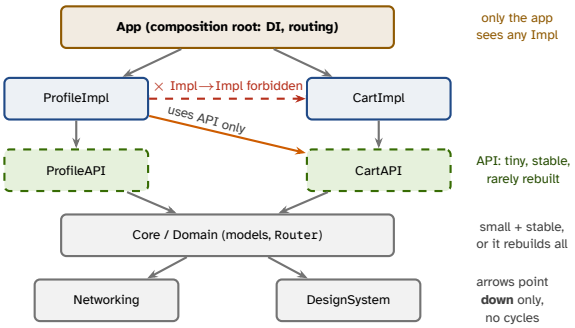
Modularization & Swift Package Manager

In one line: Cut the app into **modules** (local SPM packages) along seams that change independently; features depend on each other's **interfaces** only, dependencies point **down**, and a thin **app target** composes the concrete implementations. The payoff is faster incremental builds and **compiler-enforced** boundaries.



Edit `CartImpl` → what rebuilds?



Profile imports `CartAPI`, not `CartImpl` → no recompile.
Edit `CartAPI` instead → Cart + Profile + App all rebuild.

Access across modules (narrow → wide)

`private`	`fileprivate`	`internal`	`package`	`public`	`open`
scope / file		module (default)	same package (Swift 5.9)	other modules use it	+ subclass / override

`public class`: usable, NOT subclassable outside its module.
`@testable import` lifts `internal` (needs `ENABLE_TESTABILITY`, Debug); never `private`.

How it works

- **Why:** incremental builds (only a changed module + its dependents recompile), **enforced boundaries** (`internal` is the default, so nothing leaks by accident), **ownership** (a team per package), isolated tests + fast SwiftUI previews, **reuse** across app, widget, extensions, App Clip.
- **Layers:** App → Features → Interfaces/Core → Foundation. Acyclic; SPM *rejects* target cycles (package-to-package cycles are allowed from tools 6.0), but does *not* forbid Feature→Feature edges — that rule is policy (review, CI graph lint).
- **API/Impl split:** `FeatureAPI` = protocols, routes, DTOs (no heavy deps); `FeatureImpl` = UI + logic. Others import the API; the app injects the Impl. Cross-feature navigation = a `Router`/`Route` enum in Core, resolved by the app.
- **Package.swift:** the first line `// swift-tools-version:` picks the manifest API *and*, from **6.0**, puts every target in the **Swift 6 language mode** (data races = errors) unless `swiftLanguageModes: [.v5]`. A dependency takes two entries: the package *and* the target's `.product(name:package:)`.
- **Versions:** `from: "1.2.0"` = `>=1.2.0 <2.0.0`; `.exact`, ranges; commit `Package.resolved` for apps.
- **Resources:** `.process` (optimised: asset catalogs, localisation) or `.copy` (verbatim). Load via the synthesised `Bundle.module` — only generated when the target *has* resources.
- **platforms:** `[.iOS(.v16)]` is the package's *minimum*; it does not set the app's deployment target.
- **Binary target:** `.binaryTarget(name:url:checksum:)` (or `path:`) wraps a prebuilt `.xcframework`; checksum of the *zip* from `swift package compute-checksum`.
- **Plugins:** *build-tool* (SwiftGen, protobuf; a build command re-runs on changed inputs, a *prebuild* one every build) vs *command* (`swift package <verb>`: lint, format). Sandboxed.

Traps

- ! **Product ≠ target.** Other packages depend on *products*; a target not exposed as a product is private to its package.
- ! 40 modules that import each other's *concrete* Impl = file layout, not a graph: all the overhead, none of the incremental-build win.
- ! Image blank / `nil` inside a package → loaded from `Bundle.main`; use `Bundle.module`.
- ! **Over-modularization:** per-module manifest, link and DI cost; clean builds can get *slower*. Cut along ownership/change seams, not per screen.
- ! `ServiceLocator.shared` inside features hides dependencies — inject API protocols through `init` from the composition root.
- ! `package` (Swift 5.9) ≠ `@_spi`: `package` = visible to modules *in the same package*; `@_spi(X)` = public but only for importers writing `@_spi(X) import` (underscored, unofficial).
- ! `.library(type:)` omitted = *automatic* = **static** by default. Two copies bite only in *one process* (app + own dynamic framework): Xcode 12.5+ then builds it *dynamic* (+ dyld cost). App + extension = two processes: two copies cost only size.
- ! `.unsafeFlags` are refused only in a dependency resolved *by version* (or registry): root, `path:` and branch/revision packages may use them, the same package pulled in with `from:` may not.

Example — a feature package

```
// swift-tools-version:5.9
import PackageDescription
let package = Package(name: "Cart", platforms: [.iOS(.v16)],
  products: [.library(name: "CartAPI", targets: ["CartAPI"]),
    .library(name: "CartImpl", targets: ["CartImpl"])],
  dependencies: [.package(path: "../Core"),
    .package(url: "https://github.com/apple/swift-log",
             from: "1.5.0")],
  targets: [
    .target(name: "CartAPI"),               // protocols + models
    .target(name: "CartImpl", dependencies: ["CartAPI",
      .product(name: "Core", package: "Core"),
      .product(name: "Logging", package: "swift-log")],
      resources: [.process("Resources")]),  // -> Bundle.module
    .testTarget(name: "CartTests", dependencies: ["CartImpl"])])
```

Remember

"API up, Impl hidden, arrows down, app wires." Litmus: *can I build and preview any one feature in seconds with mocks, and does editing X never recompile Y?*

Related: linking & launch time · dependency injection · Coordinator (Router) · SOLID (DIP) · Hexagonal, Onion & Modular monolith (Clean) · Client SDK design (XCFrameworks)