will notice first. In Pennyhaven Classic, the old version of our app, everything about the budget screen lives in one view controller. It is long and tangled, but you can read it top to bottom. In the new Pennyhaven, the same screen involves a view, a view model, a use case, three repositories, and a calculator. Each piece is small and clear, but understanding the whole requires knowing how the pieces relate. We pay that price because it buys the six goals above, and we reduce it with predictable names: once you know that view models end in `ViewModel` and use cases expose a single `execute` method, the jumps become automatic.

Premature abstraction is the most expensive cost, because it is easy to mistake for good engineering. A protocol with exactly one implementation and no test double is not an abstraction; it is a second place to edit. A generic "base view model" that every screen inherits from couples every screen to decisions made for the first one. We will repeatedly ask, "what change does this abstraction make cheaper?" If the honest answer is "none that we can foresee", we leave it out.

> **WARNING**
>
> Architecture work can become a hobby that replaces shipping. If a refactoring does not make a specific, upcoming change easier, or does not remove a specific, recurring kind of bug, it is probably not worth doing this week. Tie each structural change to a concrete reason you can explain in a pull request description.

Signs an App Needs More Structure

Not every app needs everything in this book. A prototype that will be thrown away in a month needs almost none of it. How do you know when an app has outgrown its structure? These are the symptoms we watch for. None of them is proof on its own, but several together are a strong signal.

- **Fear of specific files.** The team avoids touching a certain class because changes there break unrelated screens.
- **Shotgun surgery.** Small product changes require edits in many files in many folders.

- **Untestable logic.** Important calculations live in view controllers or SwiftUI view bodies, so they can only be checked by running the app and tapping around.
- **Hidden dependencies.** Types reach for singletons such as `DataManager.shared`, so you cannot tell from an initializer what a type needs, and you cannot replace those needs in a test or a preview.
- **Merge conflicts on the same files.** Two developers working on unrelated features keep colliding in a shared file such as a giant model, a central switch statement, or the project file.
- **Slow feedback.** SwiftUI previews take a long time to refresh or fail because they try to build the whole app, and small edits trigger large rebuilds.
- **Inconsistent answers.** The same question, such as "how much did I spend on dining this month?", is computed in two places and the two results disagree.
- **Onboarding by folklore.** New developers learn where things belong only by asking someone, because the code does not make it obvious.

There are also signs of the opposite problem, too much structure for the app's current size: every screen has a protocol for its view model even though nothing ever substitutes it, simple features take days because of boilerplate, and developers copy templates without knowing why each piece exists. Architecture is a dial, not a switch. Part of the skill this book teaches is knowing where to set the dial today and when to turn it.

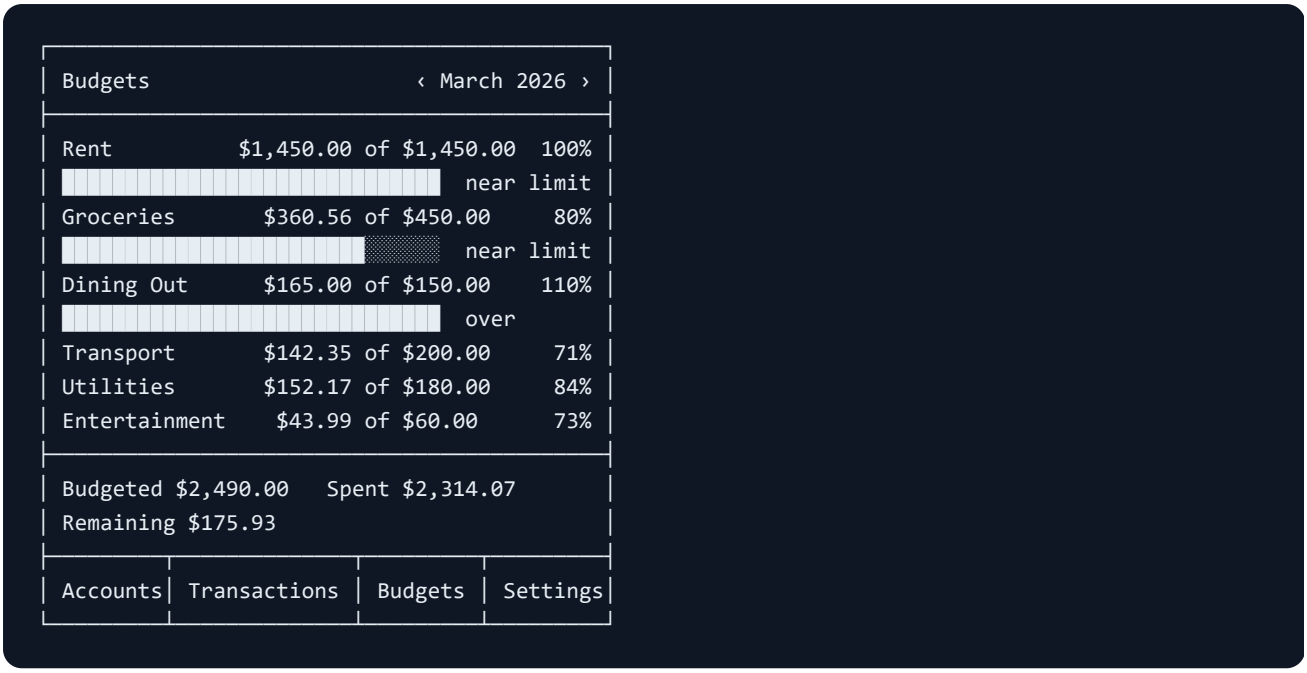
Meet Pennyhaven

To make every idea concrete, we build one app from start to finish: **Pennyhaven**, a fictional personal budgeting app with the bundle identifier `com.example.pennyhaven`. Pennyhaven is deliberately ordinary. It has the same kinds of problems your app has: lists that load from storage, forms with validation, money that must add up exactly, navigation that deep links can drive, and features that several people develop at once.

Pennyhaven has four tabs:

- **Accounts** lists the user's accounts with their current balances and the net worth across all of them. Tapping an account shows its transactions.
- **Transactions** lists the current month's transactions, newest first, with search and a category filter. You can delete a transaction, add one through a multi-step flow, or use a quick-add row.
- **Budgets** shows one row per budgeted category for the month: how much was spent, how much remains, and a status that turns amber near the limit and red when the budget is exceeded. Tapping a row shows that category's transactions.
- **Settings** shows the build environment, the API host, the app version, and, in non-production builds, switches that override feature flags.

Here is a sketch of the Budgets tab as it looks on the book's reference date, March 24, 2026. We draw interfaces as text so that the focus stays on structure rather than pixels.



The Sample Data

Pennyhaven ships with a small, fixed sample data set that we use in previews, tests, and examples. It is small enough that you can check every number by hand, which matters: a book about architecture should not ask you to trust totals you cannot verify. All dates fall in March 2026 at 12:00 UTC, and the reference "today" is March 24, 2026.

Account	Kind	Balance on March 1	After March
Everyday Checking	checking	$2,450.00	$8,547.83
Rainy Day Savings	savings	$8,200.00	$8,200.00
Travel Rewards Card	credit card	-$310.25	-$1,022.15

There are seven categories (Salary, Rent, Groceries, Dining Out, Transport, Utilities, and Entertainment), eighteen March transactions, and six monthly budgets. March brings $7,700.00 of income and $2,314.07 of spending, for a net cash flow of +$5,385.93, and net worth grows from $10,339.75 to $15,725.68. One number will follow us through the whole book: **Dining Out ends March at $165.00 against a $150.00 budget, which is 110 percent and $15.00 over.** It is our standard example of an over-budget category, and it appears in tests, previews, analytics events, and deep links.

The savings account has no transactions at all. That is intentional: an empty account gives us a realistic empty state to design and test. Small details like this make sample data useful. Data that only contains the happy path produces code that only handles the happy path.

Pennyhaven Classic: The Cautionary Tale

Pennyhaven has a history. Its first version, which we call **Pennyhaven Classic**, was built quickly in UIKit, the way many real apps start. It worked, it shipped, and users liked it. Then the team grew, the feature list grew, and every change became slower and riskier. Classic's code lives in the repository under `Legacy/PennyhavenClassic/`, and we use it in two ways: as a cautionary tale in Part I, and as a refactoring project in Part VI.

The heart of Classic is a singleton that owns all data:

Legacy/PennyhavenClassic/DataManager.swift (excerpt)

```
import Foundation

struct LegacyTransaction: Codable {
    var id: String
    var account: String
    var category: String       // the category's display name
    var payee: String
    var amount: Double         // positive = spending in Classic
    var date: Date
}

final class DataManager {
    static let shared = DataManager()

    var transactions: [LegacyTransaction] = []
    var budgets: [String: Double] = [:]    // keyed by category name

    func save() {
        if let data = try? JSONEncoder().encode(transactions) {
            UserDefaults.standard.set(data, forKey: "ph.transactions")
        }
        NotificationCenter.default.post(
            name: Notification.Name("DataManagerDidChange"),
            object: nil
        )
    }
}
```

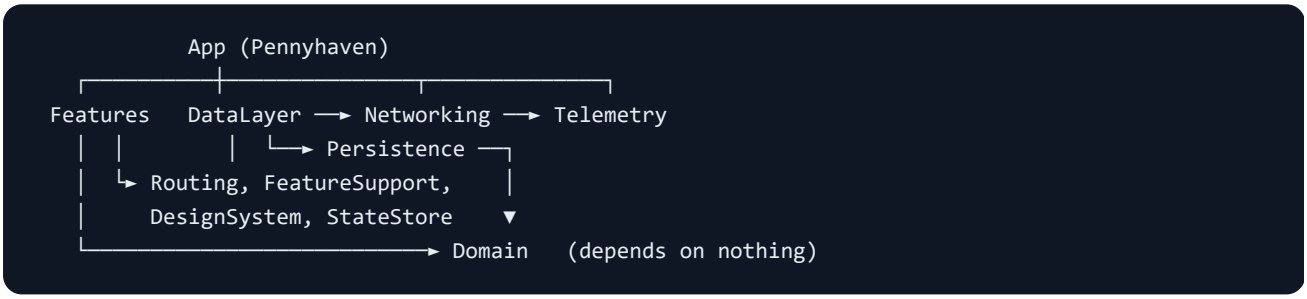
Nothing in this excerpt is shocking, and that is the point. Each line was a reasonable shortcut when it was written. Together they produce the problems listed earlier. Any code anywhere can read and mutate `DataManager.shared.transactions`, so no one knows who changes what. Money is a `Double`, which, as Chapter 4 shows, cannot represent most cent amounts exactly. Categories are identified by their display names, so renaming "Dining" to "Dining Out" silently disconnects every budget. Positive amounts mean spending, which is the opposite of the convention the server uses. A failed encode is swallowed by `try?`. And every screen listens for one global notification and reloads everything when anything changes.

Classic also contains `LegacyBudgetViewController`, a single class that loads data, computes totals, formats currency, configures table cells, and decides navigation. Chapter 2 extracts its first pure function, and Chapter 3 replaces it with a view model. We are not mocking the developers who wrote Classic. Shipping a working app quickly is a legitimate goal, and Classic achieved it. The lesson is that the shortcuts that help a first version become the costs that slow every later one, and that you can pay those costs down gradually instead of rewriting everything.

> **NOTE**
>
> Pennyhaven, Pennyhaven Classic, the team members mentioned in later chapters, the accounts, payees, amounts, and web addresses are all fictional. Domains use the reserved `.example` top-level domain, and nothing in this book is financial advice.

Where Pennyhaven Ends Up

It helps to see the destination before the journey. By the end of the book, Pennyhaven consists of a thin app target and fifteen local Swift packages, with one more added in the final case study. The arrows in the following picture mean "imports" or "depends on".



Three properties of this graph matter more than any individual box.

1. **Everything points toward Domain, and Domain points at nothing.** The domain package contains Pennyhaven's value types, its money type, its business rules, and the protocols that describe what the app needs from storage. It imports only Foundation. That makes it the most stable and the most testable part of the app.
2. **Features never import other features.** The Budgets feature does not know the Transactions feature exists. When the Budgets screen needs to show a transaction, it asks the router, which lives in a small `Routing` package that both features may import.
3. **Only the app target knows the infrastructure.** Features never import `DataLayer`, `Persistence`, or `Networking`. The app target builds the real repositories and hands them to features through initializers. This is what lets us run a feature in a preview with sample data and no database.

The following table summarizes the role of each group of packages. Chapter 17 gives the full list of allowed imports, and Chapter 23 turns that list into an automated check.

Group	Packages	Role
Core	Domain (with DomainFixtures)	Value types, money, rules, repository protocols, use cases, sample data
Data	Persistence, Networking, DataLayer	SwiftData storage, the REST client, repository implementations, sync
Shared UI and flow	DesignSystem, FeatureSupport, Routing, StateStore	Tokens and components, shared feature utilities, navigation, our store
Cross-cutting	Telemetry, Configuration	Logging, analytics, environments, feature flags
Features	AccountsFeature, TransactionsFeature, BudgetFeature, SettingsFeature	Screens, view models, reducers, coordinators
Testing	TestSupport	Recorders, failing repositories, test helpers

The screens use two different state-management styles on purpose. Accounts and Budgets use MVVM with observable view models, which suits screens that load data and react to a few user actions. Transactions uses a small unidirectional store we build ourselves in Chapters 6 and 7, because that screen has many sources of change: loading, searching, filtering, deleting, and a quick-add row. Seeing both styles in one app is more useful than a debate in the abstract.

The Repository Layout and the Skeleton From Day One

Here is the repository as it looks from the first commit of the new Pennyhaven:

```
Pennyhaven/                          repository root (Git)
  Pennyhaven.xcodeproj
  Pennyhaven/                        app target: composition root + root view
    PennyhavenApp.swift
    AppContainer.swift
    RootView.swift
    Config/                          xcconfig files (Chapter 20)
    Info.plist
  Packages/                          one local Swift package per folder
    Domain/  StateStore/  DesignSystem/  Telemetry/  Configuration/
    Networking/  Persistence/  DataLayer/  Routing/  FeatureSupport/
    AccountsFeature/  TransactionsFeature/  BudgetFeature/
    SettingsFeature/  TestSupport/
  Legacy/PennyhavenClassic/          the old UIKit code
  Scripts/check_module_graph.py      architecture checker (Chapter 23)
  docs/adr/                          Architecture Decision Records (Chapter 27)
  PennyhavenUITests/                 XCTest UI tests (Chapter 22)
```

We made one story decision that affects how you read the code in this book: **the folder skeleton exists from day one.** The team created every package folder with a stub manifest at the start of the rewrite, even though most packages were empty. A stub manifest for the domain package looks like

this:

Packages/Domain/Package.swift

```
// swift-tools-version: 6.0
import PackageDescription

let package = Package(
    name: "Domain",
    platforms: [.iOS(.v18)],
    products: [
        .library(name: "Domain", targets: ["Domain"]),
        .library(name: "DomainFixtures", targets: ["DomainFixtures"])
    ],
    targets: [
        .target(name: "Domain"),
        .target(name: "DomainFixtures", dependencies: ["Domain"]),
        .testTarget(
            name: "DomainTests",
            dependencies: ["Domain", "DomainFixtures"]
        )
    ]
)
```

The domain package has two library targets. `Domain` holds production code. `DomainFixtures` holds `SampleData` and in-memory repositories, which tests and SwiftUI previews use. Keeping fixtures in a separate target means production code that only needs the types never accidentally depends on sample data. Chapter 16 explains every line of these manifests; for now, it is enough to know that each folder under `Packages/` is a package and that a tools version of 6.0 puts the package in the Swift 6 language mode by default.

Why build the skeleton first instead of extracting packages later? There are three reasons.

- **Boundaries are cheapest before code exists.** Moving a type between modules later means fixing access modifiers, imports, and call sites. Putting it in the right package on the day it is written costs nothing.
- **The compiler enforces the graph.** If `BudgetFeature` does not declare a dependency on `DataLayer`, code in it cannot import `DataLayer`. The skeleton turns the intended architecture into something the build checks on every compile.
- **Captions stay stable.** For this book, it means every code listing can show its final location, such as `Packages/BudgetFeature/Sources/BudgetFeature/BudgetViewModel.swift`, and you never have to remember where a file used to live.

The skeleton has a cost too: fifteen manifests to maintain and a project navigator that looks busy before the app does anything. For a small app built by one person, starting with three folders inside a single app target is perfectly reasonable, and Chapter 16 shows how to extract packages from an existing target. We chose the skeleton because Pennyhaven is a rewrite by a team that already knows

which features and layers it needs. That is the situation where early boundaries pay off. If you are exploring a new product idea and do not yet know its shape, delay the packages until the shape appears.

> **TIP**
> When you create a skeleton, give every package a one-line comment at the top of its manifest that states its purpose and what it may import. That comment is the cheapest architecture documentation you will ever write, and reviewers can point to it when a pull request adds a surprising dependency.

The Patterns We Use and Where

This book covers a lot of patterns, and it helps to know in advance where each one appears in Pennyhaven and why. Treat the following table as a map; each row has a chapter that explains the decision in full, including when not to use it.

Pattern	Where in Pennyhaven	Chapter
MVVM with `@Observable`	Budgets and Accounts screens	3, 5
Value-type domain model	`Money`, typed IDs, `LedgerTransaction`	4
Unidirectional store	Transactions screen	6, 7
Router and typed routes	`AppRouter` with one path per tab	8, 10
Coordinator	The multi-step add-transaction flow	9
Composition root	`AppContainer` in the app target	11
Protocol witnesses	Dates, IDs, analytics	12
Use cases and repositories	Domain and DataLayer packages	13, 14
Feature modules	One package per feature and layer	16, 17

One more convention is worth stating early, because it affects naming everywhere. Our transaction model is called `LedgerTransaction`, not `Transaction`, and our category model is `BudgetCategory`, not `Category`. Apple frameworks already use those short names, and a clash produces confusing compiler errors in files that import both. Chapter 4 explains the details.

Target Versions

The code in this book targets the following versions. If you work with a different setup, most ideas still apply, but some APIs and diagnostics will differ.

- **Swift 6 language mode** with complete data-race checking. Concurrency mistakes are compile errors, not warnings. We treat that as a design tool: Chapter 24 shows how isolation annotations document the architecture.
- **Recent Xcode** (the Xcode 26 generation, which ships the Swift 6.2 toolchain). Menu names and build settings move between releases, so when we name a menu item, look nearby if it is not exactly where we say.
- **A deployment target of iOS 18.** That gives us the Observation framework, SwiftData, the `Tab` type for tab views, and the `Mutex` type from the Synchronization module.
- **iOS 26 features only behind availability checks.** A few newer APIs appear in later chapters inside `if #available(iOS 26, *)`, always marked as such. Check Apple's current documentation for their exact behavior.

Swift 6.2 introduced an option to make a whole module main-actor isolated by default. That option can remove a lot of annotations in UI-heavy code. Our canonical code does not rely on it: we write `@MainActor` explicitly on view models, the router, and the store, so every listing means the same thing whichever setting your module uses. Chapter 24 discusses the newer options as alternatives.

An Honest Note About Architecture Debates

iOS architecture is a field with strong opinions. MVVM versus MVC, stores versus view models, coordinators versus routers, hand-written dependency injection versus containers, many modules versus few. You will find confident claims on every side, often with numbers attached: a build that got some percentage faster, a bug count that dropped, a team that shipped more features.

We make a deliberate choice in this book: **we do not quote statistics, benchmarks, or survey results about architecture.** Results like these depend on the codebase, the team, the hardware, and the version of the tools, and they rarely transfer. Instead, we give you reasons you can evaluate against your own project, and methods to measure what matters on your own machine. When Chapter 18 discusses build times, it teaches you how to measure with Xcode's build timing summary rather than promising a number.

Third-party architecture libraries get the same fair treatment. Libraries such as The Composable Architecture or dependency injection containers solve real problems, and many teams use them successfully. Pennyhaven's code uses only Apple frameworks and packages we write ourselves, for one reason: building a small store or a composition root by hand is the best way to understand what such libraries do for you and what they cost. When we mention a library, we describe it neutrally, and you should check the project's current documentation, because these libraries evolve quickly.

How to Read This Book

The chapters build on each other, and the code in later chapters assumes the types created earlier. Reading from front to back is the most reliable path. Different readers can still prioritize differently: