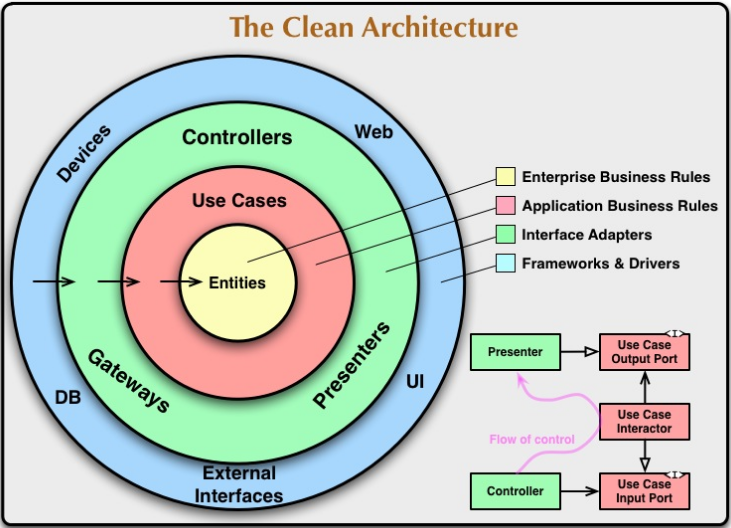
1. **Lý thuyết**

Clean Architecture là kiến trúc được đề xuất bởi Robert Martin (Uncle Bob).

* 1. Mục tiêu

Mục tiêu của Clean Architecture (CA) là ***tách biệt business logic của ứng dụng với các thành phần của framework như Database (DB), User Interface (UI).*** Sự tách biệt này đầu tiên giúp ta có những lợi ích như sau:

* Có thể dễ dàng viết và thực hiện test bởi vì ta có thể dễ dàng tạo ra các mock.
* Giảm thiểu sự ảnh hưởng do sự thay đổi UI và DB: bởi vì các thành phần như UI và DB có thể bị thay đổi trong lúc triển khai và trong quá trình recfactor (có thể do client, do khối lượng dự án…), nên sự tách biệt này sẽ giúp ta giảm số lượng code bị ảnh hưởng. Đồng thời cũng giúp cho người ra quyết định ra quyết định dễ dàng hơn vì không làm thay đổi phần lõi của ứng dụng.
  1. Nguyên tắc



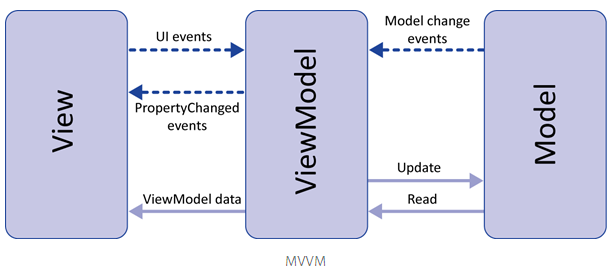
Để đạt được mục tiêu trên, CA đề ra các nguyên tắc như sau: Các thành phần của vòng tròn bên trong không nên phụ thuộc vào bất kì thành phần của vòng trong bên ngoài. Cụ thể, application logic và business logic không nên phụ thuộc vào các thành phần như UI, DB…

* 1. Cấu trúc dự án cho ứng dụng android



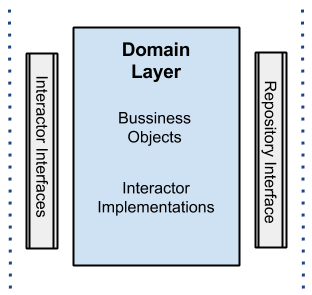
Mục đích của kiến trúc là tách các thành phần riêng biệt, không phụ thuộc lần nhau. Để đạt được điều này, ta chia dự án thành 3 layer khác nhau. Trong đó mỗi layer có mục đích và hoạt động riêng biệt với các layer khác.

1.3.1. Presentation Layer



Đây là tầng UI của ứng dụng. Ta có thể sử dụng các mô hình MVP, MVVM, MVC cho tầng này. Lưu ý model ở tầng này là model của view, đại diện (represent) cho mỗi view. Nó khác với model ở tầng Domain (hay còn gọi là entity/Bussiness Object). Tuy nhiên trong nhiều dự án có thể không có sự khác biệt giữa model của Presentation Layer và Domain Layer. Khi đó ta có thể dung chung model giữa 2 tầng mà không cần chuyển đổi (mapper). Tuy nhiên, theo quan điểm của tôi, ta nên tạo ra model riêng biệt cho mỗi tầng vì tầng view rất dễ thay đổi, dẫn đến model của nó cũng thay đổi theo. Điều đó sẽ ảnh hưởng tới Domain Layer và đó không phải là điều chúng ta mong muốn. Layer này sẽ sử dụng các UseCase ở tầng Domain để thực thi logic của nó.

1.3.2. Domain Layer

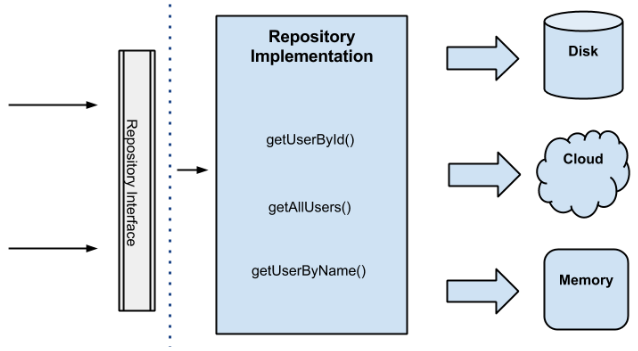


Trong cuốn sách **Pattern of Enterprise Application Architecture (**PoEAA) của Martin Fowler, ông có đề cập đến các pattern cho layer này như: Transaction script, Domain Model…. Và trong CA, Robert Martin đã đề xuất mô hình Domain Model cho layer này.

Tầng này có nhiệm vụ thực thi toàn bộ logic (application and business logic) của ứng dụng. Các class của tầng này là các class thuần Java (Kotlin). Nó mô tả toàn bộ logic lõi của bài toán. Layer này có 2 loại class là Entity và UseCase. Entity là các class đại diện cho bài toán mà ứng dụng giải quyết. Các logic của bài toán được đặt trong các class Entity. Nếu các logic này cần sự kết hợp của nhiều Entity, ta nên đưa nó vào class UseCase.

Theo DIP của nguyên lý SOLID, “Các module cấp cao không nên phụ thuộc vào module cấp thấp, cả hai nên phụ thuộc vào sự trừu tượng. Sự trừu tượng không nên phụ thuộc vào chi tiết mà nên ngược lại”. Do đó, Domain model ngoài chứa các class Entity và UseCase, nó cũng định nghĩa các interface Interactor (được gọi bởi Presenter) và Repository (được thực thi bởi Data Layer).

1.3.3 Data Layer



Tầng này có nhiệm vụ cung cấp instance của Entity cho Domain Layer và lưu trữ data. Domain layer sẽ không cần quan tầm Data layer lấy dữ liệu từ đâu (API, DB, Sharedpreference). Sự trừu tượng này cũng đưa đến những lợi ích như. Trong giai đoạn đầu của dự án, ta cần tạo nhanh data để có thể tập trung giải quyết business logic. Do đó, ta có thể tạo nhanh Data bằng Memory Data. Sau đó ta có thể thực thi lại data source để lấy data từ DB hoặc API mà không làm ảnh hưởng tới Domain Layer.



* 1. Tiến trình dự án thực thi theo CA
* Mô tả bài toán
* Xác định các use case

UseCase có nghĩa là trường hợp sử dụng của người dùng. Nghĩa là ta sẽ tìm các chức năng mà người dùng mong muốn từ ứng dụng theo góc nhìn của họ. Ta cần nhìn ứng dụng theo một góc nhìn rất thường, không nặng về yếu tố kỹ thuật.

Một UseCase sẽ bắt đầu bằng một action, đi kèm với một danh từ. Danh từ này thường là Entity của ứng dụng. Rất khó để có thể ước lượng 1 UseCase có mức độ logic thế nào là đủ. UseCase cần mô tả một hành động có mục đích rõ ràng với người dùng. Nghĩa là họ cần đạt được một thứ gì đấy sau khi thực hiện xong nó. Nếu UseCase quá lớn, ta có thể tách nó thành nhiều UseCase nhỏ hơn, và kết hợp các UseCase nhỏ tạo thành 1 UseCasse lơn đó để thỏa mãn chức năng bài toán

Ví dụ ta có một đoạn mô tả chức năng “Điều chỉnh âm lượng của Media Player”của một ứng dụng như sau:

User có thể điều chỉnh volume của bài hát với khoảng từ 0 – 100 ứng với giá trị phần cứng. Khi điều chỉnh, ứng dụng cần hiển thị cho người dùng về giá trị hiện tại của volume

Từ mô tả ta thấy có 2 action trong chức năng này là: ”điều chỉnh” và “hiển thị”. Từ đó ta có thể có 2 UseCase là “Update Volume” và “Get Volume”. Tuy nhiên với 1 vài developer, họ nhận thấy action “Get Volume” chỉ là một phần của action “Update Volume” và người dùng không có mong muốn hiển thị volume ở một giao diện khác. Do đó, họ coi action “Get Volume” là một bước trong UseCase “Update Volume” mà không phải là một UseCase độc lập. Bạn có thể lựa chọn một trong hai phương án và không có phương án nào là sai cả. Hãy nhớ, mọi cách làm giải quyết được feature của bài toán đều là cách làm đúng.

* Xác định các Entity

Để xác định các Entity, ta dựa vào các danh từ để xác định các đối tượng có tiềm năng nằm trong phạm vi mô tả của bài toán. Sau khi xác định các danh từ tiềm năng. Ta có thể dựa vào 3 câu hỏi sau để xác định danh từ đó có phải Entity hay không:

* Ứng dụng có muốn hiển thị nó lên cho người dùng không
* Ứng dụng có muốn lưu trữ nó lại hay không
* Nó có mô tả core logic của ứng dụng hay không
* Viết logic cho các entity

Các Entity chứa data và các logic của riêng nó. Hãy cố gắng tuân thủ principle “Single responseibility” khi xác định các logic cho các Entity. Nếu các logic cần sự kết hợp của nhiều hơn 2 Entity, nó sẽ được đưa vào UseCase.

* Thực thi các use case

Để thực thi các UseCase, ta cần mô tả các bước thực thi các senario, nghĩa các các trường hợp có thể xảy ra khi người dụng thực hiện UseCase này. Nó có thể là trườn hợp thành công và cũng có cả trường hợp thất bại.

Ví dụ với UseCase “Update Volume” ở trên, ta có các senario như sau:

* Senario 1 (Success):

1. Load Volume hiện tại
2. Quy đổi giá trị hiện tại với khoảng giá trị hiển thị 0 – 100
3. Hiển thị giá trị quy đổi lên cho người dùng
4. Người dùng điều chỉnh giá trị mới cho Volume
5. Chuyển đổi giá trị nhận được sang giá trị thực của Volume
6. Lưu lại giá trị vừa quy đổi

* Senario 2 (Không lấy được Volume)

1. Không lấy được Volume hiện tại
2. Hiển thị lỗi cho người dùng

* Senario 3 (Không quy đổi giá trị thực và giá trị hiển thị cho người dùng)

Giống senario 1 nhưng bỏ các bước 2 và 5

* Senario 4 (không lưu lại được giá trị đã điều chỉnh)

Giống các bước của senario 1 nhưng bước 6 không thành công, do đó bước 6 sẽ là “Lưu lỗi” và có thêm bước 7 “hiển thị lỗi cho người dùng”

…

Từ các senario, ta sẽ viết các method cho UseCase để cover tất cả các senario đó. Với các bước liên quan đến UI như: “Hiển thị giá trị cho người dùng”, “Hiển thị lỗi”… ta sẽ đưa chúng vào phần code UI của ứng dụng (ViewModel hoặc Presenter).

* Thực thi Presentation layer

Thực thi code theo pattern MVVM hoặc MVP. Viewmodel/Presenter sử dụng các UseCase ở trên và nó sẽ thực thi các logic liên quan đến UI của senario.

* Thực thi Data layer

Ta cần xác định các Entity sẽ được lưu lại. Ứng dụng có thể yêu cầu lưu data trong memory, DB hoặc API… Để tránh sự ảnh hưởng khi có sự thay đổi cách lưu trữ của bài toán. Ta sẽ trừu tượng hóa việc lưu data ở Data layer bằng pattern Repositoty.

1. Thực hành

**Ứng dụng “Monitor Location and battery”**

* 1. Mô tả bài toán

Programming an application for Android which:

• contains one screen with two buttons "START" and "STOP",

• When the user presses the "START" button, the program should create 3 different threads, say T1, T2 and T3:

* There should be a string list L to store the data collected by thread T1 and T2 as follows.
* In the first thread (T1), the information about the GPS location of the user is collected repeatedly every 6 minutes, making *1 item of data*. This data item should be stored to the list L.
* In the second thread (T2), the information about the % battery usage is collected repeatedly every 9 minutes, making *1 item of data*. This data item should also be stored to the list L.
* In the third thread (T3) every time when the number of the data items in the list L exceeds 5, T3 sends all these data to a HTTP server.

• When the user presses button STOP, all three threads should be destroyed (i.e. should stop running).

* 1. Xác định UseCase

Trong mô tả ở trên, ta thấy người dùng mong muốn các hành động như: “*cập nhật GPS location sau mỗi 6 phút*”, “*cập nhật battery usage sau mỗi 9 phút*” và “*gửi dữ liệu về Location and Battery nếu số lượng item*”. Do đó ta các định các UseCase: *UpdateLocation, UpdateBattery, SendData.*

Đồng thời ta cũng muốn người dùng biết giá trị hiện tại của Location và Battery. Với feature này ta sẽ có hai cách: hoặc là ta sẽ đưa nó vào làm một phần của UseCase *UpdateLocation/UpdateBattery,* hoặc là ta sẽ thêm hai UseCase: *GetLatestLocation* và *GetLatestBattery.* Ở đây tôi chọn thực thi theo cách số hai: tạo thêm hai UseCase mới.



* 1. Xác định các Entity và viết logic

Các danh từ ở bài toán này: button, program, thread, data, information, location, battery usage, user, minute, item, list L, number of data items.

Từ các danh từ ta thấy chỉ có “location” và “battery usage” là thông tin sẽ được hiển thị và lưu trữ. Đồng thời nó cũng đại diện cho bài toán.

Với Entity Location, nó có 2 property là: lat: Double và long: Double, ngoài ra nó còn ta cần thêm property ID cho nó. Entity Battery có 1 pproperty level: Int



* 1. Thực thi các UseCase
     1. Update location

1. Senario 1 (success)
2. User ấn start
3. Tạo timer
4. Tạo ra thread T1
5. Get location hiện tại từ harware sau mỗi 6 phút
6. Lưu current location
7. Lưu location vào chuỗi L



1. Senario 2 (cannot get hardware location )
2. User ấn start
3. Tạo ra thread T1
4. Get location hiện tại từ harward sau mỗi 6 phút
5. Hiển thị lỗi cho user
6. Senario 3 (stop)
7. User ấn stop
8. Hủy thread T1
9. Hủy timer
   * 1. Show latest location
10. Senario 1 (success)
11. Latest location được update
12. Load lastest Location
13. Hiển thị nó trên UI
    * 1. Update battery

Tương tự update location

* + 1. Show latest battery
    2. Send Data

1. Senario 1 (success)
2. Khởi tạo thread T3
3. Dữ liệu trong list L được update
4. Lấy số lượng item của list L
5. Kiểm tra số lượng item, nếu số lượng item vượt quá 5
6. send list L
7. Xóa list L
8. Senario 2 (not send data success)
9. Khởi tạo thread T3
10. Dữ liệu trong list L được update
11. Lock list L
12. Lấy số lượng item list L
13. Kiểm tra số lượng item, nếu số lượng vượt quá 5
14. Send list L fail
15. Hiển thị lỗi đến người dùng
16. Tùy vào yêu câu bài toán (Xóa list L hoặc giữ nguyên)



* 1. Thực thi Presentation
  2. Thực thi Data Layer

