44 · Feature First Architecture

Flutter Practice Notes

Contents

44 · Feature-First Architecture

Feature-First Fundamentals (Feature vs Layer)

Structuring Features & the Core Module

Feature Boundaries & Cross-Feature Dependencies

Scaling & Migrating to Feature-First

Feature-First Integration (Capstone: An App Skeleton)

44 · Feature-First Architecture

Introduction

This module covers **feature-first** (a.k.a. package-by-feature) organization: structuring a Flutter app so code is grouped **by feature** (`features/auth/` , `features/cart/`) — each a self-contained vertical slice with its own domain/data/presentation — rather than by technical **layer** across the whole app (`models/` , `views/` , `controllers/`). It walks the **fundamentals** (feature vs layer, cohesion), **structuring features + a shared** `core` , **feature boundaries & cross-feature dependencies**, and **scaling + migrating** from layer-first — with a capstone. It builds directly on Clean Architecture (Module 40) and MVVM (Module 43), and points toward modular architecture (Module 45).

Why this module exists

As apps grow, **layer-first** folders (all models together, all views together) scatter each feature across the codebase — a change to "cart" touches five distant folders, features tangle, and teams collide. **Feature-first** co-locates everything a feature needs, so slices are **cohesive, independently evolvable, and ownable by a team**, with a shared `core` for cross-cutting concerns. It's the organizing principle that makes Clean/MVVM scale across many features and the step before extracting features into packages/modules.

Module map

#	File	Topic	Level
1	01_feature_first_fundamentals.md	Feature vs layer organization, cohesion, why it scales	🔵
2	02_structuring_features_and_core.md	Feature-slice internals + shared `core` / `shared` module	🟡
3	03_feature_boundaries_and_dependencies.md	Boundaries, cross-feature dependencies, avoiding coupling	🔴
4	04_scaling_and_migration.md	Scaling to many features; migrating from layer-first; packages	🔴
5	05_feature_first_integration.md	Capstone: a feature-first app skeleton	🟡

> **Cross-references:** Clean Architecture (each slice's layers): Module 40. MVVM (presentation per feature): Module 43. Modular architecture (features as packages): Module 45. DI (per-feature wiring): Module 14. Routing (per-feature routes): Module 13. DDD (domain per bounded context): Module 46.

Prerequisites

40 Clean Architecture, 43 MVVM, 14 Dependency Injection, 13 Routing.

What you'll be able to do after this module

- Explain feature-first vs layer-first and why cohesion matters at scale.
- Structure a feature slice (domain/data/presentation) + a shared `core` module.
- Define feature boundaries and manage cross-feature dependencies without coupling.
- Scale to many features and migrate a layer-first codebase incrementally.
- Lay out a feature-first app skeleton ready to grow (and later modularize).

Capstone

Feature-first skeleton: A small app organized as `features/` (auth, home, profile — each with domain/data/presentation) + `core/` (theme, DI, routing, shared widgets, `Result`), with per-feature DI + routes, clear boundaries (no direct cross-feature imports; interaction via core/interfaces), and a documented migration/scaling note.

Summary

Feature-first organizes code by feature (cohesive vertical slices) with a shared `core`, instead of by layer across the app. It makes features independently evolvable and team-ownable, scales Clean/MVVM across many features, keeps boundaries explicit, and is the natural precursor to modular (package-based) architecture.

Feature-First Fundamentals (Feature vs Layer)

> Two ways to organize a codebase: **layer-first** (group by technical role across the whole app — `models/`, `views/`, `controllers/`, `repositories/`) or **feature-first** (group by business capability — `features/cart/`, `features/auth/`, each containing *its own* domain/data/presentation). Feature-first wins at scale because it maximizes **cohesion** (everything a feature needs is together) and minimizes **scatter** (a change to "cart" lives in one place, not five distant folders). It follows the principle: **"things that change together should live together."**

Introduction

This file establishes the core distinction — layer-first vs feature-first — and *why* co-locating by feature (cohesion) beats grouping by layer (scatter) as an app grows. It's the conceptual foundation for structuring slices and boundaries in the rest of the module.

Why this concept exists

Small apps tolerate layer-first folders; large apps don't. When every feature's pieces are spread across global `models/` / `views/` / `controllers/` folders, understanding or changing one feature means hopping across the tree, unrelated features share folders (merge conflicts, accidental coupling), and no folder maps to a team's ownership. Feature-first fixes this by making the **feature the unit of organization**.

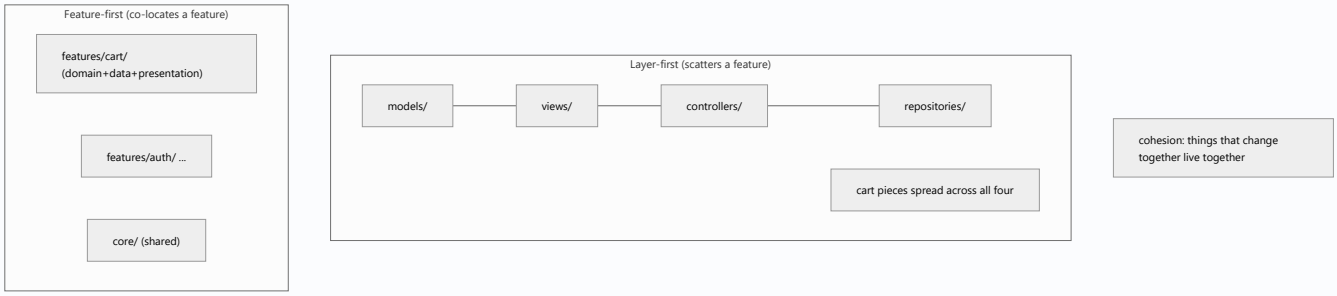
Real-world analogy

Layer-first is a **warehouse organized by material type**: all screws in one aisle, all wood in another, all paint in another — so building one piece of furniture means running across the whole warehouse. Feature-first is a **workshop with a kit per project**: everything for the "bookshelf" (its screws, wood, paint) is in one labeled bin. When you work on the bookshelf, you go to one place; when a new person joins, you hand them the whole bin.

Problem Statement

For a growing app (auth, feed, cart, profile), decide how to organize folders so a change to one feature is localized, teams can own features, and new features slot in cleanly — and articulate why feature-first scales better than layer-first. You'll compare the two layouts and justify the choice.

Internal Working



- **Layer-first (package-by-layer)**: top-level folders by **technical role** — `models/`, `views/`, `controllers/` / `viewmodels/`, `repositories/`, `services/`. A single feature's code is **scattered** across all of them.
  - **+** Familiar; fine for **tiny apps/tutorials**.
  - **−** At scale: **low cohesion** (feature spread out), **navigation friction** (jump across folders to understand one feature), **coupling risk** (unrelated features share folders), **no ownership mapping** (folders don't map to teams), **merge conflicts** (many people editing shared layer folders).
- **Feature-first (package-by-feature)**: top-level `features/<name>/`, each a **self-contained vertical slice** with its **own** `domain/`, `data/`, `presentation/` (Clean layers *inside* the feature — Module 40), plus a **shared** `core/` for cross-cutting concerns (02_structuring_features_and_core.md).
  - **+ High cohesion** (all of a feature in one place), **easy navigation** (feature = folder), **independent evolvability** (change/delete a feature locally), **team ownership** (a team owns a feature folder), **parallel work** (fewer conflicts), **modularization-ready** (a feature folder → a package — Module 45).
  - **−** Slight upfront structure; need discipline on **shared code (core)** and **cross-feature boundaries** (03_feature_boundaries_and_dependencies.md).
- **The principle — cohesion**: **"things that change together should live together."** A feature's UI, logic, and data change together → co-locate them. Technical layers *within* a feature still exist (Clean), but the **top-level axis is the feature**, not the layer.
- **Layers still exist — inside features**: feature-first ≠ abandoning layers. Each feature has domain/data/presentation *internally*; feature-first just makes **feature** the outer grouping and **layer** the inner grouping.
- **When layer-first is OK**: throwaway prototypes / single-screen demos. Beyond that, feature-first scales far better.

Memory Representation

Not runtime — a **source-tree organization**. The mental model: the top-level tree is a list of features + a core; each feature is a mini Clean app. Cohesion is measured by "how many folders must I touch to change one feature?" (ideally one).

Compiler Behavior

Feature-first enables **import-boundary discipline**: a feature imports its own code + `core`, not other features' internals — lintable/enforceable (03_feature_boundaries_and_dependencies.md), and a stepping stone to package boundaries.

Runtime Behavior

Not applicable — organization doesn't change runtime; it changes maintainability, navigation, and team velocity.

Flutter Engine Behavior

Not applicable.

Dart VM Behavior

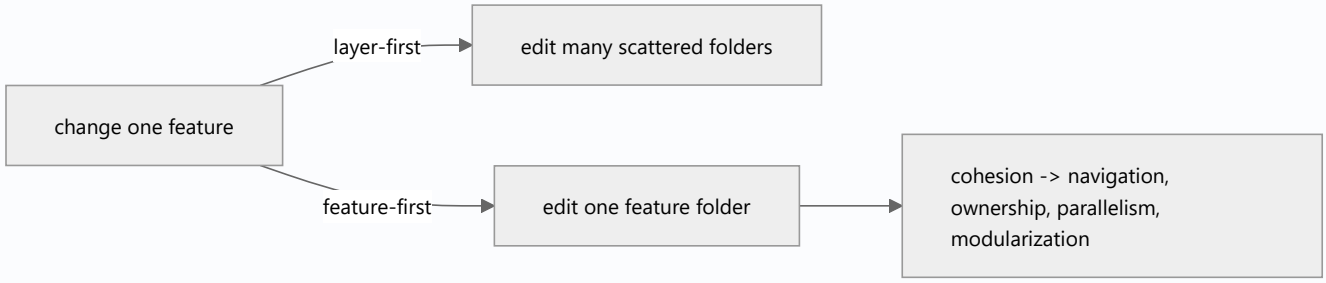
Not applicable (until features become packages — Module 45).

Examples

```
Layer-first (scatters "cart"):                Feature-first (co-locates "cart"):
  lib/                                          lib/
    models/       cart_item.dart  ...             features/
    views/        cart_screen.dart ...              cart/
    controllers/  cart_vm.dart    ...                 domain/    (entities, use cases, interfaces)
    repositories/ cart_repo.dart  ...                 data/      (dtos, sources, repo impl)
    services/     ...                                 presentation/ (view model, widgets)
  # a "cart" change touches 4-5 folders             auth/  home/  profile/  ...
                                                    core/       (theme, di, routing, Result, shared widgets)
  # a "cart" change lives in features/cart/
```

```
Cohesion test:
  "To change the cart feature, how many top-level folders do I edit?"
    layer-first  -> many (models + views + controllers + repositories)   [low cohesion]
    feature-first-> one  (features/cart/)                                [high cohesion]
```

Diagrams



Common Mistakes

Mistake	Why it's a problem	Fix
Layer-first for a large app	Scatter, coupling, conflicts	Feature-first (feature = top-level)
Thinking feature-first drops layers	Loses Clean benefits	Keep layers *inside* each feature
No shared `core`	Duplication of cross-cutting code	Add a `core` / `shared` module
Features importing each other's internals	Coupling	Boundaries via core/interfaces
One giant "feature"	Low cohesion again	Split by real capability
Full feature-first for a 1-screen demo	Overkill	Right-size (layer-first ok tiny)

Best Practices

- Organize **top-level by feature** (`features/<name>/`), each a **cohesive vertical slice** with its **own domain/data/presentation** (Clean *inside*), plus a shared `core` .
- Apply the principle **"things that change together live together"**; measure by the **cohesion test** (one feature = one folder to edit).
- Keep **feature boundaries explicit** (a feature imports its own code + `core` , not other features' internals) — lint/enforce imports.
- **Right-size**: feature-first for real/growing apps; layer-first only for throwaway demos; split by **real business capability**, not size.

Performance

No runtime performance impact — this is a maintainability/velocity concern. The "performance" gained is **developer performance**: faster navigation, safer parallel work, easier onboarding, and readiness to modularize.

Advantages / Disadvantages

- **+** High cohesion, easy navigation, independent/deletable features, team ownership, parallel work, modularization-ready.
- **−** Slight upfront structure, requires `core` + boundary discipline, overkill for trivial apps, risk of ill-defined "features."

Interview Questions

1. 🟢 **Feature-first vs layer-first — what's the difference?** — Feature-first groups top-level by business capability (each feature has its own layers); layer-first groups by technical role across the whole app.
2. 🟢 **Why does feature-first scale better?** — Cohesion: a feature's code lives together, so changes are localized, features are independently evolvable, and teams can own folders — vs layer-first scatter.
3. 🟡 **Does feature-first mean no layers?** — No — layers (domain/data/presentation) exist *inside* each feature; feature-first just makes feature the outer axis and layer the inner.
4. 🟡 **What's the guiding principle?** — "Things that change together should live together" — a feature's UI/logic/data change together, so co-locate them.
5. 🟡 **What's the cohesion test?** — "To change one feature, how many top-level folders must I edit?" — ideally one (feature-first) vs many (layer-first).
6. 🔴 **When is layer-first acceptable?** — Throwaway prototypes/single-screen demos; beyond that, feature-first is preferable.
7. 🔴 **How does feature-first prepare for modularization?** — A cohesive feature folder maps naturally to a package/module later (Module 45).

Senior Engineer Tips

- Make the top-level tree read like the product ("auth, cart, feed, profile"), not like a framework glossary ("models, views, controllers"); that alignment is the whole point.
- Keep Clean layers *inside* each feature — feature-first and Clean are complementary axes, not alternatives.
- Define "feature" by business capability and enforce import boundaries early; ill-defined or leaky features erode the cohesion you organized for.

Architect Perspective

Feature-first is the organizational expression of cohesion and the Common Closure Principle: group by reason-to-change (the feature), not by technical role (the layer). It scales Clean/MVVM across many features, aligns folders with teams and product capabilities, keeps changes local, and sets up the clean boundaries that later become package/module boundaries. Layers live inside features; the feature is the unit of organization, ownership, and (eventually) deployment (02_structuring_features_and_core.md, Module 40, Module 45).

Summary

- Feature-first groups top-level by business capability (each feature = a vertical slice with its own domain/data/presentation + a shared `core`); layer-first groups by technical role across the app.
- Feature-first wins at scale via cohesion ("things that change together live together") — localized changes, team ownership, parallelism, modularization-readiness.
- Layers persist *inside* features; right-size (layer-first only for trivial demos); define features by real capability.

Revision Notes

- Layer-first: top-level by role (models/views/controllers) → scatter, coupling, conflicts (ok for tiny apps). Feature-first: top-level `features/<name>/` (own domain/data/presentation) + `core/` → cohesion.
- Principle: "things that change together live together"; cohesion test = folders touched per feature change (one = good).
- Layers live inside features (feature = outer axis, layer = inner); enforce import boundaries; feature folder → future package (Module 45).

Practice Questions

1. Why does layer-first scatter a feature, and why does that hurt at scale?
2. Does feature-first eliminate layers? Where do they go?
3. Apply the cohesion test to a "cart" change in both layouts.

Coding Questions

1. Convert a layer-first tree sketch to feature-first for 3 features + core.
2. Show where a feature's domain/data/presentation live inside its folder.
3. Identify a cross-feature import that would violate boundaries.

Mini Project

Layout comparison (Flutter): For an app with auth/feed/cart/profile, sketch both a layer-first and a feature-first folder tree, apply the cohesion test to a "cart" change in each, and justify feature-first (cohesion, ownership, scaling). Show that Clean layers live *inside* each feature and a `core` holds shared code. Acceptance: both layouts sketched; cohesion test applied; feature-first justified (cohesion/ownership/scale); layers-inside-features + `core` shown; features defined by capability.

Structuring Features & the Core Module

> Each feature folder is a **mini Clean app** — `features/<name>/{domain, data, presentation}` — and everything **shared across features** lives in a `core/` (or `shared/`) module: cross-cutting infrastructure (DI, routing, networking client, `Result`/error types, theme, localization) and **truly reusable** widgets/utilities. The discipline is deciding **what's a feature vs what's core**: feature-specific code stays in the feature; only genuinely cross-cutting, stable, reusable code goes to core. A bloated core (dumping ground) or a leaky feature (reaching into another) are the two failure modes.

Introduction

This file details the internal structure of a feature slice and the shared `core` module: what each contains, the feature↔core relationship, and the judgment of feature-vs-core placement. It's the "what goes where" companion to the fundamentals (01_feature_first_fundamentals.md).

Why this concept exists

Feature-first only works if features are consistent (predictable internal structure) and shared code has a clear home. Without a `core`, cross-cutting concerns get duplicated or dumped into a random feature; without placement discipline, `core` becomes a junk drawer or features leak into each other. A defined feature structure + a curated core keep the app cohesive and navigable.

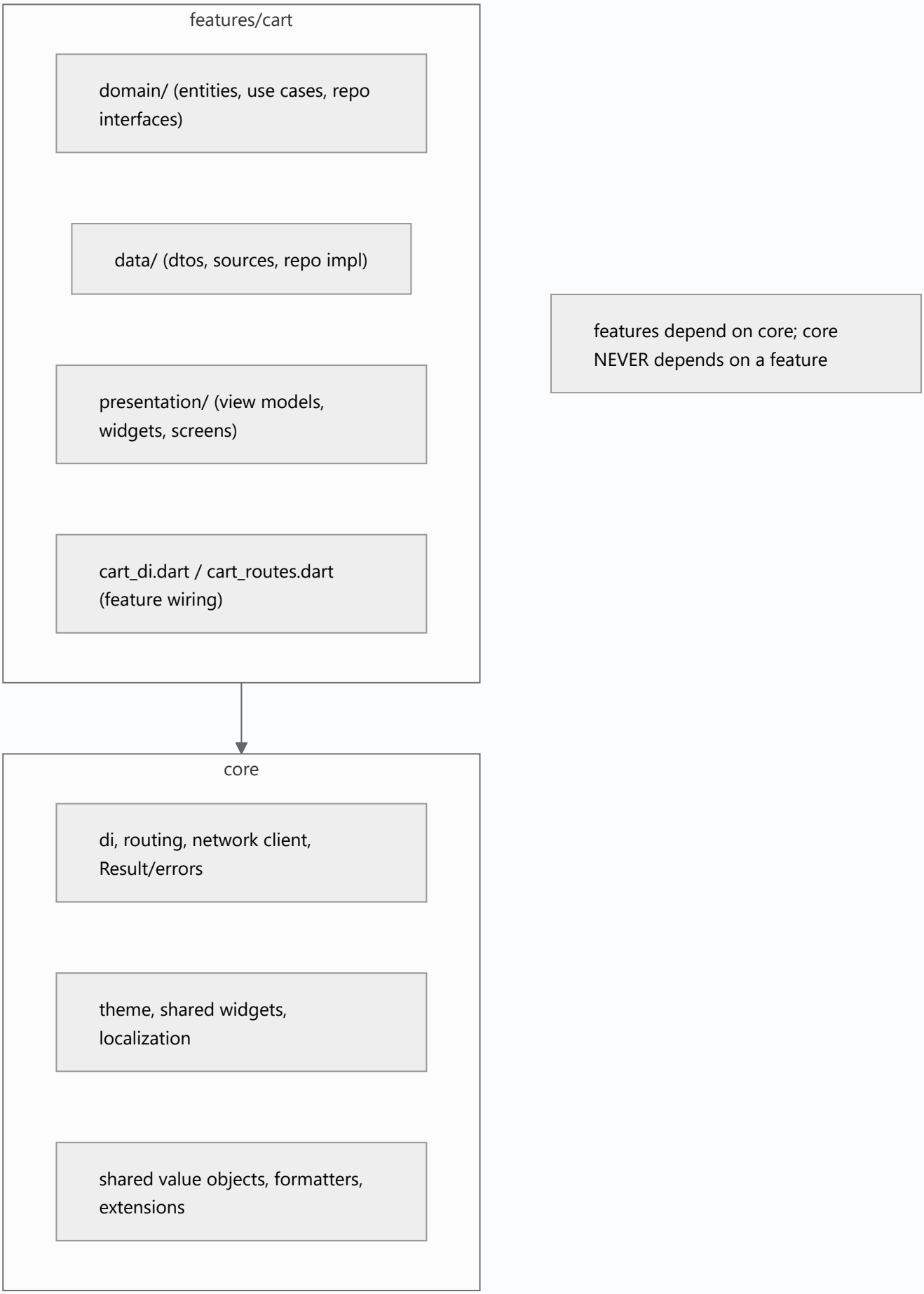
Real-world analogy

Each **feature** is a fully-equipped **food truck** (its own kitchen, menu, register — domain/data/presentation) that can operate independently. The `core` is the **commissary/shared kitchen**: shared ovens, the payment system, the brand signage, common ingredients every truck uses. You don't put the taco truck's salsa recipe in the commissary (feature-specific → stays in the truck), and you don't give each truck its own payment terminal (cross-cutting → belongs in core). A commissary crammed with every truck's private gear is chaos; a truck secretly using another truck's fryer is a boundary violation.

Problem Statement

Structure a `features/cart/` slice internally and decide, for a list of items (a `Money` value object, the `dio` client, a `PrimaryButton`, the cart repository, the app theme, a date formatter), whether each belongs in the cart feature or in `core`. You'll define the feature layout + core contents + placement rules.

Internal Working


features/cart
domain/ (entities, use cases, repo interfaces)
data/ (dtos, sources, repo impl)
presentation/ (view models, widgets, screens)
cart_di.dart / cart_routes.dart (feature wiring)
core
di, routing, network client, Result/errors
theme, shared widgets, localization
shared value objects, formatters, extensions
features depend on core; core NEVER depends on a feature

- **Feature slice internals** (consistent across features): `features/<name>/`
  - `domain/` — entities, use cases, repository **interfaces** (pure — Module 40).
  - `data/` — DTOs, data sources, repository **impls** (map/convert).
  - `presentation/` — view models + widgets/screens (MVVM — Module 43).
  - **feature wiring** — the feature's **DI registration** and **routes** (`cart_di.dart`, `cart_routes.dart`) so the feature self-registers (Module 14/Module 13).
  - Optional `<feature>.dart` **barrel** exposing the feature's **public API** (what other parts may use) while keeping internals private.
- **The `core` / `shared` module** — cross-cutting, feature-agnostic:
  - **Infrastructure**: DI container setup, router root, the **network client** (`dio` + interceptors), `Result`**/error types**, logging facade, storage helpers.
  - **UI**: **theme/design system**, **truly reusable widgets** (`PrimaryButton`, `AppScaffold`), localization.
  - **Utilities**: shared **value objects** (`Money`, `Email`), formatters (`intl`), extensions, constants.
  - Sometimes split into `core/` (infra) + `shared/` / `ui/` (design system) — pick a convention.
- **Feature-vs-core placement rules**:
  - **Feature-specific** (used by one feature) → **stays in the feature** (even if it *looks* generic — don't pre-promote).
  - **Cross-cutting / used by many, stable, reusable → core**.
  - **Rule of three (heuristic)**: promote to core when **≥2–3 features genuinely need it**; before that, keep it local (avoid premature/wrong abstractions).
- **Dependency direction (critical)**: **features depend on `core`; `core` NEVER depends on a feature.** Core is the stable base; features are the volatile leaves. A `core` importing a feature is a red flag (inverted dependency).
- **Failure modes**: **bloated core** (everything dumped in → a second layer-first app) and **leaky features** (feature A imports feature B's internals → coupling — 03_feature_boundaries_and_dependencies.md).

Memory Representation

Not runtime — a **tree + dependency direction**. Each feature is a subtree (domain/data/presentation + wiring); core is a shared subtree features import. The invariant: edges go feature→core, never core→feature.

Compiler Behavior

Barrel files + private (`_`) members let a feature expose a **public API** and hide internals; import lints can enforce feature→core-only and no core→feature imports (checkable boundaries).

Runtime Behavior

At startup, each feature registers its DI + routes (into core's container/router); core infra (client, theme) is shared. Organization doesn't change runtime behavior, only wiring locality.

Flutter Engine Behavior

Not applicable.

Dart VM Behavior

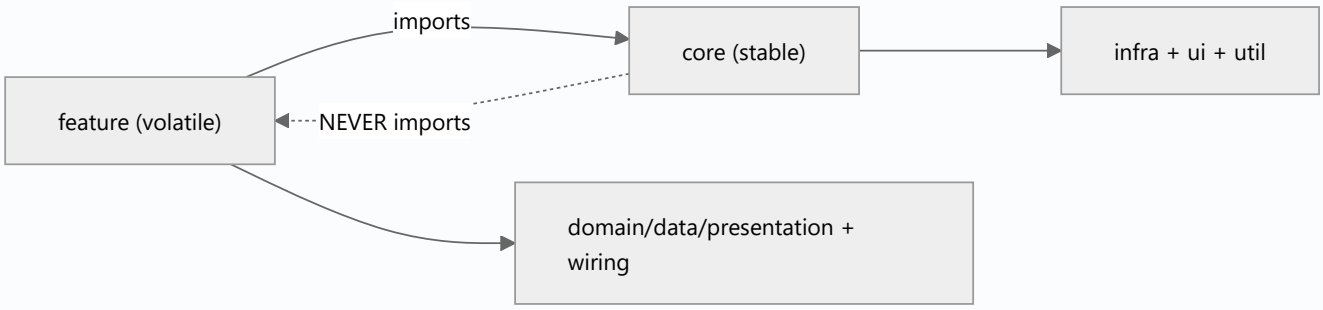
Not applicable (until features become packages — Module 45).

Examples

```
lib/
  core/
    di/            (getIt setup, registerCore)
    routing/       (root router)
    network/       (dio client + interceptors)
    error/         (Result, Failure hierarchy)
    ui/            (theme, PrimaryButton, AppScaffold)
    util/          (Money, formatters, extensions)
  features/
    cart/
      domain/      (Cart, AddToCart, CartRepository interface)
      data/        (CartDto, CartRemote/Local, CartRepositoryImpl)
      presentation/(CartViewModel, CartScreen, widgets)
      cart_di.dart      (registerCart -> registers repo/use cases/VM)
      cart_routes.dart  (cart routes)
      cart.dart         (barrel: public API only)
    auth/  home/  profile/  ...
```

```
// Feature self-registers its DI (called from core's composition root)
void registerCart(GetIt di) {
  di.registerLazySingleton<CartRepository>(() => CartRepositoryImpl(di())); // uses core network client
  di.registerFactory(() => AddToCart(di<CartRepository>()));
  di.registerFactory(() => CartViewModel(di<AddToCart>()));
}
// core/di/register_core.dart calls registerCore() then each feature's registerX(di).
// A `Money` value object used by cart + checkout + wallet -> promote to core/util (rule of three).
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`core` imports a feature	Inverted dependency	Core stays feature-agnostic (features→core only)
Bloated core (junk drawer)	Second layer-first blob	Only cross-cutting/stable/reusable in core
Pre-promoting feature-specific code to core	Wrong abstraction	Keep local until ≥2–3 features need it
Inconsistent feature internals	Hard to navigate	Standard `domain/data/presentation` + wiring
No public API/barrel	Internals leak	Barrel + private members
Feature imports another feature's internals	Coupling	Interact via core/interfaces (boundaries)
DI/routes centralized away from features	Low cohesion	Feature self-registers DI + routes

Best Practices

- Give every feature a **consistent internal structure** (`domain/data/presentation` + `*_di.dart` / `*_routes.dart` + a barrel exposing a **public API**); keep internals private.
- Put **only cross-cutting, stable, reusable** code in `core` (infra + design system + shared value objects/utils); apply the **rule of three** before promoting.

- Enforce the **dependency direction**: **features** → **core**, never **core** → **feature**; keep core a clean, feature-agnostic base.
- Let **features self-register** DI + routes into core's composition root/router; avoid a bloated core and leaky features.

Performance

No runtime cost. The benefit is developer velocity: consistent slices are fast to navigate/onboard; a lean core avoids the layer-first scatter returning through the back door. (Package extraction later can improve build times — Module 45.)

Advantages / Disadvantages

- **+** Predictable, cohesive slices; clear home for shared code; enforceable dependency direction; self-registering features; modularization-ready.
- **−** Placement judgment (feature vs core), risk of bloated core / leaky features, barrel/public-API discipline, some wiring boilerplate.

Interview Questions

1. 🟢 **What's inside a feature folder?** — Its own `domain/`, `data/`, `presentation/` plus feature wiring (DI registration, routes) and a barrel exposing its public API.
2. 🟢 **What belongs in `core`?** — Cross-cutting, feature-agnostic, stable, reusable code: DI/routing/network client/`Result`, theme/shared widgets, shared value objects/formatters.
3. 🟡 **What's the cardinal dependency rule?** — Features depend on core; **core never depends on a feature** (stable base vs volatile leaves).
4. 🟡 **When do you promote code from a feature to core?** — When ≥2–3 features genuinely need it (rule of three) — not preemptively, to avoid wrong abstractions.
5. 🟡 **How do features self-register?** — Each exposes `registerX(di)`/routes that core's composition root/router calls at startup — keeping wiring cohesive.
6. 🔴 **What are the two failure modes?** — A bloated core (junk drawer → second layer-first blob) and leaky features (importing each other's internals → coupling).
7. 🔴 **How do you hide a feature's internals?** — A barrel exposing only the public API + private (`_`) members; lint import boundaries.

Senior Engineer Tips

- Treat `core` as a curated library with the same "would I publish this?" bar; the moment it becomes a dumping ground, you've recreated layer-first inside it.

- Resist promoting code to core until multiple features truly need it — premature promotion creates wrong shared abstractions that are painful to unwind.
- Standardize the feature template (domain/data/presentation + `_di` / `_routes` + barrel) and have features self-register; consistency + cohesion are what make feature-first pay off.

Architect Perspective

The feature-slice + `core` split operationalizes stable-dependencies: `core` is the stable, feature-agnostic base; features are volatile leaves that depend on it, never the reverse. Consistent internal structure (Clean layers), self-registration, and public-API barrels make features cohesive and swappable, while a disciplined core avoids the layer-first blob returning. This structure is what a package/module boundary later formalizes (03_feature_boundaries_and_dependencies.md, Module 45, Module 40).

Summary

- Feature = `domain/data/presentation` + wiring (`*_di` / `*_routes`) + barrel (public API); consistent across features.
- `core` = cross-cutting, stable, reusable code (infra + design system + shared value objects); features → core, **never** core → feature.
- Promote to core by the rule of three; avoid bloated core / leaky features; features self-register DI + routes.

Revision Notes

- Feature internals: `domain/` (entities/use cases/interfaces), `data/` (dtos/sources/impl), `presentation/` (VM/widgets), `*_di.dart` / `*_routes.dart`, barrel (public API + private members).
- `core` / `shared` : DI/routing/network/ `Result` /errors, theme/shared widgets/l10n, shared value objects/formatters/extensions.
- Direction: features→core only (core feature-agnostic); rule of three to promote; features self-register DI+routes; avoid bloated core & leaky features.

Practice Questions

1. What's the standard internal structure of a feature?
2. What is (and isn't) allowed in `core`, and why the dependency direction?
3. When do you promote code from a feature to core?

Coding Questions

1. Lay out a `features/cart/` slice + a `core/` tree.
2. Write `registerCart(di)` self-registration using core's network client.
3. Classify five items as feature-local vs core with justification.

Mini Project

Feature slice + core (Flutter): Structure a `features/cart/` slice (domain/data/presentation + `cart_di.dart` / `cart_routes.dart` + barrel exposing a public API) and a `core/` module (DI, network client, `Result`, theme, a shared `Money` value object). Classify a given list of items as feature-local vs core (rule of three), and ensure features depend on core (never the reverse). Acceptance: consistent feature internals + wiring + barrel; core holds only cross-cutting/stable/reusable code; features→core dependency only; placement decisions justified; feature self-registers DI/routes.

Feature Boundaries & Cross-Feature Dependencies

> The value of feature-first evaporates if features reach into each other's internals, so **boundaries must be explicit**: a feature may depend on `core` and on other features **only through a published contract** (an interface in `core`, an event, or a route) — **never** by importing another feature's `domain/data/presentation` directly. When feature A needs something from feature B, invert the dependency (B implements a core interface A depends on), communicate via **events/messages**, or navigate via **routes** with parameters. This keeps features **loosely coupled, independently evolvable, and extractable into packages**.

Introduction

This file addresses the hard part of feature-first: how features interact **without** coupling. It covers boundary rules, the patterns for cross-feature communication (interfaces-in-core, events, routing), circular-dependency avoidance, and enforcement. It's what keeps the slices from silently re-tangling (02_structuring_features_and_core.md).

Why this concept exists

Real features aren't fully independent — checkout needs the cart, profile needs auth. If they satisfy those needs by importing each other's internals, you get a tangled graph (circular deps, ripple changes, un-extractable features) — layer-first coupling under a feature-first tree. Explicit boundaries + inversion patterns let features cooperate while staying decoupled.

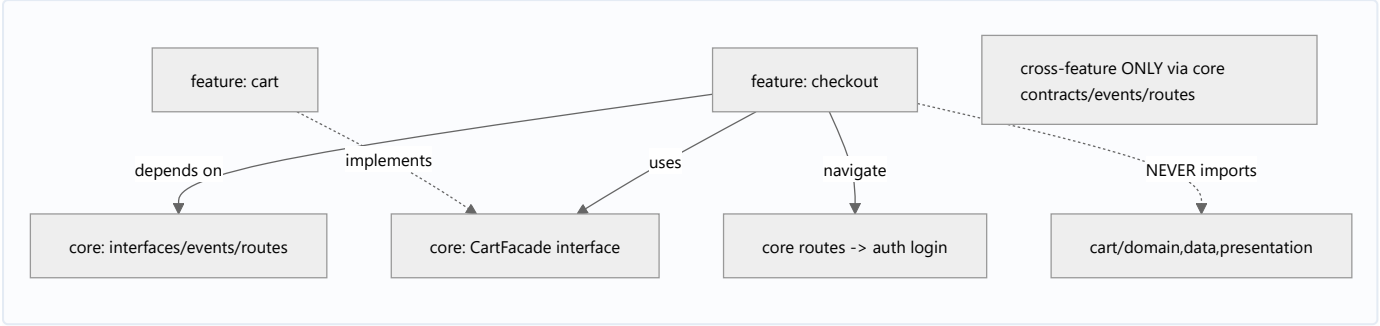
Real-world analogy

Features are **neighboring shops in a mall**. They cooperate — the coffee shop's customers use the bookstore — but they don't **walk into each other's stockrooms** (import internals). They interact through the **mall's shared facilities** (core), **public counters** (published contracts), and **the mall directory** (routing). If the coffee shop needs a book delivered, it uses the mall's **standard delivery service** (a core interface the bookstore fulfills), not by rummaging in the bookstore's back office.

Problem Statement

Checkout (feature) needs the current cart (from the cart feature) and to send the user to login (auth feature) if unauthenticated — without importing cart's or auth's internals. Design the boundaries and interaction so features stay decoupled and extractable. You'll apply interface-inversion, events, and routing.

Internal Working



- **Boundary rule**: a feature may import **its own code** + `core`. It must **not** import another feature's `domain/data/presentation` internals. Cross-feature interaction goes **through contracts published in** `core` (or events/routes).
- **Patterns for cross-feature dependency** (pick per case):
  1. **Interface in** `core` **(dependency inversion)**: define the needed capability as an **interface in core** (`CartFacade { Future<Cart> current(); }`); the **cart feature implements** it and registers it in DI; **checkout depends on the interface** (in core), not on cart. Checkout doesn't know cart exists — just the contract (Module 04).
  2. **Events / message bus**: features publish/subscribe to **domain events** via a core event bus (`UserLoggedOut` , `CartCleared`) — fully decoupled, good for reactions/notifications.
  3. **Routing**: to *go to* another feature's screen, **navigate by route** (`context.go('/login?redirect=...')`) via the core router — no import of that feature's widgets (Module 13).
- **Shared data/models**: if two features share an entity/value object, it belongs in `core` (shared kernel — Module 46); don't reach into one feature's domain for it.
- **Circular dependencies (forbidden)**: A→B and B→A (even via core if sloppy) is a red flag; invert one direction (interface in core), or extract the shared piece to core. The dependency graph among features should be **acyclic** (ideally features depend only on core, forming a star).
- **Composition root**: cross-feature wiring (which impl fulfills which interface) happens in **core's DI composition root**, keeping features unaware of each other's concretes (Module 14).
- **Enforcement**: **import lints/analysis** (or package boundaries — Module 45) to forbid `features/x/**` importing `features/y/**` ; a barrel exposes only a feature's public API.
- **Extractability payoff**: with clean boundaries, a feature can be **lifted into its own package** with minimal changes — the ultimate test that boundaries are real.

Memory Representation

Not runtime — a **feature dependency graph** that must be **acyclic** and (ideally) star-shaped around `core` . Contracts live in core; implementations register at the composition root; features hold references to interfaces, not each other.

Compiler Behavior

Import boundaries are compile-time-checkable (lints/analysis; hard-enforced once features are packages). Depending on a core interface compiles without knowing the implementing feature — the inversion.

Runtime Behavior

DI wires the implementing feature's concrete to the core interface at startup; at runtime checkout calls the interface (resolved to cart's impl) or emits/handles events or navigates by route — all without a static cross-feature import.

Flutter Engine Behavior

Not applicable (routing aside — navigation by route avoids widget imports).

Dart VM Behavior

Not applicable (until package boundaries enforce it at the module level — Module 45).

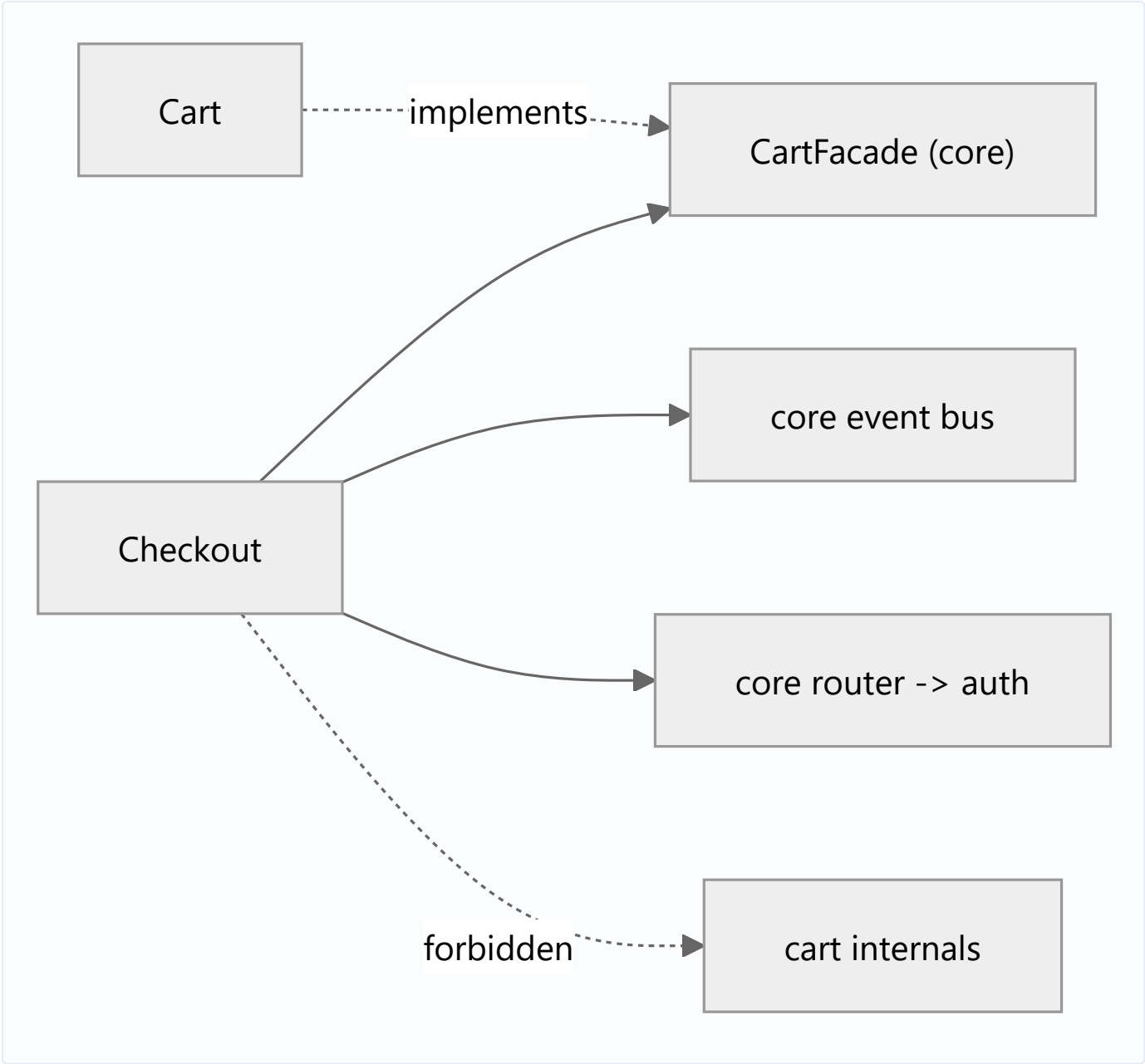
Examples

```
// core/contracts/cart_facade.dart — the published contract (in CORE)
abstract class CartFacade { Future<Cart> current(); }

// features/cart — cart IMPLEMENTS the core contract + registers it
class CartFacadeImpl implements CartFacade {
  final GetCurrentCart getCart;
  CartFacadeImpl(this.getCart);
  @override Future<Cart> current() async => (await getCart()).valueOrThrow();
}
// cart_di.dart: di.registerLazySingleton<CartFacade>(() => CartFacadeImpl(di()));

// features/checkout — depends on the CORE interface, NOT on the cart feature
class CheckoutViewModel {
  final CartFacade cart;               // core contract (checkout doesn't import cart internals)
  final Router router;                 // core router
  CheckoutViewModel(this.cart, this.router);
  Future<void> start() async {
    final c = await cart.current();     // cross-feature via contract
    if (!authed) router.go('/login?redirect=/checkout'); // cross-feature via route
  }
}
// ANTI-PATTERN: import 'package:app/features/cart/domain/cart_repository.dart'; // ❌ internal import
```

Diagrams



Common Mistakes

Mistake	Why it's bad	Fix
Feature A imports feature B internals	Coupling, re-tangling	Depend on a core contract/event/route
Circular feature dependency	Un-buildable/extractable	Invert via core interface; keep acyclic
Shared entity inside one feature	Others reach in	Move to core (shared kernel)
Concrete cross-feature refs	Tight coupling	DI to a core interface (inversion)
Navigating by importing widgets	Widget coupling	Navigate by route
No import enforcement	Boundaries erode silently	Lints/package boundaries
Everything through events	Hard to trace	Use interfaces for direct needs; events for reactions

Best Practices

- A feature depends on **its own code +** `core`, and on other features **only via published contracts** (core interface), **events**, or **routes** — **never** internal imports.
- **Invert** direct cross-feature needs: define the capability as an **interface in core**, implemented by the providing feature, depended on by the consumer; wire at the **composition root**.
- Keep the **feature graph acyclic** (ideally star-shaped around core); put **shared entities in core**; **navigate by route**, not widget imports.
- **Enforce boundaries** with import lints/analysis (and later package boundaries); expose only a feature's **public API** via a barrel.

Performance

No runtime cost; DI/interface indirection is negligible. Clean boundaries improve build times when features become packages (parallel/incremental builds — Module 45) and dramatically improve change-locality (developer performance).

Advantages / Disadvantages

- **+** Loose coupling, independently evolvable/deletable/extractable features, acyclic graph, enforceable boundaries, team autonomy.
- **−** Contract/DI boilerplate for interactions, judgment on interface-vs-event-vs-route, discipline/tooling to enforce, shared-kernel decisions.

Interview Questions

1. 🟢 **What may a feature import?** — Its own code and `core` — not another feature's internals.
2. 🟢 **How should features interact?** — Via published contracts (interfaces in core), events, or routes — never by importing each other's domain/data/presentation.
3. 🟡 **How do you let checkout use the cart without coupling?** — Define a `CartFacade` interface in core, have cart implement + register it, and let checkout depend on the interface (dependency inversion).
4. 🟡 **Where do shared entities go?** — In `core` (shared kernel) — not inside one feature that others reach into.
5. 🟡 **How do you avoid circular feature dependencies?** — Invert one direction via a core interface (or extract shared code to core); keep the feature graph acyclic (star around core).
6. 🔴 **When use events vs interfaces vs routes?** — Interfaces for direct capability needs, events for decoupled reactions/notifications, routes for navigating to another feature's screen.
7. 🔴 **How do you enforce boundaries?** — Import lints/analysis (forbid `features/x` → `features/y`), public-API barrels, and eventually package boundaries.

Senior Engineer Tips

- Make "no cross-feature internal imports" a lint rule from day one; boundaries that aren't enforced silently erode until you're back to a tangled monolith.
- Prefer interface-in-core + DI for direct needs (traceable, testable) and reserve the event bus for genuine fan-out reactions; all-events is hard to follow.
- Use "could I extract this feature into its own package with minimal changes?" as your boundary litmus test — if not, something's leaking.

Architect Perspective

Feature boundaries are Dependency Inversion applied at the feature scale: consumers depend on contracts (in the stable core), providers implement them, and the composition root wires concretes — so features cooperate without knowing each other. Keeping the feature graph acyclic and star-shaped around core, communicating via contracts/events/routes, and enforcing import boundaries is exactly what makes features independently evolvable and later extractable into packages/modules. Coupling here is the failure that turns feature-first back into a scattered monolith (02_structuring_features_and_core.md, Module 04, Module 45).

Summary

- Features import only their own code + `core`; cross-feature interaction via published contracts (core interfaces), events, or routes — never internal imports.
- Invert direct needs (interface-in-core, provider implements, consumer depends), keep the feature graph acyclic (star around core), put shared entities in core.
- Navigate by route (not widget imports); enforce boundaries with lints/packages; the test is "extractable into a package."

Revision Notes

- Boundary: feature imports own code + core only; cross-feature via core interface (DIP) / events / routes; no internal imports; acyclic feature graph (star around core).
- Patterns: interface-in-core (provider implements + DI-registers, consumer depends), event bus (decoupled reactions), routing (navigate w/o widget import); shared entities → core (shared kernel).
- Wire cross-feature at composition root; enforce with import lints/package boundaries + public-API barrels; litmus test = extractable to a package.

Practice Questions

1. How does checkout use the cart without importing it?

2. When do you use an interface vs an event vs a route for cross-feature interaction?
3. Why must the feature dependency graph be acyclic?

Coding Questions

1. Define a core `CartFacade`, implement it in cart, depend on it from checkout.
2. Add a core event bus and publish/subscribe to a `UserLoggedOut` event.
3. Add an import lint forbidding cross-feature internal imports.

Mini Project

Decoupled cross-feature interaction (Flutter): Wire checkout↔cart↔auth without internal cross-feature imports: a `CartFacade` interface in core (implemented + registered by cart, depended on by checkout), a core event/route for redirecting to auth login, and an import rule forbidding `features/x` → `features/y` internals. Verify the feature graph is acyclic (star around core). Acceptance: no cross-feature internal imports; direct need via core interface (DIP + DI); navigation via route; shared entities in core; acyclic graph; boundary enforced (lint); a feature is plausibly extractable to a package.

Scaling & Migrating to Feature-First

> Feature-first scales from a handful of folders to dozens of features and, eventually, **separate packages** (the bridge to modular architecture — Module 45): as the app grows you keep the same slice template, promote shared code to `core` by the rule of three, and extract mature/reusable features into local packages for **enforced boundaries + faster builds**. Migrating a **layer-first** codebase is done **incrementally, feature by feature** (strangler-fig): stand up `features/` + `core/`, move one feature's scattered pieces into its slice, repeat — never a risky big-bang rewrite.

Introduction

This file covers the two growth dimensions — **scaling** an existing feature-first app (many features → packages) and **migrating** a layer-first app to feature-first incrementally. It's the "how do I actually get here and keep growing" companion to the structure/boundary files.

Why this concept exists

Teams rarely start clean: they have a layer-first codebase that's tangling, or a feature-first app outgrowing a single package. Both need a **safe, incremental path** — big-bang rewrites fail. Knowing the strangler-fig migration and the scale-up-to-packages progression lets you improve architecture continuously without halting delivery.

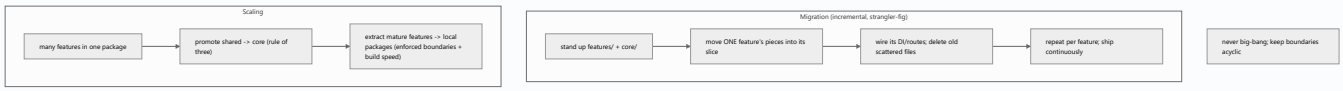
Real-world analogy

Migrating layer-first → feature-first is **renovating a house while living in it**: you don't demolish everything (big-bang); you renovate **one room at a time** (one feature), keeping the house habitable throughout. Scaling to packages is later **converting rooms into self-contained units** (apartments) with their own utilities and locked doors (enforced boundaries) as the building grows.

Problem Statement

You have a tangled layer-first app and need to migrate to feature-first without stopping shipping; later, as features multiply, decide when/how to extract features into packages. You'll plan an incremental migration and a scaling path.

Internal Working



- **Migrating layer-first → feature-first (strangler-fig, incremental)**:
  1. **Create the target structure**: add `features/` and `core/` alongside the existing layout.
  2. **Move cross-cutting first**: relocate shared infra/design-system/utilities into `core` (network client, theme, `Result`).
  3. **Migrate one feature at a time**: pick a **cohesive, low-risk** feature; gather its scattered pieces (its model/view/controller/repo) into `features/<name>/{domain,data,presentation}`; wire its **DI + routes**; **delete** the old files. Ship. Repeat.
  4. **Fix boundaries as you go**: when a migrated feature reaches into another's internals, introduce a **core contract** (03_feature_boundaries_and_dependencies.md).
  5. **Coexist during transition**: layer-first and feature-first can **coexist**; you're not blocked — the tree just gets progressively feature-first. This is the **strangler-fig** pattern (new structure grows around the old until it's replaced).
  - **Avoid big-bang rewrites** (high risk, long freeze, merge hell).
- **Scaling within a single package (dozens of features)**: keep the **consistent slice template**; promote shared code to `core` by the **rule of three**; consider **sub-grouping** features (`features/shopping/{cart,checkout}`) if natural; keep the **feature graph acyclic** (star around core).
- **Scaling to packages/modules (the bridge to Module 45)**:
  - Extract **mature, stable, reusable** features (and `core`) into **local packages** (path dependencies in a monorepo / melos workspace).
  - **Why**: **enforced boundaries** (a package literally can't import another's private code), **faster/incremental builds**, **independent versioning/ownership**, potential reuse across apps.
  - **When**: build times hurt, teams need hard isolation, or a feature is reused elsewhere — not prematurely (packages add overhead).
- **Enforcement throughout**: import lints during single-package phase; **package boundaries** once extracted — the same acyclic, star-around-core graph, now hard-enforced.
- **Right-sizing the journey**: small app → single package feature-first; large/multi-team → package-per-feature. Migrate/scale **only as pain justifies**.

Memory Representation

Not runtime — an **evolving source tree + dependency graph**. Migration transforms a layer-first tree into feature slices incrementally; scaling optionally lifts slices into packages. The invariant across all phases: acyclic, star-around-core dependencies.

Compiler Behavior

Single-package phase: import lints enforce boundaries. Package phase: the **build system enforces** boundaries (can't import a package's non-exported code) and enables **incremental compilation** per package.

Runtime Behavior

Organization/packaging don't change runtime behavior; packaging can improve **build**/tooling performance. DI composition root still wires everything at startup.

Flutter Engine Behavior

Not applicable.

Dart VM Behavior

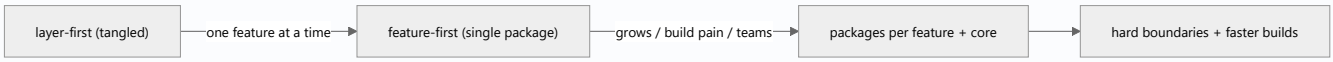
Package extraction enables **incremental/parallel builds** (only changed packages rebuild) — a tooling/build-time benefit (Module 45).

Examples

```
Migration order (strangler-fig), shipping after each step:
  1) add features/ + core/                       (structure)
  2) move network client, theme, Result -> core/ (cross-cutting first)
  3) migrate features/profile/ (low-risk, cohesive); wire DI+routes; delete old files
  4) migrate features/auth/; introduce core contracts where features interacted
  5) ... repeat per feature; layer-first shrinks as feature-first grows
```

```
# Scaling to packages (monorepo, e.g. melos): features & core become path packages
# packages/core, packages/feature_cart, packages/feature_auth ...
# app/pubspec.yaml
dependencies:
  core: { path: ../packages/core }
  feature_cart: { path: ../packages/feature_cart }   # boundaries now build-enforced
  feature_auth: { path: ../packages/feature_auth }
# feature_cart depends on core; NOT on feature_auth (acyclic, star around core)
```

Diagrams



Common Mistakes

Mistake	Why it fails	Fix
Big-bang rewrite	High risk, long freeze, merge hell	Incremental strangler-fig, feature by feature
Migrating a high-risk feature first	Risky learning curve	Start with a low-risk cohesive feature
Not moving cross-cutting to core first	Features re-scatter shared code	Relocate infra/design system to core early
Premature package extraction	Overhead without payoff	Extract only mature/reused features when pain justifies
Leaving old files after migrating	Duplication/confusion	Delete migrated originals
Ignoring boundaries during migration	Re-tangling	Introduce core contracts as needed
Cyclic feature deps at scale	Un-extractable	Keep acyclic (star around core)

Best Practices

- Migrate **incrementally (strangler-fig)**: stand up `features/` + `core/`, move **cross-cutting to core first**, then migrate **one low-risk cohesive feature at a time**, wiring DI/routes and **deleting old files**; ship after each — **never big-bang**.
- **Fix boundaries as you migrate** (introduce core contracts when features interact); keep the graph **acyclic** throughout.
- **Scale** by keeping the slice template + promoting to core (rule of three); **extract features into packages** only when **build pain/team isolation/reuse** justifies it (enforced boundaries +

faster builds).

- **Right-size**: single-package feature-first for most apps; package-per-feature for large/multi-team; enforce boundaries with lints → package boundaries.

Performance

Reorganization is runtime-neutral. **Package extraction** improves **build times** (incremental/parallel) and tooling at scale — a real payoff for large monorepos. The dominant benefit throughout is **developer velocity** (localized changes, parallel work, enforced boundaries).

Advantages / Disadvantages

- **+** Safe continuous improvement (no freeze), progressive cohesion, enforced boundaries + faster builds at scale, incremental risk.
- **−** Migration takes time (coexistence period), judgment on order/when-to-package, discipline to delete old code + fix boundaries, package overhead if premature.

Interview Questions

1. 🟢 **How do you migrate a layer-first app to feature-first?** — Incrementally (strangler-fig): add `features/` + `core/`, move cross-cutting to core, migrate one low-risk feature at a time (wire DI/routes, delete old files), ship after each — no big-bang.
2. 🟢 **Why avoid a big-bang rewrite?** — High risk, long code freeze, and merge hell; incremental migration keeps the app shippable throughout.
3. 🟡 **Which feature do you migrate first, and why?** — A low-risk, cohesive one — to learn the process safely before tackling tangled/critical features.
4. 🟡 **When do you extract features into packages?** — When build times hurt, teams need hard isolation, or a feature is reused — not prematurely (packages add overhead).
5. 🟡 **What do packages buy you over folders?** — Build-enforced boundaries (can't import private code), faster incremental/parallel builds, independent versioning/ownership.
6. 🔴 **How do you keep boundaries clean during migration?** — Introduce core contracts when migrated features need to interact, and keep the feature graph acyclic (star around core).
7. 🔴 **How do layer-first and feature-first coexist during migration?** — They live side by side; the feature-first tree grows as you migrate, and the layer-first tree shrinks — the strangler-fig pattern.

Senior Engineer Tips

- Migrate one cohesive, low-risk feature end-to-end first as a template; it de-risks the process and gives the team a concrete pattern to follow.
- Move shared infrastructure to `core` before migrating features, or you'll re-scatter it; and delete migrated originals immediately to avoid a confusing dual tree.
- Extract packages reactively (build pain, team isolation, reuse), not aspirationally; premature modularization adds friction without payoff — but keep boundaries clean so extraction is cheap when the time comes.

Architect Perspective

Scaling and migration make feature-first a **continuous, low-risk evolution** rather than a rewrite: the strangler-fig pattern converts a tangled layer-first app feature by feature while shipping, and the same cohesive slices later graduate into packages for hard boundaries and build speed. Throughout, the invariant is an acyclic, star-around-core dependency graph — which is exactly what a modular (package) architecture formalizes. The judgment is timing: migrate/scale as pain justifies, keeping boundaries clean so the next step is always cheap (03_feature_boundaries_and_dependencies.md, Module 45).

Summary

- Migrate layer-first → feature-first **incrementally** (strangler-fig): structure + core-first + one low-risk feature at a time (wire DI/routes, delete old, ship) — never big-bang; fix boundaries as you go.
- Scale by keeping the slice template + promoting to core; **extract features into packages** when build pain/team isolation/reuse justify (enforced boundaries + faster builds).
- Keep the graph acyclic (star around core) throughout; right-size; boundaries are the bridge to modular architecture.

Revision Notes

- Migration (strangler-fig): add features/+core/; move cross-cutting→core first; migrate one low-risk cohesive feature at a time (DI/routes, delete originals); ship after each; introduce core contracts as needed; layer/feature coexist; no big-bang.
- Scaling: consistent slice template + rule-of-three promotion; extract mature/reused features → local packages (monorepo/melos) for enforced boundaries + incremental builds; when build pain/team isolation/reuse justify.
- Invariant: acyclic star-around-core graph; enforce via lints → package boundaries; right-size; bridge to Module 45.

Practice Questions

1. Outline an incremental migration from layer-first and why big-bang is avoided.
2. Which feature would you migrate first and why?
3. When and why extract a feature into a package?

Coding Questions

1. Plan a step-by-step migration order for a given layer-first app.
2. Convert two features + core into path packages (monorepo) keeping the graph acyclic.
3. Add boundary enforcement (lint now, package later) for a migrated feature.

Mini Project

Migration + scaling plan (Flutter): For a tangled layer-first app (auth/feed/cart/profile + shared services), write an incremental strangler-fig migration plan (structure → core-first → per-feature order with DI/routes + deletion + ship points + boundary fixes), and a scaling plan to packages (which features to extract, when, and why — enforced boundaries/build speed), keeping the feature graph acyclic. Acceptance: incremental (no big-bang) migration with a sensible order (low-risk first) and coexistence; cross-cutting→core first; boundaries fixed via core contracts; package-extraction criteria + acyclic graph; right-sized to the app.

Feature-First Integration (Capstone: An App Skeleton)

> Assemble a complete feature-first skeleton: `features/` (each a Clean/MVVM vertical slice that **self-registers** its DI + routes) + `core/` (infra, design system, `Result`, shared value objects, the **composition root** and **root router**), with **explicit boundaries** (features → core only; cross-feature via core contracts/events/routes; acyclic star-around-core graph). This is the organizing chassis that scales Clean + MVVM across many features, aligns folders with teams and product capabilities, and is ready to grow into packages (Module 45).

Introduction

This module capstone composes fundamentals, feature/core structure, boundaries, and scaling into one working app skeleton. It shows the full tree, the composition root that wires self-registering features, and the boundary discipline that keeps slices decoupled — the template teams grow on.

Why this concept exists

The parts only pay off when assembled: consistent slices + a curated core + enforced boundaries + a composition root = an app that scales cohesively and stays navigable, ownable, and extractable. This capstone provides that reference skeleton.

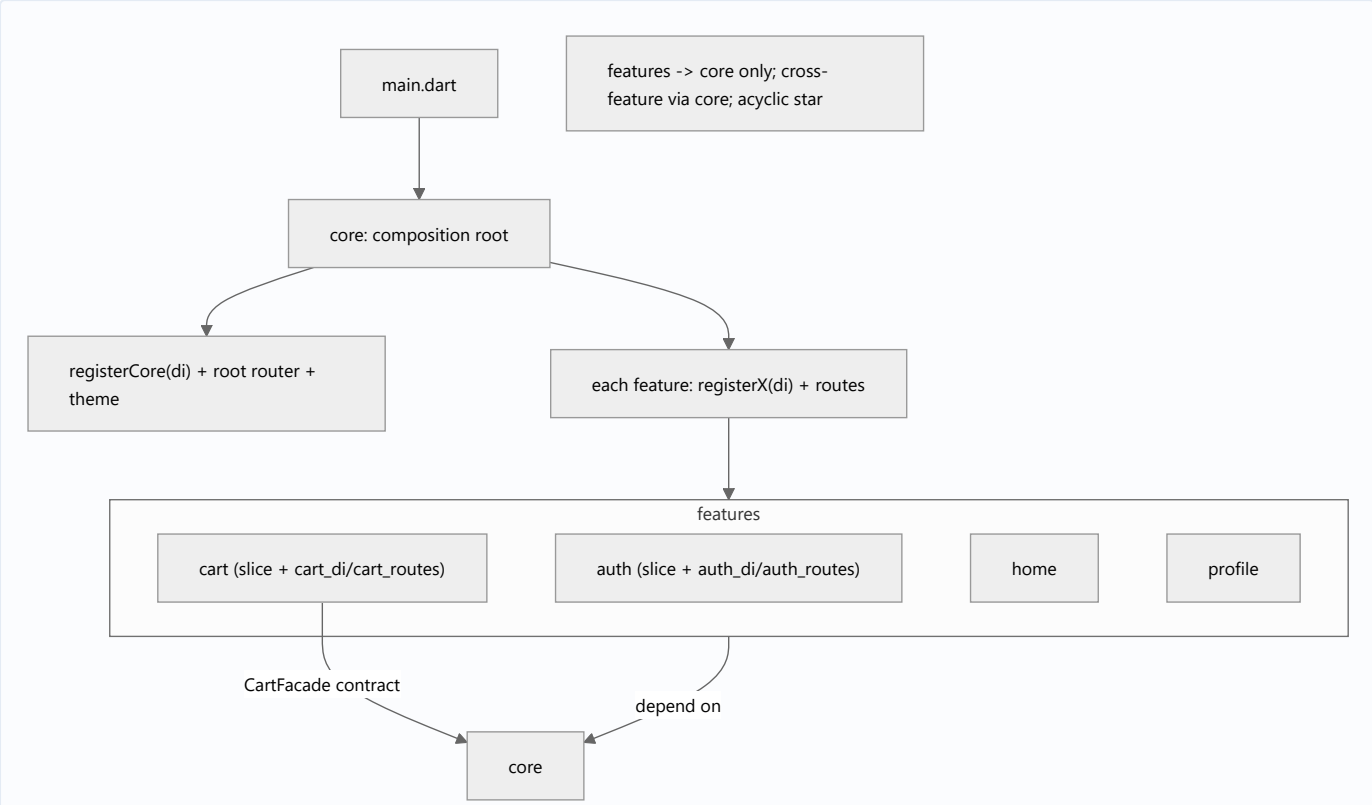
Real-world analogy

It's a **food court built to a standard stall template**: each stall (feature) is a complete mini-restaurant that plugs into the court's shared utilities (core) and registers itself in the directory (routes) and billing (DI). The court operator (composition root) just powers on each stall; stalls cooperate through court services, never by walking into each other's kitchens — and a successful stall can graduate to its own standalone location (package).

Problem Statement

Build a feature-first skeleton for an e-commerce-ish app (auth, home, cart, profile): each feature a self-registering Clean/MVVM slice, a `core` with the composition root/router/design system/ `Result`, boundaries enforced (no cross-feature internal imports; interaction via core), and a documented scaling/migration note. You'll compose every file in this module into a runnable chassis.

Internal Working



- **Tree**: `lib/core/` (di, routing, network, error/ `Result`, ui/theme + shared widgets, util/value objects, **composition root**) + `lib/features/<name>/` (domain/data/presentation + `*_di.dart` / `*_routes.dart` + barrel) — consistent slice template (02_structuring_features_and_core.md).
- **Each feature = Clean + MVVM** (Module 40/Module 43): domain (entities/use cases/interfaces), data (dtos/sources/impls), presentation (view models/screens) — and **self-registers** its DI + routes.
- **Core composition root**: `registerCore(di)` sets up shared infra; then calls each feature's `registerX(di)` and aggregates each feature's routes into the **root router** (Module 14/Module 13). `main.dart` just runs the root.
- **Boundaries** (03_feature_boundaries_and_dependencies.md): features import **own code + core** only; cross-feature via **core contracts** (e.g., `CartFacade`), **events**, or **routes**; **acyclic star-around-core** graph; enforced by import lints (→ package boundaries later).
- **Shared code in core**: `Result`, `Money`, theme, network client, reusable widgets — promoted by the **rule of three**; core never imports a feature.
- **Growth-ready** (04_scaling_and_migration.md): the skeleton scales by adding slices; mature features extract into **packages** with minimal change (boundaries already clean).
- **Right-sizing**: this full chassis suits real/growing apps; trivial apps can start lighter and grow into it.

Memory Representation

Not runtime state — a source tree + an acyclic dependency graph (features → core). At startup the composition root builds the DI graph + router by invoking each feature's self-registration.

Compiler Behavior

Domain pure; features UI-aware only in presentation; import lints enforce feature→core-only + no cross-feature internals; barrels expose public APIs. Package-ready boundaries.

Runtime Behavior

`main` → composition root → registerCore + per-feature register → app runs with shared infra + feature routes/DI wired. Cross-feature calls resolve to core-contract implementations at runtime.

Flutter Engine Behavior

Standard; only presentation touches the engine. Routing via the root router (navigate by route across features).

Dart VM Behavior

Single package now; package extraction later enables incremental/parallel builds (Module 45).

Examples

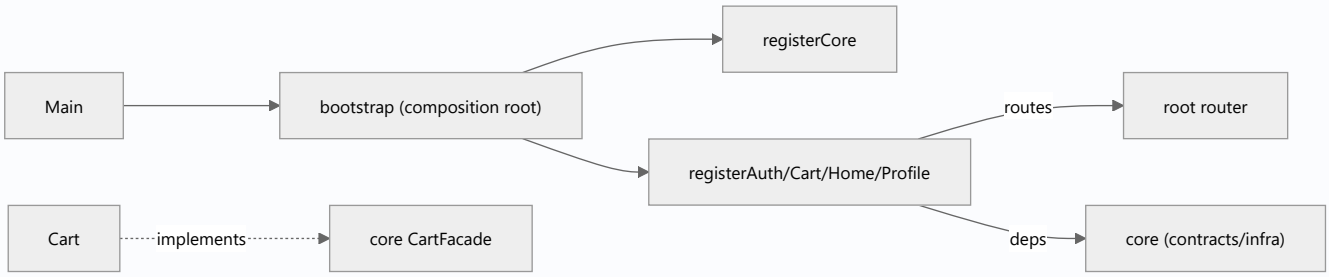
```
lib/
  core/
    di/ (register_core.dart, composition_root.dart)
    routing/ (root_router.dart)
    network/ (dio client)
    error/ (result.dart, failure.dart)
    ui/ (theme.dart, primary_button.dart)
    util/ (money.dart, formatters.dart)
    contracts/ (cart_facade.dart)          # cross-feature contracts live in core
  features/
    auth/    (domain/ data/ presentation/ auth_di.dart auth_routes.dart auth.dart)
    home/    ...
    cart/    (... cart_di.dart cart_routes.dart cart.dart)   # implements CartFacade
    profile/ ...
  main.dart
```

```
// core/di/composition_root.dart — wires self-registering features
Future<void> bootstrap(GetIt di) async {
  registerCore(di);                 // shared infra (client, Result, theme)
  registerAuth(di);                 // each feature self-registers DI + routes
  registerCart(di);                 // cart also registers its CartFacade impl
  registerHome(di);
  registerProfile(di);
}

// core/routing/root_router.dart aggregates each feature's routes:
final router = GoRouter(routes: [...authRoutes, ...cartRoutes, ...homeRoutes, ...profileRoutes]);

// main.dart — just runs the root
void main() async {
  final di = GetIt.instance;
  await bootstrap(di);
  runApp(App(router: router));      // no feature knows about another
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Centralizing all DI/routes away from features	Low cohesion	Features self-register; root aggregates
Cross-feature internal imports	Coupling	Interact via core contracts/events/routes
Bloated core / core→feature import	Re-scatter / inverted dep	Curated core; features→core only
Inconsistent slice structure	Hard to navigate	Standard template per feature
Cyclic feature deps	Un-extractable	Acyclic star around core
No boundary enforcement	Erodes to monolith	Import lints → package boundaries
Over-building for a tiny app	Overkill	Right-size; grow into it

Best Practices

- Standardize the **slice template** (Clean/MVVM + `*_di` / `*_routes` + barrel); have each feature **self-register** DI + routes into a **core composition root/root router**.
- Keep `core` **curated** (infra + design system + shared value objects + cross-feature **contracts**); enforce **features** → **core only** and **acyclic star** boundaries.
- Interact **cross-feature via core contracts/events/routes** (never internal imports); promote shared code by the **rule of three**.
- Keep the skeleton **package-ready** (clean boundaries) and **right-size** — start lighter and grow; enforce boundaries with lints → package boundaries.

Performance

Runtime-neutral; benefits are developer velocity (cohesion, parallel work, onboarding) and, at scale, build speed once features become packages. The composition root adds negligible startup wiring.

Advantages / Disadvantages

- **+** Cohesive, consistent, self-wiring features; curated core; enforced boundaries; team ownership; scales Clean/MVVM; package-ready.
- **−** Upfront chassis setup + conventions, boundary discipline/tooling, judgment on core placement, overkill for trivial apps.

Interview Questions

1. 🟢 **What's in the skeleton's `core` vs `features`?** — Core: infra, design system, `Result`, shared value objects, cross-feature contracts, composition root/router. Features: self-contained

Clean/MVVM slices + their DI/routes.

2. 🟢 **How do features get wired?** — Each self-registers DI + routes; the core composition root calls them and aggregates routes into the root router; `main` runs the root.
3. 🟡 **How do features interact without coupling?** — Via core contracts (e.g., `CartFacade`), events, or routes — never by importing each other's internals; the graph stays acyclic (star around core).
4. 🟡 **Where do shared entities/value objects live?** — In core (promoted by the rule of three); core never imports a feature.
5. 🟡 **How is this skeleton package-ready?** — Clean feature→core boundaries mean a mature feature can be lifted into a package with minimal change.
6. 🔴 **How do you enforce boundaries?** — Import lints now (forbid cross-feature internals), package boundaries later; barrels expose public APIs.
7. 🔴 **How does this scale Clean + MVVM?** — Each feature is a Clean/MVVM slice; feature-first makes feature the outer axis so many such slices coexist cohesively with a shared core.

Senior Engineer Tips

- Build one exemplary self-registering slice + the composition root first; it becomes the template every feature follows, making the codebase uniform and onboarding trivial.
- Keep `main` dumb (just bootstrap + run) and let features own their DI/routes; centralizing wiring re-creates the low-cohesion problem feature-first solves.
- Enforce feature→core-only imports from day one and keep the graph acyclic; that discipline is what makes the eventual package extraction a cheap, mechanical step.

Architect Perspective

The feature-first skeleton is the app's organizational chassis: consistent Clean/MVVM slices that self-wire into a curated core, with enforced, acyclic boundaries. It scales the presentation/layering patterns across many features, aligns structure with teams and product capabilities, keeps changes local, and graduates cleanly into a modular (package) architecture. It's the practical synthesis of everything in the architecture band so far — and the launchpad for modularization and DDD (Module 43, Module 40, Module 45, Module 46).

Summary

- Skeleton = `features/` (self-registering Clean/MVVM slices) + `core/` (infra, design system, `Result`, contracts, composition root/router); `main` just bootstraps + runs.
- Boundaries: features → core only; cross-feature via core contracts/events/routes; acyclic star-around-core; enforced by lints → packages.

- Curated core (rule of three); consistent slice template; package-ready; right-size and grow.

Revision Notes

- Tree: `core/` (di/routing/network/error/ui/util/contracts + composition root) + `features/<name>/` (domain/data/presentation + `*_di` / `*_routes` + barrel).
- Wiring: features self-register DI+routes; core composition root calls them + aggregates routes; `main` runs root; cross-feature via core contracts/events/routes.
- Boundaries: features→core only (acyclic star), curated core (rule of three), lints→package boundaries; package-ready; right-size.

Practice Questions

1. How does a feature self-register, and what does the composition root do?
2. How do features cooperate without importing each other?
3. What makes this skeleton ready to modularize?

Coding Questions

1. Lay out the full `core/` + `features/` tree with per-feature wiring.
2. Write the composition root that wires self-registering features + aggregates routes.
3. Add a cross-feature `CartFacade` contract (core) implemented by cart, used by checkout.

Mini Project

Feature-first app skeleton (capstone — Flutter): Build a runnable skeleton with `core/` (composition root, root router, network client, `Result`, theme, a `CartFacade` contract) and `features/` (auth, home, cart, profile — each a Clean/MVVM slice that self-registers DI + routes; cart implements `CartFacade`). Enforce features→core-only imports and cross-feature interaction via core; keep the graph acyclic; document the scaling/migration path. Acceptance: consistent self-registering slices + curated core; composition root wires features + aggregates routes; no cross-feature internal imports (interaction via core contract/route); acyclic star graph; boundary lint; scaling/migration note; package-ready; runs end-to-end.