



**UNIVERSIDADE ESTADUAL DO CEARÁ
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

YAGO SABOIA FELIX FROTA

**UM ESTUDO COMPARATIVO ENTRE OS DIFERENTES ESTILOS DE
CONSTRUÇÃO DA INTERFACE DE USUÁRIO PARA DISPOSITIVOS IOS**

FORTALEZA – CEARÁ

2021

YAGO SABOIA FELIX FROTA

UM ESTUDO COMPARATIVO ENTRE OS DIFERENTES ESTILOS DE
CONSTRUÇÃO DA INTERFACE DE USUÁRIO PARA DISPOSITIVOS IOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientador: Prof. Dr. Paulo Henrique Mendes Maia

FORTALEZA – CEARÁ

2021

**Dados Internacionais de Catalogação na Publicação
Universidade Estadual do Ceará
Sistema de Bibliotecas**

Frota, Yago Saboia Felix.

Um estudo comparativo entre os diferentes estilos de construção da interface de usuário para dispositivos iOS [recurso eletrônico] / Yago Saboia Felix Frota. - 2021.

81 f. : il.

Trabalho de conclusão de curso (GRADUAÇÃO) – Universidade Estadual do Ceará, Centro de Ciências e Tecnologia, Curso de Ciência da Computação, Fortaleza, 2021.

Orientação: Prof. Dr. Paulo Henrique Mendes Maia.

1. Dispositivos móveis. 2. Interface de usuário. 3. UIKit. 4. SwiftUI. 5. Construção de Interface. I. Título.

YAGO SABOIA FELIX FROTA

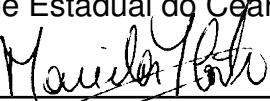
UM ESTUDO COMPARATIVO ENTRE OS DIFERENTES ESTILOS DE
CONSTRUÇÃO DA INTERFACE DE USUÁRIO PARA DISPOSITIVOS IOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em: 22/02/2021

BANCA EXAMINADORA


Prof. Dr. Paulo Henrique Mendes Maia (Orientador)
Centro de Ciências e Tecnologia - CCT
Universidade Estadual do Ceará – UECE


Profa. Dra. Mariela Inés Cortés
Centro de Ciências e Tecnologia - CCT
Universidade Estadual do Ceará – UECE


Profa. Dra. Carla Ilane Moreira Bezerra
Campus de Quixadá
Universidade Federal do Ceará - UFC

A minha mãe Milene Saboia Felix. A meu
pai Gerardo Frota Junior. Aos meus avós
Lenira de Saboia Felix e Miguel Felix Neto.

AGRADECIMENTOS

Ao Prof. Dr. Paulo Henrique Mendes Maia, pela sua orientação. Aos professores e desenvolvedores que tomaram um tempo para responder o questionário utilizado no trabalho. Aos professores e colegas de graduação que contribuíram imensamente para minha formação e para minha jornada dentro e fora da UECE. Ao meu irmão Yuri Saboia Felix Frota pela atenção e ajuda para responder dúvidas técnicas e resolver problemas. A minha família que permitiu com que eu chegassem até aqui, graças a todo o suporte oferecido.

“The best way to predict the future is to create it.”

Peter Drucker

RESUMO

Dispositivos móveis estão se tornando cada vez mais indispensáveis na sociedade, sendo cada vez mais presentes em atividades pessoais e sociais. A interface de usuário em um dispositivo móvel é uma parte essencial para a comunicação desse dispositivo com um usuário, pois é por ela que mostra o conteúdo que se quer apresentar. Na criação de uma interface em uma etapa do desenvolvimento, é necessário escolher o estilo de construção a se utilizar e, sem um conhecimento amplo, o desenvolvedor acaba tendo dificuldade em escolher a mais eficaz. Neste contexto, este trabalho realiza um estudo comparativo sobre os diferentes estilos de construção de interface para dispositivos iOS, de modo a auxiliar o desenvolvedor a compreender e melhorar a estrutura de um projeto ou de uma aplicação móvel. Com esses estilos sendo a Storyboard, o View Code e o SwiftUI. Para a análise comparativa foram utilizadas características do modelo de qualidade ISO/IEC 25010 pertencente ao modelo SQuaRE. Essas características, em conjunto com um caso de uso, foram utilizadas para a elaboração de um questionário, o qual foi respondido por desenvolvedores com experiência na área, e para a coleta de dados a partir de ferramentas do programa Xcode. Os dados coletados de cada estilos foram analisados e utilizados para a análise comparativa dos estilos, encontrando assim o melhor estilo para cada característica. Como resultado final, foi encontrado que o estilo View Code foi o melhor estilo de construção de interface, obtendo os melhores resultados na maioria das características.

Palavras-chave: Dispositivos móveis. Interface de usuário. UIKit. SwiftUI. Construção de Interface.

ABSTRACT

Mobile devices are becoming increasingly indispensable in society, being increasingly present in personal and social activities. The user interface on a mobile device is an essential part for the communication of that device with a user, as it is through it that shows the content that one wants to present. When creating an interface at a development stage, it is necessary to choose the style of construction to be used and, without extensive knowledge, the developer ends up having difficulty in choosing the most effective one. The following research conducts a comparative study on the different interface construction styles for iOS devices, in order to help the developer understand and improve the structure of a project or mobile application. With these styles being Storyboard, View Code and SwiftUI. For the comparative analysis, characteristics of the ISO / IEC 25010 quality model belonging to the SQuaRE model were used. These characteristics, together with a use case, were used to design a questionnaire, which was answered by developers with experience in the area, and to collect data from Xcode program tools. The data collected from each style was analyzed and used for the comparative analysis of the styles, thus finding the best style for each characteristic. As a final result, it was found that the View Code style was the best interface construction style, obtaining the best results in most of the characteristics.

Keywords: Mobile Devices. Interface. Interface Builder. UIKit. SwiftUI. Visual Editor

LISTA DE ILUSTRAÇÕES

Figura 1 – Arquitetura iOS	20
Figura 2 – Área Básica de trabalho do Xcode	22
Figura 3 – Padrão MVC da Apple	24
Figura 4 – Imagem exemplo de uma Storyboard	26
Figura 5 – Exemplo de interface	26
Figura 6 – Interface de detalhes de uma paisagem	31
Figura 7 – Interface de adicionar objetos em SwiftUI	32
Figura 8 – Interface de atributos de objetos em SwiftUI	32
Figura 9 – Modelo de qualidade de produto - ISO 25010	34
Figura 10 – Interface do Exemplo	42
Figura 11 – Fluxo de telas no Interface Builder	43
Figura 12 – Constraints da UIImageView	44
Figura 13 – Experiência do usuário	54
Figura 14 – Experiência do usuário	54
Figura 15 – Estilo mais utilizado pelo usuário	54
Figura 16 – Qualidade Funcional	55
Figura 17 – Interface da Storyboard	56
Figura 18 – Interface do View Code	56
Figura 19 – Interface do SwiftUI	57
Figura 20 – Expressão funcional	58
Figura 21 – Comportamento temporal 1	60
Figura 22 – Comportamento temporal 2	60
Figura 23 – Uso de recursos 1	63
Figura 24 – Uso de recursos 2	63
Figura 25 – Coexistência - Pizza	65
Figura 26 – Coexistência - Colunas	65
Figura 27 – Capacidade de reconhecer sua adequação	66
Figura 28 – Capacidade de aprendizagem	67
Figura 29 – Operabilidade	68
Figura 30 – Proteção contra erro de usuário	69
Figura 31 – Maturidade	70

Figura 32 – Capacidade de recuperação 1	70
Figura 33 – Capacidade de recuperação 2	70
Figura 34 – Modularidade 1	71
Figura 35 – Modularidade 2	72
Figura 36 – Modularidade 3	72
Figura 37 – Reusabilidade 1	73
Figura 38 – Reusabilidade 2	74
Figura 39 – Reusabilidade 3	74
Figura 40 – Analisabilidade	75
Figura 41 – Modificabilidade 1	76
Figura 42 – Modificabilidade 2	76

LISTA DE TABELAS

Tabela 1 – Tabela de subcaracterísticas	38
Tabela 2 – Tabela de tempo de compilação em segundos	58
Tabela 3 – Tabela de tempo para rodar o simulador em segundos	59
Tabela 4 – Tabela de tempo de resposta	60
Tabela 5 – Tabela de Uso de recursos - Movimentação na tabela	61
Tabela 6 – Tabela de Uso de recursos - Movimentação entre telas	62
Tabela 7 – Tabela de capacidade	64
Tabela 8 – Tabela do resultado em relação aos dados obtidos	77

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Exemplo de View Controller	27
Código-fonte 2 – Exemplo de TableViewController	28
Código-fonte 3 – UILabel na sintaxe imperativa	29
Código-fonte 4 – Text na sintaxe declarativa	29
Código-fonte 5 – List em SwiftUI	30
Código-fonte 6 – ContentPreview em SwiftUI	30
Código-fonte 7 – Protocolo utilizado no View Code	45
Código-fonte 8 – Inicialização da interface em View Code	45
Código-fonte 9 – Inicializando a InfoView a partir da chamada da LoadView()	46
Código-fonte 10–Criação de uma UITableView por View Code	47
Código-fonte 11–Utilização do protocolo CodeView e atribuição de constraints	47
Código-fonte 12–Inicialização da interface em SwiftUI	48
Código-fonte 13–Utilização da List no exemplo SwiftUI	49
Código-fonte 14–Inicialização de objetos e atribuição de modificadores	49
Código-fonte 15–Objeto Landmark	50
Código-fonte 16–Função de leitura do landmarkData	51

LISTA DE ABREVIATURAS E SIGLAS

API	Interface de Programação de Aplicação
GQM	<i>Goal-Question-Metric</i>
IEC	<i>International Electrotechnical Commission</i>
ISO	<i>International Organization of Standardization</i>
MVC	<i>Model-View-Controller</i>
WWDC	<i>Apple Worldwide Developers Conference</i>
XML	<i>eXtensible Markup Language</i>

SUMÁRIO

1	INTRODUÇÃO	16
1.1	OBJETIVOS	18
1.1.1	Objetivo Geral	18
1.1.2	Objetivos Específicos	18
1.2	ESTRUTURA DA MONOGRAFIA	18
2	FUNDAMENTAÇÃO TEÓRICA	20
2.1	INTRODUÇÃO AO AMBIENTE IOS E SUAS FERRAMENTAS	20
2.1.1	Linguagem Swift	21
2.1.2	Xcode	22
2.2	INTERFACE GRÁFICA	23
2.2.1	Framework UIKit	23
2.2.1.1	Padrão MVC	24
2.2.1.2	Storyboards	25
2.2.1.3	View Code	27
2.2.2	Sintaxe declarativa	29
2.2.3	SwiftUI	30
2.3	O CONJUNTO DE PADRÕES INTERNACIONAIS SQUARE	33
3	TRABALHOS RELACIONADOS	36
4	METODOLOGIA	37
4.1	INTRODUÇÃO DAS MÉTRICAS SELECIONADAS PARA A METODOLOGIA	37
4.2	EXPLICAÇÃO DO EXEMPLO UTILIZADO	41
4.2.1	O exemplo	41
4.2.2	Exemplo em Storyboard	43
4.2.3	Exemplo em View Code	44
4.2.4	Exemplo em SwiftUI	48
4.2.5	Elementos em comum nos três estilos	50
4.2.6	Etapas da metodologia	52
5	RESULTADOS OBTIDOS	53

5.1	APRESENTAÇÃO DOS DADOS COLETADOS E ANÁLISE COMPARATIVA DOS ESTILOS	53
5.2	SÍNTESE DOS RESULTADOS	77
5.3	AMEAÇAS À VALIDADES	78
6	CONSIDERAÇÕES FINAIS	79
	REFERÊNCIAS	80

1 INTRODUÇÃO

Dispositivos móveis estão cada vez mais presentes na sociedade. Segundo a GSMA (*Global System for Mobile Communications*), existem mais de 5,2 bilhões dispositivos móveis conectados a internet, significando que mais de 60% da população mundial possui algum tipo de dispositivo móvel, sendo no Brasil mais de 65% da população (GSMA, 2020).

Os dispositivos móveis estão cada vez mais sendo indispensáveis para o dia a dia. Um celular não é apenas mais um meio de se conectar a internet, mas sim a principal maneira para se comunicar, trabalhar, realizar negócios, se divertir e até para a locomoção (TIDWELL et al., 2020). Esses dispositivos, em sua a maioria, utilizam de uma interface gráfica, sendo essa interface essencialmente um meio de comunicação entre o usuário e o produto, onde este visa realizar tarefas para atender os objetivos do usuário (MCKAY, 2013).

Segundo Pressman (2006), a Interface é um meio de comunicação efetivo entre o ser humano e o computador. A interface gráfica de usuário (GUI, do inglês *Graphical User Interface*), sobre a qual essa pesquisa se trata, utiliza-se de elementos gráficos como botões, textos, tabelas, entre outros, para facilitar a comunicação com o usuário (PRESSMAN, 2006).

Sendo assim, a interface de usuário é essencial para a comunicação desses dispositivos com o usuário que a utiliza, tendo uma grande influência na permanência do usuário em, por exemplo, um aplicativo, visto que a primeira impressão é de extrema importância para conquistar o usuário.

A interface no ambiente iOS pode ser desenvolvida de diferentes formas, não existindo um padrão que deve ser firmemente seguido pelo desenvolvedor (BARKER, 2020). Porém, cada um dos diferentes estilos de construção de interface possui vantagens e desvantagens, competindo ao desenvolvedor optar por qual estilo utilizar em cada interface.

É comum que o estilo de construção de interface em um projeto seja escondido a partir do gosto pessoal do desenvolvedor ou de padrões e definições da própria empresa. Isso não quer dizer que não é possível implementar diferentes estilos em um só projeto, porém, sem decidir um padrão para quando utilizar cada estilo, acaba por dificultar a movimentação, manutenção e reusabilidade destes códigos.

Os estilos a serem estudados e avaliados nesta pesquisa serão:

- a) Construção a partir de código escrito (View code), na qual toda a interface é feita a partir da escrita em código, possuindo assim mais controle e conhecimento do funcionamento dessas interface.
- b) Utilização de construtores de interface como Storyboards, que podem ser utilizados para desenvolvimento iOS. Storyboard é uma representação visual da interface do usuário de um aplicativo iOS, mostrando telas de conteúdo e as conexões entre essas telas. O programa Xcode fornece um editor visual para Storyboards, pelo qual é possível criar e projetar interfaces de usuário adicionando telas, botões, tabela, textos, dentre outros.
Além disso, uma Storyboard permite conectar uma tela ao objeto do controlador e gerenciar a transferência de dados entre telas.
- c) E por fim a construção de interface a partir de sintaxes declarativas, como é o caso do SwiftUI, onde é possível apenas declarar como a interface é para ser feita e o seu comportamento.

A falta de um padrão ou um conhecimento amplo sobre os mais diferentes estilos de construção de interface acaba por dificultar a escolha de qual estilo utilizar no desenvolvimento de um certo projeto. Esta pesquisa visa fazer um estudo sobre os diferentes estilos de construção de interface e, a partir de uma comparação entre eles, busca ajudar o desenvolvedor na escolha de um estilo para o seu caso em específico, facilitando a compreensão e melhorando a estrutura de um projeto ou aplicativo.

Para esse estudo será realizada uma análise comparativa entre os três estilos de construção de interface no ambiente iOS. Serão utilizadas características do modelo de qualidade ISO/IEC 25010 para a elaboração de um questionário e para a coleta de dados relacionados a performance.

Os dados coletados de cada um dos estilos serão analisados e utilizados para uma análise comparativa dos estilos, com o objetivo de encontrar o melhor estilo para cada características. Por fim, será feita uma síntese desses resultados, encontrando o melhor estilo de construção de interface no geral, em relação a ser o melhor em maioria das características.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Este trabalho tem como objetivo comparar a utilização e aplicação de diferentes estilos de construção de interface no ambiente iOS a partir de critérios pre-selecionados com o intuito de encontrar o estilo que apresenta a maior performance e adequabilidade.

1.1.2 Objetivos Específicos

Visando atingir o objetivo principal, alguns objetivos específicos são requeridos:

- a) identificar os estilos existentes de criação de interface no ambiente iOS.
- b) Analisar as diferenças dos estilos no processo de criação da interface.
- c) Realizar experimentos a partir de características selecionadas do modelo de qualidade ISO/IEC 25010 e de casos de uso, para avaliar os estilos de construção de interface.
- d) Comparar os estilos de construção de interface a partir do resultado dos experimentos realizados, para assim, chegar a uma conclusão das vantagens e desvantagens de cada um desses estilos.

1.2 ESTRUTURA DA MONOGRAFIA

No Capítulo 2 é apresentada a base teórica desta monografia. São abordados os temas ambiente iOS, Interface Gráfica e a metodologia SQuaRE. No ambiente iOS são apresentados a linguagem de desenvolvimento escolhida para esse projeto, o Swift, e a ferramenta utilizada para o desenvolvimento e a coleta de dados relacionados a performance, o Xcode. No Tópico da interface Gráfica, foram apresentados o *framework* UIKit, com seus estilos de construção de interface que são A Storyboard e o View Code, a sintaxe declarativa e o Framework SwiftUI, que também é um dos estilos de construção de interface.

O Capítulo 3 apresenta a metodologia deste trabalho. Nessa etapa foi decidido os critérios a serem utilizados na análise comparativa a partir do modelo de

qualidade ISO/IEC 25010, e esses critérios foram utilizados para elaborar questões e chegar a métricas, que foram utilizadas para a coleta de dados. Também foi apresentado o caso de uso a ser utilizado para a comparação dos três estilos.

O Capítulo 4 apresenta os resultados obtidos a partir da coleta de dados. Neste tópico é feita uma análise de cada um dos critérios decididos no capítulo 3, e realizado uma síntese, mostrando o melhor estilo para cada critério e qual foi o melhor no geral.

O Capítulo 5 apresenta os trabalhos relacionados que foram utilizados para a compreensão do estado da arte nesta área de pesquisa.

Por fim, no Capítulo 6 foram feitas as considerações finais do trabalho, apresentando um resumo geral e indicando o melhor estilo em relação a metodologia e aos dados coletados. Neste capítulo também foi mostrado as dificuldades desse trabalho e sugestões futuras para a evolução da pesquisa.

2 FUNDAMENTAÇÃO TEÓRICA

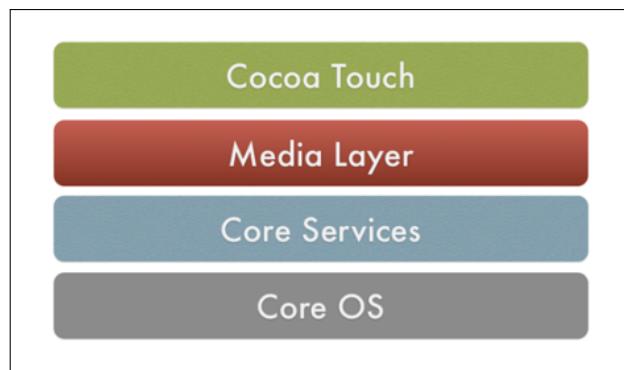
Este capítulo apresenta a base teórica utilizada para o desenvolvimento do trabalho. São abordados os temas Ambiente iOS, a interface gráfica e a metodologia SQuaRE.

2.1 INTRODUÇÃO AO AMBIENTE IOS E SUAS FERRAMENTAS

O iOS é um sistema operacional móvel desenvolvido e apresentado em 2007 pela Apple Inc. Inicialmente chamado iPhone OS, esse sistema é baseado no sistema operacional UNIX e foi feito especialmente para seu dispositivo móvel Iphone, sendo posteriormente aplicado em outros hardwares da empresa (APPLE, 2021a).

A arquitetura iOS é semelhante à arquitetura do macOS e é dividida em 4 camadas:

Figura 1 – Arquitetura iOS



Fonte – imagem obtida em (NAAVEN, 2020).

- Cocoa Touch: onde são encontrados os principais *frameworks* utilizados pelos desenvolvedores. É responsável por gerenciar serviços de notificação, multitarefa e diversos outros serviços de alto nível.
- Media: é responsável pelos serviços multimídias, sendo projetada para facilitar a implementação de arquivos multimídia. Nele estão presentes *frameworks* essenciais como o UIKit, um *framework* de grande importância para a criação de gráficos e animações.
- Core Services: possui os serviços fundamentais do sistema como leitura e escrita de arquivos, chamadas, banco de dados, entre outros.

- Core OS: gerencia segurança, bateria, memória e outros serviços de baixo nível. Utilizado pelos desenvolvedores quando é necessário lidar explicitamente com segurança ou com comunicação de hardwares externos.

Atualmente iOS está em sua versão 13 e com novas funcionalidades, como é o caso do novo modo escuro, melhorias no aspecto de privacidade e em seu desempenho.

2.1.1 Linguagem Swift

Em 2014, a linguagem Swift foi lançada na conferência de desenvolvedores anual da Apple WWDC, tornando-se, a partir dessa época, a linguagem oficial de programação da Apple. Criada com o propósito de substituir a antiga linguagem utilizada para desenvolvimento de aplicativos, o Objective-C, Swift é uma linguagem mais fácil de aprender e utilizar, sendo uma linguagem mais competitiva entre as novas linguagens de programação (HOFFMAN, 2017).

Algumas das melhorias que o Swift trouxe em comparação com a linguagem Objective-C são:

- Mais rápido: segundo a Apple, algoritmos de busca podem ser até 2,6 vezes mais rápido que em Objective-C (APPLE, 2014).
- Mais fácil: Swift possui uma sintaxe eficiente e simples de aprender, com semelhanças a muitas linguagens atuais e possuindo um amplo acervo de tutoriais disponibilizados no próprio site da Apple.
- Mais seguro: além de não possuir ponteiros, Swift foi criado pensando em segurança, produzindo um erro de compilador cada vez que se é escrito algum código incorreto.
- Melhor gerenciamento de memória: devido à ótima estabilidade e uma ótima utilização de técnicas para gerenciamento de memória.
- Mais fácil de manter: Swift possui apenas um arquivo por classe, ao contrário de Objective-C que possui dois arquivos por classe.
- *Open source*: inicialmente introduzida em 2014 como uma linguagem proprietária, Swift se tornou uma linguagem de código aberto em dezembro de 2015, sobre a licença Apache License 2.0.

A linguagem Swift pode perfeitamente ser incorporada em códigos já existentes em Objective-C. Boa parte dos *frameworks* e implementações construídos em

Objective-C também podem ser utilizados em códigos Swift, como é o caso do Cocoa e do Cocoa Touch que possuem respectivamente os principais *frameworks* para desenvolvimento iOS e macOS.

Sendo uma linguagem extremamente poderosa e intuitiva, Swift é utilizada não apenas para iOS, mas também para o desenvolvimento de outros os produtos da Apple, como macOS, watchOS, tvOS, dentre outros.

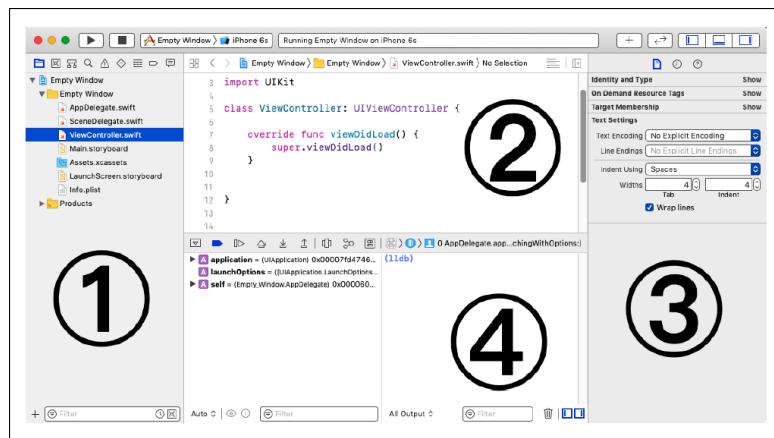
2.1.2 Xcode

Lançado em 2003, Xcode é a principal ferramenta do macOS utilizada para se desenvolver aplicativos macOS, iOS, iPadOS, watchOS e tvOS. Nele é possível executar todas as etapas de criação de um software iOS, como escrever códigos, construir interfaces de usuários e testes de software (ZVEIBIL, 2003).

O Xcode dispõe de diversas outras ferramentas para auxiliar no desenvolvimento de um aplicativo, como é o caso do Simulator e do Debugger. Simulator é uma ferramenta que simula os dispositivos da Apple, permitindo analisar o seu funcionamento em diferentes dispositivos. Com a ferramenta Debugger é possível analisar o percurso feito até um defeito, o tipo de defeito, dicas de como resolver estes defeitos, além de possibilitar a análise de elementos no meio da execução do código.

Na Figura 2 temos a área básica de trabalho do Xcode, a qual pode ser dividida em 4 partes:

Figura 2 – Área Básica de trabalho do Xcode



Fonte – Imagem obtida no livro (NEUBURG, 2019), página 454

1. Navigator Area: campo responsável pela navegação e administração de um

projeto.

2. Editor Area: campo responsável pela edição do código de um arquivo.
3. Utility Area: campo responsável pela visualização de atributos do arquivo.
4. Debug Area: campo responsável por mostrar as informações em um ponto específico quando a aplicação está sendo executada.

O padrão não apenas define a função de cada uma das camadas na aplicação como também como elas se comunicam entre si. Cada uma das camadas é separada da outra a partir de fronteiras abstratas e comunicam-se com outras a partir dessas fronteiras.

2.2 INTERFACE GRÁFICA

Mesmo sendo possível criar todo um projeto de aplicativo apenas por código, ainda existem diversas ferramentas criadas com o intuito de agilizar o desenvolvimento de um projeto, possibilitando até mesmo que poucas ou até nenhuma linha de código.

Nesta etapa serão apresentados alguns dos principais elementos que facilitam na criação de interfaces gráficas: O *framework* UIKit e as Storyboards. Também será discutido sobre a criação da interface a partir de View Code.

2.2.1 *Framework* UIKit

O *framework* UIKit fornece os principais objetos que o desenvolvedor vai precisar para construir aplicativos iOS. Ele provê os objetos necessários para mostrar o conteúdo na tela, interagir com o conteúdo e administrar a comunicação desses objetos com o sistema. O *framework* oferece diversos recursos como suporte para animação, suporte para documentos, desenhos e impressão, busca, entre outros (APPLE, 2021d).

Para o UIKit, existem duas formas de implementação da interfaces, uma sendo a partir da utilização do Interface Builder, que é o caso da Storyboard e a outra sendo a escrita da interface programaticamente, que é o caso do View Code.

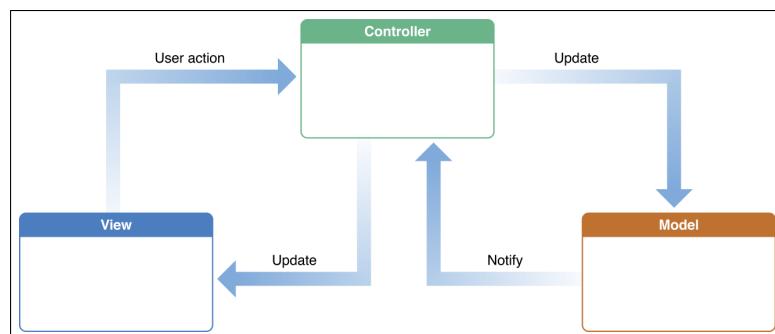
A arquitetura comumente utilizada na UIKit é a arquitetura MVC, onde a Controller é responsável por consolidar a comunicação entre a View e o Model, observando mudanças que ocorrem na view que atualizam o Model e mudanças no Model que atualizam a View.

2.2.1.1 Padrão MVC

O padrão Modelo-Visão-Controle (do inglês, *Model-View-Controller*), conhecido por sua sigla MVC, é um dos padrões de arquitetura de software mais utilizados no desenvolvimento iOS. Ele consiste em dividir os objetos da aplicação em 3 camadas: Modelo (do inglês, *Model*), Visão (do inglês, *View*) e Controle (do inglês, *Controller*) (KACZMAREK et al., 2019).

O modelo do padrão MVC aplicado pela Apple possui algumas diferenças se comparado com o padrão MVC tradicional, sendo as camadas Visão e Controle não independentes, uma vez que no modelo convencional existe uma comunicação entre o Modelo e com a Visão (APPLE, 2021b). A Figura 3 mostra a comunicação de camadas no padrão MVC aplicado pela Apple, onde cada camada é responsável por:

Figura 3 – Padrão MVC da Apple



Fonte – Imagem obtida na documentação da Apple (2021b).

- **Modelo:** camada responsável por encapsular os dados específicos de uma aplicação e define a lógica e computação de manipulação desses dados, além de ser facilmente reutilizável.
- **Visão:** camada onde a aplicação mostra os dados para o usuário, sendo responsável também por enviar as ações do usuário para o controle.
- **Controle:** camada intermediária entre modelos e visões, sendo responsável por informar a camada da visão se existiu mudanças no modelo ou vice-versa

O padrão não apenas define a função de cada uma das camadas na aplicação como também como elas se comunicam entre si. Cada uma das camadas é separada da outra a partir de fronteiras abstratas e comunicam-se com outras a partir dessas fronteiras.

2.2.1.2 Storyboards

Arquivos de interface de usuário da ferramenta Interface Builder podem ter suas extensões do tipo *.storyboard* ou *.xib*. O conteúdo desses tipos de arquivos é armazenado em um formato XML, no qual ao ser compilado pelo Xcode é transformado em um tipo de arquivo binário chamado Nib. Em tempo de execução arquivos Nibs são carregados e executados para criarem Cenas (do inglês, *Views*), também chamadas de telas da interface.

Views são os blocos fundamentais para a interface de usuário, e a classe *UIView* define o comportamento que é comum para todas as *views*. Resumidamente, a classe *UIView* é uma classe concreta a qual se pode inicializar, adicionar ou modificar propriedades e adicionar classes *subviews* (APPLE, 2021e).

Storyboard é um recurso introduzido no iOS 5 com o objetivo de agilizar o desenvolvimento de interfaces de usuário em um aplicativo iOS. Uma Storyboard é uma representação visual da interface de usuário em uma aplicação iOS, mostrando as telas com seus conteúdos e como é feita a conexão entre essas telas (APPLE, 2018).

Storyboards são compostas de uma sequência de telas, onde cada uma dessas telas representa uma *view controller* e sua respectiva *view*. A partir da ferramenta Interface Builder presente no Xcode é possível adicionar em uma *view* objetos como botões, listas, textos, entre outros com um simples arrastar, sendo possível denominar sua localização na tela a partir de limites (do inglês, *Constraints*).

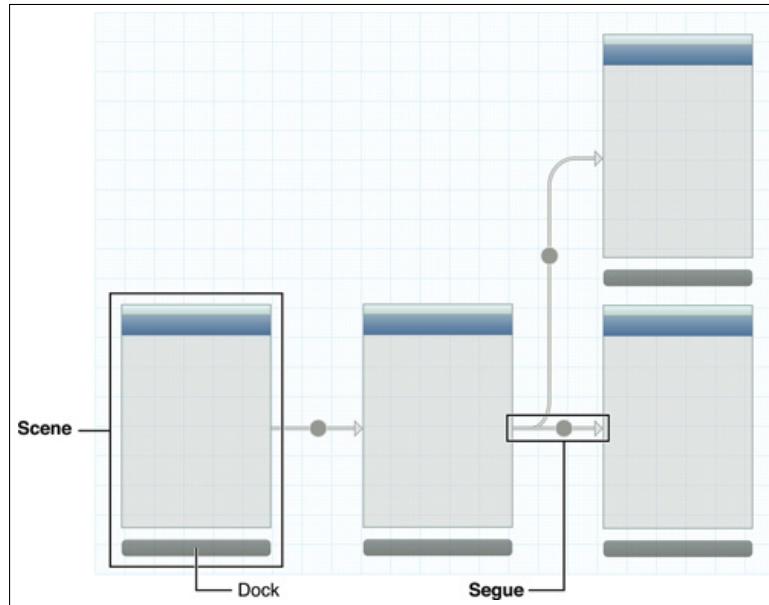
A partir da Figura 4 podemos notar que as *controllers* da *view* são conectadas por um conjunto de objetos chamados de *segue*s que representam a transição entre duas *view controllers*. Assim, em apenas um arquivo é possível projetar múltiplas interfaces de usuário e decidir como elas vão interagir entre si.

Cada *view* possui um *dock*, o qual é responsável por mostrar ícones com referência a objetos top-level como a *view controller* acoplada a esta *view*. Ele é principalmente utilizado para criar conexões de entrada e saída entre a *view* e a *view controller*.

A Figura 5 apresenta um exemplo de desenvolvimento em uma Storyboard. Esse exemplo é um aplicativo simples que ao clicar no botão é feita a troca do texto de maneira aleatória.

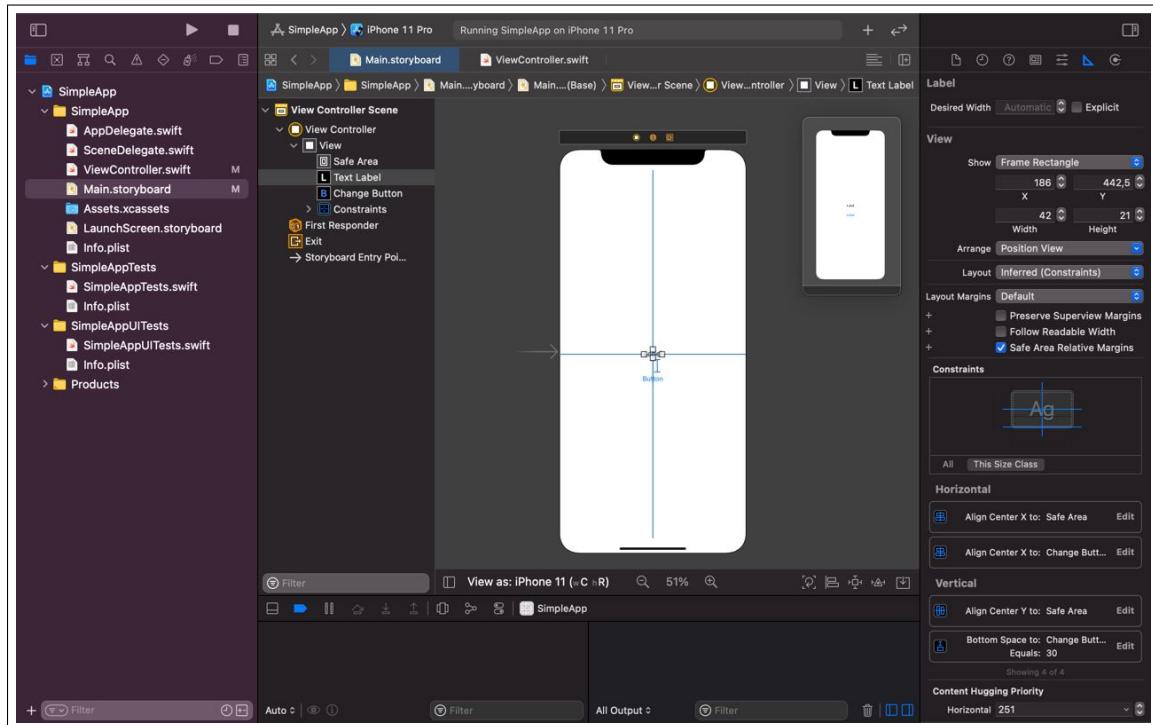
O exemplo possui uma *UIView* com subviews *UILabel*, responsável pelo

Figura 4 – Imagem exemplo de uma Storyboard



Fonte – Imagem obtida na documentação da Apple (2021c).

Figura 5 – Exemplo de interface



Fonte – Imagem obtida pelo autor diretamente do programa Xcode

texto, e UIButton, responsável pela modificação do estado da UILabel. Como podemos ver na imagem, a UILabel tem *constraints* de alinhamento ao centro horizontal e vertical em relação a sua *view* superior(no caso, a UIView). O UIButton possui *constraints* em relação a UILabel, sendo estas uma *constraint* de alinhamento vertical e um

espaçamento de 30 pixels em relação a UILabel.

Ambos o UILabel e o UIButton são conectados a ViewController da *view*, como visto no Código 1.

Código-fonte 1 – Exemplo de View Controller

```

1   import UIKit
2
3   class ViewController: UIViewController {
4
5       @IBOutlet weak var textLabel: UILabel!
6
7       @IBOutlet weak var changeButton: UIButton!
8
9       let names = ["Luiz", "José", "Bruno", "Jéssica", "Fernanda", "Luisa"]
10
11      override func viewDidLoad() {
12          super.viewDidLoad()
13
14          textLabel.text = names[0]
15          changeButton.setTitle("Trocar nome", for: .normal)
16          // Do any additional setup after loading the view.
17      }
18
19      @IBAction func change(_ sender: Any) {
20          textLabel.text = names.randomElement()
21      }
22
23 }
```

No Código 1 na classe *ViewController* é possível observar as mudanças feitas em cada uma das *subviews*. A UILabel inicialmente é atribuída o elemento inicial do *array* de nomes e o UIButton tem seu título modificado para “Trocar Nome”. Além disso, existe uma função conectada ao UIButton, onde quando se é pressionado, é escolhido um elemento aleatório do *array* de nomes e atribuído ao texto da UILabel.

2.2.1.3 View Code

Conhecida como *View Coding*, esse estilo visa criar interfaces de usuário programaticamente, isto é, sem nenhuma ajuda de um construtor de interfaces e

utilizando-se somente da linguagem Swift, permitindo apenas com o código criar telas, seus objetos e conexões para a sua comunicação.

Código-fonte 2 – Exemplo de UITableViewController

```
1 import UIKit
2
3 class ViewController: UIViewController,
4     UITableViewDelegate {
5
6     lazy var tableView = UITableView()
7
8     override func viewDidLoad() {
9         super.viewDidLoad()
10        view.addSubview(tableView)
11        tableView.translatesAutoresizingMaskIntoConstraints =
12            false
13
14        NSLayoutConstraint.activate([
15            tableView.topAnchor.constraint(equalTo: view.
16                safeAreaLayoutGuide.topAnchor),
17            tableView.leadingAnchor.constraint(equalTo:
18                view.leadingAnchor),
19            tableView.trailingAnchor.constraint(equalTo:
20                view.trailingAnchor),
21            tableView.bottomAnchor.constraint(equalTo: view
22                .bottomAnchor)
23        ])
24    }
25}
```

No Código 2 temos um exemplo de criação de uma tabela em View Code, onde em seu inicio é feita a inicialização da tabela: UITableView. Após a UITableViewController ser adicionada a memória é chamada a função *viewDidLoad*, esta adiciona como *Subview* a UITableView na UIView principal e são adicionadas as *constraints* da UITableView em relação a UIView principal, responsáveis pelo posicionamento da tabela em relação a tela.

2.2.2 Sintaxe declarativa

Antes de falar sobre a nova linguagem de criação de interface SwiftUI, é necessário explicar a sua diferença para a linguagem Swift e para o UIKit. Enquanto a linguagem Swift no UIKit se utiliza de uma sintaxe imperativa para a criação de telas, o SwiftUI veio para inovar utilizando-se de uma sintaxe declarativa.

A sintaxe declarativa é um paradigma de programa, um estilo de construção de estruturas e elementos de um programa, que expressa a lógica da computação sem descrever o seu fluxo de controle. De maneira mais clara, a sintaxe declarativa é uma maneira de descrever o código que se deseja escrever sem se preocupar com a forma que vai ser implementado (BARKER, 2020).

Código-fonte 3 – UILabel na sintaxe imperativa

```

1 let.textLabel = UILabel(frame: CGRect(x:0, y:0, width:100,
2 height:100))
3 textLabel.text = "Example"
4 textLabel.textColor = UIColor.black
5 textLabel.backgroundColor = UIColor.blue
6 textLabel.font = UIFont(name: "Arial", size: 24)
7 self.view.addSubview(textLabel)

```

O Código 3 é feito em UIKit utilizando sintaxe imperativa. Esse código cria uma UILabel e atribui os seus atributos. Neste caso é necessário explicar passo a passo como o objeto labelText tem que ser criado, quais seus atributos e por fim adicioná-lo como elemento da UIView superior.

Código-fonte 4 – Text na sintaxe declarativa

```

1 Text("Example")
2 .color(.black)
3 .background(Color.blue)
4 .font(.largeTitle)

```

Já o Código 4 é feito em SwiftUI utilizando sintaxe declarativa. Não é necessário declarar todos os passos para mostrar o texto, simplesmente é pedido um objeto Texto e adicionado como ele deve ser a partir de modificadores.

2.2.3 SwiftUI

SwiftUI é um *framework* relativamente novo escrito em Swift para Swift. Esse possui uma sintaxe declarativa, ao contrário de seus antecessores que utilizavam sintaxe imperativa, que permite uma maior fluidez e um código mais legível e fácil de ser escrito (BARKER, 2020).

Em contrapartida a arquitetura MVC utilizada pelo *framework* UIKit, o SwiftUI se utiliza de um modelo baseado em views e estados (do inglês, *states*). No caso de ser atribuído um `@state` a uma propriedade, o SwiftUI vai monitorar essa propriedade e, caso ocorra uma alteração nessa propriedade, seja por uma ação do usuário ou um evento externo, será automaticamente atualizada às partes afetadas da interface. Como resultado, o *framework* faz quase todo o trabalho das View Controllers.

Como exemplo temos o Código 5, onde caso ocorra alguma mudança na coleção de dados, `landmarkData`, a lista vai automaticamente ser atualizada na interface, sem a necessidade de fazer uma chamada de atualização como ocorre na UIKit com UITableViews.

Código-fonte 5 – List em SwiftUI

```
1 List(landmarkData) { landmark in
2     LandmarkRow(landmark: landmark)
3 }
```

Uma das principais ferramentas apresentadas com SwiftUI foi a SwiftUI Previews, que permite o usuário analisar a interface sem a necessidade de compilar e executar o aplicativo no simulador. Para a utilização da Preview, é necessário implementar uma estrutura dentro da sua classe SwiftUI.

Código-fonte 6 – ContentPreview em SwiftUI

```
1 struct ContentView_Previews: PreviewProvider {
2     static var previews: some View {
3         LandmarkDetail(landmark: landmarkData[0])
4         ForEach(["iPhone SE", "iPhone XS Max"], id: \.self)
5             { deviceName in
6                 LandmarkList()
7                     .previewDevice(PreviewDevice(rawValue:
8                         deviceName))
```

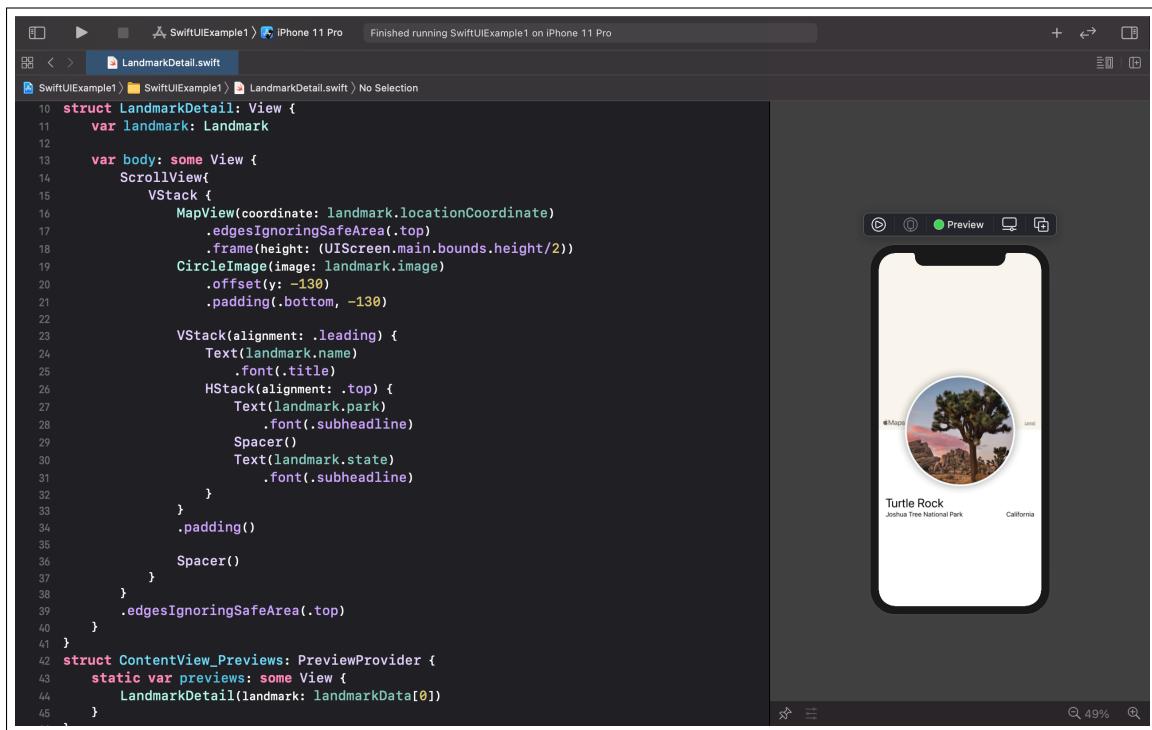
```

7         }
8     }
9 }
```

O Código 6 representa a estrutura da Preview na imagem 6, nesta estrutura é injetada um conjunto de dados na linha 3, com estes dados aparecendo na Preview.

Além disso, a Preview possibilita observar a interface em múltiplos dispositivos ao mesmo tempo, como é o caso do *ForEach* na linha 4 a 8 do Código 6, que mostra as Previews de Iphone SE e Iphone XS Max.

Figura 6 – Interface de detalhes de uma paisagem



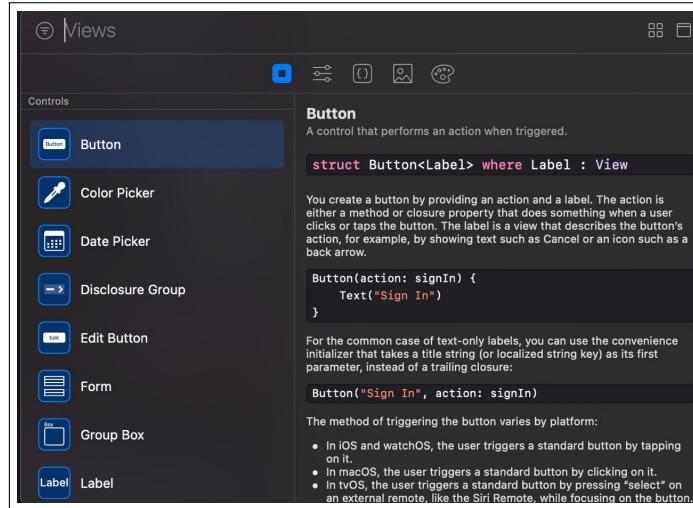
Fonte – Imagem obtida pelo autor diretamente do programa Xcode

Na Figura 6 temos uma *View* utilizada futuramente neste trabalho. Nesta imagem é possível observar como o Xcode fica dividido quando se está desenvolvendo em SwiftUI. Na parte esquerda da imagem temos o código para a criação da *view*, em SwiftUI, e na direita temos a Preview, mostrando como a interface foi montada a partir do código.

A Preview possui ferramentas semelhantes ao Interface Builder, que possibilitem adicionar e fazer alterações em objetos na Preview, que serão automaticamente adicionados no código do SwiftUI. A Figura 7 apresenta a interface com as *views*

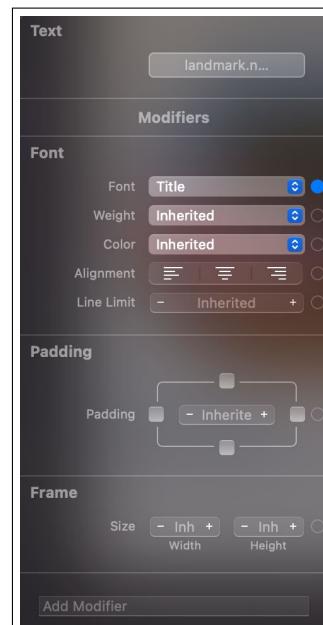
possíveis a adicionar e a Figura 8 mostra as modificações possíveis em uma *View* do tipo Text.

Figura 7 – Interface de adicionar objetos em SwiftUI



Fonte – Interface obtida pelo autor diretamente do programa Xcode

Figura 8 – Interface de atributos de objetos em SwiftUI



Fonte – Interface obtida pelo autor diretamente do programa Xcode

Ao contrário do desenvolvimento utilizando UIKit que utiliza diferentes objetos

para o desenvolvimento em diferentes dispositivos, os objetos do SwiftUI funcionam em todos os seus dispositivos. Assim, todas as funcionalidades de SwiftUI só necessitam ser construídas uma vez para suportar múltiplos dispositivos, eliminando a necessidade de escrever o código múltiplas vezes.

Esse *framework* está disponível a partir da versão 11 do Xcode e exige a versão macOS 10.15 Catalina ou maior para ser utilizado. Em relação a dispositivos é necessário ser pelo menos as versões iOS 13, iPadOS 13, macOS 15 e watchOS 6.

2.3 O CONJUNTO DE PADRÕES INTERNACIONAIS SQUARE

A qualidade de um produto, juntamente com a qualidade do processo, são essenciais para o desenvolvimento de software. O modelo de qualidade determina quais as características de qualidade serão levadas em conta quando for avaliar um produto de software, sendo importante que as características de qualidade sejam especificadas, medidas e avaliadas sempre o quanto possível utilizando-se de métodos amplamente reconhecidos (ISO, 2011).

A série de normas ISO/IEC 25000, também conhecido como SQuaRE, tem o objetivo de criar um *framework* para a avaliação da qualidade do produto de software. Ele é o resultado da evolução de diversas normas e consiste em 5 divisões:

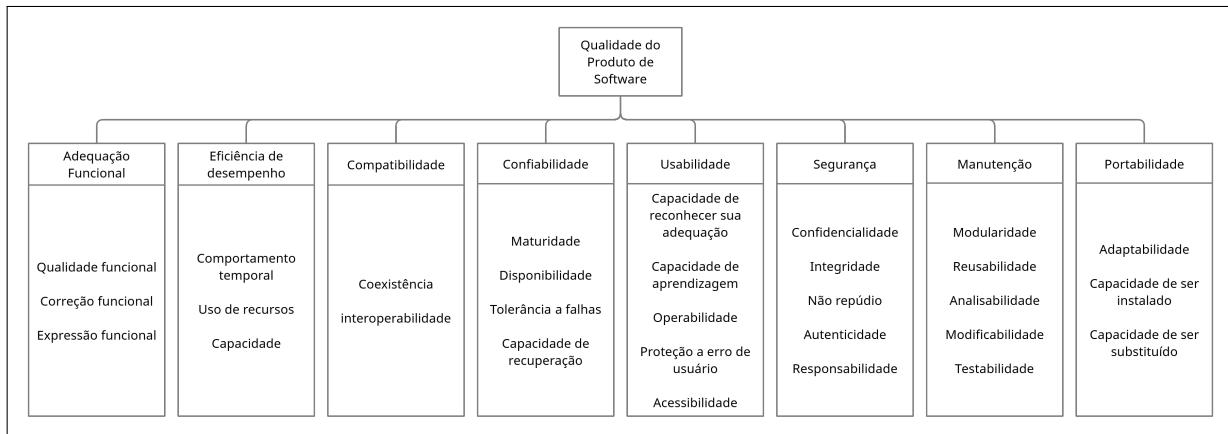
- ISO/IEC 2500n – Divisão gestão de qualidade;
- ISO/IEC 2501n – Divisão modelo de qualidade;
- ISO/IEC 2502n – Divisão medição da qualidade;
- ISO/IEC 2503n – Divisão requisitos de qualidade;
- ISO/IEC 2504n – Divisão avaliação da qualidade.

O modelo de qualidade de produto definido na ISO/IEC 25010 é composto de 8 características de qualidade, conforme a Figura 9:

Para esse estudo foram utilizadas 6 das 8 características do modelo de qualidade ISO/IEC 25010. As características segurança e portabilidade não são avaliadas nesse estudo, devido a essas características não dependerem do estilo no desenvolvimento iOS e sim do programa Xcode.

Em relação as subcaracterísticas das características selecionadas, algumas não foram aplicadas neste estudo. A interoperabilidade, tolerância a falhas e a availabilidade de suas respectivas características, compatibilidade, usabilidade e confiabilidade,

Figura 9 – Modelo de qualidade de produto - ISO 25010



Fonte – Imagem adaptada da documentação ISO (2011).

não foram utilizadas no estudo, devido a essa funcionalidade não depender do estilo e sim do programa Xcode. Já a estética da interface de usuário, da característica usabilidade, não foi utilizada, pois a análise comparativa é feita a partir da mesma interface em todos os estilos.

Sendo assim, as características e subcaracterísticas utilizadas nesse estudo são apresentadas a seguir, sendo elas adaptadas para a aplicação no estudo.

1. Adequação funcional:

- Qualidade funcional: grau em que o estilo atende todos os requisitos necessários para o desenvolvimento da interface.
- Correção funcional: grau em que o estilo atende o mínimo de requisitos de precisão da interface.
- Expressão funcional: grau em que o estilo fornece ferramentas e funcionalidades que facilitam atender os requisitos da interface.

2. Eficiência de desempenho:

- Comportamento temporal: grau em que o estilo atende os requisitos de processamento e de tempo de resposta.
- Uso de recursos: grau em que o estilo atende os requisitos de recursos quando desempenhando a sua função.
- Capacidade: grau em que o estilo atende os requisitos máximos de elementos da interface.

3. Compatibilidade:

- Coexistência: grau em que o estilo permite o desenvolvimento com múltiplos

desenvolvedores sem impactar negativamente a interface como um todo.

4. Usabilidade:

- a) Capacidade de reconhecer sua adequação: grau em que o usuário consegue reconhecer se o estilo é adequado para as necessidades da interface.
- b) Capacidade de aprendizagem: grau em que o usuário consegue aprender a utilizar o estilo de maneira eficaz e eficiente.
- c) Operabilidade: grau de facilidade em utilizar o estilo de forma eficiente e eficaz quando aprendido.
- d) Proteção a erro de usuário: grau em que o estilo protege o usuário de cometer erros.
- e) Acessibilidade: grau em que o estilo pode ser utilizado por pessoas com as mais ampla gama de características para resolver o exemplo.

5. Confiabilidade:

- a) Maturidade: grau em que o estilo atende as necessidades de confiabilidade em operação normal.
- b) Capacidade de recuperação: grau em que caso ocorra uma falha ou interrupção no sistema ou programa, seja possível recuperar todos os dados e voltar ao estado que estava antes do problema.

6. Manutenção:

- a) Modularidade: grau em que ao fazer modificações em um componente da interface não ocorra nenhum impacto ou tenha um impacto mínimo em outros componentes.
- b) Reusabilidade: grau em que o estilo possibilita reutilizar elementos da interface em outras partes da interface.
- c) Analisabilidade: grau em que o estilo possibilita identificar a causa e onde ocorreu uma falha.
- d) Modificabilidade: grau em que o estilo possibilita fazer modificações na interface de uma maneira eficiente e eficaz sem introduzir defeitos ou diminuir a qualidade da interface.
- e) Testabilidade: grau em que o estilo permite criar um conjunto de critérios de testes para avaliar a eficiência e eficácia da interface e se esses critérios são atendidos.

3 TRABALHOS RELACIONADOS

Neste capítulo são apresentados os trabalhos relacionados, os quais foram utilizados para a compreensão do estado da arte nesta área de pesquisa.

Jessica et al. (2020) avaliam os *frameworks* UIKit e SwiftUI a partir de 4 aplicações diferentes desenvolvidas pelos autores, testando componentes disponíveis em ambos os *frameworks*, como listas, textos, imagens, botões e seletores.

Para o desenvolvimento dos exemplos, os autores optaram por não utilizar Storyboards e a ferramenta Interface Builder para a UIKit, construindo a interface apenas por View Code, para se obter um comparativo mais justo em relação a quantidade de linhas de código entre os *frameworks*.

Os autores concluem em seu artigo que ao testar os *frameworks* UIKit e SwiftUI em 4 aplicações móveis diferentes houve um aumento no uso de memória e uso de CPU e uma redução de linhas de código nas implementações feitas utilizando o *framework* SwiftUI.

Os objetivos propostos pelo artigo são semelhantes aos objetivos deste trabalho, com a diferença de que o artigo visa comparar apenas a performance da execução em aplicativos moveis. Já este trabalho visa analisar os diferentes estilos de construção gráfica a partir de diversos critérios, com um destes sendo a performance.

O artigo de Gustavo e Daniela (2019) compara duas ferramentas de desenvolvimento de aplicativos móveis, React Native e Ionic, a partir dos critérios: eficiência de desempenho, adequação funcional e usabilidade retirados da (ISO, 2011).

Os critérios decididos pelos autores, em conjunto com a abordagem métrica GQM (BASILI et al., 1992), foram utilizados para elaborar um questionário. Para este questionário foi proposto um cenário de desenvolvimento de um aplicativo com as mesmas funcionalidades nas duas ferramentas.

Os autores concluem que a ferramenta Ionic demonstrou ser a melhor opção para os desafios propostos em seu artigo pois possui uma avaliação melhor na maioria dos quesitos, com React Native apenas sendo considerado mais adequado para a subcaracterística Integridade Funcional, pertencente a ISO/IEC 25010.

O artigo aplica uma metodologia semelhante a aplicada neste trabalho, utilizando-se de características do modelo de qualidade funcional ISO/IEC 25010.

4 METODOLOGIA

O trabalho de natureza básica estratégica e abordagem qualitativa busca fazer um estudo de diferentes estilos de construções de interface em aplicativos móveis, com o intuito de serem comparados e, a partir desta comparação, prover indicações sobre quando é mais viável e eficaz utilizar certos estilos de construções de interface.

Para isto, será feita uma pesquisa descritiva e exploratória utilizando-se de técnicas de pesquisa bibliográfica, documental e de casos de uso para estudar os estilos e avaliar quais os parâmetros que serão utilizados para os métodos comparativos. Por fim, esses dados serão organizados e analisados para, a partir de um método indutivo, chegar a conclusão proposta.

4.1 INTRODUÇÃO DAS MÉTRICAS SELECIONADAS PARA A METODOLOGIA

Como dito no tópico do modelo SQuaRE, foram escolhidas 6 das 8 características presentes no modelo ISO/IEC 25010 para serem aplicadas nesse trabalho. Destas características, foram escolhidas 19 subcaracterísticas ao todo.

As subcaracterísticas foram utilizadas como objetivos de avaliação. Foram elaboradas questões para refinar e determinar se tais objetivos podem ser atingidos. Com as questões elaboradas, foram definidas as métricas necessárias para responder as questões. As métricas podem ser divididas em 2 tipos:

- Métricas Objetivas: dependem apenas do objeto e não de um ponto de vista, nesse caso essas métricas foram coletadas a partir de ferramentas disponibilizadas pelo programa Xcode.
- Métricas Subjetivas: Dependem do objeto e de um ponto de vista, nesse caso essas métricas foram coletadas a partir de um questionário.

A seguir a Tabela 1 apresenta as características e subcaracterísticas escolhidas como objetivos, as questões elaboradas para cada uma dessas sub características, e por fim uma coluna com as métricas utilizadas para o questionário, no caso das subjetivas, e para a coleta de dados a partir de ferramentas, no caso das objetivas.

Tabela 1 – Tabela de subcaracterísticas

Começo da tabela		
Subcaracterísticas	Questões	Métricas
Qualidade funcional	O estilo de interface atende todos os requisitos necessários para desenvolver o exemplo?	<p>Objetivas:</p> <p>Analisar se o estilo possui os requisitos necessários para o desenvolvimento do caso de uso, os quais são: Tabela com elementos customizáveis, elemento de mapa, imagem e texto, ferramenta para organizar elementos da interface.</p> <p>Subjetivas:</p> <p>Foi possível atender todos os requisitos do exemplo com as ferramentas e funcionalidades que o estilo dispõe?</p>
Correção Funcional	Com o estilo foi possível atender os mínimos requisitos de precisão de como a interface tem que ficar?	<p>Objetivas:</p> <p>Analisar a precisão da interface nesse estilo em comparação ao caso de uso.</p>
Expressão funcional	O estilo fornece ferramentas e funcionalidades que facilitam atender os requisitos do exemplo?	<p>Subjetivas:</p> <p>O quanto bom foi o estilo para conseguir atender os requisitos do exemplo? (Em comparação aos outros) (0 a 5)</p>
Comportamento temporal	Os requisitos de processamento atendem os determinados requerimentos?	<p>Objetivas:</p> <p>Analisar o tempo de compilação, tempo de resposta e tempo para rodar no simulador.</p>
	Ocorre algum gargalo no sistema devido a utilização do estilo e suas ferramentas?	<p>Subjetivas:</p> <p>Quando se está desenvolvendo é comum ocorrer algum gargalo no Xcode? ex: Xcode travando ao abrir “x” elemento.</p> <p>Quão comumente ocorre o problema citado na questão anterior? (0 a 5)</p>

Continuação da tabela 8		
Subcaracterísticas	Questões	Métricas
Uso de recursos	Quanto de recurso é utilizado pelo programa nos determinados casos?	<p>Objetivas:</p> <p>Analisar o consumo de CPU, memória e HD nos casos do simulador e na utilização do programa de desenvolvimento.</p> <p>Subjetivas:</p> <p>Quando se está desenvolvendo é comum ocorrer algum gargalo no sistema como um todo? ex: Xcode utilizando muito recurso e o computador travando</p> <p>Quão comumente ocorre o problema citado na questão anterior?(0 a 5)</p>
Capacidade	O número de itens impacta no funcionamento da tabela?	<p>Objetivas:</p> <p>Analisar o aumento de consumo de CPU, memória e HD em relação ao aumento de elementos na tabela de paisagens.</p>
	Qual o limite de itens possível na tabela?	<p>Objetivas:</p> <p>Analisar quantos elementos são necessários para chegar a um gargalo na tabela de paisagens</p>
Coexistência	O estilo permite o desenvolvimento com dois ou mais desenvolvedores modificando o mesmo código?	<p>Objetivas:</p> <p>Analisar se é possível desenvolver com mais de uma pessoa modificando o código.</p>
	Ao desenvolvedor com outra pessoa modificando o mesmo código de interface ocorre algum problema ou dificuldade?	<p>Subjetivas:</p> <p>Ao desenvolver com outra pessoa modificando o mesmo código de interface é comum ocorrer algum problema ou dificuldade?</p>
	Quão difícil normalmente é resolver esse problema?	<p>Subjetivas:</p> <p>Quão difícil é resolver geralmente esse problema caso ocorra?(0 a 5)</p> <p>Quais são os problemas mais comuns?</p>
Capacidade de reconhecer sua adequação	O usuário consegue reconhecer se o estilo é adequado para desenvolver a interface?	<p>Subjetivas:</p> <p>Analisando a interface, qual o estilo que você acredita ser mais adequado para desenvolver o exemplo?</p>
Capacidade de aprendizagem	O quão fácil é aprender o estilo?	<p>Subjetivas:</p> <p>Quão fácil foi aprender o estilo?(0 a 5)</p>

Continuação da tabela 8		
Subcaracterísticas	Questões	Métricas
Operabilidade	Após dominado o estilo, é fácil de utilizá-lo?	Subjetivas: O quão fácil de utilizar o estilo se torna quando aprendido?(0 a 5)
Proteção contra erros de usuário	O estilo possui ferramentas que avisam/impedem o usuário de cometer erros?	Subjetivas: Ao desenvolver a interface existem ferramentas que avisam ou impedem que você cometa um erro?
Acessibilidade	O estilo possui ferramentas para auxiliar pessoas com deficiências?	Objetivas: Analizar se existem ferramentas para pessoas com diferentes necessidades.
Maturidade	É confiável desenvolver neste estilo? Isto é, ele consegue desempenhar sua função, sem falhas ou avarias, dada certa uma tarefa sob certas condições em um período de tempo?	Subjetivas: Com base na sua experiência o estilo aparenta ser confiável? Isto é, é possível desenvolver o exemplo em um período de tempo aceitável?
Capacidade de recuperação	Quando ocorre algum problema com o programa é possível restaurar todos os dados afetados e voltar para o mesmo estado que estava antes?	Subjetivas: Já ocorreu algum problema do Xcode fechar sozinho ou alguma ferramenta parar de funcionar? Caso ocorra o problema da questão anterior foi possível restaurar todos os dados e voltar para o mesmo estado que estava antes?(0 a 5)
Modularidade	Fazer modificações em um componente da interface tem algum tipo de impacto em outros componentes?	Subjetivas: Já ocorreu de quando modificado um componente da interface gerar problemas em outros componentes?
	Caso seja feita uma modificação que afete negativamente outras partes da interface é fácil de resolver?	Subjetivas: Quão fácil é geralmente resolver esses tipos de problemas?(Em relação aos outros estilos)(0 a 5)
	É comum a modularização nesse estilo?	Subjetivas: Você usualmente modulariza componentes da interface nesse estilo?

Continuação da tabela 8		
Subcaracterísticas	Questões	Métricas
Reusabilidade	É possível reutilizar elementos feitos na interface em outras partes da interface?	Subjetivas: Você já reutilizou componentes da interface em outras partes da interface? Se sim, quão fácil foi reutilizar os componentes?(0 a 5)
	O estilo possui técnicas que facilitam a reutilização de elementos? E essas técnicas são estimuladas a serem utilizadas?	Subjetivas: Na sua opinião, o estilo induz a reutilização de componentes?
Analisabilidade	Caso ocorra um problema na interface após uma modificação é possível identificar a causa e onde ocorreu a falha?	Subjetivas: Ao modificar um elemento da interface que gerou um erro, quão fácil é identificar onde ocorreu e a causa?(0 a 5)
Modificabilidade	É possível fazer modificações da interface de uma maneira eficiente e eficaz sem introduzir defeitos ou diminuir a qualidade da interface?	Subjetivas: É possível fazer modificações da interface de uma maneira eficiente e eficaz sem introduzir defeitos ou diminuir a qualidade da interface? Quão prático você acha que é fazer modificações sem introduzir defeitos ou diminuir a qualidade da interface?(0 a 5)
Testabilidade	É possível fazer testes na interface em relação a critérios pré-estabelecidos para analisar se ela cumpre tais critérios?	Objetivas: Verificar se existem ferramentas que possibilitem criar parâmetros para teste e analisar se a interface em questão cumpre esses parâmetros.
Termino da tabela		

Fonte – Elaborado pelo autor

4.2 EXPLICAÇÃO DO EXEMPLO UTILIZADO

4.2.1 O exemplo

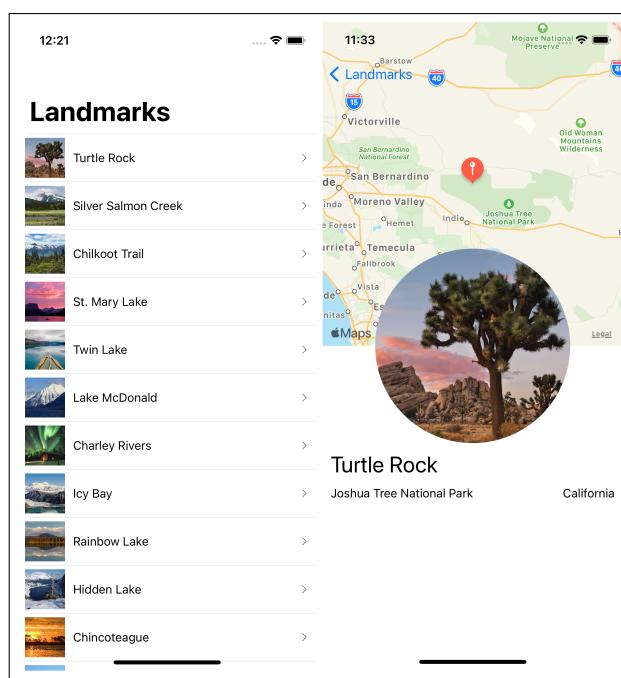
Com o intuito de comparar os três estilos de construção de interface gráficas foi desenvolvido um exemplo baseado no tutorial sobre SwiftUI “Building List and Navigations”, disponível na área SwiftUI Essentials no site da Apple (APPLE, 2020).

Esse exemplo contém elementos fundamentais no desenvolvimento de

aplicativos, como tabelas, imagens e textos, além de elementos mais específicos como mapas. Além disso, nele estão presentes estruturas de movimentação e comunicação entre múltiplas telas e manipulação de arquivos externos.

O exemplo se trata de um aplicativo chamado Paisagens, no qual é possível descobrir diversas novas paisagens e onde estão localizadas. Como visto na Figura 10, a tela à esquerda trata-se de uma tabela com diversas paisagens representadas por uma imagem do local e seu nome. Quando selecionada uma paisagem, é feita uma transição para uma nova tela, representada pela imagem da tela à direita, onde se encontram elementos de imagem, nome, estado e um mapa com sua localização da paisagem.

Figura 10 – Interface do Exemplo



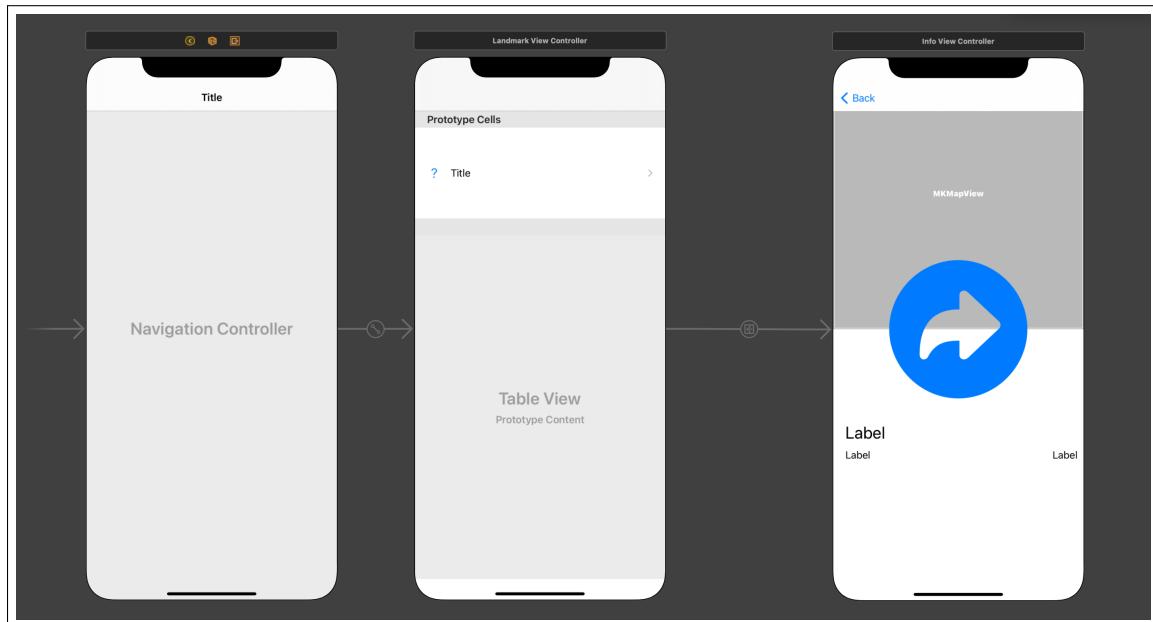
Fonte – Imagem obtida pelo autor diretamente do exemplo de View Code

Esse exemplo foi disponibilizado no questionário, para os respondentes utilizarem como base para responder o questionário, caso optem por não desenvolver o exemplo.

4.2.2 Exemplo em Storyboard

No exemplo em Storyboard, temos um arquivo chamado *main.storyboard*, o qual é responsável por armazenar a interface. Para a criação e modificação da interface é utilizado o Interface Builder, por onde são criadas todas as telas de interface. A Figura 11 apresenta todas as telas de interface seus fluxos.

Figura 11 – Fluxo de telas no Interface Builder



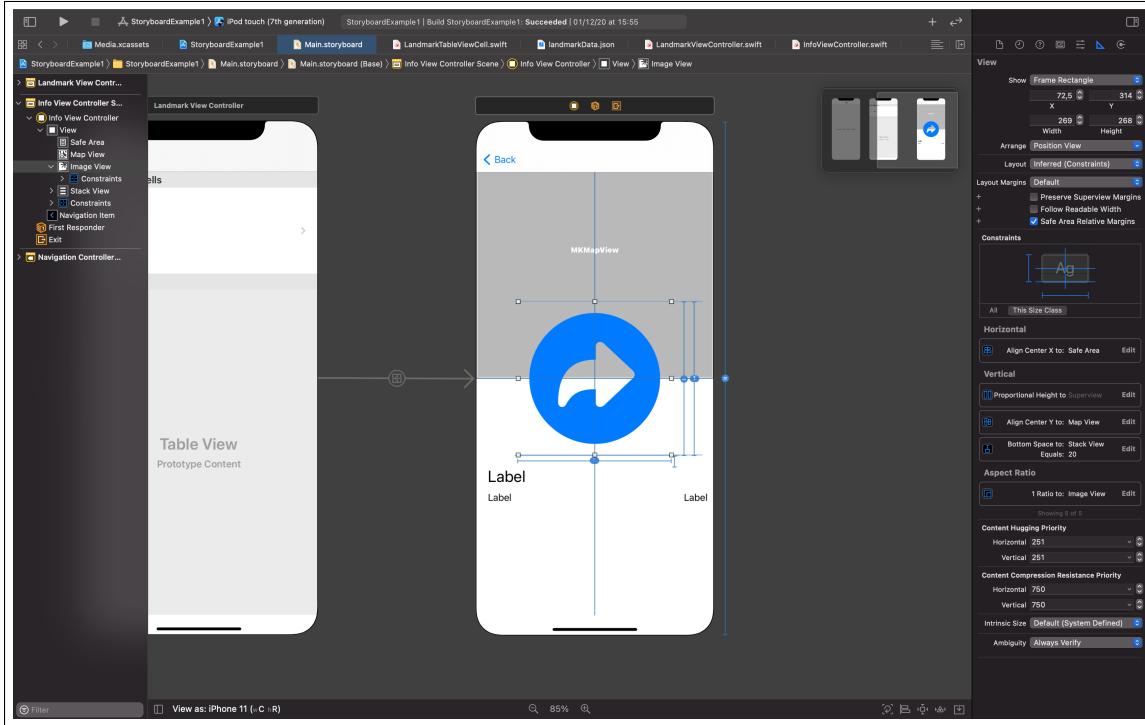
Fonte – Imagem obtida pelo autor diretamente do Xcode

A *NavigationController* é uma tela auxiliar responsável pela navegação entre *Views*, sendo um elemento responsável pela navegação dentro da aplicação. Ela representa uma estrutura de dados do tipo pilha que empilha e desempilha *controllers*, permitindo o usuário navegar para frente e voltar a *controller* anterior quando necessário.

A *LandmarkView* representa a tela das paisagens. Ela é composta de uma tabela dinâmica chamada *UITableView*, onde se encontram células que contém uma imagem (*UIImageView*) e um texto (*UILabel*) e que representam as paisagens na tabela.

A *InfoView* representa a tela de informações de uma paisagem. Ela é composta de um mapa (*MKMapView*), uma imagem (*UIImageView*) e uma pilha (*StackView*) na qual possui três textos (*UILabel*).

Figura 12 – Constraints da UIImageView



Fonte – Imagem obtida pelo autor diretamente do Xcode

Para o posicionamento dos elementos em cada tela se utiliza de Auto Layout. Ele é responsável por calcular a posição e o tamanho de todas as views em sua hierarquia, a partir da utilização de constraints. Como exemplo, temos as *constraints* de uma UIImageView apresentada na Figura 12, onde essa imagem possui as seguintes *constraints*:

- Alinhar o centro horizontal da imagem ao centro horizontal da InfoView.
- Alinhar o centro vertical da imagem ao centro vertical da InfoView.
- Altura proporcional a 30% do tamanho da InfoView.
- Largura proporcional a altura de si mesmo.
- A distância da sua parte de baixo em relação a StackView é igual a 20 pixels.

Ambas as LandmarkView e InfoView possuem *view controllers* responsáveis pelo gerenciamento da interface, preenchendo com dados e adicionando as interações de usuário necessárias.

4.2.3 Exemplo em View Code

Para criar uma interface programaticamente com sucesso, é necessário seguir uma certa ordem de passos. Para isso, foi criado um protocolo chamado

CodeView. Assim, é necessário que todas as *views* estejam conforme o protocolo. O conjunto de passos do protocolo é apresentado a seguir e sua aplicação no Código 7.

- Inicializar a hierarquia de *views*.
- Adicionar as *constraints* nas *views*.
- Fazer qualquer configuração adicional.

Código-fonte 7 – Protocolo utilizado no View Code

```

1 protocol CodeView {
2     func buildViewHierarchy()
3     func setupConstraints()
4     func setupAdditionalConfiguration()
5     func applyViewCode()
6 }
7
8 extension CodeView {
9     func applyViewCode(){
10    buildViewHierarchy()
11    setupConstraints()
12    setupAdditionalConfiguration()
13 }
14 }
```

Devido a não possuir uma storyboard, foi necessário inicializar a interface principal via código, a partir do arquivo SceneDelegate.swift. No arquivo foi criado uma UIWindow, que é um container básico responsável por receber as *views* da aplicação, e a UINavigationController, que recebe a LandmarkListViewController como a ViewController de origem, como mostrado no Código 8.

Código-fonte 8 – Inicialização da interface em View Code

```

1 class SceneDelegate: UIResponder, UIWindowSceneDelegate {
2
3     var window: UIWindow?
4
5     func scene(_ scene: UIScene, willConnectTo session:
6                 UISceneSession, options connectionOptions: UIScene.ConnectionOptions) {
7         if let windowScene = scene as? UIWindowScene{
8             let window = UIWindow(windowScene: windowScene)
9             let controller = LandmarkListViewController()
```

```

9     window.rootViewController =
10    UINavigationController(rootViewController:
11                           controller)
12    self.window = window
13    window.makeKeyAndVisible()
14
15 }
16 }
```

O exemplo em View Code possui 5 arquivos relacionados à interface:

- LandmarkListView.swift: arquivo responsável pela criação da *view* da tabela de paisagens, sendo composto por uma UITableView.
- LandmarkListViewController.swift: arquivo responsável pelo funcionamento da LandmarkListView.
- LandmarkListTableViewCell.swift: arquivo responsável pela criação da *view* das células da tabela.
- InfoView.swift: Arquivo responsável pela criação da *view* de informações de paisagem.
- InfoViewController.swift: arquivo responsável pelo funcionamento da InfoView.

Para toda *view controller* foi utilizada a função *loadView*, responsável por fazer a inicialização da UIView, como apresentado no Código 9.

Código-fonte 9 – Inicializando a InfoView a partir da chamada da LoadView()

```

1 override func loadView() {
2     if let selectedLandmark = landmark {
3         self.view = InfoView(frame: .zero,
4                               locationCoordinate: selectedLandmark.
5                               locationCoordinate, image: selectedLandmark.
6                               image, name: selectedLandmark.name, location
7                               : selectedLandmark.park, country:
8                               selectedLandmark.state)
9     }
10 }
```

Ao iniciar uma UIView, é realizada a inicialização de todas as suas *subviews*,

como pode ser observado na criação da UITableView pertencente a LandmarkListView no Código 10.

Código-fonte 10 – Criação de uma UITableView por View Code

```

1 lazy var tableView : UITableView = {
2     let view = UITableView(frame: .zero)
3     view.translatesAutoresizingMaskIntoConstraints =
4         false
5     return view
6 }()

```

No Código 11 temos um exemplo da utilização do protocolo CodeView na LandmarkListView, onde está sendo utilizado NSLayoutConstraint, responsável pelo posicionamento das subviews na landmarkListView.

Código-fonte 11 – Utilização do protocolo CodeView e atribuição de constraints

```

1 extension LandmarkListView : CodeView{
2     func buildViewHierarchy() {
3         self.addSubview(tableView)
4     }
5
6     func setupConstraints() {
7         NSLayoutConstraint.activate([
8             tableView.topAnchor.constraint(equalTo: self.
9                 safeAreaLayoutGuide.topAnchor),
10            tableView.leadingAnchor.constraint(equalTo:
11                self.leadingAnchor),
12            tableView.trailingAnchor.constraint(equalTo:
13                self.trailingAnchor),
14            tableView.bottomAnchor.constraint(equalTo: self
15                .bottomAnchor)
16        ])
17    }
18
19    func setupAdditionalConfiguration() {
20        tableView.backgroundColor = .white
21        tableView.rowHeight = 60
22        tableView.register(LandmarkListTableViewCell.self,
23            forCellReuseIdentifier: "
24                LandmarkListTableViewCell")
25    }

```

20

4.2.4 Exemplo em SwiftUI

SwiftUI se diferencia dos outros dois estilos devido à utilização de uma linguagem imperativa e um novo *framework* para o layout das telas.

O exemplo em SwiftUI possui 5 arquivos relacionados à interface:

- LandmarkList.swift: arquivo responsável pela criação e funcionamento da tabela de paisagens.
- LandmarkDetail.swift: arquivo responsável pela criação e funcionamento da tela de informações de paisagem.
- LandmarkRow.swift: arquivo responsável pela criação da *view* da célula da tabela de paisagens.
- MapView.swift: arquivo responsável pela configuração da *view* do mapa pertencente a tela de informações da paisagem.
- CircleImage.swift: arquivo responsável pela configuração da *view* da imagem pertencente a tela de informações da paisagem.

Para a inicialização da interface é necessário criar um WindowGroup, que funciona de maneira semelhante a UIWindow, o qual inicializa a tela de tabelas LandmarkList, como exibido no Código 12.

Código-fonte 12 – Inicialização da interface em SwiftUI

```

1 @main
2
3 struct SwiftUIExample1App: App {
4     var body: some Scene {
5         WindowGroup {
6             LandmarkList()
7         }
8     }
9 }
```

Para a tela de lista de paisagens, foi utilizada uma List, semelhante à UITableView, a List cria uma *view* que apresenta uma tabela com linhas de dados em uma única coluna.

List é a versão declarativa de UITableView em SwiftUI, retirando a necessidade do uso de *delegates* e *data sources* presentes no UIKit, requerendo apenas a passagem de um tipo View para compor a interface a partir de um *array*. Essa lista é representada pela estrutura LandmarkList no Código 13.

Código-fonte 13 – Utilização da List no exemplo SwiftUI

```

1 struct LandmarkList: View {
2     var body: some View {
3         NavigationView {
4             List(landmarkData) { landmark in
5                 NavigationLink(destination: LandmarkDetail(
6                     landmark: landmark)) {
7                     LandmarkRow(landmark: landmark)
8                 }
9             }
10            .navigationBarTitle(Text("Landmarks"))
11        }
12    }
13 }
```

Para a tela de detalhes de uma paisagem, foram utilizadas as estruturas VStack e HStack ,as quais organizam as *views* de maneira vertical e horizontal, estruturas de texto, mapa e imagem. Foram utilizados modificadores para fazer alterações nas *views* e alterar o posicionamento delas. O Código 14 representa a utilização desses elementos na tela de detalhes de uma paisagem no SwiftUI.

Código-fonte 14 – Inicialização de objetos e atribuição de modificadores

```

1 VStack {
2     MapView(coordinate: landmark.locationCoordinate)
3         .edgesIgnoringSafeArea(.top)
4         .frame(height: (UIScreen.main.bounds.height/2))
5     CircleImage(image: landmark.image)
6         .offset(y: -130)
7         .padding(.bottom, -130)
8
9     VStack(alignment: .leading) {
10         Text(landmark.name)
11             .font(.title)
12         HStack(alignment: .top) {
13             Text(landmark.park)
```

```

14         .font(.subheadline)
15     Spacer()
16     Text(landmark.state)
17         .font(.subheadline)
18     }
19 }
20 .padding()
21
22 Spacer()
23 }
```

4.2.5 Elementos em comum nos três estilos

Todos os exemplos possuem 3 arquivos idênticos e as mesmas 12 imagens de paisagens. Os arquivos em questão são:

- LandmarkData.json: arquivo que contém todas as paisagens a serem utilizadas.
- Landmark.swift: uma struct que representa uma paisagem com seus atributos.
- Data.swift: conjunto de funções responsáveis por ler o LandmarkData.json e transformar em um array (Do português: cadeia) de Landmarks.

Os códigos 15 e 16 representam, respectivamente, os arquivos Landmark.swift e data.swift.

Código-fonte 15 – Objeto Landmark

```

1 import Foundation
2 import UIKit
3 import CoreLocation
4
5 struct Landmark: Hashable, Codable, Identifiable {
6     var id: Int
7     var name: String
8     fileprivate var imageName: String
9     fileprivate var coordinates: Coordinates
10    var state: String
11    var park: String
12    var category: Category
13    var locationCoordinate: CLLocationCoordinate2D {
14        CLLocationCoordinate2D(
15            latitude: coordinates.latitude,
16            longitude: coordinates.longitude)
```

```

17 }
18 enum Category: String, CaseIterable, Codable, Hashable
19 {
20     case featured = "Featured"
21     case lakes = "Lakes"
22     case rivers = "Rivers"
23 }
24
25 extension Landmark {
26     var image: UIImage? {
27         get {
28             if let repoImage = UIImage(named: imageName){
29                 return repoImage
30             }else{ return nil }
31         }
32     }
33 }
34
35 struct Coordinates: Hashable, Codable {
36     var latitude: Double
37     var longitude: Double
38 }
```

Código-fonte 16 – Função de leitura do landmarkData

```

1 import CoreLocation
2
3 var landmarkData: [Landmark] = load("landmarkData.json")
4
5 func load<T: Decodable>(_ filename: String) -> T {
6     let data: Data
7     guard let file = Bundle.main.url(forResource: filename,
8         withExtension: nil)
9     else {
10         fatalError("Couldn't find \(filename) in main
11             bundle.")
12     }
13
14     do {
15         data = try Data(contentsOf: file)
16     } catch {
17         fatalError("Couldn't load \(filename) from main
```

```

                bundle:\n\(\error\)")
16
17
18 do {
19     let decoder = JSONDecoder()
20     return try decoder.decode(T.self, from: data)
21 } catch {
22     fatalError("Couldn't parse \(filename) as \(T.self)
23             :\n\(\error\)")
24 }
}

```

4.2.6 Etapas da metodologia

Com as métricas definidas e dividas, o estudo foi realizado em quatro etapas. A primeira foi uma etapa quantitativa baseada na extração dos dados das métricas objetivas em cada um três estilos apresentados nesse estudo. Essa coleta de dados foi feita a partir de ferramentas disponibilizadas pelo programa Xcode.

Na segunda etapa foi feita a aplicação de um questionário para as métricas subjetivas. Nesse questionário foi feito uma breve explicação de seu objetivo e onde vai ser utilizado. Para os participantes do questionário, foram convidadas pessoas que possuem experiência na área iOS e conhecimento sobre os estilos de construção de interface. Foi disponibilizado, para todos os respondentes, os arquivos do exemplo da Seção 4.2, sendo estes arquivos implementados nos três estilos de construção de interface. Foi pedido para que os respondentes que se baseiem no exemplo implementado e na sua experiência em cada um desses estilos para responder o questionário.

Com os dados coletados das métricas objetivas e subjetivas foi iniciada a terceira etapa. Nesta etapa foram dividas as métricas em relação a suas subcaracterísticas, para então, ser feita a análise comparativa entre os diferentes estilos e encontrar o estilo mais adequado para cada uma das características.

Na quarta etapa foi realizada uma síntese dos resultados, apresentando qual estilo obteve os melhores resultados em cada uma das subcaracterísticas. Por fim, foi identificado qual estilo obteve os melhores resultados em maioria das subcaracterísticas, com esse sendo considerado o melhor estilo de construção de interface para o estudo.

5 RESULTADOS OBTIDOS

Este capítulo apresenta os resultados obtidos a partir da aplicação da metodologia. Inicialmente será feita uma apresentação de como os dados foram coletados, seguidamente da amostragem dos dados e uma análise comparativa destes, levando em consideração os três estilos estudados anteriormente. Será apresentada uma síntese dos resultados, mostrando qual estilo foi considerado o melhor para cada subcaracterística e chegando a uma conclusão de qual foi o melhor estilo em geral. Por último temos o tópico de ameaça a validade, onde é explicado que o resultado não pode ser generalizado devido a determinados fatores.

5.1 APRESENTAÇÃO DOS DADOS COLETADOS E ANÁLISE COMPARATIVA DOS ESTILOS

Como dito na metodologia, as métricas são divididas entre objetivas e subjetivas. Para a coleta de dados das métricas objetivas foi utilizado um MacBook Pro (Retina, 13-inch, Early 2015), com um processador 2.7 Ghz Dual-Core i5, memória de 8GB 1867 Mhz DDR3, placa gráfica Intel Iris Graphics 6100 1536 MB, Disco rígido Macintosh HD 120gb e a versão 12.1(12A7403) do programa Xcode.

Para o caso das métricas subjetivas, a coleta dos dados foi realizada a partir de um questionário, o qual foi aplicado em um grupo de 9 pessoas que atuam como desenvolvedores ou professores na área.

As Figuras 13 e 14 são relacionadas às duas primeiras questões, que se referem à experiência do usuário na área e se ele possui conhecimento de todos os estilos. Todos (100%) possuem mais de 1 ano de experiência como desenvolvedor iOS e estão divididos igualmente entre 1 a 3 anos, 3 a 5 anos e mais de 5 anos. Todos os respondentes possuem conhecimento dos estilos Storyboard e View Code, com apenas um dos respondentes afirmado não ter conhecimento sobre SwiftUI.

A Figura 15 é relacionada a terceira questão, que se refere a que estilo os respondentes utilizam com mais frequência. A maioria (78%) afirma utilizar mais View Code, com nenhum afirmado utilizar SwiftUI com maior frequência.

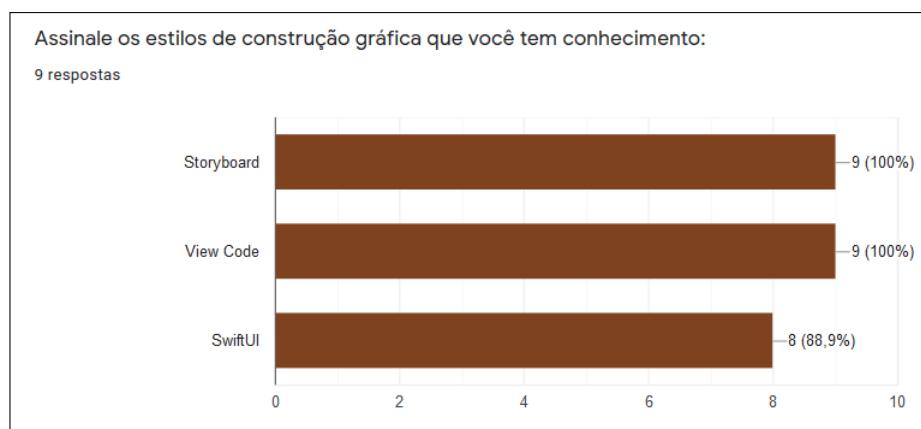
A seguir será apresentado os resultados da coleta de dados das métricas e feita a análise comparativa dos três estilos em relação as subcaracterísticas.

Figura 13 – Experiência do usuário



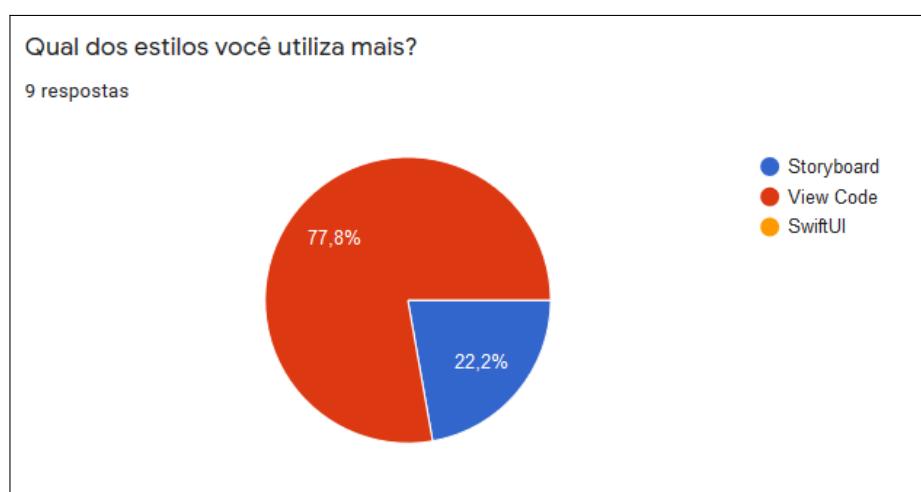
Fonte – Questionário desenvolvido pelo autor

Figura 14 – Experiência do usuário



Fonte – Questionário desenvolvido pelo autor

Figura 15 – Estilo mais utilizado pelo usuário



Fonte – Questionário desenvolvido pelo autor

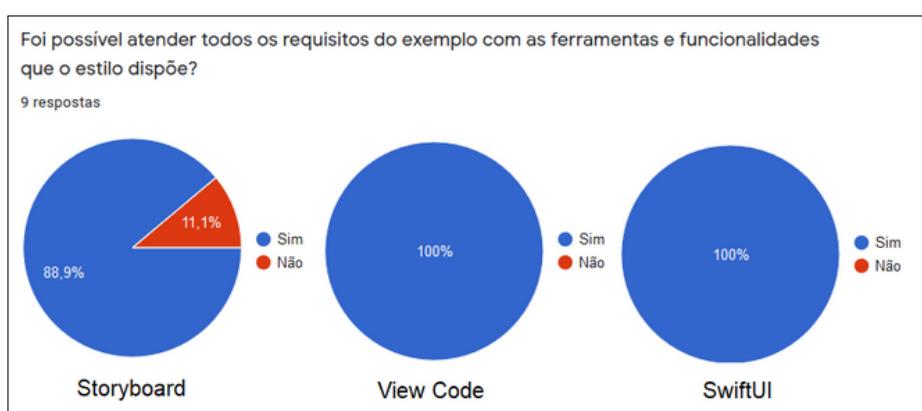
Qualidade Funcional

Na qualidade funcional foi avaliado se o estilo consegue cumprir todos os requisitos necessários para o desenvolvimento do exemplo. Esta subcaracterística foi analisada de maneira objetiva e subjetiva.

Para a métrica objetiva foi analisado se todos os estilos possuem os requisitos: Tabelas com elementos customizáveis, elementos de mapa, imagem e texto e ferramentas para organizar elementos da interface. O resultado foi que os três estilos apresentam todos os elementos necessários para o requisito, visto que o exemplo foi desenvolvido por completo nos três estilos.

Para a métrica subjetiva foi perguntado se os respondentes conseguiram atender todos os requisitos do exemplo com as ferramentas que o estilo dispõe. Analisando os dados obtidos a partir da Figura 16, podemos notar que todos conseguiram atender os requisitos em View Code e SwiftUI e apenas um não conseguiu atender a Storyboard.

Figura 16 – Qualidade Funcional

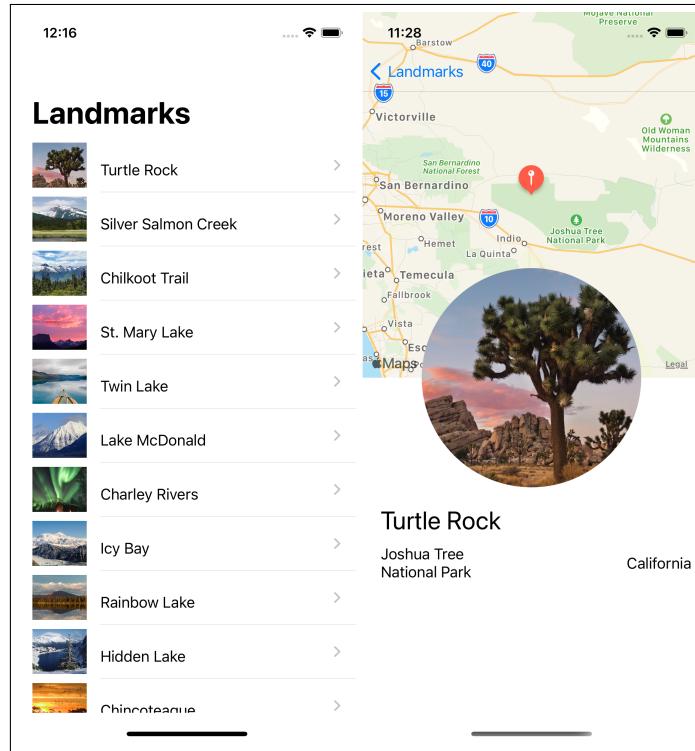


Fonte – Questionário desenvolvido pelo autor

Correção Funcional

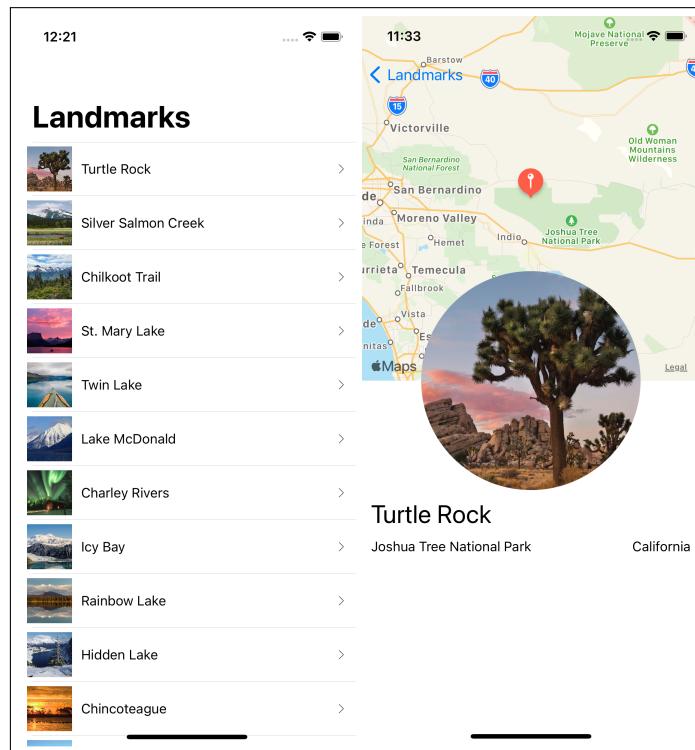
Em relação a correção funcional foi avaliado se com os estilos foi possível atender os mínimos requisitos de precisão da interface. Após analisar as interfaces nas Figuras 17, 18, 19, todos os estilos foram capazes de cumprir os mínimos requisitos de precisão.

Figura 17 – Interface da Storyboard



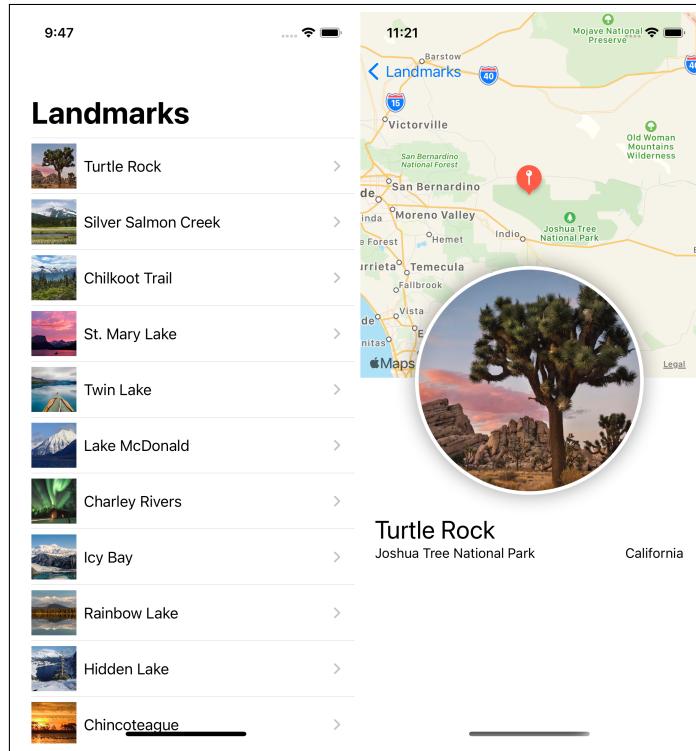
Fonte – Exemplo desenvolvido pelo autor

Figura 18 – Interface do View Code



Fonte – Exemplo desenvolvido pelo autor

Figura 19 – Interface do SwiftUI



Fonte – Exemplo desenvolvido pelo autor

Expressão Funcional

Para essa subcaracterística foi perguntado sobre a capacidade dos estilos em atender os requisitos do exemplo. Analisando os dados obtidos a partir da Figura 20, pode-se notar que a maioria dos respondentes avaliou o SwiftUI como o melhor para atender os requisitos do exemplo.

Comportamento Temporal

Para essa subcaracterística foi analisado se os requisitos de processamento atendem os determinados requisitos. Esta subcaracterística foi analisada de maneira objetiva e subjetiva.

Para a métrica objetiva foram analisados os requisitos de tempo de resposta, tempo de compilação, tempo de rodar no simulador. Cada um destes recursos foram avaliados em segundos com o programa em diferentes estados, com no mínimo 3 testes e obtida a media a partir do resultado desses testes.

O tempo de compilação foi analisado quando está sendo feita uma compi-

lação pela primeira vez e quando já foi feita uma compilação anteriormente, estando parte da compilação já salva na memória.

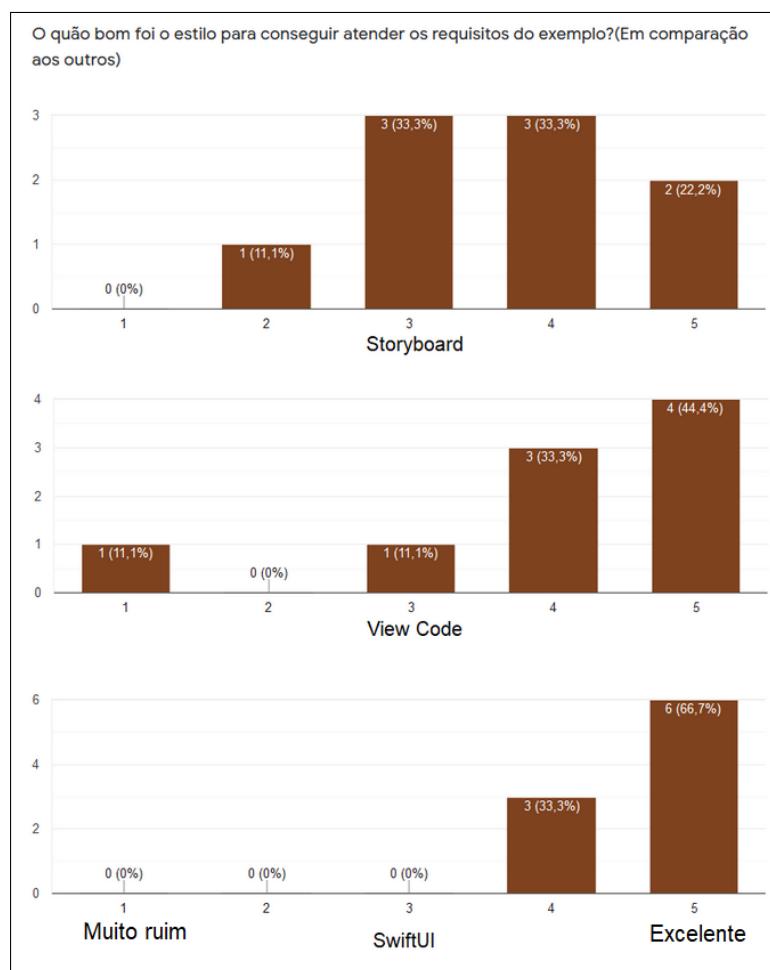
Tabela 2 – Tabela de tempo de compilação em segundos

Estilo	Média do Tempo de compilação na primeira vez	Média do Tempo de compilação salvo na memória
Storyboard	12,4s	2,7s
View Code	12,1s	2,6s
SwiftUI	13,7s	2,9s

Fonte – Elaborado pelo autor

Com o resultado apresentado na Tabela 2, podemos perceber que o melhor em relação ao tempo de compilação foi o View Code. Porém, todos os 3 estilos cumpriram o requisito mínimo.

Figura 20 – Expressão funcional



Fonte – Questionário desenvolvido pelo autor

O tempo de chegar ao estado inicial do exemplo no simulador foi analisado quando está sendo inicializado o exemplo pela primeira vez e quando já foi iniciado mais de uma vez, no qual a compilação e a instalação já ocorreram.

Tabela 3 – Tabela de tempo para rodar o simulador em segundos

Estilo	Média do tempo para chegar no estado inicial pela primeira vez	Média do tempo para chegar no estado inicial nas seguintes
Storyboard	15,6s	4,3s
View Code	16,1s	4,1s
SwiftUI	20s	4,8s

Fonte – Elaborado pelo autor

Com o resultado apresentados na Tabela 3, pode se notar que o melhor em tempo para rodar o simulador foi a Storyboard e nas vezes seguintes foi o View Code, com SwiftUI novamente com a pior média.

Em relação ao tempo de resposta foi analisado o tempo da ferramenta de interface. No caso da Storyboard foi analisado o tempo de abrir o arquivo .storyboard pela primeira vez e de abri-lo quando ele já foi aberto anteriormente. Esses arquivos são necessários para o desenvolvimento da interface.

Neste caso, o estilo View Code é desenvolvido programaticamente e a abertura do arquivo de interface é instantânea. Porém, View Code não possui uma ferramenta de visualização da interface em paralelo ao desenvolvimento, sendo necessário rodar o simulador para analisar como está a interface.

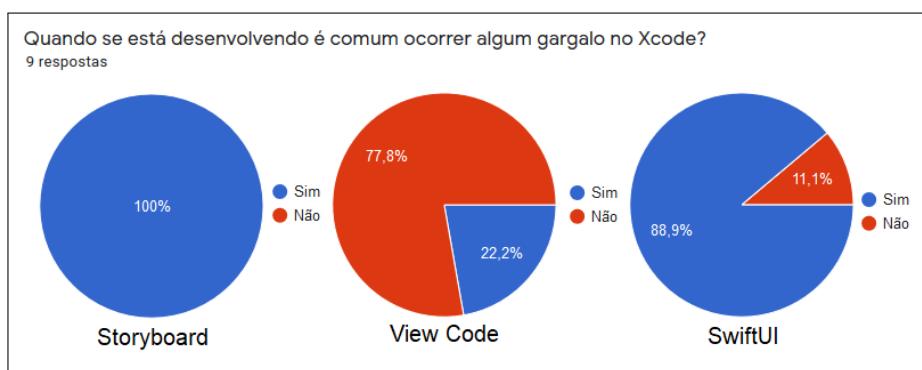
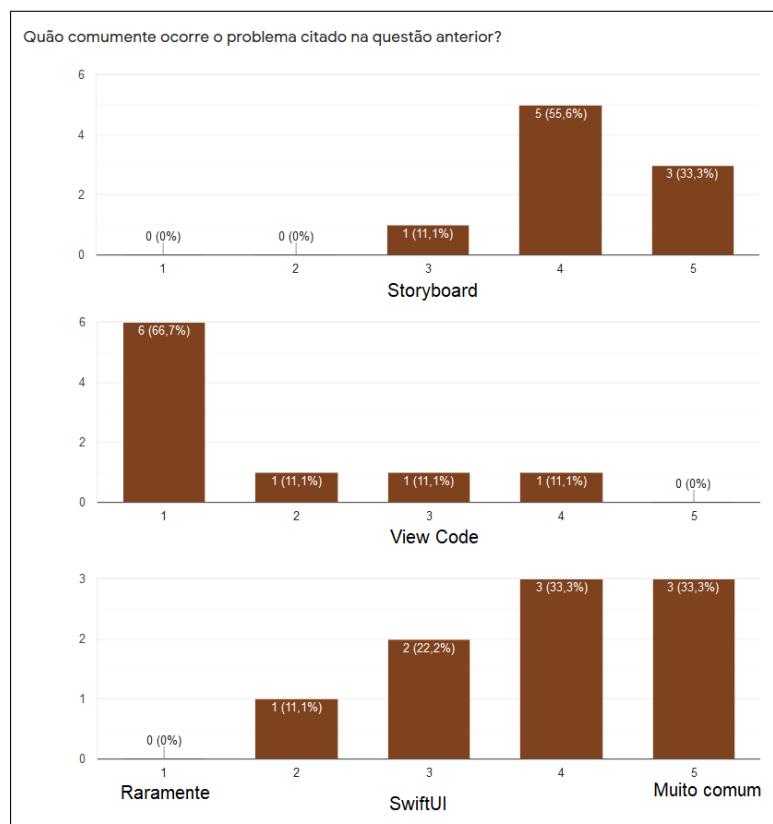
O estilo SwiftUI também é desenvolvido programaticamente, sendo sua abertura de arquivo de interface instantânea. Porém ele possui uma ferramenta para analisar a interface durante o desenvolvimento. Por isso, será analisado o tempo para a ferramenta Preview começar a funcionar na primeira vez e já tendo aberto anteriormente.

Os resultados obtidos na Tabela 4 mostram que ao querer modificar a Storyboard, é necessário esperar um período de tempo pequeno, porém existente, para abrir o arquivo. No caso de SwiftUI a Preview, que não é necessária para a modificação da interface, demora um tempo relativamente considerável para abrir pela primeira vez.

No caso das métricas subjetivas, foi perguntado se é comum ocorrer algum gargalo no programa Xcode e quão comumente ocorre este tipo de problema.

Tabela 4 – Tabela de tempo de resposta

Estilo	Média do Tempo para abrir a ferramenta de interface pela primeira vez	Média do Tempo para abrir a ferramenta de interface nas próximas vezes
Storyboard	4,3s	1,25s
SwiftUI	21s	1,1s

Fonte – Elaborado pelo autor**Figura 21 – Comportamento temporal 1****Fonte – Questionário desenvolvido pelo autor****Figura 22 – Comportamento temporal 2****Fonte – Questionário desenvolvido pelo autor**

Os resultados obtidos na Figura 21 mostram que todos os respondentes afirmaram já ter algum problema de gargalo ao utilizar Storyboards. No caso do View Code, para a maioria (77,8%) não é comum ocorrer problemas de gargalo. Em SwiftUI a maioria (88,9%) afirma ser comum ter problemas de gargalo.

Os resultados obtidos a partir da Figura 22 indicam quão comum é ocorrer este tipo de problema. No caso da Storyboard, maioria (88,8%) dos respondentes opinou entre muito comum e comum de ocorrer. No caso do View Code, a maioria (66,7%) respondeu que raramente ocorre esse tipo de problema. Em relação ao SwiftUI, a maioria (66,7%) opinou entre muito comum e comum de ocorrer esse tipo de problema.

Uso de Recursos

Para o uso de recursos foi avaliado sobre a utilização de recursos pelo programa e os impactos causados no sistema como um todo. Esta subcaracterística foi analisada de maneira objetiva e subjetiva.

Para as métricas objetivas foram analisados os seguintes requisitos: Uso de CPU, Uso de memória, uso de HD nos casos de o funcionamento do simulador e no uso da ferramenta para desenvolver a interface.

Tabela 5 – Tabela de Uso de recursos - Movimentação na tabela

Estilo	Uso de CPU	Memória
Storyboard	20~40% uso comum com máximas de 81%.	170mb media comum com máximas de 300mb.
View Code	15~25% uso comum com máximas de 81%.	155mb media comum com máximas de 175mb.
SwiftUI	20~40% uso comum com máximas de 81%.	220mb media comum com máximas de 270mb.

Fonte – Elaborado pelo autor

A Tabela 5 apresenta os dados obtidos para a movimentação na tabela de paisagens no simulador. Neste caso, a diferença de consumo de memória e de disco rígido para o estado ocioso foi praticamente nulo. Para a CPU, o consumo teve uma alta variância em todos os três estilos.

A Tabela 6 apresenta os dados obtidos em relação à movimentação entre

Tabela 6 – Tabela de Uso de recursos - Movimentação entre telas

Estilo	Uso de CPU para Navegar para os detalhes	Uso de CPU para Navegar de volta para a tabela
Storyboard	20% mínima e 95% máxima	6% mínima e 13% máxima
View Code	21% mínima e 86% máxima	6% mínima e 13% máxima
SwiftUI	10% mínima e 81% máxima	5% mínima e 20% máxima

Fonte – Elaborado pelo autor

a tela de tabelas de paisagens e a tela de detalhes da paisagem no simulador. Em relação a selecionar uma paisagem, a Storyboard apresentou a maior máxima com 95% do uso da CPU e SwiftUI apresentou a menor mínima com 10% de uso da CPU. Em relação a voltar para a tabela, todos mostraram resultados semelhantes com a maior máxima sendo 20% de uso de CPU em SwiftUI.

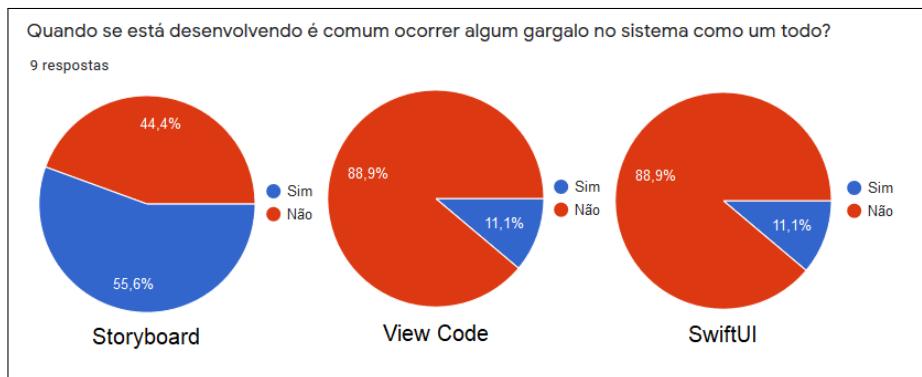
Para o uso na ferramenta de desenvolver a interface, foi avaliado o consumo de CPU, Memória e de HD em relação a máquina. A Storyboard e o SwiftUI levaram a máquina a consumir 100% de sua CPU ao abrir o arquivo .storyboard e ao inicializar o SwiftUI Preview pela primeira vez. Após abertos, o consumo nos testes não chegou a ultrapassar mais de 95% do uso da CPU. O View Code apresentou o menor consumo de CPU, não chegando a ultrapassar 70% do uso total de CPU, devido a não possuir nenhuma ferramenta de visualização da interface durante o desenvolvimento. Os valores de uso de memória e uso de HD para todos os estilos não impactaram negativamente no desenvolvimento da interface.

Para as métricas subjetivas foi perguntado se ocorre algum gargalo, como o programa Xcode utilizando tantos recursos ao ponto de travar o computador, e quanto comumente ocorre esse tipo de problema.

Os resultados obtidos nas Figuras 23 e 24 indicam se é comum e quanto comum é ocorrer este tipo de gargalo. Para a Storyboard, os resultados foram que para a maioria(55,6%) já ocorreu algum tipo de gargalo no sistema todo, com a maioria(81%) opinando que ocorre de uma maneira razoável a comum e com apenas 2 respondentes dizendo que é incomum de ocorrer.

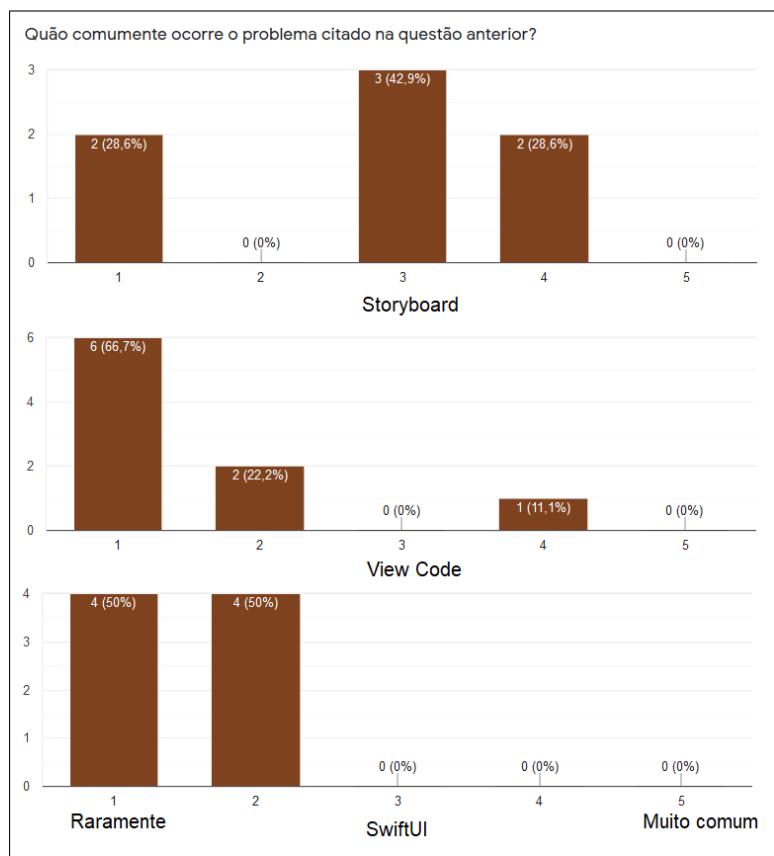
No View Code, a maioria (88,9%) respondeu que não é comum ocorrer esse tipo de gargalo, com a maioria (66,7%) opinando que ocorre muito raramente.

Figura 23 – Uso de recursos 1



Fonte – Questionário desenvolvido pelo autor

Figura 24 – Uso de recursos 2



Fonte – Questionário desenvolvido pelo autor

Em relação ao SwiftUI, a maioria (88,9%) respondeu que não é comum ocorrer esse tipo de gargalo, com as respostas divididas igualmente entre muito raramente e raramente.

Capacidade

Para a capacidade foi analisado se a quantidade de elementos na tabela impacta na utilização de recursos do computador e qual o limite de elementos possíveis na tabela até parar de funcionar.

Tabela 7 – Tabela de capacidade

Estilo	12 elementos	12000 elementos
Storyboard	CPU: 4% mínima e 13% máxima Memória:25,5mb	CPU: 14% mínima e 40% máxima Memória:31,8mb
View Code	CPU: 3% mínima e 24% máxima Memória:24,7mb	CPU: 21% mínima e 40% máxima Memória:26,5mb
SwiftUI	CPU: 18% mínima e 47% máxima Memória:28,3mb	CPU: 55% mínima e 80% máxima Memória:30,8mb

Fonte – Elaborado pelo autor

Os resultados obtidos na Tabela 7 indicam que houve um aumento considerável do uso da CPU em todos os estilos, com um aumento pequeno da memória e um uso praticamente nulo do disco rígido para todos os estilos.

No caso de limite de elementos na Storyboard, foi possível abrir cerca de 12.000 elementos. No View Code foi possível abrir cerca de 1.200.000 de elementos. Em relação ao SwiftUI foi possível abrir apenas 12.000 elementos, porém neste caso a movimentação pela tabela tornou-se inviável devido a travamentos. O funcionamento da movimentação pela tabela sem nenhum travamentos em SwiftUI foi possível com 1.200 elementos.

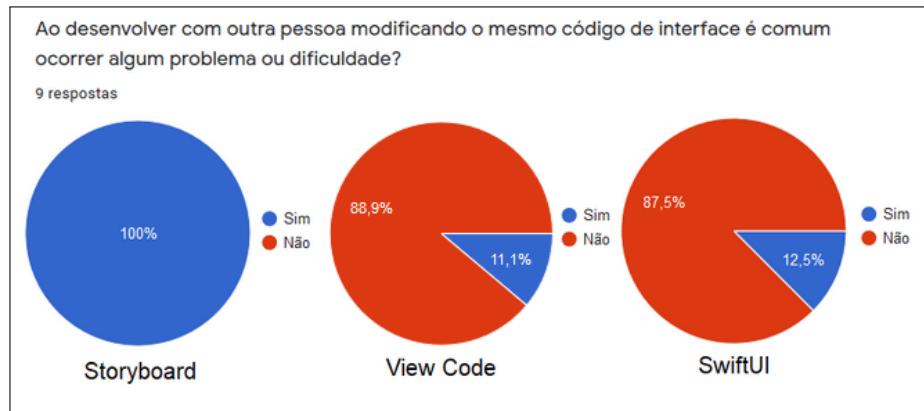
Coexistência

A coexistência foi avaliada a partir de métricas objetivas e subjetivas. Para a métrica objetiva foi avaliado se é possível dois ou mais desenvolvedores modificarem o código da interface simultaneamente. Para esse caso, foi encontrado que os estilos permitem essa modificação simultânea, porém cada um possui uma certa dificuldade para resolver problemas consequentes dessas modificações.

Para as métricas subjetivas, foi perguntado se ao desenvolver com outra pessoa modificando o mesmo código de interface é comum ocorrer algum problema ou

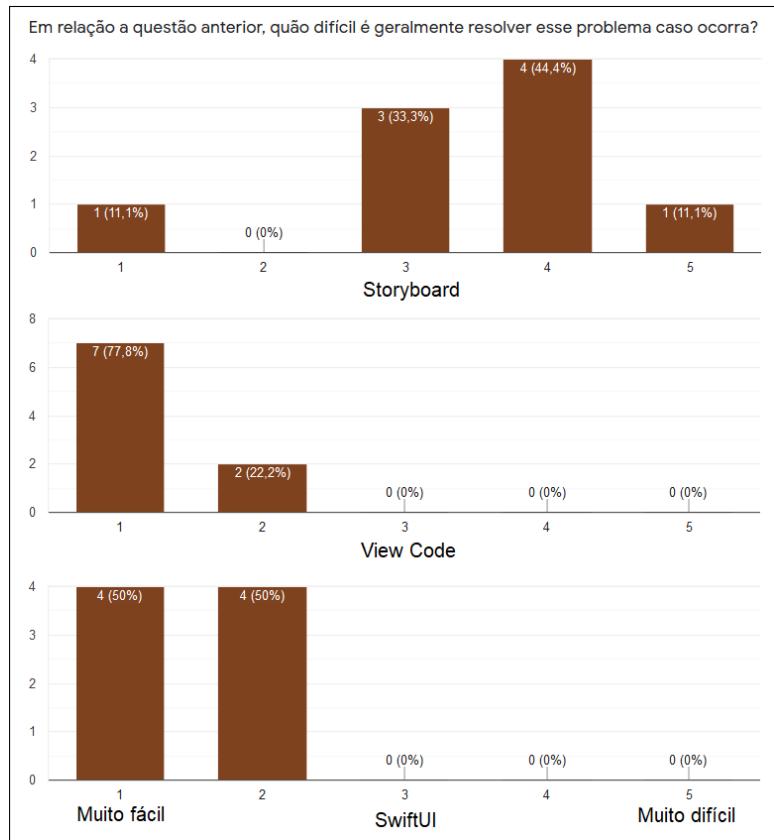
dificuldade e quão difícil é resolver esse problema quando ocorrido.

Figura 25 – Coexistência - Pizza



Fonte – Questionário desenvolvido pelo autor

Figura 26 – Coexistência - Colunas



Fonte – Questionário desenvolvido pelo autor

O resultado apresentado na Figura 25 mostra que todos os respondentes na Storyboard já tiveram algum tipo de problema quando dois ou mais desenvolvedores

modificam o código simultaneamente, sendo a maioria dos casos problemas de conflitos na hora de fazer a combinação dos códigos da Storyboard.

No caso do View Code e SwiftUI, a maioria respondeu que não é comum ocorrer esses problemas, e os que achavam comum responderam que ocorrem os mesmos conflitos de combinação, porém estes são mais fáceis de resolver.

Em relação à dificuldade de resolver os problemas, os resultados obtidos na Figura 26 indicam que todos acharam fácil ou muito fácil resolver o problema em View Code e SwiftUI. Isto se dá devido ao código de interface ser escrito em Swift, facilitando a sua leitura e modificação.

No caso da dificuldade da Storyboard, o resultado foi bem dividido, com a maioria entre uma dificuldade média e difícil. Isto se dá pelo fato de o arquivo da Storyboard ser em XML, dificultando a leitura e modificação deste tipo de arquivo.

Capacidade de reconhecer sua adequação

Neste caso avalia-se, a partir de uma métrica subjetiva, a adequação do estilo de interface antes mesmo de ser desenvolvido o exemplo.

Figura 27 – Capacidade de reconhecer sua adequação



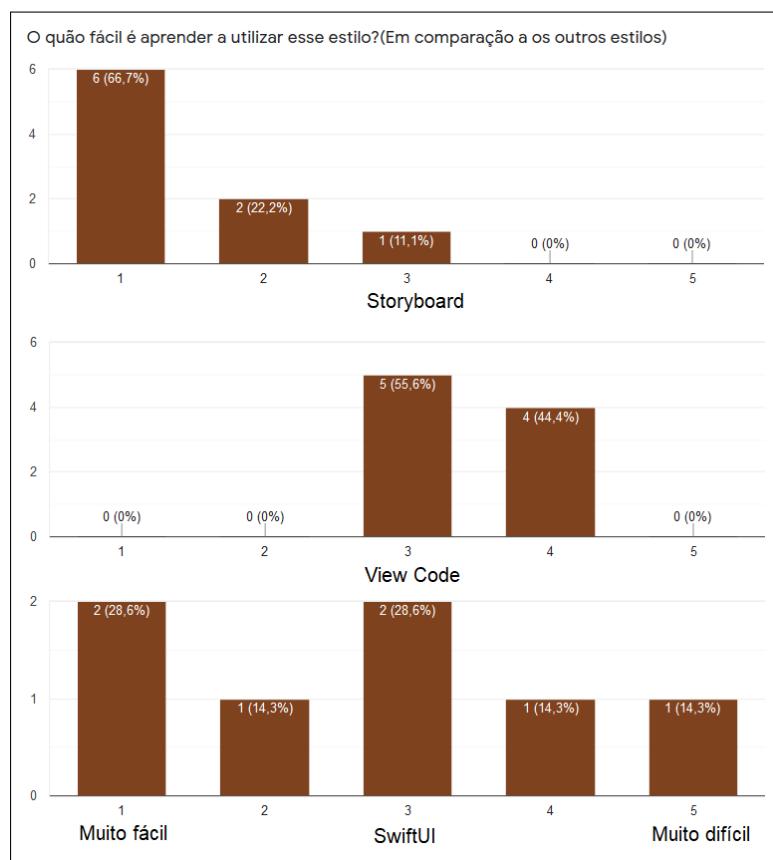
Fonte – Questionário desenvolvido pelo autor

Os resultados obtidos na Figura 27 indicam que todos os respondentes acreditam que SwiftUI é a melhor opção para desenvolver o exemplo em específico, devido à facilidade de criar projetos em SwiftUI com telas simples.

Capacidade de aprendizagem

A capacidade de aprendizagem procura avaliar o quanto fácil é aprender a utilizar o estilo de maneira eficaz e eficiente. Esse resultado é obtido a partir de uma métrica subjetiva. A Figura 28 apresenta os resultados em relação à capacidade de aprendizagem.

Figura 28 – Capacidade de aprendizagem



Fonte – Questionário desenvolvido pelo autor

No caso da Storyboard, a opinião da maioria dos respondentes foi que ela é fácil de aprender comparada aos outros estilos devido à ampla documentação disponível no site oficial da Apple.

No View Code, metade dos respondentes responderam que é uma dificuldade média e a outra metade que é difícil, devido à falta de documentação disponível e exigir um nível de entendimento maior em algumas APIs como o AutoLayout.

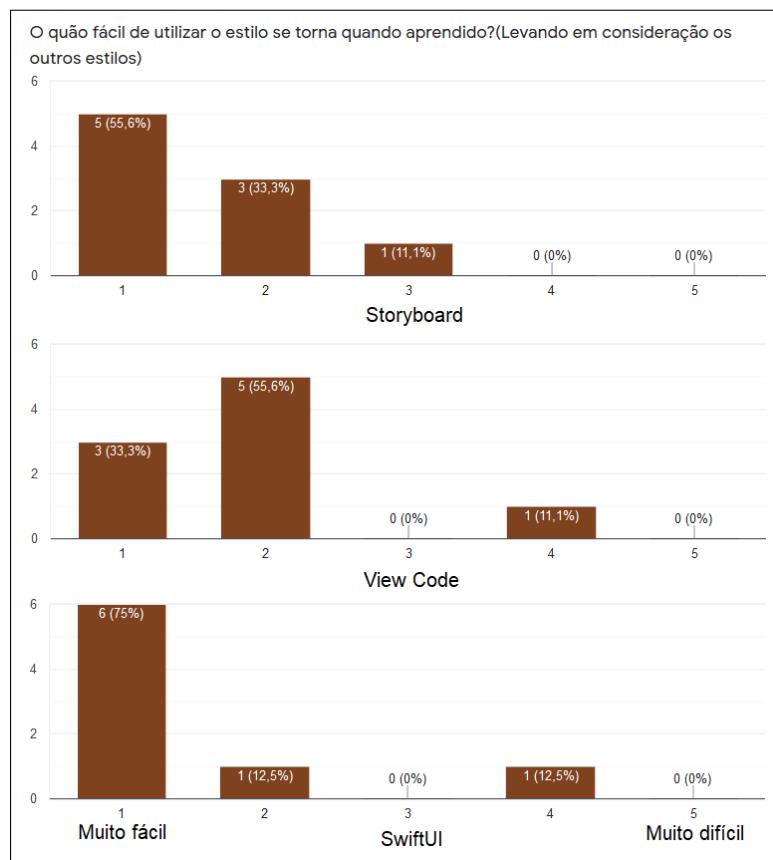
Em relação ao SwiftUI, a opinião ficou dividida em todas as opções de dificuldade, com os respondentes que colocaram difícil opinando que é uma mudança

de paradigma muito grande comparada aos seus antecessores, além de ser uma linguagem declarativa ao invés de imperativa.

Operabilidade

Em relação à operabilidade, foi analisado o quanto difícil é utilizar o estilo após este ter sido aprendido por completo.

Figura 29 – Operabilidade



Fonte – Questionário desenvolvido pelo autor

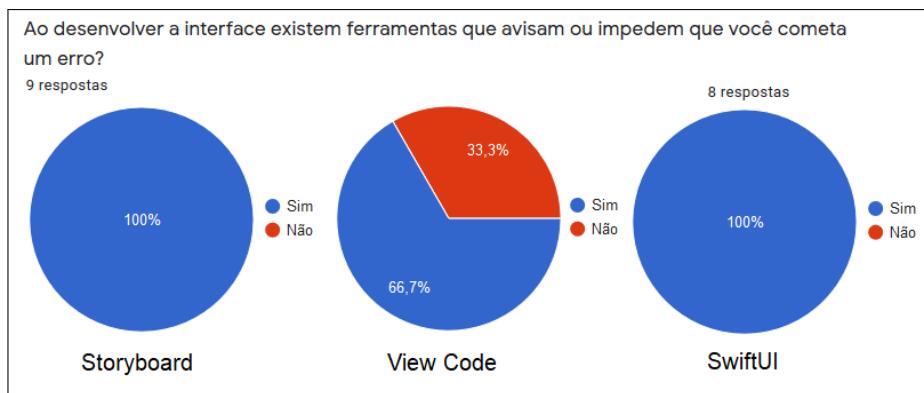
O resultado obtido a partir da Figura 29 indica que a maioria dos respondentes considerou os três estilos fáceis quando aprendidos, com apenas um considerando View Code e SwiftUI difíceis em comparação a Storyboard.

Proteção contra erros de usuário

Em relação à proteção contra erros de usuário, foi perguntado se os respondentes possuem algum conhecimento sobre ferramentas que avisam ou impedem que

se cometa um erro.

Figura 30 – Proteção contra erro de usuário



Fonte – Questionário desenvolvido pelo autor

O resultado obtido a partir da Figura 30 mostra que todos os respondentes possuem conhecimento de uma ferramenta de proteção contra erros em Storyboard e SwiftUI, já no caso do View code apenas parte (66,7%) dos respondentes tem conhecimento sobre uma ferramenta com essa função.

Acessibilidade

Em relação à acessibilidade, foi estudado se existe algum tipo de ferramenta para auxiliar pessoas com diferentes necessidades. Para esse caso, não foram encontradas ferramentas de acessibilidade específicas para o Xcode em nenhum dos três estilos.

Maturidade

Para a questão de maturidade, foi perguntado para os respondentes se eles consideram, a partir de sua própria experiência, os estilos em questão confiáveis.

Os resultados obtidos na Figura 31 indicam que todos acreditam que View Code é confiável e a maioria (88%) acredita que Storyboards e SwiftUI são confiáveis.

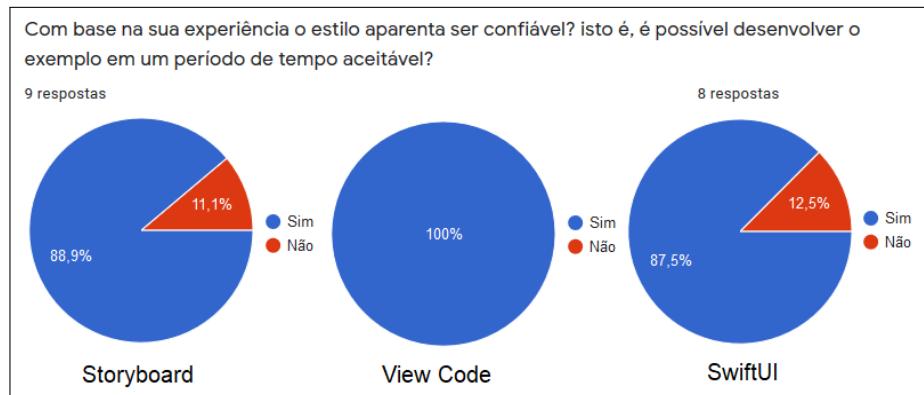
Capacidade de recuperação

A capacidade de recuperação procura avaliar se é possível recuperar os dados e o estado do programa caso tenha ocorrido alguma falha. As Figuras 32 e 33

apresentam os dados obtidos em relação a capacidade de recuperação.

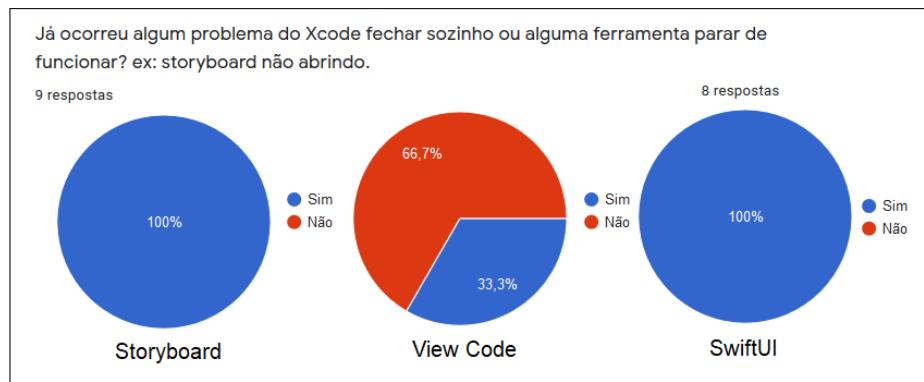
No caso da Storyboard, foi respondido por todos que já ocorreu algum tipo de problema de alguma ferramenta fechar sozinha. Destes casos, a maioria (66,7%) respondeu que foi possível recuperar todos os dados e voltar ao estado anterior.

Figura 31 – Maturidade



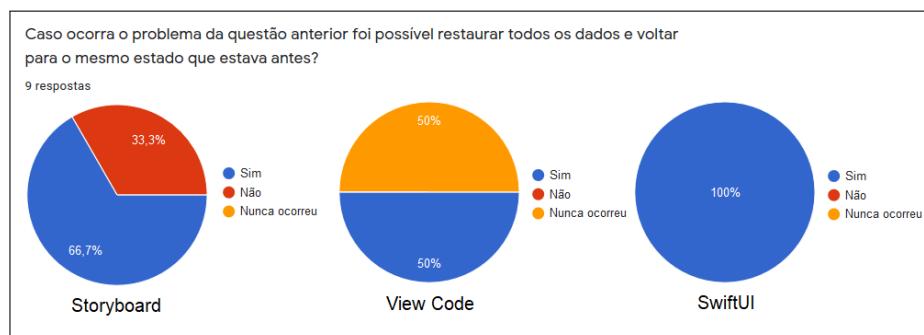
Fonte – Questionário desenvolvido pelo autor

Figura 32 – Capacidade de recuperação 1



Fonte – Questionário desenvolvido pelo autor

Figura 33 – Capacidade de recuperação 2



Fonte – Questionário desenvolvido pelo autor

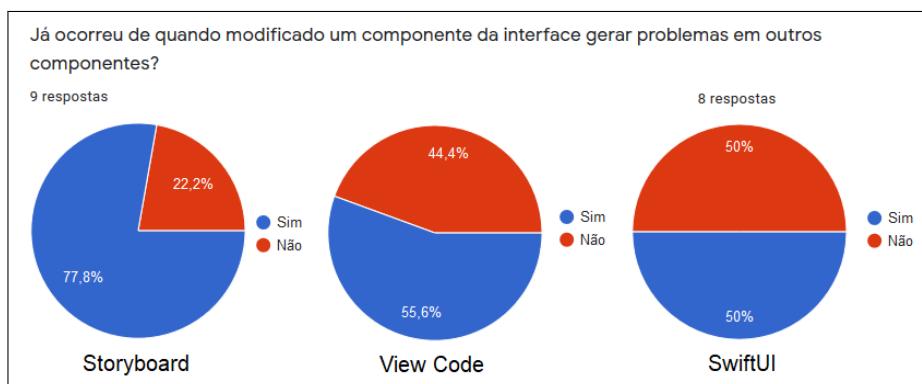
No View Code, a maioria (66,7%) dos respondentes respondeu que não ocorreu nenhum tipo de problema. Nos casos que ocorreram problema (33,3%) todos responderam que foi possível recuperar os dados e o estado anterior.

Em relação ao SwiftUI, todos responderam que já ocorreu esse tipo de problema e que foi possível recuperar todos os dados e o estado anterior.

Modularidade

Em relação à modularidade, foi perguntado se ao fazer modificações em componentes da interface ocorre algum tipo de impacto e, caso esse impacto seja negativo, se é fácil resolvê-lo. Também foi perguntado o quanto comum é fazer a modularização em cada um dos estilos. As Figuras 34, 35 e 36 apresentam os dados obtidos em relação a modularidade.

Figura 34 – Modularidade 1

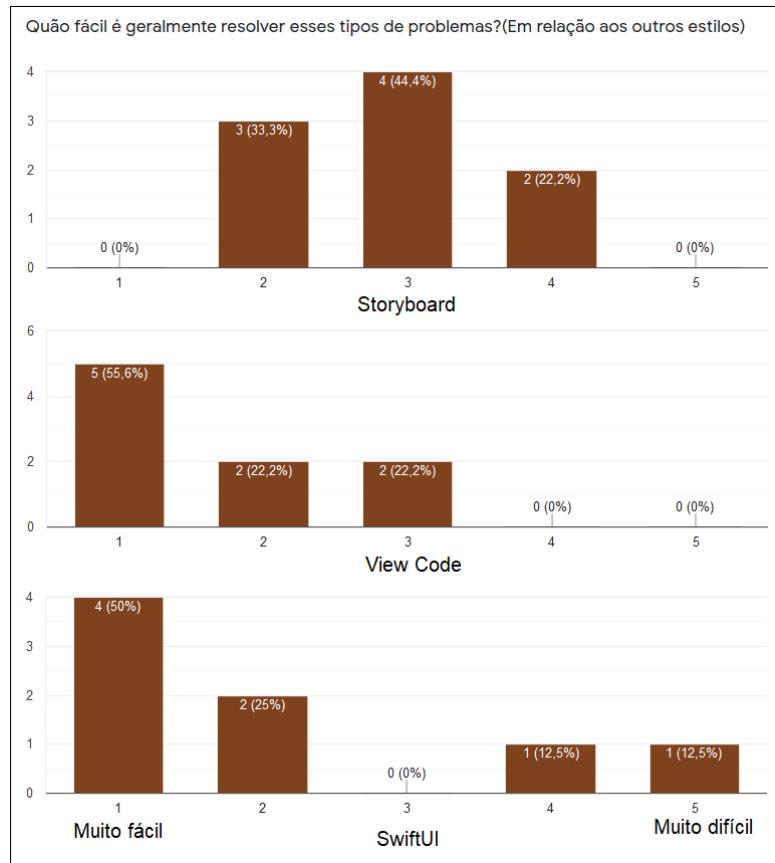


Fonte – Questionário desenvolvido pelo autor

No resultado da Storyboard, a maioria (77,8%) respondeu que já ocorreu algum problema em outros componentes ao modificar um componente. As respostas em relação à facilidade ficaram divididas entre fácil, médio e difícil. Em relação a utilizar comumente componentes na interface, 77,8% responderam que não utilizam.

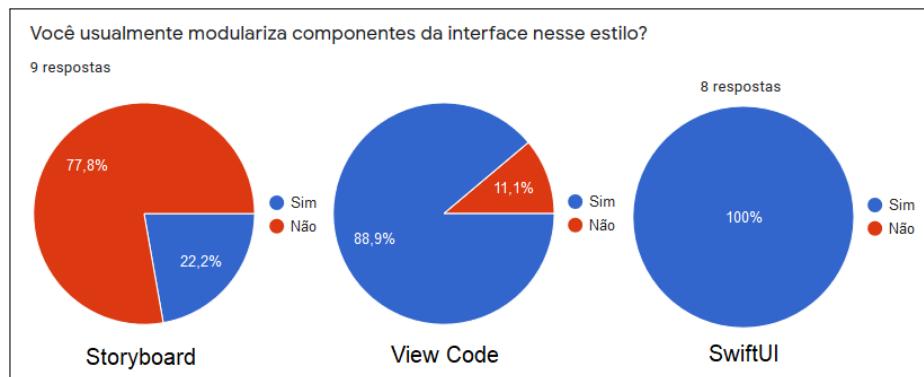
No View Code, a maioria (55,6%) também respondeu que já ocorreu algum problema em outros componentes. A maioria (55,6%) considerou muito fácil de resolver esse tipo de problema, com as outras respostas entre fácil e médio. Cerca de 88,9% dos respondentes afirmaram utilizar usualmente a modularização de componentes na interface.

Figura 35 – Modularidade 2



Fonte – Questionário desenvolvido pelo autor

Figura 36 – Modularidade 3



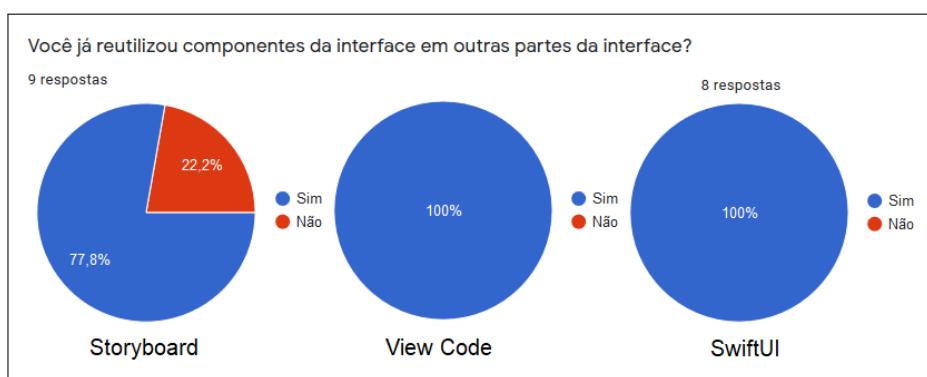
Fonte – Questionário desenvolvido pelo autor

Em relação ao SwiftUI, a metade dos respondentes afirmou que já ocorreu algum problema em outros componentes ao modificar um componente. A maioria (75%) opinou entre muito fácil e fácil para resolver esses tipos de problemas, com apenas 2 respondentes opinando difícil e muito difícil. Todos os respondentes afirmaram utilizar a modularização de componentes na interface.

Reusabilidade

Na reusabilidade foi avaliado se o respondente já reutilizou algum componente da interface em outras partes da interface e a facilidade na reutilização. Também foi perguntada a opinião do respondente em relação ao estilo promover a reutilização de componentes. As Figuras 37, 38 e 39 apresentam os dados obtidos em relação a reusabilidade.

Figura 37 – Reusabilidade 1



Fonte – Questionário desenvolvido pelo autor

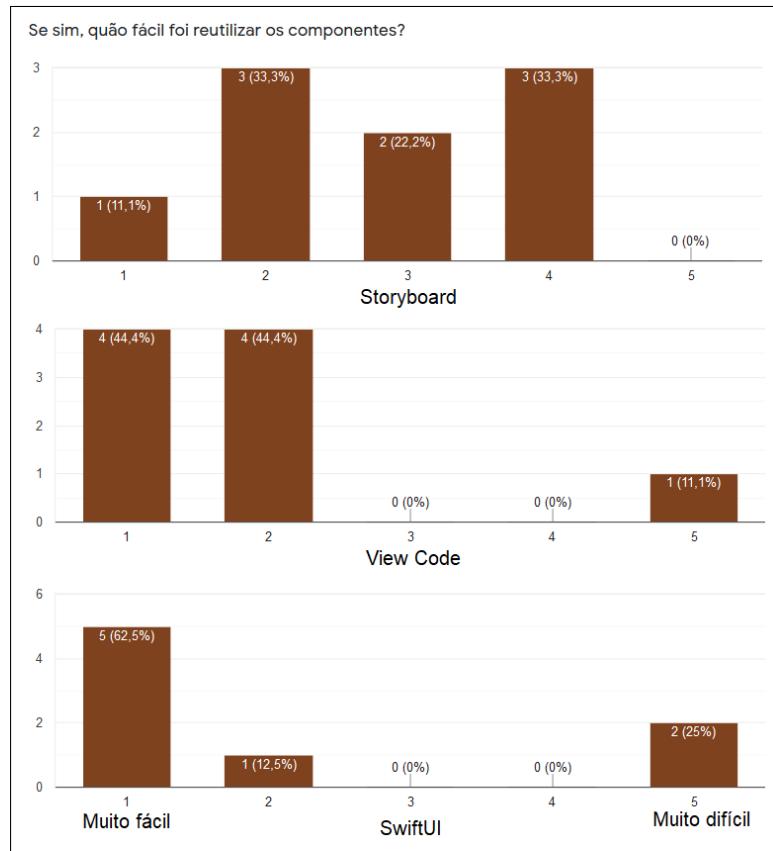
No caso da Storyboard, a maioria (77,8%) respondeu que já utilizou alguma vez componentes em outras partes da interface. Em relação à facilidade, as respostas ficaram bem diversificadas entre fácil, difícil e média, com apenas uma pessoa considerando muito fácil. Em relação a induzir a reutilização, a grande maioria (77,8%) opinou que não induz.

No View Code todos responderam que já reutilizaram, de alguma maneira componentes da interface. A maioria (88,8%) considerou a reutilização de componentes fácil, com apenas um considerando muito difícil. A maioria (88,9%) opinou que o estilo induz a reutilização.

Em relação ao SwiftUI, todos responderam que, em algum momento, já utilizaram partes da interface. Na facilidade da reutilização a maioria responderam fácil ou muito fácil (75%) com 2 respondentes marcando como muito difícil. Todos opinaram que o estilo induz a reutilização.

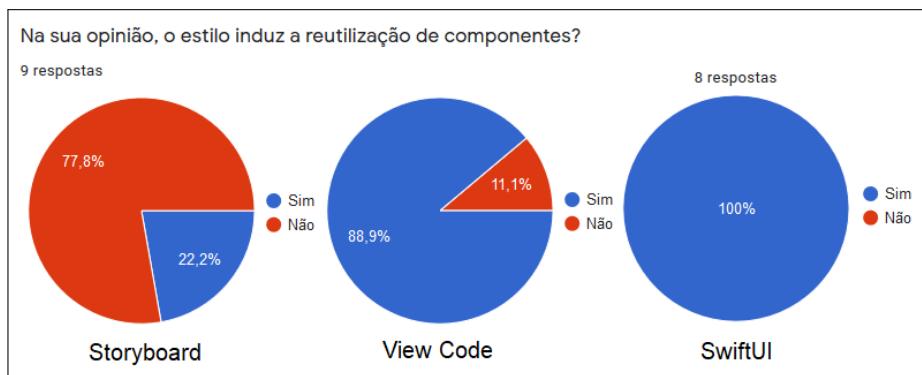
Para a reutilização em todos os casos é necessário previamente decidir quais componentes a se reutilizar e desenvolve-los num estilo que será apto a reutilização.

Figura 38 – Reusabilidade 2



Fonte – Questionário desenvolvido pelo autor

Figura 39 – Reusabilidade 3

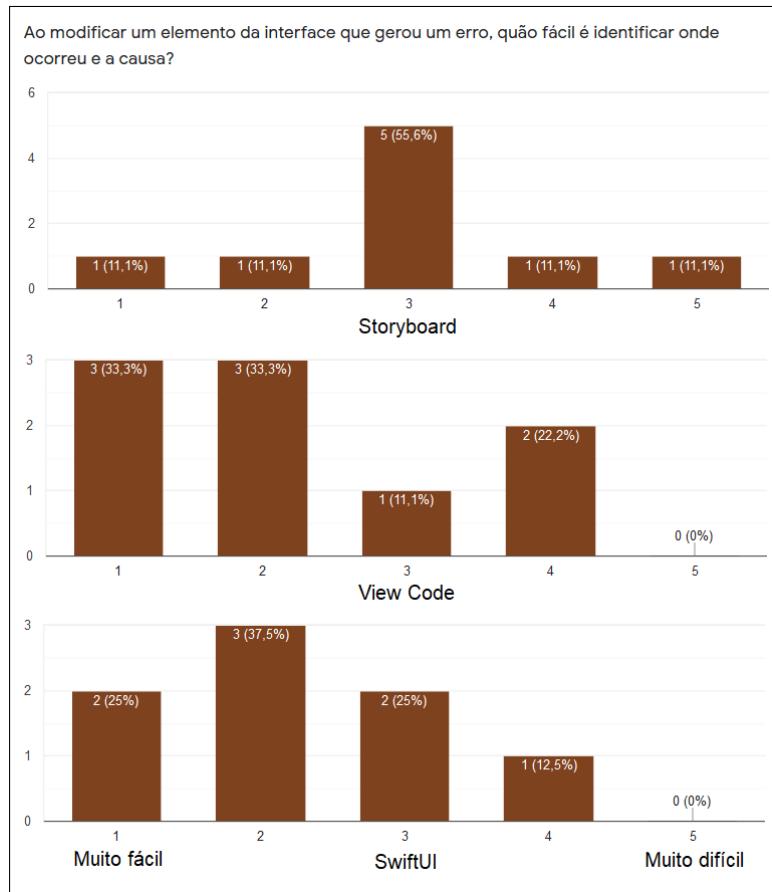


Fonte – Questionário desenvolvido pelo autor

Analisabilidade

A Analisabilidade busca analisar se é possível identificar a causa e o local caso ocorra alguma falha ao modificar uma interface. A partir dos resultados obtidos na Figura 40 percebe-se que todos os estilos tiveram opiniões diversificadas em relação à facilidade de identificar a causa e o local dessas falhas.

Figura 40 – Analisabilidade



Fonte – Questionário desenvolvido pelo autor

No caso da Storyboard, a maioria (55,6%) considerou uma dificuldade comum, com pessoas opinando em todos os outros níveis. Em relação ao View Code, a maioria opinou entre fácil e muito fácil (66,7%), com alguns opinando em média e difícil. E por último, no SwiftUI a maioria também opinou entre fácil e muito fácil (57,5%), com alguns opinando em média e difícil.

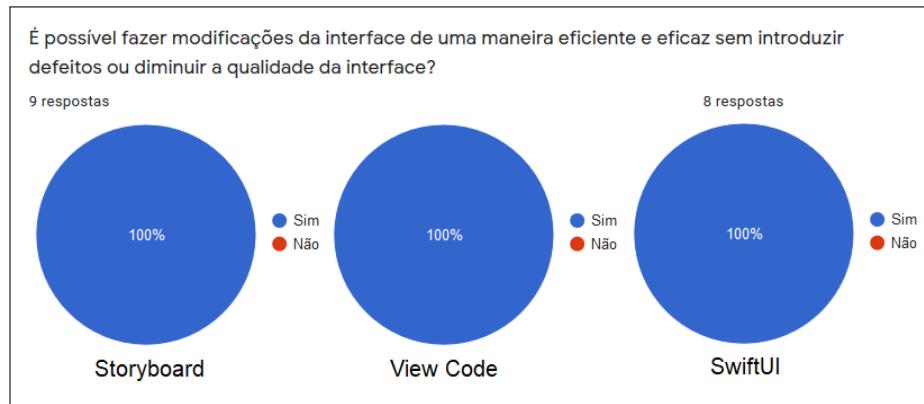
Modificabilidade

Para o caso de modificabilidade, foi perguntado aos respondentes se é possível fazer modificações na interface de uma maneira eficiente e eficaz sem introduzir defeitos ou diminuir a qualidade da interface. Também foi perguntado quão prático é fazer essas modificações. As Figuras 41 e 42 apresentam os dados obtidos em relação a modificabilidade.

Em todos os estilos é possível a modificação da interface. Em relação à praticidade, a maioria respondeu entre fácil e muito fácil no caso de View Code e

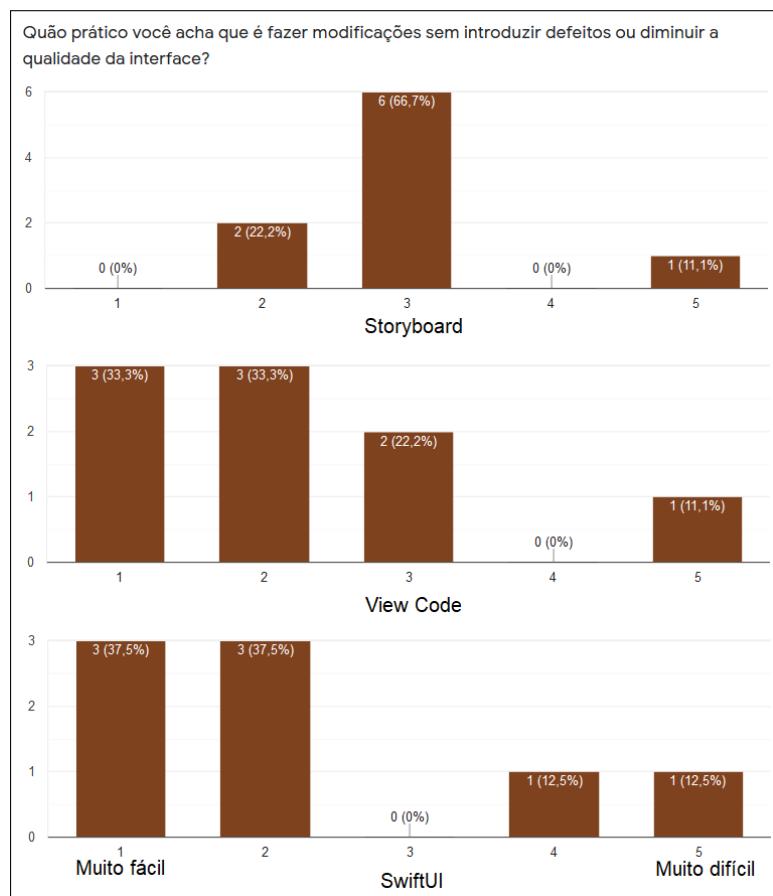
SwiftUI. No caso da Storyboard, a maioria respondeu que é uma dificuldade média.

Figura 41 – Modificabilidade 1



Fonte – Questionário desenvolvido pelo autor

Figura 42 – Modificabilidade 2



Fonte – Questionário desenvolvido pelo autor

Testabilidade

Em relação à testabilidade, foi estudado se existe alguma ferramenta na qual é possível criar parâmetros para teste e analisar se a interface em questão cumpre esses parâmetros. Todos os estilos utilizam-se da mesma ferramenta para testes unitários e testes de interface de usuário, chamada de XCTest. Nela é possível criar classes nas quais se coloca em passo a passo o caminho a ser testado. Após criada a classe, podem ser utilizadas diversas ferramentas para se obter os dados de teste.

5.2 SÍNTESE DOS RESULTADOS

Com os dados coletados e analisados, foi elaborada a Tabela 8, que indica qual estilo obteve os melhores resultados para cada uma das subcaracterísticas avaliadas, com algumas subcaracterísticas apresentando mais de um estilo.

Analizando a Tabela 8, podemos perceber que o estilo View Code foi avaliado como o melhor em maioria das subcaracterísticas. Sendo assim, podemos afirmar que o estilo de construção gráfica View Code se apresentou como o melhor estilo para o desenvolvimento em iOS, levando em consideração os dados coletados, o caso de uso utilizado para o estudo e o questionário respondido por desenvolvedores da área.

Tabela 8 – Tabela do resultado em relação aos dados obtidos

Características	Subcaracterísticas	Quem obteve os melhores resultados
Adequação funcional	Qualidade funcional	View Code e SwiftUI
	Correção funcional	View Code
	Expressão funcional	SwiftUI
Eficiência de desempenho	Comportamento temporal	View Code
	Uso de recursos	View Code
	Capacidade	View Code
Compatibilidade	Coexistência	View Code
Usabilidade	Capacidade de reconhecer sua adequação	SwiftUI
	Capacidade de aprendizagem	Storyboard
	Operabilidade	Storyboard e SwiftUI
	Proteção contra erros de usuário	Storyboard e SwiftUI
	Acessibilidade	Nenhum
Confiabilidade	Maturidade	View Code
	Capacidade de recuperação	View Code
Manutenção	Modularidade	View Code

Continuação da tabela 8		
Características	Subcaracterísticas	Quem se saiu melhor
	Reusabilidade	View Code
	Analisabilidade	View Code e SwiftUI
	Modificabilidade	View Code
	Testabilidade	Iguais

Fonte – Elaborado pelo autor

5.3 AMEAÇAS À VALIDADES

Esse trabalho possui algumas limitações que ameaçam a validade do trabalho. Em relação à eficiência de desempenho, Uso de recursos e Capacidade, não foi possível criar um ambiente perfeito para a coleta dos dados, sendo apenas capaz de minimizar o impacto de elementos externos a partir de múltiplos experimentos. Para esse trabalho foi utilizado apenas um caso de uso para a coleta de dados, impossibilitando a generalização deste resultado para outros casos de uso. As implementações em cada estilo foram feitas com base na experiência nesses estilos, podendo não ser a forma mais eficiente de desenvolver o caso de uso.

No caso da coleta de dados relacionada ao questionário, existiu uma certa dificuldade para achar um número grande de desenvolvedores iOS dispostos a responder o questionário e que tenham um conhecimento sobre os três estilos de construção de interface abordados nesse trabalho. Também é necessário levar em consideração o impacto que o nível de conhecimento de cada desenvolvedor em determinado estilo e sua tendência de possuir um estilo preferido traz para o resultado.

6 CONSIDERAÇÕES FINAIS

Existem diversos estilos de construção da interface de usuário em iOS, tendo cada um suas próprias peculiaridades, causando assim uma indecisão para o desenvolvedor de qual estilo seria mais adequado para o desenvolvimento de seu programa. Este trabalho busca ajudar desenvolvedores iOS na escolha do estilo mais adequado para o seu programa, permitindo um desenvolvimento mais eficiente e eficaz. Para isso, este trabalho propõe fazer uma comparação entre os estilos de construção da interface de usuário no ambiente iOS. Para isto, se utilizou de características da ISO/IEC 25010 (ISO, 2011) como metas para serem avaliadas. A partir dessas metas, foram elaboradas questões para determinar se essas metas podem ser atingidas. Para responder essas questões, foram elaboradas métricas, as quais são utilizadas na coleta de dados. Com os dados coletados, foi feita uma análise comparativa dos estilos e chegado a uma conclusão.

Em relação aos objetivos do trabalho, o estilo View Code se apresentou como o melhor para a maioria das subcaracterísticas. Assim, podemos afirmar que o estilo View Code foi considerado no geral o melhor estilo em relação à experiência dos respondentes com os estilos e os dados coletados do exemplo utilizado.

Em relação à adequação do estilo para o determinado exemplo, foi determinado que SwiftUI seria a melhor opção, tendo como justificativa que para casos da necessidade de escrever o código rápido e para aplicações pequenas e simples o SwiftUI apresenta ser a melhor opção. No caso da usabilidade, a Storyboard é uma excelente opção para o estilo de construção de interface, devido a facilidade de aprender o estilo e de utilizá-lo.

Devido à dificuldade de achar desenvolvedores iOS que possuam conhecimento em todos os três estilos de interface, só foi possível achar 9 respondentes para o questionário. Uma possível melhoria desta pesquisa seria o aumento do número de respondentes.

Outra sugestão para a futura evolução desta pesquisa seria utilizar um número maior de casos de uso para a avaliação da característica e da Eficiência de Desempenho. Outra sugestão seria utilizar outros métodos para analisar e comparar esses estilos de construção de interface.

REFERÊNCIAS

- APPLE. Swift. A powerful open language that lets everyone build amazing apps.** 2014. Disponível em: <<https://www.apple.com/swift/>>.
- APPLE. Storyboard.** 2018. Disponível em: <<https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html>>.
- APPLE. SwiftUI Essentials: Building Lists and Navigation.** 2020. Disponível em: <<https://developer.apple.com/tutorials/swiftui/building-lists-and-navigationw>>.
- APPLE. Apple Reinvents the Phone with iPhone.** 2021. Disponível em: <<https://www.apple.com/newsroom/2007/01/09Apple-Reinvents-the-Phone-with-iPhone/>>.
- APPLE. Model-View-Controller.** 2021. Disponível em: <<https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>>.
- APPLE. Storyboard.** 2021. Disponível em: <<https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html>>.
- APPLE. UIKit: Construct and manage a graphical, event-driven user interface for your iOS or tvOS app.** 2021. Disponível em: <<https://developer.apple.com/documentation/uikit/uikit>>.
- APPLE. UIView: An object that manages the content for a rectangular area on the screen.** 2021. Disponível em: <<https://developer.apple.com/documentation/uikit/uiview>>.
- BARKER, C. Learn SwiftUI: An introductory guide to creating intuitive cross-platform user interfaces using Swift 5.** [S.I.]: Packt Publishing Ltd., 2020.
- BASILI, V. R.; CALDIERA, G.; ROMBACH, H.** The goal question metric approach. **Encyclopedia of Software Engineering**, John Wiley & Sons, 1992.
- GSMA.** The mobile economy: Latin america 2020. **GSMA Intelligence**, GSM Association, 2020.
- GUSTAVO, R. M.; DANIELA, M.** Análise comparativa entre ferramentas de desenvolvimento de aplicativos móveis multiplataforma utilizando características da iso/iec 25010 por meio da implementação de um cenário pré-definido. Não publicado. 2019.
- HOFFMAN, J.** **Mastering Swift 4.** [S.I.]: Packt Publishing Ltd., 2017.
- ISO. ISO/IEC 25000.** 2011. Disponível em: <<https://iso25000.com/index.php/en/iso-25000-standards>>.
- JESSICA, M. L.; MATEUS, F. R.; CARLOS, H. R. G.** Um estudo comparativo entre os frameworks de desenvolvimento nativo uikit e swiftui da plataforma ios. Não publicado. 2020.
- KACZMAREK, S.; LEES, B.; BENNETT, G.** **Swift 5 for Absolute Beginners.** [S.I.]: Apress Media, 2019.

- MCKAY, E. **UI is Communication: How to design intuitive, user-centered interfaces by focusing on effective communication.** [S.I.]: Elsevier Inc, 2013.
- NAAVEN. **iOS Architecture: What is the architecture of iOS.** 2020. Disponível em: <<https://intellipaat.com/blog/tutorial/ios-tutorial/ios-architecture/>>.
- NEUBURG, M. **iOS 13 Programming Fundamentals with Swift.** [S.I.]: O'Reilly., 2019.
- PRESSMAN, R. S. **Engenharia de software.** Porto Alegre: Bookman, 2006.
- TIDWELL, J.; BREWER, C.; VALENCIA, A. **Designing Interfaces : Patterns for Effective Interaction Design.** [S.I.]: O'Reilly, 2020.
- ZVEIBIL, M. **Apple apresenta Xcode, a forma mais rápida de criar com aplicativos do Mac OS X.** 2003. Disponível em: <<https://www.apple.com/br/newsroom/2003/06/23Apple-Introduces-Xcode-the-Fastest-Way-to>Create-Mac-OS-X-Applications/>>.