Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Build-block Architecture: Uma Abordagem para Navegação Modular para Desenvolvimento Nativo para iOS

Felipe Israel de Oliveira Vidal

JUIZ DE FORA
MARÇO, 2025

Build-block Architecture: Uma Abordagem para Navegação Modular para Desenvolvimento Nativo para iOS

Felipe Israel de Oliveira Vidal

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Marco Antônio Pereira Araújo

JUIZ DE FORA

MARÇO, 2025

Build-block Architecture: Uma Abordagem para Navegação Modular para Desenvolvimento Nativo para iOS

Felipe Israel de Oliveira Vidal

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Marco Antônio Pereira Araújo
Doutor em Engenharia de Sistemas e Computação.

Luiz Maurílio da Silva Maciel
Doutor em Programa de Engenharia de Sistemas e Computação

Ronney Moreira de Castro
Doutor em Informática

JUIZ DE FORA
13 DE MARÇO, 2025

Aos meus amigos.

Aos meus pais, pelo apoio e sustento.

Resumo

Este trabalho apresenta a *Build-block Architecture*, uma abordagem modular para o desenvolvimento de aplicativos nativos em iOS utilizando SwiftUI e conceitos da arquitetura *Model-View-ViewModel* (MVVM) com o padrão *Coordinator*. A modularização é alcançada por meio do *Swift Package Manager*, onde cada módulo contém suas próprias *Views*, *Models* e *View Models*, sendo gerenciado por um *coordinator* responsável pela navegação. A proposta facilita o gerenciamento de múltiplos fluxos de navegação, tanto no contexto *push* quanto *modal*, oferecendo flexibilidade e escalabilidade para aplicativos complexos. O *Follows Audit*, um aplicativo real desenvolvido com essa arquitetura, é utilizado como estudo de caso para demonstrar as vantagens da organização modular e a eficácia na separação de responsabilidades entre os componentes. O estudo visa fornecer uma alternativa eficiente para arquiteturas mais tradicionais, como *Model-View-Controller* (MVC) e MVVM, adaptada ao desenvolvimento moderno com *SwiftUI*.

Palavras-chave: modularização, programação reativa, navegação, testabilidade e escalabilidade de software

Abstract

This work presents the "Build-block Architecture" a modular approach to developing native iOS applications using SwiftUI and concepts from the Model-View-ViewModel (MVVM) and Coordinator architectures. Modularity is achieved through Swift Package Manager, where each module contains its own Views, Models, and ViewModels, managed by a Coordinator responsible for navigation. This approach facilitates the management of multiple navigation flows, both in push and modal contexts, providing flexibility and scalability for complex applications. Follows Audit, a real-world application developed with this architecture, is used as a case study to demonstrate the advantages of modular organization and the effectiveness of separating responsibilities among components. The study aims to offer an efficient alternative to traditional architectures, such as MVC and MVVM, tailored for modern development with SwiftUI.

***Keywords*:** modularization, reactive programming, navigation, testability and software scalability

Agradecimentos

A todos os meus pais, por todo o apoio e encorajamento de sempre.

Ao professor Marco pela orientação, amizade e principalmente, pela paciência, sem a qual este trabalho não se realizaria.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos e aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

"A única maneira de fazer algo excelente é amar o que você faz. Se você ainda não a encontrou, continue procurando. Não se acomode".

Steve Jobs

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Abreviações

BbA	*Build-block Architecture*
CD	*Continuous Deployment*
CI	*Continuous Integration*
FA	*Follows Audit*
UI	*User Interface* ou Interface do Usuário
MVC	*Model-View-Controller*
MVVM	*Model-View-ViewModel*
MVVM-C	*Model-View-ViewModel-Coordinator*
SO	Sistema Operacional
SDK	*Software Development Kit*
SPM	*Swift Package Manager*
WWDC	*Worldwide Developer Conference*

1 Introdução

O desenvolvimento de aplicativos para dispositivos móveis se tornou um campo de grande complexidade, demandando abordagens e arquiteturas que conciliem flexibilidade, modularidade e escalabilidade. No contexto de desenvolvimento para iOS, *frameworks* como *SwiftUI* (APPLE, 2019d) e arquiteturas como *Model-View-ViewModel* (MVVM) e *Coordinator* (FUKSA; SPETH; BECKER, 2025) são amplamente utilizadas para organizar a lógica de interface e de dados. Com a introdução do *SwiftUI*, em 2019, a *Apple* trouxe uma abordagem declarativa para a construção de interfaces, promovendo a simplificação de *layout* e interatividade. No entanto, quando o aplicativo cresce em funcionalidades e fluxos de navegação, surgem desafios que requerem uma abordagem arquitetural mais robusta e modular.

Para lidar com esses desafios, foi proposta a ***Build-block Architecture*** (BbA), uma solução modular para o desenvolvimento de aplicativos iOS, que utiliza o *Swift Package Manager* (SPM) (APPLE, 2024b) para organizar o código em módulos independentes, cada um com suas próprias *Views*, *Models*, *View Models* ou quaisquer outros arquivos necessários para seu funcionamento. Inspirada na ideia de blocos de construção *(Lego)*, essa arquitetura permite que o aplicativo seja montado de forma flexível, conforme as necessidades do projeto. O padrão *Coordinator* é utilizado para gerenciar a navegação entre telas, criando uma separação clara de responsabilidades e garantindo a manutenção da escalabilidade do projeto.

Este trabalho explora a aplicação prática da *Build-block Architecture* por meio do *Follows Audit*, um aplicativo real desenvolvido com essa abordagem. O *Follows Audit* permite ao usuário analisar dados de interações do Instagram, mantendo a privacidade e a segurança dos dados no dispositivo. A arquitetura modular foi essencial para lidar com diferentes funcionalidades, mantendo o código organizado e fácil de manter. O estudo deste aplicativo serve como base para demonstrar as vantagens dessa nova arquitetura.

1.1 Apresentação do Tema

O rápido avanço no desenvolvimento de aplicativos móveis exige abordagens arquiteturais que não apenas favoreçam a modularidade, mas também garantam a escalabilidade, flexibilidade e a testabilidade. A *Build-block Architecture* é uma resposta a essas demandas, propondo uma abordagem em que o aplicativo é montado de forma modular, como blocos de lego, utilizando o *Swift Package Manager*. Cada módulo pode conter suas próprias *views*, *models* e *view models*, permitindo ao desenvolvedor isolar diferentes partes do aplicativo de forma eficiente. Esse tipo de organização oferece maior controle sobre a complexidade do projeto e facilita a manutenção e expansão do aplicativo.

Um dos pontos principais dessa arquitetura é a implementação do *Coordinator*, que gerencia a navegação entre os diferentes módulos do aplicativo. O *Coordinator* permite uma separação clara de responsabilidades, facilitando a navegação no contexto *push* – onde o usuário transita entre telas dentro do mesmo fluxo; e no contexto *modal* – onde um novo contexto é criado e o fluxo anterior é colocado em espera.

Dessa forma, a *Build-block Architecture* propõe um novo paradigma para o desenvolvimento de aplicativos nativos, enfatizando a importância da modularização, da testabilidade e da escalabilidade, aspectos essenciais para o sucesso em um mercado em constante evolução.

1.2 Contextualização

O *SwiftUI*, lançado pela *Apple* em 2019, como uma nova forma de desenvolver interfaces declarativas, trouxe uma série de benefícios para a criação de *layouts* e a interação entre componentes. No entanto, com a simplificação das interfaces, surgem desafios relacionados à organização de projetos mais complexos, especialmente quando se trata de navegação e modularização. Este trabalho apresenta a *Build-block Architecture*, uma abordagem que utiliza a modularização como foco e a organização de fluxos de navegação por meio do padrão *Coordinator*. Inspirada no conceito de blocos de lego, essa arquitetura permite que o desenvolvedor monte o aplicativo como uma combinação de módulos independentes, cada um com suas próprias *views*, *models* e *view models*, criando uma arquitetura flexível

e escalável.

O *Follows Audit*, um aplicativo real desenvolvido com *SwiftUI* e utilizando a *Build-block Architecture*, é utilizado como estudo de caso neste trabalho. Sua principal função é permitir que o usuário analise dados de interações no Instagram de forma privada e local, com uma estrutura modular que facilita a implementação de novos recursos e a navegação em diferentes fluxos de telas. O *Follows Audit* é o exemplo prático que foi usado para demonstrar os benefícios dessa arquitetura no desenvolvimento de aplicativos nativos.

1.3 Descrição do Problema

A construção de aplicativos móveis complexos requer uma arquitetura que permita fácil manutenção e evolução. Abordagens tradicionais, como *Model-View-Controller* (MVC), podem se tornar insuficientes à medida que a aplicação se expande, resultando em dificuldades para manter a separação clara de responsabilidades e assegurar a testabilidade. Além disso, gerenciar a navegação em *SwiftUI*, especialmente em aplicativos que exigem múltiplos fluxos e contextos, pode aumentar a complexidade se não houver uma estrutura adequada para controlar a transição eficiente entre diferentes telas.

O problema central que este trabalho busca abordar é: **como organizar um aplicativo modular em *SwiftUI*, utilizando uma arquitetura que permita navegação eficiente, testabilidade e escalabilidade, mantendo a simplicidade e flexibilidade?**

1.4 Pergunta Orientadora/Norteadora

Após apresentado o tema deste estudo, bem como os problemas das arquiteturas tradicionais, a pergunta que fica é: como a adoção da *Build-block Architecture*, utilizando *SwiftUI*, a arquitetura MVVM e o padrão *Coordinator*, impacta a modularização, navegação e escalabilidade no desenvolvimento de aplicativos nativos para iOS, em comparação com outras arquiteturas tradicionais, como o MVC?

1.5 Justificativa e Motivações

A motivação para explorar e desenvolver uma nova arquitetura vem da necessidade de encontrar uma solução que resolva as limitações de arquiteturas tradicionais no desenvolvimento moderno de aplicativos iOS. O *SwiftUI*, com sua simplicidade declarativa, oferece oportunidades para repensar as arquiteturas clássicas, como MVC, em favor de uma abordagem mais modular e eficiente.

O uso do *Swift Package Manager* para criar módulos independentes, juntamente com o padrão *Coordinator* para gerenciar fluxos de navegação, possibilita uma abordagem escalável e flexível. O *Follows Audit* é um exemplo prático de como essa arquitetura pode ser aplicada em um projeto real, o que reforça a relevância e o valor prático deste estudo.

1.6 Objetivos

O objetivo geral deste trabalho é apresentar uma proposta de uma nova abordagem como uma alternativa moderna e eficiente para a modularização e navegação de aplicativos nativos em iOS, utilizando *SwiftUI* e a arquitetura MVVM-C. A seguir estão alguns objetivos específicos que serão apresentados:

- Demonstrar como a modularização pode ser alcançada utilizando *Swift Package Manager*;
- Analisar os benefícios da separação de responsabilidades entre *Views*, *Models* e *Models*;
- Explorar a utilização do *Coordinator* para navegação flexível entre contextos *push* e *modal*;
- Aplicar a arquitetura ao *Follows Audit* como estudo de caso, evidenciando as vantagens práticas na escalabilidade, manutenção e organização do código.

1.7 Organização do Trabalho

Os capítulos que compõem este trabalho estão dispostos de modo a fornecer uma visão detalhada a respeito da construção e aplicação da nova abordagem, focando-se na modularização e navegação de aplicativos iOS desenvolvidos em *Swift*, utilizando o *framework SwiftUI* e o suporte do *Swift Package Manager*.

Durante a **Introdução**, é apresentada a problemática abordada neste trabalho, justificando e contextualizando a escolha da *Build-block Architecture* como uma solução. Também são descritos os objetivos da pesquisa e a motivação por trás da escolha desta arquitetura, bem como uma visão geral da organização dos capítulos do documento.

A **Fundamentação Teórica** apresenta os conceitos básicos sobre arquiteturas de software para iOS como MVC, MVVM, VIPER, o padrão *Coordinator*; e a abordagem declarativa do *SwiftUI*, bem como a navegação entre telas. Além disso, é dada uma atenção à modularização e sua aplicação na prática em projetos iOS, utilizando o SPM.

O **Referencial Teórico** mostra o mapeamento sistemático da literatura. Neste capítulo, são apresentadas as palavras-chave, o modelo PICOC, a questão orientadora da pesquisa, a *string* de busca utilizada e os critérios de inclusão e exclusão para seleção dos artigos e pesquisas pertinentes.

Em seguida, a ***Build-block Architecture*** é descrita como o conceito central deste trabalho. O capítulo confere os princípios da arquitetura proposta, focando-se na modularização através de blocos, e como são organizados os módulos do SPM. Também são discutidas as vantagens desta abordagem, no que se refere à escalabilidade, flexibilidade, testabilidade e manutenção, em comparação às arquiteturas tradicionais.

O Capítulo 5, **Estudo de Caso: *Follows Audit***, segue com a aplicação prática da *Build-block Architecture* em um aplicativo real. Este capítulo aborda o aplicativo, seus recursos, arquitetura e *design*; e também apresenta a implementação, modularização e o uso do *Coordinator* para a navegação.

O Capítulo 6, **Discussão e Resultados**, traz a avaliação da aplicação da *Build-block Architecture* ao *Follows Audit*: a flexibilidade para modelos complexos, os benefícios para teste e manutenção a longo prazo, bem como o impacto do desenvolvimento em comparação com a abordagem tradicional.

Por fim, conclui-se o trabalho, revisitando as contribuições da *Build-block Architecture* e sugerindo as direções futuras da investigação e do desenvolvimento.

2 Fundamentação Teórica

Este capítulo tem como objetivo apresentar os conceitos e princípios fundamentais que sustentam este trabalho. Nele, são discutidas as abordagens e tecnologias relevantes para o desenvolvimento de aplicativos iOS, fornecendo um embasamento sólido para a solução proposta.

2.1 Arquiteturas de Software em iOS

No desenvolvimento de aplicativos móveis, a escolha da arquitetura de software desempenha um papel fundamental, não apenas na organização do código, mas também em sua capacidade de ser mantido e escalado ao longo do tempo. No ecossistema iOS, várias arquiteturas surgiram para abordar os desafios de separação de responsabilidades, modularidade e testabilidade. Entre as mais utilizadas estão *Model-View-Controller* (MVC) (DOBREAN; DIOSAN, 2019), *Model-View-ViewModel* (MVVM) (FUKSA; SPETH; BECKER, 2025), *Model-View-ViewModel-Coordinator* (MVVM-C) (FUKSA; SPETH; BECKER, 2025), VIPER (BABAYEV, 2023), dentre outras. Cada uma dessas abordagens oferece soluções específicas para o controle da interação entre a interface do usuário e a lógica de negócios, bem como para a gestão de fluxos de navegação.

É importante lembrar que a escolha para a arquitetura de um software depende de vários fatores que precisam ser definidos pela equipe de desenvolvimento. Não existe uma arquitetura certa, mas sim uma ideal para o projeto específico.

2.1.1 *Model-View-Controller* (MVC)

Em 2008, quando a *Apple* publicou a primeira versão do seu *Software Development Kit* (SDK) (APPLE, 2008a) para o iPhone, foi recomendado que os desenvolvedores seguissem um documento chamado *Cocoa Fundamentals Guide* (APPLE, 2013). Nesse guia, uma das recomendações era seguir as práticas da arquitetura MVC para a criação dos aplicativos. A arquitetura era descrita como "fundamental para um bom *design* para um aplicativo

Cocoa“. Com o avanço do desenvolvimento móvel, novas abordagens foram surgindo ao longo do tempo, resultando nas arquiteturas que conhecemos hoje. Atualmente, a *Apple* não recomenda mais uma arquitetura padrão.

A premissa básica do MVC é a separação em três componentes principais: *model*, *view* e *controller*.

Na Figura 2.1, está ilustrada a arquitetura MVC bem como seus principais componentes: o ***Model***, que é a camada responsável pelos dados da aplicação e pela lógica de negócios. Ela é independente da interface gráfica, sendo capaz de realizar operações de leitura, escrita e manipulação dos dados; a ***View***, que é a camada de apresentação, responsável pela interface gráfica e por exibir os dados ao usuário. Ela escuta as mudanças no *model* e atualiza a exibição conforme necessário; por sua vez, o ***Controller*** atua como intermediário entre a *view* e o *model*, recebendo as interações do usuário e atualizando o *model* ou a *view* conforme apropriado.

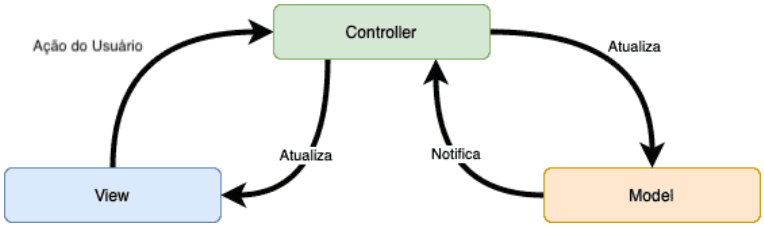

Figura 2.1: Arquitetura MVC

Apesar de ser uma abordagem simples e eficaz para aplicativos pequenos, o MVC pode se tornar problemático à medida que o projeto cresce. O *controller*, em especial, tende a acumular muitas responsabilidades, levando ao chamado *“Massive View Controller”*, que dificulta a manutenção e escalabilidade.

2.1.2 *Model-View-ViewModel* (MVVM)

O *Model-View-ViewModel* (MVVM) surgiu como uma evolução do MVC, visando reduzir a sobrecarga do *Controller* introduzindo o *View Model*. Como disse Matteo Manferdini (MANFERDINI, 2023) em seu artigo, a *Apple* não possui uma arquitetura padrão, no entanto, o *SwiftUI* é particularmente feito para a arquitetura MVVM – graças à sua abordagem declarativa e reativa.

Na Figura 2.2, são apresentados os componentes do MVVM: o ***Model***, com a

mesma responsabilidade do MVC, lidando com a lógica de negócios e os dados; a ***View***, similar à MVC, é a interface gráfica que interage com o usuário; e a ***ViewModel***, que é o ponto chave dessa arquitetura, atuando como uma "ponte" entre o *Model* e a *View*. Ela é a responsável por converter os dados do *Model* em um formato que a *View* pode apresentar e, ao mesmo tempo, interpreta as ações do usuário para atualizar o *Model*.

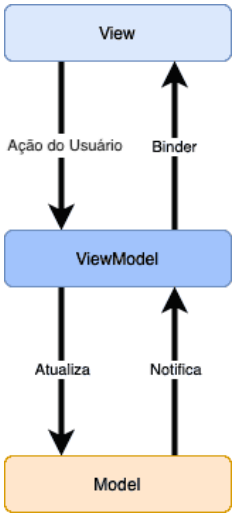

Figura 2.2: Arquitetura MVVM

O MVVM é especialmente poderoso em cenários que envolvem ligação de dados *(data binding)*, permitindo que a interface seja automaticamente atualizada quando o estado do *ViewModel* é alterado, sem que seja necessário gerenciar manualmente as mudanças na UI.

No artigo *"MVVM: An architectural coding pattern to structure SwiftUI Views"*, Avander Lee 2024 detalha como o padrão MVVM pode ser aplicado para estruturar as *views* de forma eficiente, promovendo uma separação clara entre a lógica e as regras de negócio e a interface do usuário. A seguir, apresenta-se um exemplo de como a implementação dessa arquitetura pode facilitar a escalabilidade e promover uma estrutura altamente testável e reutilizável em diversos contextos.

Inicialmente, será exibido um código com um alto acoplamento e uma sobrecarga de responsabilidades, ilustrando os problemas que emergem do MVVM. Em seguida, será apresentada uma refatoração desse código, adotando os padrões do MVVM de forma adequada, com o objetivo de evidenciar os benefícios e as melhorias obtidas em termos de organização, manutenção e flexibilidade do sistema.

No Código 2.1 é possível perceber que a *ContactView* depende fortemente do modelo, além de possuir regras de negócios, como a exclusão do contato.

```
struct Contact {
    let name: String
}
struct ContactView: View {
    let contact: Contact
    var body: some View {
        VStack {
            Text("Name: \(contact.name)")
            Button("Delete", action: deleteContact)
        }
    }

    func deleteContact() {}
}
```

Código 2.1: Implementacão da *ContactView* sem arquitetura

Para corrigir esses problemas, é implementada a *ContactViewModel* (Código 2.2), que passa a ser a responsável por fazer a ligação entre a *ContactView* e o *Contact*.

```
struct ContactViewModel {
    var name: String { contact.name }
    private let contact: Contact
    init(contact: Contact) {
        self.contact = contact
    }
    func deleteContact() {}
}
struct ContactView: View {
    let viewModel: ContactViewModel
    var body: some View {
        VStack {
            Text("Name: \(viewModel.name)")
            Button("Delete", action: viewModel.deleteContact)
        }
    }
```

```
17 }
```

Código 2.2: Implementacão da *ContactView* com MVVM

Apesar de muito utilizada, é comum encontrar diferentes implementações do MVVM, mas sempre seguindo os princípios de ser testável e reutilizável.

2.1.3 *Model-View-ViewModel-Coordinator* (MVVM-C)

O *Model-View-ViewModel-Coordinator* (MVVM-C) é uma extensão da arquitetura MVVM, que introduz o *Coordinator* para gerenciar a navegação entre as telas. No MVVM, a navegação pode se tornar complexa e acabar ficando dispersa no *view model*, prejudicando a clareza do código e misturando responsabilidades. Para resolver essa questão, o MVVM-C propõe que a navegação seja responsabilidade de um *coordinator*, o que mantém o *view model* focado em sua função principal de transformar dados em estados para a *view*.

De uma forma geral, o *coordinator* organiza a navegação do aplicativo, gerenciando a transição entre as telas. Sua responsabilidade é instanciar as *views* e suas *view models* e coordenar as interações entre diferentes fluxos de navegação.

As principais vantagens do *coordinator* são:

- Os fluxos são independentes. Ou seja, um fluxo pode ser iniciado a qualquer momento;
- Desacoplamento da *view*;
- Redução de tempo para refatorar fluxos;
- E um dos fatores mais importantes: testabilidade.

Dentre essas vantagens citadas acima, ressalta-se que o uso do *coordinator* permite a independência dos fluxos de navegação, possibilitando que um fluxo possa ser iniciado a qualquer momento, sem depender de outros componentes.

2.1.4 VIPER

O VIPER é uma arquitetura mais complexa e inspirada no princípio SOLID (RAMACHANDRAPPA, 2024), oferecendo uma separação de responsabilidades muito clara.

Cada camada dessa arquitetura tem um papel específico.

Na Figura 2.3, é apresentada a arquitetura VIPER, bem como seus principais componentes, sendo eles: a ***View***, responsável por exibir os dados e interagir com o usuário; o ***Interactor***, que contém a lógica de negócios do aplicativo, geralmente lidando com os dados fornecidos pelo *Entity*; o ***Presenter*** que atua como um intermediário entre a *view* e o *interactor* recebendo os eventos da *view* e solicitando ao *interactor* para realizar a lógica de negócios; a ***Entity*** que representa os objetos de negócio do aplicativo; e por fim, o ***Router*** que lida com a navegação e a transição entre diferentes telas.

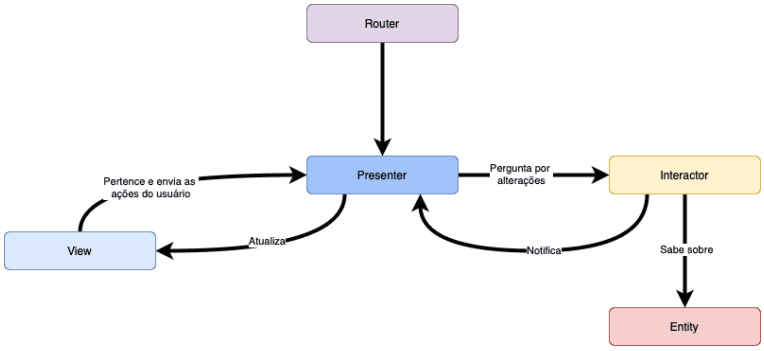

Figura 2.3: Arquitetura VIPER

2.1.5 Coordinator

O *Coordinator*, por sua vez, é um padrão que foca na separação da lógica de navegação de um aplicativo. É frequentemente usado em conjunto com arquiteturas como MVVM e VIPER, onde a complexidade da navegação entre telas pode ser um desafio. O *coordinator* tem a responsabilidade de gerenciá-las, criando e configurando *views* e *view models* quando necessário, e determinando qual tela deve ser exibida a seguir.

Ao isolar a navegação em um componente separado, o código relacionado às transições de tela é mais fácil de manter e escalar, permitindo que as *view models* se concentrem em suas funções principais.

2.1.6 Comparação entre as Abordagens

No desenvolvimento de aplicativos para iOS, a escolha da arquitetura impacta diretamente na organização, escalabilidade e manutenção do código. Diferentes abordagens foram

criadas para melhorar a separação de responsabilidades e reduzir o acoplamento, sendo as mais populares MVC, MVVM e VIPER. No entanto, cada uma apresenta vantagens e desafios, variando desde a simplicidade do MVC até a estruturação mais complexa do VIPER. A Tabela 2.1 resume as principais características dessas arquiteturas, destacando aspectos como modularização, reuso de código, acoplamento e facilidade de manutenção, auxiliando na escolha da abordagem mais adequada para cada projeto.

	MVC	**MVVM**	**VIPER**
Organização do Código	Baixa	Moderada	Alta
Acoplamento	Alto	Moderada	Baixo
Modularização	Baixa	Moderada	Alta
Complexidade	Baixa	Moderada	Alta
Escalabilidade	Baixa	Moderada	Alta
Tempo de Build	Alto	Alto	Moderado
Facilidade de Manutenção	Baixa	Moderada	Alta
Flexibilidade na Navegação	Baixa	Moderada	Alta

Tabela 2.1: Comparação entre padrões de arquitetura: MVC, MVVM e VIPER

A comparação entre as arquiteturas tradicionais revela que não existe uma solução única para todos os projetos. O MVC, apesar de simples, não escala bem em aplicações grandes devido ao forte acoplamento. O MVVM melhora a separação de responsabilidades, mas pode resultar em *view models* sobrecarregadas se não for bem estruturado. O VIPER, por outro lado, se destaca na modularização e organização do código, mas adiciona uma complexidade significativa com muitas camadas e arquivos. Dessa forma, a escolha da arquitetura deve considerar o equilíbrio entre organização, modularização e simplicidade, garantindo que a aplicação seja escalável sem comprometer a manutenção e a produtividade da equipe.

A escolha dos critérios utilizados na Tabela 2.1 foi baseada em desafios comuns encontrados no desenvolvimento de aplicações móveis, especialmente no que se refere à organização do código, manutenção e escalabilidade do projeto. Esses critérios foram definidos com base em experiências práticas e na literatura sobre arquitetura de software, visando identificar as principais diferenças entre os padrões MVC, MVVM e VIPER.

2.2 *SwiftUI* e a Abordagem Declarativa

2.2.1 Introdução ao *SwiftUI*

O *SwiftUI* (APPLE, 2019d) é um *framework* de desenvolvimento de interfaces de usuário lançado pela *Apple* em 2019, com o objetivo de simplificar e acelerar o processo de criação de aplicativos. Ao contrário do *UIKit*, que adota uma abordagem imperativa, o *SwiftUI* utiliza uma abordagem declarativa, onde é possível descrever o que a interface deve exibir em resposta a mudanças de estado (APPLE, 2008b). Esse modelo permite que o código seja mais legível, conciso e de fácil manutenção.

No *SwiftUI*, é possível construir interfaces com base em uma estrutura hierárquica de *views*, onde cada *view* é uma função do estado. À medida que o estado de um componente muda, o *SwiftUI* reage automaticamente, atualizando a interface para refletir essas mudanças. Isso reduz a complexidade associada ao gerenciamento de estados e eventos, comparado ao modelo tradicional de *delegates* e *callbacks*.

Uma das principais vantagens do *SwiftUI* é ser multiplataforma. Ou seja, com uma única base de código, é possível desenvolver interfaces que funcionam em diferentes dispositivos do ecossistema *Apple*, incluindo iOS, iPadOS, macOS, watchOS, tvOS e o mais recente, o visionOS. Isso elimina a necessidade de adaptar o código para cada sistema operacional, facilitando a criação de aplicativos consistentes e integrados.

O *framework* é fortemente integrado ao *Swift* (APPLE, 2014), a linguagem de programação atual da *Apple*, e aproveita seus recursos avançados para fornecer uma experiência de desenvolvimento fluida e intuitiva. O uso de estruturas como ***@State*** e ***@Binding*** permite que o *SwiftUI* gerencie automaticamente atualizações na interface quando o estado muda, eliminando a complexidade de gerenciar manualmente eventos de interface.

Embora seja uma ferramenta poderosa, o *SwiftUI* ainda enfrenta alguns desafios, especialmente no que diz respeito à navegação complexa e à integração com o *UIKit*, que muitas vezes é necessária para utilizar componentes que ainda não estão totalmente implementados no *SwiftUI*. Conforme o *framework* evolui, sua adoção e suporte continuam a crescer, consolidando o *SwiftUI* como uma peça central no desenvolvimento de aplicativos

nativos para o ecossistema *Apple*.

Atualmente já é possível criar aplicativos completos utilizando apenas *SwiftUI* e isso será demonstrado nos capítulos a seguir, com o *Follows Audit*.

2.2.2 Diferenças entre *SwiftUI* e *UIKit*

SwiftUI e *UIKit* são dois *frameworks* distintos para a criação de interfaces gráficas no desenvolvimento de aplicativos para o ecossistema da *Apple*. Enquanto o *UIKit* tem sido o padrão desde sua introdução em 2008, o *SwiftUI*, lançado em 2019, representa uma abordagem mais moderna e declarativa para a construção de interfaces.

2.2.2.1 Paradigma Declarativo vs. Imperativo

O **paradigma declarativo** do *SwiftUI* é uma abordagem de desenvolvimento de interfaces de usuário (UI) onde os desenvolvedores descrevem como a interface deve ser renderizada com base em um estado e nas ações do usuário.

O Código 2.3 implementa, utilizando o *SwiftUI*, uma tela em branco com um botão e um texto. Ao pressionar esse botão, o valor do texto é alterado.

```
1  import SwiftUI
2  struct ContentView: View {
3      @State private var text = "Texto inicial"
4      var body: some View {
5          VStack {
6              Text(text)
7                  .padding()
8
9              Button("Mudar Texto") {
10                 text = "Texto alterado"
11             }
12         }
13         .padding()
14     }
15 }
```

Código 2.3: Código *SwiftUI*

Por outro lado, o *UIKit* (Código 2.4) segue um **paradigma imperativo**, onde o desenvolvedor precisa especificar manualmente cada alteração na interface, ou seja, deve controlar quando e como a UI é atualizada.

```
import UIKit
class ViewController: UIViewController {
    let label = UILabel()
    let button = UIButton(type: .system)
    override func viewDidLoad() {
        super.viewDidLoad()

        label.text = "Texto inicial"
        label.textAlignment = .center
        label.translatesAutoresizingMaskIntoConstraints = false
        view.addSubview(label)
        button.setTitle("Mudar Texto", for: .normal)
        button.addTarget(self, action: #selector(buttonTapped), for: .
    touchUpInside)
        button.translatesAutoresizingMaskIntoConstraints = false
        view.addSubview(button)

        NSLayoutConstraint.activate([
            label.centerXAnchor.constraint(equalTo: view.centerXAnchor),
            label.centerYAnchor.constraint(equalTo: view.centerYAnchor),

            button.centerXAnchor.constraint(equalTo: view.centerXAnchor)
    ,
            button.topAnchor.constraint(equalTo: label.bottomAnchor,
    constant: 20)
        ])
    }
    @objc func buttonTapped() {
        label.text = "Texto alterado"
    }
}
```

Código 2.4: Código *UIKit*

A Figura 2.4 é o resultado da tela desenvolvida utilizando os dois *frameworks*, tanto o *SwiftUI*, quanto o *UIKit*. Para uma tela simples como essa, é possível concluir que, de fato, escrever interfaces utilizando o *SwiftUI* requer muito menos código do que utilizando o *UIKit*.

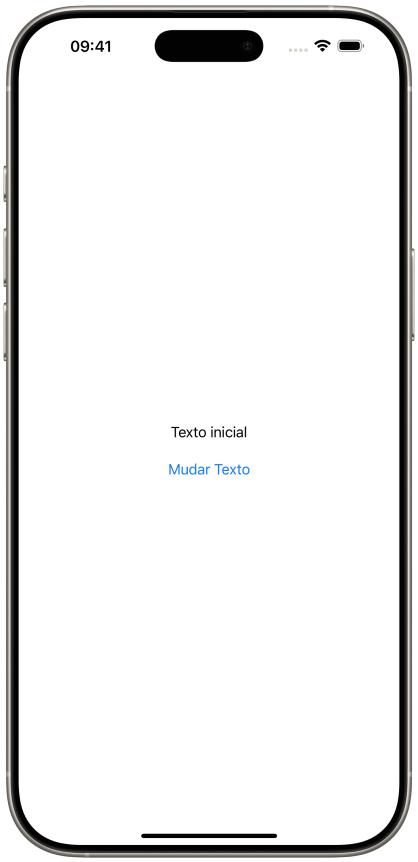

Figura 2.4: Resultado para os Códigos 2.3 e 2.4

2.2.2.2 Gerenciamento de Estados

No *SwiftUI*, o gerenciamento de estados é feito utilizando propriedades como ***@State*** e ***@Binding***, permitindo uma ligação reativa entre o estado da aplicação e a interface (APPLE, 2019b). Ou seja, quando o estado muda, a interface precisa ser alterada.

A propriedade ***@State*** armazena um valor na hierarquia da *view* (APPLE, 2019c). Quando o valor é atualizado, o *SwiftUI* se encarrega de atualizar apenas a parte da hierarquia que depende dele.

O Código 2.5 implementa um botão que incrementa o valor da variável *contador* e atualiza a tela para mostrar o valor atual.

```
1 struct PlayButton: View {
```

```
    @State private var contador: Int = 0
    var body: some View {
        Button("Tap me \(contador)") {
            contador += 1
        }
    }
}
```

Código 2.5: *SwiftUI – State*

A propriedade ***@Binding*** cria uma conexão bidirecional entre a propriedade que armazena o estado e uma *view* (APPLE, 2019a). É especialmente útil quando é preciso alterar o valor de um estado em outra *view*.

Quando a *ContentView* (Código 2.6) instancia o *PlayButton*, ele inicia a *view* passando o *binding* da propriedade de estado.

```
struct PlayButton: View {
    @Binding var contador: Int
    var body: some View {
        Button("Tap me \(contador)") {
            contador += 1
        }
    }
}
struct ContentView: View {
    @State private var contador: Int = 0
    var body: some View {
        PlayButton(contador: $contador)
    }
}
```

Código 2.6: *SwiftUI – Binding*

2.2.2.3 Simplicidade e Quantidade de Código

Como visto nos códigos implementados acima, o *SwiftUI* reduz drasticamente a quantidade de código necessária para criar interfaces complexas. Um *layout* complexo em *UIKit* pode ser escrito em poucas linhas de código usando *SwiftUI*.

2.2.2.4 Compatibilidade

O *SwiftUI* foi projetado para ser multiplataforma, o que significa que a maior parte do código pode ser reutilizada entre os diferentes sistemas operacionais da *Apple*. Enquanto isso, o *UIKit* é específico para cada sistema operacional (SO), com adaptações para cada dispositivo. Isso resulta em uma maior personalização, mas exige mais esforço ao escrever código para diferentes dispositivos.

2.2.2.5 Curva de Aprendizado

O *SwiftUI* apresenta uma curva de aprendizado mais suave para iniciantes, devido à sua simplicidade e à abordagem declarativa. Enquanto isso, o *UIKit* possui uma curva de aprendizado mais acentuada, especialmente ao lidar com *constraints* de *layout* e o ciclo de vida dos *controllers*.

2.2.3 Desafios de Navegação em *SwiftUI*

A navegação em *SwiftUI*, embora moderna e intuitiva, apresenta alguns desafios e limitações quando comparada com *frameworks* mais tradicionais, como o *UIKit*. Com as versões mais recentes do *framework*, a navegação passou por uma significativa evolução com a introdução do **NavigationStack** (APPLE, 2022), que substitui o **NavigationView** para fluxos de navegação mais robustos e flexíveis. Essa mudança foi feita para melhorar o controle sobre a pilha de navegação e resolver algumas das limitações que existiam nas versões anteriores. Ainda assim, alguns desafios persistem, especialmente quando comparados ao *UIKit*. São eles:

- **Complexidade na Navegação entre Telas**

  A medida que os fluxos de navegação vão se tornando mais complexos, a combinação de modais (APPLE, 2023) e *stacks (push)* também se torna mais complexa.

- **Customização de Transições**

  Embora o *SwiftUI* forneça transições básicas entre telas, a customização delas ainda é limitada em comparação com o *UIKit*, que permite controle total sobre animações de navegação.

- **Navegação Programática**

  No *UIKit*, controlar a navegação programaticamente com *'pushViewController'* ou *'present'* é simples, mas em *SwiftUI*, especialmente antes da introdução do **NavigationStack**, isso era uma tarefa bem trabalhosa. Isso porque a *view* se baseia em estados para atualizar, então era algo como ter que *ligar* e *desligar* chaves para navegar por outras telas.

2.3 Modularização em Desenvolvimento de Aplicativos

A modularização tem se tornado uma prática essencial no desenvolvimento de aplicativos modernos, particularmente quando se trata de projetos complexos ou de grande escala.

A divisão de um projeto em módulos facilita a organização do código, promove a reutilização de componentes e torna o desenvolvimento mais escalável e fácil de manter.

No contexto do desenvolvimento de aplicativos nativos para iOS, essa prática tem ganhado cada vez mais relevância, especialmente com a introdução do *Swift Package Manager* (SPM), que facilita a criação e a integração de módulos.

2.3.1 Conceitos de Modularização

A modularização (BARTLETT, 2024) envolve uma série de princípios e padrões que ajudam a dividir o código em componentes reutilizáveis, coesos e com responsabilidades bem definidas.

2.3.1.1 Baixo Acoplamento e Alta Coesão

Um dos conceitos mais importantes da modularização é a busca pelo baixo acoplamento e alta coesão. Isso quer dizer que os módulos devem ser o mais independentes possível. Quando um módulo depende fortemente de outro, qualquer mudança em um pode impactar o outro, dificultando a manutenção e a evolução do projeto.

Além disso, cada módulo deve ter uma única responsabilidade ou um conjunto de responsabilidades **altamente relacionadas**. Isso facilita o entendimento do que o

módulo faz, tornando-o mais fácil de manter, testar e reutilizar.

2.3.1.2 Abstração e Encapsulamento

A modularização está intimamente ligada aos princípios de abstração e encapsulamento. A ideia é ocultar a complexidade interna de um módulo, expondo apenas uma interface pública que define como esse módulo pode ser usado. Por meio de interfaces ou protocolos, um módulo define claramente o que faz, sem expor os detalhes de como realiza suas operações.

O encapsulamento, por sua vez, prevê que funcionalidades internas de um módulo são mantidas privadas, o que garante que os métodos internos não sejam acessíveis ou manipulados diretamente por outros módulos.

2.3.1.3 Separação de Responsabilidades

A modularização permite aplicar o princípio da separação de responsabilidades, herdado do *Clean Architecture*, garantindo que diferentes responsabilidades do sistema sejam tratadas de forma isolada, melhorando a organização do código e tornando o app mais flexível e escalável (MARTIN, 2017).

2.3.1.4 Dependência Explícita

Uma prática comum na modularização é a declaração explícita de dependências. Isso pode ser feito através da injeção de dependências, onde as dependências de um módulo são fornecidas de fora, em vez de serem criadas internamente. Isso facilita o teste e a manutenção, pois as dependências podem ser trocadas facilmente, como em testes unitários, onde classes reais são substituídas por *mocks* ou *stubs*.

2.3.2 Swift Package Manager

O *Swift Package Manager* é uma ferramenta própria da *Apple* e já integrada à linguagem *Swift* que facilita o gerenciamento de dependências e a modularização de projetos (APPLE, 2024b). Essa ferramenta permite a divisão de um projeto em pacotes que podem ser reutilizados em diferentes projetos ou compartilhados com outros desenvolvedores.

Uma de suas principais vantagens em relação a concorrentes, como CocoaPods ou Carthage, outros dois gerenciadores de pacotes, é a integração nativa ao *Xcode*, o que facilita o uso dentro dos projetos. Além disso, o SPM é multiplataforma, ou seja, funciona para todos os SOs da *Apple*.

O arquivo *Package.swift* é o coração de qualquer pacote gerenciado pelo SPM. Esse arquivo define a configuração do pacote, especificando como o código está organizado e quais dependências o pacote possui, bem como outras especificações mais gerais.

1. Nome do Pacote

O nome do pacote é definido na inicialização do pacote (Código 2.7) e é o identificador principal dentro do SPM.

```
let package = Package(
    name: "NomeDoProjeto"
)
```

Código 2.7: Estrutura do Package.swift – *name*

2. Plataformas

O Código 2.8 define as plataformas em que o pacote pode ser executado. É comum especificar a versão mínima para cada plataforma.

```
let package = Package(
    // ...
    platforms = [
        .iOS(.v13),
        .macOS(.v10_15)
    ]
)
```

Código 2.8: Estrutura do Package.swift – *platforms*

3. Produtos

Os produtos (Código 2.9) especificam quais bibliotecas e módulos o pacote vai expor para serem utilizados por outros pacotes ou aplicativos. Um produto pode ser uma *library* ou um *executable*.

```
let package = Package(
    // ...
```

```
    products = [
        .library(
            name: "NomeDaBiblioteca",
            targets: ["NomeDoTarget"]
        ),
    ]
)
```

Código 2.9: Estrutura do Package.swift – *products*

Uma *library* indica que o produto é uma biblioteca que será utilizada por outros pacotes. Ela é compilada para um arquivo *.framework* ou *.a*, que pode ser importado e utilizado em outros projetos ou *targets*. Já um *executable* indica que o produto é um executável (*.app* no macOS ou binário no Linux) que pode ser executado diretamente no terminal ou como um aplicativo independente.

4. Dependências

As dependências (Código 2.10) são outros pacotes que o projeto requer para funcionar. Essas dependências podem ser pacotes hospedados em repositórios públicos ou internos. O SPM faz a gestão automática dessas dependências.

```
let package = Package(
    // ...
    dependencies = [
        .package(url: "https://github.com/nome/repo.git", from: "1.0.0")
    ]
)
```

Código 2.10: Estrutura do Package.swift – *dependencies*

O SPM se encarrega de buscar o pacote na URL fornecida e faz o *download* da versão especificada ou superior.

5. *Targets*

Os *targets* (Código 2.11) são as unidades principais dentro de um pacote. Cada *target* agrupa um conjunto de arquivos de código e pode depender de outros *targets* ou de dependências externas.

```
let package = Package(
```

```
    // ...
    targets = [
        .target(
            name: "NomeDoTarget",
            dependencies: ["OutraDependencia"]
        ),
        .testTarget(
            name: "NomeDoTargetTests",
            dependencies: ["NomeDoTarget"]
        ),
    ]
)
```

Código 2.11: Estrutura do Package.swift – *targets*

O *target* define um grupo de código que será compilado como parte do produto. Já o *testTarget* define um grupo de código voltado para testes unitários ou de integração, garantindo que o pacote seja testado de forma isolada.

2.3.3 Modularização em iOS usando SPM

O SPM permite que o projeto seja dividido em módulos, que podem ser reutilizados em diferentes partes do aplicativo. Essa abordagem traz diversas vantagens, especialmente no que diz respeito ao tempo de compilação.

O *Xcode* possui um sistema de *build* "inteligente", que recompila apenas o módulo que sofreu alterações, em vez de recompilar todo o projeto. Isso significa que, ao trabalhar em um módulo específico, se for necessário refazer o *build*, o *Xcode* se concentra apenas naquele módulo em que se está trabalhando, ignorando os módulos que não foram modificados e ignorando qualquer dependência que não faça parte do módulo atual.

Como resultado disso, o tempo de *build* cai significativamente, especialmente em projetos grandes, onde a recompilação total pode ser demorada. Essa otimização aumenta a produtividade dos desenvolvedores, permitindo um ciclo de desenvolvimento mais ágil e eficiente.

Na Figura 2.5, é possível observar a estrutura de arquivos do projeto no *Xcode*,

organizada de forma modular. O projeto foi configurado utilizando o SPM, sendo dividido em um **pacote que contém os módulos** e a **aplicação principal**. Cada *target* do pacote possui seu próprio *target* de testes, enquanto a aplicação principal também inclui seus próprios arquivos de testes.

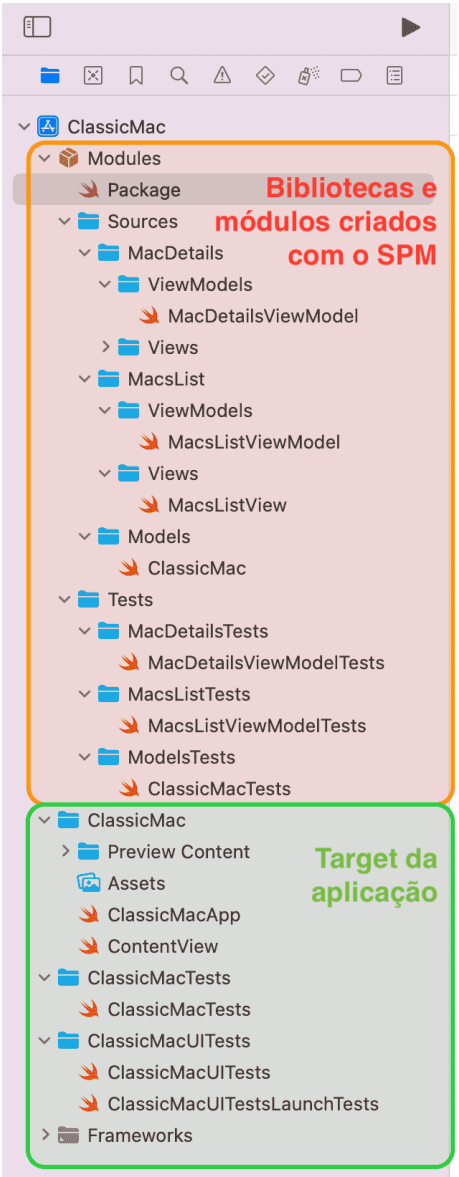

Figura 2.5: Módulos configurados no *Xcode*

O arquivo *Package.swift* foi configurado para expor duas bibliotecas: *Core* e *Modules*. Cada biblioteca possui seus *targets* independentes, que agrupam funcionalidades específicas. Um ponto fundamental na configuração é a gestão das dependências entre os *targets*. É altamente recomendado minimizar as dependências entre *targets*, especialmente aquelas dentro da biblioteca *Modules*, a fim de manter a modularidade e evitar um acoplamento excessivo. Dessa forma, é possível garantir um código mais coeso e fácil de

manter, além de melhorar a escalabilidade do projeto.

2.3.4 Benefícios da Modularização usando o SPM

O uso do *Swift Package Manager* (SPM) para modularizar projetos tem-se tornado cada vez mais comum, oferecendo uma série de vantagens que otimizam tanto o processo de desenvolvimento quanto a qualidade do *software* produzido.

Um dos principais benefícios do SPM, além da reutilização de código, é a facilidade de manutenção. Ao dividir o projeto em pequenos módulos independentes, é possível isolar funcionalidades específicas, tornando mais simples corrigir ou atualizar partes do código sem impactar outras áreas do projeto.

Para equipes de desenvolvimento, grandes ou pequenas, a modularização promove uma colaboração eficiente. Diferentes desenvolvedores podem trabalhar em módulos separados, evitando conflitos e facilitando o controle de versão.

Pacotes criados com o *Swift Package Manager* podem ser utilizados em diferentes plataformas do ecossistema *Apple*, incluindo iOS, macOS, watchOS, tvOS e visionOS. Isso permite criar funcionalidades que podem ser compartilhadas entre aplicativos destinados a diferentes dispositivos, otimizando o esforço de desenvolvimento e manutenção.

Por fim, o SPM é suportado nativamente pelo *Xcode*, facilitando a integração de *pipelines* de CI/CD, principalmente quando usado com o *Xcode Cloud* (APPLE, 2021), também uma ferramenta da *Apple*.

2.4 Conclusões do Capítulo

Este capítulo apresentou uma análise detalhada das principais arquiteturas de *software* em iOS, enfatizando a importância de abordagens modularizadas no desenvolvimento de aplicativos. A compreensão dos conceitos fundamentais, como MVC, MVVM e VIPER, é importante para garantir que as aplicações sejam escaláveis, testáveis e mantenham uma separação clara de responsabilidades. Além disso, a introdução do *SwiftUI* e a modularização através do *Swift Package Manager* são discutidas como práticas que promovem a eficiência e a organização do projeto. Ao estabelecer uma base teórica sólida, este trabalho

busca contribuir para o avanço do conhecimento na área de desenvolvimento de *software*, oferecendo uma solução que busca solucionar desafios atuais enfrentados na criação de aplicativos móveis.

3 Referencial Teórico

Neste capítulo, será apresentada a aplicação do Mapeamento Sistemático da Literatura (SILVA, 2009), detalhando a metodologia utilizada, incluindo a utilização do *framework* PICOC (SANTOS; PIMENTA; NOBRE, 2007) (População, Intervenção, Comparação, Desfecho e Contexto), que auxilia na formulação de questões de pesquisa e na definição de critérios de seleção dos estudos. Além disso, serão discutidos os principais resultados obtidos por meio da revisão sistemática, evidenciando as descobertas mais relevantes e as tendências observadas. Esta abordagem visa garantir a qualidade e a relevância da literatura, fortalecendo a precisão e a consistência dos resultados da pesquisa.

3.1 Mapeamento Sistemático da Literatura

Um Mapeamento Sistemático é uma metodologia que tem como objetivo identificar, avaliar e analisar de forma sistemática e abrangente as evidências disponíveis em relação a uma determinada questão de pesquisa ou tema de interesse.

Esta fase contou com as etapas de: i) definição do objetivo, ii) elaboração das questões de pesquisa, iii) seleção das fontes de dados, iv) composição da *string* de busca, v) definição de critérios de inclusão, exclusão e de qualidade dos trabalhos.

3.1.1 PICOC

O acrônimo PICOC (*Population*, *Intervention*, *Control*, *Outcome*, *Context*) representa uma estrutura utilizada na formulação de perguntas de pesquisa, especialmente em revisões sistemáticas. Ele ajuda a definir os elementos essenciais para uma investigação clara e focada. A **população** é o grupo de indivíduos ou contexto específico ao qual a pesquisa se refere. A **intervenção** é o tratamento, exposição ou intervenção que será avaliada. A **comparação** é o grupo de controle ou alternativa à intervenção para análise comparativa. O **desfecho** é o resultado esperado ou o efeito da intervenção sobre a população estudada. E por fim, o **contexto** é o ambiente ou cenário onde a pesquisa será

aplicada (por exemplo, hospitalar, comunitário, escolar).

Para este trabalho, foi desenvolvida a seguinte estrutura PICOC, definida na Tabela 3.1.

PICOC	Palavras-Chave
Population	*SwiftUI*
Intervention	MVVM; MVVM-C; *coordinator*
Control	*Not defined*
Outcome	SPM; *Modularization*; Navegação
Context	*Architecture*; *Pattern*

Tabela 3.1: PICOC

3.1.2 *String* de busca

A partir da tabela PICOC, foi criada a seguinte *string* de busca:

(*"swiftui"*)

AND

("mvvm" *OR* "mvvm-c" *OR* *"coordinator"*)

AND

(*"modularization"* *OR* "spm" *OR* *"modular"*) *OR* *"navigation"*

AND

(*"architecture"* *OR* *"pattern"*)

3.1.3 Critérios de Inclusão e Exclusão

Para garantir a relevância e a qualidade dos estudos selecionados para análise, foram definidos alguns critérios de inclusão: **publicações recentes:** artigos e estudos publicados nos últimos cinco anos (2019 a 2024), a fim de capturar as abordagens mais modernas e atuais em arquitetura de *software* para iOS, além de que 2019 foi o ano de lançamento do *SwiftUI* (APPLE, 2019d); **relevância temática:** estudos que abordam arquiteturas de *software* aplicadas ao desenvolvimento nativo de aplicativos para iOS, incluindo arquiteturas como *Model-View-ViewModel* (MVVM) e *Model-View-ViewModel-Coordinator* (MVVM-C) (FUKSA; SPETH; BECKER, 2025) e *SwiftUI*; **foco em modularização**

e navegação: trabalhos que tratem de modularidade e navegação no contexto de desenvolvimento móvel e que utilizam o *Swift Package Manager* (SPM) (APPLE, 2024b); **disponibilidade da pesquisa:** estudos que estão acessíveis integralmente para análise e revisão, sejam de acesso livre ou via bibliotecas acadêmicas, de forma gratuita. Artigos de outras bases de estudos *online*, de especialistas na área de desenvolvimento nativo iOS, também foram selecionados.

3.1.4 Resultados

Após serem estabelecidos os critérios de inclusão, exclusão e qualidade, e utilizando o diretório *Google Scholar*, foram encontrados e organizados os seguintes resultados na Tabela 3.2.

	Resultado
Trabalhos Encontrados	65
Trabalhos Eliminados	59
Trabalhos Utilizados	6

Tabela 3.2: Resultado Mapeamento Sistemático

Além do *Google Scholar*, outras bases como o *Medium* e outros *blogs* também foram utilizados para captura de estudos relacionados.

De forma geral, o resultado do mapeamento sistemático mostrou que a arquitetura MVVM é a principal escolhida para o desenvolvimento de aplicativos iOS, principalmente quando o *SwiftUI* é escolhido como *framework* para criação de interfaces. O MVVM tem-se mostrado bastante interessante quando utilizado juntamente com o *SwiftUI*, ainda mais quando o *Coordinator* também é utilizado.

Além disso, o mapeamento de literatura mostrou também que a separação do aplicativo em módulos vem se tornando uma abordagem cada vez mais usada quando comparada com os monolitos padrões. O *Swift Package Manager* (SPM) ainda é uma ferramenta nova, mas também se mostrou com muito potencial, ainda mais pela sua integração total com o *Xcode* e até com ferramentas de CI/CD (NILSSON, 2016) como o próprio *Xcode Cloud*.

3.1.5 Trabalhos Relacionados

Nesta seção, serão analisados os principais trabalhos e artigos acadêmicos que abordam o desenvolvimento de uma arquitetura de navegação modular para aplicações nativas em iOS. A pesquisa foi conduzida com base na *string* de busca já descrita anteriormente, buscando identificar as melhores práticas e estudos de caso que contribuem para a criação de aplicativos modulares, escaláveis e de fácil manutenção. O objetivo é contextualizar o presente estudo dentro do panorama atual, destacando abordagens já consolidadas e tendências emergentes na área de desenvolvimento móvel, com foco em modularidade e na gestão eficiente de navegação utilizando *SwiftUI*.

A pesquisa conduzida por Fuksa, Speth e Becker (2025) destaca a falta de uma documentação e detalhes sobre as diversas implementações do MVVM que são utilizadas atualmente, de uma maneira geral. A pesquisa encontrou 76 novas construções de *design* agrupadas em 29 aspectos de *design* em diferentes contextos de desenvolvimento, não apenas iOS.

Por outro lado, o trabalho de Indrawan (2023) tratou de implementar a arquitetura MVVM focando em desenvolvimento nativo para iOS para analisar o ganho de *perfomance* quando comparado com outras arquiteturas. Um ponto importante dessa pesquisa é que foi utilizada uma biblioteca chamada RXSwift (REACTIVEX, 2016) em vez do *SwiftUI*.

O artigo escrito por LEE (2024), em seu site, descreve que a arquitetura nem sempre é consistente e acaba resultando em novas variações do padrão. O texto conclui que a arquitetura é muito popular, e que, seguindo as regras, ela pode ajudar, deixando o código mais testável, reusável e fácil de entender.

O estudo feito por NAUMOV (2019), um desenvolvedor para as plataformas da *Apple* desde 2011, introduz a utilização das boas práticas do *Clean Architecture* utilizando a arquitetura MVVM. São apresentadas as três camadas da Arquitetura Limpa, bem como suas implementações em *Swift* e *SwiftUI*.

O trabalho realizado pela Margarida Rocha Raposo de Oliveira (2024) explora o conceito de *"code cities"*, onde representações espaciais de código ajudam desenvolvedores a identificar defeitos e oportunidades de refatoração em tempo real. A proposta

da ferramenta *SpatialLiveRef* é utilizar realidade aumentada para facilitar a navegação e análise de estruturas de *software* diretamente integradas a ambientes de desenvolvimento (IDE). No desenvolvimento voltado para o *Apple Vision Pro*, a autora optou por utilizar *SwiftUI* em conjunto com a arquitetura MVVM, garantindo uma implementação robusta e alinhada às melhores práticas de *design* de *software*.

Por fim, Forsanker (2024) fez um estudo sobre a qualidade de códigos gerados por *Large Language Models* (LLMs). Foram analisados os resultados obtidos por dois modelos diferentes, o ChatGPT-3.5 e o ChatGPT-4, e o resultado foi unânime: ambos os modelos geraram códigos já na arquitetura MVVM, mostrando que a arquitetura é amplamente utilizada pelos desenvolvedores.

3.1.6 Análise de Qualidade dos Artigos

Os trabalhos analisados fornecem contribuições importantes, destacando desafios e melhorias na abordagem modular baseada no MVVM. Um dos principais pontos levantados é a falta de padronização na implementação do MVVM, o que pode dificultar a testabilidade e a manutenção do código. Essa questão reforça a necessidade de uma estrutura modular bem definida, que organiza os componentes do sistema de forma clara e independente.

Outra contribuição relevante está na avaliação do impacto do MVVM na performance do sistema. Embora essa arquitetura melhore a separação de responsabilidades, sua implementação pode gerar sobrecarga caso não seja bem estruturada. Isso destaca a importância da modularização, permitindo um carregamento otimizado dos módulos e evitando tempos de compilação excessivos. Além disso, foram identificadas inconsistências comuns na aplicação do MVVM, o que evidencia a necessidade de boas práticas para garantir reuso e testabilidade do código.

A integração do MVVM com conceitos de *Clean Architecture* e *SwiftUI* também foi abordada, demonstrando que a modularização favorece a organização do código e simplifica a navegação entre telas (MARTIN, 2017). Essa visão busca a escalabilidade ao dividir o código em módulos reutilizáveis e gerenciar a navegação com o padrão *Coordinator*. Além disso, a aplicação de soluções inovadoras no desenvolvimento iOS, como ferramentas de refatoração e automação, destaca o potencial de aprimorar ainda mais a

modularização e a experiência do desenvolvedor.

Por fim, a consolidação do MVVM como um padrão amplamente adotado no desenvolvimento iOS reforça a relevância de modernizar essa abordagem com um modelo modular e eficiente.

3.2 Conclusões do Capítulo

Neste capítulo foi detalhado o processo e aplicação do mapeamento sistemático da literatura, incluindo o *framework* PICOC, que auxiliou na formulação da pergunta norteadora da pesquisa. Além disso, foram descritos os critérios de inclusão e exclusão de trabalhos relacionados. Chegou-se à conclusão de que a maioria dos textos relevantes está em bases de dados como *Medium* e *blogs* de diversos especialistas em desenvolvimento nativo com *Swift* e *SwiftUI*.

4 Build-block Architecture

No desenvolvimento de aplicativos para iOS, arquiteturas como *Model-ViewController* (MVC), *Model-View-ViewModel* (MVVM) e VIPER são amplamente utilizadas, mas muitas vezes sem um padrão bem definido. Isso leva a uma sobrecarga de responsabilidades em arquivos que não deveriam tê-las, dificultando a organização, escalabilidade e manutenção do código. O MVC, por exemplo, frequentemente resulta em *View Controllers* sobrecarregados, conhecidos como *Massive View Controllers*, que ocorre quando o *View Controller* acumula não apenas a lógica de apresentação, mas também regras de negócio e manipulação de dados, tornando o código difícil de testar e manter. Já o MVVM tenta solucionar esse problema ao delegar a lógica de apresentação para a *view model*, mas sem um padrão definido, o código pode ficar inconsistente e com *view models* também sobrecarregadas. O VIPER, por sua vez, separa claramente as responsabilidades, mas adiciona uma complexidade excessiva para a maioria dos projetos, resultando em uma grande quantidade de arquivos para cada funcionalidade.

Para resolver esses problemas, a *Build-block Architecture* (BbA) propõe uma abordagem modular dentro do MVVM, utilizando o *Swift Package Manager* (SPM) para separar o aplicativo em módulos totalmente independentes. A principal vantagem desse padrão é que cada módulo contém apenas o que é necessário para uma funcionalidade específica, reduzindo o acoplamento entre as camadas e tornando o código mais organizado e reutilizável, diferente do MVVM tradicional. Na abordagem do MVVM puro, muitas vezes todas as *Views* e *View Models* são mantidas em um único *target*, reunindo a maioria dos arquivos do projeto. A BbA busca estruturar o código em pequenos blocos reutilizáveis, melhorando a escalabilidade e a clareza da arquitetura.

Além disso, a BbA se beneficia de conceitos amplamente utilizados em outros padrões de projeto, como a Inversão de Dependência, garantindo que cada módulo dependa apenas do necessário, e os *coordinators*, que organizam a navegação de forma independente da interface de usuário. Se comparada a BbA a outros padrões de projetos conhecidos, como *Singletons*, *Factory Method* e *Facade*, que ajudam na organização de

objetos e na encapsulação da complexidade, a BbA se destaca por ser um padrão estrutural de maior abrangência. Afetando diretamente a forma como o projeto é organizado e escalado.

Outro benefício importante é a redução do tempo de *build*, um dos principais desafios em projetos grandes. Cada módulo é independente e apenas os módulos alterados precisam ser recompilados, tornando o desenvolvimento mais ágil. Além disso, essa abordagem permite que diferentes equipes trabalhem simultaneamente em diferentes partes da aplicação sem conflitos, melhorando a colaboração e a produtividade.

Em resumo, a *Build-block Architecture* combina os benefícios da modularização com a simplicidade do MVVM, proporcionando um código mais organizado, reutilizável e escalável. Enquanto arquiteturas tradicionais sofrem com falta de padronização e sobrecarga de responsabilidades, a BbA oferece uma solução prática e eficiente para o desenvolvimento iOS moderno.

4.1 Conceito da Build-block Architecture

4.1.1 Definição e Princípios

A *Build-block Architecture* é uma proposta de uma nova *"arquitetura"*, baseada nos conceitos da MVVM, e altamente modular para o desenvolvimento de aplicativos nativos para o ecossistema da *Apple* utilizando *Swift* e o *SwiftUI*. Inspirada na analogia de *blocos de lego*, onde cada bloco representa um módulo que pode ser conectado a outros através de um *Coodinator* para construir um aplicativo totalmente funcional, essa abordagem herda também vários conceitos da *Clean Architecture* (MARTIN, 2017), como a **reutilização de código** e a **separação clara de responsabilidades**, promovendo a escalabilidade e a manutenção do código.

A BbA é baseada na arquitetura bastante conhecida e utilizada atualmente, a **Model-View-ViewModel** (MVVM). Essa nova proposta herda todos os mesmos componentes – o *model*, a *view* e a *view model*. O componente central da *Build-block Architeture* é o *Coordinator*, que desempenha o papel crucial de integrar os módulos, estabelecendo o fluxo de navegação da aplicação. A combinação desses quatro componentes, aliada à es-

truturação e organização proporcionadas pelo *Swift Package Manager*, torna a BbA uma abordagem promissora para o desenvolvimento de aplicativos iOS.

Os principais objetivos propostos nesse estudo incluem: a **modularidade**, garantindo que cada funcionalidade do aplicativo seja encapsulada em um módulo independente; a **baixa dependência**, permitindo que modificações em um módulo não impactem o restante do sistema; a **escalabilidade**, que facilita a adição de novas funcionalidades ou a substituição de módulos; e a **facilidade de testes**, assegurando que cada módulo possa ser validado de forma isolada e totalmente independente.

É importante lembrar que o *Swift Package Manager* tem uma grande responsabilidade nessa arquitetura, facilitando a criação e a integração dos módulos com o restante do aplicativo.

4.1.2 Componentes da BbA

Nessa seção serão apresentados todos os componentes básicos que compõem a *Build-block Architecture*, bem como suas implementações.

4.1.2.1 *Model*

O *Model* (Código 4.1) é a camada responsável por gerenciar os dados e a lógica de negócio da aplicação.

```
1 struct ClassicMac: Identifiable, Hashable {
2     let id = UUID()
3     let name: String
4     let description: String
5 }
```

Código 4.1: Implementação do *model*

4.1.2.2 *View*

A *View* (Código 4.2) representa, única e exclusivamente, a interface do usuário. É a responsável por exibir os dados e capturar as interações do usuário. Essa camada é focada apenas em apresentar as informações fornecidas pela *view model* e enviar ações do

usuário de volta para ela.

É responsabilidade da *view* fornecer *callbacks* (ou *closures*) para notificar quando uma interação do usuário exige uma ação externa. Esses *callbacks* atuam como "ponte" entre a interface e o *coordinator*, garantindo que a *view* permaneça desacoplada das responsabilidades de navegação ou lógica de fluxo.

```
public struct MacsListView: View {
    @ObservedObject private var viewModel: MacsListViewModel
    private let didSelectMac: (ClassicMac) -> Void
    public init(
        viewModel: MacsListViewModel,
        didSelectMac: @escaping (ClassicMac) -> Void
    ) {
        self._viewModel = ObservedObject(wrappedValue: viewModel)
        self.didSelectMac = didSelectMac
    }
    public var body: some View {
        List(viewModel.classicMacs) { classicMac in
            Button(action: {
                didSelectMac(classicMac)
            }) {
                HStack(spacing: 16) {
                    VStack(alignment: .leading) {
                        Text(classicMac.name)
                            .font(.body)
                        Text(classicMac.description)
                            .font(.subheadline)
                            .foregroundStyle(.secondary)
                    }
                    .frame(maxWidth: .infinity, alignment: .leading)
                    Image(systemName: "chevron.right")
                        .imageScale(.medium)
                        .foregroundStyle(.secondary)
                }
            }
        }
```

```
31      }
32  }
```

Código 4.2: Implementação da *view*

4.1.2.3 *View Model*

A *view model* (Código 4.3), por sua vez, atua como intermediária entre o *model* e a *view*, transformando os dados do *model* em um formato que a *view* possa exibir e lidando com a lógica específica da interface, como formatação, validação e sincronização de dados.

```
1  final class MacsListViewModel: ObservableObject {
2      @Published private(set) var classicMacs: [ClassicMac] = []
3      func fetchMacs() { ... }
4      func setMacFavorite(mac: ClassicMac) { ... }
5  }
```

Código 4.3: Implementação da *view model*

4.1.2.4 *Coordinator*

O *coordinator* (Código 4.4) é o responsável por orquestrar os fluxos e navegações da aplicação. Ele usa as *views* implementadas nos módulos para organizá-los de forma que faça sentido para a aplicação.

```
1  struct MacsListCoordinator: View {
2
3      @StateObject private var viewModel = MacsListViewModel()
4
5      @State private var path = NavigationPath()
6
7      var body: some View {
8          NavigationStack(path: $path) {
9              MacsListView(
10                 viewModel: viewModel,
11                 didSelectMac: { selectedMac in
12                     path.append(selectedMac)
13                 }
```

```
14              )
15              .navigationTitle("Classic Macs")
16              .navigationDestination(for: ClassicMac.self) { selectedMac
    in
17                  MacDetailsView(
18                      viewModel: MacDetailsViewModel(
19                          mac: selectedMac
20                      )
21                  )
22              }
23          }
24      }
25  }
```

Código 4.4: Implementação da *view model*

4.1.3 Estrutura Inicial

A *Build-block Architecture* estabelece, como ponto de partida, duas bibliotecas fundamentais para a organização estrutural do projeto: ***Core*** e ***Modules***.

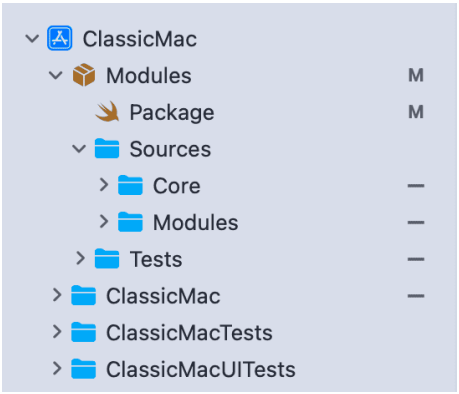

Figura 4.1: Estrutura Inicial da *Build-block Architecture*

A biblioteca ***Core*** reúne artefatos essenciais e reutilizáveis para a aplicação, incluindo extensões, componentes customizados e outros arquivos considerados fundamentais para o funcionamento geral da aplicação.

Por sua vez, a biblioteca ***Modules*** concentra os módulos específicos que implementam cada funcionalidade da aplicação. Cada módulo reflete uma divisão lógica do

sistema, encapsulando aspectos específicos daquele domínio, garantindo uma separação clara de responsabilidades. Essa organização modular permite não apenas a escalabilidade do aplicativo, mas também facilita a manutenção e a reutilização de código.

Em projetos de maior complexidade, a BbA admite a criação de bibliotecas adicionais, destinadas a atender demandas específicas, complementando as funcionalidades das bibliotecas ***Core*** e ***Modules*** e promovendo maior flexibilidade no desenvolvimento.

A BbA apresenta um crescimento vertical (Figura 4.2). Isso significa que novas bibliotecas, criadas para implementar funcionalidades específicas, devem ser organizadas em camadas acima ou abaixo da ***Core***. Dessa forma, elas podem ser utilizadas tanto pela própria ***Core*** quanto por outros módulos do sistema. Essa abordagem mantém a coesão da estrutura e promove um fluxo de dependências bem definido.

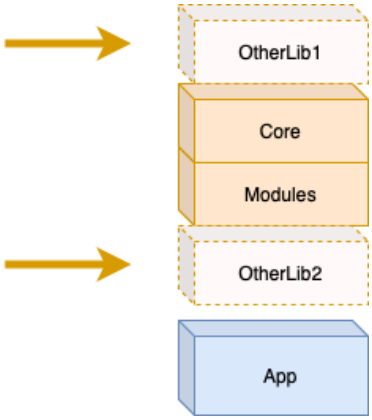

Figura 4.2: Crescimento Vertical das Bibliotecas na BbA

4.1.4 Modularização através de *blocos*

Um módulo é uma unidade independente que encapsula uma parte específica da funcionalidade do aplicativo. Ele é projetado para ser autônomo, com responsabilidades bem definidas, permitindo que seja reutilizado, modificado ou substituído sem afetar outras partes da aplicação. Um módulo pode conter componentes como *views*, modelos de dados, serviços, utilitários ou qualquer outra lógica necessária para cumprir sua função.

Na *Build-block Architecture*, cada módulo é tratado como um bloco de construção, similar às peças de *lego*, que podem ser conectadas, desconectadas e reorganizadas conforme necessário. Essa analogia destaca a flexibilidade e a simplicidade do modelo de modularização.

No diagrama 4.3, são mostradas as duas bibliotecas criadas com o SPM, a *Core* e a *Modules*, e o *target* do aplicativo principal, criado pelo *Xcode*. É essencial que não haja dependência entre os *targets* da mesma biblioteca para garantir que a arquitetura seja sempre modular e altamente desacoplada.

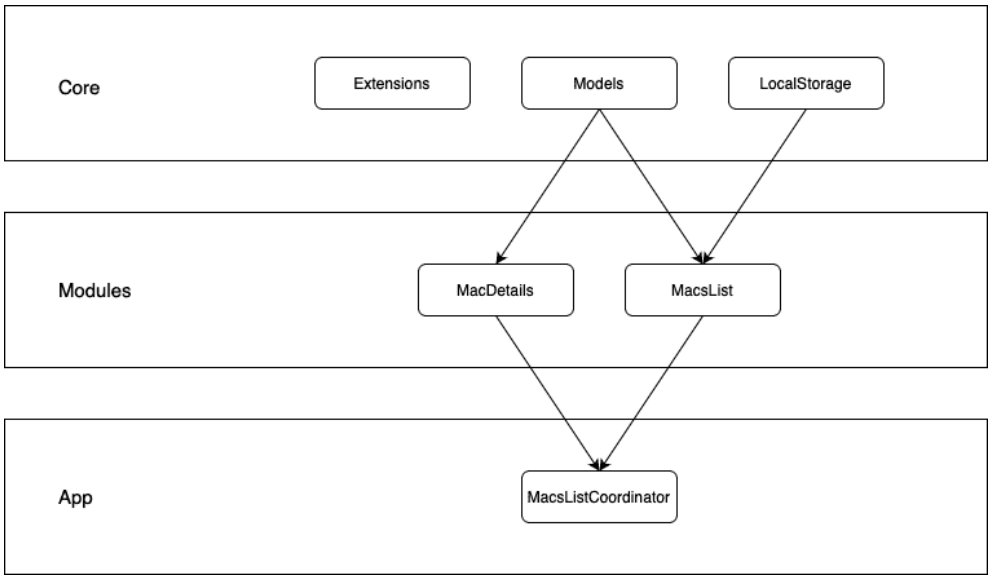

Figura 4.3: Diagrama da arquitetura básica do aplicativo *ClassicMac*

4.2 Aplicação do *Swift Package Manager*

Nesta seção será mostrada a aplicação do *Swift Package Manager* na prática, para a implementação do diagrama da Figura 4.3 apresentado na seção anterior para o aplicativo *ClassicMac*. O aplicativo *ClassicMac* 4.4 é um exemplo, retirado do vídeo *Platform State of the Union* (APPLE, 2024a) da *Worldwide Developer Conference 24* (WWDC) (APPLE, 2024c), que ilustrará a utilização da arquitetura. A implementação não será a mesma, apenas a ideia será usada para a exemplificação.

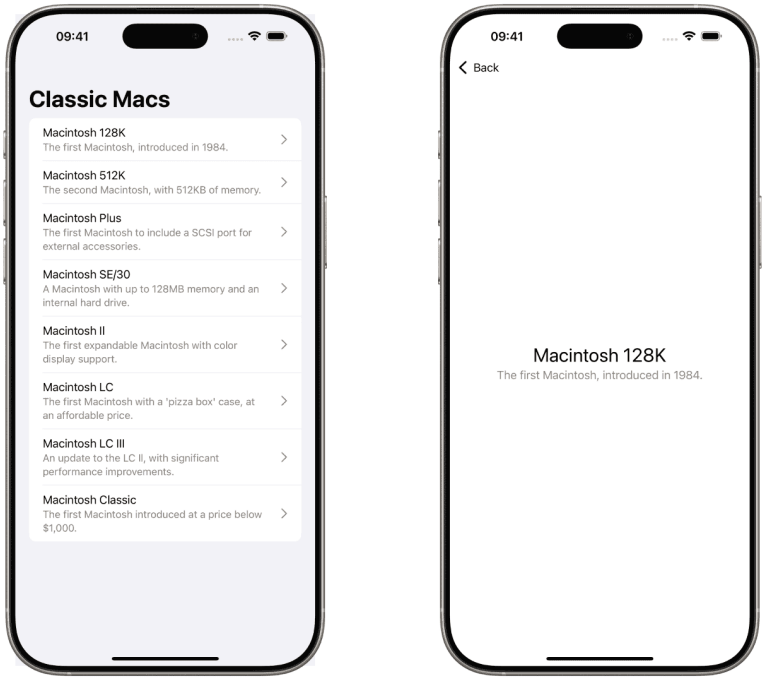

Figura 4.4: Aplicativo *ClassicMac*

4.2.1 Organização de Módulos

Cada módulo dentro da *Build-block Architecture* representa uma parte específica do aplicativo. Muitas vezes, representa uma única tela, mas também pode ser um bom local para adicionar outras telas que fazem sentido entre si.

O aplicativo em questão é muito simples. Se resume basicamente a uma lista de Macs já lançados pela *Apple* e, ao tocar em um, ele mostra uma tela com informações específicas sobre o Mac selecionado.

O primeiro passo é identificar as pequenas partes do sistema, que serão os módulos da aplicação. Para o *ClassicMac*, foram identificados dois módulos principais: o *MacsList* e o *MacDetails*. Cada módulo contém dois arquivos, a sua *view* e o seu *view model*. Sendo assim, os seguintes arquivos (Figura 4.5) compõem os módulos principais da aplicação.

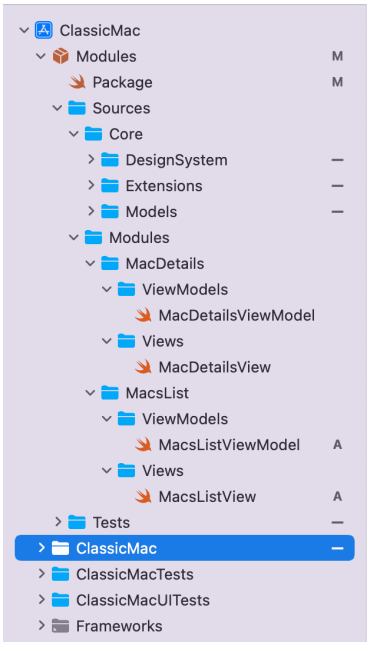

Figura 4.5: Estrutura inicial de arquivos do *ClassicMac*

Pela flexibilidade dessa abordagem, caso seja necessário adicionar uma funcionalidade para permitir que o usuário adicione um Mac aos favoritos a partir da tela de detalhes, basta implementar uma nova *view* (Figura 4.6) dentro do módulo *MacDetails*, garantindo coesão e organização ao manter a funcionalidade onde ela faz sentido.

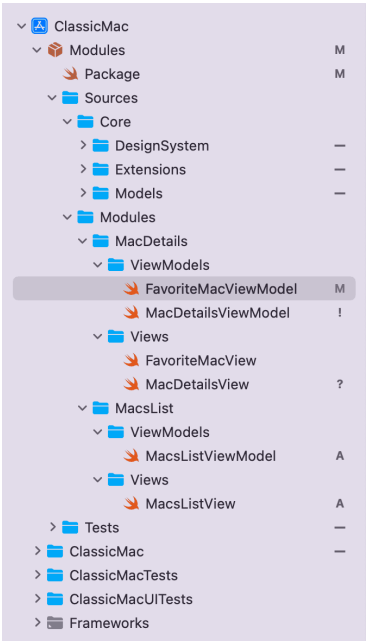

Figura 4.6: Estrutura inicial de arquivos do *ClassicMac* com a *view FavoriteMac* implementada

Nota-se que agora o módulo *MacDetails* não se limita a uma única *view*, mas contém um conjunto de componentes projetados para fazer com que esse módulo funcione

de maneira independente.

Essa abordagem permite que, à medida que o aplicativo evolua e essa tela seja reutilizada em outras partes do sistema, o módulo se consolide como uma unidade independente.

4.3 *Coordinator* como Gerenciador de Navegação

Nesta seção, é abordado um dos componentes fundamentais da *Build-block Architecture*: o ***Coordinator***.

Serão discutidas as melhores práticas para sua implementação, destacando como esse padrão pode simplificar o gerenciamento da navegação em aplicativos complexos. Além disso, será realizada uma análise das principais formas de navegação – *push* e *modal*.

4.3.1 Separação de Fluxos de Navegação

Para uma aplicação eficiente da *Build-block Architecture*, é essencial identificar e separar os fluxos básicos da aplicação que está sendo desenvolvida. Por exemplo, no caso do aplicativo *ClassicMac*, existe essencialmente um único fluxo: uma lista de modelos de Mac que, ao serem selecionados, direciona o usuário para uma tela com os detalhes do modelo escolhido.

O gerenciamento desse fluxo é responsabilidade do *coordinator*, que deve ser implementado diretamente no *target* principal da aplicação. A estrutura dessa implementação é ilustrada na Figura 4.7.

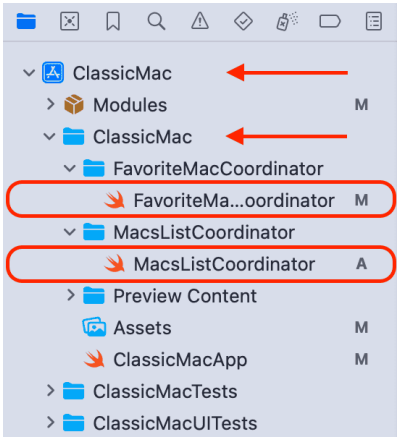

Figura 4.7: Local para implementação dos *coordinators*.

4.3.2 Implementação de Navegação *Push* e *Modal*

Os *guidelines* de *design* da *Apple* definem dois tipos principais de navegação: *push* e *modal*. A navegação ***push*** insere uma nova tela na pilha de navegação, preservando o histórico das telas anteriores. É recomendada quando o usuário permanece no mesmo contexto ou fluxo de interação. Por outro lado, a navegação ***modal*** apresenta uma nova tela sobreposta à anterior, criando um novo contexto de navegação. Essa abordagem não faz parte da pilha de navegação existente, isolando a interação atual do restante do fluxo.

Em *SwiftUI*, a criação de uma pilha de navegação *push* é realizada com o componente *NavigationStack*, que gerencia o histórico de telas. Entre as diferentes formas de implementar uma *NavigationStack*, destaca-se a abordagem baseada no controle da pilha através de um *array*. Essa técnica oferece maior flexibilidade ao Coordinator, permitindo um controle preciso sobre as telas exibidas ao usuário.

No Código 4.5, ao selecionar um Mac da lista, o item é adicionado ao *array* ***path***, resultando na apresentação de uma nova *view* contendo os detalhes do item selecionado.

```
@State private var path: [ClassicMac] = []
NavigationStack(path: $path) {
    List(classicMacs) { classicMac in
        Button(classicMac.name) {
            path.append(classicMac)
        }
    }
    .navigationDestination(for: ClassicMac.self) { classicMac in
        Text(classicMac.name)
    }
}
```

Código 4.5: Implementação da *NavigationStack*

Para a implementação de navegação *modal*, o *SwiftUI* oferece dois métodos principais: `sheet` e `fullScreenCover`. Apesar de ambos apresentarem telas modais, eles possuem diferenças significativas em termos de comportamento visual e experiência do usuário.

O `sheet` (Figura 4.8) apresenta a tela em formato de cartão que desliza de baixo para cima, ocupando apenas uma parte da tela em dispositivos como o iPhone. Esse tipo

de modal pode ser descartado deslizando-o para baixo. No iPad, o `sheet` aparece como um *popup* centralizado, mantendo o mesmo gesto de fechamento.

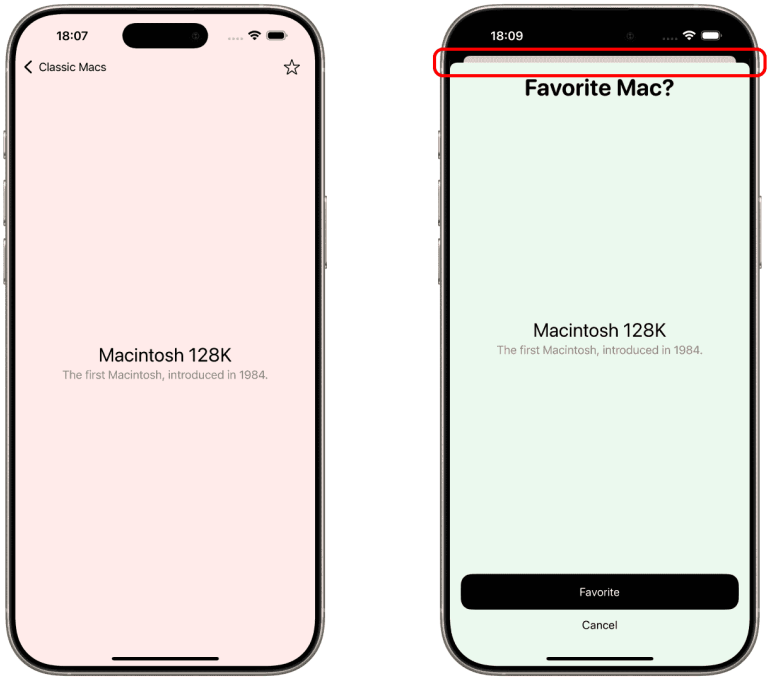

Figura 4.8: Modal – *sheet*

O `fullScreenCover` (Figura 4.9), por sua vez, exibe uma tela *modal* que ocupa toda a área visível, cobrindo completamente o conteúdo anterior. Diferentemente do `sheet`, o `fullScreenCover` não permite o gesto de deslizar para baixo para fechamento (de forma nativa).

A escolha entre `sheet` e `fullScreenCover` deve ser orientada pelas necessidades do projeto e pelas diretrizes definidas pelo time de *design*, garantindo que a experiência do usuário seja consistente e adequada ao contexto de uso.

4.3.3 Implementação do Coordinator

Com base nos conceitos de navegação apresentados nas subseções anteriores, é possível implementar um *Coordinator* na prática, compreendendo seu funcionamento e o seu papel dentro da arquitetura.

A principal função do *coordinator* na *Build-block Architecture* é gerenciar o fluxo de navegação, decidindo qual tela deve ser exibida em cada momento. Essa responsabilidade torna o *coordinator* um componente essencial para a organização e escalabilidade

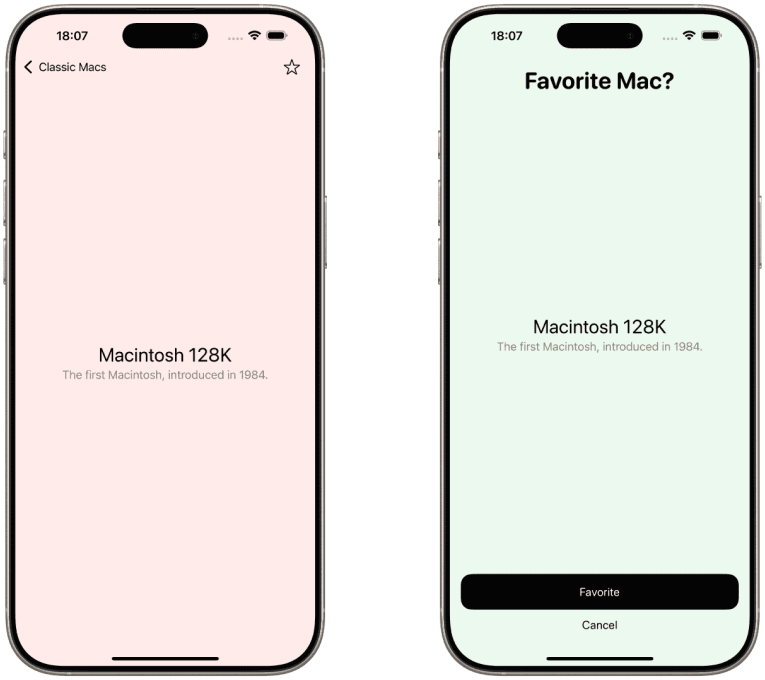

Figura 4.9: Modal – *fullScreenCover*

de aplicativos complexos.

Na Figura 4.3 da Subseção 4.1.2, foram identificados os módulos que compõem a aplicação *ClassicMac*. Esses módulos servem como ponto de partida para determinar os *coordinators* necessários. Embora cada módulo possua suas próprias *views*, é importante destacar que um módulo não corresponde necessariamente a um único *coordinator*. Um único *coordinator* pode gerenciar mais de um módulo, agrupando-os em um fluxo coeso. No diagrama, por exemplo, destacam-se dois módulos principais: *MacsList* e *MacDetails*, cujas *views* são interligadas pelo **MacsListCoordinator** (Código 4.6), que centraliza a navegação entre elas.

```
1  import SwiftUI
2  import MacsList
3  import MacDetails
4  struct MacsListCoordinator: View {
5      @StateObject private var viewModel = MacsListViewModel()
6      @State private var selectedMac: ClassicMac?
7      @State private var path = NavigationPath()
8      @State private var isPresentingFavoriteView: Bool = false
9
10     var body: some View {
```

```
        NavigationStack(path: $path) {
            MacsListView(
                viewModel: viewModel,
                didSelectMac: { selectedMac in
                    path.append(selectedMac)
                }
            )
            .navigationTitle("Classic Macs")
            .navigationDestination(for: ClassicMac.self) { mac in
                MacDetailsView(
                    viewModel: MacDetailsViewModel(mac: mac),
                    didTapFavoriteButton: {
                        isPresentingFavoriteView.toggle()
                    }
                )
                .sheet(isPresented: $isPresentingFavoriteView) {
                    FavoriteMacView(
                        viewModel: FavoriteMacViewModel(mac: mac)
                    )
                }
            }
        }
    }
}
```

Código 4.6: Implementação do MacsListCoordinator

Cada *coordinator* possui uma *View* principal, que serve como ponto inicial do fluxo. Para essa *View* inicial, é necessário instanciar sua *View Model* utilizando o marcador **@StateObject**. Esse marcador garante que o objeto será inicializado apenas uma vez durante o ciclo de vida do *Coordinator*. Ou seja, enquanto o *Coordinator* permanecer em memória, a *View Model* principal também será mantida. Ao término do fluxo, todos os objetos associados serão liberados automaticamente.

Essa implementação modularizada e baseada em *Coordinators* contribui para uma navegação organizada, aumentando a manutenibilidade e a flexibilidade do aplicativo.

4.4 Vantagens da *Build-block Architecture*

A *Build-block Architecture* é uma proposta moderna e altamente modular para o desenvolvimento de aplicativos nativos para as plataformas da *Apple*, proporcionando diversos benefícios com sua aplicação. Nesta seção, serão exploradas suas vantagens por meio de três perspectivas principais: **escalabilidade e flexibilidade**, **testabilidade e manutenção**, e por fim, **uma breve comparação com arquiteturas tradicionais**.

4.4.1 Escalabilidade e Flexibilidade

A medida que projetos vão se tornando cada vez mais complexos, surgem novos desafios para manter a organização e a manutenção dos códigos. Escalabilidade é um dos principais pontos de atenção em grandes aplicativos. Muitas aplicações nascem pequenas, fazendo sentido ser um "monolito", mas à medida que começam a crescer, tornam-se quase impossíveis de mantê-las. Por isso, abordagens modernas, escaláveis e altamente flexíveis fazem-se necessárias para trazer mais agilidade às equipes de desenvolvedores.

Pensando nisso, a BbA promove escalabilidade ao permitir que aplicativos cresçam em funcionalidade sem comprometer a organização do código. Cada módulo e cada *coordinator* são projetados para serem totalmente independentes e reutilizáveis, o que facilita a inclusão (ou a exclusão) de novos recursos ou a modificação de fluxos existentes. Por exemplo, é possível alterar o **MacsListCoordinator** (Código 4.6), criando um novo fluxo com o **FavoriteMacCoordinator** (Código 4.7) que será responsável pela ação de adicionar ou remover um Mac como favorito.

```
1 import SwiftUI
2 import MacsList
3 import MacDetails
4
5 struct FavoriteMacCoordinator: View {
6     @StateObject private var viewModel: FavoriteMacViewModel
7     init(mac: ClassicMac) {
8         _viewModel = StateObject(wrappedValue: FavoriteMacViewModel(mac:
      mac))
9     }
```

```
10      var body: some View {
11          NavigationStack {
12              FavoriteMacView(viewModel: viewModel)
13          }
14      }
15  }
16  struct MacsListCoordinator: View {
17      ...
18      var body: some View {
19          NavigationStack(path: $path) {
20              MacsListView(...)
21              .navigationTitle("Classic Macs")
22              .navigationDestination(for: ClassicMac.self) { mac in
23                  MacDetailsView(...)
24                      .sheet(isPresented: $isPresentingFavoriteView) {
25                          FavoriteMacCoordinator(mac: mac) // Remove a
    chamada da view direto, chamando agora o coordinator respons vel
26                      }
27              }
28          }
29      }
30  }
```

Código 4.7: Implementação do FavoriteMacCoordinator

A flexibilidade da BbA também é evidenciada pela capacidade de cada *coordinator* gerenciar múltiplos módulos. Por exemplo, o **MacsListCoordinator** pode implementar recursos tanto do módulo de lista quanto do de detalhes de Macs, enquanto o **FavoriteMacCoordinator** gerencia apenas a funcionalidade de favoritos. Essa abordagem reduz a necessidade de redefinir fluxos ao escalar a aplicação.

4.4.2 Testabilidade e Manutenção

A estrutura e organização da BbA foram pensadas a fim de facilitar a criação de testes isolados e bem organizados para cada módulo do projeto. O uso do SPM simplifica a estruturação e a manutenção desses testes, já que cada módulo pode ser testado de

forma independente, facilitando a detecção e correção de problemas em fluxos específicos. Além disso, enquanto o SPM gerencia os testes para os módulos independentes, o *target* principal da aplicação organiza os testes específicos para cada *coordinator*, garantindo maior cobertura e consistência durante toda a etapa de desenvolvimento e validação do aplicativo.

A separação de responsabilidades, uma das características centrais da BbA, também contribui para uma manutenção mais eficiente e previsível. Por exemplo, implementar alterações na funcionalidade de favoritos exige apenas modificações no ***FavoriteMacCoordinator*** e nos módulos diretamente associados a essa funcionalidade. Isso não apenas minimiza o risco de introdução de erros em outras áreas do aplicativo, mas também acelera o processo de desenvolvimento e manutenção.

4.4.3 Comparação com Arquiteturas Tradicionais

A BbA vem com a proposta de resolver algumas limitações de arquiteturas tradicionais como MVC, VIPER, etc. Essas arquiteturas, principalmente a MVC, muitas vezes apresentam limitações devido à centralização excessiva de responsabilidades, como ocorre em *ViewControllers*, criando-se as famosas *"God Classes"*. Nesse cenário, as regras de negócio, navegação e apresentação são combinadas em um único componente, dificultando a escalabilidade e manutenção de aplicativos mais complexos.

Surge-se então a necessidade de distribuir as responsabilidades entre os módulos e os *coordinators*, promovendo uma organização mais eficiente. Enquanto o MVC tradicional pode atender bem a aplicativos pequenos, a BbA sobressai-se em projetos maiores e mais complexos.

Além disso, arquiteturas como VIPER compartilham do objetivo de promover uma alta modularidade e a separação de responsabilidades, mas podem introduzir maior complexidade devido ao número elevado de camadas e arquivos. Em contrapartida, a BbA encontra um equilíbrio entre simplicidade e organização, tornando-se uma solução viável tanto para equipes pequenas quanto para projetos de grande escala.

Essa clara divisão de responsabilidades, aliada à reutilização e simplicidade, posiciona a BbA como uma alternativa robusta para projetos que demandam escalabilidade,

flexibilidade e manutenção eficiente, especialmente com equipes remotas.

Voltando à Tabela 2.1, criou-se a Tabela 4.1 com a adição de uma coluna para a BbA, com o objetivo de evidenciar suas vantagens em relação às demais abordagens. Como observado, a BbA se destaca como a arquitetura mais eficiente, pois aprimora os conceitos do MVVM ao estabelecer padrões que aumentam a modularidade, reduzem o acoplamento e melhoram a organização do código. Com uma estrutura altamente modular, aliada a um baixo tempo de *build* e menor complexidade em comparação ao VIPER, torna o desenvolvimento mais ágil e sustentável. Além disso, a escalabilidade e flexibilidade na navegação garantem uma arquitetura preparada para evolução contínua, facilitando a manutenção e aprimorando a eficiência do MVVM existente.

	MVC	**MVVM**	**VIPER**	**BbA**
Organização do Código	Baixa	Moderada	Alta	**Alta**
Acoplamento	Alto	Moderada	Baixo	**Baixo**
Modularização	Baixa	Moderada	Alta	**Alta**
Complexidade	Baixa	Moderada	Alta	**Baixa**
Escalabilidade	Baixa	Moderada	Alta	**Alta**
Tempo de Build	Alto	Alto	Moderado	**Baixo**
Facilidade de Manutenção	Baixa	Moderada	Alta	**Alta**
Flexibilidade na Navegação	Baixa	Moderada	Alta	**Alta**

Tabela 4.1: Comparação entre padrões de arquitetura: MVC, MVVM, VIPER e BbA

4.5 Conclusões do Capítulo

Neste capítulo, foi apresentada uma análise detalhada da *Build-block Architecture* e suas vantagens em relação às arquiteturas tradicionais, como MVC e VIPER. A BbA se destaca pela sua abordagem altamente modular, que permite uma maior flexibilidade, escalabilidade e testabilidade no desenvolvimento de aplicativos. A modularização, facilitada pelo uso do *Swift Package Manager*, permite que diferentes partes de um projeto sejam desenvolvidas, testadas e mantidas de forma independente. Isso não apenas melhora a organização do código, mas também promove uma maior flexibilidade para a introdução de novas funcionalidades e adaptações em projetos existentes.

Ao longo do capítulo, foram apresentados cenários práticos que demonstraram a eficácia da BbA em resolver desafios comuns enfrentados em arquiteturas tradicionais.

Em suma, a BbA apresenta-se como uma solução moderna e eficiente para projetos que demandam alto nível de organização, escalabilidade e manutenção ágil. Sua adoção é particularmente vantajosa em contextos de desenvolvimento dinâmico e colaborativo, consolidando-se como uma alternativa robusta para superar as limitações de arquiteturas mais tradicionais e atender às exigências do mercado atual.

5 Estudo de Caso: *Follows Audit*

O *Follows Audit* é um aplicativo desenvolvido totalmente em *Swift* e utilizando o *SwiftUI*. Com ele, é possível monitorar as informações de uma conta no Instagram de forma segura e privada, sem a necessidade de fazer *login* na plataforma. O *Follows Audit* funciona totalmente offline e os dados são tratados localmente, no dispositivo do usuário.

Esse aplicativo surgiu da necessidade de amigos em conseguirem, de forma prática e segura, as informações sobre seu círculo social no Instagram. Esse processo era todo feito de forma manual e o *Follows Audit* foi criado para deixar esse processo mais rápido, fácil e acessível a todos os públicos.

5.1 Por que o *Follows Audit*?

O *Follows Audit* foi escolhido como estudo de caso neste trabalho por ser um aplicativo desenvolvido inteiramente pelo aluno do presente trabalho, desde sua concepção até a implementação. Isso significa que a *Build-block Architecture* (BbA) não foi simplesmente aplicada ao aplicativo, mas sim **criada em conjunto** com ele, evoluindo conforme as necessidades do projeto se tornavam mais complexas.

Desde o início do desenvolvimento, havia uma preocupação em construir um aplicativo que fosse altamente **modular**, **escalável**, **testável** e principalmente, **de fácil manutenção**, características essenciais para um *software* que lida com funcionalidades variadas.

A abordagem tradicional, baseada em arquiteturas como *Model-View-Controller* (MVC) ou *Model-View-ViewModel* (MVVM) simples, não atendia plenamente às demandas do projeto, especialmente no que diz respeito à organização da navegação e separação de responsabilidades, tendo que vista que nele foi utilizada o *framework SwiftUI*, que por mais que esteja evoluindo e se tornando maduro com o passar dos anos, ainda apresenta uma alta complexidade quando se trata de navegação entre telas.

O foco da *Build-block Architecture* não é iniciar uma arquitetura do zero, mas me-

lhorar e estabelecer padrões para uma arquitetura já amplamente utilizada – a **MVVM**. Sendo assim, a Bba surge como uma resposta a esses desafios, principalmente envolvendo navegação, permitindo que o *Follows Audit* fosse estruturado de forma modular desde o primeiro momento. Cada parte do aplicativo — desde a interface até a lógica de dados — foi planejada para funcionar como blocos independentes, facilitando sua reutilização e testabilidade. Além disso, o uso do *Swift Package Manager* para separar o código em módulos garantiu um melhor controle sobre as dependências e tornou o desenvolvimento mais organizado e eficiente.

Dessa forma, a escolha do *Follows Audit* como estudo de caso torna-se natural, pois ele não apenas exemplifica a aplicação da *Build-block Architecture*, mas também foi o ambiente onde essa arquitetura nasceu.

5.2 Limitações do MVVM "puro"

Embora o MVVM seja uma das arquiteturas mais utilizadas no desenvolvimento de aplicativos iOS, sua implementação sem um controle adequado pode gerar problemas de organização e manutenção, especialmente à medida que o projeto cresce. Um dos principais desafios enfrentados no *Follows Audit* foi o acúmulo de responsabilidades dentro dos *view models*, que passaram a lidar não apenas com a **lógica de apresentação**, mas também com a **navegação entre telas**, **manipulação de dados** e **comunicação com serviços externos**. Esse cenário levou a um código altamente acoplado, dificultando a modularização do projeto.

Além disso, a estrutura tradicional do MVVM organiza os arquivos do projeto separando as *Views*, *ViewModels* e *Models* em pastas distintas, sem uma divisão lógica baseada nas funcionalidades da aplicação. Isso pode gerar dificuldades na escalabilidade, pois os desenvolvedores precisam navegar entre múltiplos arquivos para entender o funcionamento de uma única funcionalidade.

Em vez de estruturar o projeto com base apenas nos tipos de componentes (*Views*, *ViewModels* e *Models*), a *Build-block Architecture* divide o código em módulos independentes, onde cada funcionalidade do aplicativo possui sua própria estrutura isolada, incluindo todas as suas dependências. Isso garante que cada parte do sistema funcione

como um bloco reutilizável e autônomo, facilitando a manutenção e reduzindo o impacto de mudanças em outras áreas do projeto.

Outra característica essencial da Bba é a introdução do *Coordinator*, componente responsável exclusivamente pelo gerenciamento da navegação. No MVVM puro, a transição entre telas muitas vezes acaba sendo responsabilidade dos *ViewModels*, o que gera alto acoplamento e dificulta os testes de unidade. Com a separação da lógica de navegação dos módulos, os *ViewModels* podem se concentrar apenas na manipulação do estado da interface, tornando o código mais coeso.

Além de resolver os problemas de organização e separação de responsabilidades, essa abordagem também melhora a testabilidade do código. Como cada funcionalidade está encapsulada dentro de um "pacote"específico, é possível testar componentes individuais sem depender de outras partes da aplicação. Isso reduz o tempo e a complexidade dos testes, além de facilitar a identificação de possíveis falhas.

Ao aplicar essa abordagem no *Follows Audit*, a arquitetura permitiu que o aplicativo crescesse sem comprometer a clareza e a manutenção do código. A modularização garantiu maior controle sobre a organização do projeto, enquanto o uso do *Coordinator* trouxe mais flexibilidade para gerenciar fluxos de navegação.

5.3 Descrição Geral do *Follows Audit*

Nessa seção serão discutidos todos os detalhes, tanto de escopo do projeto, quanto de implementação do aplicativo *Follows Audit* (Figura 5.1).

5.3.1 Escopo do Projeto

O *Follows Audit* é um aplicativo nativo para iOS, desenvolvido em *SwiftUI*, que tem como objetivo fornecer uma análise detalhada das interações do usuário no Instagram. Ele é projetado para oferecer *insights* sobre a atividade e as conexões do usuário, permitindo uma visão clara sobre aspectos como seguidores, contas bloqueadas, curtidas, comentários e outras interações.

As principais funcionalidades que incluem o Follows Audit são:

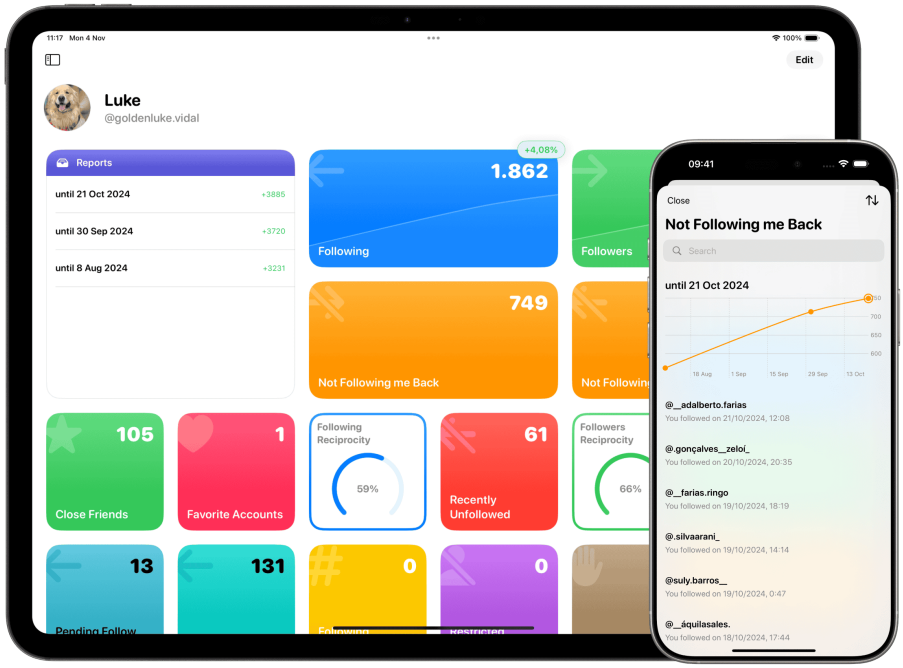

Figura 5.1: Capturas de tela do *Follows Audit* para iPhone e iPad

- **Gestão de Múltiplas Contas**
  - O usuário pode adicionar várias contas do Instagram para análise;
  - Cada conta possui suas próprias configurações e relatórios armazenados separadamente.
- **Relatórios Detalhados**
  - Gerar relatórios que incluem métricas como número de seguidores, contas recentemente seguidas ou deixadas de seguir, contas favoritadas, entre outros;
  - Permitir comparações entre relatórios para identificar mudanças nas interações ou conexões.
- **Customização com *Widgets***
  - O usuário deve poder customizar a tela inicial do aplicativo para cada conta com *widgets* dinâmicos;
  - Exemplos de *Widgets*;
    - Seguidores;
    - Seguindo;

    - Contas bloqueadas;
    - Contas que eu sigo mas não me seguem de volta;
    - Contas que me seguem mas eu não sigo de volta;
    - Reciprocidade de seguidores;
    - Reciprocidade de seguindo;
    - Contas que deixaram de me seguir;
    - Contas que eu deixei de deguir.
  - Ao tocar em um *widget*, deve mostrar a lista com as contas relacionadas à interação selecionada.

- **Processamento e Armazenamento de Dados**
  - Processar grandes volumes de dados em formatos JSON, convertendo-os para modelos estruturados e otimizados;
  - Utilizar um banco de dados local, SQLite, para organizar e salvar as informações de contas, relatórios e interações.

5.3.2 *Design* do *Follows Audit*

Por ser um aplicativo desenvolvido exclusivamente para o ecossistema do iOS, o *Follows Audit* foi projetado seguindo as diretrizes de *design* (*guidelines*) recomendadas pela *Apple*. Durante o planejamento, um dos principais objetivos era criar uma interface que proporcionasse familiaridade e intuitividade para os usuários. Essa abordagem levou à implementação de uma funcionalidade central do aplicativo: os *widgets*.

O *Follows Audit* conta com uma biblioteca diversificada de *widgets*, permitindo que os usuários selecionem aqueles que melhor atendam às suas necessidades e os posicionem na tela inicial do aplicativo de forma personalizada. Esses *widgets* oferecem informações relevantes e práticas, como o número de seguidores, o número de contas seguidas pelo usuário, o número de contas que o usuário segue, mas que não o seguem de volta, entre outros.

Além disso, os *widgets* são dinâmicos. Sempre que um novo relatório é importado, os dados apresentados são automaticamente atualizados, garantindo que as informações

exibidas estejam sempre alinhadas com os relatórios mais recentes. Esse sistema garante o compromisso do *Follows Audit* em fornecer uma experiência visualmente atraente e funcional, priorizando a interação e a praticidade para o usuário.

5.4 Implementação da *Build-block Architecture* no *Follows Audit*

5.4.1 Modularização do Código

O *Follows Audit* pode ser classificado como um aplicativo de alta complexidade, considerando sua diversidade de fluxos e funcionalidades. O primeiro passo para aplicar a *Build-block Architecture* (BbA) consiste em realizar um mapeamento detalhado dos módulos que compõem o projeto.

Como visto anteriormente, a BbA utiliza, inicialmente, duas bibliotecas fundamentais para estruturar o projeto: ***Core*** e ***Modules***. Em projetos de maior complexidade, torna-se necessária a criação de bibliotecas adicionais que auxiliem na implementação de funcionalidades específicas. No caso do *Follows Audit*, foi desenvolvida, de maneira complementar, a biblioteca ***FAFoundation***, cuja principal responsabilidade é fornecer a implementação de protocolos como ***Entity***, ***Model*** e ***PrimitiveMetric***. Esses protocolos servem como base para a implementação de funcionalidades específicas em outras bibliotecas do projeto, como no ***Core***, que agrega funções essenciais para atender às necessidades dos módulos.

A Figura 5.2 mostra a hierarquia proposta pela BbA, dos módulos identificados para o *Follows Audit*. Percebe-se que há, de fato, uma hierarquia sendo seguida: a biblioteca ***Core*** implementa os protocolos criados na ***FAFoundation***, camada acima dela. E a biblioteca ***Modules***, por sua vez, implementa protocolos e classes da biblioteca ***Core***. A ideia é que a arquitetura seja totalmente flexível para atender às demandas do projeto.

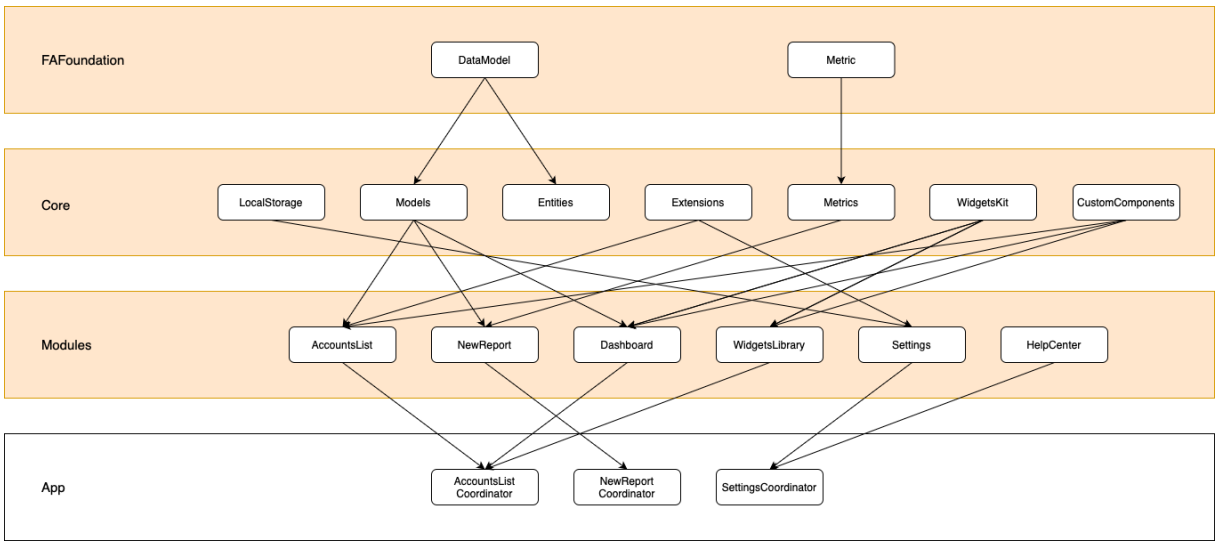

Figura 5.2: Diagrama dos módulos do *Follows Audit*

5.4.2 Uso do *Coordinator* para Navegação

Na *Build-block Architecture*, os módulos são conectados através de um ***Coordinator***, que é responsável por gerenciar a apresentação e navegação entre as telas. O *coordinator* funciona como uma ponte, garantindo que os módulos sejam combinados de forma coerente para formar fluxos completos da aplicação.

Essa abordagem permite que a navegação seja controlada de maneira centralizada, evitando que cada módulo precise conhecer os detalhes de outros, o que preserva a independência dos blocos. Além disso, facilita a alteração de fluxos, já que o *coordinator* pode ser modificado sem que os módulos precisem ser alterados.

No contexto do *Follows Audit*, o primeiro fluxo que é identificado ao iniciar a aplicação é a tela com a lista de contas disponíveis. Ao tocar em uma conta, o usuário deve ser direcionado para a tela de *dashboard* com os *widgets* escolhidos.

O Código 5.1 mostra a implementação do **AccountsListCoordinator**. O ***NavigationSplitView*** foi utilizado no lugar do **NavigationStack** apenas para deixar a interface otimizada para o iPad. O *coordinator* então, cria a *view* **AccountsListView** e recebe um evento quando alguma conta da lista foi selecionada. Nesse momento, é importante reparar que o *coordinator* é o responsável por fazer a navegação. Nenhuma *view* tem conhecimento de outra, apenas os *coordinators*.

```
1 import SwiftUI
2 import AccountsList
```

```
3  import Dashboard
4
5  struct AccountsListCoordinator: View {
6      @StateObject private var viewModel = AccountsListViewModel()
7
8      @State private var selectedAccountId: Int? = nil
9      @State private var activeSheet: AccountsListActiveSheet? = nil
10
11     var body: some View {
12         NavigationSplitView {
13             AccountsListView(
14                 viewModel: viewModel,
15                 didSelectAccount: { index, selectedAccountId in
16                     if entitlementManager.hasPro || index < Constants.
    REPORTS_LIMIT_FREE_VERSION {
17                         selectAccount(accountId: selectedAccountId)
18                     } else {
19                         activeSheet = .buyPro
20                     }
21                 },
22                 didTapImportReport: {
23                     importNewReport()
24                 },
25                 didTapSettingsButton: {
26                     activeSheet = .settings
27                 },
28                 didTapProButton: {
29                     activeSheet = .buyPro
30                 }
31             )
32         } detail: {
33             if let selectedAccountId, let selectedAccount = viewModel.
    getAccount(by: selectedAccountId) {
34                 DashboardCoordinator(
35                     account: selectedAccount,
36                     didTapDeleteAccount: { account in
```

```
37                          viewModel.deleteAccount(accountId: account.id) {
38                              selectAccount(accountId: nil)
39                          }
40                      }
41                  )
42              }
43          }
44      }
45
46      private func selectAccount(accountId: Int?) {
47          selectedAccountId = accountId
48      }
49  }
```

Código 5.1: Implementação do AccountsListCoordinator

5.5 Desafios e Limitações

A adoção da *Build-block Architecture* no desenvolvimento do *Follows Audit* trouxe melhorias significativas na organização do código, modularização e escalabilidade do aplicativo. No entanto, sua implementação também apresentou desafios e limitações que exigiram adaptações ao longo do processo.

A modularização por meio do *Swift Package Manager* (SPM) permitiu a criação de componentes reutilizáveis, mas também **aumentou a complexidade da organização do projeto**. Com múltiplos módulos independentes, a comunicação entre eles precisou ser cuidadosamente planejada para evitar dependências circulares e garantir um acoplamento mínimo entre os componentes.

Além disso, a adoção de uma arquitetura modular baseada em MVVM e *Coordinator* exige um tempo maior de adaptação por parte dos desenvolvedores não familiarizados com essa abordagem. A necessidade de compreender a estrutura modular, a separação de responsabilidades e a comunicação entre módulos representa um desafio inicial.

5.6 Conclusões do Capítulo

Neste capítulo, foram aplicados todos os conceitos da *Build-block Architecture* em uma aplicação real, o ***Follows Audit***. Foi realizada a análise do aplicativo e seus recursos, sua arquitetura e *design*, além de descrever a implementação e a modularização usando a abordagem proposta.

A análise realizada permitiu identificar a flexibilidade da arquitetura proposta, que se revela crucial para a gestão dos complexos fluxos de navegação e funcionalidades presentes em aplicativos de alta complexidade. Um dos aspectos centrais abordados foi a modularização do código, que facilita não apenas a manutenção e a evolução do aplicativo ao longo do tempo, mas também promove uma maior testabilidade dos diversos módulos que compõem a aplicação.

A divisão do código em bibliotecas como *Core*, *Modules* e *FAFoundation*, por exemplo, permite que a implementação de novos recursos seja realizada de maneira incremental e organizada, sem comprometer a estrutura existente do aplicativo. Essa característica é relevante em projetos onde as demandas e os requisitos podem evoluir rapidamente.

Ainda, o uso do *coordinator* para gerenciar fluxos de navegação demonstrou ser uma estratégia eficaz para garantir uma navegação escalável e intuitiva.

6 Conclusões

A *Build-block Architecture* propõe uma abordagem altamente modular para o desenvolvimento de aplicativos para as plataformas da *Apple*, estruturando aplicações por meio da composição de blocos reutilizáveis. A comparação com arquiteturas tradicionais permite avaliar as vantagens e limitações da metodologia, destacando aspectos como desacoplamento entre módulos, flexibilidade na manutenção do código e tempo de *build*.

Os resultados obtidos reforçam a viabilidade da BbA como uma solução eficaz para modularização, ao mesmo tempo em que apontam possibilidades de aprimoramento e adaptações para contextos específicos.

6.1 Avaliação da *Build-block Architecture*

A eficácia de uma arquitetura de *software* está diretamente relacionada à sua capacidade de lidar com a complexidade do projeto, garantindo flexibilidade, testabilidade e facilidade de manutenção. A *Build-block Architecture*, ao adotar um "modelo" modular baseado na composição de blocos independentes, apresenta características que impactam diretamente esses aspectos.

Além disso, dois outros princípios essenciais da engenharia de *software* foram levados em conta no desenvolvimento da BbA, como **alta coesão** e **baixo acoplamento**, fatores determinantes para a escalabilidade e sustentabilidade da aplicação a longo prazo.

6.1.1 Flexibilidade em Projetos Complexos

Arquiteturas modulares são fundamentais para projetos de grande escala, nos quais a separação de responsabilidades e a reutilização de componentes desempenham um papel essencial. A *Build-block Architecture* facilita a organização do código ao estruturar a aplicação em módulos coesos, reduzindo dependências desnecessárias e permitindo a evolução do sistema sem comprometer sua estabilidade.

A abordagem baseada em alta coesão e baixo acoplamento favorece a flexibilidade,

pois permite que cada módulo seja desenvolvido, atualizado ou substituído de forma independente, útil em cenários onde requisitos de negócio sofrem alterações frequentes, garantindo maior adaptabilidade e menor impacto em termos de retrabalho.

6.1.2 Testabilidade e Manutenção a Longo Prazo

A modularização promovida pela *Build-block Architecture* contribui significativamente para a testabilidade do sistema, uma vez que a separação clara entre os módulos permite a implementação de testes unitários mais eficientes e de forma isolada, reduzindo efeitos colaterais indesejados.

O uso do *Swift Package Manager* (SPM) fortalece essa abordagem ao possibilitar a separação dos testes por módulo, garantindo que cada parte do sistema seja testada de forma independente. Essa estratégia reduz o tempo de execução dos testes, pois evita a necessidade de rodar verificações desnecessárias em módulos que não sofreram modificações.

6.2 Próximos Passos

Dando continuidade a este estudo, uma das etapas fundamentais para a consolidação da *Build-block Architecture* no desenvolvimento de aplicativos nativos para as plataformas da *Apple* é a automação de testes e sua integração a um *pipeline* contínuo de entrega e implantação. A implementação de uma estratégia de *Continuous Integration/Continuous Deployment* (CI/CD) possibilita a validação automatizada da aplicação a cada modificação no código, garantindo uma maior confiabilidade e reduzindo riscos.

A integração do *Xcode Cloud* (APPLE, 2021), uma solução nativa do *Xcode* para automação de *build*, testes e distribuição, pode melhorar ainda mais esse processo. O uso do *Xcode Cloud* possibilita a execução de testes automatizados em diferentes configurações de ambiente, permitindo a detecção de falhas em estágios iniciais do desenvolvimento. Ao incorporar essa ferramenta ao pipeline de CI/CD, é possível estabelecer um fluxo contínuo de validação e implantação, reduzindo a complexidade do gerenciamento manual e acelerando a entrega de novas versões do aplicativo.

Além disso, uma pesquisa com desenvolvedores usando a BbA na prática se faz de suma importância no sentido de validar essa abordagem em um ambiente real.

6.3 Considerações Finais

A *Build-block Architecture* se apresenta como uma abordagem eficiente para o desenvolvimento de aplicativos nativos para as plataformas da *Apple*, proporcionando uma alta coesão e baixo acoplamento entre os módulos do projeto, favorecendo não apenas a organização do código, mas também a melhoria de aspectos como flexibilidade, testabilidade e manutenção a longo prazo à medida que a complexidade vai aumentando. A adoção do SPM como ferramenta nativa para a modularização reforça esses benefícios, permitindo a separação clara entre os componentes e facilitando a evolução da aplicação sem comprometer sua integridade.

Além disso, a viabilidade dessa abordagem pode ser ampliada com a integração de práticas de CI/CD, garantindo que cada módulo seja testado e validado de maneira automatizada. O uso de ferramentas como *Xcode Cloud* possibilita a execução de testes contínuos e a distribuição eficiente do app, consolidando uma esteira de desenvolvimento mais previsível e confiável.

Dessa forma, este estudo contribui para o entendimento da *Build-block Architecture* como uma solução viável para projetos que demandam escalabilidade e organização modular. Trabalhos futuros podem explorar sua aplicação em diferentes domínios e tecnologias, bem como investigar aprimoramentos na esteira de desenvolvimento, a fim de maximizar seus benefícios na prática.

Bibliografia

APPLE. *Apple Announces iPhone 2.0 Software Beta*. 2008. Acessado em: 4 de novembro de 2024. Disponível em: ⟨https://www.apple.com/newsroom/2008/03/06Apple-Announces-iPhone-2-0-Software-Beta/⟩.

APPLE. *UIKit*. 2008. Acessado em: 4 de novembro de 2024. Disponível em: ⟨https://developer.apple.com/documentation/uikit/⟩.

APPLE. *Cocoa Fundamentals Guide*. 2013. Acessado em: 4 de novembro de 2024. Disponível em: ⟨https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/Introduction/Introduction.html⟩.

APPLE. *Swift*. 2014. Acessado em: 7 de novembro de 2024. Disponível em: ⟨https://www.swift.org/documentation/⟩.

APPLE. *Binding*. 2019. Acessado em: 7 de novembro de 2024. Disponível em: ⟨https://developer.apple.com/documentation/SwiftUI/Binding⟩.

APPLE. *Managing user interface state*. 2019. Acessado em: 4 de novembro de 2024. Disponível em: ⟨https://developer.apple.com/documentation/swiftui/managing-user-interface-state⟩.

APPLE. *State*. 2019. Acessado em: 7 de novembro de 2024. Disponível em: ⟨https://developer.apple.com/documentation/SwiftUI/State⟩.

APPLE. *SwiftUI*. 2019. Acessado em: 4 de novembro de 2024. Disponível em: ⟨https://developer.apple.com/documentation/swiftui⟩.

APPLE. *Xcode Cloud*. 2021. Acessado em: 8 de novembro de 2024. Disponível em: ⟨https://developer.apple.com/xcode-cloud/⟩.

APPLE. *NavigationStack*. 2022. Acessado em: 4 de novembro de 2024. Disponível em: ⟨https://developer.apple.com/documentation/swiftui/navigationstack/⟩.

APPLE. *Modality*. 2023. Acessado em: 4 de novembro de 2024. Disponível em: ⟨https://developer.apple.com/design/human-interface-guidelines/modality⟩.

APPLE. *Platform State of the Union*. 2024. Acessado em: 11 de dezembro de 2024. Disponível em: ⟨https://www.youtube.com/watch?v=YJZ5YcMsgD4&t=1576s⟩.

APPLE. *Swift Package Manager*. 2024. Acessado em: 6 de novembro de 2024. Disponível em: ⟨https://github.com/swiftlang/swift-package-manager⟩.

APPLE. *WWDC 2024*. 2024. Acessado em: 11 de dezembro de 2024. Disponível em: ⟨https://developer.apple.com/wwdc24/⟩.

BABAYEV, N. The challenges and possibilities of using viper architecture in swiftui. 2023. Disponível em: ⟨https://www.theseus.fi/bitstream/handle/10024/806261/Babayev_Natig.pdf?sequence=2&isAllowed=y⟩.

BARTLETT, J. *Modular Architecture for Apps*. 2024. Acessado em: 6 de novembro de 2024. Disponível em: ⟨https://blog.jacobstechtavern.com/p/modular-architecture-for-apps⟩.

DOBREAN, D.; DIOSAN, L. An analysis system for mobile applications mvc software architectures. In: *ICSOFT*. [S.l.: s.n.], 2019. p. 178–185.

FORSANKER, A. *Evaluating the Code Quality of iOS Applications Generated by Large Language Models*. 2024.

FUKSA, M.; SPETH, S.; BECKER, S. Mvvm revisited: Exploring design variants of the model-view-viewmodel pattern. 2025.

INDRAWAN DANA SULISTYO KUSUMO, S. Y. P. D. *ANALYSIS OF THE IMPLEMENTATION OF MVVM ARCHITECTURE PATTERN ON PERFORMANCE OF IOS MOBILE-BASED APPLICATIONS*. 2023. Acessado em: 8 de novembro de 2024. Disponível em: ⟨https://jurnal.stkippgritulungagung.ac.id/index.php/jipi/article/view/3293/1469⟩.

LEE, A. V. D. *MVVM: An architectural coding pattern to structure SwiftUI Views*. 2024. Acessado em: 4 de novembro de 2024. Disponível em: ⟨https://www.avanderlee.com/swiftui/mvvm-architectural-coding-pattern-to-structure-views/⟩.

MANFERDINI, M. *MVVM in SwiftUI for a Better Architecture [with Example]*. 2023. Acessado em: 4 de novembro de 2024. Disponível em: ⟨https://matteomanferdini.com/mvvm-swiftui/⟩.

MARTIN, R. C. *Clean architecture*. [S.l.]: Prentice Hall, 2017.

NAUMOV, A. *Clean Architecture for SwiftUI*. 2019. Acessado em: 8 de novembro de 2024. Disponível em: ⟨https://nalexn.github.io/clean-architecture-swiftui/⟩.

NILSSON, S. *Implementation of a Continuous Integration and Continuous Delivery System for Cross-Platform Mobile Application Development*. 2016.

OLIVEIRA, M. R. R. de. Spatialized live refactoring. 2024.

RAMACHANDRAPPA, N. C. Solid design principles in software engineering. *International Journal of Computer Trends and Technology*, v. 72, n. 9, p. 18–23, 2024.

REACTIVEX. *RXSwift*. 2016. Acessado em: 8 de novembro de 2024. Disponível em: ⟨https://github.com/ReactiveX/RxSwift⟩.

SANTOS, C. M. d. C.; PIMENTA, C. A. d. M.; NOBRE, M. R. C. The pico strategy for the research question construction and evidence search. *Revista Latino-Americana de Enfermagem*, Escola de Enfermagem de Ribeirão Preto / Universidade de São Paulo, v. 15, n. 3, p. 508–511, Jun 2007. ISSN 0104-1169. Disponível em: ⟨https://doi.org/10.1590/S0104-11692007000300023⟩.

SILVA, R. P. d. Métodos para revisão e mapeamento sistemático da literatura. 2009. Disponível em: ⟨https://www.researchgate.net/publication/303497814_Metodos_para_Revisao_e_Mapeamento_Sistematico_da_Literatura_Methods_for_Systematic_Literature_Reviews_and_Systematic_Mapping_Studies⟩.