Ссылка на видео этого руководства на сайте: borisproit.expert

🟢 БАЗА (Junior)

Что такое архитектура в приложении? What is application architecture?

Архитектура — это то, как устроено приложение внутри.
То есть как разложена логика по разным частям и как эти части между собой общаются.

Она нужна, чтобы:

— код не превращался в хаос
— было понятно, где что менять
— можно было легко добавлять новые функции
— разные части приложения не мешали друг другу

Проще говоря — это как план дома.
Если всё продумано, жить удобно. Если нет — всё начинает ломаться.

Architecture is how the app is structured inside.
It defines how different parts are organized and how they talk to each other.

It helps to:

— keep the code from becoming messy
— understand where to change things
— easily add new features
— prevent parts of the app from interfering with each other

In simple words — it’s like a house blueprint.
With a good plan, everything works smoothly.

Зачем разделять код на слои? Why do we separate code into layers?

Разделение на слои нужно, чтобы каждая часть приложения делала только свою работу.

Тогда:

— код становится понятнее
— проще что-то менять, не ломая всё остальное
— легче тестировать
— новые функции добавляются без хаоса

Ссылка на видео этого руководства на сайте: borisproit.expert

Например, если поменяется источник данных (сеть → база),
UI вообще не должен об этом знать.

Проще говоря — чтобы не было "одной огромной функции, которая делает всё".

We separate code into layers so each part of the app has its own responsibility.

This helps to:

— keep code easier to understand
— change things without breaking everything
— test logic more easily
— add new features without chaos

For example, if the data source changes (network → database),
the UI should not care.

In simple terms — to avoid having “one giant function that does everything”.

Что такое MVVM? What is MVVM?

MVVM — это архитектурный подход, который помогает разделить приложение на понятные части.

Он состоит из трёх ролей:

— **Model** — данные (сервер, база, репозиторий)
— **View** — экран (Activity / Fragment / Compose UI)
— **ViewModel** — посредник между ними

View показывает данные.
Model хранит данные.
ViewModel связывает их и решает, что показывать.

Главная идея — экран не должен напрямую работать с данными.

MVVM is an architecture pattern that separates an app into clear parts.

It has three roles:

— **Model** — data (API, database, repository)
— **View** — the screen (Activity / Fragment / UI)
— **ViewModel** — the middle layer between them

Ссылка на видео этого руководства на сайте: borisproit.expert

View displays data.
Model provides data.
ViewModel connects them and decides what should be shown.

Main idea — the UI should not talk to data directly.

Чем MVVM отличается от MVC? What is the difference between MVVM and MVC?

Главное отличие — куда уходит логика.

В **MVC**:

— View напрямую общается с Model
— Controller управляет логикой

Но на практике Controller часто разрастается и превращается в “God object”.

В **MVVM**:

— View не знает про Model
— вся логика находится в ViewModel
— View просто наблюдает и показывает данные

То есть MVVM лучше разделяет ответственность
и делает UI менее зависимым от данных.

The main difference is where the logic lives.

In **MVC**:

— View talks directly to Model
— Controller handles logic

But in reality, the Controller often grows too big.

In **MVVM**:

— View does not know about Model
— all logic lives in ViewModel
— View only observes and displays data

So MVVM provides better separation
and makes UI less dependent on data.

Ссылка на видео этого руководства на сайте: borisproit.expert

Что делает View? What does View do?

View — это то, что видит пользователь.

Она:

— показывает данные
— принимает действия пользователя (клики, ввод)
— передаёт события дальше

Важно: View **не должна содержать бизнес-логику**.
Её задача — только отображать.

Проще говоря — View ничего “не решает”, она просто показывает.

View is what the user sees.

It:

— displays data
— handles user actions (clicks, input)
— passes events forward

Important: View should **not contain business logic**.
Its job is only to display.

In simple terms — View does not “decide”, it just shows.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// View (Activity)
class MainActivity(private val viewModel: UserViewModel) {

    fun onButtonClick() {
        val name = viewModel.getUserName()
        showName(name)
    }

    fun showName(name: String) {
        println(name) // отображаем пользователю
    }
}
```

Что делает ViewModel? What does ViewModel do?

ViewModel — это посредник между экраном и данными.

Она:

— получает данные
— обрабатывает их
— решает, что показать
— отдаёт готовый результат View

ViewModel содержит логику, но не знает ничего про UI.

Проще говоря — ViewModel “думает”, а View просто показывает.

ViewModel is the middle layer between UI and data.

It:

— gets data
— processes it
— decides what should be shown
— provides the result to the View

ViewModel contains logic, but does not know about UI.

Ссылка на видео этого руководства на сайте: borisproit.expert

In simple terms — ViewModel “thinks”, View just displays.

```
// Model
class UserRepository {
    fun getUser(): String = "John"
}

// ViewModel
class UserViewModel(private val repository: UserRepository) {

    fun getUserName(): String {
        val user = repository.getUser()
        return user.uppercase() // логика обработки
    }
}
```

Может ли ViewModel работать без UI? Can ViewModel work without UI?

Да, может.

ViewModel не зависит от экрана.
Она не знает ничего про Activity, Fragment или Compose.

Она просто:

— получает данные
— обрабатывает их
— отдаёт результат

Поэтому её можно использовать:

— в тестах
— в фоновой логике
— даже в другом приложении

Проще говоря — ViewModel умеет работать сама по себе.

Yes, it can.

Ссылка на видео этого руководства на сайте: borisproit.expert

ViewModel does not depend on UI.
It does not know about Activity, Fragment, or Compose.

It simply:

— gets data
— processes it
— provides the result

That's why it can be used:

— in tests
— in background logic
— even outside UI

In simple terms — ViewModel can work on its own.

Где должна жить логика загрузки данных? Where should data loading logic live?

Логика загрузки данных должна жить не во View, а во ViewModel (или UseCase / Repository).

Экран не должен сам ходить в сеть или базу.

Его задача — только сказать:
👉 "нужно загрузить данные"

А уже ViewModel решает:

— откуда брать данные
— когда их загружать
— что показать

Data loading logic should not live in the View.
It should live in ViewModel (or UseCase / Repository).

The screen should not call API or database itself.

Its job is only to say:
👉 "load the data"

Then ViewModel decides:

— where to get data
— when to load it

Ссылка на видео этого руководства на сайте: borisproit.expert

— what to show

Что такое Repository? What is Repository?

Repository — это слой, который отвечает за получение данных.

Он знает:

— брать данные из сети
— брать из базы
— брать из кэша

И решает, откуда именно их взять.

Для ViewModel это просто “источник данных”.

Проще говоря — Repository прячет всю работу с данными внутри себя.

Repository is a layer that provides data.

It knows how to:

— get data from network
— get data from database
— get data from cache

And decides where to take it from.

For ViewModel, it’s just a data source.

In simple terms — Repository hides all data handling inside.

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
// Repository решает откуда взять данные
class UserRepository {

    fun getUser(): String {
        val fromCache = null

        return fromCache ?: "User from API"
    }
}

// ViewModel просто просит данные
class UserViewModel(private val repository: UserRepository) {

    fun loadUser(): String {
        return repository.getUser()
    }
}
```

Кто вызывает Repository? Who calls Repository?

Repository вызывает не экран, а ViewModel.

Экран говорит:
👉 “мне нужны данные”

ViewModel решает:
👉 “ок, беру их из Repository”

То есть:

View → ViewModel → Repository

Экран не должен напрямую обращаться к данным.

Repository is called by ViewModel, not by the UI.

Ссылка на видео этого руководства на сайте: borisproit.expert

The screen says:
👉 “I need data”

ViewModel decides:
👉 “ok, I’ll get it from Repository”

So the flow is:

View → ViewModel → Repository

UI should not talk to data directly.

```
// Repository
class UserRepository {
    fun getUser(): String = "John"
}

// ViewModel вызывает Repository
class UserViewModel(private val repository: UserRepository) {

    fun loadUser(): String {
        return repository.getUser()
    }
}

// View вызывает ViewModel
class MainActivity(private val viewModel: UserViewModel) {

    fun onScreenOpened() {
        val user = viewModel.loadUser()
        println(user)
    }
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Что такое состояние экрана? What is screen state?

Состояние экрана — это то, что сейчас происходит на экране.

Например:

— идёт загрузка
— есть данные
— произошла ошибка
— экран пустой

То есть это “текущая ситуация”, которую должен показать UI.

View не придумывает состояние сама —
его формирует ViewModel.

Screen state is what is currently happening on the screen.

For example:

— loading
— data is shown
— error happened
— empty screen

It’s the “current situation” that UI should display.

View does not create it —
ViewModel provides it.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Possible UI states
sealed class UserState {
    object Loading : UserState()
    data class Success(val name: String) : UserState()
    object Error : UserState()
}

// ViewModel controls state
class UserViewModel(private val repository: UserRepository) {

    fun loadUser(): UserState {
        return try {
            UserState.Success(repository.getUser())
        } catch (e: Exception) {
            UserState.Error
        }
    }
}
```

Где должен храниться UI state? Where should UI state live?

UI state должен храниться во ViewModel.

Не во View.

Потому что:

— экран может пересоздаваться (поворот, пересоздание)
— логика состояния не должна зависеть от UI
— данные не должны теряться

View просто отображает состояние,
а ViewModel его хранит и управляет им.

UI state should live in ViewModel.

Ссылка на видео этого руководства на сайте: borisproit.expert

Not in the View.

Because:

— UI can be recreated (rotation, etc.)
— state logic should not depend on UI
— data should not be lost

View only displays state,
ViewModel stores and manages it.

Что такое Data слой? What is Data layer?

Data слой — это часть приложения, которая отвечает за получение данных.

Он:

— работает с сетью
— работает с базой
— работает с кэшем

И отдаёт данные дальше — в ViewModel.

UI не должен знать, откуда пришли данные.
Этим занимается Data слой.

Data layer is the part of the app that handles data.

It:

— works with network
— works with database
— works with cache

And provides data to ViewModel.

UI should not know where data comes from.
That's the job of the Data layer.

Почему UI не должен зависеть от сети напрямую? Why shouldn't UI depend directly on network?

Потому что UI должен только показывать данные, а не заниматься их загрузкой.

Ссылка на видео этого руководства на сайте: borisproit.expert

Если экран напрямую ходит в сеть:

— код становится хрупким
— сложно тестировать
— при смене API нужно менять UI
— логика смешивается с отображением

UI должен быть простым — показать состояние.

А вся работа с сетью должна быть спрятана в Repository.

Because UI should only display data, not load it.

If UI calls the network directly:

— code becomes fragile
— testing becomes harder
— UI must change if API changes
— logic mixes with presentation

UI should stay simple — just show state.

All network work should live in Repository.

Что такое ответственность класса? What is class responsibility?

Ответственность класса — это то, за что он отвечает.

То есть — какую одну задачу он должен выполнять.

Хороший класс:

— делает одну вещь
— понятен по названию
— не пытается решать всё сразу

Например:

Repository — отвечает за данные
ViewModel — за логику
View — за отображение

Если класс делает всё подряд — это плохой дизайн.

Ссылка на видео этого руководства на сайте: borisproit.expert

Class responsibility is what a class is responsible for.

It means — what single job it should do.

A good class:

— does one thing
— is clear by its name
— does not try to do everything

For example:

Repository — handles data
ViewModel — handles logic
View — handles UI

If a class does everything — that's bad design.

Почему лучше работать через интерфейсы? Why is it better to work through interfaces?

Потому что интерфейсы уменьшают зависимость от конкретной реализации.

Это даёт:

— гибкость
— лёгкую замену реализации
— удобное тестирование

Например, сегодня данные приходят из сети,
а завтра — из базы.

Если всё завязано на интерфейс — менять код почти не нужно.

Because interfaces reduce dependency on concrete implementations.

This gives:

— flexibility
— easy replacement
— better testing

For example, today data comes from network,
tomorrow from database.

If everything depends on an interface — changes are minimal.

Ссылка на видео этого руководства на сайте: borisproit.expert

Где должна происходить обработка ошибок? Where should error handling happen?

Обработка ошибок должна происходить не в UI, а в логике — чаще всего во ViewModel или Repository.

UI не должен разбираться в причинах ошибки.
Он должен просто показать состояние:

— ошибка
— сообщение
— retry

ViewModel решает:

— что делать при ошибке
— что показать экрану

Error handling should not happen in the UI.
It should live in logic — usually in ViewModel or Repository.

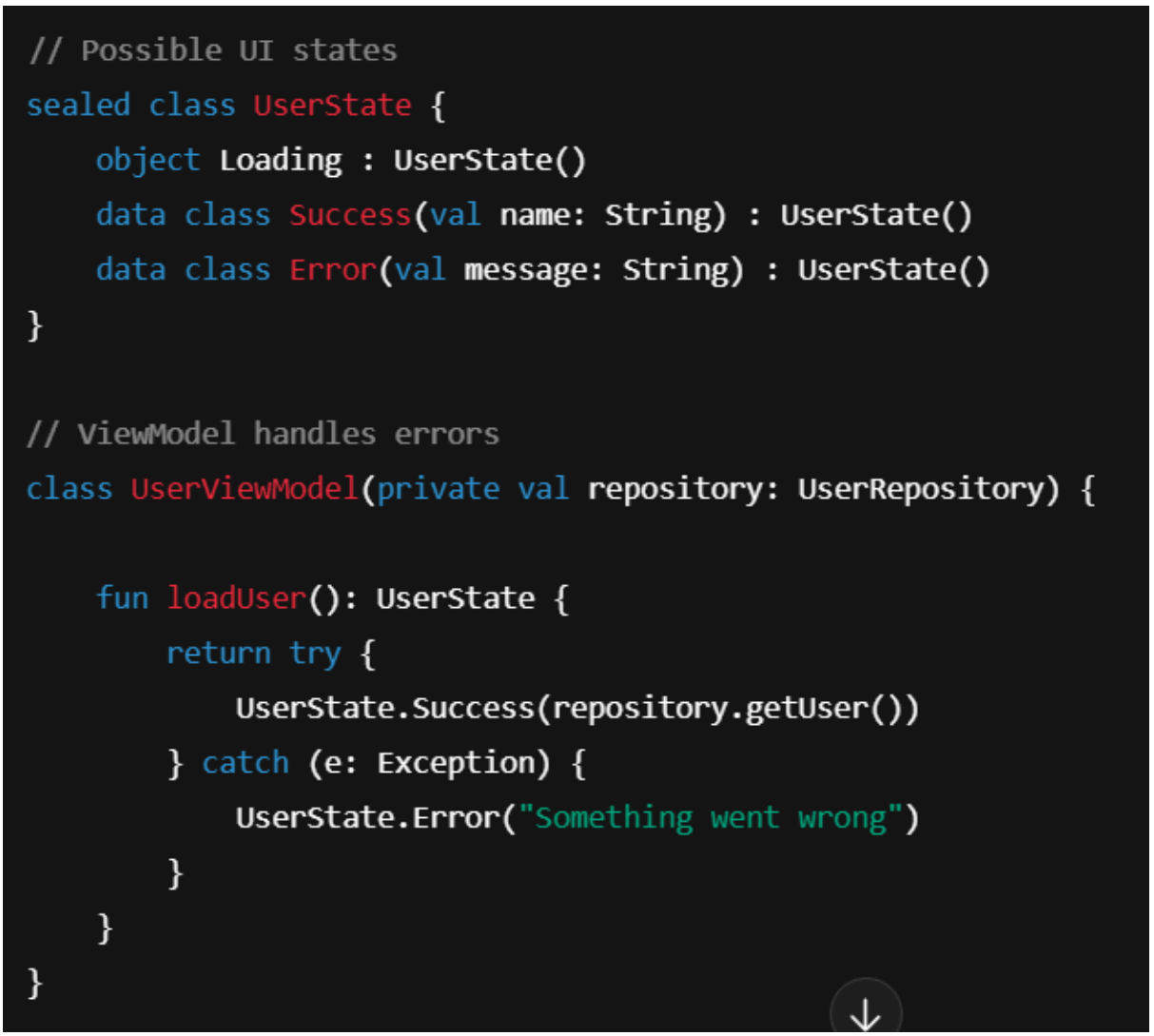
UI should not deal with why something failed.
It should only display:

— error
— message
— retry

ViewModel decides:

— what to do on failure
— what UI should show

Ссылка на видео этого руководства на сайте: borisproit.expert



```
// Possible UI states
sealed class UserState {
    object Loading : UserState()
    data class Success(val name: String) : UserState()
    data class Error(val message: String) : UserState()
}

// ViewModel handles errors
class UserViewModel(private val repository: UserRepository) {

    fun loadUser(): UserState {
        return try {
            UserState.Success(repository.getUser())
        } catch (e: Exception) {
            UserState.Error("Something went wrong")
        }
    }
}
```

Где должна происходить подготовка данных для UI? Where should data preparation for UI happen?

Подготовка данных должна происходить во ViewModel.

Не в UI.

Потому что UI должен только показывать готовый результат,
а не форматировать, фильтровать или преобразовывать данные.

ViewModel берёт “сырые” данные из Repository
и превращает их в то, что удобно отображать.

Data preparation should happen in ViewModel.

Ссылка на видео этого руководства на сайте: borisproit.expert

Not in UI.

Because UI should only display ready-to-use data,
not format or transform it.

ViewModel takes “raw” data from Repository
and converts it into something ready for display.

```
// Repository returns raw data
class UserRepository {
    fun getUser(): String = "john"
}

// ViewModel prepares data for UI
class UserViewModel(private val repository: UserRepository) {

    fun getUserName(): String {
        val raw = repository.getUser()
        return raw.uppercase() // format for UI
    }
}
```

Что такое state-driven UI? What is state-driven UI?

State-driven UI — это подход, где экран зависит только от состояния.

UI не “решает”, что показывать сам.
Он просто смотрит на текущее состояние и отображает его.

Например:

— Loading → показываем прогресс
— Success → показываем данные
— Error → показываем ошибку

То есть UI = функция от состояния.

Ссылка на видео этого руководства на сайте: borisproit.expert

State-driven UI is an approach where UI depends only on state.

UI does not decide what to show.
It just reacts to the current state.

For example:

— Loading → show progress
— Success → show data
— Error → show error

UI becomes a function of state.

Где должен храниться state в Compose? Where should state live in Compose?

State должен храниться не в UI, а во ViewModel.

Compose экран может пересоздаваться много раз,
и если state лежит внутри него — он потеряется.

ViewModel переживает пересоздание экрана,
поэтому именно там должен жить основной state.

UI просто читает его и отображает.

State should live in ViewModel, not in UI.

Compose screens can be recreated many times,
and if state is inside UI — it may be lost.

ViewModel survives recomposition,
so main state should live there.

UI just reads and displays it.

🟡 СРЕДНИЙ УРОВЕНЬ (Middle)

Какие роли у View, ViewModel и Model? What are the roles of View, ViewModel and Model?

Ссылка на видео этого руководства на сайте: borisproit.expert

View — показывает данные и принимает действия пользователя.
Ничего не решает.

ViewModel — связывает данные и экран.
Обрабатывает данные и решает, что показать.

Model — источник данных.
Сеть, база, репозиторий.

View — displays data and handles user actions.
Does not make decisions.

ViewModel — connects data and UI.
Processes data and decides what to show.

Model — data source.
Network, database, repository.

```
// Model
class UserRepository {
    fun getUser(): String = "John"
}

// ViewModel
class UserViewModel(private val repository: UserRepository) {
    fun loadUser(): String = repository.getUser()
}

// View
class MainActivity(private val viewModel: UserViewModel) {
    fun showUser() {
        println(viewModel.loadUser())
    }
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Должна ли ViewModel знать про Android framework? Should ViewModel know about Android framework?

Нет, не должна.

ViewModel должна быть независимой от Android.

Она не должна знать про:

— Context
— Activity
— Fragment
— View

Тогда её можно:

— легко тестировать
— переиспользовать
— запускать без UI

No, it should not.

ViewModel should be independent from Android.

It should not know about:

— Context
— Activity
— Fragment
— View

This makes it:

— easy to test
— reusable
— usable without UI

Чем State отличается от Event? What is the difference between State and Event?

State — это текущее состояние экрана.
Оно длительное и может быть восстановлено.

Например:

Ссылка на видео этого руководства на сайте: borisproit.expert

— Loading
— Data
— Error

Event — это разовое действие.
Оно происходит один раз и не должно повторяться.

Например:

— показать тост
— перейти на экран
— открыть диалог

State is the current screen condition.
It is long-lived and can be restored.

Examples:

— Loading
— Data
— Error

Event is a one-time action.
It should happen only once.

Examples:

— show toast
— navigate
— open dialog

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// State (persistent)
sealed class ScreenState {
    object Loading : ScreenState()
    data class Success(val name: String) : ScreenState()
}

// Event (one-time)
sealed class ScreenEvent {
    object ShowToast : ScreenEvent()
    object NavigateNext : ScreenEvent()
}
```

Почему навигацию нельзя хранить как State? Why navigation should not be stored as State?

Потому что State живёт долго и может повторяться.

А навигация — это разовое действие.

Если хранить её как State:

— при пересоздании экрана навигация может выполниться снова
— пользователь может попасть на экран повторно
— появляются баги

Навигация — это Event, а не состояние.

Because State is persistent and can repeat.

Navigation is a one-time action.

If stored as State:

— it may trigger again after recreation
— user can navigate twice
— bugs appear

Navigation is an Event, not State.

Ссылка на видео этого руководства на сайте: borisproit.expert

Как обрабатывать одноразовые события? How to handle one-time events?

Одноразовые события нужно передавать отдельно от состояния.

Не через State.

Обычно:

— через Event
— через поток
— через канал

Смысл — событие должно произойти один раз и исчезнуть.

One-time events should be sent separately from state.

Not through State.

Usually:

— via Event
— via stream
— via channel

The idea — it should happen once and disappear.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// One-time event
sealed class ScreenEvent {
    object ShowToast : ScreenEvent()
}

// ViewModel emits event
class UserViewModel {

    var event: ScreenEvent? = null
        private set

    fun onError() {
        event = ScreenEvent.ShowToast
    }
}
```

Где должен жить Repository? Where should Repository live?

Repository должен жить в Data слое.

Он не относится к UI
и не должен находиться рядом с экраном или ViewModel.

Repository — это часть работы с данными.

Repository should live in the Data layer.

It does not belong to UI
and should not be placed next to screens or ViewModel.

Repository is part of data handling.

Ссылка на видео этого руководства на сайте: borisproit.expert

Что такое UseCase? What is UseCase?

UseCase — это отдельный класс, который выполняет одну бизнес-задачу.

Например:

— загрузить пользователя
— оформить заказ
— сохранить данные

Он берёт данные из Repository
и выполняет нужную логику.

Проще говоря — это “одно конкретное действие”.

UseCase is a class that performs one business action.

For example:

— load user
— place order
— save data

It gets data from Repository
and applies business logic.

In simple terms — it’s one specific action.

```
// UseCase example

class GetUserUseCase(private val repository: UserRepository) {

    fun execute(): String {
        return repository.getUser()
    }
}
```

Где должны жить UseCases? Where should UseCases live?

Ссылка на видео этого руководства на сайте: borisproit.expert

UseCases должны жить между ViewModel и Data слоем.

Это слой бизнес-логики.

Они:

— принимают решение
— используют Repository
— возвращают готовый результат

UseCase не относится ни к UI, ни к Data напрямую.

UseCases should live between ViewModel and Data layer.

This is the business logic layer.

They:

— make decisions
— use Repository
— return ready result

UseCase belongs neither to UI nor Data directly.

Что такое DTO? What is DTO?

DTO — это объект, который используется только для передачи данных.

Он:

— не содержит логики
— просто хранит данные
— приходит из сети или базы

Обычно это “сырой” формат данных.

DTO is an object used only to transfer data.

It:

— has no logic
— just holds data
— comes from network or database

Ссылка на видео этого руководства на сайте: borisproit.expert

It’s usually raw data.

```
// DTO from API
data class UserDto(
    val name: String,
    val age: Int
)
```

Чем отличается DTO от Domain модели? What is the difference between DTO and Domain model?

DTO — это “сырой” объект данных.
Он приходит из сети или базы.

Он:

— отражает формат API
— может быть неудобным
— не предназначен для логики

Domain модель — это объект, удобный для работы внутри приложения.

Она:

— используется в логике
— может отличаться от API
— понятна бизнес-уровню

DTO is a raw data object.
It comes from network or database.

It:

— reflects API format
— may be inconvenient
— is not for logic

Domain model is used inside the app.

It:

Ссылка на видео этого руководства на сайте: borisproit.expert

— is used in logic
— may differ from API
— fits business needs

```
// DTO from API
data class UserDto(
    val first_name: String
)

// Domain model
data class User(
    val name: String
)
```

пример DTO vs Domain Model? Give an example of DTO vs Domain Model?

DTO — это формат, как данные приходят извне.
Domain — это формат, как удобно работать внутри приложения.

DTO может быть “грязным”,
Domain — всегда чистый и понятный.

DTO is how data comes from outside.
Domain is how we use it inside the app.

DTO may be “messy”,
Domain is clean and usable.

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
// DTO from API
data class UserDto(
    val first_name: String,
    val last_name: String,
    val birth_year: Int
)

// Domain model used inside app
data class User(
    val fullName: String,
    val age: Int
)

// Mapping DTO -> Domain
fun UserDto.toDomain(): User {
    return User(
        fullName = "$first_name $last_name",
        age = 2026 - birth_year
    )
}
```

Где должна происходить валидация данных? Where should data validation happen?

Валидация должна происходить в логике — обычно в UseCase или Domain слое.

Не в UI
и не в Repository.

UI только собирает ввод,
а UseCase решает — корректны данные или нет.

Validation should happen in logic — usually in UseCase or Domain layer.

Ссылка на видео этого руководства на сайте: borisproit.expert

Not in UI
and not in Repository.

UI just collects input,
UseCase decides if it's valid.

```
// UseCase handles validation
class RegisterUserUseCase {

    fun execute(name: String): Boolean {
        return name.isNotBlank() // validate input
    }
}
```

Зачем нужен Repository как интерфейс? Why use Repository as an interface?

Чтобы не зависеть от конкретной реализации.

Интерфейс позволяет:

— легко менять источник данных
— подменять реализацию в тестах
— не привязывать ViewModel к сети или базе

ViewModel работает с абстракцией,
а не с конкретным классом.

To avoid depending on a specific implementation.

An interface allows:

— easy switching of data sources
— replacing implementation in tests
— keeping ViewModel independent from network or DB

ViewModel works with abstraction,
not with a concrete class.

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
// Repository interface
interface UserRepository {
    fun getUser(): String
}

// Real implementation
class NetworkUserRepository : UserRepository {
    override fun getUser(): String = "User from API"
}

// ViewModel depends on interface
class UserViewModel(private val repository: UserRepository) {
    fun loadUser(): String = repository.getUser()
}
```

Когда UseCase оправдан? When is UseCase justified?

UseCase оправдан, когда появляется бизнес-логика.

Например:

— нужно что-то проверить
— объединить несколько источников данных
— выполнить последовательность действий

Если ViewModel начинает “умнеть” —
логика должна уйти в UseCase.

В простых задачах он может быть лишним.

UseCase is justified when business logic appears.

For example:

— validation is needed
— combine multiple data sources
— perform a sequence of actions

Ссылка на видео этого руководства на сайте: borisproit.expert

If ViewModel becomes too “smart” —
logic should move to UseCase.

In simple cases, it may be unnecessary.

```
// UseCase handles business logic
class RegisterUserUseCase(
    private val repository: UserRepository
) {

    fun execute(name: String): Boolean {
        if (name.isBlank()) return false // business rule
        repository.saveUser(name)
        return true
    }
}
```

Когда UseCase — лишний? When is UseCase unnecessary?

Когда нет бизнес-логики.

Если ViewModel просто:

— вызывает Repository
— ничего не проверяет
— ничего не объединяет

то UseCase не даёт ценности
и только добавляет лишний слой.

В таком случае ViewModel может работать напрямую с Repository.

When there is no business logic.

If ViewModel only:

Ссылка на видео этого руководства на сайте: borisproit.expert

— calls Repository
— does no validation
— does no coordination

then UseCase adds no value
and becomes extra complexity.

In this case, ViewModel can work directly with Repository.

```
// UseCase is unnecessary here

class UserViewModel(private val repository: UserRepository) {

    fun loadUser(): String {
        return repository.getUser() // simple pass-through
    }
}
```

Может ли ViewModel напрямую вызывать Repository? Can ViewModel call Repository directly?

Да, может.

Это нормально в простых сценариях,
где нет бизнес-логики.

ViewModel может напрямую получать данные
из Repository.

UseCase нужен только тогда,
когда появляется логика.

Yes, it can.

This is normal in simple cases
when there is no business logic.

ViewModel can directly get data
from Repository.

Ссылка на видео этого руководства на сайте: borisproit.expert

UseCase is needed only
when logic appears.

Где должен происходить кеш? Where should caching happen?

Кеш должен происходить в Data слое — внутри Repository.

UI и ViewModel не должны знать:

— есть ли кеш
— откуда пришли данные
— из сети или памяти

Repository сам решает:

👉 взять из кеша
👉 или загрузить заново

Caching should happen in the Data layer — inside Repository.

UI and ViewModel should not know:

— if cache exists
— where data came from
— network or memory

Repository decides:

👉 use cache
👉 or fetch fresh data

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Repository handles caching
class UserRepository {

    private var cachedUser: String? = null

    fun getUser(): String {
        return cachedUser ?: fetchFromNetwork().also {
            cachedUser = it
        }
    }

    private fun fetchFromNetwork(): String {
        return "User from API"
    }
}
```

Где должна обрабатываться ошибка сети? Where should network error be handled?

Ошибка сети должна обрабатываться в Data слое — обычно в Repository.

UI не должен знать:

— что это была сеть
— какой код ошибки
— что именно сломалось

Repository ловит ошибку
и передаёт дальше уже понятный результат.

Network error should be handled in the Data layer — usually in Repository.

UI should not know:

— it was a network issue
— error codes
— what exactly failed

Ссылка на видео этого руководства на сайте: borisproit.expert

Repository catches the error
and passes a clear result upward.

```
// Repository handles network error
class UserRepository(private val api: UserApi) {

    fun getUser(): Result<String> {
        return try {
            Result.success(api.getUser())
        } catch (e: Exception) {
            Result.failure(e) // network error handled here
        }
    }
}
```

Что такое orchestration? What is orchestration?

Orchestration — это координация нескольких действий.

Когда нужно:

— вызвать несколько источников данных
— объединить результат
— выполнить шаги в нужном порядке

Кто-то должен управлять процессом.

Обычно этим занимается UseCase или ViewModel.

Orchestration is coordinating multiple actions.

When you need to:

— call several data sources
— combine results
— execute steps in order

Someone must manage the flow.

Ссылка на видео этого руководства на сайте: borisproit.expert

Usually handled by UseCase or ViewModel.

```
// UseCase orchestrates multiple sources
class GetUserProfileUseCase(
    private val userRepository: UserRepository,
    private val statsRepository: StatsRepository
) {

    fun execute(): String {
        val user = userRepository.getUser()
        val stats = statsRepository.getStats()
        return "$user - $stats"
    }
}
```

Что такое SOLID? What is SOLID?

SOLID — это набор принципов, которые помогают писать чистый и поддерживаемый код.

Они говорят, как правильно разделять ответственность
и уменьшать зависимости.

SOLID помогает:

— избегать хаоса
— делать код гибким
— упростить изменения

SOLID is a set of principles for writing clean and maintainable code.

They guide how to separate responsibilities
and reduce dependencies.

SOLID helps to:

Ссылка на видео этого руководства на сайте: borisproit.expert

— avoid messy code
— make systems flexible
— simplify changes

Дай краткое описание к каждому пункту SOLID? Give short description of each SOLID principle?

S — Single Responsibility
Класс должен делать только одну вещь.

O — Open / Closed
Код должен быть открыт для расширения, но закрыт для изменений.

L — Liskov Substitution
Подкласс должен спокойно заменять родителя.

I — Interface Segregation
Лучше много маленьких интерфейсов, чем один большой.

D — Dependency Inversion
Зависеть нужно от абстракций, а не от конкретных классов.

S — Single Responsibility
A class should do only one thing.

O — Open / Closed
Code should be open for extension, closed for modification.

L — Liskov Substitution
A child class should replace its parent without breaking behavior.

I — Interface Segregation
Prefer many small interfaces over one large one.

D — Dependency Inversion
Depend on abstractions, not concrete classes.

Что такое SRP и как он применим к ViewModel? What is SRP and how does it apply to ViewModel?

SRP — это принцип одной ответственности.

Класс должен отвечать только за одну задачу.

Для ViewModel это значит:

Ссылка на видео этого руководства на сайте: borisproit.expert

Она должна управлять состоянием экрана
и подготовкой данных для UI.

Но не должна:

— работать с сетью напрямую
— форматировать сложные бизнес-правила
— заниматься сохранением данных

SRP means Single Responsibility Principle.

A class should have only one job.

For ViewModel, it means:

It should manage screen state
and prepare data for UI.

But should not:

— call network directly
— contain complex business logic
— handle persistence

Что такое OCP на практике? What is OCP in practice?

OCP — это когда ты можешь добавить новое поведение,
не меняя существующий код.

То есть не переписывать старое,
а расширять через новые классы.

Например:

Сегодня есть один способ оплаты,
завтра добавляем второй —
и старый код не трогаем.

OCP means you can add new behavior
without changing existing code.

You extend, not modify.

For example:

Ссылка на видео этого руководства на сайте: borisproit.expert

Today you have one payment method,
tomorrow you add another —
without touching old logic.

```kotlin
// Abstraction
interface Payment {
    fun pay(): String
}

// Existing implementation
class CardPayment : Payment {
    override fun pay() = "Pay with card"
}

// New behavior added without modifying old code
class PaypalPayment : Payment {
    override fun pay() = "Pay with PayPal"
}

// Works with abstraction
fun process(payment: Payment) {
    println(payment.pay())
}
```

Что такое DIP и как он связан с Repository? What is DIP and how is it related to Repository?

DIP — это принцип, что зависеть нужно от абстракций,
а не от конкретных классов.

ViewModel не должна зависеть от конкретного Repository.

Она должна работать через интерфейс.

Ссылка на видео этого руководства на сайте: borisproit.expert

Тогда можно легко:

— заменить реализацию
— тестировать
— менять источник данных

DIP means depending on abstractions,
not concrete implementations.

ViewModel should not depend on a concrete Repository.

It should depend on an interface.

This allows:

— swapping implementations
— easy testing
— changing data sources

```
// Abstraction
interface UserRepository {
    fun getUser(): String
}

// Implementation
class NetworkUserRepository : UserRepository {
    override fun getUser(): String = "User from API"
}

// ViewModel depends on abstraction
class UserViewModel(private val repository: UserRepository) {
    fun loadUser(): String = repository.getUser()
}
```

Что такое ISP в Android контексте? What is ISP in Android context?

Ссылка на видео этого руководства на сайте: borisproit.expert

ISP — это принцип: лучше несколько маленьких интерфейсов, чем один большой.

В Android это значит:

ViewModel или UseCase не должны зависеть
от интерфейса, где есть лишние методы.

Каждый должен получать только то, что ему нужно.

ISP means: prefer small focused interfaces over one big one.

In Android:

ViewModel or UseCase should not depend
on an interface with unused methods.

Each class should use only what it needs.

```
// ❌ Bad: big interface
interface UserRepository {
    fun getUser(): String
    fun saveUser(name: String)
    fun deleteUser()
}

// ViewModel needs only read
class UserViewModel(private val repository: UserRepository)

// ✅ Good: split interfaces
interface UserReader {
    fun getUser(): String
}

class UserViewModel(private val repository: UserReader)
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Как LSP может быть нарушен в архитектуре? How can LSP be violated in architecture?

LSP нарушается, когда замена родителя на наследника ломает поведение.

То есть:

Мы ожидаем одно поведение,
а получаем другое.

Например, Repository обещает вернуть данные,
но одна реализация вдруг кидает исключение или возвращает null.

Код начинает ломаться.

LSP is violated when replacing a parent with a child breaks behavior.

We expect the same behavior,
but get something different.

For example, Repository promises data,
but one implementation throws or returns null.

Code breaks.

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
// Parent contract
interface UserRepository {
    fun getUser(): String
}

// Correct implementation
class NetworkRepository : UserRepository {
    override fun getUser() = "John"
}

// ❌ Violates LSP
class BrokenRepository : UserRepository {
    override fun getUser(): String {
        throw Exception("Not supported") // breaks expectation
    }
}
```

Что такое композиция? What is composition?

Композиция — это когда класс использует другой класс внутри себя.

Не наследуется от него,
а просто содержит его.

То есть:

не “является”,
а “имеет”.

Composition is when a class uses another class inside it.

Ссылка на видео этого руководства на сайте: borisproit.expert

It does not inherit from it,
it contains it.

So it does not “is”,
it “has”.

```
// Composition example

class Engine {
    fun start() = "Engine started"
}

class Car(private val engine: Engine) {

    fun drive(): String {
        return engine.start() // Car uses Engine
    }
}
```

Почему композиция лучше наследования? Why is composition better than inheritance?

Потому что композиция делает систему гибкой.

Класс не привязан к одному родителю.
Он просто использует нужное поведение.

Это позволяет:

— легко менять реализацию
— не ломать существующий код
— избегать жёсткой связи

Ссылка на видео этого руководства на сайте: borisproit.expert

Наследование же связывает навсегда.

Because composition makes the system flexible.

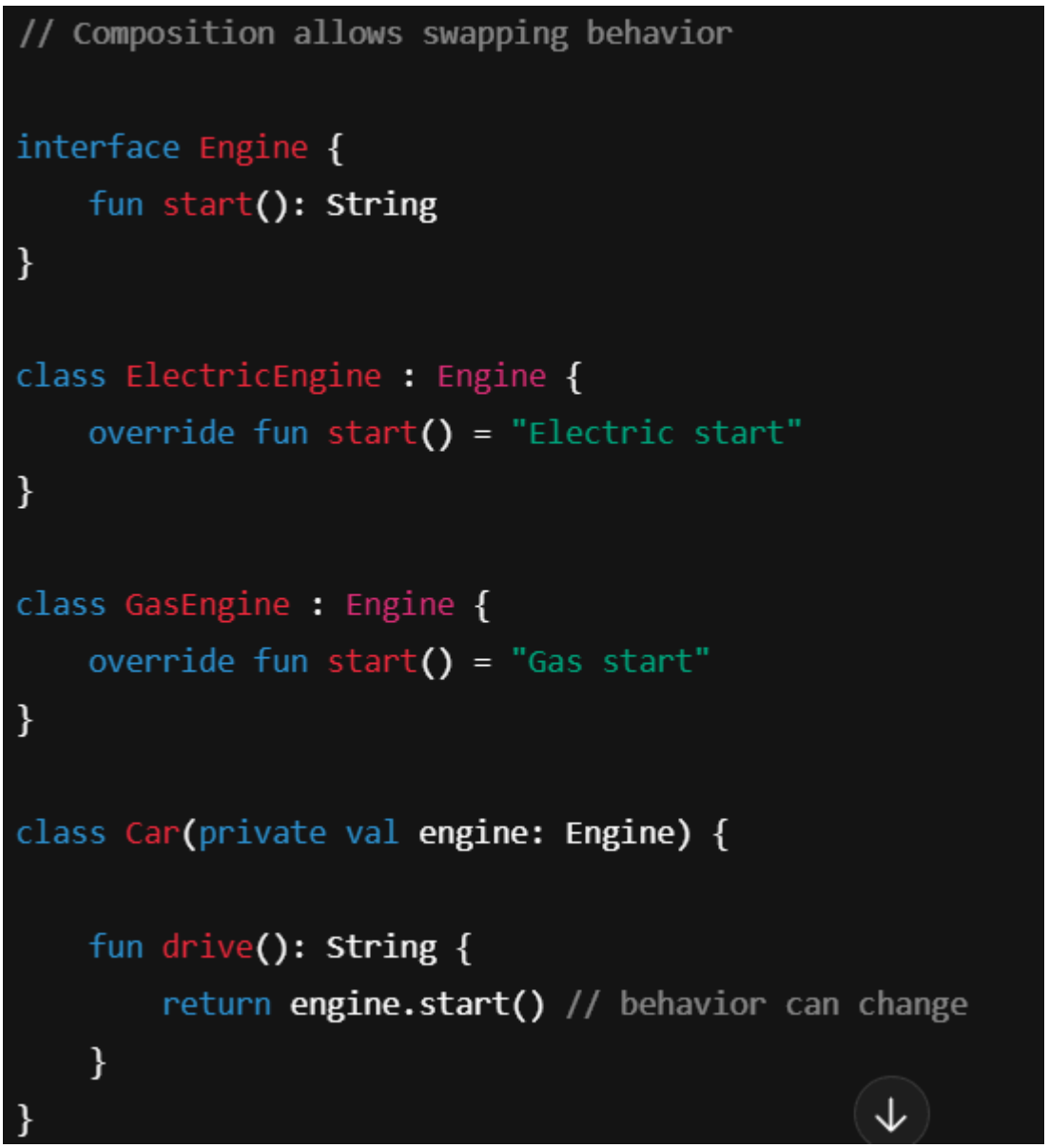
A class is not tied to a parent.
It just uses needed behavior.

This allows:

— easy replacement
— safer changes
— less tight coupling

Inheritance creates rigid coupling.

Ссылка на видео этого руководства на сайте: borisproit.expert



```
// Composition allows swapping behavior

interface Engine {
    fun start(): String
}

class ElectricEngine : Engine {
    override fun start() = "Electric start"
}

class GasEngine : Engine {
    override fun start() = "Gas start"
}

class Car(private val engine: Engine) {

    fun drive(): String {
        return engine.start() // behavior can change
    }
}
```

Когда использовать interface vs abstract class? When to use interface vs abstract class?

Interface — когда нужен контракт поведения.

Когда важно:

— что класс умеет делать
— но не важно, как именно

Abstract class — когда есть общая база.

Ссылка на видео этого руководства на сайте: borisproit.expert

Когда нужно:

— общее состояние
— частичная реализация
— общий код

Interface — use when you need a behavior contract.

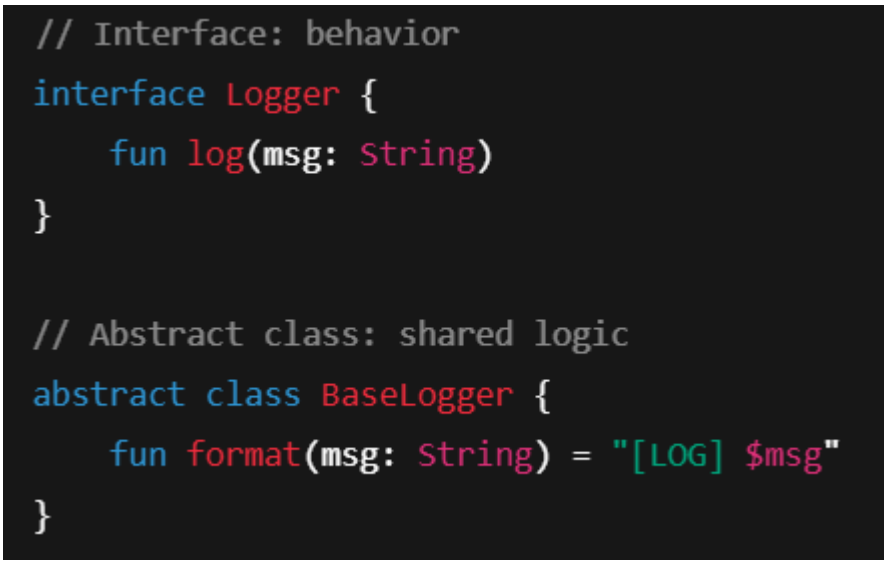
It defines:

— what a class can do
— not how

Abstract class — use when you have shared base logic.

It allows:

— shared state
— partial implementation
— common code



Где полезен полиморфизм? Where is polymorphism useful?

Полиморфизм полезен, когда есть разные реализации одного поведения.

Например:

Ссылка на видео этого руководства на сайте: borisproit.expert

— разные источники данных
— разные способы оплаты
— разные стратегии

Код работает с абстракцией
и не знает о конкретной реализации.

Polymorphism is useful when there are multiple implementations of the same behavior.

For example:

— different data sources
— payment methods
— strategies

Code works with abstraction
without knowing the concrete implementation.

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
// Common abstraction
interface UserRepository {
    fun getUser(): String
}

// Different implementations
class NetworkRepository : UserRepository {
    override fun getUser() = "User from API"
}

class CacheRepository : UserRepository {
    override fun getUser() = "User from cache"
}

// Works polymorphically
fun load(repository: UserRepository) {
    println(repository.getUser())
}
```

Почему sealed class полезен для state? Why is sealed class useful for state?

Потому что он ограничивает все возможные состояния.

Мы заранее знаем:

— какие варианты могут быть
— что ничего лишнего не появится

Это делает UI предсказуемым
и избавляет от неожиданных состояний.

Ссылка на видео этого руководства на сайте: borisproit.expert

Because it limits all possible states.

We know in advance:

— what states exist
— that nothing extra will appear

This makes UI predictable
and avoids unexpected cases.

```
// All possible states are defined
sealed class UserState {
    object Loading : UserState()
    data class Success(val name: String) : UserState()
    object Error : UserState()
}
```

Когда стоит использовать delegation? When should you use delegation?

Delegation нужен, когда:

класс хочет использовать чужую логику,
но не должен становиться этим классом.

То есть:

он не “является Logger”,
но хочет уметь логировать.

Вместо наследования
он просто передаёт работу другому объекту.

Use delegation when:

a class wants to use behavior,
but should not inherit from it.

It is not a Logger,
but wants logging ability.

Ссылка на видео этого руководства на сайте: borisproit.expert

Instead of inheriting,
it forwards work to another object.

```
// Behavior
interface Logger {
    fun log(msg: String)
}

class ConsoleLogger : Logger {
    override fun log(msg: String) = println(msg)
}

// Delegation instead of inheritance
class UserService(private val logger: Logger) {

    fun loadUser() {
        logger.log("Loading user") // forwards responsibility
    }
}
```

Что такое Single Source of Truth? What is Single Source of Truth?

Single Source of Truth — это когда данные хранятся в одном месте.

Не в нескольких.

Это значит:

— одно состояние
— одна версия данных
— нет рассинхронизации

UI всегда читает данные из одного источника.

Single Source of Truth means data lives in one place.

Not in many.

Ссылка на видео этого руководства на сайте: borisproit.expert

This ensures:

— one state
— one version of data
— no inconsistencies

UI always reads from one source.

🔴 ПРОДВИНУТЫЙ УРОВЕНЬ (Senior)

Что такое maintainability? What is maintainability?

Maintainability — это насколько легко поддерживать код.

То есть:

— понимать
— менять
— исправлять
— расширять

Хорошая архитектура повышает maintainability
и делает работу безопаснее.

Maintainability is how easy it is to maintain code.

Meaning:

— understand
— change
— fix
— extend

Good architecture improves maintainability
and makes changes safer.

Как SOLID улучшает maintainability? How does SOLID improve maintainability?

SOLID делает код проще для изменений.

Он:

Ссылка на видео этого руководства на сайте: borisproit.expert

— разделяет ответственность
— уменьшает зависимости
— делает поведение предсказуемым

В итоге:

меняя одну часть —
мы не ломаем остальные.

SOLID makes code easier to maintain.

It:

— separates responsibilities
— reduces dependencies
— keeps behavior predictable

So when one part changes,
others stay stable.

Как SOLID улучшает тестируемость? How does SOLID improve testability?

SOLID уменьшает связанность между классами.

Классы:

— зависят от интерфейсов
— делают одну задачу
— легко заменяются

Это позволяет:

👉 подменять зависимости
👉 писать изолированные тесты

SOLID reduces coupling between classes.

Classes:

— depend on interfaces
— have one responsibility
— are easy to replace

This allows:

Ссылка на видео этого руководства на сайте: borisproit.expert

👉 mocking dependencies
👉 writing isolated tests

Когда SOLID становится overengineering? When does SOLID become overengineering?

Когда мы начинаем добавлять абстракции без реальной необходимости.

Например:

— интерфейс ради интерфейса
— UseCase без логики
— лишние слои в простом флоу

Код становится сложнее,
а пользы нет.

SOLID нужен там, где есть сложность.
В простых задачах — он может мешать.

SOLID becomes overengineering
when abstractions are added without need.

For example:

— interface for no reason
— UseCase with no logic
— extra layers for simple flows

Code gets harder,
with no benefit.

SOLID helps with complexity.
In simple cases, it may hurt.

Как SOLID связан с архитектурными паттернами (например MVVM)?
How is SOLID related to architectural patterns (like MVVM)?

SOLID — это принципы,
а MVVM — это структура.

Ссылка на видео этого руководства на сайте: borisproit.expert

MVVM говорит:
👉 раздели на View / ViewModel / Model

SOLID говорит:
👉 как правильно реализовать каждую часть

Например:

— SRP → ViewModel отвечает только за state
— DIP → ViewModel работает через Repository interface
— OCP → можно добавить новый источник данных без изменения ViewModel

---

SOLID are principles,
MVVM is structure.

MVVM says:
👉 split into View / ViewModel / Model

SOLID says:
👉 how to implement each part correctly

For example:

— SRP → ViewModel manages only state
— DIP → ViewModel depends on repository interface
— OCP → new data source can be added without modifying ViewModel

Как рефакторить класс, нарушающий SRP? How to refactor a class that breaks SRP?

Если класс делает “всё подряд”, SRP нарушен. Рефактор обычно такой:

1. **Найти разные задачи внутри класса**
   Например: загрузка из сети, кеш, форматирование, логирование, обработка ошибок, навигация.

2. **Разделить по ролям**
   — работа с данными → Repository / DataSource
   — бизнес-правила → UseCase
   — подготовка для UI / состояние → ViewModel
   — отображение → View

3. **Вынести куски в отдельные классы**
   Оставить в исходном классе только одну ответственность.

Ссылка на видео этого руководства на сайте: borisproit.expert

4. **Подключить через интерфейсы** (чтобы потом легко менять и тестировать)

If a class “does everything”, it breaks SRP. Typical refactor:

1. **Identify different responsibilities**
2. **Split by roles**
3. **Extract into separate classes**
4. **Use interfaces for dependencies** (easier testing and swapping)

Какие антипаттерны приводят к нарушению SOLID?
What anti-patterns lead to SOLID violations?

God Object

Один класс делает всё.

→ нарушает **Single Responsibility Principle**
(у класса больше одной ответственности)

Tight Coupling

Классы напрямую создают зависимости.

→ нарушает **Dependency Inversion Principle**
(зависимость идёт от конкретных классов, а не от абстракций)

Fat Interface

Огромный интерфейс со множеством методов.

→ нарушает **Interface Segregation Principle**
(классы вынуждены реализовывать лишнее)

Ссылка на видео этого руководства на сайте: borisproit.expert

Switch / if логика для расширения

Новые типы добавляются через when / if.

→ нарушает **Open Closed Principle**
(приходится менять существующий код вместо расширения)

Throwing “Not Supported”

Реализация интерфейса не поддерживает метод.

→ нарушает **Liskov Substitution Principle**
(подкласс не может заменить родителя без поломки)

Hardcoded Dependencies

Создание зависимостей внутри класса.

→ нарушает **Dependency Inversion Principle**
(нет зависимости от абстракции)

Common anti-patterns:

— God Object → breaks **Single Responsibility Principle**
— Tight Coupling → breaks **Dependency Inversion Principle**
— Fat Interface → breaks **Interface Segregation Principle**
— if/when extension → breaks **Open Closed Principle**
— NotSupported → breaks **Liskov Substitution Principle**
— Hardcoded deps → breaks **Dependency Inversion Principle**

Почему instanceof или switch по типу может нарушать SOLID?
Why can instanceof / type switch violate SOLID?

Когда код проверяет тип:

Ссылка на видео этого руководства на сайте: borisproit.expert

— `if (x is A)`
— `when (x)`

это значит, что поведение зависит от конкретных классов.

Чтобы добавить новый тип —
придётся менять существующий код.

Это нарушает:

👉 **Open Closed Principle**
(код должен расширяться, а не изменяться)

Также часто нарушается:

👉 **Liskov Substitution Principle**
(мы начинаем относиться к наследникам по-разному)

When code checks types:

— `if (x is A)`
— `when (x)`

it means behavior depends on concrete classes.

To add a new type —
existing code must be modified.

This breaks:

👉 **Open Closed Principle**
(code should be open for extension, closed for modification)

And often:

👉 **Liskov Substitution Principle**
(children are no longer treated the same)

Как обеспечить соблюдение SOLID в проекте?

Ссылка на видео этого руководства на сайте: borisproit.expert

Как определить, какие SOLID принципы нарушены в системе?

🔴 MODERN COMPOSE ARCHITECTURE

Меняется ли архитектура при использовании Compose?
Does architecture change when using Compose?

Нет, сама архитектура не меняется.

MVVM остаётся той же:

— View
— ViewModel
— Model

Но меняется реализация View.

Вместо Activity / Fragment
используется Composable-функция.

Compose лучше подходит для:

👉 state-driven UI
👉 Single Source of Truth

No, architecture itself does not change.

MVVM stays the same:

— View
— ViewModel
— Model

But the View implementation changes.

Instead of Activity / Fragment,
you use Composable functions.

Compose fits naturally with:

👉 state-driven UI
👉 Single Source of Truth

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Compose View
@Composable
fun UserScreen(viewModel: UserViewModel) {
    val state = viewModel.state
    Text(state.toString())
}
```

Чем MVVM в Compose отличается от классического MVVM?
How is MVVM in Compose different from classic MVVM?

Структура та же:

— View
— ViewModel
— Model

Но меняется способ работы UI.

В классическом MVVM:

View подписывается на изменения
и вручную обновляет экран.

В Compose:

UI автоматически перерисовывается
когда меняется state.

UI становится функцией состояния.

Structure stays the same:

— View
— ViewModel
— Model

But UI behavior changes.

In classic MVVM:

Ссылка на видео этого руководства на сайте: borisproit.expert

View observes
and updates manually.

In Compose:

UI automatically recomposes
when state changes.

UI becomes a function of state.

```
// Compose UI reacts to state automatically
@Composable
fun UserScreen(state: UserState) {
    when (state) {
        is UserState.Loading -> Text("Loading...")
        is UserState.Success -> Text(state.name)
        is UserState.Error -> Text("Error")
    }
}
```

Что заменяет View в MVVM при использовании Compose?
What replaces View in MVVM when using Compose?

View заменяется Composable-функцией.

Она:

— показывает состояние
— принимает события пользователя
— ничего не решает

То есть роль View остаётся,
меняется только форма.

View is replaced by a Composable function.

It:

— displays state
— handles user input
— makes no decisions

Ссылка на видео этого руководства на сайте: borisproit.expert

The role stays the same,
only the form changes.

Что такое unidirectional data flow в Compose?
What is unidirectional data flow in Compose?

Это когда данные движутся в одном направлении.

Поток такой:

👉 UI отправляет событие
👉 ViewModel обрабатывает
👉 обновляет state
👉 UI перерисовывается

UI не меняет state напрямую.

It means data flows in one direction.

Flow:

👉 UI sends event
👉 ViewModel processes
👉 updates state
👉 UI recomposes

UI does not change state directly.

Почему Compose лучше работает с immutable state?
Why does Compose work better with immutable state?

Потому что Compose отслеживает изменения через сравнение значений.

Когда state неизменяемый:

— любое изменение = новый объект
— легко понять, что произошло обновление
— UI корректно перерисовывается

Mutable state может измениться “тихо”
и Compose может не заметить.

Ссылка на видео этого руководства на сайте: borisproit.expert

Compose tracks changes by comparing values.

With immutable state:

— any change = new object
— updates are obvious
— UI recomposes correctly

Mutable state may change silently
and Compose may miss it.

Как ViewModel взаимодействует с Composable?
How does ViewModel interact with Composable?

Через state и события.

ViewModel:

— отдаёт состояние
— принимает действия пользователя

Composable:

— читает state
— отправляет события

Through state and events.

ViewModel:

— provides state
— receives user actions

Composable:

— reads state
— sends events

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Composable reads state and sends event
@Composable
fun UserScreen(viewModel: UserViewModel) {

    val state = viewModel.state

    Button(onClick = { viewModel.onLoadClick() }) {
        Text("Load")
    }

    Text(state.toString())
}

// ViewModel updates state
class UserViewModel {

    var state: UserState = UserState.Loading
        private set

    fun onLoadClick() {
        state = UserState.Success("John")
    }
}
```

Где должен происходить state mapping для UI?
Where should state mapping for UI happen?

State mapping должен происходить во ViewModel.

Не в UI
и не в Repository.

Repository отдаёт Domain данные,
а ViewModel превращает их в UI state.

State mapping should happen in ViewModel.

Ссылка на видео этого руководства на сайте: borisproit.expert

Not in UI
and not in Repository.

Repository provides domain data,
ViewModel converts it into UI state.

```
// Domain model
data class User(val name: String)

// UI state
sealed class UserState {
    data class Success(val displayName: String) : UserState()
}

// ViewModel maps Domain -> UI
class UserViewModel(private val repository: UserRepository) {

    fun loadUser(): UserState {
        val user = repository.getUser()
        return UserState.Success(user.name.uppercase())
    }
}
```

Где должен храниться UI state в Compose?
Where should UI state live in Compose?

UI state должен храниться во ViewModel.

Не в Composable.

Composable может пересоздаваться много раз,
а ViewModel сохраняет состояние.

UI только читает state
и отображает его.

UI state should live in ViewModel.

Not in Composable.

Composable can be recreated many times,
but ViewModel keeps the state.

Ссылка на видео этого руководства на сайте: borisproit.expert

UI just reads and displays it.

Чем StateFlow лучше LiveData в Compose?
Why is StateFlow better than LiveData in Compose?

StateFlow обычно удобнее в Compose, потому что он “родной” для корутин и state-driven подхода.

Что чаще всего выигрывает на практике:

— **Единый стек**: корутины + Flow везде (данные, state, события), без отдельной “LiveData-экосистемы”
— **Работает вне Android**: StateFlow не привязан к Android framework, проще тестировать и переиспользовать
— **Предсказуемый state**: у StateFlow всегда есть текущее значение (как “одно место правды”)
— **Compose-интеграция**: удобно собирать state через `collectAsStateWithLifecycle()` и UI сам перерисуется
— **Лучше подходит для потока данных**: легко комбинировать, маппить, фильтровать (map/flatMapLatest/combine и т.д.)

LiveData тоже работает, но чаще воспринимается как “старый UI-инструмент” из мира XML, который в Compose просто менее удобен по стилю.

StateFlow is usually a better fit for Compose because it matches coroutines and state-driven UI.

Common practical benefits:

— **Single stack**: coroutines + Flow everywhere
— **Not Android-tied**: easier testing and reuse
— **Always has a current value**
— **Compose-friendly** with `collectAsStateWithLifecycle()`
— **Powerful operators** for mapping/combining state

LiveData still works, but it’s more of a legacy UI tool compared to Flow/StateFlow in modern Compose apps.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// ViewModel exposes UI state via StateFlow
class UserViewModel {
    private val _state = MutableStateFlow<UserState>(UserState.Loading)
    val state: StateFlow<UserState> = _state

    fun load() {
        _state.value = UserState.Success("John")
    }
}

// Composable collects state and recomposes
@Composable
fun UserScreen(viewModel: UserViewModel) {
    val state = viewModel.state.collectAsStateWithLifecycle().value
    Text(state.toString())
}
```

Как обрабатывать events в Compose?
How to handle events in Compose?

Events обрабатываются отдельно от state.

Flow такой:

👉 ViewModel отправляет event
👉 UI его получает
👉 выполняет действие

Event не должен храниться в state.

Events should be handled separately from state.

Flow:

👉 ViewModel emits event
👉 UI receives it
👉 performs action

Event should not be stored in state.

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
// One-time event
sealed class UiEvent {
    object ShowToast : UiEvent()
}

// ViewModel emits event
class UserViewModel {

    private val _event = MutableSharedFlow<UiEvent>()
    val event = _event

    suspend fun onError() {
        _event.emit(UiEvent.ShowToast)
    }
}

// UI collects event
@Composable
fun UserScreen(viewModel: UserViewModel) {

    LaunchedEffect(Unit) {
        viewModel.event.collect {
            // handle event once
        }
    }
}
```

Что такое hoisted state? What is hoisted state?

Hoisted state — это ключевой принцип в Compose.

Идея:

Composable не хранит важное состояние внутри себя.
Он получает его извне.

Ссылка на видео этого руководства на сайте: borisproit.expert

---

Зачем это нужно:

👉 чтобы UI был предсказуемым
👉 чтобы состояние не терялось
👉 чтобы компонент можно было переиспользовать
👉 чтобы логика не смешивалась с отображением

Как это выглядит концептуально:

State поднимается вверх
→ в родитель
→ во ViewModel

Composable становится “глупым”:

он только:

— показывает данные
— сообщает о действиях

Что меняется:

Вместо:

👉 внутри Composable есть state

мы делаем:

👉 state снаружи
👉 Composable получает:

state + callback

Почему это важно:

Это делает UI:

— stateless
— тестируемым
— легко управляемым

Ссылка на видео этого руководства на сайте: borisproit.expert

Hoisted state is a core Compose principle.

Composable does not own important state.
It receives it from above.

Benefits:

👉 predictable UI
👉 no state loss
👉 reusable components
👉 clean separation

Composable becomes:

— display-only
— event sender

```
// Stateless composable
@Composable
fun LoginField(
    text: String,
    onTextChange: (String) -> Unit
) {
    TextField(
        value = text,
        onValueChange = onTextChange
    )
}

// State is hoisted
@Composable
fun LoginScreen() {

    var text by remember { mutableStateOf("") }

    LoginField(
        text = text,
        onTextChange = { text = it }
    )
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Как избежать бизнес-логики в Composable?
How to avoid business logic in Composable?

Composable должен только:

— показывать данные
— отправлять события

Он не должен:

— принимать решения
— валидировать
— загружать данные

Бизнес-логика должна жить во ViewModel или UseCase.

UI получает готовый state
и сообщает о действиях.

Composable should only:

— display data
— send events

It should not:

— make decisions
— validate
— load data

Business logic should live in ViewModel or UseCase.

UI receives ready state
and reports actions.

Можно ли использовать MVI-подход с Compose?
Can you use MVI with Compose?

Да, можно — и часто это даже очень удобно.

Compose отлично подходит для MVI, потому что он естественно работает как state-driven UI:

Ссылка на видео этого руководства на сайте: borisproit.expert

— есть один UI state
— UI рисуется из state
— события идут наружу
— state обновляется в одном месте

Yes, you can — and it often fits very well.

Compose works great with MVI because it is naturally state-driven:

— single UI state
— UI is rendered from state
— events go outward
— state is updated in one place

Чем Compose усиливает Single Source of Truth?
How does Compose reinforce Single Source of Truth?

Compose строит UI напрямую из state.

Это означает:

— UI не хранит свою копию данных
— нет ручной синхронизации
— экран всегда отражает одно состояние

Composable — это функция от state.

Поэтому источник данных должен быть один.

Compose builds UI directly from state.

This means:

— UI does not keep its own copy
— no manual syncing
— screen always reflects one state

Composable is a function of state.

So there must be a single source of truth.

Ссылка на видео этого руководства на сайте: borisproit.expert

Как Compose влияет на тестируемость архитектуры?
How does Compose affect architecture testability?

Compose делает UI проще для тестирования,
потому что он основан на state.

UI — это просто функция:

state → экран

Нет необходимости:

— мокать Activity
— проверять lifecycle
— синхронизировать вручную

Можно тестировать:

👉 ViewModel отдельно
👉 UI отдельно

Compose makes UI easier to test
because it is state-driven.

UI becomes a function:

state → screen

No need to:

— mock Activity
— handle lifecycle
— sync manually

You can test:

👉 ViewModel separately
👉 UI separately