42 · MVP

Flutter Practice Notes

Contents

42 · MVP

Introduction

This module covers **Model-View-Presenter (MVP)**: the pattern where a **passive View** (implementing a **view interface/contract**) delegates all logic to a **Presenter** that drives it explicitly (`view.showLoading()` , `view.showData(...)`), with the **Model** holding data/rules. It walks the **fundamentals**, the **presenter ↔ view contract** (MVP's defining feature), its **standout testability** (mock the view, test the presenter without a UI), and its **fit and friction in Flutter** plus comparison to MVC/MVVM — tied together in a capstone. It sits with the other presentation patterns (Module 41 MVC, Module 43 MVVM) over the Clean backbone (Module 40).

Why this module exists

MVP's contribution is **maximum testability via an explicit view contract**: the presenter talks to an interface, so you can mock the view and test all presentation logic in plain Dart. That's valuable to understand — but MVP's **imperative "presenter drives the view"** model fits Flutter's **reactive/declarative** UI awkwardly (even more than MVC), so it's rarely the idiomatic choice. Knowing MVP clarifies the testability idea (which MVVM achieves differently) and why reactive Flutter gravitates to MVVM.

Module map

#	File	Topic	Level
1	01_mvp_fundamentals.md	Roles, passive view, presenter, data flow	🔵
2	02_presenter_and_view_contract.md	The view interface/contract (MVP's core), presenter driving the view	🟡
3	03_mvp_testability.md	Mock the view, test the presenter — MVP's standout strength	🟡
4	04_mvp_in_flutter_and_comparison.md	Flutter fit/friction; MVP vs MVC/MVVM	🔴
5	05_mvp_integration.md	Capstone: an MVP feature with a mocked-view test	🟡

> **Cross-references:** MVC: Module 41. MVVM (reactive fit): Module 43. Clean Architecture: Module 40. Testing (mocking): Module 49. SOLID (DIP/interfaces): Module 04. State management: Module 11.

Prerequisites

41 MVC, 04 SOLID (interfaces/DIP), 40 Clean Architecture, 49 Testing (mocking).

What you'll be able to do after this module

- Explain MVP's roles and the passive-view + presenter-driven-view flow.
- Define a view contract (interface) and a presenter that drives it.
- Test all presentation logic by mocking the view (MVP's key strength).
- Articulate MVP's friction in reactive Flutter and how it compares to MVC/MVVM.
- Decide when (rarely) MVP is worth its boilerplate in Flutter.

Capstone

MVP feature: A login/list feature with a `View` interface (`showLoading/showData/showError`), a `Presenter` that calls use cases and drives the view, and a widget implementing the view — with a full **presenter unit test using a mocked view** (asserting `showLoading` then `showData` / `showError`), plus a note on why MVVM is usually the better Flutter fit.

Summary

MVP = passive View (behind an interface) + Presenter (drives the view, holds presentation logic) + Model (data/rules). Its defining strength is **testability via the view contract** (mock the view). Its imperative flow fits reactive Flutter poorly, so it's uncommon here — but the testability lesson informs MVVM, the idiomatic choice.

MVP Fundamentals (Passive View & Presenter)

> MVP splits UI into **Model** (data + rules), **View** (a *passive* UI that implements a **view interface** and does nothing but display what it's told), and **Presenter** (holds all presentation logic, handles input, calls the Model, and **drives the view explicitly** via the interface: `view.showLoading()`, `view.showData(...)`). Unlike MVC, the **View and Presenter have a 1:1 relationship** and the View is **fully passive** — it never reads the Model directly; the Presenter pushes everything to it. This tight, interface-mediated coupling is what makes MVP exceptionally testable.

Introduction

This file establishes MVP's roles and its distinctive **passive-view + presenter-drives-view** flow, contrasting it with MVC. It's the foundation for the view-contract and testability files.

Why this concept exists

MVP evolved from MVC to make presentation logic **fully testable** and the View **truly dumb**. By routing *all* view updates through the Presenter via an interface, the View has zero logic (so nothing untestable lives in the UI) and the Presenter can be tested against a **fake view**. It trades some boilerplate for a clean, mockable seam.

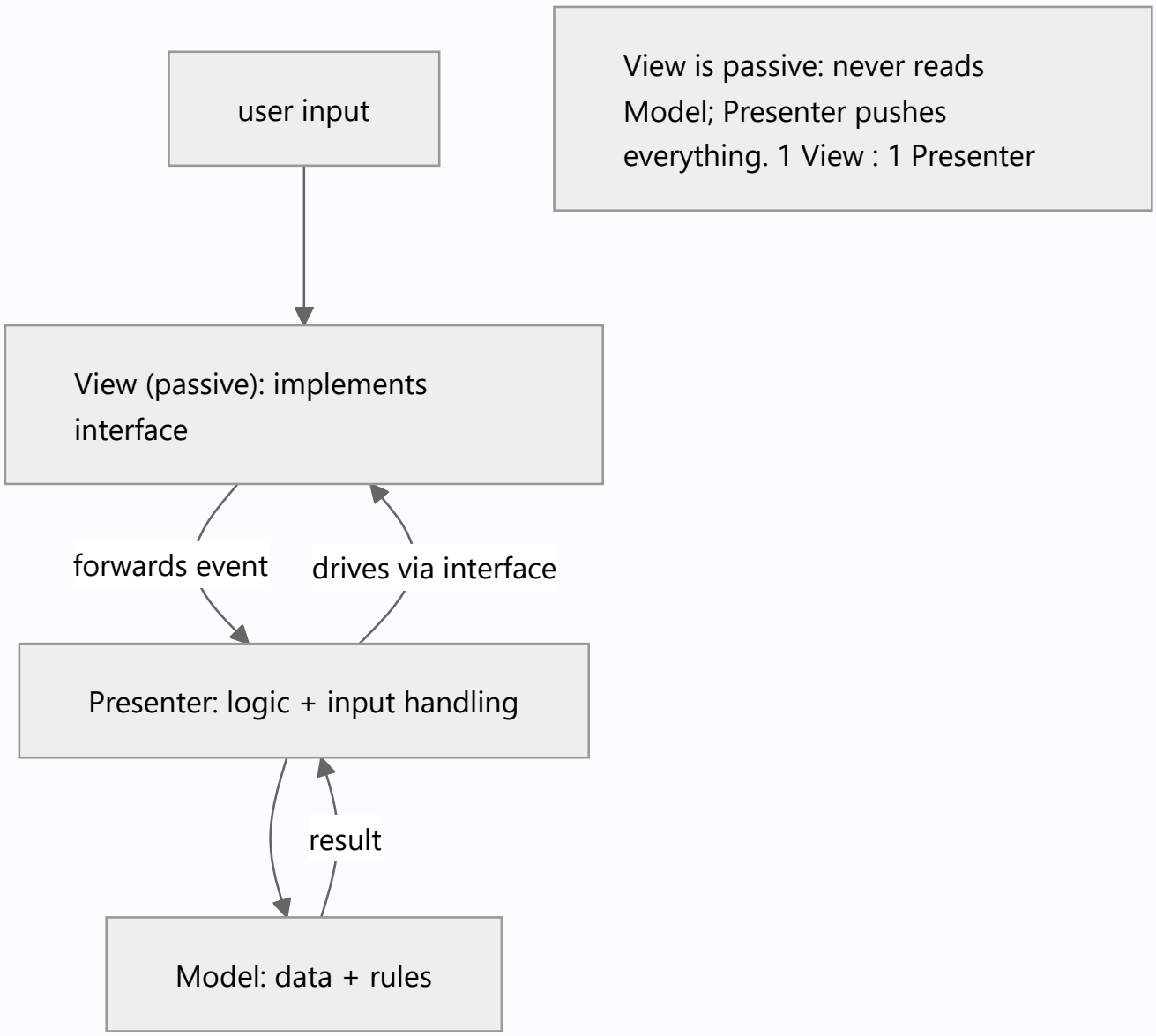
Real-world analogy

MVP is a **puppeteer (Presenter) and a puppet (View)**: the puppet has no will of its own — it only moves exactly as the puppeteer pulls its strings (`showData`, `showError`). The puppeteer consults the script/props (Model) and decides every movement. Because the puppet is completely passive, you can swap it for a **test dummy** and verify the puppeteer pulls the right strings in the right order — that's MVP's testability.

Problem Statement

For a login feature, assign roles: a passive View interface (methods the presenter calls), a Presenter (input handling + logic + Model calls + driving the view), and a Model (auth rules). Trace the flow of "tap login." You'll map MVP's roles and its presenter-driven data flow.

Internal Working



- **Model**: data + business rules + state (UI-agnostic), same as MVC (Module 41) — validates, computes, persists (often behind repositories/use cases — Module 40).
- **View (passive)**: implements a **view interface** exposing methods the Presenter calls (`showLoading` , `showData(list)` , `showError(msg)` , `navigateHome`). It **forwards user events** to the Presenter and **does nothing else** — no logic, no direct Model access, no formatting decisions. It's a **dumb rendering surface**.
- **Presenter**: holds **all presentation logic** — receives forwarded events, calls the **Model**, decides what to show, and **explicitly drives the View** through the interface. It holds a **reference to the View interface** (not the concrete widget) and typically maps Model data into display form before pushing it.
- **Key differences from MVC**:

    - **Passive View**: MVP's View **never reads the Model**; the Presenter **pushes** everything (in MVC the View often observes/reads the Model).
    - **1:1 View–Presenter**: each View has its own Presenter (MVC controllers can serve multiple views).
    - **Interface-mediated**: the Presenter talks to a **view interface**, enabling mocking (03_mvp_testability.md).
- **Data flow**: input → View forwards to Presenter → Presenter calls Model → Presenter drives View via interface. **Imperative** ("show this now"), not reactive.
- **Presenter lifecycle**: attach/detach the View (`attach(view)` / `detach()`) to avoid calling a disposed view — an explicit lifecycle concern.

Memory Representation

Presenter holds a reference to the **View interface** + Model/use cases. View holds a reference to the Presenter (to forward events). Communication is method calls both ways (View→Presenter events; Presenter→View interface methods) — no observer/binding.

Compiler Behavior

The view interface is a normal Dart `abstract class` /interface; the Presenter compiles against it (not the widget), enabling a mock implementation for tests.

Runtime Behavior

Input triggers presenter methods; the presenter imperatively calls view-interface methods to update the UI. There's no automatic reactivity — the presenter must call the right view methods.

Flutter Engine Behavior

Covered in 04_mvp_in_flutter_and_comparison.md — imperative view-driving conflicts with Flutter's rebuild-from-state model.

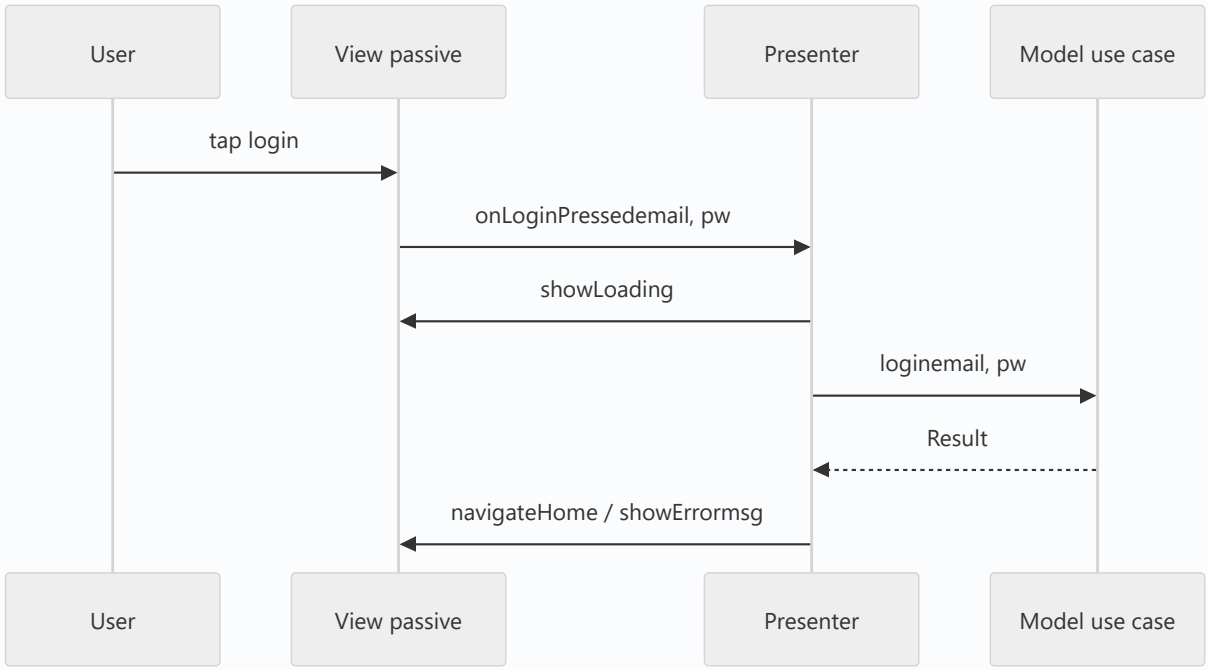
Dart VM Behavior

Presenter is plain Dart (fast tests); the view interface + mock enable device-free testing.

Examples

```
// VIEW INTERFACE — the contract the Presenter drives (passive view implements it)
abstract class LoginView {
  void showLoading();
  void showError(String message);
  void navigateHome();
}

// MODEL side — rules via a use case (domain)
class LoginUser {
  final AuthRepository auth;
  LoginUser(this.auth);
  Future<Result<void>> call(String email, String password) => auth.login(email, password);
}

// PRESENTER — all logic; drives the view through the interface (imperative)
class LoginPresenter {
  final LoginUser loginUser;
  LoginView? _view;                                  // holds the INTERFACE, not the widget
  LoginPresenter(this.loginUser);

  void attach(LoginView view) => _view = view;       // lifecycle: attach/detach
  void detach() => _view = null;

  Future<void> onLoginPressed(String email, String password) async {
    _view?.showLoading();                            // drive the view explicitly
    final r = await loginUser(email, password);
    if (r is Success) { _view?.navigateHome(); }
    else { _view?.showError('Login failed'); }       // push result to the passive view
  }
}
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
View reads the Model directly	Breaks passive-view rule	Presenter pushes all data to the view
Logic in the View	Untestable UI	All logic in the Presenter
Presenter references the concrete widget	Can't mock/test	Depend on the view interface
No attach/detach	Calling a disposed view → errors	Manage presenter/view lifecycle
One presenter for many views	Breaks 1:1 relationship	One presenter per view
Presenter importing Flutter UI	Couples/untestable	Presenter plain Dart + interface
Business rules in the Presenter	Belongs in Model/use case	Delegate rules downward

Best Practices

- Keep the **View fully passive** (implements the interface, forwards events, **never reads the Model**) and put **all presentation logic in the Presenter**.
- Have the Presenter **depend on the view interface** (not the widget) and **drive it explicitly**; maintain **attach/detach** lifecycle.
- Keep the **Presenter plain Dart** (no Flutter UI imports) and **delegate business rules** to the Model/use cases (Module 40).

- Maintain **1:1 View↔Presenter**; map Model data to display form in the Presenter before pushing.

Performance

Not a perf concern at this level; imperative updates are direct method calls. (Flutter-specific efficiency/friction is in 04_mvp_in_flutter_and_comparison.md.)

Advantages / Disadvantages

- **+** Fully testable presentation logic (mock the view), truly dumb view, explicit contract, clear separation.
- **−** Boilerplate (view interfaces + attach/detach), imperative (doesn't fit reactive UIs), 1:1 rigidity, presenter can bloat.

Interview Questions

1. 🟢 **What are MVP's three roles?** — Model (data/rules), passive View (implements an interface, forwards events), Presenter (all logic, drives the view via the interface).
2. 🟢 **What makes MVP's View "passive"?** — It has no logic and never reads the Model; the Presenter pushes everything to it via the interface.
3. 🟡 **How does MVP differ from MVC?** — Passive view (vs view reading the model), 1:1 view–presenter, and interface-mediated presenter→view calls (vs controller mediation).
4. 🟡 **Why does the Presenter depend on a view interface?** — So it can be tested against a mock view and isn't coupled to the concrete widget.
5. 🟡 **What lifecycle concern does MVP add?** — attach/detach — the presenter must not call a detached/disposed view.
6. 🔴 **Where do business rules live in MVP?** — In the Model/use cases; the Presenter holds presentation logic and delegates rules downward.
7. 🔴 **Why is MVP's flow "imperative"?** — The Presenter explicitly calls view methods (`showX`) to update the UI, rather than the view reacting to observable state.

Senior Engineer Tips

- Keep the presenter free of Flutter imports and depending only on the view interface; that single rule is what unlocks MVP's whole testability story.
- Enforce the passive-view rule ruthlessly — the moment a widget reads the model or formats data, MVP's benefit evaporates.
- Manage attach/detach carefully (null-safe `_view?.`) so async results never call into a disposed widget.

Architect Perspective

MVP's insight is the **explicit view contract**: routing all view updates through an interface makes presentation logic fully mockable and the view truly dumb. That testability idea is MVP's lasting contribution — but its **imperative presenter-drives-view** flow predates reactive UIs, which is why MVVM (achieving similar testability via observable state) fits Flutter better. Understand MVP for the contract/testability concept; reach for MVVM in practice (02_presenter_and_view_contract.md, Module 43).

Summary

- MVP: Model (data/rules) + passive View (implements interface, forwards events, never reads Model) + Presenter (all logic, drives view via interface).
- 1:1 view–presenter, interface-mediated, imperative flow; presenter plain Dart + attach/detach lifecycle; rules delegated to Model/use cases.
- Passive view + interface = MVP's testability foundation.

Revision Notes

- Model = data/rules (UI-agnostic); View = passive (implements interface, forwards events, no Model access/logic); Presenter = all logic, drives view via interface (imperative), plain Dart, attach/detach.
- 1:1 view↔presenter; presenter depends on view interface (mockable); rules → Model/use cases.
- Flow: input → view forwards → presenter → Model → presenter drives view (`showLoading/showData/showError`).

Practice Questions

1. What does "passive view" mean and why does it matter?
2. How does the presenter update the UI, and how is that different from MVC?
3. Why does the presenter depend on an interface, not the widget?

Coding Questions

1. Define a view interface + a presenter that drives it for a login flow.
2. Add attach/detach lifecycle and null-safe view calls.
3. Delegate the auth rule to a use case (keep the presenter presentation-only).

Mini Project

MVP role mapping (Flutter): For a login feature, define a `LoginView` interface (`showLoading/showError/navigateHome`), a `LoginPresenter` (plain Dart, drives the view, calls a `LoginUser` use case, attach/detach), and a widget implementing the view (passive: forwards events, renders what it's told). Acceptance: view passive (no logic/Model access); presenter holds logic + drives via interface (no Flutter imports); attach/detach lifecycle; rules in the use case; imperative flow correct.

The Presenter & View Contract

> MVP's defining feature is the **view contract** — an explicit **interface** listing every way the presenter can affect the UI (`showLoading()` , `showItems(list)` , `showError(msg)` , `navigateTo(route)`). This contract is the **seam**: the presenter depends only on the interface (never the widget), so it's decoupled and mockable, and the passive view's entire job is to **implement the contract**. Designing the contract well — **presentation-oriented** methods (states/effects), not leaky domain/UI types — is what makes MVP clean, testable, and the view swappable.

Introduction

This file focuses on MVP's core artifact: the view contract (interface) and how the presenter uses it to drive the view. It covers designing good contract methods, the two-way references, one-off effects vs state, and keeping domain/framework types out of the contract — the craft that makes MVP work.

Why this concept exists

The interface is what converts MVP from "presenter with a widget reference" (untestable, coupled) into "presenter with a mockable contract" (testable, decoupled). It's the application of the Dependency Inversion Principle (Module 04) to the presenter↔view relationship: the presenter depends on an abstraction the view implements.

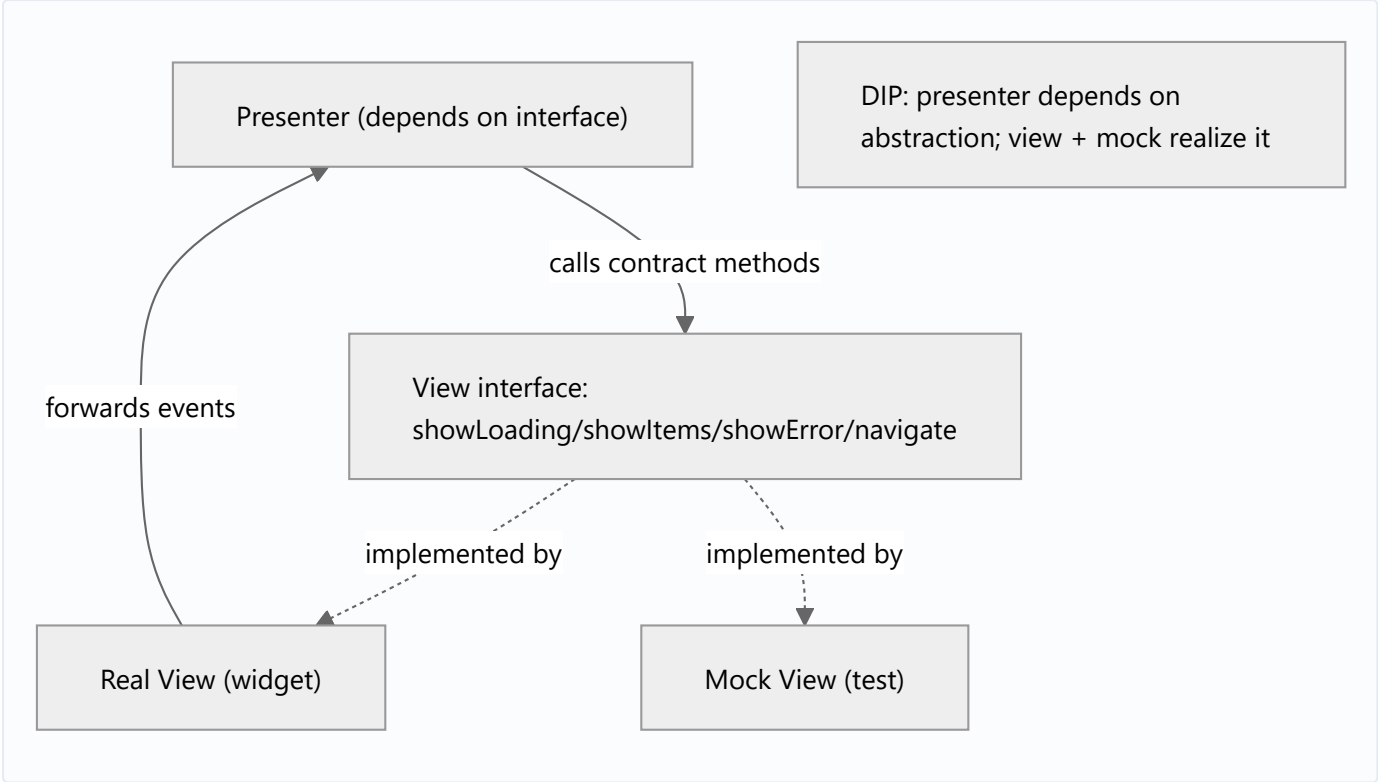
Real-world analogy

The contract is a **stagehand's cue sheet**: it lists exactly the cues the director (presenter) can call ("lights up," "curtain," "show slide 3") — nothing more. Any stage crew (view implementation) that can perform those cues works, and you can rehearse the director against a **stand-in crew** (mock) that just records which cues were called. A vague or leaky cue sheet ("do whatever with the raw script") breaks the whole arrangement.

Problem Statement

Design the view contract for a product-list screen: methods for loading/data/empty/error and one-off effects (navigation/snackbar), using presentation types (not domain entities or Flutter widgets), so the presenter drives it and a mock can verify calls. You'll design the interface + presenter interaction.

Internal Working



- **The contract (view interface)**: an `abstract class` enumerating **every presenter→view action** — typically **state renders** (`showLoading` , `showItems(List<ItemVm>)` , `showEmpty` , `showError(String)`) and **one-off effects** (`navigateTo(route)` , `showSnackbar(msg)`). This is the **entire surface** through which the presenter affects the UI.
- **Presentation-oriented methods**: methods should express **what the UI should do/show**, in **presentation terms** — pass **view models / primitives / messages**, **not** domain entities (leaks domain into UI) or Flutter widgets (leaks framework into the presenter). Map domain → view model in the presenter first.
- **Two-way references**: the **view implements the contract** and holds a **presenter** (to forward events); the **presenter holds the view interface** (to drive it). Wire this at attach time (01_mvp_fundamentals.md).
- **State vs one-off effects**: **state methods** (loading/data/error) are idempotent renders; **effect methods** (navigate/snackbar/toast) are one-shot. Distinguishing them avoids re-firing effects on re-render (a classic bug) — model effects as commands the view executes once.
- **Contract granularity**: prefer a **small, cohesive** contract (few meaningful methods) over many fine-grained setters; too granular → chatty/brittle, too coarse → leaky. Group by UI state.
- **DIP payoff**: because the presenter depends on the **interface**, you provide the **real widget** in production and a **mock** in tests — the seam that yields MVP's testability (03_mvp_testability.md).
- **No framework/domain leakage**: contract signatures use presentation types only; keep `BuildContext` , widgets, and raw entities out of the interface.

Memory Representation

The contract is an interface (no state). The presenter holds the interface reference + presentation state it maps from the Model. The view holds the presenter reference + its own widget state. Mock views record method calls/args for assertions.

Compiler Behavior

The interface + implementations are checked at compile time: the widget and the mock must implement all contract methods. The presenter compiles against the interface, so swapping implementations requires no presenter change.

Runtime Behavior

Presenter calls contract methods; the bound implementation (widget or mock) executes them. Event forwarding runs the reverse direction. Effects should fire once (view executes them), state methods can be called repeatedly.

Flutter Engine Behavior

The real view translates contract calls into imperative widget updates (`setState` /controllers) — the friction with reactive Flutter is detailed in 04_mvp_in_flutter_and_comparison.md.

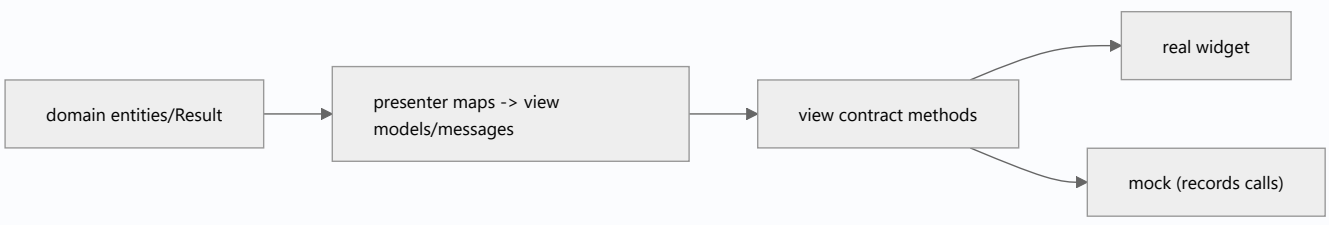
Dart VM Behavior

Interface dispatch is normal virtual calls; the presenter (plain Dart) + interface enable fast, device-free tests.

Examples

```
// VIEW CONTRACT — presentation-oriented; no domain entities / no Flutter types
abstract class ProductListView {
  void showLoading();
  void showItems(List<ProductVm> items);   // view models, NOT domain Product
  void showEmpty();
  void showError(String message);          // message string, NOT a Failure/exception
  void navigateToDetail(String id);        // one-off effect (command)
}

// PRESENTER — maps domain -> view models, drives the contract (depends on interface)
class ProductListPresenter {
  final GetProducts getProducts;           // domain use case
  ProductListView? _view;
  ProductListPresenter(this.getProducts);
  void attach(ProductListView v) => _view = v;
  void detach() => _view = null;

  Future<void> load() async {
    _view?.showLoading();
    final r = await getProducts();
    switch (r) {
      case Success(:final value) when value.isEmpty: _view?.showEmpty();
      case Success(:final value): _view?.showItems(value.map(ProductVm.from).toList()); // map!
      case Failure(:final error): _view?.showError(messageFor(error));                  // message!
    }
  }
  void onItemTapped(String id) => _view?.navigateToDetail(id); // effect
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Domain entities in the contract	Leaks domain into UI	Pass view models
`Failure`/exceptions in the contract	Leaks error internals	Pass mapped message strings
`BuildContext`/widgets in the contract	Leaks framework into presenter	Presentation types only
Effects modeled as state	Re-fire on re-render	Model effects as one-off commands
Too-granular contract (many setters)	Chatty/brittle	Cohesive state-based methods
Presenter holds the concrete widget	Untestable/coupled	Depend on the interface
View implements logic in contract methods	View not passive	Contract methods just render

Best Practices

- Design the contract with **cohesive, presentation-oriented methods** (state renders + one-off effects), passing **view models/primitives/messages — no domain entities, `Failure`s, or Flutter types**.
- Have the **presenter depend on the interface** (DIP) and **map domain → presentation** before driving it; keep the **view passive** (contract methods just render).
- **Distinguish state (idempotent) from effects (one-shot)** to avoid re-firing navigation/snackbars on re-render.
- Keep the contract **small and stable**; wire two-way references via **attach/detach**; the interface is the seam enabling **mock-based tests**.

Performance

Interface dispatch is negligible. The relevant concern is not re-firing effects (correctness) and, in Flutter, translating imperative calls to efficient rebuilds (04_mvp_in_flutter_and_comparison.md). No architectural perf cost.

Advantages / Disadvantages

- **+** Decoupled, mockable presenter↔view seam; swappable view; clear, explicit UI surface; DIP-clean.
- **−** Boilerplate (interface + implementations), risk of leaky/over-granular contracts, effect-vs-state handling, imperative feel.

Interview Questions

1. 🟢 **What is the view contract in MVP?** — An interface enumerating every action the presenter can perform on the UI (state renders + effects); the passive view implements it.
2. 🟢 **Why does the presenter depend on the interface, not the widget?** — Decoupling + testability (DIP): you can supply a real widget or a mock without changing the presenter.
3. 🟡 **What types should contract methods use?** — Presentation types (view models, primitives, messages) — never domain entities, `Failure`s, or Flutter widgets.
4. 🟡 **Why distinguish state methods from effect methods?** — State renders are idempotent (safe to repeat); effects (navigate/snackbar) must fire once, or they re-trigger on re-render.
5. 🟡 **How coarse/fine should the contract be?** — Cohesive and small (state-based methods); too granular is chatty/brittle, too coarse leaks concerns.
6. 🔴 **How does the contract map to Dependency Inversion?** — The presenter (policy) depends on an abstraction (the view interface); the view + mock (details) implement it — dependencies point to the abstraction.
7. 🔴 **What's a common effect-related bug and its fix?** — Re-navigating/re-showing snackbars on re-render because effects were modeled as state; model them as one-off commands the view executes once.

Senior Engineer Tips

- Treat the contract as the presenter's public API to the UI: keep it presentation-only and stable, and it stays mockable and swappable.
- Map domain → view models/messages in the presenter before touching the contract; a `Product` or `Failure` in the interface is a leak that couples UI to domain.
- Separate effects from state explicitly (a one-shot command channel) to kill the "snackbar fires twice / navigates on rebuild" class of bugs.

Architect Perspective

The view contract is MVP's essence and its DIP realization: an explicit, presentation-typed interface that decouples presenter from view and makes both mockable and swappable. Designing it well (cohesive, leak-free, effects-vs-state) is what delivers MVP's testability. This same "depend on an abstraction, map at the boundary" discipline recurs in Clean Architecture's interfaces and MVVM's observable state — MVP just makes the view side of it explicit (03_mvp_testability.md, Module 04, Module 40).

Summary

- The view contract (interface) is MVP's core: cohesive, presentation-oriented methods (state + one-off effects) with view models/messages — no domain/Flutter types.
- Presenter depends on the interface (DIP), maps domain→presentation, drives it; view passively implements it; attach/detach wires the two.
- Distinguish idempotent state from one-shot effects; the interface is the seam for mock-based testing.

Revision Notes

- Contract = interface of presenter→view actions (state renders + one-off effects); passive view implements; presenter depends on it (DIP, mockable).
- Presentation types only (view models/primitives/messages); map domain→presentation in presenter; no entities/`Failure`/`BuildContext`/widgets in contract.
- State (idempotent) vs effects (one-shot command); cohesive/small contract; attach/detach two-way refs.

Practice Questions

1. Why must contract methods avoid domain entities and Flutter types?
2. How do you prevent navigation/snackbars from re-firing on re-render?
3. How is the contract an application of DIP?

Coding Questions

1. Design a cohesive view contract (state + effects) for a list screen.
2. Write a presenter that maps domain results to contract calls.
3. Model a one-off effect (navigation) so it fires exactly once.

Mini Project

View contract design (Flutter): For a product-list screen, design a `ProductListView` contract (state: `showLoading/showItems(vm)/showEmpty/showError(msg)`; effect: `navigateToDetail(id)`), a presenter that maps domain results to contract calls, and a widget implementing it passively. Ensure no domain/Flutter types leak into the contract and effects fire once. Acceptance: cohesive presentation-only contract; presenter maps domain→view models/messages and depends on the interface; effects modeled as one-shot; view passive; ready to mock for tests.

MVP Testability (Mock the View, Test the Presenter)

> MVP's headline benefit: because the presenter depends on a **view interface**, you **mock the view** and unit-test the presenter in **plain Dart** — asserting it calls the right contract methods, in the right order, with the right arguments (`verify(view.showLoading())` then `verify(view.showItems(...))`). All presentation logic is testable **without a widget, device, or Flutter binding**, and the passive view has **no logic to test** (only a thin widget test that it renders contract calls). This is MVP's strongest selling point — and exactly the capability MVVM achieves differently.

Introduction

This file shows how MVP's view contract turns presentation logic into fast, device-free unit tests via a mocked view, what to assert (calls/order/args), and how it splits testing responsibilities (presenter = logic tests; view = thin render tests). It's the payoff of the contract (02_presenter_and_view_contract.md).

Why this concept exists

Presentation logic (loading→data/error sequencing, mapping, decisions) is where UI bugs live, and testing it through the real UI is slow and flaky. MVP isolates that logic in the presenter behind a mockable interface, so you can test it exhaustively and quickly — the reason MVP was created (to make MVC's logic testable).

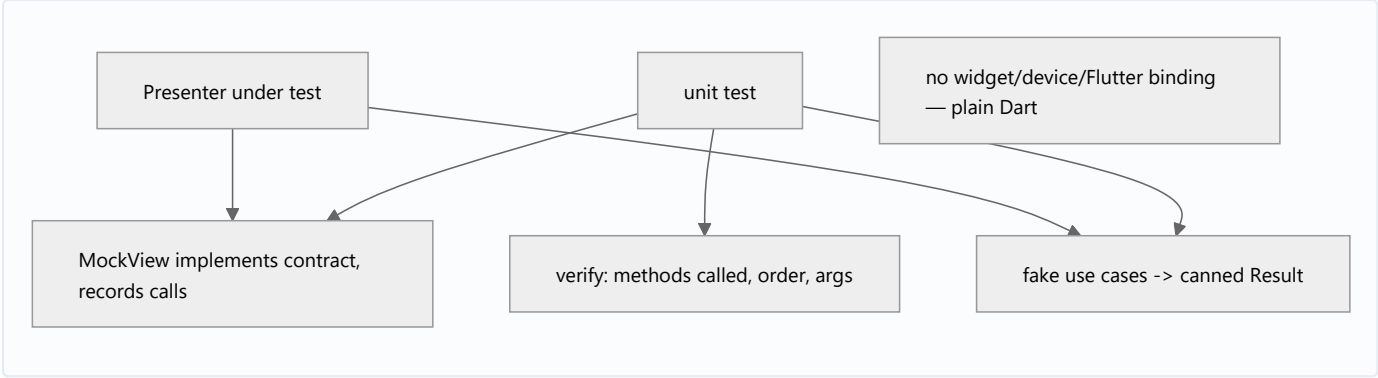
Real-world analogy

Testing the presenter with a mock view is like **rehearsing a director with a stand-in crew that just writes down every cue called**. You don't need the real, expensive stage (device/UI) — you verify the director calls "lights, curtain, slide 3" in the correct order for each scenario. If the director's logic is right against the stand-in, it's right on the real stage.

Problem Statement

Unit-test a `ProductListPresenter` : verify that `load()` calls `showLoading()` then `showItems(...)` on success and `showError(...)` on failure, and that a tap triggers `navigateToDetail(id)` — all with a **mock view**, no device. You'll write presenter tests using a mocked contract.

Internal Working



- **Mock the view**: create a **mock implementing the view interface** (via `mockito` / `mocktail` or a hand-written recorder) that captures which methods were called with what args (Module 49).
- **Fake the dependencies**: inject **fake use cases/repositories** returning canned `Result`s so you control the scenario (success/empty/failure).
- **Drive the presenter**: `attach(mockView)`, call presenter methods (`load()`, `onItemTapped(id)`), then **assert on the mock**:
  - **Which methods** were called (`showLoading`, `showItems` / `showError`).
  - **Order** (loading before data/error).
  - **Arguments** (correct view models/messages, mapped correctly).
  - **Effects** (navigate called once with the right id).
- **No Flutter binding**: presenter + interface + fakes are **plain Dart** → tests run in the fast unit tier (no `WidgetsFlutterBinding`, no device) (Module 40).
- **View tests are thin**: the passive view has no logic, so its test is a **small widget test** verifying it renders the right UI when contract methods are invoked (or is skipped if trivial). The bulk of testing is on the presenter.
- **What you get**: exhaustive, fast coverage of presentation logic (sequencing, mapping, decisions, error handling) — MVP's core value.
- **Same idea in MVVM**: MVVM gets equivalent testability by asserting the **view model's emitted state sequence** (no mock view needed) — a more reactive-friendly variant (Module 43).

Memory Representation

The mock view records call events (method + args) for assertions; fakes hold canned results. The presenter holds the mock (as the interface) + fakes. No widget tree/element tree exists in these tests.

Compiler Behavior

The mock must implement the full view interface (compile-checked). The presenter compiles against the interface, so the mock substitutes seamlessly.

Runtime Behavior

Tests run synchronously/async in the Dart VM: call presenter method → it calls mock contract methods → assertions inspect recorded calls. Fast and deterministic (no I/O, no UI).

Flutter Engine Behavior

None — these are pure Dart unit tests (no engine/binding). Only optional thin view tests use the Flutter test binding.

Dart VM Behavior

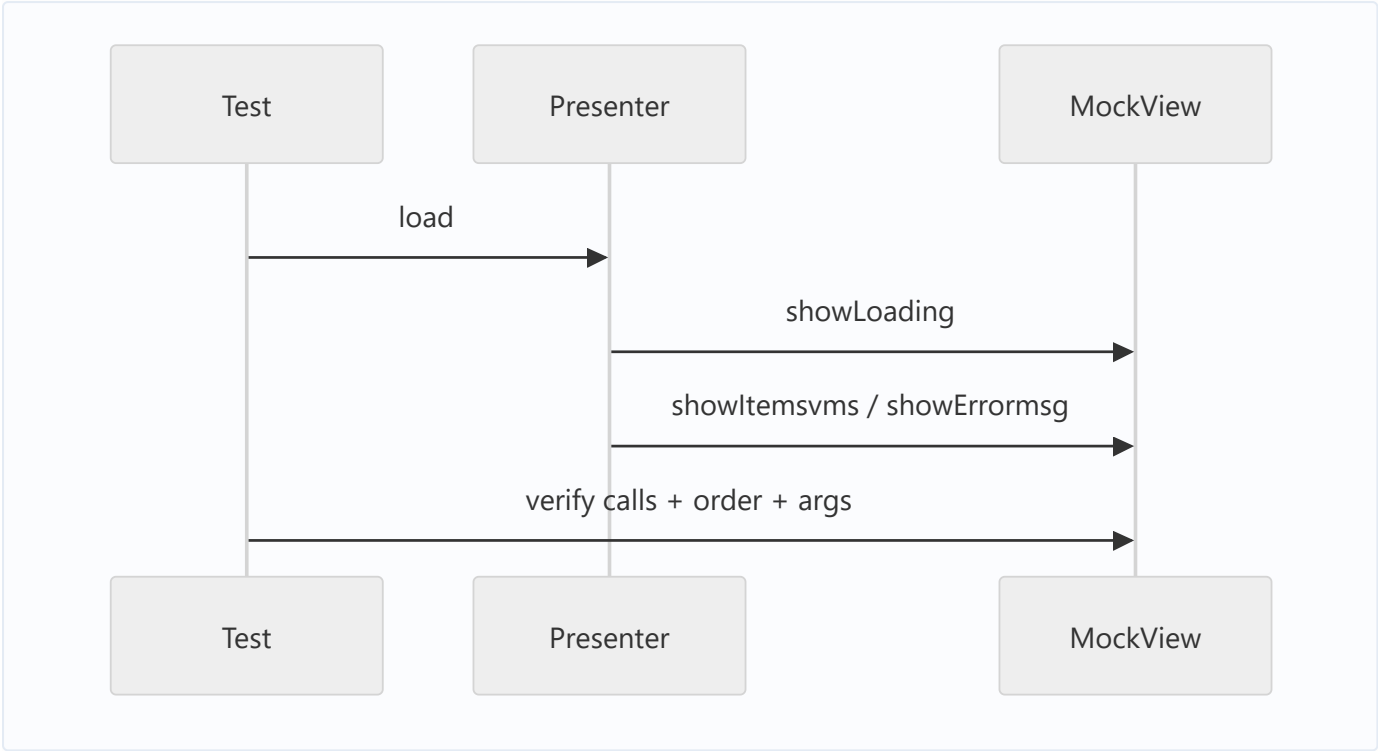
Fastest test tier (no Flutter binding); mocks/fakes are cheap objects.

Examples

```
import 'package:mocktail/mocktail.dart';
import 'package:test/test.dart';

class MockView extends Mock implements ProductListView {} // mock the contract
class FakeGetProducts extends Fake implements GetProducts {
  final Result<List<Product>> result;
  FakeGetProducts(this.result);
  @override
  Future<Result<List<Product>>> call() async => result;
}

void main() {
  test('load: shows loading then items on success', () async {
    final view = MockView();
    final presenter = ProductListPresenter(FakeGetProducts(Success([Product('1','A')])));
    presenter.attach(view);

    await presenter.load();

    verifyInOrder([                       // assert methods + ORDER
```

```
      () => view.showLoading(),
      () => view.showItems(any()),        // (could capture + assert mapped view models)
    ]);
    verifyNever(() => view.showError(any()));
  });

  test('load: shows error on failure', () async {
    final view = MockView();
    final presenter = ProductListPresenter(FakeGetProducts(Failure(NetworkFailure())));
    presenter.attach(view);

    await presenter.load();

    verify(() => view.showLoading()).called(1);
    verify(() => view.showError(any())).called(1);   // mapped message
  });

  test('tap navigates once with id', () {
    final view = MockView();
    final presenter = ProductListPresenter(FakeGetProducts(Success([]))..attach(view);
    presenter.onItemTapped('42');
    verify(() => view.navigateToDetail('42')).called(1); // effect fired once
  });
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Testing through the real widget	Slow/flaky	Mock the view; unit-test the presenter
Presenter depends on the concrete widget	Can't mock	Depend on the view interface
Logic in the view	Untestable via presenter tests	Keep view passive; logic in presenter
Not asserting order/args	Misses sequencing/mapping bugs	`verifyInOrder` + capture args
Real I/O in tests	Slow/nondeterministic	Fake use cases/repositories
Not testing effect-once	Re-fire bugs slip through	Assert effect called exactly once
Skipping failure/empty scenarios	Unhappy paths untested	Cover success/empty/error

Best Practices

- **Mock the view** (implement the contract) and **fake the dependencies** (canned `Result`s); **unit-test the presenter** in plain Dart (no device).
- Assert **which methods, in what order, with what arguments** (`verifyInOrder`, arg capture) — covering **loading→data/error** sequencing, **mapping** correctness, and **effects fired once**.
- Cover **success/empty/failure** scenarios; keep the **view passive** so presenter tests cover the logic and view tests stay thin.

- Recognize MVVM offers the **same testability** by asserting view-model **state sequences** — choose the reactive variant for Flutter.

Performance

Presenter tests are the fast unit tier (no binding/UI/I/O) — run in milliseconds, ideal for CI. This test speed is MVP's practical payoff. Thin view tests (if any) are the slower widget tier but minimal since the view has no logic.

Advantages / Disadvantages

- + Fast, exhaustive, device-free tests of all presentation logic; unhappy paths easy to cover; clear split (presenter logic vs thin view).
- − Requires interface + mocks (boilerplate); imperative call-verification can be verbose; MVVM achieves the same with less ceremony in Flutter.

Interview Questions

1. 🟢 **How do you test presentation logic in MVP?** — Mock the view (implement the contract), fake the dependencies, drive the presenter, and verify the contract calls (methods/order/args) — plain Dart, no device.
2. 🟢 **Why is the presenter easy to test?** — It depends on the view interface and injected fakes, so it runs without a widget/device and its behavior is fully observable via the mock.
3. 🟡 **What should presenter tests assert?** — Which contract methods were called, in what order (loading before data/error), with what (mapped) arguments, and that effects fire exactly once.
4. 🟡 **How much do you test the view?** — Little — it's passive (no logic); a thin widget test that it renders contract calls, if anything.
5. 🟡 **Why fake dependencies instead of using real I/O?** — For fast, deterministic tests; you control success/empty/failure scenarios.
6. 🔴 **How does MVVM achieve similar testability differently?** — By asserting the view model's emitted state sequence (loading→data/error) instead of verifying imperative view calls — more reactive-friendly.
7. 🔴 **What bug class does asserting "effect called once" catch?** — Re-firing navigation/snackbars on re-render (effects mismodeled as state).

Senior Engineer Tips

- Verify order and arguments, not just that a method was called — most presentation bugs are wrong sequencing (data before loading cleared) or wrong mapping, which only order/arg

assertions catch.

- Cover the unhappy paths (empty, network failure, validation) explicitly; MVP makes them trivial to test, so there's no excuse to skip them.
- If the ceremony of mocking imperative view calls feels heavy, that's a signal MVVM (state-sequence assertions) may suit your Flutter app better — same testability, less boilerplate.

Architect Perspective

MVP's testability is a direct dividend of the view contract + DIP: isolate presentation logic behind a mockable interface and it becomes fast, exhaustive, device-free unit tests. This is MVP's enduring lesson — testable presentation logic — and it's why the pattern mattered. In reactive Flutter, MVVM inherits the lesson while fitting the framework, asserting state sequences instead of mocked calls; either way, the principle is: **presentation logic belongs in a plain-Dart, testable unit, not the widget** (02_presenter_and_view_contract.md, Module 43, Module 49).

Summary

- MVP's key benefit: mock the view (contract) + fake dependencies → unit-test the presenter in plain Dart (fast, device-free).
- Assert methods/order/args (loading→data/error, mapping, effect-once); cover success/empty/failure; view tests stay thin (passive view).
- MVVM achieves the same testability via state-sequence assertions — the reactive-friendly variant.

Revision Notes

- Mock the view (implements contract) + fake use cases (canned `Result`); drive presenter; `verifyInOrder`/arg capture (methods/order/args, effect-once).
- Plain-Dart fast tier (no binding/device); passive view → thin/no view tests; cover success/empty/failure.
- Same testability in MVVM via view-model state-sequence assertions (less ceremony, reactive fit).

Practice Questions

1. What exactly do you assert in a presenter unit test?
2. Why can these tests run without a device or Flutter binding?
3. How does MVVM test the same logic differently?

Coding Questions

1. Write presenter tests with a mock view for success, empty, and failure.
2. Assert method order and mapped arguments (`verifyInOrder` + capture).
3. Verify a one-off effect (navigation) fires exactly once.

Mini Project

Presenter test suite (Flutter): For the product-list presenter, write a plain-Dart test suite using a mocked view + fake use cases covering: `load` → `showLoading` then `showItems` (success), `showEmpty` (empty), `showError` (failure), and tap → `navigateToDetail(id)` once — asserting order and mapped arguments. Acceptance: presenter tested via mock view (no device); order + args asserted; success/empty/failure covered; effect-once verified; runs in the fast unit tier.

MVP in Flutter & Comparison

> MVP fits Flutter **worse than MVVM** because its **imperative presenter-drives-view** flow (`view.showX()`) fights Flutter's **declarative rebuild-from-state** model: you end up translating presenter calls back into `setState`/controllers, fighting effect-vs-state re-fires, and juggling attach/detach against the widget lifecycle. MVP's prize — **testable presentation logic via a mockable contract** — is real, but **MVVM delivers the same testability** by asserting **observable state sequences**, without the imperative friction. So in Flutter, MVP is **rarely idiomatic**; prefer MVVM, and understand MVP mainly for the testability lesson and legacy/Android-influenced codebases.

Introduction

This file assesses MVP against Flutter's grain and compares it to MVC and MVVM, giving a clear recommendation. It's the decision-context before the capstone (05_mvp_integration.md).

Why this concept exists

Developers coming from Android (where MVP was popular) or seeking testability may reach for MVP in Flutter. Knowing precisely where MVP's imperative model clashes with Flutter's reactivity — and that MVVM offers equal testability with better fit — prevents adopting a pattern that fights the framework.

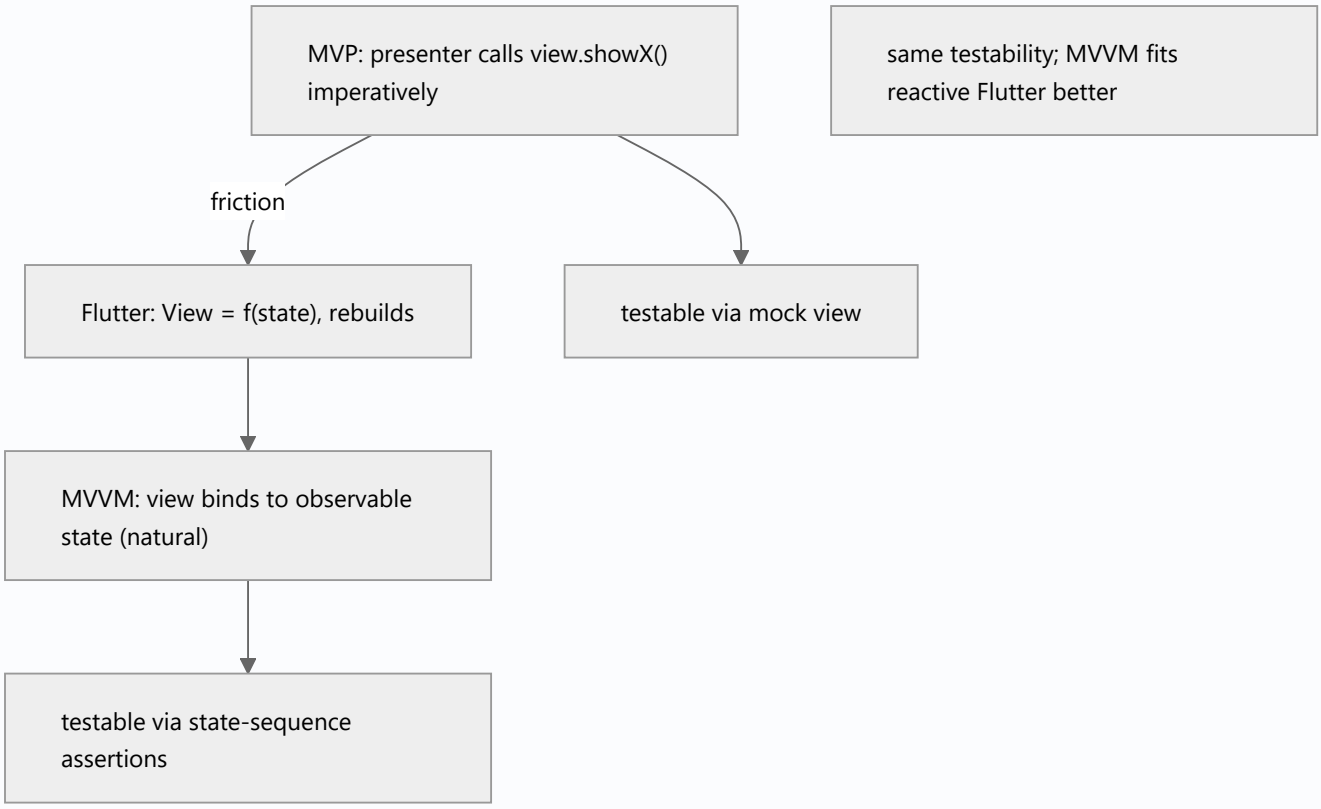
Real-world analogy

MVP in Flutter is like **using a manual film camera's workflow (load, wind, expose each frame imperatively) on a digital camera**: it works, but you're fighting a device designed for continuous live preview. MVVM is the **digital-native workflow** (compose the shot, the screen updates live). Both can produce great photos and both let you review results (testability), but one matches the tool.

Problem Statement

Decide whether to use MVP for a Flutter feature: weigh its testability against the reactive friction, and compare with MVC and MVVM. You'll evaluate fit and pick the idiomatic pattern.

Internal Working



- **Where MVP strains in Flutter**:
  - **Imperative vs declarative**: MVP presupposes a persistent view you **call methods on**; Flutter **rebuilds** the view from state. You must translate `showItems(...)` into `setState`/holding widget state — re-introducing the very state-in-widget MVP tried to remove.
  - **Effect vs state re-fires**: mapping one-off `navigate/showSnackbar` onto a rebuild model risks re-firing on rebuild (02_presenter_and_view_contract.md); you must add command-once handling.
  - **Lifecycle**: attach/detach must track the widget lifecycle (dispose) carefully to avoid calling a gone view — extra bookkeeping Flutter's reactive holders handle more naturally.
  - **Boilerplate**: a view interface per screen + wiring, on top of Flutter's already-declarative UI.
- **What MVP still gives**: a **fully mockable view contract** → exhaustive presentation-logic tests. But…
- **MVVM achieves the same testability differently**: the view model exposes **observable state**; you test by asserting the **emitted state sequence** (loading→data/error), with **no mock view** and **no imperative calls** — and the view **binds** naturally (UI = f(state)). Same guarantee, better fit (Module 43).
- **Comparison**:

    - **MVC** (Module 41): looser, controller mediates/view may read model; in Flutter → reactive controller (≈ MVVM). Simpler, less testability discipline than MVP.
    - **MVP**: strict passive view + mockable contract → best explicit testability contract, worst reactive fit + most boilerplate.
    - **MVVM**: observable state the view binds to → **best Flutter fit**, equal testability (state sequences), less imperative friction. **Recommended for Flutter.**
    - **Clean Architecture**: orthogonal layering; combine with MVVM (or MVP) in the presentation layer.
- **Recommendation**: In Flutter, **prefer MVVM**; use **MVP** only for **legacy/ported** codebases or teams with strong MVP conventions who accept the friction. Understand MVP for the **testability principle**, which MVVM carries forward.

Memory Representation

Not applicable — comparative. MVP holds view-interface refs + attach/detach state; MVVM holds observable state the view subscribes to. The difference is imperative refs vs reactive subscriptions.

Compiler Behavior

Both compile against abstractions (MVP: view interface; MVVM: none needed for view — it binds to state). MVP requires implementing the full interface in the widget.

Runtime Behavior

MVP: presenter calls view methods → widget imperatively updates. MVVM: state changes → bound widgets rebuild. The latter matches Flutter's engine model; the former requires manual translation.

Flutter Engine Behavior

Flutter is built to rebuild from state; MVVM's binding aligns with the diff/rebuild pipeline, while MVP's imperative calls must be adapted into that pipeline (via `setState`/controllers) — friction, not impossibility.

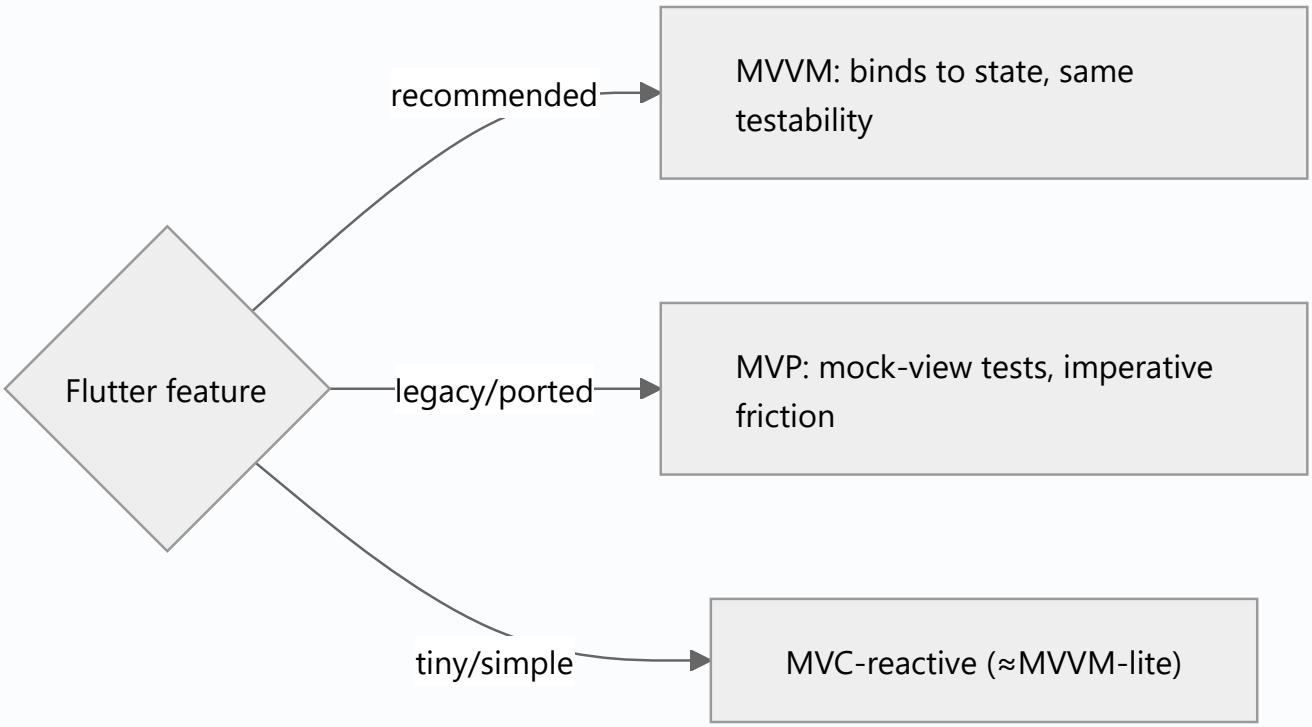
Dart VM Behavior

Both keep presentation logic in plain Dart (testable). MVP tests verify mock calls; MVVM tests verify state streams — both fast, device-free.

Examples

```
// Same feature, two flavors:


// MVP (imperative) — presenter drives the view; widget must translate to setState
// presenter.load(); -> view.showLoading(); view.showItems(vms);
// class _S extends State implements ProductListView {
//   @override void showItems(List<ProductVm> v) => setState(() => _items = v); // translate!
// }


// MVVM (reactive) — view model exposes state; widget binds; no imperative calls
class ProductListViewModel extends ChangeNotifier {
  ProductListState state = const ProductListState.loading();
  final GetProducts getProducts;
  ProductListViewModel(this.getProducts);
  Future<void> load() async {
    state = const ProductListState.loading(); notifyListeners();
    final r = await getProducts();
    state = r is Success ? ProductListState.data(r.value.map(ProductVm.from).toList())
                         : const ProductListState.error('Failed');
    notifyListeners();                                  // view rebuilds from state
  }
}
// Test: assert the state sequence [loading, data] — no mock view, no imperative calls.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Choosing MVP for testability in Flutter	MVVM gives it with better fit	Prefer MVVM
Fighting reactivity with imperative view calls	Reintroduces state-in-widget	Bind to state (MVVM)
Ignoring effect-once in MVP	Re-fires on rebuild	Command-once handling
Sloppy attach/detach	Calls to disposed view	Track lifecycle carefully
Treating Clean as an MVP alternative	Orthogonal	Combine layering + presentation pattern
Porting Android MVP verbatim	Framework mismatch	Adapt to MVVM in Flutter

Best Practices

- In Flutter, **prefer MVVM** (view binds to observable state) — it gives **MVP-equal testability** (state-sequence assertions) **without** the imperative friction/boilerplate.
- Use **MVP only** for legacy/ported code or strong-convention teams accepting the friction; if you do, add **effect-once** handling and careful **attach/detach**.
- Keep presentation logic in **plain-Dart, testable** units regardless of pattern; **combine** the chosen presentation pattern with **Clean layering** as needed.

- Take MVP's **lesson** (mockable, testable presentation logic) forward into MVVM; **don't fight** Flutter's reactive model.

Performance

No inherent perf difference; MVVM's scoped bindings can be very efficient, while MVP's imperative updates still ultimately rebuild widgets. The friction is developer-facing (boilerplate/bugs), not runtime. Scope rebuilds regardless (Module 21).

Advantages / Disadvantages

- **MVP** **+** explicit contract, strong mockable testability, truly passive view; **−** imperative friction in Flutter, boilerplate, effect/lifecycle handling.
- **MVVM** **+** reactive fit, equal testability, less friction; **−** binding plumbing (minor). **In Flutter, MVVM wins.**

Interview Questions

1. 🟢 **Why does MVP fit Flutter worse than MVVM?** — Its imperative presenter-drives-view flow fights Flutter's declarative rebuild-from-state; you must translate calls into `setState`, reintroducing widget state.
2. 🟢 **What's MVP's main advantage, and does MVVM share it?** — Testable presentation logic via a mockable contract; yes — MVVM achieves equal testability via state-sequence assertions.
3. 🟡 **How do MVP and MVVM test presentation logic differently?** — MVP verifies imperative calls on a mock view; MVVM asserts the emitted observable-state sequence (no mock view).
4. 🟡 **What extra concerns does MVP add in Flutter?** — Effect-once handling (avoid re-fire on rebuild) and careful attach/detach against the widget lifecycle.
5. 🟡 **When is MVP justified in Flutter?** — Legacy/ported (e.g., Android) codebases or teams with strong MVP conventions accepting the friction.
6. 🔴 **Is Clean Architecture an alternative to MVP?** — No — it's orthogonal layering; combine it with a presentation pattern (MVVM recommended).
7. 🔴 **What's the recommendation and why?** — Prefer MVVM in Flutter: same testability, natural reactive fit, less boilerplate/friction than MVP.

Senior Engineer Tips

- Reach for MVVM by default in Flutter; if someone proposes MVP "for testability," point out MVVM gives the same guarantee via state-sequence tests without fighting rebuilds.

- If you must maintain MVP (ported code), add explicit one-shot effect handling and rigorous attach/detach — those are where MVP-in-Flutter bugs cluster.
- Keep the transferable lesson: presentation logic in a plain-Dart testable unit — MVP taught it, MVVM delivers it idiomatically.

Architect Perspective

MVP and MVVM share a goal (testable presentation logic, dumb view) but differ in mechanism: MVP's **imperative contract** vs MVVM's **reactive binding**. Flutter's declarative engine makes MVVM the natural realization, so MVP's value in Flutter is mostly conceptual/legacy. The architectural takeaway is pattern-agnostic and enduring: isolate presentation logic in a testable unit and let a dumb view reflect it — with MVVM the idiomatic Flutter form and Clean Architecture the orthogonal layering (03_mvp_testability.md, Module 43, Module 40).

Summary

- MVP's imperative presenter-drives-view clashes with Flutter's declarative rebuild-from-state (friction + boilerplate + effect/lifecycle handling).
- MVP's testability (mock view) is matched by MVVM via state-sequence assertions — with better reactive fit.
- In Flutter, prefer **MVVM**; use MVP only for legacy/ported code; combine any presentation pattern with Clean layering.

Revision Notes

- MVP imperative (`view.showX()`) vs Flutter declarative (View=f(state)) → friction: translate to `setState`, effect-once, attach/detach lifecycle, boilerplate.
- Testability: MVP mocks the view; MVVM asserts state sequence (same guarantee, reactive fit, less ceremony).
- Comparison: MVC (loose→reactive), MVP (strict contract, worst fit), MVVM (recommended); Clean = orthogonal layering; prefer MVVM in Flutter, MVP for legacy only.

Practice Questions

1. Precisely where does MVP's flow conflict with Flutter's?
2. How does MVVM match MVP's testability without a mock view?
3. When, if ever, would you choose MVP in Flutter?

Coding Questions

1. Rewrite an MVP presenter+view as an MVVM view model + bound view.
2. Show the MVP effect-once problem and its fix.
3. Write the equivalent test both ways (mock-view calls vs state sequence).

Mini Project

MVP vs MVVM comparison (Flutter): Implement the same list feature twice — MVP (view interface + presenter driving it + widget translating calls, with effect-once + attach/detach) and MVVM (view model with observable state + bound view) — and write the equivalent test for each (mock-view call verification vs state-sequence assertion). Document the friction points and recommend one. Acceptance: both implemented + tested; MVP friction (imperative translation, effect-once, lifecycle) demonstrated; MVVM shown as the idiomatic fit with equal testability; recommendation justified.

MVP Integration (Capstone: An MVP Feature + Mocked-View Test)

> Build one complete MVP feature — **Model** (use cases/repository), **view contract** (interface), **Presenter** (drives the view, plain Dart, attach/detach, effect-once), and a **passive widget** implementing the contract — then write the **hallmark test**: unit-test the presenter against a **mocked view**, asserting the contract calls (order/args) for success/empty/failure. Finish with an honest note that in reactive Flutter you'd usually pick **MVVM** (same testability, less friction), making this capstone a demonstration of the **pattern + its testability**, not a production recommendation.

Introduction

This module capstone assembles fundamentals, the view contract, testability, and the Flutter comparison into one working MVP feature with its defining test. It shows MVP done correctly (so you understand it fully) and closes with the practical recommendation.

Why this concept exists

Seeing MVP end-to-end — especially the mocked-view presenter test — cements both its mechanics and its core value (testable presentation logic). Building it also makes the imperative friction tangible, reinforcing why MVVM is the idiomatic Flutter choice while carrying MVP's testability lesson forward.

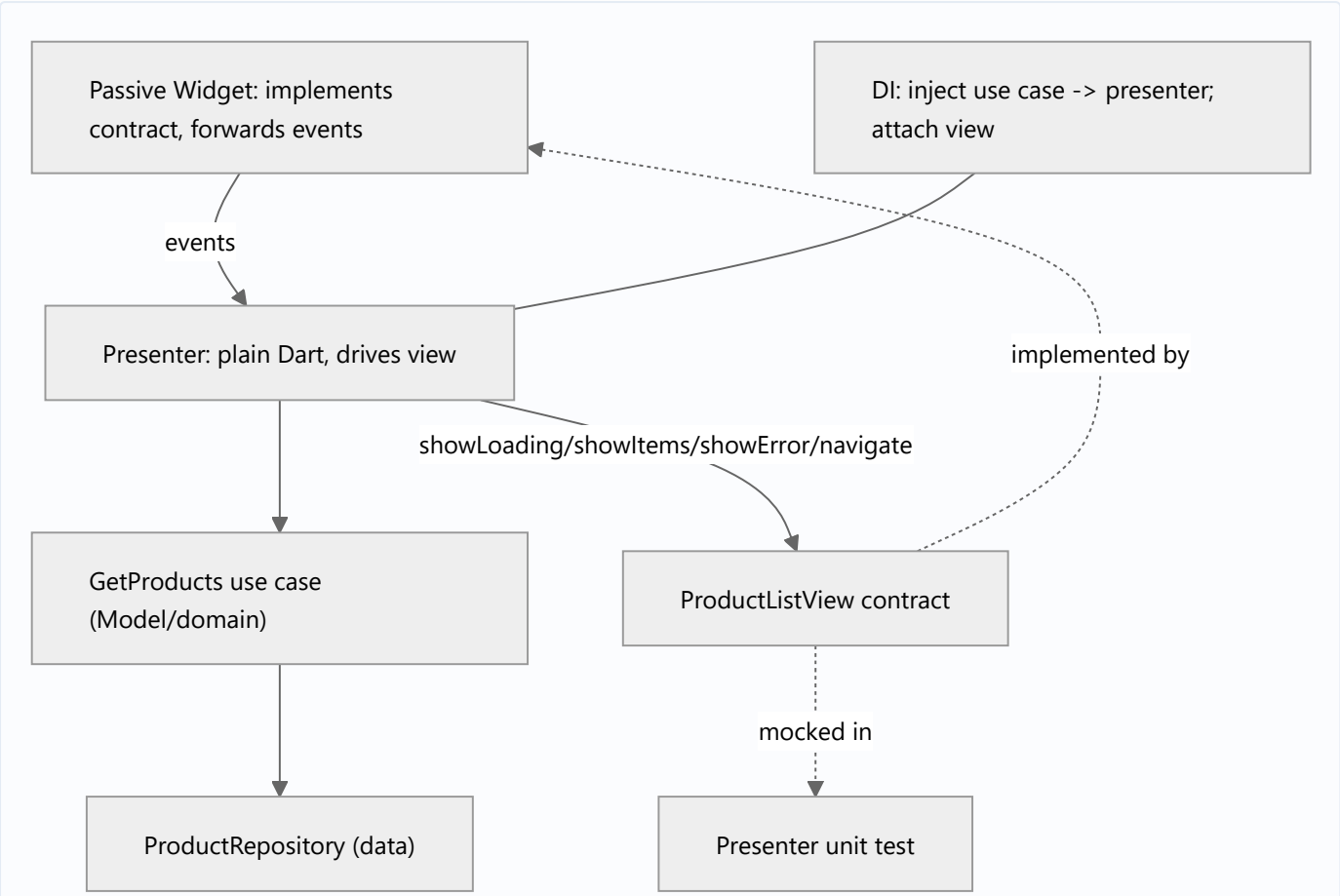
Real-world analogy

This is a **full dress rehearsal with a stand-in crew**: you stage the whole production (Model/contract/presenter/view), then rehearse the director (presenter) against a stand-in that records every cue (mock view) to prove the logic — while noting that on this particular (digital/reactive) stage, a live-preview workflow (MVVM) would be the natural way to run the show.

Problem Statement

Build a "products" feature in MVP: `GetProducts` use case + repository (Model), a `ProductListView` contract, a `ProductListPresenter` (drives the view, attach/detach, effect-once navigation), and a passive widget implementing the contract — then unit-test the presenter with a mocked view for success/empty/failure/tap. You'll compose the module into one slice + its test.

Internal Working



- **Model** (Module 40): `GetProducts` use case over a `ProductRepository`; rules/IO live here, returning `Result`.
- **View contract** (02_presenter_and_view_contract.md): `ProductListView` with presentation-only methods — state (`showLoading/showItems(vm)/showEmpty/showError(msg)`) + effect (`navigateToDetail(id)`).
- **Presenter** (01_mvp_fundamentals.md): plain Dart; `attach/detach`; on `load()` drives `showLoading` → maps `Result` → `showItems` / `showEmpty` / `showError`; on tap → `navigateToDetail` (once). No Flutter imports, no rules/IO (delegated).
- **Passive widget**: implements `ProductListView`, translating contract calls into UI (`setState` /controllers), and forwards events (`initState` → `presenter.attach(this)` + `load()`; `dispose` → `detach()`; button→ `presenter.onItemTapped(id)`). No logic beyond rendering.
- **Wiring (DI)**: inject the use case into the presenter; the widget owns/attaches the presenter (Module 14).
- **The hallmark test** (03_mvp_testability.md): mock `ProductListView`, fake `GetProducts`, drive the presenter, and `verify` the contract calls (order/args) for success/empty/failure + effect-once — plain Dart, no device.
- **Honest note** (04_mvp_in_flutter_and_comparison.md): document the imperative friction (contract-call translation, effect-once, attach/detach) and that **MVVM** would give equal

testability with better fit — so MVP here is educational/legacy, not the default.

Memory Representation

Presenter holds the view interface + use case (+ attach state). Widget holds the presenter + local UI state it sets from contract calls. Test holds a mock (records calls) + fakes (canned results).

Compiler Behavior

Presenter compiles UI-free (against the interface); widget + mock must implement the full contract. DI wires use case → presenter.

Runtime Behavior

`initState` attaches + loads → presenter calls `showLoading` / `showItems` /etc. → widget updates; tap → `navigateToDetail` once; `dispose` detaches (no calls to a gone view). Imperative throughout.

Flutter Engine Behavior

The widget translates contract calls into `setState` /rebuilds — the adaptation of MVP's imperative model to Flutter's reactive engine (04_mvp_in_flutter_and_comparison.md).

Dart VM Behavior

Presenter/use case/model tests are fast plain Dart; only the (thin) widget test uses the Flutter binding.

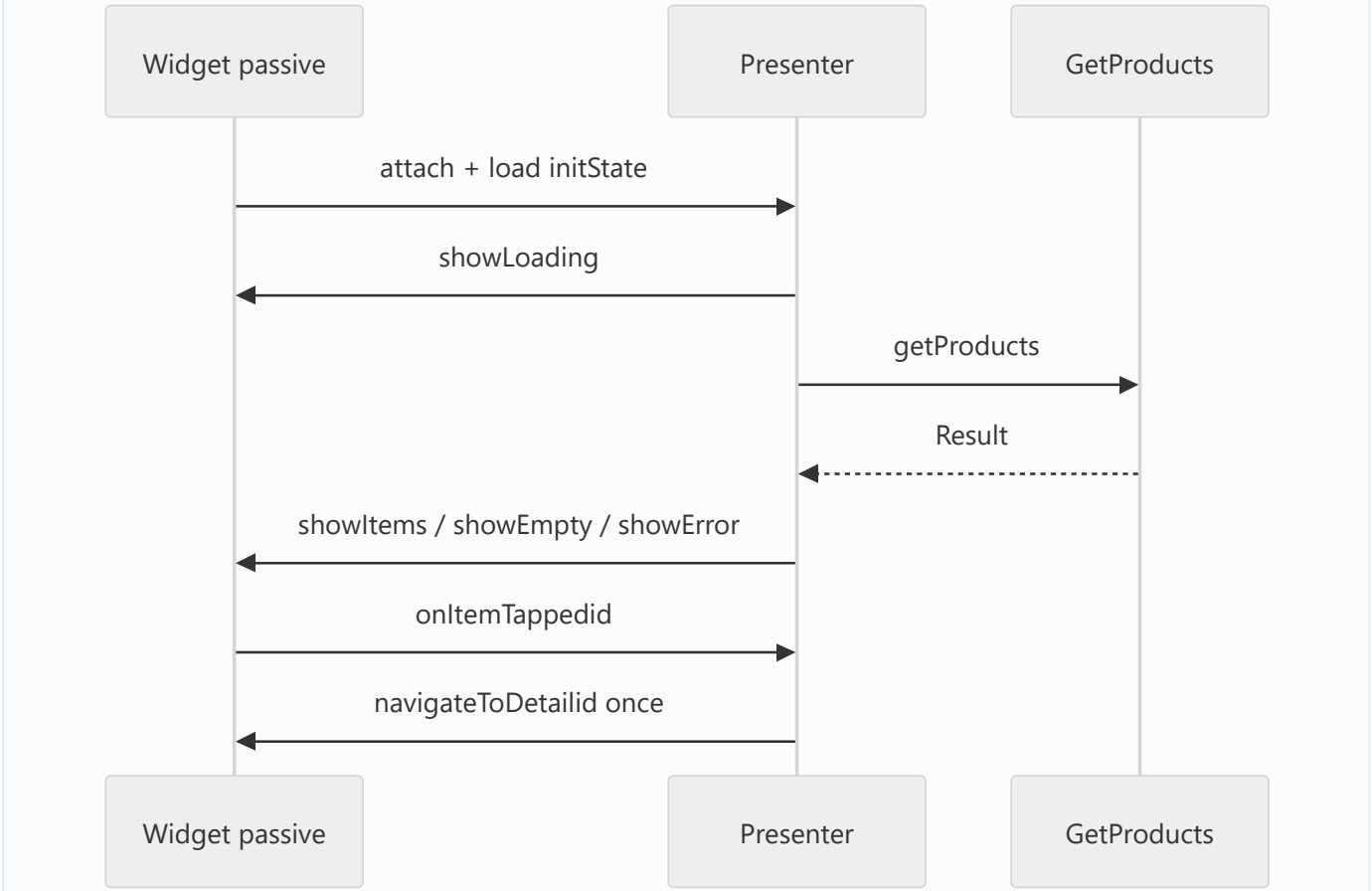
Examples

```
// PRESENTER — drives the contract; plain Dart; attach/detach; effect-once
class ProductListPresenter {
  final GetProducts getProducts;
  ProductListView? _view;
  ProductListPresenter(this.getProducts);
  void attach(ProductListView v) => _view = v;
  void detach() => _view = null;

  Future<void> load() async {
    _view?.showLoading();
    final r = await getProducts();
    switch (r) {
      case Success(:final value) when value.isEmpty: _view?.showEmpty();
      case Success(:final value): _view?.showItems(value.map(ProductVm.from).toList());
      case Failure(:final error): _view?.showError(messageFor(error));
    }
  }
  void onItemTapped(String id) => _view?.navigateToDetail(id); // one-off effect
}
```

```
// PASSIVE WIDGET — implements the contract, translates calls to UI, forwards events
class ProductListScreen extends StatefulWidget { /* ... */ }
class _State extends State<ProductListScreen> implements ProductListView {
  late final ProductListPresenter presenter = ProductListPresenter(getIt());
  Widget _body = const SizedBox();
  @override void initState() { super.initState(); presenter.attach(this); presenter.load(); }
  @override void dispose() { presenter.detach(); super.dispose(); }         // lifecycle
  @override void showLoading() => setState(() => _body = const CenteredSpinner());
  @override void showItems(List<ProductVm> v) => setState(() => _body = ProductList(v,
      onTap: presenter.onItemTapped));
  @override void showEmpty() => setState(() => _body = const EmptyState('No products'));
  @override void showError(String m) => setState(() => _body = ErrorView(message: m, onRetry:
presenter.load));
  @override void navigateToDetail(String id) => Navigator.pushNamed(context, '/product/$id'); // effect
once
  @override Widget build(BuildContext c) => Scaffold(body: _body);
}
```

```
// HALLMARK TEST — mock the view, verify contract calls (no device)
test('load success: showLoading then showItems', () async {
  final view = MockProductListView();
  final p = ProductListPresenter(FakeGetProducts(Success([Product('1','A')])))..attach(view);
  await p.load();
  verifyInOrder([() => view.showLoading(), () => view.showItems(any())]);
  verifyNever(() => view.showError(any()));
});
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Logic in the widget	View not passive	All logic in the presenter
Presenter imports Flutter/holds widget	Untestable/coupled	Plain Dart + view interface
No attach/detach	Calls to disposed view	Lifecycle in initState/dispose
Effect re-fires	Navigate on rebuild	One-off effect handling
Rules/IO in presenter	Wrong layer	Delegate to use cases/repo
Only happy-path tests	Misses unhappy paths	Test success/empty/failure/effect
Presenting MVP as the Flutter default	Ignores reactive friction	Note MVVM is idiomatic

Best Practices

- Compose **Model (use cases/repo) + view contract + plain-Dart Presenter (attach/detach, effect-once) + passive widget**; wire via DI.
- Keep the **presenter UI-free** (drives the interface, delegates rules/IO) and the **widget passive** (translate contract calls, forward events).
- Write the **mocked-view presenter test** covering **success/empty/failure + effect-once** (order/args) — MVP's defining payoff.
- **Document the friction + recommend MVVM** for reactive Flutter; treat this as understanding-the-pattern, not a production default.

Performance

No architectural overhead; the widget still ultimately rebuilds via `setState`. Presenter tests are the fast unit tier. The costs are boilerplate/friction (dev-facing), which is exactly why MVVM is preferred in Flutter.

Advantages / Disadvantages

- **+** Demonstrates a clean, fully-testable contract; passive view; strong mocked-view tests; conceptual clarity.
- **−** Imperative friction (contract→ `setState` translation), boilerplate, effect/lifecycle handling; not idiomatic Flutter (MVVM preferred).

Interview Questions

1. 🟢 **What are the pieces of an MVP feature?** — Model (use cases/repo), view contract (interface), presenter (drives the view, plain Dart), passive widget implementing the contract.

2. 🟢 **What's the hallmark MVP test?** — Unit-testing the presenter against a mocked view, verifying contract calls (order/args) for success/empty/failure/effect — no device.
3. 🟡 **How does the passive widget adapt to Flutter?** — It translates contract calls (`showItems`, etc.) into `setState`/rebuilds and forwards events — the imperative-to-reactive adaptation.
4. 🟡 **Where do rules/IO and effect-once handling live?** — Rules/IO in use cases/repo (delegated); effect-once handled so navigation fires once (not on rebuild).
5. 🟡 **What lifecycle wiring is required?** — attach in `initState` (+ load), detach in `dispose`, null-safe view calls.
6. 🔴 **Why is this educational rather than the default in Flutter?** — The imperative friction/boilerplate; MVVM gives equal testability with a natural reactive fit.
7. 🔴 **How would you convert this to MVVM?** — Replace the contract+presenter with a view model exposing observable state; the widget binds and rebuilds; test by asserting the state sequence.

Senior Engineer Tips

- Build the presenter test first (mock view, fakes) — it clarifies the contract and proves the logic before any widget exists; that's MVP's whole point realized.
- Keep the widget a pure translator of contract calls and event forwarder; the instant it holds logic, MVP's benefit is gone.
- Ship the honest recommendation: understand MVP, prefer MVVM in Flutter — and know how to convert one to the other.

Architect Perspective

This capstone demonstrates MVP's mechanics and its enduring value (mockable, testable presentation logic) while making its reactive-Flutter friction concrete. The transferable takeaway — presentation logic in a plain-Dart testable unit, a dumb view reflecting it — is exactly what MVVM realizes idiomatically. Combined with Clean layering underneath, the pattern choice becomes: **MVVM presentation + Clean layering** for Flutter, with MVP understood as the explicit-contract ancestor (03_mvp_testability.md, Module 43, Module 40).

Summary

- MVP feature = Model (use cases/repo) + view contract + plain-Dart presenter (attach/detach, effect-once) + passive widget; wired by DI.
- Hallmark test: mock the view, verify contract calls (order/args) for success/empty/failure/effect — fast, device-free.
- Document the imperative friction and recommend MVVM for Flutter; know the MVP→MVVM conversion.

Revision Notes

- Pieces: use case/repo (Model) + view contract (interface) + presenter (drives view, plain Dart, attach/detach, effect-once, delegates rules/IO) + passive widget (translate calls, forward events); DI-wired.
- Test: mock view + fake use case; `verifyInOrder`/args for success/empty/failure + effect-once (no device).
- Friction (imperative→`setState`, effect-once, lifecycle) → document; prefer MVVM in Flutter; MVP→MVVM = contract/presenter → observable state/view model.

Practice Questions

1. Trace load() through the MVP pieces and its contract calls.
2. What exactly does the presenter test assert, and why no device?
3. How would you convert this feature to MVVM?

Coding Questions

1. Build the full MVP products feature (Model/contract/presenter/passive widget) + DI.
2. Write the mocked-view presenter test (success/empty/failure/effect).
3. Convert the presenter+contract to an MVVM view model + bound view.

Mini Project

MVP feature + presenter test (capstone — Flutter): Build a "products" feature in MVP — `GetProducts` use case + repository, a `ProductListView` contract, a plain-Dart `ProductListPresenter` (attach/detach, effect-once navigation, delegates rules/IO), and a passive widget implementing the contract — wired by DI. Write the hallmark presenter unit test with a mocked view (success/empty/failure/effect, order/args). Document the imperative friction and sketch the MVVM conversion. Acceptance: view passive; presenter UI-free + drives contract + delegates; attach/detach + effect-once handled; presenter unit-tested via mock view (no device); friction documented + MVVM conversion sketched; runs end-to-end.