# Практическая работа №7 «Паттерны разработки приложений»

Как правило, приложение под Android состоит из множества компонентов, включая Activity, Fragment, Service, Content Provider и Broadcast Receiver. Большинство компонентов приложения разработчик описывает в манифесте приложения. Далее с помощью этого файла Android OS решает, как интегрировать приложение в пользовательский интерфейс устройства.

Учитывая, что среднестатистическое приложение для Android может содержать множество компонентов, а пользователи зачастую взаимодействуют с несколькими приложениями в течение короткого промежутка времени, приложениям необходимо адаптироваться к разнообразным процессам и задачам пользователя.

Мобильные устройства ограничены в ресурсах: операционная система в любое время может убить какой-нибудь процесс приложения, чтобы освободить место для новых. Не исключено, что компоненты приложения будут запускаться по отдельности и в неправильном порядке, а операционная система или пользователь в любой момент смогут их закрыть.

Так как нет контроля над этими события, не следует хранить или держать какие бы то ни было данные приложения или состояния в компонентах приложения. Сами компоненты приложения не должны зависеть друг от друга.

## Общепринятые архитектурные принципы

Если компоненты приложения не рекомендуется использовать для хранения данных и состояний приложения, как тогда проектировать приложение? Важно создать архитектуру, которая позволит масштабировать приложение, сделает его надёжнее и облегчит его тестирование.

Архитектура приложения устанавливает границы между разными частями приложения и их ответственностями. Чтобы удовлетворить описанные выше потребности, нужно создавать архитектуру приложения в соответствии с принципами:

- разделения ответственностей,
- построения UI на основе модели данных.

**Разделение ответственностей**

Принцип разделения приложения на функциональные блоки, как можно меньше перекрывающие функции друг друга, или принцип разделения ответственностей является одним из самый важных в разработке программных продуктов.

Писать весь код в Activity или Fragment — распространённая ошибка. Эти классы находятся в слое UI: в них должна содержаться только логика, которая обрабатывает взаимодействия UI и операционной системы. Если максимально облегчить эти классы, можно избежать проблем с жизненным циклом компонентов и упростить тестирование классов.

Вы не владеете реализациями Activity и Fragment — они, скорее, просто связующие звенья, которые представляют собой контракт между ОС Android и приложением. ОС в любой момент может их уничтожить при определенных взаимодействиях пользователя с приложением или состояниях системы — например, при недостатке оперативной памяти. При необходимости создать приемлемый пользовательский опыт и облегчить обслуживание приложения, лучше свести к минимуму зависимости от них.

**Построение UI на основе моделей данных**

UI нужно строить на основе моделей данных. Желательно — поддерживающих постоянное хранение данных.

Модели данных дают представление о данных, используемых приложением. Они:

- Не зависят от элементов UI и других компонентов.
- Не ограничены жизненным циклом компонентов приложения и UI.
- Будут уничтожены, когда ОС решит удалить из памяти процесс приложения.

Модели, поддерживающие постоянное хранение данных, идеально подходят по следующим причинам:

- Пользователи не потеряют данные, если Android закроет приложение, чтобы освободить ресурсы.
- Приложение продолжит работать, если подключение к Интернету нестабильно или недоступно.

Построив архитектуру приложения на основе классов моделей данных, вы сделаете его тестируемым и надежным.

## Общие рекомендации по созданию архитектуры приложения

Согласно общепринятым архитектурным принципам, у каждого приложения должно быть как минимум два слоя:

- Слой UI, который отображает данные приложения на экране.
- Слой данных, который содержит бизнес-логику приложения и открывает доступ к данным приложения.

К ним можно добавить ещё один слой — доменный. Он помогает упростить и переиспользовать взаимодействия между слоями UI и данных.

Общая схема приложения проиллюстрирована на рисунке 1.

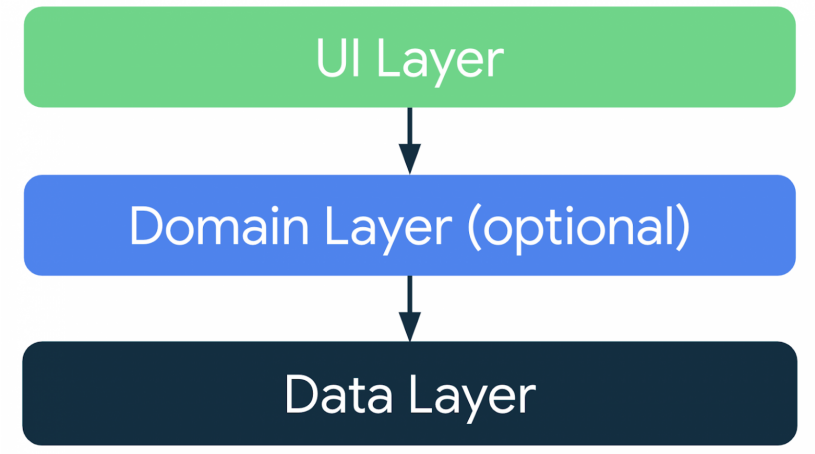


Рисунок 1 – общая схема архитектуры приложения

**Слой UI**

Роль слоя UI (или слоя представления) — отображать на экране данные приложения. Если данные меняются — из-за взаимодействия с пользователем (например, нажатия кнопки) или внешнего воздействия (например, отклика сети) — UI должен обновиться и отразить изменения.

Слой UI состоит из двух частей (рисунок 2):

- Элементов UI, которые отображают данные на экране. Создаются с помощью функций Jetpack Compose или View.
- Экземпляров state holder (например, классов ViewModel), которые хранят данные, открывают к ним доступ для UI и работают с логикой.

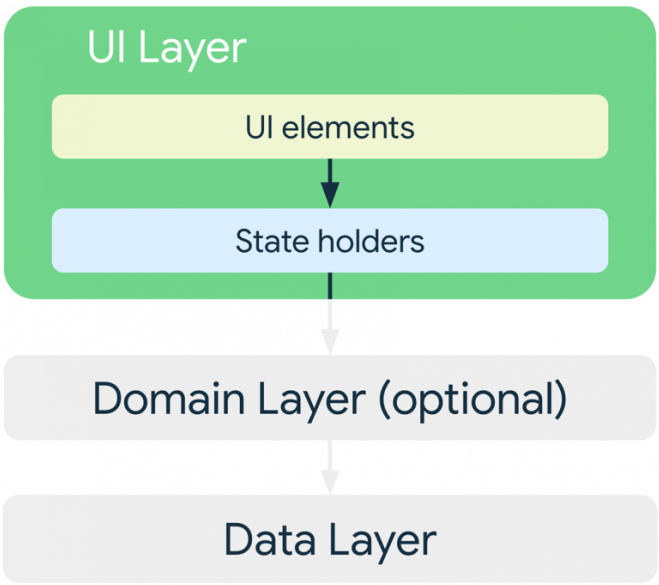


Рисунок 2 – содержимое UI слоя

**Слой данных**

Слой данных в приложении содержит логику хранения данных, а именно правила, по которым приложение создаёт, хранит и изменяет данные. Именно благодаря данной логике приложение имеет ценность.

Слой данных состоит из классов Repository (рисунок 3). В каждом может содержаться множество классов DataSource или не быть ни одного. Отдельный класс Repository следует создать для каждого уникального типа данных в приложении. В дальнейшем паттерн Repository будет рассмотрен более подробно.

Каждый DataSource-класс должен отвечать за работу только с одним источником данных. Им может оказаться файл, сетевой источник или локальная база данных. DataSource-классы — связующее звено между приложением и системой для работы с данными.

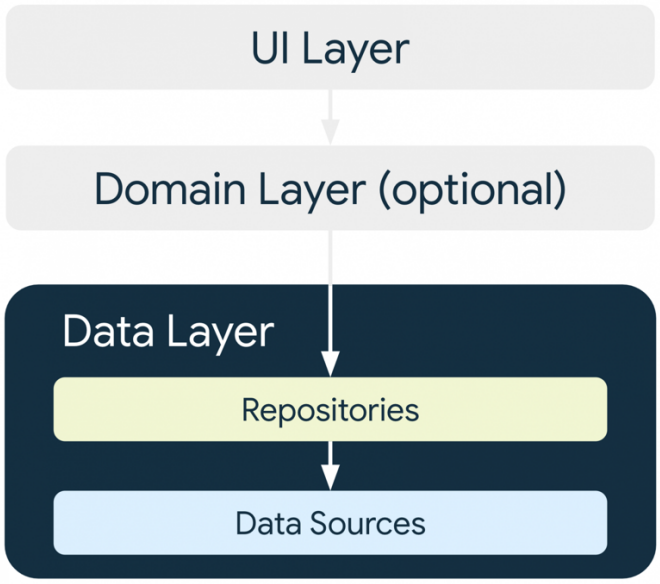


Рисунок 3 – Содержимое слоя данных в архитектуре приложения

**Доменный слой**

Доменный слой располагается между слоями UI и данных (рисунок 4).

Доменный слой отвечает за инкапсуляцию сложной бизнес-логики или простой бизнес-логики, которую переиспользуют несколько ViewModel.

Этот слой необязателен, так как не всем приложениям он нужен. Внедрять его следует только при необходимости. Например, при работе со сложной логикой или чтобы переиспользовать логику.

Классы этого слоя, как правило, называют UseCase или Interactor. Каждый UseCase должен отвечать только за одну функциональность. Например, в приложении может быть один класс GetTimeZoneUseCase и несколько ViewModel, которые в зависимости от часового пояса отображают соответствующее сообщение на экране.

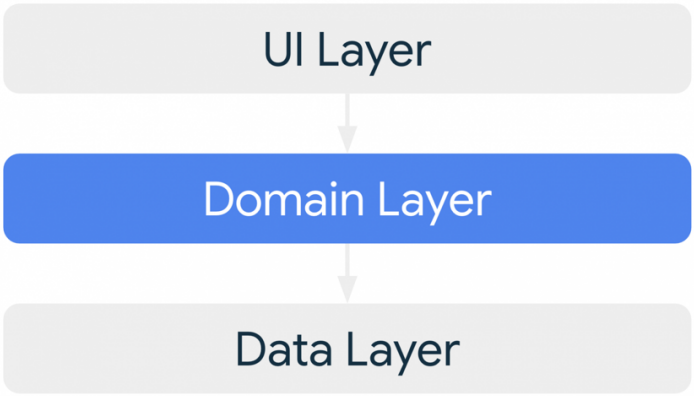


Рисунок 4 – Расположения доменного слоя в архитектуре приложения

## Архитектурные паттерны, используемые в Android приложения

При разработке сложных приложений можно столкнуться с проблемами, которые, вероятно, возникали раньше и уже имеют большое количество решений. Такие решения называются паттернами (шаблонами). Как правило, говорят о паттернах дизайна и паттернах архитектуры. Они упрощают разработку приложений, поэтому целесообразно их использовать, если такая возможность есть.

Архитектура приложений играет жизненно важную роль в структурировании слабосвязанной кодовой базы. Вы можете использовать ее где угодно, независимо от платформы. Архитектура приложений помогает писать легко тестируемый, расширяемый и несвязанный код.

Существует множество архитектур приложений, используемых для создания надежных и поддерживаемых кодовых баз, но в этой практической работе будут разобраны следующие из них:

- Model View Controller – MVC
- Model View Presenter – MVP
- Model View ViewModel – MVVM

Основная идея любого из паттернов MVP, MVC, MVVM заключается в разделении логики и UI-части приложения так, чтобы их можно было тестировать по отдельности.

Сложно не заметить, что все эти паттерны содержат View и Model, а отличия заключаются в последнем элементе, который управляет логикой. В случае MVC это Controller, в случае MVP – Presenter, в MVVM – ViewModel. Для удобства можно назвать этот отличающийся объект – делегатом. Разумеется, различия заключаются не только в названии, но и в том, как именно делегат управляет логикой и взаимодействует с View и Model.

Начнем рассмотрение паттернов с их общих частей – View и Model:

- Model – это обычные классы объектов, которые используются при взаимодействии View с делегатом. Плюс их использования заключается в том, что мы разделяем сущности, что может упрощать понимание.
- View отображает данные, получаемые либо от Model, либо от делегата, что зависит от конкретного паттерна. View – эта та часть системы, которая видна пользователю и которая взаимодействует с ним. При этом View не должна содержать логику, а передавать результаты взаимодействия делегату, который будет управлять этой View.

При этом сами паттерны достаточно сильно отличаются между собой, поэтому далее будем рассматривать их по-отдельности.

**Паттерн Model View Controller**

Model View Controller, или MVC, относится к одному из самых популярных архитектурных шаблонов в программных продуктах, от которого происходят многие другие. Схема этого паттерна проиллюстрирована на рисунке 5.

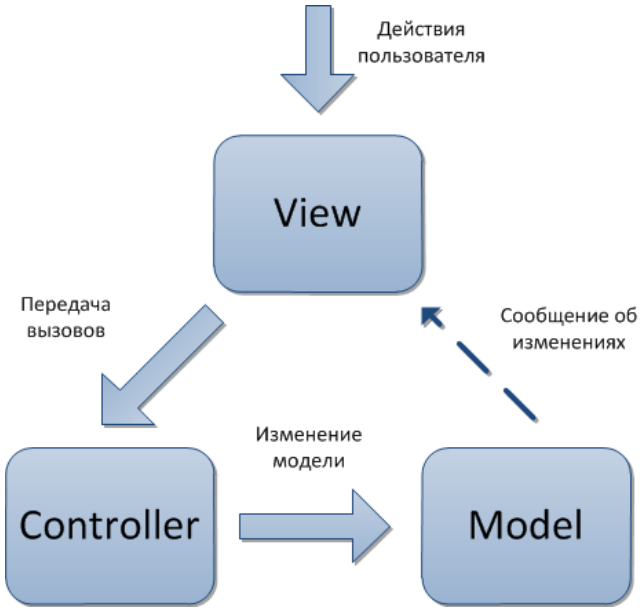


Рисунок 5 – схематичное представление паттерна MVC

Когда пользователь взаимодействует со View (к примеру, нажимает на кнопку), View передает информацию об этом действии в Controller. Controller обрабатывает это событие в соответствии с логикой системы и изменяет Model. View отслеживает состояние модели, поэтому при изменении Model View получает уведомление и отображает новую информацию. Это активная модель, которая применяется чаще всего. Также существует пассивная модель, в которой View обновляется через Controller.

И есть еще один важный момент – Controller может управлять несколькими View, при этом Controller определяет, какая именно View будет отображаться в текущий момент. Именно из-за этой особенности паттерн MVC не слишком удобно применять в Android. В Android в роли View может выступать Activity, которую просто так сменить, разумеется, невозможно. Есть определенный вариант использования MVC в Android, когда разными View являются фрагменты, которые переключает Controller. В таком случае View может отслеживать изменение состояния через, к примеру, ContentProvider. Но, к сожалению, чаще всего попытки реализовать MVC в Android вели к использованию Activity в качестве God object (когда все сущности, и вся логика находится в Activity).

**Паттерн Model View Presenter**

MVP имеет несколько основных отличий от MVC. Во-первых, Presenter управляет только одной View и взаимодействует с ней через специальный интерфейс. Во-вторых, View управляется только с помощью Presenter-a, а не отслеживает изменение Model. Presenter получает все данные из слоя данных, обрабатывает их в соответствии с требуемой логикой и управляет View. Схема паттерна MVP проиллюстрирована на рисунке 6.


Рисунок 6 – схематичное представление паттерна MVP

Такое разделение слоёв позволяет лучше тестировать приложение, поскольку заранее известно, на каком уровне нужно искать код.

**Пример реализации архитектурного паттерна MVP в Android приложении**

Для лучшего понимания работы архитектурного паттерна MVP в Android приложении далее будет разобран не большой пример по реализации мобильного приложения, которое будет по нажатию на кнопку загружать строку из хранилища данных и отображать в TextView. Для простоты реализации наша импровизированное хранилище данных содержит список лучших ресторанов города.

В первую очередь для простоты реализации и дальнейшей читаемости кода будет реализован контракт нашей связки экрана, делегата и слоя данных. В качестве класса реализатора будет использован интерфейс, позволяющий не предоставлять конкретных

реализаций описываемых методов сейчас и переназначить логику описанных методов в дальнейшем. Описание контракта приведено на рисунке 7.

```
public interface MainContract {
    interface View {
        void showText();
    }

    interface Presenter {
        void onButtonWasClicked();
        void onDestroy();
    }

    interface Repository {
        String loadMessage();
    }
}
```

Рисунок 7 – реализация контракта

Следующим этапом выполним реализацию нашего слоя данных. В связи с уровнем нашего приложения в качестве хранилища данных будет использоваться оперативная память устройства, в которой будет лежать одно значение, которое и будет возвращено при обращении в слой данных. Реализация слоя данных изображена на рисунке 8.

```
public class MainRepository implements MainContract.Repository {

    private String singleValue = "Ресторан на Арбатской";

    @Override
    public String loadMessage() {
        return singleValue;
    }
}
```

Рисунок 8 – реализация слоя данных

Далее реализуем используемый в приложении делегат. Так как приложение строится на архитектурном паттерне MVP, то в качестве реализуемого делегата будет выступать класс Presenter. В нем необходимо реализовать все описанные методы в ранее созданном контракте. В качестве метода обработки нажатия на кнопку на экране будет выступать метод onButtonWasClicked(), а логика окончания работы делегата будет обрабатываться в методе onDestroy(). С реализацией делегата можно ознакомиться на рисунке 9.

```java
public class MainPresenter implements MainContract.Presenter {
    private static final String TAG = "MainPresenter";

    // Компоненты MVP приложения
    private MainContract.View mView;
    private MainContract.Repository mRepository;

    // Сообщение
    private String message;


    // Экземпляр View передается аргументом конструктора,
    // Repository создаётся внутри логики свмого конструктором.
    public MainPresenter(MainContract.View mView) {
        this.mView = mView;
        this.mRepository = new MainRepository();
    }

    //View сообщает, что кнопка была нажата
    @Override
    public void onButtonWasClicked() {
        message = mRepository.loadMessage();
        mView.showText(message);
        Log.d(TAG, "onButtonWasClicked()");
    }

    @Override
    public void onDestroy() {
        // Если бы реализовывалась работа с RxJava, в этом классе стоило бы отписываться от подписок
        // Кроме того, при работе с асинхронными методами,здесь необходимо их завершить
    }
}
```

Рисунок 9 – реализация делегата

Последний компонент архитектурного паттерна MVP – View, реализуется посредством создания собственного класса наследника Activity – MainActivity. Его реализацию можно увидеть на рисунке 10.

```java
public class MainActivity extends AppCompatActivity implements MainContract.View {

    private static final String TAG = "MainActivity";

    private MainContract.Presenter mPresenter;

    private Button mButton;

    private TextView myTv;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //Создаём Presenter и в аргументе передаём ему this - объект самого MainActivity, реализатор интерфейса MainContract.View
        mPresenter = new MainPresenter(this);

        myTv = (TextView) findViewById(R.id.text_view);
        mButton = (Button) findViewById(R.id.button);

        mButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                mPresenter.onButtonWasClicked();
            }
        });
    }

    @Override
    public void showText(String message) {
        myTv.setText(message);
    }

    //Вызываем у Presenter метод onDestroy, чтобы избежать утечек контекста и прочих неприятностей.
    @Override
    public void onDestroy() {
        super.onDestroy();
        mPresenter.onDestroy();
    }
}
```

Рисунок 10 – реализация компонента View

Реализовав все компоненты паттерна, подведем итог тому, что происходит в приложении и в каком порядке:

- Activity, она же View, в методе onCreate() создаёт экземпляр делегата Presenter и передаёт ему в конструктор собственный объект;
- Presenter при создании явно получает View и создаёт экземпляр Repository;
- При нажатии на кнопку, View передает вызов делегату и сообщает, что кнопка была нажата;
- Presenter обращается к Repository с запросом на предоставление ему необходимых данных;
- Repository производит выгрузку данных и передает их делегату Presenter;
- Presenter обращается к View передавая ему данные на отрисовку.

**Паттерн Model View ViewModel**

Этот архитектурный шаблон с довольно запутанным названием похож на шаблоны MVC и MVP. Компоненты Model и View выступают в качестве тех же элементов, что и в ранее разобранных паттернах.

Данный подход позволяет связывать элементы представления со свойствами и событиями View-модели. Объект ViewModel является связующим звеном между UI слоем и слоем данных, но работает иначе, чем делегаты Controller или же Presenter. Вместо этого он предоставляет команды для View и привязывает View к Model. Когда Model обновляется, соответствующие View обновляются через связывание данных.

Точно так же, когда пользователь взаимодействует с View, связывание работает в противоположном направлении, автоматически обновляя модель. Можно утверждать, что каждый слой этого паттерна не знает о существовании другого слоя, но при этом зависит от взаимодействия или изменения другого компонента в данной связке.

Так же, как и в паттерне MVP на каждый компонент View создается собственный делегат, обрабатывающий и связывающий его с Model. Однако в отличии от MVP, при пересоздании View нет необходимости пересоздавать делегат, так как связь между компонентами паттерна реализуется не на прямую, а косвенно, посредством подписки на изменение данных. Помимо этого, положительным фактором в возможность такой логики выступает факт того, что ViewModel имеет собственный жизненный цикл и не зависит от жизненного цикла View. Все ViewModel хранятся и зависят от состояния ViewModelScope, который прекращает свое существование только на момент уничтожения своего ViewModelStoreOwner. Пример жизненного цикла ViewModel, у которого в качестве ViewModelStoreOwner выступает класс наследник Activity можно увидеть на рисунке 11.

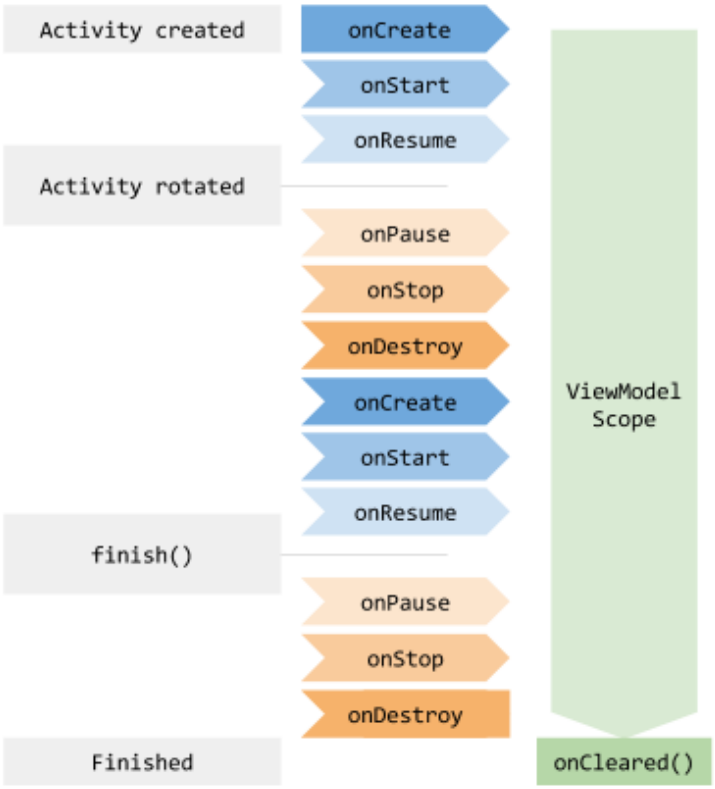


Рисунок 11 – Жизненный цикл ViewModel

Схема данного архитектурного паттерна проиллюстрирована на рисунке 11.

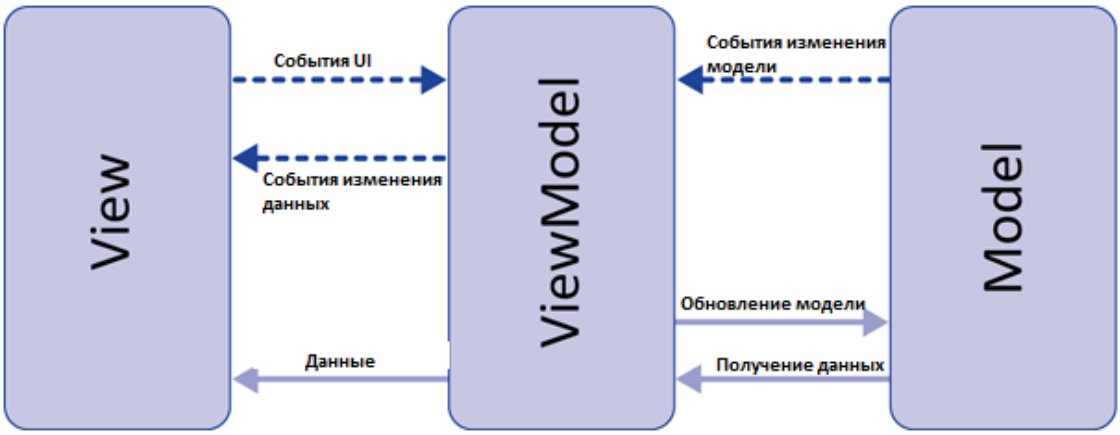


Рисунок 11 – Схема архитектурного паттерна MVVM

**Компонент LiveData**

Компонент LiveData не является архитектурным паттерном или частью одного из выше разобранных паттернов. Однако, компонент LiveData часто используют в паттерне MVVM для реализации принципа «зависимости» одной части паттерна от другой.

Компонент LiveData — предназначен для хранения объекта и предоставления возможности подписаться на его изменения. Ключевой особенностью является то, что компонент осведомлен о жизненном цикле и позволяет не беспокоится о том, на каком этапе сейчас находиться подписчик, в случае уничтожения подписчика, компонент отпишет его от себя.

Класс LiveData, является абстрактным дженерик классом, инкапсулирующим всю логику работы компонента. Если есть необходимость создать хранилище некоторых данных в приложении, в котором данные могут появиться не сразу, а спустя время, то необходимо получить объект LiveData с указанием типа данных содержимого в качестве дженерик типа. После получения объекта данного типа появляется возможность произвести подписку на изменение данных внутри данного хранилища. Для этого необходимо вызвать у данного объекта метод observe, с передачей в него жизненного цикла подписанта и логики обработки изменений данных в хранилище. Логика обработки изменений предоставляется в виде объекта абстрактного дженерик класса Observer с перегруженным у него методом onChanged, в котором указывается получаемое измененное значение в качестве входного параметра. Логику получения объекта LiveData с дальнейшей подпиской на изменение данных в ней можно увидеть на рисунке 12.

```
LiveData<String> liveData = viewModel.getData();

liveData.observe(this, new Observer<String>() {
    @Override
    public void onChanged(@Nullable String value) {
        textView.setText(value)
    }
});
```

Рисунок 12 – логика подписки на измененные данные в LiveData

Класс MutableLiveData, является расширением LiveData, с отличием в том, что это не абстрактный класс, а методы setValue(T) и postValue(T) публичные. По факту класс является помощником для тех случаев, когда мы не хотим помещать логику обновления значения в LiveData, а лишь хотим использовать его как держателя данных.

**Пример реализации архитектурного паттерна MVVM совместно с компонентом LiveData в Android приложении**

Для примера реализации приложения по архитектурному паттерну MVVM с использованием компонента LiveData создадим основные составляющие мобильного приложения, позволяющие пользователю бросить игральные кости и увидеть результат броска.

В первую очередь необходимо реализовать структуру данных. Для этого будет использоваться класс DiceUIState, описывающий в себе значение двух брошенных кубиков и количество совершенных попыток. Реализацию модели можно увидеть на рисунке 13.

```
public class DiceUiState {
    private final Integer firstDieValue;
    private final Integer secondDieValue;
    private final int numberOfRolls;
}
```

Рисунок 13 – Реализация компонента Model

Далее необходимо реализовать делегат, производящий работу нашего «броска» игральных костей. В связи с тем, что мы используем архитектурный паттерн MVVM в связке с компонентом LiveData, логика зависимости ранее описанной модели результата броска и компонента View будет производиться через хранилище LiveData и возможность подписаться на это хранилище. Результат исполнения делегата проиллюстрирован на рисунке 14.

```
public class DiceRollViewModel extends ViewModel {

    private final MutableLiveData<DiceUiState> uiState =
        new MutableLiveData(new DiceUiState(null, null, 0));
    public LiveData<DiceUiState> getUiState() {
        return uiState;
    }

    public void rollDice() {
        Random random = new Random();
        uiState.setValue(
            new DiceUiState(
                random.nextInt(7) + 1,
                random.nextInt(7) + 1,
                uiState.getValue().getNumberOfRolls() + 1
            )
        );
    }
}
```

Рисунок 14 – Реализация делегата ViewModel

Крайним элементом, который необходимо реализовать, является компонент View. В качестве него будет реализован класс Activity с получением из хранилища ViewModel-ей ViewModelProvider нашего делегата и подписки на изменение значения лежащего в нем хранилища, описанного в прошлом шаге, в методе onCreate. Реализацию логики компонента View можно увидеть на рисунке 15.

```
public class MyActivity extends AppCompatActivity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Create a ViewModel the first time the system calls an activity's onCreate() method.
        // Re-created activities receive the same MyViewModel instance created by the first activity.
        DiceRollViewModel model = new ViewModelProvider(this).get(DiceRollViewModel.class);
        model.getUiState().observe(this, uiState -> {
            // update UI
        });
    }
}
```

Рисунок 15 – Реализация компонента View

Реализовав все компоненты паттерна, подведем итог тому, что происходит в приложении и в каком порядке:

- Activity на момент своего создания запрашивает из хранилища ViewModelProvider реализацию нашего делегата ViewModel.
- После получения объекта делегата, Activity получает объект хранилища результата броска, которое является полем созданного делегата.
- Когда объект хранилища получен Activity производит подписку на изменение данных в хранилище посредством передачи своего жизненного цикла и

реализации логики обработки пользовательского интерфейса на момент обновления данных.

- Так же Activity имеет возможность задать действие «броска» игральных костей посредством вызова у делегата метода rollDice(), который обновит данные в хранилище и запустит процесс обновления данных у компонента View.

**Паттерн Repository**

Когда в приложении появляется более одного источника данных начинает появляться вопрос, какой из этих источников считать в качестве основного? Это один из стандартных вопросов проблемы единого источника правды или как ее называют сокращенно – SSOT. В проектировании и теории информационных систем единый источник правды (SSOT) — это практика структурирования информационных моделей и связанных схем таким образом, чтобы каждый элемент данных сохранялся ровно один раз. Как раз для решения этой проблемы используется паттерн Repository.

Паттерн Repository — это паттерн структурного проектирования. Этот инструмент полезен для организации доступа к данным. Наиболее распространенными операциями являются создание, чтение, обновление и удаление данных (также известные как CRUD – creating, reading, updating, deleting data). Этим операциям иногда нужны параметры, которые определяют, как их запускать. Например, параметр может быть поисковым запросом для фильтрации результатов.

Репозиторий не является частью Android Architecture Components или частью Android. Это абстракция, скрывающая реализацию доступа к источнику данных. Такой способ является полезным и очень популярным, потому что в этом случае вы жестко не завязаны на какую-то конкретную реализацию источника данных. С помощью репозитория разработанный код (например код из Activity или Fragment) не знает к какому источнику данных он обращается. Например, вам нужно получить список слов. Вы можете получить данные из сети, из файла или из локальной базы данных. Но, если вы не используете паттерн репозиторий, то каждый раз, когда у вас меняется источник данных, вам необходимо переписывать и менять код в вашем Activity или Fragment-е. Используя же абстракцию в виде репозитория вы обращаетесь только к классу, который реализует интерфейс репозитория и детали скрыты. Таким образом ваш код устойчив к изменениям, является более гибким и соответствует принципу single responsibility. Схема работы паттерна Repository продемонстрирована на рисунках 12 и 13.

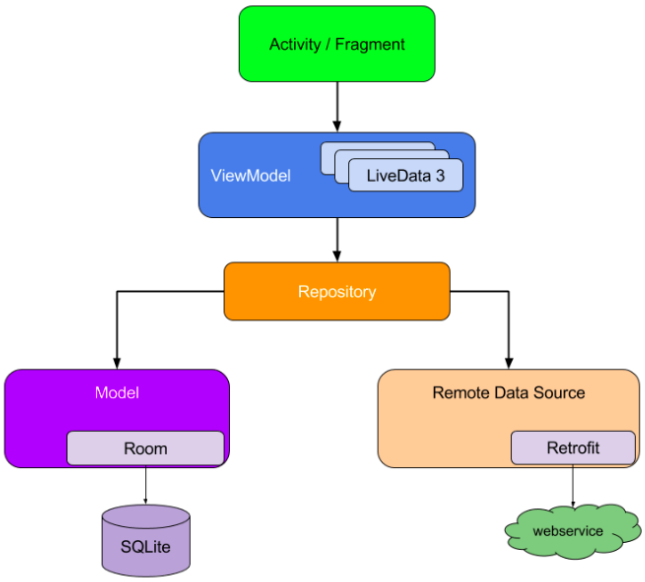


Рисунок 12 – Использование паттерна Repository совместно с паттерном MVVM и двумя источниками данных

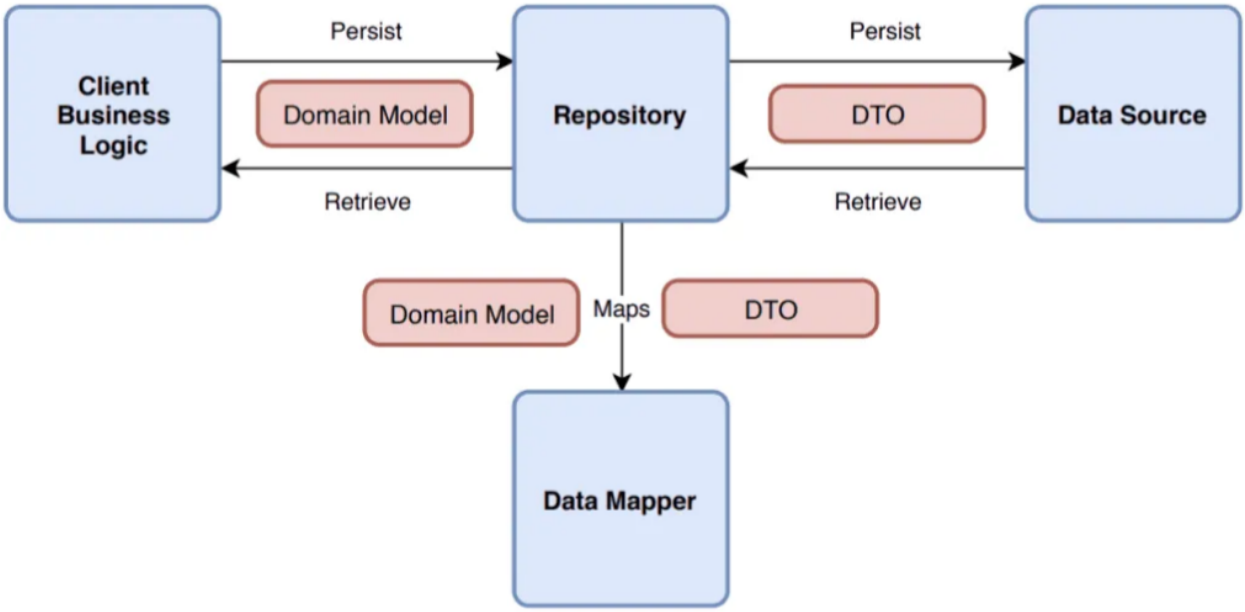


Рисунок 13 – Использование паттерна Repositoty с Data Mapper-ом

## Задание на практическую работу:

1. Обновить структуру разработанного на прошлых практических работах мобильного приложения по выбранной предметной области на основе общих рекомендаций к архитектуре мобильных приложений,
    1.1. Реализовать или реструктурировать уже реализованные файлы с программным кодом пользовательского интерфейса м хранилищ состояния пользовательского интерфейса в подпакет UI слоя.
    1.2. Реализовать или реструктурировать уже реализованные файлы с программным кодом моделей данных и источниками данных в подпакет слоя данных.
    1.3. Реализовать при необходимости предметной области или реструктурировать уже реализованные файлы с программным кодом переиспользуемой бизнес-логики в подпакет доменного слоя.
2. Реализовать архитектуру разрабатываемого приложения по паттерну MVVM с использованием компонентов LiveData в элементах паттерна.
    2.1. Реализовать по предметной области модели данных, используемых в приложении.
    2.2. Реализовать по предметной области и/или реструктуризировать ранее разработанные классы пользовательского интерфейса в компоненты View паттерна MVVM. В приложении должно быть не менее 5 отдельных компонентов View, обусловленных предметной областью и имеющей логическую и смысловую нагрузку, связанных между собой данными и навигационными переходами. Обязательной парой таких компонентов должны быть выполненные в практической работе №4 экран со списком элементов и экран с демонстрацией информации элемента списка.
    2.3. Реализовать по предметной области делегаты ViewModel, необходимые для работы компонентов View в разрабатываемом приложении.
    2.4. Реализовать по предметной области Repository с несколькими простейшими DataSource классами. Данные в DataSource-классах могут храниться в виде массивов и ассоциативных массивов.