**CLEAN ARCHITECTURE + TDD**

# Tổng quan

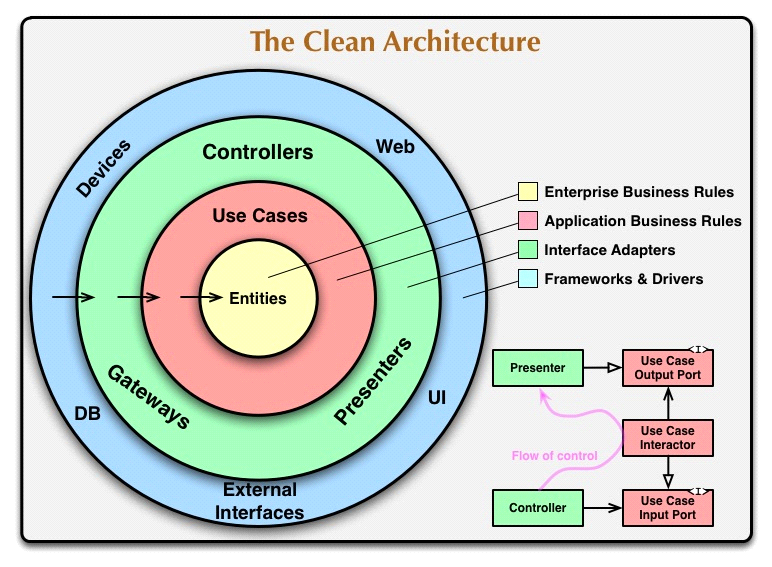
## Clean Architecture là gì?

Clean Architecture được xây dựng dựa trên tư tưởng "độc lập" kết hợp với các nguyên lý thiết kế hướng đối tượng (đại diện tiêu biểu là Dependency Inversion). Kiến trúc này tạo ra hệ thống:

* Independent of Frameworks (Độc lập các framework)
* Testable (Dễ kiểm thử)
* Independent of UI (Độc lập giao diện)
* Independent of DB (Độc lập về dữ liệu)
* Independent of any external agency

Kiến trúc của Clean Architecture chia thành 4 layer với một quy tắc phụ thuộc. Các layer bên trong không nên biết bất kỳ điều gì về các layer bên ngoài.

Dù định dạng dữ liệu của layer ngoài cùng như thế nào thì sẽ không ảnh hưởng đến layer bên trong, hay việc thay đổi business rule ở layer bên trong cũng không ảnh hưởng đến việc hiển thị ở layer bên ngoài.



**Entities:** là khái niệm dùng để mô tả các Business Logic. Đây là layer quan trọng nhất, là nơi bạn thực hiện giải quyết các vấn đề - mục đích khi xây dựng app. Layer này không chứa bất kỳ một framework nào, bạn có thể chạy nó mà không cần emulator. Nó giúp bạn dễ dàng test, maintain và develop phần business logic.

**Use case:** chứa các rule liên quan trực tiếp tới ứng dụng cục bộ (application-specific business rules).

**Interface Adapter:** tập hợp các adapter phục vụ quá trình tương tác với các công nghệ.

**Framework and Drivers:** chứa các công cụ về cơ sở dữ liệu và các framework, thông thường bạn sẽ không phải lập trình nhiều ở tầng này. Tuy nhiên cần chắc chắn về mức ưu tiên sử dụng các công cụ này trong project.

**\*\* Ưu điểm:**

* Mạch lạc, dễ xem
* Linh hoạt - thể hiện ở khả năng độc lập, không phụ thuộc vào framework, database, application server.
* Dễ mở rộng.
* Dễ kiểm thử - testable

**\*\* Nhược điểm:**

* Không thể sử dụng framework theo cách mỳ ăn liền- do luật dependency inversion.
* Khó áp dụng.
* Indirect - quá nhiều interface?
* Cồng kềnh - thể hiện ở việc có quá nhiều class so với các project cùng mục tiêu (tuy nhiên các class được thêm vào đều có chủ ý và đáp ứng đúng quy định khi triển khai kiến trúc)

## TDD (Test Driven Development) là gì?

TDD được hiểu nôm na là mô hình phát triển với trọng tâm hướng về việc kiểm thử. Mục đích chính của TDD là viết code rõ ràng, đơn giản và ít lỗi.

TDD được xây dựng theo hai tiêu chí:

* Test-First (Kiểm thử trước)
* Refactoring (Điều chỉnh mã nguồn)

Các bước thực hiện theo quy trình TDD:

* Tạo test
* Chạy test và check xem chắc chắn có lỗi
* Viết code
* Chạy test và refactor code (đương nhiên là refactor để pass test)
* Lặp lại các bước trên

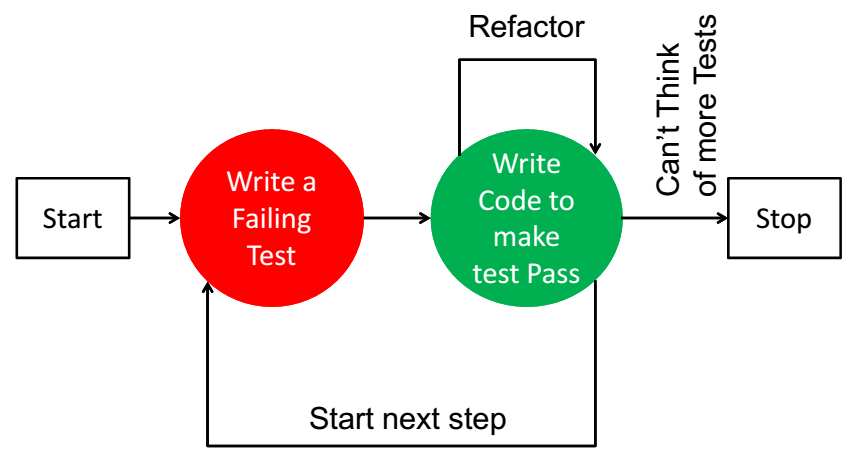


Figure 1. Các bước triển khai mô hình TDD

# Áp dụng mô hình Clean Architecture vào Flutter

## I. Explanation & Project Structure

Project được chia thành 2 phần chính:

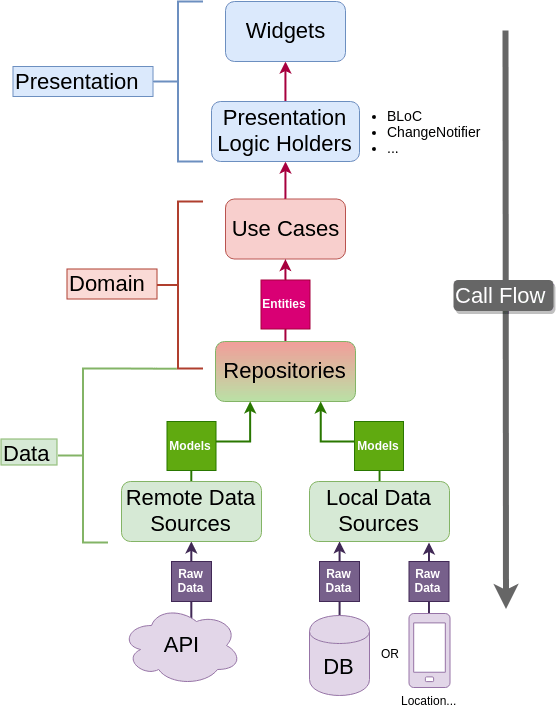
* Core: thực hiện các chức năng chung, cơ bản, dễ dàng share code giữa các layer như: lấy dữ liệu từ API, lưu cache local, error handling, validation, …
* Features: thực hiện các chức năng riêng biệt

+ Presentation (Outer layer)

+ Data (Middle layer)

+ Domain (Inner layer)

State management sử dụng BLOC pattern



### Domain

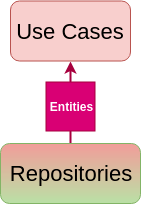


Figure 2. Domain layer

Đây là phần core layer chứa code ở mức level cao nhất, là phần phía trong cùng. Nó chỉ chứa các phần cốt lõi: **business logic**(**use cases**) and **business objects** (**entities**), an abstract **Repository** class: lớp này định nghĩa cái mà repository phải làm, sau đó triển khai các “contract” này ở phần Data layer.

Tầng Domain hoàn toàn độc lập với các tầng khác, có thế chạy ở bất kể máy ảo nào và không phụ thuộc vào framework nào



Figure 3. Domain folder structure

### Data

Tầng Data bao gồm:

* **Datasources:** lấy dữ liệu từ API, lưu bộ nhớ đệm dữ liệu
* **Repository implementation** (triển khai các “contract” từ tầng domain)
* **Models: data sources** không trả về dưới dạng **entities** mà là **models**, xử lý các dữ liệu thô thành các dữ liệu tương thích**.**

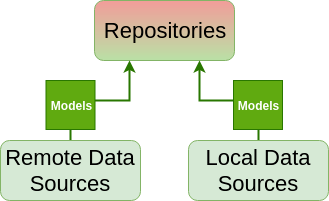


Figure 4. Data layer

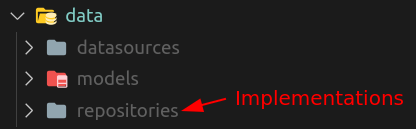


Figure 5. Data folder structure

### Presentation

Chứa UI, các widget, presentation logic

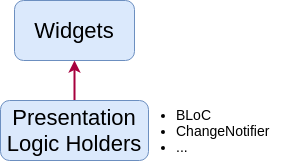


Figure 6. Presentation layer

Tầng Presentation bao gồm:

* **Bloc**: Xử lý phần logic giữa giao diện và dữ liệu
* **Pages**: Hiển thị phần giao diện
* **Widgets**: Chứa các thành phần con của giao diện

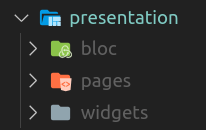


Figure 7. Presentation folder structure

## Entities & Use cases

**Usecases**: nơi thực thi business logic

**Usecase** sẽ get **Entity** từ **repository,** sau đó pass các entities -> **presentation** layer => Usecase sử dụng Future<> asynchronous

**Either<L, R>:**

**L là Failure, F là Result,**

**Handle error không dùng try/catch**

## Domain Layer Refactoring

Method call: thay vì object.call() => object()

## Data layer overview & models

Là nơi giao tiếp với APIs, 3rd party lib.

Repository implement Repository (Domain layer)

Trước khi impl, make remote and local Data Sources.

Models: là Entities có các chức năng bổ sung, parsed Json to Dart obj

## Contracts of Data Sources

Repository như là bộ não của Data layer. Nó handle data từ remote và local Data sources, quyết định Data Source.

Repository impl; fulfill “contract” ở Domain layer

the **Repository** needs lower level **Data Sources** to get the actual data from.

## Repository Implementation

## Network Info

## Local Data Source

## Remote Data Source

## Bloc Scaffolding & Input Conversion

## Bloc Implementation

Presentation Logic Holder

## Dependency Injection

Được gọi ngay khi khởi tạo hàm main()

Singleton

Get\_It stored inside sl (service locator)

Get\_it supports create singletons and instance Factories

Dùng **Factory** khi luôn muốn nhận về một **instance** mới mỗi khi sử dụng mà không liên quan gì đến **instance** trước để tránh trường hợp dùng lại các **data** cũ đã **init** từ **instance** trước

**Factory** không nên dùng nếu như **object** của bạn có chứa các **logic code quá phức tạp** dẫn đến việc làm chậm quá trình khởi tạo và lãng phí tài nguyên

**Singleton** trái ngược với **factory**, chỉ tạo ra một **instance** **duy nhất** kể từ khi app khởi động, sau đó nếu bất kì chỗ nào có dùng dến thì sẽ chỉ trả về instance đã tạo trước đó. Do đó xuyêt suốt app, bạn sẽ chỉ sử dụng một **instance** của **object** đó mà thôi.

Nên dùng **singleton** khi chỉ muốn khởi tạo **object một lần** và **dùng ở nhiều chỗ**, tránh lãng phí tài nguyên. Không nên dùng nếu như nó phụ thuộc quá nhiều về mặt giá trị và pointer, dễ gây lỗi app về mặt logic nếu không xử lí cẩn thận.

**Lazy-singleton** thì giống như singleton, chỉ khác là nó sẽ **được khởi tạo vào lần gọi lấy instance đầu tiên**, chứ không phải khi app khởi động.

## User Interface