Desenvolvimento Mobile Profissional

App Dev 2023

Mark Joselli

mark.joselli@pucpr.br

Cronograma

- ~~Aula 1 05/08/2023 – Introdução e TDD~~
- ~~Aula 2 12/08/2023 – Threads e performance optimization~~
- Aula 3 02/09/2023 - Dependency Injection, Clean Code, SOLID
- Aula 4 02/09/2023 – Mobile Architectures
- Aula 5 16/09/2023 (EAD) - Design Patterns
- Aula 6 14/10/2023 (EAD) - Projeto

Regras Gerais

Regras Gerais

- Siga as convenções
- Se você começou agora em um projeto ou acabaram de definir suas convenções, siga-as!
- Se utilizam por exemplo constantes em maiúsculo, enumeradores com E como prefixo, não importa!
- Siga sempre os padrões do projeto.

Regras Gerais

- KISS
- Mantenha as coisas simples!
- Este conceito vem até de outro livro, e particularmente acho que é a base de uma boa solução.
- Normalmente tendemos a complicar as coisas que poderiam ser muito mais simples.
- Então, Keep It Stupid Simple (Mantenha isto estupidamente simples - KISS)!

Regras Gerais

- Regra do escoteiro
- "Deixe sempre o acampamento mais limpo do que você encontrou!" O mesmo vale para nosso código.
- Devolva (Check in) sempre o código melhor do que você o obteve.
- Se todo desenvolvedor no time tiver esta visão, e devolver um pedacinho de código melhor do que estava antes, em pouco temos teremos uma grande mudança.

Regras Gerais

- Causa raiz
- Sempre procure a causa raiz do problema, nunca resolva as coisas superficialmente.
- No dia-a-dia, na correria, tendemos a corrigir os problemas superficialmente e não adentrar neles, o que muitas vezes causa o re-trabalho!
- Tente sempre procurar a causa raiz e resolver assim o problema de uma vez por todas!

Dependency Injection

injeção de dependências

- A injeção de dependências é um padrão de design usado em programação para aumentar a eficiência e a modularidade do código.
- É uma forma de implementar o Princípio da Inversão de Dependência (DIP), que é um dos cinco princípios SOLID de design de software orientado a objetos.
- Como funciona:
  - Definição de Dependências: Primeiro, você define as dependências que seu código precisa para funcionar. Essas dependências podem ser interfaces, classes abstratas ou implementações concretas.
  - Injeção de Dependências: Em seguida, você "injeta" essas dependências no código que as necessita, em vez de criar as dependências diretamente no código. A injeção pode ser feita através do construtor, de um método setter ou de um método de fábrica.
  - Uso de Dependências: O código que recebe as dependências injetadas pode então usá-las como se tivesse criado as dependências ele mesmo.

injeção de dependências

- Vantagens:
  - Desacoplamento: A injeção de dependências ajuda a desacoplar o código, tornando-o mais modular e flexível. Por exemplo, se um componente depende de uma interface, em vez de uma implementação concreta, você pode facilmente substituir a implementação sem modificar o componente.
  - Testabilidade: A injeção de dependências facilita a escrita de testes unitários, porque você pode injetar implementações falsas ou simuladas das dependências para testar o código em isolamento.
  - Reutilização de Código: Como o código é mais modular e desacoplado, é mais fácil reutilizar componentes em diferentes partes do aplicativo ou em aplicativos diferentes.
  - Manutenibilidade: O código é mais fácil de manter porque as dependências são explícitas e claras. Isso facilita a compreensão de como o código funciona e como as diferentes partes do código estão relacionadas.
  - Configuração Flexível: Você pode alterar as dependências de um componente sem modificar o componente em si. Isso é útil se você quiser alterar o comportamento do aplicativo em tempo de execução ou para diferentes configurações.
- Lembrando que, embora a injeção de dependências tenha muitas vantagens, ela também pode tornar o código mais complexo e difícil de entender se não for usada corretamente. É importante usar o bom senso e não seguir as regras cegamente.

Injeção de dependências Android

- Primeiro, crie uma interface para a dependência que você deseja injetar. Por exemplo, se você tem uma classe NetworkService que é responsável por fazer chamadas de rede, crie uma interface INetworkService que a NetworkService implementará.

```
interface INetworkService {
    fun fetchData(): String
}
```

Injeção de dependências Android

- Em seguida, implemente a interface na classe de serviço.

```
class NetworkService : INetworkService {
    override fun fetchData(): String {
        // fetch data from network
        return "Data from network"
    }
}
```

Injeção de dependências Android

- Crie uma classe de injeção de dependências: Crie uma classe que será responsável por fornecer as dependências para o restante do aplicativo. Esta classe terá métodos que retornarão instâncias das dependências necessárias.

```
class DependencyInjector {
    fun provideNetworkService(): INetworkService {
        return NetworkService()
    }
}
```

Injeção de dependências Android

- Use a injeção de dependências: Em vez de criar uma instância da NetworkService diretamente na sua Activity ou Fragment, use a DependencyInjector para obter uma instância da INetworkService.

```
class MainActivity : AppCompatActivity() {
    private lateinit var networkService: INetworkService
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        // inject dependencies
        val dependencyInjector = DependencyInjector()
        networkService = dependencyInjector.provideNetworkService()
        // use the networkService
        val data = networkService.fetchData()
        println(data)
    }
}
```

Injeção de dependências Android

- Neste exemplo, a MainActivity não precisa saber sobre a implementação da INetworkService, ela apenas sabe que precisa de algum objeto que implemente a INetworkService.
- Isso torna o código mais flexível e testável, pois você pode facilmente substituir a NetworkService por uma implementação diferente para testes ou para fornecer funcionalidades diferentes.

Frameworks Android

- Dagger: É um dos frameworks de injeção de dependências mais populares e amplamente utilizados na comunidade Android. Ele é conhecido por ser extremamente eficiente em termos de desempenho, pois usa geração de código em tempo de compilação. No entanto, ele tem uma curva de aprendizado íngreme e pode ser complexo de configurar, especialmente para projetos grandes ou para desenvolvedores que são novos na injeção de dependências.
- Hilt: É um framework de injeção de dependências construído em cima do Dagger, que visa simplificar a configuração e o uso do Dagger no Android. Ele fornece uma série de convenções e anotações que reduzem a quantidade de código de configuração que você precisa escrever. O Hilt é recomendado pela Google e é mais fácil de usar do que o Dagger puro, especialmente para desenvolvedores que são novos na injeção de dependências.
- Koin: É outro framework de injeção de dependências popular no Android. Ele é escrito em Kotlin e usa as características da linguagem para tornar a configuração e o uso da injeção de dependências mais simples e mais idiomáticas para o Kotlin. O Koin é mais fácil de aprender e configurar do que o Dagger, mas pode não ser tão eficiente em termos de desempenho, pois usa reflexão em tempo de execução.

Frameworks iOS

- EnvironmentObject é parte do framework SwiftUI da Apple.
- O SwiftUI é um framework de interface do usuário declarativo que permite construir interfaces de usuário para todos os dispositivos da Apple usando Swift.
- EnvironmentObject é um tipo de propriedade especial no SwiftUI que é usado para compartilhar dados entre diferentes views.
- Você pode usar um EnvironmentObject para injetar uma instância de um objeto observável (um objeto que conforma ao protocolo ObservableObject) em uma hierarquia de views, para que todas as views na hierarquia possam acessar e observar o objeto.

Frameworks iOS

```
class UserData: ObservableObject {
    @Published var name = "John Doe"
}
let userData = UserData()
let contentView = ContentView().environmentObject(userData)
struct ContentView: View {
    @EnvironmentObject var userData: UserData

    var body: some View {
        Text("Hello, \(userData.name)!")
    }
}
```

Clean Code

Clean Code

- O Clean Code, em tradução livre "Código Limpo", é um conjunto de práticas e diretrizes que visam produzir código fonte de alta qualidade, legível, bem estruturado e de fácil manutenção.
- No contexto do desenvolvimento mobile, seja para Android ou iOS, o Clean Code desempenha um papel crucial na criação de aplicativos eficientes, robustos e escaláveis.
- O Clean Code se refere a um estilo de programação que vai além de simplesmente fazer o código funcionar.
- Ele se concentra em tornar o código compreensível para outros desenvolvedores (incluindo o seu "eu" futuro) e em reduzir a complexidade, para que a manutenção, a colaboração e as melhorias possam ser realizadas de forma eficaz.

Importância no Desenvolvimento Mobile

- Legibilidade e Compreensão: Em projetos mobile, onde as equipes podem ser diversas e os ciclos de desenvolvimento podem ser rápidos, o Clean Code se torna essencial. Código limpo é mais fácil de ler e entender, o que acelera a integração de novos membros da equipe e facilita a manutenção.
- Redução de Erros: Código confuso e desorganizado é propenso a erros. Manter um código limpo ajuda a minimizar a introdução de bugs e simplifica a identificação e correção de problemas quando eles ocorrem.
- Agilidade e Iterações Rápidas: No desenvolvimento mobile, a agilidade é fundamental. Código limpo permite que as equipes iterem rapidamente sobre funcionalidades, adicionem melhorias e corrijam falhas sem o fardo de um código difícil de entender e modificar.

Importância no Desenvolvimento Mobile

- Economia de Tempo e Recursos: Código limpo resulta em menos tempo gasto na depuração, refatoração e correção de problemas. Isso libera os desenvolvedores para se concentrarem em adicionar valor real ao aplicativo.
- Escalabilidade: À medida que um aplicativo mobile cresce, a manutenção se torna um desafio. Código limpo simplifica a escalabilidade, permitindo que novos recursos sejam adicionados sem que o código se torne um labirinto de dependências.
- Melhoria da Colaboração: Desenvolvedores frequentemente trabalham em equipe, e um código limpo promove uma colaboração mais eficaz. Cada membro da equipe pode entender o código dos outros com mais facilidade, tornando a colaboração mais suave.

Importância no Desenvolvimento Mobile

- Documentação Viva: Código limpo muitas vezes dispensa a necessidade de comentários excessivos, pois ele próprio se torna uma forma de documentação. Isso evita que os comentários se tornem desatualizados e enganosos.
- Preparação para Mudanças Futuras: A manutenção de aplicativos mobile é inevitável. Ter um código limpo prepara o terreno para futuras mudanças e atualizações, tornando-as menos trabalhosas.

Princípios do Clean Code

- Nomes Significativos: Escolha nomes descritivos e não ambíguos para variáveis, funções, classes e outros identificadores.
- Funções: As funções devem ser pequenas, fazer apenas uma coisa e fazer bem. Elas devem ter um número mínimo de argumentos.
- Comentários: Comentários devem ser usados com moderação e devem adicionar valor ao código. Código bem escrito muitas vezes é autoexplicativo.
- Formatação: A formatação do código deve ser consistente e seguir um conjunto de regras estabelecidas.
- Objetos e Estruturas de Dados: Use objetos para agrupar dados e comportamentos relacionados e use estruturas de dados para manter os dados separados do comportamento.

Princípios do Clean Code

- Tratamento de Erros: Trate os erros de forma adequada e não os ignore. Use exceções para indicar situações excepcionais.
- Limites: Mantenha as dependências entre os módulos claras e bem definidas.
- Testes: Escreva testes unitários e os mantenha limpos. Os testes são tão importantes quanto o código de produção.
- Classes: As classes devem ser pequenas e ter uma única responsabilidade.
- Sistemas: Mantenha os sistemas pequenos e bem organizados.

Princípios do Clean Code

- Emergência: Mantenha o código o mais simples possível e refatore o código conforme necessário para mantê-lo limpo e eficiente.
- Concorrência: A concorrência deve ser tratada com cuidado e deve ser bem compreendida.
- Sucessivos Refinamentos: O código deve ser continuamente refinado e melhorado.
- Cheiro de Código: Aprenda a reconhecer os "cheiros de código" (code smells) que indicam problemas no código e saiba como corrigi-los.

Nomes Significativos

- A prática de usar nomes significativos é fundamental para criar código limpo e compreensível. Nomes bem escolhidos ajudam a transmitir a intenção do código de forma clara, o que torna o processo de leitura e manutenção mais suave. Aqui estão algumas diretrizes para escolher nomes significativos:
- Nomes Descritivos: Escolha nomes que expliquem claramente a finalidade da variável, função, classe ou método. Um nome bem escolhido deve indicar qual é o propósito e o que a entidade representa.
- Evite Abreviações Obscuras: Embora algumas abreviações sejam amplamente conhecidas e aceitas, é preferível evitar abreviações que possam ser confusas ou difíceis de entender para alguém que não esteja familiarizado com o projeto.
- Use Nomes Completos: Nomes completos geralmente são mais descritivos do que nomes curtos. Por exemplo, em vez de usar "usrCt" para "usuárioContador", use o nome completo para que outros possam entender facilmente.

Nomes Significativos

- Seja Consistente: Mantenha uma convenção de nomenclatura consistente ao longo do código. Isso ajuda os desenvolvedores a entender rapidamente a estrutura e a semântica do código.
- Nomes Pronunciáveis: Escolha nomes que possam ser pronunciados facilmente. Isso facilita a discussão sobre o código entre membros da equipe.
- Nomes para Funções/Métodos: Escolha nomes de funções/métodos que descrevam o que a função faz. Um nome de função deve ser um verbo ou uma frase que começa com um verbo.
- Nomes para Variáveis: Escolha nomes que descrevam o conteúdo ou o papel da variável. Evite usar nomes genéricos como "temp" ou "data".
- Evite Nomes Enganosos: Não use nomes que possam levar a interpretações erradas sobre a função ou o comportamento do código.

Nomes Significativos - Exemplo Variáveis:

```
// Ruim
val a = 10

// Bom
val numberOfUsers = 10
```

Nomes Significativos - Exemplo Funções:

```
// Ruim
fun calc(a: Int, b: Int): Int {
    return a * b
}

// Bom
fun calculateProduct(a: Int, b: Int): Int {
    return a * b
}
```

Nomes Significativos - Exemplo Classes:

```
// Ruim
class Data {
    // ...
}

// Bom
class UserData {
    // ...
}
```

Nomes Significativos - Exemplo funções:

```
// Ruim
fun printInfo(u: User) {
    println("Nome: ${u.name}, Idade: ${u.age}")
}

// Bom
fun printUserInfo(user: User) {
    println("Nome: ${user.name}, Idade: ${user.age}")
}
```

Nomes Significativos - Exemplo Iteração:

```
// Ruim
for (i in 0 until 10) {
    // ...
}

// Bom
for (index in 0 until 10) {
    // ...
}
```

Nomes Significativos - Exemplo Teste Condicional:

```
/ Ruim
if (s == "Y") {
    // ...
}

// Bom
if (input == "Yes") {
    // ...
}
```

Funções/Métodos Pequenos

- A prática de manter funções e métodos pequenos é fundamental para garantir um código mais legível, compreensível e fácil de manter.
- Vantagens de Funções Pequenas:
  - Legibilidade: Funções curtas são mais fáceis de entender e ler, tornando o código mais compreensível para você e outros desenvolvedores.
  - Manutenção: Quando uma função precisa de ajustes, é mais fácil identificar o local do problema em uma função curta.
  - Reusabilidade: Funções pequenas podem ser mais facilmente reutilizadas em diferentes partes do código.
  - Testabilidade: Funções curtas são mais fáceis de testar, pois se concentram em uma única tarefa.
  - Colaboração: Facilita a colaboração em equipe, já que os membros da equipe podem revisar e compreender o código de forma mais eficaz.
- Lembre-se de que a regra "Uma função, um propósito" é um guia valioso para manter o código organizado e de fácil manutenção.

Funções/Métodos Pequenos

```
// Ruim
fun processUserData(data: List<User>) {
    for (user in data) {
        if (user.isActive) {
            val userDetails = fetchUserDetails(user.id)
            val formattedDetails = formatUserDetails(userDetails)
            saveFormattedDetailsToFile(formattedDetails)
        }
    }
    // Mais código relacionado ao processamento de dados...
}
```

```
// Bom
fun processActiveUsersData(data: List<User>) {
    for (user in data) {
        if (user.isActive) {
            processActiveUser(user)
        }
    }
}

fun processActiveUser(user: User) {
    val userDetails = fetchUserDetails(user.id)
    val formattedDetails = formatUserDetails(userDetails)
    saveFormattedDetailsToFile(formattedDetails)
}
```

Comentários Claros

- A prática de escrever comentários claros é importante para fornecer contexto e esclarecimento sobre partes do código que não são imediatamente óbvias.
- No entanto, é crucial utilizar comentários de maneira eficaz para não sobrecarregar o código com informações redundantes ou desnecessárias.
- Vantagens de Comentários Claros:
  - Explicação de Intenções: Comentários podem explicar a intenção por trás de decisões de design ou implementação, o que ajuda outros desenvolvedores a entender o porquê de certas escolhas.
  - Lógica Complexa: Quando há lógica complexa ou algoritmos não triviais, comentários podem explicar a abordagem utilizada.
  - Regras de Negócio: Comentários podem ajudar a relacionar partes do código com regras de negócios específicas.
  - Prevenção de Mal-Entendidos: Comentários evitam que outros desenvolvedores interpretem erroneamente o código.
- No entanto, lembre-se de que comentários não devem ser usados como um substituto para um código bem escrito e autoexplicativo. Idealmente, o código deve ser claro o suficiente para que a maioria dos detalhes seja compreendida sem a necessidade de comentários. Use comentários apenas para fornecer informações adicionais que não são óbvias a partir do código em si.

Comentários Claros

//Ruim

val total = calculateTotal(items) // Calcula o total dos itens

Neste exemplo, o comentário é redundante, pois o próprio código já diz que a função calculateTotal está sendo chamada para calcular o total dos itens. O comentário não acrescenta informações relevantes e pode se tornar desatualizado se a função mudar.

//Bom

val discount = calculateDiscount(total) // Calcula o desconto com base no total antes de impostos

Neste exemplo, o comentário fornece informações adicionais que não são imediatamente claras apenas olhando para o código. Ele explica por que a função calculateDiscount está sendo chamada, ajudando a entender que o desconto está sendo calculado com base no total antes de impostos.

Formatação

- A formatação é um aspecto crucial do clean code. Um código bem formatado é mais fácil de ler, entender e manter. Aqui estão algumas dicas para a formatação de código:
- Indentação: Use a indentação de forma consistente para mostrar a estrutura do código. A maioria das IDEs e editores de texto tem funcionalidades para formatar o código automaticamente.
- Linhas em Branco: Use linhas em branco para separar blocos de código relacionados. Isso ajuda a organizar o código e torná-lo mais legível.
- Comprimento da Linha: Mantenha o comprimento da linha dentro de um limite razoável. Um limite comum é 80-120 caracteres por linha.
- Espaçamento: Use espaçamento de forma consistente. Por exemplo, sempre coloque espaços ao redor dos operadores (a + b em vez de a+b).

Formatação

- Organização do Código: Organize o código de forma lógica. Por exemplo, coloque as variáveis e funções relacionadas perto umas das outras.
- Convenções de Nomenclatura: Siga as convenções de nomenclatura do idioma que você está usando. Por exemplo, em Kotlin, as variáveis e funções são geralmente em camelCase (myVariable) e as classes em PascalCase (MyClass).
- Comentários: Use comentários de forma eficaz. Eles devem ser usados para explicar o "porquê" do código, não o "o que". O código em si deve ser autoexplicativo.
- Consistência: Seja consistente na formatação do código. Se você está trabalhando em equipe, considere o uso de uma ferramenta de formatação de código automática para garantir a consistência em todo o projeto.

Indentação

```
// Ruim
fun soma(a: Int, b: Int): Int {
return a + b
}
```

```
// Bom
fun soma(a: Int, b: Int): Int {
    return a + b
}
```

Linhas em Branco e espaçamento

```
// Ruim
fun soma(a:Int,b:Int):Int{return a+b}
```

```
// Bom
fun soma(a: Int, b: Int): Int {
    return a + b
}
```

Comprimento da Linha

```
// Ruim

fun somaDeTresNumerosComNomesMuitoGrandes(numeroUm: Int,
numeroDois: Int, numeroTres: Int): Int {

    return numeroUm + numeroDois + numeroTres

}
```

```
// Bom

fun somaDeTresNumeros(

    numeroUm: Int,

    numeroDois: Int,

    numeroTres: Int

): Int {

    return numeroUm + numeroDois + numeroTres

}
```

Objetos e Estruturas de Dados

- No contexto do clean code, a maneira como você organiza e manipula objetos e estruturas de dados é fundamental para a legibilidade e manutenibilidade do seu código. Aqui estão algumas dicas para trabalhar com objetos e estruturas de dados:
- Abstração de Dados: Não exponha os detalhes internos dos seus objetos. Use métodos de acesso (getters e setters) para manipular os dados dos objetos.
- Objetos vs. Estruturas de Dados: Objetos ocultam suas implementações internas e expõem operações. Estruturas de dados expõem seus dados e não têm operações significativas.
- Lei de Demeter: Um objeto só deve se comunicar com seus vizinhos imediatos. Em outras palavras, um método M de um objeto O só deve invocar os métodos de O, objetos parâmetro de M, objetos que M cria, ou objetos que M obtém diretamente de O.

Objetos e Estruturas de Dados

- Transferência de Dados: Use objetos de transferência de dados (DTOs) para agrupar múltiplos elementos de dados em um único objeto.
- Classe vs. Estrutura de Dados: Se você está criando uma classe, faça-a completamente encapsulada e forneça métodos que operem em seus dados. Se você está criando uma estrutura de dados, faça-a completamente exposta sem métodos significativos.
- Preferir Objetos a Estruturas de Dados: Prefira usar objetos para agrupar dados e comportamentos relacionados.
- Encapsulamento: Encapsule os detalhes internos de um objeto tanto quanto possível. Isso torna mais fácil mudar a implementação interna do objeto sem afetar o código que o utiliza.

Abstração de Dados

```
// Ruim
class Pessoa {
    var nome: String = ""
}

val pessoa = Pessoa()
pessoa.nome = "João"
```

```
// Bom
class Pessoa(private var nome: String) {
    fun getNome(): String {
        return nome
    }
    fun setNome(nome: String) {
        this.nome = nome
    }
}

val pessoa = Pessoa("João", 30)
pessoa.setNome("José")
```

Lei de Demeter

```
//Ruim
class Endereco(val rua: String)

class Pessoa(val endereco: Endereco)

fun imprimirRua(pessoa: Pessoa) {
    println(pessoa.endereco.rua)
}
```

```
// Bom
class Endereco(val rua: String)

class Pessoa(private val endereco: Endereco) {
    fun getRua(): String {
        return endereco.rua
    }
}

fun imprimirRua(pessoa: Pessoa) {
    println(pessoa.getRua())
}
```

Encapsulamento

```
class Conta {
    var saldo: Double = 0.0
    fun depositar(valor: Double) {
        saldo += valor
    }

    fun sacar(valor: Double) {
        saldo -= valor
    }
}

val conta = Conta()
conta.saldo = 100.0
conta.depositar(50.0)
conta.sacar(30.0)
```

```
class Conta(private var saldo: Double) {
    fun getSaldo(): Double {
        return saldo
    }
    fun depositar(valor: Double) {
        saldo += valor
    }
    fun sacar(valor: Double) {
        if (valor <= saldo) {
            saldo -= valor
        } else {
            println("Saldo insuficiente")
        }
    }
}
val conta = Conta(100.0)
conta.depositar(50.0)
conta.sacar(30.0)
```

Testes e Testabilidade

- Testes e testabilidade são conceitos fundamentais no clean code.
- Eles se referem à facilidade com que o código pode ser testado e à prática de escrever testes para o código.
- A implementação de testes e a melhoria da testabilidade do código trazem várias vantagens:
  - Detecção Precoce de Bugs: Os testes ajudam a detectar bugs no código antes que eles cheguem à produção. Isso pode economizar muito tempo e esforço no futuro.
  - Facilita as Mudanças: Os testes tornam mais fácil fazer mudanças no código. Se você tem uma boa cobertura de testes, pode ter mais confiança de que suas mudanças não vão quebrar nada.
  - Documentação: Os testes servem como documentação para o código. Eles mostram como o código deve ser usado e o que se espera que ele faça.
  - Design Melhor: Escrever testes muitas vezes leva a um design de código melhor. Para tornar o código testável, muitas vezes é necessário torná-lo mais modular e menos acoplado.
  - Confiança: Os testes dão confiança de que o código funciona como deveria. Isso é especialmente importante em projetos grandes e complexos.
  - Automatização: Os testes permitem a automatização da verificação do código. Isso significa que você pode executar seus testes automaticamente a cada mudança no código, garantindo que nada seja quebrado inadvertidamente.
  - Integração Contínua: A implementação de testes facilita a integração contínua e a entrega contínua (CI/CD). Isso pode levar a um processo de desenvolvimento mais eficiente e a entregas mais rápidas.

Escreva Código Testável: O código deve ser escrito de forma que seja fácil de testar. Isso geralmente significa que o código deve ser modular, com cada parte do código tendo uma única responsabilidade.

```
// Ruim
class OrderProcessor(private val emailSender: EmailSender) {
    fun processOrder(order: Order) {
        // ...
        emailSender.sendEmail(order)
    }
}
```

```
// Bom
class OrderProcessor(private val emailSender: EmailSender) {
    fun processOrder(order: Order) {
        // ...
        sendOrderConfirmationEmail(order)
    }

    private fun sendOrderConfirmationEmail(order: Order) {
        emailSender.sendEmail(order)
    }
}
```

Use Injeção de Dependências: A injeção de dependências facilita a substituição de partes do código por mocks ou stubs durante os testes.

```
// Ruim
class OrderProcessor {
    private val emailSender = EmailSender()

    fun processOrder(order: Order) {
        // ...
    }
}
```

```
// Bom
class OrderProcessor(private val emailSender: EmailSender) {
    fun processOrder(order: Order) {
        // ...
    }
}
```

Escreva Testes de Unidade: Testes de unidade são testes que verificam o funcionamento de uma única unidade de código (como uma função ou um método). Eles são essenciais para garantir que cada parte do código funcione como esperado.

```
class OrderProcessorTest {


    private val emailSender = mock(EmailSender::class.java)
    private val orderProcessor = OrderProcessor(emailSender)


    @Test
    fun `processOrder should send an email`() {
        val order = Order()


        orderProcessor.processOrder(order)


        verify(emailSender).sendEmail(order)
    }
}
```

Escreva Testes de Integração: Testes de integração são testes que verificam o funcionamento de várias unidades de código juntas. Eles são importantes para garantir que as diferentes partes do código funcionem bem juntas.

```
class OrderIntegrationTest {


    private val emailSender = EmailSender()
    private val orderProcessor = OrderProcessor(emailSender)


    @Test
    fun `processOrder should send an email`() {
        val order = Order()


        orderProcessor.processOrder(order)


        // Verifique se o email foi enviado
    }
}
```

Tratamento de Erros

- Use Exceções em vez de Códigos de Retorno: Use exceções para indicar erros em vez de retornar códigos de erro.
- Crie Exceções Informativas: Inclua informações úteis nas mensagens de exceção. Isso pode ajudar a diagnosticar o problema mais rapidamente.
- Não Retorne Null: Retornar null pode levar a erros de ponteiro nulo. Em vez disso, retorne um objeto especial que indica a ausência de valor.
- Não Passe Null: Não passe null como argumento de um método. Isso pode levar a erros de ponteiro nulo.

Tratamento de Erros

- Trate os Erros o Mais Cedo Possível: Trate os erros assim que eles ocorrem. Isso pode evitar que os erros se propaguem pelo sistema.
- Use o Bloco Try-Catch-Finally: Use o bloco try-catch-finally para garantir que os recursos sejam liberados, mesmo que ocorra um erro.
- Não Ignore as Exceções: Não ignore as exceções. Mesmo que você não possa fazer nada sobre o erro, pelo menos registre-o.
- Não Crie Exceções Genéricas: Evite criar exceções genéricas que não dão informações sobre o tipo de erro que ocorreu.

Use Exceções em vez de Códigos de Retorno

```
// Ruim
fun dividir(a: Int, b: Int): Int {
    if (b == 0) {
        return -1
    }
    return a / b
}

val resultado = dividir(10, 0)
if (resultado == -1) {
    println("Erro: divisão por zero")
}
```

```
//Bom
fun dividir(a: Int, b: Int): Int {
    if (b == 0) {
        throw IllegalArgumentException("Divisor não pode ser zero")
    }
    return a / b
}

try {
    val resultado = dividir(10, 0)
} catch (e: IllegalArgumentException) {
    println(e.message)
}
```

Não Retorne Null

```
// Ruim
fun encontrarPessoa(id: Int): Pessoa? {
    // ...
    return null
}

val pessoa = encontrarPessoa(123)
println(pessoa!!.nome)
```

```
// Bom
fun encontrarPessoa(id: Int): Pessoa {
    // ...
    return Pessoa("Desconhecido", 0)
}

val pessoa = encontrarPessoa(123)
println(pessoa.nome)
```

Não Passe Null

```
// Ruim
fun imprimirNome(pessoa: Pessoa?) {
    println(pessoa!!.nome)
}

imprimirNome(null)
```

```
// Bom
fun imprimirNome(pessoa: Pessoa) {
    println(pessoa.nome)
}

imprimirNome(Pessoa("Desconhecido", 0))
```

Trate os Erros o Mais Cedo Possível

```
// Ruim
fun dividir(a: Int, b: Int): Int {
    return a / b
}

try {
    val resultado = dividir(10, 0)
} catch (e: ArithmeticException) {
    println("Erro: divisão por zero")
}
```

```
// Bom
fun dividir(a: Int, b: Int): Int {
    if (b == 0) {
        throw IllegalArgumentException("Divisor não pode ser zero")
    }
    return a / b
}

try {
    val resultado = dividir(10, 0)
} catch (e: IllegalArgumentException) {
    println(e.message)
}
```

Use o Bloco Try-Catch-Finally

```
// Ruim
val linha = bufferedReader.readLine()
println(linha)

bufferedReader.close()
```

```
// Bom
try {
    val linha = bufferedReader.readLine()
    println(linha)
} finally {
    bufferedReader.close()
}
```

Não Ignore as Exceções

```
// Ruim
try {
    val resultado = dividir(10, 0)
} catch (e: IllegalArgumentException) {
    // Ignorar
}
```

```
// Bom
try {
    val resultado = dividir(10, 0)
} catch (e: IllegalArgumentException) {
    println(e.message)
}
```

Não Crie Exceções Genéricas

```
// Ruim
fun dividir(a: Int, b: Int): Int {
    if (b == 0) {
        throw Exception("Divisor não pode ser zero")
    }
    return a / b
}
```

```
// Bom
fun dividir(a: Int, b: Int): Int {
    if (b == 0) {
        throw IllegalArgumentException("Divisor não pode ser zero")
    }
    return a / b
}
```

Limites

- No contexto do clean code, "limites" refere-se às fronteiras entre diferentes partes do sistema, como entre um pacote e outro, entre a aplicação e uma biblioteca externa, ou entre o servidor e o cliente. Algumas dicas para lidar com limites no seu código:
  - Use Interfaces para Integrar com Códigos Externos: Use interfaces para integrar com códigos externos. Isso cria uma camada de abstração entre o seu código e o código externo.
  - Não Retorne Tipos de Dados de Códigos Externos: Não retorne tipos de dados de códigos externos diretamente. Em vez disso, retorne seus próprios tipos de dados.
  - Não Passe Tipos de Dados de Códigos Externos: Não passe tipos de dados de códigos externos diretamente. Em vez disso, passe seus próprios tipos de dados.
  - Trate os Dados de Entrada na Fronteira: Trate os dados de entrada na fronteira do seu sistema. Isso pode evitar que dados inválidos se propaguem pelo sistema.
  - Use Adaptadores para Converter Dados: Use adaptadores para converter dados entre diferentes formatos ou entre o seu código e um código externo.

Use Interfaces para Integrar com Códigos Externos

```
// Ruim
class MeuServico(val bibliotecaExterna: BibliotecaExterna) {
    fun fazerAlgo() {
        bibliotecaExterna.metodoExterno()
    }
}
```

```
// Bom
interface ServicoExterno {
    fun metodoExterno()
}
class Adaptador(val bibliotecaExterna: BibliotecaExterna) : ServicoExterno {
    override fun metodoExterno() {
        bibliotecaExterna.metodoExterno()
    }
}
class MeuServico(val servicoExterno: ServicoExterno) {
    fun fazerAlgo() {
        servicoExterno.metodoExterno()
    }
}
```

Não Retorne Tipos de Dados de Códigos Externos

```
// Ruim
fun obterDadosExterno(): DadosExterno {
    val bibliotecaExterna = BibliotecaExterna()
    return bibliotecaExterna.obterDados()
}
```

```
// Bom
fun obterDadosExterno(): MeusDados {
    val bibliotecaExterna = BibliotecaExterna()
    val dadosExterno = bibliotecaExterna.obterDados()
    return MeusDados(dadosExterno)
}
```

Trate os Dados de Entrada na Fronteira

```
// Ruim
fun processarDados(dados: String) {
    // ...
}

val dados = obterDadosDeEntrada()
processarDados(dados)
```

```
// Bom
fun processarDados(dados: String) {
    if (dados.isBlank()) {
        throw IllegalArgumentException("Dados não podem ser vazios")
    }
    // ...
}

val dados = obterDadosDeEntrada()
try {
    processarDados(dados)
} catch (e: IllegalArgumentException) {
    println(e.message)
}
```

Use Adaptadores para Converter Dados

```
// Ruim
class MeuServico(val bibliotecaExterna: BibliotecaExterna) {
    fun enviarDados(dados: MeusDados) {
        val dadosExterno = DadosExterno(dados.valor)
        bibliotecaExterna.enviarDados(dadosExterno)
    }
}
```

```
// Bom
class AdaptadorDeDados {
    fun converterMeusDadosParaDadosExterno(dados: MeusDados): DadosExterno {
        return DadosExterno(dados.valor)
    }
}
class MeuServico(val bibliotecaExterna: BibliotecaExterna, val adaptadorDeDados: AdaptadorDeDados) {
    fun enviarDados(dados: MeusDados) {
        val dadosExterno = adaptadorDeDados.converterMeusDadosParaDadosExterno(dados)
        bibliotecaExterna.enviarDados(dadosExterno)
    }
}
```

Classes

- É um princípio de design de software que ajuda a organizar o código de forma que cada parte ou módulo do código tenha uma responsabilidade única e claramente definida. Isso é muitas vezes referido como o Princípio da Responsabilidade Única (SRP, do inglês Single Responsibility Principle).
- Vantagens:
  - Manutenibilidade: É mais fácil manter e modificar o código. Quando cada parte do código tem uma única responsabilidade, é mais fácil fazer alterações sem afetar outras partes do sistema.
  - Testabilidade: É mais fácil escrever testes para o código. Quando cada parte do código tem uma única responsabilidade, você pode escrever testes focados para cada parte individualmente.
  - Reusabilidade: É mais fácil reutilizar o código em outras partes do sistema ou em outros projetos. Quando cada parte do código tem uma única responsabilidade, é mais provável que você possa reutilizar essa parte em outros lugares.
  - Legibilidade: O código é mais fácil de entender. Quando cada parte do código tem uma única responsabilidade, é mais fácil para outros desenvolvedores (ou você mesmo no futuro) entender o que cada parte do código está fazendo.
  - Redução de Bugs: Menos propenso a bugs. Quando cada parte do código tem uma única responsabilidade, há menos chance de efeitos colaterais inesperados e, portanto, menos chance de bugs.
  - Facilita a Refatoração: Facilita a refatoração do código. Quando cada parte do código tem uma única responsabilidade, é mais fácil refatorar o código sem afetar outras partes do sistema.
  - Escalabilidade: Facilita a escalabilidade do código. Quando cada parte do código tem uma única responsabilidade, é mais fácil escalar o código, pois você pode modificar ou estender cada parte individualmente.

Use Nomes Significativos

```
// Ruim
class C {
    fun m(d: Int) {
        // ...
    }
}
```

```
// Bom
class Calculadora {
    fun somar(numero: Int) {
        // ...
    }
}
```

Use Classes Pequenas

```
// Ruim
class Pessoa {
    var nome: String = ""
    var idade: Int = 0
    fun salvar() {
        // ...
    }
    fun imprimir() {
        // ...
    }
}
```

```
// Bom
class Pessoa(var nome: String, var idade: Int)

class Repositorio {
    fun salvar(pessoa: Pessoa) {
        // ...
    }
}
class Impressora {
    fun imprimir(pessoa: Pessoa) {
        // ...
    }
}
```

Use um Número Mínimo de Campos

// Ruim

```
class Pessoa(var nome: String, var idade: Int, var endereco: String, var telefone: String, var email: String)
```

```
class Pessoa(var nome: String, var idade: Int, var contato: Contato)

class Contato(var endereco: String, var telefone: String, var email: String)
```

Use Encapsulamento

```
// Ruim
class Pessoa {
    var nome: String = ""
    var idade: Int = 0
}

val pessoa = Pessoa()
pessoa.nome = "João"
pessoa.idade = 30
```

```
//Bom
class Pessoa(private var nome: String, private var idade: Int) {
    fun getNome(): String {
        return nome
    }

    fun getIdade(): Int {
        return idade
    }
}

val pessoa = Pessoa("João", 30)
val nome = pessoa.getNome()
val idade = pessoa.getIdade()
```

Evite Usar Classes com Muitos Métodos

```
// Ruim
class Calculadora {
    fun somar(a: Int, b: Int): Int {
        return a + b
    }
    fun subtrair(a: Int, b: Int): Int {
        return a - b
    }
    fun multiplicar(a: Int, b: Int): Int {
        return a * b
    }
    fun dividir(a: Int, b: Int): Int {
        return a / b
    }
}
```

```
//Bom
class Calculadora {
    fun somar(a: Int, b: Int): Int {
        return a + b
    }
    fun subtrair(a: Int, b: Int): Int {
        return a - b
    }
}
class Matematica {
    fun multiplicar(a: Int, b: Int): Int {
        return a * b
    }

    fun dividir(a: Int, b: Int): Int {
        return a / b
    }
}
```

Sistema

- No contexto do clean code, "sistemas" refere-se à organização e estruturação do código em um nível mais alto.
- Aqui estão algumas dicas para criar sistemas limpos:
- Separe a Construção da Utilização: Separe o código que constrói o sistema do código que o utiliza.
- Esconda Detalhes de Construção: Esconda os detalhes de construção do sistema atrás de uma interface.
- Use o Princípio da Inversão de Dependência: Dependa de abstrações, não de implementações concretas.
- Evite Singleton: Evite usar o padrão singleton, pois ele pode tornar o código difícil de testar.
- Use Injeção de Dependência: Use injeção de dependência para fornecer as dependências de um objeto.
- Use Módulos: Organize o código em módulos com responsabilidades claras e bem definidas.
- Use Princípios de Design: Siga os princípios de design, como o SOLID, para criar um sistema bem estruturado.

Separe a Construção da Utilização

```
// Ruim
class MeuServico {
    val repositorio = Repositorio()

    fun fazerAlgo() {
        val dados = repositorio.obterDados()
        // ...
    }
}

val meuServico = MeuServico()
meuServico.fazerAlgo()
```

```
// Bom
class MeuServico(val repositorio: Repositorio) {
    fun fazerAlgo() {
        val dados = repositorio.obterDados()
        // ...
    }
}

val repositorio = Repositorio()
val meuServico = MeuServico(repositorio)
meuServico.fazerAlgo()
```

Esconda Detalhes de Construção

```
// Ruim
class MeuServico(val repositorio: Repositorio) {
    fun fazerAlgo() {
        val dados = repositorio.obterDados()
        // ...
    }
}

val repositorio = Repositorio()
val meuServico = MeuServico(repositorio)
meuServico.fazerAlgo()
```

```
// Bom
interface Servico {
    fun fazerAlgo()
}

class MeuServico(val repositorio: Repositorio) : Servico {
    override fun fazerAlgo() {
        val dados = repositorio.obterDados()
        // ...
    }
}

val repositorio = Repositorio()
val servico: Servico = MeuServico(repositorio)
servico.fazerAlgo()
```

Use o Princípio da Inversão de Dependência

```
// Ruim
class Repositorio {
    fun obterDados(): List<Dados> { ...
    }
}
class MeuServico {
    val repositorio = Repositorio()
    fun fazerAlgo() {
        val dados = repositorio.obterDados()   ...
    }
}
val meuServico = MeuServico()
meuServico.fazerAlgo()
```

```
// Bom
interface IRepositorio {
    fun obterDados(): List<Dados>
}
class Repositorio : IRepositorio {
    override fun obterDados(): List<Dados> { ...
    }
}
class MeuServico(val repositorio: IRepositorio) {
    fun fazerAlgo() {
        val dados = repositorio.obterDados(). ...
    }
}
val repositorio: IRepositorio = Repositorio()
val meuServico = MeuServico(repositorio)
meuServico.fazerAlgo()
```

Evite Singleton

```
// Ruim
object MeuSingleton {
    fun fazerAlgo() {
        // ...
    }
}
class MeuServico {
    fun fazerAlgo() {
        MeuSingleton.fazerAlgo()
    }
}
val meuServico = MeuServico()
meuServico.fazerAlgo()
```

```
// Bom
interface IMeuServico {
    fun fazerAlgo()
}
class MeuServico : IMeuServico {
    override fun fazerAlgo() {...
    }
}
class MeuOutroServico(val meuServico: IMeuServico) {
    fun fazerAlgo() {
        meuServico.fazerAlgo()
    }
}
val meuServico: IMeuServico = MeuServico()
val meuOutroServico = MeuOutroServico(meuServico)
meuOutroServico.fazerAlgo()
```

Use Injeção de Dependência

```
// Ruim
class MeuServico {
    val repositorio = Repositorio()

    fun fazerAlgo() {
        val dados = repositorio.obterDados()
        // ...
    }
}

val meuServico = MeuServico()
meuServico.fazerAlgo()
```

```
// Bom
class MeuServico(val repositorio: IRepositorio) {
    fun fazerAlgo() {
        val dados = repositorio.obterDados()
        // ...
    }
}

val repositorio: IRepositorio = Repositorio()
val meuServico = MeuServico(repositorio)
meuServico.fazerAlgo()
```

Separação de Responsabilidades

- É um princípio de design de software que ajuda a organizar o código de forma que cada parte ou módulo do código tenha uma responsabilidade única e claramente definida. Isso é muitas vezes referido como o Princípio da Responsabilidade Única (SRP, do inglês Single Responsibility Principle).
- Vantagens:
  - Manutenibilidade: É mais fácil manter e modificar o código. Quando cada parte do código tem uma única responsabilidade, é mais fácil fazer alterações sem afetar outras partes do sistema.
  - Testabilidade: É mais fácil escrever testes para o código. Quando cada parte do código tem uma única responsabilidade, você pode escrever testes focados para cada parte individualmente.
  - Reusabilidade: É mais fácil reutilizar o código em outras partes do sistema ou em outros projetos. Quando cada parte do código tem uma única responsabilidade, é mais provável que você possa reutilizar essa parte em outros lugares.
  - Legibilidade: O código é mais fácil de entender. Quando cada parte do código tem uma única responsabilidade, é mais fácil para outros desenvolvedores (ou você mesmo no futuro) entender o que cada parte do código está fazendo.
  - Redução de Bugs: Menos propenso a bugs. Quando cada parte do código tem uma única responsabilidade, há menos chance de efeitos colaterais inesperados e, portanto, menos chance de bugs.
  - Facilita a Refatoração: Facilita a refatoração do código. Quando cada parte do código tem uma única responsabilidade, é mais fácil refatorar o código sem afetar outras partes do sistema.
  - Escalabilidade: Facilita a escalabilidade do código. Quando cada parte do código tem uma única responsabilidade, é mais fácil escalar o código, pois você pode modificar ou estender cada parte individualmente.

Use Funções Pequenas: Cada função deve fazer uma coisa e fazer bem. Isso torna o código mais legível e testável.

```
// Ruim
fun processUserInput(input: String) {
    val sanitizedInput = input.trim().toLowerCase()
    val isValid = validateInput(sanitizedInput)
    if (!isValid) {
        println("Input inválido")
        return
    }
    val result = performAction(sanitizedInput)
    println("Resultado: $result")
}
```

```
// Bom
fun processUserInput(input: String) {
    val sanitizedInput = sanitizeInput(input)
    if (!validateInput(sanitizedInput)) {
        println("Input inválido")
        return
    }
    val result = performAction(sanitizedInput)
    println("Resultado: $result")
}
```

Use Classes Pequenas: Cada classe deve ter uma única responsabilidade. Isso torna o código mais modular e flexível.

```
// Ruim
class OrderProcessor {
    fun validateOrder(order: Order) { /* ... */ }
    fun calculateTotal(order: Order): Double { /* ... */ }
    fun sendEmail(order: Order, total: Double) { /* ... */ }
}
```

```
// Bom
class OrderValidator {
    fun validate(order: Order) { /* ... */ }
}

class OrderCalculator {
    fun calculateTotal(order: Order): Double { /* ... */ }
}

class EmailSender {
    fun sendEmail(order: Order, total: Double) { /* ... */ }
}
```

Evite Efeitos Colaterais: Funções e classes não devem ter efeitos colaterais inesperados. Eles devem fazer exatamente o que dizem e nada mais.

```
// Ruim
fun calculateTotal(order: Order): Double {
    sendEmail(order)
    // ...
}
```

```
// Bom
fun calculateTotal(order: Order): Double {
    // ...
}
```

Evitar repetição

- A prática de evitar repetição, conhecida como DRY (Don't Repeat Yourself), é fundamental para manter o código limpo, eficiente e de fácil manutenção.
- Repetição desnecessária pode levar a inconsistências, aumento do esforço de manutenção e dificuldade em fazer alterações.
- Vantagens de Evitar Repetição:
  - Manutenção Simplificada: Alterações precisam ser feitas em um único lugar, reduzindo a chance de inconsistências.
  - Código Mais Limpo: Eliminar repetição torna o código mais conciso e legível.
  - Redução de Erros: Repetição pode levar a erros ao copiar e colar código.
  - Facilitação de Refatoração: Alterações na lógica podem ser feitas sem afetar várias partes do código.
- Lembre-se de que a repetição pode ocorrer não apenas em código, mas também em estruturas de dados, consultas a bancos de dados, etc. Sempre procure oportunidades para centralizar a lógica e eliminar redundâncias.

Evitar repetição

```
//Ruim
fun calculateAreaOfRectangle(width: Double, height: Double): Double {
    val area = width * height
    println("A área é: $area")
    return area
}
fun calculateAreaOfSquare(side: Double): Double {
    val area = side * side
    println("A área é: $area")
    return area
}
```

Neste exemplo, há repetição na lógica de cálculo da área e na exibição da mensagem. Isso torna o código mais difícil de manter, porque qualquer mudança precisa ser feita em várias partes.

```
//Bom
fun calculateArea(width: Double, height: Double): Double {
    val area = width * height
    return area
}
fun printArea(area: Double) {
    println("A área é: $area")
}
```

Neste exemplo, a repetição foi eliminada, combinando a lógica de cálculo da área em uma única função. A função printArea é responsável apenas por exibir a mensagem. Isso torna o código mais enxuto, mais fácil de manter e mais flexível para futuras alterações.

Estruturas de Controle Simples

- O princípio chamado "Estruturas de Controle Simples" refere-se à importância de manter as estruturas de controle (como loops e condicionais) o mais simples e legível possível, facilitando a manutenção e a compreensão do código por outros desenvolvedores.
- Vantagens:
  - Legibilidade: Códigos mais simples são mais fáceis de ler e entender. Isso é especialmente importante em projetos de equipe, onde várias pessoas podem estar trabalhando no mesmo código.
  - Manutenção: Códigos mais simples são mais fáceis de manter. Se o código é fácil de entender, é mais fácil de modificar sem introduzir bugs.
  - Testabilidade: Códigos mais simples são mais fáceis de testar. Se uma função faz apenas uma coisa e faz bem, é mais fácil escrever testes para essa função.
  - Reusabilidade: Códigos mais simples são mais fáceis de reutilizar. Se você mantiver suas funções pequenas e focadas, é mais provável que você possa reutilizá-las em outras partes do seu código.
  - Redução de Bugs: Quanto mais complexa é uma estrutura de controle, maior é a chance de cometer um erro. Simplificar as estruturas de controle ajuda a reduzir a chance de bugs.
  - Colaboração: Facilita a colaboração entre os membros da equipe. Quando o código é simples e limpo, é mais fácil para os outros membros da equipe entenderem e contribuírem com o código.
  - Performance: Códigos mais simples muitas vezes levam a uma melhor performance, pois há menos lógica para executar.

Evite Aninhamento Profundo: Tente evitar o aninhamento profundo de estruturas de controle. Isso pode tornar o código difícil de ler e entender.

```
// Ruim
for (i in 1..10) {
    for (j in 1..10) {
        if (i * j > 50) {
            println("$i * $j = ${i * j}")
        }
    }
}
```

```
// Bom
for (i in 1..10) {
    for (j in 1..10) {
        printProductIfGreaterThan50(i, j)
    }
}

fun printProductIfGreaterThan50(i: Int, j: Int) {
    if (i * j > 50) {
        println("$i * $j = ${i * j}")
    }
}
```

Use Expressões de Controle: Em Kotlin, você pode usar expressões de controle, como if e when, que retornam um valor. Isso pode ajudar a simplificar o código.

```
// Ruim
val max: Int
if (a > b) {
    max = a
} else {
    max = b
}
```

```
// Bom
val max = if (a > b) a else b
```

Evite else quando possível: O uso excessivo de else pode tornar o código difícil de ler. Em muitos casos, você pode usar return ou continue para evitar o uso de else.

```
// Ruim
fun findEvenNumber(numbers: List<Int>): Int? {
    for (number in numbers) {
        if (number % 2 == 0) {
            return number
        } else {
            continue
        }
    }
    return null
}
```

```
// Bom
fun findEvenNumber(numbers: List<Int>): Int? {
    for (number in numbers) {
        if (number % 2 == 0) {
            return number
        }
    }
    return null
}
```

SOLID

Princípios SOLID

- Os princípios SOLID são um conjunto de cinco princípios de design de software que ajudam a criar sistemas mais manuteníveis, flexíveis e robustos. Eles são amplamente utilizados na programação orientada a objetos. Aqui estão os cinco princípios:
- S - Princípio da Responsabilidade Única (Single Responsibility Principle - SRP)
- O - Princípio Aberto/Fechado (Open/Closed Principle - OCP)
- L - Princípio da Substituição de Liskov (Liskov Substitution Principle – LSP)
- I - Princípio da Segregação de Interface (Interface Segregation Principle – ISP)
- D - Princípio da Inversão de Dependência (Dependency Inversion Principle – DIP)

Single Responsibility Principle - SRP

- O Princípio da Responsabilidade Única (Single Responsibility Principle - SRP) é um dos cinco princípios SOLID de design de software orientado a objetos.
- Este princípio afirma que uma classe deve ter apenas um motivo para mudar, ou seja, deve ter apenas uma responsabilidade.
- Por exemplo, considere um aplicativo Android que tem uma tela para mostrar informações de um usuário e também para salvar as informações do usuário no banco de dados.
  - De acordo com o SRP, deveríamos ter uma classe separada para buscar as informações do usuário do banco de dados, outra classe para mostrar as informações do usuário na tela, e outra classe para salvar as informações do usuário no banco de dados.

SRP

```
// Ruim
class UsuarioActivity : AppCompatActivity() {
    fun carregarUsuario() {
        // Carregar usuário do banco de dados
    }
    fun mostrarUsuario() {
        // Mostrar usuário na tela
    }
    fun salvarUsuario() {
        // Salvar usuário no banco de dados
    }
}
```

Neste exemplo, a UsuarioActivity tem três responsabilidades: carregar o usuário do banco de dados, mostrar o usuário na tela e salvar o usuário no banco de dados. Isso viola o SRP.

SRP

Neste exemplo, a UsuarioRepository tem a responsabilidade de carregar e salvar o usuário no banco de dados, e a UsuarioActivity tem a responsabilidade de mostrar o usuário na tela. Cada classe tem apenas uma responsabilidade, portanto, este exemplo segue o SRP.

```
// Ruim
class UsuarioActivity : AppCompatActivity() {
    fun carregarUsuario() {
        // Carregar usuário do banco de dados
    }
    fun mostrarUsuario() {
        // Mostrar usuário na tela
    }
    fun salvarUsuario() {
        // Salvar usuário no banco de dados
    }
}
```

```
// Bom
class UsuarioRepository {
    fun carregarUsuario(): Usuario {
        // Carregar usuário do banco de dados
    }
    fun salvarUsuario(usuario: Usuario) {
        // Salvar usuário no banco de dados
    }
}
class UsuarioActivity : AppCompatActivity() {
    private val usuarioRepository = UsuarioRepository()
    fun mostrarUsuario() {
        val usuario = usuarioRepository.carregarUsuario()
        // Mostrar usuário na tela
    }
    fun salvarUsuario(usuario: Usuario) {
        usuarioRepository.salvarUsuario(usuario)
    }
}
```

Single Responsibility Principle - SRP

- Por exemplo, considere um aplicativo iOS que tem uma tela para mostrar informações de uma lista de tarefas e também por pegar as informações de uma API.
- De acordo com o SRP, deveríamos ter uma classe separada para buscar as informações da API, outra classe para mostrar as informações das tarefas.

SRP

```
// Ruim
struct ContentView: View {
    @State private var tasks: [Task] = []
    var body: some View {
        List(tasks) { task in
            Text(task.name)
        }
        .onAppear {
            // Buscar tarefas da API
            APIService.fetchTasks { fetchedTasks in
                self.tasks = fetchedTasks
            }  }}
struct Task: Identifiable {
    var id: Int
    var name: String
}
class APIService {
    static func fetchTasks(completion: @escaping ([Task]) -> Void) {// Buscar tarefas da API}}
```

Neste exemplo, o ContentView é responsável por mostrar a lista de tarefas e também por buscar as tarefas da API. Isso viola o SRP.

SRP

Neste exemplo, o TaskViewModel é responsável por buscar as tarefas da API, e o ContentView é responsável por mostrar a lista de tarefas. Cada classe tem apenas uma responsabilidade, portanto, este exemplo segue o SRP.

```
// Bom
struct ContentView: View {
    @StateObject private var taskViewModel = TaskViewModel()
    var body: some View {
        List(taskViewModel.tasks) { task in
            Text(task.name)}
        .onAppear {
            taskViewModel.fetchTasks()
        }}}
class TaskViewModel: ObservableObject {
    @Published var tasks: [Task] = []
    func fetchTasks() {
        APIService.fetchTasks { fetchedTasks in
            self.tasks = fetchedTasks
        }}}
struct Task: Identifiable {
    var id: Int
    var name: String
}
class APIService {
    static func fetchTasks(completion: @escaping ([Task]) -> Void) {// Buscar tarefas da API
    }}
```

Open/Closed Principle - OCP

- O Princípio Aberto/Fechado (Open/Closed Principle - OCP) é outro dos cinco princípios SOLID de design de software orientado a objetos. Este princípio afirma que entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação.
- Isso significa que você deve ser capaz de adicionar novas funcionalidades ao sistema sem modificar o código existente.
- Por exemplo, considere um aplicativo que envia notificações para os usuários. Inicialmente, o aplicativo suporta apenas notificações por email, mas no futuro, queremos adicionar suporte para notificações por SMS.

```
//Ruim
struct Notificacao {
    func enviar(mensagem: String, tipo: String) {
        switch tipo {
        case "email":
            // Enviar email
        case "sms":
            // Enviar SMS
        default:
            break
        }}}
struct ContentView: View {
    var body: some View {
        Button(action: {
            let notificacao = Notificacao()
            notificacao.enviar(mensagem: "Olá, mundo!", tipo: "email")
        }) {
            Text("Enviar Notificação")
        }}}
```

Neste exemplo, se quisermos adicionar suporte para notificações por push, teríamos que modificar a classe Notificacao e adicionar um novo caso ao switch. Isso viola o OCP.

SRP

Neste exemplo, se quisermos adicionar suporte para notificações por push, podemos simplesmente criar uma nova classe que implementa a interface Notificacao e passá-la para o NotificacaoService.
Não precisamos modificar nenhuma das classes existentes, portanto, este exemplo segue o OCP.

```
protocol Notificacao { func enviar(mensagem: String)}
struct Email: Notificacao { func enviar(mensagem: String) {}}
struct SMS: Notificacao { func enviar(mensagem: String) {}}
struct NotificacaoService {
    private let notificacao: Notificacao
    init(notificacao: Notificacao) {
        self.notificacao = notificacao
    }
    func enviar(mensagem: String) {
        notificacao.enviar(mensagem: mensagem)
    }
}
struct ContentView: View {
    var body: some View {
        Button(action: {
            let notificacaoService = NotificacaoService(notificacao: Email())
            notificacaoService.enviar(mensagem: "Olá, mundo!")
        }) {
            Text("Enviar Notificação")
        } }}
```

Liskov Substitution Principle - LSP

- O Princípio de Substituição de Liskov (Liskov Substitution Principle - LSP) é um dos cinco princípios SOLID de design de software orientado a objetos.
- Este princípio afirma que objetos de uma superclasse devem ser substituíveis por objetos de suas subclasses sem causar problemas no programa.

- Por exemplo, considere um aplicativo que tem uma classe Ave e uma classe Pato que herda de Ave. A classe Ave tem um método voar que é implementado na classe Pato.

```
open class Ave {
    open fun voar() {}
}
class Pato : Ave() {
    override fun voar() {}
}
class Avestruz : Ave() {
    override fun voar() {
        throw UnsupportedOperationException("Avestruzes não podem voar")
    }
}
fun main() {
    val aves: List<Ave> = listOf(Pato(), Avestruz())
    for (ave in aves) {
        ave.voar()
    }
}
```

Neste exemplo, a Avestruz é uma subclasse de Ave, mas ela não pode voar. Isso viola o LSP porque não podemos substituir Ave por Avestruz sem alterar o comportamento do programa.

Neste exemplo, Ave é uma interface e Avestruz implementa Ave sem lançar uma exceção. Podemos substituir Ave por Pato ou Avestruz sem alterar o comportamento do programa, portanto, este exemplo segue o LSP.

```
nterface Ave {
    fun voar()
}
class Pato : Ave {
    override fun voar() {
        // Voar como um pato
    }
}
class Avestruz : Ave {
    override fun voar() {
        // Não fazer nada
    }
}
fun main() {
    val aves: List<Ave> = listOf(Pato(), Avestruz())
    for (ave in aves) {
        ave.voar()
    }
}
```

Interface Segregation Principle - ISP

- O Princípio da Segregação de Interface (Interface Segregation Principle - ISP) é um dos cinco princípios SOLID de design de software orientado a objetos.
- Este princípio afirma que os clientes não devem ser forçados a depender de interfaces que não usam.
- Em outras palavras, é melhor ter várias interfaces específicas do que uma interface geral.
- Por exemplo, considere um aplicativo que tem uma interface Impressora com métodos para imprimir, escanear e fax. Uma classe ImpressoraMultifuncional implementa todos os métodos, mas uma classe ImpressoraSimples só pode imprimir.

```
interface Impressora {
    fun imprimir()
    fun escanear()
    fun enviarFax()
}
class ImpressoraMultifuncional : Impressora {
    override fun imprimir() {
}
    override fun escanear() {
}
    override fun enviarFax() {
}
}
class ImpressoraSimples : Impressora {
    override fun imprimir() {
     }
    override fun escanear() {
        throw UnsupportedOperationException("Esta impressora não pode escanear")
    }
    override fun enviarFax() {
        throw UnsupportedOperationException("Esta impressora não pode enviar fax")
    }
}
```

Neste exemplo, a ImpressoraSimples é forçada a implementar métodos que não precisa. Isso viola o ISP.

Neste exemplo, temos três interfaces específicas e a ImpressoraSimples implementa apenas a interface que precisa. Portanto, este exemplo segue o ISP.

```
interface Impressora {
    fun imprimir()
}
interface Scanner {
    fun escanear()
}
interface Fax {
    fun enviarFax()
}
class ImpressoraMultifuncional : Impressora, Scanner, Fax {
    override fun imprimir() {}
    override fun escanear() {}
    override fun enviarFax() {}
}
class ImpressoraSimples : Impressora {
    override fun imprimir() {}
}
```

Dependency Inversion Principle - DIP

- O Princípio da Inversão de Dependência (Dependency Inversion Principle - DIP) é o último dos cinco princípios SOLID de design de software orientado a objetos.
- Este princípio afirma que módulos de alto nível não devem depender de módulos de baixo nível.
  - Ambos devem depender de abstrações.
  - Além disso, abstrações não devem depender de detalhes.
  - Detalhes devem depender de abstrações.
- Por exemplo, considere um aplicativo que tem uma classe Aplicativo que depende de uma classe Servico.

```
class Servico {
    fun executar() {
        // Executar serviço
    }
}

class Aplicativo {
    private val servico = Servico()

    fun iniciar() {
        servico.executar()
    }
}

fun main() {
    val aplicativo = Aplicativo()
    aplicativo.iniciar()
}
```

Neste exemplo, a classe Aplicativo de alto nível depende diretamente da classe Servico de baixo nível. Isso viola o DIP.

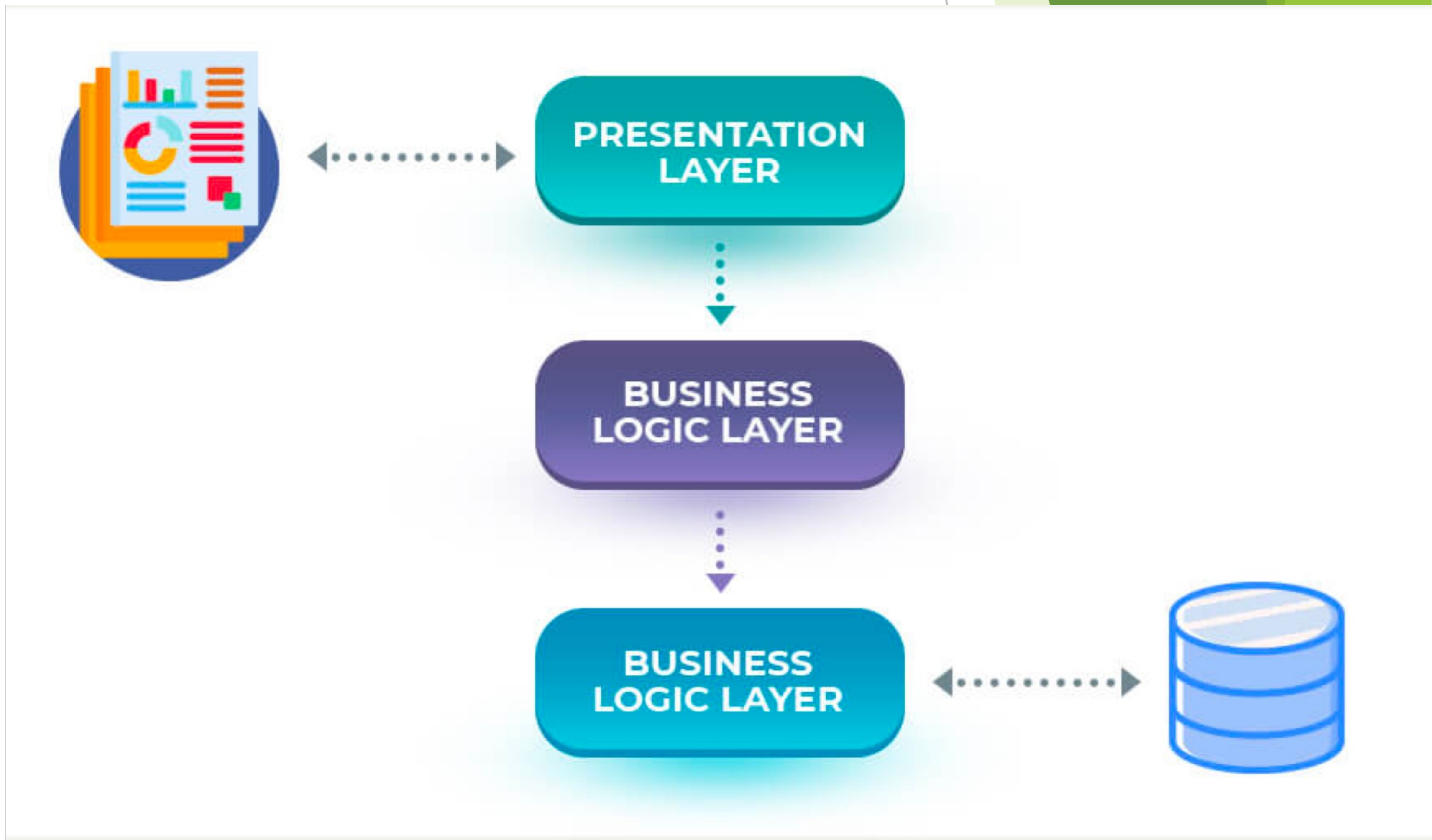
Neste exemplo, a classe Aplicativo de alto nível e a classe ServicoImpl de baixo nível dependem da interface Servico. Portanto, este exemplo segue o DIP.

```
interface Servico {
    fun executar()
}

class ServicoImpl : Servico {
    override fun executar() {
        // Executar serviço
    }
}

class Aplicativo(private val servico: Servico) {
    fun iniciar() {
        servico.executar()
    }
}

fun main() {
    val servico: Servico = ServicoImpl()
    val aplicativo = Aplicativo(servico)
    aplicativo.iniciar()
}
```

Arquiteturas Mobile

Padrões de arquitetura

- Técnicas e padrões usados para estruturar os projetos de Apps



Importância da arquitetura de aplicativos móveis

- Manutenibilidade: Uma boa arquitetura facilita a manutenção do código. Isso significa que será mais fácil e mais rápido fazer alterações no código, corrigir bugs e adicionar novos recursos.
- Testabilidade: Uma arquitetura bem projetada facilita a escrita de testes para o aplicativo. Isso é crucial para garantir que o aplicativo funcione corretamente e para evitar a introdução de novos bugs ao fazer alterações no código.
- Reutilização de Código: Uma arquitetura bem projetada promove a reutilização de código. Isso significa que você pode usar o mesmo código em diferentes partes do aplicativo ou até mesmo em diferentes aplicativos.

Importância da arquitetura de aplicativos móveis

- Desempenho: Uma arquitetura bem projetada pode ajudar a melhorar o desempenho do aplicativo. Isso é especialmente importante para aplicativos móveis, onde os recursos do dispositivo são limitados.
- Escalabilidade: Uma boa arquitetura facilita a escalabilidade do aplicativo. Isso significa que será mais fácil adicionar novos recursos e funcionalidades ao aplicativo à medida que ele cresce.
- Colaboração: Uma arquitetura clara e bem definida facilita a colaboração entre diferentes membros de uma equipe de desenvolvimento. Isso é especialmente importante em projetos maiores, onde várias pessoas podem estar trabalhando no mesmo código.

Importância da arquitetura de aplicativos móveis

- Qualidade do Código: Uma boa arquitetura ajuda a manter a qualidade do código. Isso significa que o código será mais limpo, mais organizado e mais fácil de entender.
- Redução de Bugs: Uma arquitetura bem projetada ajuda a reduzir a quantidade de bugs no aplicativo. Isso é crucial para a satisfação do usuário e para a reputação do aplicativo.
- Facilita a Refatoração: Uma arquitetura bem estruturada facilita a refatoração do código, o que é importante para manter a qualidade do código ao longo do tempo.
- Flexibilidade: Uma boa arquitetura permite flexibilidade para mudanças futuras, seja na adição de novos recursos, na modificação de funcionalidades existentes ou na integração com outros sistemas.

Vantagens do uso de padrões de arquitetura

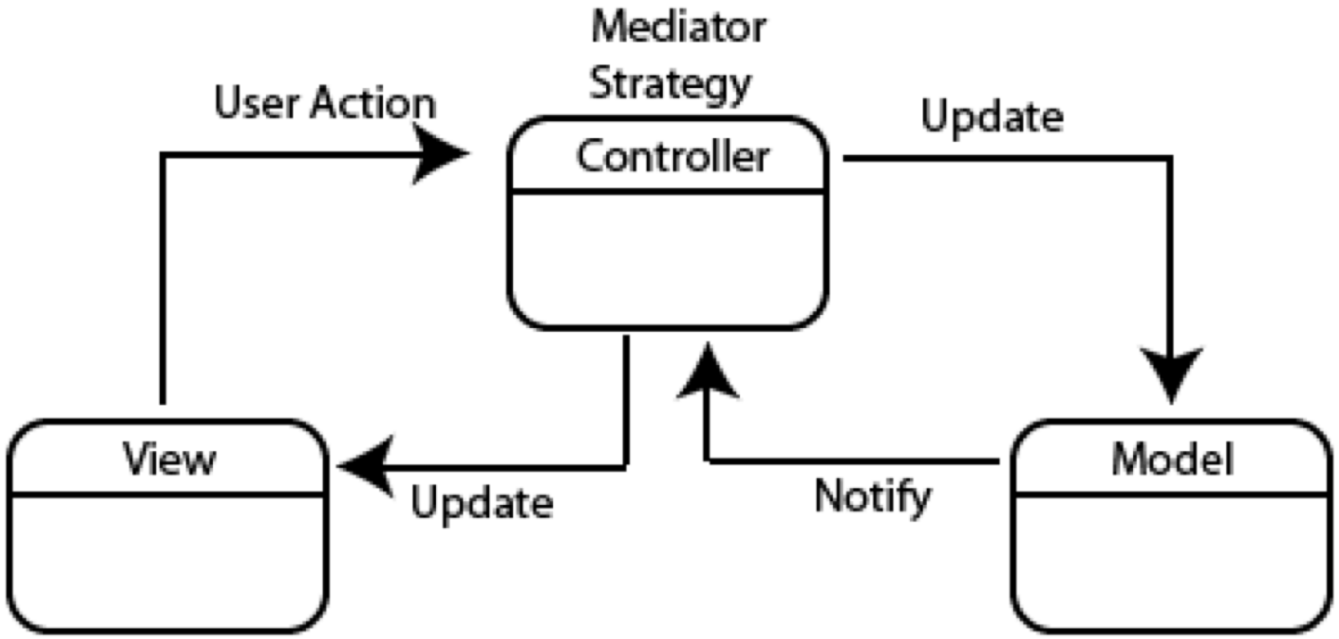
- Maior qualidade do código
- Maior padronização
- Maior facilidade para testes
- Facilidade de manutenção do código;
- Maior tempo no começo do desenvolvimento - e menor conforme o projeto cresce

MVC

- O MVC (Model-View-Controller) é um padrão de design amplamente utilizado para organizar o código de um aplicativo de maneira lógica e modular. Ele divide o código em três componentes principais:
- Model: Representa os dados e a lógica de negócios do aplicativo. É responsável por armazenar o estado do aplicativo e manipular esses dados conforme necessário. O Model não sabe nada sobre a interface do usuário e pode ser usado em diferentes interfaces ou até mesmo sem interface, como em um aplicativo de linha de comando.
- View: Representa a interface do usuário e a apresentação dos dados. A View observa o Model e se atualiza quando os dados no Model mudam. A View não sabe nada sobre a lógica de negócios e pode ser reutilizada com diferentes Models.
- Controller: Atua como intermediário entre o Model e a View. Ele processa a entrada do usuário, atualiza o Model e, em seguida, atualiza a View. O Controller contém a lógica de controle do aplicativo e é responsável por coordenar o Model e a View.

MVC

- A principal vantagem do MVC é a separação de preocupações, o que torna o código mais modular, mais fácil de entender, testar e manter.
- Cada componente tem uma responsabilidade clara e bem definida, e o código pode ser organizado de forma que os diferentes componentes possam ser desenvolvidos, testados e modificados independentemente uns dos outros.


Mediator
Strategy
User Action
Controller
Update
View
Update
Notify
Model

MVC

vantagens do mvc

- Separação de Preocupações: O MVC separa o aplicativo em três componentes lógicos: Model, View e Controller. Isso facilita a organização do código e a separação das responsabilidades, tornando o código mais modular e mais fácil de entender, testar e manter.
- Reusabilidade de Código: A separação de preocupações facilita a reutilização do código. Por exemplo, o mesmo Model pode ser usado com diferentes Views, e o mesmo View pode ser usado com diferentes Models.
- Testabilidade: A separação de preocupações também facilita a escrita de testes unitários para o código. Por exemplo, o Model pode ser testado independentemente da interface do usuário (UI).
- Flexibilidade: O MVC permite que diferentes desenvolvedores trabalhem em diferentes partes do aplicativo ao mesmo tempo. Por exemplo, um desenvolvedor pode trabalhar na View enquanto outro trabalha no Controller.
- Escalabilidade: O MVC facilita a escalabilidade do aplicativo, pois diferentes partes do aplicativo podem ser desenvolvidas e escaladas independentemente.

desvantagens do mvc

- Complexidade: Para aplicativos simples, o MVC pode parecer excessivamente complexo e pode levar mais tempo para desenvolver do que um aplicativo monolítico simples.
- Sobrecarga de Desempenho: A separação de preocupações e a comunicação entre o Model, View e Controller podem levar a uma sobrecarga de desempenho, especialmente para aplicativos muito grandes ou complexos.
- Curva de Aprendizado: Para desenvolvedores novos no MVC, pode haver uma curva de aprendizado inicial para entender como o padrão funciona e como usar efetivamente.
- Risco de Desenvolvimento Desorganizado: Se não for implementado corretamente, o MVC pode levar a um desenvolvimento desorganizado, onde a lógica do aplicativo é distribuída de forma confusa entre o Model, View e Controller.

MVP

- O MVP (Model-View-Presenter) é um padrão de design que é uma variação do padrão MVC (Model-View-Controller).
- Ele é usado para organizar o código de forma que a lógica de negócios, a interface do usuário e a apresentação dos dados sejam separadas.
- O MVP divide o aplicativo em três componentes principais:

MVP

- Model: Representa os dados e a lógica de negócios do aplicativo. É responsável por armazenar o estado do aplicativo e manipular esses dados conforme necessário. O Model não sabe nada sobre a interface do usuário e pode ser usado em diferentes interfaces ou até mesmo sem interface, como em um aplicativo de linha de comando.

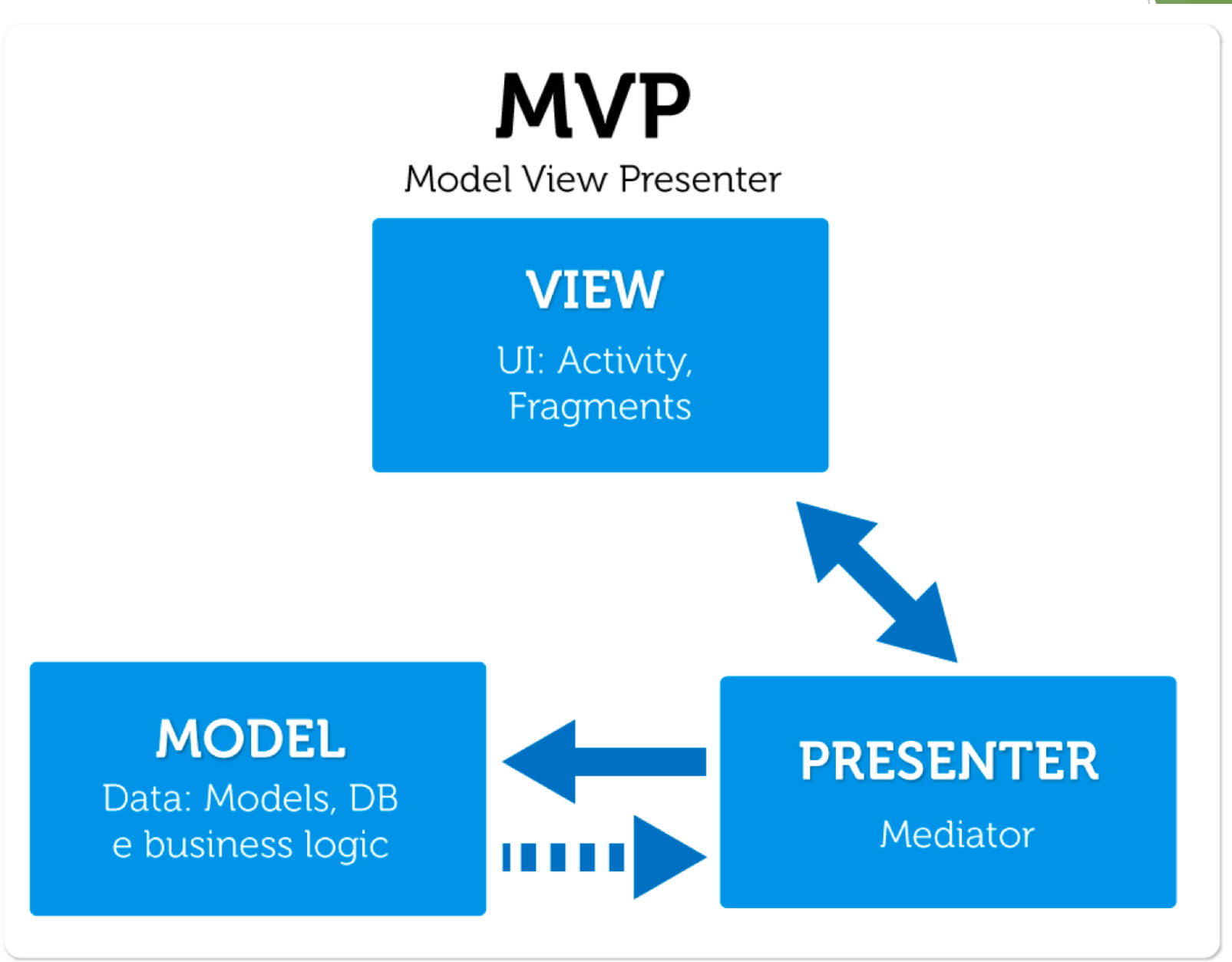
- View: Representa a interface do usuário e a apresentação dos dados. A View recebe as entradas do usuário e as passa para o Presenter. A View não sabe nada sobre a lógica de negócios e pode ser reutilizada com diferentes Presenters.

- Presenter: Atua como intermediário entre o Model e a View. Ele recupera os dados do Model, os processa e os passa para a View para serem exibidos. O Presenter também recebe as entradas do usuário da View e as processa, o que pode envolver a atualização do Model. O Presenter não sabe nada sobre a interface do usuário.

MVP

- A principal diferença entre o MVP e o MVC é que no MVP, o Presenter é responsável por processar as entradas do usuário, enquanto no MVC, o Controller é responsável por esse processamento.
- Modificação do MVC, separando totalmente a lógica dos dados e da UI
- Model -igual ao MVC
- View - responsável por tudo relacionado a visualização incluindo tratamento de inputs
- Presenter - igual ao MVC, mas simplificado e sem depender da view

MVP


MVP
Model View Presenter
VIEW
UI: Activity,
Fragments
MODEL
Data: Models, DB
e business logic
PRESENTER
Mediator

MVP - vantagens

- Separação de Preocupações: O MVP promove uma clara separação de preocupações entre a lógica de negócios, a interface do usuário e a apresentação dos dados, o que torna o código mais modular e fácil de entender, testar e manter.
- Testabilidade: A separação entre a View e o Presenter facilita a escrita de testes unitários para a lógica de negócios sem a necessidade de interagir com a interface do usuário.
- Reusabilidade: O Presenter não tem conhecimento da View, o que significa que o mesmo Presenter pode ser reutilizado com diferentes Views.
- Manutenção: Como o MVP separa a lógica de negócios da interface do usuário, é mais fácil fazer alterações na interface do usuário sem afetar a lógica de negócios e vice-versa.

MVP - desvantagens

- Complexidade: Para aplicativos simples, o MVP pode parecer excessivamente complexo e pode levar mais tempo para desenvolver do que um aplicativo monolítico simples.
- Curva de Aprendizado: Para desenvolvedores novos no MVP, pode haver uma curva de aprendizado inicial para entender como o padrão funciona e como usar efetivamente.
- Desempenho: A necessidade de comunicar-se entre a View e o Presenter pode levar a uma sobrecarga de desempenho, especialmente para aplicativos muito grandes ou complexos.
- Risco de Desenvolvimento Desorganizado: Se não for implementado corretamente, o MVP pode levar a um desenvolvimento desorganizado, onde a lógica do aplicativo é distribuída de forma confusa entre o Model, View e Presenter.

MVVM

- O MVVM (Model-View-ViewModel) é um padrão de design que é uma evolução do padrão MVC (Model-View-Controller).
- Ele é especialmente útil para desenvolvimento de aplicativos que utilizam frameworks de vinculação de dados, como o Android Data Binding, Android JetPack Composer e o SwiftUI com Combine.
- O MVVM divide o aplicativo em três componentes principais:

MVVM

- Model: Representa os dados e a lógica de negócios do aplicativo. É responsável por armazenar o estado do aplicativo e manipular esses dados conforme necessário. O Model não sabe nada sobre a interface do usuário e pode ser usado em diferentes interfaces ou até mesmo sem interface, como em um aplicativo de linha de comando.
- View: Representa a interface do usuário e a apresentação dos dados. A View observa o ViewModel e se atualiza quando os dados no ViewModel mudam. A View não sabe nada sobre a lógica de negócios e pode ser reutilizada com diferentes ViewModels.
- ViewModel: Atua como intermediário entre o Model e a View. Ele expõe os dados do Model de uma maneira que é fácil de usar para a View. O ViewModel não sabe nada sobre a interface do usuário e pode ser reutilizado com diferentes Views.

MVVM

- A principal vantagem do MVVM é que ele facilita a vinculação de dados entre a View e o ViewModel, o que pode reduzir a quantidade de código de cola necessária e tornar o código mais limpo e mais fácil de entender e manter.

MVVM - Vantagens

- Separação de Preocupações: O MVVM promove uma clara separação de preocupações entre a interface do usuário e a lógica de negócios, o que torna o código mais modular e fácil de entender, testar e manter.
- Testabilidade: A separação entre a View e o ViewModel facilita a escrita de testes unitários para a lógica de negócios sem a necessidade de interagir com a interface do usuário.
- Reusabilidade: O ViewModel não tem conhecimento da View, o que significa que o mesmo ViewModel pode ser reutilizado com diferentes Views.
- Vinculação de Dados: O MVVM facilita a vinculação de dados entre a View e o ViewModel, o que pode reduzir a quantidade de código de cola necessária e tornar o código mais limpo.
- Manutenção: Como o MVVM separa a lógica de negócios da interface do usuário, é mais fácil fazer alterações na interface do usuário sem afetar a lógica de negócios e vice-versa.

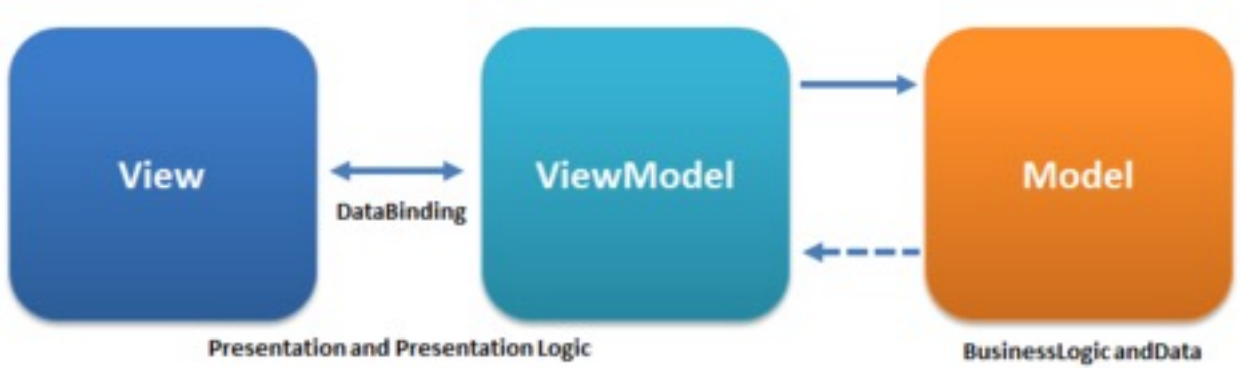
MVVM - desvantagens

- Complexidade: Para aplicativos simples, o MVVM pode parecer excessivamente complexo e pode levar mais tempo para desenvolver do que um aplicativo monolítico simples.
- Curva de Aprendizado: Para desenvolvedores novos no MVVM, pode haver uma curva de aprendizado inicial para entender como o padrão funciona e como usar efetivamente.
- Desempenho: A vinculação de dados e a observação de mudanças nos dados podem levar a uma sobrecarga de desempenho, especialmente para aplicativos muito grandes ou complexos.
- Risco de Desenvolvimento Desorganizado: Se não for implementado corretamente, o MVVM pode levar a um desenvolvimento desorganizado, onde a lógica do aplicativo é distribuída de forma confusa entre o Model, View e ViewModel.

MVVM

- Model -igual ao MVC/MVP
- View - igual ao do MVP
- ViewModel tem a mesma funcionalidade que no MVC/MVP, mas a comunicação é feita através de observadores e notificações
- Usado no novo framework SwiftUI e pela Google no pacote Jetpack Compose

MVVM


View
DataBinding
ViewModel
Model
Presentation and Presentation Logic
BusinessLogic andData

Viper

- O VIPER é um padrão de arquitetura de software que visa organizar o código de um aplicativo em módulos lógicos e separados.
- O nome VIPER é um acrônimo para View, Interactor, Presenter, Entity e Router, que são os cinco componentes principais do padrão.

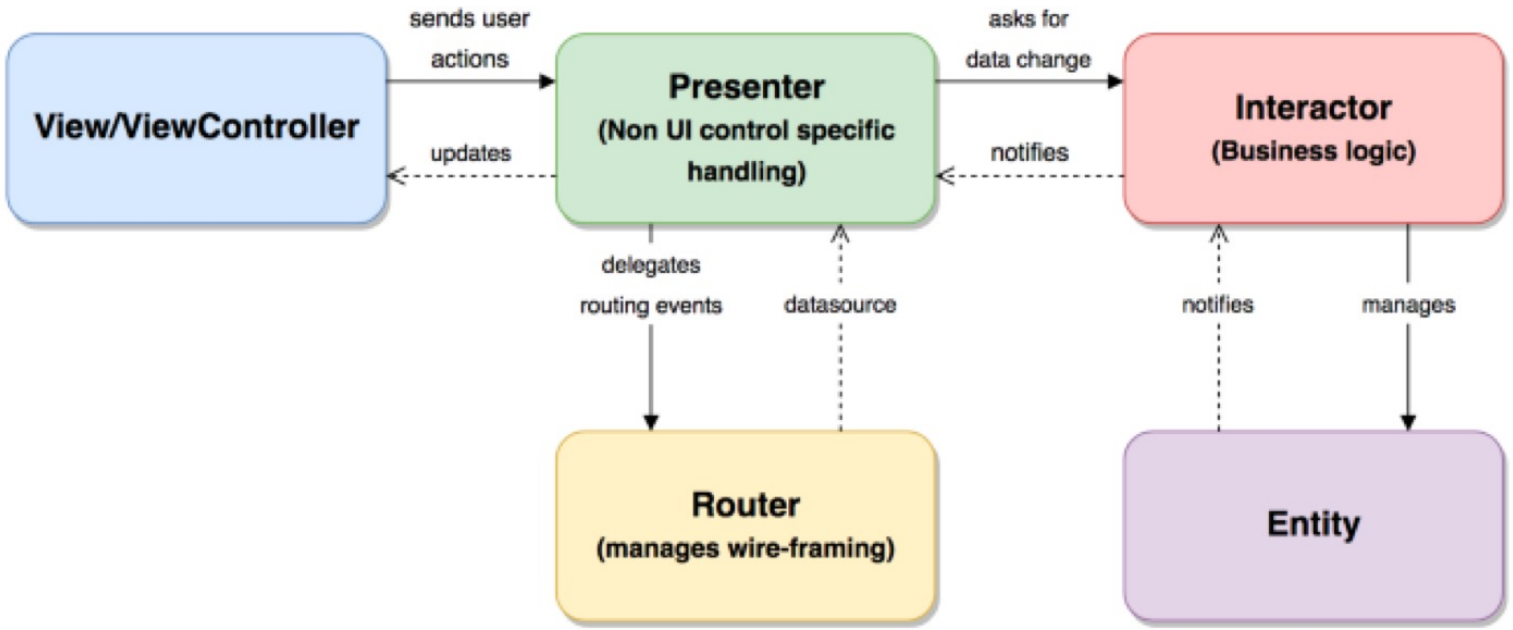
Viper

- View: É responsável pela apresentação da interface do usuário e pela captura de ações do usuário. A View comunica-se diretamente com o Presenter.
- Interactor: Contém toda a lógica de negócios relacionada ao módulo em questão. Ele recebe instruções do Presenter e, após processar os dados, retorna os resultados ao Presenter.
- Presenter: Atua como um intermediário entre a View e o Interactor. Ele recebe eventos da View, processa-os com a ajuda do Interactor e retorna os dados formatados de volta para a View.
- Entity: Representa os objetos de domínio ou modelos de dados. É basicamente o que o Interactor usa para realizar operações de negócios.
- Router: Gerencia a navegação entre diferentes módulos ou telas. Ele é chamado pelo Presenter para iniciar novas Views.

Viper

- View: responsável pela UI
- Presenter: responsável por tratar as ações por trás das entradas e preparar os dados para visualização
- Interactor: possui lógica de negócios
- Entity: contendo os Models usados pelo Interactor
- Router: possui a lógica da navegação

Viper


View/ViewController
sends user actions
updates
Presenter
(Non UI control specific handling)
asks for data change
notifies
Interactor
(Business logic)
delegates routing events
datasource
notifies
manages
Router
(manages wire-framing)
Entity

Viper - vantagens

- Separação de Responsabilidades: Cada componente tem responsabilidades bem definidas, tornando o código mais organizado e facilitando a manutenção.
- Testabilidade: Devido à separação de responsabilidades, é mais fácil escrever testes unitários para cada componente.
- Reusabilidade: Os componentes são independentes e podem ser reutilizados em diferentes partes do aplicativo ou até mesmo em diferentes projetos.
- Escalabilidade: VIPER é adequado para projetos grandes e complexos, onde a escalabilidade e a manutenibilidade são preocupações importantes.

Viper - desvantagens

- Complexidade: VIPER pode ser excessivamente complexo para aplicativos pequenos e simples, tornando-se um exagero para tais projetos.
- Curva de Aprendizado: Devido à sua complexidade e ao número de componentes, pode haver uma curva de aprendizado íngreme para desenvolvedores que são novos no VIPER.
- Verbosidade: A necessidade de criar vários componentes para cada módulo pode levar a um código mais extenso e, possivelmente, a duplicação de código.
- Menos Comunidade e Recursos: Embora seja popular, VIPER ainda não tem tantos recursos de aprendizado e exemplos de código disponíveis como outras arquiteturas como MVC ou MVVM.

Viper - desvantagens

- Em resumo, VIPER é uma arquitetura robusta que oferece muitos benefícios em termos de manutenibilidade e testabilidade, mas pode ser excessiva para projetos menores. É mais adequada para aplicativos grandes e complexos, onde a complexidade adicional pode ser justificada.

RIBs

- O RIBs é um framework de arquitetura de aplicativos móveis desenvolvido pelo Uber.
- Ideia de separar a lógica de negócios

- O nome RIBs é um acrônimo para Router, Interactor e Builder, que são os três componentes principais do framework.

RIBs

- Router: O Router é responsável por lidar com a navegação entre as telas do aplicativo. Ele decide qual tela mostrar a seguir e controla a transição entre as telas.

- Interactor: O Interactor contém a lógica de negócios do aplicativo. Ele interage com o Presenter para atualizar a View e com o Router para navegar entre as telas.

- Builder: O Builder é responsável por construir as dependências necessárias para o Router e o Interactor. Ele cria instâncias do Router, Interactor e Presenter e injeta as dependências necessárias.

RIBs

- Além desses três componentes, o RIBs também inclui a View e o Presenter, semelhantes ao MVP (Model-View-Presenter):

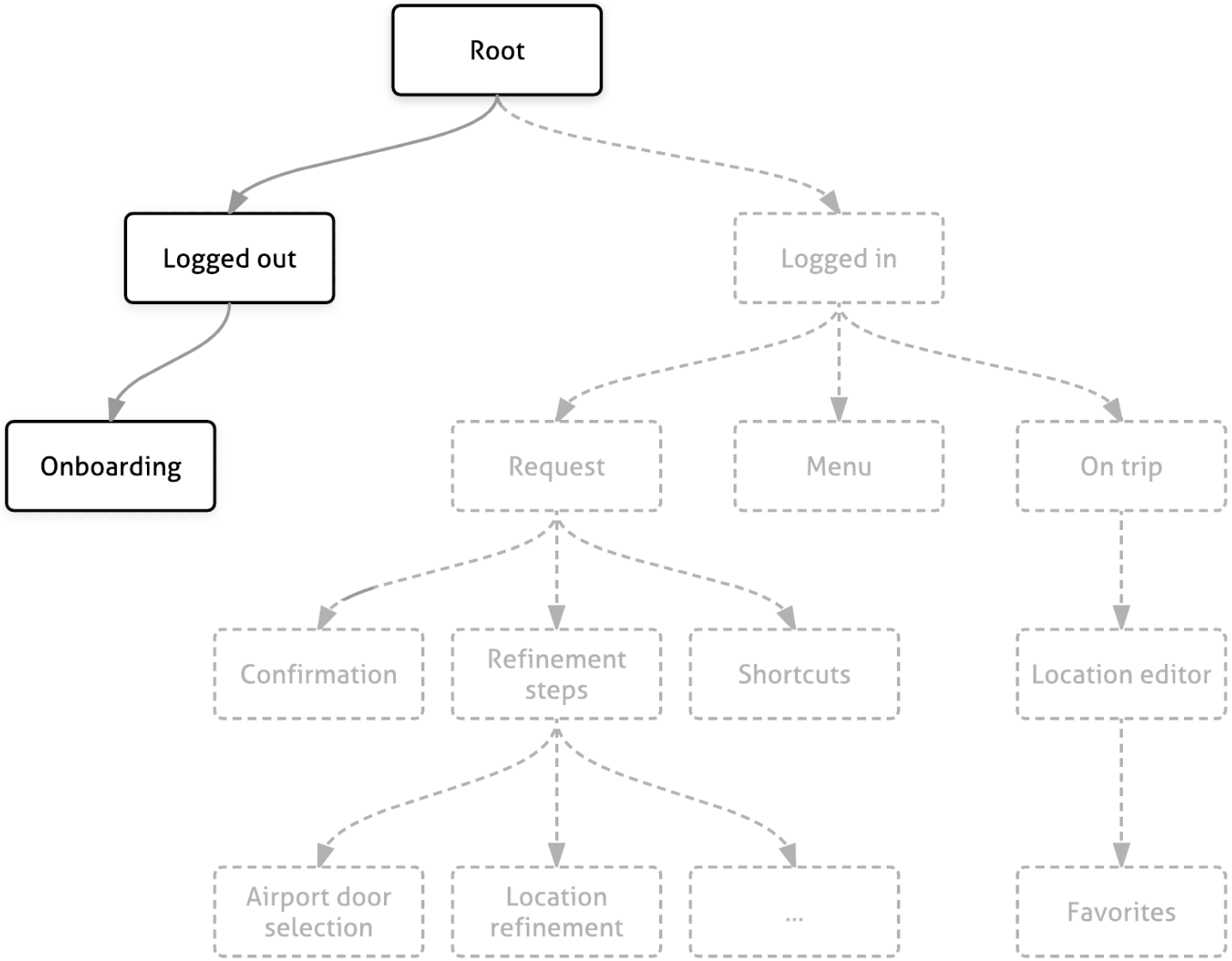
- View: A View é a interface do usuário do aplicativo. Ela define os métodos que o Presenter usará para atualizar a interface do usuário.

- Presenter: O Presenter interage com a View para atualizar a interface do usuário e com o Interactor para obter os dados necessários.

RIBs

- O RIBs organiza o código em uma árvore de RIBs, onde cada RIB é um módulo que contém seu próprio Router, Interactor, Builder, View e Presenter.
- Cada RIB na árvore tem um pai e pode ter vários filhos.
- O RIB raiz é o ponto de entrada do aplicativo e é responsável por iniciar o primeiro RIB filho.

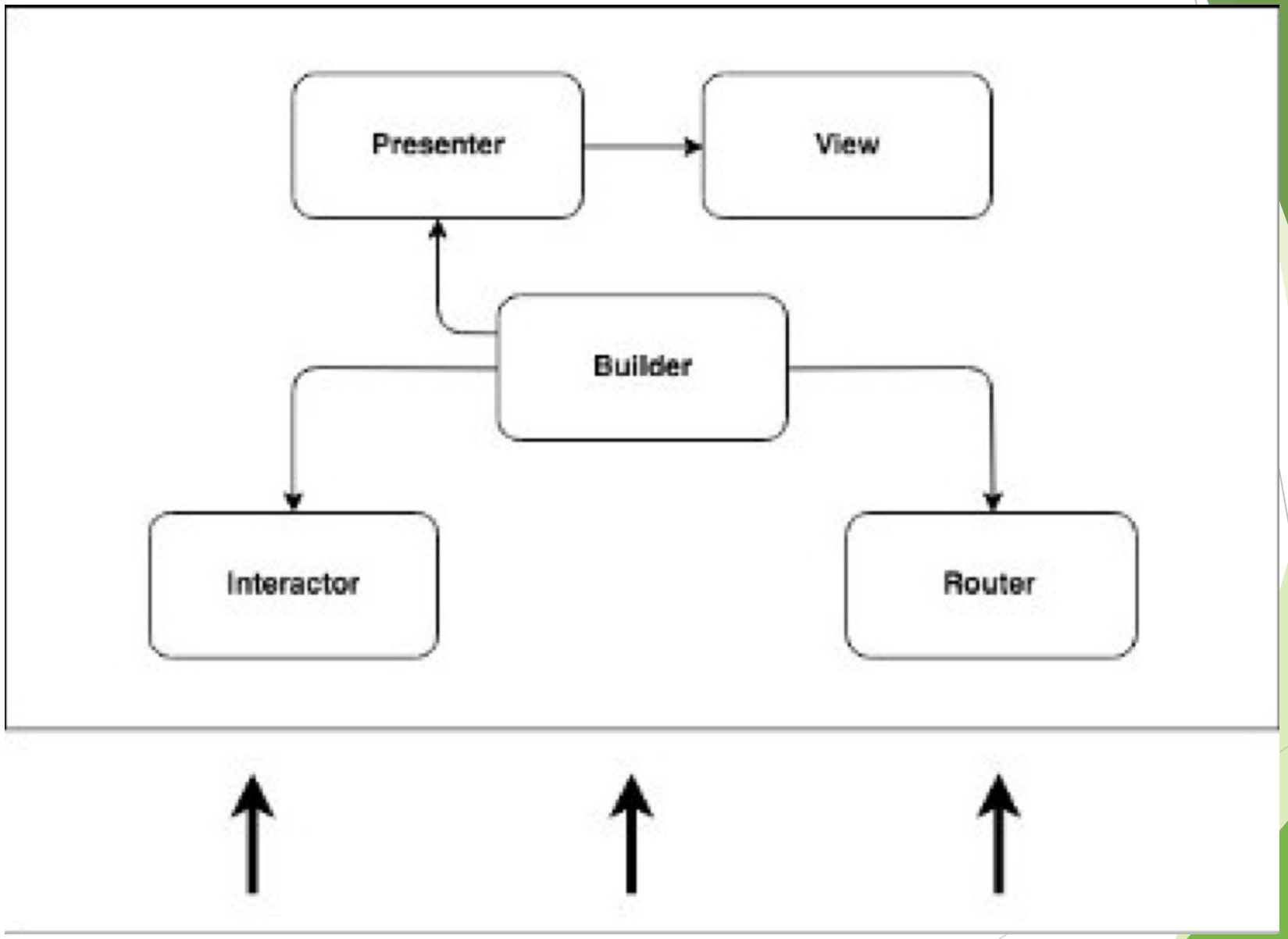
RIBs



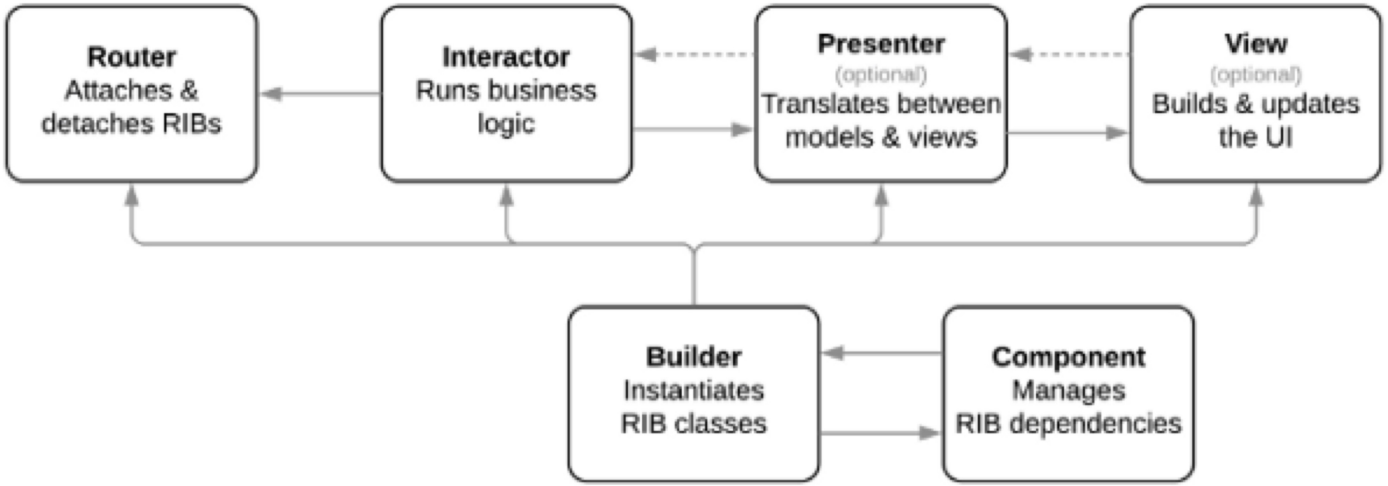
Step 1 Step 2 Step 3 Step 4 Step 5

User is signed out. Onboarding RIB is attached.

RIBs



RIBs


Router
Attaches & detaches RIBs
Interactor
Runs business logic
Presenter
(optional)
Translates between models & views
View
(optional)
Builds & updates the UI
Builder
Instantiates RIB classes
Component
Manages RIB dependencies

RIBs

- A arquitetura RIBs ajuda a organizar o código de forma mais modular e testável.
- Cada RIB é um módulo independente que pode ser testado isoladamente. Além disso, a separação de responsabilidades entre o Router, Interactor, Builder, View e Presenter torna o código mais limpo e fácil de entender.

RIBs - Vantagens

- Modularidade: O RIBs organiza o código em módulos independentes (RIBs), o que facilita a reutilização de código e a colaboração entre diferentes equipes de desenvolvedores.
- Testabilidade: Cada componente do RIBs (Router, Interactor, Builder, View, Presenter) pode ser testado isoladamente, o que facilita a escrita de testes unitários e ajuda a garantir a qualidade do código.
- Separação de Responsabilidades: O RIBs segue o princípio da separação de responsabilidades, o que torna o código mais limpo, organizado e fácil de entender.
- Escalabilidade: A arquitetura RIBs é projetada para ser escalável, o que a torna adequada para aplicativos grandes e complexos.
- Navegação: O RIBs oferece uma maneira estruturada de lidar com a navegação entre as telas, o que ajuda a gerenciar a complexidade da navegação em aplicativos grandes.

RIBs - Desvantagens

- Curva de Aprendizado: O RIBs tem uma curva de aprendizado mais íngreme em comparação com outras arquiteturas, como MVP ou MVVM. Os desenvolvedores precisam entender os conceitos de Router, Interactor, Builder, além de View e Presenter.

- Complexidade: A arquitetura RIBs pode ser excessivamente complexa para aplicativos pequenos ou simples. Pode ser mais eficiente usar uma arquitetura mais simples, como MVP ou MVVM, para aplicativos menores.

- Verbosidade: O RIBs requer a criação de várias classes e interfaces para cada tela do aplicativo, o que pode levar a mais código e maior complexidade.

- Menos Recursos de Aprendizado: Existem menos recursos de aprendizado disponíveis para o RIBs em comparação com arquiteturas mais populares, como MVP ou MVVM. Isso pode tornar mais difícil para os desenvolvedores novos no RIBs aprenderem e começarem a usar o framework.

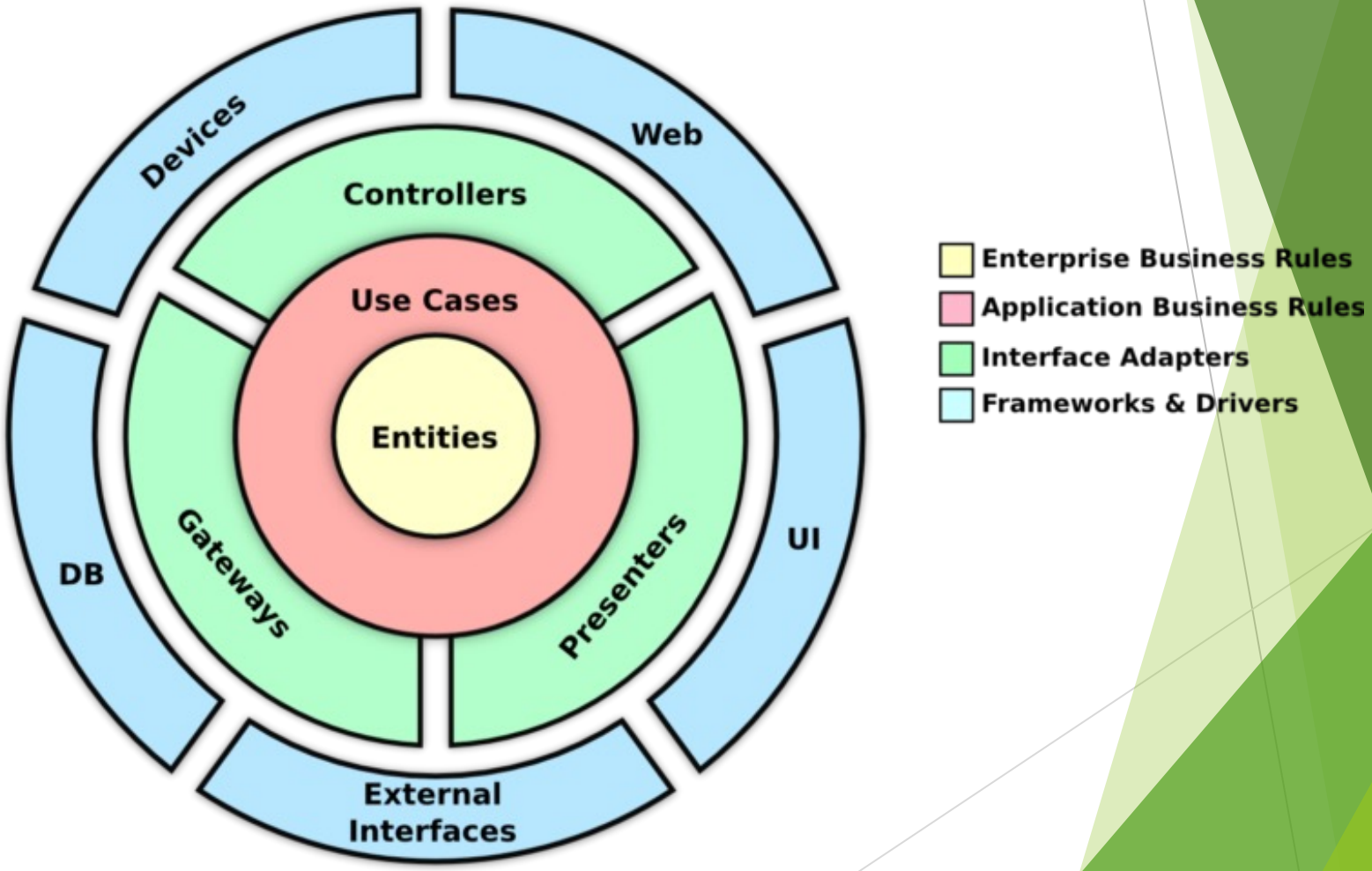
Arquitetura Clean

- A Arquitetura Clean, também conhecida como Arquitetura Hexagonal, é um padrão de design de software que visa separar as preocupações do código em camadas, tornando o sistema mais organizado, flexível e testável.
- Foi proposta por Robert C. Martin (Uncle Bob) e é baseada em alguns princípios do SOLID.

Arquitetura Clean

- Camada de Domínio: Esta é a camada mais interna e contém as regras de negócio do aplicativo. É completamente independente de qualquer outra camada. Esta camada contém Entidades e Casos de Uso.
  - Entidades: São os objetos de domínio que contêm as regras de negócio que são fundamentais para o aplicativo.
  - Casos de Uso: Contêm as regras de negócio específicas do aplicativo. Eles orquestram o fluxo de dados entre as Entidades e a Camada de Interface.
- Camada de Interface: Esta camada contém código que é necessário para interagir com o sistema externo, como banco de dados, web services, etc. Ela é dividida em Adaptadores e Portas.
  - Adaptadores: São implementações específicas para interagir com o sistema externo, como um banco de dados ou uma API web.
  - Portas: São interfaces que definem os métodos que os Adaptadores devem implementar.
- Camada de Apresentação: Esta é a camada mais externa e contém código relacionado à interface do usuário. Ela é responsável por receber as entradas do usuário e apresentar os resultados ao usuário.

Arquitetura Clean



Arquitetura Clean - vantagens

- Separação de Responsabilidades: Cada camada tem uma responsabilidade claramente definida, o que torna o código mais organizado e fácil de entender.
- Testabilidade: A separação de responsabilidades facilita a escrita de testes unitários para cada componente do sistema.
- Flexibilidade: A arquitetura permite a substituição de componentes externos, como banco de dados ou APIs, sem afetar o código de domínio.
- Independência de Framework: O código de domínio não depende de nenhum framework específico, o que torna mais fácil mudar para um framework diferente, se necessário.

Arquitetura Clean - desvantagens

- Complexidade: A arquitetura pode ser excessivamente complexa para aplicativos pequenos e simples.
- Curva de Aprendizado: Pode haver uma curva de aprendizado íngreme para desenvolvedores que são novos na Arquitetura Clean.
- Verbosidade: A necessidade de criar várias interfaces e implementações pode levar a um código mais extenso.

Outras Arquiteturas

- MVI
- VIP
- MVVM-C

Aula 05 - EAD

- Faça o CodeLab: https://developer.android.com/codelabs/android-hilt?hl=pt-br#0
- Pesquisar:
  - Design Patterns Comuns
    - Singleton, Factory, Builder, Prototype, Adapter, Decorator, Facade, Observer, Command, Strategy, State, Template Method
  - Design Patterns em Android
    - ViewHolder, Singleton, Factory, Observer, Strategy, Command
  - Design Patterns em iOS
    - Delegate, Singleton, Factory, Observer, Strategy, Command

Projeto Final

- Criar um aplicativo (iOS, Android, Hibrido...) que tenha:
  - As diretrizes básicas do Clean Code
  - Utilização de uma arquitetura (MVC, MVVM, VIPER...)
  - Uso de Dependency Injection
  - Uso de TDD
  - Uso de design patterns
  - Tenha pelo menos 3 telas.
  - Entrega um vídeo de 5 minutos ou um relatório escrito explicando onde está cada requisito.
  - Pode ser feito em grupos de até 3 pessoas.
  - Entrega até dia 16/10/2023