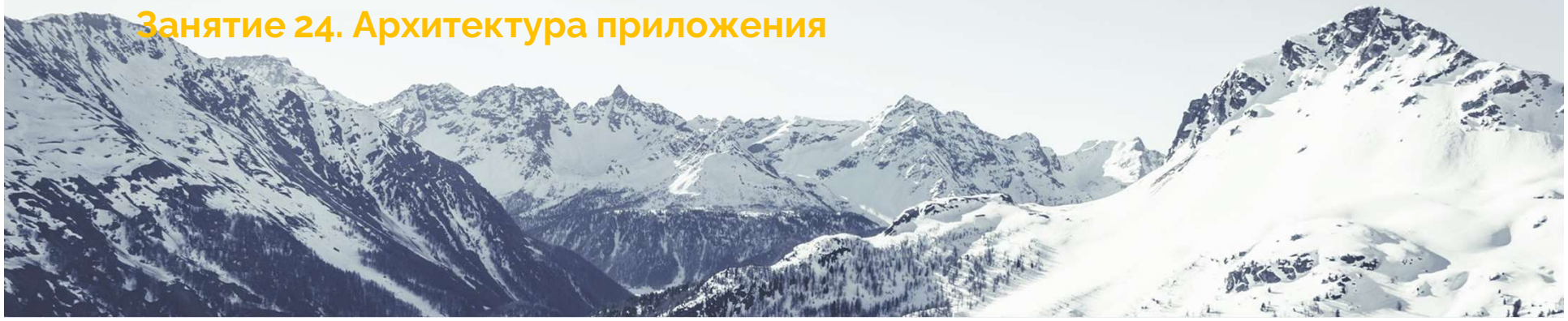

<TeachMeSkills/>

курс
Android разработчик

Занятие 24. Архитектура приложения



Агенда занятия

MVVM/P/C/I

MVVM + LiveData

Clean architecture

▶ MVVM/P/C/I

MVVM + LiveData

Clean architecture

Архитектура приложения

Архитектура программного обеспечения — это набор ключевых решений об организации программной системы. Это как чертеж, который показывает главные части программы, как они работают вместе и правила их соединения. Включает в себя основные компоненты системы, их взаимодействие и принципы, по которым они соединяются. это план или схема, которая определяет, как будет устроена программная система. Она служит фундаментом для разработки, помогает решать проблемы бизнеса в масштабируемости, надежности и удобстве сопровождения, переиспользование кода.

Архитектура UI слоя и MV*

MVVM, MVC, MVP и MVI — это архитектурные паттерны, которые используются для разделения UI слоя от бизнес логики в приложениях. Они помогают разделить обязанности между различными компонентами приложения, улучшая его структурированность, тестируемость и поддерживаемость.

MVC

MVC — Model View Controller. Архитектуру MVC можно охарактеризовать двумя пунктами:
- Model (Модель): Управляет данными и бизнес-логикой. Она отвечает за получение данных и их обработку.
- View (Представление): Отвечает за отображение данных пользователю. Оно получает обновления от контроллера. Представление — это визуальная проекция модели.
- Controller (Контроллер): Посредник между Model и View. Обрабатывает пользовательский ввод, обновляет модель и изменяет представление. Это соединение между пользователем и системой
Главная идея MVC — это то, что представление полностью управляется контроллером. Помимо этого, код четко разделен на два уровня: бизнес логику и логику отображения:
Бизнес логика — то, как работает приложение
Логика отображения — то, как выглядит приложение

MVC

MVC Passive View

Характеризующий элемент:
Представление, полностью управляемое контроллером

Идея:
Разделение между бизнес-логикой,
и логикой отображения

Бизнес-логика - это бизнес правила, которые
реализует приложение. Как приложение работает.
Реализована в контроллерах.

Логика отображения - как приложение выглядит,
какие цвета и текст будут отображены.
Реализовано в слое представления

MVC

Преимущества:

- Простота и интуитивная понятность.
- Хорошо подходит для небольших и средних приложений.

Недостатки:

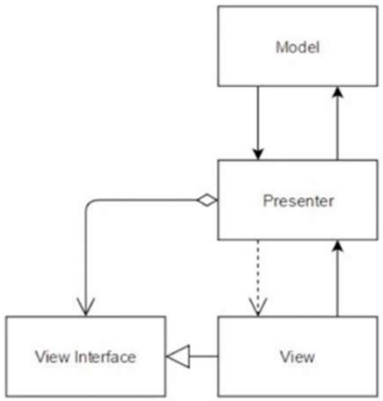
- Сложности с тестированием, так как View и Controller часто тесно связаны.
- Может привести к монолитному контроллеру, который сложно поддерживать.
- Практически не осталось реальных проектов использующих этот паттерн

MVP

MVP — Model View Presenter.
MVP является эволюцией более простого паттерна MVC (Model-View-Controller), который разделяет ответственность между тремя компонентами:
• Model — управляет данными и бизнес-логикой приложения.
• View — отвечает за отображение данных и взаимодействие с пользователем.
• Presenter — служит посредником между Model и View, обрабатывая данные и передавая их для отображения.
В отличие от MVC, в MVP роль View сводится только к отображению данных, а весь код бизнес-логики и работы с данными перемещается в Presenter. Это делает MVP более гибким и позволяет разделить задачи, что улучшает тестируемость и читабельность кода.

MVP

MVP



Характеризующий элемент:
View закрыт за интерфесом

Идея:
Сделать Presenter независимым от View

+ Presenter очень легко тестируется
+ Presenter можно повторно использовать
+ Presenter'ы могу использоваться в общих модулях
- Необходимо создавать и поддерживать интерфейсы для представлений (Views)
- Лишний шаблонный код

MVP

Преимущества MVP:

• Четкое разделение ответственности. Presenter полностью абстрагирует View от бизнес-логики.

• Тестируемость. Presenter можно легко тестировать, поскольку он не связан напрямую с UI.

• Упрощение View. UI-компоненты (Activity/Fragment) становятся проще и содержат только код, связанный с отображением данных.

Недостатки MVP:

• Увеличение количества кода. Разделение логики между View и Presenter может привести к дублированию кода и необходимости создания множества классов.

• Трудности с поддержкой сложного UI. При большом количестве взаимодействий с пользователем Presenter может стать перегруженным.

Проектов использующих этот паттерн осталось не так много.

MVP

```kotlin
// User.kt
data class User(val id: Int, val name: String, val email: String)

// UserModel.kt
class UserModel {

    private val users = listOf(
        User( id: 1,  name: "John Doe",  email: "john.doe@example.com"),
        User( id: 2,  name: "Jane Smith",  email: "jane.smith@example.com")
    )

    fun getUserById(userId: Int): User? {
        return users.firstOrNull { it.id == userId }
    }
}
```

```kotlin
// UserView.kt
interface UserView {
    fun displayUser(user: User)
    fun showError(message: String)
}

// UserPresenter.kt
class UserPresenter(private val view: UserView, private val model: UserModel) {

    fun loadUser(userId: Int) {
        val user = model.getUserById(userId)
        if (user != null) {
            view.displayUser(user)
        } else {
            view.showError("User not found")
        }
    }
}
```

MVP

```
class UserActivity : AppCompatActivity(), UserView {

    private lateinit var userNameTextView: TextView
    private lateinit var userEmailTextView: TextView
    private lateinit var loadUserButton: Button

    private lateinit var userPresenter: UserPresenter

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_user)

        userNameTextView = findViewById(R.id.userName)
        userEmailTextView = findViewById(R.id.userEmail)
        loadUserButton = findViewById(R.id.loadUserButton)

        userPresenter = UserPresenter( view: this, UserModel())

        loadUserButton.setOnClickListener {
            userPresenter.loadUser( userId: 1) // Загрузка пользователя с ID = 1
        }
    }

    override fun displayUser(user: User) {
        userNameTextView.text = user.name
        userEmailTextView.text = user.email
    }

    override fun showError(message: String) {
        userNameTextView.text = message
        userEmailTextView.text = ""
    }
}
```

MVVM

MVVM расширяет идеи MVP, вводя новый компонент — ViewModel. Это промежуточный слой между Model и View, который позволяет динамически обновлять UI в зависимости от изменений в данных.

• Model — управляет данными и бизнес-логикой.

• View — отображает данные и реагирует на взаимодействие с пользователем.

• ViewModel — содержит логику, которая соединяет Model и View, но при этом не знает о существовании конкретных элементов интерфейса.

Ключевым преимуществом MVVM является привязка данных (Data Binding), которая позволяет автоматически обновлять UI при изменении данных в ViewModel.

MVVM

Преимущества MVVM:

• Упрощение синхронизации данных. Благодаря двустороннему связыванию данных, View автоматически обновляется при изменении данных в ViewModel.

• Легкость тестирования. Как и в MVP, ViewModel не содержит ссылок на View, что облегчает его тестирование.

• Снижение зависимости UI от логики. Логика отображения и бизнес-логика разделены, что упрощает поддержку.

• Рекомендованный подход Google. Используется в большинстве проектов.

Недостатки MVVM:

• Может быть сложным для понимания и реализации для новичков.

• Привязка данных может привести к затруднениям в отладке.

MVVM


Model
Business Logic
and data
Processed
Data
Data
Changed
ViewModel
Presentation
Logic
Command and
Data Binding
View
Presentation
XAML

MVI

MVI — это более современный подход, который основывается на идее потоков данных и реактивного программирования. Этот паттерн предусматривает односторонний поток данных.

• Model — отвечает за данные и бизнес-логику.

• View — отвечает за отображение состояния.

• Intent — представляет собой действия пользователя или события, которые изменяют состояние приложения.

В MVI взаимодействие строится вокруг одного источника истины — состояния (State). Вся логика взаимодействия заключается в том, что View отображает текущее состояние, а Intent вызывает изменения этого состояния.

MVI

Преимущества MVI:

• Простота управления состоянием. MVI предлагает четкую и понятную модель управления состояниями приложения.

• Однозначность потоков данных. Все данные передаются в одном направлении, что исключает случайные ошибки при обработке событий.

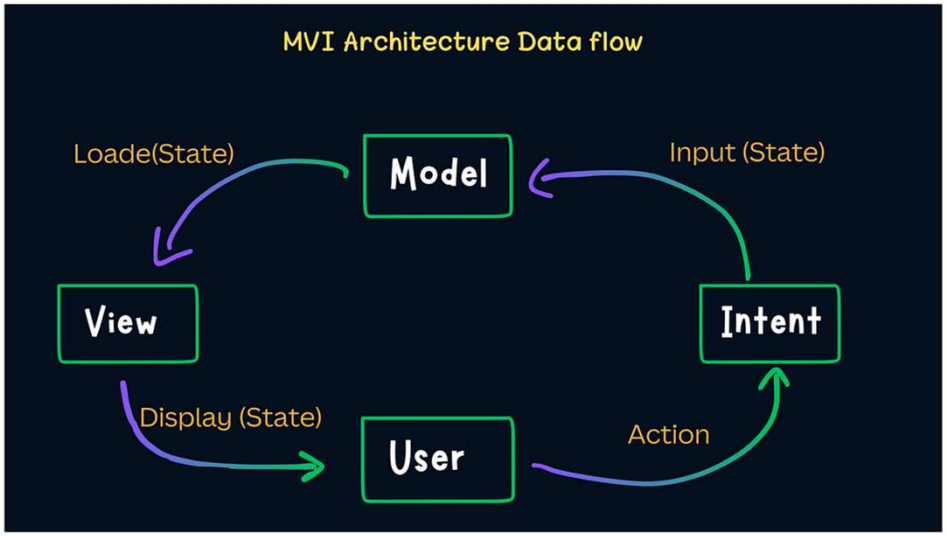
• Новый подход, постепенно заменяет MVVM в приложениях, чаще используется с Jetpack Compose.

Недостатки MVI:

• Сложность. Внедрение MVI может быть сложным для разработчиков, не знакомых с реактивным программированием.

• Много boilerplate-кода. Для каждого изменения состояния и UI требуется создавать дополнительные классы и интерфейсы.

MVI


MVI Architecture Data flow
Loade(State)
Model
Input (State)
View
Intent
Display (State)
User
Action

Какой выбрать?

Современные проекты чаще всего отдают предпочтение MVVM и гугл рекомендует его. Однако в комбинации с Jetpack Compose и не только выбор все чаще падает на MVI. В больших и старых проектах редко можно встретить MVP в комбинации с библиотекой Moxy и без. MVC же застать практически невозможно, разве что в Legacy но вряд ли над таким проектом все еще ведется разработка.

MVVM/P/C/I

▶ MVVM + LiveData

Clean architecture

Что важно знать о ViewModel

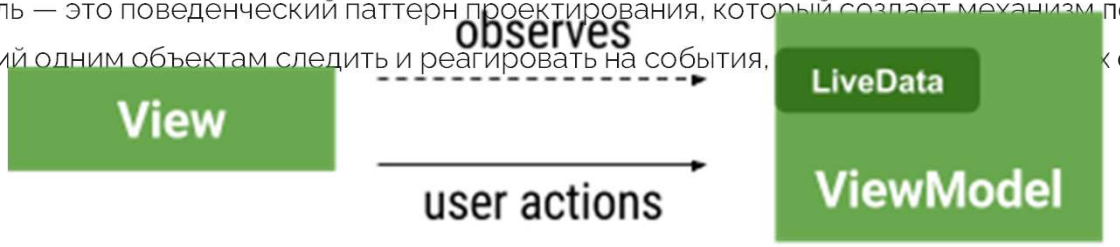
• Жизненный цикл ViewModels отличается от жизненного цикла активити или фрагмента. Тем временем, как ViewModel инициализирована и работает, активити может пройти через несколько состояний жизненного цикла. ViewModel может ничего не знать о том, что активити и фрагменты были уничтожены и созданы.

• ViewModels сохраняется при изменениях конфигурации

• Передача ссылки на View (активити или фрагмент) в ViewModel является серьезным риском. Предположим, ViewModel запросила данные из сети, и они придут чуть позже. В этот момент ссылка на View может быть уничтожена или активити может больше не отображаться — это приведет к утечке памяти, а возможно и к крэшу приложения.

Observer Pattern (шаблон проектирования «Наблюдатель»)

Рекомендуемый способ коммуникации между ViewModel и View — использование observer pattern, при помощи LiveData или observable из других библиотек.
Наблюдатель — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, [illegible] объектах.



Observer Pattern (шаблон проектирования «Наблюдатель»)

Удобным способом дизайна презентационного слоя в Android является, когда View (активити или фрагмент) наблюдает (подписана на изменения в) ViewModel. Т.к. ViewModel не знает ничего про Android, она также не знает о том, как часто Android убивает View. У этого подхода есть несколько преимуществ:

• ViewModel сохраняются при изменении конфигурации, поэтому нет нужды в повторном запросе внешних источников данных (к примеру, базы данных или сетевого хранилища) при произведенном повороте устройства.

• Когда заканчивается какая-либо длительная операция, observable в ViewModel обновляются. Не важно, велось ли наблюдение за данными или нет. NPE не произойдет даже при попытке обновления несуществующего View.

• ViewModel не содержат ссылки на View, что снижает риск возникновения утечек памяти.

LiveData

• LiveData — это компонент библиотеки Android Architecture Components, который представляет собой наблюдаемый контейнер данных.
• Он позволяет компонентам UI подписываться на изменения данных и автоматически обновляться, когда данные изменяются.
• LiveData учитывает жизненный цикл компонентов, что предотвращает утечки памяти и ошибки, связанные с жизненным циклом.

Пример

```kotlin
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import com.study.android.myapplication.domain.CatModel

class MainActivityViewModel : ViewModel() {

    private val _cats = MutableLiveData<CatModel>()
    val cats: LiveData<CatModel> get() = _cats

    fun loadUser(catId: String) {
        // Симуляция загрузки данных
        _cats.value = CatModel(catId, imageUrl: "https://teachmeskills.ru/", isLiked: true)
    }
}
```

```kotlin
import ...

class MainActivity : AppCompatActivity() {
    private val catsViewModel: MainActivityViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContentView(R.layout.activity_main)
        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v, insets ->
            val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())
            v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom
            insets
        }
        // Подписка на изменения данных
        catsViewModel.cats.observe( owner: this, Observer { cat ->
            // Обновление UI
            findViewById<TextView>(R.id.cats).text = cat?.id
        })

        // Загрузка данных
        catsViewModel.loadUser( catId: "1")
    }
}
```

Пример работы

Смотрим проект. Git

Обработка нажатия на элемент RecyclerView

Эффективное обновление списка

▶ Clean architecture

Clean architecture

Clean Architecture — это архитектурный подход, предложенный Робертом С. Мартином (дядей Бобом), который помогает разработчикам создавать приложения с четким разделением обязанностей и высокой степенью независимости от фреймворков, баз данных, UI и других внешних систем. Основная цель Clean Architecture — обеспечить простоту изменения и тестирования кода, а также его долговечность.

Основные идеи Clean Architecture

Независимость от фреймворков:
• Архитектура не должна быть зависимой от конкретных фреймворков или библиотек. Это позволяет легко заменять технологический стек без значительных изменений в бизнес-логике.
Тестируемость:
• Бизнес-логика должна быть легко тестируемой и независимо от внешних систем, что позволяет писать модульные тесты без необходимости в сложных моках.
Независимость от UI:
• Изменения в пользовательском интерфейсе не должны влиять на бизнес-логику приложения.
Независимость от базы данных:
• Бизнес-логика не должна зависеть от способа хранения данных. Это позволяет легко менять способ хранения данных без изменений в бизнес-логике.
Независимость от внешних систем:
• Внешние интерфейсы и API должны быть легко заменяемыми.

Слои Clean Architecture

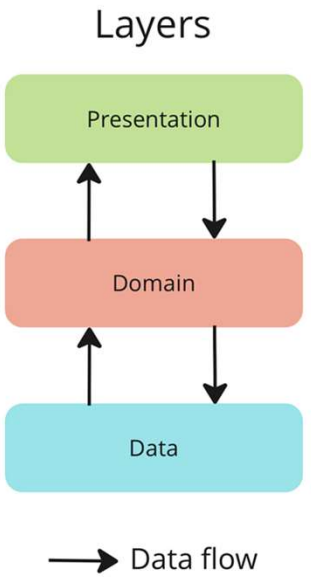
• Представление (Presentation Layer): Этот слой отвечает за отображение данных пользователю и обработку пользовательского ввода. Он содержит элементы интерфейса пользователя (Activity, Fragment) и презентеры (Presenter), которые отвечают за взаимодействие с остальными слоями.

• Бизнес-логика (Domain Layer): Этот слой содержит основную бизнес-логику приложения. Он определяет модели данных, правила бизнес-логики и интеракторы (Interactors), которые обрабатывают запросы из представления.

• Хранилище данных (Data Layer): В этом слое находится код для доступа к данным, таким как базы данных или API. Репозитории (Repositories) отвечают за извлечение и сохранение данных.

Слои Clean Architecture

Presentation: отвечает за отображение пользовательского интерфейса и реагирование на его события
Он содержит элементы интерфейса пользователя (Activity, Fragment) и презентеры(Presenter)/ViewModel./Controller, которые отвечают за взаимодействие с остальными слоями
Domain: бизнес-логика, изолированная от деталей реализации, определяет правила и операции, как приложение должно взаимодействовать с данными
Data: в этом слое находится код для доступа к данным, таким как базы данных или API. Репозитории (Repositories) отвечают за извлечение и сохранение данных.



Компоненты слоя Domain

- Entity: представляет объекты данных, с которыми работает бизнес-логика
- Repository: обеспечивает контракт для доступа к данным

```kotlin
package com.study.android.myapplication.domain

// *Model = Entity в Clean Architecture
data class CatModel(
    val id: String,
    val imageUrl: String,
    val isLiked: Boolean,
)
```

```kotlin
package com.study.android.myapplication.domain

interface CatRepository {
    suspend fun getCats(limit: Int): List<CatModel>
    suspend fun setLike(value: Boolean, id: String)
}
```

Компоненты слоя Domain

Interactor или UseCase: содержит функциональность или операции

В контексте чистой архитектуры, термины "Interactor" и "Use Case" часто используются почти взаимозаменяемо, хотя в некоторых случаях могут быть небольшие различия в их восприятии или использовании.
UseCase: представляет собой конкретную операцию
Interactor: является комплексом операций над сущностью

```kotlin
interface GetCatsUseCase {
    fun invoke(): List<CatModel>
}

class GetCatsUseCaseImpl: GetCatsUseCase{
    override fun invoke(): List<CatModel> {
        TODO( reason: "Not yet implemented")
    }
}
```

odel.kt | CatRepository.kt | CatInteractor.kt

```kotlin
package com.study.android.myapplication.domain

interface CatInteractor {
    fun getAllCats()
    fun setLike()
    fun getLikedCats()
    fun getNotLikedCats()
}

class CatInteractorImpl : CatInteractor {
    override fun getAllCats() {
        TODO( reason: "Not yet implemented")
    }

    override fun setLike() {
        TODO( reason: "Not yet implemented")
    }

    override fun getLikedCats() {
        TODO( reason: "Not yet implemented")
    }

    override fun getNotLikedCats() {
        TODO( reason: "Not yet implemented")
    }
}
```

Компоненты слоя Data

Базовыми компонентами данного модуля являются:
• Repository Implementation: реализованные контракты репозиториев, которые мы определили в domain модуле
• Data Sources: конкретные реализации, обеспечивающие доступ к данным из различных источников (база данных, API)
• Data Models: объекты данных, представляющие хранимую информацию. Включает в себя DTO (Data Transfer Objects), объекты для операций CRUD (Create, Read, Update, Delete), а также объекты запросов (Query) и т.д
• Mappers: преобразование объектов данных, к примеру в Entity

Компоненты слоя Data

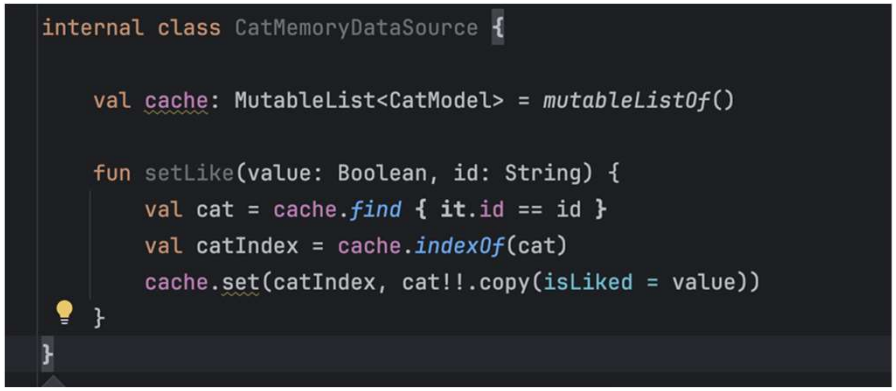
• Data Models: объекты данных, представляющие хранимую информацию. Включает в себя DTO (Data Transfer Objects), объекты для операций CRUD (Create, Read, Update, Delete), а также объекты запросов (Query) и т.д
• Mappers: преобразование объектов данных, к примеру в Entity

```
1  package com.study.android.myapplication.data
2
3
4  internal data class CatResponseModel(
5      val id: String,
6      val url: String,
7  )
```

```
package com.study.android.myapplication.data

import com.study.android.myapplication.domain.CatModel

internal fun CatResponseModel.mapToDomainModel(isLiked: Boolean) =
    CatModel(id, url, isLiked)

internal fun List<CatResponseModel>.mapToDomainModels(
    isLikedGetter: (String) -> Boolean,
) = map { it.mapToDomainModel(isLiked = isLikedGetter(it.id)) }
```

Компоненты слоя Data

• Data Sources: конкретные реализации, обеспечивающие доступ к данным из различных источников (база данных, API)
Они могут быть разделены на три типа:
Remote: обращения к внешнему API для получения данных
Memory: взаимодействие с данными, хранящимися в оперативной памяти
Local: чтение данных из локальной базы данных



Компоненты слоя Data

• Repository Implementation: реализованные контракты репозиториев, которые мы определили в domain модуле

```kotlin
internal class CatRepositoryImpl(
    private val catRemoteDataSource: CatRemoteDataSource,
    private val catMemoryDataSource: CatMemoryDataSource,
) : CatRepository {

    override suspend fun getCats(limit: Int): List<CatModel> {
        val cats = catRemoteDataSource.getCats(limit)
            .mapToDomainModels(isLikedGetter = { catMemoryDataSource.getLike(it) })
        catMemoryDataSource.save(cats)
        return cats
    }

    override suspend fun setLike(value: Boolean, id: String) =
        catMemoryDataSource.setLike(value, id)
}
```

Clean Architecture

Смотрим проект. Git

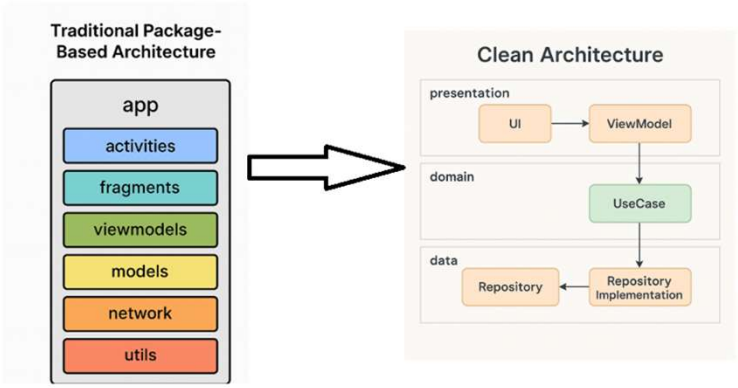
Clean Architecture недостатки

Clean Architecture не всегда стоит применять. Например для следующих типов проектов:

• В маленьких проектах или прототипах, где важна скорость разработки, а не гибкость и расширяемость.

• В проектах с очень узким функционалом, где количество логики невелико, и разделение на слои только усложнит архитектуру.

Clean Architecture vs Традиционное разде



Clean Architecture vs Традиционное разделение по пакетам

Плюсы обычного разделения по пакетам:

- Легко начать и понять.
- Хорошо подходит для небольших проектов.

Минусы:

- Сложнее тестировать (тесная связь между слоями).
- Трудно масштабировать (добавление новых функций становится запутанным).
- Нет четкого разделения обязанностей.
- Clean Architecture используется больше чем в 90% проектов, и онбординг новых разработчиков за счет этого ускоряется

Практика

Задачи

Задача 1: Счетчик кликов

Создайте простое приложение с кнопкой и текстовым полем, которое отображает количество нажатий на кнопку. Используйте MVVM и LiveData для обновления счетчика в пользовательском интерфейсе.

Задачи

Задача 2: Фильтрация списка

Создайте приложение, которое отображает список пользователей. Пользователь может ввести имя в текстовое поле для фильтрации списка по нажатию кнопки "Отфильтровать". Используйте MVVM и LiveData для управления состоянием списка и фильтрацией.

Задачи

Задача 3: Перенесите логику хранения списка по правилам Clean Architecture

ViewModel должно лишь реагировать на действия пользователя и обновлять LiveData

Задачи

Задача 4: Таймер обратного отсчета

Создайте приложение с таймером обратного отсчета, который начинает отсчет при нажатии кнопки "Старт". Таймер должен обновлять интерфейс каждую секунду через LiveData.

Q&A

Домашнее задание

Задачи

Задача 1: Clean Architecture + MVVM

Реализуйте экран, отображающий список заметок и дату их добавления.
Хранение заметок реализовать в файле или в переменной.
Над списком должно быть текстовое поле и кнопка “Добавить”.
При сохранении заметка должна сохраняться в хранилище и отображаться в списке.
Используйте правила Clean Architecture, MVVM + LiveData

Задачи

Задача 2*: Удаление заметки из списка

Реализуйте удаление заметки по нажатию на нее или кнопку рядом с ней.

** Добавьте возможность редактирования заметки

Q&A

Ваши вопросы


Спасибо
TeachMeSkills