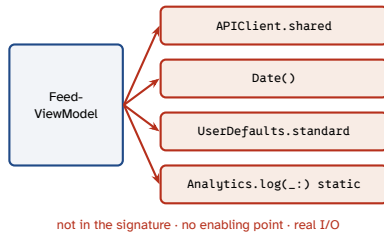
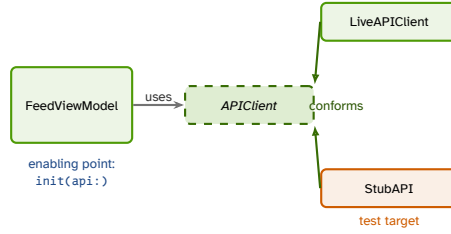


In one line: A **seam** is “a place where you can alter behaviour in your program without editing in that place” (Michael Feathers, *Working Effectively with Legacy Code*, 2004); its **enabling point** is where you choose the behaviour. In Swift the seam is a **protocol** and the enabling point is the **init** — code that *reaches for* `.shared`, `Date()` or `UserDefaults.standard` has none.

BEFORE — hidden dependencies**AFTER — object seam****Feathers' 3 seam types**

object — swap via protocol/subclass
← the Swift default

link — swap a library/module at build

preprocessing — `#if DEBUG`

Legacy loop

characterize → **seam** → **test**

→ **refactor** (never refactor first)

How it works

- DI** = the type *receives* collaborators instead of creating them. **Constructor** injection (preferred: `required`, `let`, never half-built) · **property** (late/optional, e.g. `delegate`) · **method** (varies per call). Wire concretes in one **composition root** (`@main/SceneDelegate`).
- Why these four hurt tests:**
 - `singleton .shared` — hidden, no seam; its state leaks between tests (breaks **Independent**);
 - `static call` — statically dispatched, nothing to substitute;
 - `Date()`, `UUID()`, `random` — differ every run (breaks **Repeatable**);
 - `UserDefaults.standard` — persists to *disk* across tests and runs. Inject `UserDefaults(suiteName:)`; in `tearDown` call `removePersistentDomain(forName:)`.
- @testable import App** — tests see internal symbols (and may subclass/override public ones). Needs the module built with *Enable Testability* (`-enable-testing`, on in Debug). **Never** exposes `private/fileprivate` — needing that means “extract a type”.
- Legacy code** = “code without tests” (Feathers). **Dilemma:** to change safely you need tests; to add tests you must change code. So the first edit is the smallest *mechanical*, IDE/compiler-checked one that creates a seam.
- Characterization test** (golden master): pins what the code does *today*, bugs included. Recipe: assert a deliberately wrong value, run, copy the *actual* from the failure message into the assertion.

Dependency-breaking moves (Feathers)

Extract Interface	protocol the concrete type already fits; depend on it
Parameterize	<code>init(api: APIClient = Live...)</code> — callers unchanged
Constructor	
Wrap / Adapt	own thin protocol around a static SDK or framework type
Extract & Override	subclass SUT, override the I/O method — needs non-final; a stepping stone
Sprout	new logic in a new, fully-tested unit; old code calls it
Method/Class	
Wrap Method/Class	rename old, new one runs old + new behaviour before/after

Remember

Receive, don't reach. The four hidden deps: **Singleton** · **Static** · **Date/UUID** · **Defaults** — “**SSDD**”. Legacy order: **Characterize** → **Seam** → **Test** → **Refactor**.

Example — parameterize the constructor

```
protocol APIClient { func feed() async throws -> [Post] }
final class LiveAPIClient: APIClient { // old singleton
    static let shared = LiveAPIClient()
    func feed() async throws -> [Post] { /* URLSession */ [] } }
final class FeedViewModel {
    private let api: APIClient
    private let now: () -> Date
    private let defaults: UserDefaults
    init(api: APIClient = LiveAPIClient.shared, // prod call sites
         now: @escaping () -> Date = Date.init, // stay unchanged
         defaults: UserDefaults = .standard) {
        self.api = api; self.now = now; self.defaults = defaults } }
// test: FeedViewModel(api: StubAPI(),
//                      now: { Date(timeIntervalSince1970: 0) },
//                      defaults: UserDefaults(suiteName: #function)!)
```

The brief form `init(api: APIClient = .shared)` compiles when `APIClient` is the concrete class; with a protocol, name the type (`LiveAPIClient.shared`).

Interview traps

- ! **Live-default leak:** a test that forgets one argument silently gets the *real* `LiveAPIClient` → network in tests. Pass every dep in tests.
- ! Making things public “for tests” — `@testable` already opens internal; test private logic through behaviour.
- ! Swapping static `var shared` in a test without resetting it in `tearDown/addTeardownBlock` = order-dependent flakes.
- ! “Refactor the class first, then test” — refactoring untested code changes behaviour silently. Characterize first.
- ! Service Locator (`Container.resolve(X.self)` inside the body) is *not* DI — the dependency is still hidden.
- ! 8 init parameters is an **SRP** smell, not a DI problem — split the type.
- ! **DI** ≠ **DIP**: DI is the technique; DIP says the *protocol belongs to the high-level side*.

Likely questions

- What is a seam? — alter behaviour without editing there; has an enabling point.
- Test code that uses `URLSession.shared`? — protocol + defaulted init param.
- What does `@testable` do? — exposes internal; never private.
- Characterization test? — pins current behaviour before you change it.
- Why is `Date()` a problem? — non-repeatable; inject `()` → `Date`.