Um estudo comparativo entre padrões arquiteturais para o desenvolvimento de aplicativos para a plataforma iOS

Ícaro Lima Magalhães

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2018

Ícaro Lima Magalhães

Um estudo comparativo de padrões arquiteturais para o desenvolvimento de aplicativos para a plataforma iOS

Monografia apresentada ao curso Ciência da Computação do Centro de Informática, da Universidade Federal da Paraíba, como requisito para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Raoni Kulesza

Novembro de 2018

Catalogação na publicação
Seção de Catalogação e Classificação

M188e Magalhaes, Icaro Lima.
Um estudo comparativo entre padrões arquiteturais para o desenvolvimento de aplicativos para a plataforma iOS / Icaro Lima Magalhaes. - João Pessoa, 2018.
52 f. : il.

Orientação: Raoni Kulesza.
TCC (Especialização) - UFPB/CI.

1. Dispositivos móveis. 2. Plataformas de desenvolvimento. 3. Arquitetura de software. I. Kulesza, Raoni. II. Título.

UFPB/BC

CENTRO DE INFORMÁTICA

UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Ciência da Computação intitulado **Um estudo comparativo entre padrões arquiteturais para o desenvolvimento de aplicativos para a plataforma iOS** de autoria de Ícaro Lima Magalhães, aprovado pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Raoni Kulesza
Universidade Federal da Paraíba

Prof. Dr. Derzu Omaia
Universidade Federal da Paraíba

Prof. Dr. Lincoln David Nery e Silva
Universidade Federal da Paraíba

João Pessoa, 12 de Novembro de 2018

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

DEDICATÓRIA

Dedico este trabalho primeiramente aos meu pais, Franco e Jeanne. Que os frutos dessa conquista diminuam a distância e a saudade que tenho de vocês. Aos meus irmãos sempre me apoiaram nessa longa jornada. À minha amada Janiele, um dos pilares da minha vida. Dedico aos meus cachorros, que conseguem me fazer sorrir mesmo nos dias mais tristes. Aos bons amigos e colegas de trabalho que tive a oportunidade de conhecer graças às incontáveis oportunidades que a universidade me deu, e em especial, aos professores, que formaram não só mais um cientista, mas um ser humano apto a construir um mundo melhor.

AGRADECIMENTOS

Agradeço em primeiro lugar e de uma maneira especial à professora Tatiana Aires Tavares. Talvez ela não saiba, mas foi minha segunda mãe durante bastante tempo. Tatiana me ensinou muitas lições importantes que levarei para a vida toda.

Agradeço ao professor Tiago Maritan por ter me dado a oportunidade de participar do núcleo LAViD, lugar onde aprendi que com esforço, responsabilidade e dedicação é possível sim transformar o mundo num lugar melhor.

Ao professor Lincoln David Nery e Silva por ter acompanhado minha equipe, Mateus Pires e Marcello Marques, em uma das jornadas mais divertidas e recompensadoras que tive a oportunidade de participar durante estes longos anos de curso.

Agradeço ao professor Raoni Kulesza por ter me ensinado uma das lições de maior valor que carregarei por toda a minha vida profissional, assim como pelo seu apoio e suporte na escrita de meu trabalho de conclusão.

Aos professores Antônio Gomes, Christian Azambuja Pagot e Lucídio dos Anjos Formiga Cabral por suas personalidades inspiradoras, assim como a todos os demais professores que contribuíram, mesmo que indiretamente, com minha formação.

É impossível citar todos os colegas de curso ou de trabalho que tive a oportunidade de conhecer e que me impactaram positivamente de alguma maneira. À minha banca examinadora, à coordenação, ao pessoal da limpeza e à instituição como um todo, deixo aqui o meu muito obrigado.

RESUMO

Com a evolução dos smartphones, aplicativos para dispositivos móveis têm se tornado poderosas ferramentas com interações e serviços cada vez mais complexos. Com esse avanço, o aspecto de desenvolvimento foi impactado profundamente nessa categoria de software, trazendo requisitos cada vez mais exigentes para atender as necessidades dos usuários. O estado da arte no desenvolvimento de aplicativos móveis apresenta uma vasta quantidade de documentação e frameworks que oferecem diversas APIs para suportar o desenvolvimento de aplicativos simples até os mais complexos. Toda essa complexidade vêm aliada de um custo de desenvolvimento, evolução e manutenção. Mudanças na base de código fonte de um aplicativo móvel podem se fazer necessárias por uma infinidade de motivos. Nesse contexto, uma aplicação móvel projetada a partir de uma arquitetura pré-existente apoiada por padrões de projeto pode apresentar diversas vantagens em aspectos cruciais como reuso, testabilidade, modularização e baixo acoplamento, permitindo mudanças durante o ciclo de vida da aplicação de modo que os impactos indesejáveis são diminuídos. Entretanto, apesar da popularidade dessas soluções, poucos trabalhos avaliam esses ganhos. Este trabalho compara dois padrões arquiteturais no desenvolvimento de aplicativos móveis para dispositivos iOS, discutindo quais os reais impactos de uma mudança arquitetural nos estágios iniciais de desenvolvimento.

Palavras-chave: Dispositivos móveis, plataformas de desenvolvimento, arquitetura de software.

ABSTRACT

The smartphone evolution has brought increasingly powerful, complex and interactive applications for mobile devices. In this category, software developers face new challenges to meet industry's ever-growing criteria and user's demands. The State of the Art of applications development has a wide framework of documents which offer several APIs to help developers from the basic to the most sophisticated levels. This framework comes with a high cost of maintenance due to constantly changing source-codes, which in turn might be required for a large range of reasons. In this context, a mobile application designed from a pre-existing architecture and supported by strong project patterns may benefit in crucial aspect such as reusage, testability, modularization and low coupling, allowing changes during the application's life cycle and limiting undesirable impacts. However, as popular as these solutions may be, only a few academic works have addressed its real gains. Here we compare two architectural patterns for applications development in iOS devices and evaluate the impacts of an architectural change in the early stage of a developing process.

Key-words: Mobile devices, development platforms, software architecture.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

API	Application Programming Interface
APP	Aplicativo
IDE	Integrated Development Environment
IOS	iPhone Operating System
MVC	Model, View, Controller
MVVM	Model, View, ViewModel
MVP	Model, View, Presenter
UFPB	Universidade Federal da Paraíba
UI	Interface com o Usuário
USP	Universidade de São Paulo
VIP	View, Interactor, Presenter
VIPER	View, Interactor, Presenter, Entity, Routing

SUMÁRIO

1 INTRODUÇÃO

Com a evolução da indústria de aplicações móveis, projetos e produtos cada vez mais complexos apresentam dificuldades recorrentes que adicionam complexidade aos processos de desenvolvimento. Adição de novas funcionalidades e aplicativos com cada vez mais opções de interação assim como constantes mudanças nos requisitos do sistema criam demandas de ajustes e remodelagem constantes na estrutura do código. Deste modo, a necessidade de ambientes flexíveis, reuso e de facilidade de manutenção se torna um ponto crítico do desenvolvimento de aplicativos, tornando a arquitetura de um software um fator determinante para seu sucesso no mercado. Maior frequência no lançamento de ajustes, melhor acurácia nas estimativas, mudanças em menos tempo e sistemas que se recuperam melhor de impactos estruturais são alguns dos benefícios ao utilizar uma arquitetura de software adequada.

1.1 Definição do problema

O aumento do número de aplicativos aliado à demanda crescente e às urgências de usuários cada vez mais exigentes fez com que desenvolvedores colocassem à prova a robustez e eficiência do modelo arquitetural Cocoa MVC[9], uma variação da arquitetura MVC, acrônimo de Model View Controller, para dispositivos iOS. Esta arquitetura ganhou popularidade no desenvolvimento de aplicações iOS ao ser adotado como arquitetura principal da plataforma, tendo seu uso encorajado pela Apple. Este modelo é visto como um bom projeto central para aplicações iOS, oferecendo código reusável, interfaces bem definidas e cooperando muito bem com as tecnologias Cocoa existentes[21]. Entretanto, o emprego do MVC traz um acoplamento bastante estrito, que impacta negativamente no aspecto de reuso das aplicações.

Interações cada vez mais complexas e evolução nas APIs e no framework iOS como um todo expuseram o problema dos *ViewControllers* massivos no processo desenvolvimento, fazendo com que os desenvolvedores buscassem alternativas ao padrão MVC na comunidade ou em outros padrões arquiteturais já consagrados[26]. Desenvolvedores passaram a buscar arquiteturas alternativas mais flexíveis e adaptáveis nos estágios iniciais de desenvolvimento se tornou cada vez mais importante. Este ponto é crucial no ciclo de vida e de desenvolvimento de um produto, pois impacta diretamente na velocidade e no custo de desenvolvimento, visto que a maior parte do

orçamento de um projeto sério é representada pela fase pós release, ou seja, sua evolução e manutenção[18].

Uma das maiores dificuldades encontradas por desenvolvedores está em criar código que evolua bem e resista ao teste do tempo à medida que a complexidade dos seus sistemas aumenta. Ao escolher um modelo arquitetural adequado ao seu problema, o projetista de software visa criar uma base de código que não seja hostil ao desenvolvedor, de modo que manutenções e a adição de novas funcionalidades se torne um trabalho mais fácil. Segregar componentes de acordo com suas funcionalidades é não só um desejo, mas um processo vital à saúde do software.

1.2 Premissas e hipóteses

O desenvolvimento de aplicativos iOS introduz novos problemas que resistem ao modelo arquitetural inicialmente proposto pela Apple dez anos atrás. De um ponto de vista arquitetural, MVC consegue fazer um bom trabalho quando estamos tratando de sistemas simples, porém à medida que os projetos deste tipo crescem, tarefas que deveriam ser naturais tornam-se gargalos no processo de desenvolvimento. Projetos que utilizam esta arquitetura tendem a apresentar forte acoplamento, classes massivas e pouco testáveis. Tendo isso em vista, a comunidade de desenvolvedores busca novos meios de resolver tais problemas. Percebendo padrões e problemas recorrentes, desenvolvedores naturalmente constroem um novo ferramental para abordar diferentes tipos de problema. Encontrar arquiteturas alternativas que se encaixam bem ao processo de desenvolvimento não é uma tarefa fácil. As alternativas devem ser postas à prova e comparadas diretamente com o já consagrado padrão Cocoa MVC.

1.2.1 Objetivo geral

O principal objetivo deste trabalho é realizar um estudo comparativo entre modelos arquiteturais para o desenvolvimento de aplicativos iOS.

1.2.2 Objetivos específicos

No que diz respeito aos objetivos específicos, pretende-se:

1. Apresentar uma visão geral sobre aplicativos móveis para dispositivos iOS, assim como sobre os modelos arquiteturais mais empregados no seu desenvolvimento.

2. Implementar diferentes versões de um aplicativo a partir dos mesmos requisitos e casos de uso, partindo de diferentes arquiteturas de software, descrevendo todo o processo e as tecnologias utilizadas no desenvolvimento da solução.
3. Descrever ferramentas utilizadas e todo o processo de obtenção e análise das métricas coletadas a partir do código fonte do aplicativo desenvolvido.
4. Avaliar os impactos arquiteturais causados pela substituição do modelo arquitetural no aplicativo desenvolvido.

1.3 Estrutura da monografia

No capítulo 2 é apresentado todo o alicerce que objetiva familiarizar o leitor com diversos aspectos do desenvolvimento de aplicações móveis para dispositivos iOS. São discutidos os principais modelos arquiteturais aplicados no contexto apresentado, seguindo então para o Capítulo 3, que apresenta toda a metodologia e os processos que foram executados durante a pesquisa. Todos os resultados são sumarizados e apresentados no Capítulo 4, seguido as conclusões no Capítulo 5.

2 CONCEITOS GERAIS

Em 2007 Steve Jobs apresentou ao público o que mais tarde viera a se tornar o projeto mais ambicioso da gigante Apple, o iPhone, trazendo um sistema operacional inovador com funcionalidades jamais vistas em qualquer outro dispositivo móvel. O iOS, *iPhone Operating System,* completa dez anos de existência em 2018[1]. Na sua décima segunda versão, o sistema entrega a seus usuários mais de dois milhões de aplicativos por meio da *App Store*, principal serviço de distribuição e download de aplicativos no ecossistema Apple.

O processo natural de evolução da plataforma introduziu ao longo dos anos diversas funcionalidades e APIs, dando mais poder aos desenvolvedores. Novos níveis de complexidade surgiram, tais como notificações ricas, métodos de pagamento, novos gestos e maneiras de interagir com aplicativos que à pouco não passavam simples interfaces com finalidades básicas. Com isso, usuários cada vez mais exigentes demandam dos novos aplicativos eficiência e qualidade em todos os seus aspectos. Pequenas mudanças que antes não faziam tanta diferença agora são fatores decisivos no ciclo de vida dos aplicativos do mundo moderno, que passaram de simples interfaces gráficas para interfaces entre o usuário e serviços em escala global.

O aumento da complexidade nas interações, interfaces e operações oferecidas pelos aplicativos trouxeram implicações fortíssimas no seu ciclo de desenvolvimento e manutenção. Para estender a discussão é preciso entender como um aplicativo iOS funciona, quais são seus principais componentes, seu ciclo de vida e como a tecnologia tem amadurecido ao longo dos anos.

A discussão sobre a anatomia de um aplicativo iOS, assim como seus modelos arquiteturais mais conhecidos, abre espaço para um melhor entendimento sobre quais são os reais impactos da evolução dos aplicativos nos seus estágios de concepção, desenvolvimento e manutenção. Ao entender os principais modelos arquiteturais aplicados no desenvolvimento de aplicativos é possível entender qual o verdadeiro impacto de uma mudança arquitetural em um projeto, e como a comunidade se comporta em relação ao estágio atual do desenvolvimento iOS.

2.1 A Anatomia de Um Aplicativo iOS

Para oferecer interações ricas e responsivas aos seus usuários, a Apple disponibiliza para seus desenvolvedores *Frameworks* que tornam possível o desenvolvimento de aplicativos

[1] Apple: The App Store turns 10. Disponível em: https://www.apple.com/newsroom/2018/07/app-store-turns-10/

complexos, fluidos e robustos. Com o amadurecimento da plataforma, o desenvolvimento passou a se concentrar principalmente no uso de frameworks como *UIKit, Foundation* e *CoreData*. Estes possuem um conjunto de operações que podem ser acessadas e por meio de código nativo. Comandos são traduzidos em comportamentos que o desenvolvedor deseja espelhar em seu aplicativo a partir destes frameworks.

Como consequência da sua evolução, o desenvolvimento de aplicativos iOS sofreu uma grande mudança em 2014. Até então só poderiam ser escritos por meio da linguagem de programação Objective-C. Ao ouvir a comunidade, a Apple introduziu a linguagem Swift foi introduzida no desenvolvimento iOS. Swift trouxe melhorias muito bem vindas tais como uma tipagem mais forte e a adição de extensões[2], permitindo métodos mais curtos, mais seguros e de menor complexidade. Estas diferenças impactaram profundamente no desenvolvimento de aplicativos no ecossistema iOS, melhorando muitos aspectos da sua linguagem antiga, Objective-C[13].

2.1.1 UI, Layouts e Eventos

A UI em aplicativos iOS é definida principalmente por *Storyboards* e *ViewControllers*. Cada aplicativo é composto de pelo menos um *ViewController* que define comportamentos e interações entre a interface com o usuário e os dados da aplicação. Toda a UI tem como elementos básicos as Views, que podem ser especializadas de modo a exibir, por exemplo, imagens, textos e componentes de alta complexidade. Views respondem a eventos e podem ser coordenadas por manipulações programáticas tais como animações, transformações e transições.

O conceito de *Delegates* permite que um objeto possa agir em razão de outro. Esta coordenação é dada por meio de troca de mensagens e pode definir como componentes devem reagir a determinados eventos. Tomando como exemplo uma tela, isto é, uma *ViewController*, aberta para extensão de modo que ao definir uma implementação do componente *TableViewDelegate* torna-se responsável por responder aos eventos emitidos por uma *TableView*.

2.1.2 O ciclo de vida de um aplicativo iOS

O ponto de partida de um aplicativo iOS é denominado *UIApplicationDelegate*. Este componente é de implementação obrigatória é o responsável por gerenciar grande parte dos

[2] Extensions - The Swift Programming Language, 2018, Disponível em: https://docs.swift.org/swift-book/LanguageGuide/Extensions.html. Acesso em: 20 ago. 2018.

comportamentos de alto nível de uma aplicação. Os estados do *UIKit* que dizem respeito ao ciclo de vida podem ser resumidos em cinco estados principais. Estes eventos notificam o delegate que por sua vez deve responder de forma apropriada a situações onde o aplicativo é inicializado, suspenso e até mesmo finalizado.

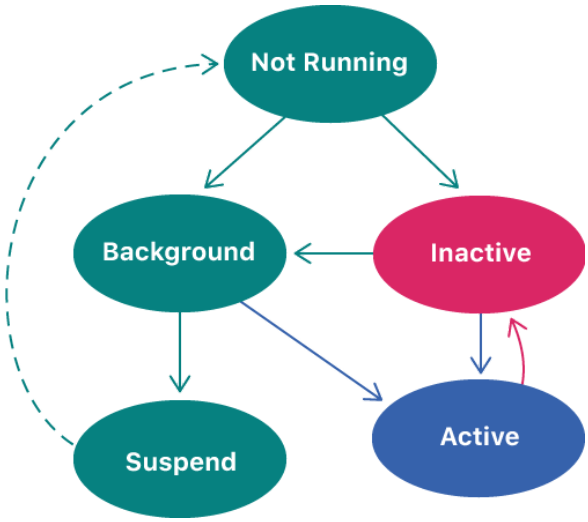

Figura 1: Ciclo de Vida de Aplicativos iOS [4]

Em alguns casos, tratar o ciclo da maneira correta pode introduzir bastante complexidade ao projeto. Um aplicativo, por exemplo, não é inicializado no estado de execução. Ao invés disso, ele é inicializado em estado inativo, tornando-se ativo somente quando é lançado pelo sistema operacional. Importante notar que ele pode ser, inclusive, enviado imediatamente para background. Um aplicativo é dito suspenso se não está sendo apresentado na tela, sendo assim candidato a voltar ao estado de execução. O ciclo de vida também pode ser afetado por alertas que podem ser, por exemplo, relativas ao consumo de memória do processo. Complicações como estas normalmente são alvo de negligência por parte de desenvolvedores menos experientes. Estes tendem a não implementar tratamentos para casos específicos causados por comportamentos relativos ao ciclo de vida do aplicativo.

Conhecer o ciclo de vida e os respectivos eventos que são disparados em determinadas situações permite que o desenvolvedor configure, inicialize, suspenda ou finalize objetos e serviços com mais segurança, fazendo melhor gerenciamento de recursos e tornando o aplicativo mais seguro e responsivo[3].

2.2 Padrões Arquiteturais

No ecossistema iOS, os padrões normalmente são divididos em duas categorias: Os padrões MV*, onde se encontra o MVC, e os padrões baseados na Arquitetura Limpa, tais como VIP, VIPER e *CleanSwift*. Mesmo compartilhando de intenções tais como a separação de responsabilidades em camadas, melhorias na testabilidade e modularização, o uso de diferentes modelos aplicados ao desenvolvimento iOS pode trazer implicações que vão muito além da base de código de um projeto. Apesar do grande número de padrões arquiteturais citados e do número ainda maior de padrões conhecidos e difundidos na comunidade, são discutidos os mais populares e aplicáveis ao ecossistema da Apple, analisando-os e levando sempre em consideração sua aplicação e impactos no desenvolvimento de aplicações iOS.

2.3 Padrões Arquiteturais MV*

Discutir os papéis dos padrões da classe MV* é um ponto crucial para um bom entendimento a respeito da evolução das interfaces de usuário. O termo MV* representa uma família de modelos arquiteturais que dão suporte à separação de responsabilidades na base de código de um projeto. As duas primeiras letras da sigla MV* representam Model e View, respectivamente. O asterisco, por sua vez, representa um terceiro componente que pode variar. No caso do modelo arquitetural MVC, por exemplo, o asterisco está para Controller. Dentre as variações que pertencem à classe, além do MVC existe um espectro enorme de padrões arquiteturais, variando de padrões amplamente conhecidos, tais como *Model-View-Presenter (MVP)* e *Model-View-ViewModel (MVVM)* até implementações com caráter mais experimental, tais como *ModelAdapter - ViewBinder* e *ModelView - ViewModel - Coordinator*[23].

2.4 MVC e Cocoa MVC

Proposto na década de 70, o padrão arquitetural *Model-View-Controller (MVC)* emergiu como um dos padrões de design mais influentes para aplicações com interfaces gráficas complexas[1]. O MVC tradicional utiliza o conceito de *Controllers*. Estes são os pontos de entrada da aplicação e recebem sinais enviados pela interface com o usuário. Deste modo, o *Controller* possui a habilidade de processar entradas do usuário e enviar comandos para que Models sejam atualizados com base em seus estados, podendo implicar em mudanças nas Views à medida que

mudanças são observadas. Em suma, *Controllers*, que são mediadores que em geral coordenam o modo que modelos reagem a interações do usuário com as views.

2.4.1 MVC Tradicional

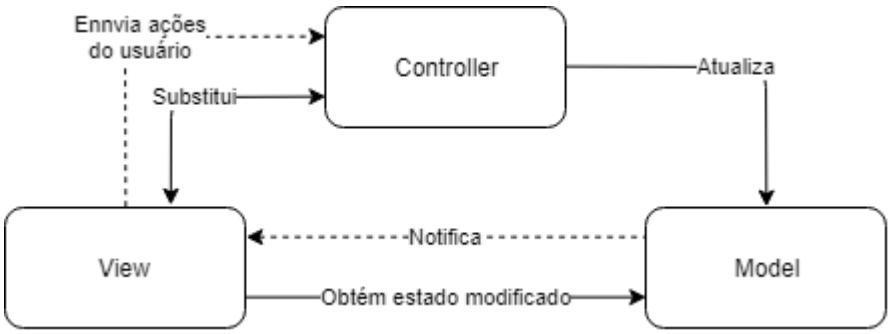

Figura 2: Padrão MVC Tradicional

Neste padrão, *Models*, não devem depender de *Controllers* ou *Views*. Tradicionalmente, o padrão MVC utiliza views que não possuem estado. Ao invés disso, são renderizadas uma vez que seus modelos são alterados. Discutir o padrão MVC tradicional é especialmente importante para um bom entendimento de que, na prática, não representa apenas vantagens para desenvolver aplicativos iOS, pois estas três entidades são altamente acopladas. Importante notar que muitas outras operações ocorrem durante o ciclo de vida de uma aplicação. Operações relacionadas a persistência ou networking não são vistas como preocupações do padrão MVC, que foi concebido para resolver problemas de interações via interface do usuário.

2.4.2 Cocoa MVC

Na intenção de adaptar uma solução já consagrada para o framework iOS, a Apple propõe uma variação do padrão MVC onde o fluxo de ações foi alterado de modo os eventos são emitidos diretamente pela View. Pode-se usar como exemplo o toque na tela do dispositivo móvel, de modo que uma View seja encarregada de notificar um observer. Isso coloca o componente Controller como um mediador que observa eventos da View por meio de delegates, callbacks ou outros mecanismos. Os eventos são então convertidos em comandos e passados para o Model, que por sua vez é atualizado, notificando o Controller que finalmente sinaliza a View para que seja renderizada refletindo o novo estado da UI. Deste modo, em contraste à abordagem MVC tradicional, View e Model não se conhecem.

Como Views não referenciam Models diretamente, a camada de formatação e apresentação costuma ser delegada ao Controller, assim como operações de networking, gerenciamento de eventos, etc. Esta organização acaba se tornando um problema, pois ainda que o desenvolvedor tenha a opção de quebrar sua interface no que são conhecidas como Subviews, o que é visto na prática são os famosos ViewControllers massivos.

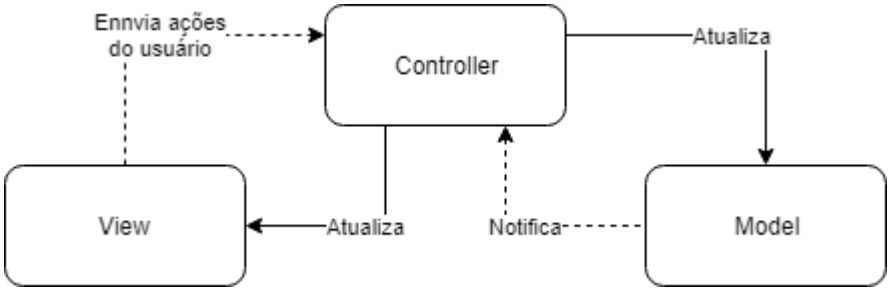

Figura 3: Padrão Apple MVC - Expectativa

Ainda que seja proposto como uma melhoria ao ser comparado com o padrão MVC original, desenvolvedores continuam o modelo adotado pela Apple recebe críticas constantes de desenvolvedores mais versados em modelos arquiteturais. Isso se dá pelo fato do padrão Cocoa MVC encorajar a escrita de *UIViewControllers*, controllers altamente acoplados à camada de visão. Também são atribuídas muitas responsabilidades ao ViewController, tornando-o *Delegate* e *Data Source* de muitos componentes ou entidades presentes no projeto. Isso ocorre com muita frequência, fazendo com que ViewControllers cresçam ainda mais, e por isso a discussão a respeito de situações tais como "*View controller offloading*" é algo muito comum entre desenvolvedores iOS que buscam criar *ViewControllers* mais magros[5].

O problema dos ViewControllers massivos se torna ainda mais evidente ao tentarmos desenvolver testes unitários. Torna-se difícil testar responsabilidades de componentes quando estes possuem muitas responsabilidades, o que é o caso do *ViewController*, demandando ainda mais disciplina por parte do desenvolvedor, que deve escrever testes não só para validar modelos, como também para testar sua responsividade e fidelidade visual em diferentes tipos de tela.

Todas estas características tornam este padrão um dos mais simples e utilizados, que exigem a menor quantidade de código escrito dentre os outros padrões que são discutidos no trabalho e adotados pela comunidade. Como custo desta facilidade, temos impactos negativos na manutenção e testabilidade do código, tornando o desenvolvimento propenso a problemas de acoplamento,

separação de responsabilidades e testabilidade. Ainda assim, é a melhor opção para projetos pequenos que não exigem arquiteturas mais elaboradas e que não têm a intenção de se preocupar com um processo de manutenção mais rigoroso.

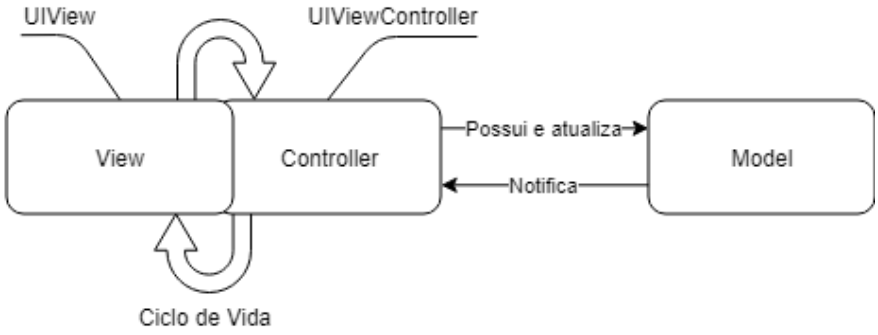

Figura 4: Padrão MVC Apple - Realidade

Neste padrão, a View é o ponto de interação entre usuários e a aplicação, ficando responsável por exibir os dados armazenados em modelos e permitir edição destes dados. Como intermediário, o Controller é o canal através do padrão em que mudanças na camada de Visão ou Modelos são aprendidas pelo outro. A interação do usuário feita no View é interpretada pelo Controller, que notifica o estado alterado nos dados para a camada Model. Qualquer alteração que ocorra no objeto Modelo é registrada pelo controlador por meio de implementações do padrão observer[10].

Ao observar as desvantagens dos ViewControllers imensos introduzidos por este padrão somos levados a questionar o motivo da popularidade do padrão MVC no desenvolvimento de aplicações iOS. A própria Apple encoraja fortemente seu uso. Não é difícil encontrar razões que levam o desenvolvedor a optar por usá-lo como primeira alternativa. A herança de projetos que usam MVC torna mais fácil de colocar exemplos de código encontrados em fóruns e na própria documentação oficial do framework em produção. Prazos curtos impostos pela indústria prezando por componentes com boa interação e qualidades gráficas também podem representar um motivo relevante, ainda que por baixo sua parte operacional seja um completo caos.

Uma vez que o desenvolvedor consegue implementar a funcionalidade desejada, seu tempo e orçamento curtos o forçam a seguir adiante e talvez nunca mais voltar a tocar no trecho de código recém implementado, a não ser que seja forçado, e provavelmente será, pois os temidos "View Controllers Massivos" acoplam uma imensa porção do código, fazendo com que mudanças simples

se propaguem em lugares indesejados. Apesar de todos os problemas que o MVC insiste em introduzir em projetos de larga escala, continua sendo a arquitetura prevalente em projetos iOS.

2.5 MVVM

A concepção da arquitetura MVVM - Model-View-ViewModel - apresenta a proposta de diminuir o código excessivo de apresentação das Views, fazendo com que o componente ViewModel se encarregue pela lógica de apresentação[11]. Isso faz de MVVM uma das alternativas mais populares que surgem na intenção de combater os problemas apresentados pelo padrão Cocoa MVC. Nesta variação MV*, é introduzido o conceito de ViewModel, um mediador entre Views e Models que substitui os controladores apresentados no padrão MVC. Neste caso, é preciso considerar Views como entidades que representam UIViewControllers completas, pois estas passam a refletir uma representação dos dados oferecida por ViewModels. Os dados, por sua vez, são representações das Views com base em seus estados.

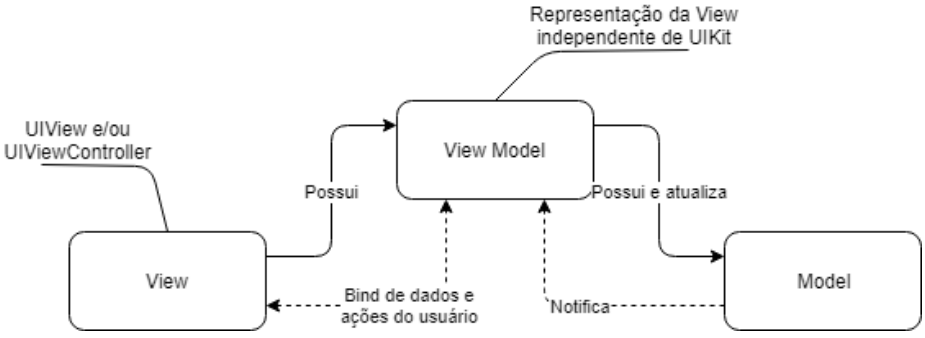

Figura 5: Padrão MVVM

Grande parte da popularidade do padrão MVVM se dá à sua natureza reativa, abrindo as portas para frameworks consagrados no desenvolvimento iOS, tais como ReactiveCocoa[3] ou RxSwift[4]. Daí surge uma necessidade indesejada de amarrar a aplicação a dependências de terceiros, além da necessidade de educar o desenvolvedor em um novo paradigma, a programação orientada a sinais. Isso também requer maior disciplina por parte do desenvolvedor, além de introduzir complexidade em operações de debugging, etc.

Dentre suas vantagens estão uma melhor distribuição, que apesar de depender de bindings, desacopla Modelos das Views, tornando ViewControllers menos massivos e mais testáveis sem a

[3] Extensões reativas para o framework Cocoa.
[4] Implementação do padrão ReactiveX para Swift

necessidade de adição de muito código. Em termos de tamanho e complexidade se assemelha ao modelo MVP, porém sem a complexidade de encaminhar eventos a um componente de apresentação, já que o MVVM representa essas operações por meio de bindings e observables. Em algumas situações MVVM pode ser visto como uma forma evoluída do Cocoa MVC, onde os componentes principais são mantidos e a lógica do domínio é delegada para uma nova mecânica orientada a eventos e sinais. Desde que existam bindings entre uma View e um ViewModel, ambos são atualizados de acordo. A adoção desta arquitetura tende a resultar em ViewControllers mais magros.

2.6 Arquitetura Limpa

Apesar de todas as arquiteturas apresentadas neste trabalho apresentarem variações em seus detalhes, seus objetivos tendem a se alinhar: Dividir o software em camadas bem definidas. Devem ser testáveis e o mais independente possível de qualquer agente externo, tais como Frameworks e bibliotecas. A partir de uma regra de dependência a arquitetura limpa prega que componentes do código fonte só devem apontar para dentro, isto é, camadas mais externas devem apontar apenas para camadas mais internas. Isso pode ser visto na figura que apresenta um diagrama circular onde tais princípios são respeitados.

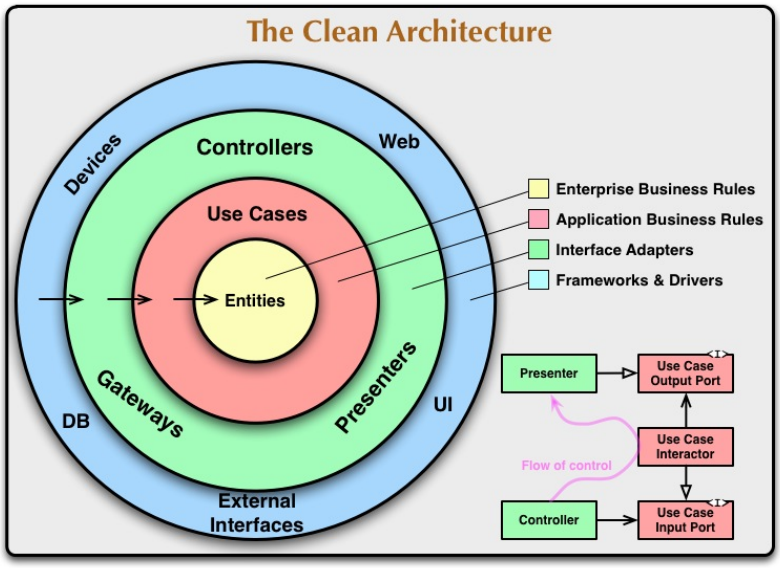

Figura 6: Arquitetura Limpa[5]

[5] Diagrama disponível em: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Na prática, isto significa que círculos exteriores devem conter mecânicas de alto nível, e camadas mais internas não devem depender destas de modo algum. Enquanto a camada de Entidades contém regras de negócio, na camada mais externa (Frameworks e Drivers) são encontradas ferramentas como bancos de dados, frameworks e outras mecânicas que não devem ser acopladas aos mecanismos mais internos.

Para realizar a comunicação entre as camadas, tipicamente usa-se estruturas de dados simples, tais como structs ou objetos de dados simples. Não devem ser passadas entidades ou dados brutos de um banco de dados, pois ao fazer isso a regra da dependência seria violada. Estar de acordo com estas regra invariavelmente faz do sistema um organismo mais testável, assim como torna a substituição de componentes obsoletos uma tarefa muito mais simples.

2.6.1 VIPER

A arquitetura VIPER (*View, Interactor, Presenter, Entity e Routing*) possui dois objetivos principais: Resolver o problema dos ViewControllers massivos e introduzir melhorias na testabilidade do projeto. O acrônimo VIPER representa suas cinco camadas principais que servem como um framework arquitetural, onde há casos que menos ou mais camadas devem ser usadas pelo desenvolvedor[24]. As cinco camadas são:

- *View*: Exibe o que é ordenado pelo Presenter.
- *Interactor*: Contém regras de negócio especificadas por um caso de uso.
- *Presenter*: Contém a lógica de exibição da *View* e prepara elementos para serem exibido baseando-se em dados obtidos do componente *Interactor*.
- *Entity*: contains basic model objects used by the Interactor.
- *Routing*: Contém lógica de navegação e ordem de exibição de telas.

Esta arquitetura foca em problemas relacionados a dependências de classes e testabilidade, ao invés de problemas relacionados a dependências entre módulos[20]. Esta é uma arquitetura particularmente interessante, pois não se encaixa na categoria MV* e é considerada bastante popular no desenvolvimento de aplicações iOS, pois aproxima o desenvolvedor dos conceitos apresentados pela Arquitetura Limpa, que por sua vez é fundamentada em princípios consagrados de design orientado a objeto.

A introdução de Entities e Routers possuem grande importância neste padrão, assim como sua característica de possibilitar a separação de responsabilidades em componentes tão pequenos quanto se queira. O componente Interactor cuida da lógica relacionada às entidades, networking ou

persistência. Isso faz com que as entidades no modelo VIPER sejam implementadas como estruturas de dados brutos, tais como structs e enums.

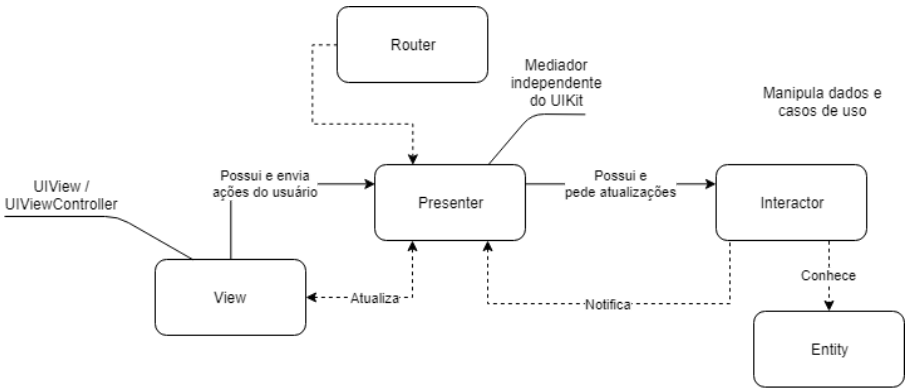

Figura 7: Modelo VIPER

As principais diferenças entre VIPER e as arquiteturas da classe MV* estão na responsabilidade atribuída ao seu componente Presenter. Ainda que este seja independente do framework UIKit, possui responsabilidades de coordenar lógicas de negócio relacionadas à UI. Também pode ser destacada a introdução de mecânicas que favorecem navegação e roteamento, cujo componente responsável é o Router.

Dentre suas desvantagens conhecidas se destacam a grande quantidade de código boilerplate, isto é, código que sempre é repetido sem nenhuma alteração em qualquer cenário que utilize tal arquitetura. Também possui um número excessivo de interfaces para classes com responsabilidades minúsculas. Importante citar o componente Presenter, que possui fluxos de dados multidirecionais, divergindo do que é proposto na Arquitetura Limpa, que originalmente encoraja o uso de fluxos de controle unidirecionais. Estas características podem fazer com que este modelo arquitetural seja considerado exagerado para aplicações simples com equipes de desenvolvimento reduzidas.

Gigantes do mercado como UBER se beneficiaram fortemente dos aspectos oferecidos por VIPER, adotada como um framework para solucionar problemas de testabilidade e dificuldades na adição de novas funcionalidades no seu aplicativo[6]. Isso os levou a implementar sua própria variação deste modelo, conhecido como Riblets[7], permitindo que seus desenvolvedores trabalhem em

[6] Uber: A engenharia por trás do novo aplicativo Rider, Disponível em: https://eng.uber.com/new-rider-app/
[7] RIBS: Plataforma de arquitetura de software mobile, Disponível em: https://github.com/uber/RIBs.

componentes com o máximo de desacoplamento, vencendo as barreiras dos Massive View Controllers originalmente introduzidos pelo uso do modelo MVC.

2.6.2 Clean Swift e o ciclo VIP

Clean Swift é mais uma tentativa de aplicar os conceitos introduzidos pela Arquitetura Limpa ao desenvolvimento de projetos iOS escritos em Swift. Essa abordagem de desenvolvimento é sustentada pelo ciclo VIP, e assim como a arquitetura VIPER, Clean Swift foca em testabilidade e na solução de ViewControllers massivos[25]. Um ponto crucial da arquitetura Clean Swift é seu fluxo de dados. Enquanto VIPER realiza comunicação componentes de modo multidirecional, o ciclo VIP utiliza a abordagem unidirecional.

Apesar de se basear no ciclo VIP, a implantação de Clean Swift em um projeto iOS requer que um processo bem definido seja seguido, por isso encoraja o uso de templates para facilitar sua implementação[22]. Apesar de tornar o desenvolvimento mais conveniente, o uso destes templates pode criar um framework limitado de soluções. A navegação entre telas se dá por meio do componente Router, assim como em VIPER. A comunicação entre seus outros componentes é feita via protocolos que trafegam os dados de maneira cíclica, de modo que Workers se comunicam com o Interactor, que por sua vez se comunica com um Presenters. Por fim o Presenter será encarregado de se comunicar com o ViewController que atualiza o estado da UI.

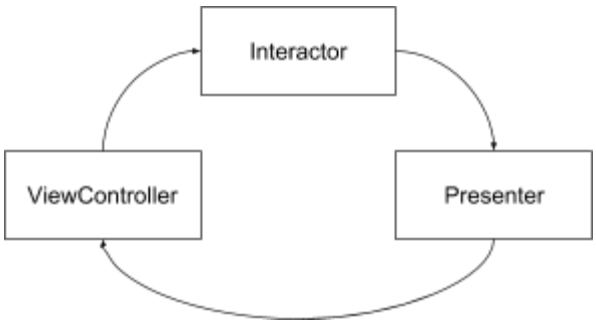

Figura 8: Diagrama VIP (View, Interactor, Presenter)

Em Clean Swift, cada modelo contém objetos de requisição e resposta. Para trafegar dados entre suas camadas são usadas estruturas de dados básicas como structs e enums, assim como interfaces. Nesta arquitetura, modelos são os únicos componentes totalmente independentes e desacoplados do template oferecido pelo framework.

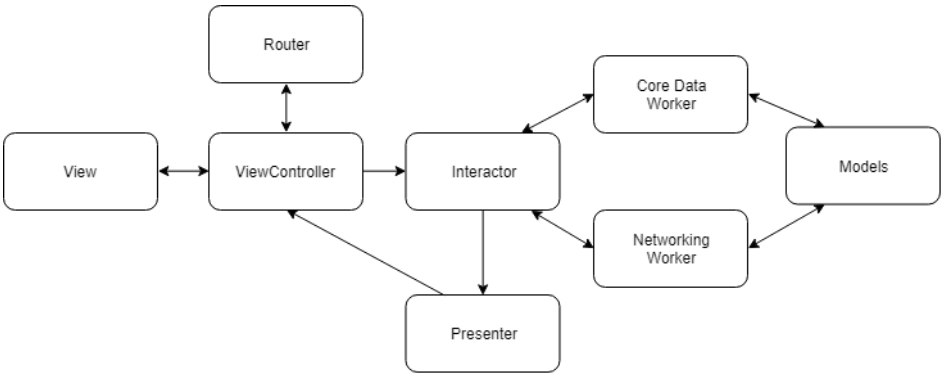

Figura 9: Clean Swift (VIP), Adaptado de [8]

Sua proposta é introduzir uma maneira sistemática de desenvolver aplicações com o mínimo de acoplamento possível, aliando os conceitos da arquitetura limpa com fluxo unidirecional de dado tornando seus componentes facilmente testáveis. Ao definir bem as responsabilidades de cada componente, é esperado que o projeto possua ViewControllers mais enxutas, pois atividades como roteamento, apresentação e interação são delegadas a classes do framework. Além da ser uma arquitetura, Clean Swift é visto um processo bem definido que deve ser seguido e requer bastante disciplina do desenvolvedor.

Ao introduzir o ciclo VIP, a arquitetura Clean Swift demonstra uma das suas principais vantagens: A solução do controle de fluxo encontrado na arquitetura VIPER, tornando-o unidirecional. Outra vantagem é não depender de reatividade como na arquitetura MVVM, pois Clean Swift tem uma natureza cíclica. Clean Swift é baseado fortemente nos princípios SOLID, educando o desenvolvedor por meio de um processo sistemático.

2.7 Métricas de Código

Métricas de código são medidas importantes que oferecem insights sobre aspectos de um software que dizem respeito a, por exemplo, sua manutenibilidade e complexidade, oferecendo uma visão melhor sobre o código desenvolvido. Normalmente métricas de código são pouco difundidas entre desenvolvedores, que acabam por não conhecer suas vantagens, deixando de identificar potenciais riscos e pontos que devem ser refatorados em seus projetos[6].

Apesar de subjetiva, a análise de métricas de código pode apontar problemas em alguns pontos específicos do código fonte de um projeto. Diversas heurísticas e métricas estão à disposição de desenvolvedores por meio de ferramentas, permitindo-lhes localizar trechos problemáticos no

código que podem levar a possíveis complicações. As métricas de software costumam ser usadas como um filtro. Devido à sua subjetividade, podem apresentar falsos-positivos, mas ainda assim auxiliam a avaliação automatizada sobre bases de código. O número e a definição dos parâmetros definidos para diferentes projetos é algo subjetivo. Apesar de existirem ferramentas que permitem calcular métricas[7], não é comum encontrá-las. Além destas, existem pesquisas que tentam deduzir números, valores ou parâmetros ótimos para determinadas heurísticas analisando repositórios de código aberto e observando como estes parâmetros se distribuem ao longo do código[19].

2.7.1 Tipos de métrica

Diversas métricas e heurísticas estão à disposição dos desenvolvedores para a análise de seus projetos. A ferramenta Visual Studio, por exemplo, implementa diversas métricas em sua IDE[6]. A complexidade ciclomática, por exemplo, mede a complexidade estrutural do código ao calcular do número de caminhos diferentes no fluxo de um programa. A Profundidade de herança, por sua vez, informa a respeito da hierarquia de uma classe. Métricas mais simples como contagem de parâmetros em métodos, tamanho de classes e até mesmo métodos de análise sintática, tais como Complexidade Ciclomática são oferecidas pelas ferramentas. Também são disponibilizadas métricas para a Análise de Coesão ou de Acoplamento.

2.7.2 Ferramentas de Métricas de Código

É normal que ferramentas de deste tipo apresentem baixa eficácia devido às métricas subjetivas que disponibilizam, mas algumas delas possuem funcionalidades interessantes. A ferramenta Sonar, por exemplo, oferece uma análise que mantém histórico de alterações e dados sobre o código de um projeto de maneira automática, livrando o desenvolvedor da responsabilidade de executar uma análise de tempos em tempos. Outros bons exemplos são as ferramentas MetricMiner[8], desenvolvida pela USP, e uma evolução sua, a ferramenta RepoDriller[12]. Ambas focam em flexibilidade, pois não assumem quais informações o pesquisador deseja minerar em um repositório, dando-lhe a opção de definir suas próprias métricas por meio da criação de casos de estudo e regras de visita a sessões do código.

Neste trabalho é feito uso de análise estática de código para a linguagem de programação Swift no desenvolvimento de aplicativos iOS. Dentre as ferramentas mais difundidas estão Taylor[9] e

[8] MetricMiner: Ferramenta deenvolvida para mineração de repositórios de software
[9] Taylor: Ferramenta de código aberto para melhoria na qualidade de código Swift.

Tailor[10]. Apesar de possuírem nomes similares, possuem características bastante diferentes. Ainda assim, ambas disponibilizam diferentes métodos e métricas de análise, analisando arquivos de código fonte em repositórios de maneira recursiva e provendo um relatório final reportando a conformidade do código analisado com as métricas escolhidas pelo desenvolvedor.

A ferramenta Taylor disponibiliza seis métricas de código. As análises executadas pela ferramenta são: Complexidade ciclomática, Profundidade de blocos aninhados e Complexidade N-Path, Tamanho de classes, Tamanho de métodos, Quantidade de métodos e Tamanho de listas de parâmetros. A ferramenta Tailor, por sua vez, disponibiliza trinta métricas, das quais a grande maioria diz respeito a análise puramente sintática e de estilo de código. Algumas destas métricas são apresentadas pela ferramenta Taylor, citada no parágrafo anterior. Dentre as métricas introduzidas pela ferramenta Tailor, destacam-se as análises de tamanho de closures, Tamanho de arquivos, Tamanho de Structs e Quantidade de imports por arquivo.

2.8 Conclusão do Capítulo

Este capítulo discute que constroem um alicerce para a metodologia e a pesquisa que são apresentadas nos capítulos subsequentes. Foi discutido desde a visão histórica do desenvolvimento de aplicações para dispositivos iOS e sua anatomia básica aos modelos arquiteturais, suas características e os benefícios da utilização de métricas de código no desenvolvimento de software.

[10] Tailor: Analisador sintático de código aberto para Swift.

3 METODOLOGIA

Diversos métodos e procedimentos foram adotados nesta pesquisa a fim de analisar comparativamente a qualidade do código e os impactos na estrutura de duas implementações do mesmo aplicativo utilizando diferentes modelos arquiteturais. Métodos quantitativos baseados em estatísticas coletadas por meio de métricas de código assim como a metodologia utilizada na análise dos coletados durante a pesquisa são apresentados neste capítulo.

3.1 Problemas

Para o propósito deste trabalho, comparar diferentes modelos arquiteturais no desenvolvimento móvel requer o uso de ferramentas maduras ou fortemente customizáveis para coleta de métricas de código. Apesar do framework iOS oferecer análise sintática a partir de sua IDE XCode, a análise de componentes, modularização e aspectos que dizem respeito à arquitetura da aplicação não estão presentes. Para tornar viável um estudo comparativo são necessárias métricas específicas que podem ser encontradas em algumas ferramentas de código aberto.

Comparar dois projetos requer que ambos reflitam os mesmos requisitos e casos de uso, assim como devem seguir à risca o processo de implementação de cada arquitetura estudada. Deste modo, a coleta de métricas tende a mostrar resultados significativos em projetos de qualquer escala. Neste trabalho foi implementado um aplicativo utilizando dois modelos arquiteturais. Ambas as implementações foram comparadas por meio de métricas de código e análise de diagrama.

3.2 O Aplicativo Create Order

Em 2011 em um meetup organizado pela Skills Matter, Robert C. Martin - popularmente conhecido na comunidade como Uncle Bob - apresentou em sua palestra intitulada *"Why can't anyone get Web architecture right?"* a implementação do caso de uso Create Order para aplicativos Java para Web[15] Sua palestra disponibilizada em formato ScreenCast[14] traz como principal questionamento o fato de mesmo após anos a fio onde designers e arquitetos aconselham estruturas que foquem em baixo acoplamento, abstração e padrões sólidos e testados, os frameworks web continuam cometendo os mesmos erros, levando os desenvolvedores a caírem nos mesmos problemas recorrentes. Robert então prega que WEB não é uma arquitetura, e que a aplicação será uma aplicação independente de sua plataforma, seja web, mobile ou um terminal.

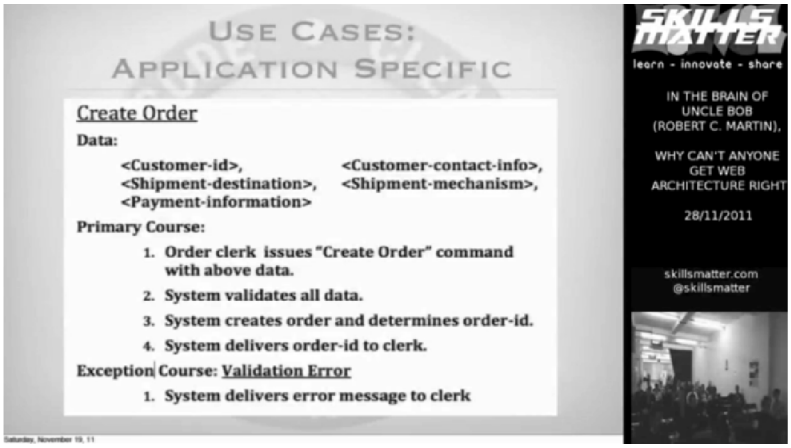

Figura 10: O Caso de Uso Create Order

O aplicativo iOS usado no estudo reflete uma implementação do caso de uso Create Order, reforçando o fato de que, independente da plataforma, o padrão arquitetural é desconexo do domínio da aplicação e deve ser implementado como tal à medida do possível, focando sempre nos princípios da Arquitetura Limpa. Este capítulo é iniciado com uma discussão a respeito das principais funcionalidades do aplicativo assim como sua interface com o usuário, suas mecânicas e casos de uso.

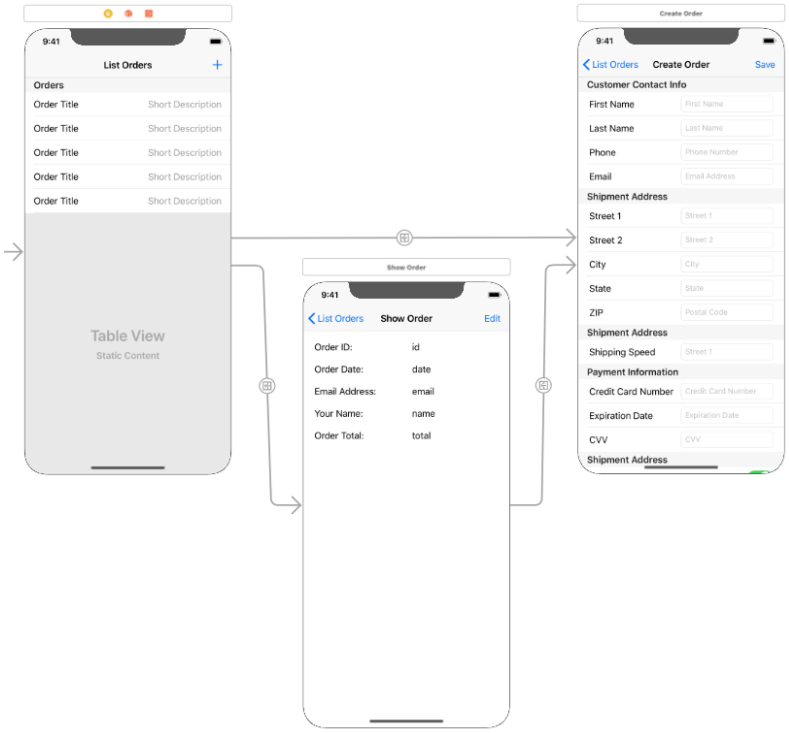

Figura 11: O Aplicativo Create Order

3.2.1 Interface com o Usuário

O aplicativo desenvolvido consiste em três telas que implementam os casos Listar, Criar e Editar itens. A primeira tela consiste na listagem. Sua lógica de negócio se resume a obter dados sobre itens já criados armazenados em cache. A tela reage à adição de novos itens, exibindo-os em forma de lista. Ao tocar nos itens da lista, o usuário é levado a uma tela de detalhes sobre o item em questão.

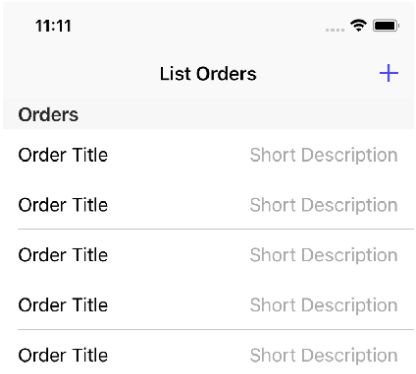

Figura 12: Tela de Listagem de Itens

A segunda tela é responsabilizada pela criação de itens. A entrada de dados se dá por meio de um formulário simples, processado e armazenados em cache. Sua lógica de negócio consiste em validar e mapear os dados da UI aos modelos da aplicação. A mesma tela encapsula toda a lógica de negócio relativa à edição de itens.

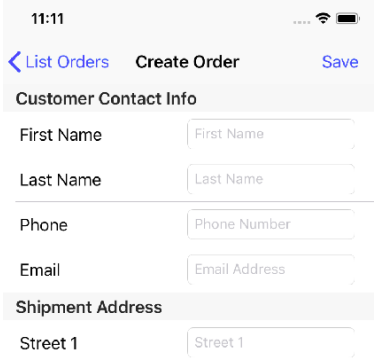

Figura 13: Tela de Criação e Edição de Itens

A terceira tela se trata da mais simples de todas, pois sua tarefa consiste somente em detalhar itens e lançar navegar para a tela de edição edição relacionada ao item que está sendo exibido caso o usuário assim deseje. A complexidade extra introduzida neste tela está em exibir dados atualizados imediatamente após a tela responsável pela edição sinalizar uma atualização do item atual.

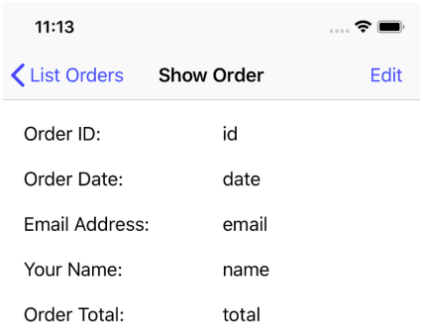

Figura 14: Tela de Detalhes

3.2.2 Fontes de dados

No desenvolvimento foram utilizados dois níveis de persistência de dados: Cache em memória e armazenamento em disco utilizando CoreData[16]. Utilizar duas mecânicas de persistência permite levantar observações a respeito do acoplamento da aplicação em relação à interface de acesso aos dados. Para isso, será definida uma interface que permitirá a execução de operações para criação, leitura e atualização de dados, assim como a troca do sistema de armazenamento sem maior esforço.

3.3 Implementação Clean Swift

A implementação do caso de uso Create Order depende da criação dos seus componentes principais, assim como suas telas e modelos seguindo os padrões definidos. Na organização da implementação seguindo Arquitetura Limpa, foi utilizado o conceito de Cenas. Todo o código é aberto e está disponível para acesso na plataforma GitHub[11]. Para isso foram criados os grupos CreateOrder, ListOrder e ShowOrder contendo todos os arquivos respectivos aos componentes necessários para a cena funcionar.

[11] Repositório do aplicativo CreateOrder, Disponível em: https://github.com/icaromagalhaes/ios-architectures-showcase

3.4 Coleta de Métricas de Código

Após a implementação do aplicativo Create Order, as ferramentas de análise estática Tailor e Taylor foram usadas para a obtenção de métricas. Cada ferramenta foi executada na raiz do diretório do projeto. Após a execução, arquivos com estatísticas a respeito das métricas escolhidas são gerados no formato CSV e sumarizados para a análise comparativa. Cada implementação possui um conjunto de dados exportados que são interpretados a fim de observar características divergentes assim como impactos causados pela mudança de arquitetura nos dois projetos.

4 APRESENTAÇÃO E ANÁLISE DOS RESULTADOS

Neste capítulo são apresentadas as métricas coletadas a partir do aplicativo desenvolvido, assim como a interpretação de seus resultados. Em seguida é apresentada uma comparação a nível arquitetural feita por meio de diagramas representando dependências, acoplamento e modularização do código. Explicar qual o real impacto da mudança arquitetural em um projeto e afirmar se isso realmente representa uma melhoria é uma tarefa difícil, pois se trata de algo subjetivo estando sujeito à qualidade das métricas obtidas. Um resultado, ainda que inconclusivo, pode dizer muito a respeito de como projetos similares se comportam em face a uma mudança arquitetural.

4.1 Resultados Obtidos a Partir Das Métricas Coletadas

A partir da análise sintática foram detectadas diversas violações no código fonte, assim como diferenças consideráveis na quantidade de componentes entre os dois projetos. Estes resultados são importantes para entender onde se concentram as reais diferenças e quais impactos trazem ao código fonte do projeto, dando menos foco aos aspectos arquiteturais de cada implementação. Para tornar a visualização dos resultados coletados pelas ferramentas de métrica foram utilizados gráficos de barra.

4.1.1 Análise de violações sintáticas

Para comparar o impacto da mudança arquitetural no projeto, foi realizada uma análise de violações a partir da coleta de métricas de código fazendo uso das ferramentas Taylor e Tailor. Esta etapa da análise busca identificar características peculiares nos códigos fonte das as duas implementações.

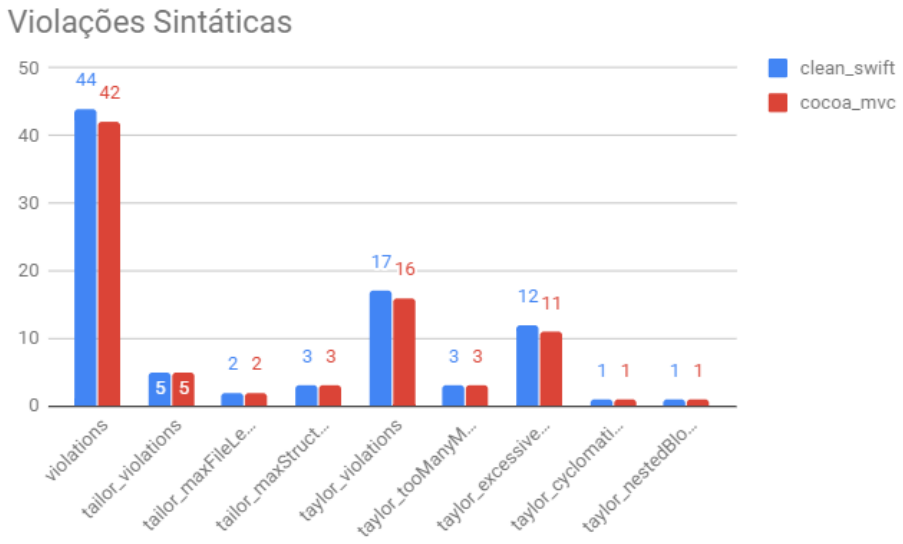

Figura 15: Clean Swift: Violações Sintáticas

A partir dos resultados obtidos foi observado que, apesar dos dois padrões arquiteturais adotados utilizarem meios para organizar e separar componentes de modos consideravelmente diferentes, a análise sintática ofusca detalhes arquiteturais e revela poucas mudanças a respeito da sintaxe do código. Uma explicação plausível está no fato de que toda a lógica de negócio da aplicação é implementada utilizando os mesmos métodos, independente do tipo de arquitetura utilizada. Apesar da diferença no fluxo de dados entre os dois padrões, os principais métodos de manipulação e passagem de dados se mantêm intactos com poucas variações. Pode presumir-se que a maior diferença entre as duas implementações deve ser acusada por uma análise de componentes, onde se concentra a grande diferença entre os dois modelos arquiteturais.

4.1.2 Contagem de componentes

Nesta etapa da análise, foram coletadas métricas a respeito dos componentes comuns a todos os aplicativos iOS, tais como classes, extensões, protocolos, estruturas e enumerações. Pode-se notar que, diferente da análise de violações sintáticas, houve uma grande diferença na quantidade de componentes utilizados entre as duas implementações.

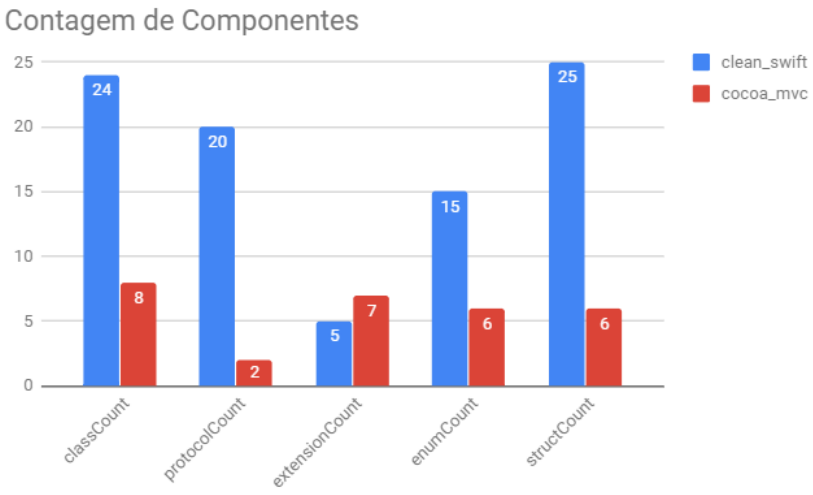

Figura 16: Clean Swift: Contagem de Componentes

Isto pode ser explicado pelo fato da implementação Clean Swift introduzir todos os componentes do ciclo VIP, assim como os componentes extras que dizem respeito a passagem de dados, navegação e roteamento. A implementação MVC, por sua vez, encoraja o encapsulamento aspectos que são delegados a outras classes na implementação Clean Swift a seus próprios ViewControllers. Nota-se a maior diferença na quantidade de classes, protocolos e structs. Isso se dá pelo fato de Clean Swift utilizar protocolos para comunicar seus componentes, e estruturas para passagem de objetos Request e Response entre eles. A menor diferença é notada na contagem de extensões. Extensões são uma característica da linguagem Swift que permitem a adição de funcionalidades em classes com acesso privado[14], e isto se torna mais evidente nos ViewControllers, onde é mais comum surgir a necessidade de desacoplar da classe principal parte da lógica ligada à UI do aplicativo, que por sua vez é compartilhada entre as duas implementações, tornando-as similares.

4.2 Análise Arquitetural

Apesar das ferramentas de métrica de código acusarem diferenças sintáticas não tão significativas entre as duas implementações analisadas, a base de código foi analisada sob um ponto de vista arquitetural. Deste modo muitas características interessantes foram notadas na implementação Clean Swift.

4.2.1 Análise de Acoplamento

A arquitetura Clean Swift aliada ao ciclo VIP introduz flexibilidade que permite uma variação entre o número de camadas de separação do projeto. Ainda assim, o core do sistema deve seguir o ciclo VIP. Deste modo, o ciclo pode ser visto como um padrão de design organizacional, resolvendo problemas de dependências entre classes como os ViewControllers massivos e a testabilidade do sistema como um todo. Ainda assim, foi observado que o desenvolvedor pode acabar criando código muito acoplado, ainda que seja testável.

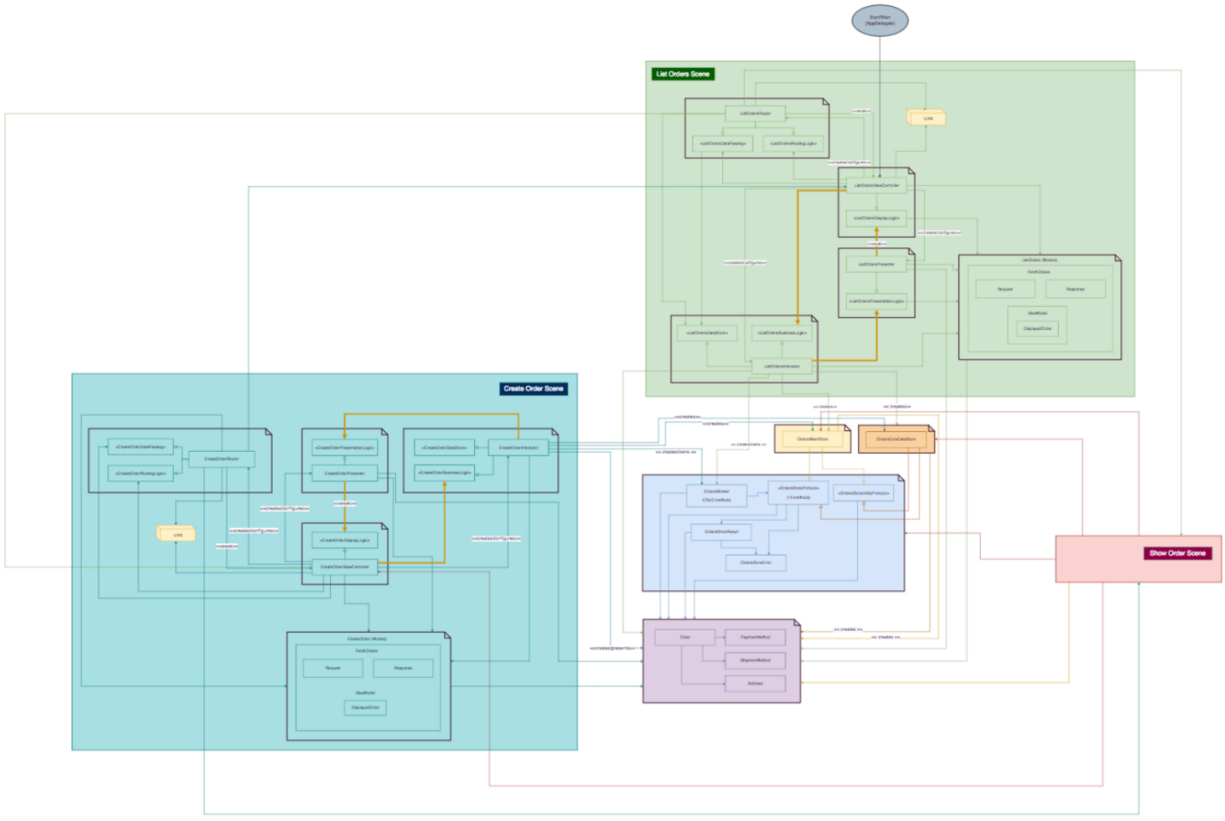

Figura 17: Acoplamento da Implementação Clean Swift

4.2.2 A Cena ListOrders

Na implementação baseada em Clean Swift, no momento que a aplicação é iniciada e instancia a primeira cena, List Orders, diversas outras classes provenientes dos templates são instanciadas, criando imediatamente um forte acoplamento entre o ponto de entrada do aplicativo e os componentes do ciclo VIP, que é representado pelas setas destacadas na figura.

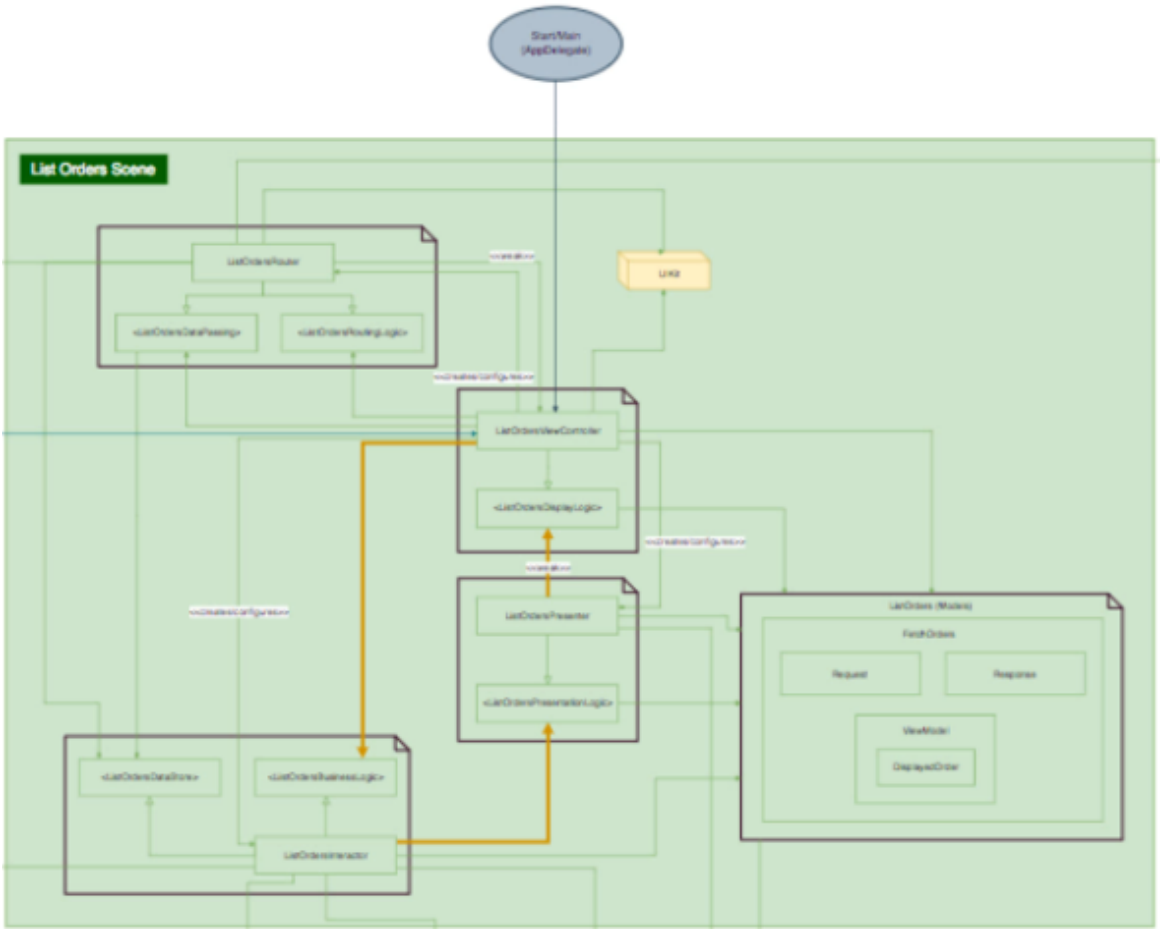

Figura 18: Acoplamento Entre Componentes do Ciclo VIP

O problema de forte acoplamento impacta diretamente na qualidade de modularização do sistema, uma vez que alguns componentes são obrigatoriamente dependentes de outros. Neste caso, a arquitetura não se preocupa em resolver este tipo de problema. Ao invés disso, encoraja o desenvolvedor a criar estes componentes em cada cena a partir de templates. Até mesmo os componentes chamados de Workers definem lógica que deve estar em conformidade com determinados protocolos, criando assim um código ainda mais acoplado. A única maneira de resolver este tipo de dependência é fugir do modelo proposto pelo autor ao implementar uma aplicação baseada em Clean Swift.

Todos estes pontos fazem de Clean Swift um padrão ideal para quem pretende criar uma base de código altamente testável, ao preço de tornar o projeto engessado aos templates. O alto acoplamento pode tornar a colaboração difícil.

4.2.3 Impactos do uso de Templates em CleanSwift

Templates para implementar projetos utilizando Clean Swift são uma adição muito conveniente ao processo de desenvolvimento, porém se analisarmos a fundo é fácil perceber que acaba criando framework limitado de soluções, baseando-se no fato de que não há um template genérico que possa resolver todos os tipos de problema. É preferível que o desenvolvedor faça uso de metodologias que possam ser facilmente estendidas, visando evitar que o processo para solucionar um problema se resuma a tentar encaixá-lo em um padrão pré definido no seu problema.

4.2.4 Diagrama de dependências

A partir da figura 19 é possível analisar o acoplamento em um nível de abstração ainda maior.

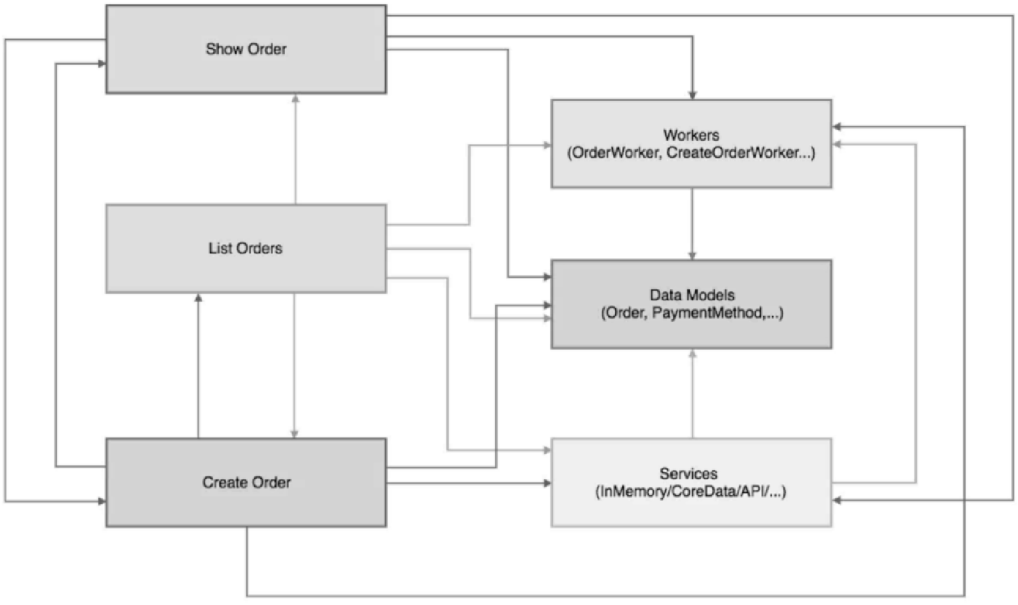

Figura 19: Diagrama de Dependências de Alto Nível

O forte acoplamento pode representar perigos à saúde do projeto. Uma simples mudança nos Data Models pode quebrar todos os outros módulos, pois todos eles estão amarrados. Também é fácil observar que mudanças nos Workers também podem quebrar outros módulos, com exceção dos Data Models, pois estes são independentes.

4.3 Implementação CleanSwift vs Arquitetura Limpa

Na Arquitetura Limpa definida por Uncle Bob, existe a preocupação com dependências entre classes e entre módulos. Além de limpo, o código deve ser testável e modular, facilitando mudanças e colaboração.

Figura 20: Camadas - Clean Swift

No diagrama circular se torna claro que os Data Models não dependem de nenhum outro módulo. Do mesmo modo, Workers dependem dos Data Models e nada mais. O mesmo ocorre para a camada de serviços. O real problema surge nas camadas mais externas. O fato de que componentes que se encontram na camada externa possam depender de camadas mais internas vai contra os princípios da arquitetura limpa[17]. E ainda pior, existem componentes da camada mais

externa que dependem de mais de uma camada mais interna. Deste modo, modificar um dos anéis internos pode propagar problemas para camadas mais externas.

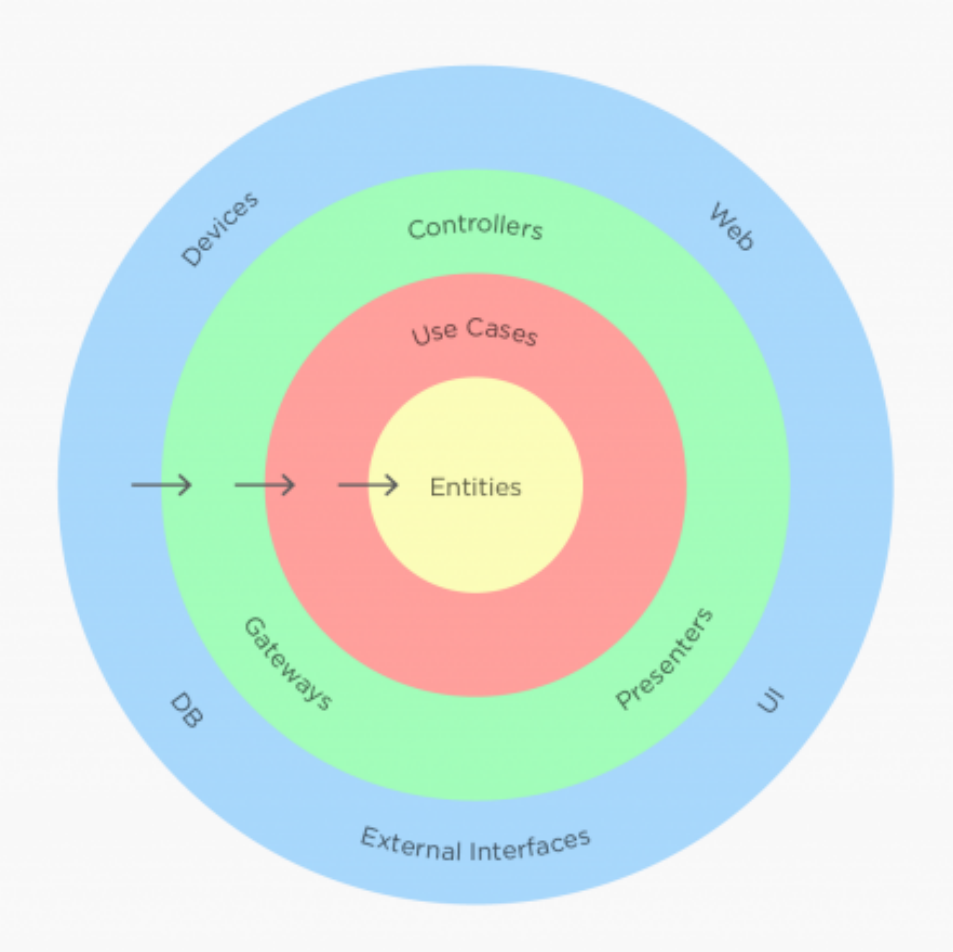

Figura 21: Camadas da Arquitetura Limpa

Diferente do é pregado pela arquitetura limpa, na implementação executada temos frameworks e drivers nas camadas centrais. Isso não representa necessariamente um problema, porém representa alto acoplamento.

5 CONCLUSÕES E TRABALHOS FUTUROS

Na busca por sistemas de softwares que possuem modularidade, testabilidade, reuso e facilidade de manutenção o arquiteto de software analisa diferentes modelos e padrões levando em consideração cada aspecto do projeto em questão. Um ponto crucial nesta análise está na percepção de que projetos que possuem diferentes requisitos, ainda que desenvolvidos a partir de um mesmo *framework*, podem apresentar comportamentos divergentes quando implementados sob diferentes modelos arquiteturais. A escolha de uma boa arquitetura passa a depender de algum modo da experiência prévia do arquiteto, porém este deve ter sempre em mente que cada sistema é de certa forma único e deve ser cuidadosamente construído.

A análise executada neste trabalho tornou possível observar sob uma nova perspectiva como as responsabilidades passaram a ser delegadas entre os componentes particulares de cada arquitetura utilizada. A partir disso, é possível fazer observações a respeito de como o código se comporta a partir dos pontos de vista sintático e estrutural. Ainda que pouco conclusivas, tais observações dizem muito sobre como projetos iOS de maior porte tendem a se comportar sob tais circunstâncias.

A partir das métricas de código observadas, nota-se que na abordagem *Clean Swift* o projeto tende a possuir muitos componentes em comparação à abordagem Cocoa MVC, principalmente, devido à introdução dos componente utilizados no ciclo VIP e componentes extras que se encarregam de roteamento e passagem de dados. O uso dos templates *Clean Swift*, apesar de melhorarem a testabilidade , colocam o projeto na contramão do que a arquitetura limpa prega a respeito de modularização e acoplamento. É bom ter em vista que, neste caso, o alto acoplamento não significa necessariamente algo ruim. Ainda que isso aconteça, a separação de responsabilidades entre componentes é melhorada a custo de mais complexidade no código, o que pode aumentar a curva de aprendizado para desenvolvedores iniciantes e até mesmo para os mais experientes, mas sem conhecimento prévio deste modelo arquitetural.

Outro ponto que foi observado é que os templates de *Clean Swift* invariavelmente tornam o código mais acoplado ao *framework* e aos serviços, pois camadas mais externas se comunicam com camadas internas para respeitar a ordem dos componentes do ciclo VIP. A comunicação entre estes componentes se dá de modo complicado. Isso porque, apesar de manter o fluxo de dados unidirecional, o que pode ser visto como uma vantagem, *Clean Swift* aborda de modo complexo atividades que podem ser resolvidas com simplicidade na implementação Cocoa MVC. Ainda

assim, *Clean Swift* exibe características de uma boa arquitetura para projetos que necessitam de alta testabilidade. Ainda que mantenha forte acoplamento violando princípios relacionados à coesão e responsabilidade de classes, todos os componentes introduzidos por este modelo podem ser independentemente testáveis. Dessa forma, a principal constatação deste trabalho é que padrões arquiteturais recentes e alternativos aos modelos de variação MV*, como por exemplo o Clean Swift, não necessariamente trazem benefícios, pelo contrário, podem ter impactos negativos quando se analisa os aspectos de qualidade de arquitetura de software.

5.1 Trabalhos Futuros

Durante a pesquisa, oportunidades de expansão foram vislumbradas em diversos caminhos. Tendo em vista que a qualidade dos resultados obtidos neste trabalho é afetada pelo fato de somente um projeto ter sido analisado, a análise executada sob projetos de grande porte pode apresentar resultados mais significativos no que diz respeito às métricas coletadas. No que diz respeito à modularização e componentização, projetos maiores que utilizam MVC tendem a apresentar *ViewControllers* ainda maiores e com mais responsabilidades, assim como novas regras e interações mais complexas, cada vez mais dependentes do framework.

Diferentes modelos arquiteturais presentes na comunidade como os da classe MV* também podem ser comparados entre si, não só em quesitos arquiteturais, mas também em aspectos de desempenho que analisem, por exemplo, qual o impacto dos sinais e bindings utilizados no padrão MVVM que notificam *Views* ao observarem alterações em *ViewModels*. Observar como *Clean Swift* se compara em relação ao modelo VIPER no quesito estrutural também pode oferecer discussões interessantes a respeito de quais componentes são opcionais e quais devem receber maior importância no desenvolvimento iOS.

REFERÊNCIAS

[1] REF Krasner GE, Pope ST. A cookbook for using the model - view controller user interface paradigm in Smalltalk - 80. 1550 Plymouth Street Mountain View; 1998.

[2] ORLOV, Bohdan. iOS Architecture Patterns - iOS App Development. US: Medium, 2018. Disponível em: https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52. Acesso em: 23 jun. 2018.

[3] APPLE Developer Documentation. 2018. Disponível em: https://developer.apple.com/documentation/. Acesso em: 25 jul. 2018.

[4] Managing Your App Lifecycle. 2018. Disponível em: https://developer.apple.com/documentation/uikit/core_app/managing_your_app_s_life_cycle.. Acesso em: 22 ago. 2018.

[5] EIDHOF, Chris. Lighter View Controllers. 2013. Disponível em: https://www.objc.io/issues/1-view-controllers/lighter-view-controllers/. Acesso em: 20 ago. 2018.

[6] Understand the complexity and maintainability of your code using Code Metrics in Visual Studio. 2018. Disponível em: https://dailydotnettips.com/understand-the-complexity-and-maintainability-of-your-coding-using-code-metrics-in-visual-studio/. Acesso em: 01 set. 2018.

[7] Microsoft: Medições de software. 2018. Disponível em: https://docs.microsoft.com/pt-br/visualstudio/code-quality/code-metrics-values?view=vs-2017. Acesso em: 30 set. 2018.

[8] ATANASOV, Dejan. Introducing Clean Swift Architecture (VIP). 2017. Disponível em: https://hackernoon.com/introducing-clean-swift-architecture-vip-770a639ad7bf. Acesso em: 12 ago. 2018.

[9] Model-View-Controller. US: Apple Inc., 2018. Disponível em: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html. Acesso em: 29 jun. 2018.

[10] Introduction to Key-Value Observing Programming Guide. US: Apple Inc., 2018. Disponível em: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.html. Acesso em: 21 jun. 2018.

[11] The MVVM Pattern. US: Microsoft, 2012. Disponível em: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10). Acesso em: 29 jun. 2018.

[12] ANICHE, Maurício. A tool to support researchers on mining software repositories studies. 2018. Disponível em: https://github.com/mauricioaniche/repodriller.

[13] HABCHI, Sarra. "**Code Smells in iOS Apps:** How Do They Compare to Android?" 2017. Disponível em: https://ieeexplore.ieee.org/document/7972725/. Acesso em: 30 ago. 2018.

[14] MARTIN, Robert. Why can't anyone get Web architecture right? **Skills Matter**, 2011. Disponível em: https://skillsmatter.com/skillscasts/2437-uncle-bob-web-architecture. Acesso em: 02 set. 2018.

[15] MARTIN, Robert. Why can't anyone get Web architecture right? **Skills Matter**, 2011. Disponível em: https://skillsmatter.com/meetups/1111-uncle-bob-architecture. Acesso em: 02 set. 2018

[16] Core Data: Managed object graphs and object lifecycle, including persistence. Disponível em: https://developer.apple.com/documentation/coredata. Acesso em: 09 set. 2018

[17] MARTIN, Robert. The Clean Architecture. **The Clean Code Blog**, 2012. Disponível em: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. Acesso em: 12 set. 2018

[18] R. L. Glass, "Frequently forgotten fundamental facts about software engineering," IEEE software, vol. 18, no. 3, pp. 112–111, 2001.

[19] ANICHE, Maurício. **Orientação a Objetos e SOLID para Ninjas: Projetando classes flexíveis.** 1. ed. São Paulo: Casa do Código, 2015.

[20] APOSTOLAKIS, Mike; ZULLO, Caio. Clean iOS Architecture pt.6: VIPER – Design Pattern or Architecture? **Essential Developer,** 2018. Disponível em: https://www.essentialdeveloper.com/articles/clean-ios-architecture-pt-6-viper-design-pattern-or-architecture. Acesso em: 02 nov. 2018

[21] **Model-View-Controller.** 2018. Disponível em: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html. Acesso em: 03 out. 2018.

[22] RAYMOND. **Clean Swift templates now update for Xcode 8.3.3 & 9.0 and Swift 3 & 4.** 2018. Disponível em: https://clean-swift.com/swift-3-compatible/. Acesso em: 05 out. 2018

[23] EIDHOF, Chris. GALLAGHER, Matt. KUGLER, Florian. **App Architecture iOS Application Design Patterns in Swift.** 2018. Disponível em: https://www.objc.io/books/app-architecture/. Acesso em: 09 out. 2018.

[24] GILBERT, Jeff; STOLL, Conrad. **Architecting ios apps with viper.** 2014. Disponível em: https://www.objc.io/issues/13-architecture/viper/. Acesso em: 08 set. 2018..

[25] RAYMOND. **Clean Swift iOS Architecture for Fixing Massive view controller.** 2018. Disponível em: http://clean-swift.com/clean-swift-ios-architecture/. Acesso em: 03 ago. 2018.