PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA DE CIÊNCIAS EXATAS E DA COMPUTAÇÃO
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

FRAMEWORK PARA DESENVOLVIMENTO DE APLICATIVOS IOS COM INTERFACE DE USUÁRIO PROGRAMADA DE FORMA DECLARATIVA

BRENNO GIOVANINI DE MOURA

GOIÂNIA
2020

BRENNO GIOVANINI DE MOURA

FRAMEWORK PARA DESENVOLVIMENTO DE APLICATIVOS IOS COM INTERFACE DE USUÁRIO PROGRAMADA DE FORMA DECLARATIVA

Trabalho de Conclusão de Curso apresentado à Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para a obtenção do título de Bacharel em Engenharia de Computação.
Orientador(a):
Prof[a]. Ma. Ludmilla Reis Pinheiro dos Santos
Banca examinadora:
Prof. Me. Rafael Leal Martins
Esp. Arthur Junqueira Cançado

GOIÂNIA
2020

BRENNO GIOVANINI DE MOURA

FRAMEWORK PARA DESENVOLVIMENTO DE APLICATIVOS IOS COM INTERFACE DE USUÁRIO PROGRAMADA DE FORMA DECLARATIVA

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel em Engenharia de Computação, em 02/12/2020.

____
Orientadora: Profª. Ma. Ludmilla Reis Pinheiro dos Santos

____
Profª. Ma. Ludmilla Reis Pinheiro dos Santos
Coordenadora de Trabalho de Conclusão de Curso

GOIÂNIA
2020

À Universidade e aos professores que estimularam o conhecimento.

Ao apoio da minha família.

À minha mãe, meu pai, meus irmãos e colegas pelos momentos e parcerias.

AGRADECIMENTOS

A professora Ludmilla Reis Pinheiros dos Santos, orientadora acadêmica, pela disciplina, confiança e compromisso com o desenvolvimento deste trabalho.

Aos professores do curso de Engenharia de Computação por permitirem o desenvolvimento de novas ideias e por estimularem a pesquisa e o estudo em todos os trabalhos realizados durante o curso.

A Pontifícia Universidade Católica de Goiás pela infraestrutura e compromisso com o ensino de engenharia, que sempre cuidou dos laboratórios de informática e realizou diversas palestras e eventos que abordam conteúdos contemporâneos.

A minha família por apoiar e auxiliar em todos os momentos durante o curso, permitindo o desenvolvimento e execução de projetos pessoais e do curso de engenharia.

Aos meus colegas do curso de Engenharia de Computação pelos momentos e trabalhos realizados em equipe que contribuíram para a nossa formação, especialmente o Henrique Coelho dos Santos e a Laryssa Salustiana de Oliveira Pires.

Aos meus companheiros de trabalho e estágio que me apresentaram e me conduziram para a realização deste trabalho, principalmente o Túlio Vilela e o Ramon Vicente.

Aos demais que contribuíram diretamente e indiretamente com o desenvolvimento deste trabalho.

“Que tipo de programação desejamos ter? Queremos poder dizer ao computador o que fazer, de uma maneira fácil, confiável e que prontamente nos dá o resultado que queremos.”

Butler Lampson

RESUMO

Este trabalho implementa um *framework* para construção de interfaces de usuário usando o paradigma de programação declarativa nos dispositivos iOS. O desenvolvimento é composto de vários conceitos e recursos de programação, discutindo a integração com a linguagem de programação Swift e com o sistema operacional iOS. Os objetos declarativos encapsulam os objetos de interface do *framework* UIKit, sendo implementados os essenciais para permitir o uso prático em aplicações reais. A avaliação sobre *framework* desenvolvido por este trabalho consistiu em aplicar casos de teste que simulam uma tela ou um aplicativo prático, coletando e analisando dados de desempenho. Com isso, o trabalho mostrou-se consistente e aplicável, necessitando do desenvolvimento de novos conceitos para integralizar o *framework*.

Palavras-Chave: *Programação Declarativa, Interface, Framework, iOS.*

ABSTRACT

This work implements a framework for building user interfaces using declarative programming paradigm on iOS devices. The development consists of several concepts and programming resources, discussing integration with the Swift programming language and the iOS operating system. Declarative objects encapsulate the UIKit framework interface objects, the essential ones being implemented to allow practical use in real applications. The evaluation of the framework developed by this work consisted of applying test cases that simulate a screen or a practical application, collecting and analyzing performance data. With that, the study proved to be consistent and applicable, requiring the development of new concepts to integrate the framework.

Keywords*: Declarative Programming, Interface, Framework, iOS.*

LISTA DE FIGURAS

Figura 1 - Extensão do objeto Double para conversão direta de distâncias. 26

Figura 2 - A arquitetura MVC do UIKit. 29

Figura 3 - Ciclo de vida do UIKit. 30

Figura 4 - Debug da hierarquia de interface do aplicativo Olá Mundo. 31

Figura 5 - Métodos da UIViewController que informam o estado atual da *view* na hierarquia. 33

Figura 6 - Estados da UIViewController que informam o momento que a *view* se encontra na hierarquia. 34

Figura 7 - Layout hexagonais para notas musicais. 35

Figura 8 - Trecho de código usando a instrução ANEL, tornando o código tolerante a falhas. 36

Figura 9 - Construção do algoritmo de exemplo usando o framework DDFlow. 37

Figura 10 - Código comparativo entre *callback* com e sem a marcação de escape. 41

Figura 11 - Resultado da execução do código conforme a Figura 10. 41

Figura 12 - Código exemplo usando a programação declarativa para construir um alerta. 43

Figura 13 - Alerta construído usando a programação declarativa. 44

Figura 14 - Código exemplo da UIAlertController para mostrar um alerta. 45

Figura 15 - Alerta construído usando a programação imperativa. 46

Figura 16 - Fluxo lógico da programação declarativa. 47

Figura 17 - *Callbacks* em estrutura recursiva. 49

Figura 18 - Protocolo com variável *image* implementada usando a extensão. 50

Figura 19 - Protocolo com tipo genérico associado. 52

Figura 20 - Estágios de renderização da UIView. 53

Figura 21 - Implementação do estágio não renderizado na classe View. 54

Figura 22 - Protocolo com tipo associado utilizado como parâmetro genérico da função isEqual<Element>(Element, Element). 55

Figura 23 - Protocolo ViewCreator. 56

Figura 24 - Classe MutableBox. 58

Figura 25 - UICNavigation com título igual a "Início" e *view* vazia com cor de fundo cinza claro. 62

Figura 26 - Execução do UICNavigation com título igual a "Início" e *view* vazia com cor de fundo cinza claro. 63

Figura 27 - UICTab com duas views "Início" e "Ajustes". 64

Figura 28 - Execução do UICTab com a view "Início" selecionada. 65

Figura 29 - UICVStack com três *views* distribuídas igualmente. 66

Figura 30 - Execução da UICVStack com três *views* distribuídas igualmente. 67

Figura 31 - UICVScroll com uma *view* de altura igual a 175. 68

Figura 32 – Execução da UICVScroll com *view* de altura igual a 175. 68

Figura 33 - UICLabel com o texto "Olá mundo!". 69

Figura 34 - Execução da UICLabel com o texto "Olá mundo!". 70

Figura 35 - UICImage com símbolo do sol nascente. 71

Figura 36 - Execução da UICImage com símbolo do sol nascente. 71

Figura 37 - UICButton com texto "Aperte". 72

Figura 38 - Execução do UICButton com texto "Aperte". 73

Figura 39 - UICText com campo igual a "Nome". 74

Figura 40 - Execução do UICText com campo igual a "Nome". 74

Figura 41 - Evento para observar a alteração do valor armazenado no Value. 81

Figura 42 - Evento para ler o valor em Value. 81

Figura 43 - Evento para atribuir um novo valor ao Value. 82

Figura 44 - UICForEach como conteúdo dinâmico da UICVStack. 84

Figura 45 - Execução da UICVStack com UICForEach dinâmico. 84

Figura 46 - Exemplo de implementação do protocolo UITableViewDataSource. 86

Figura 47 - Protótipo de código para sequência de objetos como conteúdo da lista. 88

Figura 48 - Fluxo de atualização da ação de remoção da UITableView. 89

Figura 49 - Implementação do cabeçalho da seção com auto layout. 94

Figura 50 - Execução do cabeçalho da seção com auto layout. 94

Figura 51 - Utilizando a UICList para mostrar números de zero a nove. 97

Figura 52 - Execução da UICList com números de zero a nove. 98

Figura 53 - UICList com cabeçalho utilizando a classe UICHeader. 99

Figura 54 - Execução da UICList com cabeçalho definido pela classe UICHeader. 100

Figura 55 – UICList listando célula com ação editar à direita. 101

Figura 56 - Execução da UICList listando célula com ação editar à direita. 102

Figura 57 - Protótipo do layout das células definidos por valores relativos e fixos. 105

Figura 58 - Coleção UICFlow que utiliza a UICollectionViewFlowLayout para distribuir as células. 106

Figura 59 - Execução da UICFlow formada por grupos e células com largura relativa. 107

Figura 60 - Comparação entre consumo de memória nos casos de teste 1 ao 4. 117

Figura 61 - Comparação entre uso de processamento na inicialização nos casos de teste 1 ao 4 118

Figura 62 - Comparação entre uso de processamento nos casos de teste 1 ao 4. 119

Figura 63 - Comparação entre uso das memórias HEAP & VM nos casos de teste 1 ao 4. 120

Figura 64 - Comparação entre tempo de processamento nos casos de teste 1 ao 4. 121

Figura 65 - Comparação entre quantidade de linhas nos casos de teste 1 ao 4. 122

LISTA DE TABELAS

Tabela 1 - Comparação entre código SQL (declarativo) e código Python (imperativo). 23

Tabela 2 - Resultados da execução do Caso de Teste 1. 110

Tabela 3 - Resultados da execução do Caso de Teste 2. 111

Tabela 4 - Resultados da execução do Caso de Teste 3. 111

Tabela 5 - Resultados da execução do Caso de Teste 4. 112

Tabela 6 - Resultados da execução do Caso de Teste 5. 113

Tabela 7 - Resultados da execução do Caso de Teste 6. 113

Tabela 8 - Resultados da execução do Caso de Teste 7. 114

Tabela 9 - Resultados da execução do Caso de Teste 8. 115

LISTA DE SIGLAS

ARC	*Automatic Reference Counting*
DCL	*Data Control Language*
DDL	*Data Definition Language*
DML	*Data Manipulation Language*
FRP	*Functional Reactive Programming*
HTML	*HyperText Markup Language*
IDE	*Integrated Development Environment*
IoT	*Internet of Things*
MVC	*Model View Controller*
MVVM	*Model View View Model*
RVR	*Reactive Values and Relations*
NPD	*Network Programming Defects*
SDK	*Software Development Kit*
SO	Sistema Operacional
SQL	*Structured Query Language*
UI	*User Interface*
WWDC	*WorldWide Developers Conference*
XML	*eXtensible Markup Language*
Yacc	*Yet another compiler compiler*
SSOT	*Single Source Of Truth*

SUMÁRIO

1 INTRODUÇÃO 16
1.1 Objetivos gerais 17
1.2 Objetivos específicos 18
1.3 Metodologia 18
1.4 Justificativa 18
1.5 Organização do trabalho 19

2 REFERENCIAL TEÓRICO 21
2.1 Programação declarativa e programação imperativa 21
2.2 Programação funcional 23
2.3 Programação reativa 24
2.4 Linguagem de programação Swift 25
2.5 O Kit de Desenvolvimento de Software iOS 28
2.6 Trabalhos relacionados 34
2.6.1 The Arpeggigon: Declarative Programming of A Full-Fledged Musical Application *35*
2.6.2 Anel: Robust Mobile Network Programming Using a Declarative Language *36*
2.6.3 DDFlow: Visualized Declarative Programming for Heterogeneous IoT Networks *37*
2.6.4 Using Declarative Programming in an Introductory Computer Science Course for High School Students *38*
2.6.5 Análise dos Trabalhos Relacionados *39*

3 DESENVOLVIMENTO 40
3.1 Programação declarativa 40
3.1.1 Callback *40*
3.1.2 Sintaxe declarativa *42*
3.1.3 Empilhamento de chamadas *48*
3.1.4 Protocolos declarativos *49*
3.2 Composição declarativa 52
3.2.1 Encapsulamento da classe UIView *52*
3.2.2 Protocolo declarativo ViewCreator *54*
3.2.3 Unindo o protocolo e a instância da UIView *57*
3.2.4 Classe de renderização *59*
3.3 Estrutura e objetos de interface do UIKit 60
3.3.1 Hierarquia de objetos de interface *60*
3.3.2 Objetos principais da classe UIViewController *61*
3.3.3 Objetos principais da classe UIView *65*

4 COMPONENTES REATIVOS 77
4.1 Estrutura de notificações 77
4.2 A classe ReactiveCenter 79
4.2.1 Single Source of Truth *79*
4.2.2 Eventos Reativos *80*
4.3 Construtor dinâmico UICForEach 82
4.4 UICList 85

4.4.1 Objetos construtores para seção, cabeçalho, rodapé e célula 87
4.4.2 Classes que representam o estado da lista... 90
4.4.3 Implementação da TableViewCell e TableHeaderFooterView.................... 92
4.4.4 Implementando os protocolos UITableViewDataSource e UITableViewDelegate... 95
4.5 UICFlow 102
4.5.1 Layout da célula a partir do UICCollectionLayoutManager 104

5 RESULTADOS.. 108
5.1 Esquematização dos casos de testes 108
5.2 Análise dos casos de teste 110
5.3 Análise dos casos de teste 1 ao 4 116
5.4 Análise dos casos de teste 5 ao 8 122

6 CONSIDERAÇÕES FINAIS 125
6.1 Trabalhos futuros.. 126

REFERÊNCIAS... 128

APÊNDICE A – CASOS DE TESTE 135

ANEXO A – TERMO DE AUTORIZAÇÃO DE PUBLICAÇÃO DE PRODUÇÃO ACADÊMICA 140

1 INTRODUÇÃO

A programação declarativa está presente nas principais plataformas de desenvolvimento. Ela é um paradigma de programação, em que o programador se preocupa com o que será feito e não em como será feito, como acontece com a programação imperativa. A sua adoção nas plataformas *mobile*, *web* e *desktop* traz o benefício de permitir que a interface de usuário, do inglês *User Interface* (UI) escrita usando linguagens de marcação, como por exemplo o *eXtensible Markup Language* (XML) e o *HyperText Markup Language* (HTML), possam ser codificadas usando uma linguagem de programação (BREWSTER, 2019).

A melhor abordagem para integrar os conceitos da programação declarativa para UI é por meio da implementação de um *framework*. Ele é um conjunto de regras, que integradas, resolvem um problema de domínio específico (SAUVÉ, 2020). O SwiftUI, implementado pela Apple em 2019, é um *framework* que alterou o padrão de desenvolvimento ao integrar o paradigma de programação declarativa e a programação reativa nos projetos desenvolvidos para o Sistema Operacional (SO) iOS. Ele foi escrito em Swift, lançado em 2014, redesenhando a sintaxe e modernizando vários conceitos de otimização e manipulação de memória (APPLE, 2020; SWIFT, 2020).

O Ambiente de Desenvolvimento Integrado, do inglês *Integrated Development Environment* (IDE), Xcode explora todos os recursos disponíveis no Swift e no iOS, apesar de existir outros IDE para o desenvolvimento iOS, por exemplo o Android Studio (GOOGLE, 2020). No Xcode estão definidos recursos de inspeção de memória, de desempenho, de acessibilidade, dentre outros recursos adotados pelo iOS. A implementação de projetos utilizando esse IDE é baseada na arquitetura *Model View Controller* (MVC), definida pelo *framework* UIKit (APPLE, 2020).

A MVC é uma arquitetura de projeto que define os três objetos necessários para gerenciar a interação do software com o usuário. O *Model* é encarregado de armazenar e manipular os dados à medida que as interações são realizadas. A *View* é encarregada de mostrar as informações na tela do dispositivo. O *Controller*, ou controlador, gerencia a interação com o usuário, recebendo os eventos e enviando para o Model, que retorna as informações processadas para a *View*. Além dessa arquitetura, no iOS, também pode ser utilizado a arquitetura *Model View View Model* (MVVM), abstraindo a camada do controlador (MEDEIROS, 2013; STRAWN, 2018).

O UIKit é um *framework* estrutural para todos os aplicativos desenvolvidos para iOS, não sendo possível implementar uma aplicação sem o uso dele. O UIKit define todas as classes para implementação de interface, imagem, cor e várias outras com funções específicas para usar os recursos do iOS. A UIViewController, definida pelo UIKit, é uma classe que representa a arquitetura MVC utilizada na hierarquia de interface no iOS (APPLE, 2020).

Segundo Jirásek (2019), os projetos baseados no *framework* UIKit precisam ser reestruturados para se adaptar ao SwiftUI, principalmente, devido a mudança da arquitetura MVC para a arquitetura MVVM. Existem *frameworks* de interface declarativa que compartilham a mesma ideia, mas seguem propostas diferentes, por exemplo: (a) o SwiftUI integra o iOS, o compilador e o Xcode para a construção do projeto; (b) o Litho e o ComponentKit, do Facebook, otimizam a renderização dos objetos de interface; (c) o Flutter, do Google, é uma solução híbrida para o desenvolvimento de um único código fonte para plataformas diferentes; (d) o React, do Facebook, é um *framework* de código aberto para as plataformas *desktop*, *web* e *mobile*, utilizando uma linguagem de construção para interface próximo da linguagem de marcação HTML (APPLE, 2020; FACEBOOK, 2020; GOOGLE, 2020).

Nesse cenário, este trabalho visa responder as seguintes questões: (a) É possível desenvolver um *framework* para construção de interface usando a programação declarativa que seja suportado pelo iOS 10 e superiores?; e (b) É possível manter os padrões de projeto e suas arquiteturas aplicando os mesmos conceitos do SwiftUI sem a necessidade da reescrita do *back-end*?

1.1 Objetivos gerais

Desenvolver o *framework* UICreator para iOS que permita a implementação da interface de usuário usando a programação declarativa, que suporte as versões do iOS 10 e superiores, avaliando o desempenho com o uso de casos de teste definidos no Apêndice A.

Implementar um conjunto de regras para a construção da interface, mantendo a arquitetura dos projetos e códigos incorporados nos aplicativos existentes baseados no *framework* UIKit.

1.2 Objetivos específicos

- Desenvolver objetos de interface declarativos para criar e personalizar as telas que compõem o aplicativo para iOS;
- Otimizar o desenvolvimento de estruturas complexas como listas e coleções, exigindo menos código para o desenvolvedor e sem afetar a experiência do usuário;
- Definir regras de portabilidade para o uso na estrutura da declaração de interface como forma de permitir que os objetos de interface escritos usando funções e métodos do UIKit possam ser facilmente integrados.

1.3 Metodologia

A implementação deste trabalho para responder à questão de pesquisa foi feita com base em pesquisas, artigos científicos e outros materiais para compor o referencial teórico. As definições e regras do *framework* serão obtidas a partir da documentação da Apple. Com os conceitos organizados, o *framework* foi implementado e, posteriormente, foram definidos os casos de teste, com base nas regras e classes implementadas. Após a conclusão do desenvolvimento dos módulos, são testadas e avaliadas as estruturas desenvolvidas e que compõem o *framework*.

1.4 Justificativa

A interface de um aplicativo é essencial para que o usuário possa interagir com o sistema, usando os controles e visualizando as informações, adaptando de acordo com as necessidades e habilidades do usuário. Cada interface tem total responsabilidade pelo sucesso de um produto, uma vez que representa todas as ações que o usuário possa realizar dentro do aplicativo, necessitando serem autoexplicativas. A trajetória de uma interface sempre pode ser aprimorada, ganhar funções mais complexas e facilitar o uso de determinado produto pelo usuário (COPADATA, 2020).

A programação imperativa é formada por um conjunto de instruções sequenciais que executam determinada tarefa, familiarizando com a definição de algoritmo. O seu uso na construção de interface pode ser definido como uma forma de programação de baixo nível, em que é possível o programador implementar qualquer recurso usando os métodos e regras definidas para cada objeto de interface.

Então, com as mudanças nas interfaces dos *softwares*, o uso da linguagem imperativa é uma ótima ferramenta para o programador, porém junto com a complexidade das interfaces, há o aumento na quantidade de instruções necessárias para atingir o resultado desejado (WATT, 2004).

A programação declarativa é formada por um conjunto de funções que desempenham um papel genérico dentro do código e que cada uma tem apenas um único objetivo. A concepção de cada função deve ser feita dado um objetivo, por exemplo configurar a cor de fundo dos objetos de interface, podendo ser implementada usando outras funções ou usando a linguagem imperativa para realizar o objetivo proposto. Com isso, o desenvolvimento de um *framework* nesse sentido precisa que o programador conheça profundamente as regras que compõem o desenvolvimento imperativo, para que possa definir as regras de sua solução declarativa. O maior esforço, nesse cenário, estará em otimizar as soluções para cada regra definida (SMARAGDAKIS, 2019).

A implementação do *framework* para interface declarativa exige uma reformulação estrutural do código e a compreensão do ambiente de desenvolvimento. Dessa maneira, serão propostas novas formas de implementar determinado recurso atendendo os requisitos de otimização das regras envolvidas, o que permite o enriquecimento de informações a respeito da programação de interface em qualquer sistema. Com isso, os resultados da integração da programação declarativa com o software, deve ser tema de discussão entre os pesquisadores, em que seja considerado as diversas formas de implementação para cada módulo de interface conhecido.

1.5 Organização do trabalho

No capítulo 1 apresentou-se uma introdução sobre o tema e os objetivos deste trabalho, citando as tecnologias que serão utilizadas.

O capítulo 2 apresenta os principais recursos de programação para o desenvolvimento do *framework* e cita alguns trabalhos envolvendo o paradigma declarativo.

O capítulo 3 desenvolve os conceitos e implementa algumas estruturas de programação considerando o ambiente de desenvolvimento.

O capítulo 4 implementa os principais objetos declarativos para a construção da interface de usuário no iOS.

O capítulo 5 analisa e valida com base nos casos de teste o desempenho do *framework* desenvolvido.

O capítulo 6 mostra os pontos positivos e negativos, apresentando sugestões para trabalhos futuros a partir deste trabalho.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta os principais conceitos para a compreensão do tema deste trabalho, como a programação declarativa, funcional e reativa, a linguagem de programação Swift e uma abordagem rápida do SO iOS. Por último, estão referenciados trabalhos relacionados e uma breve análise sobre eles.

2.1 Programação declarativa e programação imperativa

Um dos vários paradigmas de programação existentes é a programação declarativa. Sua definição engloba diversos outros conceitos que juntos compõem toda sua lógica e estrutura. Por isso, pesquisas nessa área não apenas podem ser feitas com base nesse paradigma, mas também em outros que a compõe, por exemplo, a programação lógica. As contribuições para a programação declarativa podem complementar do desenvolvimento de outros assuntos que a compõe, como o avanço da programação funcional, da programação lógica e outros paradigmas (LAMPSON, 2010).

Existem comparações para as linguagens de programação em relação ao nível de abstração que se aplica aos paradigmas de programação. A linguagem *assembly* flexibiliza e liberta a implementação dos algoritmos por conter um conjunto de instruções e atributos de código menor quando comparado as linguagens C ou Java. Por conta disso, com o aumento da complexidade do algoritmo, soluções escritas usando o *assembly* podem se tornar um verdadeiro quebra-cabeças na busca de encaixar todas as instruções bases para solucionar o problema. Essa lógica se aplica ao paradigma declarativo e o imperativo, uma vez que o primeiro deve ser usado para abstrair vários comandos da programação imperativa, enquanto o segundo deve ser usado para implementar soluções simples, flexíveis e otimizadas (FISCHER, 2010).

A programação declarativa se encaixa nesses mesmos conceitos e abstrai estruturas que a programação imperativa exige em sua programação. Dependendo do domínio compreendido pela solução declarativa, as estruturas algorítmicas condicionais e de repetição, como dos operadores aritméticos e lógicos, podem deixar de existir, sendo necessário analisar caso a caso. Esse nível de abstração permite concepções mais complexas, mas limita em termos de controle do algoritmo da solução. Por isso, deve ser feito um *fallback* das soluções declarativas, permitindo o

programador usar, quando necessário, a programação imperativa (LAMPSON, 2010; FISCHER, 2010; KMETIUK, 2018).

Segundo Wirth (1976), a programação imperativa define o programa como o resultado da combinação do algoritmo com as estruturas de dados. O algoritmo é um conjunto de instruções, que se executadas permitem a solução de algum problema. As estruturas de dados são representações de um tipo de dado ou um contexto, permitindo operações e armazenamento tanto em memória quanto em disco. Essas definições estão presentes em várias linguagens de programação, como o C e o C++ (O'Regan, 2008; FARRER, 2011).

A linguagem de programação C++ foi desenvolvida por Bjarne Stroustrup (2014), seguindo o paradigma de programação imperativa. A linguagem é um estado da arte para a computação, pois define desde estruturas simples até estruturas complexas como classes e sobrescrita de operadores. É uma linguagem referência para códigos otimizados e de baixo consumo de tempo de processamento, e também é utilizada em compiladores como no caso do compilador da linguagem Swift, escrito em C++. Por isso, a programação imperativa é uma base sólida para a programação, enquanto a programação declarativa é uma abstração sobre ela.

O gerador de analisador sintático, o *Yet Another Compiler Compiler* (Yacc), é escrito usando os padrões da programação imperativa, porém desde que foi criado, a linguagem de entrada, de domínio específico e de livre contexto, é padronizada usando os conceitos da programação declarativa. Com isso, o analisador usa a linguagem de entrada para encontrar padrões e gerar as árvores de contexto, para então encontrar a árvore de saída. De certa forma, o seu trabalho é lembrado como um dos primeiros que utilizou a linguagem declarativa em analisadores sintáticos (LAMPSON, 2010; JOHNSON, 1975).

O *Structured Query Language* (SQL) é outra referência para a programação declarativa para armazenamento de dados. O SQL é composto por três definições: (a) *Data Definition Language* (DDL), que define um conjunto de comandos que compõem o esquema do banco de dados; (b) *Data Manipulation Language* (DML), suportando operações com os dados; (c) *Data Control Language* (DCL), permitindo a configuração de segurança no banco de dados. O SQL permite que o desenvolvedor se preocupe em programar no contexto da linguagem declarativa, abstraindo interações e checagens como os comandos *se* e *senão* (SILVA; ALMEIDA; QUEIROZ, 2016).

O código SQL pode ser comparado com estruturas simples da programação imperativa o que permite compreender o quanto de código pode ser abstraído ao usar a programação declarativa. A Tabela 1 apresenta uma comparação de um comando escrito usando o SQL para seleção dos estudantes com idade maior que dezoito anos com os mesmos comandos usando a linguagem de programação Python, permitindo analisar a quantidade de passos necessários para chegar ao mesmo resultado. O SQL abstrai estruturas de repetição e condicionais, disponibilizando comandos como o *SELECT* e o *WHERE* para executar a seleção dos dados restritos a uma condição ou não.

Tabela 1 - Comparação entre código SQL (declarativo) e código Python (imperativo).

SQL	**Python**

"SELECT * FROM students WHERE age > 18;"	selectedStudents = []
	for student in students:
	    if student.age > 18:
	        selectedStudents.append(student)
	print(selectedStudents)

Fonte: Elaborado pelo Autor.

Com isso, não é preciso compreender como é feito a busca, a filtragem e a seleção dos dados usando a linguagem SQL. Também, fica abstraído onde os dados estão armazenados e qual estrutura de dados é utilizada. Assim, o programador se preocupa em o que deve ser feito ao utilizar soluções que fazem o uso da programação declarativa.

2.2 Programação funcional

A programação funcional possui conceitos que auxiliam na compreensão da programação declarativa, porém não é citada como um dos paradigmas que a compõem em sua definição. Ela pode ser comparada com uma equação matemática que possui as variáveis de entrada e as variáveis de saída, sem que o seu estado mude, tendo como característica de ser imutável. Na programação declarativa esses conceitos de imutabilidade podem ser utilizados parcialmente ou integralmente, de acordo com o objetivo da funcionalidade (STONE, 2018).

Devido ao escopo dos métodos serem restritos aos dados internos, isso é, sem acessar valores externos e alterar o estado do método ou da aplicação, eles são caracterizados como componentes *stateless*. Essa característica é inversa se

comparado com a programação orientada a objetos, em que os dados são encapsulados e alterados conforme a mudança dos estados da aplicação. Com isso, é possível prever o resultado dos métodos uma vez que o estado continua o mesmo. No caso da programação orientada a objetos, um método de encapsulamento simples fica dependente do estado atual da aplicação (MICHAELSON, 2011).

Essas características da programação funcional permitem classificar como um dos paradigmas recomendados para aplicações que vem se tornando mais complexas e referência para código modulares. Com a independência entre os códigos, é possível adequar à programação paralela, uma vez que os métodos não interferem e nem competem por recursos da aplicação. Além disso, o código se torna de fácil compreensão, já que os métodos funcionais têm suas variáveis bem definidas, não existindo outra forma de executar sem os valores de entrada especificados (JANSEN, 2019).

2.3 Programação reativa

Uma alternativa ao processamento de dados é a programação reativa. Sua estrutura lógica é de propagar alterações, de maneira assíncrona, em todo o código que esteja escutando aquela alteração. Com esse paradigma, é possível reduzir a complexidade do código para lidar com as variáveis, sendo uma alternativa para o processamento de dados, e desenvolver uma estrutura lógica mais simples para controle de tarefas e eventos assíncronos. Dessa forma, aplicações que se concentravam apenas no *thread* principal, podem, com o benefício da programação reativa, tornar o código dinâmico e suportar multitarefas, como fazer requisições na internet e processar eventos da interface de usuário, sem ocorrer competição por processamento no *thread* principal (ZAITSEV, 2017).

O uso da linguagem reativa em conjunto com a linguagem imperativa tem seus desafios, por conta dos *callbacks* de propagação de mudança serem assíncronos. Por serem assíncronos, são executados fora da *thread* principal e, por terem o escopo livre de contexto, eles mantém disponíveis apenas as variáveis globais. Isso é devido aos conceitos de sinal, utilizado na programação reativa, e de operações assíncronas. Os blocos que escutam as alterações aguardam, de maneira assíncrona, por um sinal para que seja executado o *callback* de alterações. Assim, é possível tanto criar uma variável reativa, onde as mudanças de valores são propagadas para toda a aplicação,

como transformar eventos do sistema reativos, observáveis de forma simples e descomplicada (SCHUSTER; FLANAGAN, 2016).

A programação tem grandes desafios quando é discutido o gerenciamento de *threads* e propagação de alterações. A programação reativa deve rastrear as dependências do fluxo de dados e automaticamente garantir a ausência de falhas. O RxSwift, um *framework* reativo para iOS, trata alterações simultâneas em uma única variável reativa usando contagem de atualizações enquanto uma está ocorrendo. Quando em um ciclo de propagação é gerado uma nova alteração, o RxSwift identifica como um erro de programação. Apesar desses desafios, a programação reativa é um paradigma que gerência eventos e mantem síncrono variáveis da aplicação, tornando o desenvolvimento dinâmico e flexível (DRECHSLER; MOGK; SALVANESCHI; MEZINI, 2018).

2.4 Linguagem de programação Swift

O Swift é uma linguagem de programação desenvolvida pela Apple e lançada em 2014, na Conferência Mundial de Desenvolvedores da Apple, do inglês *WorldWide Developers Conference* (WWDC). A linguagem foi idealizada para atualizar a antiga linguagem de programação, o Objective-C, utilizada no desenvolvimento de aplicações para diversas plataformas da Apple. Em seu projeto, os desenvolvedores usaram diversos conceitos modernos de ponteiros, valores nulos, sintaxe e os principais erros que os programadores cometem para, então, compilar uma nova linguagem de programação (SEL, 2020).

O projeto do Swift é de código aberto recebendo diversas contribuições em cada versão para tornar uma linguagem completa e com recursos funcionais. Desde o lançamento, a linguagem se tornou cada vez mais popular atingindo em 2020 a 9ª posição no mercado em janeiro de 2020, segundo o índice TIOBE. O Swift é uma linguagem com marcações de sintaxe objetivas, sem ponto e vírgula, com *callbacks* e de forte tipagem. As variáveis podem ser declaradas tanto com aspecto mutável quanto imutável usando as marcações *var* e *let*, respectivamente, e sem que haja a necessidade de declarar o tipo da variável, pois o compilador compreende por meio de inferência o tipo da variável (KACZMAREK; LEES; BENNETT, 2019).

A linguagem de programação implementa o conceito de *extension*, ou extensão, para que sejam declarados métodos, construtores e variáveis fora dos

escopos de: (a) *class*, ou classe; (b) *struct*, ou estrutura; (c) *enum*, ou enumerador; e (d) *protocol*, ou protocolo. Linguagens de programação como o C++ precisam que todas as propriedades estejam declaradas dentro da classe para que o compilador consiga calcular o tamanho de cada objeto. O Swift abstrai essa necessidade e permite que todos os tipos possam ser estendidos conforme a Figura 1 (CHAUDHARY, 2020; SWIFT, 2020).

Figura 1 - Extensão do objeto Double para conversão direta de distâncias.

```
1   extension Double {
2       var km: Double { return self * 1_000.0 }
3       var m: Double { return self }
4       var cm: Double { return self / 100.0 }
5       var mm: Double { return self / 1_000.0 }
6       var ft: Double { return self / 3.28084 }
7   }
8   let oneInch = 25.4.mm
9   print("One inch is \(oneInch) meters")
10  // Prints "One inch is 0.0254 meters"
11  let threeFeet = 3.ft
12  print("Three feet is \(threeFeet) meters")
13  // Prints "Three feet is 0.914399970739201 meters"
```

Fonte: Swift, 2020.

O Swift suporta os tipos genéricos em funções e em classes, permitindo que o código se torne menos dependente de tipos de variáveis declaradas, abstrato e reutilizável. Os tipos genéricos estão em diversos objetos do Swift como o Array<Element> e o Dictionary<Key, Value>, implementados pelo *framework* Foundation. Com isso, funções como *swap* podem ser declaradas com valores de entrada genéricos e, consequentemente, suportando qualquer tipo como parâmetro da função. (SWIFT, 2020).

Além dos tipos de dados concretos, existem no Swift os tipos abstratos, definidos pelos protocolos. Eles são modelos de objeto semelhantes às interfaces do Java em que são definidas as propriedades que um objeto concreto deve declarar, sendo aplicados em classes, estruturas e em enumeradores tanto de forma direta quanto usando as extensões dos objetos. Em um protocolo podem conter a declaração de métodos, de construtores e de variáveis de leitura ou de escrita, usando os atributos dinâmicos *getter* e *setter*, respectivamente. No caso de um objeto

estender um protocolo usando as extensões, o Swift não permite que variáveis sejam concretas e devem implementar os atributos *getter* e o *setter*; sendo apenas concretas quando o protocolo é estendido no escopo do objeto (MARTIN, 2019).

Os protocolos permitem associar um tipo genérico dentro do objeto, se tornando um tipo alias ou genérico. As implementações de ambas estruturas de dados pilha e fila podem compartilhar o mesmo protocolo coleção, com tipo genérico associado Element. Dessa forma, a implementação das classes pilha e fila com extensão ao protocolo coleção poderá definir o tipo Element como um tipo alias ou como um tipo genérico como no caso da estrutura Array. No caso de ser um tipo alias, a classe define que o Element seja apenas de um único tipo, podendo ser um tipo Int, String, e até mesmo um tipo FilaGenerica<Int> (GARCÍA et al., 2018).

Os objetos são armazenados na memória usando o mecanismo de contagem de referência automática, do inglês *Automatic Reference Counting* (ARC), que rastreia e gerencia a memória da aplicação. Assim, os desenvolvedores não precisam se preocupar em como os dados são armazenados e se ficou algum dado perdido na memória, porque o Swift se encarrega de criar e excluir regiões alocadas para as variáveis. Apenas as classes são gerenciadas usando a contagem de referência, uma vez que estruturas e enumeradores são tipos primitivos e toda vez que seu valor é atribuído a outra variável, o Swift cria uma cópia mutável ou imutável dependendo da marcação *var* ou *let* (TRIPP; HYDE; GROSSMAN-PONEMON, 2018; SWIFT, 2020).

Em cada atribuição de uma mesma classe, o Swift entende que deve ser somado o número de referência e, enquanto as variáveis referenciadas na classe existirem, o objeto não será destruído. O compilador propõe o uso de dois atributos ao declarar uma variável para lidar com o ARC, a marcação *weak* e a *unowned*. A primeira marcação torna a variável do tipo opcional, em que seu valor pode assumir valor nulo. A segunda marcação mantém a variável de tipo não nulo, porém o acesso deve acontecer enquanto a classe existir, caso contrário irá ocorrer um erro fatal (GRAY, 2016).

Esses recursos dinamizam o processo de desenvolvimento permitindo soluções mais complexas dependendo de cada aplicação. O Swift tem se expandido em diversas plataformas de desenvolvimento, como o Windows e o Linux, além de ter projetos que permitem a sua utilização na programação de aplicativos para o Android (SWIFT, 2020).

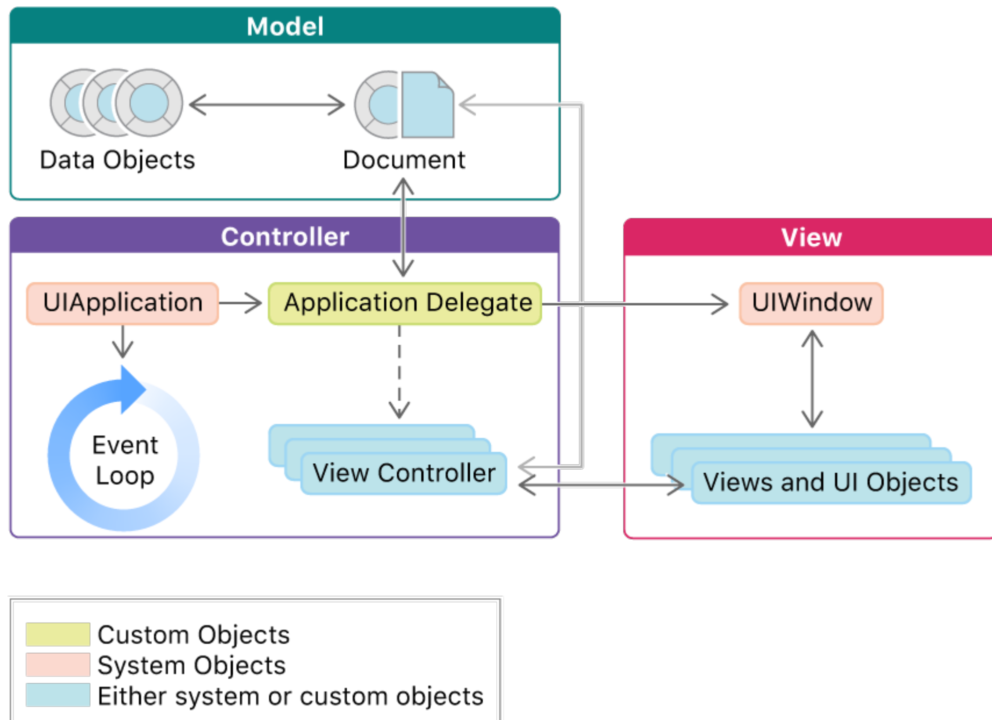
2.5 O Kit de Desenvolvimento de Software iOS

O iOS é um sistema desenvolvido pela Apple e utilizado pelos dispositivos da empresa desde 2007 com o lançamento do primeiro iPhone. O SO amadureceu nos conceitos de experiência do usuário e nos recursos do Kit de Desenvolvimento de Software, do inglês *Software Development Kit* (SDK). Com a chegada da oitava versão do SO e com o Xcode 6.0, os desenvolvedores tiveram acesso as ferramentas de desenvolvimento para escrever aplicativos usando a linguagem Swift, substituindo o Objective-C (NOVAC et al., 2017).

O SDK iOS é amplo e contém desde recursos simples como imagens a recursos que dão suporte a modelos de *Machine Learning*. O UIKit é um *framework* incluído no SDK iOS que implementa toda a parte da interface de usuário, suportando fontes, cores, imagens, gestos, janelas, controles e vários outros recursos de interface, recebendo o prefixo UI em suas classes. Os principais elementos que estão diretamente envolvidos na construção de interface são as *views* (UIView), os controladores (UIViewController), os gestos (UIGestureRecognizer), os controles (UIControl), as janelas (UIWindow) e a UIApplication (THAKKAR, 2019; APPLE, 2020).

A arquitetura utilizada pelo *framework* UIKit é a MVC, Figura 2, em que o programa está divido em três contextos. O *Model* define todas as regras de negócio do aplicativo, a *View* define a representação gráfica dos dados e o *Controller* gerencia os dados de acordo com a manipulação da *View*. O UIKit implementa duas classes que representam os controladores: (a) UIApplication, encarregada de iniciar o programa e monitorar a fila de eventos da aplicação; e (b) UIViewController, utilizada pelos desenvolvedores para gerenciar a aplicação (APPLE, 2020).

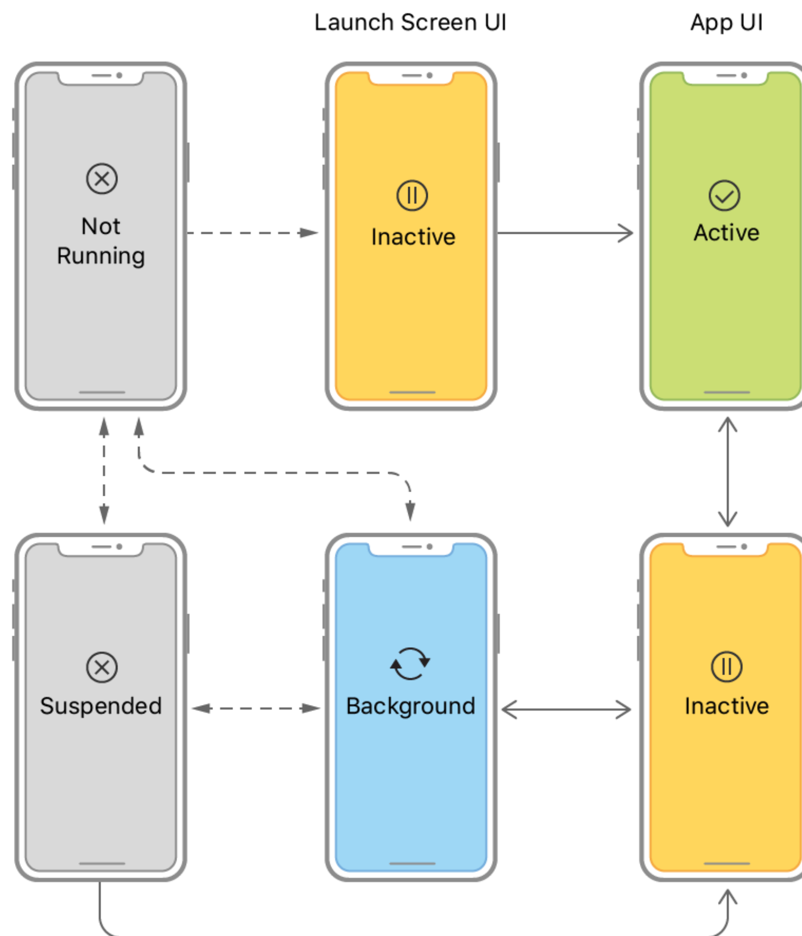
Figura 2 - A arquitetura MVC do UIKit.



Fonte: Apple, 2020.

Por meio do UIApplication, o UIKit notifica as alterações do estado da aplicação no sistema iOS. Todo programa inicia no estado *not running*, indo para o estado *inactive* que, posteriormente, muda para o estado *active* ou *background*. Os principais estados são: (a) *active*, quando o programa está em execução; (b) *background*, em que os recursos estão acessíveis, porém com a interface desativada; e (c) *inactive*, quando o aplicativo está executando com diversos recursos restringidos. Os estados *not running* e *suspended* referem ao momento de inicialização e ao momento em que não é possível executar o código, respectivamente. A Figura 3 mostra como os estados são alterados conforme a execução do programa (WANG, 2019).

Figura 3 - Ciclo de vida do UIKit.



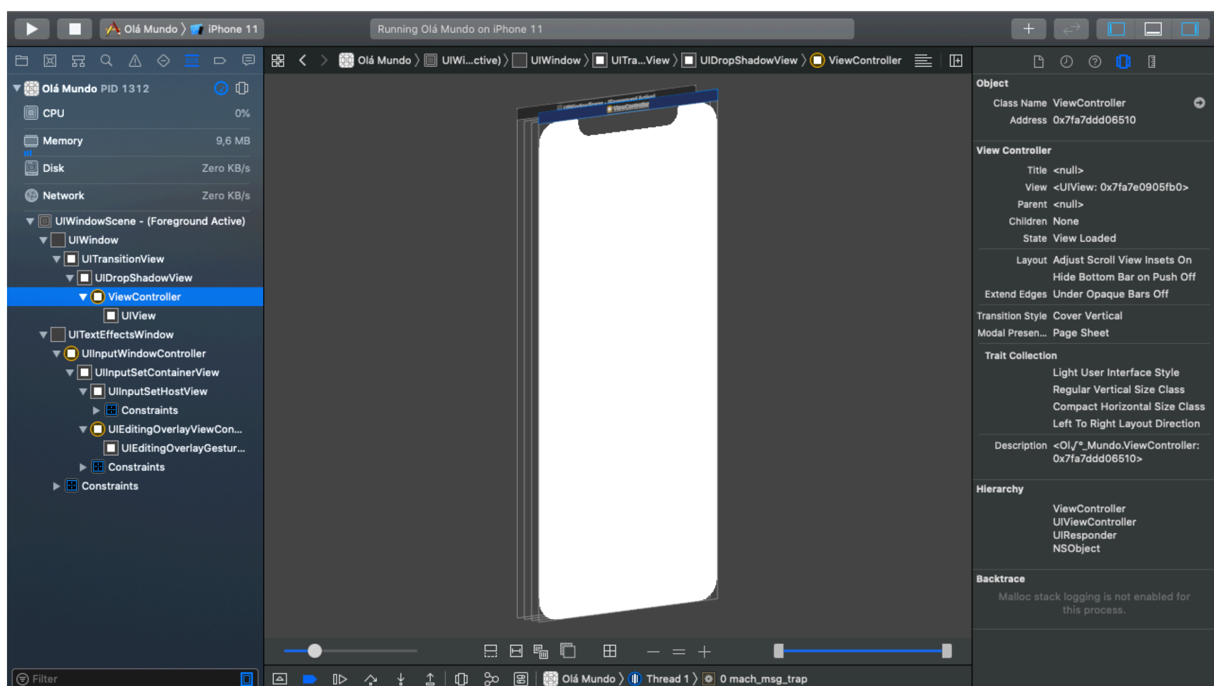
Fonte: Apple, 2020.

O UIApplication é uma classe estática que pode ser acessada em qualquer contexto usando sua propriedade estática *shared*. Dessa forma, podem ser tratados e acessados valores que refletem o estado atual da aplicação. Ações como abrir uma URL externamente também é implementado por métodos do UIApplication. Além disso, o UIKit exige a implementação do AppDelegate, uma classe utilitária que recebe os eventos do sistema como: (a) aplicativo iniciou; (b) aplicativo entrou em *background*; e (c) aplicativo recebeu uma notificação *push*. A partir do AppDelegate, o desenvolvedor informa a janela que a aplicação deve carregar iniciando o fluxo de interação com o usuário (ADAMS; LESEV, 2016).

As janelas, representadas pela classe UIWindow, estendem a classe UIView. Cada janela precisa ser configurada com uma UIViewController para que seja mostrado a interface na tela do usuário. Também, o desenvolvedor deve informar ao AppDelegate qual janela é a principal usando a sua propriedade interna *window* e

chamando os métodos da UIWindow makeKey() ou makeKeyAndVisible(). Desse modo, a janela será carregada com o controlador associado a ela e, então, é criada a hierarquia de interface. Por meio do Xcode, é possível fazer o *debug* da hierarquia e acompanhar quais elementos estão sendo mostrados, assim como valores de referência de memória, tamanho em altura e largura de cada elemento e propriedades específicas de cada *view*, conforme a Figura 4 (DOWNEY, 2016; APPLE, 2020).

Figura 4 - Debug da hierarquia de interface do aplicativo Olá Mundo.



Fonte: Elaborado pelo Autor.

A UIView encapsulada pela UIViewController é um objeto base para as *views* do UIKit. A classe implementa todo o controle da hierarquia de *views* com métodos para adicionar e remover *view*, acessar *views* filhas e animar mudanças usando o método animate(withDuration: TimeInterval, animations: () -> Void). A UIView também implementa propriedades como cor de fundo, largura, altura e transparência. Além disso, a *view* contém os seguintes métodos:

a) willMove(toSuperview: UIView?);
b) didMoveToSuperview();
c) didMoveToWindow().

O método (a) informa que a *view* está prestes a entrar na hierarquia. O (b) notifica que a *view* está na hierarquia mais próxima. Enquanto o (c) informa que a *view* está na hierarquia completa com uma janela conhecida. A partir desses três métodos, é possível gerenciar o estado atual na hierarquia de *views* (NEUBURG, 2017).

A classe UIControl compartilha os mesmos métodos da UIView, porém adiciona recursos de propagação de eventos. Com essa classe é possível notificar o sistema que eventos ocorreram como: (a) *valueDidChange*, utilizado para notificar que o valor alterou; (b) *editingDidBegin*, para quando a edição de algum valor começou; (c) *editingChange*, quando a edição mudou; e (d) *editingDidEnd*, utilizada para notificar que a edição terminou. Ao todo, são cerca de 20 eventos utilizados para notificar eventos da UIControl, por meio do método addTarget(Any?, action: Selector, for: UIControl.Event), que executa o Selector, uma estrutura do Objective-C para ponteiros de funções, informando que o evento ocorreu (APPLE, 2020).

As *views* são elementos primários na hierarquia gráfica e para gerenciar os estados delas como um grupo de sub hierarquias, o UIKit define a classe UIViewController. Essa classe é uma casca sobre a UIView que tem todas as propriedades necessárias para manipular a hierarquia. Segundo a documentação do UIKit, a UIViewController é usada para gerenciar a hierarquia de *views* e detalhar melhor o estado atual que a sub hierarquia se encontra. Além disso, os desenvolvedores não devem confiar nos métodos da UIView, pois eles não são tão precisos para representar o estado de cada elemento (APPLE, 2020).

A classe UIViewController possui um ciclo de vida e é usado em toda aplicação como um representante da arquitetura MVC. Quando um controlador é criado, nenhum método é executado, sendo dependente da primeira requisição para carregar a *view* que ela gerencia. A partir disso, é chamado o método informando que a *view* carregou para que o desenvolvedor possa fazer todas as configurações importantes relacionadas as regras da aplicação. Depois disso, o controlador notifica o estado atual da *view* quanto a sua hierarquia por meio dos métodos, conforme as Figuras Figura 5 e Figura 6 (WANG, 2019):

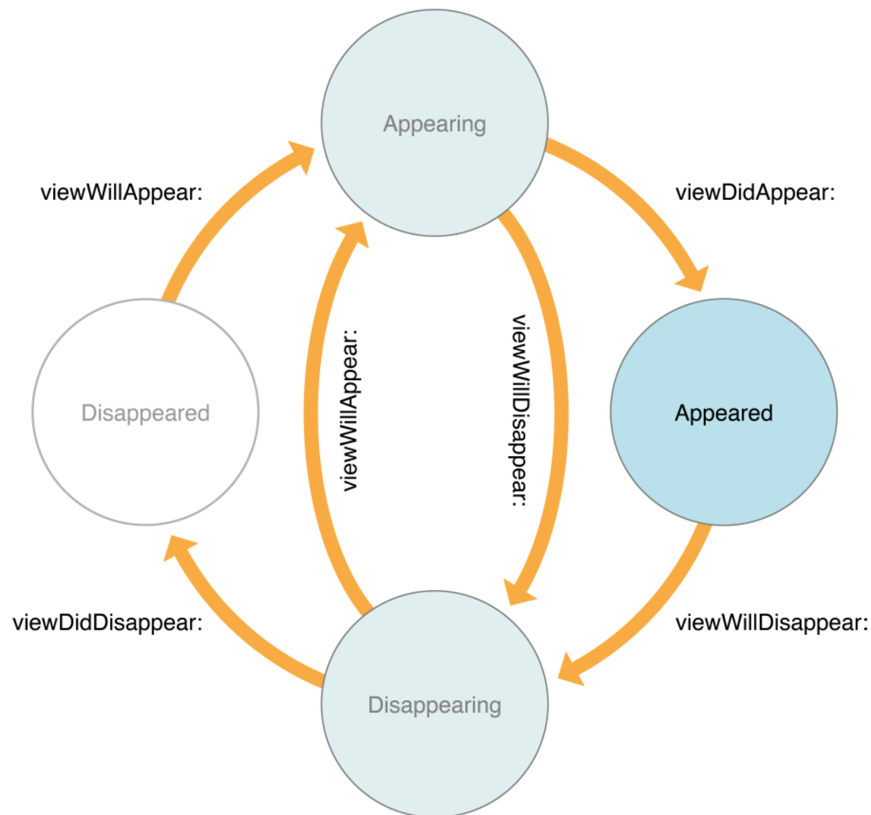
a) viewWillAppear(Bool);
b) viewDidAppear(Bool);
c) viewWillDisappear(Bool);
d) viewDidDisappear(Bool).

Figura 5 - Métodos da UIViewController que informam o estado atual da *view* na hierarquia.

```
1  import UIKit
2
3  class ViewController: UIViewController {
4
5      override func viewDidLoad() {
6          super.viewDidLoad()
7
8          // A view carregou
9      }
10
11     override func viewWillAppear(_ animated: Bool) {
12         super.viewWillAppear(animated)
13
14         // A view irá aparecer
15     }
16
17     override func viewDidAppear(_ animated: Bool) {
18         super.viewDidAppear(animated)
19
20         // A view apareceu
21     }
22
23     override func viewWillDisappear(_ animated: Bool) {
24         super.viewWillDisappear(animated)
25
26         // A view irá desaparecer
27     }
28
29     override func viewDidDisappear(_ animated: Bool) {
30         super.viewDidDisappear(animated)
31
32         // A view desapareceu
33     }
34 }
```

Fonte: Elaborado pelo Autor.

Figura 6 - Estados da UIViewController que informam o momento que a *view* se encontra na hierarquia.



Fonte: Apple, 2020.

Os principais componentes como a UIViewController e a UIView são classes de interface maduras e implementam recursos importantes para o *layout* da interface nos dispositivos a serem desenvolvidos pelo programador. O ciclo de vida no UIKit é um conceito que organiza a arquitetura dos objetos, criando momentos ideais para que trechos de códigos possam ser executados. A UIView contém uma série de métodos que são chamados pelo UIKit, montando o *layout* e configurando elementos em suas várias propriedades. Dessa forma, compreendendo esses conceitos básicos do UIKit, é possível ter uma visão geral da arquitetura do *framework* (APPLE, 2020).

2.6 Trabalhos relacionados

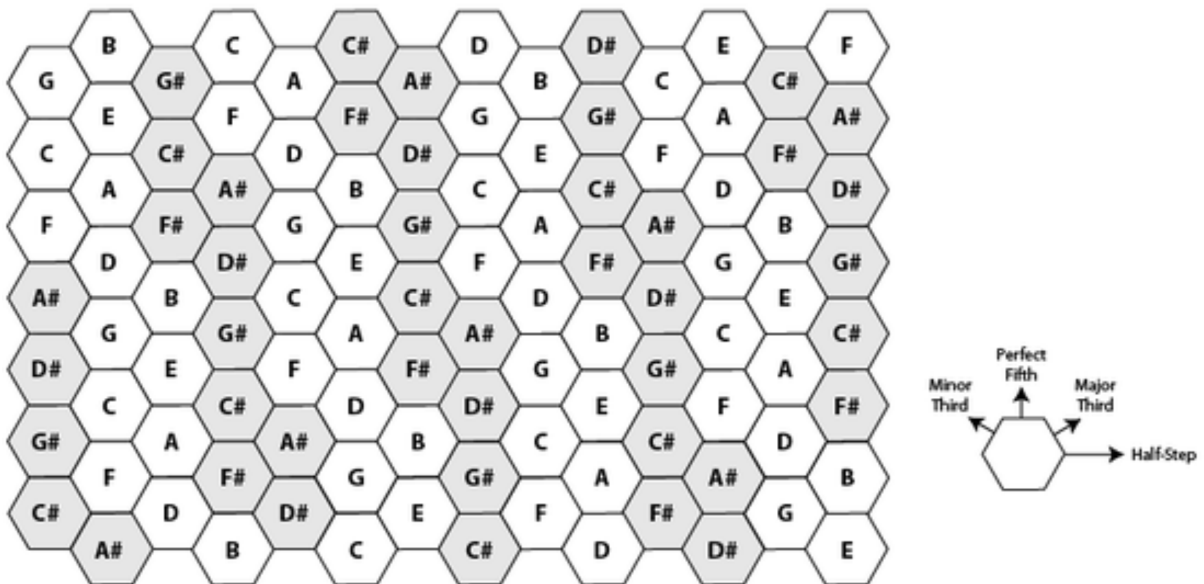
Essa seção descreve os trabalhos relacionados com o paradigma de programação declarativa e faz uma breve análise sobre eles.

2.6.1 The Arpeggigon: Declarative Programming of A Full-Fledged Musical Application

O artigo “The Arpeggigon: Declarative Programming of A Full-Fledged Musical Application” teve como objetivo implementar um software para produção musical. A música é composta de elementos declarativos, como a notação musical, o tempo e o fluxo síncrono de dados. Combinando a *Functional Reactive Programming* (FRP), o *Reactive Values and Relations* (RVR) e a interface de usuário, foi possível obter uma abordagem viável para o desenvolvimento de um software musical, conforme o paradigma de programação declarativa (NILSSON; CHUPPIN, 2017).

O *layout* do software Arpeggigon utiliza um arranjo de figuras hexagonais situando as notas musicais em uma lógica matemática, conforme a Figura 7. Foi necessário adaptar o projeto para os dispositivos com tela sensível ao toque (NILSSON; CHUPPIN, 2017).

Figura 7 - Layout hexagonais para notas musicais.



Fonte: Nilsson; Chuppin, 2017.

O software desenvolvido expressou de forma clara e precisa a lógica central da aplicação, tornando útil para composições musicais. Grande parte do código é declarativo, com sequência de definições de interface e como estão relacionados. O Arpeggigon apresentou resultado satisfatório de desempenho se comparado aos outros instrumentos musicais em relação a composição e execução da música (NILSSON; CHUPPIN, 2017).

2.6.2 Anel: Robust Mobile Network Programming Using a Declarative Language

O artigo “Anel: Robust Mobile Network Programming Using a Declarative Language” teve como objetivo suprimir erros de conexão com o auxílio da programação declarativa. A linguagem declarativa foi usada, com sucesso, para preencher as lacunas do desenvolvimento de softwares que não tratam os erros de conexão, gerando comportamentos inesperados nas aplicações (JIN; GRISWOLD; ZHOU, 2018).

Existem várias abordagens para testes de detecção de erros de conexão, informando a causa da falha no aplicativo, conhecido como defeitos de programação de rede, do inglês *Network Programming Defects* (NPD). Entretanto, mesmo em ambiente controlado, existem erros que apenas se manifestam quando a conexão de rede é intermitente. Apesar das bibliotecas para programação de rede existirem em grande número, até mesmo expondo as falhas aos desenvolvedores, os NPD ainda são predominantes (JIN; GRISWOLD; ZHOU, 2018).

Usando a linguagem ANEL, conforme a Figura 8, o programador adiciona ao código fonte uma instrução, de padrão declarativo, que retira a necessidade de tratar os erros NPD pelo desenvolvedor (JIN; GRISWOLD; ZHOU, 2018).

Figura 8 - Trecho de código usando a instrução ANEL, tornando o código tolerante a falhas.

```
View onCreateView() {
   …
   load();
}

void load() {
   @Anel_property({“UserReq”})
   AsyncHttpClient client=new AsyncHttpClient();
   RequestParams params = new RequestParams();
   client.get(apiUrl, params,
              new AsyncHttpResponseHandler());
}
```

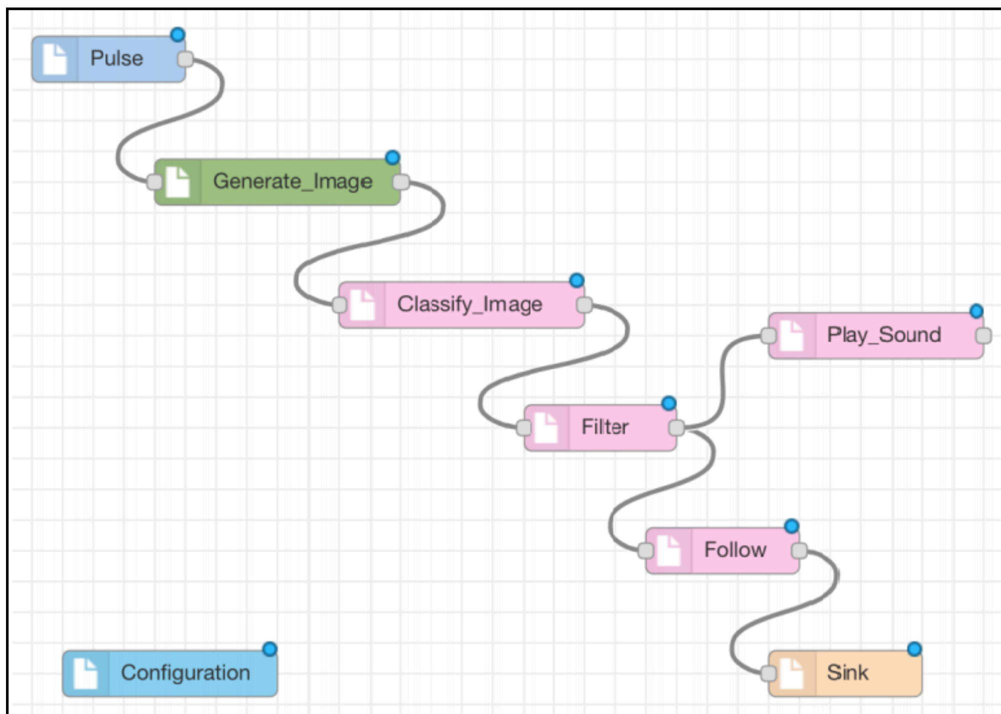
Fonte: Jin; Griswold; Zhou, 2018.

O estudo recrutou desenvolvedores com experiência inicial e constatou uma facilidade em compreender as instruções declarativas. O principal objetivo de suprimir o NPD das aplicações foi atingido (JIN; GRISWOLD; ZHOU, 2018).

2.6.3 DDFlow: Visualized Declarative Programming for Heterogeneous IoT Networks

O artigo “DDFlow: Visualized Declarative Programming for Heterogeneous IoT Networks” teve com objetivo implementar um *framework* usado na implementação de aplicações heterogêneas para Internet of Things (IoT), ou Internet das Coisas. Por meio da programação declarativa visual, estendida da ferramenta Node-RED, o *framework* faz uma abstração de macroprogramação, em que o programador se concentra com o que a aplicação deve fazer, simplificando o desenvolvimento de aplicações distribuídas, conforme o exemplo da Figura 9 (NOOR et al., 2019).

Figura 9 - Construção do algoritmo de exemplo usando o framework DDFlow.



Fonte: Noor; Tseng; Garcia; Srivastava, 2019.

Como resultado, o framework atingiu seu objetivo, facilitando o desenvolvimento de aplicações de redes heterogêneas IoT ao usar a programação declarativa, tornando o desenvolvimento intuitivo e de fácil compreensão (NOOR et al., 2019).

2.6.4 Using Declarative Programming in an Introductory Computer Science Course for High School Students

O artigo “Using Declarative Programming in an Introductory Computer Science Course for High School Student” apresenta uma metodologia e ressalta a importância de ensinar e aprender os conceitos que envolvem o paradigma de programação declarativa. Os cursos de introdução a ciência da computação são voltados para o paradigma de programação imperativa e a programação orientada a objetos. É importante que os alunos aprendam conceitos básicos da programação declarativa. O artigo apresenta a implantação de um curso de quatro semanas para o paradigma de programação declarativa (REYES et al., 2016).

Considerando que os alunos haviam estudado outros dois cursos de computação sobre fundamentos da programação e algoritmos, foi ministrado o curso sobre a programação declarativa. Os alunos estariam aptos a: (a) identificar os conceitos que envolvem um problema; (b) identificar relações entre os objetos e o contexto que eles estão inseridos no problema identificado; e (c) escrever soluções declarativas para o problema usando o paradigma de programação declarativa (REYES et al., 2016).

O curso foi ministrado para dezesseis alunos, com idade média de quinze anos, sendo nove homens e sete mulheres. Desse número de alunos, a maioria cursava a 10ª série, três cursava a 11ª série e dois a 12ª série. O curso foi ministrado por um professor de ciências da computação, com auxílio de três assistentes de ensino para suporte aos alunos com dúvidas (REYES et al., 2016).

Ao final do curso foi apresentado um questionário aos alunos sobre a experiência com o curso, o que resultou positivo indicando aptidão de todos os alunos para solução de problemas usando os conceitos da linguagem declarativa. Com isso, os autores pontuam a importância de estudar o paradigma de programação declarativa da mesma forma que se estuda as abordagens imperativas e orientadas a objetos. A resolução de um problema é simplificada, pois na programação declarativa se preocupa com as especificações do problema e não como o problema é resolvido, possibilitando o desenvolvimento de soluções mais complexas (REYES et al., 2016).

2.6.5 Análise dos Trabalhos Relacionados

A linguagem declarativa se aplica em diversas áreas da computação com resultados positivos quando comparados com as atuais implementações utilizando linguagens imperativas. Analisando os artigos apresentados, constatou-se o uso da programação declarativa para simplificar o processo de codificação, minimizar a complexidade em utilizar e integrar os recursos do software, bem como o desenvolvimento de um software para composição de música digital.

A programação declarativa necessita de um estudo aprofundado sobre o problema para que seja elaborado uma solução específica e de menor complexidade ao comparar com a solução imperativa. Dessa forma, os autores não explicitaram exatamente quais foram os obstáculos envolvendo a aplicação da programação declarativa e a lógica de implementação.

3 DESENVOLVIMENTO

Este capítulo apresenta os conceitos fundamentais para a implementação do *framework* declarativo, bem como as estruturas e algoritmos responsáveis por encapsular os objetos de interface do UIKit.

3.1 Programação declarativa

As próximas subseções apresentam os recursos da linguagem Swift para auxiliar na implementação do *framework* declarativo.

3.1.1 Callback

Um recurso importante na linguagem Swift são os *callbacks*, semelhantes as funções lambdas de algumas linguagens de programação, porém tem acesso as variáveis externas livres de contexto. Eles são um tipo de dado que podem ser armazenados em memória e executados como uma função. Existem dois tipos de *callbacks*: (a) executados dentro do fluxo de instruções de um bloco de código; e (b) executados fora do fluxo de instruções. O primeiro significa que ao declará-lo como parâmetro de uma função, ele será executado até o final da execução da função. O segundo significa que o *callback*, recebendo a marcação escape, no Swift usando a marcação *escaping*, poderá ser executado em algum momento futuro, não apenas enquanto a função existir ou o contexto em que foi criado existir.

A Figura 10 apresenta quando um *callback* é executado dentro da função e quando ele é executado fora da função. Quando não se utiliza a marcação de escape, é garantido ao compilador que o *callback* será executado dentro do escopo da função exemploCallbackSemEscape(() -> Void). Uma outra forma é usando a marcação de escape, em que a *callback* é executado depois que a função é retornada, ou como na Figura 11, executado após o período de 1 segundo gerando o resultado depois que todos os outros métodos foram retornados. Isso ocorreu por conta da classe DispatchQueue que gerencia as filas de execução assíncrona e síncrona do Swift. Dessa forma, um melhor aproveitamento dos *callbacks* está quando se utiliza para operações assíncronas como quando é feito uma requisição web, no qual a resposta será obtida no futuro, pois eles mantêm as variáveis externas disponíveis.

Figura 10 - Código comparativo entre *callback* com e sem a marcação de escape.

```
1  import Foundation
2
3  func exemploCallbackSemEscape(_ handler: () -> Void) {
4      handler()
5  }
6
7  func exemploCallbackComEscape(_ handler: @escaping () -> Void) {
8      DispatchQueue.main.asyncAfter(deadline: .now() + 1) {
9          handler()
10     }
11 }
12
13 print("Passo 1")
14
15 exemploCallbackSemEscape {
16     print("Callback sem escape")
17 }
18
19 print("Passo 2")
20
21 exemploCallbackComEscape {
22     print("Callback com escape")
23 }
24
25 print("Passo 3")
```

Fonte: Elaborado pelo Autor.

Figura 11 - Resultado da execução do código conforme a Figura 10.

```
Passo 1
Callback sem escape
Passo 2
Passo 3
Callback com escape
```

Fonte: Elaborado pelo Autor.

Um segundo benefício dos *callbacks* está em ser um tipo de dado, assim como os inteiros e os textos. Por ser um tipo, eles podem ser armazenados em memória, usados em algum momento desejado e possuem a característica de nunca expirar, podendo ser chamados várias vezes sem restrição. Entretanto, quando são utilizados com a marcação de escape e mantidos em memória, o Swift entende que as variáveis usadas dentro do *callback* devem ser seguradas em memória, usando o ARC ou por

meio de cópia de objetos, exigindo do desenvolvedor uma atenção maior ao usá-los, principalmente para evitar vazamentos de memória. Esses vazamentos são objetos que permanecem em memória mesmo quando não estão sendo mais utilizados, geralmente, quando ocorre um ciclo de referências, resultado em uma estrutura de dados que nunca será desalocada. Então, o controle desses objetos dentro do *callback* deve ser feito usando as marcações de acesso *weak* ou *unowned*.

3.1.2 Sintaxe declarativa

O paradigma de programação declarativa não especifica um padrão de sintaxe, além de existirem diversas formas de escrever códigos declarativos. No Swift, é possível usar a notação declarativa como a @IBOutlet que atua instanciando as variáveis dentro da classe controlador UIViewController de acordo com as especificações do arquivo *storyboard*. Outra maneira que enquadra a programação imperativa em programação declarativa, é usando funções implementadas dentro de um objeto retornando à referência do mesmo objeto, como uma função implementada no tipo de dado String que retorna o texto em maiúsculo. Então, usando essa ideia aliada a outros recursos do Swift, o padrão declarativo será moldado em funções que manipulam o objeto e retornam uma cópia do objeto ou simplesmente a referência dele.

A cópia automática dos objetos funciona apenas com variáveis do tipo estruturas e enumeradores com exceção as classes. Para obter o mesmo efeito com as classes, é preciso implementar o protocolo NSCopying que cria uma nova instância do objeto. No entanto, para este trabalho, quando o objeto for uma classe e for implementado um método declarativo no seu escopo, a função deve retornar a referência à classe, gerando redundância, para tornar possível a implementação do fluxo declarativo.

O objetivo principal está em desenvolver uma lógica em que o objeto declarativo será instanciado na primeira linha, enquanto as próximas linhas de código serão para configurar propriedades do objeto manipulado, usando o retorno redundante das funções. Além disso, deve ser possível, dependendo de cada caso, manipular o objeto internamente usando a programação imperativa, suportando operações usando o UIKit ou outras bibliotecas não declarativas.

Conforme a Figura 12, um exemplo de código da biblioteca AlertFactory, é possível criar um fluxo lógico para mostrar alertas no iOS usando o padrão declarativo, como resultado, o alerta é mostrado na tela, como na Figura 13. Usando os métodos do AlertFactory, é possível determinar o título, a mensagem e as ações que o usuário pode interagir. Na linha 9 da Figura 12 é instanciado a classe AlertFactory com o tipo genérico definido como UIAlertController, do UIKit, para apresentar no objeto ViewController. Entre as linhas 10 a 12 são definidos o título, a mensagem e a ação cancelar, que suprime detalhes para dispensar o alerta quando o usuário interagir com o botão. Para consumir o objeto e encerrar as instruções declarativas, é chamado o método present(), responsável por mostrar o alerta na tela (UMOBI, 2019).

Figura 12 - Código exemplo usando a programação declarativa para construir um alerta.

```
1  import UIKit
2  import AlertFactory
3
4  class ViewController: UIViewController {
5
6      override func viewDidAppear(_ animated: Bool) {
7          super.viewDidAppear(animated)
8
9          AlertFactory<UIAlertController>(viewController: self)
10             .with(title: "Olá Mundo!")
11             .with(text: "Este é um alerta do iOS usando a programação declarativa.")
12             .cancelButton(title: "Cancelar")
13             .present()
14     }
15 }
```

Fonte: Elaborado pelo Autor.

Figura 13 - Alerta construído usando a programação declarativa.

Fonte: Elaborado pelo Autor.

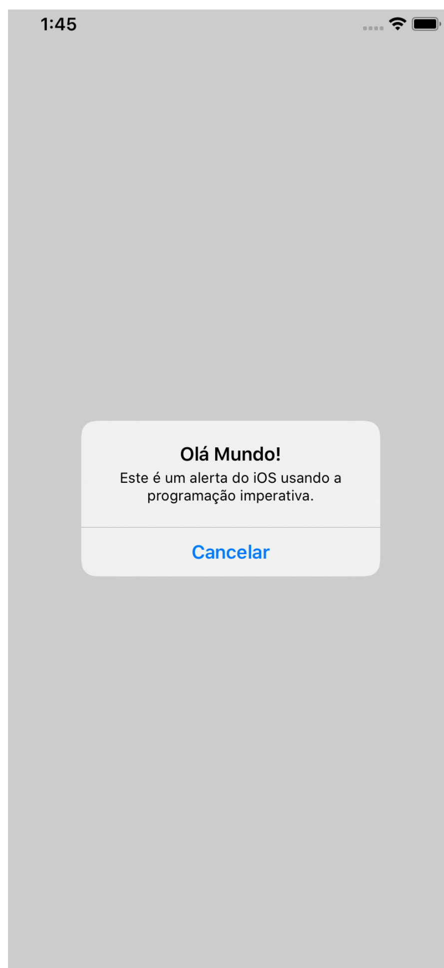
Em comparação com um código usando a classe UIAlertController, conforme a Figura 14, é possível identificar que a estrutura lógica da programação imperativa é organizada de forma diferente. Ao contrário de declarar, deve-se conhecer os métodos e propriedades da classe imperativa, além de seguir uma ordem de execução pré-definida. Nas linhas 8 a 12 são instanciados o alerta com título, mensagem e estilo. Entre as linhas 14 a 22 é definido e criado a ação cancelar. Por último, usando os métodos do UIKit da UIViewController, o alerta é mostrado, como na Figura 15.

Figura 14 - Código exemplo da UIAlertController para mostrar um alerta.

```
1  import UIKit
2
3  class ViewController: UIViewController {
4
5      override func viewDidAppear(_ animated: Bool) {
6          super.viewDidAppear(animated)
7
8          let alertController = UIAlertController(
9              title: "Olá Mundo!",
10             message: "Este é um alerta do iOS usando a programação imperativa.",
11             preferredStyle: .alert
12         )
13
14         let cancelAction = UIAlertAction(
15             title: "Cancelar",
16             style: .cancel,
17             handler: { [weak alertController] _ in
18                 alertController?.dismiss(animated: true, completion: nil)
19             }
20         )
21
22         alertController.addAction(cancelAction)
23         self.present(alertController, animated: true, completion: nil)
24     }
25 }
```

Fonte: Elaborado pelo Autor.

Figura 15 - Alerta construído usando a programação imperativa.



Fonte: Elaborado pelo Autor.

Com o uso da programação declarativa na criação de alertas no iOS, em comparação com a forma imperativa do UIKit, o algoritmo se torna mais abstrato quanto ao uso das instruções, uma vez que se utiliza os métodos declarativos para determinar as propriedades do objeto ou da aplicação. Um outro benefício está na reutilização de código, pois a implementação imperativa de métodos utilitários que define os alertas pode aumentar ainda mais a complexidade. Nesse caso dos alertas, o primeiro problema ao torná-lo genérico e reutilizável está em abstrair as ações, em que o UIAlertController exige a configuração do título do botão, o estilo e o *callback* executado no momento da ação, além do programador ter que dispensar o alerta em todas as ações, caso contrário, o alerta continua visível.

Outro problema está na configuração dos objetos que possuem propriedades definidas apenas no momento da sua inicialização, sendo solucionado ao implementar uma estrutura de duas camadas. A primeira camada é a classe declarativa, que salva

todas as propriedades de forma mutável, podendo ser atribuídas várias vezes, e a segunda camada é o objeto final, nesse caso as classes e estruturas imperativas. Um exemplo desse problema é a classe UIAlertController que contém a variável constante *preferredStyle* definida no momento da inicialização da classe. Com as funções declarativas, o objeto é encapsulado e instanciado posteriormente, no momento que for requisitado a sua leitura, permitindo, de forma redundante, chamar o método with(preferredStyle:) sem gerar erros de compilação e de execução.

O paradigma de programação declarativa tem um fluxo lógico definido conforme a Figura 16. Os objetos declarativos têm início, meio e fim. O primeiro ocorre quando o objeto é criado, informando nenhum ou poucos valores. O segundo são os métodos de configuração diretamente ligados ao contexto do objeto como, no caso dos alertas, sendo declarados as propriedades e comportamentos do objeto. O terceiro, marcado com borda pontilhada, ocorre diretamente ou indiretamente. O encerramento direto é quando o objeto construtor encerra o fluxo declarativo com funções que se auto consomem, usando a notação @discartableResult, ou quando o método retorna vazio, como a função present() do AlertFactory. O encerramento indireto é quando o fluxo declarativo é capturado por outro objeto, variável ou método.

Figura 16 - Fluxo lógico da programação declarativa.

Fonte: Elaborado pelo Autor.

Esse fluxo permite dividir a solução em estágios. O uso dessa lógica para objetos de interface se encaixa com o uso geral de qualquer objeto de interface baseado na UIView. O estágio início deve criar a *view* internamente com algumas propriedades especificas sendo atribuídas ou nenhuma, dependendo de cada caso. O estágio meio define as propriedades e comportamentos da UIView. O estágio fim encerra colocando a *view* na hierarquia do objeto construtor mais próximo ou na *view* mais próxima.

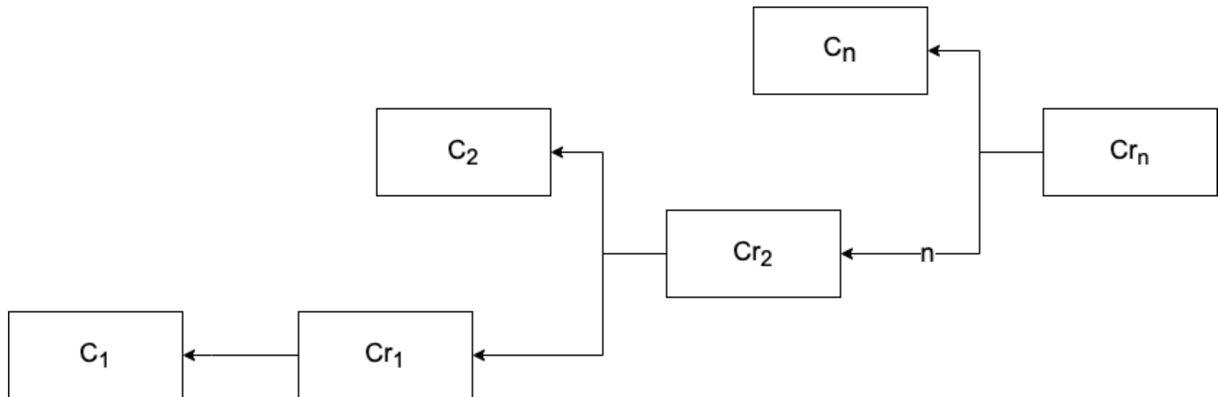
3.1.3 Empilhamento de chamadas

O uso da programação declarativa em conjunto com os *callbacks* permite isolar o objeto construtor do objeto manipulado, garantindo independência da UIView. A UIView é uma classe sujeita ao ARC, por isso, quanto menos complexidade de manipulação da UIView, menos problemas surgirão envolvendo vazamento de memória. Entretanto, os *callbacks* usados como parâmetro de funções não mantêm um escopo compartilhado e precisam ser armazenados ou associados a outra função para que sejam executados posteriormente.

O problema envolvendo esse cenário está em associar a programação declarativa à classe UIView, pois será necessário ter uma instância dessa classe armazenada dentro do objeto construtor e que servirá de parâmetro para os *callbacks*. Assim feito, não será possível garantir a independência da *view* e encapsulá-la para adaptar a programação declarativa. Quando se trata de alterar estados do objeto manipulado, uma melhor abordagem envolvendo a programação declarativa está em desassociar as alterações do fluxo declarativo. De maneira semelhante ao que é feito na biblioteca AlertFactory, as alterações são salvas em um envelope que são aplicadas ao objeto manipulado ao final da execução, evitando erros fatais de acesso ou problemas envolvendo as regras de uso do objeto.

Como a UIView tem várias propriedades, algumas delas dependem que outros objetos estejam instanciados para que possam ser manipulados e, a melhor forma de manter a independência é usando o empilhamento de chamadas dos *callbacks*. Os *callbacks* de escopo imperativo serão armazenados para serem executados no momento ideal. Assim, o armazenamento dos *callbacks* pode ser feito utilizando o conceito de recursividade, como estrutura de dados, apresentando característica de ser uma função que chama a si mesma dentro do mesmo escopo, conforme apresentado na Figura 17. Isso garante que o armazenamento dos *callbacks* mantenha com seu tipo próprio de função, no Swift como tipo primitivo, sem a necessidade de ser manipulado por uma classe, o que exigiria mais recursos de memória.

Figura 17 - *Callbacks* em estrutura recursiva.



Fonte: Elaborado pelo Autor.

O empilhamento de funções precisa que seja especificado uma variável de armazenamento *callbacksRecursivos* do mesmo tipo que o *callback* e de tipo opcional, no caso do Swift. Então, com a chegada do *callback* C_1, a função de empilhamento deve criar um novo *callback* Cr_1 com a junção do C_1 com o armazenado na variável *callbacksRecursivos*, no primeiro momento nulo, e armazenando novamente na variável. Repetindo essa função n vezes, o primeiro da pilha será o Cr_n que deve executar o C_n e o Cr_{n-1} até executar o Cr_1 que irá chamar o último da pilha, o C_1.

Dessa forma, aplicando ao contexto da UIView, todos os *callbacks* têm parâmetro de entrada de tipo UIView e retornam vazio. Como a UIView é uma classe e o ARC garante que não seja feito uma cópia dela e sim uma contagem de referência, o objeto manipulado é o mesmo em todos os cenários e as alterações são feitas no mesmo objeto.

3.1.4 Protocolos declarativos

O Swift implementa o protocolo como um modelo de objeto à classe ou ao objeto que o estender, com métodos e propriedades a serem declarados. Além disso, com a extensão é possível implementar variáveis e funções dentro do escopo do protocolo, permitindo a classe que o estender ter acesso a elas. Associando o protocolo com a programação declarativa, é esperado que algumas propriedades e métodos sejam obrigatórios, enquanto todas as regras devem ser implementadas dentro do protocolo, porém usando as extensões. Usando a extensão, conforme a Figura 18, o protocolo ImageType recebe a implementação da variável *image* fora do

seu escopo, e essa variável pode ser acessada por qualquer objeto que estender o protocolo.

Figura 18 - Protocolo com variável *image* implementada usando a extensão.

```
1  import Foundation
2  import UIKit
3
4  protocol ImageType: RawRepresentable where RawValue == String {
5      var rawValue: RawValue { get }
6  }
7
8  extension ImageType {
9      var image: UIImage! {
10         UIImage(named: self.rawValue)
11     }
12 }
13
14 enum Images: String, ImageType {
15     case addIcon = "icon.add"
16     case clockIcon = "icon.clock"
17     case calendarIcon = "icon.calendar"
18 }
```

Fonte: Elaborado pelo Autor.

O protocolo ImageType requer a implementação do RawRepresentable, um protocolo que permite a criação de modelos de objeto para enumeradores, apesar de ser permitido estender o protocolo em classes ou em estruturas. No escopo do protocolo ImageType é exigido a implementação da variável de apenas leitura *rawValue*, propriedade interna dos enumeradores. Com a extensão, a variável *image* é implementada e ela utiliza da variável *rawValue* do protocolo para criar uma imagem UIImage.

Como benefício, os protocolos, apesar de serem apenas um modelo de objeto, são um recurso da linguagem de programação Swift que generaliza estruturas e compartilha métodos entre as classes que o implementam. Um detalhe importante dos protocolos integrados a biblioteca Objective-C Runtime é a possibilidade de, usando as funções objc_getAssociatedObject(Any, UnsafeRawPointer) e objc_setAssociatedObject(Any, UnsafeRawPointer, Any?, objc_AssociationPolicy), armazenar valores em variáveis implementadas dentro das extensões, um recurso que é limitado pelo Swift ao impedir a declaração de variáveis concretas dentro de

extensões. Então, ao usar o protocolo como fonte de recursos para as classes que o estenderão, pode ser garantido toda uma estrutura por fora da classe, diretamente no protocolo, funcionado da mesma forma como se a classe estivesse herdando outra classe.

A programação declarativa encapsula os objetos imperativos e não deve aumentar a complexidade de implementação por parte dos programadores. O protocolo é uma forma segura de implementar recursos sem a necessidade de criar classes específicas para complementar a estrutura declarativa. Além disso, o protocolo auxilia a implementação, uma vez que os métodos e as variáveis obrigatórias são exigidos pelo compilador como atributos da classe. Isso garante que o objeto esteja em conformidade com a estrutura do *framework* e, ao mesmo tempo, por não herdar nenhuma classe, se torna um objeto singular que implementa o protocolo especificado.

Outro recurso importante associado aos protocolos são os tipos genéricos. O Swift permite que os protocolos tenham a declaração de tipos associados que podem estar restritos a algum outro tipo ou não. Conforme a Figura 19, o protocolo Number tem o tipo associado Value sem tipo definido, permitindo que qualquer tipo seja associado ao tipo do protocolo. Na extensão do protocolo Number é possível restringir o escopo para tornar disponíveis os métodos apenas quando o tipo associado Value for igual a um tipo ou estender algum outro protocolo ou classe. Nesse caso, são implementados os métodos toDouble() e toFloat() disponíveis apenas quando o tipo Value for igual ao tipo Int. Portanto, é possível desenvolver protocolos que implementam várias funções limitadas ao tipo associado.

Figura 19 - Protocolo com tipo genérico associado.

```
1  import Foundation
2
3  protocol Number {
4      associatedtype Value
5      var value: Value { get }
6  }
7
8  extension Number where Value == Int {
9      func toDouble() -> Double {
10         Double(self.value)
11     }
12
13     func toFloat() -> Float {
14         Float(self.value)
15     }
16 }
```

Fonte: Elaborado pelo Autor.

A interface de usuário contém diversos objetos que podem ser encapsulados pela programação declarativa, principalmente usando o conceito dos protocolos da linguagem Swift. Para isso, além dos protocolos simples, deve ser adicionado ao protocolo o tipo associado restrito ao tipo UIView, que é uma classe herdada por todos os outros objetos que compõem a interface de usuário. Então, um protocolo de tipo UIElement com tipo associado Body restrito a classe UIView, pode ter métodos específicos para quando o Body for igual a classe UIButton ou a classe UIStackView, garantindo que comportamentos específicos de cada objeto de interface seja mantido.

3.2 Composição declarativa

Essa seção define os principais componentes para abstrair os objetos que compõem o *framework* UIKit.

3.2.1 Encapsulamento da classe UIView

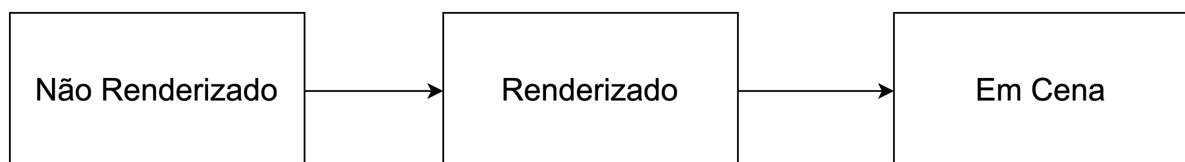
Implementações usando a UIView seguem um fluxo bem determinado. Retirando a construção de interface usando os *storyboards* e considerando apenas a programação imperativa, é possível identificar um padrão de programação. O primeiro momento é quando a *view* é acessada para configurar propriedades internas

independentes de outras variáveis. O segundo momento ocorre quando a UIView é manipulada com base na *view* pai definida, isso significa que a *view* está na hierarquia mais próxima. O terceiro momento ocorre quando a manipulação *da* UIView depende de algum elemento fora da hierarquia próxima, considerando tanto a hierarquia inferior quanto a hierarquia superior até a janela, representada pela classe UIWindow.

Os estágios da UIView serão definidos com base no ciclo de vida do objeto, implementado pelo *framework* UIKit. Esses estágios começam a se modificar quando o objeto é atribuído a uma *view* pai por meio do método addSubview(UIView). A partir desse momento, é disparado dois eventos internos do objeto filho. O primeiro método, willMove(toSuperview:), e o segundo método, didMoveToSuperview(). Posteriormente, quando a *view* mais próxima da raiz estiver na hierarquia da janela é chamado o método didMoveToWindow() de todos os objetos. Dessa forma, usando a estrutura do UIKit, é possível dividir o processo de renderização em três estágios, tornando possível configurar propriedades em momentos específicos, com base nesses três métodos da UIView.

Conforme a Figura 20, podem ser definidos os estágios: (a) não renderizado; (b) renderizado; e (c) em cena. Alinhado ao conceito de *callback*, cada estágio resulta na chamada de um *callback* específico, configurando a UIView no momento correspondente. Para que seja executado o *callback* no momento ideal, é necessário sobrescrever os métodos da UIView que melhor representam cada estágio, como na Figura 21, que sobrescreve o método willMove(toSuperview:) para o estado não renderizado. Assim, implementando a estrutura de *callbacks* e de estágios para cada *view* do UIKit, é possível abstrair a linguagem imperativa para uma solução da programação declarativa.

Figura 20 - Estágios de renderização da UIView.



Fonte: Elaborado pelo Autor.

Figura 21 - Implementação do estágio não renderizado na classe View.

```
1  import UIKit
2
3  class View: UIView {
4      var onNotRenderedHandler: ((UIView) -> Void)? = nil
5
6      override func willMove(toSuperview newSuperview: UIView?) {
7          super.willMove(toSuperview: newSuperview)
8
9          guard newSuperview != nil else {
10             return
11         }
12
13         self.onNotRenderedHandler?(self)
14     }
15 }
```

Fonte: Elaborado pelo Autor.

O UIKit conta com vários tipos de *views,* cada uma representando um elemento de interface importante para a experiência do usuário com a plataforma iOS. Implementar essa solução para cada uma delas geraria uma perda significativa de reutilização de estruturas e limitaria o código. Além disso, existem várias bibliotecas que implementam recursos extras que não estão presentes no UIKit e essa abordagem tornaria o código incompatível com a proposta de solução. Outra opção seria implementar os métodos e funções direto na UIView, o que tornaria o código genérico, porém iria poluir a estrutura da UIView. Lembrando que deve ser mantido a acessibilidade da programação imperativa e mínimas alterações nas classes bases.

Nesse caso, o uso dos protocolos auxilia a tornar o código independente de classes e métodos específicos. Isso permite criar os objetos do início e encapsular as instâncias de outros objetos internamente, como ao usar o tipo associado e declarar uma variável interna desse tipo para ser manipulada pelos métodos que compõem o protocolo.

3.2.2 Protocolo declarativo ViewCreator

Criar um objeto declarativo que encapsula um objeto imperativo se encaixa no contexto de objeto construtor que envolve o objeto concreto. Para garantir essa abstração da UIView, é especificado o protocolo ViewCreator e o protocolo UIViewCreator. O primeiro protocolo é a base para todos os métodos declarativos e

não possui tipo associado, não devendo ser estendido por outras classes. O segundo é referência para os objetos construtores, pois especifica o tipo associado View restrito à classe UIView. O protocolo UIViewCreator estende o protocolo ViewCreator, porque o compilador do Swift entende que um protocolo com tipo associado é genérico o suficiente para ser utilizado como tipo de dado concreto.

Isso significa que, ao definir uma variável como sendo do tipo UIViewCreator, é impossível deduzir qual será o tipo do objeto associado View. Dessa forma, as possíveis soluções para esse problema são: (a) estender outro protocolo sem tipo associado; ou (b) usar o protocolo como parâmetro genérico de funções ou classes, como representado na Figura 22. Nesse caso, como será feito a abstração da UIView utilizando os protocolos, é necessário definir um protocolo base e sem tipo associado para permitir que o protocolo possa ser utilizado como tipo de dado em funções e em variáveis.

Figura 22 - Protocolo com tipo associado utilizado como parâmetro genérico da função isEqual<Element>(Element, Element).

```
1  import Foundation
2
3  func isEqual<Element>(
4        _ first: Element,
5        _ second: Element) -> Bool where Element: Equatable {
6        return first == first
7  }
```

Fonte: Elaborado pelo Autor.

A implementação do protocolo ViewCreator utiliza os conceitos de *callback*, os conceitos de empilhamento de chamada e algumas funções da biblioteca Objective-C Runtime para armazenar os *callbacks* e o objeto UIView encapsulado. Os métodos principais são: (a) onNotRendered((UIView) -> Void); (b) onRendered((UIView) -> Void); e (c) onInTheScene((UIView) -> Void). Em cada um dos métodos, é reservado uma variável por meio do uso de extensão para armazenar os *callbacks* correspondentes, sendo marcadas como variáveis privadas e de acesso exclusivo do protocolo.

O fluxo lógico declarativo precisa que o objeto construtor esteja sempre referenciando a si mesmo. A Figura 23 utiliza o tipo Self como retorno das funções do protocolo declarativo, com isso o compilador deduz como sendo o mesmo tipo do

objeto que implementa o método. Então, caso seja declarado uma classe ContentView que estende o protocolo ViewCreator, a função onNotRendered(_:) tem o tipo de retorno Self traduzido pelo compilador como sendo o tipo ContentView. Além disso, para que a função retorne o tipo Self, é preciso retornar a variável implícita *self*, que é uma variável de contexto que referencia os métodos e as propriedades do objeto.

Figura 23 - Protocolo ViewCreator.

```
1  import Foundation
2  import UIKit
3
4  protocol ViewCreator {
5      /// When the current state of UIView is not rendered
6      func onNotRendered(_ handler: @escaping (UIView) -> Void) -> Self
7
8      /// When the current state of UIView is rendered
9      func onRendered(_ handler: @escaping (UIView) -> Void) -> Self
10
11     /// When the current state of UIView is in the scene
12     func onInTheScene(_ handler: @escaping (UIView) -> Void) -> Self
13 }
```

Fonte: Elaborado pelo Autor.

Por conta da diferença entre mutabilidade da estrutura e da classe, além da necessidade de atribuir valores as variáveis implementadas dentro de uma extensão, o protocolo ViewCreator precisa estender o tipo *class* que impede a implementação do protocolo por estruturas ou enumeradores. Dessa forma, é possível utilizar as funcionalidades da biblioteca Objective-C Runtime, como os métodos objc_getAssociatedObject(_:, _:) e objc_setAssociatedObject(_:, _:, _:, _:), que necessitam de uma referência mutável e de da instância do objeto para vincular o endereço da referência ao objeto, permitindo atribuir e ler valores como uma variável.

No caso do objeto ser uma estrutura, o Swift trata esses tipos primitivos usando cópia imutável, impossibilitando armazenar valores relativos nesses objetos. Quando um tipo primitivo é usado como parâmetro de funções ou atribuído a alguma variável, o Swift faz uma cópia e, dependendo do tipo da variável ser mutável ou imutável, é possível alterar valores internos do objeto.

3.2.3 Unindo o protocolo e a instância da UIView

O gerenciamento do objeto concreto pelo protocolo será feito de maneira independente. Os *callbacks* referentes aos estágios de renderização retiram qualquer necessidade de manipular a UIView diretamente. Por isso, o protocolo deve instanciar a UIView sob demanda usando um *callback* sem parâmetro de entrada, permitindo a existência do protocolo em memória independentemente da instância da UIView. Após a instanciação, é preciso tratar o ARC, pois ambos objetos são classes e uma referência forte bidirecional resultaria em um ciclo de memória. O objeto de interface precisa ter uma referência do objeto construtor em todo o momento de existência para garantir o correto funcionamento do *framework* e, o objeto construtor uma referência do objeto de interface para manipular os estados e hierarquia.

As variáveis que irão armazenar as classes sujeitas ao ARC precisam se comportar como um *switch*. Existem duas formas de armazenar valores no Swift: (a) de forma fraca, usando a marcação *weak*; e (b) de forma forte, sem marcação. No conflito entre UIView e ViewCreator existem dois cenários que a variável precisa mudar: (a) quando a UIView não estiver na hierarquia; e (b) quando a UIView entrar na hierarquia. No primeiro cenário, a referência da variável ViewCreator dentro da UIView deve ser fraca e da variável UIView dentro do ViewCreator forte. No segundo cenário, as referências devem inverter indo de fraca para forte e de forte para fraca, evitando um ciclo de memória por conta do ARC.

Para tornar uma variável um *switch* de valores, é preciso implementar o enumerador MemorySwitch com três casos: (a) weak(WeakOpaque); (b) strong(StrongOpaque); e (c) nil. Será necessário implementar o protocolo Opaque que estende o tipo *class* para que seja possível armazenar valores usando a marcação *weak*. A estrutura WeakOpaque contém uma variável *weak* que armazena classes que estendem o protocolo Opaque. A estrutura StrongOpaque não contém a marcação de referência fraca, sendo compreendido pelo Swift por ser uma referência forte. Dessa forma, a UIView e o ViewCreator devem estender o protocolo Opaque para serem armazenadas pelo MemorySwitch.

Os métodos utilitários do Objective-C Runtime, em alguns momentos, se tornam imprevisíveis gerando erros de acesso a valores que já foram removidos da memória. As estruturas apresentam pouco risco de serem removidas, uma vez que elas são constantemente copiadas e destruídas quando não se utiliza mais o objeto

dentro do escopo. Por isso, deve ser implementado um banco de variáveis utilizando o conceito de estrutura para que tanto a UIView quanto o ViewCreator possam armazenar as propriedades necessárias para abstrair a UIView em métodos declarativos.

As estruturas, por serem copiadas, precisam de uma variável do tipo classe para manter o comportamento de mutabilidade. Implementando a classe MutableBox, de acordo com a Figura 24, é possível atribuir várias vezes a variável *object* garantindo a imutabilidade da estrutura. Como a estrutura é copiada constantemente e as classes são referenciadas, a cópia não fará com que a estrutura perca o acesso ao valor atual armazenado pela classe MutableBox. Além disso, a classe conta com um tipo genérico para manter o tipo da propriedade manipulada. Com isso, apesar de criar instâncias extras para armazenar uma propriedade, é evitado o uso constante dos métodos do Objective-C Runtime e os problemas de acesso são minimizados.

Figura 24 - Classe MutableBox.

```
1  import Foundation
2
3  class MutableBox<Object> {
4      var object: Object
5
6      init(_ object: Object) {
7          self.object = object
8      }
9  }
10
```

Fonte: Elaborado pelo Autor.

Como é inviável atualizar diretamente as variáveis usando os métodos do Objective-C Runtime, a solução é implementar uma estrutura que agrupa todas essas variáveis. Unindo essa proposta a classe MutableBox, as variáveis mutáveis são encapsuladas por ela e a estrutura mantém sua característica imutável. Dessa forma, ao acessar a única variável implementada na extensão da classe, como a UIView ou o ViewCreator que é um protocolo do tipo *class*, e realizar operações seguras de leitura e escrita sem causar erros fatais na aplicação por conta dos métodos do *framework* Objective-C Runtime.

3.2.4 Classe de renderização

A classe que gerencia os estágios da UIView precisa executar os *callbacks* *notRenderedHandler*, *renderedHandler* e o *inTheSceneHandler* sempre que o estágio altera. Isso significa que todas as *views* filhas estão no estágio maior ou igual ao estágio da *view* pai. Não é possível o objeto ir para o estágio em cena e os filhos continuarem no estado não renderizado, porém o contrário pode acontecer. Então, a classe que o gerencia precisa informar para as *views* filhas que o estágio mudou, caso elas não tenham alterado. Isso garante que os objetos estejam sincronizados e em harmonia quanto ao estado atual dos elementos de interface.

Os *callbacks* dos estágios nem sempre são atribuídos antes da renderização mudar, o que deve ser entendido como um *callback* tardio que precisa ser executado. Pode ocorrer da *view* estar no estágio na cena e um *callback* não renderizado ficar pendente. Usando as regras do *thread* principal, que limita as alterações de interface apenas sob o comando dela, a classe deve enfileirar os *callbacks* tardios para serem executados no *thread* principal usando a OperationQueue, uma classe do Foundation. Assim, é mantido o sincronismo de operações exigido pelo UIKit.

A classe deve gerenciar os *callbacks* sob demanda e removê-los quando forem executados, evitando que o execute mais de uma vez. Quando um *callback* de renderização é atribuído, a classe deve entender que o estado está pendente e verificar se o estado já foi executado, se for verdade, é gerado uma solicitação para execução dos *callbacks* tardios usando a OperationQueue. Então, utilizando o conceito de recursividade para armazenar os *callbacks*, somente são executados quando a UIView ou a OperationQueue informar que o *callback* deve ser executado.

Por ser uma classe que armazena tanto os *callbacks* quanto o estado da UIView, é preciso que ela esteja sempre acessível por meio do ViewCreator. Além disso, a UIView que for encapsulada pelo objeto construtor deve sobrescrever os métodos do UIKit para alterar e executar os *callbacks* referente a cada estágio. Dessa forma, integrando todos os recursos, o ViewCreator é capaz de atribuir os *callbacks* à classe Render, que gerencia os estados da UIView, para que possam ser executados no momento ideal.

Ao sobrescrever os métodos da UIView e para manter uma reutilização de código por outras *views* implementadas, deve ser implementado a estrutura RenderManager. Essa estrutura é instanciada com uma referência da classe de

interface, encontrando o objeto construtor associado àquela *view*. Os métodos que compõem essa estrutura recebem os mesmos nomes das funções que devem ser sobrescritas pela UIView. Então, quando a UIView chama o método willMove(toSuperview:) referente ao estágio não renderizado, o RenderManager pode verificar regras e informar ao Render do ViewCreator associado a UIView para executar o estágio não renderizado. Dessa forma, a sobrescrita dos métodos das classes convergem para os métodos do RenderManager, compartilhando o mesmo bloco de instrução.

3.3 Estrutura e objetos de interface do UIKit

A definição de estágios de renderização permite implementar os principais objetos de interface e respeitar a estrutura de interface implementada pelo UIKit. Essa seção apresenta a árvore de objetos de interface e define os objetos construtores que encapsulam as principais classes do UIKit.

3.3.1 Hierarquia de objetos de interface

A hierarquia de interface é um conjunto de objetos de interface que são organizados usando a estrutura de dados em árvore, sendo possível obter a raiz e as folhas que representam os objetos de interface. A raiz principal do UIKit é a classe UIWindow que inicia a composição da tela. Apesar de ser um elemento único, o *framework* permite que em uma única tela possa existir mais de uma janela visível. A subárvore é formada por objetos da classe UIView. Essa estrutura é importante para a interface, pois permite encontrar objetos tanto em direção da raiz ou em direção das folhas.

Com a abstração da UIView pelo ViewCreator, o framework precisa fazer buscas de elementos únicos de interface que estão associados a classes específicas do UIKit. A UIViewController contém propriedades fundamentais para o controle da tela, transições e informa valores para outras UIViewController. Retirar essa dependência, em muitos casos, exigirá a reescrita do UIKit para transferir toda a carga que o controlador carrega para a UIView. Dessa forma, elementos importantes de interface perderiam sua função e as características do sistema seriam aproximadas,

já que o UIKit implementa recursos importantes de transição e de navegação nos aplicativos.

Com o uso da árvore de interface, os objetos específicos, como a UIViewController, UINavigationController e outros, podem ser mantidos. Quando o ViewCreator for atribuir alguma propriedade que depende desses objetos, apenas será preciso buscá-los na hierarquia.

3.3.2 Objetos principais da classe UIViewController

A classe UIViewController é base para diversas classes do UIKit com funções envolvendo a apresentação da *view* na hierarquia. Existem controladores como o UINavigationController, o UITabBarController, o UIPageController e o próprio UIViewController que é usado para implementar os controladores que descrevem as regras de negócio do aplicativo. No caso do ViewCreator, quando uma propriedade restrita da classe dos controladores é requisitada, deve ser buscado a UIViewController mais próxima para acessar e atribuir os valores desejados.

A classe UICHostingController estende a classe UIViewController e implementa o ambiente para que o ViewCreator possa entrar na hierarquia de interface. O carregamento da UIView pelo controlador é realizado pelo UIKit sob demanda, quando a variável *view* é acessada pela primeira vez ou quando o método loadViewIfNeeded() é chamado. Posteriormente, é executado o método viewDidLoad(), notificando que a *view* está carregada. É nesse momento que o ViewCreator é atribuído a hierarquia, com a UICHostingController requisitando a UIView encapsulada pelo objeto construtor e a adiciona como *view* filha da *view* do controlador.

O protocolo NavigationRepresentable é uma abstração dos objetos formados pelo UINavigationController. Esse protocolo implementa as ações de transição do navegador chamando os métodos nativos do UIKit, como o pushViewController(UIViewController, animated: Bool), o popViewController(animated: Bool) e outros. A classe UICNavigation implementa esse protocolo para ser utilizado durante a declaração da interface. Como o controlador de navegação pode ser implementado por outras bibliotecas e pelo programador do projeto, a classe UICNavigation pode ser personalizada usando o protocolo UICNavigationExtendable para instanciar outra classe que implemente a UINavigationController.

A manipulação da classe de navegação pelos controladores é feita usando a variável *navigationController*, acessando os métodos e propriedades implementadas pela classe UINavigationController. O encapsulamento da classe de navegação pelos protocolos declarativos impede o acesso direto a ela, principalmente por conta da falta do controlador de *view*. Então, o protocolo ViewCreator contém a variável *navigation* que busca na hierarquia o objeto construtor do tipo NavigationRepresentable.

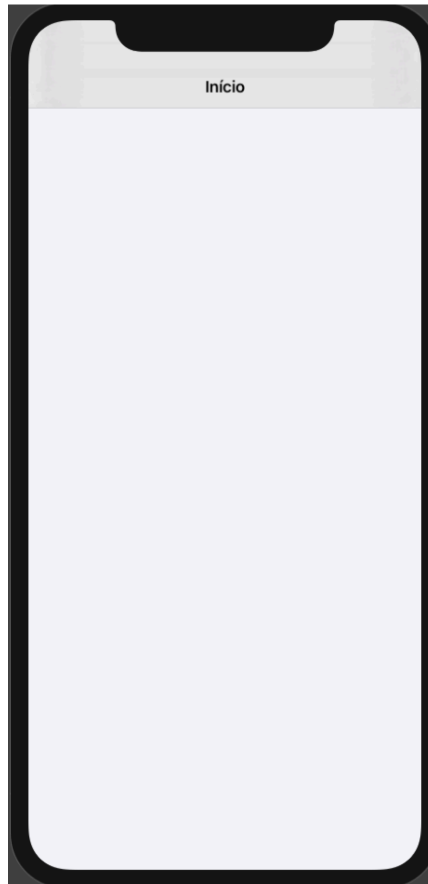
Com a classe UICNavigation, foi implementado uma interface de usuário vazia como título “Início” na barra de navegação. Conforme as Figura 25 e Figura 26 é possível utilizar a UINavigationController para criar fluxos de navegação. Estão expostos vários métodos para customizar o controlador de navegação, como as funções navigation(leftButton: () -> ViewCreator), navigation(rightButton: () -> ViewCreator), navigation(titleView: () -> ViewCreator) entre outras.

Figura 25 - UICNavigation com título igual a "Início" e *view* vazia com cor de fundo cinza claro.

```
1  import UICreator
2
3  class ContentView: UICView {
4      var body: ViewCreator {
5          UICNavigation {
6              UICSpacer()
7                  .backgroundColor(.systemGray6)
8                  .navigation(title: "Início")
9          }
10     }
11 }
```

Fonte: Elaborado pelo Autor.

Figura 26 - Execução do UICNavigation com título igual a "Início" e *view* vazia com cor de fundo cinza claro.



Fonte: Elaborado pelo Autor.

Da mesma forma que foi implementado o objeto construtor para navegação é implementado o objeto construtor UICTab, com o protocolo personalizável UICTabExtendable. O controlador UITabBarController cria uma barra inferior que controla qual UIViewController é mostrada por ele. Para isso, o UIKit exige que o controlador informe e manipule o UITabBarItem, uma outra variável incluída no escopo dos controladores. Essa dependência limita os objetos construtores a manipular o *tabBarItem* dos controladores e dificulta o fluxo lógico de implementação, caso essas manipulações ocorressem a partir de chamadas de métodos do ViewCreator.

A instanciação da classe UICTab exige que o parâmetro do construtor seja um vetor de UICTabItem. Esse objeto implementa todas as propriedades que podem ser configuradas no UITabBarItem como título, cor e imagem, além de definir um *callback* que retorna uma instância do ViewCreator. Dessa forma, o UICTab consegue montar todos os controladores a partir do UICHostingController para informar à classe UITabBarController os controladores gerenciados por ele. Outro benefício do

UICTabItem é que ele retorna o objeto UITabBarItem preparado para ser atribuído ao respectivo controlador.

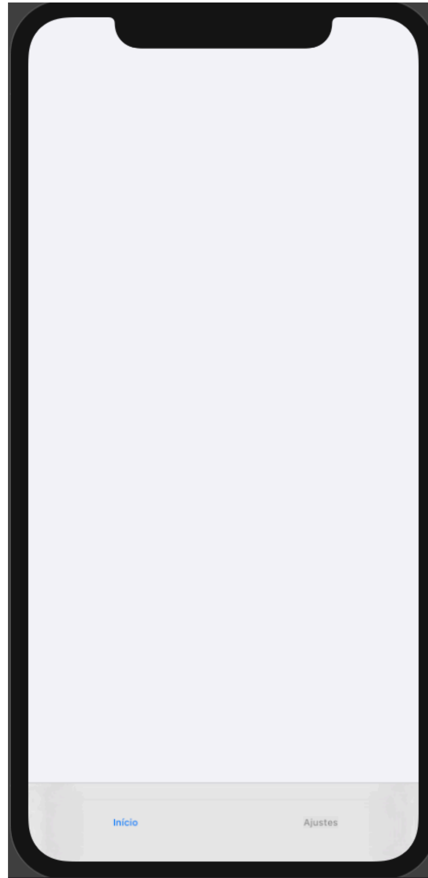
Com a UICTab e o UICTabItem foi implementado uma interface com duas *views* para formar a navegação por *tabs*, uma com o título “Início” e outra com o título “Ajustes”. Apesar de não conter informação, ambas as telas contêm o mesmo estilo, conforme as Figura 27 e Figura 28, sendo possível modificar as propriedades do UITabBarItem usando as funções públicas do ViewCreator como a tabBarItem(image: UIImage?), a tabBarItem(selectedImage: UIImage?) e outras.

Figura 27 - UICTab com duas views "Início" e "Ajustes".

```
1  import UICreator
2  
3  class ContentView: UICView {
4      var body: ViewCreator {
5          UICTab {
6              UICTabItem(title: "Início") {
7                  UICSpacer()
8                      .backgroundColor(.systemGray6)
9              }
10 
11             UICTabItem(title: "Ajustes") {
12                 UICSpacer()
13                     .backgroundColor(.systemGray6)
14             }
15         }
16     }
17 }
```

Fonte: Elaborado pelo Autor.

Figura 28 - Execução do UICTab com a view "Início" selecionada.



Fonte: Elaborado pelo Autor.

Além dessas classes construtoras, a *view controller* também pode se tornar um objeto declarativo, por meio do protocolo UICViewControllerRepresentable. Esse protocolo exige a implementação do método makeUIViewController() retornando o tipo associado ViewController. Dessa forma, nos casos onde a *view controller* é de terceiros ou contém uma estrutura complexa, ainda é possível utilizá-la no fluxo declarativo.

3.3.3 Objetos principais da classe UIView

Os objetos de interface do UIKit são classes que herdam a classe UIView. Essa classe implementa diversas propriedades envolvendo a interface, como a cor de fundo, o tamanho do quadro, a posição e outras. Também, ela define a hierarquia controlando quem é a sua raiz mais próxima e quais são suas filhas. Os principais objetos que herdam a UIView são as classes: (a) UIStackView; (b) UIScrollView; (c) UILabel; (d) UIImageView; (e) UIButton; e (f) UITextField.

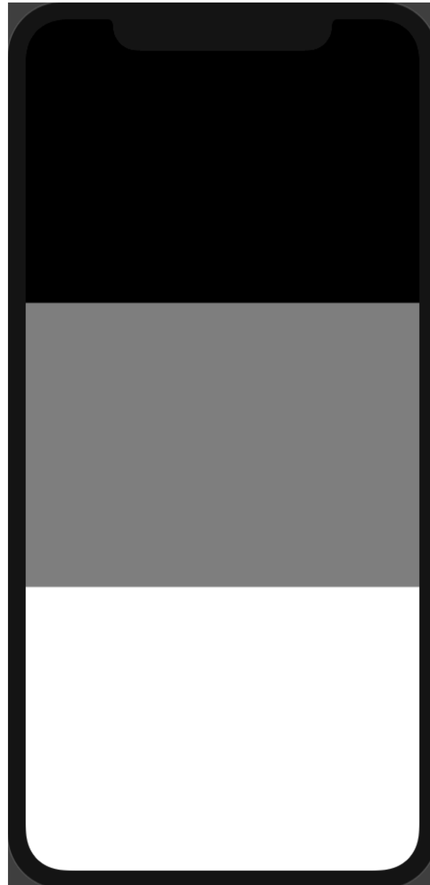
A classe UIStackView é encapsulada pelo objeto construtor UICStack, que é instanciado com um vetor de objetos construtores. A UIStackView implementa arranjos de disposição de *view* com atributos para definir o espaçamento entre elas, a forma que serão distribuídas, o alinhamento e a direção. As Figura 29 e Figura 30 mostram a implementação utilizando a UICVStack, com disposição vertical, um arranjo de três *views* com cores de fundo variando da cor escura para a branca. S função distribution(UIStackView.Distribution) foi usada para distribuir as *views* igualmente dentro do *frame* da UIStackView. Além dessa função, existem as funções spacing(CGFloat), alignment(UIStackView.Alignment) e axis(NSLayoutConstraint.Axis) que complementam o comportamento da *view*.

Figura 29 - UICVStack com três *views* distribuídas igualmente.

```
1  import UICreator
2
3  class ContentView: UICView {
4      var body: ViewCreator {
5          UICVStack {
6              UICSpacer()
7                  .backgroundColor(.black)
8
9              UICSpacer()
10                 .backgroundColor(.gray)
11
12             UICSpacer()
13                 .backgroundColor(.white)
14         }
15         .distribution(.fillEqually)
16     }
17 }
```

Fonte: Elaborado pelo Autor.

Figura 30 - Execução da UICVStack com três *views* distribuídas igualmente.



Fonte: Elaborado pelo Autor.

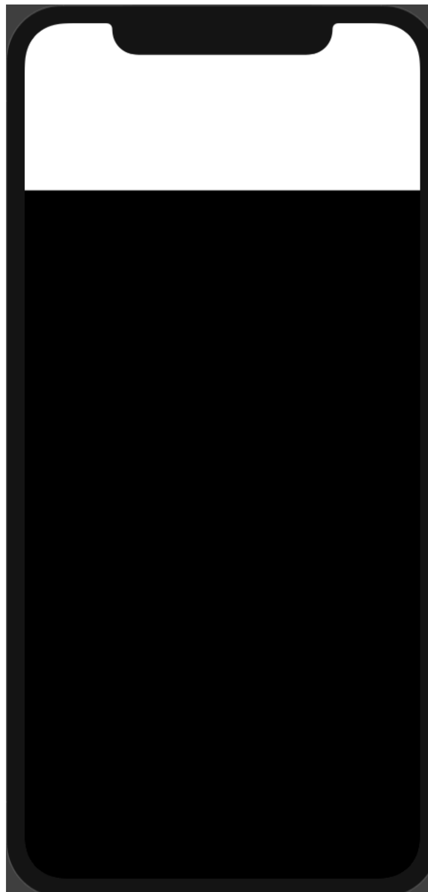
A UIScrollView mostra um conteúdo de tamanho variável que não obedece os limites do *frame* disponível para a *view*, se deslocando a partir da interação do usuário com a tela. O seu comportamento é definido pelo programador, porém está restrito ao conteúdo que é mostrado por ela, esperando três tipos de comportamento: (a) rolagem vertical; (b) rolagem horizontal; e (c) rolagem nas diagonais. As Figura 31 e Figura 32 utilizaram a UICVScroll para mostrar um conteúdo menor, de altura igual a 175, resultando em uma *view* branca que ocupa uma faixa da tela. A UICScroll implementa algumas funções para a UIScrollView, como a insets(behavior: UIScrollView.ContentInsetAdjustmentBehavior), alwaysBounce(vertical: Bool), content(insets: UIEdgeInsets), dentre outras.

Figura 31 - UICVScroll com uma *view* de altura igual a 175.

```
1   import UICreator
2
3   class ContentView: UICView {
4       var body: ViewCreator {
5           UICVScroll {
6               UICSpacer()
7                   .backgroundColor(.white)
8                   .height(equalTo: 175)
9           }
10          .backgroundColor(.black)
11      }
12  }
```

Fonte: Elaborado pelo Autor.

Figura 32 – Execução da UICVScroll com *view* de altura igual a 175.



Fonte: Elaborado pelo Autor.

A UILabel renderiza textos com propriedades para definir a fonte, a cor do texto, alinhamento e outras. O UICLabel encapsula a UILabel definindo texto e a cor, através de funções como a text(String?), textColor(UIColor), font(UIFont) e outras que configuram propriedades da view. As Figura 33 e Figura 34 representam a implementação utilizando UICLabel para mostrar o texto “Olá Mundo!”, com a fonte do texto igual ao estilo *title1* e alinhamento centralizado.

Figura 33 - UICLabel com o texto "Olá mundo!".

```
1  import UICreator
2
3  class ContentView: UICView {
4      var body: ViewCreator {
5          UICLabel("Olá Mundo!")
6              .textAlignment(.center)
7              .font(.title1)
8      }
9  }
```

Fonte: Elaborado pelo Autor.

Figura 34 - Execução da UICLabel com o texto "Olá mundo!".

Fonte: Elaborado pelo Autor.

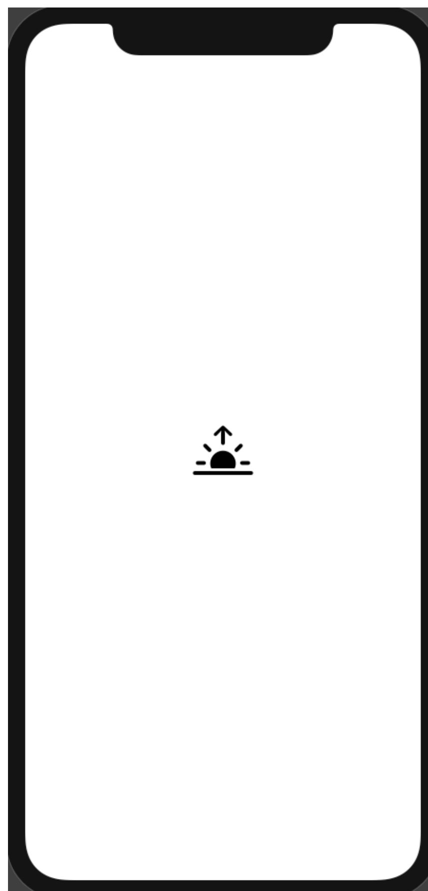
A UIImageView renderiza o objeto UIImage na tela. Com ela, é possível definir como a imagem será escalonada e a cor de tingimento da imagem. A UICImage é um objeto construtor que encapsula a UIImageView e implementa as funções para configurar as propriedades da classe de interface, como os métodos image(UIImage?), tintColor(UIColor?), content(mode: UIView.ContentMode). Conforme as Figura 35 e Figura 36, foi renderizado uma imagem de sol nascente na tela com altura e largura igual a 75.

Figura 35 - UICImage com símbolo do sol nascente.

```
1  import UICreator
2  import UIKit
3
4  class ContentView: UICView {
5      var body: ViewCreator {
6          UICCenter {
7              UICImage(image: UIImage(systemName: "sunrise.fill"))
8                  .height(equalTo: 75)
9                  .aspectRatio()
10                 .tintColor(.black)
11                 .content(mode: .scaleAspectFit)
12         }
13     }
14 }
```

Fonte: Elaborado pelo Autor.

Figura 36 - Execução da UICImage com símbolo do sol nascente.



Fonte: Elaborado pelo Autor.

A classe UIButton representa um botão e herda a classe UIControl, implementando funções de controle do UIKit. Representa um objeto de interface que pode ser personalizado combinando imagem e texto. Além disso, o botão define os

estados selecionados, em foco e desativado, permitindo que as propriedades sejam diferentes para cada estado. Por ser uma classe do UIControl, a *view* consegue gerar eventos de controle que podem ser capturadas por outras *views*. O principal evento envolvendo os botões ocorre quando o usuário seleciona o botão, gerando o evento *touchUpInside*. O objeto construtor é o UICButton que pode ser instanciado de várias formas, conforme a implementação do UIButton e os eventos do UIControl que são capturados usando os métodos onTouchInside((UIView) -> Void) e o método onEvent(UIControl.Event, (UIView) -> Void).

Conforme as Figura 37 e Figura 38, o botão foi criado com a UICLabel para mostrar o texto “Aperte” como forma de customizar o conteúdo mostrado pelo UIButton. No caso do texto, foi adicionado outras propriedades para inserir margem em relação as bordas de 15 na horizontal e 7,5 na vertical e a cor azul.

Figura 37 - UICButton com texto "Aperte".

```
1   import UICreator
2
3   class ContentView: UICView {
4       var body: ViewCreator {
5           UICCenter {
6               UICRounder(radius: 7.5) {
7                   UICButton {
8                       UICLabel("Aperte")
9                           .font(.body(weight: .bold))
10                          .textColor(.white)
11                          .padding(15, .horizontal)
12                          .padding(7.5, .vertical)
13                          .backgroundColor(.blue)
14                  }
15              }
16          }
17      }
18  }
```

Fonte: Elaborado pelo Autor.

Figura 38 - Execução do UICButton com texto "Aperte".

Fonte: Elaborado pelo Autor.

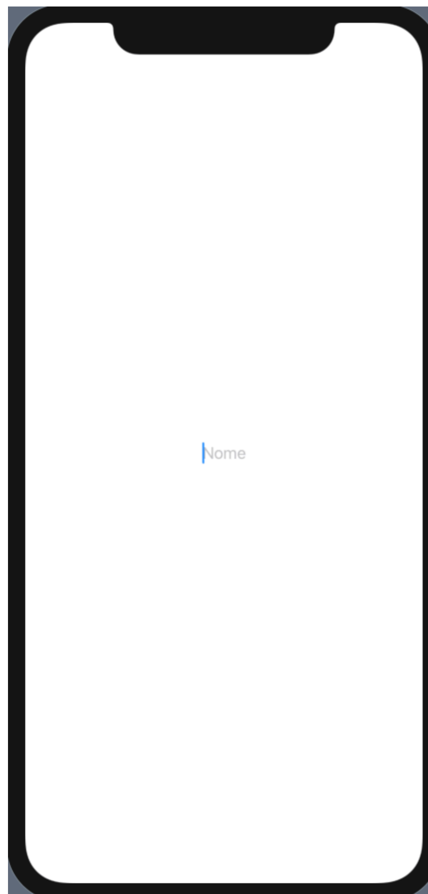
A UITextField também herda a classe UIControl e representa um campo de texto editável. Existem duas formas de controlar o comportamento desse objeto, o primeiro usando os eventos do UIControl e o segundo usando o protocolo UITextFieldDelegate que deve ser atribuído à propriedade *delegate* do UITextField. Além disso, é possível determinar o texto de guia ao usuário, todas as propriedades envolvendo a classe UILabel, além de definir comportamentos envolvendo o teclado. O objeto construtor referente a essa classe de interface é o UICText, implementando os métodos placeholder(String?), textColor(UIColor?), do UIControl o onEditingDidEnd((UIView) -> Void) e outros. As Figura 39 e Figura 40 implementam um texto editável com texto de guia “Nome”.

Figura 39 - UICText com campo igual a "Nome".

```
1  import UICreator
2
3  class ContentView: UICView {
4      var body: ViewCreator {
5          UICCenter {
6              UICText(placeholder: "Nome")
7          }
8      }
9  }
```

Fonte: Elaborado pelo Autor.

Figura 40 - Execução do UICText com campo igual a "Nome".



Fonte: Elaborado pelo Autor.

O protocolo que deve ser usado pelos programadores para implementar as *views* substituindo o UIViewController é o UICView. Esse protocolo estende o ViewCreator e obriga a declaração da variável de leitura *body*. Com essa variável, os

programadores podem implementar a hierarquia de *views* e gerenciar o comportamento de cada tela a partir da classe que implementa o UICView. Para manter compatível com a programação já existente dos projetos que utilizam o UIKit e o *storyboard*, o protocolo tem o método de implementação opcional viewDidLoad(), onde os desenvolvedores podem migrar chamada às funções de configuração da UIViewController para dentro desse método.

O ViewCreator encapsula a UIView e impede um acesso direto a ela fora do fluxo declarativo. Ao utilizar o atributo declarativo @propertyWrapper, é implementado a estrutura imutável UICOutlet que armazena a UIView. Esse atributo declarativo altera a forma em que a estrutura pode ser acessada. Além disso, ele exige a implementação da variável *wrappedValue* que deve retornar o objeto encapsulado pela estrutura, nesse caso da UICOutlet, a referência da UIView. A variável opcional *projectedValue* permite o acesso de forma explícita ou por meio do símbolo $ antes do nome da variável que armazena a estrutura com o @propertyWrapper. Então, para obter a instância da UIView encapsulada pelo ViewCreator, é implementado o método as(Outlet<UIView>).

Apesar das classes específicas que encapsulam objetos do UIKit, pode ser necessário transformar uma classe escrita usando a UIView compatível com a estrutura declarativa. Para que isso ocorra, é implementado o protocolo UICViewRepresentable que estende o protocolo UIViewCreator e o protocolo ViewRepresentable. No momento em que a UIView dos objetos construtores é requisitada, o ViewCreator verifica se ele estende o protocolo ViewRepresentable e chama o método _makeView().

O UICViewRepresentable implementa esse método e retorna a *view* encapsulada usando a função a ser implementada pelo programador makeUIView(). Para manter os estados de renderização, a *view* é atribuída a ViewAdaptor, uma classe que sobrescreve os métodos da UIView referente aos estágios de renderização, mantendo o correto funcionamento do código pelo objeto construtor UICViewRepresentable.

O UIKit contém uma série de gestos implementados com base na classe UIGestureRecognizer. Para permitir que os objetos construtores possam adicionar gestos ao objeto de interface, é implementado os objetos construtores de gestos, seguindo a mesma ideia implementada no ViewCreator, mantendo a referência forte

ao UIGestureRecognizer até que ela entre na hierarquia. Com isso, o ViewCreator implementa os métodos onTap((UIView) -> Void), onPinch((UIView) -> Void) e outros para que o programador possa adicionar gestos a UIView encapsulada por ele.

Outro recurso a ser implementado é o posicionamento dos objetos nos quadros das *views*. O UIKit implementa duas soluções: (a) o *layout* baseado em quadro; e (b) o *auto layout*. O ViewCreator utiliza o *auto layout* para manter o posicionamento da UIView na tela. Para que o programador possa alterar os valores da posição da *view*, o objeto construtor define os métodos top(equalTo: CGFloat), bottom(equalTo: CGFloat), leading(equalTo: CGFloat), trailing(equalTo: CGFloat) e outros referentes a altura, a largura e a disposição na tela. Todos esses métodos são implementados utilizando o método onInTheScene((UIView) -> Void), pois dependem da hierarquia completa ou a hierarquia mais próxima para efetivar as regras de *layout*.

No ViewCreator é adicionado a *view* pai com regras de *layout* para expandir em todas as direções até a margem do objeto pai. Um ViewCreator que permite dimensionar o posicionamento da UIView sem que seja aplicado essas regras é o objeto construtor UICContent. Com isso, a UIView pode ser posicionada obedecendo o centro, a margem do topo e da esqueda, a margem de baixo ou outras formas implementadas pela classe. Assim, *views* como a UILabel e a UIButton, que tem a altura e a largura relativa ao tamanho do conteúdo renderizado, com a utilização da classe UICContent, o *layout* pode ser respeitado.

4 COMPONENTES REATIVOS

Este capítulo apresenta a implementação de componentes reativos para sincronizar dados entre os objetos de interface e a aplicação, bem como o desenvolvimento de classes declarativas para listagem *views* dentro do fluxo declarativo.

4.1 Estrutura de notificações

A programação reativa é integrada aos objetos de interface para refletir o dado em uma propriedade ou comportamento do objeto, simplificando a programação. A comunidade de código aberto do ReactiveX implementa diversos *frameworks* para as principais linguagens de programação, em que as classes e objetos tornam-se reativos. Esse paradigma de programação traz benefício ao projeto, quando simplifica diversas estruturas para atualizar as informações que estão na tela do usuário e, também, garante maior controle sobre os dados (FEILER, 2018).

O framework Foundation implementa os conceitos centrais da programação reativa, embora não em toda a sua definição, utilizando a propagação de notificações. Para isso, é definido o emissor e o ouvinte que possuem a habilidade de publicar e de escutar as notificações, respectivamente. Entretanto, apesar de ser possível adaptar essa estrutura desenvolvida pela Apple para a programação reativa, ela é utilizada para escutar mudanças de estados e de valores dentro do *framework* UIKit, como por exemplo, o tamanho do teclado que será mostrado na tela.

São quatro principais objetos utilizados para emitir e escutar as notificações: (a) a classe NotificationCenter; (b) a estrutura Notification; (c) a estrutura NSNotification.Name; e (d) o protocolo NSObjectProtocol (Apple, 2020).

A classe NotificationCenter é responsável por propagar as notificações e gerenciar os observadores. Existem duas formas de acessar a instância dessa classe, a primeira criando uma instância do objeto utilizando o init() e a segunda por meio da variável global e estática *default*. O UIKit e outros *frameworks* que compõem o SDK do iOS utilizam a NotificationCenter através da variável global para publicar as notificações. Com a instância da classe, que deve ser comum entre o emissor e o observador, é possível chamar os três principais métodos:

a) post(name: NSNotification.Name, object: Any?, userInfo: [AnyHashable: Any]?);
b) addObserver(forName: NSNotification.Name?, object: Any?, queue: OperationQueue?, using: (Notification) -> Void) -> NSObjectProtocol;
c) removeObserver(Any).

O método (a) tem como parâmetros o nome da notificação e o dicionário *userInfo*. O primeiro parâmetro é utilizado pela NotificationCenter para gerar notificações específicas e, com isso, podem ser escutadas pelos observadores. O segundo parâmetro é usado para anexar uma coleção de valores e propriedades, onde o observador tem acesso ao dicionário na notificação propagada. Dessa forma, quando o método é chamado, com base nesses dois parâmetros, a NotificationCenter inicializa a estrutura Notification e propaga o objeto entre os observadores registrados.

O nome das notificações é atribuído pela estrutura Name, definida dentro do escopo da NSNotification, quando inicializada pelos métodos init(String) ou init(rawValue: String). Cada emissor utiliza de um nome para propagar as notificações, sem a certeza de que esse nome é único, principalmente quando a instância da NotificationCenter é global ou compartilhada. Entretanto, em um cenário com controle total sobre as notificações propagadas, isto é, com uma instância privada da NotificationCenter, é possível garantir que as notificações sejam emitidas com nomes únicos, pois todos os emissores são conhecidos.

Os observadores são criados por meio do método (b) com dois parâmetros principais. O *forName* é usado para escutar as notificações com o nome equivalente ao informado, enquanto o *using* é um *callback* para tratar a notificação propagada, possibilitando ter acesso a variável *userInfo* e ler o dicionário anexado. Esse método retorna uma instância de um objeto que implementa o protocolo NSObjectProtocol, o qual deve ser armazenado pelo observador. Dessa forma, quando o observador for desalocado ou tornar-se inválido, por meio da referência do NSObjectProtocol, é possível chamar o método (c) para remover o *callback using* e impedir que novas propagações sejam informadas ao observador desativado.

4.2 A classe ReactiveCenter

A necessidade de implementar uma estrutura reativa para este projeto está na simplificação de manter as *views* atualizadas, quando elas são criadas a partir de métodos declarativos. Caso contrário, seria utilizado as propriedades da *view* do UIKit para atualizá-la. Existem uma série de propriedades que podem ser sincronizadas a uma fonte de dados, por exemplo, o texto da UILabel, o título do UIButton e outras encapsuladas pelo ViewCreator. Então, com a implementação dos métodos reativos para o objeto construtor, será possível conectar a UIView encapsulada a uma fonte de dados diretamente no fluxo lógico declarativo.

Utilizar a instância global da NotificationCenter pode gerar diversas inconsistências dentro da estrutura reativa do *framework* declarativo, uma vez que as notificações podem ser geradas por outros emissores desconhecidos. Além disso, instanciar a NotificationCenter impossibilita o desenvolvimento de novos métodos necessários para garantir a integridade da aplicação em um cenário reativo. Dessa forma, é implementado a classe ReactiveCenter, interna ao *framework,* com a variável estática *shared*.

4.2.1 Single Source of Truth

O contexto em que será utilizado a estrutura reativa desenvolvida é um cenário onde o emissor está vinculado a *view* pai e o observador existe enquanto a referência do emissor existir. Por exemplo, a MainView que implementa o protocolo UICView, contém um texto dinâmico criado pela UICLabel. O objeto construtor e a UILabel encapsulada por ele existirão em memória enquanto a MainView existir. Dessa forma, vinculando os observadores ao emissor garante-se que as notificações sejam propagadas enquanto o objeto que instanciou o emissor estiver em memória.

Nesse sentido é definido as estruturas Value<Value> e a Relay<Value>. A primeira é responsável por armazenar o valor que será propagado nas notificações. Existem ao menos duas notificações principais: (a) quando o valor altera; e (b) quando é requisitado uma leitura do valor armazenado. A notificação pública (a) ocorre sempre que um valor é atribuído na estrutura Value, anexando o novo valor na notificação e informando aos observadores. A notificação privada (b) é enviada pelo observador no

momento em que é feito uma requisição de leitura do valor armazenado, recebendo-o anexado em um outro evento privado propagado pelo Value.

A segunda estrutura, Relay<Value>, é capaz de ler e escrever o valor diretamente na Value fonte e observar as alterações. Usando o conceito de *thread* e a implementação da variável de leitura e escrita através dos atributos *get* e *set*, é encapsulado dois métodos. O método de leitura bloqueia a *thread,* em que o código está sendo executado, faz a requisição para receber o valor atual e ao recebê-lo libera a *thread*, retornando o valor. O método de escrita gera uma notificação para gravar um novo valor na estrutura fonte Value, anexando o valor por meio do *userInfo*. Por último, para escutar as mudanças, são implementados dois métodos:

(a) sync((Value) -> Void);
(b) next((Value) -> Void).

Ambos os métodos recebem como parâmetro um *callback* para ser executado quando o valor é propagando nas notificações. O primeiro método faz uma requisição do valor inicial e escuta as próximas alterações, enquanto o segundo apenas escuta a mudança de novos valores.

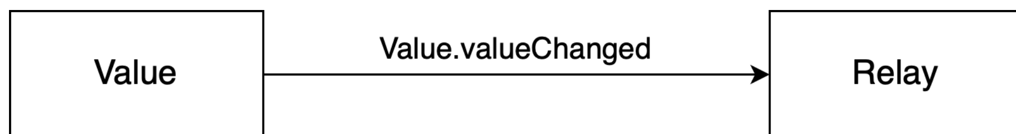
Com essa abordagem, a estrutura Value é semelhante ao conceito de *Single Source of Truth* (SSOT), em que somente ela é responsável por armazenar os valores. Com o Relay, essa característica é mantida, uma vez que é feito requisições a estrutura Value constantemente para retornar o valor atual (VACCARO, 2020).

4.2.2 Eventos Reativos

As duas estruturas Value e Relay publicam notificações no ReactiveCenter e, também, são observadores de eventos públicos e privados. Os nomes dos eventos são formados pela instância do ObjectIdentifier, inicializado com a referência do Value fonte, que retorna quando convertido para String o endereço de memória do objeto, no formato ObjectIdentifier(0x0000FFFF). Isso garante que apenas os objetos que contém essa chave possam publicar e escutar as notificações, além de manter a integridade do conceito SSOT. Serão utilizados o nome da estrutura Value para compor o nome dos eventos neste trabalho, porém, na prática, esse termo é substituído pelo ObjectIdentifier. Os nomes são: (a) Value.valueChanged; (b) Value.request.getter.value; (c) Value.getter.value; (d) e Value.request.setter.value.

Ambas as estruturas reativas são implementadas com o atributo declarativo @propertyWrapper, onde o tipo da variável *wrappedValue* é definido pelo tipo genérico Value. Além disso, a estrutura Value retorna uma instância da estrutura Relay na variável *projectedValue*, enquanto o Relay retorna uma cópia da sua própria instância. Quando o Relay é inicializado, é informado a ele o endereço do Value para a observação e publicação de eventos. Conforme a Figura 41, quando um novo valor é atribuído ao Value, a notificação Value.*valueChanged* é propagada. Porém, o Relay não observa essa notificação o tempo todo, para não gerar cópias do valor propagado, apenas quando as funções sync(_:) e next(_:) são chamadas.

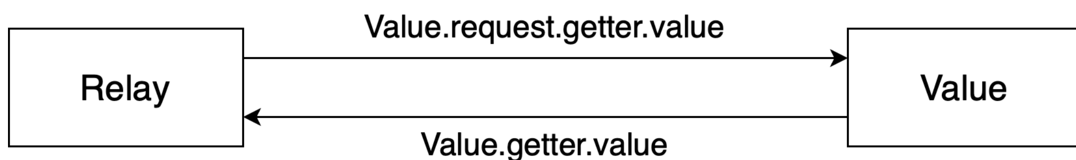
Figura 41 - Evento para observar a alteração do valor armazenado no Value.



Fonte: Elaborado pelo Autor.

Um segundo recurso dentro dessas duas estruturas reativas é a possibilidade de ler, através do Relay, o valor armazenado no Value. Para isso ocorrer, como na Figura 42, o Relay dispara o evento Value*.request.getter.value* e aguarda a resposta do Value com a notificação Value.*getter.value*. A estrutura Value observa esse evento de requisição desde o momento em que foi instanciada, enquanto o ReactiveCenter é encarregado de gerenciar essa transação. Quando o Relay inicia o processo, o ReactiveCenter dispara o evento de requisição de leitura e cria um observador para o evento de retorno com o valor lido. Após o retorno do valor, o ReactiveCenter desativa o observador utilizando o NSObjectProtocol retornado pela função de escuta.

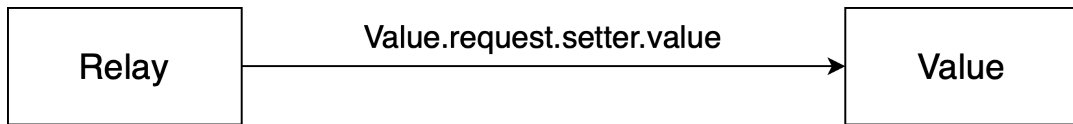
Figura 42 - Evento para ler o valor em Value.



Fonte: Elaborado pelo Autor.

A possibilidade de atribuir valores no Value através do Relay, é um recurso para propagar alterações na *view* filha para a *view* pai. Nesse caso, ao alterar o valor do Relay, é disparado a notificação Value.*request.setter.value*, conforme a Figura 43. A estrutura Value escuta essa notificação e atribui o valor a sua variável.

Figura 43 - Evento para atribuir um novo valor ao Value.



Fonte: Elaborado pelo Autor.

Dessa forma, a partir desses eventos e das duas funções next(_:) e sync(_:), foi possível implementar diversos operadores para simplificar o processo de codificação. O primeiro operador é o map((Value) -> OtherValue) que retorna um novo Relay a partir da mesma fonte de propagação, porém, quando um novo valor é atribuído, o *callback* do operador converte o valor propagado em outro. O segundo operador cria um novo Value com o tipo OtherValue e, sempre que um novo valor é propagado, o operador chama o *callback* e escuta as alterações do Relay retornado, atribuindo os valores ao Value<OtherValue>. A nova instância criada pelo operador é retornada como Relay<OtherValue> utilizando a variável *projectedValue*.

Além desses operadores dinâmicos, foi implementado alguns operadores aritméticos para valores booleanos. É possível utilizar os operadores &&, || e ! no Relay onde o tipo genérico é o tipo Bool.

4.3 Construtor dinâmico UICForEach

Com a implementação do ReactiveCenter e das estruturas Value e Relay, foi desenvolvido o objeto construtor UICForEach. Essa classe implementa o protocolo ViewCreator e instancia uma *view* vazia e temporária. A proposta é transformar o vetor de elementos em um vetor de *views,* permitindo recarregar o conteúdo na tela de forma dinâmica quando um novo valor for atribuído ao Value. Além disso, para que seja recarregado as *views* na tela, a *view* pai deve implementar o protocolo SupportForEach e atribuir ao UICForEach sua referência.

Os elementos de interface que foram implementados para mostrar uma coleção de *views* pelo UIKit são: (a) UIStackView; (b) UITableView; e (c) UICollectionView. No caso da UIView, apesar de ser possível mostrar várias *views* dentro do seu *frame*, ela não é caracterizada como *view* dinâmica e nem é usada para mostrar coleções de objetos. Para isso, pode ser utilizado a *view* (a) que mostra uma coleções de *views* arranjadas na direção vertical ou horizontal. As *views* (b) e (c) exigem a implementação de dois protocolos para recarregar o conteúdo que será abordado na seção 4.4. Por último, a UIStackView exige o gerenciamento das *views* arranjadas, removendo e as adicionando.

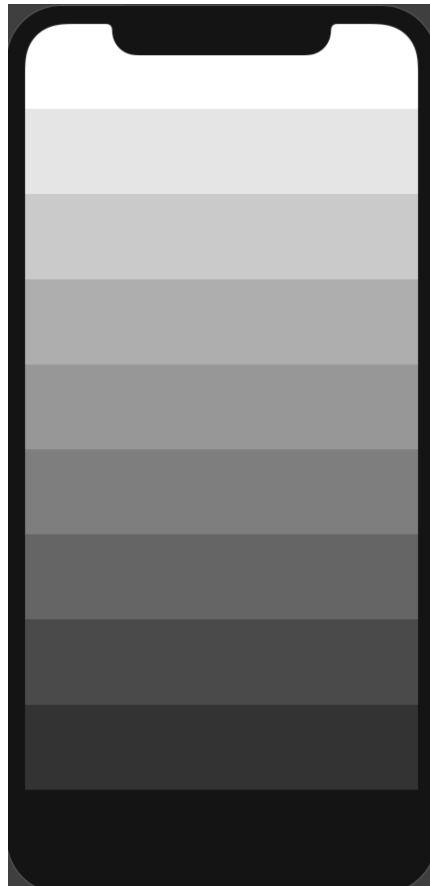
O protocolo SupportForEach exige a implementação da função viewsDidChange(placeholderView: UIView!, (Relay<[ViewCreator]> -> Void)). O primeiro parâmetro é a referência da *view* criada pelo UICForEach e o segundo é o Relay mapeado com as *views* a serem atualizadas. O desenvolvimento dessa função no escopo da UICStack ocorre procurando a referência da *view* do UICForEach para reconstruir as *views* arranjadas, salvando o índice de início e fim. Na segunda tentativa, essa busca é feita com base nos índices salvos, removendo as *views* antigas e substituindo pelas novas. O resultado esse processo está representado nas Figura 44 e Figura 45.

Figura 44 - UICForEach como conteúdo dinâmico da UICVStack.

```
1  import UICreator
2  import CoreGraphics
3
4  class ContentView: UICView {
5      var body: ViewCreator {
6          UICSpacer {
7              UICVStack {
8                  UICForEach(0..<10) {
9                      UICSpacer()
10                         .backgroundColor(.black)
11                         .alpha(CGFloat($0) / 10)
12                 }
13             }
14             .distribution(.fillEqually)
15         }
16         .backgroundColor(.white)
17     }
18 }
```

Fonte: Elaborado pelo Autor.

Figura 45 - Execução da UICVStack com UICForEach dinâmico.



Fonte: Elaborado pelo Autor.

O construtor UICForEach pode ser inicializado de duas formas, uma com o vetor de dados dinâmico e outra com o vetor estático. O segundo parâmetro é a função que transforma os valores do vetor em uma coleção de ViewCreator. Após a inicialização, o UICForEach aguarda a sua *view* encapsulada entrar na hierarquia para iniciar o processo de recarregar as *views*. Então, com o operador map(_:), quando o novo vetor é propagado pelo ReactiveCenter, o UICForEach transforma os valores e chama a função viewsDidChange(placeholderView:,_:) por meio da referência que implementa o protocolo SupportForEach.

4.4 UICList

A UITableView é uma *view* que lista conteúdos na tela categorizados por seção. Cada seção contém um cabeçalho, um rodapé e um conjunto de células. Os dois primeiros são representados pela classe UITableHeaderFooterView, enquanto a terceira é representada pela classe UITableViewCell. Quando uma lista é instanciada, ela não tem nenhum registro quanto as células e seções que devem ser listadas. Além disso, a lista do UIKit utiliza o conceito de reciclagem de *views* para manter o uso de memória baixo. Então, todos os objetos que constituem a estrutura da célula devem ser registrados na lista e atribuído um identificador único para que possam ser instanciados e reutilizados.

A implementação das classes customizadas para mostrar conteúdo na lista, por exemplo a classe UserViewCell que estende a UITableViewCell e é registrada na lista. Realizar esses dois passos não é o suficiente para que a lista possa ser montada. A listagem é a representação de um conjunto de dados em forma de elementos gráficos, atribuindo cada dado do conjunto a uma *view*. O processo de transformar os dados em um conjunto de *views*, no UIKit, é feito por meio da variável *dataSource*, que recebe uma instância de um objeto que implementa o protocolo UITableViewDataSource.

Conforme a Figura 46, é necessário implementar os métodos tableView(UITableView, numberOfRowsInSection section: Int) e tableView(UITableView, cellForRowAt: IndexPath). O primeiro método é usado para informar quantos células existem e o segundo retorna a instância de uma

UITableViewCell. Nesse segundo caso, a instância da célula pode ser obtida por meio do inicializador da classe ou, utilizando o método de reciclagem de células da UITableView, por meio do método dequeueReusableCell(withIdentifier: String, for: IndexPath).

Figura 46 - Exemplo de implementação do protocolo UITableViewDataSource.

```
1  import UIKit
2
3  extension MainViewController: UITableViewDataSource {
4      func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
5          self.users.count
6      }
7
8      func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
           UITableViewCell {
9          let userCell = tableView.dequeueReusableCell(withIdentifier: "user.identifier", for:
               indexPath) as! UserTableViewCell
10
11         userCell.titleLabel.text = self.users[indexPath.row].name
12
13         return userCell
14     }
15 }
```

Fonte: Elaborado pelo Autor.

A adaptação da UITableView para uma estrutura declarativa exige atender diversos cenários como: (a) listagem de células; (b) listagem de seções com células; (c) listagem de seções com cabeçalho, rodapé e células. Além desse requisito, é preciso otimizar o uso de memória e de processamento, uma vez que os dados são grandes e os métodos que obtém as instâncias das *views* são chamados constantemente. Também, como o objetivo é transformar os dados em instâncias do ViewCreator, será necessário armazenar os objetos montados para manter a integridade da lista.

O armazenamento dos objetos construtores garante que alguns dados dinâmicos sejam preservados, como: o número de células, o número de seções e os objetos necessários para a listagem. Para minimizar os problemas de memória, os objetos construtores não serão armazenados e, para obtê-los será utilizado os *callbacks*. Então, a estrutura utilizada para otimizar o desempenho será os vetores aninhados, imitando a estrutura da lista em seção, cabeçalho, rodapé e célula.

Com essa abordagem, a lista passa a se comportar por estado. Ela inicia no estado vazio e, sempre que é recarregada, um novo estado é configurado. Além das células, a lista também precisa da implementação do *delegate* para obter as *views* de

cabeçalho e de rodapé. Então, o estado precisa ser compartilhado entre os dois protocolos, e isso é feito criando uma nova classe de lista a UICTableView que estende a UITableView. Essa classe implementa o *delegate* e o *dataSource*, como também conhece o seu estado para realizar a listagem.

O primeiro passo será implementar os objetos construtores que representam a seção, o cabeçalho, o rodapé e a célula. Posteriormente, será desenvolvido o objeto que representa o estado da lista, a implementação dos objetos de interface da UITableView e, por útlimo, a implementação do *delegate* e do *dataSource*. A implementação do estado precisa ser compatível com o protocolo SupportForEach e manter a estrutura de lista, seção, cabeçalho, rodapé e célula. A classe que representa a lista e o seu estado é a ListManager. A classe que representa a seção é o SectionManager. Enquanto a classe RowManager representa o cabeçalho, o rodapé e a célula.

4.4.1 Objetos construtores para seção, cabeçalho, rodapé e célula

A montagem de uma lista de forma declarativa precisa obedecer a estrutura suportada pela UITableView com a definição de seção, cabeçalho, rodapé e célula. Isso permite que seja validado todas as *views* no momento da inicialização da lista. Também é preciso ser aceito como um objeto parte da lista o UICForEach que implementa o protocolo ViewCreator. Por essa razão, os objetos têm como base o protocolo declarativo para configurar as *views* da lista, embora o propósito de cada objeto é de apenas definir qual o tipo da *view*. Então, a seção é representada pela classe UICSection, o cabeçalho pela classe UICHeader, o rodapé pela classe UICFooter e a célula pela classe UICRow.

A estrutura declarativa da UICList é como representado na Figura 47, onde utilizando o *callback* é definido o cenário em que a lista se encaixa. Nesse caso, a lista contém a UICSection com cabeçalho, rodapé e uma única célula. A integração com o UICForEach é detalhada na seção 4.4.2, porém como um objeto construtor dentro da estrutura declarativa da lista, ele pode repetir dois tipos de *views*: a UICSection ou a UICRow. Dessa forma, com esses objetos seguindo esse formato e ordem, pode-se verificar se a sequência e se os objetos que estão na estrutura declarativa está em conformidade com a UITableView.

Figura 47 - Protótipo de código para sequência de objetos como conteúdo da lista.

```
1  import UIKit
2
3  class MainView: UICView {
4      var body: ViewCreator {
5          UICList {
6              UICSection {
7                  UICHeader {
8                      UICLabel("Cabeçalho")
9                  }
10
11                 UICRow {
12                     UICLabel("Célula")
13                 }
14
15                 UICFooter {
16                     UICLabel("Rodapé")
17                 }
18             }
19         }
20     }
21 }
```

Fonte: Elaborado pelo Autor.

A classe declarativa UICSection implementa o protocolo ViewCreator, porém não implementa a *view* encapsulada, impedindo de ser utilizado como objeto de interface. Essa classe é utilizada para verificar o conteúdo declarativo com o qual ela foi inicializada, sendo formado na regra: de zero a um objeto UICHeader, **n** objetos UICRow ou UICForEach, e de zero a um objeto UICFooter. Além disso, a UICSection é opcional, utilizada para implementar várias seções diferentes ou aninhada em um objeto UICForEach. Caso contrário, a lista é configurada com uma seção padrão sem cabeçalho ou rodapé.

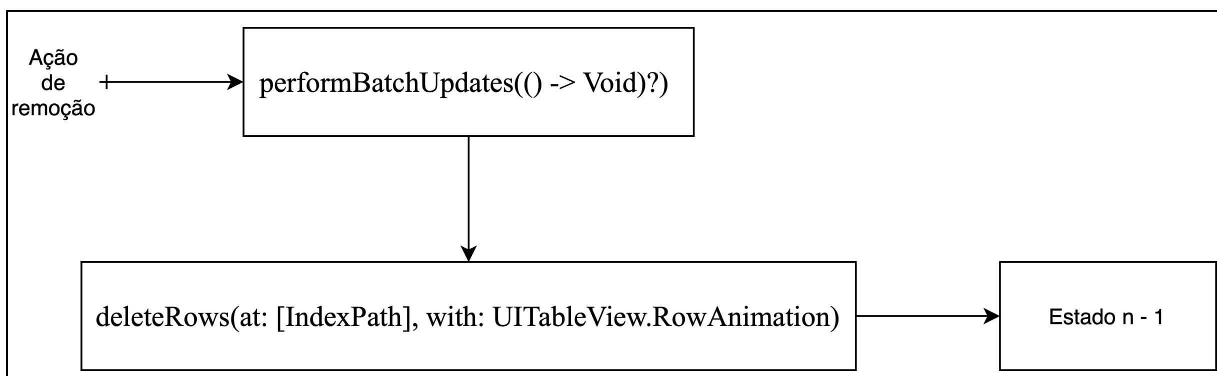
As classes declarativas UICHeader, UICFooter e UICRow seguem o mesmo padrão de algoritmo e não contêm a UIView encapsulada. Com exceção da classe UICRow que, além de marcar o tipo da *view* na lista, contém outras propriedades com relação aos recursos da UITableView como, as propriedades para configurar ações à direita e à esquerda, quando o usuário desliza a célula nesses dois sentidos revelando

botões de ação; e a propriedade para definir o tipo do acessório da UITableViewCell, utilizando o enum do UIKit AcessoryType. Essas três propriedades são definidas usando os métodos declarativos trailingActions(() -> RowAction), leadingActions(() -> RowAction) e accessoryType(UITableViewCell.AccessoryType).

O RowAction é um protocolo implementado pelas classes UICContextualAction e a UICRowAction. Essas duas classes possuem similaridade quanto a implementação dos recursos, onde a primeira encapsula a classe do UIKit UIContextualAction, disponível para o iOS 11 em diante, enquanto a segunda encapsula a classe UITableViewRowAction, descontinuada no iOS 13. Ambas as classes permitem a configuração de propriedades como título, imagem e cor de fundo, alterando apenas a forma em que as ações são executadas e como os botões são configurados no processo de montagem da lista pelo *delegate*. Existem duas ações suportadas pelo UIKit, uma para deletar um elemento da lista e a outra sem regra específica.

A ação de remoção precisa ser sincronizada com os métodos de atualização da lista, o performBatchUpdates(() -> Void)?) e o deleteRows(at: [IndexPath], with: UITableView.RowAnimation), gerando um problema quando a lista é representada por estados. No momento em que o *callback* de remoção do RowAction é executado, é necessário executar as duas funções de atualização da lista. Entretanto, a função deleteRows(at:, with:) contém a regra de que a quantidade de elementos será igual a n – 1, caso contrário, o UIKit gera um erro fatal à aplicação. Conforme a Figura 48, é preciso executar as funções de atualização da lista antes de alterar o estado dela. A inversão desse fluxo geraria uma inconsistência ao executar o método deleteRows(at:, with:), porque a lista estaria atualizada com n – 1 elementos.

Figura 48 - Fluxo de atualização da ação de remoção da UITableView.



Fonte: Elaborado pelo Autor.

Esses objetos são desmontados pelo ListManager e transformados em SectionManager e RowManager. A primeira é o resultado da transformação da UICSection, enquanto a segunda é das classes UICHeader, UICFooter e UICRow. Com esses objetos transformados, o último passo é listar os objetos usando as classes que estendem a UITableViewCell e a UITableHeaderFooterView, discutido na seção 4.4.3.

4.4.2 Classes que representam o estado da lista

O ListManager é uma classe que representa o estado da lista, sendo que sua inicialização é feita com uma coleção de objetos ViewCreator que deve conter os objetos UICSection, o UICRow e o UICForEach. O *for each* se comporta de duas formas quando seu tipo dinâmico é igual ao tipo da seção ou, igual ao tipo da célula. A coleção é caracterizada pelos objetos que ela contém, sendo aceito duas formas: (a) coleção de seções, onde o ListManager transforma as UICSection e o UICForEach em objetos do tipo SectionManager, representando o estado da seção; e (b) coleção de células, transformado em uma única instância da seção. Assim, o ListManager verifica e garante que o programador inicializou corretamente a lista, armazenando as seções em sua variável para leitura *sections*.

O SectionManager armazena as células, o cabeçalho e o rodapé que serão mostrados na tela, gerenciando não só o carregamento dinâmico do UICForEach como, também, os identificadores das *views* e seus índices de busca para quando a UITableView estiver carregando o conteúdo. Os identificadores são usados para registrar e reciclar as *views* que formam a estrutura da lista. Após a inicialização e configuração do ListManager, os identificadores são registrados usando o método register(AnyClass?, forCellReuseIdentifier: String) da lista.

Os identificadores são organizados de forma que para cada UICSection, ou SectionManager, é atribuído um índice de seção. No caso da lista conter dez seções, serão atribuídos para cada seção um valor entre zero e nove. Quando a seção é dinâmica, gerenciada pelo UICForEach, é compartilhado o mesmo índice para todas as seções retornadas pelo *for each*. Nesse caso, a lista pode conter, por exemplo, nove seções estáticas, enquanto a décima é dinâmica, onde todas que forem geradas pelo *for each* naquele índice, terão seu índice de seção igual a nove.

Após a definição das seções e do seu identificador, a SectionManager recebe o vetor de *views* que deve formar as células, o cabeçalho e o rodapé. É verificado a integridade do vetor que deve conter, de forma opcional, no primeiro índice um objeto do tipo UICHeader e, no último índice um objeto do tipo UICFooter. Nos demais índices, é esperado que as *views* sejam do tipo UICRow ou UICForEach com tipo dinâmico igual a célula. Assim, a mesma regra aplicada a UICSection é usada para atribuir, aos objetos que formam as células, o correspondente índice para compor o identificador.

A SectionManager converte cada elemento dentro do vetor de *views* em um objeto da classe RowManager. Essa classe armazena as propriedades de configuração da célula como, o *callback* que retorna a instância do ViewCreator, o identificador montado pela SectionManager, o índice formado por {seção, linha} e as propriedades para definição das ações e o acessório da UITableViewCell. Com essas definições primárias, é possível montar a lista de conteúdo estático, sem suporte ao protocolo SupportForEach.

A adaptação dessas três classes, em classes dinâmicas, necessita implementar dois protocolos: (a) ListSectionDelegate; e (b) ListContentDelegate. O protocolo (a) é implementado pelo ListManager, que atribui sua instância ao objeto SectionManager, para permitir a seção informar que o estado da lista mudou. No caso das seções serem dinâmicas, associadas a um *for each*, o SectionManager cria uma cópia de si para cada seção retornada pelo UICForEach e, atribui as *views* armazenadas pelo novo UICSection a sua cópia. Assim, a SectionManager consegue manipular a seção estática ou dinâmica e notificar que o estado da lista foi atualizado.

O protocolo (b) é implementado pela SectionManager, porém a RowManager contém apenas a referência do ListManager. Quando a classe que gerencia uma célula é dinâmica, ela deve informar a seção que a sua quantidade de células mudou, propagando essa alteração ao ListManager. Isso é feito usando a referência do estado da lista, pois o RowManager não deve armazenar a SectionManager, uma vez que ambas as classes são dinâmicas e podem expirar da memória em determinado momento. Por isso, a única garantia é a instância do ListManager que contém o método section(at: Int), obtendo a instância da SectionManager.

A implementação do SupportForEach pela RowManager segue o mesmo algoritmo aplicado na classe SectionManager. Para cada *ViewCreator* retornada pelo

for each, é criado uma cópia da RowManager. Após a transformação do vetor de *views* em vetor de estado de célula, é informado à seção que a quantidade de células mudou. A seção atualiza o vetor de células para manter a integridade da lista e informa ao ListManager que uma alteração ocorreu. Dessa forma, o ListManager recarrega a UITableView, refletindo o novo estado.

Além da classe ListManager, existem as classes ListManager.Append e o ListManager.Delete. Essas duas classes são definidas como estado de transição da lista, que é sempre equivalente a configuração do ListManager. A primeira classe é usada para impedir que a lista seja recarregada enquanto ela estiver no estado de adição de novas células ou seções, permitindo a execução correta dos métodos da UITableView para efetuar a ação de adição. A segunda classe é usada para simular a remoção das células ou seções deletadas. Essa ação é feita ignorando a leitura e requisição das estruturas da lista que foram deletadas, permitindo a UITableView preencher a lista com a quantidade de células igual ou inferior a n – 1.

A classe ListManager.Delete também é usada pelo protocolo RowAction. Ambas as classes UICContextualAction e UICRowAction utilizam do ListManager.Delete para simular a ação de remoção da célula que eles foram configurados para remover. Quando a ação termina, o RowAction executa o *callback* para que a aplicação gere o novo estado da lista, voltando para o tipo ListManager.

A UITableView exige que os métodos sejam exatos para retornar corretamente as células que compõem a seção e as seções que compõem a lista. Enquanto ela está montando a lista ou requisitando os objetos de interface, é necessário que o algoritmo tenha eficiência para não bloquear a *thread* principal e gerar demora no processo de listagem, caso contrário, a experiência do usuário é afetada com gargalos durante a visualização dos conteúdos. Também, essa representação dos estados nessas cinco classes não sobrecarregam a memória, uma vez que os objetos de interfaces são obtidos por meio dos *callbacks* e a estrutura é organizada utilizando vetores.

4.4.3 Implementação da TableViewCell e TableHeaderFooterView

Os objetos de interface mostrados pela lista são sobrescritos para comportar o estado da lista. A classe TableViewCell extende a classe UITableViewCell e a classe TableHeaderFooterView extende a classe UITableHeaderFooterView. Ambas

compartilham o mesmo código que é implementado pelo protocolo ReusableView. Esse protocolo recebe o objeto RowManager, executa o *callback* e insere o ViewCreator à *view* que implementa o protocolo. Por conta do *auto layout*, as *views* recebem regras para centralizar no quadro da sua *superview*. Além disso, o tamanho do quadro é monitorado para atualizar a altura dos objetos que compõem a lista, evitando os problemas de *auto layout* da UITableView.

O protocolo UITableViewDelegate contém vários métodos para determinar o tamanho das células, dos cabeçalhos e dos rodapés. Porém, quando a *view* tem seu tamanho definido usando o *auto layout*, é utilizado o tamanho automático para que a lista atualize o tamanho da célula para corresponder o tamanho da *view*. Entretanto, se o tamanho da célula não for calculado antes da *view* ser montada, a célula é configurada com o tamanho padrão da lista pela variável *estimatedRowHeight*. Após, a *view* recalcular o seu tamanho, a UITableView pode causar conflitos de *layout* e, em algums casos, não atualizar a célula para corresponder.

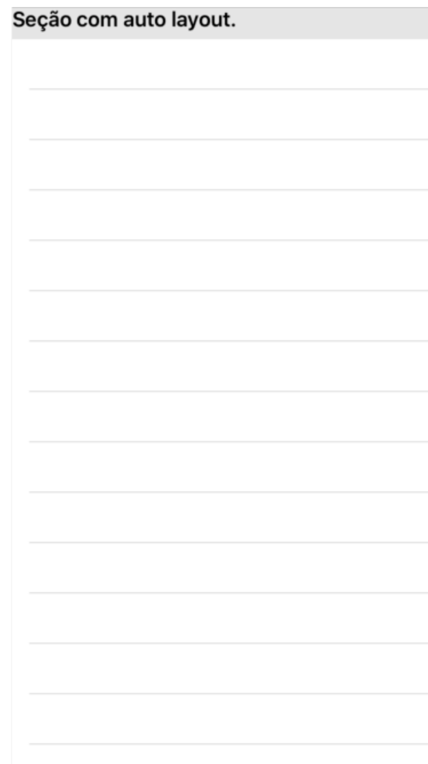
No caso da UITableHeaderFooterView, o problema é ainda mais presente, uma vez que a lista não reflete corretamente o tamanho da *view*. Para isso, é necessário implementar alguns métodos e estruturas, discutidos na seção 4.4.4, além de ser necessário chamar funções da UITableView para recarregar os cabeçalhos e rodapés atualizados. Conforme as Figura 49 e Figura 50, é possível reproduzir esse problema do *auto layout* da lista nos objetos que implementam a classe UITableHeaderFooterView. Embora a *view* tenha o tamanho superior ao calculado pela UITableView, é mantido a definição inicial o que resulta em uma perda de conteúdo na tela.

Figura 49 - Implementação do cabeçalho da seção com auto layout.

```
1  import UIKit
2  import PlaygroundSupport
3
4  final class TableHeaderView: UITableViewHeaderFooterView {
5      weak var titleLabel: UILabel!
6
7      func prepare() {
8          self.titleLabel.text = "Seção com auto layout.\nA UITableView não
               atualiza a altura, mesmo quando está configurada para ser
               automática"
9      }
10 }
11
12 extension ViewController: UITableViewDelegate {
13     func tableView(_ tableView: UITableView, heightForHeaderInSection
           section: Int) -> CGFloat {
14         UITableView.automaticDimension
15     }
16 }
```

Fonte: Elaborado pelo Autor.

Figura 50 - Execução do cabeçalho da seção com auto layout.



Fonte: Elaborado pelo Autor.

O UIKit exige o tratamento de reutilização das *views* no processo de listagem, implicando na atualização dos dados para refletir o índice da célula e não na

reconstrução da hierarquia da *view*. Essa regra não é adotada pelo *framework* declarativo devido a falta de controle dos estados da UIView. Além disso, é necessário comparar a hierarquia atual da célula com a nova hierarquia a ser renderizada ou atualizada. Por exemplo, a célula é definida com o UICSpacer que foi declarado com cor de fundo azul; durante a reciclagem, não pode acontecer dois momentos: (a) o novo índice para essa célula continua sendo uma view da classe UICSpacer; e (b) o novo índice é uma nova *view* da classe UICLabel.

O primeiro caso poderia ser simples ao iterar na hierarquia da célula e trocar o ViewCreator responsável pela UIView, conforme o encapsulamento dos objetos construtores proposto por este trabalho. Ao final de iteração, deve ser executado os *callbacks* referente ao estado de renderização. Entretanto, os *callbacks* podem conter uma série de propriedades que já foram configuradas, como no exemplo da cor de fundo azul ou, em um caso mais grave, da *constraint* do *auto layout* for a definição da altura igual a 45 px, resultado em uma nova *constraint* a UIView. Como as células são recicladas várias vezes, a *view* poderia ser sobrecarregada com atributos que não foram reciclados causando um colapso de memória ou da quebra do *auto layout*.

O segundo caso é uma derivação do primeiro onde, até um certo ponto da hierarquia, existiria uma hierarquia distinta. Nesse caso, seria necessário remover a parte diferente e trocar pela parte correta. O pior caso é ter que remover toda a hierarquia de *views* e inserir a nova, como está implementado atualmente no *framework declarativo*, exigindo a renderização de todos os objetos novamente. Dessa forma, a própria reciclagem da hierarquia poderia causar uma sobrecarga para o processador, implicando em um tempo de reutilização equivalente ou maior ao necessário para renderizar novamente a hierarquia de *views*.

4.4.4 Implementando os protocolos UITableViewDataSource e UITableViewDelegate

O processo de listagem do UIKit necessita da implementação do *dataSource* para definir a quantidade de seções e de células. É com base nesse protocolo que as células são criadas e recicladas. O *delegate* é um protocolo usando para definir várias outras propriedades da lista como, por exemplo, os cabeçalhos, os rodapés, as ações e a altura das *views*. O primeiro passo é implementar o *dataSource* com base na

instância do ListManager. O segundo passo é implementar o *delegate* para montar as estruturas restantes da tabela, assim como as ações e altura das *views*.

Com o protocolo UITableViewDataSource se define a quantidade de seções com a função numberOfSections(in: UITableView), que retorna um valor inteiro. A estrutura do ListManager é organizada em seções e, para obter o número de seções que a lista deve mostrar, é necessário contar o número de seções no vetor. Após a definição das seções, o método tableView(UITableView, numberOfRowsInSection: Int) define a quantidade de células por seção. Então, usando o ListManager e acessando a posição da seção pelo índice informado pelo *dataSource*, se obtém a instância que contém o vetor de células, sendo necessário contar a quantidade de elementos.

O *dataSource* define os índices {seção, célula} com base nos valores retornados pelas duas funções e, então, chama a função tableView(UITableView, cellForRowAt: IndexPath). Nessa função, se utiliza a instância da lista para reciclar as células e definir as propriedades dos objetos de interface com base no dado associado ao índice da célula. Devido as abstrações do UICForEach, do UICRow e do RowManager e do ListManager, não é possível reciclar a hierarquia da célula, sendo necessário retirar o ViewCreator montado e atribuir o ViewCreator daquele índice à célula reciclada. Além disso, é responsabilidade do protocolo ReusableView montar as células que estendem as classes UITableViewCell e UITableHeaderFooterView.

No momento em que o ListManager é montado e os identificadores são definidos, são chamados os métodos para registar na tabela as classes de células (TableViewCell), de cabeçalho e rodapé (TableHeaderFooterView) que serão reutilizadas. Para registrar a célula é chamado o método register(AnyClass?, forCellReuseIdentifier: String) e para registrar o cabeçalho e o rodapé é chamado o método register(AnyClass?, forHeaderFooterViewReuseIdentifier: String). O valor do primeiro parâmetro nos dois casos são os valores TableViewCell.self e o TableHeaderFooterView.self, respectivamente.

Com o registro dos identificadores e das classes que serão reutilizadas, no método para montar a célula do *dataSource* é usado o identificador para reutilizar a célula com o método dequeueReusableCell(withIdentifier: String, for: IndexPath). Então, é obtido a instância da classe TableViewCell, onde são chamados os métodos

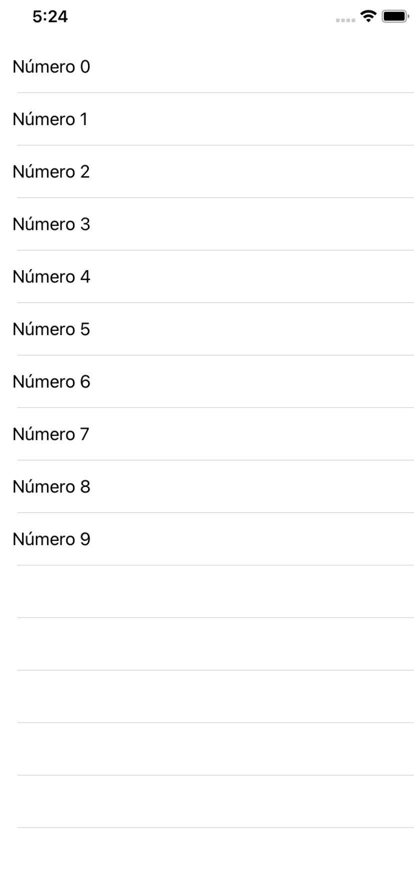
para montar a célula com a instância do RowManager. Dessa forma, a lista pode ser montada conforme a Figura 51, obtendo como o resultado a Figura 52.

Figura 51 - Utilizando a UICList para mostrar números de zero a nove.

```
1  import UICreator
2
3  class ContentView: UICView {
4      var body: ViewCreator {
5          UICList(style: .plain) {
6              UICForEach(0..<10) { index in
7                  UICRow {
8                      UICSpacer(spacing: 15) {
9                          UICLabel("Número \(index)")
10                             .font(.body)
11                             .textColor(.black)
12                     }
13                 }
14             }
15         }
16         .separator(style: .singleLine)
17     }
18 }
```

Fonte: Elaborado pelo Autor.

Figura 52 - Execução da UICList com números de zero a nove.



Fonte: Elaborado pelo Autor.

A implementação do *delegate* é utilizada para obter alguns eventos das células como quando elas são selecionadas pelo usuário. Para este trabalho, a implementação desse protocolo será para montar o cabeçalho, o rodapé e as ações das células. O algoritmo usado para reutilizar as células é replicado nas funções tableView(UITableView, viewForHeaderInSection: Int) e tableView(UITableView, viewForFooterInSection: Int). No entanto, as instâncias do RowManager para preencher as *views* do TableHeaderFooterView são obtidas por meio das propriedades *header* e *footer* da SectionManager. Além disso, a reutilização do cabeçalho e do rodapé é feita utilizando a função dequeueReusableHeaderFooterView(withIdentifier: String).

Conforme as Figura 53 e Figura 54, os objetos UICHeader e UICFooter são transformados, respectivamente, no cabeçalho e rodapé. A abstração do *delegate* e dos algoritmos para recalcular o tamanho das *views* são todos feitos pelo *framework*

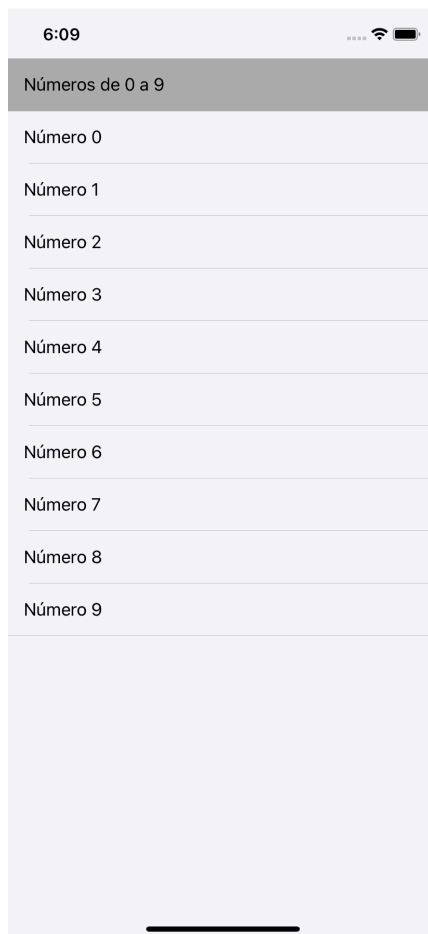
desenvolvido, tornando o código da *view* independente dos métodos do *dataSource* e do *delegate* da UITableView.

Figura 53 - UICList com cabeçalho utilizando a classe UICHeader.

```
1  import UICreator
2
3  class ContentView: UICView {
4      var body: ViewCreator {
5          UICList(style: .grouped) {
6              UICSection {
7                  UICHeader {
8                      UICSpacer(spacing: 15) {
9                          UICLabel("Números de 0 a 9")
10                     }
11                     .backgroundColor(.lightGray)
12                 }
13
14                 UICForEach(0..<10) { index in
15                     UICRow {
16                         UICSpacer(spacing: 15) {
17                             UICLabel("Número \(index)")
18                                 .font(.body)
19                                 .textColor(.black)
20                         }
21                     }
22                 }
23             }
24         }
25         .separator(style: .singleLine)
26     }
27 }
```

Fonte: Elaborado pelo Autor.

Figura 54 - Execução da UICList com cabeçalho definido pela classe UICHeader.



Fonte: Elaborado pelo Autor.

Como as alturas das *views* da lista são observadas, sempre que for necessário recarregar a altura de uma célula, ou cabeçalho ou rodapé, as funções tableView(UITableView, heightForRowAt: IndexPath), tableView(UITableView, heightForHeaderInSection: Int) e tableView(UITableView, heightForFooterInSection: Int) são chamadas pelo *delegate*. Dessa forma, a lista ajusta o tamanho das *views* sem comprometer as informações que estão sendo mostradas.

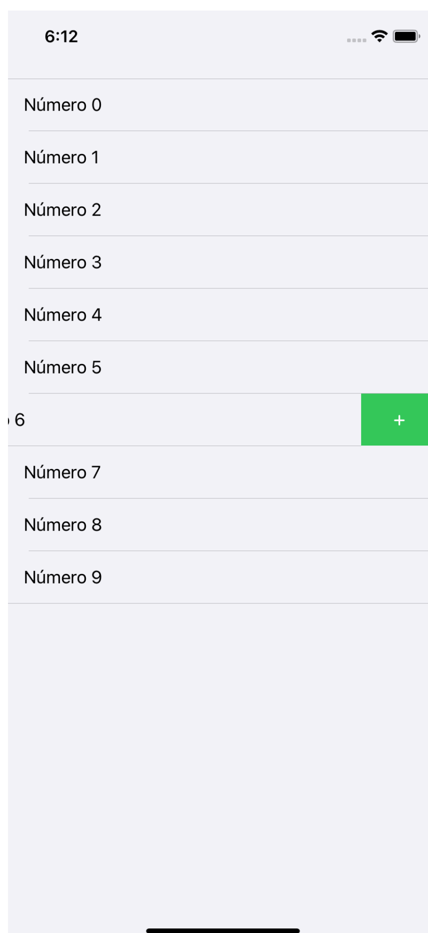
As ações são configuradas pelas funções tableView(UITableView, trailingSwipeActionsConfigurationForRowAt: IndexPath) e tableView(UITableView, leadingSwipeActionsConfigurationForRowAt: IndexPath). Usando a instância do RowManager, as ações são obtidas e montadas na lista, permitindo o usuário deslizar a célula para direita ou para a esquerda, acessando as ações disponíveis, conforme as Figura 55 e Figura 56.

Figura 55 – UICList listando célula com ação editar à direita.

```
1  import UICreator
2
3  class ContentView: UICView {
4      var body: ViewCreator {
5          UICList(style: .grouped) {
6              UICForEach(0..<10) { index in
7                  UICRow {
8                      UICSpacer(spacing: 15) {
9                          UICLabel("Número \(index)")
10                             .font(.body)
11                             .textColor(.black)
12                     }
13                 }
14                 .trailingActions {
15                     UICContextualAction(
16                         "Adicionar",
17                         .add,
18                         style: .normal
19                     )
20                     .backgroundColor(.systemGreen)
21                     .onAction { indexPath in /* Executa a ação */
22                         return true
23                     }
24                 }
25             }
26         }
27         .separator(style: .singleLine)
28     }
29 }
```

Fonte: Elaborado pelo Autor.

Figura 56 - Execução da UICList listando célula com ação editar à direita.



Fonte: Elaborado pelo Autor.

Além das ações da UITableView, existem outros recursos que são utilizados para customizar a lista ou integrar com recursos do iOS. Apesar disso, a UICList contempla os recursos bases para a UITableView como as seções, as células e as ações laterais. Abordar todos os recursos da lista na UICList necessita de uma análise para discutir se é interessante disponibilizar o recurso ou se seria melhor implementar uma nova classe que encapsula a lista do UIKit e modificar o *delegate* e o *dataSource* para incluir o novo recurso.

4.5 UICFlow

A UICollectionView é uma classe de *view* que mostra células com tamanho variável tanto na vertical quanto na horizontal. Ela utiliza a classe UICollectionViewLayout que deve ser implementada como a UICollectionViewFlowLayout para gerar o *layout* das células ou como elas serão

distribuídas dentro da *collection view*. A UICollectionViewFlowLayout é uma classe do UIKit que distribuí as células naturalmente, quando a barra de rolagem é vertical, as células são distribuídas uma após a outra até caber na horizontal; quando a rolagem é horizontal, as células são distribuídas até caber na vertical.

A estrutura da coleção é semelhante a estrutura da lista, exceto pela classe de *layout* que deve ser utiliza na montagem das células e a ausência do cabeçalho e do rodapé. Adaptar a classe UICollectionViewLayout para uma solução declarativa pode não ser uma tarefa simples devido a forma em que o UIKit trata as *views* dentro do *layout*. Então, será implementado a classe UICFlow que utiliza o *layout* UICollectionViewFlowLayout para encapsular a UICollectionView. Como não existem mudanças na estrutura da coleção, onde será usado o ListManager para montá-la, esta seção será dedicada as classes de *layout* declarativo.

O *layout* das células da coleção não segue um padrão, podendo ter tamanhos diferentes. Abstrair essa característica necessita criar dois conceitos: (a) item; (b) grupo; e (c) seção. O item é o *layout* da célula, enquanto o grupo pode agrupar tanto outros grupos quanto vários itens. A seção é para coleções que dividem o conteúdo em seções e, então, o *layout* pode ser diferente entre uma seção e outra, agrupando outros grupos e itens. Também, o *layout* é a combinação de largura e altura, podendo ser definido de três formas: (a) relativo; (b) fixo; e (c) automático. O atributo relativo é usado para calcular a altura e a largura com base em outro valor, podendo ser o tamanho do grupo ou da coleção, o fixo define um valor para altura e largura e o automático permite que a *auto layout* seja aplicado a célula utilizando o valor calculado por ele para definir o tamanho dela.

Então para gerenciar o *layout* da coleção foi implementado a classe UICCollectionLayoutManager que será usada pelo *delegate* da coleção para montar o *layout* das células. A seção é representada pela UICCollectionLayoutSection, enquanto as classes UICCollectionLayoutGroup e UICCollectionLayoutItem implementam o protocolo UICCollectionLayoutElement. Também, o enumerador CollectionLayoutSizeConstraint contém os atributos relativos para definir a largura e altura das células.

4.5.1 Layout da célula a partir do UICCollectionLayoutManager

A abstração do *layout* da coleção precisa obedecer ao processo de listagem que conta com células com índices {seção, célula}. Para isso, a classe UICCollectionLayoutManager contém um vetor que armazena as seções, que armazenam os objetos que calculam o *layout*. Além disso, enquanto na listagem todas as células são conhecidas, no *layout* é definido o padrão que é replicado a todas as células, contendo apenas uma regra ou um conjunto delas. Por isso, quando o gerenciador de *layout* da coleção acessa um índice na posição {x, y}, ele pode acessar o mesmo valor na posição {w, z}.

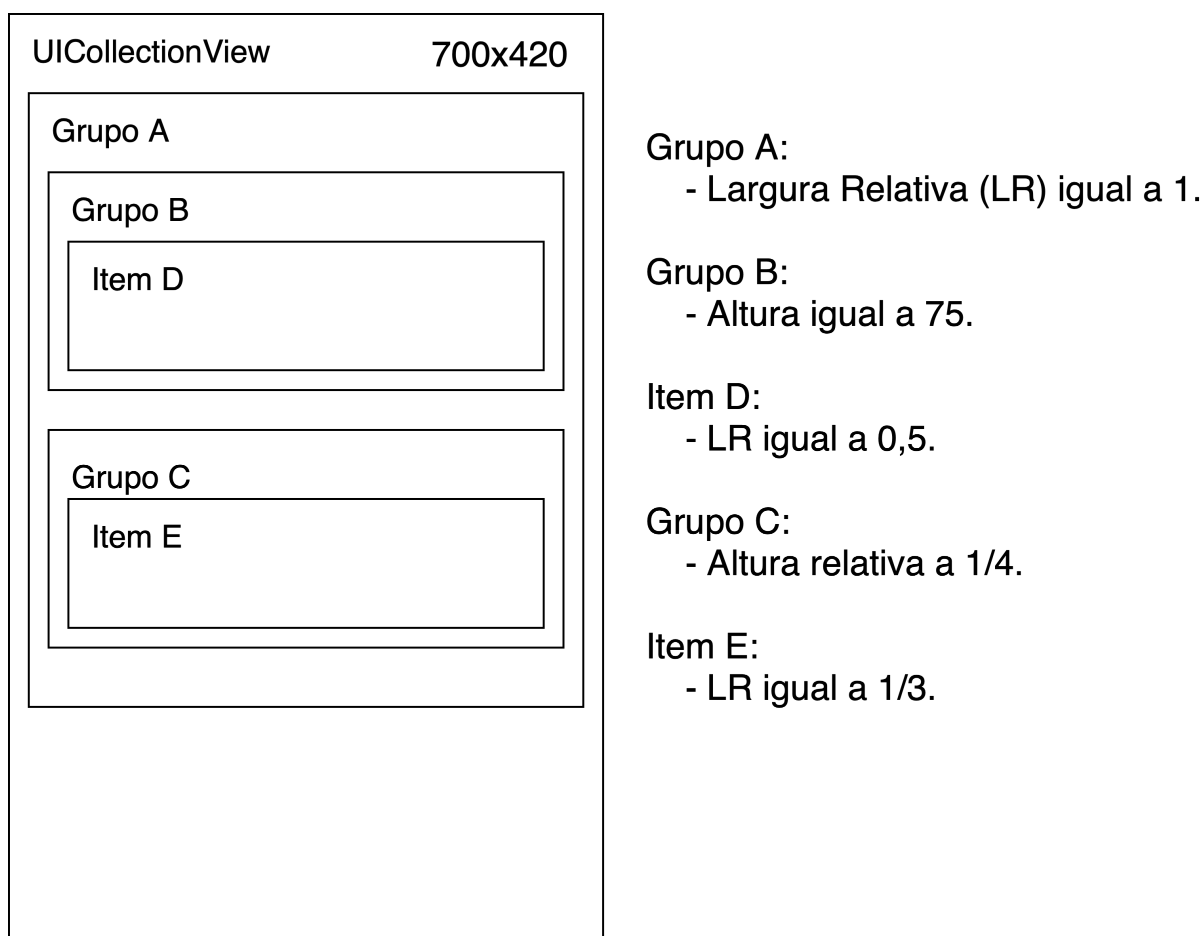
As seções podem ser definidas no fluxo declarativo ou criada pelo gerenciador para receber as regras de *layout*. A seção armazena um vetor de objetos que implementam o protocolo UICCollectionLayoutElement. Esse protocolo não contém variáveis ou funções que devem ser implementadas, mas limita os objetos que podem ser utilizados para instanciar a seção. A classe UICCollectionLayoutSection contém a função size(inside: CGSize, at: IndexPath) que retorna o tamanho da célula no índice informado, enquanto o primeiro parâmetro é o tamanho da UICollectionView na tela, que será usado para definir os valores do *layout*.

Quando a coleção pergunta o tamanho da célula para o gerenciador, ele procura a seção e chama a função para calcular o tamanho no índice informado pela coleção. A partir disso, a seção pergunta para os objetos quem contém o tamanho da célula no índice x e y para que seja calculado e retornado para o *delegate* da coleção. A classe UICCollectionLayoutGroup é uma estrutura recursiva que pode armazenar tanto outros grupos quanto os itens, estruturas finais. Dessa forma, ela consegue determinar quantas instancias da classe UICCollectionLayoutItem que possui e, utilizando o operador módulo, procurar o item que contém o índice informado para o cálculo da célula.

Ambas as classes contêm duas variáveis: (a) vertical; e (b) horizontal. Essas variáveis são do tipo CollectionLayoutSizeConstraint e armazenam a regra que será aplicada sobre o tamanho da coleção informado para determinar o tamanho da célula. No caso de serem automáticos ou fixos, os valores da coleção são ignorados, no entanto, quando são relativos, é utilizado o tamanho da coleção para calcular o tamanho da célula. Conforme a Figura 57, são definidos os grupos A, B e C e os itens D e E. Calculando os valores com base na altura e largura da coleção igual a 700x420,

o item D possui tamanho igual a 75x210 e o item E com tamanho igual a 175x140. Apesar desse cenário simples, a proposta é que os itens podem ser utilizados em até n células e um grupo possa ter vários itens dentro deles, gerando um layout bem complexo.

Figura 57 - Protótipo do layout das células definidos por valores relativos e fixos.



Fonte: Elaborado pelo Autor.

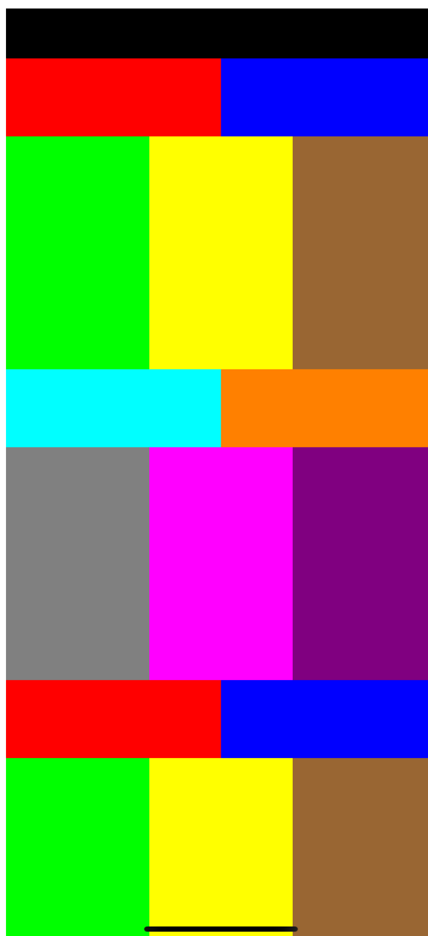
Com essas classes implementadas, o gerenciador pode ser utilizado pela coleção para montar o *layout* das células no *delegate* por meio do método collectionView(UICollectionView, layout: UICollectionViewLayout, sizeForItemAt: IndexPath). Além disso, o desenvolvedor define o tamanho das células utilizando a função declarativa layoutMaker(() -> UICCollectionLayoutSectionElement). Conforme a Figura 58 e Figura 59, foi criado uma coleção com células que contém o mesmo *layout* que o definido na Figura 57.

Figura 58 - Coleção UICFlow que utiliza a UICollectionViewFlowLayout para distribuir as células.

```
1  import UICreator
2
3  class ContentView: UICView {
4      var body: ViewCreator {
5          UICFlow {
6              UICForEach(0..<100) { index in
7                  UICRow {
8                      UICSpacer()
9                          .backgroundColor(colors[index%10])
10                         .insets()
11                 }
12             }
13         }
14         .layoutMaker {
15             UICCollectionLayoutGroup {
16                 UICCollectionLayoutGroup(vertical: .equalTo(75)) {
17                     UICCollectionLayoutItem(horizontal: .flexible(0.5), numberOfElements: 2)
18                 }
19
20                 UICCollectionLayoutGroup(vertical: .flexible(0.25)) {
21                     UICCollectionLayoutItem(horizontal: .flexible(1/3), numberOfElements: 3)
22                 }
23             }
24         }
25     }
26 }
```

Fonte: Elaborado pelo Autor.

Figura 59 - Execução da UICFlow formada por grupos e células com largura relativa.



Fonte: Elaborado pelo Autor.

O algoritmo que gerencia o *layout* da coleção consegue determinar os valores de altura e de largura de cada célula. Existem alguns casos em que os métodos implementados falham em calcular o tamanho das *views*, principalmente, com o aumento da quantidade de grupos aninhados. Nesses casos, embora as dimensões estejam corretas, foi identificado falhas quanto ao arredondamento de valores quebrados ou da impossibilidade de tratar dizimas periódicas no primeiro momento. Dessa forma, a atual implementação do objeto declarativo UICCollection e UICFlow não são estáveis e apresentam uma série de erros de *layout*.

O *framework* está disponível no GitHub através do *link* https://github.com/umobi/UICreator.

5 RESULTADOS

Este capítulo faz a análise de desempenho do *framework* desenvolvido com base na execução dos casos de testes estipulados.

5.1 Esquematização dos casos de testes

A avaliação do *framework* desenvolvido consiste em coletar dados sobre o desempenho e uso de memória por meio da execução dos casos de teste definidos no Apêndice A. Foram coletados diferentes tipos de dados usando tanto as ferramentas do Xcode quanto o Instruments, que é uma aplicação da Apple que coleta diversas métricas sobre o aplicativo. Os dados sobre a execução da aplicação são: (a) sobrecarga da CPU na inicialização; (b) sobrecarga no uso de memória; (c) uso da CPU durante a execução; (d) uso das memórias HEAP & VM; (e) tempo de execução; (f) tamanho do aplicativo; (g) tempo de compilação; e (h) quantidade de linhas. A partir desses dados, é possível compreender as estruturas que necessitam ser otimizadas e pontuar as qualidades de utilizar a programação declarativa.

A sobrecarga da CPU na inicialização é importante para conhecer o quanto a aplicação está gastando para inicializar todas as variáveis globais. A sobrecarga no uso de memória, apesar de necessitar de um intervalo de dados que represente todo o período em que a aplicação ficou ativa, permite detectar se houve uso excessivo de memória. O uso médio da CPU durante a execução permite detectar se houve algum estresse no processador para executar a aplicação. Ações do usuário, ou o carregamento de listas ou objetos que enviam diversas notificações tendem a consumir tempo de processamento. O consumo das memórias HEAP & *anonymous* VM (memória virtual anônima) é um dado mais específico quanto ao uso de memória global, uma vez que na memória HEAP estão todas as classes alocadas pela a aplicação e na memória virtual estão os endereços lógicos da aplicação (APPLE, 2013; SWIFT, 2019).

O tempo de execução é uma medida para avaliar quanto tempo demorou para o teste ser concluído, apesar de ser possível obter quais funções consumiram parte desse tempo. O tamanho do aplicativo é o resultado da compilação do código e conversão para binário executável, sendo uma medida pra avaliar a relação linhas de código e tamanho da aplicação. O tempo de compilação permite verificar o tempo

gasto pelo compilador para gerar o binário executável, apesar de ser importante destacar que o *framework* UIKit é pré-compilado, reduzindo o tempo de compilação em uma aplicação que apenas o utiliza. A quantidade de linhas é uma medida que detalha o quanto de código foi produzido para o teste, somatório do código editável, isolando os *frameworks* (HAMZA, 2018).

Os testes foram executados em um iPhone 11, rodando o iOS 14.0.1, e compilados para o iOS 10 em um Mac Mini 2018, rodando o macOS 11 Beta 9. Foram utilizados como ferramenta de desenvolvimento o Xcode 12 e o Swift 5.3. O dispositivo de teste conta com o processador A13 Bionic de 7 nm com *clock* máximo de 2666 MHz e 4 GB de memória RAM. O dispositivo de compilação conta com 16 GB de memória operando a 2666 MHz e processor i7-8700B com *clock* máximo de 4.6 GHz. Para efetuar os testes, os dispositivos tiveram sua carga de trabalho reduzida, embora não cessada, o que pode causar diferenças em relação aos dados coletados (EVERYMAC, 2018; FRUMUSANU, 2019; PRIDAY, 2019).

Os testes podem ser divididos em duas categorias, a primeira sendo os testes que não necessitam do usuário para completar a avaliação, enquanto o segundo são testes que dependem das ações do usuário para finalizá-los. Para os testes da segunda categoria, foram implementados em seu código estruturas que imitam o usuário e assim garante que tenham o mesmo desempenho independente da aplicação, uma vez que o usuário se comporta igualmente. Ações que necessitam de toques na tela ou de deslizamento sobre a tela foram feitas por meio de um usuário real, podendo impactar nos dados obtidos caso não seja reproduzido na mesma velocidade e precisão. Cada teste foi executado duas vezes devido o uso das duas ferramentas de avaliação, o Xcode e o Instruments.

Cada caso de teste resultou em dois aplicativos semelhantes, um construído utilizando apenas o *framework* UIKit e o outro utilizando o *framework* desenvolvido para este trabalho. Além disso, cada aplicativo recebeu três diretrizes de compilação diferentes sobre a otimização do Swift. A ideia era perceber os diferentes resultados que poderiam ser gerados caso a compilação alterasse. A primeira diretriz é para compilar utilizando a otimização de espaço, a segunda para otimização de tempo e a terceira sem otimização. Cada aplicativo recebeu o código A para aqueles usando somente o UIKit ou o código B utilizando o UICreator. A variante 1, 2 e 3 refere-se, respectivamente, as compilações de tempo, nenhuma e de espaço. Com isso, os

aplicativos foram nomeados concatenando: TC1, onde o “1” é o número do caso de teste; e A2, onde “A” é o uso do *framework* e “2” é a otimização do Swift.

A avaliação consiste em provar a viabilidade do uso do *framework* declarativo para o desenvolvimento de aplicativos iOS e não detalhes de otimização de uso de memória ou de processamento. Conforme as questões de pesquisa, será analisado se foi possível desenvolver um *framework* funcional que utilize os conceitos da programação declarativa e se é possível manter os padrões de projeto e de arquitetura sem reescrever o *back-end*. Dessa forma, os casos de teste foram construídos com base em fragmentos de uma aplicação real, uma vez que as telas de qualquer aplicativo são compostas de um número finito de elementos de interface, onde os mais cruciais são aqueles que listam coleções de objetos.

5.2 Análise dos casos de teste

O caso de teste CT01 avalia a diferença mínima entre os desempenhos dos dois *frameworks*. Conforme a Tabela 2, o uso de memória foi 0,8% maior no *framework* declarativo e houve uma carga de 4,9% maior da CPU em relação ao desempenho do UIKit. O tempo de processamento para completar o caso de teste foi 4,3% maior resultando em um consumo médio de 69,7 ms com o UIKit e 72,7 ms com o UICreator. O tempo de compilação e tamanho do aplicativo são resultado da soma dos dois *frameworks*, onde o UICreator conta com 23 mil linhas de código, resultando em um acréscimo de 36 segundos no tempo de compilação e 8,4 MB a mais no tamanho da aplicação, aproximadamente. Por último, o número de linhas foi 0,1% menor para um caso de teste com um elemento gráfico na tela. Demais variáveis não foram comentadas por terem o mesmo desempenho.

Tabela 2 - Resultados da execução do Caso de Teste 1.

Caso de Teste	Memória	Início CPU	CPU	HEAP & VM	Tempo	Tamanho	Compilação	Linhas
TC1A1	37,3 MB	14%	-	1,89 MB	72 ms	113,7 MB	39 s	724
TC1A2	37,1 MB	14%	-	1,89 MB	70 ms	113,8 MB	37 s	723
TC1A3	37,2 MB	15%	-	1,89 MB	67 ms	113,8 MB	37 s	724
TC1B1	37,4 MB	15%	-	1,88 MB	75 ms	122,2 MB	75 s	723
TC1B2	37,4 MB	14%	-	1,9 MB	70 ms	122,2 MB	73 s	722
TC1B3	37,8 MB	16%	-	1,95 MB	73 ms	122,2 MB	73 s	723

Fonte: Elaborado pelo Autor.

O caso de teste CT02 avalia os métodos do UIControl que utiliza do Selector do Objetive-C para executar uma função após um evento ocorrer. Nesse caso teste, o evento a ser capturado é o *touchUpInside*. Conforme a Tabela 3, o uso de memória nesse cenário foi 1% maior, com carga da CPU de 2% a mais na inicialização se comparado com a solução usando apenas o UIKit. As memórias HEAP & VM tiveram consumo superior de 3,2%. Durante a execução, a CPU variou igualmente em todos os casos. O número de linhas implementadas diminuiu 1,3% em relação a solução imperativa.

Tabela 3 - Resultados da execução do Caso de Teste 2.

Caso de Teste	Memória	Início CPU	CPU	HEAP & VM	Tempo	Tamanho	Compilação	Linhas
TC2A1	38,6 MB	14%	4%	3,15 MB	229 ms	113,7 MB	38 s	757
TC2A2	38,7 MB	16%	5%	3,13 MB	220 ms	113,8 MB	37 s	756
TC2A3	38,6 MB	15%	4%	3,12 MB	233 ms	113,8 MB	37 s	757
TC2B1	39 MB	15%	4%	3,2 MB	233 ms	122,2 MB	73 s	741
TC2B2	39 MB	15%	5%	3,33 MB	224 ms	122,3 MB	73 s	740
TC2B3	39,1 MB	16%	4%	3,18 MB	215 ms	122,2 MB	72 s	741

Fonte: Elaborado pelo Autor.

O caso de teste CT03 é uma evolução do primeiro caso pois adiciona a interface mais um objeto de texto e outro para organizar o arranjo das *views* na tela. Nesse caso, a UIStackView adiciona *constraints* de *layout* e calcula os tamanhos das *views*. Além disso, é verificado o funcionamento da classe declarativa UICStack. Conforme a Tabela 4, o uso de processamento durante a inicialização e de memória foram 2% e 1,3% maiores que os valores do UIKit. A memória HEAP & VM consumiram 4,8% a mais, enquanto o tempo de processamento aumentou 3,7%. O tamanho da aplicação foi 8,5 MB superior e o tempo de compilação gastou mais 34 segundos para completar. Nesse caso de teste, o número total de linhas implementadas foi equivalente.

Tabela 4 - Resultados da execução do Caso de Teste 3.

Caso de Teste	Memória	Início CPU	CPU	HEAP & VM	Tempo	Tamanho	Compilação	Linhas
TC2A1	37,4 MB	15%	-	2,08 MB	67 ms	113,7 MB	36 s	737

TC2A2	37,5 MB	15%	-	2,06 MB	76 ms	113,8 MB	38 s	736
TC2A3	37,3 MB	14%	-	2,07 MB	74 ms	113,8 MB	36 s	737
TC2B1	37,8 MB	15%	-	2,2 MB	78 ms	122,3 MB	74 s	737
TC2B2	38,1 MB	16%	-	2,19 MB	76 ms	122,3 MB	70 s	736
TC2B3	37,8 MB	15%	-	2,15 MB	71 ms	122,3 MB	70 s	737

Fonte: Elaborado pelo Autor.

O caso de teste CT04 verifica os elementos de interface UITextField que também são classes do UIControl. Conforme a Tabela 5, o uso de memória de o consumo de processamento na inicialização foram, respectivamente, 1,2% e 2% superiores. Durante a execução do caso de teste, onde o usuário teria que digitar um texto para e-mail e senha, além de pressionar um botão para mostrar o alerta na tela, a CPU e as memórias HEAP & VM foi equivalente. O tempo de processamento para finalizar o caso de teste foi 0,9% superior ao nativo. O tempo de compilação foi de 33 segundos e a diferença entre o tamanho dos aplicativos 8,5 MB. O aplicativo também ficou 2,5% menor em relação ao número de linhas implementadas usando a solução declarativa.

Tabela 5 - Resultados da execução do Caso de Teste 4.

Caso de Teste	Memória	Início CPU	CPU	HEAP & VM	Tempo	Tamanho	Compilação	Linhas
TC2A1	40,7 MB	15%	8%	5,05 MB	2,1 s	113,7 MB	39 s	798
TC2A2	40,9 MB	15%	8%	4,95 MB	2,15 s	113,8 MB	38 s	797
TC2A3	40,7 MB	15%	8%	4,92 MB	2,15 s	113,8 MB	37 s	798
TC2B1	41,4 MB	16%	8%	4,97 MB	2,13 s	122,3 MB	72 s	776
TC2B2	41,3 MB	15%	8%	5,01 MB	2,18 s	122,3 MB	68 s	775
TC2B3	41,3 MB	15%	8%	4,94 MB	2,15 s	122,3 MB	74 s	776

Fonte: Elaborado pelo Autor.

O caso de teste CT05 avalia o impacto utilizando a UIScrollView com 100 elementos de interface. Conforme a Tabela 6, o tempo de processamento na inicialização e o uso de memória foram, respectivamente, 3,5% e 2,1% superiores quando comparado com a solução imperativa. O processamento durante o deslocamento da *view* e o tempo gasto para concluir a tarefa, deslocando até o último elemento da *scroll*, apresentaram, respectivamente, 50% e 78,4% superiores ao

desempenho reportado pelos testes com UIKit. O uso das memórias HEAP & VM mostrou 36,4% superior ao *framework* nativo. O tempo de processamento e o tamanho da aplicação foi equivalente aos valores avaliados no caso de teste anterior. Por último, o projeto encolheu 48,7% em relação ao número de linhas implementadas.

Tabela 6 - Resultados da execução do Caso de Teste 5.

Caso de Teste	Memória	Início CPU	CPU	HEAP & VM	Tempo	Tamanho	Compilação	Linhas
TC2A1	38 MB	16%	2%	2,19 MB	241 ms	113,8 MB	37 s	1475
TC2A2	38 MB	17%	2%	2,18 MB	242 ms	113,8 MB	38 s	1474
TC2A3	38 MB	17%	2%	2,17 MB	227 ms	113,8 MB	38 s	1475
TC2B1	38,8 MB	17%	4%	3,09 MB	412 ms	122,3 MB	74 s	757
TC2B2	38,9 MB	17%	4%	2,92 MB	444 ms	122,3 MB	72 s	756
TC2B3	38,8 MB	18%	4%	3,08 MB	411 ms	122,3 MB	71 s	757

Fonte: Elaborado pelo Autor.

O caso de teste CT06 avalia um pequeno projeto de aplicativo, onde o usuário informa alguns dados cadastrais e é direcionado a uma segunda tela para conferi-los. Conforme a Tabela 7, o desempenho em relação ao uso de memória e tempo de processamento na inicialização foram, respectivamente, 1,2% e 1,7% superiores ao UIKit. Para concluir o caso de teste, o consumo de CPU foi 1,3% maior e o tempo de execução 6% superior. O uso das memórias HEAP & VM aumentou 4,4%. O tamanho da aplicação e o tempo de compilação não se alterou. A quantidade de linhas implementadas para o caso de teste foi 12,8% inferior quando na forma declarativa.

Tabela 7 - Resultados da execução do Caso de Teste 6.

Caso de Teste	Memória	Início CPU	CPU	HEAP & VM	Tempo	Tamanho	Compilação	Linhas
TC2A1	41,5 MB	16%	8%	4,73 MB	2,55 s	113,8 MB	39 s	1062
TC2A2	41,4 MB	18%	7%	4,75 MB	2,68 s	113,9 MB	38 s	1061
TC2A3	41,3 MB	16%	7%	4,73 MB	2,69 s	113,9 MB	38 s	1062
TC2B1	41,9 MB	17%	8%	4,96 MB	2,88 s	122,4 MB	75 s	925
TC2B2	41,9 MB	17%	7%	4,95 MB	2,88 s	122,4 MB	72 s	924
TC2B3	42 MB	17%	8%	4,95 MB	2,85 s	122,4 MB	73 s	925

Fonte: Elaborado pelo Autor.

O caso de teste 7 avalia a listagem utilizando a UITableView e a classe UICList, considerando as telas para adicionar e visualizar os contatos. Esse caso foi executado com 500 contatos cadastrados, sendo suficiente para detectar as diferenças entre as duas soluções. Conforme a Tabela 8, o uso de memória foi 52,8% superior e o tempo de processamento médio para a inicialização da aplicação foi de 43% contra 18,3% necessário pelo UIKit. Durante a execução do caso de teste, o *framework* declarativo consumiu 46,9% a mais de processamento se comparado com a solução imperativa. O tempo de processamento foi 43,2% superior para concluir o caso de teste. O uso das memórias HEAP & VM foi 16,7% maior. A diferença no tempo de compilação foi de 33 segundos. Enquanto a quantidade de linhas implementadas reduziu 18,9% com o *framework* declarativo.

Tabela 8 - Resultados da execução do Caso de Teste 7.

Caso de Teste	Memória	Início CPU	CPU	HEAP & VM	Tempo	Tamanho	Compilação	Linhas
TC2A1	42,9 MB	16%	21%	6,28 MB	5,24 s	114 MB	40 s	1451
TC2A2	43 MB	21%	21%	6,4 MB	5,4 s	114 MB	39 s	1450
TC2A3	43 MB	18%	22%	6,37 MB	5,26 s	114,2 MB	39 s	1451
TC2B1	65,8 MB	42%	30%	7,3 MB	7,54 s	122,5 MB	72 s	1176
TC2B2	65,6 MB	45%	34%	7,35 MB	7,67 s	122,7 MB	72 s	1175
TC2B3	65,8 MB	42%	30%	7,4 MB	7,55 s	122,5 MB	73 s	1176

Fonte: Elaborado pelo Autor.

O caso de teste 8 avalia o desempenho da listagem usando a UICollectionView implementada pela classe declarativa UICFlow, utilizando o UICollectionViewFlowLayout para dimensionar as *views*. Conforme a Tabela 9, o uso de processamento na inicialização e o uso de memória foram, respectivamente, 10,8% e 2,4% superiores comparado com o desempenho do UIKit. O consumo das memórias HEAP & VM foi 21% maior do que o avaliado na solução imperativa. O uso da CPU foi 11 vezes maior e o tempo de processamento para concluir o caso de teste foi 3,1 vezes superior ao *framework* nativo. O tempo de compilação foi de 36 segundos de diferença e o tamanho do aplicativo variou 8,4 MB. A quantidade de linhas necessárias para implementar esse caso de teste foi 5,6% inferior.

Tabela 9 - Resultados da execução do Caso de Teste 8.

Caso de Teste	Memória	Início CPU	CPU	HEAP & VM	Tempo	Tamanho	Compilação	Linhas
TC2A1	37,5 MB	15%	6%	1,88 MB	3,68 s	113,8 MB	43 s	814
TC2A2	37,6 MB	15%	6%	1,87 MB	3,74 s	113,9 MB	39 s	813
TC2A3	37,5 MB	14%	6%	1,84 MB	3,73 s	113,9 MB	40 s	814
TC2B1	38,4 MB	17%	65%	2,25 MB	11,52 s	122,3 MB	77 s	769
TC2B2	38,5 MB	16%	67%	2,25 MB	11,47 s	122,4 MB	76 s	768
TC2B3	38,3 MB	16%	67%	2,4 MB	11,56 s	122,3 MB	77 s	769

Fonte: Elaborado pelo Autor.

A interpretação dos dados será dividida em dois níveis: (a) geral; e (b) específico. Os dados são produto de um conjunto de componentes que desempenham funções específicas tanto do iOS e do Swift quanto do UIKit para a renderização e posicionamento gráficos dos objetos de interface. Por essa razão, compreender a dimensão do desempenho em cada cenário será relevante para uma análise completa dos pontos positivos e negativos quanto a implementação deste *framework*.

A especificação de casos de teste para a análise teórica em relação ao nível de complexidade dos métodos não foi realizada por requerer uma interpretação mais profunda dos motores gráficos do UIKit que tem suas *views* implementadas com base no *framework* CoreAnimation. Além disso, existem outras possibilidades de implementação nesse cenário das *views* utilizando recursos mais modernos como o *framework* Metal. Este trabalho consiste em implementar uma ferramenta em cima do UIKit, sendo transferido a responsabilidade quanto ao desempenho de renderização e otimização dos objetos gráficos. No entanto, as estruturas declarativas, bem como os métodos que permitam a configuração da *view* quanto aos seus estados de renderização, estes são foco de estudo para a análise direcionada deste trabalho.

Em âmbito geral, o desempenho do *framework* está sujeito as aplicações que são desenvolvidas para a resolução do problema. Neste caso, é utilizado para a construção de interface com número de telas limitadas e quantidade de objetos de interface limitados. Também, as telas contêm pelo menos uma *view* e pode conter até infinitas. Porém, o próprio motor gráfico do UIKit contém restrição devido aos objetos associados aos elementos gráficos, que começam a consumir memória e tempo de processamento para telas muito complexas. No âmbito específico, é avaliado as

estruturas quanto ao seu contexto. Em relação aos dados obtidos, os objetos críticos foram os responsáveis pela listagem de *views*, principalmente por remover a reciclagem de objetos. Além disso, será pontuado outros conceitos que colaboram para a construção da interface no iOS.

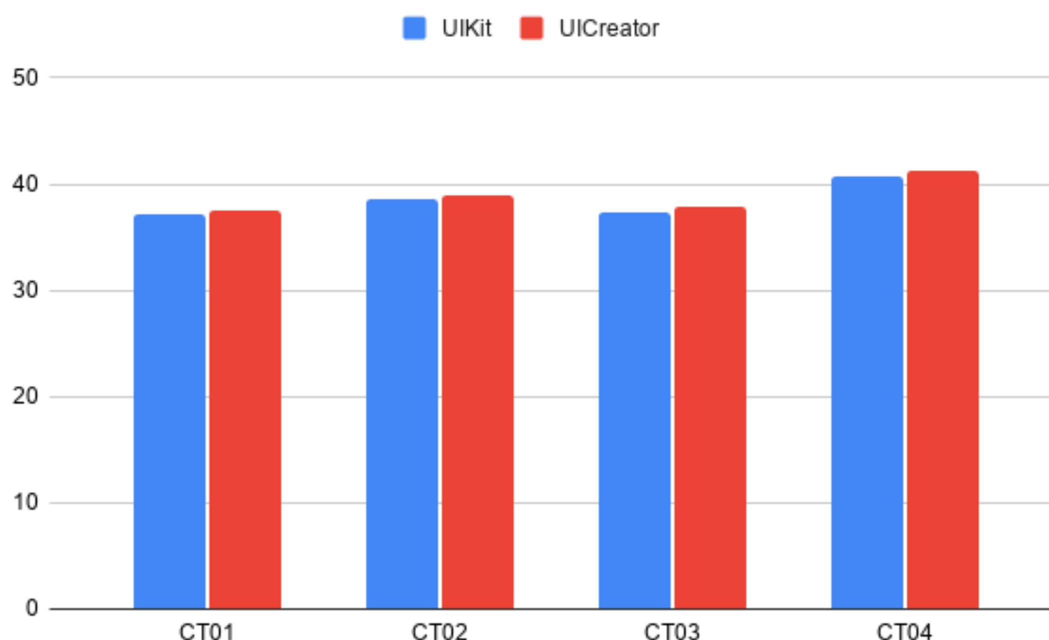
Os casos de teste 1 ao 4 serão analisados sob a perspectiva geral, pois utilizam componentes gráficos que simulam um produto mínimo. Para cada um, é possível avaliar o desempenho de um único elemento quanto aos seus estados de renderização definidos neste trabalho. Os casos de teste 5 ao 8 serão analisados de maneira específica considerando características específicas dos elementos gráficos.

5.3 Análise dos casos de teste 1 ao 4

A análise dos casos será subdividida quanto aos atributos coletados em cada caso de teste, porém alinhando as discussões a respeito do *framework* implementado. De acordo com a análise anterior, os casos de teste se mostraram relativamente estáveis, onde houve um aumento de custo tanto em memória quanto em processamento. Esse comportamento é esperado, uma vez que as estruturas implementadas são uma extensão do *framework* UIKit, não tratando de otimizações internas do *framework* nativo. Conforme as análises serão feitas, essa distinção será evidenciada. Além disso, os dados sobre o tamanho da aplicação e tempo de compilação se mostraram constantes e não farão parte dessa análise.

Em relação ao uso de memória, os aplicativos declarativos tiveram um impacto leve variando de 0,8% a 1,3%. Cada objeto de interface contém uma série de outros objetos alocados que gerenciam seus estados, assim como outras suas propriedades intrínsecas de cada um. Conforme a Figura 60, é possível identificar o leve impacto de memória nos casos de teste avaliado, sendo um forte indicativo de estabilidade para implementação de aplicações mais complexas, uma vez que é mínimo o impacto em memória considerando a execução do UIKit. Então, aplicações que utilizam os elementos de interface avaliados, como os botões, os textos e os campos de texto editáveis ligados as estruturas reativas, mostram-se capazes de um baixo incremento de memória quando na forma declarativa.

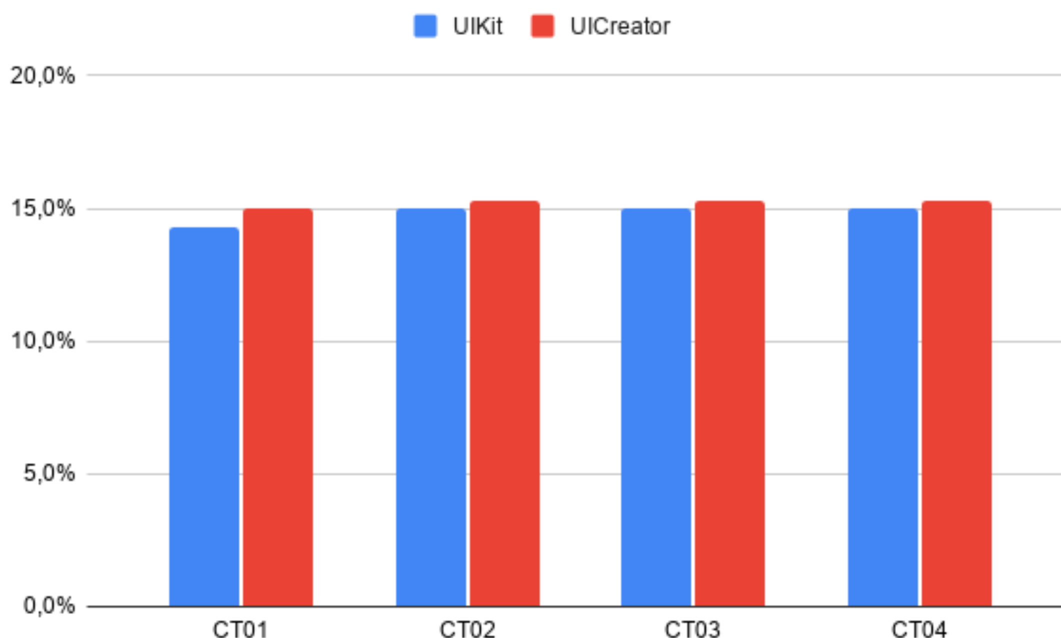
Figura 60 - Comparação entre consumo de memória nos casos de teste 1 ao 4.



Fonte: Elaborado pelo Autor.

O consumo da CPU durante o início da aplicação foi leve e manteve entre 2% a 4,9% comparado com as soluções do UIKit. Esse consumo reflete o carregamento da primeira tela como, também, o início da classe UIApplication com a renderização da janela e das *views*. Conforme a Figura 61, os passos necessários para montar as *views* no formato declarativo pouco impactou no desempenho geral da aplicação e, mesmo em casos mais complexos, como no caso de teste 4, não houve aumento significativo para a renderização da tela. Dessa forma, o tempo perdido para montar as telas que são criadas pelo *storyboard*, carregadas usando o NSCoder, foi transferido para a montagem dos objetos declarativos assim como a execução dos *callbacks* de estado de renderização.

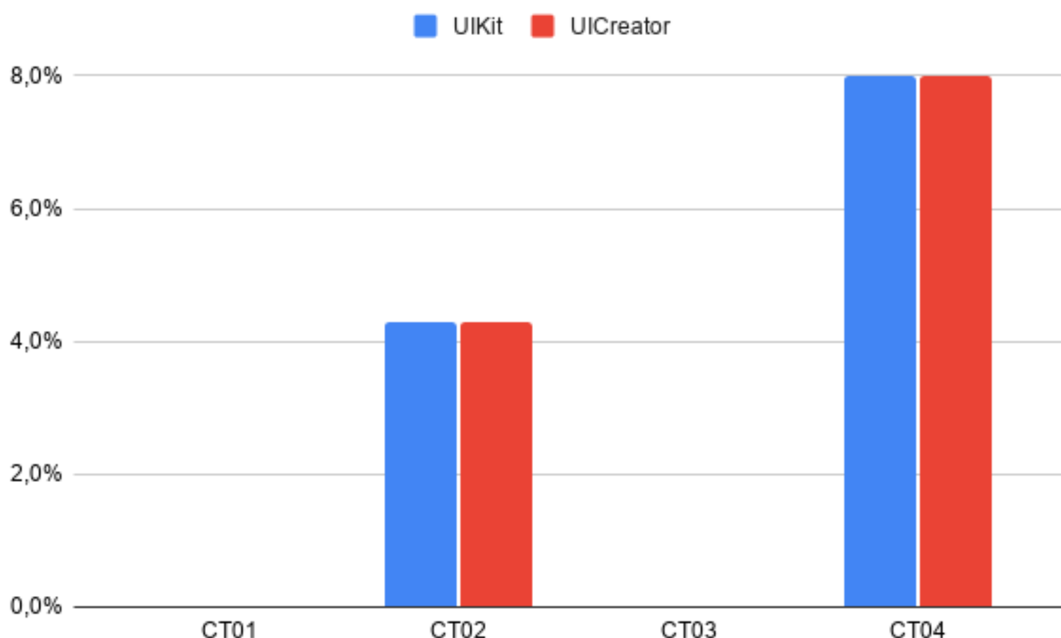
Figura 61 - Comparação entre uso de processamento na inicialização nos casos de teste 1 ao 4



Fonte: Elaborado pelo Autor.

O consumo durante a execução dos casos de teste manteve-se inalterado tanto no caso de teste 2 e 4. Conforme a Figura 62, apesar dos objetos serem configurados de forma diferente, um utilizando a programação imperativa e outro a programação declarativa, os métodos executados pelos objetos do UIKit são equivalentes, resultando no mesmo uso de processamento enquanto no estado de execução. No caso de teste 4, há o uso das estruturas reativas que monitoram a edição dos campos de texto com a notificação *editingChanged* da UIControl. Com isso, apesar do aumento dos passos necessários para a efetivar a solução declarativa, isso não se mostrou carregado para o processador durante a execução, pois os métodos reativos são simples e implementados com base no NotificationCenter.

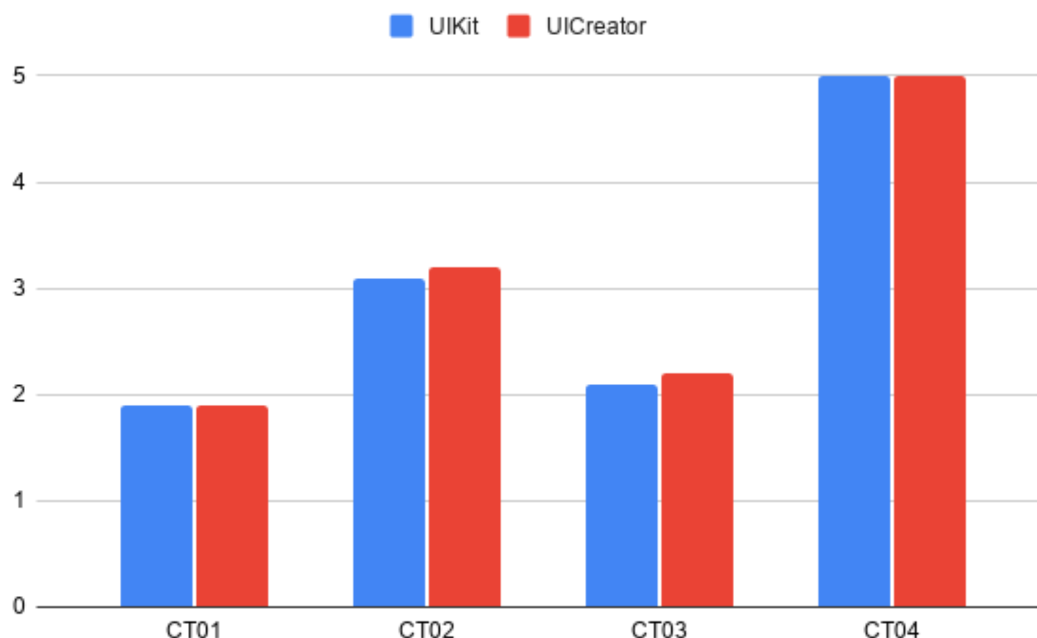
Figura 62 - Comparação entre uso de processamento nos casos de teste 1 ao 4.



Fonte: Elaborado pelo Autor.

As memórias HEAP & VM foram levemente consumidas nos casos de teste, não apresentando uma variação considerável quando comparado com o desempenho do UIKit. Conforme a Figura 63, os casos de teste 1 e 4 não apresentaram valores distintos, enquanto o caso de teste 2 e 3 tiveram um aumento no consumo das memórias. Apesar disso, esses valores representam uma média da execução dos casos de teste e por isso são interpretados como um consumo leve para os quatro casos de teste. Além disso, o UIKit instancia diversas classes e contém uma série de nós em memória, no caso de teste 1 foram 202 instancias contra 188 na solução declarativa, somente do UIKitCore; enquanto isso, o UICreator instanciou 14 objetos.

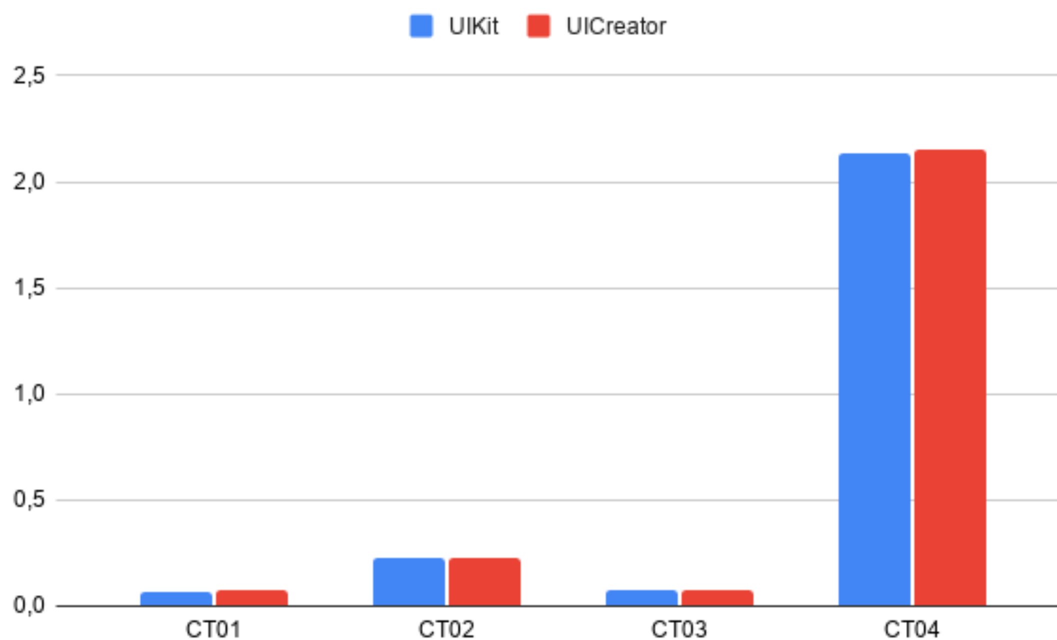
Figura 63 - Comparação entre uso das memórias HEAP & VM nos casos de teste 1 ao 4.



Fonte: Elaborado pelo Autor.

O tempo gasto para concluir o caso de teste considerando apenas o tempo gasto na *main thread*, onde todas as operações de interface ocorrem no UIKit. Conforme a Figura 64, exceto pelo caso de teste 2, foi identificado um consumo leve em tempo de processamento, indicando um uso estável do tempo para renderizar os objetos de interface e alterar os estados dos objetos declarativos. No caso de teste 2, onde é avaliado um botão da classe UIButton, a diferença de 3,3 ms reportado se deve aos métodos do UIKit para conectar os objetos do *storyboard* a UIViewController com o NSCoder, uma vez que demais métodos seguem as mesmas chamadas e estruturas após a renderização da interface.

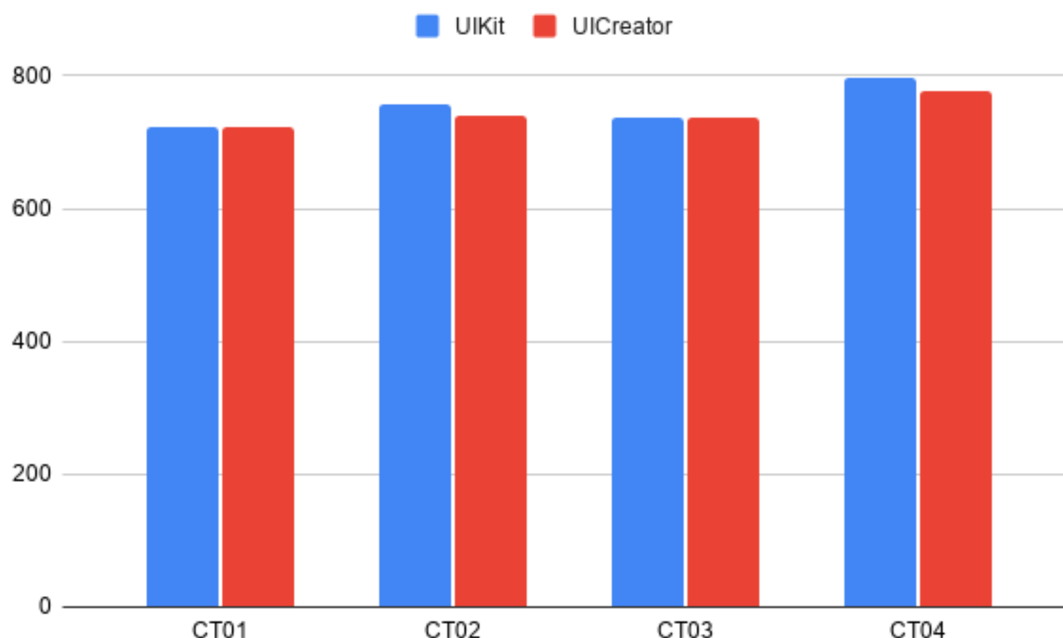
Figura 64 - Comparação entre tempo de processamento nos casos de teste 1 ao 4.



Fonte: Elaborado pelo Autor.

A quantidade de linhas segue a tendência negativa de redução das aplicações quanto ao tempo necessário para codificar uma solução utilizando o UIKit. O maior decremento foi de 2,5% em uma aplicação de uma tela com dois campos de texto e um botão. Conforme a Figura 65, apenas o caso de teste 3 não houve redução no número de linhas, aumentando a diferença conforme a quantidade de elementos de interface aumenta na aplicação. Com a programação declarativa, os métodos para integrar os dados da aplicação com os dados do objeto de interface é simplificado, além da eliminação do *storyboard* e da UIViewController com vários *delegates* e métodos para sincronizar e validar dados.

Figura 65 - Comparação entre quantidade de linhas nos casos de teste 1 ao 4.



Fonte: Elaborado pelo Autor.

Com base nesses dados e informações, os métodos implementados para integrar a programação declarativa aos projetos com o UIKit se mostraram estáveis em relação ao consumo de memória e de processamento. Com isso, os benefícios de utilizar esses recursos e paradigma de programação tende a tornar mais simples a arquitetura do projeto, além de retirar trechos de códigos necessários para sincronizar informações de interface e de dado. Evidentemente, não houve redução da carga computacional com o *framework* implementado, sendo um fator de discussão para trabalhos futuros ou uso de outras técnicas para este problema. O UIKit se mostrou fortemente adaptável a programação declarativa, assim como os benefícios dessa integração foram reportados nesta análise.

5.4 Análise dos casos de teste 5 ao 8

A análise específica será dividida nos quatro casos de teste, do 5 ao 8. Cada caso explora alguns recursos específicos do UIKit e representam telas padrões que estão presentes em diversos aplicativos. O primeiro caso de teste a ser analisado, o CT05, é um exemplo comum onde é utilizado a UIScrollView para mostrar várias *views*. O caso de teste CT06 representa os formulários de cadastro com vários

campos como nome, CPF, e-mail e outros. O terceiro caso de teste, CT07, é comumente implementado para listagem de *views* onde existe um vetor de dados homogêneo e que será utilizado para prencher a lista. O último caso de teste, CT08, também é utilizado em aplicações como a App Store ou o aplicativo Fotos, porém, por se tratar da UICollectionView, contém implementações diferentes no *layout* da coleção.

O caso de teste 5, apesar de ser formado pela UIScrollView, tem o carregamento da hierarquia no momento da inicialização. Conforme os resultados, não houve impacto significativo de processamento. O custo computacional de memória e de processamento ocorre durante a rolagem do conteúdo onde ocorre chamadas aos métodos de *layout* da UIView. A partir desse momento, a UIScrollView inicia o processo de atualização do *layout*, gerando um envio recursivo para efetivar os *callbacks* de *layout* referente as funções onAppear((UIView) -> Void), onDisappear((UIView) -> Void) e onLayout((UIView) -> Void) executadas pelo RenderManager. Esses métodos auxiliam na integração de outros componentes do UIKit e também fazem parte do ciclo de vida da UIView. Dessa forma, houve uma perda significativa tanto de memória quanto de processamento por conta dos métodos de *layout* para manter integro o *framework* declarativo.

O caso de teste 6 avalia os métodos da classe UITextField semelhante ao caso de teste 4, porém passando os dados informados pelo usuário para a tela de conferência. Conforme a avaliação geral, os dados coletados mostraram que mesmo com o aumento na quantidade de *views* e controle de fluxo, o impacto continuou leve nas aplicações baseadas em formulários. Apesar disso, a implementação do UIKit não escuta as alterações no texto para manter salvo em uma variável como no *framework* declarativo, que utiliza nas estruturas reativas para manter a UITextField e a variável correspondente sempre atualizadas. Dessa forma, a atual implementação do caso de teste 6 mostrou ser coerente com os métodos e estruturas desenvolvidas pelo *framework* declarativo como, também, a redução de 12,8% na quantidade de linhas para implementar as telas.

O caso de teste 7 avalia todo o algoritmo desenvolvido para manter a integridade da UITableView. Segundo os dados coletados, houve uma sobrecarga considerável tanto no uso de processamento quanto no uso de memória. Além da falta da reciclagem de *views*, as células escutam as alterações de *layout* de sua hierarquia

para pedir a lista que recarregue as células com os tamanhos corretos. Esse procedimento busca suprimir os erros no cálculo do *auto layout*, uma vez que as células são todas instanciadas dinamicamente. Esses erros são evitados quando as listas são definidas pelo *storyboard* e, no *framework* declarativo acabam gerando *loops* de atualização para que as células correspondem o tamanho correto. Dessa forma, é necessário revisar todas as classes implementadas para reduzir o consumo de memória como também corrigir os problemas envolvendo o *layout* das *views*.

O caso de teste 8 combina a solução da UITableView com o gerenciamento declarativo do *layout* da coleção. Com uma hierarquia composta de uma *view*, esse caso de teste evitou a recursividade dos métodos de *layout* durante a montagem da coleção, porém apresentou falhas de renderização durante a rolagem da tela. Apesar do *layout* da coleção ser declarativo foi utilizado a UICollectionFlowLayoutView para montar o quadro das células, não utilizando todas as possibilidades envolvendo essa implementação. Seria necessário implementar uma classe para gerar os quadros corretos para a célula bem com a disposição de cada uma na tela. Os erros de renderização ocorreram por conta do *auto layout* devido a falta de *constraints* na célula, fazendo com que elas fossem criadas com zero de altura e de largura, corrigindo à medida que o usuário visualizasse os conteúdos. Dessa forma, a coleção envolve duas partes para gerenciar o *layout* da célula o que se mostrou inviável em termos de custo computacional.

6 CONSIDERAÇÕES FINAIS

Este trabalho apresentou os conceitos envolvendo o desenvolvimento do *framework* declarativo para iOS, considerando os recursos atuais de programação. Além disso, há o suporte mínimo para o iOS 10 e o tvOS 10, por compartilharem o UIKit como *framework* de interface gráfica. Após a implementação, foi aplicado os casos de testes para avaliar o desempenho de cada componente, desde os mais simples ao mais complexos como as listas e coleções. Apesar do profundo estudo envolvendo o Swift e o UIKit, existem uma série de questões a serem trabalhas para entregar um *framework* completo e com desempenho ótimo.

A arquitetura MVC é uma característica fundamental do UIKit e, apesar de ser possível utilizar outras arquiteturas, o código base desse *framework* foi estruturado utilizando essa arquitetura. A interface começa na UIViewController que chama os métodos de ciclo de vida e faz a montagem da tela para renderizar as informações contidas no *model*. O *framework* resultante desse trabalho manteve a função *viewDidLoad* e implementou a estrutura UICOutlet para que seja possível aplicar os códigos imperativos de projetos consolidados no UIKit, evitando a necessidade da reescrita no *back-end* da aplicação.

O UIKit contém uma série de *views* que são específicas para cada elemento gráfico implementado por ele. Os essenciais foram adaptados para a solução declarativa, permitindo a criação das telas e personalização de cada elemento que compõe o aplicativo para iOS. Com a implementação dos objetos UICList e UICCollection, mesmo que não completos em termos de recursos e características contidas nos objetos UITableView e UICollectionView, foi possível suprimir os códigos envolvendo o processo de listagem com o UICForEach e os objetos UICRow, UICHeader e UICFooter. No entanto, por conta do *auto layout* e da falta de reciclagem de *views*, os testes mostraram um desempenho ruim quando o número de elementos aumenta e até mesmo problemas envolvendo o *layout* das células.

Os projetos que estejam utilizando o UIKit podem ser traduzidos para utilizar o *framework* declarativo tanto reescrevendo partes da interface com a programação declarativa, ou utilizando os protocolos UICViewRepresentable e UICViewControllerRepresentable, tornando as *views* e *view controllers* em objetos declarativos. Como o *framework* é escrito em cima do UIKit, o programador ainda

consegue acessar objetos de interface que estejam na hierarquia e configurar propriedades desejadas e que não estejam disponíveis na solução declarativa.

Além das estruturas discutidas neste trabalho, a proposta permitiu explorar vários outros recursos de programação para simplificar a codificação de interface no iOS. Isso devido a estrutura declarativa implementada com base nos estágios de renderização da UIView e com a programação reativa das estruturas Value e Relay. O contínuo estudo e avanço sobre o tema desde trabalho permitirá conhecer novos conceitos envolvendo a interface gráfica, além, no melhor cenário, criar um *framework* que seja multiplataforma incluindo os dispositivos da Apple como também aqueles com Android, Windows e, talvez, até mesmo a plataforma web.

6.1 Trabalhos futuros

Como discutido anteriormente, existem duas formas de implementar um objeto no Swift, utilizando as classes ou estruturas. O *framework* foi construído desde o início para que todos os objetos implementados sejam classes e não estruturas. Isso gerou um impacto na memória e também em tempo de processamento, uma vez que os objetos declarativos e os objetos de interface ficam conectados usando um recurso para manter as referências entre esses dois objetos seguras, para não causar ciclo de memória. Com o desenvolvimento desse trabalho foi possível identificar uma série de características que não foram conhecidas antecipadamente, permitindo afirmar que não há necessidade de manter os objetos declarativos em memória, uma vez que não há dependência do objeto declarativo com o objeto de interface.

Foi detectado que o *auto layout* causou um impacto de processamento na aplicação e gerou ambiguidade enquanto uma *view* era renderizada, principalmente naquelas que formavam a interface de uma célula para a listagem de conteúdo. Por conta disso, este trabalho necessita que um estudo sobre os conceitos de *layout* de *views* que aborde os cálculos de *layout* de forma a suprimir as diversas opções que o *constraint layout* do UIKit fornece e que não é utilizada na estrutura declarativa, gerando principalmente consumo de tempo de processamento na aplicação.

Outra questão fundamental e que exige um estudo profundo é sobre a reciclagem dos objetos. Este trabalho não foi discutido esse conceito e nem o impacto dele, por não ter sido identificado uma forma para comparar dois objetos declarativos e atualizar a UIView conforme a igualdade desses objetos. No caso da listagem, as

células formadas por um UICForEach são idênticas, mudando apenas os dados que serão renderizados, dessa forma poderia ser reutilizado a UIView e atualizado apenas seus valores. Neste trabalho, há um gasto para retirar a *view* que está na célula e adicioná-la novamente, exigindo do UIKit renderizar novamente a *view* e calcular uma série de propriedades que estavam calculadas.

As *views* podem ser reescritas para remover a dependência com o UIKit, uma vez que elas são resultado da implementação sobre o CoreAnimation com o CALayer. O SDK do iOS contém o *framework* Metal, utilizado em desenvolvimento de jogos, mas que permite a implementação de *views* que são renderizadas pela GPU dos dispositivos da Apple, gerando um ganho de *performance*. Também, poderia ser restruturado as *views* e implementado uma solução própria usando o CoreAnimation como é feito por outros projetos que levam o desenvolvimento Android para a plataforma iOS, por exemplo.

Por último, podem ser feitas pesquisas e análises sobre o impacto da programação declarativa nas empresas e trabalhos profissionais. A programação declarativa permite abstrair uma série de instruções imperativas e o uso desse paradigma pode causar impactos nas etapas de construção do *software*.

REFERÊNCIAS

ADAMS, Radoslava Leseva; LESEV, Hristo. **“Hello, Swift!” - A Tutorial for Building an iOS App. Migrating To Swift From Flash And Actionscript**, Berkeley, CA, EUA, p. 71-103, 2016. Apress. http://dx.doi.org/10.1007/978-1-4842-1666-8_5.

APPLE. **Memory Usage Performance Guidelines: About the Virtual Memory system**. 2013. Disponível em: <https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/AboutMemory.html>. Acesso em: 28 set. 2020.

APPLE. **NotificationCenter: A notification dispatch mechanism that enables the broadcast of information to registered observers**. Disponível em: <https://developer.apple.com/documentation/foundation/notificationcenter>. Acesso em: 30 ago. 2020.

APPLE. **Submit Your Apps to the App Store**. Disponível em: <https://developer.apple.com/ios/submit/>. Acesso em: 6 mar. 2020.

APPLE. SwiftUI: **Better apps. Less code**. Disponível em: <https://developer.apple.com/xcode/swiftui/>. Acesso em: 6 mar. 2020.

APPLE. **UIControl**: **The base class for controls, which are visual elements that convey a specific action or intention in response to user interactions**. Disponível em: <https://developer.apple.com/documentation/uikit/uicontrol>. Acesso em: 27 abril. 2020.

APPLE. **UIKit**: **Construct and manage a graphical, event-driven user interface for your iOS or tvOS app**. Disponível em: <https://developer.apple.com/documentation/uikit>. Acesso em: 6 mar. 2020.

APPLE. **UIView**: **An object that manages the content for a rectangular area on the screen**. Disponível em: <https://developer.apple.com/documentation/uikit/uiview>. Acesso em: 27 abr. 2020.

APPLE. **UIViewController**: **An object that manages a view hierarchy for your UIKit app**. Disponível em: <https://developer.apple.com/documentation/uikit/uiviewcontroller>. Acesso em: 29 fev. 2020.

APPLE. **UIWindow**: **The backdrop for your app’s user interface and the object that dispatches events to your views**. Disponível em: <https://developer.apple.com/documentation/uikit/uiwindow>. Acesso em: 27 abr. 2020.

BREWSTER, Keith. **What the Heck is Declarative Programming, Anyways?** 2019. Disponível em: <https://dev.to/brewsterbhg/what-the-heck-is-declarative-programming-anyways-2bj2>. Acesso em: 29 fev. 2020.

CHAUDHARY, Amit. **Swift 5 vs. Objective-C: Compared Each Topic Of Both Programming Languages With Examples**. [s.l.], eBook Kindle, 2019.

COPADATA. **O que é HMI?** Disponível em: <https://www.copadata.com/pt/produtos/interface-homem-maquina-hmi/>. Acesso em: 22 mar. 2020.

DOWNEY, Eric. **Practical Swift**. [s.l.]: Apress, 2016.

DRECHSLER, Joscha; MOGK, Ragnar; SALVANESCHI, Guido; MEZINI, Mira. **Thread-safe reactive programming. Proceedings Of The Acm On Programming Languages**, [s.l.], v. 2, n. , p. 1-30, 24 out. 2018. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/3276477.

EVERYMAC. **Apple Mac mini "Core i7" 3.2 (Late 2018) Specs**. 2018. Disponível em: <https://everymac.com/systems/apple/mac_mini/specs/mac-mini-core-i7-3.2-late-2018-specs.html>. Acesso em: 28 set. 2020.

FACEBOOK. **A declarative UI framework for iOS**. Disponível em: <https://componentkit.org/>. Acesso em: 29 mar. 2020.

FACEBOOK. **Litho**: **A declarative UI framework for Android**. Disponível em: <https://fblitho.com/>. Acesso em: 6 mar. 2020.

FACEBOOK. **React**: **O que é um *framework*?**. Disponível em: <https://pt-br.reactjs.org/>. Acesso em: 6 mar. 2020.

FARRER, Harry. **Algoritmos Estruturados**. 3. ed. Belo Horizonte: LTC - Livros Técnicos e Científicos Editora S.A., 2011.

FEILER, Jesse. **Beginning Reactive Programming with Swift**. Plattsburgh, Nova Iorque: Apress, 2018.

FISCHER, Sebastian. **On Functional Logic Programming and its Application to Testing**. 2010. Disponível em: <https://www-ps.informatik.uni-kiel.de/~sebf/thesis.pdf>. Acesso em: 2 abr. 2020.

FRUMUSANU, Andrei. **The Apple iPhone 11, 11 Pro & 11 Pro Max Review: Performance, Battery, & Camera Elevated**. 2019. Disponível em: <https://www.anandtech.com/show/14892/the-apple-iphone-11-pro-and-max-review/2>. Acesso em: 28 set. 2020.

GARCÍA, Cristian González; ESPADA, Jordán Pascual; BUSTELO, Begoña Cristina Pelayo García; LOVELLE, Juan Manuel Cueva. **Swift vs. Objective-C: A New Programming Language**. 2018. Disponível em: https://documat.unirioja.es/servlet/articulo?codigo=5574309. Acesso em: 21 abr. 2020.

GOOGLE. **Android Studio**: **Android Studio provides the fastest tools for building apps on every type of Android device**. Disponível em: <https://developer.android.com/studio>. Acesso em: 6 mar. 2020.

GOOGLE. **Design beautiful apps**. Disponível em: <https://flutter.dev/>. Acesso em: 6 mar. 2020.

GRAY, Anthony. **Swift Pocket Reference: Programming for IOS and OS X**. 2. ed. Sebastopol, Eua: O'reilly Media, Inc., 2016.

HAMZA, Yousef. **Creating and Distributing an iOS Binary Framework**. 2018. Disponível em: <https://instabug.com/blog/ios-binary-framework/>. Acesso em: 28 set. 2020.

JIN, Xinxin; GRISWOLD, William G.; ZHOU, Yuanyuan. ANEL: robust mobile network programming using a declarative language. **Proceedings Of The 5th International Conference On Mobile Software Engineering And Systems - Mobilesoft '18**, [s.l.], p. 202-213, 2018. ACM Press. http://dx.doi.org/10.1145/3197231.3197237

JIRÁSEK, Matěj Kašpar. **MVC without the C: What will SwiftUI change in app architecture?** 2019. Disponível em: <https://blog.thefuntasty.com/mvc-without-the-c-what-will-swiftui-change-in-app-architecture-c9ce3f49d256>. Acesso em: 6 mar. 2020.

JANSEN, Remo H. **Hands-On Functional Programming with TypeScript**. Birmingham, Uk: Packt Publishing, 2019.

JOHNSON, Stephen C. **Yacc: Yet Another Compiler-Compiler**. 1975. Disponível em: <http://web.wlu.ca/science/physcomp/ikotsireas/CP465/W3-BNF-LEX-YACC/Yacc_Introduction.pdf>. Acesso em: 02 abr. 2020.

KACZMAREK, Stefan; LEES, Brad; BENNETT, Gary. **Swift 5 For Absolute Beginners**. 5. ed. Nova Iorque, EUA: Apress, 2019.

KMETIUK, Anatolii. **Mastering Functional Programming**. Birmingham, ING: Packt Publishing, 2018.

LAMPSON, Butler. **Declarative Programming: the light at the end of the tunnel**. 2010. Disponível em: <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/Lampson-Declarative-Programming-final.pdf>. Acesso em: 2 abr. 2020.

LASO-MARSETTI, Felipe. **Model-View-Controller (MVC) in iOS – A Modern Approach.** 2019. Disponível em: <https://www.raywenderlich.com/1000705-model-view-controller-mvc-in-ios-a-modern-approach>. Acesso em: 5 mar. 2020.

MARTIN, MG. **Swift: Advanced Detailed Approach To Master Swift Programming With Latest Updates**. [s.l], eBook Kindle, 2019.

MEDEIROS, Higor. **Introdução ao Padrão MVC.** 2013. Disponível em: <http://www.devmedia.com.br/introducao-ao-padrao-mvc/29308>. Acesso em: 29 fev. 2020.

MICHAELSON, Greg. **An Introduction to Functional Programming Through Lambda Calculus**. Mineola, EUA: Dover Publications, Inc., 2011.

NEUBURG, Matt. **Programming IOS 10: Dive Deep Into Views, View Controllers, and Frameworks**. 7. ed. Sebastopol, Ca, Eua: O'reilly Media, Inc., 2017.

NILSSON, Henrik; CHUPIN, Guerric. **The Arpeggigon: declarative programming of a full-fledged musical application**. 2017. Disponível em: <http://eprints.nottingham.ac.uk/38657/1/padl2017-techreport.pdf>. Acesso em: 20 abr. 2020.

NOOR, Joseph; TSENG, Hsiao-yun; GARCIA, Luis; SRIVASTAVA, Mani. DDFlow. **Proceedings Of The International Conference On Internet Of Things Design And Implementation**, [s.l.], p. 172-177, 15 abr. 2019. ACM. http://dx.doi.org/10.1145/3302505.3310079.

NOVAC, Ovidiu Constantin; NOVAC, Mihaela; GORDAN, Cornelia; BERCZES, Tamas; BUJDOSO, Gyongyi. Comparative study of Google Android, Apple iOS and Microsoft Windows Phone mobile operating systems. **2017 14th International Conference On Engineering Of Modern Electric Systems (emes)**, [s.l.], p. 154-159, jun. 2017. IEEE. http://dx.doi.org/10.1109/emes.2017.7980403.

O'REGAN, Gerard. **A Brief History of Computing**. Mallow, IR: Springer, 2008.

PRIDAY, Richard. **iPhone 11 and iPhone 11 Pro Battery Size and RAM Revealed**. 2019. Disponível em: https://www.tomsguide.com/news/iphone-11-and-iphone-11-pro-battery-size-and-ram-revealed. Acesso em: 28 set. 2020.

REYES, Maritza; PEREZ, Cynthia; UPCHURCH, Rocky; YUEN, Timothy; ZHANG, Yuanlin. **Using Declarative Programming in an Introductory Computer Science Course for High School Students**. 2016. Disponível em: <https://dl.acm.org/doi/abs/10.5555/3016387.3016496>. Acesso em: 1 maio 2020.

SAUVÉ, Jacques. **Frameworks: o que é um framework?.** Disponível em: <http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/frame/oque.htm>. Acesso em: 29 fev. 2020.

SCHUSTER, Christopher; FLANAGAN, Cormac. Reactive programming with reactive variables. **Companion Proceedings Of The 15th International Conference On Modularity - Modularity Companion 2016**, [s.l.], p. 29-33, 2016. ACM Press. http://dx.doi.org/10.1145/2892664.2892666.

SEL, Tam. **SWIFT PROGRAMMING FOR BEGINNERS: learn coding fast**. [s.l.], eBook Kindle, 2020. 132 p.

SILVA, Yasin N.; ALMEIDA, Isadora; QUEIROZ, Michell. SQL: from traditional databases to big data. **Proceedings Of The 47th Acm Technical Symposium On Computing Science Education - Sigcse '16**, [s.l.], p. 413-418, 2016. ACM Press. http://dx.doi.org/10.1145/2839509.2844560.

SMARAGDAKIS, Yannis. **Next-Paradigm Programming Languages: What Will They Look Like and What Changes Will They Bring?** 2019. University of Athens. Disponível em: <https://arxiv.org/pdf/1905.00402.pdf>. Acesso em: 22 mar. 2020.

STONE, John David. **Algorithms for Functional Programming**. Grinnel, EUA: Springer, 2018.

STRAWN, Jay. **Design Patterns by Tutorials: MVVM.** 2018. Disponível em: <https://www.raywenderlich.com/34-design-patterns-by-tutorials-mvvm>. Acesso em: 6 mar. 2020.

STROUSTRUP, Bjarne. **Programming: Principles and Practice Using C++**. 2. ed. Indianapolis, EUA: Pearson Education, 2014.

SWIFT. **Automatic Reference Counting.** Disponível em: <https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html>. Acesso em: 12 abr. 2020.

SWIFT. **Extensions.** Disponível em: <https://docs.swift.org/swift-book/LanguageGuide/Extensions.html>. Acesso em: 12 abr. 2020.

SWIFT. **Generics.** Disponível em: <https://docs.swift.org/swift-book/LanguageGuide/Generics.html>. Acesso em: 12 abr. 2020.

SWIFT. **Protocols.** Disponível em: <https://docs.swift.org/swift-book/LanguageGuide/Protocols.html>. Acesso em: 12 abr. 2020.

SWIFT. **Welcome to Swift.org.** Disponível em: <https://swift.org/>. Acesso em: 6 mar. 2020.

SWIFT, Sarin. **Memory Management in Swift: Heaps & Stacks**. 2019. Disponível em: <https://heartbeat.fritz.ai/memory-management-in-swift-heaps-stacks-baa755abe16a>. Acesso em: 28 set. 2020.

THAKKAR, Mohit. **Beginning Machine Learning In iOS**, [S.L.]: Apress, 2019.

TIOBE. **The Swift Programming Language**. Disponível em: https://www.tiobe.com/tiobe-index/swift/. Acesso em: 10 out. 2020.

TRIPP, Charles; HYDE, David; GROSSMAN-PONEMON, Benjamin. FRC: a high-performance concurrent parallel deferred reference counter for C++. **Acm Sigplan Notices**, [s.l.], v. 53, n. 5, p. 14-28, 7 dez. 2018. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/3299706.3210569.

UMOBI. **AlertFactory: Declarative alert creator**. 2019. Disponível em: <https://github.com/umobi/AlertFactory>. Acesso em: 04 mai. 2020.

VACCARO, Elisa. **Single Source of Truth…and Why it Matters**. Disponível em: https://medium.com/@elisavaccaro/single-source-of-truth-and-why-it-matters-a68e28b8c175. Acesso em: 30 ago. 2020.

WANG, Wallace. **Pro iPhone Development With Swift 5**. [S.L.]: Apress, 2019.

WATT, David A. **Programming Language Design Concepts.** Chichester, ING: John Wiley & Sons, Ltd, 2004.

WIRTH, Niklaus. **Algorithms + Data Strutures = Programs**. Englewood Cliffs, EUA: Prentice-hall, Inc., 1976.

ZAITSEV, O.O. **THE BENEFITS OF REACTIVE PROGRAMMING**. 2017. Disponível em: <https://er.nau.edu.ua/bitstream/NAU/27935/1/Zaitsev%20O.O.pdf>. Acesso em: 30 abr. 2020.

APÊNDICE A – CASOS DE TESTE

Identificador	CT01	
Nome	ContentView com texto “Olá Mundo”	
Sumário	Esse caso de teste deve avaliar o uso do protocolo UICView e da classe UICLabel.	
Pré-condição	Deve ser declarado uma ContentView que estende o protocolo UICView.	
Sequência Ordinária	**Passo**	**Ação**
	1	Declarar a variável de leitura *body.*
	2	Instanciar a classe UICContent para posicionar o conteúdo ao centro.
	3	Instanciar a classe UICLabel com o título “Olá Mundo”.
	4	Definir a cor do texto como laranja.
	5	Definir a fonte padrão do sistema e tamanho igual a 18 pt.
Resultado Esperado	Ao executar o teste, o aplicativo mostra o texto “Olá Mundo” centralizado, com cor do texto de cor laranja, fonte equivalente ao do sistema iOS e tamanho 18pt.	

Identificador	CT02	
Nome	ContentView com botão “Aperte”	
Sumário	Esse caso de teste deve avaliar o uso do protocolo UICView e da classe UICButton.	
Pré-condição	Deve ser declarado uma ContentView que estende o protocolo UICView.	
Sequência Ordinária	**Passo**	**Ação**
	1	Declarar a variável de leitura *body.*
	2	Instanciar a classe UICContent para centralizar o conteúdo.
	3	Instanciar a classe UICButton com o título “Aperte”.
	4	Definir a cor do texto como branca.
	5	Definir a cor de fundo como azul.
	6	Arredondar borda do botão para 5 pt.
	7	Adicionar o método onTouchInside(_:) definir um *callback* que mostre o alerta.

	8	Definir o *callback* do método para mostrar o alerta "Olá Mundo!" com a mensagem "Esse é o primeiro caso de teste usando a programação declarativa".
Resultado Esperado	Ao executar o teste, o aplicativo mostra o botão "Aperte" centralizado, com cor do texto igual a branco, cor de fundo azul e borda arredondada em 5 pt. Enquanto selecionado, o botão deve mudar a cor de fundo para azul claro. Então, o usuário deve apertar o botão e o aplicativo mostra o alerta.	

Identificador	CT003	
Nome	Mostrar título e descrição usando o objeto UICStack.	
Sumário	Utilizando a classe UICStack, pode ser mostrado várias *views* de maneira ordenada verticalmente ou horizontalmente.	
Pré-condição	Deve ser declarado uma ContentView que estende o protocolo UICView.	
Sequência Ordinária	**Passo**	**Ação**
	1	Declarar a variável de leitura *body.*
	2	Instanciar a classe UICContent para centralizar o conteúdo.
	3	Instanciar a classe UICStack com posicionamento vertical.
	4	Declarar o título UICLabel com o texto "Olá Mundo!".
	5	Declarar a descrição UICLabel com o texto "Esta interface foi programada de forma declarativa"
	6	Definir o espaçamento da UICStack como 5 pt
Resultado Esperado	O aplicativo deve mostrar o título "Olá Mundo" centralizado e ao topo. Também, deve ser mostrado a descrição centralizada e embaixo do título.	

Identificador	CT004	
Nome	Montar um formulário de login.	
Sumário	Utilizando a classe UICText para os campos e-mail e senha com um botão enviar.	
Pré-condição	Deve ser declarado uma ContentView que estende o protocolo UICView.	
Sequência Ordinária	**Passo**	**Ação**
	1	Declarar a variável de leitura *body.*
	2	Instanciar a classe UICContent para centralizar o conteúdo.
	3	Instanciar a classe UICStack com posicionamento vertical.

	4	Declarar o campo e-mail com a UICText
	5	Declarar o campo senha com a UICText, definindo o campo como seguro para ocultar a senha do usuário.
	6	Escutar as edições com o método onEditingDidEnd(_:) do campo e-mail (x) e da senha (y) para armazenar o dado na ContentView.
	7	Adicionar o botão "Entrar" com o método onTouchInside(_:)
	8	Ao selecionar o botão, o aplicativo mostra um alerta com o título "Dados Inseridos" e com a mensagem "E-mail: x; Senha: y".
Resultado Esperado	Ao preencher as informações e selecionar o botão, o aplicativo deve mostrar um alerta com as informações inseridas.	

Identificador	CT005	
Nome	Listar 100 *views* com o protocolo UICViewRepresentable usando UICScroll com cores diferentes.	
Sumário	Utilizando a classe UICScroll com direção vertical, deve ser mostrado um total de 100 views implementadas com o protocolo UICViewRepresentable usando cores de fundo diferentes.	
Pré-condição	Deve ser declarado uma ContentView que estende o protocolo UICView.	
Sequência Ordinária	**Passo**	**Ação**
	1	Declarar a variável de leitura *body.*
	2	Instanciar a classe UICScroll com direção vertical.
	3	Instanciar a classe UICStack com posicionamento vertical.
	4	Implementar a StaticView com o protocolo UICViewRepresentable
	5	Adicionar 100 StaticView com cor de fundo únicas e para cada uma configurar altura igual a 100 pt.
Resultado Esperado	O aplicativo deve mostrar uma tela que deslize na vertical com 100 views configuradas com cores distintas.	

Identificador	CT006
Nome	Aplicativo com duas telas para preenchimento e conferência dos dados.
Sumário	Usando a classe UICNavigation implementar a primeira tela com um formulário simples com dados de usuário. Ao pressionar o botão "Conferir", o aplicativo deve apresentar a segunda tela com os mesmos dados inseridos, porém utilizando a classe UICLabel.
Pré-condição	Deve ser declarado uma ContentView que estende o protocolo UICView.

Sequência Ordinária	Passo	Ação
	1	Declarar a variável de leitura *body.*
	2	Instanciar a classe UICNavigation para mostrar a FormView.
	3	Implementar a FormView, que estende o protocolo UICView, utilizando a UICText para os campos "Nome", "Sobrenome", "CPF" e "RG".
	4	Adicionar na classe FormView o botão "Conferir" que, ao pressioná-lo, deve ser apresentado CheckView com os dados inseridos.
	5	Definir o título da barra de navegação da FormView usando o método navigationBar(title:) como "Formulário".
	6	Implementar a CheckView utilizar a UICLabel para mostrar o texto para os campos "Nome", "Sobrenome", "CPF" e "RG".
	7	Definir como título da CheckView como "Conferência".
Resultado Esperado	O aplicativo deve inicializar mostrando o formulário com os campos editáveis "Nome", "Sobrenome", "CPF" e "RG". Ao pressionar o botão "Conferir", deve ser apresentado uma tela com as mesmas informações inseridas na tela anterior.	

Identificador	CT007	
Nome	Aplicativo para listagem de contatos com recurso para adicionar e remover.	
Sumário	Utilizando a UICNavigation, a aplicação deve conter três telas. A primeira deve listar os contatos, a segunda permitir cadastrar novos contatos e a terceira visualizar o contato.	
Pré-condição	Deve ser declarado as classes ContactsView, NewContactView, ContactView que estendem o protocolo UICView.	
Sequência Ordinária	**Passo**	**Ação**
	1	Declarar a variável de leitura *body.*
	2	Instanciar a classe UICNavigation para mostrar a ContactsView.
	3	Implementar a ContactsView, estendendo o protocolo UICView, utilizando a UICList para listar os contatos.
	4	Inserir o botão mais para adicionar novos contatos na barra de navegação, ao lado direito.
	5	Definir o título da navegação igual a "Contatos".
	6	Implementar a classe NewContactView com que permita adicionar ao contato uma imagem, nome, sobrenome, email e telefone.
	7	Definir título da NewContactView igual a "Novo Contato".
	8	Implementar a classe ContactView para mostrar as informações do contato.
	9	Definir o títutlo da ContactView igual ao nome do contato.
Resultado Esperado	O aplicativo deve inicializar mostrando uma lista de contatos vazia com o botão adicionar na parte superior direita. Ao adicionar um novo contato, a lista deve ser atualizada. Selecionando um contato na lista, é mostrado as informações do contato. Na lista, ao arrastar uma célula para a esquerda, é possível deletar o contato.	

Identificador	CT008
Nome	Lisagem de *views* utilizando a UICFlow.

<table>
<tr><td>Sumário</td><td colspan="2">Com a classe UICFlow, deve ser listado 500 views formando um degrade com cor inicial preto e cor final branco, incrementando na escala de cinza, distribuídas em 2 colunas.</td></tr>
<tr><td>Pré-condição</td><td colspan="2">Deve ser declarado a classe ContentView que estende o protocolo UICView.</td></tr>
<tr><td rowspan="7">Sequência Ordinária</td><td>Passo</td><td>Ação</td></tr>
<tr><td>1</td><td>Declarar a variável de leitura body.</td></tr>
<tr><td>2</td><td>Instanciar a classe UICFlow.</td></tr>
<tr><td>3</td><td>Adicionar a UICForEach com 500 elementos.</td></tr>
<tr><td>4</td><td>Declarar a UICRow com a UICSpacer com cor de fundo entre 0 e 1, onde 0 significa preto e 1 significa branco.</td></tr>
<tr><td>5</td><td>Adicionar a UICFlow o método layoutMaker(_:)</td></tr>
<tr><td>6</td><td>Incluir ao layout das células a UICCollectionLayoutItem com largura igual a metade da UICollectionView.</td></tr>
<tr><td>Resultado Esperado</td><td colspan="2">O aplicativo deve mostrar 500 views formando um degrade na escala de cinza que devem ser distribuídas formando duas colunas orientadas na vertical.</td></tr>
</table>

ANEXO A – TERMO DE AUTORIZAÇÃO DE PUBLICAÇÃO DE PRODUÇÃO ACADÊMICA

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
GABINETE DO REITOR

Av. Universitária, 1069 ● Setor Universitário
Caixa Postal 86 ● CEP 74605-010
Goiânia ● Goiás ● Brasil
Fone: (62) 3946.1000
www.pucgoias.edu.br ● reitoria@pucgoias.edu.br

RESOLUÇÃO n° 038/2020 – CEPE

ANEXO I

APÊNDICE ao TCC

Termo de autorização de publicação de produção acadêmica

O(A) estudante Brenno Giovanini de Moura do Curso de Engenharia da Computação, matrícula 2016.1.0033.0504-7, telefone: (62) 92000-8483 e-mail brennobemoura@gmail.com, na qualidade de titular dos direitos autorais, em consonância com a Lei nº 9.610/98 (Lei dos Direitos do autor), autoriza a Pontifícia Universidade Católica de Goiás (PUC Goiás) a disponibilizar o Trabalho de Conclusão de Curso intitulado Framework para desenvolvimento de aplicativos IOS com interface de usuário programada de forma declarativa, gratuitamente, sem ressarcimento dos direitos autorais, por 5 (cinco) anos, conforme permissões do documento, em meio eletrônico, na rede mundial de computadores, no formato especificado (Texto (PDF); Imagem (GIF ou JPEG); Som (WAVE, MPEG, AIFF, SND); Vídeo (MPEG, MWV, AVI, QT); outros, específicos da área; para fins de leitura e/ou impressão pela internet, a título de divulgação da produção científica gerada nos cursos de graduação da PUC Goiás.

Goiânia, 08 de dezembro de 2020.

Assinatura do(s) autor(es): Brenno de Moura

Nome completo do autor: Brenno Giovanini de Moura

Assinatura do professor-orientador: Ludmilla R. P. dos Santos

Nome completo do professor-orientador: Ludmilla Reis Pinheiro dos Santos