ĐẠI HỌC KHOA HỌC TỰ NHIÊN TPHCM

KHOA CÔNG NGHỆ THÔNG TIN



**KIẾN TRÚC PHẦN MỀM**

Đề tài: Đồ án Kiến Trúc Phần Mềm

PHẦN MỀM HỌC TIẾNG ANH QUA VIDEO CHAT

C:\Users\tdqua_000\Dropbox\SS-Slides\DeCuong-CDIO\Template CDIO v4.2\Templates\Hinh anh\LogoTruong.png

Bộ môn Công nghệ phần mềm

Khoa Công nghệ thông tin

Đại học Khoa học tự nhiên TP HCM

MỤC LỤC

[1 Thông tin cá nhân 3](#_Toc503750283)

[2 Thang điểm tự đánh giá 3](#_Toc503750284)

[3 Thông tin dự án 3](#_Toc503750285)

[4 Công nghệ sử dụng 4](#_Toc503750286)

[**a.** **Client side** 4](#_Toc503750287)

[**b.** **Server side** 4](#_Toc503750288)

[5 Kỹ thuật và kiến trúc được dùng trong phần mềm 4](#_Toc503750289)

[**a.** **Software architecture pattern:** 4](#_Toc503750290)

[**b.** **Programming paradigms** 11](#_Toc503750291)

[**c.** **Design Pattern** 13](#_Toc503750292)

[**a.** **Creational pattern:** 13](#_Toc503750293)

[**ii.** **Builder** 17](#_Toc503750294)

[**iii.** **Factory method:** 20](#_Toc503750295)

[**iv.** **Singleton** 21](#_Toc503750296)

[**vii.** **Abstract class** 22](#_Toc503750297)

[**b.** **Structural pattern** 22](#_Toc503750298)

[**i.** **Adapter:** 22](#_Toc503750299)

[**ii.** **ViewHolder** 23](#_Toc503750300)

[**iii.** **Façade** 24](#_Toc503750301)

[**iv.** **Repository** 26](#_Toc503750302)

[**c.** **Behavioral pattern** 29](#_Toc503750303)

[**i.** **Observer** 29](#_Toc503750304)

[**ii.** **Strategy** 31](#_Toc503750305)

# Thông tin cá nhân

|  |  |  |  |
| --- | --- | --- | --- |
| MSSV | Họ Tên | Email | SDT |
| 1412197 | Đoàn Thị Phương Huyền | dtphuyen2506@gmail.com | 0969938215 |
| 1412363 | Trần Thị Nhã | [tranthinha160296@gmail.com](mailto:tranthinha160296@gmail.com) |  |
| 1412477 | Đoàn Hiếu Tâm | [nhoxbypass@gmail.com](mailto:nhoxbypass@gmail.com) | 01684934109 |

# Thang điểm tự đánh giá

Đánh giá dựa trên tiêu chí tỉ lệ số kĩ thuật áp dụng thành công/số kĩ thuật có thể áp dụng được cho Android. (vì một số kĩ thuật không thể hoặc không phù hợp với Android).

* Software architecture pattern (MVVM): 10.
* Programming paradigm: 10.
* Design pattern: Creational (9.5), Structural (9.75), Behavior (9.75).
* Middleware (8)

**Tổng kết: 9.75**.

# Thông tin dự án

**English Now** là một ứng dụng Android nhằm giúp đỡ mọi người học tiếng Anh thông qua các cuộc đối thoại bằng video chat, viết essay và chat text. Qua đó các kĩ năng speaking, writing và listening sẽ được cải thiện.

Link github: <https://github.com/HCMUS-AssignmentWarehouse/EnglishNow-Android-MVVM/>

Đồ án được xây dựng phục vụ riêng cho môn Kiến trúc phần mềm.

Dựa trên phần mềm English Now phiên bản iOS mà nhóm đã xây dựng ở học kì trước (trong môn phát triển ứng dụng mobile). Link github của phiên bản iOS kèm demo để hiểu rõ hơn phiên bản Android (Android chưa kịp làm demo): <https://github.com/HCMUS-AssignmentWarehouse/EnglishNow/> . Link video walkthrough: <https://www.youtube.com/watch?v=Sw4Gj1eF8is/>

# Công nghệ sử dụng

Môi trường phát triển: **Android Studio 3.0** và **Ubuntu 16.04 LTS**.

1. **Client side**

English Now sử dụng:

* **Android** framework với minSdk 21 và buildtools version 27.0.1 để xây dựng.
* **OpenTok client sdk** để xây dựng client side cho webRTC
* **Firebase** để xây dựng hệ thống authentication và realtime database.
* **Dagger 2** để ứng dụng dependency injection.
* **RxJava, RxAndroid** để ứng dụng observer pattern và reactive programming.
* **Android Acrchitecture Component** của Google để áp dụng MVVM và LiveData.
* **Retrofit 2** cho HTTP client.
* **EventBus** cho việc giao tiếp giữa các thành phần trong hệ thống bằng message.
* Và một số thư viện UI khác.

Để biết rõ hơn, tham khảo readme của projet tại [đây](https://github.com/HCMUS-AssignmentWarehouse/EnglishNow-Android-MVVM/blob/master/README.md).

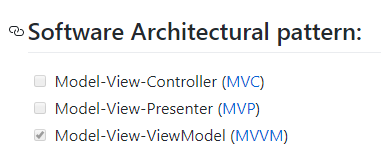
1. **Server side**

Sử dụng **nodeJS**, **expressJS** và **OpenTok server sdk** để tạo một server side đơn giản và deploy lên herokuapp (<https://englishnow.herokuapp.com/> ).

# Kỹ thuật và kiến trúc được dùng trong phần mềm

Tham khảo danh sách thu gọn các kỹ thuật/kiến trúc đã áp dụng hoặc chưa áp dụng được trong list requirement [sau](https://github.com/HCMUS-AssignmentWarehouse/EnglishNow-Android-MVVM/blob/master/requirement.md).

1. **Software architecture pattern:**



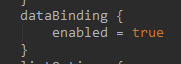
Vấn đề: Project mang tính chất realtime, cần databinding và cập nhật view theo trạng thái của data liên tục

Giải pháp: Nhóm chọn sử dụng mô hình **Model-View-ViewModel (MVVM)** trong đó:

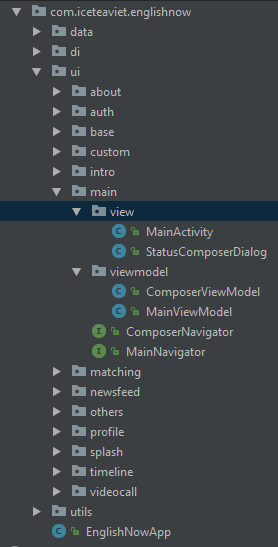
* **View**: đại diện cho giao diện, dùng để hiển thị trạng thái lên màn hình và thông báo cho ViewModel khi có tương tác từ user.
* **ViewModel**: xử lý các dòng dữ liệu, xử lý logic và và chuyển lại cho View.
* **Model**: hay còn gọi là DataModel, là các abstract data source. Được ViewModel sử dụng trong việc lấy và lưu data.

MVVM tách rời trạng thái của View ra khỏi các hành vi của nó, chú trọng vào **event driven** programming. View sẽ có thể có reference tới ViewModel, nhưng ViewModel sẽ **không quan tâm** cũng như **không biết** gì về trạng thái của View, chỉ nhận yêu cầu, lấy data, xử lý và trả kết quả.

MVVM ứng dụng trong project theo hướng **two-way data binding** giữa View và ViewModel với sự hỗ trợ của Android Data Binding bằng cách bật trong **app/build.gradle**



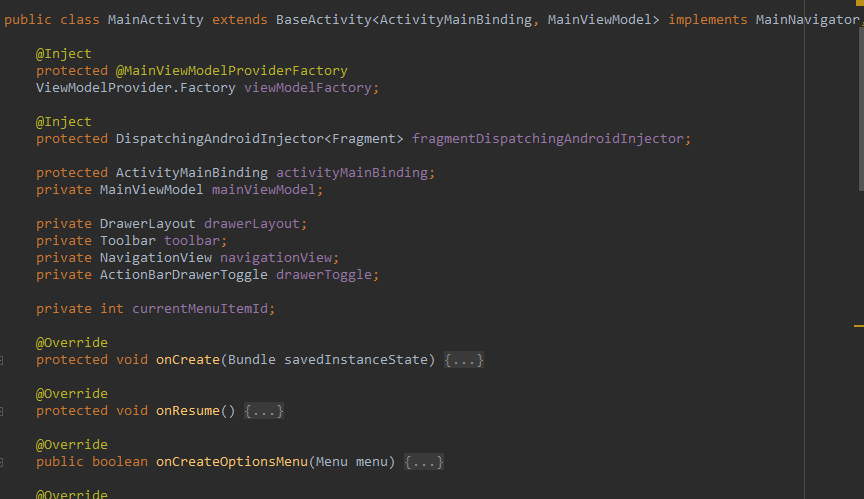
Project có cấu trúc như sau:

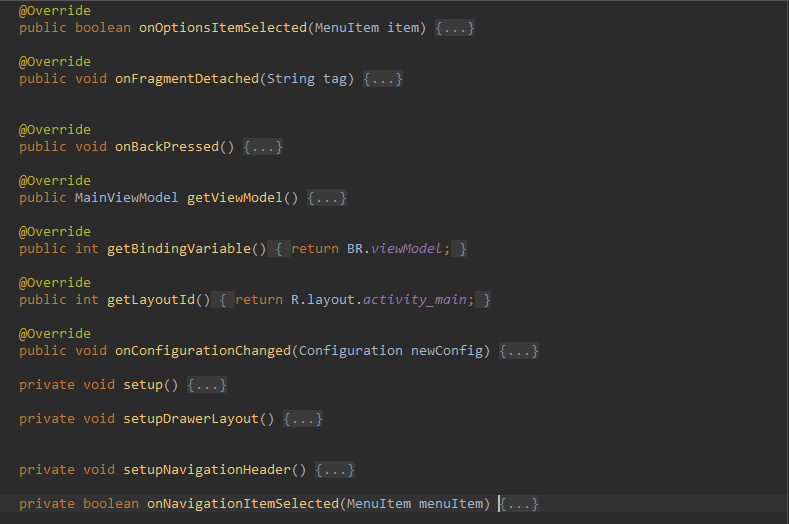


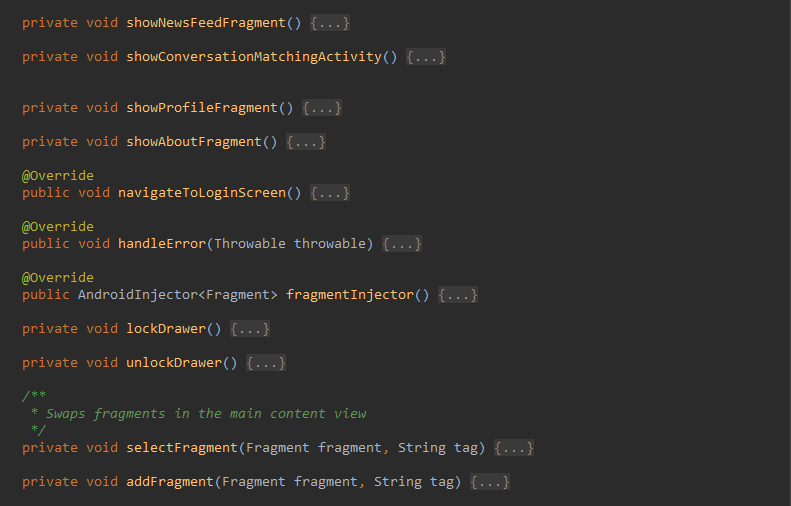
Nhóm phân chia các package của project theo feature mà không theo layer vì khi ta cần sửa một tính năng gì, ta chỉ cần sửa gọn trong package của nó. Và vì project áp dụng abstract class/ interface rất triệt để cho các class của từng layer nên nếu ta cần thêm code cho tất cả các class của một layer nào đó chỉ cần thêm vào base class của layer đó.

Ta sẽ đến với phần **View trong MVVM**: Trong Android, phần View thường là Activity/Fragment. Tất cả các đoạn code trong View layer không được phép chứa logic hay các thao tác với dữ liệu, vì bản chất của MVVM là tách rời logic và các dòng dữ liệu ra khỏi View để tiện xử lý.

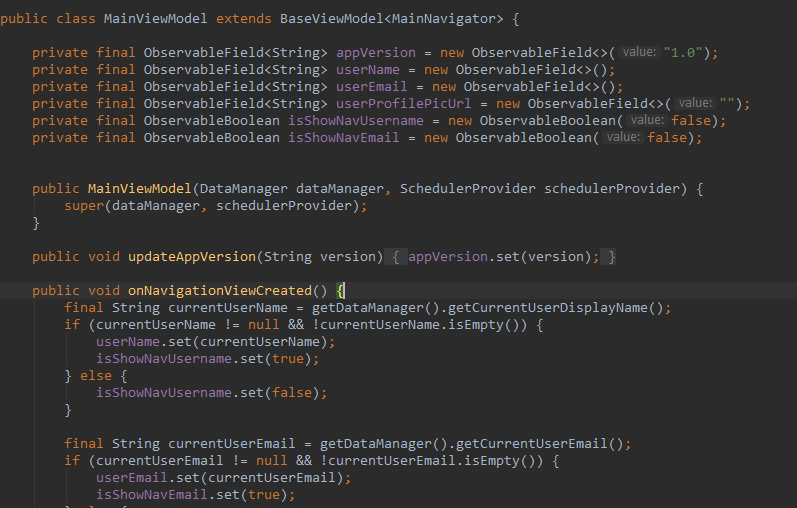
Vì vậy mọi xử lý phải thông qua ViewModel





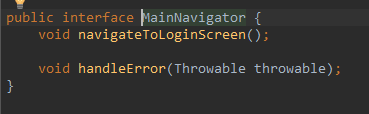


Tiếp theo ta sẽ đến với phần **ViewModel trong MVVM**: ViewModel đảm nhiệm các phần tính toán logic, và xử lý các dòng dữ liệu

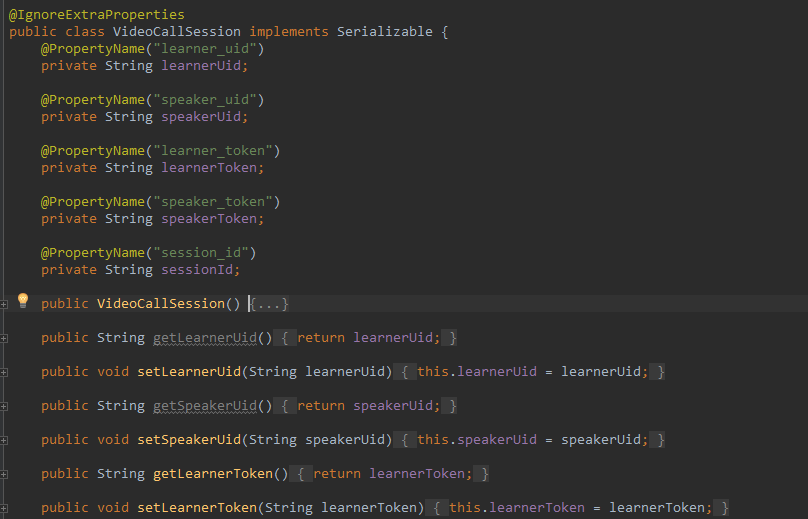




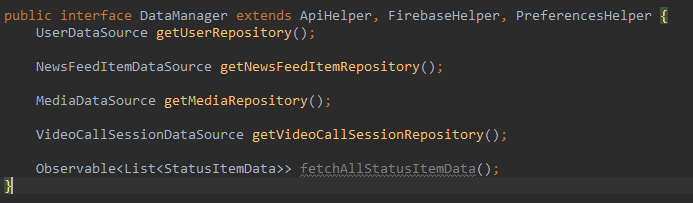
ViewModel giao tiếp với View thông qua một interface **Navigator** được View implement các method:

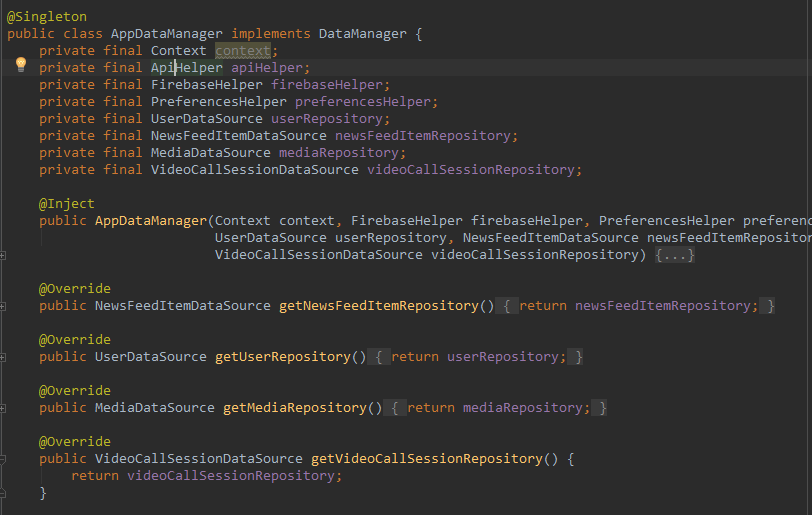


Cuối cùng là **Model trong MVVM**: Model chịu trách nhiệm làm các “vật chứa” để ViewModel thao tác với database (read/write/remove).



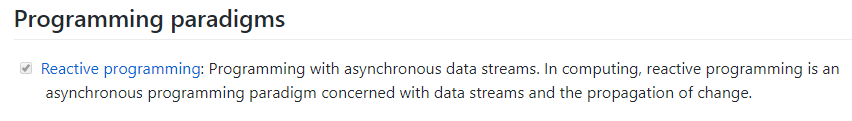
Để sử dụng Model, ViewModel cần một **Façade** tổng hợp các thao tác gọi là **AppDataManager**, một concrete class của **interface DataManager**





Lý do tại sao không tạo class trực tiếp mà sử dụng Interface rồi implement lại là vì cần thiết cho việc thiết kế các testcase và sẽ được giải thích trong phần sau.

1. **Programming paradigms**



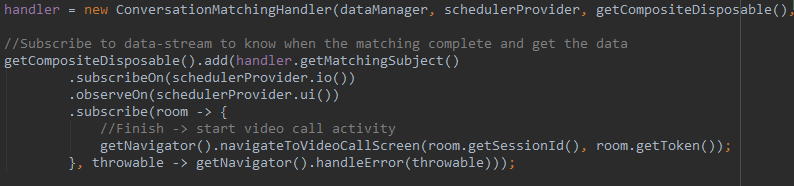
Vấn đề: project là phần mềm tương tác online 90%, **dữ liệu và trạng thái của các thành phần trong app luôn thay đổi liên tục**.

Giải pháp: nhóm quyết đỉnh áp dụng **Reactive Programming** để tương tác với **các dòng dữ liệu bất đồng bộ** giữa app và server.

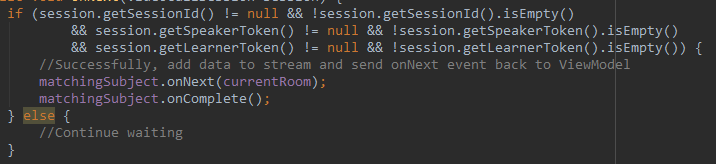
Ví dụ: Trong khi thực hiện việc matching giữa hai user để tạo room cho video chat. ViewModel cần biết khi nào việc matching hoàn thành để lấy thông tin ra và chuyển sang màn hình khác, ta sử dụng PublishSubscriber để tạo ra datastream liên kết giữa ViewModel và các data handler.



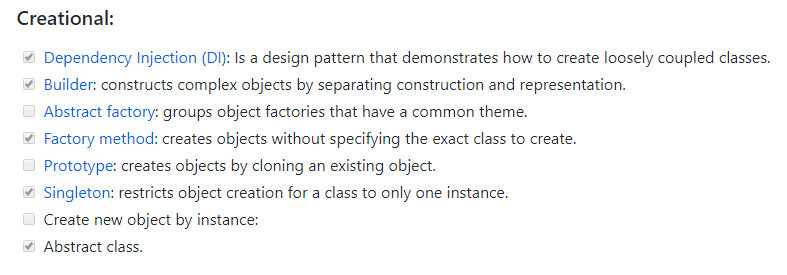
ViewModel subscribe vào datastream này và chờ sợ thay đổi của trạng thái và dữ liệu.



Khi dữ liệu thay đổi, ta thêm data vào data-stream và trigger event cho các subscriber biết mà cập nhật dữ liệu.

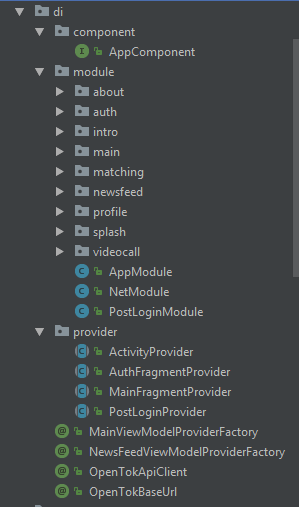


1. **Design Pattern**
   1. **Creational pattern:**

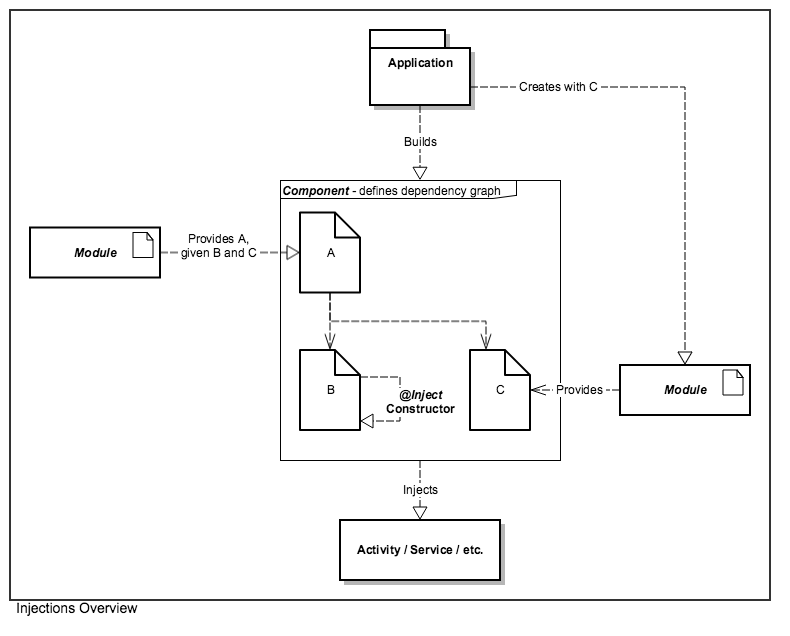


* + 1. **Dependency Injection (DI)**: Vì project có rất nhiều các thành phần được dùng chung và có thể tái sử dụng nên nhóm quyết định sử dụng Dagger 2 để apply dependency injection để tạo các object đó trước ở ngoài và inject vào các chỗ cần thiết.

Project package structure được sắp xếp như sau:

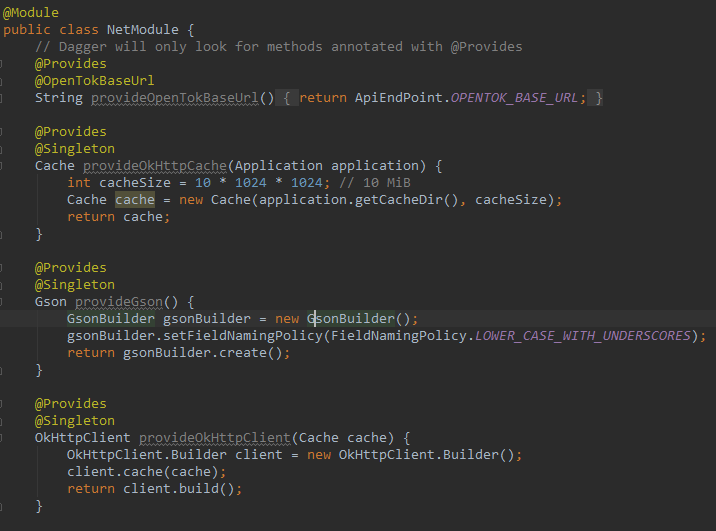


Mô hình thường thấy nhất khi áp dụng DI trong Android là:



Ở đây ta thấy có ba thành phần chính: **Module**, **Component** và **những nơi cần sử dụng dependencies**.

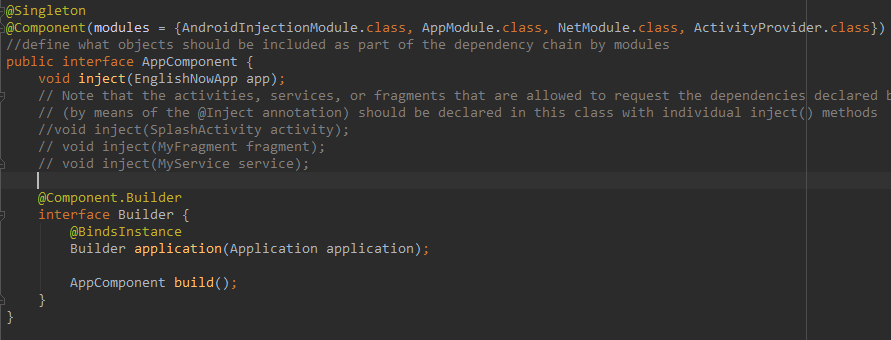
Ta đi vào tìm hiểu **@Module**: Là nơi cung cấp các dependency cần thiết cho Component. Ví dụ **NetModule.java, AppModule.java**:



Các annotation **@Provides** đánh dấu đây là các dependency sẽ được cung cấp cho component, các annotation **@Singleton** đánh dấu các object này chỉ cần tạo một lần và sử dụng xuyên suốt trong project.

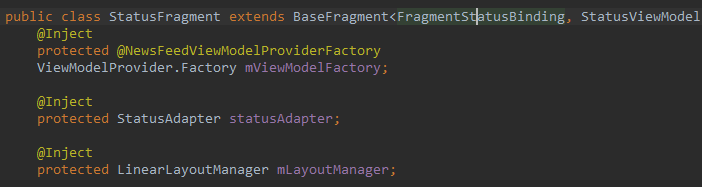
Có rất nhiều @Module được tạo ra trong project.

Tiếp theo ta đến với **@Component**: Component là một graph. Chúng ta xây dựng Component, Component sẽ cung cấp các **injected instance** bằng cách sử dụng các @Module. Ví dụ **AppComponent.java**:

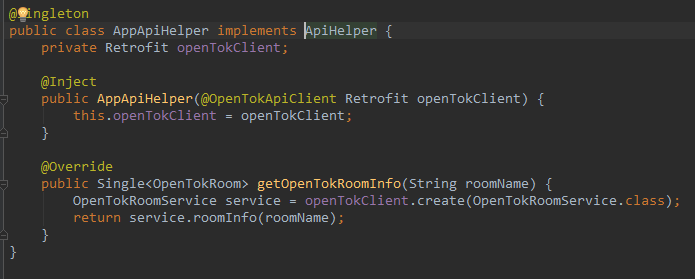


Sau khi setup xong, mỗi khi cần dependency nào chúng ta có thể inject vào bằng 2 cách.

Field inject:



Và constructor inject



* + 1. **Builder**

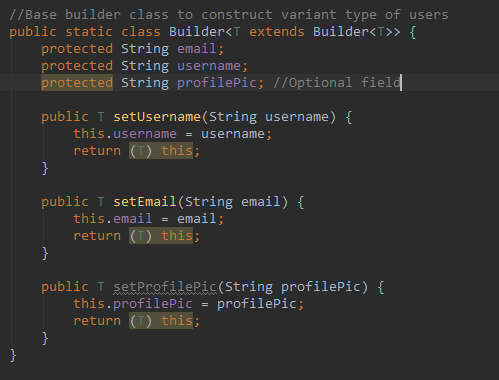
Vấn đề: Các model của project có rất nhiều biến thể và có các **optional field**, cần tối giản quá trình tạo lập object.

Giải pháp: Áp dụng Builder pattern để tạo các đổi tượng phức tạp chỉ bằng các câu lệnh đơn giản.

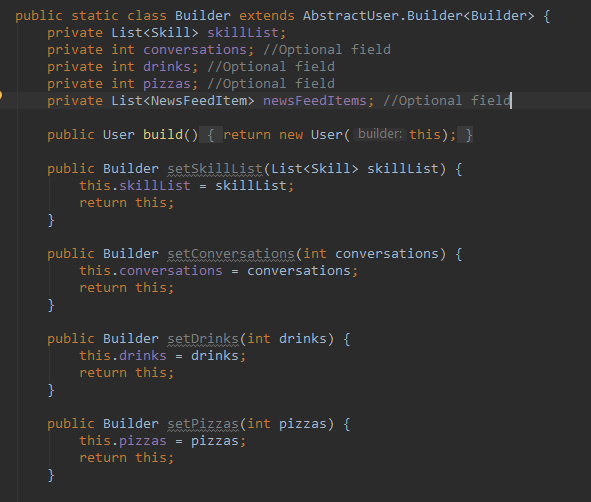
Ví dụ:

Khi khởi tạo User lúc đăng ký, ta chưa có nhiều thông tin về User đó, vậy khi đó ta chỉ cần tạo User với các thông tin cần thiết để push vào database, sau này khi có đầy đủ thì ta có thể update lại thông tin.

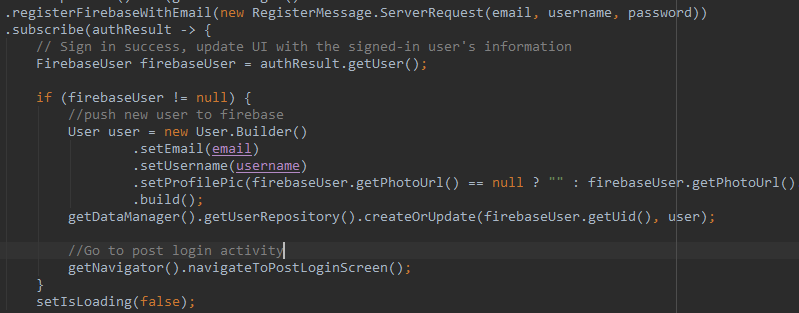
Lưu ý: Vì project có nhiều loại user nền cần dùng kế thừa/đa hình. Vì vậy Builder cũng được ứng dụng cho base class **AbstractUser.java**, sử dũng **Generic<T>** trong java.



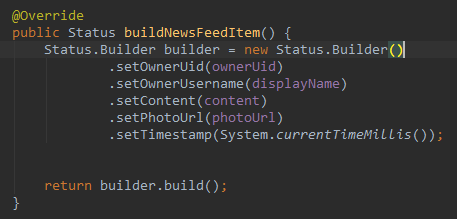
Trong class con **User.java**



Sử dụng:



Trong bài Builder pattern còn được sử dụng với các **NewsFeedItem**:

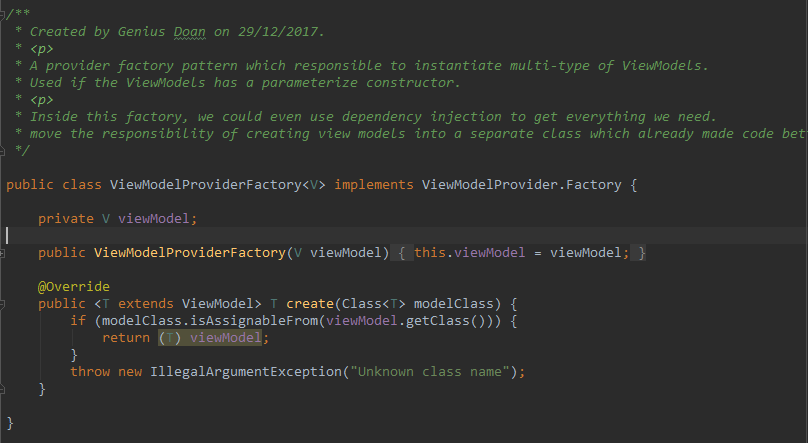


* + 1. **Factory method:**

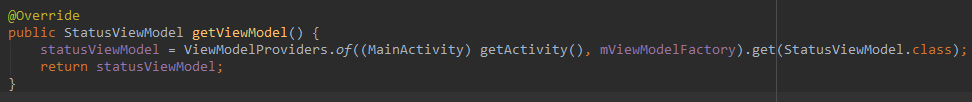
Vấn đề: Khi áp dụng MVVM ở màn hình NewsFeed sẽ có 2 tab Status và Video tương ứng với 2 list dữ liệu khác nhau (một list các status dạng text kèm hình, một list các video). Điều đó đồng nghĩa với việc cần có 2 ViewModel khác nhau nhưng lại có những mối quan hệ với nhau.

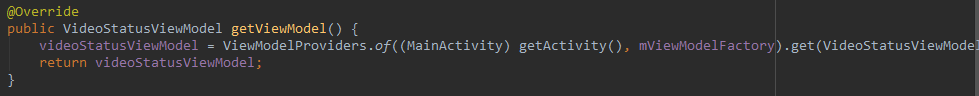
Giải pháp: Áp dụng Factory method để tạo ra các ViewModel nào cần điều kiện khi tạo.

Ta tạo class **ViewModelProviderFactory.java**



Ta xét các điều kiện sau đó gọi hàm **ViewModelProviderFactory#get()** để tạo các object



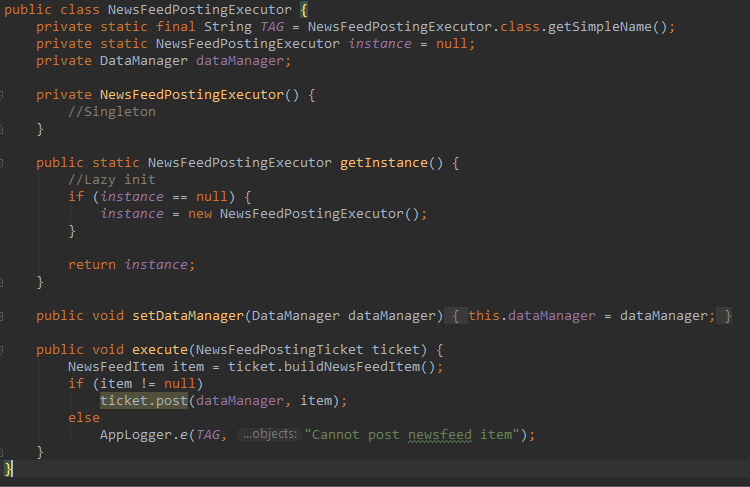


* + 1. **Singleton**

Vấn đề: Có những object chỉ cần **tạo một lần** và sử dụng nhiều nơi trong project.

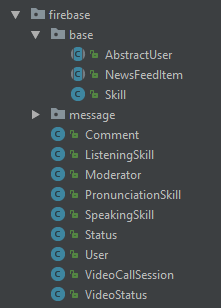
Giải pháp: Nếu object đó tạo bằng DI ta sử dụng annotation @Singleton. Nếu không ta tạo một biến **static** để lưu instance, dùng **private/protected constructor** để giới hạn việc tạo lập instance của class đó.

Ví dụ:

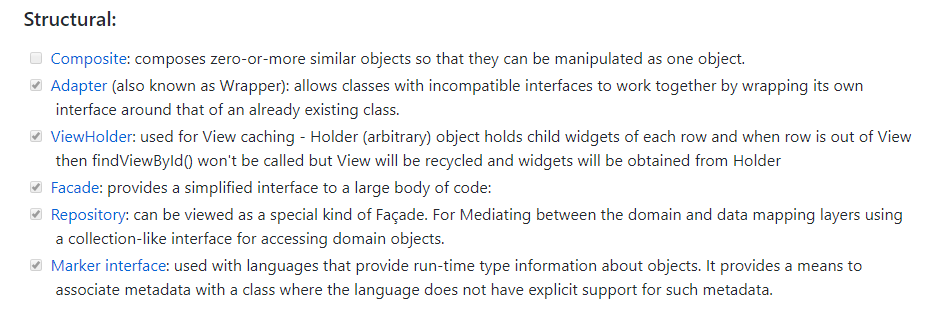


1. **Abstract class**

Project áp dụng abstract class cho **tất cả** các model chính. Khi cần mở rộng để thêm các object trong cùng một nhóm chỉ cần extends base abstract class và implement thêm các thông tin cần thiết mà không boiterplate code.



* 1. **Structural pattern**

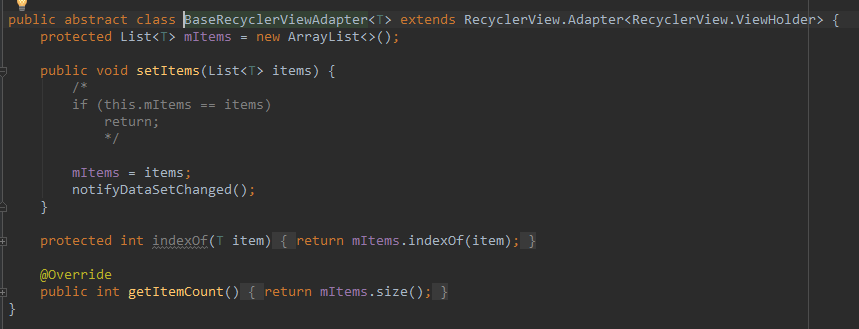


* + 1. **Adapter:**

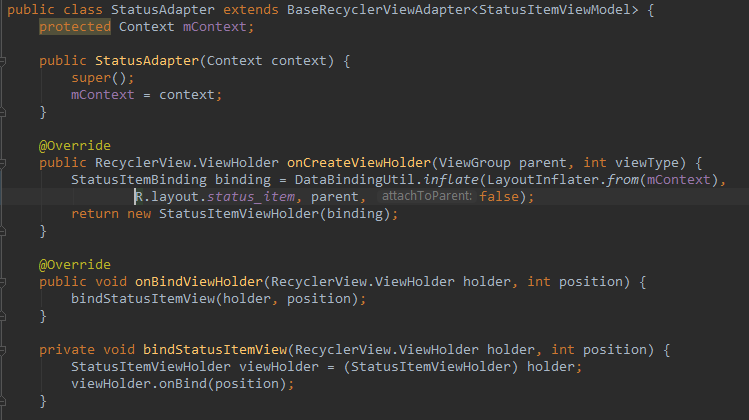
Vấn đề: Trong việc hiển thị và cập nhật danh sách các newsfeed item, ta cần giao tiếp giữa hai thành phần khác nhau hoàn toàn là **list dữ liệu trong memory** và các **layouts/views**.

Giải pháp: Sử dụng Adapter pattern của Android.

Tạo base class **BaseRecyclerViewAdapter<T>** để làm khung, đồng thời giữ một List dữ liệu **mItems**



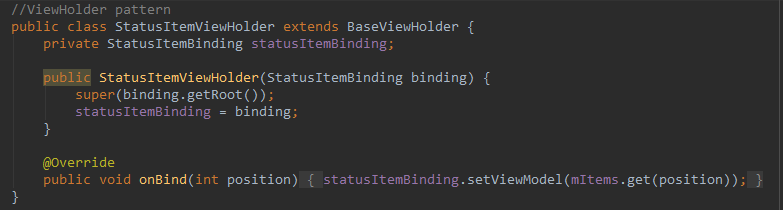
Và class con **StatusAdapter.java** sử dụng hàm **bindStatusItemView()** để convert từ dữ liệu sang layout với thông tin và trạng thái tương ứng.



* + 1. **ViewHolder**

Vấn đề: Trong Android, việc tìm view từ các id rất **tốn thời gian và tài nguyên của máy**, với một list dữ liệu lớn mỗi lần chúng ta muốn cập nhật thông tin đều phải tìm view bằng cách gọi **findViewById()** thì cực kì mất thời gian và **ảnh hưởng đến hiệu năng**.

Giải pháp: Sử dụng ViewHolder pattern, chúng ta sẽ **cache/hold** các instance của các View cần thiết lại và lấy ra **tái sử dụng** khi cần cập nhật dữ liệu của View.

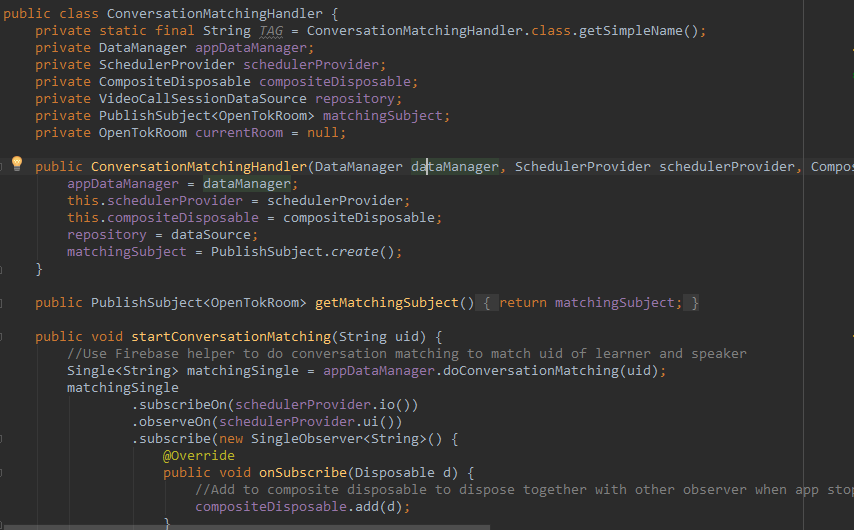


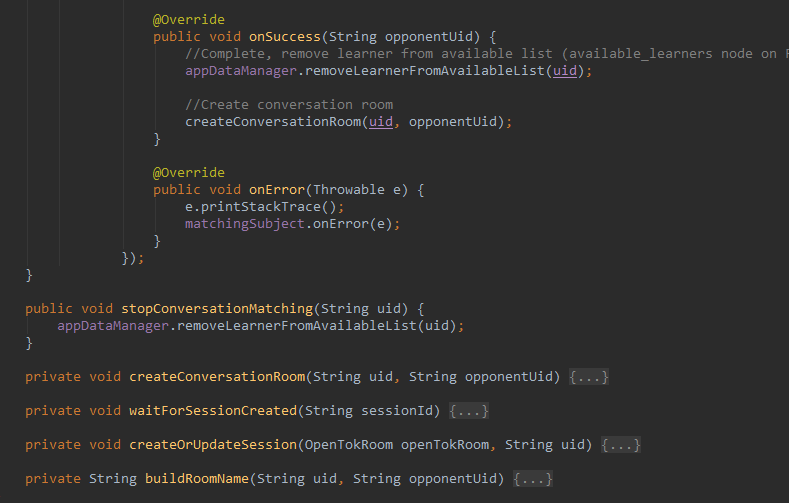
* + 1. **Façade**

Vấn đề: Trong project có những thao tác được thực hiện **nhiều lần** theo một **trình tự nhất định**, sử dụng **nhiều nơi** trong ứng dụng. Lập trình viên khó khăn khi phải nhớ mình cần làm cái gì, mặc khác nếu copy/paste những đoạn code đó thì sẽ tạo ra **boiterplate code**.

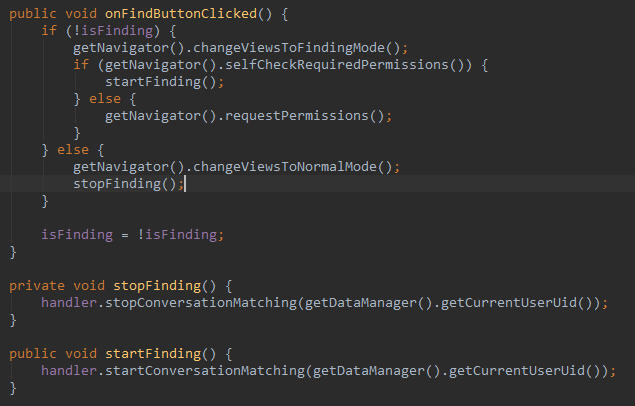
Giải pháp: Sử dụng Façade pattern, tạo một **Handler/Executor/Manager** đại diện, cung cấp cho nó những thành phần cần thiết và chỉ cần thực thi bất cứ chỗ nào cần.

Ví dụ: class **ConversationMatchingHandler.java**





Sử dụng trong ViewModel một cách đơn giản, chỉ cần gọi 1 trong 2 hàm cần sử dụng, mọi thông tin về việc nó vẫn hành thế nào ta không cần quan tâm.



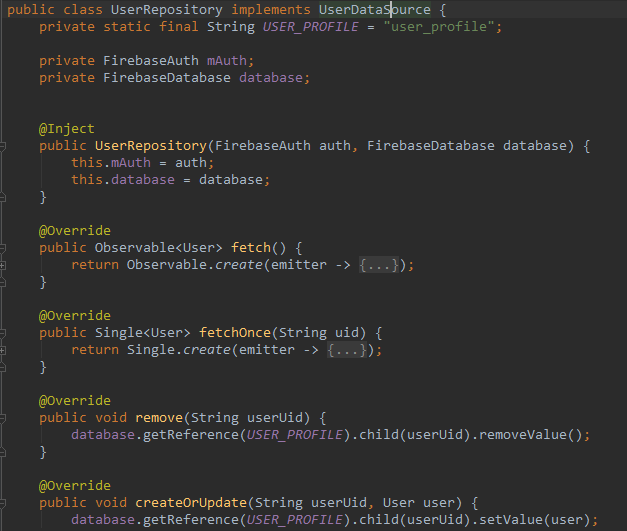
Ngoài ra trong project còn áp dụng Façade pattern cho NewsFeedPostingExecutor, và các data manager khác.

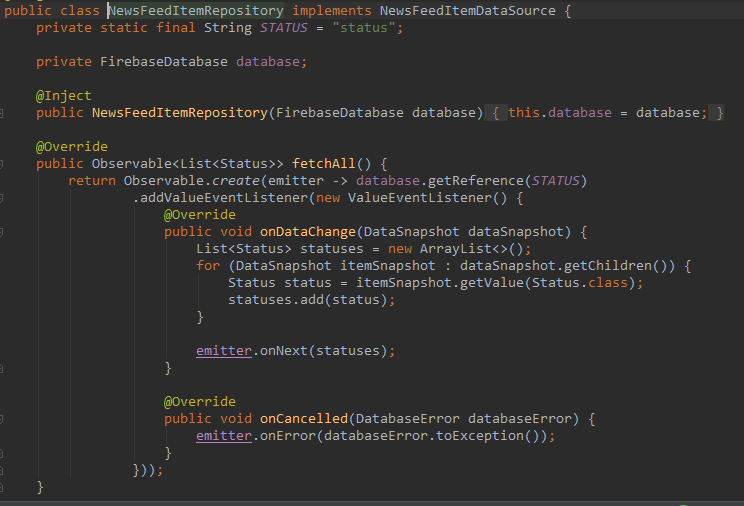
* + 1. **Repository**

Vấn đề: Vì project thao tác với dữ liệu rất nhiều, nên cần một cách để thống nhất và dễ dàng thao tác với dữ liệu.

Giải pháp: Sử dụng Repository pattern để ViewModel có thể dễ dàng tương tác với data layer.

Ví dụ:





Mặc khác khi áp dụng Repository pattern nói riêng và các data handler/helper nói chúng, nhóm có tạo các **interface DataSource** làm **phần khung** trước sau đó mới implement lại (một biến thể của Template pattern)

Việc này giúp ta có thể dễ dàng tạo thêm một class tương tự (**UserRepositoryMock**/**UserRepositoryFake**) để thực hiện **Mock** dữ liệu cho các **unit test/intergration test**.

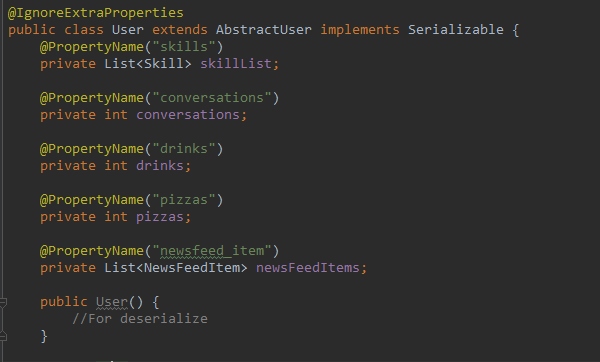
Trong project còn các Repository khác như MediaRepository, VideoCallSessionRepository

* + 1. **Marker interface**

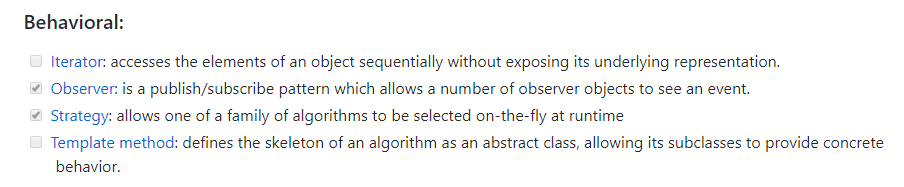
Vấn đề: Trong project Android để chuyển dữ liệu giữa các Activity chúng ta convert nó thành byte-stream để gửi đi và tạo lại object từ byte-stream nhận được. Vì vậy cần một thứ gì đó **đánh dấu hiệu** là một object nào đó có thể gửi như vậy.

Giải pháp: Sử dụng marker interface Serializable của Android

Ví dụ:



* 1. **Behavioral pattern**



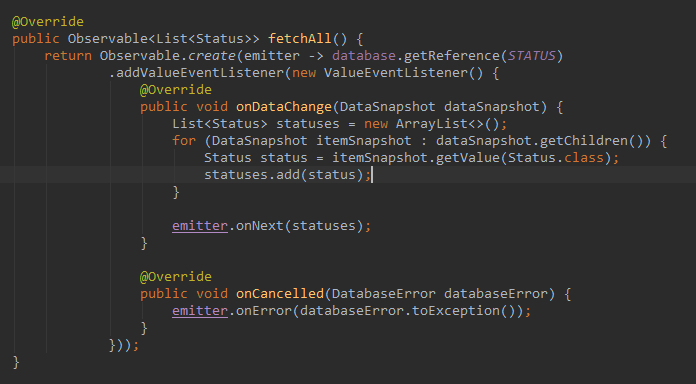
* + 1. **Observer**

Vấn đề: Vì là một app phụ thuộc vào realtime database, nên sẽ có rất nhiều đối tượng trong app như user info, newsfeed items, chat message, incoming request,… phụ thuộc vào dữ liệu và các đối tượng khác.

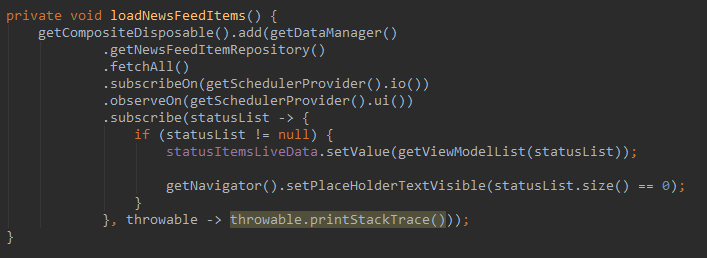
Giải pháp: Sử dụng **Observable/Single** của RxJava hoặc **LiveData** của Android. Khi realtime database cập nhật trạng thái mới, thì các thành phần trong app phải cập nhật thay đổi theo.

Ví dụ:

Để fetch và observe dữ liệu của các status ta sử dụng hàm **fetchAll().** Trong hàm này ta tạo ra một object kiểu Observable (có thể observe được). Mỗi khi realtime database cập nhật, ta dùng **emmiter** gửi một event kèm theo dữ liệu bằng **emitter.onNext()**.



Trong ViewModel hoặc bất cứ chỗ nào cần nhận event từ Observable trên, ta tạo một observer và subscribe vào. Mỗi khi có **event** được **emit**, các observer sẽ được **notify**, sau đó lấy dữ liệu trạng thái mới ra và cập nhật.

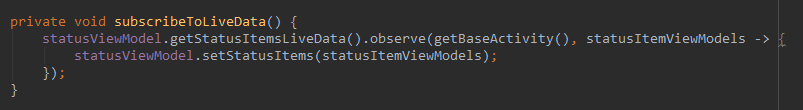


Hoặc sử dụng LiveData của Android.

Khai báo trong class **StatusFragment.java**



Tạo các Observer để observe vào LiveData.



Mỗi khi giá trị của LiveData thay đổi, các observer sẽ được notify để cập nhật trạng thái mới.

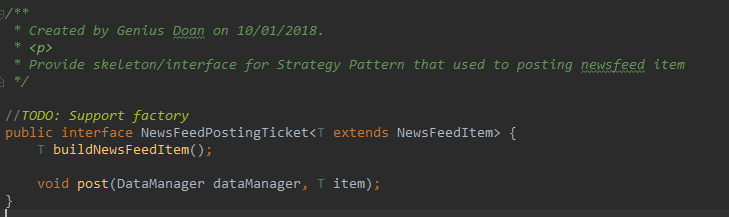
* + 1. **Strategy**

Vấn đề: Trong project, user có thể chọn post các status thông thường (text kèm hình ảnh) hoặc video status. Việc lựa chọn cách post **diễn ra run-time**.

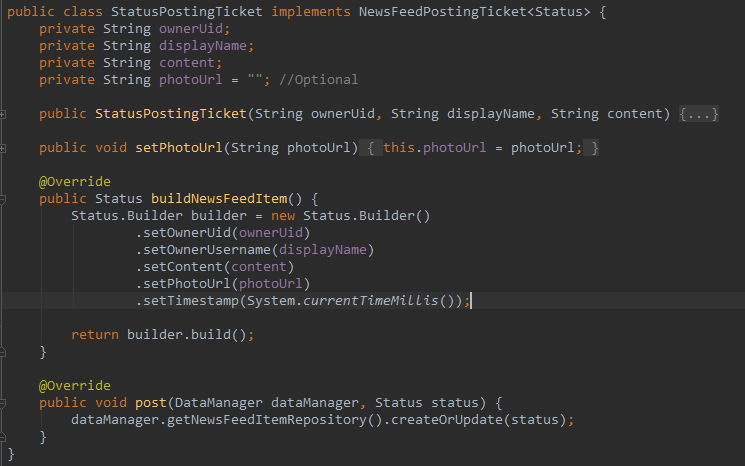
Giải pháp: Sử dụng Strategy kết hợp với Façade để post các loại Status.

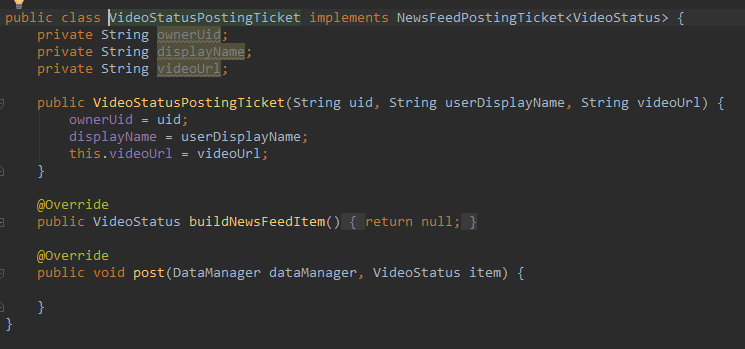
Ví dụ:

Đầu tiên ta tạo base class cho các strategy:



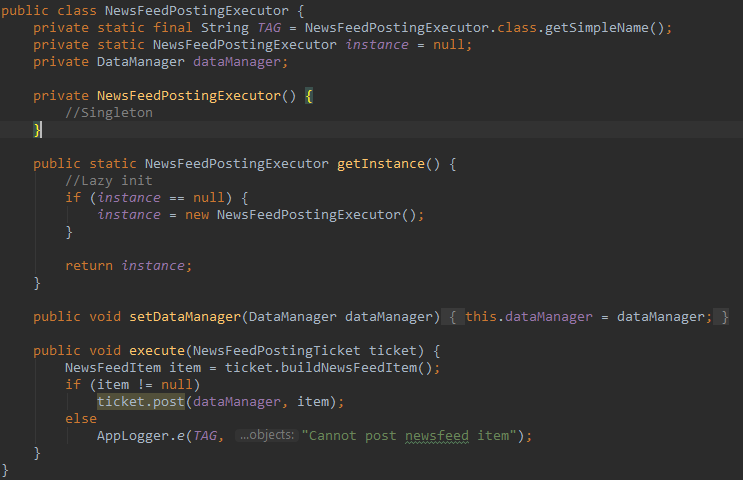
Implement các Strategy



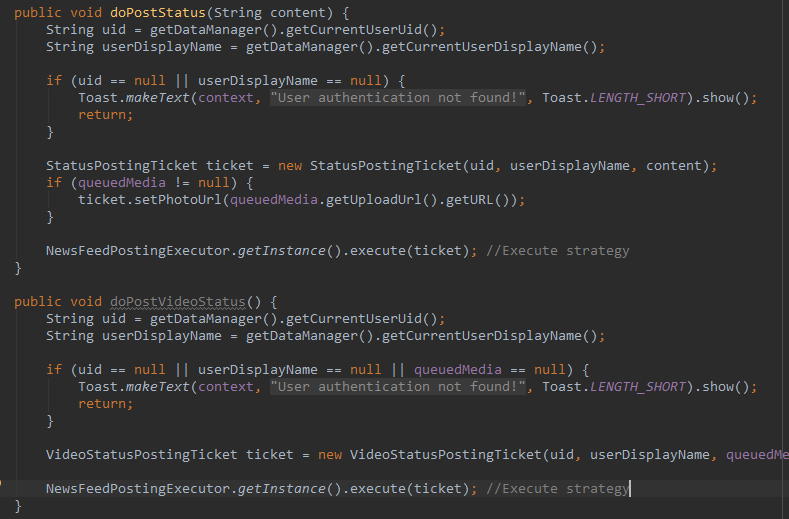


Chúng ta có thể tạo thêm Strategy nếu sau này muốn mở rộng project ra để post audio, file, văn bản,..

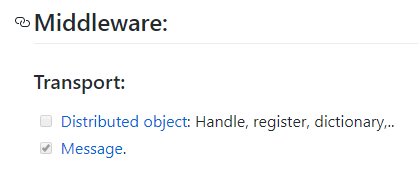
Sử dụng Façade làm context để thực thi các Strategy, tạo class **NewsFeedPostingExecutor.java**:



Sử dụng trong các ViewModels:



1. **Middleware**

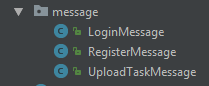


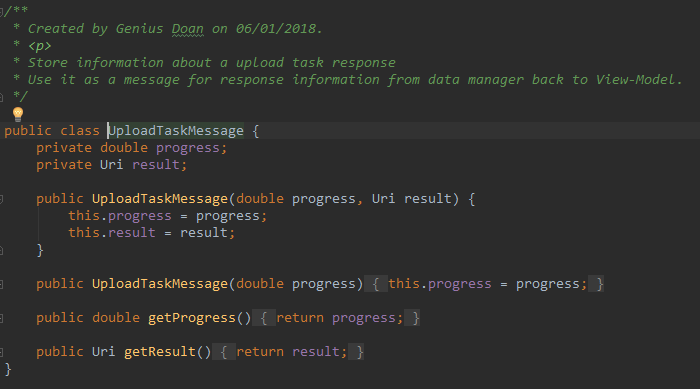
Vấn đề: Project ngoài việc thao tác dữ liệu với database, còn phải luân chuyển **transport** dữ liệu **giữa các thành phần của hệ thống** (Activity/Fragment/Service,..)

Giải pháp: Sử dụng Transport Message.

Ví dụ:

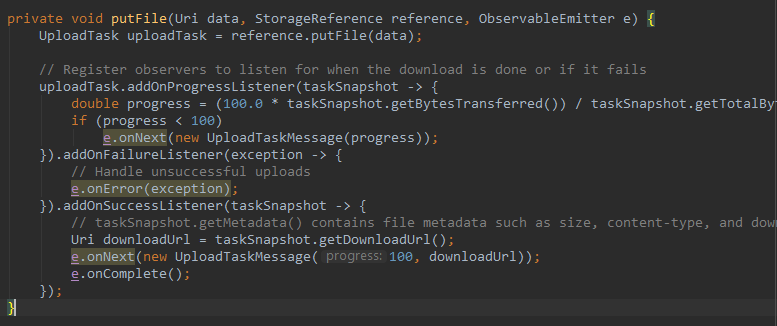
Tạo các class model Message để chứa dữ liệu cần truyền





Sau đó nạp thông tin vào và truyền dữ liệu qua lại giữa ViewModel/Data manager

Bên gửi:



Bên nhận

