45 · Modular Architecture

Flutter Practice Notes

Contents

45 · Modular Architecture

Introduction

This module covers **modular architecture** — taking feature-first organization (Module 44) to its conclusion by making each feature (and `core`) a **separate Dart/Flutter package** with **compiler-enforced boundaries** and its own dependencies. It walks the **fundamentals** (packages as hard boundaries, why/when), **monorepo management with melos** (workspace, path deps, scripts), **module boundaries & contracts** (package exports/public API, dependency graph, contract packages), and **build/versioning/team benefits** — with a capstone. It's the formalization of feature-first and the bridge to enterprise-scale and DDD (Module 46, Module 47).

Why this module exists

Feature-first folders rely on **discipline/lints** to keep boundaries — which erode silently. Turning features into **packages** makes boundaries **compiler-enforced** (a package literally cannot import another's private code), unlocks **incremental/parallel builds** (only changed packages rebuild), enables **independent versioning/ownership**, and allows **reuse across apps**. But modularity adds real overhead (workspace tooling, contract design, wiring), so it must be adopted **when the scale justifies it** — the module teaches both the how and the when.

Module map

#	File	Topic	Level
1	01_modular_fundamentals.md	Packages as hard boundaries, why/when to modularize	🔴
2	02_monorepo_and_melos.md	Monorepo layout, melos, path deps, workspace scripts	🔴
3	03_module_boundaries_and_contracts.md	Package exports/public API, contracts, dependency graph	🔴
4	04_build_versioning_and_teams.md	Build speed, versioning, ownership, CI	🟡
5	05_modular_integration.md	Capstone: a modular monorepo skeleton	🔴

> **Cross-references:** Feature-first (the precursor): Module 44. Clean Architecture (module internals): Module 40. DI (cross-module wiring): Module 14. DDD (module per bounded context): Module 46. Scalable apps: Module 47. CI/CD: Module 50.

Prerequisites

44 Feature First Architecture, 40 Clean Architecture, 14 Dependency Injection, Dart package/pubspec basics.

What you'll be able to do after this module

- Explain packages as compiler-enforced boundaries and decide when modularization is justified.
- Structure a monorepo and manage it with melos (path deps, bootstrap, scripts).
- Design module public APIs (exports/barrels), contract packages, and an acyclic dependency graph.
- Reason about build speed, versioning, ownership, and CI implications of modules.
- Lay out a modular monorepo skeleton (app + feature packages + core/contracts).

Capstone

Modular monorepo skeleton: A melos workspace with an `app` package depending on feature packages (`feature_auth`, `feature_cart`) + shared packages (`core`, `contracts`), each a Clean slice with a controlled public API (`export`s), an acyclic dependency graph (features → core/contracts, never each other), cross-module wiring via contracts + DI, and workspace scripts (bootstrap/analyze/test) — with a documented "when to modularize" rationale.

Summary

Modular architecture makes features **packages** with **compiler-enforced boundaries**, managed in a **monorepo (melos)**: hard isolation, incremental/parallel builds, independent versioning/ownership, cross-app reuse. Modules expose controlled public APIs and interact via contract packages + DI on an acyclic graph. Powerful but with overhead — adopt when scale (build time, team count, reuse) justifies it.

Modular Fundamentals (Packages as Hard Boundaries)

> Modular architecture turns each feature (and `core`) into a **separate Dart/Flutter package** so boundaries are **enforced by the compiler/build system**, not by discipline: a package can only use what another package `export`**s**, and if it doesn't declare a dependency in `pubspec.yaml`, it **can't import it at all**. This upgrades feature-first's *lint-enforced* boundaries to *build-enforced* ones, and adds **incremental builds, independent versioning, ownership, and reuse** — at the cost of workspace tooling and wiring overhead. So the core question isn't *how* but **when**: modularize when scale (build time, team count, reuse) justifies the overhead.

Introduction

This file establishes what a "module" is (a package), why package boundaries are qualitatively stronger than folder boundaries, the concrete benefits, and — crucially — the **when-to-modularize** decision. It's the conceptual base for the monorepo/contracts/build files.

Why this concept exists

Feature-first folders keep boundaries by convention + lints, which erode silently as teams grow. Packages make boundaries **structural**: the dependency must be declared and only public API is reachable. This structural enforcement, plus per-package builds and versioning, is what large, multi-team codebases need — but it's overhead a small app doesn't.

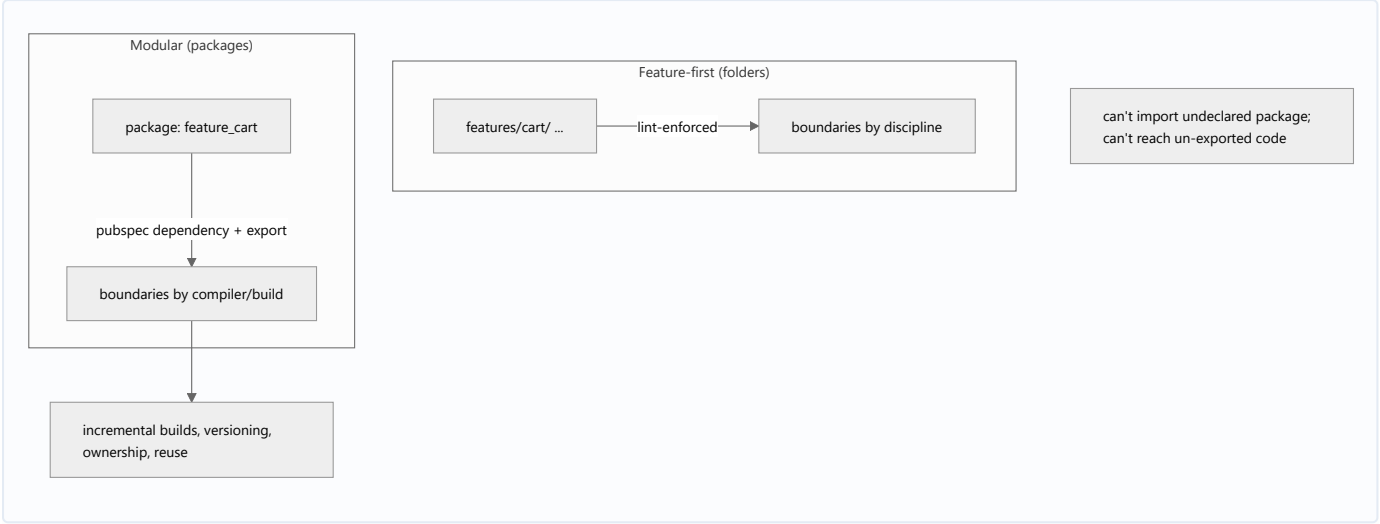
Real-world analogy

Feature-first folders are **rooms in an open-plan office** — labeled zones, but anyone can wander in (discipline/lints keep order). Modules are **separate offices with locked doors and a reception desk (public API)**: you can only enter through reception, and to visit at all you need a listed appointment (declared dependency). Building 2 offices is easy; running 40 needs a facilities system (melos) — and you only build offices when the open plan gets too crowded.

Problem Statement

Decide whether to modularize a growing feature-first app: weigh compiler-enforced boundaries + build speed + team autonomy against tooling/wiring overhead, and identify the signals that justify it. You'll articulate packages-as-boundaries and a when-to-modularize rubric.

Internal Working



- **A module = a package**: each feature/core becomes a Dart/Flutter package (its own `pubspec.yaml`, `lib/`, tests). In a monorepo they're **path dependencies** (02_monorepo_and_melos.md).
- **Boundaries become structural (the key upgrade)**:
  - **Declared dependencies**: a package can only import another if it lists it in `pubspec.yaml` — no accidental cross-feature imports.
  - **Public API only**: consumers can only use what a package `export`**s** from its main library; everything else is effectively private (03_module_boundaries_and_contracts.md).
  - This is **compiler/build-enforced**, not lint-enforced — boundaries can't erode by carelessness.
- **Benefits over feature-first folders**:
  - **Enforced isolation** (structural, not convention).
  - **Incremental/parallel builds**: only changed packages (+ dependents) rebuild/re-test → faster CI at scale (04_build_versioning_and_teams.md).
  - **Independent versioning + ownership**: teams own packages with their own version/changelog.
  - **Reuse across apps**: a package (e.g., `core`, `design_system`) can be shared by multiple apps.
- **Costs (be honest)**:
  - **Tooling overhead**: workspace management (melos), `pubspec` maintenance, bootstrapping.
  - **Wiring overhead**: cross-module interaction via contracts + DI (more ceremony than a folder import).
  - **Refactor friction**: moving code across package boundaries is heavier than across folders.
- **When to modularize (the real question)** — signals that justify the overhead:
  - **Build/test times hurting** (large single package → slow CI).

    - **Multiple teams** needing hard isolation + independent ownership.
    - **Reuse** across apps/white-label products.
    - **Very large codebase** where folder discipline no longer holds.
    - **Not** for small/single-team apps → feature-first folders suffice; **grow into** modules from a clean feature-first base (Module 44).
- **Same internals**: each module is still a **Clean slice** (domain/data/presentation — Module 40); modularization changes the **boundary mechanism**, not the internal architecture.

Memory Representation

Not runtime — a **package dependency graph** (must be **acyclic**). Each node is a package with a public API; edges are declared `pubspec` dependencies. The invariant: features → core/contracts, never feature → feature.

Compiler Behavior

The whole point: the compiler/pub resolves only declared dependencies and only exported symbols are visible — cross-package access is **structurally impossible** unless intended. Undeclared import = compile error.

Runtime Behavior

No runtime change vs feature-first — same code runs; DI composition root still wires modules at startup. Packaging affects **build/tooling**, not runtime behavior.

Flutter Engine Behavior

Not applicable.

Dart VM Behavior

Package granularity enables **incremental compilation** — changed packages (and dependents) recompile, others are cached — improving build times at scale.

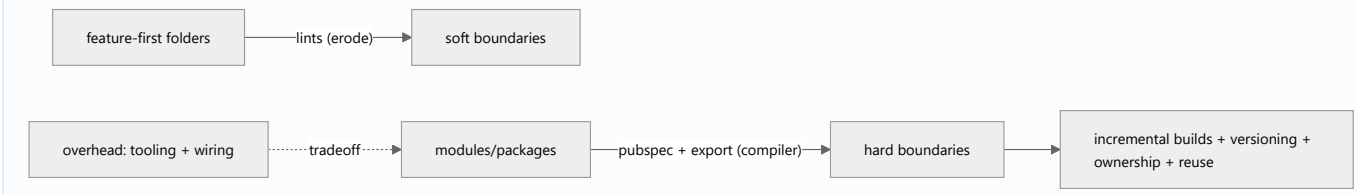
Examples

```yaml
# packages/feature_cart/pubspec.yaml — declares ONLY what it may use
name: feature_cart
dependencies:
  core: { path: ../core }          # allowed
  contracts: { path: ../contracts } # allowed
  # feature_auth is NOT listed -> importing it is a COMPILE ERROR (structural boundary)
```

```dart
// packages/feature_cart/lib/feature_cart.dart — the PUBLIC API (only these are visible)
export 'src/presentation/cart_screen.dart';
export 'src/cart_di.dart' show registerCart;   // controlled surface
// src/** internals are NOT exported -> other packages can't reach them
```

```
When-to-modularize checklist:
  slow builds/CI on a big package ..... modularize (incremental builds)
  multiple teams needing isolation/ownership ... modularize
  reuse across apps / white-label ..... modularize (shared packages)
  small, single-team app ..... DON'T (feature-first folders)
  ->  grow INTO modules from a clean feature-first base
```

Diagrams



Common Mistakes

Mistake	Why it's a problem	Fix
Modularizing a small/single-team app	Overhead without payoff	Feature-first folders; grow into modules
Feature package depending on another feature	Coupling, cyclic risk	Depend on core/contracts only (acyclic)
Exporting everything (`src/**`)	No real boundary	Export only the public API
Big-bang split into packages	Risky/disruptive	Extract mature features incrementally
Different internal architecture per module	Inconsistent	Keep each a Clean slice
Ignoring build/versioning benefits	Miss the payoff	Leverage incremental builds + per-package versions
Circular package deps	Won't resolve/build	Keep the graph acyclic (star around core)

Best Practices

- Treat a **module as a package** with **compiler-enforced boundaries** (declared deps + controlled `export`s); keep the package graph **acyclic** (features → core/contracts, never each other).
- **Modularize when scale justifies it** (slow builds, multiple teams, reuse) — **not** for small/single-team apps; **grow into** modules from a clean feature-first base.
- Keep each module a **Clean slice** internally; expose a **minimal public API**; wire cross-module via **contracts + DI**.
- Leverage the payoffs deliberately: **incremental/parallel builds, independent versioning, ownership, reuse**; accept the **tooling/wiring overhead** consciously.

Performance

Runtime-neutral; the performance win is **build/CI time** via incremental compilation (only changed packages rebuild/re-test) and parallelism — significant on large codebases, negligible on small ones (where overhead dominates). This is the quantitative half of the when-to-modularize decision (04_build_versioning_and_teams.md).

Advantages / Disadvantages

- **+** Compiler-enforced isolation, incremental/parallel builds, independent versioning/ownership, cross-app reuse, team autonomy.
- **−** Tooling (workspace/melos) + wiring (contracts/DI) + refactor overhead; overkill for small apps; requires acyclic-graph discipline.

Interview Questions

1. 🟢 **What is a module in modular architecture?** — A separate Dart/Flutter package (own pubspec/lib/tests), typically a feature or shared library, with a controlled public API.
2. 🟢 **How are package boundaries stronger than folder boundaries?** — They're compiler/build-enforced: you can only import declared dependencies and only their exported symbols — no reliance on discipline/lints.
3. 🟡 **What are the main benefits of modularizing?** — Enforced isolation, incremental/parallel builds, independent versioning/ownership, and cross-app reuse.
4. 🟡 **What are the costs?** — Workspace tooling, cross-module wiring via contracts/DI, and heavier refactors across boundaries.
5. 🟡 **When should you modularize (and not)?** — When build times hurt, multiple teams need isolation, or reuse is needed; not for small/single-team apps — grow into it from feature-first.
6. 🔴 **Does modularization change a module's internal architecture?** — No — each module is still a Clean slice; only the boundary mechanism (package vs folder) changes.
7. 🔴 **Why must the package graph be acyclic?** — Cyclic package dependencies can't be resolved/built; keep features depending on core/contracts (star), never on each other.

Senior Engineer Tips

- Modularize reactively on real signals (CI pain, team count, reuse), not aspirationally; premature package splits add tooling/wiring cost with no payoff on a small app.
- Keep the package graph a star around core/contracts and export only the public API; the two ways modular apps rot are cyclic deps and packages that export their internals.
- Build a clean feature-first base first so extraction is mechanical; modules are just feature slices with a locked door — the internal Clean architecture is unchanged.

Architect Perspective

Modular architecture is feature-first with **structural enforcement**: packages make boundaries a build-time fact, enabling incremental builds, independent versioning/ownership, and reuse. It's the same cohesion/DIP discipline, now enforced by the compiler and rewarded by tooling — but it carries overhead, so it's a **scale decision**, not a default. Adopted at the right time (build/team/reuse pressure) from a clean feature-first base, it's how large Flutter codebases stay buildable, ownable, and evolvable — and it aligns naturally with a module-per-bounded-context DDD split (Module 44, Module 46, Module 47).

Summary

- A module is a package; boundaries become compiler/build-enforced (declared deps + exports only) — stronger than folder/lint boundaries.
- Benefits: incremental/parallel builds, independent versioning/ownership, reuse; costs: tooling + wiring + refactor overhead.
- Modularize when scale justifies (build time/teams/reuse), not for small apps; keep modules Clean slices on an acyclic star-around-core graph.

Revision Notes

- Module = package (pubspec/lib/tests); boundaries compiler-enforced (declared deps + `export` public API only) vs feature-first lint boundaries.
- Benefits: incremental/parallel builds, versioning, ownership, cross-app reuse. Costs: melos tooling, contract/DI wiring, refactor friction.
- When: slow builds / multi-team / reuse → modularize (grow from feature-first); not for small/single-team. Acyclic star (features→core/contracts). Internals still Clean.

Practice Questions

1. Why are package boundaries qualitatively stronger than folder boundaries?
2. List concrete benefits and costs of modularizing.
3. Give the signals for when to (and not to) modularize.

Coding Questions

1. Write a feature package `pubspec.yaml` declaring only core/contracts deps.
2. Define a package's public API via `export`s (hiding `src/**`).
3. Draw the acyclic package graph for app + 2 features + core/contracts.

Mini Project

Modularization decision + boundary sketch (Flutter): For a growing feature-first app, write a when-to-modularize rationale (signals present/absent) and sketch the target package graph (app → feature packages → core/contracts, acyclic star), including one feature package's `pubspec.yaml` (declared deps only) and its public-API `export`s (hiding internals). Acceptance: packages-as-hard-boundaries explained; benefits/costs weighed; decision justified by real signals; acyclic star graph; example pubspec (declared deps only) + minimal public API; internals stay Clean.

Monorepo & Melos

> Modular Flutter apps live in a **monorepo**: one repository holding the `app` package plus many local packages (`core` , `contracts` , `feature_*`), wired together with **path dependencies**. **Melos** is the standard tool to manage this workspace — it **bootstraps** (links local packages), runs **scripts across packages** (`melos run analyze/test`), handles **selective/incremental** execution (only changed packages), and coordinates **versioning/publishing**. The monorepo keeps everything **atomically versioned + refactorable in one PR** while packages give you **enforced boundaries + incremental builds**.

Introduction

This file covers the practical mechanics: monorepo layout, path dependencies, and using melos to bootstrap, script, scope, and version the workspace. It's the tooling layer under the fundamentals (01_modular_fundamentals.md).

Why this concept exists

Many packages need coordination: linking local deps, running lint/test/build across all of them, doing only what changed, and versioning together. Doing this by hand (per-package `pub get` , manual script loops) is painful and error-prone. Melos + a monorepo standardize it — you get package isolation without multi-repo overhead (no cross-repo version juggling; atomic cross-package changes).

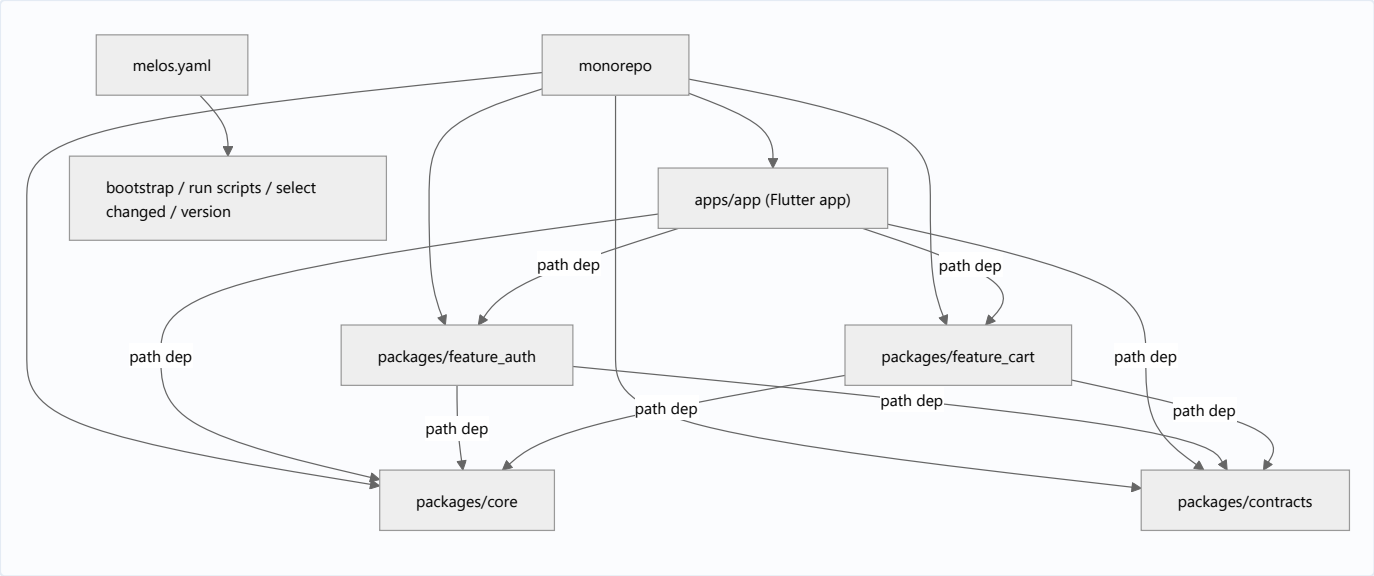
Real-world analogy

A monorepo is a **single campus with many specialized buildings** (packages) connected by internal walkways (path deps). **Melos** is the **campus operations team**: it hooks up utilities to every building (bootstrap), runs campus-wide drills (scripts across packages), only services the buildings that changed (selective runs), and coordinates campus-wide upgrades (versioning). One campus (repo) beats scattered buildings across the city (multi-repo) for coordinated work.

Problem Statement

Set up a monorepo for an `app` + `core` + `contracts` + `feature_auth` + `feature_cart` , link them via path deps, and use melos to bootstrap, analyze/test across packages (only changed ones in CI), and version them. You'll structure the workspace + melos config + scripts.

Internal Working



- **Monorepo layout**: one repo, e.g. `apps/app/` (the Flutter application) + `packages/{core, contracts, feature_auth, feature_cart, design_system, ...}`. Each is a normal Dart/Flutter package with its own `pubspec.yaml` / `lib/` / `test/`.
- **Path dependencies**: packages depend on each other via `path:` deps in `pubspec.yaml` (e.g., `feature_cart` → `core`, `contracts`). The **app** depends on the feature packages + core. The graph stays **acyclic** (features → core/contracts, never each other — 03_module_boundaries_and_contracts.md).
- **Melos** (`melos.yaml` at the root): declares `packages:` globs and **scripts**.
  - **Bootstrap** (`melos bootstrap` / `melos bs`): resolves + links all local packages (runs `pub get` everywhere, wiring path deps) — the setup step after clone/dep changes.
  - **Scripts** (`melos run <name>`): run a command **across packages** — `analyze`, `test`, `format`, `build_runner`, custom. Define once, run everywhere.
  - **Selective/changed execution**: run only **packages that changed** (or depend on changed ones) via `--since`/filters — the basis for **fast CI** (04_build_versioning_and_teams.md).
  - **Filtering/scoping**: target subsets (`--scope`, `--ignore`, by directory/flag) — e.g., only feature packages, or only those with tests.
  - **Versioning/publishing** (`melos version` / `publish`): conventional-commit-based version bumps + changelogs across packages (for shared/published packages).
- **Why monorepo over multi-repo**: **atomic changes** (a cross-package refactor is one PR), **no version juggling** between repos, **shared tooling/CI**, easy local linking — while packages still give **enforced boundaries**. (Multi-repo trades this for stronger org isolation + independent release cadence — rarely needed for one app.)
- **CI integration**: CI runs `melos bootstrap` then scoped `melos run analyze/test` on **changed** packages → incremental, fast pipelines (Module 50).

- `.gitignore` **/tooling**: ignore per-package `.dart_tool` /build outputs; commit `melos.yaml` + lockfiles per policy.

Memory Representation

Not runtime — a **repo tree + a package graph** melos operates on. Melos holds the workspace config (package globs, scripts) and computes changed-package sets for selective runs.

Compiler Behavior

Path deps + `pub get` (via bootstrap) make local packages resolvable; the compiler still enforces declared-deps + exports (boundaries). Incremental compilation caches unchanged packages.

Runtime Behavior

No runtime effect — this is dev/build/CI tooling. The built app is identical; only how it's assembled/tested is coordinated.

Flutter Engine Behavior

Not applicable.

Dart VM Behavior

Incremental builds: only changed packages (+ dependents) recompile; melos scoping mirrors this for analyze/test to keep CI fast.

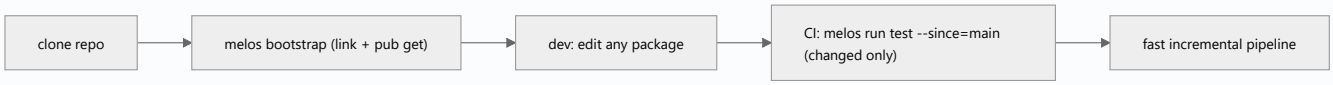
Examples

```yaml
# melos.yaml (repo root)
name: my_app_workspace
packages:
  - apps/**
  - packages/**
scripts:
  analyze: { run: dart analyze, exec: { concurrency: 5 } }
  test:    { run: flutter test, exec: { concurrency: 5 }, packageFilters: { dirExists: test } }
  gen:     { run: dart run build_runner build -d }
```

```yaml
# packages/feature_cart/pubspec.yaml — path deps to local packages (acyclic)
name: feature_cart
dependencies:
  core: { path: ../core }
  contracts: { path: ../contracts }
  # NOT feature_auth (features never depend on each other)
```

```
# Workflow
melos bootstrap                    # link + pub get all packages
melos run analyze                  # analyze across all packages
melos run test --since=origin/main  # test ONLY changed packages (fast CI)
melos version                      # bump versions + changelogs (conventional commits)
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Manual per-package `pub get` /scripts	Error-prone, slow	Use `melos bootstrap` / `run`
Cyclic path deps	Won't resolve	Keep graph acyclic (features→core/contracts)
Running everything every CI build	Slow pipelines	Scope with `--since` /filters (changed only)
Feature package path-depending on another feature	Coupling	Depend only on core/contracts
Multi-repo for one app	Version juggling, no atomic refactors	Monorepo (melos)
Not bootstrapping after dep changes	Broken links	Re-run `melos bootstrap`
Publishing versioning on purely-local packages	Unneeded	Version/publish only shared/published packages

Best Practices

- Use a **monorepo** (`apps/` + `packages/`) with **path dependencies**, kept **acyclic** (features → core/contracts); manage it with **melos**.

- **Bootstrap** after clone/dep changes; define **workspace scripts** (analyze/test/format/gen) once and run across packages; **scope to changed packages** (`--since`) for fast CI.
- Prefer **monorepo over multi-repo** for one app (atomic cross-package refactors, no version juggling, shared tooling); version/publish only **shared/published** packages.
- Keep tooling consistent (root analysis options, ignored build outputs); integrate melos into **CI** for incremental pipelines (Module 50).

Performance

Melos + incremental/selective execution is the **build-time** payoff: CI runs analyze/test only on changed packages (+ dependents), and Dart caches unchanged packages — pipelines scale sub-linearly with codebase size. Runtime is unaffected.

Advantages / Disadvantages

- **+** Coordinated multi-package workflow, atomic cross-package changes, incremental/selective CI, shared tooling, easy local linking + versioning.
- **−** Learning/tooling overhead (melos config, bootstrap step), monorepo size/CI setup, discipline to keep the graph acyclic + scripts maintained.

Interview Questions

1. 🟢 **What is a monorepo in this context?** — One repository holding the app + many local packages (core/contracts/features) wired via path dependencies.
2. 🟢 **What does melos do?** — Manages the workspace: bootstraps (links local packages), runs scripts across packages, scopes to changed packages, and coordinates versioning/publishing.
3. 🟡 **What is `melos bootstrap` for?** — Resolving and linking all local path dependencies (`pub get` everywhere) so packages reference each other correctly.
4. 🟡 **How does melos speed up CI?** — Selective execution: run analyze/test only on packages that changed (or depend on changed ones) via `--since` /filters.
5. 🟡 **Why monorepo over multi-repo for one app?** — Atomic cross-package refactors in one PR, no cross-repo version juggling, and shared tooling/CI — while packages still enforce boundaries.
6. 🔴 **How do packages depend on each other locally?** — Via `path:` dependencies in `pubspec.yaml` , kept acyclic (features → core/contracts, never each other).
7. 🔴 **What must remain true of the package graph, and why?** — Acyclic — cyclic path deps can't be resolved/built; keep a star around core/contracts.

Senior Engineer Tips

- Wire CI to `melos bootstrap` + scoped `melos run test --since=main` from the start; running the whole workspace every build erases modular architecture's build-speed payoff.
- Enforce the acyclic star (features → core/contracts) in path deps and review; a single feature→feature path dep starts the slide back to a tangled monolith.
- Keep one repo for one app (monorepo) unless org/release-cadence pressures truly demand multi-repo — the coordination cost of multi-repo rarely pays off for a single product.

Architect Perspective

The monorepo + melos is the operational substrate of modular architecture: it makes many packages practical to develop, test, and version together while preserving hard boundaries and unlocking incremental CI. It gives the best of both worlds — package isolation (enforced boundaries, ownership, reuse) with monorepo cohesion (atomic changes, shared tooling) — and is the platform on which enterprise-scale and CI/CD strategies are built (01_modular_fundamentals.md, 04_build_versioning_and_teams.md, Module 50).

Summary

- Monorepo: `apps/app` + `packages/{core,contracts,feature_*}` linked by path deps (acyclic star), managed with melos.
- Melos: bootstrap (link/`pub get`), run scripts across packages, scope to changed packages (`--since`) for fast CI, coordinate versioning.
- Monorepo beats multi-repo for one app (atomic refactors, no version juggling, shared tooling); runtime-neutral, build-time win.

Revision Notes

- Layout: `apps/app` + `packages/{core, contracts, feature_*}`; path deps in pubspec (acyclic, features→core/contracts).
- Melos: `melos.yaml` (packages globs + scripts); `bootstrap` (link/pub get), `run <script>` across packages, `--since`/filters (changed only), `version` / `publish`.
- Monorepo > multi-repo for one app (atomic changes, shared tooling); CI = bootstrap + scoped run; keep graph acyclic; runtime-neutral.

Practice Questions

1. What does `melos bootstrap` do and when do you run it?
2. How do you make CI only test changed packages?

3. Why prefer a monorepo over multi-repo for a single app?

Coding Questions

1. Write a `melos.yaml` with analyze/test scripts and package globs.
2. Add path deps to a feature package (core/contracts only, acyclic).
3. Show the CI commands for bootstrap + changed-only tests.

Mini Project

Melos monorepo setup (Flutter): Lay out a monorepo (`apps/app` + `packages/core` , `contracts` , `feature_auth` , `feature_cart`) with path deps (acyclic star), a `melos.yaml` (packages globs + analyze/test/gen scripts), and CI commands (`bootstrap` + `run test --since=main`). Ensure no feature→feature path dep. Acceptance: monorepo layout + path deps (acyclic); melos scripts defined; bootstrap + selective/changed CI commands; features depend only on core/contracts; workspace-manageable with melos.

Module Boundaries & Contracts

> A module's boundary is its **public API** — precisely the symbols it `export`**s** from its main library (everything under `lib/src/` is private by convention). Cross-module interaction goes through a `contracts` **package**: shared interfaces/DTOs/events that feature packages **implement** or **depend on**, so features never depend on each other — they depend on the neutral contract. This keeps the **package graph acyclic** (features → contracts/core; a thin `app`/composition package wires the concretes) and makes modules independently buildable, replaceable, and reusable.

Introduction

This file covers designing module public APIs (`export` s + `lib/src/` privacy), the **contracts package** pattern for cross-module interaction, and keeping the dependency graph acyclic. It's the boundary-design layer over fundamentals (01_modular_fundamentals.md) and the package-level formalization of feature-first boundaries (Module 44).

Why this concept exists

Packages enforce *that* boundaries exist, but you must *design* them: what's public vs private, and how modules cooperate without depending on each other. A minimal public API + a shared contracts package (dependency inversion at the package level) is what keeps modules loosely coupled and the graph acyclic — the difference between clean modules and a package spaghetti.

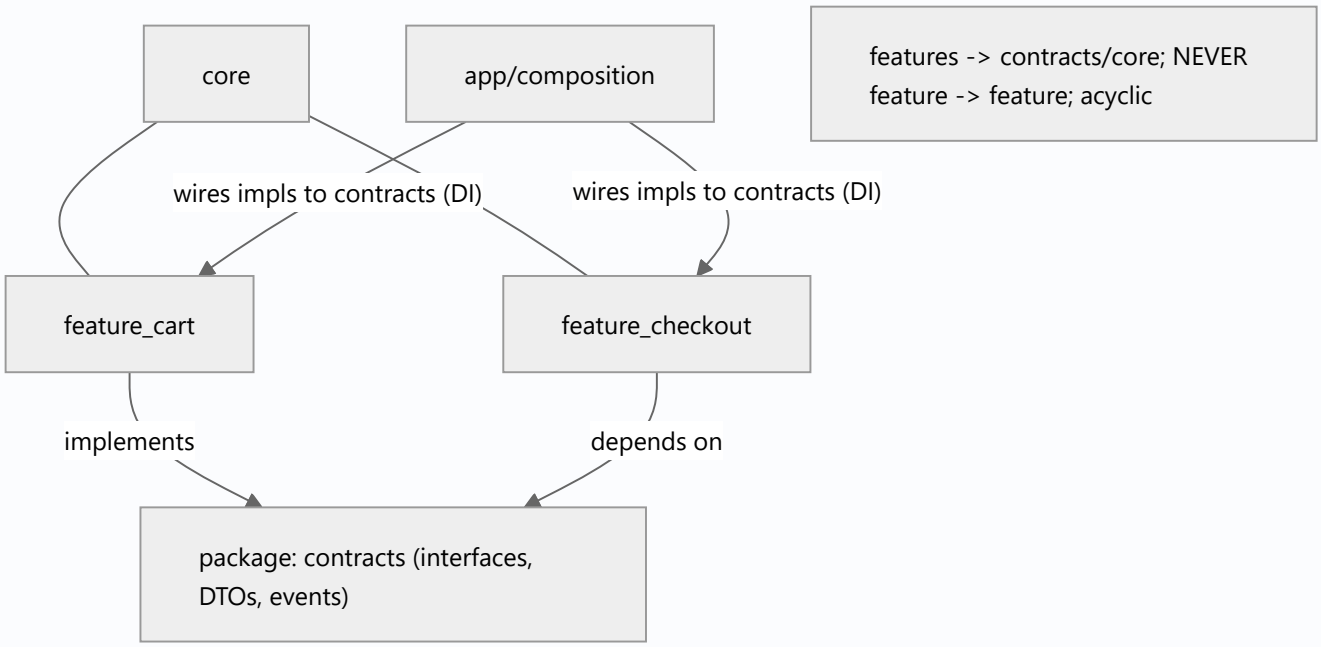
Real-world analogy

Each module is a **company with a published catalog** (public API via `export` s) and **private operations** (`lib/src/`) no outsider sees. Companies don't call each other's internal extensions; they transact through **industry-standard contracts** (the contracts package) — a "supplier agreement" spec that any company can fulfill or rely on. A **general contractor** (the app/composition package) hires specific companies to fulfill specific contracts, so no supplier needs to know another supplier exists.

Problem Statement

Design `feature_cart` 's public API, let `feature_checkout` use the cart without depending on the cart package, via a `contracts` package (a `CartFacade` interface + shared `Cart` DTO), and keep the whole graph acyclic. You'll define exports, the contracts package, and the wiring.

Internal Working



- **Public API via `export` + `lib/src/`**: put implementation under `lib/src/` (private by convention) and expose only intended symbols from the package's **main library** (`lib/<pkg>.dart`) via `export ... show ...`. Consumers can only use exported symbols — the **API surface is deliberate + minimal**. Prefer a **barrel** that curates exactly the public API.
- **Contracts package (the key pattern)**: a small, **dependency-light** package holding **shared interfaces, DTOs, and events** used across modules (`CartFacade`, `AuthGateway`, `Cart` DTO, `UserLoggedOut` event). It's the **neutral ground**:
  - **Providers** (e.g., `feature_cart`) **implement** a contract.
  - **Consumers** (e.g., `feature_checkout`) **depend on the contract** — not on the provider package.
  - This is **Dependency Inversion at package scale** (Module 04): both sides depend on the abstraction (contracts), so features don't depend on each other.
- **Acyclic graph (star)**: features depend on **contracts + core**, never on **each other**. A thin `app` **/composition package** depends on all features to **wire implementations to contracts** (DI composition root — Module 14). Result: a **DAG** — features are leaves, contracts/core are roots, app is the top wiring node.
- **Where shared types live**:
  - **Shared domain interfaces/DTOs/events → contracts**.
  - **Shared infra + design system + value objects → core** (Module 44).
  - A feature's own entities/DTOs stay **private** to that feature.

- **Communication mechanisms** (same as feature-first, now across packages): **contract interfaces** (direct capability, DI-wired), **events** (decoupled reactions via a contracts/core event bus), **routes** (navigate across modules without importing widgets — Module 13).
- **Versioning the contract**: the contracts package is a **stable, slowly-changing API**; changing it ripples to all consumers, so treat it as a **published interface** (version carefully — 04_build_versioning_and_teams.md).
- **Enforcement**: package deps + `lib/src/` privacy are **compiler-enforced**; the acyclic rule is maintained by not adding feature→feature deps (reviewable in pubspecs).

Memory Representation

Not runtime — a **DAG of packages** with public-API surfaces. Contracts is a root (no deps on features); features implement/consume it; app wires concretes to abstractions at the composition root. Private code lives in `lib/src/` (unreachable across packages).

Compiler Behavior

Only exported symbols are importable; `lib/src/` is inaccessible from other packages. A consumer depending on `contracts` compiles without knowing which package implements it — the inversion. Undeclared/cyclic deps fail resolution.

Runtime Behavior

DI (in the app/composition package) binds a provider's implementation to the contract interface; consumers call the interface, resolved to the concrete at runtime — no static consumer→provider import.

Flutter Engine Behavior

Not applicable (routing aside — navigate across modules by route).

Dart VM Behavior

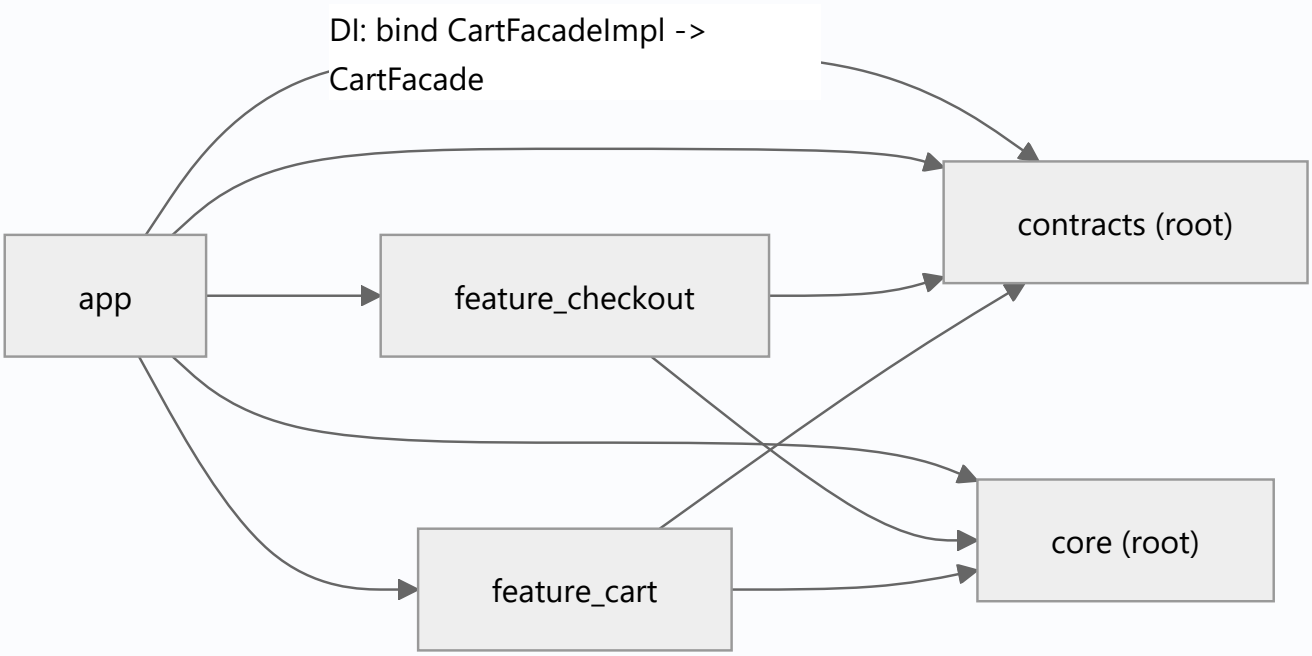
Package granularity + minimal public APIs support incremental builds; a stable contracts package changing forces rebuilds of dependents (reason to keep it stable).

Examples

```
// packages/contracts/lib/contracts.dart — neutral shared abstractions (no feature deps)
export 'src/cart_facade.dart';   // abstract class CartFacade { Future<CartDto> current(); }
export 'src/cart_dto.dart';      // shared DTO
export 'src/events.dart';        // e.g., UserLoggedOut

// packages/feature_cart/lib/feature_cart.dart — public API only
export 'src/presentation/cart_screen.dart';
export 'src/cart_di.dart' show registerCart;   // src/** internals stay private

// packages/feature_cart/lib/src/cart_facade_impl.dart — cart IMPLEMENTS the contract
class CartFacadeImpl implements CartFacade {     // depends on contracts, not on checkout
  @override Future<CartDto> current() async => /* ... */;
}

// packages/feature_checkout/lib/src/checkout_vm.dart — depends on the CONTRACT, not on cart
class CheckoutVm {
  final CartFacade cart;                          // from contracts package
  CheckoutVm(this.cart);
}
// app/lib/composition.dart — wires impl -> contract (only the app knows both)
// di.registerSingleton<CartFacade>(() => CartFacadeImpl(...));  // provided by feature_cart
```

```
# feature_checkout/pubspec.yaml — depends on contracts + core, NOT on feature_cart
dependencies:
  contracts: { path: ../contracts }
  core: { path: ../core }
```

Diagrams



Common Mistakes

Mistake	Why it's bad	Fix
Feature package depends on another feature	Coupling, cyclic risk	Depend on contracts (both sides)
Exporting `src/**`	No real boundary	Export only the public API
Fat/unstable contracts package	Ripples to all consumers	Keep contracts small + stable
Shared DTO living in a feature	Others must depend on that feature	Put shared DTOs in contracts
Cyclic package deps	Won't build	Acyclic star (features→contracts/core)
Wiring impls inside feature packages	Features learn of each other	Wire in the app/composition package
No composition package	Nowhere to bind impls→contracts	Thin app/composition root wires DI

Best Practices

- Expose a **minimal public API** (`export ... show` ; keep impl in `lib/src/`); curate it via a barrel — the boundary *is* the export list.
- Put **shared interfaces/DTOs/events in a `contracts` package**; providers **implement** it, consumers **depend on it** — features **never** depend on each other (package-level DIP).
- Keep the **graph acyclic (star)**: features → contracts/core; a **thin app/composition package** wires implementations to contracts via **DI**.

- Treat **contracts as a stable published API** (version carefully); shared infra/design system/value objects → **core**; feature-private types stay private.

Performance

Minimal public APIs + stable contracts maximize incremental-build benefit (fewer forced rebuilds); a churning contracts package rebuilds all dependents (keep it stable). Runtime-neutral; the payoff is build isolation + change locality.

Advantages / Disadvantages

- **+** Deliberate/minimal surfaces, decoupled features (no feature→feature deps), acyclic graph, replaceable/reusable modules, DIP at scale.
- **−** Contracts-package design + versioning care, composition-package wiring, export/ `src` discipline, more packages to maintain.

Interview Questions

1. 🟢 **What is a module's public API?** — Exactly the symbols it `export`s from its main library; everything in `lib/src/` is private and unreachable from other packages.
2. 🟢 **What is the contracts package for?** — Holding shared interfaces/DTOs/events so features interact through neutral abstractions instead of depending on each other.
3. 🟡 **How do two features cooperate without depending on each other?** — Both depend on the contracts package: the provider implements the contract, the consumer depends on it; the app wires the impl to the interface via DI (package-level DIP).
4. 🟡 **Why must the package graph be acyclic, and what shape is ideal?** — Cyclic deps can't build; ideal is a star/DAG — features → contracts/core, app → everything for wiring.
5. 🟡 **Where do shared DTOs/entities live?** — Shared cross-module DTOs/interfaces/events in `contracts`; shared infra/value objects in `core`; feature-private types stay in the feature.
6. 🔴 **Why keep the contracts package small and stable?** — Every consumer depends on it, so changes ripple widely and force rebuilds — treat it as a versioned published API.
7. 🔴 **Why wire implementations in the app/composition package, not in features?** — So features never learn of each other's concretes; only the app knows both sides and binds impl→contract.

Senior Engineer Tips

- Design the export list deliberately (small, `show`-listed) and keep everything else in `lib/src/`; a package that exports its internals has a boundary in name only.

- Route all cross-feature needs through the contracts package and wire concretes in the app; the moment `feature_a` lists `feature_b` in its pubspec, you've broken the acyclic star.
- Guard the contracts package like a public API — small, stable, versioned; it's the highest-leverage (and highest-blast-radius) package in the repo.

Architect Perspective

Module boundaries + contracts are Dependency Inversion realized at the package level: minimal public APIs define surfaces, and a neutral contracts package lets features depend on abstractions rather than each other, keeping the graph an acyclic star with a thin composition root wiring concretes. This is what makes modules independently buildable, replaceable, and reusable — the structural backbone for multi-team, multi-app, and DDD (module-per-bounded-context) architectures (01_modular_fundamentals.md, Module 46, Module 04).

Summary

- A module's boundary = its `export`ed public API (`lib/src/` private); curate it minimally via a barrel.
- Cross-module interaction via a `contracts` package (shared interfaces/DTOs/events): providers implement, consumers depend on it — features never depend on each other (package DIP).
- Keep the graph acyclic (star: features→contracts/core; app wires impls→contracts via DI); treat contracts as a stable versioned API.

Revision Notes

- Public API = `export ... show` from `lib/<pkg>.dart`; impl in `lib/src/` (private cross-package). Curate via barrel.
- `contracts` package = shared interfaces/DTOs/events; provider implements, consumer depends on it (package-level DIP); app/composition wires impl→contract via DI.
- Acyclic star: features→contracts/core, never feature→feature; app→all (wiring). Shared infra/value objects→core; contracts small+stable+versioned.

Practice Questions

1. How do you define and minimize a package's public API?
2. How does a contracts package let features cooperate without coupling?
3. Why must implementations be wired in the app, not in feature packages?

Coding Questions

1. Define a `contracts` package with a `CartFacade` + shared DTO, exported.
2. Implement it in `feature_cart`, depend on it from `feature_checkout` (no feature→feature dep).
3. Wire the impl→contract binding in the app composition root; show the acyclic pubspecs.

Mini Project

Contracts-based module boundaries (Flutter): Create a `contracts` package (`CartFacade` interface + `CartDto` + an event), have `feature_cart` implement it (public API via `export`s, internals in `lib/src/`), and `feature_checkout` depend only on `contracts` + `core` (no dep on `feature_cart`); wire the impl→contract in the `app` composition root via DI. Verify the package graph is an acyclic star. Acceptance: minimal public APIs (`export` / `src` privacy); cross-feature interaction via contracts (package DIP); no feature→feature deps; app wires concretes; acyclic star graph; contracts kept small/stable.

Build, Versioning & Teams

> The concrete payoffs of modularization are operational: **build/CI speed** (incremental — only changed packages and their dependents rebuild/re-test), **independent versioning** (per-package versions + changelogs, so shared packages evolve on their own cadence), **team ownership** (a team owns a package, its API, and its release), and **cleaner CI** (scoped runs via melos `--since`). These come from the package graph — but they demand discipline: a **stable, low-churn** `core` / `contracts` (churn there rebuilds everything), **careful contract versioning** (breaking a contract ripples to all consumers), and **clear ownership** to avoid diffuse responsibility.

Introduction

This file covers the operational/organizational dimension of modules: how packages accelerate builds/CI, how versioning works across packages, how ownership maps to packages, and the discipline required (stable shared packages, contract versioning). It's the "why it pays off (and what to watch)" companion to the structural files.

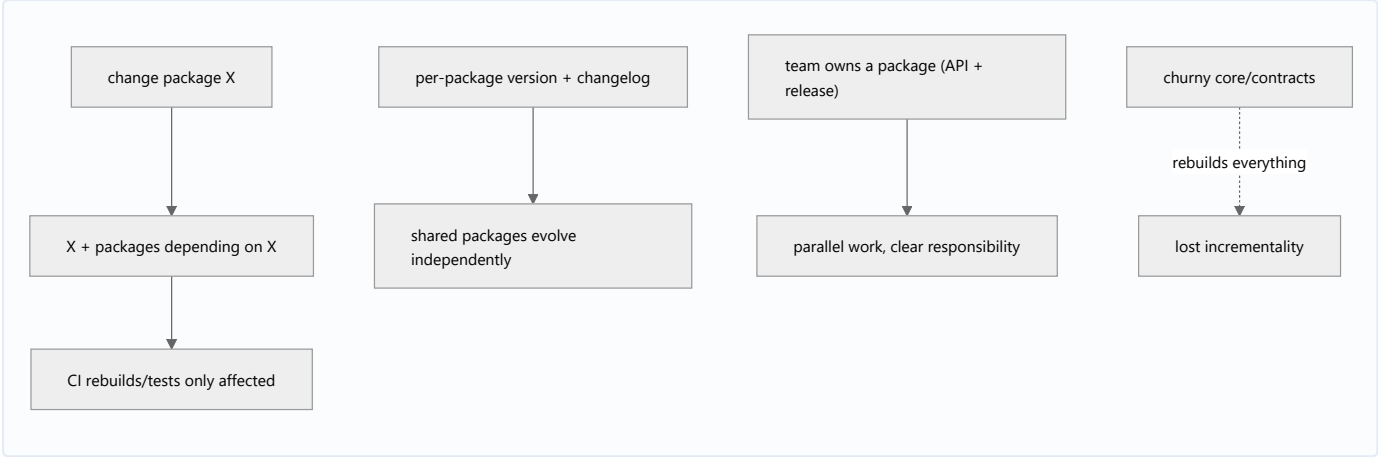
Why this concept exists

Modularization is adopted for these outcomes — faster CI, autonomous teams, controlled evolution — not for aesthetics. Understanding *how* they arise (and their failure modes) lets you realize the payoff rather than pay the overhead for nothing. It's also the quantitative side of the when-to-modularize decision (01_modular_fundamentals.md).

Real-world analogy

Modules are **independent factory lines**: if one line changes, you **retool only that line and the lines that consume its parts** (incremental build), not the whole plant. Each line has its **own model year** (version) and **shift crew** (team). But the **shared parts depot** (core/contracts) must be **rock-stable** — change a shared part spec and **every line retools** (whole-repo rebuild + ripple). Clear line ownership prevents "everyone and no one" responsibility.

Internal Working



- **Incremental build/CI (the headline win)**: with packages, a change recompiles/re-tests only the **changed package + its dependents**, not the whole app. Melos `--since` scopes analyze/test to affected packages (02_monorepo_and_melos.md) → **CI time scales with change size, not repo size**. Leaf features change often + cheaply; roots (core/contracts) rarely.
- **The stability corollary (critical)**: because **everything depends on** `core` / `contracts`, a change there **rebuilds/re-tests everything** — so keep them **stable, small, low-churn**. Volatile shared packages **destroy** the incremental benefit. Depend on stable things (Module 04 SDP).
- **Independent versioning**: each package has its own `version` + `CHANGELOG`. Melos (conventional commits) bumps versions per package. Shared/published packages (`design_system`, `core`) can evolve on their **own cadence**; the app pins versions. (Purely-local path-dep packages may skip formal versioning.)
- **Contract versioning (highest blast radius)**: the `contracts` **package** is a **published API** — a breaking change ripples to **all consumers**. Treat it with **semver discipline**, additive/backward-compatible changes where possible, and coordinated migration for breaks. This is the main coupling risk in a modular repo.
- **Team ownership**: a **package = an ownership unit** — a team owns its code, **public API**, tests, and release. `CODEOWNERS` maps packages to teams; PRs to a package need its owners. Enables **autonomy + parallel work** with **clear responsibility** (vs diffuse ownership of shared folders).
- **CI implications** (Module 50): `melos bootstrap` → scoped `melos run analyze/test --since` → per-package build/test → optional per-package publish. Parallelizable across packages. Caching keyed by package.
- **Governance discipline**: enforce **acyclic graph** (feature→contracts/core only), **contract review** (breaking changes gated), **stable roots** (change core/contracts deliberately), and **ownership** (CODEOWNERS) — else you get slow CI (churny roots), ripple breakages (loose contracts), or ownership gaps.
- **When the payoff is real**: large repos (build pain), many teams (autonomy), shared/published packages (reuse). For small/single-team apps the versioning/ownership machinery is overhead

(01_modular_fundamentals.md).

Memory Representation

Not runtime — a package graph annotated with **versions + owners**, and a CI notion of the **changed set** (affected packages). Incrementality = rebuild/test only the changed set; stability of roots keeps that set small.

Compiler Behavior

Package-granular incremental compilation: unchanged packages are cached; changed packages + dependents recompile. Contract/core changes invalidate broad swaths (hence keep them stable).

Runtime Behavior

None — build/versioning/ownership are dev/CI/org concerns. The shipped app is identical regardless.

Flutter Engine Behavior

Not applicable.

Dart VM Behavior

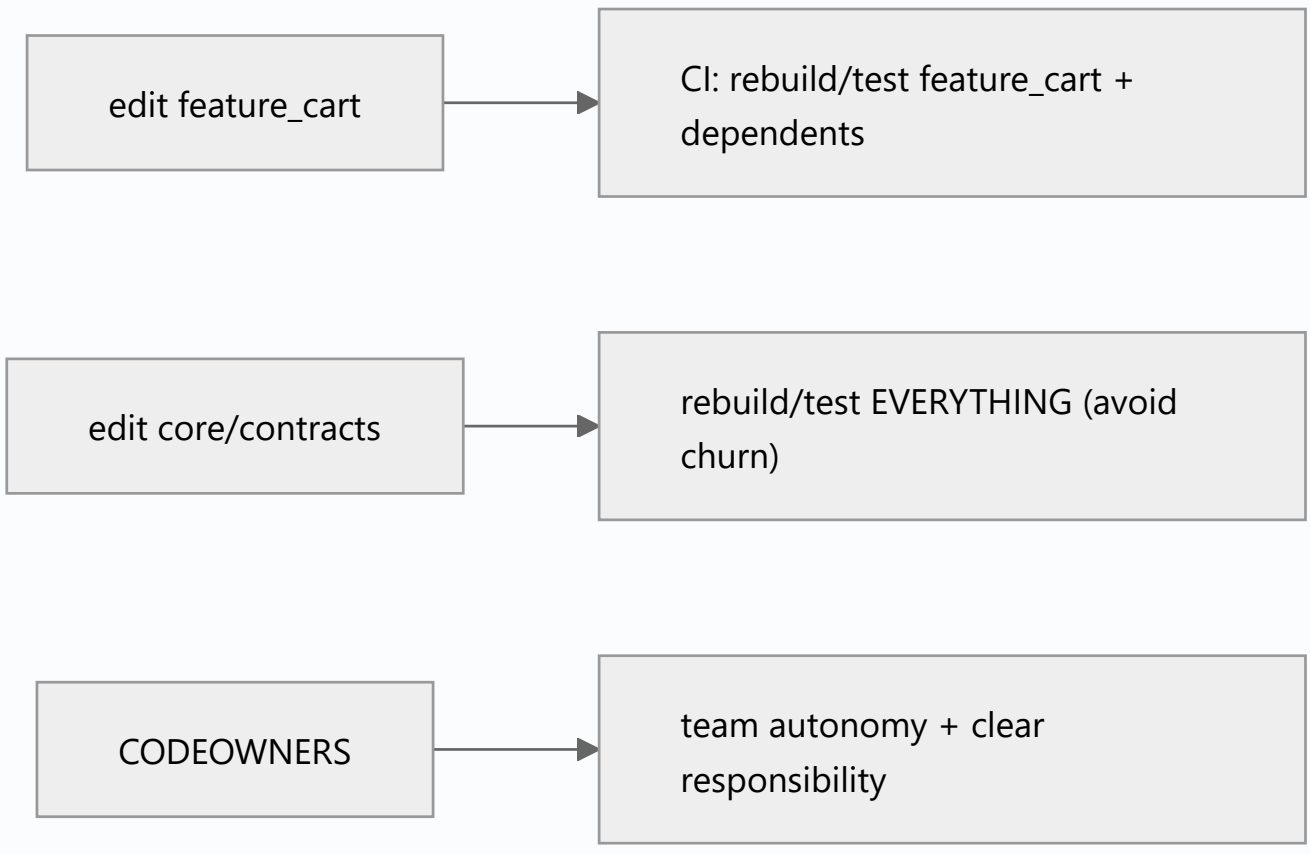
Incremental compilation per package is the mechanism behind faster builds; broad invalidation (core/contracts change) is the cost to manage.

Examples

```
# CI: incremental — only affected packages analyzed/tested
melos bootstrap
melos run analyze --since=origin/main
melos run test    --since=origin/main   # scales with change size, not repo size
melos version                           # per-package version bump + changelog (conventional commits)
```

```
# CODEOWNERS — package = ownership unit
/packages/feature_cart/     @team-commerce
/packages/feature_auth/     @team-identity
/packages/contracts/        @team-architecture   # gate breaking contract changes
/packages/core/             @team-platform       # keep stable/low-churn

Stability rule of thumb:
  leaf feature packages  -> change often, cheap (rebuild themselves + few dependents)
  core / contracts       -> change rarely, expensive (rebuild EVERYTHING) -> keep stable
```

Diagrams



Common Mistakes

Mistake	Why it's a problem	Fix
Churny `core` / `contracts`	Rebuilds everything (kills incrementality)	Keep roots small, stable, low-churn
Breaking the contracts API casually	Ripples to all consumers	Semver + additive changes + gated review
Running full CI every build	No incremental payoff	Scope with `--since` /affected packages
No package ownership	Diffuse responsibility	CODEOWNERS per package
Versioning purely-local packages needlessly	Overhead	Version/publish only shared/published
Deep dependency chains	Wide rebuild ripple	Shallow, star-shaped graph
Modular machinery on a small app	Overhead > payoff	Right-size (feature-first folders)

Best Practices

- Realize the **incremental build/CI** payoff (melos `--since`, per-package cache); design so **leaves change often (cheap)** and **roots (core/contracts) change rarely (stable)**.
- Keep `core` / `contracts` **small, stable, low-churn**; treat **contracts as a versioned published API** (semver, additive, gated breaking changes).
- Use **per-package versioning + changelogs** (melos) for shared/published packages; map **ownership to packages** (CODEOWNERS) for autonomy + clear responsibility.
- Enforce an **acyclic, shallow star graph**; integrate melos into **CI** for scoped runs; **right-size** — adopt this machinery only at justifying scale.

Performance

Build-time is the win: CI cost scales with the **changed set**, not repo size — provided **roots are stable** (churny core/contracts erase it) and the graph is **shallow** (deep chains widen ripple). Parallel per-package CI compounds the gain. Runtime unaffected.

Advantages / Disadvantages

- **+** Fast incremental CI, independent versioning/cadence, team autonomy + clear ownership, parallelizable pipelines, controlled evolution.
- **−** Requires stable roots + contract-version discipline, CODEOWNERS/governance, versioning overhead, only pays off at scale.

Interview Questions

1. 🟢 **How does modularization speed up builds/CI?** — Only changed packages (+ dependents) rebuild/re-test (incremental), scoped via melos `--since` — CI scales with change size, not repo size.
2. 🟢 **Why must `core` / `contracts` be stable?** — Everything depends on them, so any change rebuilds/re-tests the whole repo — churn there destroys incrementality.
3. 🟡 **How does versioning work across packages?** — Each package has its own version + changelog (melos/conventional commits); shared/published packages evolve on their own cadence, pinned by the app.
4. 🟡 **Why is contract versioning the highest-risk?** — The contracts API is depended on by all consumers; a breaking change ripples everywhere — use semver, additive changes, and gated review.
5. 🟡 **How does ownership map to modules?** — A package is an ownership unit (code + API + tests + release); CODEOWNERS assigns teams, enabling autonomy + clear responsibility.
6. 🔴 **What graph properties keep CI fast?** — Acyclic + shallow + stable roots (star around core/contracts); deep chains or churny roots widen rebuild ripple.
7. 🔴 **When is this machinery not worth it?** — Small/single-team apps — the versioning/ownership/tooling overhead exceeds the build/autonomy payoff.

Senior Engineer Tips

- Guard `core` / `contracts` like production APIs — small, stable, review-gated; their churn is the single biggest way teams lose the incremental-build payoff they modularized for.
- Wire CI to affected-package runs (`--since`) and CODEOWNERS to packages from day one; incrementality and clear ownership are the concrete reasons to modularize.
- Prefer additive, backward-compatible contract changes and coordinate the rare breaks; a careless breaking change to `contracts` is a repo-wide fire drill.

Architect Perspective

The build/versioning/ownership layer is *why* modularization is worth its cost at scale: incremental CI, independent cadence, and autonomous teams — all flowing from the package graph, all contingent on **stable roots** and **disciplined contract evolution**. This is the Stable-Dependencies/Stable-Abstractions principle made operational: depend on stable things, keep shared abstractions stable, and let volatile features be cheap leaves. Realized well, it's the foundation of scalable, multi-team, multi-app Flutter delivery (01_modular_fundamentals.md, Module 47, Module 50).

Summary

- Modules give incremental build/CI (only changed packages + dependents), independent versioning/cadence, and team ownership — via the package graph.
- Payoff depends on **stable, low-churn core/contracts** (churn rebuilds everything), **disciplined contract versioning** (semver/additive/gated), and **CODEOWNERS** ownership.
- Keep the graph acyclic + shallow (star), scope CI with `--since`, and adopt this machinery only at justifying scale.

Revision Notes

- Incremental CI: rebuild/test changed package + dependents (`melos run --since`); scales with change size — **only if roots stable**.
- Stability: core/contracts churn → rebuild everything (avoid); keep small/stable/low-churn (SDP). Contracts = versioned published API (semver, additive, gated breaks).
- Versioning per-package (melos/conventional commits, own cadence for shared); ownership = package (CODEOWNERS); acyclic + shallow star; right-size to scale.

Practice Questions

1. Why does core/contracts stability determine your CI speed?
2. How is versioning and ownership organized across packages?
3. Why is a breaking contracts change the riskiest kind?

Coding Questions

1. Write CI commands for incremental analyze/test with melos `--since`.
2. Author a CODEOWNERS mapping packages to teams (incl. gated contracts).
3. Show an additive vs breaking contract change and how you'd manage each.

Mini Project

Build/versioning/ownership plan (Flutter): For a modular monorepo (app + core + contracts + feature packages), write the CI plan (melos bootstrap + `--since` scoped analyze/test), a CODEOWNERS mapping (features to teams, contracts/core to platform/architecture with gated changes), and a versioning/contract-evolution policy (per-package versions; additive-preferred, semver, gated breaking contract changes). Note the stability rule (keep core/contracts low-churn) and when this machinery is justified. Acceptance: incremental CI plan (`--since`); CODEOWNERS per package (gated contracts/core); per-package versioning + contract-evolution policy; stable-roots rule stated; right-sizing justified.

Modular Integration (Capstone: A Modular Monorepo Skeleton)

> Assemble the full modular architecture: a **melos monorepo** with an `app` package (composition root) depending on **feature packages** (`feature_auth` , `feature_cart`) and **shared packages** (`core` , `contracts`), each a **Clean slice** with a **minimal public API** (`export` s over `lib/src/`), interacting **only via contracts + DI** on an **acyclic star graph**, managed with **melos scripts** and **incremental CI**, with **per-package ownership/versioning**. This is feature-first with compiler-enforced boundaries and build/team payoffs — the enterprise-scale chassis and the bridge to DDD and scalable apps (Module 46, Module 47).

Introduction

This module capstone composes fundamentals, melos, boundaries/contracts, and build/versioning into one runnable modular monorepo skeleton — the reference structure teams grow on. It shows the package graph, the composition root wiring, and the operational setup, plus an honest right-sizing note.

Why this concept exists

The pieces pay off only assembled: packages + melos + contracts + composition root + incremental CI + ownership = a codebase that's hard-bounded, fast to build, ownable, and reusable at scale. This capstone provides that assembled reference and cements the when/how.

Real-world analogy

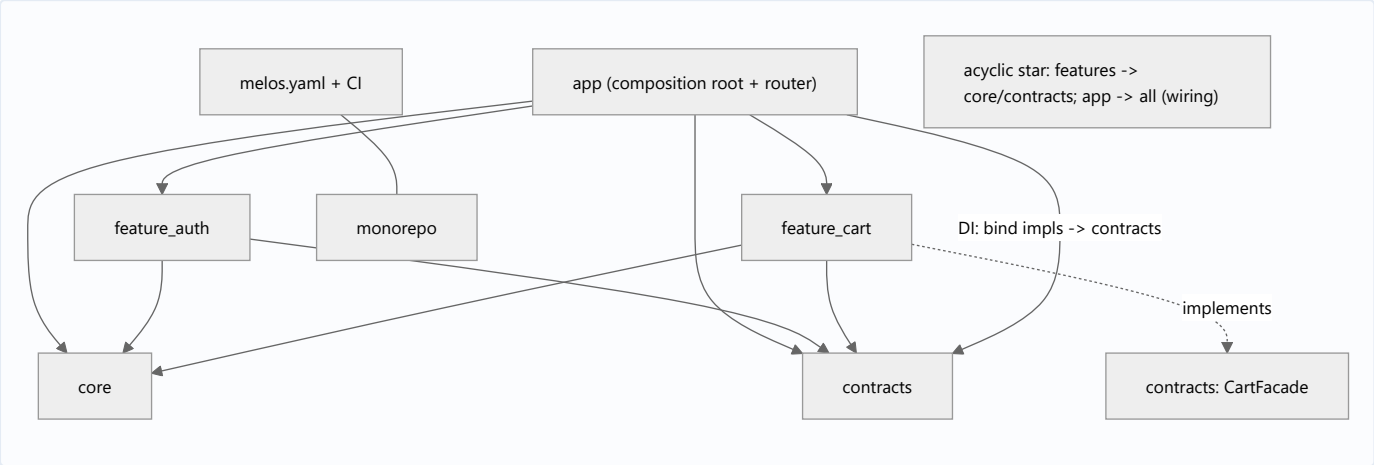
It's a **planned industrial park**: standardized plots (packages) with locked gates (public APIs), a shared utilities hub (core) and a standards office (contracts), a central operations building that connects tenants (app/composition root), and a park management system (melos + CI) that services only the plots that changed and tracks who owns which plot. Tenants collaborate through standards, never by trespassing — and a successful plot can be franchised elsewhere (reuse).

Problem Statement

Build a modular monorepo skeleton: `app` (composition root + router) + `core` + `contracts` + `feature_auth` + `feature_cart` , path-linked (acyclic star), each a Clean slice with a minimal public API, cross-module interaction via `contracts` + DI wired in `app` , melos scripts + incremental CI,

CODEOWNERS + versioning policy — with a documented when-to-modularize rationale. You'll compose every file in this module.

Internal Working



- **Monorepo layout** (02_monorepo_and_melos.md): `apps/app` + `packages/{core, contracts, feature_auth, feature_cart}`; path deps; `melos.yaml` (globs + analyze/test/gen scripts).
- **Package graph (acyclic star)** (03_module_boundaries_and_contracts.md): features → **core + contracts** (never each other); **app** → **everything** (to wire). Contracts/core are stable roots; features are leaves; app is the top.
- **Each module = a Clean slice** (Module 40) with a **minimal public API** (`export`s; `lib/src/` private) and **self-registration** (`registerX(di)` + routes).
- **Contracts + DI wiring**: shared interfaces/DTOs/events in `contracts`; `feature_cart` **implements** `CartFacade`; `feature_auth`/`checkout` **depend on the contract**; the `app` **composition root** binds impls→contracts and aggregates routes (Module 14).
- **Operational setup** (04_build_versioning_and_teams.md): melos **bootstrap** + `--since` **incremental CI**; **CODEOWNERS** per package; **per-package versioning**; **stable core/contracts**.
- **Right-sizing (honest)**: this full chassis suits **large/multi-team/reuse** scenarios; a small app should stay **feature-first folders** and **grow into** this from a clean base (Module 44).
- **Payoff realized**: compiler-enforced boundaries, incremental builds, ownership, reuse — the assembled benefits of the module.

Memory Representation

Not runtime — a repo of packages + an acyclic package graph annotated with versions/owners; melos manages the workspace; the app composition root builds the DI graph + router at startup by invoking each feature's self-registration.

Compiler Behavior

Declared path deps + `export` / `lib/src/` privacy enforce boundaries structurally; incremental compilation caches unchanged packages; undeclared/cyclic deps fail. Consumers compile against contracts without knowing implementing features.

Runtime Behavior

`main` → app composition root → registerCore + per-feature register (incl. cart's `CartFacade` impl) + route aggregation → app runs; cross-module calls resolve to contract implementations via DI. Identical runtime to feature-first; packaging affects build/CI only.

Flutter Engine Behavior

Standard; routing via the app's root router (navigate across modules by route).

Dart VM Behavior

Package-granular incremental builds; melos scopes analyze/test to changed packages for fast CI.

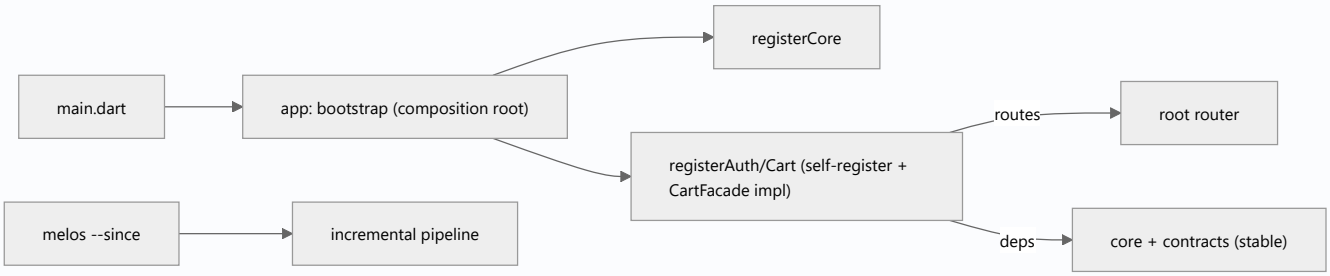
Examples

```
monorepo/
  melos.yaml
  apps/app/        (composition root, root router, main.dart) -> depends on all packages
  packages/
    core/          (di, network, Result, theme, value objects)   [stable root]
    contracts/     (CartFacade, DTOs, events)                     [stable root]
    feature_auth/  (Clean slice; depends on core+contracts)       [leaf]
    feature_cart/  (Clean slice; implements CartFacade)           [leaf]
```

```
// apps/app/lib/composition.dart — the ONLY place that knows all modules; wires impls->contracts
Future<void> bootstrap(GetIt di) async {
  registerCore(di);                                  // core (stable)
  registerAuth(di);                                  // feature self-registration
  registerCart(di);                                  // registers CartFacadeImpl -> CartFacade
}
final router = GoRouter(routes: [...authRoutes, ...cartRoutes]); // aggregate feature routes

// packages/feature_checkout/... depends on contracts only:
class CheckoutVm { final CartFacade cart; CheckoutVm(this.cart); } // no dep on feature_cart
```

```
# Operational
melos bootstrap
melos run analyze --since=origin/main
melos run test    --since=origin/main    # incremental CI
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Feature package depends on another feature	Coupling/cyclic	Depend on contracts/core; wire in app
Exporting internals (`src/**`)	No real boundary	Minimal public API
Churny core/contracts	Rebuilds everything	Keep roots stable/low-churn
Full CI every build	No incremental payoff	`--since` scoped runs
Wiring impls inside features	Features learn of each other	Wire in app composition root
No ownership/versioning	Diffuse responsibility	CODEOWNERS + per-package versions
Modular chassis on a small app	Overhead > payoff	Feature-first folders; grow into it

Best Practices

- Structure a **melos monorepo** (`apps/app` + `packages/{core, contracts, feature_*}`) with **path deps** on an **acyclic star** (features → core/contracts; app → all).
- Make each module a **Clean slice** with a **minimal public API** (`export` / `src`) that **self-registers**; interact **cross-module via contracts + DI** wired in the **app composition root**.
- Keep **core/contracts stable/low-churn** + **versioned** (contracts as published API); use **incremental CI** (`--since`) and **CODEOWNERS** per package.
- **Right-size**: adopt this at justifying scale; otherwise stay feature-first and **grow into** modules from a clean base.

Performance

Build/CI scales with the changed set (incremental, `--since`) given **stable roots** and a **shallow acyclic graph**; parallel per-package CI compounds it. Runtime is identical to feature-first. The chassis's value is build speed + team autonomy + reuse at scale.

Advantages / Disadvantages

- **+** Compiler-enforced boundaries, incremental/parallel CI, ownership + independent versioning, cross-app reuse, consistent Clean slices — enterprise-ready.
- **−** Workspace tooling + contract/DI wiring + versioning/governance overhead; overkill for small apps; requires stable-roots + acyclic discipline.

Interview Questions

1. 🟢 **What's in a modular monorepo skeleton?** — `apps/app` (composition root/router) + `packages/{core, contracts, feature_*}`, path-linked on an acyclic star, managed by melos.
2. 🟢 **How do modules interact?** — Via a `contracts` package (interfaces/DTOs/events): providers implement, consumers depend on it; the app composition root wires impls→contracts via DI — no feature→feature deps.
3. 🟡 **Where is cross-module wiring done, and why there?** — In the `app` composition root — the only place that knows all modules — so features stay unaware of each other's concretes.
4. 🟡 **How is CI kept fast?** — melos `--since` runs analyze/test only on changed packages + dependents; stable core/contracts keep the changed set small.
5. 🟡 **How do ownership and versioning work?** — Package = ownership unit (CODEOWNERS) with its own version/changelog; contracts/core are stable, gated, versioned as published APIs.
6. 🔴 **Why must the graph be an acyclic star and roots stable?** — Cyclic deps can't build; churny roots rebuild everything — a star with stable core/contracts preserves incrementality.

7. 🔴 **When is this chassis justified vs overkill?** — Justified at scale (build pain, multiple teams, reuse); overkill for small/single-team apps — stay feature-first and grow into it.

Senior Engineer Tips

- Build one exemplary feature package + the app composition root + melos/CI wiring first; it's the template every module follows and proves the incremental-build payoff.
- Keep the app the sole wiring point (impls→contracts) and keep core/contracts stable and gated; those two disciplines protect both the acyclic graph and your CI speed.
- Modularize from a clean feature-first base only when signals justify it; extraction should be mechanical (locked-door feature slices), and premature modularization is pure overhead.

Architect Perspective

The modular monorepo is the structural endgame of the feature-first arc: cohesive Clean slices with **compiler-enforced** boundaries, interacting through a neutral contracts package on an acyclic star, wired at a single app composition root, and operated with incremental CI + per-package ownership/versioning. It delivers isolation, build speed, autonomy, and reuse — the platform for enterprise-scale delivery and a natural fit for module-per-bounded-context DDD. Its power is real and so is its overhead, so it's a **scale-justified** choice built on a clean feature-first foundation (Module 44, Module 46, Module 47, Module 50).

Summary

- Skeleton: melos monorepo — `apps/app` (composition root/router) + `packages/{core, contracts, feature_*}`, path deps, acyclic star (features→core/contracts; app→all).
- Each module = Clean slice + minimal public API + self-registration; cross-module via contracts + DI wired in app; incremental CI (`--since`) + CODEOWNERS + per-package versioning; stable core/contracts.
- Delivers enforced boundaries + build speed + ownership + reuse; right-size — grow into it from feature-first.

Revision Notes

- Layout: `apps/app` (composition root + router) + `packages/{core, contracts, feature_auth, feature_cart}`; path deps; melos scripts.
- Graph: acyclic star (features→core/contracts, app→all); each module Clean slice + minimal public API (`export` / `src`) + self-register; cross-module via contracts+DI wired in app.
- Ops: bootstrap + `--since` incremental CI; CODEOWNERS per package; per-package versioning; stable/low-churn core/contracts; right-size (grow from feature-first).

Practice Questions

1. Sketch the package graph and explain why it's an acyclic star.
2. Where and how is cross-module interaction wired?
3. What operational payoffs does this skeleton deliver, and what discipline sustains them?

Coding Questions

1. Lay out the monorepo (app + core + contracts + 2 features) with path deps + melos.yaml.
2. Wire the app composition root (self-registering features + contract impl bindings + route aggregation).
3. Add incremental CI commands + a CODEOWNERS mapping.

Mini Project

Modular monorepo skeleton (capstone — Flutter): Build a melos monorepo — `apps/app` (composition root + root router), `packages/core`, `packages/contracts` (`CartFacade` + DTO), `feature_auth`, `feature_cart` (implements `CartFacade`) — each a Clean slice with a minimal public API + self-registration, path-linked on an acyclic star (features→core/contracts; app→all), with cross-module interaction via contracts + DI wired in `app`, melos scripts + `--since` CI, CODEOWNERS, and a versioning/stability policy. Document the when-to-modularize rationale. Acceptance: acyclic star package graph (no feature→feature deps); minimal public APIs (`export` / `src`); contracts + DI wiring in app (self-registering features); incremental CI + CODEOWNERS + versioning; stable core/contracts; right-sizing justified; runs end-to-end.