Ссылка на видео этого руководства на сайте: borisproit.expert

🟢 FOUNDATION (Junior)

Что такое Dependency Injection?
What is Dependency Injection?

Dependency Injection — это способ передачи зависимостей объекту извне, вместо их создания внутри класса.
Это уменьшает связанность, упрощает тестирование и делает код более гибким и расширяемым.

Dependency Injection is a technique where dependencies are provided to a class from the outside instead of being created inside it.
It reduces coupling, improves testability, and makes the code more flexible and maintainable.

```
// ❌ Tight coupling: dependency created inside the class
class UserRepository {
    private val api = ApiService()
}

// ✅ Dependency is injected from outside
class UserRepository(private val api: ApiService)
```

Зачем нужен DI?
Why is DI needed?

DI нужен для уменьшения связанности между классами.
Он позволяет легко заменять зависимости (например, на mock в тестах), упрощает поддержку кода и делает архитектуру более масштабируемой.

DI is used to reduce coupling between classes.
It allows easy replacement of dependencies (for example, with mocks in tests), simplifies maintenance, and makes the architecture more scalable.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Without DI: hard to test
class UserService {
    private val repository = UserRepository()
}

// With DI: easy to replace dependency
class UserService(private val repository: UserRepository)
```

Что такое зависимость?
What is a dependency?

Зависимость — это объект или компонент, который требуется другому классу для выполнения своей работы.
Если класс использует другой класс внутри себя, значит он от него зависит.

A dependency is an object or component that another class needs to perform its work.
If a class uses another class inside it, it depends on it.

```
// ApiService is a dependency of UserRepository
class UserRepository(private val apiService: ApiService) {

    fun loadUser() {
        apiService.getUser() // uses the dependency
    }
}
```

Почему создавать зависимости внутри класса — плохая идея?
Why is creating dependencies inside a class a bad idea?

Когда класс сам создаёт свои зависимости, он становится жёстко связанным с конкретной реализацией.
Это усложняет тестирование (невозможно подменить зависимость), нарушает принцип инверсии зависимостей и делает код менее гибким.

When a class creates its own dependencies, it becomes tightly coupled to a specific implementation.

Ссылка на видео этого руководства на сайте: borisproit.expert

This makes testing harder (you cannot replace the dependency), violates the Dependency Inversion Principle, and reduces flexibility.

```
// ❌ Tight coupling: dependency created inside
class UserService {
    private val repository = UserRepository()

    fun load() {
        repository.getUser()
    }
}

// ✅ Loose coupling: dependency injected
class UserService(private val repository: UserRepository) {

    fun load() {
        repository.getUser()
    }
}
```

Что такое tight coupling?
What is tight coupling?

Tight coupling — это сильная зависимость одного класса от конкретной реализации другого класса.
Изменение одной части системы требует изменений в другой, код сложно тестировать и расширять.

Tight coupling is a strong dependency of one class on a specific implementation of another class.
Changes in one part of the system require changes in another, and the code becomes harder to test and extend.

Как DI помогает уменьшить связанность?
How does DI reduce coupling?

Ссылка на видео этого руководства на сайте: borisproit.expert

DI уменьшает связанность, потому что класс больше не создаёт конкретную реализацию зависимости.
Он получает абстракцию (например, интерфейс) извне, поэтому его можно легко расширять, менять реализацию и тестировать без изменения самого класса.

DI reduces coupling because a class no longer creates a specific implementation of a dependency.
Instead, it receives an abstraction (for example, an interface), which allows easy replacement, extension, and testing without modifying the class.

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

// Concrete implementation
class UserRepositoryImpl : UserRepository {
    override fun getUser() { }
}

// Service depends on abstraction, not implementation
class UserService(private val repository: UserRepository) {

    fun load() {
        repository.getUser()
    }
}
```

Какие есть способы внедрения зависимостей?
What are the types of dependency injection?

Основные способы внедрения зависимостей:

1. Constructor injection — зависимость передаётся через конструктор (самый предпочтительный способ).

2. Field injection — зависимость внедряется напрямую в поле класса.

Ссылка на видео этого руководства на сайте: borisproit.expert

3. Method injection — зависимость передаётся через метод.

The main types of dependency injection are:

1. Constructor injection — dependency is provided through the constructor (most recommended).
2. Field injection — dependency is injected directly into a field.
3. Method injection — dependency is passed through a method.

Что такое constructor injection?
What is constructor injection?

Constructor injection — это способ внедрения зависимостей через конструктор класса.
Зависимость передаётся при создании объекта, что гарантирует, что объект всегда будет в корректном состоянии.
Это самый предпочтительный способ DI, потому что делает зависимости явными и облегчает тестирование.

Constructor injection is a way of providing dependencies through a class constructor.
The dependency is supplied when the object is created, ensuring the object is always in a valid state.
It is the preferred DI approach because it makes dependencies explicit and improves testability.

```
// Dependency
class UserRepository

// Constructor injection
class UserService(private val repository: UserRepository)

// Creating object with dependency
val service = UserService(UserRepository())
```

Что такое field injection?
What is field injection?

Ссылка на видео этого руководства на сайте: borisproit.expert

Field injection — это способ внедрения зависимости напрямую в поле класса после его создания.
Зависимость не передаётся через конструктор, поэтому объект может существовать во временно некорректном состоянии.
Менее предпочтительный способ по сравнению с constructor injection.

Field injection is a way of injecting a dependency directly into a class field after the object is created.
The dependency is not provided through the constructor, so the object may temporarily exist in an invalid state.
It is less preferred compared to constructor injection.

```
class UserService {

    @Inject
    lateinit var repository: UserRepository

    fun load() {
        println(repository.getUser())
    }
}
```

Что такое method injection?
What is method injection?

Method injection — это способ внедрения зависимости через параметр метода.
Зависимость передаётся только в момент вызова конкретного метода, а не хранится как поле класса.
Подходит для временных или редко используемых зависимостей.

Method injection is a way of providing a dependency through a method parameter.
The dependency is passed only when the method is called and is not stored as a class field.
It is useful for temporary or rarely used dependencies.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
class UserService {

    private lateinit var repository: UserRepository

    @Inject
    fun setRepository(repository: UserRepository) {
        this.repository = repository
    }

    fun load() {
        println(repository.getUser())
    }
}
```

В каких случаях нам приходится использовать Field injection или Method injection, вместо Constructor injection?
In what cases do we use Field injection or Method injection instead of Constructor injection?

Constructor injection — предпочтительный способ, но бывают ситуации, когда его использовать нельзя или неудобно.

Field injection используют, когда объект создаётся фреймворком (например, Activity, Fragment), и мы не контролируем его конструктор.
Также применяется при интеграции с DI-фреймворками, где зависимости внедряются после создания объекта.

Method injection применяют, когда зависимость нужна только для конкретной операции и не должна храниться в классе.
Это удобно для временных зависимостей или когда поведение может меняться от вызова к вызову.

Constructor injection is preferred, but sometimes it cannot be used or is impractical.

Field injection is used when an object is created by a framework (for example, Activity or Fragment), and we cannot control its constructor.
It is also common when integrating with DI frameworks that inject dependencies after object creation.

Ссылка на видео этого руководства на сайте: borisproit.expert

Method injection is used when a dependency is needed only for a specific operation and should not be stored in the class.
It is useful for temporary dependencies or when behavior changes per call.

Как DI связан с тестируемостью?
How is DI related to testability?

DI напрямую улучшает тестируемость, потому что позволяет подменять реальные зависимости на mock или fake-реализации.
Класс не создаёт зависимости сам, поэтому в тесте можно передать контролируемую реализацию и изолировать поведение.

DI directly improves testability because it allows replacing real dependencies with mocks or fakes.
Since a class does not create its own dependencies, tests can provide controlled implementations and isolate behavior.

```
// Production dependency
class UserRepository {
    fun getUser(): String = "Real user"
}

// Service depends on abstraction
class UserService(private val repository: UserRepository) {
    fun load() = repository.getUser()
}

// Test fake dependency
class FakeUserRepository : UserRepository() {
    override fun getUser(): String = "Test user"
}

// In test
val service = UserService(FakeUserRepository())
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Можно ли реализовать DI без библиотек?
Can DI be implemented without libraries?

Да, DI можно полностью реализовать вручную.
Достаточно передавать зависимости через конструктор или методы при создании объектов.
Библиотеки (например, Hilt, Dagger) лишь автоматизируют процесс и упрощают управление графом зависимостей.

Yes, DI can be fully implemented manually.
You just pass dependencies through constructors or methods when creating objects.
Libraries like Hilt or Dagger only automate the process and simplify dependency graph management.

🟡 MIDDLE

Что такое Service Locator?
What is a Service Locator?

Service Locator — это паттерн, при котором объект сам запрашивает зависимость из общего «хранилища» или контейнера.
В отличие от DI, зависимость не передаётся извне явно — класс сам обращается к глобальному провайдеру.

Service Locator is a pattern where an object requests its dependency from a shared registry or container.
Unlike DI, the dependency is not explicitly provided from the outside — the class retrieves it itself.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Simple Service Locator
object ServiceLocator {
    val userRepository = UserRepository()
}

class UserService {

    fun load() {
        val repository = ServiceLocator.userRepository
        repository.getUser()
    }
}
```

Чем DI отличается от Service Locator?
What is the difference between DI and Service Locator?

В DI зависимости **передаются объекту извне**, и они явно видны в конструкторе или параметрах.
В Service Locator объект **сам запрашивает зависимости** из общего контейнера.

DI делает зависимости явными и улучшает тестируемость.
Service Locator скрывает зависимости, увеличивает связанность и усложняет тестирование.

With DI, dependencies are **provided from the outside**, and they are explicit in the constructor or method parameters.
With Service Locator, the object **retrieves its dependencies itself** from a shared container.

DI makes dependencies explicit and improves testability.
Service Locator hides dependencies, increases coupling, and makes testing harder.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// DI approach
class UserService(private val repository: UserRepository)

// Service Locator approach
object ServiceLocator {
    val repository = UserRepository()
}

class UserService {
    fun load() {
        ServiceLocator.repository.getUser()
    }
}
```

Что такое Inversion of Control (IoC)?
What is Inversion of Control (IoC)?

Inversion of Control — это принцип, при котором управление созданием и передачей зависимостей передаётся внешнему механизму, а не самому классу.
Класс больше не контролирует процесс создания зависимостей — он только использует их.

Inversion of Control is a principle where the control of creating and providing dependencies is delegated to an external mechanism instead of the class itself.
The class no longer controls dependency creation — it only uses them.

```
// Without IoC: class controls dependency creation
class UserService {
    private val repository = UserRepository()
}

// With IoC: dependency provided from outside
class UserService(private val repository: UserRepository)
```

Как DI реализует DIP из SOLID?
How does DI implement the Dependency Inversion Principle (DIP)?

Ссылка на видео этого руководства на сайте: borisproit.expert

DIP говорит, что высокоуровневые модули не должны зависеть от низкоуровневых — оба должны зависеть от абстракций.
DI помогает этому, потому что зависимость передаётся через интерфейс, а не создаётся напрямую как конкретная реализация.

В итоге высокоуровневый класс зависит от интерфейса, а конкретная реализация подставляется извне.

DIP states that high-level modules should not depend on low-level modules — both should depend on abstractions.
DI supports this by injecting dependencies through interfaces instead of creating concrete implementations directly.

As a result, the high-level class depends on an abstraction, while the concrete implementation is provided externally.

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

// Low-level implementation
class UserRepositoryImpl : UserRepository {
    override fun getUser() { }
}

// High-level module depends on abstraction
class UserService(private val repository: UserRepository)
```

Почему зависимости должны передаваться извне?
Why should dependencies be provided from the outside?

Потому что класс не должен отвечать за создание своих зависимостей.
Когда зависимости передаются извне, класс становится менее связанным, его легче тестировать и заменять реализации без изменения самого класса.

Это разделяет ответственность: класс выполняет свою логику, а создание зависимостей управляется отдельно.

Ссылка на видео этого руководства на сайте: borisproit.expert

Because a class should not be responsible for creating its own dependencies.
When dependencies are provided from the outside, the class becomes less coupled, easier to test, and implementations can be replaced without modifying the class.

This separates responsibilities: the class handles its logic, while dependency creation is managed externally.

Что такое интерфейс как контракт зависимости?
What is an interface as a dependency contract?

Интерфейс как контракт — это соглашение о том, **какие методы должен реализовать объект**, но без указания конкретной реализации.
Класс зависит не от конкретного класса, а от интерфейса, что позволяет свободно менять реализацию без изменения кода клиента.

An interface as a contract defines **what methods must be implemented**, without specifying the concrete implementation.
A class depends on the interface, not on a specific class, allowing implementations to be swapped without changing client code.

```
// 1. Contract
interface PaymentProcessor {
    fun pay(amount: Double)
}

// 2. Two implementations
class StripeProcessor @Inject constructor() : PaymentProcessor {
    override fun pay(amount: Double) {
        println("Paid $amount with Stripe")
    }
}

class PaypalProcessor @Inject constructor() : PaymentProcessor {
    override fun pay(amount: Double) {
        println("Paid $amount with PayPal")
    }
}
```

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
@Module
@InstallIn(SingletonComponent::class)
abstract class PaymentModule {

    @Binds
    @Stripe
    abstract fun bindStripeProcessor(
        impl: StripeProcessor
    ): PaymentProcessor

    @Binds
    @Paypal
    abstract fun bindPaypalProcessor(
        impl: PaypalProcessor
    ): PaymentProcessor
}

// 5. Client selects which implementation to use
class CheckoutService @Inject constructor(
    @Stripe private val processor: PaymentProcessor // change to @Paypal to switch
) {
    fun checkout(amount: Double) {
        processor.pay(amount)
    }
}
```

Почему ViewModel не должна создавать Repository?
Why shouldn't a ViewModel create a Repository?

Если ViewModel сама создаёт Repository, она становится жёстко связанной с конкретной реализацией.
Это нарушает принцип инверсии зависимостей, усложняет тестирование и делает архитектуру менее гибкой.
ViewModel должна получать Repository извне, чтобы можно было подменять реализацию (например, в тестах).

If a ViewModel creates its own Repository, it becomes tightly coupled to a specific implementation.
This violates the Dependency Inversion Principle, makes testing harder, and reduces architectural flexibility.
A ViewModel should receive its Repository from the outside so the implementation can be replaced (for example, in tests).

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// ❌ Tight coupling
class UserViewModel : ViewModel() {
    private val repository = UserRepository()
}

// ✅ Dependency injected
class UserViewModel(private val repository: UserRepository) : ViewModel()
```

Где должен происходить wiring зависимостей?
Where should dependency wiring happen?

Wiring зависимостей должно происходить на уровне композиции приложения — в точке входа (composition root).
Это может быть Application-класс, DI-модуль (Hilt/Dagger), или отдельный объект-конфигуратор.
Бизнес-классы (ViewModel, Repository, UseCase) не должны создавать зависимости — они только получают их.

Dependency wiring should happen at the application composition root.
This can be the Application class, a DI module (Hilt/Dagger), or a dedicated configuration layer.
Business classes (ViewModel, Repository, UseCase) should not create dependencies — they only receive them.

Что такое dependency graph?
What is a dependency graph?

Dependency graph — это схема всех зависимостей в приложении и их связей друг с другом.
Он показывает, какие объекты зависят от каких, и в каком порядке они должны создаваться.
DI-фреймворки автоматически строят и управляют этим графом.

A dependency graph is a structure that represents all dependencies in an application and how they relate to each other.
It shows which objects depend on others and in what order they must be created.
DI frameworks automatically build and manage this graph.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Example dependency chain:
// ViewModel -> Repository -> ApiService

class ApiService

class UserRepository(private val api: ApiService)

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

Что такое manual DI?
What is manual DI?

Manual DI — это внедрение зависимостей без использования DI-фреймворков.
Объекты создаются вручную, а зависимости передаются через конструкторы или методы в точке композиции приложения.

Manual DI is dependency injection without using a DI framework.
Objects are created manually, and dependencies are passed through constructors or methods at the application's composition root.

```
// Dependencies
class ApiService
class UserRepository(private val api: ApiService)
class UserViewModel(private val repository: UserRepository)

// Manual wiring (composition root)
fun main() {
    val api = ApiService()
    val repository = UserRepository(api)
    val viewModel = UserViewModel(repository)
}
```

Когда manual DI достаточно?
When is manual DI sufficient?

Manual DI достаточно в небольших и средних проектах, где граф зависимостей простой и легко управляем вручную.

Ссылка на видео этого руководства на сайте: borisproit.expert

Если количество классов небольшое, нет сложных scope'ов и модулей, то фреймворк может быть излишним.

Когда проект растёт, появляются разные уровни зависимостей, lifecycle scope и сложная конфигурация — тогда DI-фреймворк становится оправданным.

Manual DI is sufficient in small to medium-sized projects where the dependency graph is simple and manageable by hand.
If there are few classes and no complex scopes or modules, a DI framework may be unnecessary.

As the project grows and introduces multiple layers, scopes, and complex configuration, a DI framework becomes justified.

Что такое Hilt?
What is Hilt?

Hilt — это DI-фреймворк для Android, построенный поверх Dagger.
Он автоматически создаёт и управляет dependency graph, упрощает wiring зависимостей и интегрируется с жизненным циклом Android-компонентов (Application, Activity, Fragment, ViewModel).

Hilt is a dependency injection framework for Android built on top of Dagger.
It automatically creates and manages the dependency graph, simplifies dependency wiring, and integrates with Android component lifecycles (Application, Activity, Fragment, ViewModel).

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
// Providing dependency
@Module
@InstallIn(SingletonComponent::class)
object AppModule {

    @Provides
    fun provideRepository(): UserRepository {
        return UserRepository()
    }
}

// Injecting dependency
@HiltViewModel
class UserViewModel @Inject constructor(
    private val repository: UserRepository
) : ViewModel()
```

Что такое Dagger?
What is Dagger?

Dagger — это compile-time DI-фреймворк для Java и Kotlin.
Он генерирует код во время компиляции для создания dependency graph и внедрения зависимостей.
Это делает его быстрым и эффективным по сравнению с runtime-решениями.

Dagger is a compile-time dependency injection framework for Java and Kotlin.
It generates code at compile time to build the dependency graph and inject dependencies without reflection.
This makes it fast and efficient compared to runtime-based solutions.

Чем Hilt отличается от Dagger?
What is the difference between Hilt and Dagger?

Hilt — это надстройка над Dagger, созданная специально для Android.
Он автоматически создаёт компоненты для Activity, Fragment, ViewModel и упрощает настройку dependency graph.
Dagger — более низкоуровневый и гибкий, но требует больше ручной конфигурации.

Ссылка на видео этого руководства на сайте: borisproit.expert

Hilt уменьшает boilerplate и упрощает интеграцию с lifecycle.
Dagger даёт больше контроля, но требует явного создания компонентов и связей.

Hilt is built on top of Dagger and is designed specifically for Android.
It automatically generates components for Activity, Fragment, ViewModel, and simplifies dependency graph setup.
Dagger is lower-level and more flexible, but requires more manual configuration.

Hilt reduces boilerplate and integrates easily with lifecycle.
Dagger provides more control but requires explicit component and dependency setup.

🟡 LIFECYCLE / SCOPES

Что такое scope в DI?
What is a scope in DI?

Scope в DI определяет **время жизни зависимости**.
Он указывает, как долго объект должен существовать и будет ли использоваться одна и та же его версия в рамках определённого компонента.

Например, `Singleton` означает один экземпляр на всё приложение,
а `ActivityScope` — один экземпляр на время жизни Activity.

A scope in DI defines the **lifecycle of a dependency**.
It determines how long an object should live and whether the same instance is reused within a specific component.

For example, `Singleton` means one instance for the entire application,
while `ActivityScope` means one instance per Activity lifecycle.

```
// Singleton scope example
@Singleton
class UserRepository @Inject constructor()
```

Ссылка на видео этого руководства на сайте: borisproit.expert

Зачем нужны scopes?
Why are scopes needed?

Scopes нужны для управления временем жизни объектов и повторным использованием экземпляров.
Они предотвращают создание лишних объектов, экономят ресурсы и позволяют контролировать, где именно должна использоваться одна и та же зависимость.

Без scope каждый раз создавался бы новый экземпляр, что может привести к лишнему потреблению памяти и неконсистентному состоянию.

Scopes are needed to control object lifetimes and instance reuse.
They prevent unnecessary object creation, save resources, and define where the same dependency instance should be reused.

Without scopes, a new instance would be created every time, which may lead to wasted memory and inconsistent state.

Что такое Singleton scope?
What is Singleton scope?

Singleton scope означает, что в рамках всего приложения создаётся **только один экземпляр** зависимости.
Этот экземпляр переиспользуется везде, где он запрашивается.

Обычно применяется для объектов, которые должны существовать всё время жизни приложения: базы данных, сетевые клиенты, репозитории.

Singleton scope means that **only one instance** of a dependency is created for the entire application.
The same instance is reused everywhere it is requested.

It is commonly used for objects that should live for the entire app lifecycle: databases, network clients, repositories.

Когда singleton вреден?
When is a singleton harmful?

Singleton вреден, когда объект хранит состояние, зависящее от экрана или пользователя.

Ссылка на видео этого руководства на сайте: borisproit.expert

В таком случае состояние может протекать между разными частями приложения и вызывать трудноуловимые баги.
Также singleton усложняет тестирование, если он используется как глобальная точка доступа.

Singleton подходит для stateless или инфраструктурных объектов, но не для UI-state или краткоживущих зависимостей.

A singleton is harmful when the object holds state that depends on a screen or a user.
In that case, state may leak across different parts of the app and cause subtle bugs.
It also makes testing harder if used as a global access point.

Singletons are suitable for stateless or infrastructure objects, but not for UI state or short-lived dependencies.

Что такое Activity scope?
What is Activity scope?

Activity scope означает, что зависимость создаётся один раз на время жизни конкретной Activity и переиспользуется внутри неё.
Когда Activity уничтожается, scoped-объект тоже уничтожается.

Это удобно для зависимостей, которые должны жить столько же, сколько экран, но не дольше.

Activity scope means that a dependency is created once per Activity lifecycle and reused within that Activity.
When the Activity is destroyed, the scoped object is also destroyed.

This is useful for dependencies that should live as long as the screen, but no longer.

```
// Activity-scoped dependency (Hilt example)
@ActivityScoped
class ScreenTracker @Inject constructor()
```

Что такое ViewModel scope?
What is ViewModel scope?

Ссылка на видео этого руководства на сайте: borisproit.expert

ViewModel scope означает, что зависимость создаётся один раз на время жизни конкретной ViewModel и переиспользуется внутри неё.
Когда ViewModel очищается (`onCleared()`), scoped-объект тоже уничтожается.

Это удобно для зависимостей, которые должны жить столько же, сколько ViewModel, например, UseCase или state-holder.

ViewModel scope means that a dependency is created once per ViewModel lifecycle and reused within that ViewModel.
When the ViewModel is cleared (`onCleared()`), the scoped object is also destroyed.

This is useful for dependencies that should live as long as the ViewModel, such as a UseCase or a state holder.

```
// ViewModel-scoped dependency (Hilt example)
@ViewModelScoped
class LoadUserUseCase @Inject constructor()
```

Почему важно соответствие lifecycle и scope?
Why is matching lifecycle and scope important?

Важно, чтобы время жизни зависимости совпадало с временем жизни компонента, который её использует.
Если scope слишком длинный — возможны утечки памяти и «протекание» состояния.
Если слишком короткий — объект будет пересоздаваться лишний раз, теряя состояние и ресурсы.

Правильное соответствие lifecycle и scope делает поведение приложения предсказуемым и безопасным.

It is important that a dependency's lifetime matches the lifecycle of the component that uses it.
If the scope is too long, it may cause memory leaks or state leakage.
If it is too short, the object may be recreated unnecessarily, losing state and wasting resources.

Proper lifecycle-scope alignment makes the application predictable and safe.

Ссылка на видео этого руководства на сайте: borisproit.expert

🔴 ADVANCED

Как DI улучшает тестируемость архитектуры?
How does DI improve architectural testability?

DI улучшает тестируемость, потому что зависимости можно легко подменить на mock или fake-реализации.
Классы не создают свои зависимости сами, поэтому их можно тестировать изолированно, контролируя входные данные и поведение.

Это позволяет писать unit-тесты без реальной базы данных, сети или Android-компонентов.

DI improves testability because dependencies can be easily replaced with mocks or fake implementations.
Since classes do not create their own dependencies, they can be tested in isolation while controlling inputs and behavior.

This enables writing unit tests without real databases, network calls, or Android components.

Ссылка на видео этого руководства на сайте: borisproit.expert

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

// Production implementation
class RealUserRepository : UserRepository {
    override fun getUser() = "Real user"
}

// Service depends on abstraction
class UserService(private val repository: UserRepository) {
    fun load() = repository.getUser()
}

// Test fake implementation
class FakeUserRepository : UserRepository {
    override fun getUser() = "Test user"
}

// In test
val service = UserService(FakeUserRepository())
```

Почему DI облегчает модульность?
Why does DI improve modularity?

DI облегчает модульность, потому что модули зависят от абстракций, а не от конкретных реализаций.
Каждый модуль можно разрабатывать, тестировать и заменять независимо, не изменяя другие части системы.

Это снижает связанность между модулями и делает архитектуру более расширяемой.

DI improves modularity because modules depend on abstractions rather than concrete implementations.
Each module can be developed, tested, and replaced independently without modifying other parts of the system.

Ссылка на видео этого руководства на сайте: borisproit.expert

This reduces coupling between modules and makes the architecture more extensible.

```
// Core module (defines abstraction)
interface UserRepository {
    fun getUser()
}

// Data module (provides implementation)
class UserRepositoryImpl : UserRepository {
    override fun getUser() { }
}

// Feature module depends only on abstraction
class UserService(private val repository: UserRepository)
```

Как DI помогает заменять реализации?
How does DI help replace implementations?

DI помогает заменять реализации, потому что класс зависит от абстракции (интерфейса), а конкретная реализация передаётся извне.
Чтобы поменять поведение, достаточно передать другую реализацию, не изменяя код самого класса.

Это позволяет легко переключаться между production, debug и test-реализациями.

DI helps replace implementations because a class depends on an abstraction (an interface), while the concrete implementation is provided externally.
To change behavior, you only need to supply a different implementation without modifying the class itself.

This makes it easy to switch between production, debug, and test implementations.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Abstraction
interface Logger {
    fun log(message: String)
}

// Production implementation
class FileLogger : Logger {
    override fun log(message: String) { }
}

// Debug implementation
class ConsoleLogger : Logger {
    override fun log(message: String) { }
}

// Class depends on abstraction
class UserService(private val logger: Logger)
```

Что такое binding?
What is binding?

Binding — это связывание абстракции (например, интерфейса) с конкретной реализацией в DI-графе.
Он говорит DI-контейнеру, какую реализацию использовать, когда запрашивается определённый тип.

Binding is the association between an abstraction (such as an interface) and a concrete implementation in the dependency graph.
It tells the DI container which implementation to provide when a specific type is requested.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Abstraction
interface UserRepository

// Implementation
class UserRepositoryImpl @Inject constructor() : UserRepository

// Binding in Hilt
@Module
@InstallIn(SingletonComponent::class)
abstract class AppModule {

    @Binds
    abstract fun bindUserRepository(
        impl: UserRepositoryImpl
    ): UserRepository
}
```

Что такое interface binding?
What is interface binding?

Interface binding — это привязка интерфейса к его конкретной реализации в DI-графе.
Он определяет, какой класс должен быть создан, когда запрашивается интерфейс.

Это позволяет зависеть от абстракции, а не от конкретной реализации.

Interface binding is the mapping of an interface to its concrete implementation in the dependency graph.
It defines which class should be created when the interface is requested.

This allows depending on abstractions rather than concrete implementations.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Abstraction
interface PaymentProcessor {
    fun pay(amount: Double)
}

// Implementation
class StripeProcessor @Inject constructor() : PaymentProcessor {
    override fun pay(amount: Double) { }
}

// Binding in Hilt
@Module
@InstallIn(SingletonComponent::class)
abstract class PaymentModule {

    @Binds
    abstract fun bindPaymentProcessor(
        impl: StripeProcessor
    ): PaymentProcessor
}
```

Когда использовать multiple implementations?
When should you use multiple implementations?

Multiple implementations используют, когда одна абстракция должна иметь **разное поведение в разных сценариях**.
Например: разные источники данных (remote/local), разные способы оплаты, debug и production логгеры, feature-флаги.

Это позволяет менять стратегию без изменения клиентского кода.

Multiple implementations are used when one abstraction needs **different behavior in different scenarios**.
For example: different data sources (remote/local), different payment methods, debug vs production loggers, feature flags.

This allows switching strategies without changing client code.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Abstraction
interface DataSource {
    fun load()
}

// Implementation 1
class RemoteDataSource : DataSource {
    override fun load() { }
}

// Implementation 2
class LocalDataSource : DataSource {
    override fun load() { }
}

// Client depends on abstraction
class Repository(private val dataSource: DataSource)
```

Что такое qualifiers?
What are qualifiers?

Qualifiers — это аннотации в DI, которые позволяют различать **несколько реализаций одного и того же типа**.
Они используются, когда в dependency graph есть несколько binding'ов одного интерфейса, и нужно указать, какой именно внедрить.

Qualifiers are DI annotations used to distinguish **multiple implementations of the same type**.
They are needed when the dependency graph contains multiple bindings for the same interface, and you must specify which one to inject.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Qualifiers
@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class Remote

@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class Local

// Implementations
class RemoteDataSource @Inject constructor() : DataSource
class LocalDataSource @Inject constructor() : DataSource

// Providing with qualifiers
@Module
@InstallIn(SingletonComponent::class)
object DataModule {

    @Provides
    @Remote
    fun provideRemote(): DataSource = RemoteDataSource()

    @Provides
    @Local
    fun provideLocal(): DataSource = LocalDataSource()
}
```

Что такое lazy injection?
What is lazy injection?

Lazy injection — это способ внедрения зависимости, при котором объект **не создаётся сразу**, а инициализируется только в момент первого использования.
Это помогает избежать лишнего создания тяжёлых объектов и может ускорить старт приложения.

Ссылка на видео этого руководства на сайте: borisproit.expert

Lazy injection is a way of injecting a dependency where the object is **not created immediately**, but only when it is first accessed.
This helps avoid unnecessary creation of heavy objects and can improve startup performance.

```
// Hilt / Dagger lazy injection
class UserService @Inject constructor(
    private val repository: dagger.Lazy<UserRepository>
) {

    fun load() {
        val repo = repository.get() // created on first access
        repo.getUser()
    }
}
```

Когда lazy полезен?
When is lazy useful?

Lazy полезен, когда зависимость тяжёлая или используется не всегда.
Он откладывает создание объекта до первого обращения, что снижает время старта и экономит ресурсы.
Также полезен для разрыва циклических зависимостей.

Lazy is useful when a dependency is heavy or not always needed.
It delays object creation until first use, reducing startup time and saving resources.
It is also helpful for breaking circular dependencies.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Lazy injection example
class AnalyticsService @Inject constructor()

class UserService @Inject constructor(
    private val analytics: dagger.Lazy<AnalyticsService>
) {

    fun performAction() {
        if (someCondition) {
            analytics.get().track() // created only if needed
        }
    }
}
```

Когда lazy вреден?
When is lazy harmful?

Lazy вреден, когда зависимость должна быть доступна сразу и её отложенная инициализация может привести к неожиданным задержкам или ошибкам.
Также lazy может скрывать реальные зависимости класса и усложнять понимание кода.
Если зависимость используется всегда — откладывать её создание нет смысла.

Lazy стоит применять осознанно, а не по умолчанию.

Lazy is harmful when a dependency is required immediately and delayed initialization may cause unexpected latency or runtime issues.
It can also hide real dependencies and make the code harder to understand.
If a dependency is always used, there is no benefit in delaying its creation.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// ❌ Unnecessary lazy: dependency always used
class UserService @Inject constructor(
    private val repository: dagger.Lazy<UserRepository>
) {

    fun load() {
        repository.get().getUser() // always accessed anyway
    }
}
```

🔴 COMPOSE / MODERN ANDROID

Нужно ли внедрять зависимости в `Composable`?
Should you inject dependencies into a `Composable`?

Обычно нет. `Composable` должен быть максимально простым и получать данные/колбэки через параметры.
Зависимости лучше внедрять во `ViewModel` (или в слой выше), а в `Composable` передавать уже готовое состояние и события.
Исключение — получение `ViewModel` через `hiltViewModel()`, это нормально, потому что `Composable` не создаёт зависимости вручную.

Usually no. A `Composable` should stay simple and receive state and callbacks via parameters.
Dependencies should be injected into the `ViewModel` (or higher layers), and the `Composable` should only consume state and events.
An exception is getting a `ViewModel` via `hiltViewModel()`, which is fine because the `Composable` is not manually creating dependencies.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// ✅ Preferred: Composable gets state and callbacks
@Composable
fun UserScreen(
    state: UserUiState,
    onRefresh: () -> Unit
) {
    // UI code
}

// ✅ Acceptable: obtain ViewModel via Hilt
@Composable
fun UserRoute(
    viewModel: UserViewModel = hiltViewModel()
) {
    val state = viewModel.state.collectAsState().value
    UserScreen(
        state = state,
        onRefresh = { viewModel.refresh() } // callback to ViewModel
    )
}
```

Где должен происходить injection в Compose?
Where should injection happen in Compose?

Injection должен происходить **на границе UI и бизнес-логики**, обычно во `ViewModel`.
`Composable` должен получать уже готовый `ViewModel` или state через параметры.
Чаще всего injection происходит через `@HiltViewModel` и `hiltViewModel()` в верхнем уровне экрана (route-слой).

Injection should happen **at the boundary between UI and business logic**, typically in the `ViewModel`.
A `Composable` should receive an already provided `ViewModel` or state via parameters.
In most cases, injection is done with `@HiltViewModel` and `hiltViewModel()` at the top-level screen (route layer).

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// ViewModel with DI
@HiltViewModel
class UserViewModel @Inject constructor(
    private val repository: UserRepository
) : ViewModel()

// Route layer: injection happens here
@Composable
fun UserRoute(
    viewModel: UserViewModel = hiltViewModel()
) {
    val state = viewModel.state.collectAsState().value
    UserScreen(state = state)
}

// Pure UI layer: no injection here
@Composable
fun UserScreen(state: UserUiState) {
    // UI only
}
```

Можно ли использовать DI для state holders?
Can DI be used for state holders?

Да, можно — но важно правильно выбрать scope.
State holder можно внедрять через DI, если его жизненный цикл совпадает с компонентом (например, `ViewModelScoped`).
Но UI-state внутри `Composable` обычно не внедряют через DI — он должен управляться `ViewModel` или через `remember`.

DI подходит для бизнес-state,

но не для локального UI-state внутри composable.

Yes, you can — but the scope must match the lifecycle.
A state holder can be injected if its lifecycle matches the component (for example, `ViewModelScoped`).

Ссылка на видео этого руководства на сайте: borisproit.expert

However, UI state inside a `Composable` is usually not injected via DI — it should be managed by a `ViewModel` or with `remember`.

```
// ViewModel-scoped state holder
@ViewModelScoped
class UserStateHolder @Inject constructor()

@HiltViewModel
class UserViewModel @Inject constructor(
    private val stateHolder: UserStateHolder
) : ViewModel()
```

Как DI помогает при тестировании Compose?
How does DI help when testing Compose?

DI помогает, потому что ты можешь подменить реальные зависимости на fake/mock и тестировать UI в изоляции.
В Compose правильный подход — держать `Composable` “чистыми”: передавать `state` и callbacks параметрами. Тогда UI-тесты не зависят от сети, базы, репозиториев и Hilt.

Если нужен тест Route-слоя с `hiltViewModel()`, DI позволяет заменить binding’и на тестовые и получить ViewModel с нужными зависимостями.

DI helps because you can replace real dependencies with fakes/mocks and test UI in isolation.
In Compose, the recommended approach is to keep Composables “pure” by passing `state` and callbacks as parameters. This makes UI tests independent from network, database, repositories, and Hilt.

If you test the Route layer that uses `hiltViewModel()`, DI allows replacing bindings with test ones and getting a ViewModel with the desired dependencies.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// ✅ Pure UI: easy to test (no DI needed here)
@Composable
fun UserScreen(
    state: UserUiState,
    onRefresh: () -> Unit
) {
    // UI code
}

// ✅ Test can pass fake state and callbacks
val fakeState = UserUiState(name = "Test")
UserScreen(
    state = fakeState,
    onRefresh = { /* fake action */ }
)
```

🔴 ARCHITECTURAL THINKING

Где должен жить DI слой?
Where should the DI layer live?

DI слой должен жить на уровне композиции приложения — там, где собираются модули и связываются реализации.
Обычно это отдельный `di` пакет/модуль рядом с `app`, а для многомодульной архитектуры — в `app` (composition root) плюс локальные DI-модули внутри feature/data модулей.

Главная идея: DI — это “wiring” слой, он не должен смешиваться с UI и бизнес-логикой.

The DI layer should live at the application composition root — where modules are assembled and implementations are wired together.
Typically it’s a separate `di` package/module in the `app` layer, and in a multi-module setup it’s the `app` module (composition root) plus local DI modules inside feature/data modules.

Key idea: DI is a wiring layer and should not be mixed with UI or business logic.

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
// app/di/AppModule.kt
@Module
@InstallIn(SingletonComponent::class)
object AppModule {

    @Provides
    fun provideApi(): ApiService = ApiService()
}
```

Должен ли Domain слой зависеть от DI?
Should the Domain layer depend on DI?

Нет. Domain слой не должен зависеть от DI-фреймворков и аннотаций.
Он должен содержать чистую бизнес-логику и контракты (use cases, интерфейсы репозиториев), чтобы быть независимым от Android и инфраструктуры.

DI находится во внешнем слое (обычно app/data) и связывает реализации с доменными интерфейсами.

```kotlin
// Domain: no DI annotations
interface UserRepository {
    fun getUser(): User
}

class GetUserUseCase(private val repository: UserRepository) {
    fun execute() = repository.getUser()
}

// Data/App: DI wires implementation to the interface
class UserRepositoryImpl : UserRepository {
    override fun getUser(): User = User()
}
```

Почему DI не должен «протекать» в бизнес-логику?
Why should DI not “leak” into business logic?

Потому что бизнес-логика должна быть независимой от инфраструктуры.
Если в Domain появляются аннотации DI или зависимость от фреймворка, слой перестаёт быть чистым и теряет переносимость, тестируемость и изоляцию.

Ссылка на видео этого руководства на сайте: borisproit.expert

Domain должен знать только про контракты и свою логику.
Создание и связывание зависимостей — ответственность внешнего слоя (composition root).

Business logic must remain independent from infrastructure.
If DI annotations or framework dependencies appear in the Domain layer, it stops being clean and loses portability, testability, and isolation.

The Domain layer should only know about contracts and its own logic.
Dependency creation and wiring belong to the outer layer (composition root).

```
// ❌ Bad: DI leaking into Domain
@Singleton
class GetUserUseCase @Inject constructor(
    private val repository: UserRepository
)

// ✅ Good: pure Domain
class GetUserUseCase(
    private val repository: UserRepository
)
```

Как DI связан с Clean Architecture?
How is DI related to Clean Architecture?

DI помогает реализовать Clean Architecture, потому что позволяет внешним слоям подключать реализации к внутренним контрактам.
Внутренние слои (Domain) объявляют интерфейсы и use case'ы, а внешние слои (Data/App) предоставляют конкретные реализации и выполняют wiring в composition root.

Так сохраняется правило зависимостей: зависимости направлены внутрь, а детали остаются снаружи.

DI supports Clean Architecture by letting outer layers provide implementations for inner-layer contracts.
Inner layers (Domain) define interfaces and use cases, while outer layers (Data/App) provide concrete implementations and perform wiring at the composition root.

Ссылка на видео этого руководства на сайте: borisproit.expert

This preserves the dependency rule: dependencies point inward, while details stay on the outside.

Может ли DI нарушить архитектуру?
Can DI break architecture?

Да, может — если использовать его неправильно.
Если DI-аннотации и фреймворк-зависимости попадают в Domain-слой, нарушается принцип изоляции и правило зависимостей.
Также архитектура ломается, если через DI начинают передавать всё подряд, скрывая реальные зависимости или создавая глобальные синглтоны без контроля lifecycle.

DI — это инструмент.
Если wiring смешивается с бизнес-логикой или слои начинают зависеть от фреймворка — архитектура деградирует.

Yes, it can — if used incorrectly.
If DI annotations and framework dependencies leak into the Domain layer, isolation and the dependency rule are violated.
Architecture can also degrade if DI is used to inject everything blindly or to create uncontrolled global singletons.

DI is a tool.
If wiring is mixed with business logic or layers start depending on the DI framework, the architecture breaks.

```
// ❌ DI leaking into Domain
@Singleton
class GetUserUseCase @Inject constructor(
    private val repository: UserRepository
)

// ✅ Clean Domain
class GetUserUseCase(
    private val repository: UserRepository
)
```

Как DI может привести к overengineering?
How can DI lead to overengineering?

Ссылка на видео этого руководства на сайте: borisproit.expert

DI приводит к overengineering, когда его используют там, где он не нужен:
— маленький проект с простым графом зависимостей
— создание интерфейса для каждого класса “на всякий случай”
— избыточные модули, scope’ы и binding’и без реальной необходимости

Это увеличивает boilerplate, усложняет навигацию по коду и повышает порог входа.

DI должен решать проблему, а не добавлять архитектурную сложность ради «правильности».

DI leads to overengineering when it is used where it is not needed:
— small projects with simple dependency graphs
— creating an interface for every class “just in case”
— excessive modules, scopes, and bindings without real necessity

This increases boilerplate, makes code harder to navigate, and raises the learning curve.

DI should solve a real problem, not add architectural complexity for the sake of being “proper”.

🟣 HILT PRACTICAL

Зачем нужна аннотация `@HiltAndroidApp`?
Why is the `@HiltAndroidApp` annotation needed?

`@HiltAndroidApp` ставится на класс `Application` и служит точкой входа для Hilt.
Она запускает генерацию кода, создаёт базовый `SingletonComponent` и инициализирует dependency graph всего приложения.
Без этой аннотации Hilt не сможет работать.

`@HiltAndroidApp` is placed on the `Application` class and acts as the entry point for Hilt.
It triggers code generation, creates the base `SingletonComponent`, and initializes the app-wide dependency graph.
Without this annotation, Hilt will not work.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Application class with Hilt enabled
@HiltAndroidApp
class MyApp : Application()
```

Что такое Hilt Module?
What is a Hilt Module?

Hilt Module — это класс, в котором объявляются правила создания зависимостей.
В модуле указывается, как именно Hilt должен предоставить объект: через `@Provides` или `@Binds`.
Модуль подключается к определённому компоненту через `@InstallIn`.

A Hilt Module is a class where dependency creation rules are defined.
It tells Hilt how to provide an object using `@Provides` or `@Binds`.
The module is attached to a specific component using `@InstallIn`.

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {

    @Provides
    fun provideApiService(): ApiService {
        return ApiService() // object creation rule
    }
}
```

Зачем используется `@Provides`?
Why is `@Provides` used?

`@Provides` используется, когда зависимость нельзя создать через constructor injection (например, класс из сторонней библиотеки или требуется особая логика создания).
Он указывает Hilt, как именно создать объект.

Ссылка на видео этого руководства на сайте: borisproit.expert

`@Provides` is used when a dependency cannot be created via constructor injection (for example, a third-party class or when custom creation logic is needed).
It tells Hilt exactly how to create the object.

```
@Module
@InstallIn(SingletonComponent::class)
object NetworkModule {

    @Provides
    fun provideRetrofit(): Retrofit {
        return Retrofit.Builder()
            .baseUrl("https://api.example.com")
            .build()
    }
}
```

Что такое Entry Point в Hilt?
What is an Entry Point in Hilt?

Entry Point в Hilt — это способ получить зависимости из Hilt-контейнера в классах, которые Hilt не может создать автоматически.
Обычно используется в случаях, когда объект создаётся системой или сторонним кодом (например, ContentProvider, BroadcastReceiver до инициализации Hilt).

An Entry Point in Hilt is a way to access dependencies from the Hilt container in classes that Hilt cannot create automatically.
It is typically used when an object is created by the system or third-party code (for example, a ContentProvider or early-initialized component).

Когда класс **создаётся не Hilt**, например:

- `ContentProvider`
- сторонняя библиотека
- класс, созданный системой

В этих случаях нельзя использовать `@Inject` или `@AndroidEntryPoint`.

Ссылка на видео этого руководства на сайте: borisproit.expert

```kotlin
// Define entry point
@EntryPoint
@InstallIn(SingletonComponent::class)
interface AppEntryPoint {
    fun apiService(): ApiService
}

// Accessing dependency manually
val entryPoint = EntryPointAccessors.fromApplication(
    context,
    AppEntryPoint::class.java
)

val api = entryPoint.apiService()
```

Что такое Hilt Components?
What are Hilt Components?

Hilt Components — это автоматически создаваемые контейнеры зависимостей, которые привязаны к lifecycle Android-компонентов.
Они определяют, где и как долго живут зависимости (scope) и какие объекты могут их получать.

Например:
`SingletonComponent` — живёт всё время приложения,
`ActivityComponent` — живёт столько же, сколько Activity,
`ViewModelComponent` — живёт столько же, сколько ViewModel.

Hilt Components are automatically generated dependency containers tied to Android lifecycles.
They define where dependencies live (scope) and which objects can access them.

For example:
`SingletonComponent` — lives for the entire application,
`ActivityComponent` — lives as long as the Activity,
`ViewModelComponent` — lives as long as the ViewModel.

Ссылка на видео этого руководства на сайте: borisproit.expert

```
@Module
@InstallIn(ViewModelComponent::class)
object ViewModelModule {

    @Provides
    fun provideUseCase(): LoadUserUseCase {
        return LoadUserUseCase()
    }
}
```

🧠 ARCHITECTURAL IMPACT

В чём разница между созданием зависимости и её использованием?
What is the difference between creating a dependency and using it?

Создание зависимости — это процесс её инициализации: выбор реализации, конфигурация, управление временем жизни.
Использование зависимости — это применение уже готового объекта для выполнения логики.

В Clean Architecture создание происходит во внешнем слое (composition root / DI),
а использование — во внутренних слоях (ViewModel, UseCase, Service).

Creating a dependency means instantiating and configuring it, deciding which implementation to use and managing its lifecycle.
Using a dependency means consuming an already created object to perform business logic.

In Clean Architecture, creation happens in the outer layer (composition root / DI),
while usage happens in inner layers (ViewModel, UseCase, Service).

Ссылка на видео этого руководства на сайте: borisproit.expert

```
// Creation (composition root)
val repository = UserRepository()
val useCase = GetUserUseCase(repository)

// Usage (business logic)
class GetUserUseCase(
    private val repository: UserRepository
) {
    fun execute() = repository.getUser()
}
```

Почему DI помогает разделять ответственность?
Why does DI help separate responsibilities?

DI помогает разделять ответственность, потому что класс больше не отвечает за создание своих зависимостей.
Он отвечает только за свою бизнес-логику, а создание и конфигурация объектов выносятся во внешний слой.

Это соответствует принципу Single Responsibility:
— бизнес-класс выполняет логику,
— DI-слой управляет созданием и связыванием зависимостей.

DI helps separate responsibilities because a class no longer creates its own dependencies.
It focuses only on its business logic, while object creation and configuration are handled externally.

This aligns with the Single Responsibility Principle:
— the business class handles logic,
— the DI layer handles object creation and wiring.

Как DI помогает масштабируемости приложения?
How does DI improve application scalability?

DI помогает масштабируемости, потому что новые реализации и модули можно добавлять без изменения существующего кода.
Классы зависят от абстракций, поэтому систему можно расширять, не ломая уже написанную логику.

Ссылка на видео этого руководства на сайте: borisproit.expert

Также DI упрощает разделение приложения на модули: каждый модуль предоставляет свои реализации, а связывание происходит централизованно.

DI improves scalability because new implementations and modules can be added without modifying existing code.
Classes depend on abstractions, so the system can be extended without breaking existing logic.

DI also makes modularization easier: each module provides its own implementations, while wiring happens centrally.

Почему DI делает код более гибким?
Why does DI make code more flexible?

DI делает код гибким, потому что классы зависят от абстракций, а не от конкретных реализаций.
Поведение можно менять, подставляя другую реализацию, не изменяя сам класс.

Это позволяет легко переключать источники данных, стратегии, окружения (debug/prod), не переписывая бизнес-логику.

DI makes code flexible because classes depend on abstractions rather than concrete implementations.
Behavior can be changed by supplying a different implementation without modifying the class itself.

This makes it easy to switch data sources, strategies, or environments (debug/prod) without rewriting business logic.

Как DI помогает переиспользованию кода?
How does DI help with code reuse?

DI помогает переиспользованию, потому что классы не привязаны к конкретным реализациям.
Они зависят от абстракций, поэтому могут использоваться в разных проектах, модулях и окружениях без изменений.

Это позволяет переносить бизнес-логику между Android, backend или тестовой средой, просто подставляя другую реализацию зависимости.

Ссылка на видео этого руководства на сайте: borisproit.expert

Почему DI уменьшает дублирование кода?
Why does DI reduce code duplication?

DI уменьшает дублирование, потому что создание зависимостей централизуется в одном месте.
Один и тот же объект (или правило его создания) не нужно повторять в разных классах — он предоставляется через DI.

Также переиспользуются абстракции и реализации, вместо копирования логики в нескольких местах.

DI reduces duplication because dependency creation is centralized in one place.
The same object (or its creation logic) does not need to be repeated across multiple classes — it is provided through DI.

Abstractions and implementations are reused instead of copying logic in different parts of the codebase.

Как DI помогает отделить бизнес-логику от Android framework?
How does DI help separate business logic from the Android framework?

DI позволяет вынести создание Android-зависимостей (Context, Retrofit, Room и т.д.) во внешний слой,
а бизнес-логика получает только абстракции и не знает про Android.

Domain и UseCase остаются чистыми Kotlin-классами, без зависимостей от Activity, Fragment, Context или Hilt.

DI moves Android-specific object creation (Context, Retrofit, Room, etc.) to the outer layer,
while business logic depends only on abstractions and remains unaware of Android.

Domain and UseCases stay as pure Kotlin classes without depending on Activity, Fragment, Context, or Hilt.