Архитектура

План

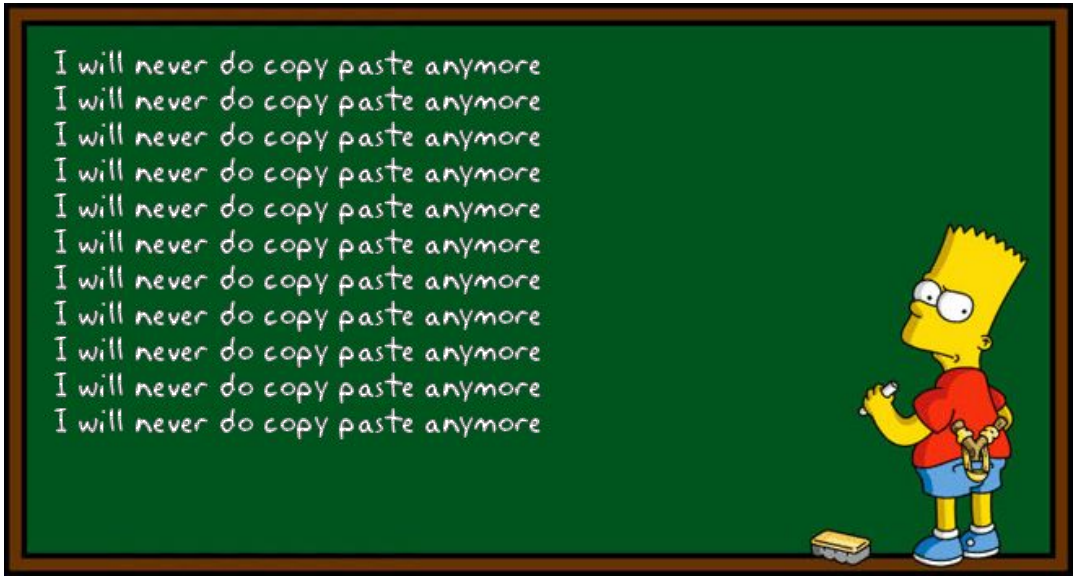
- Базовые принципы проектирования
  - DRY
  - KISS
  - YAGNI
  - SOLID
- Архитектурные паттерны
  - MVC
  - MVP
  - MVVM
  - MVI
- Clean Architecture

DRY

Don’t Repeat Yourself

Не повторяйся!

Этот принцип заключается в том, что нужно избегать повторений одного и того же кода. Когда вы замечаете, что пишите повторяющийся код, стоит задуматься о том, чтобы выделить его в общий метод/класс/модуль и переиспользовать.



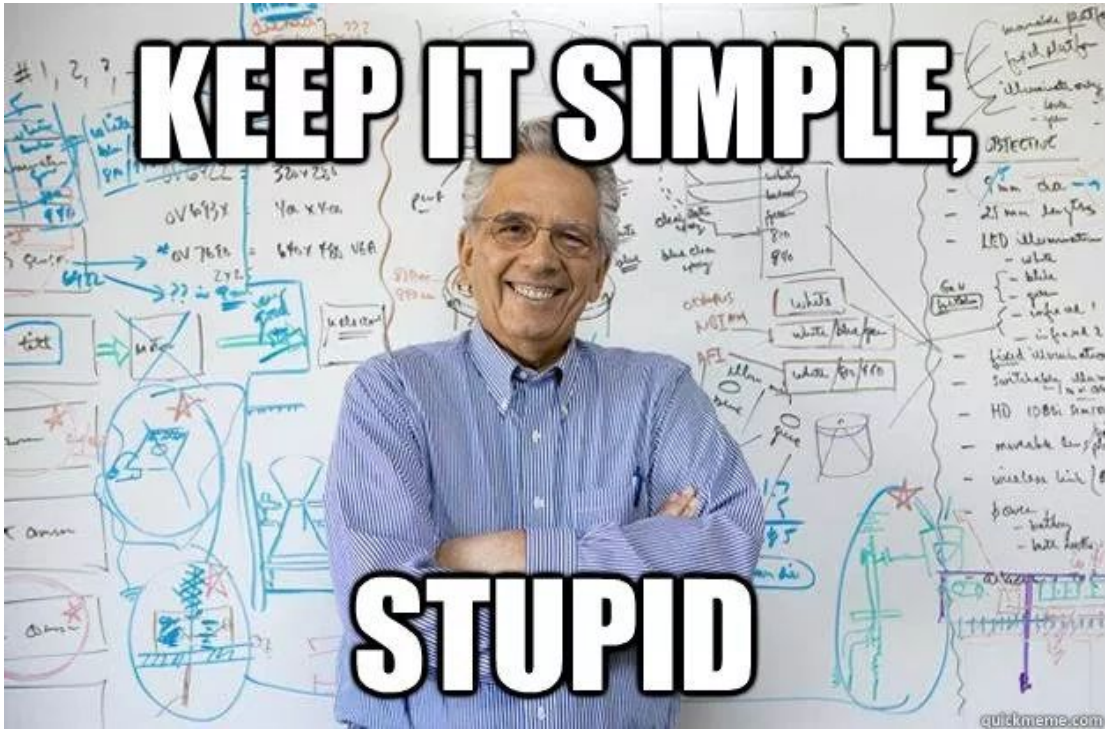
KISS

Keep It Simple, Stupid

Не усложняй!

Смысл этого принципа заключается в том, что стоит писать простой и понятный код.

Хорошо, когда код читается легко без всяких комментариев.

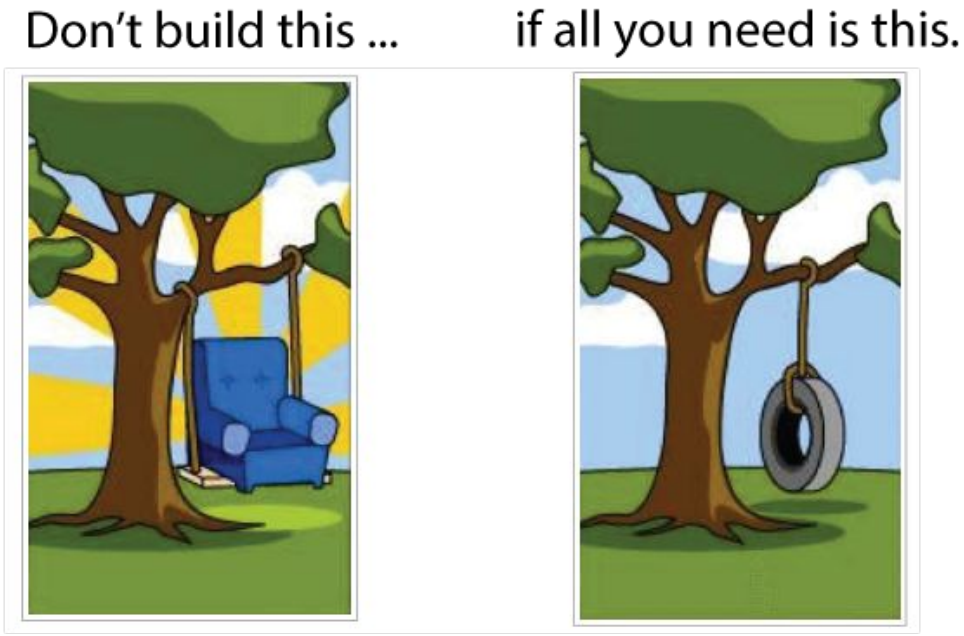


YAGNI

You Ain’t Gonna Need It

Вам это не понадобится!

Принцип проектирования ПО, при котором в качестве основной цели декларируется отказ от избыточной функциональности, — то есть отказ добавления функциональности, в которой нет непосредственной надобности.



SOLID

- **Single responsibility**
- **Open-closed**
- **Liskov substitution**
- **Interface segregation**
- **Dependency inversion**

Single Responsibility Principle

Принцип единственной ответственности

Каждый класс выполняет лишь одну задачу

```
public class NoSRP
{
    public void SaveUser(User userData)
    {
        //Save the user data in database
    }

    public void SendEmail(String smtpUserName, String smtpPassword)
    {
        // Send Email to the user
    }
}
```

```
public class ManageUser
{
    public void SaveUser(User userData)
    {
        //Save the user data in database
    }
}

public class EmailService
{
    public void SendEmail(String smtpUserName, String smtpPassword)
    {
        // Send Email for different purposes
    }
}
```

Open/Closed Principle

Принцип открытости/закрытости

«программные сущности … должны быть открыты для расширения, но закрыты для модификации.»

```
public class NOOCP
{
    public void LogErrors(Int32 type)
    {
        switch (type)
        {
            case 1:
                // Save using Type 1 Logger
                break;
            case 2:
                // Save using Type 2 Logger
                break;
        }
    }
}
```

```
public interface ILoggers
{ void Log(); }

public class Logger1 : ILoggers
{
    public void Log()
    {
        // Use Logger of Type one
    }
}
public class Logger2 : ILoggers
{
    public void Log()
    {
        // Use Logger of Type two
    }
}
public class OCP
{
    ILoggers _logger;
    public OCP(ILoggers logger)
    {
        this._logger = logger;
    }
    public void LogErrors()
    {
        _logger.Log();
    }
}
```

Liskov substitution Principle

Принцип подстановки Барбары Лисков

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом (поведение наследников не должно противоречить поведению базового класса)

```
public class Rectangle {

    private int length;
    private int breadth;

    public int getLength() {
        return length;
    }
    public void setLength(int length) {
        this.length = length;
    }
    public int getBreadth() {
        return breadth;
    }
    public void setBreadth(int breadth) {
        this.breadth = breadth;
    }
    public int getArea() {
        return this.length * this.breadth;
    }
}
```

```
public class Square extends Rectangle {

    @Override
    public void setBreadth(int breadth) {
        super.setBreadth(breadth);
        super.setLength(breadth);
    }
    @Override
    public void setLength(int length) {
        super.setLength(length);
        super.setBreadth(length);
    }
}
```

Liskov substitution Principle

Принцип подстановки Барбары Лисков

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом (поведение наследников не должно противоречить поведению базового класса)

```
public class LSPDemo {
    public void calculateArea(Rectangle r) {
        r.setBreadth(2);
        r.setLength(3);
        //
        assert r.getArea() == 6 : printError("area", r);
        //
        //
        assert r.getLength() == 3 : printError("length", r);
        assert r.getBreadth() == 2 : printError("breadth", r);
    }
```

```
public class Square implements HasArea {
    private int side;


    public int getSide() {
        return side;
    }

    public void setSide(int side) {
        this.side = side;
    }

    @Override
    public int getArea() {
        return side * side;
    }
}
```

```
public interface HasArea {
    int getArea();
}
```

Interface segregation

Принцип разделения интерфейса

Большие интерфейсы необходимо дробить на мелкие, т.о. чтобы изменение метода интерфейса не изменяло поведение клиентов, не использующих этот метод

```
public interface Athlete {

    void compete();

    void swim();

    void highJump();

    void longJump();

}
```

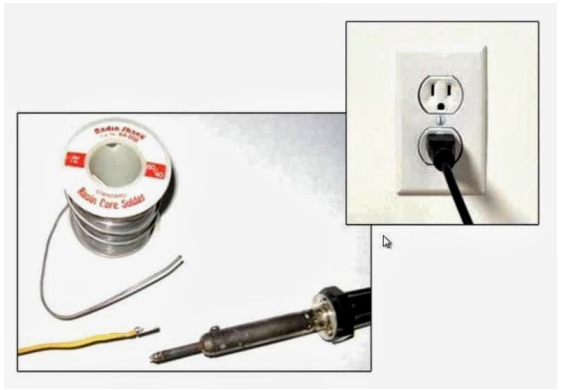
```
public interface Athlete {

void compete();
}

public interface SwimmingAthlete extends Athlete {

    void swim();

}


public interface JumpingAthlete extends Athlete {

    void highJump();

    void longJump();

}
```

Dependency inversion

Принцип инверсии зависимостей

верхние уровни не должны зависеть от нижних, абстракции не должны зависеть от деталей

```
public class XMLLogger
{
    public void Log()
    {
        // Perform the logging operation
    }
}
public class DbInteraction
{
    public void SaveData()
    {
        // Create Logger instance
        XMLLogger _xmlLogger = new XMLLogger();

        // Save data into database

        //Log into the XML database
        _xmlLogger.Log();
    }
}
```



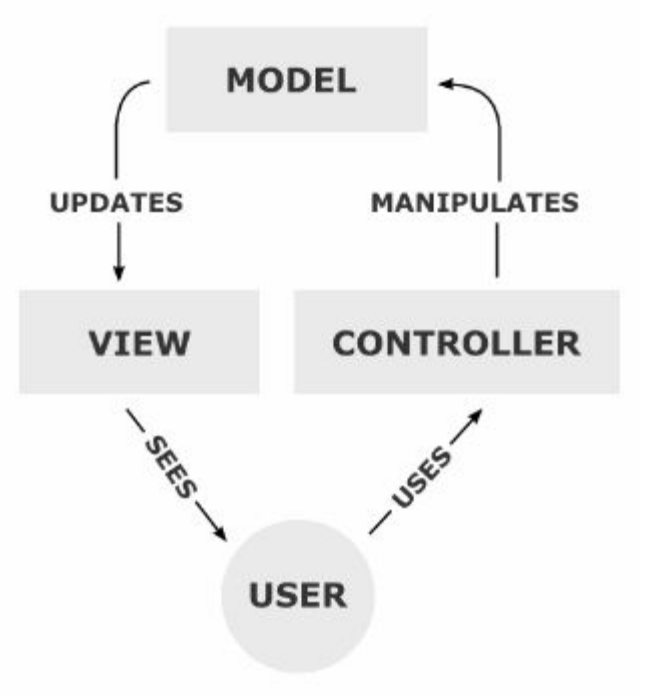
```
public interface ILogger
{
    void Log();
}
public class XMLLogger : ILogger
{
    public void Log()
    {
        // Perform the logging operation
    }
}
public class DbInteraction
{
    ILogger _iLogger;

    public void SaveData()
    {
        _iLogger = new XMLLogger();

        // Save data into database

        //Log into the XML database
        _iLogger.Log();
    }
}
```

Architectural Patterns

- MVC
- MVP
- MVVM
- MVI

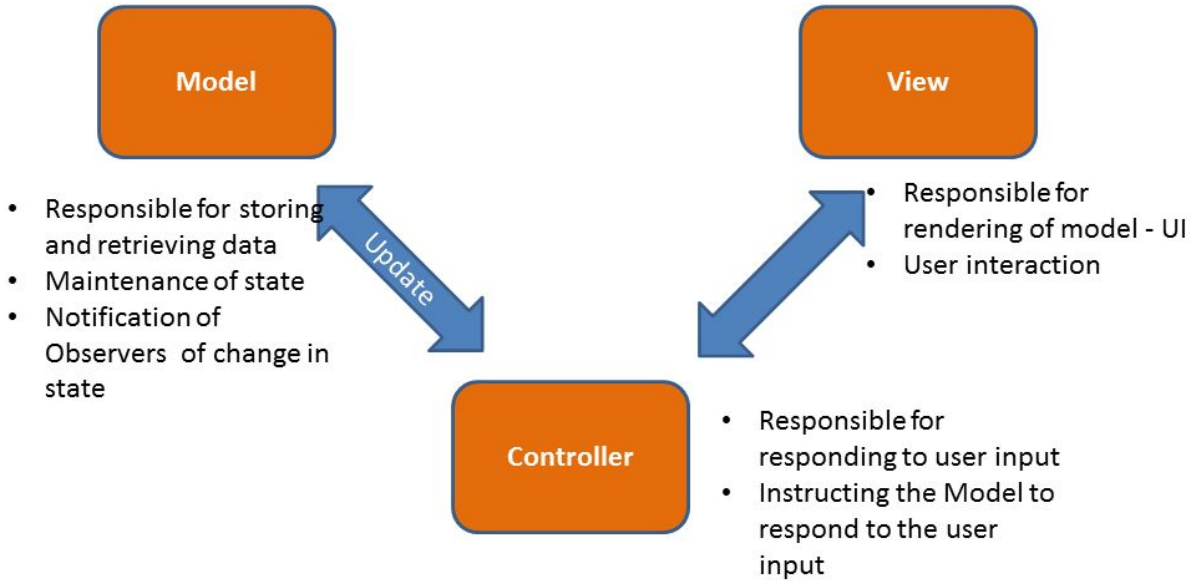
MVC

- **Model** - источник данных
- **View** - отображает данные на экране
- **Controller** - обрабатывает события пользовательского интерфейса, содержит бизнес логику



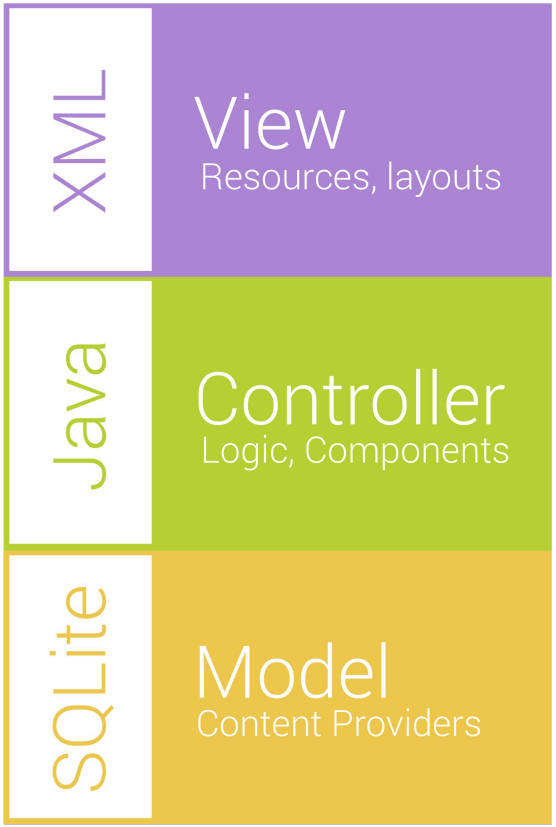
MVC

Model View Controller (MVC) Arch Passive Pattern

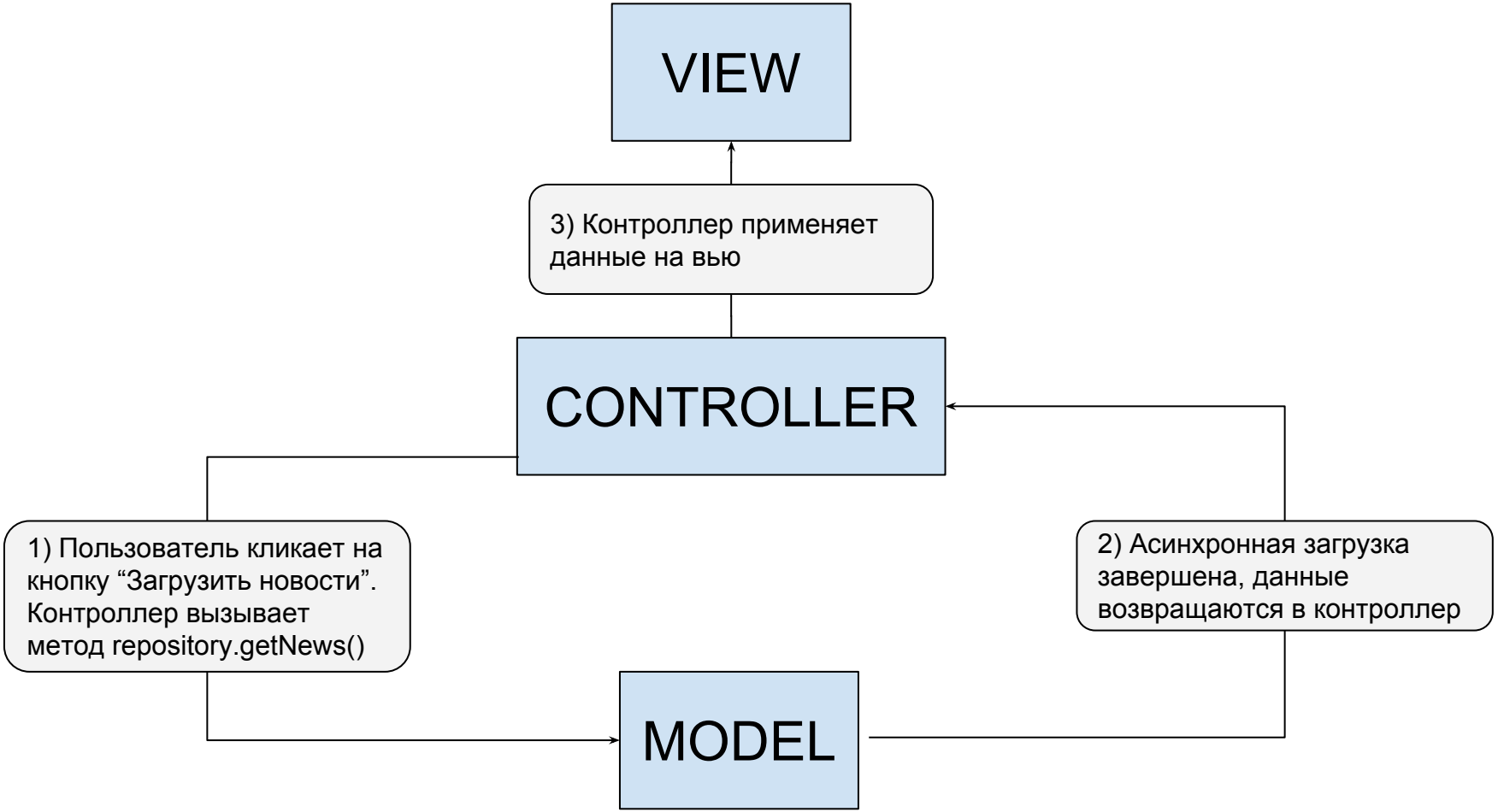


MVC in Android

- Model - сущности предметной области
- View - XML лейауты
- Controller - Activity или Fragment

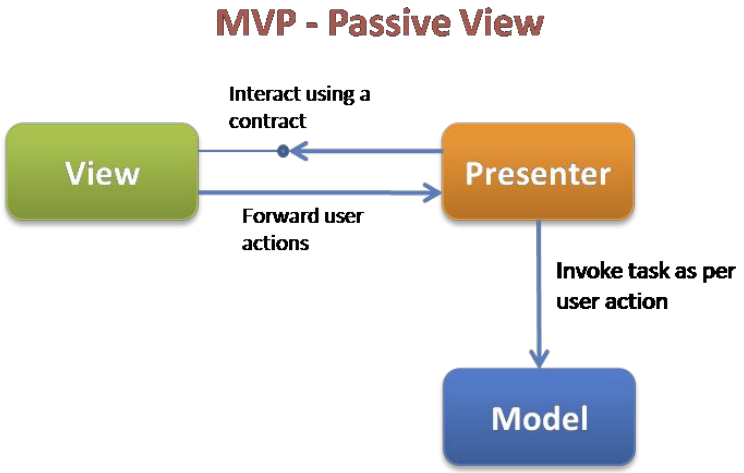


MVC in Android

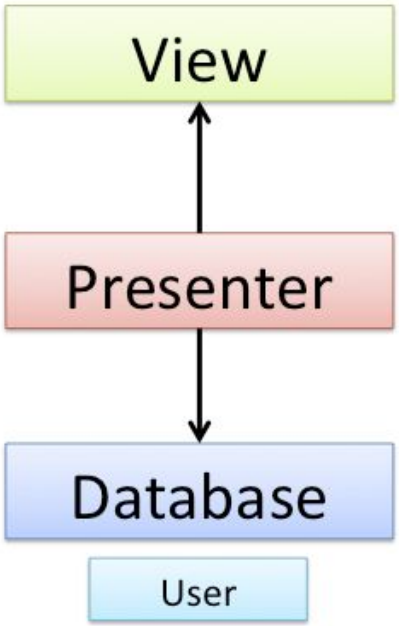


MVP

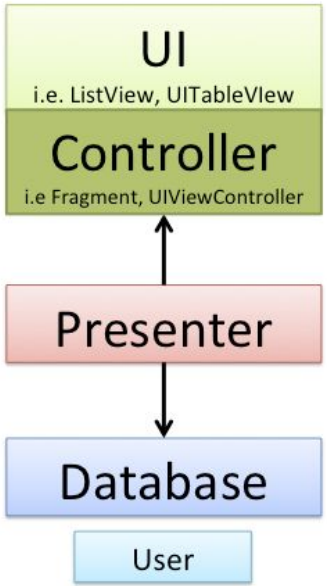
- **Model** - источник данных
- **View** - отображает данные и доставляет действия пользователя в Presenter
- **Presenter** - связывает View и Model, подготавливает данные из Model для отображения, изменяет Model в ответ на действия пользователя



MVP in Android



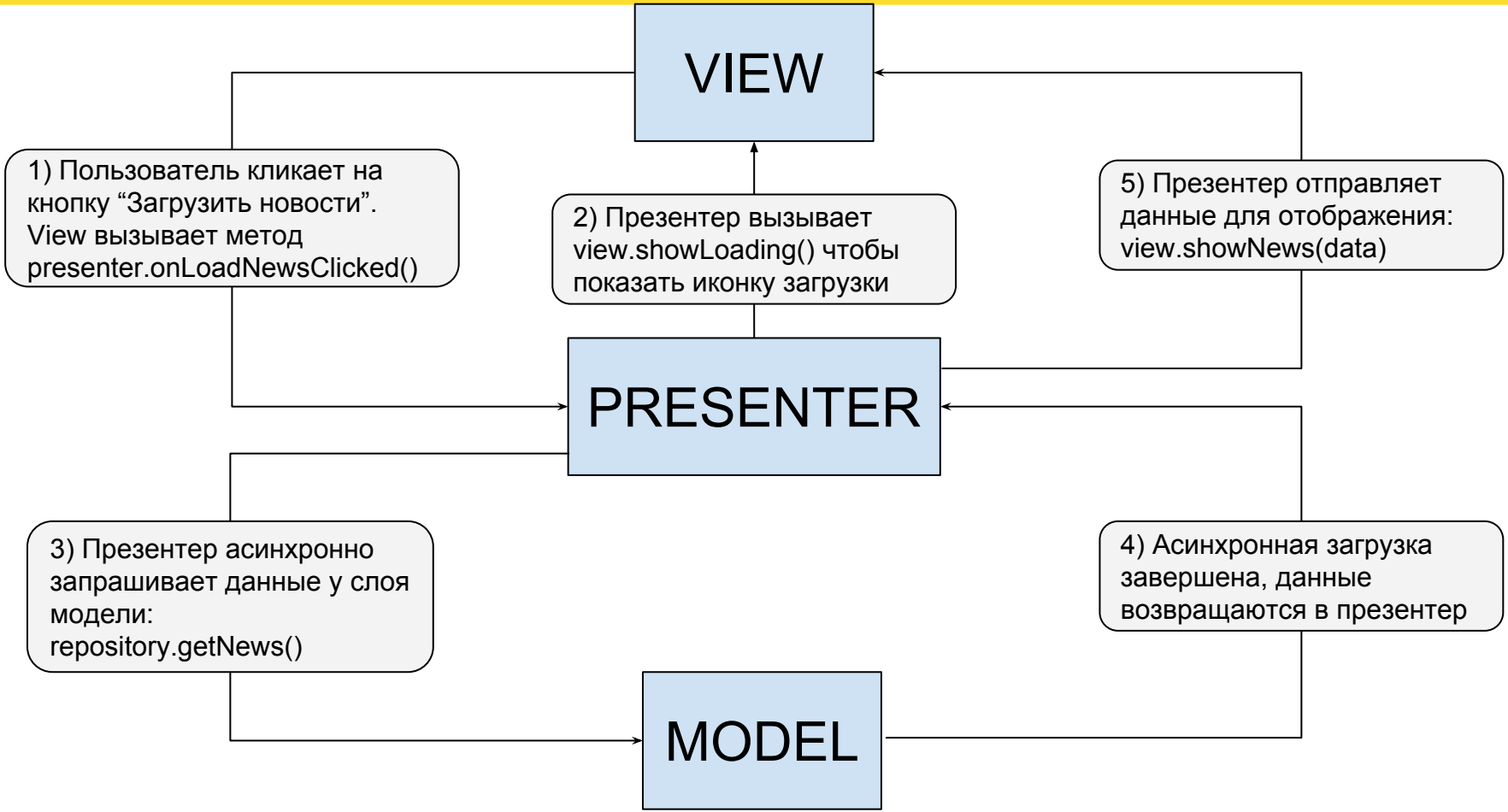
This layer is called "Model". I would rather call it "business logic"



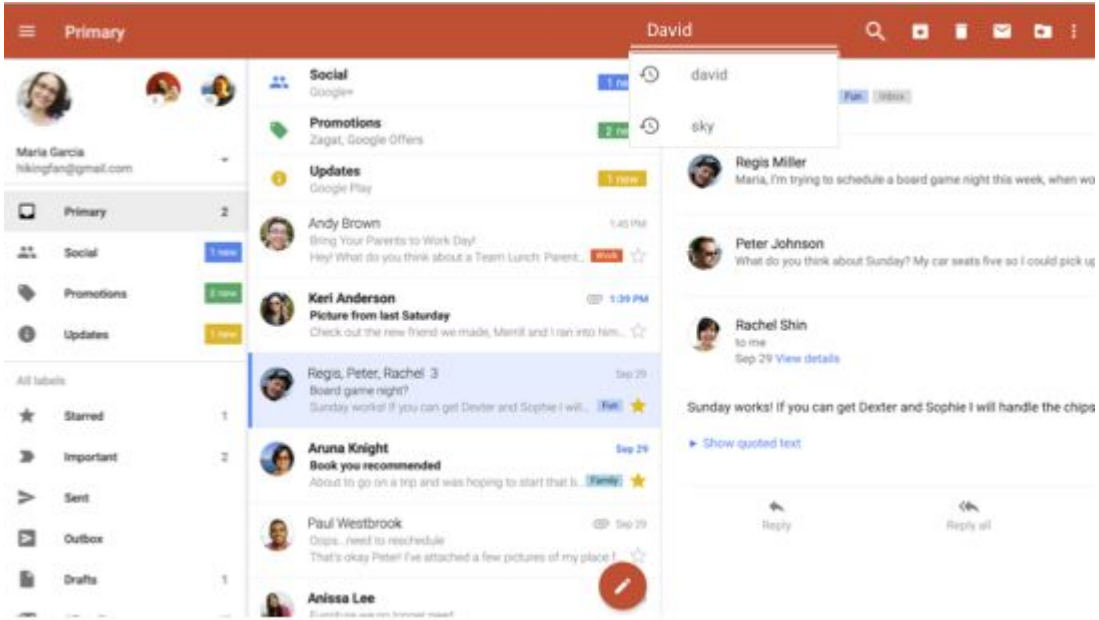
I call this layer **VIEW**.
It consists of two sub-layers:
1) UI (User Interface) Components
2) Controller

This layer is called **MODEL**. I would rather call it "business logic"

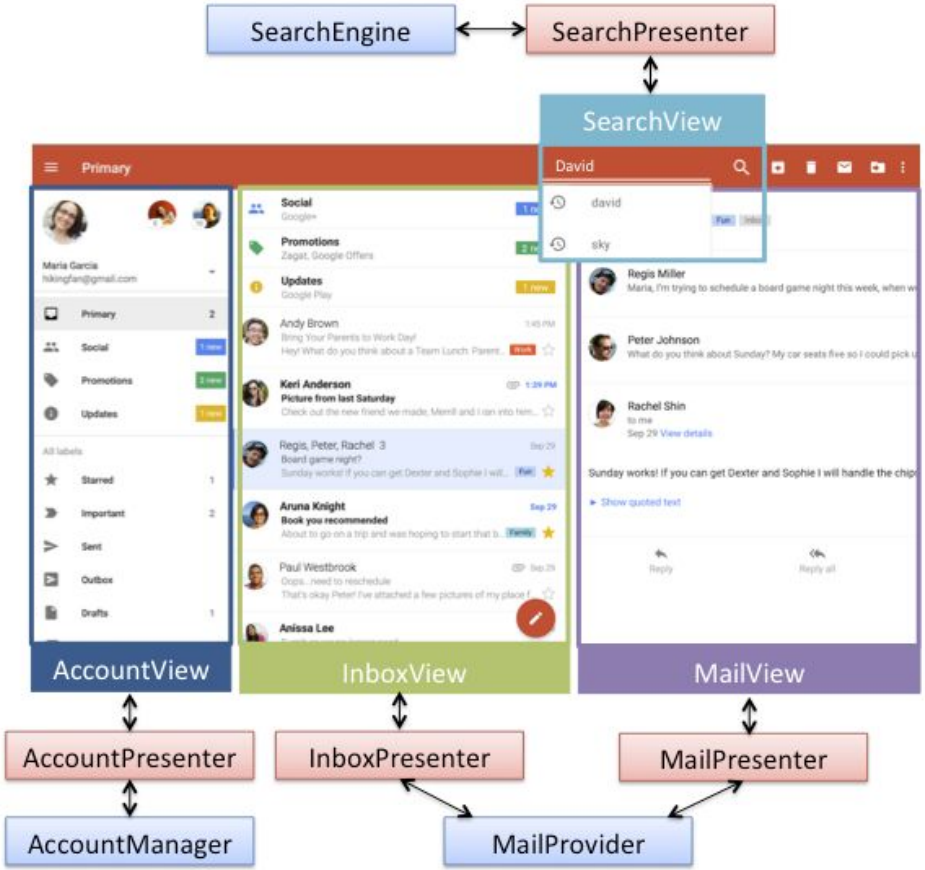
MVP in Android



MVP in Android

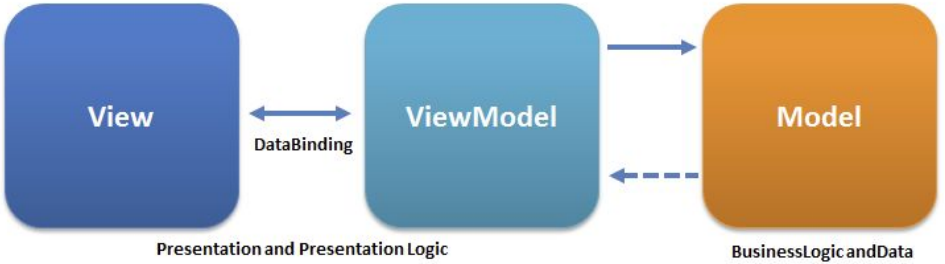


MVP in Android



MVVM

- **Model** - источник данных
- **View** - интерфейс пользователя, аналогично MVC / MVP
- **ViewModel** - преобразует модель для интерфейса и содержит binding-и



DataBinding

```xml
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable name="user" type="com.example.User"/>
    </data>
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.firstName}"/>
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.lastName}"/>
    </LinearLayout>
</layout>
```

```java
public class User {
    private final String firstName;
    private final String lastName;
    public User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String getFirstName() {
        return this.firstName;
    }
    public String getLastName() {
        return this.lastName;
    }
}
```

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    MainActivityBinding binding = DataBindingUtil.setContentView(this, R.layout.main_activity);
    User user = new User("Test", "User");
    binding.setUser(user);
}
```

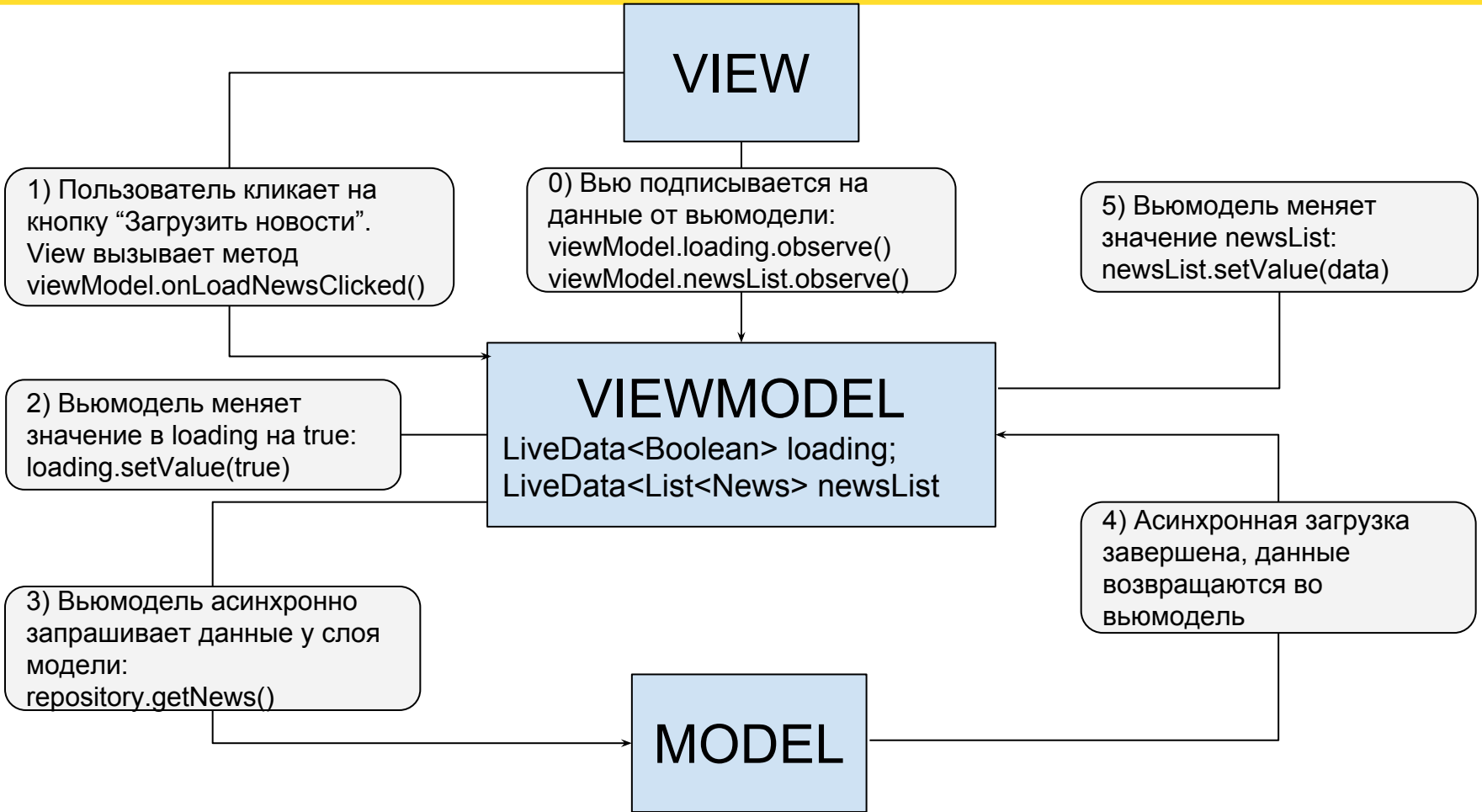
https://developer.android.com/topic/libraries/data-binding/index.html

Android Architecture Components

- ViewModel, которая переживает пересоздание Activity
- LiveData для трансляции данных из ViewModel во View

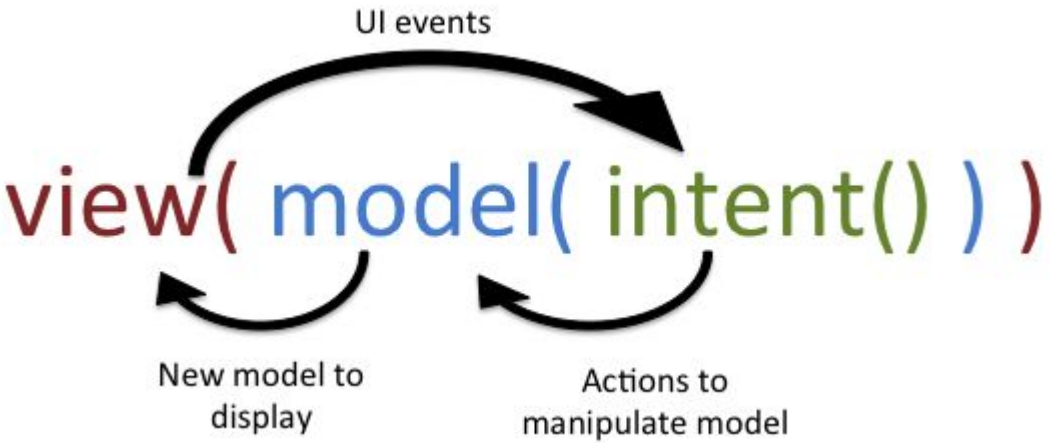
https://developer.android.com/topic/libraries/architecture/viewmodel.html

MVVM in Android



MVI

- **Model** - структура данных для ui
- **View** - render Model
- **Intent** - преобразованные UI события

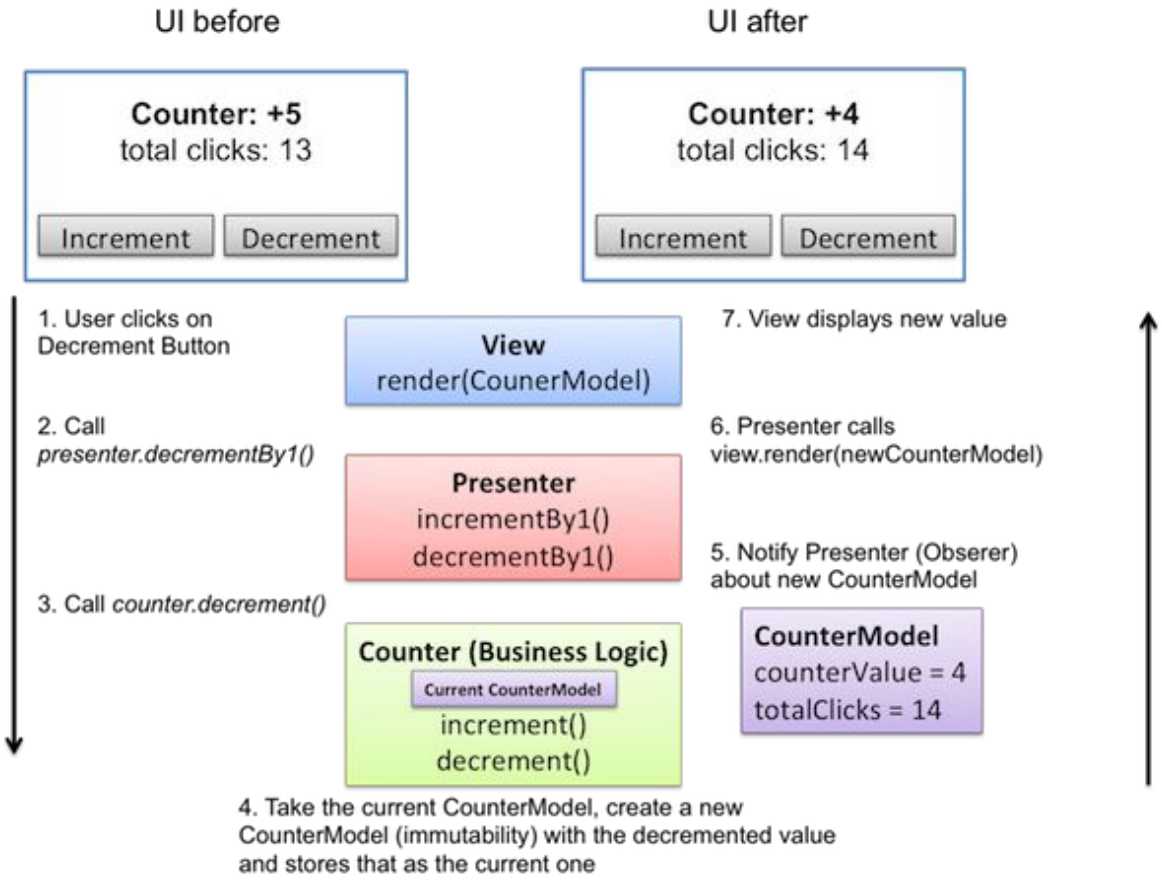


MVI

```
class PersonsModel {
  // В реальном приложении поля будут приватными
  // и у нас будут геттеры для доступа к ним
  final boolean loading;
  final List<Person> persons;
  final Throwable error;

  public(boolean loading, List<Person> persons, Throwable error){
    this.loading = loading;
    this.persons = persons;
    this.error = error;
  }
}
```

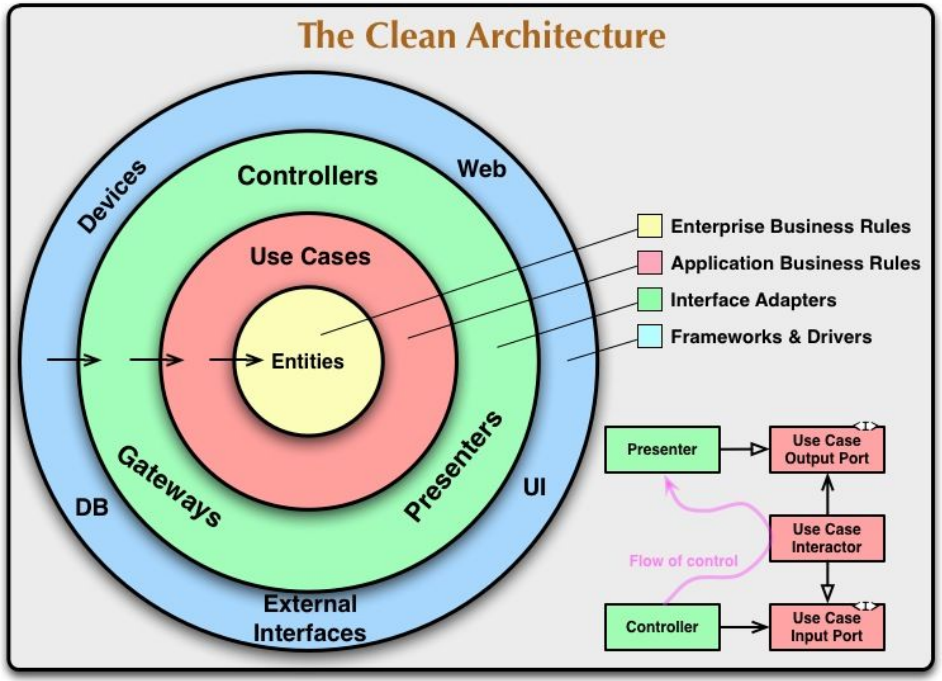
```
class PersonsPresenter extends Presenter<PersonsView> {

  public void load(){
    getView().render( new PersonsModel(true, null, null) ); // Показать ProgressBar

    backend.loadPersons(new Callback(){
      public void onSuccess(List<Person> persons){
        getView().render( new PersonsModel(false, persons, null) ); // Показать список
 людей
      }

      public void onError(Throwable error){
          getView().render( new PersonsModel(false, null, error) ); // Показать сообщен
ие об ошибке
      }
    });
  }
}
```

MVI

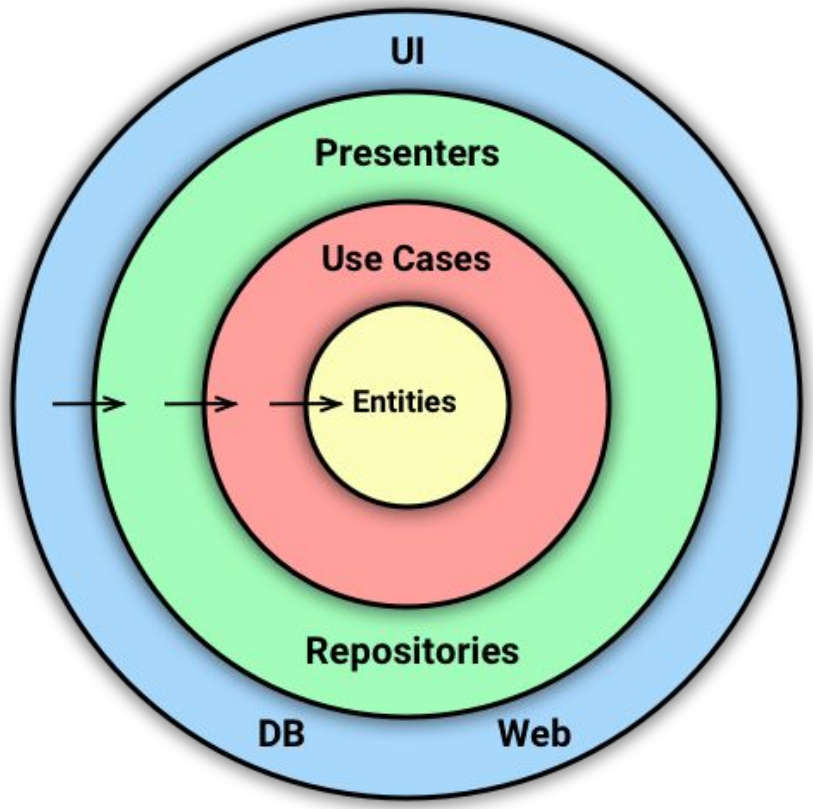

UI before
Counter: +5
total clicks: 13
Increment
Decrement
UI after
Counter: +4
total clicks: 14
Increment
Decrement
1. User clicks on Decrement Button
2. Call presenter.decrementBy1()
3. Call counter.decrement()
View
render(CounerModel)
Presenter
incrementBy1()
decrementBy1()
Counter (Business Logic)
Current CounterModel
increment()
decrement()
4. Take the current CounterModel, create a new CounterModel (immutability) with the decremented value and stores that as the current one
5. Notify Presenter (Obserer) about new CounterModel
CounterModel
counterValue = 4
totalClicks = 14
6. Presenter calls view.render(newCounterModel)
7. View displays new value

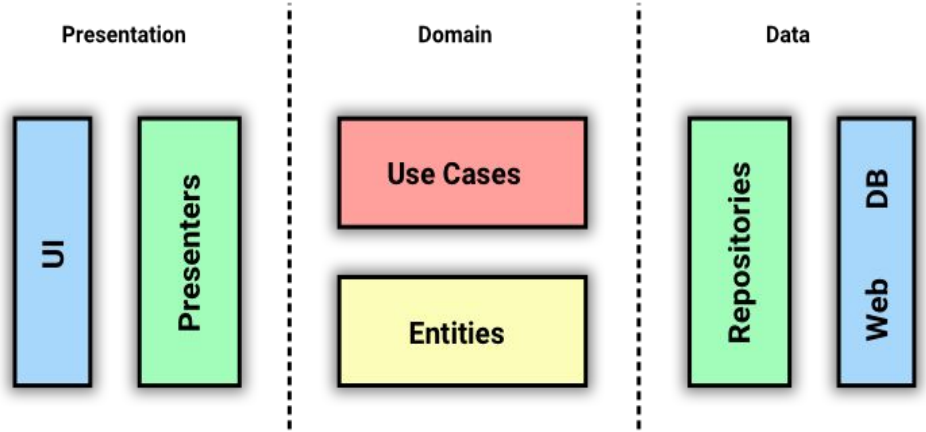
Clean Architecture

- Dependency Rule
- Crossing Boundaries

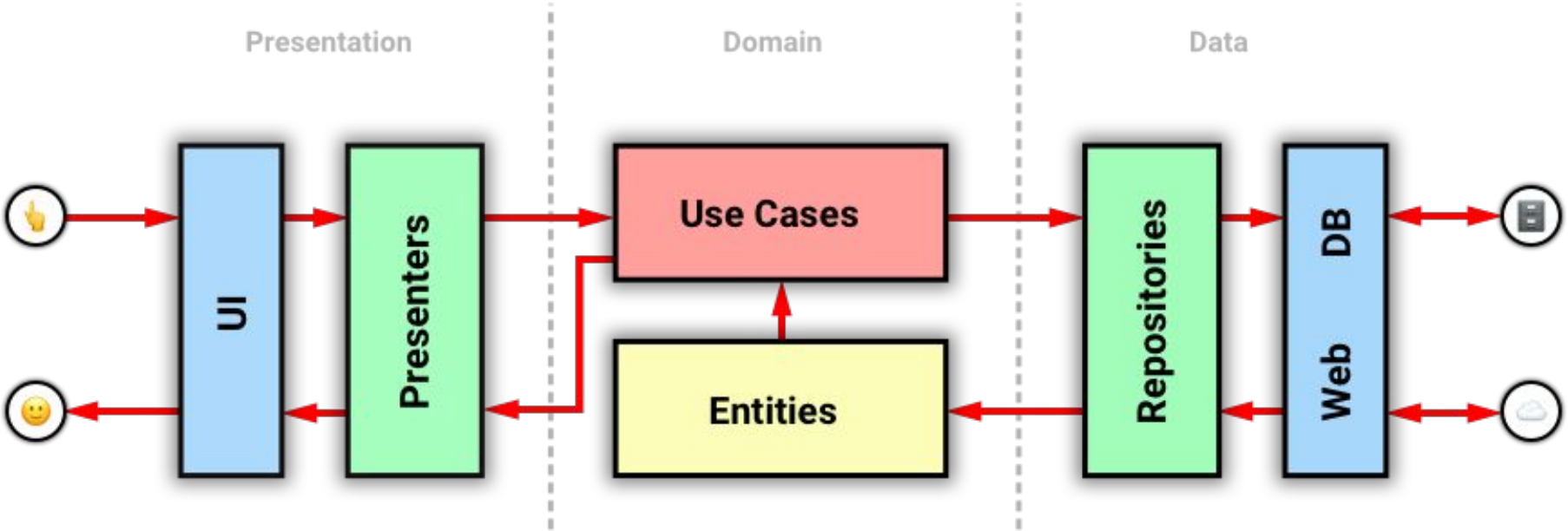


Clean Architecture

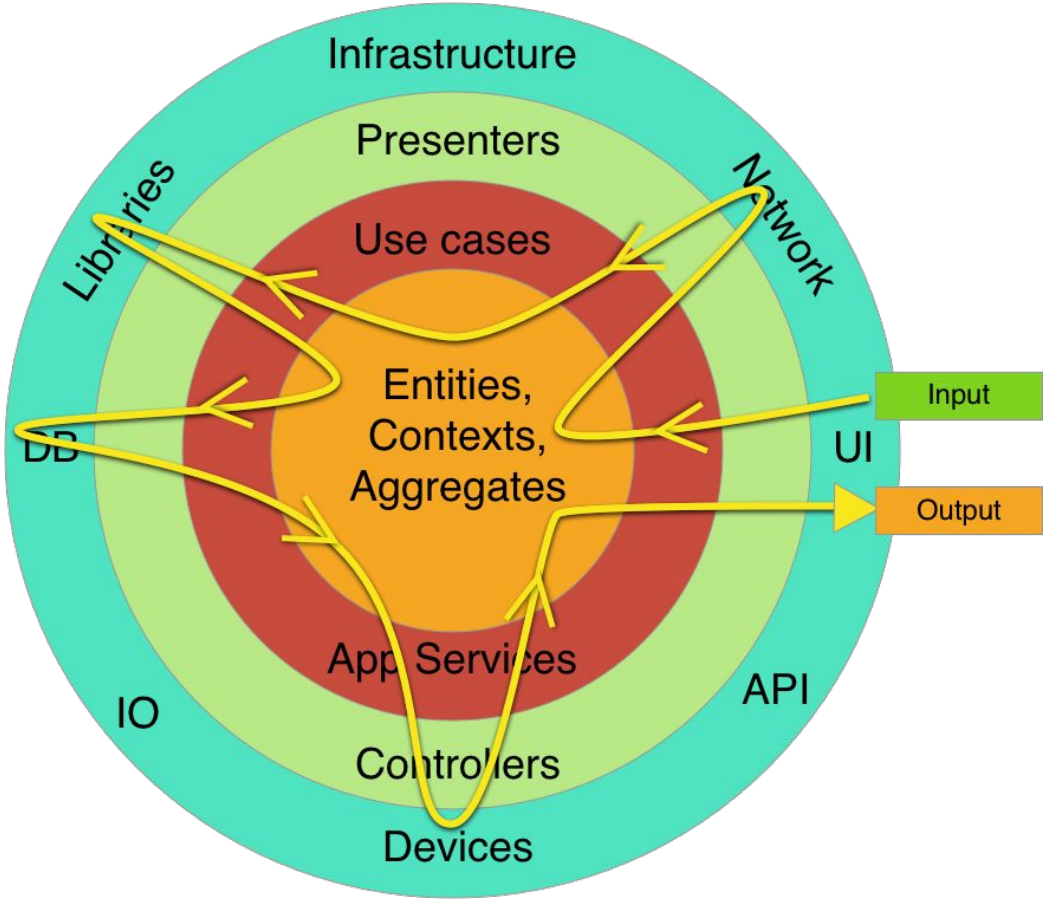


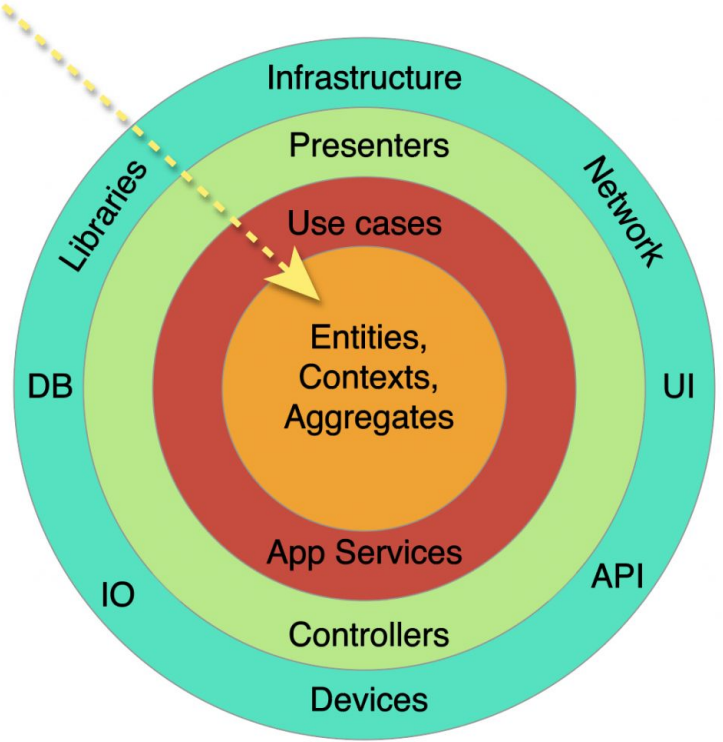


Clean Architecture



Clean Architecture





Ссылки

- Приемы объектно-ориентированного проектирования. Паттерны проектирования
  http://www.ozon.ru/context/detail/id/2457392/
- Рефакторинг. Улучшение существующего кода
  http://www.ozon.ru/context/detail/id/1308678/
- Common Design Patterns for Android with Kotlin
  https://www.raywenderlich.com/109843/common-design-patterns-for-android
- MVI (Model-View-Intent)
  http://hannesdorfmann.com/android/mosby3-mvi-1
- Реактивные приложения с Model-View-Intent
  https://habr.com/company/tinkoff/blog/325376/
  https://habr.com/company/tinkoff/blog/338558/
  https://habr.com/company/tinkoff/blog/348908/
- Clean Architecture and Design by Robert C Martin
  https://www.youtube.com/watch?v=Nsjsiz2A9mg
- Заблуждения Clean Architecture
  https://habr.com/company/mobileup/blog/335382/
- Примеры использования различных архитектурных подходов
  https://github.com/googlesamples/android-architecture

Спасибо за внимание