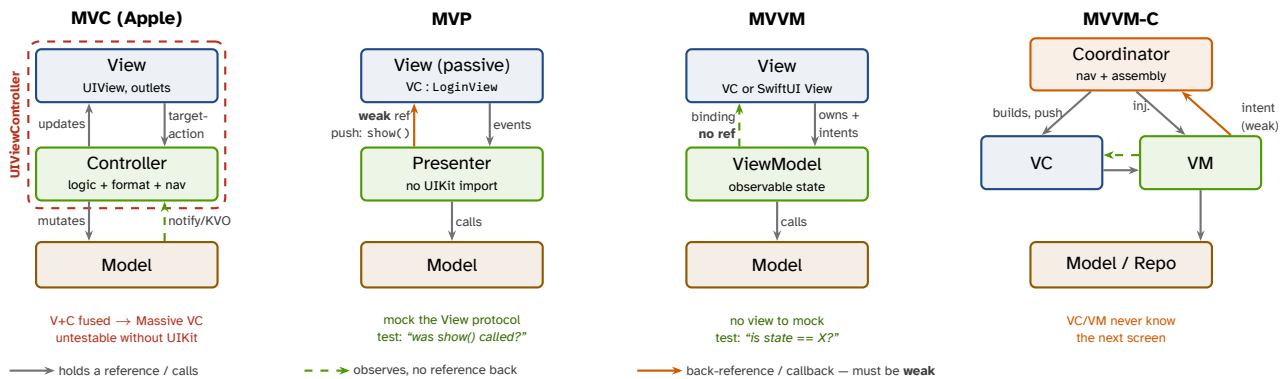


In one line: All four keep the **same Model**; they differ in **where presentation logic lives** and **who holds a reference to whom**. Each step moves logic *out* of the UIKit-bound view into a plain, unit-testable object: **MVC** (controller does it all) → **MVP** (presenter pushes to a view protocol) → **MVVM** (view *binds* to state; VM never sees the view) → **MVVM-C** (a Coordinator also takes *navigation* out).



How it works — who knows whom

	MVC	MVP	MVVM
logic in view is	Controller (= the VC) fused w/ controller	Presenter passive, behind a protocol	ViewModel binds to VM state
to view	VC sets outlets	Presenter <i>pushes</i> : view?.show()	View <i>observes</i> (bindings)
knows view?	yes, owns it	yes — via protocol (weak)	no
test via	UI / host app	mock View, verify calls	assert on state
fits	small screens	UIKit, strict tests	SwiftUI, Combine

- **Classic (Smalltalk) MVC:** the View observes the Model directly. **Apple MVC:** Controller mediates both ways, so V and M stay reusable. But UIViewController is view + controller, so networking, parsing, routing pile in → **Massive VC**: violates SRP, testable only with UIKit.
- **MVP:** View forwards every event, renders only what it is told. Presenter holds weak var view: LoginView?, formats data, calls explicit methods. No bindings. Presenter imports no UIKit ⇒ fully unit-testable against a mock View.
- **MVVM:** VM exposes formatted observable state + intent methods, holds *no* view reference and imports no UIKit. Binding tech: @Observable/@Published+Combine, KVO, closures/Box<T>. SwiftUI binds for free; **UIKit has none** — add one.
- **MVVM-C:** MVVM solves *presentation*, not *navigation*. Coordinator: childCoordinators, a presenter (UINavigationController), start(). Builds VC+VM, injects deps, performs push/present. VM only *signals* intent (closure / delegate / Combine subject).

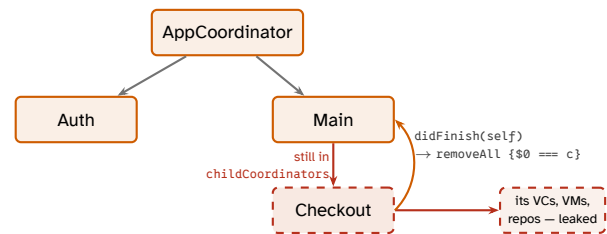
Example — same screen, MVP vs MVVM

```
// MVP: presenter KNOWS the view (protocol) and PUSHES
protocol LoginView: AnyObject { func show(error: String) }
final class LoginPresenter {
    weak var view: LoginView? // VC owns presenter
    func loginTapped(_ u: String) {
        if u.isEmpty { view?.show(error: "Empty") } }
}

// MVVM: VM knows NOTHING of the view; the view OBSERVES
@Observable final class LoginVM {
    var error: String? // bound by the view
    var onLoggedIn: (() -> Void)? // intent -> Coordinator
    func loginTapped(_ u: String) {
        if u.isEmpty { error = "Empty" } else { onLoggedIn?() } }
}
```

Test MVP: spy.shownErrors == ["Empty"]. Test MVVM: XCTAssertEqual(vm.error, "Empty") — no view at all.

Picture — the MVVM-C leak



parent → child strong, child → parent weak; not removed = leaks every visit

Interview traps

- ! **Forgetting MVP** (you did). Name all three, and for each say where logic lives, who references whom, how it is tested.
- ! "MVVM is just MVP." No: Presenter *knows* the view and *pushes*; VM has *no* view ref and the view *observes*.
- ! Frame by **testability**: MVP ≈ MVVM > MVC, because Presenter/VM import no UIKit. That is the whole point of the evolution.
- ! "MVVM in UIKit just works" — no binding mechanism exists; name yours (Combine sink/assign, closures, KVO, RxSwift).
- ! VM must not import UIKit nor push screens (that is the Coordinator's job); its nav closure captures [weak self].
- ! MVVM-C leak: child coordinator never removed. System **back button** and **swipe-to-dismiss** bypass didFinish — detect via UINavigationControllerDelegate didShow / presentationControllerDidDismiss. Remove with ==.
- ! Opposite bug: a child coordinator kept only in a local dies after start() — its screens show, its weak callbacks are nil.

Remember

C does it • P pushes it • VM publishes it • C_{oord} routes it.
 Place any pattern: testable without UIKit? holds the view?

Likely questions

1. MVP vs MVVM? — P holds a weak view protocol + pushes; VM holds none.
2. Why Massive VC? — Apple fuses V+C; everything lands there.
3. Best for SwiftUI? — MVVM: View binds to an @Observable VM.
4. Navigation in MVVM? — a Coordinator; the VM emits an intent.
5. Classic MVVM-C bug? — finished child never removed.