

# Aula 26

## Arquitetura

# Agora somos desenvolvedores Android



Não basta sair  
digitando código,  
temos que ter  
disciplina, organização,  
planejamento e  
padronização.



# Escalabilidade e testabilidade



Quais pacotes devo criar?

Devo criar classe para isso?

...?





**S**ingle Responsibility

**O**pen-closed Principle

**L**iskov Substitution Principle

**I**nterface Segregation Principle


**D**ependency Inversion Principle





# Princípio da Responsabilidade Única

Uma classe deve ter um, e somente um, motivo para mudar.





```
class Empregado {  
    private lateinit var id: Long  
    private lateinit var nome: String  
    private lateinit var salario: Double  
    private lateinit var funcao: String  
  
    fun login(): Boolean {  
        //Abre conexão com o banco e realiza o login do empregado no sistema  
    }  
  
    companion object {  
        const val URL = "jdbc:mysql://localhost:3306/employee?useSSL=false"  
    }  
}
```



# Princípio Aberto-Fechado

Você deve ser capaz de estender um comportamento de uma classe, sem modificá-lo.

Você pode criar extensões de classes através de herança, classes abstratas e interfaces, por exemplo



```
private class AuthenticateLogin {  
    public login(User user, String provider): Boolean {  
        var connectionDAO = new ConnectionDAO("root", "");  
        var connection = connectionDAO.createConnection();  
  
        if (provider.equalsIgnoreCase("Linkedin")){  
            //autêntica o login com o Oauth Linkedin  
        }  
        else{  
            //autêntica o login com informações do banco de dados  
        }  
    }  
}
```

# Princípio da Substituição de Liskov

As classes base devem ser substituíveis por suas classes derivadas.

```
class L_LiskovSubstitutionProblem {
    class BasicAccount {
        val balance = 0.0;

        open fun yield() {
            this.balance += (this.balance * 0.15);
        }
    }

    class SalaryAccount: BasicAccount() {
        override fun yield() {
            throw UnsupportedOperationException("Salary account can't yield")
        }
    }

    fun main() {
        val accountList = AccountDAO().getAllAccounts()
        accountList.forEach(account → account.yield())
    }
}
```

# Princípio da Segregação da Interface

Muitas interfaces específicas são melhores do que uma interface única.

```
class I_InterfaceSegregationProblem {

    interface Car {
        fun charge()
        fun openDoor()
    }

    class Fiesta: Car {
        override charge() {
            throw UnsupportedOperationException("I'm not an electric car")
        }

        override openDoor() {
            //abrir a porta
        }
    }

    class FocusElectric: Car {
        override fun charge() {
            //recargar
        }

        override fun openDoor() {
            //abrir porta
        }
    }
}
```



# Princípio da inversão da dependência

Dependa de uma abstração e não de uma implementação.


1. Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender da abstração.
2. Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

# Princípio da inversão da dependência


- Útil para injeção de dependência (DI)
- Diminuição do acoplamento da classe



```
interface IProvider {  
    fun logar(user: User)  
}
```

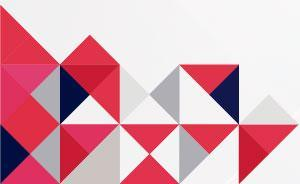


Os princípios SOLID devem ser aplicados para se obter os benefícios da orientação a objetos





- Seja fácil de se manter, adaptar e se ajustar às alterações de escopo;
- Seja testável e de fácil entendimento;
- Seja extensível para alterações com o menor esforço necessário;
- Que forneça o máximo de reaproveitamento;
- Que permaneça o máximo de tempo possível em utilização.

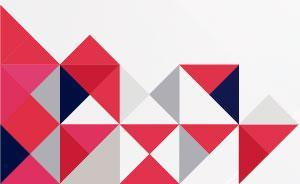


# Evita problemas comuns





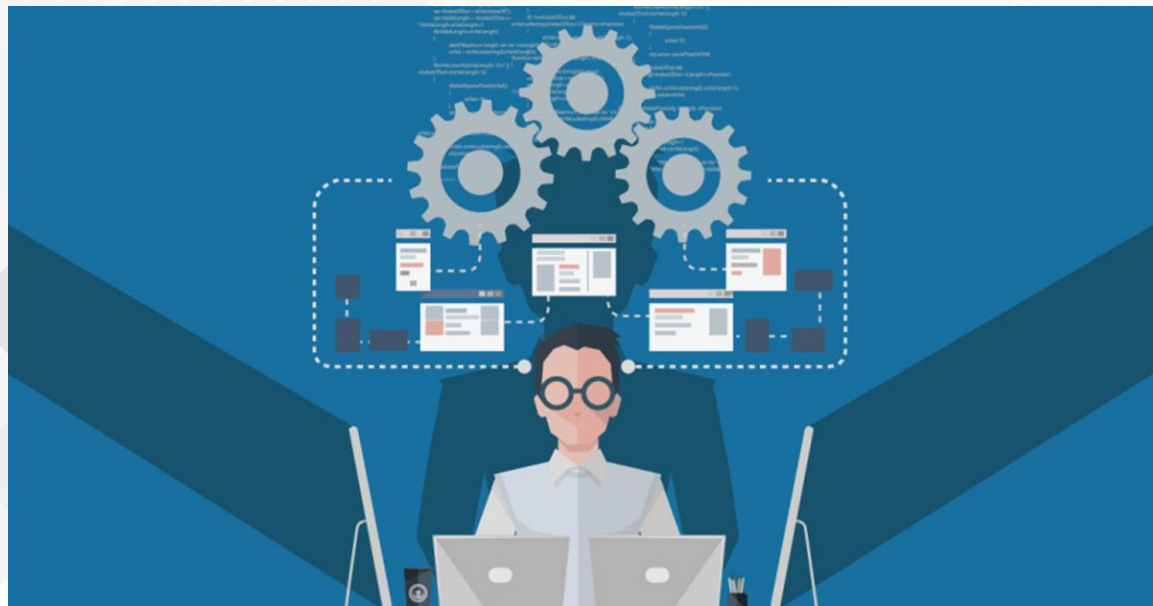
- Dificuldade na testabilidade / criação de testes de unidade;
- Código espaguete, sem estrutura ou padrão;
- Dificuldades de isolar funcionalidades;
- Duplicação de código, uma alteração precisa ser feita em N pontos;
- Fragilidade, o código quebra facilmente em vários pontos após alguma mudança.



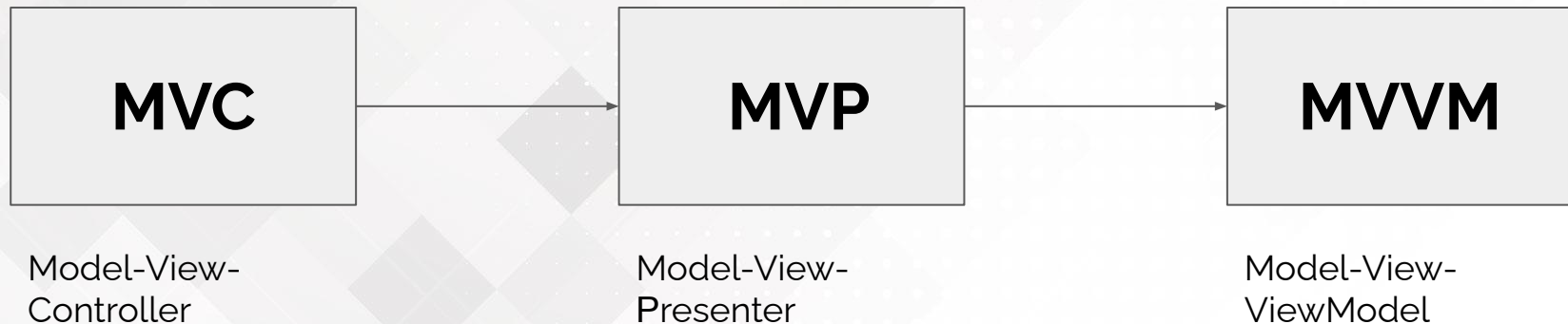
# Arquitetura

# O que é arquitetura?

Organização fundamental de um sistema, de seus componentes, suas relações entre si e com o meio ambiente e os princípios orientadores da sua concepção e evolução.



# Tipos de padrão de arquitetura



# MVC

# Model

Sempre que você pensar em manipulação de dados, pense em model. Ele é responsável pela leitura e escrita de dados, e também de suas validações.

# View

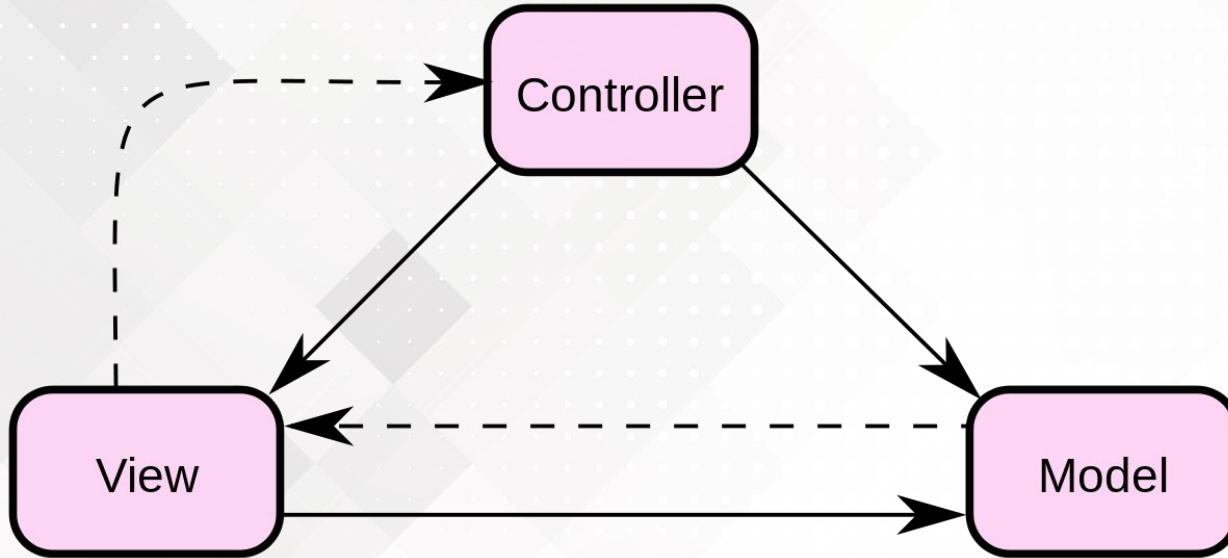
A camada de interação com o usuário. Ela apenas faz a exibição dos dados



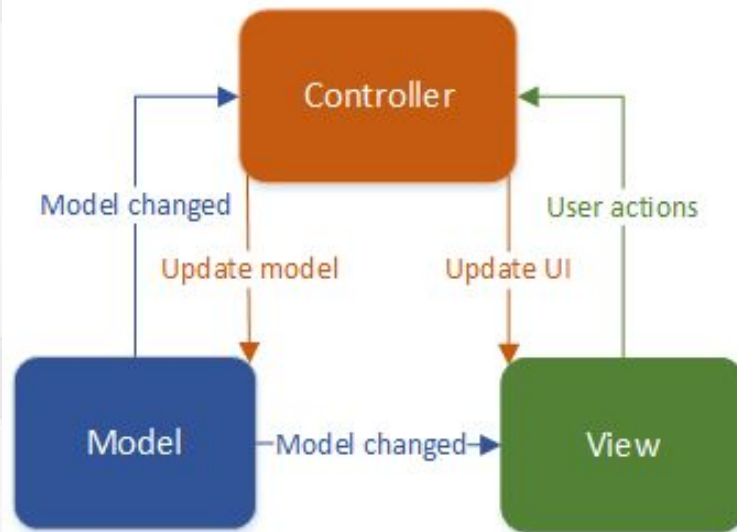
# Controller


O responsável por receber todas as requisições do usuário. Seus métodos chamados actions são responsáveis por uma página, controlando qual model usar e qual view será mostrado ao usuário.

# MVC



# MVC

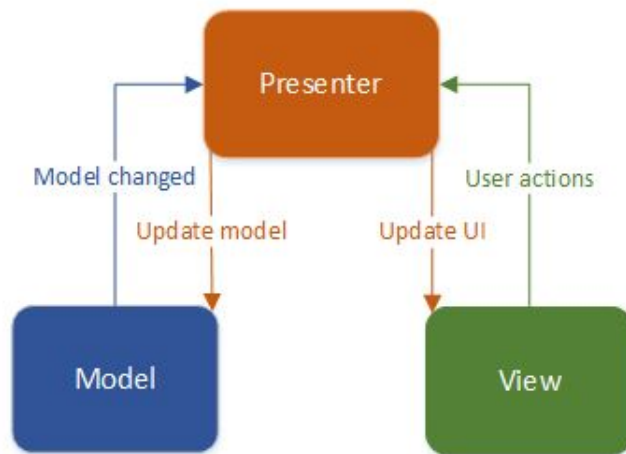


- 
- Testabilidade - O controlador está tão ligado às APIs do Android que é difícil testar a unidade.
  - Modularidade e flexibilidade - Os Controllers estão bem acoplados às Views. Pode também ser uma extensão da View.
  - Se mudarmos a View, devemos voltar e mudar o Controller.
  - Manutenção - Ao longo do tempo, particularmente em aplicações com modelos anêmicos, cada vez mais o código começa a ser transferido para os Controllers, tornando-os cheios, complexos e com grande facilidade de crashes.

The background features a large, faint, light-gray geometric pattern of overlapping squares and diamonds. In the top-left and bottom-left corners, there are clusters of small triangles in red, white, and dark blue. A similar pattern is in the top-right corner, and a smaller one is in the bottom-left corner.

# MVP

# MVP





# Model

Mesma coisa do MVC

# View

A única alteração é que a Activity/Fragment agora é considerada parte da View.

# Presenter

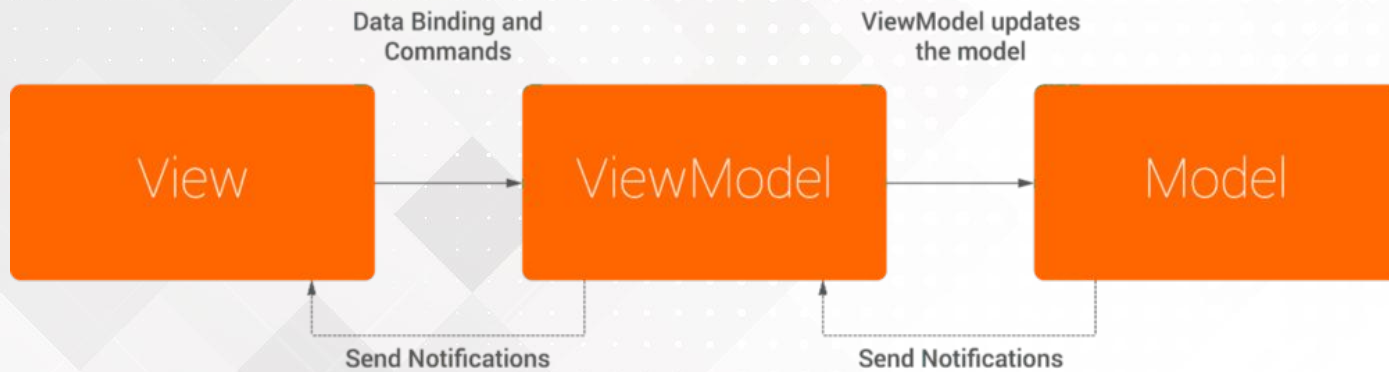
Quase a mesma coisa do controller do MVC, porém ele não controla a View, mas sim delega o que exibir

- Manutenção - Os Presenters, assim como os Controllers, são propensos a colecionar lógica comercial adicional, espalhados com o tempo.
- Eventualmente, os desenvolvedores deparam-se com grandes Presenters difíceis de separar.

The background features a large, faint, light-gray geometric pattern of overlapping squares and diamonds. In the top-left and bottom-left corners, there are clusters of small triangles in red, white, and dark blue. A similar pattern is in the top-right corner, and a smaller one is in the bottom-left corner.

# MVVM

# A arquitetura mais utilizada em apps Android MVVM





# Model


Mesma coisa do MVC e MVP


# View

A View liga-se a variáveis Observable e ações expostas pelo ViewModel de forma flexível.

# ViewModel

A ViewModel é responsável por apresentar funções, métodos e comandos para manter o estado da View, operar a Model e ativar os eventos na View.

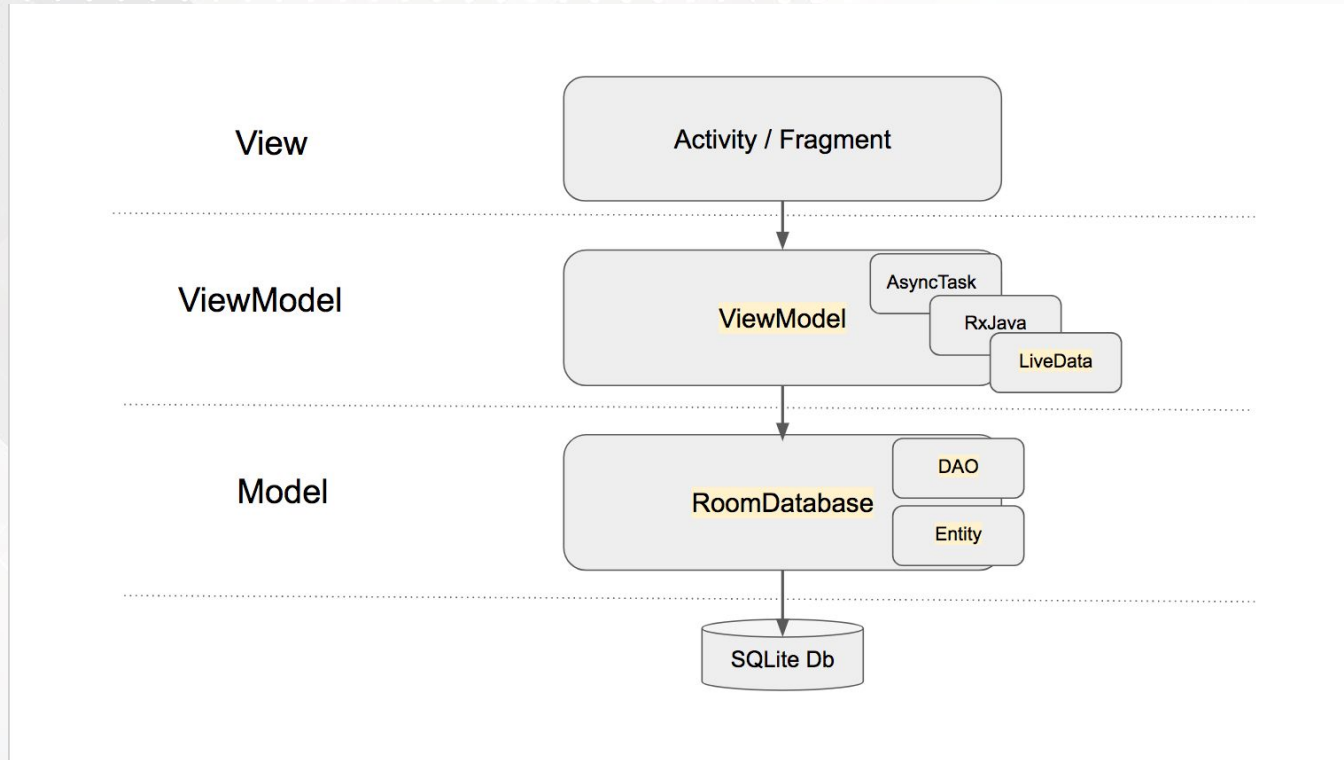
- 
- Manutenção - É possível entrar em partes menores e focadas do código
  - Testabilidade - Cada parte do código permanece granular
  - Extensibilidade - Devido ao aumento de partes granulares de código e limites de separação, às vezes ele se mistura com a capacidade de manutenção. Se você planeja reutilizar qualquer uma dessas partes terá mais chances de fazê-lo.



O principal objetivo de escolher a arquitetura MVVM é abstrair as Views para que o código por trás da lógica de negócios possa ser reduzido a uma extensão.

1. A camada lógica e de apresentação é fracamente acoplada.
2. Sem qualquer tipo de automação e interação da interface do usuário, você pode testá-lo.
3. O código orientado a eventos ou code-behind, no ViewModel fica fácil de executar o teste unitário.

# MVVM - Exemplo



# Exemplo prático e exercícios...

