UNIVERSIDADE DO SUL DE SANTA CATARINA

AMANDA ROVER

PADRÕES ARQUITETURAIS DE SOFTWARE PARA A CONSTRUÇÃO DE SOLUÇÕES MOBILE

Florianópolis

2018

AMANDA ROVER

PADRÕES ARQUITETURAIS DE SOFTWARE PARA A CONSTRUÇÃO DE SOLUÇÕES MOBILE

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação da Universidade do Sul de Santa Catarina, como requisito parcial à obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Flavio Ceci, Dr.

Florianópolis

2018

AMANDA ROVER

PADRÕES ARQUITETURAIS DE SOFTWARE PARA A CONSTRUÇÃO DE SOLUÇÕES MOBILE

Este Trabalho de Conclusão de Curso foi julgado adequado à obtenção do título de Bacharel em Sistemas de Informação e aprovado em sua forma final pelo Curso de Graduação em Sistemas de Informação da Universidade do Sul de Santa Catarina.

Florianópolis, 19 de novembro de 2018.

Professor e orientador Flávio Ceci, Dr.
Universidade do Sul de Santa Catarina

Professora Alessandra A. Wust.
Universidade do Sul de Santa Catarina

Professor Richard H. Souza.
Universidade do Sul de Santa Catarina

1

Dedico esta, bem como todas as minhas demais conquistas, aos meus amados pais João José Rover e Márcia Tomasi Rover, e meu irmão João Ricardo Rover.

AGRADECIMENTOS

Agradeço aos meus pais, pela capacidade de acreditar em mim e investir em mim. Mãe, seus conselhos e cuidados foi o que deram a esperança para seguir nessa conquista. Pai, que apesar de todas as dificuldades que enfrentei, me fortaleceu, sua presença significou segurança e confiança de que não estou sozinha.

Agradeço ao meu irmão, pelos valiosos conselhos e sabedoria que recebi durante o curso. Ao meu namorado, por me ajudar muitas vezes a achar soluções quando elas pareciam não aparecer e que não me deixou ser vencida pelo cansaço.

Agradeço também a todos os professores que me acompanharam durante a graduação, em especial ao Prof. Dr. e orientador Flavio Ceci que teve paciência e compreensão a me orientar e concluir este trabalho.

RESUMO

Criar uma arquitetura de software é uma tarefa considerada difícil. Para facilitar esse desenvolvimento, existem alguns padrões arquiteturais que garantem a construção de uma aplicação com qualidade, pois a qualidade é um ponto indispensável para o desenvolvimento. Com isso, esta monografia tem como objetivo analisar uma seleção de padrões arquiteturais para soluções mobile, levando em consideração a facilidade na criação de testes e qualidade do código. Assim, utilizar um padrão arquitetural para o desenvolvimento mobile facilita o processo de criação e de manutenção do *software*, tanto quanto o aumento de produtividade da pessoa desenvolvedora. As etapas metodológicas desta monografia se caracterizam a modelar o cenário da solução proposta, aplicar os padrões arquiteturais apresentados, escolher o padrão arquitetural que mais condiz com a aplicação e avaliar a escolha com a criação e a aplicação de um formulário. Para o desenvolvimento foi apresentado a diferenciação dos padrões arquiteturais, aplicado esses padrões arquiteturais na solução proposta, feita uma análise em cima da diferenciação entre os modelos e apresentado a forma como foi criado e aplicado o questionário. Com base neste desenvolvimento e nos critérios abordados, foi possível chegar à conclusão que a melhor escolha do padrão arquitetural levando em consideração a facilidade na criação de testes e qualidade de código é o padrão chamado de Model View View-Model, também conhecido como MVVM. A partir desta conclusão, é levantado, para trabalhos futuros, o desenvolvimento real da solução mobile apresentada seguindo esse padrão arquitetural escolhido.

Palavras-chave: Padrões arquiteturais. Arquitetura mobile. Testabilidade.

LISTA DE ILUSTRAÇÕES

LISTA DE LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

A introdução dos dispositivos mobile na sociedade foi bem impactante, como mostra Morris (2007, p. 9) ao comentar da importância desta introdução, possibilitando novos usos sociais, como novos padrões comportamentais que rapidamente se tornaram normais.

O uso dos dispositivos móveis alterou as formas de se fazer negócios e alterou relações familiares. Muitas outras tecnologias não tiveram esse mesmo efeito e avanço rápido que se têm hoje com os dispositivos mobile (MORRIS 2007, p. 9).

Pode-se perceber um grande crescimento no desenvolvimento de aplicativos para esses dispositivos. Assim, como Tarkoma (2009, p. 2) cita, desde 2006 se tem percebido um grande crescimento no número de dispositivos *mobile*, e eles têm se tornado cada vez mais exigentes quanto a funcionalidades avançadas.

Para que o processo de construção de software complexos seja feito com alta qualidade, é essencial aplicar um padrão arquitetural no projeto, conhecido como *Design Pattern.* Com isso, é possível gerar processos práticos de desenvolvimento com menos esforço (IIDA 1999).

Vários dos autores citados nesta monografia se referenciam ao *Design Patterns* de múltiplas formas. No Brasil, o *Design Patterns* é comumente chamado de Padrões de Projetos. Como o objetivo é descrever sobre o desenvolvimento da arquitetura de softwares, então este trabalho é focado em padrões de projetos arquiteturais, que são chamados de padrões arquiteturais.

Ambler (1998), citado por Iida (1999, p. 2, tradução nossa), explica na sua visão o que é o padrão arquitetural:

> Ele define o padrão arquitetural como um padrão que descreve uma técnica de gerenciamento comum ou uma estrutura de organização potencial, enquanto o padrão de processo é definido como um padrão que descreve uma abordagem comprovada e bem-sucedida e ou uma série de ações para o desenvolvimento de software.

A definição de Gamma e outros (1994) também segue esse pensamento. Ele afirma que o padrão arquitetural nomeia, abstrai e identifica os principais aspectos de uma estrutura arquitetural comum que o torna útil para criar um design reutilizável. O padrão arquitetural identifica as classes e instâncias participantes, suas funções e colaborações e a distribuição de responsabilidade.

Para Martin (2017, p. 27, tradução nossa) um dos objetivos dos padrões arquiteturais é principalemente: “[...] minimizar os recursos humanos requeridos para construir e manter um sistema”.

As diferentes formas de arquitetura de desenvolvimento, que existem atualmente e que serão descritos e especificados posteriormente, são os modelos chamados de Model View Controller (MVC), Model-View-Presenter (MVP), Presentation Model (PM) e o Model-View-ViewModel (MVVM).

No decorrer deste trabalho, são apresentados, com mais detalhes, cada um desses padrões arquiteturais citados nesse capítulo.

1.1 PROBLEMÁTICA

Criar uma arquitetura de software para Gamma e outros (1994) é considerado difícil, e criar uma arquitetura de software reutilizável é ainda mais difícil. Gamma explica que se deve encontrar objetos pertinentes, classificá-los com a granularidade correta, definir interfaces de classe e hierarquias e estabelecer relacionamentos-chave entre eles. Por isso ele conclui que é uma tarefa mais custosa de se fazer.

Por ser considerado uma tarefa complexa, é necessário investir tempo de estudo para a criação ou utilização de um padrão arquitetural. Com isso, muitas empresas decidem não investir em um padrão arquitetural para o sistema.

Quando não se constrói uma arquitetura estruturada para o sistema, dá a impressão que as pessoas que a desenvolvem criam código mais rapidamente. Escrever código, sem essa organização, é rápido em curto prazo, mas atrasa a criação de código no longo prazo. Esse código ficará desorganizado e será menos compreensível. Isso fará com que a pessoa que desenvolve o sistema enfrente maiores problemas ao inserir novos códigos no sistema (MARTIN 2017).

Nesse cenário, ao longo do tempo, torna-se mais difícil criar e manter o sistema, porque o código ficará confuso e diminuirá sua qualidade. No futuro, será preciso adicionar novas funções ao sistema, e isso demandará cada vez mais esforço das pessoas que as programam. Enfrentando essa dificuldade, uma empresa pode pensar em aumentar o número de pessoas desenvolvedoras no time para aumentar o número de entregas, mas ainda, assim, a

produtividade desse time continuará baixa, isso porque o código criado inicialmente se tornou cada vez mais confuso e bagunçado, demandando mais tempo do programador e diminuindo a produtividade de todos. Aumentando o time de pessoas que desenvolvem o sistema, mas mantendo o número de entregas, aumentará o custo de desenvolvimento do software, tornando-se inviável para a empresa (MARTIN 2017).

Para solucionar ou evitar todos esses problemas, precisa-se utilizar um padrão arquitetural correto e específico para o software.

1.2 OBJETIVOS

Esta seção aborda os objetivos definidos para este trabalho, citando o objetivo geral e os objetivos específicos.

1.2.1 Objetivo geral

O objetivo desta monografia é analisar os padrões arquiteturais para soluções mobile, levando em consideração a facilidade na criação de testes e qualidade do código.

1.2.2 Objetivo específico

Os objetivos específicos desta monografia estão listados a seguir.

- identificar os padrões arquiteturais que podem ser utilizados para o desenvolvimento mobile;
- analisar as características de cada um dos padrões arquiteturais listados;
- aplicar cada um dos padrões arquiteturais listados na solução proposta;
- selecionar o padrão arquitetural que mais condiz com a solução final

- avaliar a análise feita via questionário com profissionais da área.

1.3 JUSTIFICATIVA

Desde que as pessoas começaram a criar interfaces para softwares, o padrão arquitetural começou a ficar mais popular justamente para facilitar o processo de criação e manutenção do software (MIHAILESC 2010).

Iida (1999, p. 1, tradução nossa) comenta que "para poder produzir software em larga escala e de qualidade alta em um cronograma específico, o planejamento preciso e o gerenciamento adequado dos projetos são essenciais".

Existem muitos benefícios para quem busca decompor um software em unidades flexíveis. Um dos maiores benefícios inclui a separação das preocupações do sistema e a reutilização de uma unidade. Por ser uma vantagem desse método, essa prática de construção de aplicações complexas, a partir de unidades menores, evoluiu significativamente nas últimas décadas (TARKOMA 2009, p. 26).

Além de facilitar o processo de criação, outras vantagens de se ter um sistema com o padrão arquitetural correto, que supre as necessidades do negócio, é que, assim, facilita no aumento da produtividade das pessoas que desenvolvem, facilitando na hora de adicionar novas responsabilidades ao sistema e facilitando, também, a manutenção de código. Consequentemente, acaba diminuindo o custo de produção do software, pois se torna mais rápida qualquer ação ou qualquer adição de código no sistema (MARTIN 2017).

Outro ponto vantajoso, citado por Martin (2017), é que utilizar um padrão arquitetural bem planejado, estruturado e específico para o software, irá prolongar a vida do sistema construído e se tornará mais benéfico, sendo mais vantajoso para as empresas.

Para Gamma e outro (1994), para alguns projetos, os padrões arquiteturais resolvem problemas específicos que pode não ocorrer em outros projetos e ajudam a deixá-los mais flexíveis, elegantes e reutilizáveis.

Este trabalho tem como foco facilitar o entendimento dos padrões arquiteturais atuais e analisar as vantagens citadas anteriormente, neste capítulo, sobre utilizar um padrão de arquitetura de software em um sistema específico. Pretende-se com a construção deste trabalho

contribuir na decisão de escolha de um padrão arquitetural que facilite a criação de testes e que ajude a melhorar a qualidade do código do sistema para soluções *mobile*.

1.4 ESTRUTURA DO TRABALHO

No capítulo 2, é realizado um levantamento sobre alguns dos padrões arquiteturais conhecidos. No capítulo 3, é apresentado o método escolhido. No capítulo 4, é feito a modelagem da solução apresentada. No capítulo 5, são descritos os pontos positivos e negativos de cada padrão arquitetural apresentado nesta monografia, comparando entre cada um deles e definindo qual é o padrão arquitetural que mais condiz com o sistema proposto e, por fim, no capítulo 6, são apresentadas as conclusões da pesquisa realizada, assim como do modelo proposto, dando sugestões para trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo explica sobre a importância e as aplicações da arquitetura de desenvolvimento que existem atualmente.

A seguir, é explicado o que são e onde são aplicados os modelos de padrões arquiteturais, trazendo exemplos técnicos. São citados alguns tipos de padrões arquiteturais e explicado os padrões: MVC, MVP, PM e MVVM.

Então, é falado sobre a definição do princípio SOLID criado por Robert Cecil Martin.

Na sequência, explica-se como funciona o desenvolvimento *mobile* e quais ferramentas se podem utilizar para desenvolver aplicações *mobile*.

Como último ponto, são explicados os testes automatizados e seus tipos de testes para *mobile*.

2.1 PADRÕES ARQUITETURAIS DE PROJETOS

Nesta seção, é apresentada a arquitetura de software, explicando sobre os padrões arquiteturais e sistemas que não utilizam arquitetura bem definida.

2.1.1 Arquitetura de software

A arquitetura de software pode ser definida de várias formas, contudo todas as definições se convergem para o mesmo sentido. Para Hanmer (2013), a arquitetura de software tem o propósito de criar uma solução para uma necessidade comercial ou técnica. É uma estrutura que determina um fluxo de informações por um sistema de software. É uma série de decisões feitas que define como é organizado o software e como ele se comporta a partir deste momento (WILSON 2015).

Buschmann e outros (1996, p. 8) descrevem a visão sobre o que é a arquitetura de software:

> A arquitetura de software é uma descrição dos subsistemas e componentes de um sistema de software e as relações entre eles. Os subsistemas e componentes são normalmente especificados em diferentes visualizações para mostrar as propriedades funcionais e não funcionais relevantes de um sistema de software. A arquitetura de software de um sistema é um artefato. É o resultado da atividade de *design* de software.

Para Medvidovic e Taylor (2010, p. 471) qualquer aplicação que tenha um tamanho significativo ou complexidade significativa, no início do projeto deve ter uma arquitetura definida com antecedência. Fazendo uma comparação da arquitetura de software com a arquitetura de um prédio, para criar um edifício com uma qualidade boa é necessário a consideração de sua arquitetura antes da construção.

A arquitetura de software abrange os aspectos técnicos, metodológicos e processuais da engenharia de software. Aborda as necessidades de desenvolvimento e manutenção de software, contendo um forte impacto na qualidade final de um sistema (BUSCHMANN et al. 1996, p. 391).

A arquitetura de software, segundo Medvidovic e Taylor (2010, p. 471), deve estar no centro do *design* e desenvolvimento de sistemas de software. Deve estar em primeiro plano, estando acima do processo, acima da análise e, certamente, acima de programação. Somente dando atenção e proeminência adequadas à arquitetura de um sistema de software, durante todo o seu ciclo de vida, o desenvolvimento desse sistema e a evolução de longo prazo podem ser eficazes ou eficientes em qualquer sentido significativo.

Medvidovic e Taylor (2010, p. 471, tradução nossa) explicam:

> As arquiteturas de software que são particularmente úteis para famílias de sistemas são frequentemente codificadas em padrões de arquitetura, estilos arquitetônicos e arquiteturas de referência reutilizáveis e parametrizadas. Abrangendo o conjunto de decisões de projeto principais em relação a um sistema estão os princípios de design que guiam o arquiteto.

Martin (2017) comenta que a arquitetura de um sistema deve ser fácil de ser alterada e que deveria ser prática e rápida como um *stakeholder* muda de ideia sobre uma *feature*. Ele explica que a dificuldade de alteração da arquitetura deve ser proporcional apenas para o escopo das alterações e não para a forma das alterações. Caso isso não aconteça, novas *features* se tornarão cada vez mais difíceis de serem implementadas, alteradas ou estendidas nesta estrutura.

Caso o time de desenvolvimento não der importância para a arquitetura, o sistema vai se tornar mais custoso para desenvolver e será praticamente impossível fazer eventuais alterações no código.

Sistemas que não utilizam uma arquitetura bem definida são sistemas que não oficializaram nenhum um tipo de arquitetura de software. Não se pode dizer que um sistema não tem uma arquitetura, assim, como Hanmer (2013, p. 9) explica, todo sistema necessariamente tem uma arquitetura. Mesmo os sistemas que não foram criados com uma arquitetura planejada e apresentam uma estrutura desorganizada e difícil de entender. Essa arquitetura não foi oficialmente planejada, mas será uma arquitetura que pode ser classificada como algum tipo de estrutura que sustenta todo o sistema.

Medvidovic e Taylor (2010) concordam com Hanmer sobre os sistemas que não utilizam uma arquitetura com um padrão definido quando comparam uma construção de arquitetura de software com uma construção de prédio. Eles concluíram que, assim como todo prédio tem uma arquitetura, todos os sistemas de software têm uma arquitetura, pois todos refletem as opções de design escolhidas no momento do planejamento do prédio ou do sistema.

Assim como os prédios podem ter arquiteturas elegantes ou mal planejadas, os softwares também podem sofrer essa variação de arquitetura. A estrutura de um aplicativo pode ser limpa e eficaz ou desajeitada e disfuncional. Escolher a arquitetura que mais atende os requisitos do software fornece de maneira confiável arquiteturas boas e elegantes ao software, (MEDVIDOVIC; TAYLOR, 2010).

Essas arquiteturas de construção de software, definidas anteriormente, foram selecionadas e padronizadas, trazendo o conceito de Padrões de Projetos que serão explicados na seção a seguir.

2.1.2 Padrões de Projetos

No início da era dos computadores digitais, na década de 1950, Garlan e Shaw (2017, p. 2) explicam que o software era escrito em linguagens de máquina, onde as pessoas que programavam adicionavam dados e instruções manualmente a esses computadores. Após a evolução, muitos processos foram automatizados, assim como o layout de memória e a atualização de referências, e então ficou claro que certos padrões de execução eram comumente úteis. O progresso no padrão da linguagem continuou com a introdução de módulos para fornecer proteção para procedimentos relacionados e estruturas de dados e com a separação da especificação de um módulo de sua implementação.

Esses padrões de linguagem, que foram criados para solucionar os problemas citados na seção anterior, são chamados de Padrões de Projetos.

Gamma e outros (1994, p. 21), divide os padrões de projetos em padrões de criação de projeto, estrutural e comportamental. O padrão de criação *Singleton* garante que uma classe tem apenas uma instância e fornece um acesso global; O padrão estrutural *Adapter* converte a interface de uma classe em outra interface, assim o adaptador permite que as classes trabalhem juntas; O padrão comportamental *State* permite que um objeto altere seu comportamento quando seu estado interno é alterado; Já o padrão comportamental *Strategy* define uma família de algoritmos, encapsula cada um deles e torna-os permutável.

Outra vertente dos padrões de projetos são os padrões arquiteturais. Esta monografia terá foco em explicar sobre os padrões arquiteturais, que é melhor explorado na seção a seguir.

2.1.3 Padrões Arquiteturais

Conforme os sistemas de software evoluem e o tamanho e a complexidade aumentam, o problema de design aumenta também, criando complexidade além dos algoritmos e das estruturas de dados do sistema, ou seja, projetar e especificar a estrutura geral para um sistema grande e complexo se torna um novo tipo de problema (GARLAN; SHAW 2017, p. 2).

Para Wilson (2015, p. 18), o padrão arquitetural é uma solução específica para um problema comum no desenvolvimento de software. Utilizando uma linguagem comum para definir esses padrões, se torna mais fácil a discussão entre pessoas que desenvolvem software com objetivo de discutir os problemas enfrentados e as soluções propostas. Sendo transferível entre as linguagens de software, o padrão arquitetural não é específico de nenhuma linguagem de programação.

Os padrões arquiteturais são modelos para arquiteturas de software concretas. Eles especificam as propriedades estruturais de todo o sistema de um aplicativo e têm um impacto na arquitetura de seus subsistemas. A seleção de um padrão de arquitetura é, portanto, uma decisão fundamental do projeto ao desenvolver um sistema de software (BUSCHMANN et al. 1996, p. 12).

Garlan e Shaw (2017, p. 2) explicam sobre a importância da escolha certa de um padrão arquitetural para o projeto:

> Obter a arquitetura certa é muitas vezes crucial para o sucesso de um projeto de sistema de software, o errado pode levar a resultados desastrosos. O conhecimento detalhado das arquiteturas de software permite que o engenheiro faça escolhas baseadas em princípios entre as alternativas de projeto. Uma representação do sistema arquitetural é frequentemente essencial para a análise e descrição das propriedades de alto nível de um sistema complexo.

Na visão de Gamma e outros (1994), o padrão arquitetural nomeia, abstrai e identifica os principais aspectos de uma estrutura arquitetural comum que o torna útil para criar um design reutilizável. O padrão arquitetural identifica as classes e instâncias participantes, suas funções e colaborações e a distribuição de responsabilidade.

O padrão de arquitetura de software, segundo Buschmann e outros (1996, p. 8), descreve um problema de *design* recorrente específico que surge em contextos de design específicos e apresenta um esquema genérico comprovado para sua solução. O esquema de solução é especificado descrevendo seus componentes constituintes, suas responsabilidades e relacionamentos, e as maneiras pelas quais eles colaboram. Esses padrões arquiteturais expressam um esquema de organização estrutural fundamental para sistemas de software.

"Os Padrões Arquiteturais de Software abordam vários problemas da Engenharia de Software, como limitações de desempenho do Hardware do Computador, alta disponibilidade e minimização do risco do negócio" (AAHN INFOTECH 2018).

O que não pode faltar na hora de construir um padrão de arquitetura de software, segundo Gamma e outros (1994, p. 11, tradução nossa), é que: "[...] o padrão arquitetural do software deve ser específico para o problema, mas também deve ser genérico o suficiente para endereçar futuros problemas e requisitos. É preciso evitar o redesenho ou ao menos diminuí-lo".

Além de se ter um padrão arquitetural genérico para o sistema, pode-se aplicar um padrão arquitetural específico para solucionar um problema específico. Assim, como Buschmann e outros (1996, p. 16) explicam, os padrões arquiteturais podem existir em conjunto, "a maioria dos padrões arquiteturais de software levantam problemas que podem ser resolvidos com padrões menores. Padrões arquiteturais geralmente não existem isoladamente".

Wilson (2015, p. 6), também, afirma que é possível utilizar um padrão arquitetural diferente em cada estrutura do sistema, assim atendendo suas necessidades de organização e de estruturação de código. Um exemplo citado é que se pode utilizar o padrão arquitetural MVC para o *front-end* e uma estrutura API no *back-end*.

Alguns exemplos de padrões arquiteturais que podem ser utilizados para o desenvolvimento mobilem são: o *Broker Patterns, Layers Patterns, Pipes and Filters*, MVC, MVP, PM, MVVM, entre outros.

Aahn Infotech (2018) explica sobre o padrão chamado de *Broker Patterns*, é conhecido como um padrão que estrutura os sistemas distribuídos. É composto por componentes desacoplados que interagem por invocações de serviço remotos que são responsáveis pela comunicação, como solicitações de encaminhamento e transmissão de resultados e exceções.

Layers Patterns é um padrão arquitetural que ajuda a estruturar os sistemas. Nesse padrão específico, os sistemas podem ser decompostos em grupos de subtarefas, e cada subtarefa tem um nível de abstração específico. (AAHN INFOTECH 2018).

O padrão arquitetural, chamado *Pipes and Filters,* fornece uma estrutura para processar um determinado fluxo de dados. Cada etapa de processamento é encapsulada em um componente de filtro. Esses dados que serão processados são transmitidos por meio de canos. É possível recombinar filtros que permitem a construção de famílias de sistemas relacionados (AAHN INFOTECH 2018).

Para poder explicar melhor os padrões arquiteturais, Kouraklis (2016) explica as três camadas mais comuns de um sistema, conhecidas por *presentation layer*, traduzida como a camada de apresentação*, Business layer* (traduzida como a camada de negócios) e a *Data Access layer* (traduzida como a camada de dados).

A camada de apresentação representa a interface do usuário, em que serão mostrados vários tipos de dados e informações ao usuário. A camada de negócios representa a camada que contém validação de dados e regras de negócio. Já, a camada de dados é a representação dos dados, essa camada manterá esses dados no banco de dados.

A seção, a seguir, apresenta os outros quatro tipos de padrões arquiteturais selecionados: MVC, MVP, PM e MVVM.

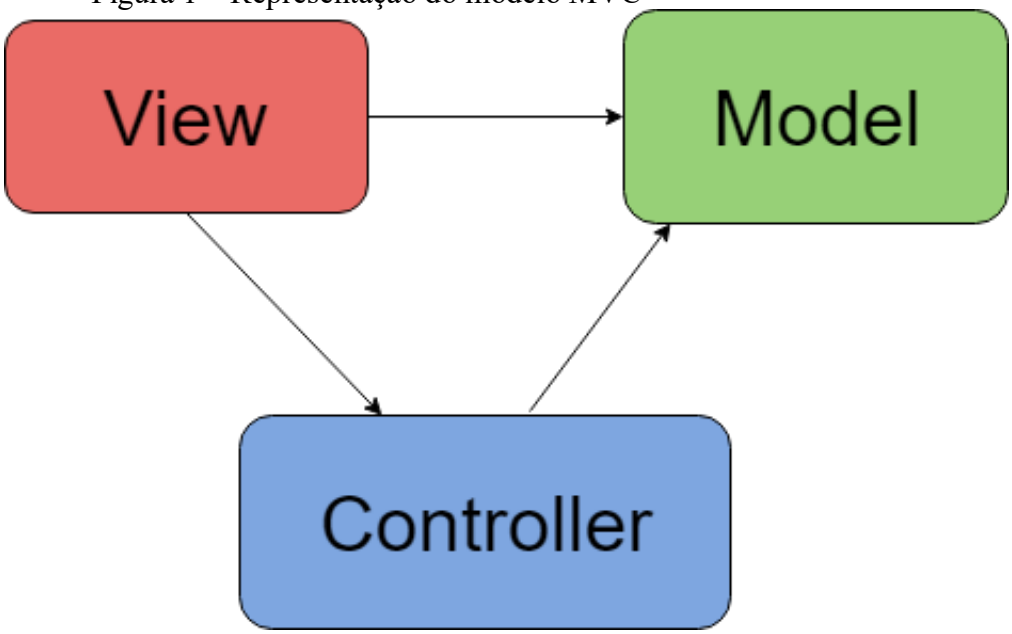
2.1.3.1 MVC

Dentre os padrões arquiteturais, pode-se citar o Model View Controller (MVC) sendo o mais conhecido entre todos. Gamma et al. (1994, p. 14) explicam que esse padrão

consiste em três tipos de objetos: *Model, View* e *Controller,* conforme é apresentado na Figura 1. Em uma visão mais superficial, o *Model* é o objeto da aplicação, a *View* é a representação da tela e o *Controller* define como a interface reagirá com a inserção do usuário.

O padrão Model-View-Controller (MVC) divide a estrutura da aplicação em três componentes. Ele explica que o Model é dividido entre as funcionalidades principais e os dados, a View mostra as informações aos usuários e os Controllers manipulam a entrada de dados do usuário. A junção de Views e Controllers compõem a interface do usuário. O que garante a consistência entre a interface do usuário e o modelo é um mecanismo específico de propagação de mudança (AAHN INFOTECH 2018).

Figura 1 – Representação do modelo MVC



Fonte: Autoria própria

A seguir, Wilson (2015, p. 62) explica mais detalhadamente os três componentes do padrão MVC. O primeiro componente apresentado é o model. Segundo ele, os models são as representações de dados, eles manterão esses dados no banco de dados da aplicação. Na linguagem de programação, a camada Model é composta por classes Model que representam os dados a serem manipulados pelo model. Um exemplo técnico de uma classe Model é a simples implementação de uma classe chamada usuário com seus atributos.

Um ponto de atenção trazido por Wilson (2015, p. 68) “o código responsável por persistir os dados para o banco de dados é uma parte da camada model, mas não deve ser parte da nossa classe model”. Ainda afirma que existe uma diferença entre a camada Model e a classe model. A classe model, que representa os dados de um Model específico, faz parte da camada

model, junto com o código responsável pela persistência dos dados, que de fato persiste os dados no banco de dados.

Na figura 1 é apresentado o modelo MVC. A classe Model pode ser chamada de *Entities* ou Entidade, pois "[...] representa simplesmente o estado das coisas que têm identidades e atributos únicos para essa identidade. Como por exemplo: usuário, produto, processos, etc. Todos esses exemplos podem ser entidades" (WILSON 2015, p. 68).

Outro ponto de atenção explicado por Wilson (2015, p. 68) diz que a camada Model não pode ser referida como sendo a camada de persistência de dados. A camada de persistência de dados é a camada responsável pelos códigos que persistem os dados das entidades.

Já, a View é o que contém os componentes visuais. São os arquivos que contêm o código com a interface a ser mostrado ao usuário. Também, é responsável pela interação do usuário com a aplicação, como, por exemplo, a interação do usuário com botões, links, *input* e outros. Essas ações podem ser tratadas inteiramente na camada da View, ou, caso precise, pode enviar essas ações ao servidor para carregar outros dados e Views (WILSON 2015, p. 63).

A camada View, sendo bem objetiva, deve mostrar os dados para o usuário e não processar nada além de um loop para mostrar os dados em uma tabela. Não é feito nenhuma pesquisa diretamente no banco, ela apenas recebe, manipula e mostra os dados ao usuário. A View não deve ter nenhum tipo de lógica de negócios (WILSON 2015, p. 63).

Nessa linha, é possível um Model ser representado por várias Views. Um exemplo prático é um Model de usuário ter uma View que mostra a listagem de usuários e outra View que cria um usuário. Essas duas Views estarão ligadas com o mesmo Model, pois tratam e mostram o mesmo tipo de dado (GAMMA et al 1994, p. 14).

O componente Controller é o responsável por implementar a regra do negócio do software. Na prática, é responsável por interpretar e responder a requisição do usuário, ou seja, ele analisa uma solicitação do usuário e determina o que será feito a partir disso (WILSON 2015, p. 64).

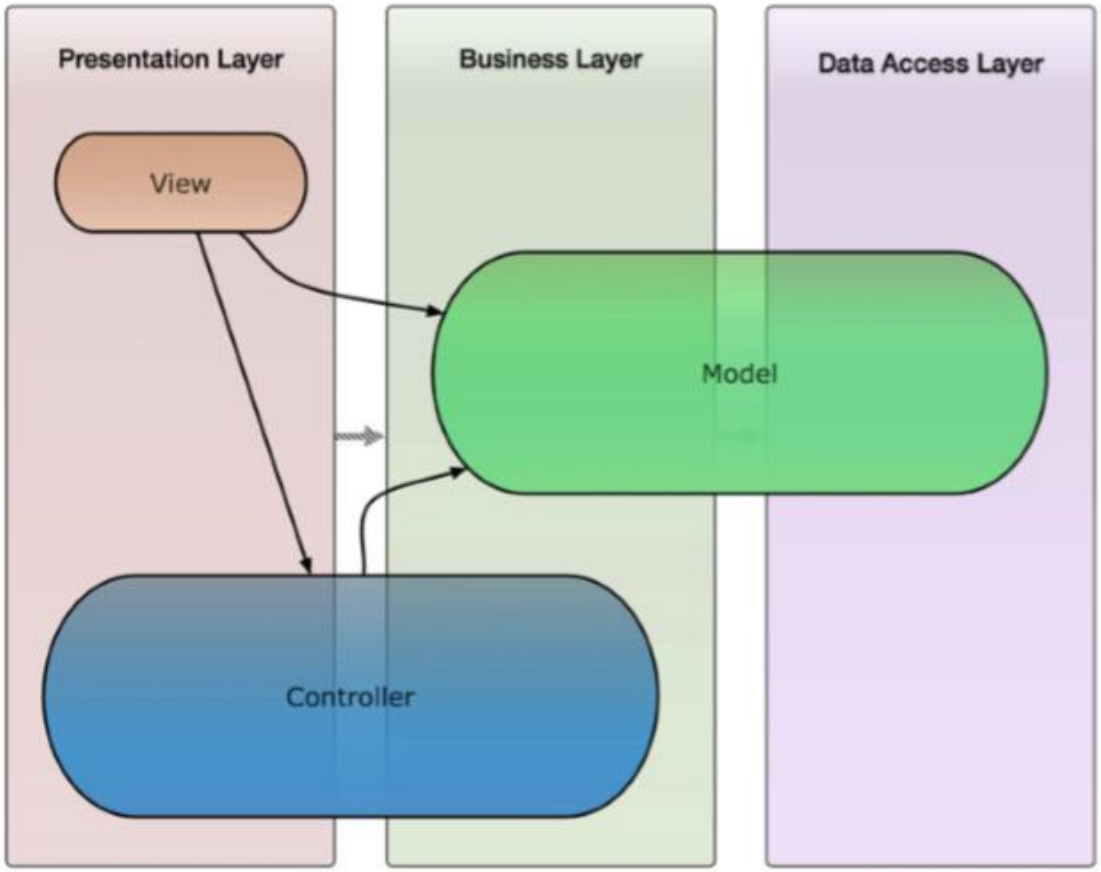
Com a requisição do usuário, o Controller carrega as informações do Model correspondente à requisição e envia as informações para a View, onde será interpretado e mostrado na tela. O fluxo de comunicação da View com o Controller também é válido, pois recebe as informações de uma View, como algum *input* feito pelo usuário e traduz e envia esses dados para um Model específico, que irá persistir no banco da aplicação (WILSON 2015, p. 64).

Wilson (2015, p. 65, tradução nossa) resume as ações dos três componentes dizendo que "os Controllers apenas se importam em responder às requisições. Ele usa o a camada do

Model para recuperar os dados e, assim, passá-los para a camada View para o processamento e exibição dos dados".

A seguir, é mostrada a relação das três camadas mais comuns de um sistema, explicadas na seção anterior, com o modelo MVC. A Figura 2, a seguir, mostra a separação dos três componentes, de acordo com a visão de Kouraklis (2016, p. 15).

Figura 1 – Relação entre as três camadas mais comuns de um sistema e o modelo MVC



Fonte: Kouraklis (2016, p. 15).

A arquitetura desacopla esses três objetos para aumentar a flexibilidade e tornar possível a reutilização de código entre eles, permitindo ligar múltiplas Views em somente um Model. A comunicação do Model com as Views é feita quando o valor do Model é alterado, e a comunicação da View com o Model é feita, quando o View precisa acessar os dados da Model (GAMMA et al 1994, p. 14).

2.1.3.2 MVP

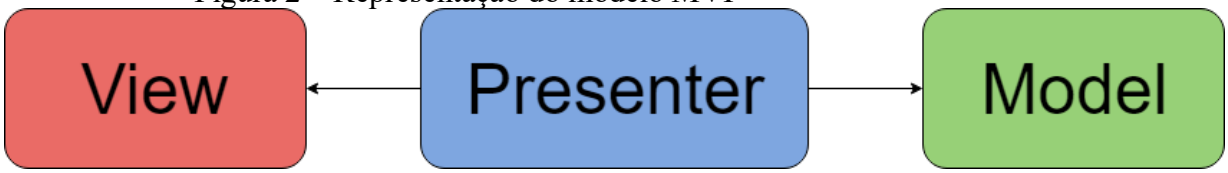
Conforme Kouraklis (2016, p. 16), esse modelo foi inspirado no modelo MVC. A diferença é que no modelo MVP foi substituído o Controller pelo Presenter, como mostrado na figura 3. As responsabilidades, deveres e capacidades do componente Presenter mudaram, assim, existe uma clara separação entre o View e o Model, sendo que a sincronização é executada pelo Presenter.

A representação do padrão Model-View-Presenter (MVP) pode ser explicada por Mihailesc (p. 2). As estruturas utilizadas nesse modelo são: a View que é o componente que se vê na tela do dispositivo; O model, que mostra os dados do sistema; O Presenter que liga a View e o Model juntos. O funcionamento da View depende de um Presenter para popular o Model com os dados que um usuário adiciona. Para manter a sincronicidade, o modelo frequentemente atualiza os dados das Views.

Para Hall (2010), o componente Presenter se comporta como um mediador entre a View e o Model, pois recebe da View uma requisição de entrada de dados do usuário, modifica os dados caso necessário, e encaminha as alterações para o Model em resposta.

Na compreensão de Hall (2010), o Presenter é capaz de consultar a View para buscar dados, e para isso, o Presenter deve ter conhecimento da View e do Model. Isso indica uma dependência em duas direções entre a View e o Presenter.

Figura 2 – Representação do modelo MVP



Fonte: Autoria própria.

Nesse modelo, Kouraklis (2016, p. 16) afirma que a View não deve estar ciente sobre Model, o inverso também se aplica, assim, isso é chamado comumente de *Passive View*. Também afirma que o Presenter recebe entradas do usuário a partir do componente View, manipula o mapeamento entre o View e o Model e, ainda, realiza lógica de negócios complexa.

A relação do modelo MVP com as três camadas mais comuns do sistema é muito parecida com o modelo MVC, porém trocando o Controller pelo Presenter, explica Kouraklis (2016, p. 17). Os componentes View e Model terão a mesma responsabilidade que no modelo MVC, porém o Presenter é responsável pela camada de negócios, pois esse componente é responsável por muitas validações e também por manter a maioria dos estados das Views.

Kouraklis (2016, p. 17) afirma que esse modelo oferece uma testabilidade melhor, se comparada com o modelo MVC, já que nos testes o Model e a View podem ser substituídos por unidades *mock* ou por diferentes implementações desacopladas.

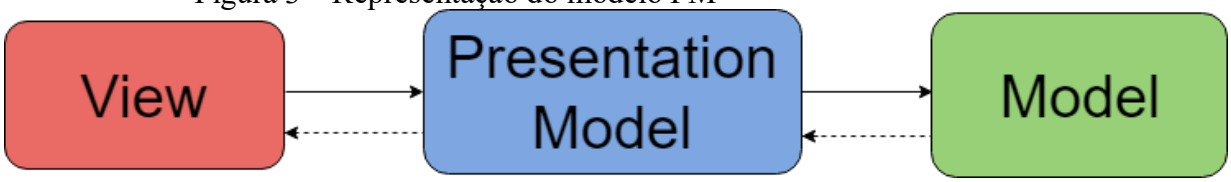
Além desses benefícios, para que esse modelo seja implementado e que a interface do usuário se torne mais sofisticada, há a necessidade de se escrever mais código. Porém, escrever mais código significa mais oportunidades para problemas técnicos também significa um aumento no esforço necessário para manter o código (KOURAKLIS 2016, p. 17).

2.1.3.3 PM

Outro padrão arquitetural explicado por Pan et al. (2011) é o Presentation Model (PM) diferentemente do padrão arquitetural MVC e MVP, esse modelo armazena o estado da View e não disponibiliza um mecanismo de ligação entre a apresentação da interface do usuário e a lógica de negócios, embora desvincule as duas partes.

Como afirma a Microsoft (2010), esse modelo utiliza 3 componentes diferentes, o componente View, o Presentation Model e o Model. Na figura 4, é mostrado a representação desse padrão arquitetural. Os componentes View e Model se comportam igualmente como nos modelos anteriores. A partir do modelo PM, foi criada uma derivação, chamada de MVVM.

Figura 3 – Representação do modelo PM



Fonte: Autoria própria.

Segundo Fowler (2018), o Presentation Model retira o estado e o comportamento da View para adicionar em uma classe de Model que faz parte do componente Presentation Model. A View pode armazenar todo o seu estado no Presentation Model ou sincronizar seu estado com o Presentation Model frequentemente.

Na visão de Fowler (2018), esse modelo "representa o estado e o comportamento da apresentação, independentemente dos controles da interface gráfica do usuário usados na interface".

É possível considerar o Presentation Model como uma abstração da View que não depende de uma estrutura da interface gráfica do usuário específica. Várias Views podem utilizar o mesmo Presentation Model, porém cada View deve exigir apenas um componente de Presentation Model. Um Presentation Model pode conter uma ou mais instâncias do Presentation Model filho, mas cada filho terá apenas um componente de Presentation Model (FOWLER, 2018).

Microsoft (2010) explica quais são as responsabilidades de cada componente:

> Esse modelo separa as responsabilidades do código visual e as responsabilidades do estado e comportamento da interface do usuário para classes de View e Presentation Model. A classe de View gerencia os controles na interface do usuário e a classe de Presentation Model atua como a fachada no modelo com comportamento e estado específicos da UI, encapsulando o acesso ao modelo e fornecendo a interface pública que é fácil de ser consumida da View.

Para que esse modelo funcione, o Presentation Model terá campos de dados para todas as informações dinâmicas da View, ou seja, conteúdo dos Controllers, se os Controllers estão habilitados ou não, entre outros. Em geral, o Presentation Model mantém qualquer estado que possa mudar durante a interação do usuário para então sincronizar seus dados. Portanto, se um campo estiver sempre ativado, não haverá dados extras para seu estado no Presentation Model (MICROSOFT, 2010).

Conforme Pan et al. (2011, p. 2), esse modelo atualiza frequentemente o componente View para que as informações estejam sempre sincronizadas. O código que sincroniza essas informações é representado como uma classe *Presentation Model*.

A Microsoft (2010) explica a comunicação entre os componentes, o Model não conhece o Presentation Model. Se o Model sofre alterações por outros componentes que não sejam o Presentation Model, então o Model deverá avisar o Presentation Model.

Na visão de Fowler (2018, tradução nossa), o funcionamento desse modelo:

> a essência de um Presentation Model é de uma classe totalmente independente que representa todos os dados e comportamentos da janela da interface do usuário, mas sem qualquer um dos controles usados para processar essa interface do usuário na tela. Uma View então simplesmente projeta o estado do modelo de apresentação no vidro.

A Microsoft (2010) afirma que esse padrão se torna mais útil para os seguintes cenários: maximizar a quantidade de código que pode ser testado com testes automatizados, pois Views são difíceis de testar; Compartilhar um trecho de código entre as Views que contêm o mesmo comportamento, ou, quando é necessário, separar a lógica de negócios da lógica de interface do usuário para tornar o código mais fácil de entender e manter.

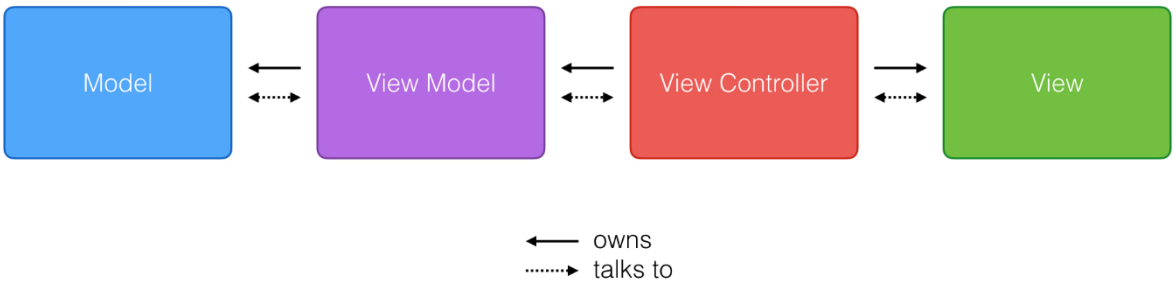
2.1.3.4 MVVM

Pan e outros (2011) definem o padrão arquitetural Model-View-ViewModel (MVVM), sendo representado por três entidades chamadas Model, View e ViewModel. O Model e a View têm a mesma definição que os modelos anteriores. Já, o ViewModel pode guardar a lógica de negócios e o estado da interface do usuário.

Na concepção de Mihailesc (2010, p. 2), em 2005, John Grossman, um dos arquitetos da Microsoft Silverlight, foi quem revelou o modelo MVVM. Esse modelo é muito parecido com o modelo PM apresentado por Martin Fowler, em que a única diferença é que no modelo MVVM o componente ViewModel não precisa da implementação de uma referência para uma View.

Na visão de Jacobs (2017, p. 3), apesar do nome, esse modelo arquitetural é composto por quatro componentes: Model, View, ViewModel e o ViewController, como é mostrado na figura 5.

Figura 4 – Representação do modelo MVVM contendo o componente View Controller



Fonte: Jacobs (2017).

Mas para Pan e outros (2011) e Kouraklis (2016, p. 18), esse modelo tem apenas três componentes, sendo eles: Model, ViewModel e View, assim como é mostrado ao longo dessa seção. Na figura 6, é apresentado o modelo MVVM com esses três componentes.

Figura 5 – Representação do modelo MVVM sem o componente View Controller

Fonte: Autoria própria.

Nesta monografia, quando referenciar ao modelo MVVM, é considerado apenas o modelo representado com as três camadas: View, ViewModel e Model.

Para Kouraklis (2016, p. 18) o componente ViewModel irá substituir os componentes Presenter e Controller dos outros dois modelos visto anteriormente. O Model e a View continuarão responsáveis pela mesma área que no modelo MVP.

A estruturação do modelo MVVM, segundo Pan e outros (2011, p. 4), pode ser dividida em cima das três camadas mais comuns do sistema, citadas anteriormente. Afirmam que o componente View é responsável pela camada de apresentação, pois seu objetivo é cuidar

inteiramente da aparência da interface do usuário e do recebimento de entrada de dados do usuário. Este componente apenas exibe os dados obtidos da camada de negócios e não processa ou armazena dados.

O componente ViewModel, para Pan e outros (2011, p. 4), é responsável pela camada de negócios, em que ela se compromete a manipular a lógica de negócios da aplicação, armazenando e processando o estado dos widgets da interface do usuário. Já o componente Model é responsável pela camada de dados da aplicação, em que serão mantidos e persistidos os dados da aplicação. O seu comportamento se repete como no modelo MVP e MVC.

Para Kouraklis (2016, p. 18):

> O ViewModel representa a maneira como a View deve ser (estado da View) e espera-se que ele se comporte com as interações do usuário (lógica de visualização). É o Model da View no sentido em que descreve um conjunto de princípios e estruturas que apresentam dados específicos conforme recuperados pelo Model. O ViewModel lida com a comunicação entre o View e o Model passando todos os dados necessários da View para o Model em um formulário que o Model possa digerir. A validação é executada no componente ViewModel.

A comunicação entre os componentes, segundo Kouraklis (2016, p. 18), se comporta da seguinte maneira: a View está ciente e se comunica com o ViewModel e o ViewModel não está ciente da View. Essa consciência unidirecional justifica a apresentação linear do padrão na figura 6.

Pan e outros (2011, p. 2) concluem que se pode salvar o mapeamento da View, fazendo com que o ViewModel possa criar a lógica de negócios e que se possa testar o estado da View sem utilizar ferramentas de interface. Essa separação permite que o ViewModel possa interagir com a View somente quando precisar sincronizar dados e estados.

Assim, Pan e outros (2011, p. 3, tradução nossa) afirmam: “Através do comando fornecido pelo ViewModel, a camada de negócios permite que a View da camada de apresentação possa interagir com os dados do Modelo localizado na camada de dados”.

Pan e outros (2011, p. 3) ainda concluem que os três componentes do modelo MVVM precisam sofrer interação apenas quando é preciso transferir dados de um componente para outro.

Com a separação dos componentes entre ViewModel e Model, cada um dos componentes pode ser testado com teste automatizado unitário. Essa estrutura permite que o código seja testado isoladamente sem a confusão de uma interface de usuário. Isso resulta em uma aplicação com menos problemas técnicos, que é mais fácil e rápido de compreender, manter e criar o código (PAN et al., 2011).

2.2 O PRINCÍPIO SOLID

Na visão de Martin (2017), ainda utilizando a analogia de arquitetura de software com arquitetura de construção de prédios, se os tijolos de um prédio não estão bem feitos, a arquitetura desse prédio não vai importar muito, pois uma arquitetura boa não é feita apenas com tijolos bem feitos. Em outra visão, é possível construir uma arquitetura ruim para um prédio que contém tijolos de boa qualidade. É isso que o princípio SOLID defende.

Para Martin (2017, p. 72), "o princípio SOLID nos conta como organizar nossas funções e estruturas de dados em classes, e como essas classes devem se interconectar".

O objetivo desse princípio é a criação de uma estrutura de software que tolera mudanças, são fáceis de entender e são a base de componentes que pode ser usada em vários sistemas de software (MARTIN 2017).

Alguns sintomas de um sistema com uma arquitetura pobre ou ruim são: rigidez do código, que faz a arquitetura ser difícil de alterar; fragilidade, mostra que a arquitetura é fácil de quebrar; Imobilidade, mostra que a arquitetura é difícil de se reutilizar; Viscosidade, mostra que é difícil de fazer a coisa certa e, entre outros sintomas citados por Martin (2002). Assim, os princípios SOLID ajudam a combater esses sintomas para criar um sistema bem arquitetado.

Quem começou a reunir os princípios SOLID foi Robert C. Martin, segundo Martin (2017). Nas próximas seções, é mostrado cada um dos pontos que o princípio SOLID reúne.

2.2.1 SRP: The Single Responsibility Principle

Esse princípio, segundo Wilson (2015), que significa o princípio de única responsabilidade, afirma que uma classe ou um objeto deve ter apenas um propósito.

Um exemplo desse princípio é citado por Wilson (2015, p. 78), quando fala que a melhor maneira de criar de um Controller, é garantir que realmente seja criado o mínimo de

código possível para um Controller. Cada Controller e cada ação desse Controller devem seguir esse princípio.

Se a classe tiver mais de uma responsabilidade, então, essas responsabilidades se tornarão duplicadas. Essa duplicação cria uma fragilidade na arquitetura sistema que pode fazer com que o sistema quebre inesperadamente (MARTIN, 2002).

O melhor design para isso, na visão de Martin (2002), seria separar as duas responsabilidades em duas classes diferentes.

2.2.2 OCP: The Open-Closed Principle

Esse segundo princípio é explicado por Martin (2002, p. 99) como: "Entidades de software (classes, módulos, funções, etc.) que devem estar abertos para expansão, mas fechados para modificação".

Quando é necessário fazer uma alteração no sistema e essa mudança resulta em alterações em cascata em módulos dependentes, então a arquitetura apresentará o sintoma de rigidez. O princípio Open-Closed fará com que o sistema não tenha mais essa rigidez (MARTIN 2002).

Para Wilson (2015), esse princípio significa que as pessoas que desenvolvem esse sistema não serão permitidas ou encorajadas a modificar essa classe já existente, mas, ao invés, serão encorajadas a encontrar outras formas de estender essa classe para assim poder adicionar novos códigos.

Martin (2017, p. 88, tradução nossa) conclui que o objetivo desse princípio

> [...] é tornar o sistema fácil de estender sem incorrer em um alto impacto de mudança. Essa meta é realizada particionando o sistema em componentes e organizando esses componentes em uma hierarquia de dependências que protege componentes de nível superior contra alterações em componentes de nível inferior.

A vantagem desse princípio, segundo Wilson (2015, p. 44), é a limitação da modificação de código existente. Quanto mais é adicionado código ao sistema, mais é provável que será inserido um código que será um débito técnico. Com esse princípio em mente, na hora de adicionar mais código, o escopo para potenciais débitos técnicos ficará limitado a um código específico.

2.2.3 LSP: The Liskov Substitution Principle

Esse princípio foi escrito primeiramente por Barbara Liskov em 1988. Para Barbara Liskov (1998), citado por Martin (2002, p. 136, tradução nossa):

> [...] se para cada objeto O1 de tipo S existe um objeto O2 de tipo T de tal modo que para todos os programas P definem em termos de T, o comportamento de P é inalterado quando O1 é substituído por O2 e então S é um subtipo de T.

Para Wilson (2015, p. 44, tradução nossa), "o LSP indica que todas as implementações concretas devem ser intercambiáveis sem afetar o comportamento do programa".

Na visão de Martin (2002, p. 136, tradução nossa), "os subtipos devem ser adequados para seus tipos de base". Esse princípio resolve a fragilidade do sistema, que é um dos sintomas citados anteriormente e fortalece a reutilização e a robustez do sistema.

Um sistema atenderá esse princípio quando o comportamento de uma classe não depende de forma alguma, de qual dos dois subtipos ele usa. Uma simples violação da substituibilidade pode fazer com que a arquitetura do sistema seja poluída com uma quantidade significativa de mecanismos extras (MARTIN 2017).

Wilson (2015) afirma que esse princípio é a solução para que o código seja facilmente refatorável. Ainda, nos permite modificar o comportamento fornecendo uma instância diferente de uma interface e sem realmente modificar o código do programa. Qualquer código que dependa do código dessa interface continuará a funcionar corretamente, independentemente da implementação.

2.2.4 ISP: The Interface Segregation Principle

Esse princípio, segundo Martin (2002), lida com interfaces implementadas com muito código. Classes que têm esse tipo de interfaces são classes cujas interfaces não são coesas, ou seja, essa interface poderia ser dividida em grupos de métodos, em que cada grupo serviria

um conjunto diferente de clientes. Ainda dita que os clientes devem conhecer uma classe abstrata que tem interfaces coesas.

Wilson (2015) explica que esse princípio resolverá o problema das interfaces que definem muitas assinaturas de métodos. Assim, esse princípio é brevemente ligado com o primeiro princípio, O SRP, pois, para resolver esse problema, é necessário que a interface tenha apenas uma responsabilidade.

Para Martin (2002, p. 162, tradução nossa), esse princípio explica que "clientes não devem ser forçados a depender de métodos que eles não usam". Quando isso acontece, esses clientes estarão sujeitos a mudança por causa desses métodos, ou seja, serão dependentes desses métodos.

Na visão de Wilson (2015), caso a interface não tenha apenas uma responsabilidade, os clientes terão excesso de métodos com a qual o código do cliente também deve ser associado.

O objetivo desse princípio, segundo Wilson (2015), é fornecer códigos desacoplados. Assim, todo código do cliente utilizado na implementação de uma interface é acoplado a todos os métodos dessa interface, mesmo caso eles não sejam utilizados.

A conclusão de Martin (2017) sobre esse princípio é que, depender de algo que carrega bagagem, ou seja, algo com muitas coisas que você não precisa, pode causar alguns problemas.

2.2.5 DIP: The Dependency Inversion Principle

Esse princípio, segundo Martin (2002, p. 164, tradução nossa), são "módulos de alto nível não devem depender de módulos de baixo nível". Também é uma definição desse princípio, quando Martin (2002) diz que "ambos devem depender de abstrações. Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações".

Para Wilson (2015), os módulos de níveis mais baixos são considerados coisas que não mudam com frequência, já os níveis mais altos são considerados coisas que mudam frequentemente. Esse princípio defende que as camadas de níveis mais baixos não devem depender das camadas de nível mais alto.

Para criar código desacoplado, é necessário chegar a um ponto em que a dependência flua apenas para dentro. As camadas de alto nível, que são coisas que mudam com

frequência, só devem depender de níveis mais baixos, ou seja, coisas que sofrem bastantes alterações (WILSON 2015).

Para Wilson (2015), para criar código desacoplado, é necessário chegar a um ponto em que a dependência flua apenas para dentro. As camadas de alto nível só devem depender de níveis mais baixos e os níveis mais baixos não devem depender das camadas de nível mais alto.

Esse princípio, segundo Wilson (2015) facilita que alguma mudança seja feita no futuro, e essa mudança terá o menor impacto possível em cima do código já existente.

2.3 DESENVOLVIMENTO MOBILE

Pode-se perceber a rápida evolução dos dispositivos nos últimos anos. Pilar (2013, p. 17) explica que há dados que, em meados dos anos 90, surgiu o primeiro dispositivo móvel chamado PDA (Personal Digital Assistant). Esse dispositivo tinha como objetivo possuir mais funcionalidades e ser mais portátil que os antigos computadores, mesmo possuindo capacidade inferior de processamento e armazenamento. Com a evolução do PDA, foram adicionadas funcionalidades como acesso à internet móvel, câmera integrada e sincronização com desktops, entre outros. Assim se criou os celulares e *smartphones*.

O desenvolvimento de softwares para dispositivos móveis pode ser um pouco complexo por ter que responder às características, tais como: necessidade de resposta em tempo real, memória limitada para se tornar viável à mobilidade do dispositivo, canais de entrada e saída limitados, além de cada dispositivo ter uma forte relação com a dependência de hardware e diferentes processadores (PILAR, 2013).

Existem vários tipos de ferramentas que ajudam no desenvolvimento de código para dispositivos mobile. Algumas dessas ferramentas utilizadas são: o Android Studio que é focado para o desenvolvimento mobile com sistema operacional Android; O Xcode que é focado para o desenvolvimento mobile com sistema operacional iOS, essa ferramenta também fornece um simulador mobile para teste de interface.

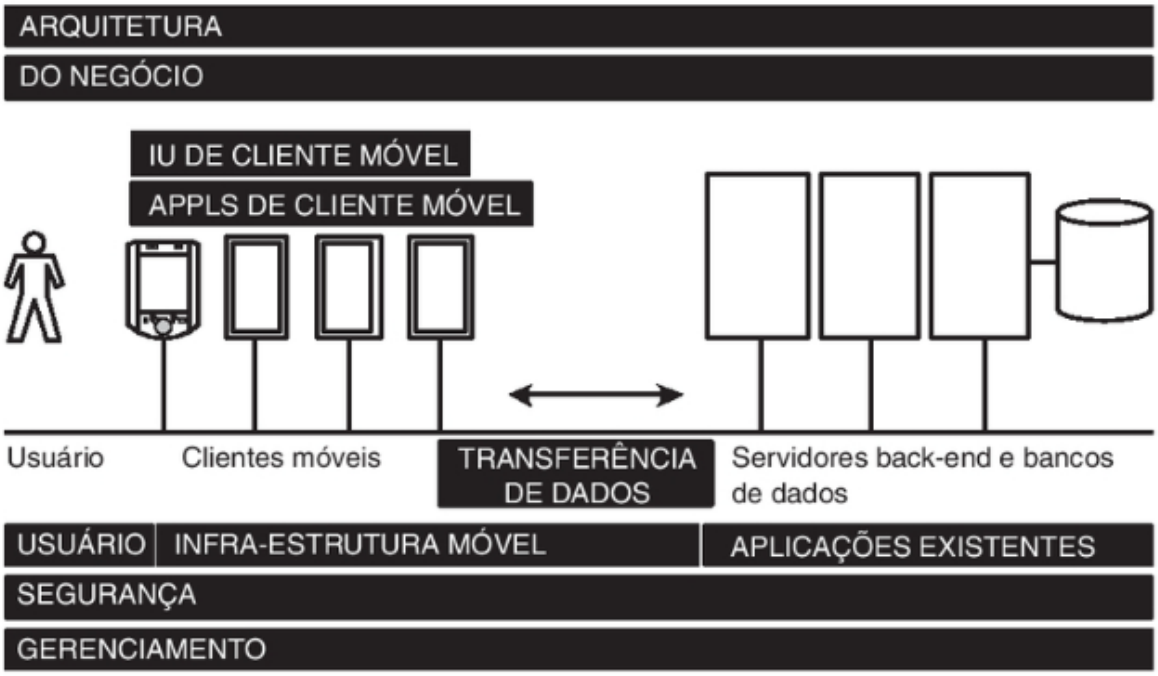
A seguir, são explicados as características de desenvolvimento mobile, como a comunicação de sua arquitetura, tipos de clientes e a hospedagem de páginas Web.

2.3.1 Características do desenvolvimento mobile

A arquitetura específica para dispositivos mobile ou dispositivos móveis é chamada pelo Celepar (2011, p. 11) por Arquiteturas Nativas. É conhecida desse modo, pois nativamente induz que o processamento será direto e dentro do dispositivo móvel, sem passar informação ou sem se comunicar com o lado externo. Tendo uma arquitetura nativa, é possível acessar todos os recursos dos aparelhos, criando mais possibilidades para o desenvolvimento de aplicações mais ricas. Alguns exemplos desses recursos seria o GPS, a câmera, lista de contatos e acelerômetro, entre outros.

Lee (2005, p. 23, tradução nossa) explica que, em geral, os dispositivos móveis operam em um de três modos, como mostrados na figura 7. “Um dispositivo móvel pode estar sempre conectado a um sistema back-end. Alternativamente, um dispositivo móvel pode operar inteiramente sem conexão a um sistema back-end. Os primeiros dois modos são de importância crucial”.

Figura 6 – Representação da solução mobile



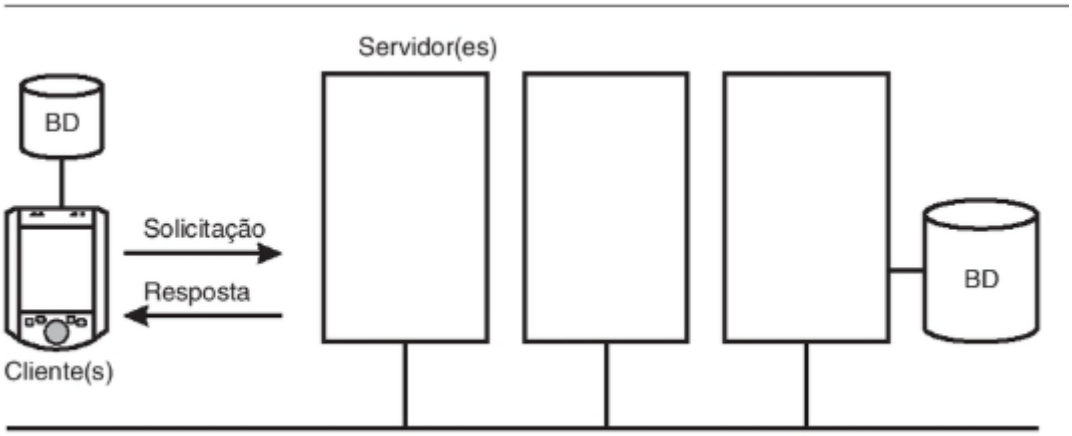
Fonte: Lee (2005, p. 7).

Segundo Lee (2005, p. 23), a arquitetura de comunicação das aplicações para dispositivos mobile é feita pela arquitetura cliente-servidor que é explicada no tópico a seguir.

2.3.1.1 Cliente-Servidor

Segundo Lee (2005, p. 23, tradução nossa), “as arquiteturas de aplicação são comumente modeladas em termos de uma arquitetura cliente-servidor, em que um ou mais dispositivos clientes solicitam informações a um servidor. O servidor, em geral, responde com as informações solicitadas”. A estrutura dessa arquitetura é representada na figura 8.

Figura 7 – Arquitetura mobile do tipo cliente-servidor



Fonte: Lee (2005, p. 24).

Para entender a relação entre cliente e servidor, a seção seguinte mostra a definição, os tipos e exemplos de clientes.

2.3.1.2 Clientes

A definição de cliente para Vasanta (2018) é um aplicativo que é executado no dispositivo e depende do servidor para executar algumas operações. Já a definição de servidor é um computador ou dispositivo portátil dedicado em uma rede que gerencia recursos de rede e executa tarefas do servidor.

Como existem muitos tipos de clientes, como, por exemplo, os celulares, tablets e PCs, Lee (p. 26, 2005) divide-os em *thin client* e os *thick client*. Os *thin clients* são os clientes que não possuem código personalizado da aplicação, dependem do servidor para que a aplicação funcione corretamente e não dependem do sistema operacional. Em geral, os *thin clients* utilizam os navegadores Web para exibir as páginas de conteúdo da aplicação desenvolvida.

Para Vasanta (2018), o *thin client* "[...] se comunica com um servidor de processamento central, o que significa que há pouco hardware e software instalados no dispositivo do usuário". O *thin client* é caracterizado por não precisar de que o software da aplicação seja instalado no dispositivo. "Isso permite que os sistemas de todos os usuários finais

sejam gerenciados centralmente, e o software implantado em um local de servidor central, em vez de instalado em cada dispositivo individual".

Os *thick clients* que Lee (2005, p. 27) apresenta, em geral, possuem, no mínimo, uma e, no máximo, três camadas de código da aplicação. Por um certo período, podem operar independentemente de um servidor. Em geral, esses clientes são mais úteis em situações em que não há comunicação permanentemente entre o cliente e o servidor, permitindo que o usuário continue trabalhando mesmo se estiver sem contato com o servidor. A desvantagem é que dependem muito do sistema operacional e do tipo de dispositivo móvel utilizado, com isso, é mais difícil de distribuir e liberar o código.

Já, para Vasanta (2018), os *thick clients* fornecerão mais recursos, gráficos e opções aos usuários, aumentando a personalização dos aplicativos. Ele não depende de um servidor de processamento central, pois isso é feito localmente no dispositivo do usuário, e o servidor é acessado apenas para ações de armazenamento.

Vasanta (2018) ainda explica que:

> É por esse motivo, os *thick client* geralmente não são adequados para ambientes públicos. Para manter um *thick client*, o provedor de aplicativos deve liberar patches regulares para atualizações e a atualização deve ser feita para cada dispositivo que um usuário final tiver, em vez de apenas manter os aplicativos no servidor.

As vantagens dos *thin clients* sobre os *thick clients* são grandes. Lee (2005, p. 27) cita a manutenção e suporte facilitado. Porém, ele também cita os problemas que os *thin clients* apresentam, como, o dever de estar em comunicação constante com o servidor, pois é sua fonte para atualizar seus dados.

2.3.1.3 Hospedagem de páginas Web

Também é possível acessar páginas Web a partir de um dispositivo móvel, explica Lee (2005, p. 27), mesmo quando esse dispositivo está conectado periodicamente com a rede e aos sistemas back-end. Para essa comunicação funcionar, é necessário ter um servidor Web de pequeno porte nos dispositivos móveis.

De acordo com Lee (2005, p. 27, tradução nossa):

> A Microsoft lançou no mercado um servidor HTTP que roda em um Pocket PC apenas com essa finalidade. Os dados inseridos por um usuário em uma página Web são disponibilizados pelo servidor HTTP e armazenados em um banco de dados local até que possam ser carregados para um servidor, quando a conectividade tiver sido restaurada.

Segundo Lee (2005, p. 27), a grande diferença entre a hospedagem de páginas Web e o *thick client* é que a camada de apresentação exibe e utiliza diretamente as páginas Web ao invés da resposta que o *thick client* recebe.

2.4 TESTES MOBILE

Este capítulo explica o que são os testes, quais as vantagens de se utilizar em um sistema e explica também os testes criados em ambiente *mobile*.

2.4.1 O que são testes

Para Martin (2015), criar testes "é a capacidade de criar unidades de programação falsificáveis que tornam valiosa a programação estruturada hoje".

Segundo Martin (2002), criar um teste unitário é mais um ato de design e de documentação de código do que um ato de verificação do código. A primeira vantagem de criar testes automatizados para o sistema é que todas as funções ou métodos terão uma validação própria em que será testada a sua funcionalidade. Essa suíte de teste serve como um batente para os futuros códigos. Esses testes avisam quando uma funcionalidade existente para de funcionar indevidamente.

Outro ponto importante sobre a criação de testes no sistema que Martin (2002) traz é que criando os testes antes de implementar o código do sistema oferece um ponto de vista diferente, pois, é necessário pensar inteiramente no sistema antes de implementar suas funções.

2.4.2 Testes automatizados mobile

Criar uma arquitetura para um sistema de software é um passo importante para o desenvolvimento do sistema, Wilson (2015) explica que, ao descobrir que a arquitetura escolhida para o sistema não é a escolha correta, ela acabará trazendo muitos problemas e dificuldades mais tarde. Pode tornar mais difícil testar o software, ou até impossibilitar que o software seja testado.

É fundamental o código do software ter uma cobertura de testes automatizados, assim, como Wilson (2015, p. 3) defende essa visão, quando fala que “escrever testes é supremamente importante para ter uma estável e duradoura aplicação de software”.

Como pessoas que programam estão continuamente produzindo código, elas estão continuamente evoluindo a aplicação, substituindo códigos funcionais antigos que agora podem estar defasados por códigos novos e mais performáticos. Essa mudança progressiva no código torna a suíte de testes indispensáveis para o sistema, pois os testes dão a garantia de que o código antigo foi substituído com sucesso pelo código novo, sem sofrer alterações na regra de negócio do sistema. (WILSON 2015, p. 70).

Ter essas suítes de teste já implementadas no sistema ajuda muito no processo de refatoração de código e de testes manuais, segundo Wilson (2015, p. 70, tradução nossa). Ele afirma que “pode ser literalmente a diferença entre prevenir rapidamente alguns bugs ou desperdiçar centenas de horas, tentando consertar esses bugs”.

Segundo Wilson (2015, p. 70), o teste automatizado se encaixa para qualquer tipo de sistema, seja ele um sistema grande e complexo ou simples e pequeno. O teste previne a regressão de erros no software, por isso é um fator muito importante para qualquer sistema. Além disso, os testes automatizados ajudam o processo de refatoração e de teste de código, sendo boas práticas desenvolver uma suíte de testes mais robustos e disponíveis para todas as pessoas.

3 MÉTODO

Este capítulo retrata sobre a caracterização da pesquisa utilizada, a abordagem do problema, a caracterização dos objetivos, o procedimento técnico, assim como as etapas metodológicas e as delimitações desta monografia.

3.1 CARACTERIZAÇÃO DO TIPO DE PESQUISA

A caracterização da pesquisa desta monografia é considerada uma pesquisa aplicada.

Segundo Silva (2005), esta monografia se classifica como uma pesquisa aplicada pois ela possui o objetivo de gerar conhecimentos para utilizá-los de forma prática na solução de um problema específico.

Quanto à abordagem do problema, é utilizada a abordagem qualitativa.

Na afirmação de Silva (2005), a abordagem da pesquisa como qualitativa com relação ao estudo e à análise do mundo real é caracterizada por capturar os dados de informações, atributos, relações e processos para análise, os que não são possíveis de representação por números, apresentando, assim, uma forma ordenada, lógica e conclusiva.

Por isso, nesta monografia, é utilizada a abordagem da pesquisa como qualitativa com relação ao estudo e à análise do mundo real.

Quanto aos objetivos, nesta monografia, é utilizada a pesquisa exploratória.

Gil (2008, p.27) afirma que

> As pesquisas exploratórias têm como principal finalidade desenvolver, esclarecer e modificar conceitos e ideias, tendo em vista a formulação de problemas mais precisos ou hipóteses pesquisáveis para estudos posteriores. De todos os tipos de pesquisa, estas são as que apresentam menor rigidez no planejamento. Habitualmente envolvem levantamento bibliográfico e documental, entrevistas não padronizadas e estudos de caso. Procedimentos de amostragem e técnicas quantitativas de coleta de dados não são costumeiramente aplicados nestas pesquisas.

Por isso, pode-se considerar a pesquisa como exploratória. Nesta monografia, foram realizados inúmeros levantamentos bibliográficos a respeito de padrões arquiteturais de

software e automações de testes, também foi feita uma entrevista com pessoas que tiveram conhecimento sobre o problema abordado.

O procedimento técnico utilizado, nesta monografia, é a pesquisa bibliográfica.

A pesquisa bibliográfica, segundo Gil (2008), é desenvolvida com base no conteúdo já desenvolvido e elaborado, que é constituída, principalmente, por livros e artigos científicos. Mesmo que muitos estudos sejam exigidos algum tipo de trabalho dessa natureza, há pesquisas desenvolvidas exclusivamente a partir de fontes bibliográficas. Grande parte dos estudos exploratórios podem ser definidos como pesquisas bibliográficas.

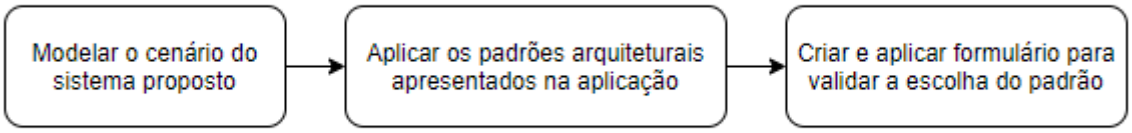
Esta monografia se caracteriza como uma pesquisa bibliográfica por se basear em livros e artigos científicos para se ter compreensão referente a assuntos de padrões arquiteturais de software.

3.2 ETAPAS METODOLÓGICAS

Esta monografia apresenta a escolha de um padrão arquitetural para uma aplicação Android. A mesma foi elaborada com um dos padrões arquiteturais apresentados no capítulo de revisão bibliográfica que é mais compatível, segundo suas características.

O fluxograma das etapas criadas para esta monografia é representado da imagem 9.

Figura 8 – Representação do fluxograma das etapas



Fonte: Autoria própria.

Para avaliar a hipótese da escolha do padrão arquitetural, é feita uma coleta de dados de um público específico. Esses dados serão posteriormente analisados para avaliar a decisão arquitetural escolhida para a solução do problema apresentado.

3.3 DESCRIÇÃO DO OBJETO DE ESTUDO, AMOSTRA OU POPULAÇÃO

O método de coleta e análise de dados é feito a partir de uma aplicação de um formulário. Esse formulário apresentará as informações e as características da aplicação a serem implementadas e, em seguida, apresentará perguntas sobre os padrões arquiteturais citados no capítulo de referência bibliográfica.

O formulário foi respondido por pessoas que trabalham e/ou que têm experiência na área de desenvolvimento de softwares. Essa experiência pode ser tanto para desenvolvimento mobile para iOS, Android ou Windows Phone, quanto para desenvolvimento de aplicações web, independentemente da linguagem utilizada.

3.4 DELIMITAÇÕES

Esta monografia se delimita a apresentar a escolha de um padrão arquitetural que mais condiz para a aplicação Android. Para isso, foram utilizados apenas os padrões exemplificados nesta monografia.

Na escolha desta arquitetura, não foram consideradas algumas características como o desenvolvimento da aplicação, a segurança do aplicativo, a usabilidade da aplicação.

Para concluir qual o padrão arquitetural mais compatível com a aplicação, foi aplicada a teoria dos padrões citados com os diagramas da solução proposta. Porém, o sistema proposto e os testes não são desenvolvidos nesta monografia.

4 MODELAGEM DO SISTEMA PROPOSTO

Para criar a solução, primeiramente é implementado o UML do projeto contendo o Diagrama de Classes.

Nesta seção, são apresentadas as etapas que compõem a modelagem do sistema proposto utilizando a metodologia de desenvolvimento de software ICONIX. Entre elas estão: listagem de requisitos funcionais, listagem de requisitos não funcionais, listagem de regras de negócio, o diagrama de classes, diagrama de casos de uso e diagrama de domínio.

Para finalizar, são demonstrados protótipos de tela referente ao software proposto.

4.1 REQUISITOS

A seguir, são apresentados os requisitos funcionais, requisitos não funcionais e as regras de negócio da solução proposta.

A definição de requisitos de um sistema, segundo Sommerville (2007, p. 79), é apresentada como: “os requisitos de um sistema são descrições dos serviços fornecidos pelo sistema e suas restrições operacionais. Esses requisitos refletem as necessidades dos clientes de um sistema que ajuda a resolver algum problema [...]”.

4.1.1 Requisitos Funcionais

Segundo Sommerville (2007, p. 80), os requisitos funcionais “são as declarações de serviços que o sistema deve fornecer. Como o sistema, deve reagir a entradas específicas e de como o sistema deve se comportar em determinadas situações".

A partir disso, é apresentada uma lista de requisitos funcionais da solução, proposta no quadro 1.

Quadro 1 – Requisitos Funcionais

Número	Requisito
RF01	O sistema deve permitir o cadastro de um usuário
RF02	O sistema deve permitir fazer o *login* de um usuário existente
RF03	O sistema deve permitir a criação de uma viagem
RF04	O sistema deve listar as viagens já criadas
RF05	O sistema deve permitir excluir uma viagem
RF06	O sistema deve permitir a edição de uma viagem
RF07	O sistema deverá listar os eventos de uma viagem
RF08	O sistema deve permitir a criação de um evento
RF09	O sistema deve permitir a edição de eventos
RF10	O sistema deve permitir deletar um evento

Fonte: Autoria própria.

Na próxima seção, serão listados os requisitos não-funcionais

4.1.2 Requisitos Não Funcionais

Já a definição de requisitos não funcionais, para Sommerville (2007, p. 80) significa "restrições sobre os serviços ou as funções oferecidas pelo sistema. Eles incluem restrições de timing, restrições sobre o processo de desenvolvimento e padrões".

O quadro 2 apresenta uma lista de requisitos não funcionais da solução proposta.

Quadro 2 – Requisitos Não Funcionais

Número	Requisito
RNF01	O sistema deve executar em smartphones Android versão 5.0 e superior
RNF02	Os dados de uma conta só podem ser alterados pelo usuário da respectiva conta
RNF03	O sistema deve estar disponível 24h durante 7 dias da semana
RNF04	O sistema deve ter testes automatizados unitários
RNF05	Todos os erros em produção devem ser registrados

Fonte: Autoria própria.

Para Sommerville (2011), também existe características que um sistema não atenderá. No caso desse sistema, o quadro 3 representa algumas características que não serão suportadas pelo sistema apresentado.

Quadro 3 – Requisitos Não Funcionais

Número	Característica
01	O sistema funcionará 100% sem internet
02	O sistema não terá sincronização com a Nuvem

Fonte: Autoria própria.

Na próxima seção, são listadas as regras de negócio do software proposto.

4.1.3 Regras de Negócio

Segundo Von Halle (2001, p.40, tradução nossa), uma regra de negócio é considerada como "[...] uma declaração que define ou restringe alguns aspectos do negócio. Pretende-se afirmar a estrutura do negócio ou controlar ou influenciar o comportamento do negócio".

O quadro 4 apresenta uma lista de regras de negócio da solução proposta.

Quadro 4 – Regras de negócio

Número	Regra de negócio
RN01	Um usuário poderá criar viagens sem limite definido
RN02	Um usuário poderá criar eventos sem limite definido
RN03	Todas as mudanças de evento em uma viagem serão salvas automaticamente
RN04	Quando inserido o horário de início e o horário final de um evento, deverá ser calculado o tempo de duração do evento
RN05	Quando inserido o horário de início e a duração de um evento, deverá ser calculado o horário final do evento
RN06	Todo evento, na hora da criação, deve ter preenchido os dados: tipo, nome, data de início, os horários de início e termino e duração
RN07	Quando inserido a data de início, deverá ser preenchida a mesma data no campo data final
RN08	Todos os eventos do dia serão ordenados por data e horário de início

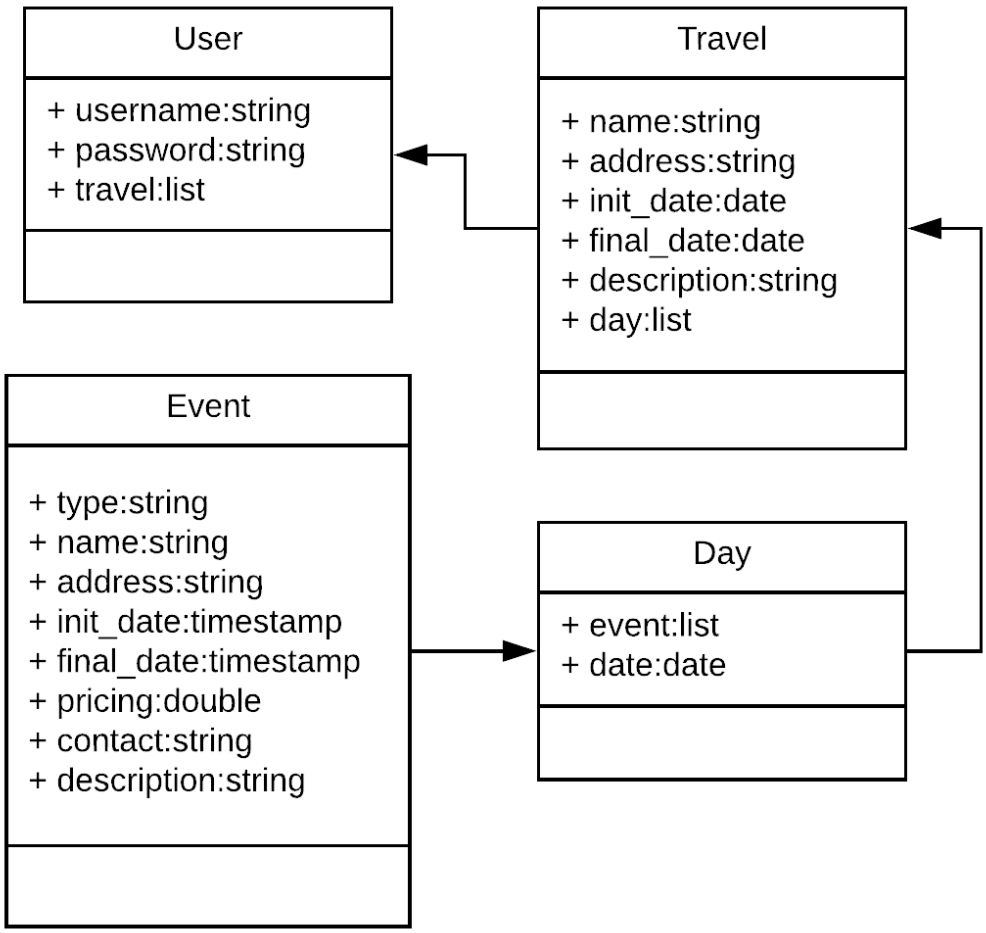
Fonte: Autoria própria.

Na próxima seção, são listados e explicados os protótipos de tela do software proposto.

4.2 DIAGRAMA DE CLASSES

Para desenvolver o sistema utilizando um dos Padrões Arquiteturais explicados anteriormente, é necessário também criar um Diagrama de Classes da solução proposta.

Figura 9 – Classe do modelo da solução proposta



Fonte: Autoria própria.

A figura 10 representa as classes do modelo de domínio do sistema proposto. Ela compõe as classes de usuário, viagem, dias e eventos.

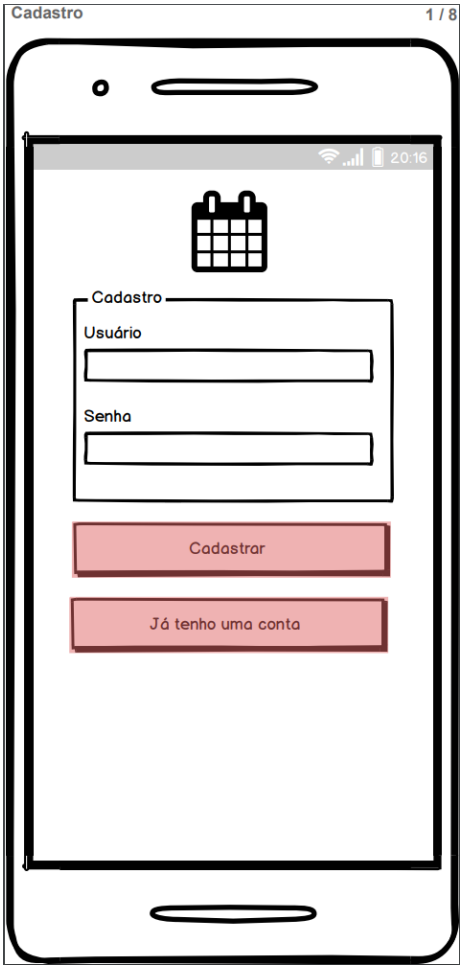
4.3 PROTÓTIPOS DE TELA

Para Sommerville (2011, p. 30), “Um protótipo é uma versão inicial de um sistema de software, usado para demonstrar conceitos, experimentar opções de projeto e descobrir mais sobre o problema e suas possíveis soluções.”

No decorrer desta seção, são apresentados os protótipos de média fidelidade das principais telas do sistema, formando o layout básico do sistema.

Deverá ser cadastrado um usuário através da página de Cadastro, preenchendo os campos necessários com as respectivas informações, conforme apresentado na figura 11.

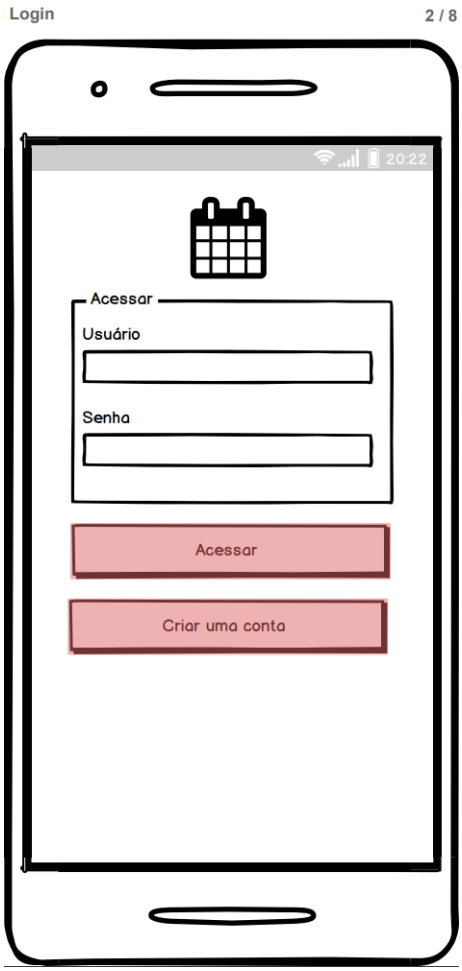
Figura 10 – Protótipo da tela de Cadastro



Fonte: Autoria própria.

Clicando em "Já tenho uma conta", redirecionará para a página de Acesso como mostrada na figura 12.

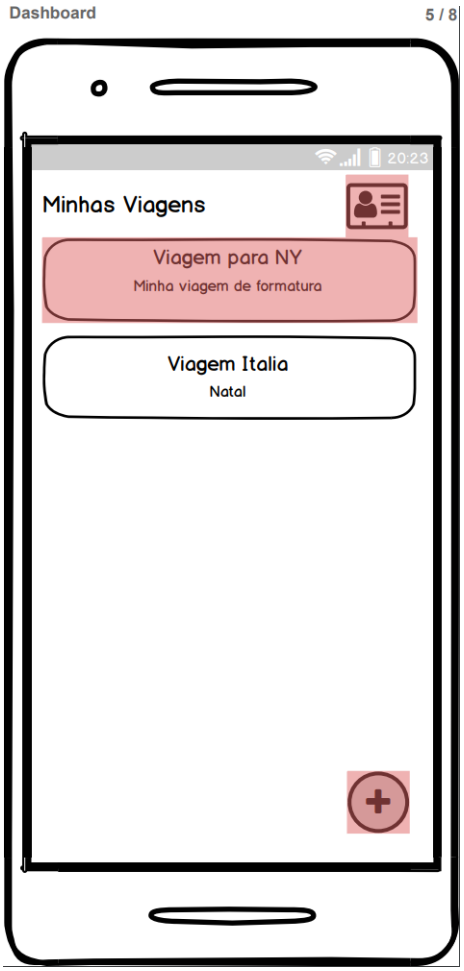
Figura 11 – Protótipo da tela de Acesso



Fonte: Autoria própria

Clicando em "Acessar" da tela de Acesso e clicando em "Cadastrar" na tela de Cadastro, liberará o acesso do usuário e, assim, redirecionará para a página de Listagem de Viagens, como mostrado na figura 13.

Figura 12 – Protótipo da tela de Listagem de Viagens

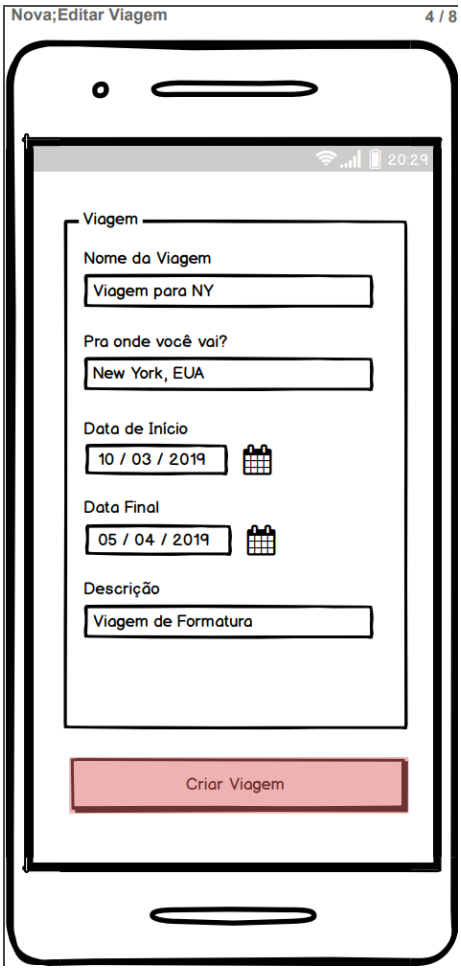


Fonte: Autoria própria.

Quando na tela de Listagem de Viagens, como mostrado na figura 13, é possível criar viagens, clicando no botão de mais no canto inferior da tela. Também, é possível acessar uma viagem já criada, clicando em cima do nome da viagem escolhida. Caso clicar no símbolo de ficha de inscrição no lado direito superior, será feito o *logoff* do usuário da conta.

Assim que clicar para criar uma viagem, será redirecionado para a página de Criação de Viagens, em que deverá cadastrar o nome da viagem, o local de destino, a data de início, a data final e uma breve descrição da viagem, assim, como mostra na figura 14.

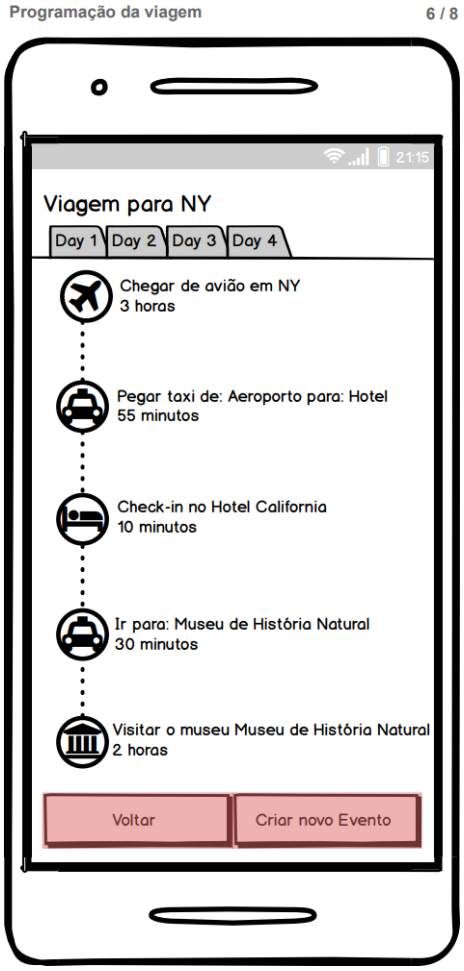
Figura 13 – Protótipo da tela de Criação de Viagens



Fonte: Autoria própria.

Quando na tela de Listagem de Viagens, como mostrado na figura 13, é possível acessar uma viagem, clicando no nome da viagem listada e será redirecionado para a página de Programação da Viagem, como mostrado na figura 15.

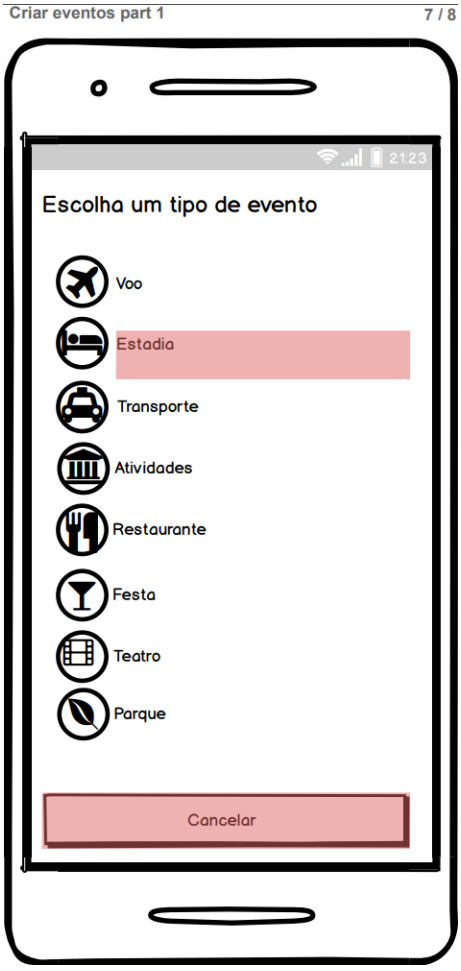
Figura 14 – Protótipo da tela de Programação de Viagens



Fonte: Autoria própria.

A página de Programação de Viagens listará todos os eventos criados pelo usuário a ser feito no respectivo dia da viagem. Caso o usuário clicar no botão "Voltar", será redirecionado para a página de Listagem de Viagens. É possível criar eventos de diversos tipos, clicando no botão "Criar novo Evento", que assim, será redirecionado para a tela de Criação de eventos, como mostra na figura 16.

Figura 15 – Protótipo da tela de Criação de Eventos – Escolha do tipo de evento



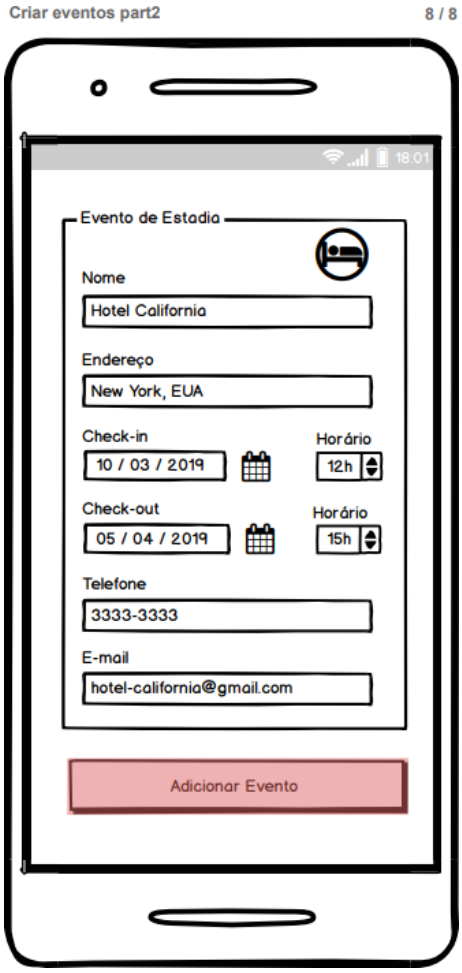
Fonte: Autoria própria.

A página de Criação de Eventos é dividida em duas, onde a primeira é a escolha do tipo de evento e a segunda é a inserção de dados de um determinado tipo de evento.

Os possíveis tipos de eventos são: voo, estadia, transporte, atividades, restaurantes, festa, teatro, parque.

Assim que for decidido um tipo de evento, clicando no nome do evento escolhido, será redirecionado para a tela de Criação de Eventos onde são inseridos os dados do evento, como mostra na figura 17.

Figura 16 – Protótipo da tela de Criação de Eventos – Inserção dos dados



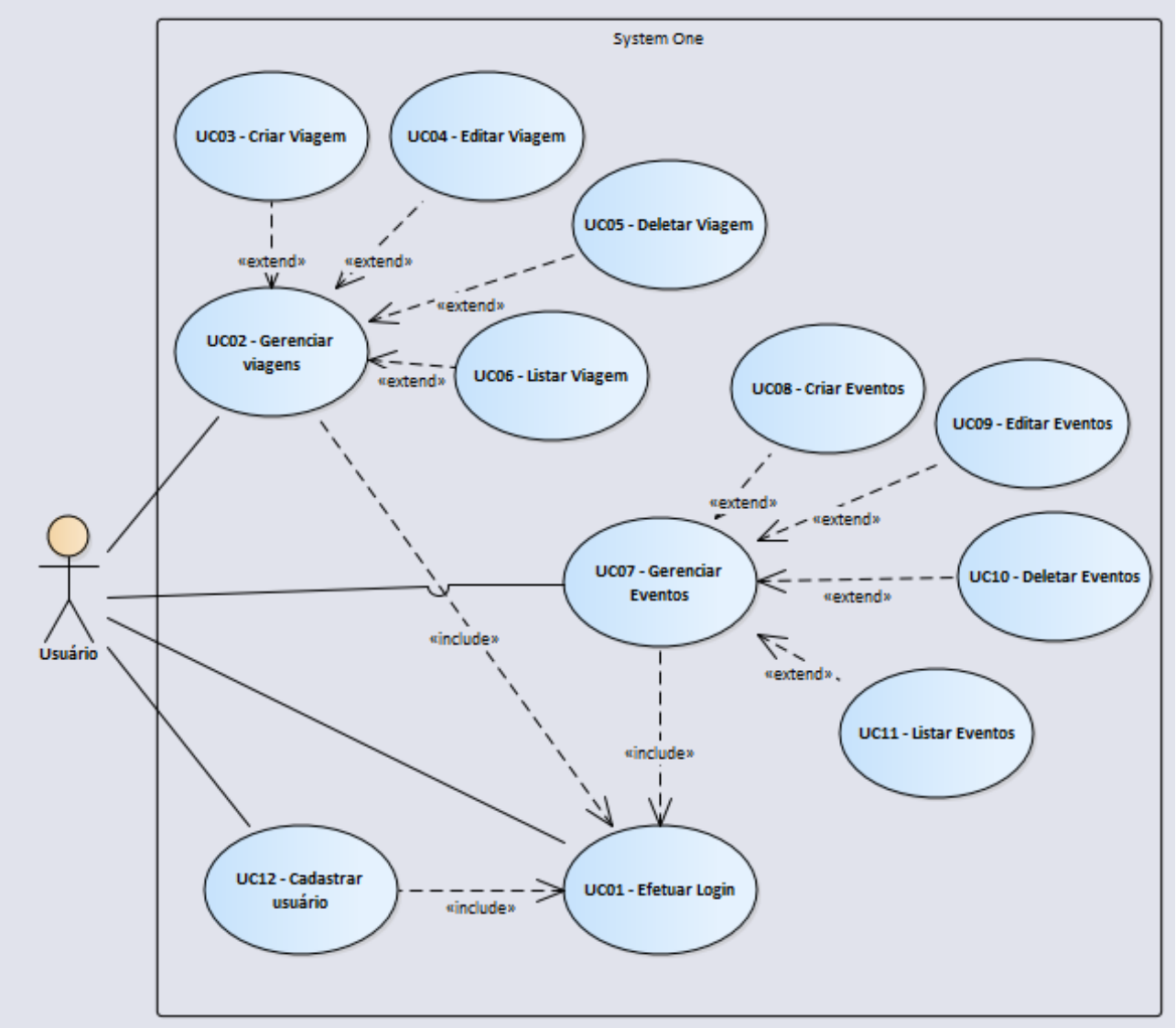
Fonte: Autoria própria.

Assim que clicar no botão "Adicionar Evento" dessa página, será criado o respectivo evento e adicionado à lista de Programação da viagem e, então, será redirecionada à página para a Programação da viagem escolhida.

4.4 DIAGRAMA DE CASO DE USO

A seguir é definido o diagrama de caso de uso, originado dos requisitos funcionais, mostrando as funcionalidades do sistema. O atore que a desempenha, são mostrados nas figuras seguintes, separados por módulos para melhor entendimento.

Figura 17 – Diagrama de caso de uso da solução proposta



Fonte: Autoria própria.

A figura 18 representa o diagrama casos de uso do sistema proposto composta por atores e casos de uso do sistema proposto. A seguir, baseado nos casos de uso criados, é apresentado suas respectivas condições, bem como a descrição dos fluxos principais e alternativos.

No quadro 5, é demonstrado o caso de uso “Efetuar login” (UC01). Todos os casos de uso apresentados utilizam este como requisito, sendo necessário sempre efetuar o *login* para ter acesso às outras funcionalidades do sistema.

Quadro 5 – Quadro de definição do caso de uso “Efetuar *login*”.

UC01 – Efetuar *Login*
Descrição: Autenticar usuários que possuem acesso ao sistema.
Pré-condições: Estar cadastrado no sistema.
Pós-condições: O ator é autenticado e possui acesso ao aplicativo.
Requisitos funcionais: RF002
- Fluxo Principal **Passo 1:** Ator acessa a tela de *login.* **Passo 2:** Sistema solicita os dados de acesso (*login* e senha). **Passo 3:** Ator preenche os dados de acesso. **Passo 4:** O sistema verifica os dados inseridos. **Passo 5:** O sistema redireciona para a tela referente ao acesso.
- Fluxo alternativo A **Passo 1:** No Passo 3, do fluxo principal, ocorre um erro na autenticação. **Passo 2:** O sistema informa o erro para o ator. **Passo 3:** O sistema retorna para o Passo 2.

Fonte: Autoria própria

As condições do caso de uso “Cadastrar usuário” (UC12) são apresentadas no quadro 6, bem como a descrição dos fluxos principais e alternativo.

Quadro 6 – Quadro de definição do caso de uso “Cadastrar usuário”.

UC12 – Cadastrar usuário
Descrição: Cadastrar usuários que possuem acesso ao sistema.
Pré-condições: Não estar cadastrado no sistema.
Pós-condições: O ator é autenticado e possui acesso ao aplicativo.
Requisitos funcionais: RF001
- Fluxo Principal A **Passo 1:** Ator acessa a tela de cadastro. **Passo 2:** Sistema solicita os dados de acesso (*login* e senha). **Passo 3:** Ator preenche os dados de acesso. **Passo 4:** O sistema verifica os dados inseridos. **Passo 5:** O sistema redireciona para a tela referente ao acesso.
- Fluxo alternativo A **Passo 6:** No Passo 3, do fluxo principal, ocorre um erro na autenticação. **Passo 7:** O sistema informa o erro para o ator. **Passo 8:** O sistema retorna para o Passo 2.

Fonte: Autoria própria

As condições do caso de uso “Listar viagem” (UC06) são apresentadas no quadro 7, bem como a descrição dos fluxos principais e alternativo.

Quadro 7 – Quadro de definição do caso de uso “Listar viagem”.

UC06 – Listar viagem
Descrição: Listar todas as viagens criadas.
Pré-condições: Ter efetuado o *login* no sistema.
Pós-condições: O ator visualiza uma lista de todas as viagens criadas.
Requisitos funcionais: RF004
- Fluxo Principal **Passo 1:** Ator acessa a tela listagem de viagem. **Passo 2:** O sistema lista todas as viagens existentes.
- Fluxo alternativo A

Passo 3: No Passo 2, do fluxo principal, é lançada uma mensagem de registro vazio caso não haja viagem. **Passo 4:** O sistema retorna ao passo 1 do fluxo principal.

Fonte: Autoria própria

As condições do caso de uso “Criar viagem” (UC03) são apresentadas no quadro 8, bem como a descrição dos fluxos principais e alternativo.

Quadro 8 – Quadro de definição do caso de uso “Criar viagem”.

UC03 – Criar viagem
Descrição: Criação de uma nova viagem no sistema.
Pré-condições: Ter efetuado o *login* no sistema.
Pós-condições: O ator cria uma viagem.
Requisitos funcionais: RF003
- Fluxo Principal **Passo 1:** Ator acessa a tela de criação de viagens. **Passo 2:** Sistema solicita os dados, como, nome da viagem, destino, data de início, data final e descrição da viagem. **Passo 3:** Ator preenche os dados de acesso. **Passo 4:** O sistema verifica os dados inseridos. **Passo 5:** O sistema lança a mensagem de sucesso de cadastro de viagem. **Passo 6:** O sistema atualiza a listagem de viagens. **Passo 7:** O sistema redireciona para a tela referente a listagem de viagem.
- Fluxo alternativo A **Passo 8:** Ocorre um erro no Passo 4 porque alguns dados inseridos não passam nas validações do sistema. **Passo 9:** O sistema lança um erro e retorna para o passo 2 do fluxo principal.

Fonte: Autoria própria

As condições do caso de uso “Editar viagem” (UC04) são apresentadas no quadro 9, bem como a descrição dos fluxos principais e alternativo.

Quadro 9 – Quadro de definição do caso de uso “Editar viagem”.

UC04 – Editar Viagem
Descrição: Editar uma viagem já criada.
Pré-condições: Ter efetuado o *login* no sistema. Ter ao menos uma viagem criada.
Pós-condições: Viagem atualizada com os novos dados.
Requisitos funcionais: RF006
- Fluxo Principal **Passo 1:** Ator acessa a tela de listagem de viagem. **Passo 2**: Ator seleciona uma viagem listada. **Passo 3:** O sistema redireciona para a página de criação de viagem. **Passo 4:** O sistema exibe todas informações da viagem selecionada. **Passo 5:** O ator altera as informações necessárias **Passo 6:** O sistema valida os dados inseridos. **Passo 7:** O sistema atualiza as informações alteradas. **Passo 8:** O sistema lança uma mensagem de sucesso.
- Fluxo alternativo A **Passo 9:** Ocorre um erro no passo 6. **Passo 10:** O sistema lança uma mensagem de erro. **Passo 11:** O sistema retorna ao passo 4 do fluxo principal.

Fonte: Autoria própria

As condições do caso de uso “Deletar viagem” (UC05) são apresentadas no quadro 10, bem como a descrição dos fluxos principais e alternativo.

Quadro 10 – Quadro de definição do caso de uso “Deletar viagem”.

UC05 – Deletar Viagem
Descrição: Deletar uma viagem já criada.
Pré-condições: Ter efetuado o *login* no sistema. Ter ao menos uma viagem criada.
Pós-condições: Ator deletar uma viagem.
Requisitos funcionais: RF005
- Fluxo Principal **Passo 1:** Ator acessa a tela de listagem de viagem.

Passo 2: Ator seleciona uma viagem listada. **Passo 3:** O sistema redireciona para a página de criação de viagem. **Passo 4:** O sistema exibe todas informações da viagem selecionada. **Passo 5:** O ator clica no botão de Excluir. **Passo 6:** O sistema pergunta se o usuário tem certeza. **Passo 7:** O usuário clica em "sim". **Passo 8:** O sistema exclui a viagem do sistema. **Passo 9:** O sistema lança uma mensagem de sucesso.
- Fluxo alternativo A **Passo 10:** O usuário clica em "não". **Passo 11:** O sistema retorna ao passo 4 do fluxo principal. **- Fluxo alternativo B** **Passo 12:** Ocorre um erro no passo 8. **Passo 13:** O sistema apresenta uma mensagem de erro. **Passo 14:** O sistema retorna ao passo 4 do fluxo principal.

Fonte: Autoria própria

As condições do caso de uso "Criar eventos" (UC08) são apresentadas no quadro 11, bem como a descrição dos fluxos principais e alternativo.

Quadro 11 – Quadro de definição do caso de uso "Criar eventos".

UC08 – Criar eventos
Descrição: Criação de um novo evento para uma viagem.
Pré-condições: Ter efetuado o *login* no sistema. Ter ao menos uma viagem criada
Pós-condições: O ator cria um evento para uma viagem específica.
Requisitos funcionais: RF008
- Fluxo Principal **Passo 1:** Ator acessa a tela de listagem de viagem. **Passo 2:** Ator seleciona uma viagem específica. **Passo 3:** Ator acessa a tela de criação de evento. **Passo 4:** Sistema solicita o tipo de evento a ser criado. **Passo 5:** Ator seleciona o tipo do evento. **Passo 6:** Sistema redireciona para a tela de inserção de dados para a criação de eventos

Passo 7: Sistema solicita dados, como, nome, endereço, data de início, data final, horário de início, horário final, telefone, e-mail. **Passo 8:** Ator preenche os dados de acesso. **Passo 9:** O sistema verifica os dados inseridos. **Passo 10:** O sistema lança a mensagem de sucesso de cadastro de evento. **Passo 11:** O sistema atualiza a listagem de viagens. **Passo 12:** O sistema redireciona para a tela referente a listagem de eventos da viagem selecionada.
- Fluxo alternativo A **Passo 13:** Ocorre um erro no Passo 9 porque alguns dados inseridos não passam nas validações do sistema. **Passo 14:** O sistema lança um erro e retorna para o passo 7 do fluxo principal.

Fonte: Autoria própria

As condições do caso de uso “Listar eventos” (UC11) são apresentadas no quadro 12, bem como a descrição dos fluxos principais e alternativo.

Quadro 12 – Quadro de definição do caso de uso “Listar eventos”.

UC11 – Listar eventos
Descrição: Criação de um novo evento para uma viagem.
Pré-condições: Ter efetuado o *login* no sistema. Ter ao menos uma viagem criada
Pós-condições: O ator verá uma lista de eventos para uma viagem específica.
Requisitos funcionais: RF007
- Fluxo Principal **Passo 1:** Ator acessa a tela de listagem de viagem. **Passo 2**: Ator seleciona uma viagem listada. **Passo 3:** O sistema lista todos os eventos existentes para a viagem selecionada.
- Fluxo alternativo A **Passo 4:** No Passo 3, do fluxo principal, é lançada uma mensagem de registro vazio caso não haja evento criados. **Passo 5:** O sistema retorna ao passo 1 do fluxo principal.

Fonte: Autoria própria

As condições do caso de uso “Editar eventos” (UC09) são apresentadas no quadro 13, bem como a descrição dos fluxos principais e alternativo.

Quadro 13 – Quadro de definição do caso de uso “Editar eventos”.

UC09 – Editar eventos
Descrição: Editar um evento já criado para uma viagem.
Pré-condições: Ter efetuado o *login* no sistema. Ter ao menos uma viagem criada com ao menos um evento criado.
Pós-condições: Evento atualizado com os novos dados.
Requisitos funcionais: RF009
- Fluxo Principal **Passo 1:** Ator acessa a tela de listagem de viagem. **Passo 2**: Ator seleciona uma viagem listada. **Passo 3: S**istema lista todos os eventos existentes para a viagem selecionada. **Passo 4:** Ator seleciona o evento a ser alterado. **Passo 5:** O sistema redireciona para a página de inserção de dados de criação de evento. **Passo 6:** O sistema exibe todas informações do evento selecionado. **Passo 7:** O ator altera as informações necessárias **Passo 8:** O sistema valida os dados inseridos. **Passo 9:** O sistema atualiza as informações alteradas. **Passo 10:** O sistema lança uma mensagem de sucesso.
- Fluxo alternativo A **Passo 11:** Ocorre um erro no passo 8. **Passo 12:** O sistema lança uma mensagem de erro. **Passo 13:** O sistema retorna ao passo 6 do fluxo principal.

Fonte: Autoria própria

As condições do caso de uso “Deletar eventos” (UC10) são apresentadas no quadro 14, bem como a descrição dos fluxos principais e alternativo.

Quadro 14 – Quadro de definição do caso de uso “Deletar evento”.

UC10 – Deletar eventos
Descrição: Deletar um evento existente de uma viagem já criada.
Pré-condições: Ter efetuado o *login* no sistema. Ter ao menos uma viagem criada com ao menos um evento criado.
Pós-condições: Ator deletar um evento de uma viagem específica.
Requisitos funcionais: RF010
- Fluxo Principal **Passo 1:** Ator acessa a tela de listagem de viagem. **Passo 2**: Ator seleciona uma viagem listada. **Passo 3:** O sistema redireciona para a página de criação de viagem. **Passo 4: S**istema lista todos os eventos existentes para a viagem selecionada. **Passo 5:** Ator seleciona o evento a ser alterado. **Passo 6:** O sistema redireciona para a página de inserção de dados de criação de evento. **Passo 7:** O sistema exibe todas informações do evento selecionado. **Passo 8:** O ator clica no botão de Excluir. **Passo 9:** O sistema pergunta se o usuário tem certeza. **Passo 10:** O usuário clica em “sim”. **Passo 11:** O sistema exclui o evento do sistema. **Passo 12:** O sistema lança uma mensagem de sucesso.
- Fluxo alternativo A **Passo 10:** O usuário clica em “não”. **Passo 11:** O sistema retorna ao passo 4 do fluxo principal. **- Fluxo alternativo B** **Passo 12:** Ocorre um erro no passo 11. **Passo 13:** O sistema apresenta uma mensagem de erro. **Passo 14:** O sistema retorna ao passo 4 do fluxo principal.

Fonte: Autoria própria

Essas tabelas apresentadas representam o conjunto de casos de uso para o aplicativo proposto que foram criados a partir dos requisitos funcionais apresentados anteriormente.

5 DESENVOLVIMENTO

Neste capítulo são abordadas as diferenças, e os pontos positivos e negativos entre os padrões arquiteturais explicados anteriormente. Então, é aplicado no sistema especificado cada um dos padrões arquiteturais, exemplificado nesta monografia.

Assim, é possível escolher o padrão que mais se encaixa no cenário deste sistema e a escolha é comprovada com uma pesquisa de campo feita às pessoas que trabalham e têm experiência na área de Tecnologia de Informação.

5.1 DIFERENCIAÇÃO ENTRE OS PADRÕES ARQUITETURAIS

Este capítulo apresenta as diferenças, e pontos positivos e negativos entre cada um dos padrões arquiteturais apresentados nesta monografia. Esses pontos são levantados por alguns autores referenciados nesta monografia e, ao longo do capítulo, é feita uma compilação desses argumentos apresentados em um quadro de comparação.

Como citado anteriormente, um dos padrões analisados nesta monografia é o padrão MVC.

Falando sobre o padrão arquitetural MVC, Wilson (2015, p. 67) traz uma visão negativa sobre o padrão MVC. O autor afirma que esse padrão não é o suficiente para a maioria dos sistemas implementados. Ele ainda retrata o ponto negativo de este padrão ser complexo quando se fala de organização do código pois o código é distribuído em poucas camadas, apenas as três camadas do padrão MVC.

Esse cenário, segundo Wilson (2015, p. 67), possibilita a adição de mais responsabilidades na camada do model criando uma dependência entre a camada de dados e a camada de persistência. Caso isso ocorra, se torna mais difícil separar essas duas camadas quando surgir a necessidade de migrar algo do sistema, pois se tem um acoplamento entre essas camadas. Caso ocorra essa junção de responsabilidade entre as camadas, será violado o princípio de Responsabilidade Única, apresentado anteriormente nesta monografia.

Como dito anteriormente no capítulo 2 desta monografia, os códigos da camada de persistência são adicionados dentro das classes do model pois é a camada onde mais encaixa

esse contexto se comparada com todas as outras camadas, e com isso ocorre o acoplamento das responsabilidades nas classes de model. Segundo, Wilson (2015, p. 68), o código de persistência de dados pertence a camada model, porém as pessoas desenvolvedoras acabam adicionando esse código na classe de model.

A comunicação das camadas deste padrão é livre entre o model e o controller, isso quer dizer que eles podem se comunicar entre si, esse ponto foi levantado no capítulo 2 desta monografia. Ter acesso a essa comunicação traz um ponto negativo, que é a criação de dependência entre essas camadas. Essa dependência resulta na dificuldade de implementação de código e consequentemente implica na dificuldade de implementação de testes para este código

Outra característica positiva sobre esse padrão apresentada no capítulo 2 é a possibilidade de criar várias Views para uma entidade Model. Isso facilita a adesão deste padrão pois se torna prático essa utilização, mesmo que aplicado em um aplicativo de pequeno porte ou grande porte.

Outro padrão arquitetural analisado nesta monografia é o padrão arquitetural MVP.

Hall (2010, p. 77) afirma que neste modelo, o Presenter consegue se comunicar com a View para recuperar dados, ou seja, o presenter tem conhecimento sobre a view e o model ao mesmo tempo. Isso indica uma certa dependência nas duas direções entre o View e o Presenter. Esse fato torna mais difícil a ação de criar códigos que não estejam acoplados entre essas duas camadas.

Um ponto positivo que foi comentado anteriormente no capítulo 2 sobre este padrão é o fato de apresentar uma testabilidade melhor do que se comparado com o padrão MVC. Isso se dá pois com esse padrão é possível criar código desacoplado do sistema para simular uma resposta referente a cada camada.

O próximo padrão arquitetural analisado nesta monografia é o padrão arquitetural PM.

Para uma maior portabilidade, com esse padrão, um Presentation Model pode ter várias views, porém se deve considerar também que cada View deve apenas exigir um Presentation Model filho, segundo Fowler (2018).

Outro ponto positivo a ser listado com esse padrão, conforme visto no capítulo 2 desta monografia, é o aumento de código que pode ser testado pois o Presentation Model tem como uma das responsabilidades os códigos referentes a lógica de View.

Um fato levantado no capítulo 2 que é considerado um ponto negativo é que este padrão precisa da implementação de código para garantir a comunicação entre as camadas deste padrão. Essa conclusão levanta outros pontos negativos pois conforme citado no capítulo 2 desta monografia, quanto mais código criado no seu aplicativo, mais possibilidade de encontrar uma falha na implementação.

O último padrão arquitetural exemplificado nesta monografia é o padrão arquitetural MVVM.

Pode-se considerar um ponto levantado por Hall (2010, p. 78), diz que pode ser muito custoso criar uma aplicação com o padrão arquitetural MVVM considerando que será desenvolvido e mantido por apenas uma, ou poucas, pessoas desenvolvedoras. Caso esse aplicativo seja de pequeno porte e com poucas pessoas para desenvolvê-lo, esse padrão arquitetural não é o melhor padrão para se escolher.

Porém, como descrito anteriormente no capítulo 2 desta monografia, um ponto positivo levantado é a comunicação entre as camadas do padrão MVVM são feitas somente quando é necessário a sincronização dos dados de uma camada para a outra. Esse fato também contribui para a diminuição do acoplamento e dependência da comunicação, tanto quanto a conclusão explicada a seguir.

Pan et al. (2011, p. 4) traz sobre o padrão MVVM, é que esse padrão foi criado com apenas uma ligação de comunicação de mão única entre a ViewModel e a View. Isso diminui drasticamente a dependência de comunicação entre essas camadas.

Segundo Mihailesc (2010, p. 2), comparando este padrão com o padrão PM, a maior diferença entre eles é que o padrão MVVM não precisar implementar a comunicação entre as camadas do padrão, pois isso é automatizado com esse padrão.

Para organizar e concluir as análises feitas sobre as características de cada um dos padrões arquiteturais foi elaborado um quadro.

O quadro 15 apresenta uma lista das diferentes características entre os padrões arquiteturais listados nesta monografia apresentados pelos capítulos 2 e 5 pelos autores citados anteriormente.

Quadro 15 – Quadro com diferenciação entre padrões arquiteturais

Padrão Arquitetural	Características Positivas	Características Negativas
MVC	Permite a criação de várias views para apenas um model.	Difícil entender as responsabilidades de cada camada deste padrão.
	Permite a criação de várias views para apenas um model.	Propensão a criar a camada de persistência nas classes do model criando um acoplamento.
MVP	Comparado com o MVC, esse padrão apresenta uma maior testabilidade.	Dependência entre as camadas em duas direções pois o presenter tem consciência tanto da View quanto do Model.
PM	Maximiza a quantidade de código que pode ser testado.	Precisa da implementação de código para garantir a comunicação entre as camadas.
	Várias Views podem utilizar o mesmo Presentation Model, mas cada View deve exigir apenas um Presentation Model filho.	
MVVM	A estrutura deste padrão, com responsabilidades únicas e o mínimo acoplamento, permite o teste unitário entre cada camada sem utilizar ferramentas para teste de interface.	Muito custoso criar uma aplicação de pequeno porte com este modelo e com poucos desenvolvedores.
	Menor acoplamento possível e dependência da comunicação entre as camadas deste padrão.	

Fonte: Autoria própria

O quadro apresentado na Figura 19, exemplifica as características positivas e negativas citadas anteriormente extraídas do capítulo 2 e 5 desta monografia sobre cada padrão arquitetural listado.

A partir desses dados e dessas conclusões, é possível entender como é feita a seleção do melhor padrão arquitetural para o aplicativo apresentado.

5.2 APLICAÇÃO DOS PADRÕES ARQUITETURAIS

Para a escolha do padrão arquitetural do sistema proposto, o diagrama de classes do sistema deve ser composto em cima de cada um dos padrões arquiteturais mostrando a modelagem de cada uma das opções. Assim, é escolhido o melhor padrão arquitetural que se encaixe com o sistema proposto.

A seção a seguir mostra a implementação da modelagem em cima dos diagramas dos padrões arquiteturais conceituados anteriormente para definir qual o padrão é seguido para a implementação nesta monografia.

Nos diagramas apresentados a seguir não é considerado a ligação entre as entidades como foi exemplificado no diagrama de classes da solução proposta. Dentro de cada uma dessas camadas está exemplificada cada classe a ser criada com esse padrão arquitetural e as ligações entre as camadas estão apresentadas pelas setas. Todas as formações dos padrões seguem como base os diagramas apresentados no capítulo 2 sobre cada um de seus respectivos padrões.

As camadas do padrão arquitetural MVC são apresentadas na figura 19 como as colunas de Model, View e Controller.

As camadas do padrão arquitetural MVP são apresentadas na figura 20 como as colunas de Model, View e Presenter.

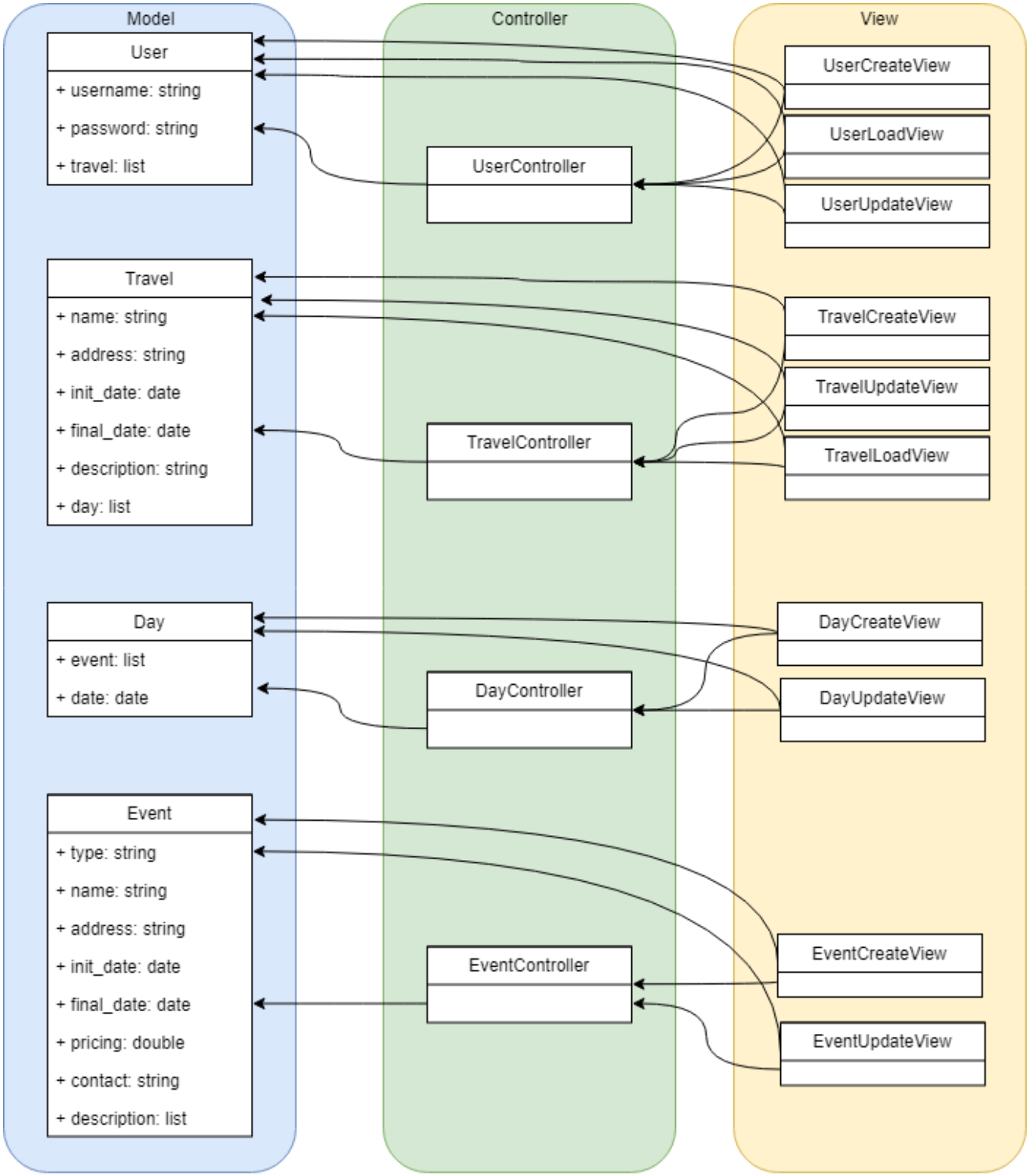
As camadas do padrão arquitetural PM são apresentadas na figura 21 como as colunas de Model, View e Presentation Model.

As setas pontilhadas na figura 21 representam uma comunicação de baixa relação e acoplamento, mas que ainda assim é necessário que exista uma comunicação neste sentido. Como a camada de View tem uma comunicação apenas para sincronizar seu estado com o Presentation Model, ela é considerada como uma comunicação de relação baixa e acoplamento representada por setas pontilhadas.

As camadas do padrão arquitetural MVVM são apresentadas na figura 22 como as colunas de Model, View e ViewModel.

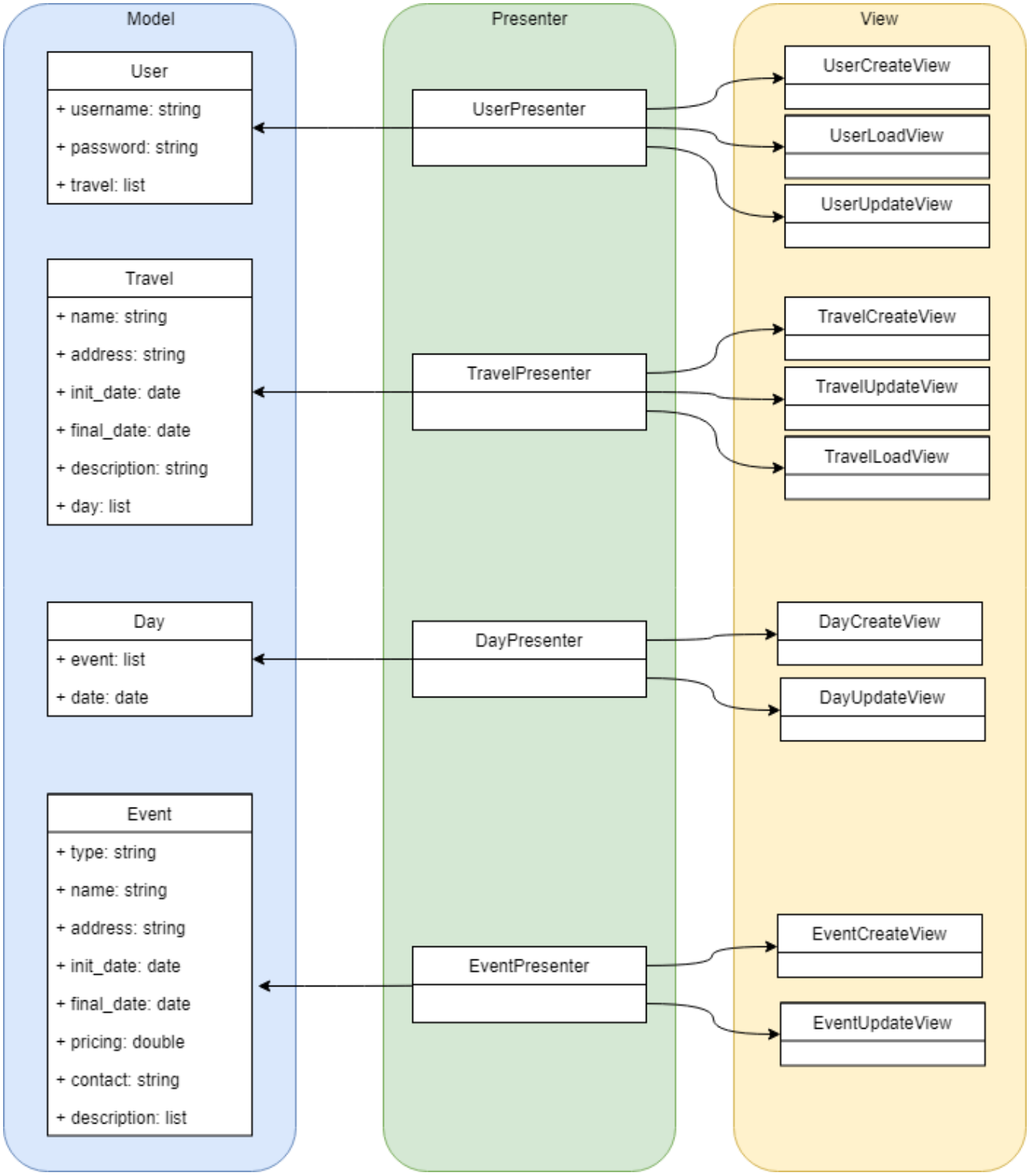
Como a comunicação entre as camadas ViewModel e a camada View são de baixa relação e acoplamento, o diagrama também pode ser representado com setas pontilhadas entre essas camadas.

Figura 18 – Diagrama MVC aplicado na solução proposta



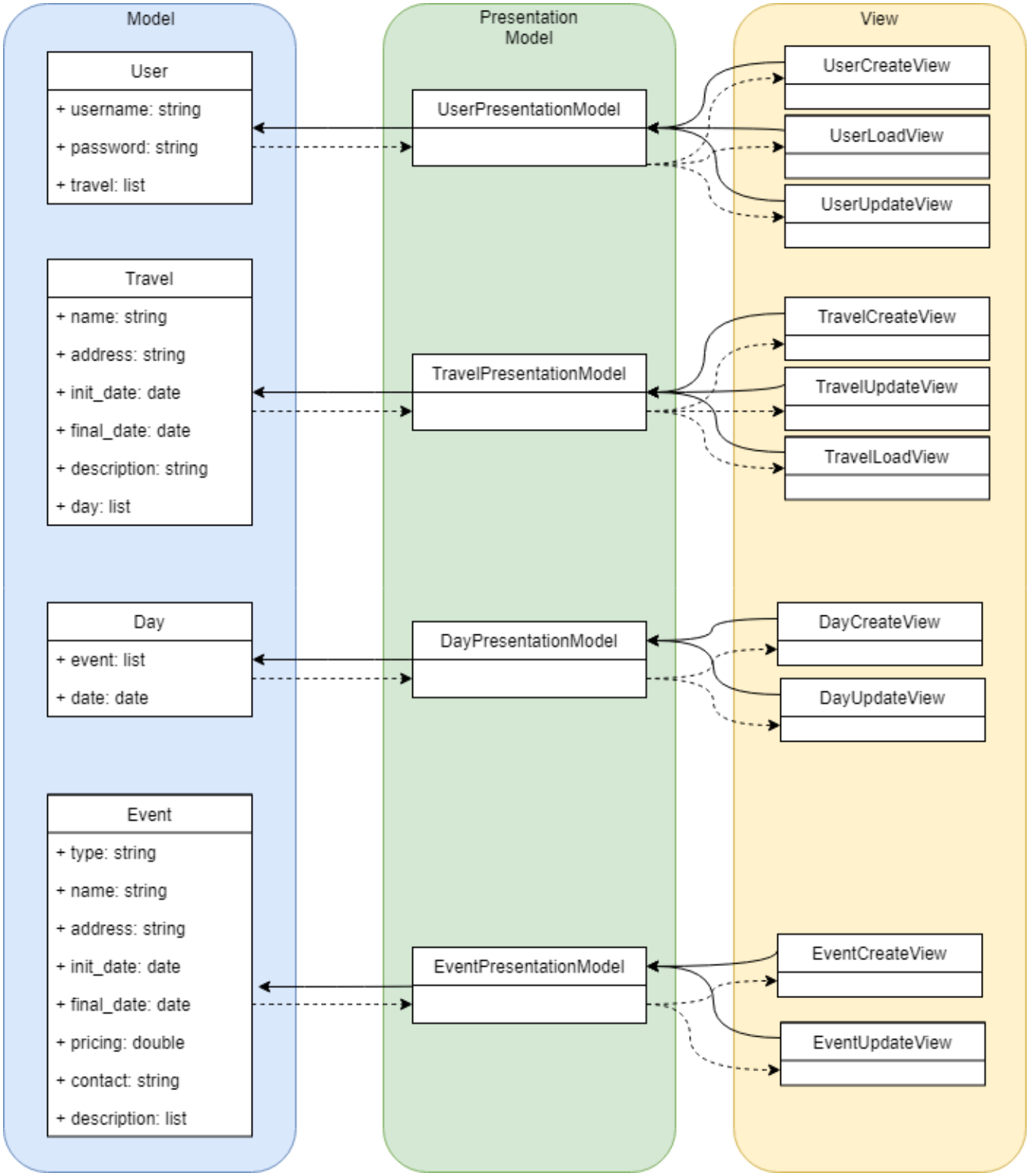
Fonte: Autoria própria

Figura 19 – Diagrama MVP aplicado na solução proposta



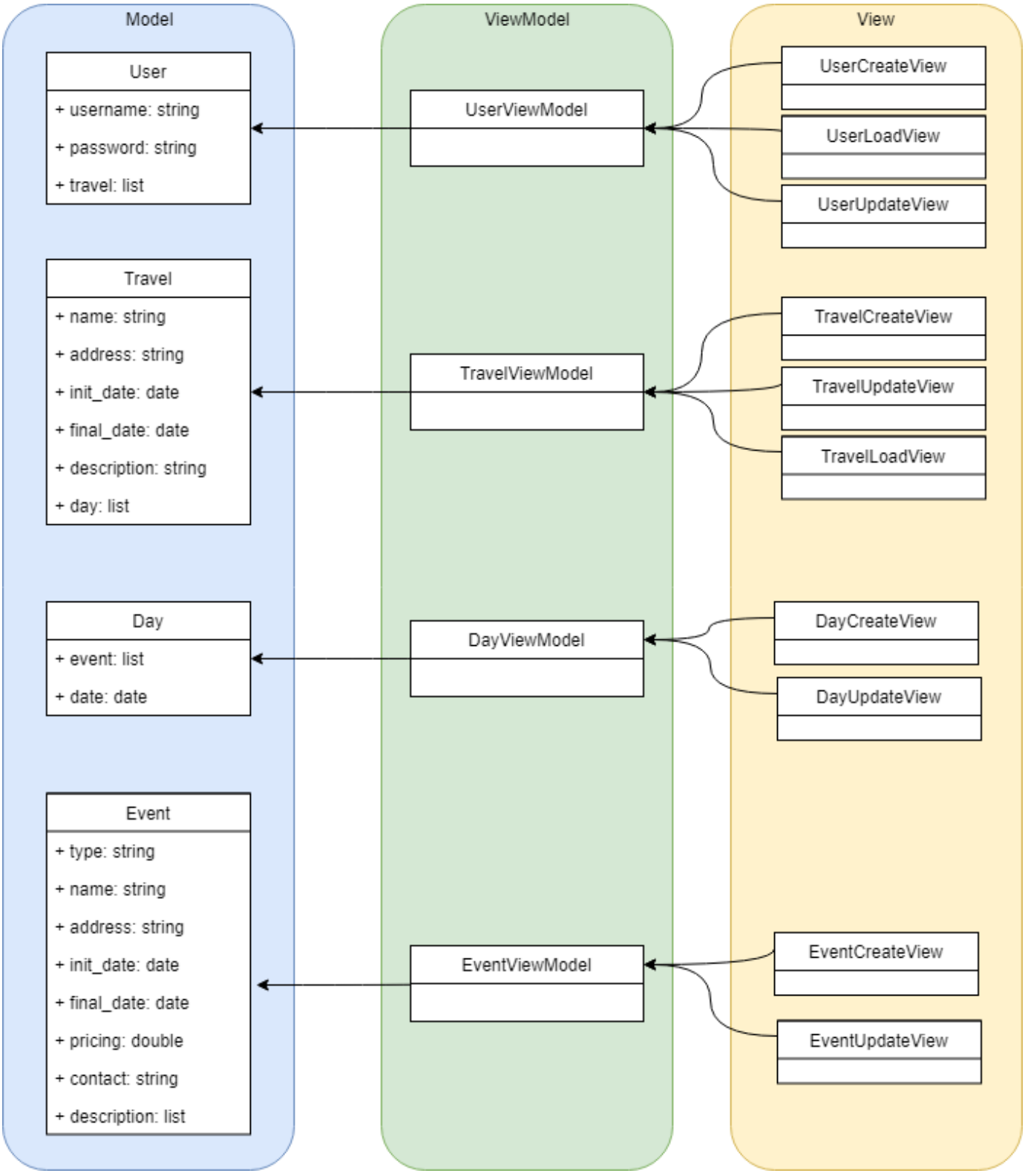
Fonte: Autoria própria

Figura 20 – Diagrama PM aplicado na solução proposta



Fonte: Autoria própria

Figura 21 – Diagrama MVVM aplicado na solução proposta



Fonte: Autoria própria

As imagens anteriores são as representações da aplicação de cada padrão arquitetural aplicado no diagrama da solução do aplicativo proposto.

A seguir, será apresentado uma análise de dados, baseada nos resultados obtidos nesta seção, entre os padrões apresentados.

5.3 ANÁLISE ENTRE OS PADRÕES ARQUITETURAIS

Esta seção foca na apresentação das análises feitas pela autora desta monografia, em que é considerado os dados e os resultados da aplicação dos modelos arquiteturais definidos sobre a solução mobile oferecida.

Alguns pontos sobre o padrão MVC são apresentados a seguir.

O padrão arquitetural MVC não tem uma definição específica de como receber e tratar os códigos da camada de persistência. Pessoas desenvolvedoras, que não tem prática com esse padrão arquitetural, podem desenvolver a aplicação criando código de persistência nas classes de model, o que acarreta na difícil extração do acoplamento desse código no futuro.

Outro ponto levantado é o fato de ser difícil entender a responsabilidade de cada uma das camadas e classes do MVC, de acordo com isso, também será difícil criar e entender a responsabilidade de cada teste que deve ser implementado. Isso cria a possibilidade do aplicativo ter uma brecha na cobertura de testes por esse acoplamento das responsabilidades e isso pode tornar o aplicativo menos confiável e seguro e mais propenso a falhas.

Porém, um ponto positivo deste padrão é que muitos profissionais que tem uma experiência geral em programação, como experiência de desenvolvimento web ou mobile, já conhecem esse padrão arquitetural. Isso porque ele é muito utilizado por algumas empresas.

A análise final sobre o padrão MVC é que há uma dificuldade da criação da camada de dados sem o acoplamento da camada de persistência e que há uma grande dependência entre as camadas deste padrão. Esses dois fatores aumentam a dificuldade da implementação de um aplicativo com esse padrão, além de que será muito mais complexo testar ou migrar seu código para outras tecnologias.

O próximo padrão arquitetural a ser analisado é o padrão MVP.

Sobre o ponto negativo ponto levantado no capítulo 5 sobre este padrão arquitetural, a dependência entre o View e o Presenter de via dupla dificulta a implementação de código e

consequentemente dificulta a implementação de testes para este código. Se comparar essa característica com o padrão MVC, é possível perceber que o padrão MVP tem menos dependência que o padrão MVC, porém, continua sendo conceituado como um ponto negativo porque a dependência que existe ainda é considerada como uma grande dependência.

Outro ponto positivo deste padrão é a consideração de que toda comunicação entre View e Model deve ser passada pelo Presenter. Isso resulta na padronização do envio e do tratamento de dados pois o envio de dados sempre será tratado pela mesma camada.

A análise final do padrão MVP, quando comparada com o padrão MVC, resulta em uma pequena diminuição da dependência de comunicação entre as camadas, padronização das respostas pelo Presenter, e uma facilidade maior na implementação de testes.

A seguir é levantado as conclusões sobre o padrão arquitetural PM

Quando esse padrão foi criado, foi levado em consideração a dependência de comunicação, então este padrão, se comparado com o padrão MVP, apresenta um ponto positivo pois teve uma queda considerável na dependência da comunicação entre as suas camadas, diminuindo o acoplamento de código e a facilidade de implementar testes para suas funcionalidades.

A análise final do padrão PM, é de um padrão bem desenvolvido com melhorias a partir dos outros padrões como o MVC e MVP. Se comparado com os padrões MVC e MVP, tem uma estrutura menos acoplada, dando liberdade para criação de código mais consistente e com uma testabilidade maior. Porém apresenta um ponto negativo onde é necessário criar código para a comunicação entre as camadas.

O último padrão arquitetural a ser analisado nesta monografia é o padrão MVVM.

Construir um padrão MVVM realmente pode sim mais custoso ao criar um aplicativo de pequeno porte e mais simples, pois no começo do desenvolvimento será custoso manter e atender as estruturas que o padrão define. O desenvolvimento do aplicativo será mais lento por conta da padronização, onde deverá criar todos os pilares que irão sustentar esse padrão para depois conseguir implementar o código do aplicativo. Porém, se o objetivo do aplicativo for crescer a longo prazo, e caso se tornar uma aplicação com muitas features, é recomendado utilizar esse padrão.

Deve-se organizar e planejar o seu aplicativo antes de começar a implementação, é necessário planejar esses pontos e verificar a possibilidade do aplicativo crescer a longo prazo. Isso deve ser analisado anteriormente porque se deve separar um tempo inicial somente para a criação da estrutura desse padrão no aplicativo. Isso será um investimento a curto prazo. Conforme a aplicação vai crescendo, será mais fácil, rápido e escalável adicionar códigos de

features novas. O padrão arquitetural MVVM garante a escalabilidade das alterações que serão implementadas no código. Adicionar features ou migrar códigos, como por exemplo migrar uma parte do código para ser micro serviços, será mais rápido e fácil essas alterações.

Por conta dessa escalabilidade, é possível perceber um aumento da produtividade e melhor entendimento a longo prazo sobre o aplicativo das pessoas desenvolvedoras.

O fato de o código que faz a comunicação entre as camadas ser automatizado diretamente pelo padrão e não precisa ser implementado, facilita e agiliza a implementação do aplicativo pois com essa automatização da comunicação entre as camadas é necessário apenas invocar no código os componentes nativos, assim a comunicação entre as camadas estará garantida. Isso traz uma grande agilidade no momento da implementação do aplicativo.

Sabendo que o ViewModel não tem conhecimento da View, tem apenas conhecimento do model, pode-se listar mais um ponto positivo pois esse fato diminui a dependência de comunicação entre as camadas deste padrão, que por consequência, torna o código mais legível, aumenta a testabilidade do código e diminui o acoplamento do código. Suas camadas são fracamente ligadas, por conta disso, aumenta a testabilidade deste padrão.

A análise final da estrutura do padrão MVVM, quando comparada com todos os outros padrões apresentados nesta monografia, ele se torna o padrão que mais apresenta suas responsabilidades como únicas e com o mínimo de acoplamento entre suas camadas. Assim, permite a implementação de testes automatizados de uma maneira mais prática e simples, e sem a utilização de ferramentas para testes automatizados para testar o comportamento da view. Quando comparada com o padrão PM se pode perceber uma grande semelhança entre eles. Como o MVVM foi criado a partir do padrão PM, ele foi desenvolvido desconsiderando o ponto negativo do padrão PM pois o padrão MVVM apresenta a automatização da comunicação entre as camadas.

5.4 QUESTIONÁRIO

Para garantir a conformidade das informações levantadas nesta monografia e comprovar os pontos citados anteriormente é aplicado a construção e aplicação de um questionário para um certo público-alvo.

Esta seção fica em explicar sobre a forma como é feita a pesquisa junto com a apresentação dos dados obtidos.

A forma utilizada para avaliação da decisão do padrão arquitetural foi através da elaboração de um questionário no Google Forms. As pessoas selecionadas para responder este questionário foram profissionais da área com algum nível de experiência em desenvolvimento mobile e foi submetida por 48 entrevistados. O questionário ficou disponível para ser respondida a partir do dia 27/09/2018 até o dia 10/10/2018 e ficou disponível por 13 dias. Ao total, o questionário teve 13 perguntas.

A pergunta 01 do questionário, segundo a figura 23, é a seguinte “Você já desenvolveu para aplicações mobile”. Ela se refere a possibilidade de o entrevistado ter experiência com aplicações mobile.

Figura 22 – Pergunta 01 do questionário

Fonte: Autoria própria por Google Forms.

A pergunta 01 obteve 100% de respostas na opção “Sim”.

A pergunta 02 do questionário, segundo a figura 24, é a pergunta “Quanto tempo você tem experiência na área de desenvolvimento mobile?”. Ela se refere a quantidade de tempo que o entrevistado tem com aplicações mobile justificando as futuras respostas.

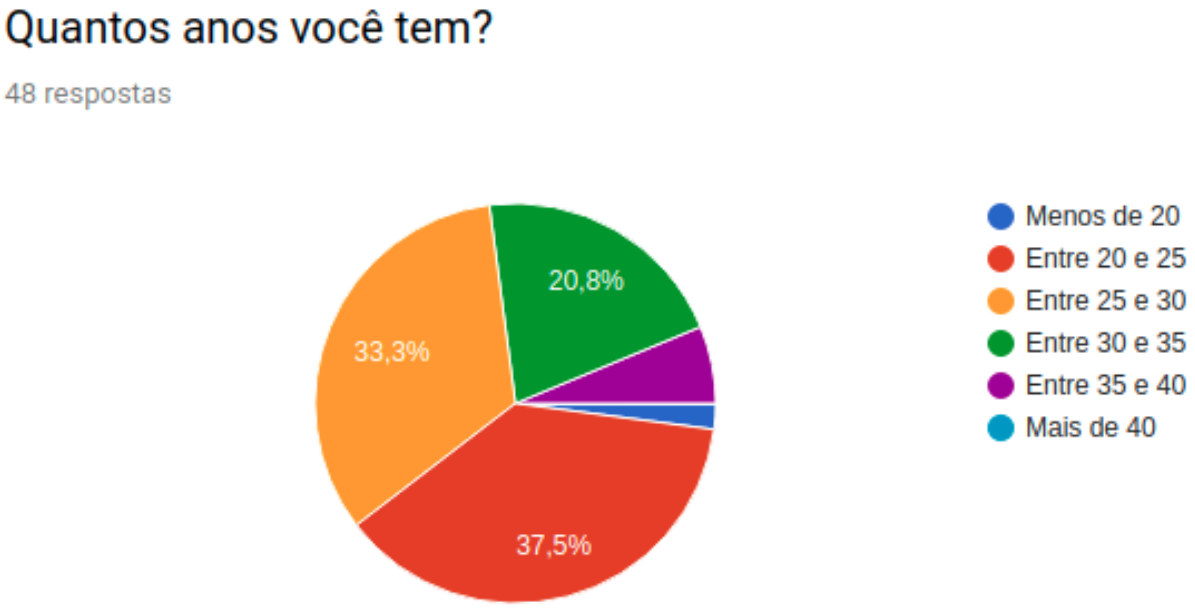
Figura 23 – Pergunta 02 do questionário

Fonte: Autoria própria por Google Forms.

A pergunta 02, obteve 50% das respostas na opção “Entre 1 e 5 anos”.

A pergunta 03 do questionário, conforme a figura 25, e diz respeito a “Quantos anos você tem?”. Ela se refere a conhecer melhor o perfil dos entrevistados.

Figura 24 – Pergunta 03 do questionário



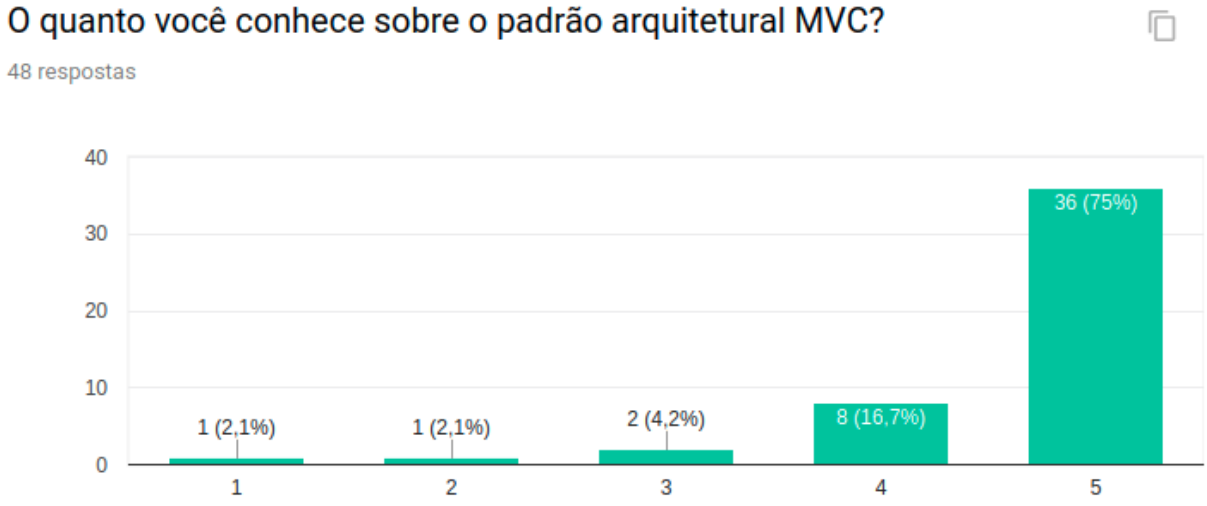
Fonte: Autoria própria por Google Forms.

A pergunta 03, obteve 37,5% das respostas na opção “Entre 20 e 25” e 33,3% das respostas na opção “Entre 25 e 30”.

As próximas 4 perguntas do questionário se referem a quantidade de conhecimento sobre os tipos de padrões arquiteturais apresentados. Ela se refere a entender o nível de conhecimento dos entrevistados aos tipos de padrões arquiteturais apresentados.

A pergunta 04 do questionário, conforme a figura 26, se refere a “O quanto você conhece sobre o padrão arquitetural MVC?”.

Figura 25 – Pergunta 04 do questionário

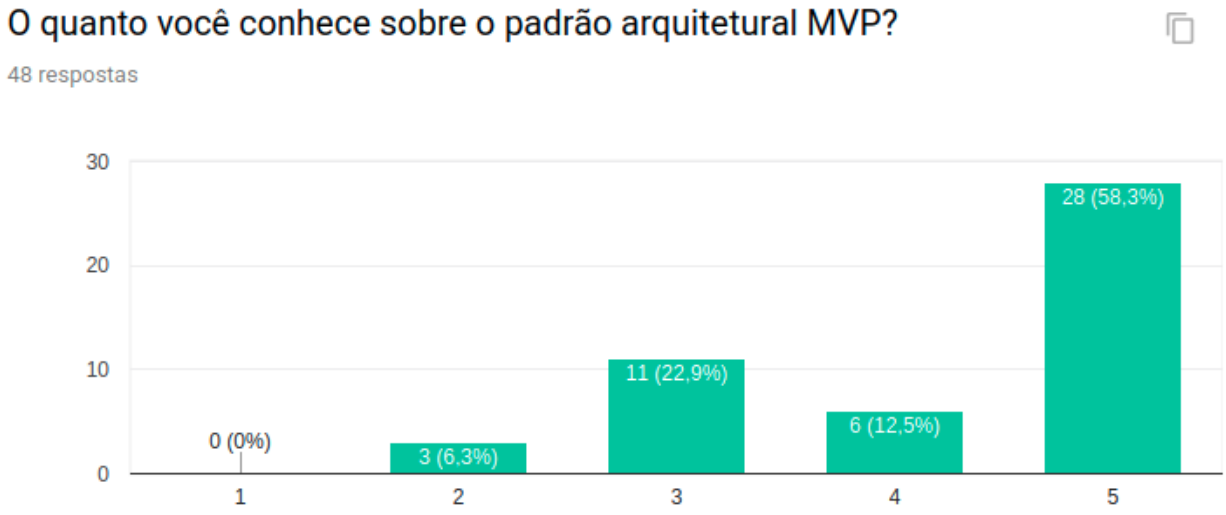


Fonte: Autoria própria por Google Forms.

A pergunta 04, obteve 75% das respostas na opção “5” e 16,7% das respostas na opção “4”, considerando 1 como “Nunca ouvi falar” e 5 como “Já utilizei em projetos”. De acordo com essas repostas, o padrão MVC foi o que se mostrou mais conhecido e mais utilizado pelos entrevistados.

A pergunta 05 do questionário, conforme a figura 27, se refere a “O quanto você conhece sobre o padrão arquitetural MVP?”.

Figura 26 – Pergunta 05 do questionário

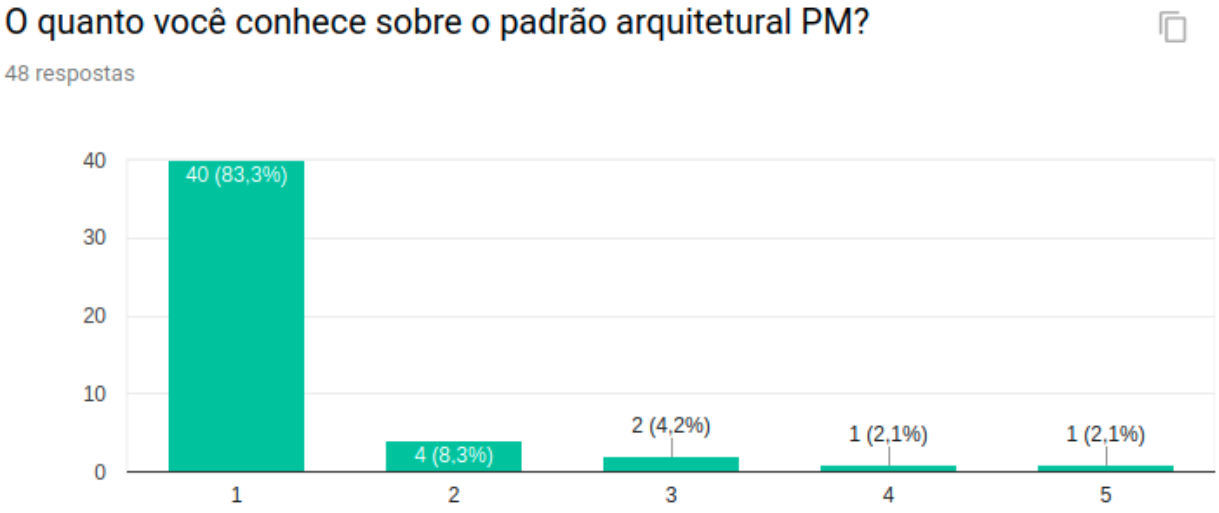


Fonte: Autoria própria por Google Forms.

A pergunta 05, obteve 58,3% das respostas na opção “5” e 22,9% das respostas na opção “3”, considerando 1 como “Nunca ouvi falar” e 5 como “Já utilizei em projetos”.

A pergunta 06 do questionário, conforme a figura 28, se refere a “O quanto você conhece sobre o padrão arquitetural PM?”.

Figura 27 – Pergunta 06 do questionário

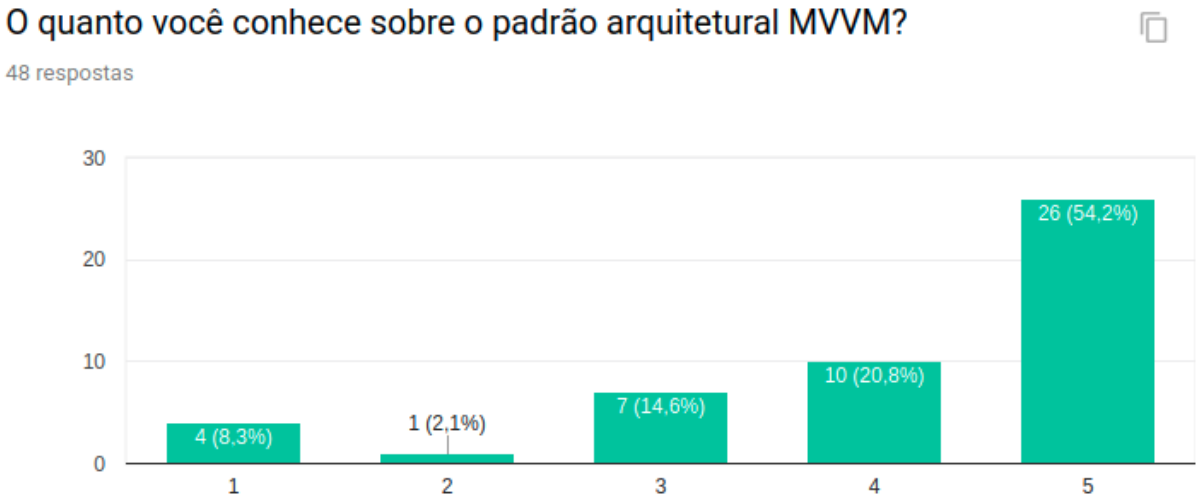


Fonte: Autoria própria por Google Forms.

A pergunta 06, obteve 83,3% das respostas na opção “1” e 8,3% das respostas na opção “2”, considerando 1 como “Nunca ouvi falar” e 5 como “Já utilizei em projetos”. Esse padrão se mostrou como o padrão menos conhecido pelos entrevistados.

A pergunta 07 do questionário, conforme a figura 29, se refere a “O quanto você conhece sobre o padrão arquitetural MVVM?”.

Figura 28 – Pergunta 07 do questionário

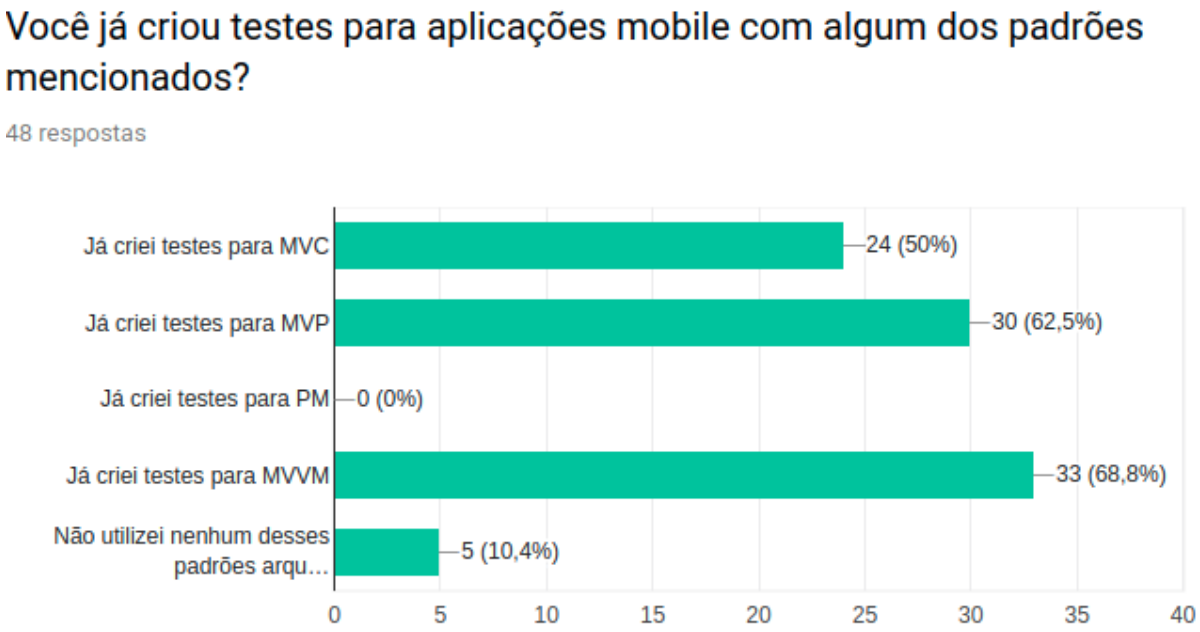


Fonte: Autoria própria por Google Forms.

A pergunta 07, obteve 54,2% das respostas na opção “5” e 20,8% das respostas na opção “4”, considerando 1 como “Nunca ouvi falar” e 5 como “Já utilizei em projetos”.

A pergunta 08 do questionário, conforme a figura 30, se refere a testabilidade dos padrões arquiteturais apresentados, em que a pergunta é “Você já criou testes para aplicações mobile com algum dos padrões mencionados?”.

Figura 29 – Pergunta 08 do questionário



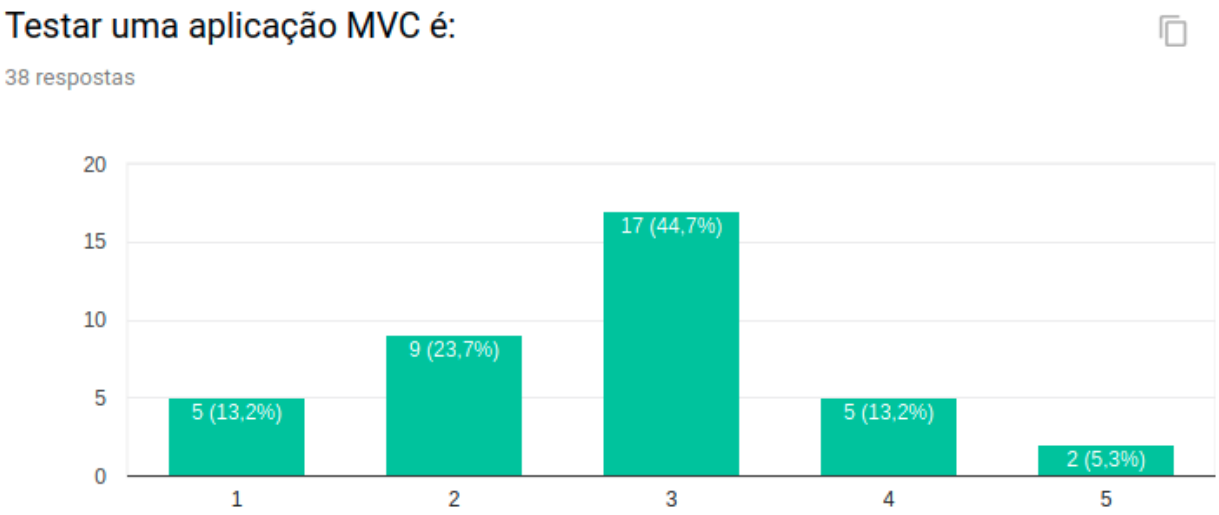
Fonte: Autoria própria por Google Forms.

A pergunta 08, obteve 68,8% das respostas na opção “Já criei testes para MVVM” que foi a mais votada. A segunda resposta mais votada foi “Já criei testes para MVP” com 62,5% de votação. Com isso se pode ver que o padrão mais testado foi o MVVM, e nenhum dos entrevistados criou testes para o padrão PM.

As próximas 4 perguntas do questionário se referem a quantidade de conhecimento sobre os tipos de padrões arquiteturais apresentados e não são obrigatórias para a submissão do questionário pois dependem da prática com os testes de cada padrão. Ela se refere a entender a prática de testabilidade que cada entrevistado obteve durante seu período de experiência.

A pergunta 09 do questionário, conforme a figura 31, se refere a entender o grau da dificuldade de testar um padrão arquitetural MVC com a pergunta “Testar uma aplicação MVC é:” onde as respostas vão de 1 a 5, sendo que a resposta 1 corresponde a “Muito difícil” e 5 corresponde a “Muito fácil”.

Figura 30 – Pergunta 09 do questionário

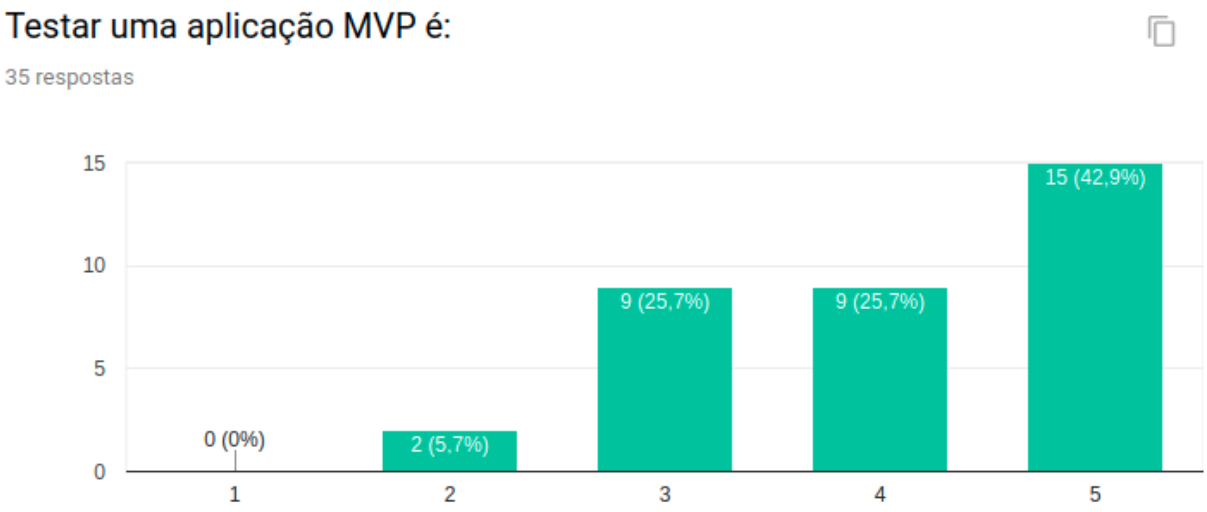


Fonte: Autoria própria por Google Forms.

A pergunta 09, obteve 44,7% das respostas na opção “3” e 23,7% das respostas na opção “2”, considerando 1 como “Muito difícil” e 5 a “Muito fácil”.

A pergunta 10 do questionário, conforme a figura 32, se refere a entender o grau da dificuldade de testar um padrão arquitetural MVP com a pergunta “Testar uma aplicação MVP é:” onde as respostas vão de 1 a 5, sendo que a resposta 1 corresponde a “Muito difícil” e 5 corresponde a “Muito fácil”.

Figura 31 – Pergunta 10 do questionário

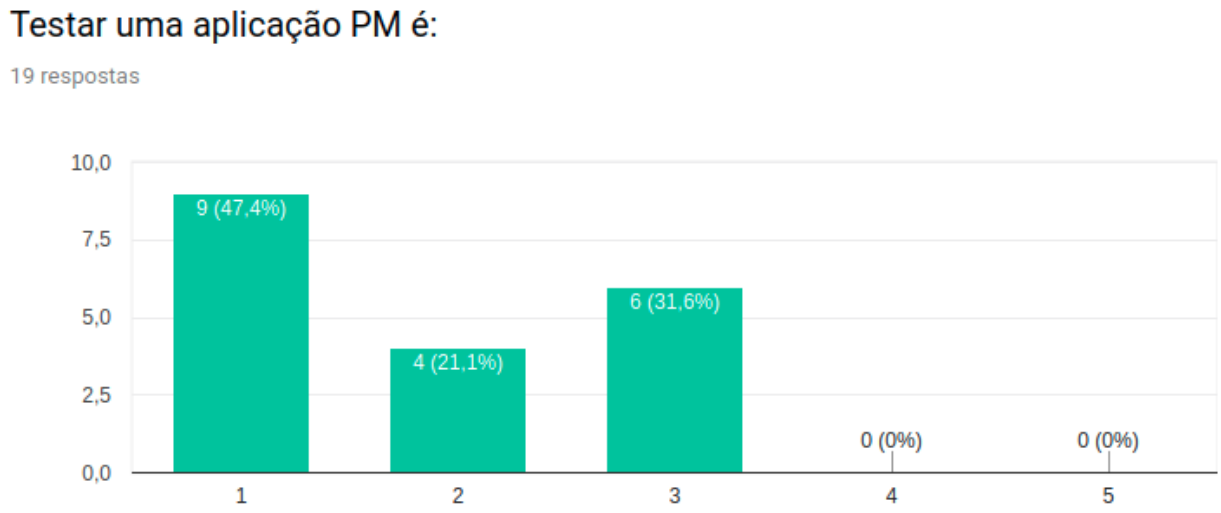


Fonte: Autoria própria por Google Forms.

A pergunta 10, obteve 42,9% das respostas na opção “5” e 25,7% tanto na opção “4” quanto na opção “3”, considerando 1 como “Muito difícil” e 5 a “Muito fácil”.

A pergunta 11 do questionário, conforme a figura 33, se refere a entender o grau da dificuldade de testar um padrão arquitetural PM com a pergunta “Testar uma aplicação PM é:” onde as respostas vão de 1 a 5, sendo que a resposta 1 corresponde a “Muito difícil” e 5 corresponde a “Muito fácil”.

Figura 32 – Pergunta 11 do questionário

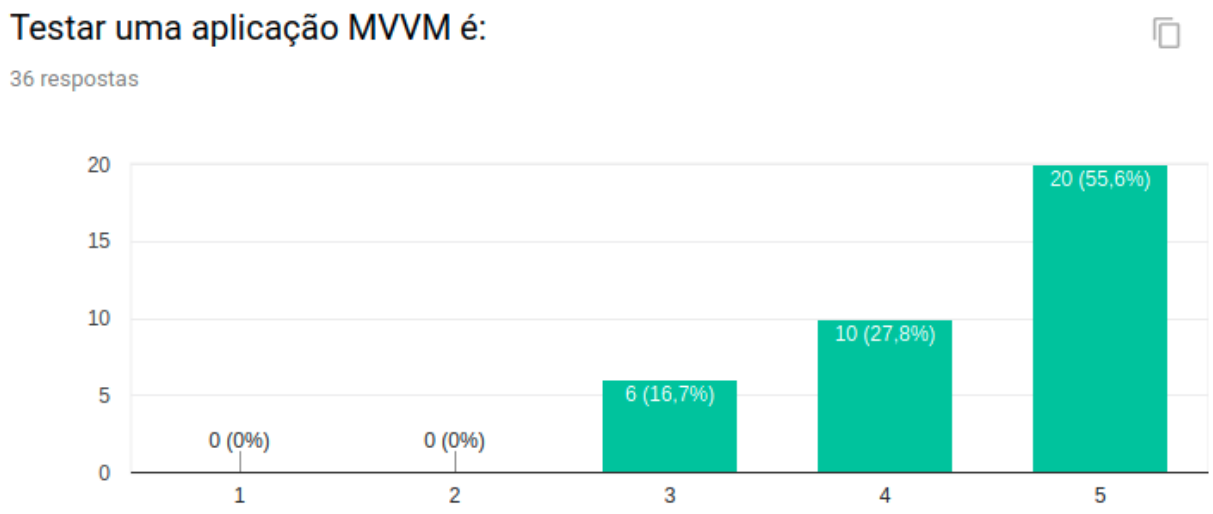


Fonte: Autoria própria por Google Forms.

A pergunta 11, obteve 47,4% das respostas na opção “1” e 31,6% na opção “3”, considerando 1 como “Muito difícil” e 5 a “Muito fácil”, mesmo considerando que nenhuma das pessoas entrevistadas tiveram prática com construção de teste para padrões PM.

A pergunta 12 do questionário, conforme a figura 34, se refere a entender o grau da dificuldade de testar um padrão arquitetural MVVM com a pergunta “Testar uma aplicação MVVM é:” onde as respostas vão de 1 a 5, sendo que a resposta 1 corresponde a “Muito difícil” e 5 corresponde a “Muito fácil”.

Figura 33 – Pergunta 12 do questionário

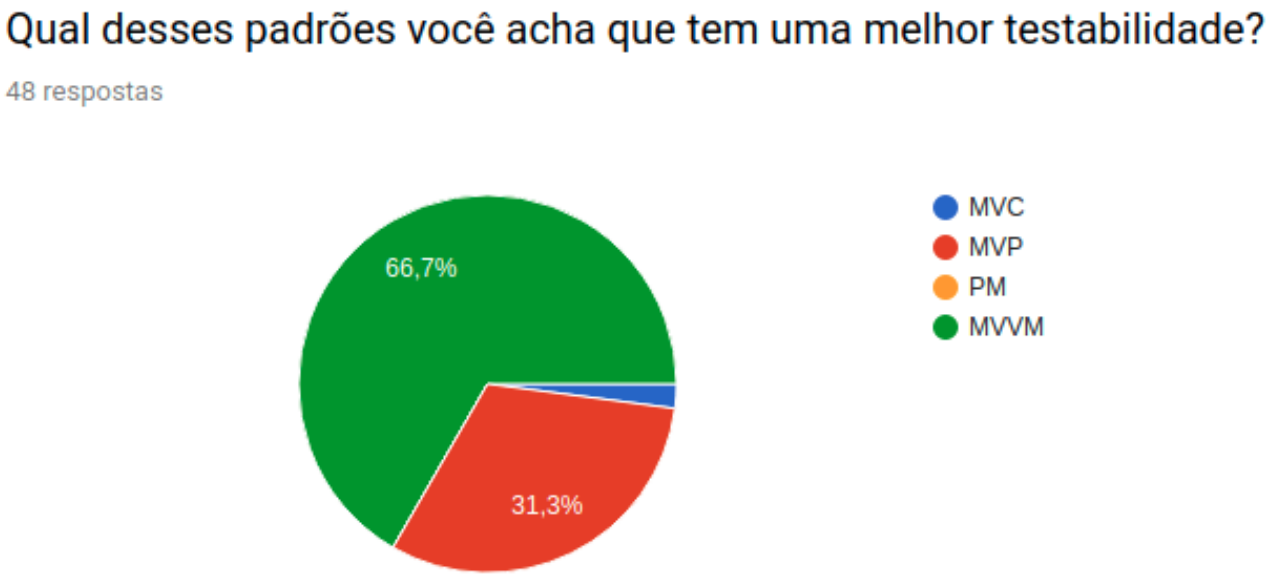


Fonte: Autoria própria por Google Forms.

A pergunta 12, obteve 55,6% das respostas na opção “5” e 27,8% na opção “4”, considerando 1 como “Muito difícil” e 5 a “Muito fácil”. Nenhum dos entrevistados selecionou as opções “1” e “2” para caracterizar que o padrão MVVM é difícil ou muito difícil de se criar testes.

A pergunta 13 do questionário, conforme a figura 35, se refere a entender a comparação de cada entrevistado, considerando o tempo de experiência e a prática de testabilidade, entre cada um dos padrões arquiteturais apresentados.

Figura 34 – Pergunta 13 do questionário



Fonte: Autoria própria por Google Forms.

A pergunta 13, obteve 66,7% das respostas na opção “MVVM” que foi a mais votada. A segunda resposta mais votada foi “MVP” com 31,3% de votação. A terceira opção mais votada foi o padrão “MVC” com o total de 1 votação representando 2,1% do gráfico.

Nas perguntas 10 e 12 do questionário, pode-se analisar que os dois padrões, tanto o MVP e MVVM, podem ser considerando dois padrões que tem uma facilidade maior na criação de testes. Mesmo que os resultados da pergunta 10 foram positivos para o padrão MVP, na pergunta 13, pode-se perceber a preferência para o padrão MVVM.

5.5 ESCOLHA DO PADRÃO ARQUITETURAL

Esse capítulo explica sobre a definição do padrão arquitetural para o desenvolvimento do aplicativo proposto.

Para fazer a escolha do padrão arquitetural é necessário levantar um ponto importante que afeta a decisão da escolha. Nesta monografia foi apresentado o projeto inicial

de um aplicativo. Esse aplicativo foi planejado para ser desenvolvido continuamente e futuramente apresentar diversas ferramentas e *features*.

Considerando o estudo feito nesta monografia, os critérios a serem avaliados para a escolha do padrão arquitetural a ser implementado no aplicativo descrito anteriormente são: testabilidade do aplicativo; facilidade para criar testes; criação de códigos com o mínimo de acoplamento; menor dependência possível de comunicação entre as camadas; automatização da comunicação entre as camadas; e quantidade de pontos positivos e negativos apresentados na tabela 4, análises e resultados obtidos no questionário.

Levando em conta todos os critérios e os padrões arquiteturais avaliados nesta monografia, o padrão arquitetural selecionado para a implementação do aplicativo apresentado nesta monografia é o padrão arquitetural MVVM.

6 CONCLUSÃO

Ao longo da pesquisa desta monografia foram analisados diversos conceitos, processos e padrões, levantando alguns padrões que podem ser utilizados para o desenvolvimento mobile, tais como os padrões: o *Broker Patterns, Layers Patterns, Pipes and Filters*, MVC, MVP, PM, MVVM, entre outros. Desta forma cumprindo o objetivo 1 desta monografia.

Para o objetivo 2, foram coletadas as características específicas de cada padrão arquitetural de várias referências reconhecidas pela área de desenvolvimento. Assim, foi possível fazer a análise das características de cada um dos padrões arquiteturais selecionados anteriormente. Os autores referenciados levantam pontos positivos e pontos negativos sobre os padrões arquiteturais listados nesta monografia. Entre esses os pontos, foram levantados e coletados afirmações sobre a estrutura dos padrões, a definição de cada uma das camadas, pontos como a comunicação entre cada um dos padrões, a dependência entre cada camada e, também, foram levantados pontos positivos e negativos individuais sobre a testabilidade dos padrões arquiteturais apresentados.

Conforme especificado no objetivo 3, para ter a melhor escolha sobre o padrão arquitetural a ser utilizado na solução do aplicativo proposto, foi necessário apresentar a aplicação de todos os padrões arquiteturais listados na monografia.

Considerando a criação e aplicação dos padrões na solução do aplicativo proposto, pode-se concluir que sempre será divido em três camadas principais e manterá os mesmos e arquivos de classes e interface do usuário.

Outra conclusão para o objetivo 3, é o fato de que os padrões demonstrados nas figuras 21 e 24, representam respectivamente o padrão MVP e o padrão MVVM, serem os padrões que menos apresentaram ligações e dependências entre cada camada dos padrões e segundo as referências bibliográficas feitas ao longo desta monografia, quando menos acoplamento e dependência sobre as camadas, melhor será a solução proposta.

Para o objetivo 4, foi feita uma análise entre os padrões arquiteturais apresentados para definir qual o padrão com uma maior testabilidade e que mais condiz com a solução do aplicativo proposto. Foram analisados vários critérios e com isso, concluído que primeiramente, antes de começar a implementar a aplicação, é necessário planejar, mapear as possíveis *features* e organizá-lo. Além disso, é preciso que a aplicação seja escalável com a solução para que a produtividade das pessoas que programam nunca diminuir, mas sim, sempre aumentar. Outro

ponto importante é garantir a responsabilidade única entre cada camada dos padrões e ter o mínimo de acoplamento de dependência. A análise de todos esses critérios resultou na escolha do padrão arquitetural MVVM para a implementação da solução proposta.

O objetivo 5 se refere a avaliação e análise feita via questionário com profissionais da área. Para atender esse objetivo, foi aplicado o questionário para 48 entrevistados durante 13 dias. Com a aplicação deste questionário é possível levantar algumas informações relevantes.

A primeira conclusão retirada do questionário é sobre o padrão arquitetural mais conhecido. Pode-se afirmar que o padrão mais conhecido entre as pessoas entrevistadas é o padrão MVC com um total de 46 pontos com as opções escolhidas entre “3”, “4” e “5”. O segundo padrão mais conhecido entre os entrevistados é o MVP com um total de 45 pontos com as opções escolhidas entre “3”, “4” e “5”.

Também pode-se validar a prática que o entrevistado teve com cada um dos padrões arquiteturais, como foi feito na pergunta de número 08 do questionário. Assim, foi obtido 33 votos para a resposta “Já criei testes para MVVM”, concluindo que o padrão que os entrevistados tiveram mais contato e prática foi o padrão MVVM, seguido pelo padrão MVP com 30 votos e o padrão MVC com 24 votos.

Considerando a facilidade de criação de testes, o padrão MVC obteve a 31 pontos com as opções escolhidas entre “1”, “2” e “3”, caracterizando-o como o padrão com maior dificuldade de criação de testes. O segundo padrão com maior dificuldade na criação de testes é o padrão MVP com um total de 19 pontos com as opções escolhidas entre “1”, “2” e “3”.

Ainda considerando a facilidade de criação de teste, o padrão MVVM se caracterizou como o padrão com uma maior facilidade para criação de testes com um total de 36 pontos com as opções escolhas entre “3”, “4” e “5” e tendo como segundo colocado, o padrão MVP contendo 33 pontos com as opções escolhidas entre “3”, “4” e “5”.

Outro ponto levantado neste questionário são os padrões MVP e MVVM que foram os padrões mais escolhidos na pergunta de número 13 que pergunta qual dos padrões pode ser considerado o padrão com uma melhor testabilidade. Com isso, se conclui que o padrão MVVM é o padrão com uma melhor testabilidade, pois foi o mais escolhido, contendo 66,7% de votos.

Assim, foi atingido todos os objetivos desta monografia.

6.1 TRABALHOS FUTUROS

Como proposta de trabalhos futuros, é indicado desenvolver o aplicativo proposto para se ter um protótipo e torna-lo funcional, além da implementação de testes unitários. Outro ponto indicado são os testes de usabilidade nas telas para tornar o aplicativo a ter mais facilidade em sua usabilidade.

REFERÊNCIAS

AAHN INFOTECH. **Architectural Patterns.** Disponível em: https://web.archive.org/web/20120623081009/http:/aahninfotech.com/arct_pattern.html. Acesso em: 13 maio 2018.

BUSCHMANN, Frank et al. **Pattern-Oriented Software Architecture:** A system of patterns. Of Siemens Ag, Germany: John'wiley & Sons Ltd, 1996.

CELEPAR. **Arquiteturas para Aplicações Móveis.** Paraná: Celepar, 2011. Disponível em: <http://www.documentador.pr.gov.br/documentador/pub.do?action=d&uuid=@gtf-escriba@758c86db-5c88-43ac-9e90-36880790c56c>. Acesso em: 13 maio 2018.

FOWLER, Martin. **Presentation Model**. Disponível em: <https://martinfowler.com/eaaDev/PresentationModel.html>. Acesso em: 20 maio 2018.

GAMMA, Erich et al. **Design Patterns:** Elements of Reusable Object-Oriented Software. Addison-wesley, 1994. 395 p.

GARLAN, David; SHAW, Mary. An Introduction to Software Architecture. 2017. 39 p. - Curso de Ciências da Computação, School Of Computer Science, Carnegie Mellon University, Pittsburgh, 2017.

GIL, Antônio Carlos**. Métodos e técnicas de pesquisa social**. 6 eds. São Paulo: Atlas, 2008. 216 p. Disponível em: <https://ayanrafael.files.wordpress.com/2011/08/gil-a-c-mc3a9todos-e-tc3a9cnicas-de-pesquisa-social.pdf>. Acesso em: 1 jun. 2018.

HALL, Gary Mclean. **Pro WPF and Silverlight MVVM:** Effective Application Development with Model-View-ViewModel. New York: Apress, 2010.

IIDA, Hajimu. **Pattern-Oriented Approach to Software Process Evolution.** Nara: Information Technology Center, Nara Institute of Science and Technology, 1999.

JACOBS, Bart. **Mastering MVVM With Swift**. 2017.

KOURAKLIS, John. **MVVM in Delphi:** Architecting and Building Model View ViewModel Applications. London, United Kingdom: Apress, 2016.

LEE, Valentino. **Aplicações móveis: arquitetura, projeto e desenvolvimento**. 1ª edição, 2005. Pearson Education do Brasil, São Paulo.

MARTIN, Robert C. **Agile Software Development:** Principles, Patterns, and Practices. New Jersey: Pearson Prentice Hall, 2002.

MARTIN, Robert C. **Clean Architecture:** A Crafstman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p. (Robert C. Martin Series).

MEDVIDOVIC, Nenad; TAYLOR, Richard N. **Software architecture:** foundations, theory, and practice. Cape Town, South Africa, p.471-472, maio 2010.

MICROSOFT (Org.). **Presentation Model.** 2010. Disponível em: <https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff921080(v=pandp.10)>. Acesso em: 21 maio 2018.

MIHAILESC, Marius Iulian. **Model-View-ViewModel (MVVM) Design Pattern using Windows Presentation Foundation (WPF) Technology.** 2010.

MORRIS, Ben. **The Symbian OS Architecture Sourcebook**: Design and Evolution of a Mobile Phone OS. Hoboken: John Wiley & Sons Inc, 2007.

PAN, Haihong et al. **A Scalable Graphics User Interface Architecture for CNC Application based - on WPF and MVVM**. Switzerland: Trans Tech Publications, 2011. 6 p.

PILAR, Carina Ponzoni. **Avaliação das Arquiteturas de Desenvolvimento para Dispositivos Móveis.** Curso de Sistemas de Informação, Universidade de Caxias do Sul, Caxias do Sul, 2013, 89 p. Disponível em: https://repositorio.ucs.br/xmlui/bitstream/handle/11338/1245/TCC Carina Ponzoni Pilar.pdf?sequence=1&isAllowed=y>. Acesso em: 13 maio 2018.

SILVA, Edna Lúcia, MENEZES, Estera Muszkat Menezes . **Metodologia da pesquisa e elaboração de dissertação**, UFSC Florianópolis, 4 Edição, 2005.

SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

SOMMERVILLE, Ian. **Engenharia de software**. 8. ed. São Paulo: Pearson Addison Wesley, 2007.

TARKOMA, Sasu. **Mobile Middleware:** Architecture, Patterns and Practice. Finland: John Wiley & Sons Ltd, 2009.

VASANTA, Rajaneesh. **Thin Client vs Thick Client Mobile Applications**. Disponível em: <http://mobile-application-testing.blogspot.com.br/2011/05/thin-client-vs-thick-client-mobile.html>. Acesso em: 13 maio 2018.

VON HALLE, Barbara. **Business Rules Applied: Building Better Systems Using the Business Rules Approach**. Wiley, 2001

WILSON, Kristopher. **The Clean Architecture in PHP**. Disponível em: <https://phutai.me/wp-content/uploads/2016/10/The-Clean-Architecture-in-PHP-Kristopher-Wilson.pdf>. Acesso em: 13 maio 2018 Leanpub, 2015.