

The background features a complex network of thin grey lines and dots, forming a web-like structure. Scattered throughout are various triangles of different sizes and orientations, some filled with a light grey color and others as outlines. The overall aesthetic is modern and technological.

PROGRAMA DEV VENTURE

Desenvolvimento Android

AQUECIMENTO

Navegando em um
projeto

01

POO

Programação Orientada
à Objetos

02

ARQUITETURA

Motivação, modelos

03

MVVM

& Guia android de
arquitetura

04

ARCHITECTURE COMPONENTS

Android Jetpack

05

Aula 13 & 14 AGENDA



01

AQUECIMENTO





POKEDEX

Clone o projeto

<https://github.com/mrcsxsiq/Kotlin-Pokedex>





COMO O PROJETO ESTÁ ESTRUTURADO

O que a MainActivity faz?

Qual o primeiro fragment que é aberto quando a app se inicializa?

Ao clicar no botão Pokedex, de onde os dados dos pokémons vem?

Existe algum banco de dados na aplicação?





COMO VOCÊ FARIA?

Faça um pequeno teste: rode a aplicação com o emulador no modo avião.
Clique no botão "Pokedex"
O que acontece?





COMO VOCÊ FARIA?

Queremos melhorar este comportamento:

Dado que um erro de conexão aconteceu, queremos que um Toast apareça avisando ao usuário.





DESAFIO

Implemente o Toast que será mostrado no caso de erro de conexão.



DESAFIO

```
private var _showError = MutableLiveData<Boolean>()
val showError = _showError
```

```
override fun onFailure(call: Call<List<Pokemon>?>?, t: Throwable?) {
    _showError.value = true
}
```

```
pokedexViewModel.showError.observe(viewLifecycleOwner, Observer { it: Boolean
    if(it) {
        Toast
            .makeText(
                context,
                "There is a network problem, try again later",
                Toast.LENGTH_LONG
            ).show()
    }
})
```

02

P00



PROGRAMAÇÃO OO

Objeto: coisa material ou abstrata que pode ser percebida pelos sentidos e descrita por meio das suas características, comportamento e estado atual.





QUAIS OBJETOS EXISTEM NO PROJETO?

**descrita por meio das suas características,
comportamento e estado atual.**





CLASSES

É um modelo de código de programa extensível para criar objetos, fornecendo valores iniciais para o estado (variáveis de membro) e implementações de comportamento (funções ou métodos de membro)





DATA CLASS PAGE

Procure no código a Data Class Page
Quais parâmetros são necessários para criar um objeto Page?
Se existisse um título padrão para as Pages, como poderíamos defini-lo?





SOLID

O **S.O.L.I.D** é um acrônimo que representa cinco princípios da programação orientada a objetos e design de código teorizados Robert C. Martin. Michael Feathers foi responsável pela criação do acrônimo.

[S]ingle Responsibility Principle (Princípio da Responsabilidade Única)

[O]pen/Closed Principle (Princípio do Aberto/Fechado)

[L]iskov Substitution Principle (Princípio da Substituição de Liskov)

[I]nterface Segregation Principle (Princípio da Segregação de Interfaces)

[D]ependency Inversion Principle (Princípio da Inversão de Dependências)





SOLID

Os princípios do SOLID não são totalmente isolados. Existe uma conexão bem forte entre eles, criando uma espécie de "efeito dominó" quando aplicamos ou infringimos.

Você vai perceber que muitas das soluções existentes para cumprir um princípio, acaba aplicando também outros





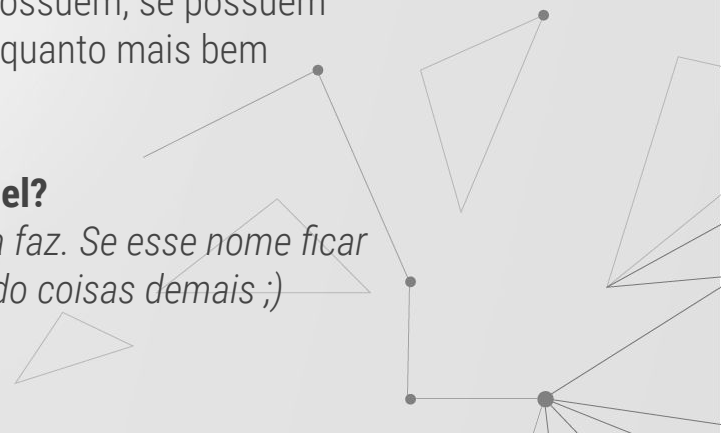
RESPONSABILIDADE ÚNICA

“uma classe, método, componente deve ter apenas um motivo para mudar”

Basicamente, esse princípio trata especificamente a coesão. A coesão é definida como a afinidade funcional dos elementos de um módulo. Se refere ao relacionamento que os membros desse módulo possuem, se possuem uma relação mais direta e importante. Dessa forma, quanto mais bem definido o que sua classe faz, mais coesa ela é.

Qual responsabilidade da Classe PokedexViewModel?

No nome da classe/metodo, tente descrever o que ela faz. Se esse nome ficar grande mais, talvez essa classe/metodo esteja fazendo coisas demais ;)





ABERTO/FECHADO

“Objetos abertos para extensão e fechados para modificação”

Basicamente, esse princípio trata de evitarmos modificar o que já existe para adicionar novos comportamentos.
Mas como colocar isso em prática?



ABERTO/FECHADO

```
class PokeBattle {  
    val attacks: Attacker()  
    fun attack(player: Pomekon) {  
        if(player.type == watter) {  
            attacks.throwWatterJet()  
        } else if (player.type == fire) {  
            attacks.setFireOnIt()  
        } |  
    }  
}
```

Observe o código ao lado. Desejamos adicionar um novo tipo de pokémon. **Pokémons do tipo elétrico. Como você faria esta alteração?**

ABERTO/FECHADO

```
class PokeBattle {  
    val attacks: Attacker()  
    fun attack(player: Pomekon) {  
        if(player.type == watter) {  
            attacks.throwWatterJet()  
        } else if (player.type == fire) {  
            attacks.setFireOnIt()  
        } else if (player.type == eletric) {  
            attacks.thunderAttack()  
        }  
    }  
}
```

Ao acrescentar uma nova condição na classe Battle estamos infringindo o princípio aberto/fechado. E qual o problema disso?

O risco de criarmos bugs no que já funciona aumenta.

- Adicionando mais condicionais, deixamos o código menos legível
- Aumentamos também a responsabilidade da classe Battle que agora precisa conhecer mais um tipo de pokémon
- ...

```
interface Pokemon {  
    fun attack()  
}  
  
class Fire: Pokemon {  
    override fun attack() {  
        this.setFireOnIt()  
    }  
  
    private fun setFireOnIt() {  
        ...  
    }  
}  
  
class Eletric: Pokemon {  
    override fun attack() {  
        this.thunderAttack()  
    }  
  
    override fun thunderAttack() {  
        ...  
    }  
}  
  
class Battle {  
    ...  
  
    fun playerAttack(pokemon) {  
        pokemon.attack()  
    }  
}
```

ABERTO/FECHADO

Como resolver este problema?

- Separe o comportamento extensível por trás de uma interface
- Inverta as dependências!
- Modelo mental de plugin




PRINCÍPIO DA SUBSTITUIÇÃO DE LISKOV

Uma classe derivada deve ser substituível por sua classe base sem quebrar o comportamento do software.

*Todas classes que estendem/implementam Pokemon seguem o **contrato do método attack()***

Sabemos que independente do tipo do pokémon, ele vai atacar quando chamado para a batalha

```
class Battle {  
    ...  
  
    fun playerAttack(pokemon) {  
        pokemon.attack()  
    }  
}
```





PRINCÍPIO DA SEGREGAÇÃO DE INTERFACES

Se uma classe implementa uma interface, ela deve implementar todo seu comportamento. Caso isso não seja possível, talvez a interface esteja genérica demais.





PRINCÍPIO DA INVERSÃO DE DEPENDÊNCIAS

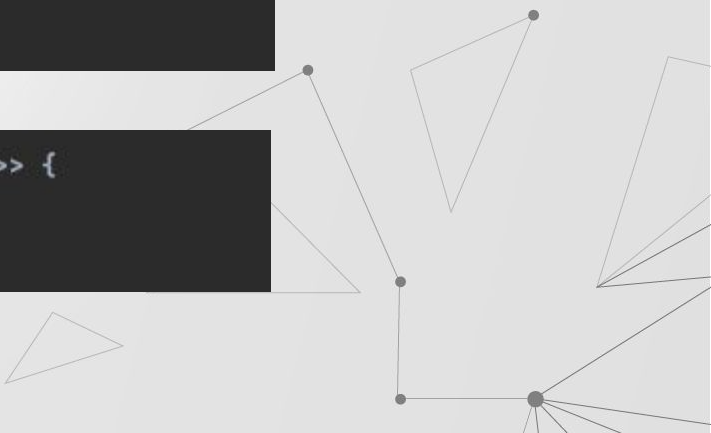
Depender de abstrações e não implementações

"Módulos de alto nível não devem depender de módulos de baixo nível"

"Abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações"

```
fun getListPokemon(): LiveData<List<Pokemon>> {  
    return pokemonDAO.all()  
}
```

```
fun getListPokemon(): LiveData<List<Pokemon>> {  
    return repository.getAll()  
}
```



02

ARQUITETURA DE SOFTWARE



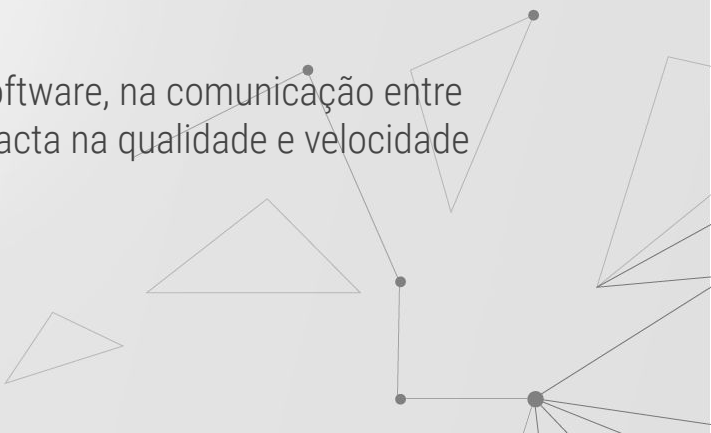


ARQUITETURA

A arquitetura de software de um sistema consiste na definição dos componentes de software, suas propriedades externas, e seus relacionamentos com outros softwares.

O projeto de arquitetura está preocupado com a compreensão de como um sistema deve ser organizado e com a estrutura geral desse sistema.

A definição de uma arquitetura facilita na organização do software, na comunicação entre as pessoas que trabalham neste software. Ela também impacta na qualidade e velocidade das entregas





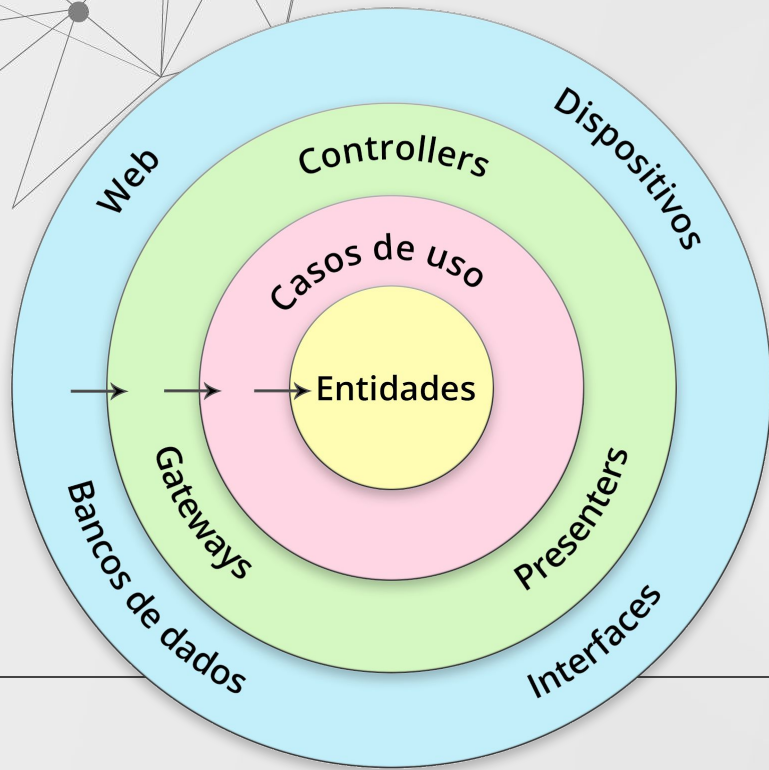
PADRÕES DE ARQUITETURA EM APPS

Existem vários padrões que são utilizados em aplicações mobile. No nosso curso focaremos em apenas um: o MVVM.

O MVVM é uma extensão de um padrão chamado Clean Architecture



CLEAN ARCHITECTURE

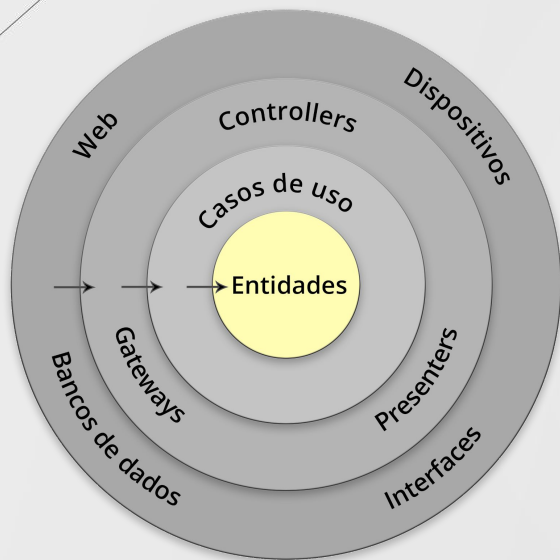


Os círculos representam diferentes áreas do software. Os círculos externos são mecanismos. Os círculos internos são políticas.

As dependências do código-fonte só podem apontar para dentro. Nada em um círculo interno pode saber absolutamente nada sobre algo em um círculo externo.

Da mesma forma, os formatos de dados usados em um círculo externo não devem ser usados por um círculo interno, especialmente se esses formatos forem gerados por uma estrutura em um círculo externo. **Não queremos que nada em um círculo externo afete os círculos internos.**

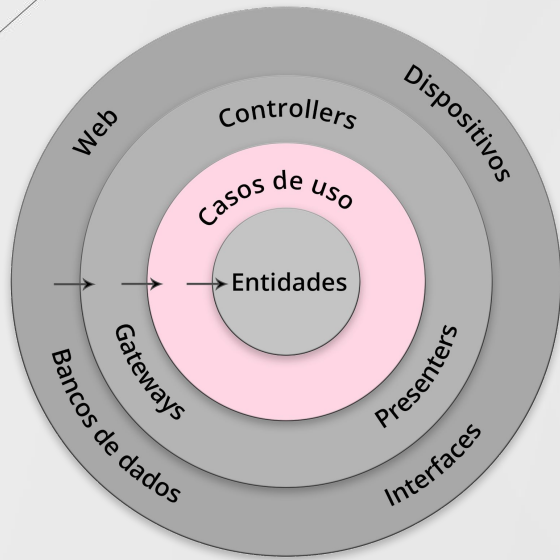
ENTIDADES



Regras de negócio do tipo

enterprise

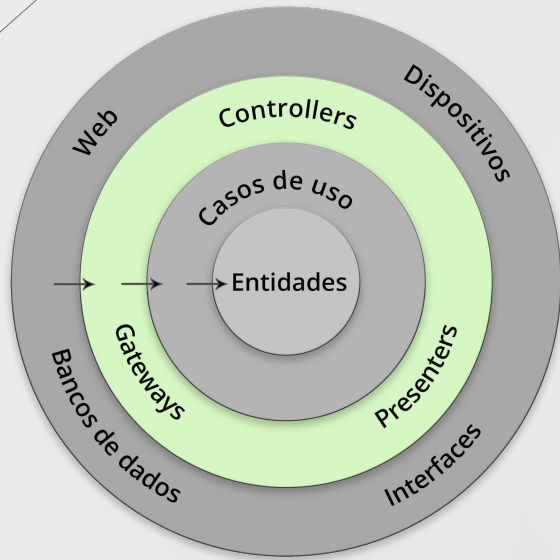
CASOS DE USO



Regras de negócio do tipo

aplicação

CONTROLLERS

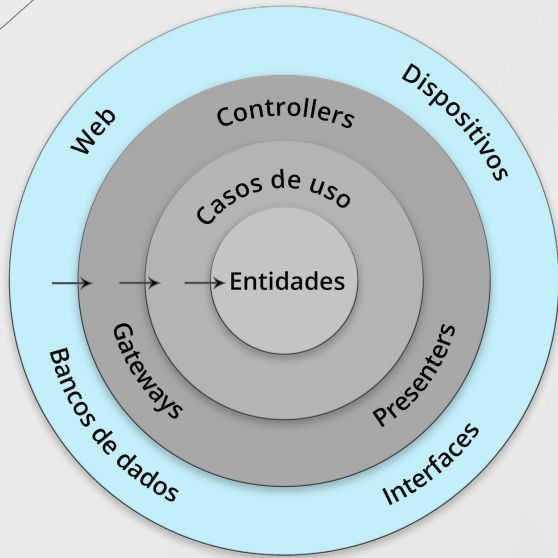


Conversores de dados para
**formatos mais
convenientes**
para interação com agentes
externos

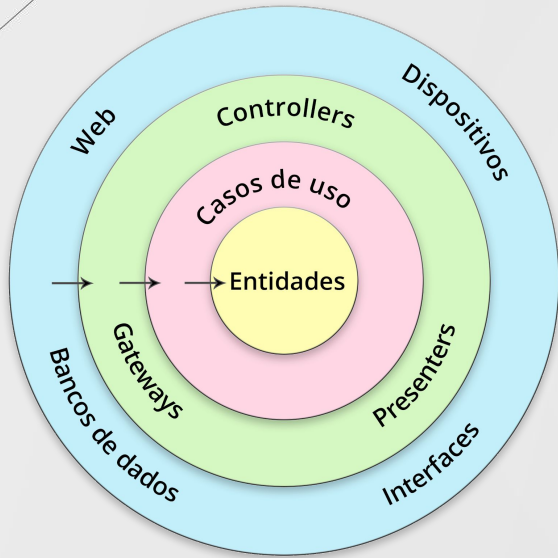
INTERFACES EXTERNAS

Chamada para frameworks,
drivers, UI, bases de dados

**sem lógica
de negócio**



REGRA DE DEPENDÊNCIA



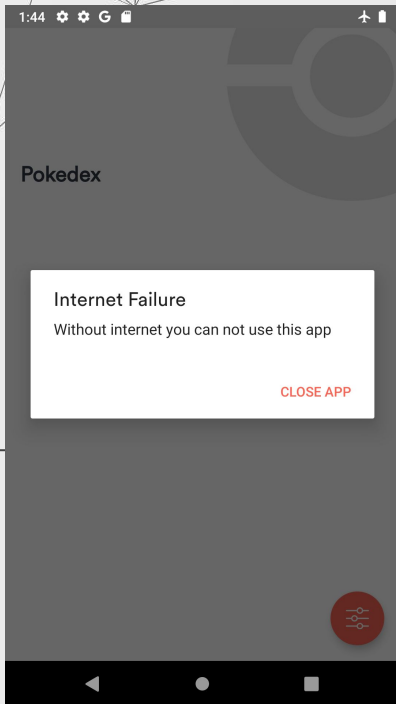
Os círculos são um esquema

Em alguns casos é necessário criar mais camadas.

Porém, existe uma única regra que nunca deve ser quebrada:

A regra de dependência.

DESAFIO



Vamos incrementar nosso projeto com um novo comportamento:

Ao invés de mostrar um Toast. Caso aconteça um erro de conexão, um alerta irá aparecer.

Dica: use um AlertDialog no lugar do toast

<https://developer.android.com/guide/topics/ui/dialogs?hl=pt-br>

04

MVVM



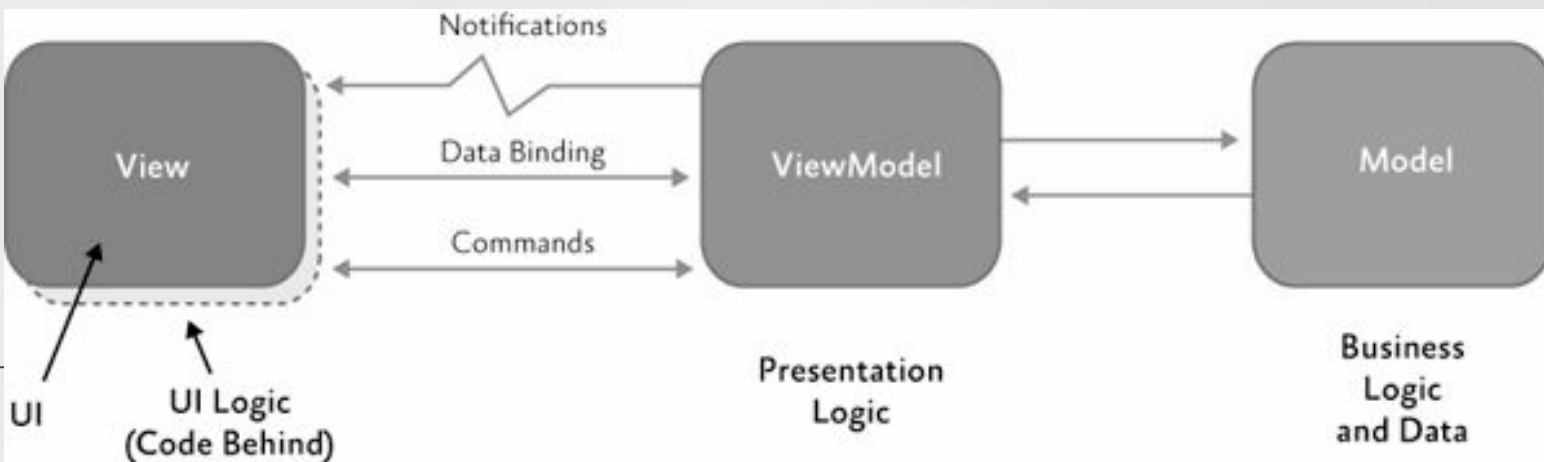


MVVM

Model-view-viewmodel (MVVM) é um padrão de arquitetura de software que facilita a separação de responsabilidades dentro de uma aplicação.



MVVM



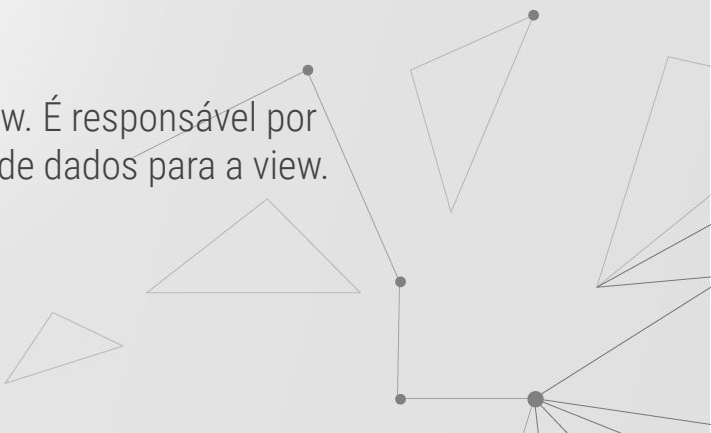


MVVM

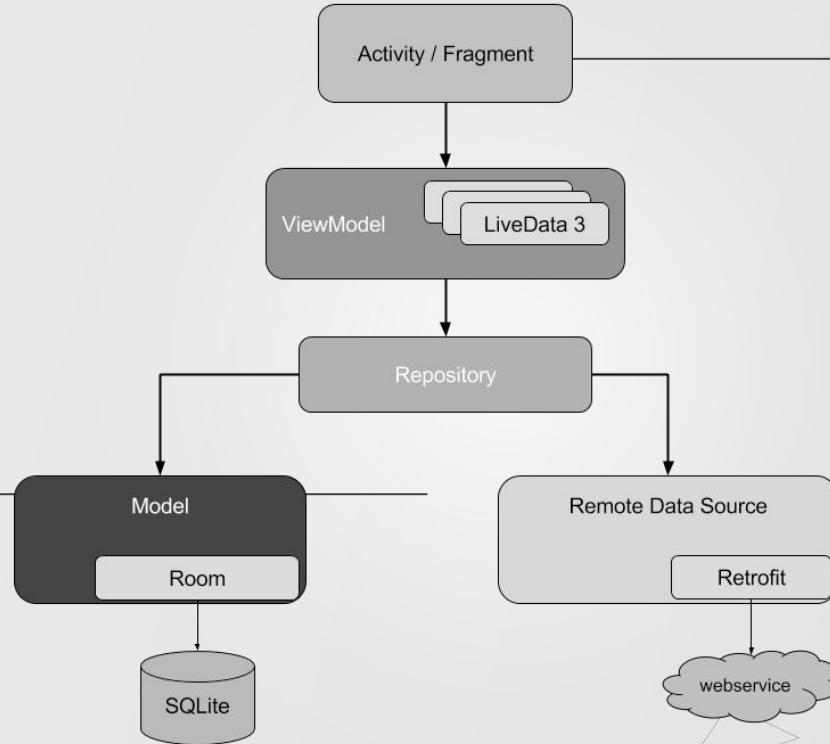
Model: contém os dados do aplicativo. Não se comunicam diretamente com as Views. Geralmente, é recomendado expor os dados ao ViewModel por meio de Observables.

View: representa a UI do aplicativo desprovida de qualquer lógica. Ela observa o ViewModel.

ViewModel: atua como um link entre o model e a view. É responsável por transformar os dados do modelo. Ele fornece fluxos de dados para a view. Usa de hooks ou callbacks atualizar a view.



GUIA ANDROID DE ARQUITETURA





GUIA ANDROID DE ARQUITETURA

Interface com usuário: as classes activities ou fragments ficam responsável por tratar a interação com usuário. Tarefas como exibir dados, capturar interações são de responsabilidade delas.



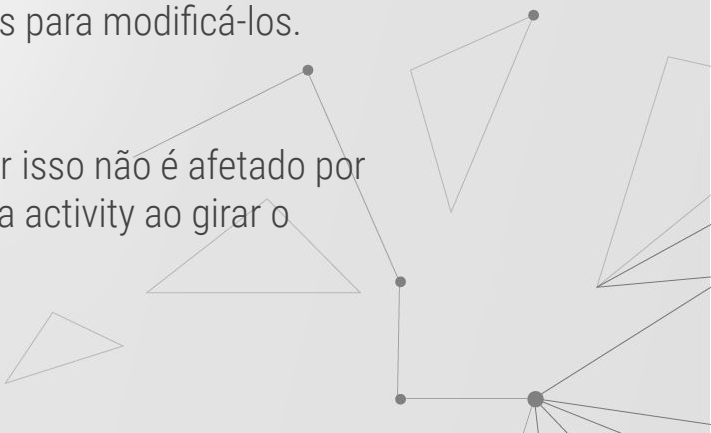


GUIA ANDROID DE ARQUITETURA

Um objeto ViewModel fornece os dados para um componente de UI específico, como um fragment ou activity, e contém lógica de manipulação de dados para se comunicar com o modelo.

Por exemplo, o ViewModel pode chamar outros componentes para carregar os dados e pode encaminhar solicitações de usuários para modificá-los.

O ViewModel não sabe sobre componentes de UI, por isso não é afetado por mudanças de configuração, como a recriação de uma activity ao girar o dispositivo.





GUIA ANDROID DE ARQUITETURA

Repositories manipulam operações de dados.

Eles disponibilizam uma interface limpa para que o restante do app possa recuperar dados com facilidade. Eles sabem onde coletar os dados e quais chamadas de API precisam ser feitas quando os dados são atualizados. Repositories são como mediadores entre fontes de dados diferentes, por exemplo, bancos de dados, serviços da Web e caches.



05

ARCHITECTURE COMPONENTS





ARCHITECTURE COMPONENTS

Os architecture components do Android são um conjunto de bibliotecas que ajuda você a projetar apps robustos, testáveis e de fácil manutenção. Essas libs oferecem classes para gerenciar, por exemplo, o ciclo de vida de IU e lidar com a persistência de dados.



ARCHITECTURE COMPONENTS



 **GDDIndia**
Instructor-led Training

**Build an App
with Architecture
Components**





EXERCÍCIOS

Qual a diferença entre uma Data Class e uma Class em Kotlin?

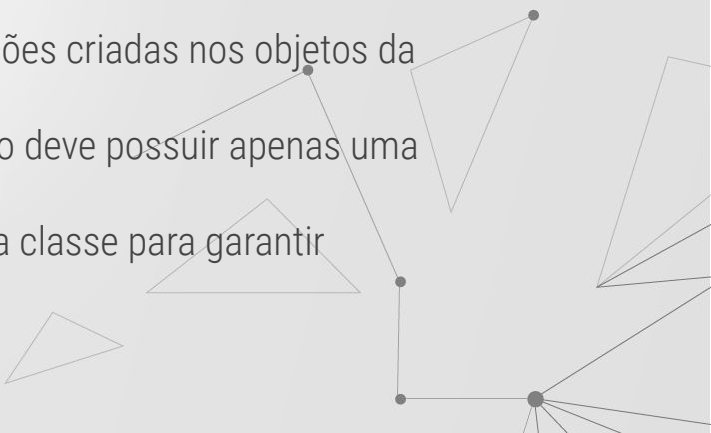
- A Data Class só pode ser usada para modelar entidades de bancos de dados
 - Uma Data Class só possui estados e não executa nenhum tipo de operação
 - A Data Class deve implementar os métodos `create()` e `update()`
 - Uma Class deve obrigatoriamente herdar comportamentos de alguma interface
-





EXERCÍCIOS

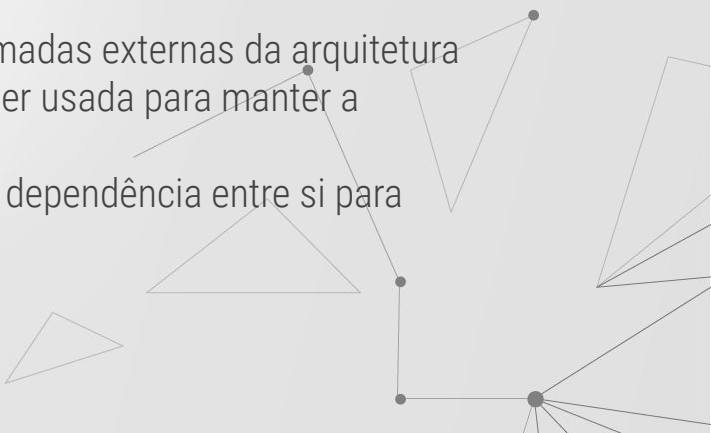
Sobre os princípios do SOLID. A Letra L representa:

- Princípio de Substituição de Liskov, que afirma que uma subclasse deve manter todos os comportamentos da sua superclasse
 - Liquid Base Principle, se refere à fluidez das abstrações criadas nos objetos da POO
 - Princípio de Liskov: cada classe, entidade ou método deve possuir apenas uma responsabilidade
 - Laura Principle: testes devem ser aplicados em cada classe para garantir qualidade
-
- 



EXERCÍCIOS

O que é a Regra de Dependência do Clean Architecture?

- Todas as camadas devem ser dependentes entre si, para garantir coesão do software
 - As camadas internas nunca devem depender das camadas externas da arquitetura
 - Uma ferramenta de Injeção de Dependências deve ser usada para manter a arquitetura limpa
 - Métodos de uma classe devem possuir uma grande dependência entre si para estarem juntos.
-
- 



Palavras chave da aula de hoje:

- POO
- SOLID
- Arquitetura de software
- Clean Architecture
- MVVM
- Dialog
- Architecture Components

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**.

Please keep this slide for attribution.