**Модель**

Под Моделью, обычно понимается часть содержащая в себе функциональную бизнес-логику приложения. Модель должна быть полностью независима от остальных частей продукта. Модельный слой ничего не должен знать об элементах дизайна, и каким образом он будет отображаться. Достигается результат, позволяющий менять представление данных, то как они отображаются, не трогая саму Модель.

**Представление (View)**

В обязанности Представления входит отображение данных полученных от Модели. Однако, представление не может напрямую влиять на модель. Можно говорить, что представление обладает доступом «только на чтение» к данным.

**Признаки View-модели:**

* Двухсторонняя коммуникация с представлением;
* View-модель — это абстракция представления. Обычно означает, что свойства представления совпадают со свойствами View-модели / модели
* View-модель не имеет ссылки на интерфейс представления (IView). Изменение состояния View-модели автоматически изменяет представление и наоборот, поскольку используется механизм связывания данных (Bindings)
* *Один экземпляр View-модели связан с одним отображением.*

User

События UI

Получение данных

Model

View

ViewModel

Данные

Обновление модели

Слой Представления

Interactor interface

JAVA Objects\*

Слой бизнес-логики

Repository interface

Data Mining Methods

Слой Данных

Database

Наша цель — **разделение задач** таким образом, чтобы бизнес-логика ничего не знала о внешнем мире, так, чтобы можно было тестировать её без никаких зависимостей и внешних элементов.   
Для достижения этого я предлагаю **разбить проект на 3 слоя**, каждый из которых имеет свою цель и может работать независимо от остальных.   
  
Стоит отметить, что каждый слой использует свою модель данных, таким образом можно достигнуть необходимой независимости (вы можете увидеть в коде, для выполнения трансформации данных необходим преобразователь данных (data mapper). Это вынужденная плата за то, чтобы модели внутри приложения не пересекались). Вот схема, как это выглядит:

## **Presentation Layer (Слой представления)**

Здесь логика связывается с Views (Представлениями) и происходят анимации. Это не что иное, как **Model View Presenter**(то есть, [MVP](http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter)), но вы можете использовать любой другой паттерн вроде MVC или MVVM. Я не буду вдаваться в детали, но fragments и activities — это всего лишь views, там нету никакой логики кроме логики UI и рендеринга этого самого отображения. Presenter-ы на этом слое связываются с **interactors (посредниками)**, что предполагает работу в новом потоке (не в UI-потоке), и передачи через коллбэки информации, которая будет отображена во view.

## **Domain Layer (Слой бизнес-логики)**

**Вся логика реализована в этом слое.** Рассматривая проект, вы увидите здесь реализацию interactor-ов (Use Cases — методы использования).   
  
**Этот слой — модуль на чистой джаве без никаких Android-зависимостей.** Все внешние компоненты используют интерфейсы для связи с бизнес-объектами.

## **Data Layer (Слой данных)**

## **Data Layer (Слой данных)**

**Все данные, необходимые для приложения, поставляются из этого слоя через реализацию UserRepository (интерфейс находится в domain layer — слое бизнес-логики), который использует [Repository Pattern](http://martinfowler.com/eaaCatalog/repository.html) со стратегией, которая, через фабрику, выбирает различные источники данных, в зависимости от определенных условий.**  
  
Например, для получения конкретного пользователя по id источником данных выбирается дисковый кэш, если пользователь уже загружен в него, в противном случае облаку отправляется запрос на получение данных для дальнейшего сохранения в тот же кэш.   
  
Идея всего этого заключается в том, что происхождение данных является понятным для клиента, которого не волнует, поступают данные из памяти, кэша или облака, ему важно только то, что данные будут получены и доступны.  
  
Например, для получения конкретного пользователя по id источником данных выбирается дисковый кэш, если пользователь уже загружен в него, в противном случае облаку отправляется запрос на получение данных для дальнейшего сохранения в тот же кэш.   
  
Идея всего этого заключается в том, что происхождение данных является понятным для клиента, которого не волнует, поступают данные из памяти, кэша или облака, ему важно только то, что данные будут получены и доступны.