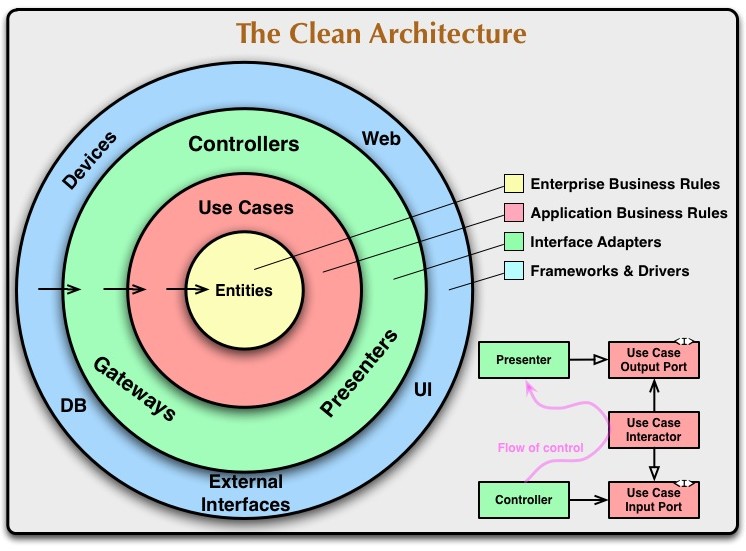
CẤU TRÚC PROJECT

VÀ QUY ƯỚC CODE ỨNG DỤNG FLUTTER

# Cấu trúc Project

* 1. **Kiến trúc Clean**

Để dễ dàng mở rộng, bảo trì, chuyển giao,… ứng dụng trong và sau khi phát triển team dự án lựa chọn chia source code thành các layer tách biệt dựa theo kiến trúc Clean.



Các lợi ích kiến trúc Clean mang lại:

* + - ***Độc lập với các framework***: không phụ thuộc vào sự tồn tại của các thư viện
    - ***Test dễ dàng:*** Các quy tắc nghiệp vụ có thể được test mà không có Giao diện, Cơ sở dữ liệu, Máy chủ Web,…
    - ***Không phụ thuộc vào giao diện người dùng***: Giao diện người dùng có thể thay đổi dễ dàng mà không làm thay đổi phần còn lại của hệ thống.
    - ***Độc lập với Cơ sở dữ liệu:*** Các quy tắc, nghiệp vụ không bị ràng buộc với cơ sở dữ liệu.
* **Quy tắc phụ thuộc (**The Dependency Rule**)**

Quan hệ phụ thuộc source code luôn luôn hướng vào trong. Càng vào trong mức độ trừu tượng càng tăng và đóng gói các nguyên tắc cấp cao hơn. Vòng tròn trong cùng là chung nhất. Bất cứ điều gì ở vòng tròn bên ngoài không được tác động đến vòng tròn trong. Đặc biệt tên của cái gì đó khai báo ở vòng tròn ngoài không được đề cập bởi code ở vòng tròn trong.

# Các layer

Các vòng tròn và thành phần trong vòng tròn như ảnh trên chỉ là phác hoạ, trên thực tế có thể nhiều hơn. Quan trọng là **luôn áp dụng quy tắc phụ thuộc.**

* + **Entities:** chứa các đối tượng, thực thể nghiệp vụ chính, ko phụ thuộc lib, framework nào.
  + **Use Cases:** nơi xử lý các quy tắc nghiệp vụ. Đóng gói và thực hiện tất cả các chức năng của hệ thống. Lớp này không bị ảnh hưởng bởi những thay đổi ngoại cảnh như DB, UI hoặc bất kỳ lib, framework nào.
  + **Interface Adapters:** có nhiệm vụ chuyển dữ liệu từ định dạng thuận tiện nhất cho các use case và entitity, thành định dạng thuận tiện nhất cho các chương trình bên ngoài như Database hoặc Web,… .Không có code bên trong vòng này nên biết bất kỳ điều gì về Database.
  + **Frameworks and Drivers:** là nơi chứa tất cả các chi tiết: DB, UI,…
* **Nguyên tắc Dependency Inversion** (Đảo ngược phụ thuộc)

Khi áp dụng được kiến trúc Clean cần áp dụng kèm Dependency Inversion:

1. Các module cấp cao không nên phụ thuộc vào các module cấp thấp. Cả 2 nên phụ thuộc vào abstraction.
2. Các class giao tiếp với nhau qua interface, không thông qua implementation

# Áp dụng kiến trúc Clean vào Project

# 

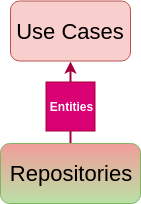
Core

Ứng dụng được chia thành 3 lớp chính: **presentation**, **domain** và **Infrastructure**

* **Presentation**

# 

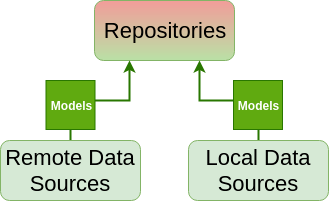
* + Chứa các widgets để hiển thị lên màn hình, quản lý trạng thái ứng dụng (BLoC,…), biến đổi và validation cơ bản các dữ liệu đầu vào…
  + Sử dụng Use case để xử lý nghiệp vụ chính
  + Các thành phần mở rộng nếu cần: utility, constant, cấu hình theme,…
* **Core**



* + Không dễ bị ảnh hưởng bởi những ý tưởng bất chợt của việc thay đổi nguồn dữ liệu hoặc chuyển đổi các thư viện có sẵn. Nó chỉ chứa logic nghiệp vụ cốt lõi và các thực thể, hoàn toàn độc lập với mọi lớp khác.
  + Repositories (Kho lưu trữ dữ liệu) là các abstract class xác định Repository phải làm những gì.

Ví dụ usecase Login cần một Repository nó có hàm login

* + Các thành phần mở rộng nếu cần: utility, constant,…
* **Infrastructure**



* + Triển khai Repository và Datasource thường lấy từ xa (gọi API) hoặc từ cache.
  + Repository quyết định trả dữ liệu mới hay đã lưu cache, khi nào lưu cache,…
  + Datasource không trả về Entity mà là Model giúp chuyển đổi dữ liệu raw như JSON sang dạng dễ dàng sử dụng nhất.
  + Các class Model chỉ là mở rộng của Entity bổ sung thêm các hàm, các trường dữ liệu,…
  + Các thành phần mở rộng nếu cần: utility, constant,…
* **injection\_container.dart:** nơi áp dụng Dependency Injection