UNIVERSIDADE FEDERAL DE OURO PRETO
ESCOLA DE MINAS
COLEGIADO DO CURSO DE ENGENHARIA DE CONTROLE
E AUTOMAÇÃO - CECAU

ANNA CRISTYNA MARTINS BARROS

ESTUDO DE CASO: MODERNIZAÇÃO INCREMENTAL DE UMA INTERFACE REFERENTE A COLETA DE DADOS DE PUBLICAÇÕES

MONOGRAFIA DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Ouro Preto, 2022

ANNA CRISTYNA MARTINS BARROS

ESTUDO DE CASO: MODERNIZAÇÃO INCREMENTAL DE UMA INTERFACE REFERENTE A COLETA DE DADOS DE PUBLICAÇÕES

Monografia apresentada ao Curso de Engenharia de Controle e Automação da Universidade Federal de Ouro Preto como parte dos requisitos para a obtenção do Grau de Engenheiro de Controle e Automação.

Orientador: Prof. Dr. Bruno Nazário Coelho

Coorientador: Prof. Dr. Rodrigo Geraldo Ribeiro

Ouro Preto
Escola de Minas – UFOP
2022

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

B277e Barros, Anna Cristyna Martins.
Estudo de caso [manuscrito]: modernização incremental de uma interface referente a coleta de dados de publicações. / Anna Cristyna Martins Barros. - 2022.
48 f.

Orientador: Prof. Dr. Bruno Nazário Coelho.
Coorientador: Prof. Dr. Rodrigo Geraldo Ribeiro.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Escola de Minas. Graduação em Engenharia de Controle e Automação .
Área de Concentração: Engenharia de Controle e Automação.

1. software. 2. Evolução. 3. Automação. I. Coelho, Bruno Nazário. II. Ribeiro, Rodrigo Geraldo. III. Universidade Federal de Ouro Preto. IV. Título.

CDU 681.5

Bibliotecário(a) Responsável: Angela Maria Raimundo - SIAPE: 1.644.803

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
ESCOLA DE MINAS
DEPARTAMENTO DE ENGENHARIA CONTROLE E AUTOMACAO

FOLHA DE APROVAÇÃO

Anna Cristyna Martins Barros

Estudo de caso: modernização incremental de uma interface referente a coleta de dados de publicações

Monografia apresentada ao Curso de Engenharia de Controle e Automação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Engenheiro de Controle e Automação

Aprovada em 12 de Janeiro de 2022

Membros da banca

Doutor – Bruno Nazário Coelho - Orientador – Universidade Federal de Ouro Preto
Doutor – Rodrigo Geraldo Ribeiro - Coorientador – Universidade Federal de Ouro Preto
Doutora – Adrielle de Carvalho Santana - Universidade Federal de Ouro Preto
Doutor – Paulo Marcos de Barros Monteiro - Universidade Federal de Ouro Preto

Bruno Nazário Coelho, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 31/01/2022

Documento assinado eletronicamente por **Bruno Nazário Coelho**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 31/01/2022, às 18:06, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0267843** e o código CRC **B7019599**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: 3135591533 - www.ufop.br

Este trabalho é dedicado aos meus pais que, com amor e extrema dedicação, me proporcionou essa dádiva que é viver e fazer parte da vida deles.

AGRADECIMENTOS

À Deus, pela proteção, vida e saúde que Ele me proporcionou. Sei que tudo está nas mãos Dele, e que embora o caminho pareça escuro, Ele estará sempre a nos guiar.

À minha mãe, a quem sou imensamente grata pela infinita dedicação e amor por mim. Suportando a tarefa de me criar sozinha em minha adolescência até os dias de hoje. Todo crescimento e todas conquistas são devotos a ela. Agradeço por me ensinar o caminho da verdade e da vida, por me ensinar todos os princípios mais valorosos e por me ensinar a lidar não só com as flores, mas também com as pedras.

Ao meu pai (*In memoriam*), pela dedicação em propiciar o melhor futuro para mim, por sempre preocupar comigo e com meus estudos. Por ser o melhor companheiro que pode ser para minha mãe.

Ao grandes mestres da universidade, que me ensinaram não só os conceitos das disciplinas mas muitos conceitos da matéria da vida. A dedicação e o empenho em aprender e ensinar me mostraram o caminho a seguir e onde eu quero chegar.

Aos grandes mestres, desde do jardim da infância até o Ensino Médio, por me envolverem em seus ensinamentos e reflexões que me ensinaram a viver e amar a busca por conhecimento.

Aos meus amigos da Universidade, por todo apoio e ajuda durante esse período que passamos juntos. Foi uma etapa a qual tenho prazer em ter em minha memória, e meu coração se alegra em saber que tive pessoas tão especiais nesse percurso. Agradeço todo momento, convívio e por todas as risadas compartilhadas.

Aos meus amigos da empresa, por todo apoio, paciência em me ensinar as tecnologias e os métodos da empresa. Por me mostrarem os valores importantes para se ter no trabalho e em todas as áreas da vida: respeito, cooperação, convergir visões, curiosidade, transparência, honra e autoconhecimento. Sou grata, em especial, ao Bryan por todo aprendizado, por ser meu professor, parceiro de trabalho e amigo. E ao Rodolfo, por todo companheirismo, amizade, apoio e dedicação em me ajudar.

À minha tia Marlene (*In memoriam*) que sempre sonhou com a conclusão dessa fase da minha vida e me teve como filha, me ensinando desde de princípios até a base de matemática que tenho hoje. Por todos os mimos e alegria que ela me proporcionou durante minha infância.

"Diga-me e eu esquecerei; ensina-me e eu poderei lembrar; envolva-me e eu aprenderei."
(Benjamin Franklin)

RESUMO

A crescente demanda por ferramentas mais intuitivas e performáticas gerou, nos últimos anos, uma grande quantidade de bibliotecas e *frameworks* para a resolução de problemas que afligem as empresas. Com o mercado de *software* competitivo, muitas versões são lançadas, atualizadas e depreciadas. Dessa forma, os desenvolvedores de *softwares* precisam ficar atentos à novas ferramentas interessantes e em caso de novas versões conflitantes. A manutenção e a evolução são necessárias para prolongar o ciclo de vida dos *softwares* e mantê-los competitivos. O presente trabalho faz um estudo de caso de uma modernização incremental de uma parte considerada legada do código. Tal parte está relacionada à renderização da interface responsável por mostrar o consumo de limite de coleta de publicações de redes sociais. Além disso, essa parte também é responsável por lidar com as regras de negócios envolvidas no consumo da coleta. A modernização foi realizada em conjunto com uma nova funcionalidade no sistema. Esta nova funcionalidade muda a forma como a contagem é feita e muda também o *layout* da interface. Apesar de aumentar significativamente a estimativa de tempo da tarefa, a nova versão se mostrou mais intuitiva e de rápido desenvolvimento. A tarefa abordada no presente trabalho foi importante para a adequação do código legado, cuja versão da linguagem estava em processo de depreciação.

Palavras-chaves: *software*, modernização, evolução.

ABSTRACT

The growing demand for more intuitive and performant tools has generated, in recent years, a large number of libraries and frameworks for solving problems that afflict companies. With the competitive software market, many versions are released, updated, and deprecated. In this way, software developers need to be aware of new interesting tools and in case of new conflicting versions. Maintenance and evolution are necessary to prolong the life cycle of software and keep it competitive. The present work makes a case study of an incremental modernization of a part considered legacy of the code. This part is related to the rendering of the interface responsible for showing the consumption of the collection limit of social network publications. In addition, this part is also responsible for handling the business rules involved in consuming the collection. The modernization was carried out in conjunction with new functionality in the system. This new feature changes the way counting is done and also changes the interface layout. Despite significantly increasing the time estimate of the task, the new version proved to be more intuitive and faster to develop. The task addressed in the present work was important for the adequacy of the legacy code, whose language version was in the process of being deprecated.

Key-words: software, modernization, evolution.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
DS	*Design System*
JS	JavaScript
GAE	*Google App Engine*
IBM	*International Business Machines Corporation*
Paas	Plataforma como serviço
Py	Python
QA	Quality Assurance
TI	Tecnologia da Informação
TDD	*Test Driven Development*

SUMÁRIO

1 INTRODUÇÃO

A crescente utilização da tecnologia pelas empresas gerou maior demanda por aplicações com características, identidade e peculiaridades próprias. Assim, novas necessidades surgiram para atender as regras de negócio. Novas linguagens de programação e ferramentas foram criadas a fim de atender a essas necessidades e propiciar mais praticidade e performance aos desenvolvedores e usabilidade aos usuários.

A digitalização dos processos de forma geral causou uma explosão de demanda por profissionais de TI[1], fato evidenciado principalmente no ano de 2020, onde até empresas muito tradicionais buscaram essa adesão em busca de sobrevivência.

Isso é o que mostra o estudo feito pela IBM "COVID-19 e o futuro dos negócios", onde 59% das organizações afirmam que a pandemia acelerou o processo de transformação digital delas e 66% afirmaram que havia resistência nas iniciativas da migração tecnológica (IBM, 2020).

O estudo mencionado, "COVID-19 e o futuro dos negócios"reuniu contribuições de ao menos 3.800 executivos em 20 países e 22 setores, e também apontou que 94% deles pretendem possuir modelos de negócios baseados em plataforma até 2022. Houve mudanças significativas de priorização nas organizações, onde 62% enfatizam a transformação digital. Sendo que essa estratégia não é uma solução transitória, 55% dos participantes dizem que a pandemia resultou em “mudanças permanentes em nossa estratégia organizacional”. Como destaque, a solução de *Cloud* e automação são as mais adotadas pelos colaboradores da pesquisa. Além disso, 75% planejam priorizar a resiliência de TI ao longo dos próximos dois anos (IBM, 2020).

O benefício consequente da digitalização é a redução de custos, maior competitividade e funcionários resilientes (IBM, 2020). Porém, o processo de adequação à plataforma, ou seja, disponibilizar seu produto online, não é rápido, demandam planejamento detalhado, contratação de uma empresa que dispõe desse serviço e capacitação e treinamento dos funcionários da empresa. E além disso, precisam ficar atentos ao lançamento de novas versões e ferramentas e notícias a respeito das tecnologias escolhidas.

O desenvolvimento de aplicações está em constante atualização, tecnologias envolvidas possuem ciclos menores de vida, sendo substituídos por uma nova versão ou por alguma concorrente. Através do video Statistics e Data (2020), onde o autor agrupou dados e estatísticas sobre as linguagens mais populares de 1965 até 2021 e os projetou de forma a visualizar a evolução conforme o tempo. Inicialmente, FORTRAN, COLBOL e ALGOL eram as mais populares, até meados de 1970. Após isso, não só o uso de linguagens de programação aumentou

[1] Tecnologia da informação: área responsável por recursos de computação que visam a produção, o armazenamento, a transmissão, o acesso, a segurança e o uso das informações.

como também observa-se surgimento de novas linguagens, de nível mais alto, que substituíram outras que possuíam padrões considerados mais rígidos e obsoletos, com estruturas semelhantes à de máquina. Ao final do video, percebe-se que *Java*, *JavaScript* e *Python* são as que mais se destacam nos últimos anos.

Além disso, para que essas tecnologias permaneçam no mercado, diversos profissionais promovem novas funcionalidades. Os *Frameworks*[2] e bibliotecas são desenvolvidos como *open source* e são disponibilizados para que os desenvolvedores possam ter mais facilidade e dinamismo ao seu trabalho, corroborando para a codificação mais rápida.

Além disso, muitos profissionais promovem novos estudos de técnicas e de metodologias para promover dinamismo, saúde e resiliência à equipe e ao desenvolvimento. Dessa forma, melhoram seus produtos, tornando mais escaláveis e intuitivos, o que ocasiona diversificação de soluções para os diversos problemas enfrentados.

Tendo como exemplo, a linguagem *JavaScript*, Greif e Benitte (2020) reuniu contribuições de 23.765 desenvolvedores em 137 países para entender melhor a utilização, popularidade e opiniões sobre bibliotecas JS e *Frameworks* desde de 2016 até 2020. Na figura 1 resume-se o levantamento, mostrando a relação ao longo desses anos, entre o conhecimento da tecnologia com o sentimento das opiniões dos desenvolvedores que as utilizaram ou que querem as utilizar. Ao observá-la, é possível perceber o perfil de uso de cada uma, perceber padrões e tendências. Muitas nascem com uma perspectiva neutra, em consequência, possuem pouco engajamento para o uso, como Koa.

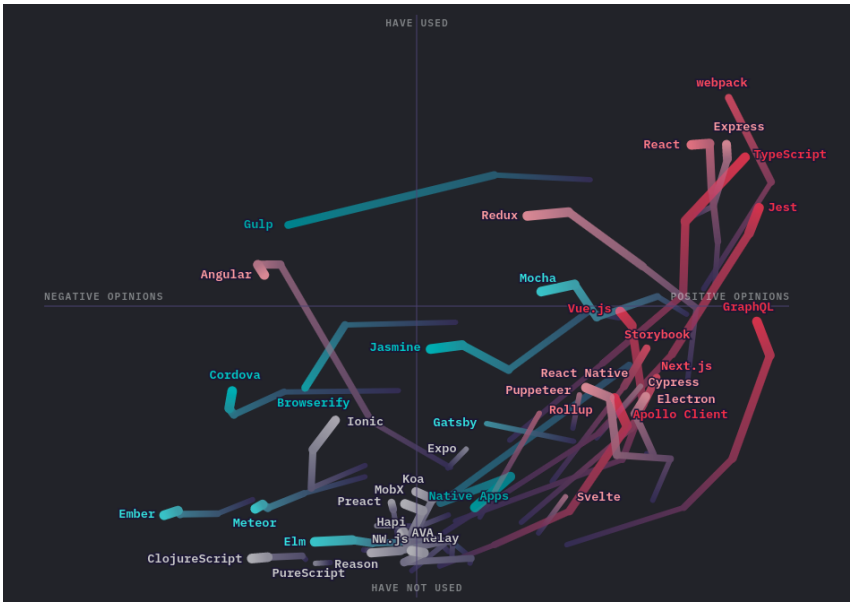

Figura 1 – Correlação das tecnologias JS entre uso e opiniões

Fonte: Greif e Benitte (2020)

[2] Pacote de códigos genéricos que oferecem funcionalidades para facilitar o desenvolvimento do software

Já outras ganham popularidade bem rápido, se mostrando em uma curva quase exponencial, a exemplo do Webpack e Jest, tendências que podem influenciar as diretrizes do desenvolvimento em *JavaScript*. Após o aumento da utilização, é possível notar, também, que o sentimento positivo associado diminui, isto indica que há alguns pontos negativos percebidos com a prática, o que induz as ferramentas a avaliar tais contras e lançar novas melhorias. Além disso, há outras que não foram bem sucedidas mesmo com alta adesão, associando-se a opiniões negativas.

Ademais, na figura 2 mostra-se *Frameworks* de testes conforme o uso ao longo dos anos. É plenamente observável que tecnologias antigas decaem a medida que novas surgem. Eventualmente, muitas ferramentas ou versões dessas serão depreciadas. Cabe, então, ao desenvolvedor, providenciar meios para que a plataforma não seja severamente afetada. É nesse momento que surge umas das razões da modernização de *software*.

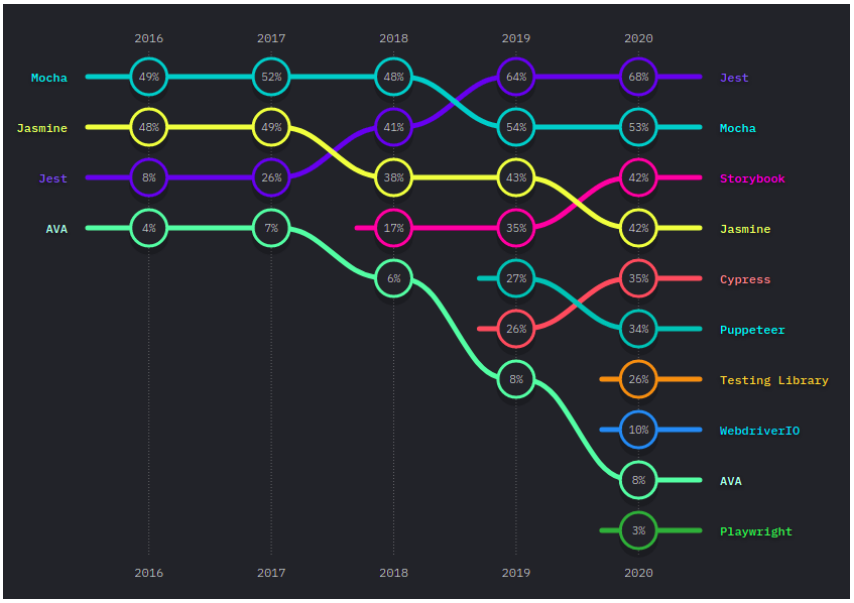

Figura 2 – Correlação das tecnologias JS entre uso e opiniões

Fonte: Greif e Benitte (2020)

Com tantas evoluções e lançamentos, trabalhar com inovação disruptiva se torna raro e a frequente troca ou atualização inadequada pode quebrar as aplicações, atrasar o desenvolvimento, e até causar perdas de clientes e uma crise na empresa. Por isso, torna-se necessário uma análise profunda do sistema, averiguação dos riscos e caso compense, o planejamento para uma futura adequação.A criticidade destes passos aumenta se código não possuir testes que garantem que *bugs*[3] não ocorram em produção. Caso não haja nenhuma revisão, corre o risco do sistema se tornar obsoleto, ou legado, e agravar a performance e satisfação dos funcionários e dos clientes a longo prazo. Em um cenário crítico, pode inviabilizar o uso, provando perdas e prejuízos.

Para que o *software* permaneça útil e continue agregando valor para a empresa durante

[3] falha ou comportamento inesperado

sua vida útil, é necessário a manutenção deste. Estima-se que cerca de dois terços dos custos organizacionais de software se referem aos custos de evolução. Novas melhorias são geradas quando há mudanças nas requisições dos clientes ou nos negócios e variações nas necessidades dos usuários. Correções são feitas através de relatos de defeitos ou por alterações de outros sistemas em um ambiente de software. E a modernização abrange uma grande variedade de razões, desde de operações custosas e não performáticas até por não oferecerem suporte de outra tecnologia (SOMMERVILLE, 2019).

1.1 Estado da arte

Para trabalhar com um sistema legado, é preciso entender melhor sua definição, o que ele representa e como solucionar a situação. Há várias discussões a cerca da definição de um sistema legado, muitos enfatizam a má qualidade, os anos de vida ou a complexidade. Mas o caminho geralmente adotado para solucionar é o mesmo, adequá-lo, refatorando e criando testes. A substituição deste sistema não é convencional, pois é necessário desconcentrar o time de desenvolvimento para a criação de outro, que se não for bem planejado e implementado, se transforará em legado rapidamente.

Chervenski (2019) coletou dados de vários documentos na literatura para analisá-los através da Teoria Fundamentada em Dados (*Grounded Theory-GT*). O trabalho teve como objetivo propor uma definição mais generalista para sistemas legados e definir as soluções mais aderidas pelas empresas. Após analise, o autor pôde concluir que um sistema passa a ser considerado legado quando este possui tecnologia obsoleta, com manutenção custosa, mas fundamental, já que a empresa não pode descartá-lo, porque nele há regras de negócio implementadas.

Independente da linguagem, Perito (2019) afirma que as consequências mais comuns do software legado são baixa velocidade de desenvolvimento e alta complexidade. Além disso, os desenvolvedores se sentem inseguros para alterar algo, já que a modificação podem causar *bugs* que dificultam ou até mesmo impedem alguma utilização do cliente, o que pode gerar prejuízos à empresa.

Algo passível de questionar é, como um sistema passa a ter essas características indesejáveis, se tornando um fardo necessário para as empresas? Não há uma trajetória definida, mas o sistema pode dar indícios como falta de documentação. Há diversas técnicas e boas práticas que ajudam o software ser padronizado e por consequência, ser mais intuitivo e acompanhar-se de melhor performance de desenvolvimento. Um código de má qualidade pode injetar *bugs* e provocar quebras, uma vez que sua criação e manutenção é complexa e custosa.

Catolino (2018) estudou se havia alguma relação entre a qualidade do código e a satisfação dos usuários de aplicativos Android. Para tal, o autor fez uso de *Machine learning* com intuito de usufruir de uma análise mais completa e automática. Para treinamento e classificação, o autor empregou um conjunto de dados contendo avaliações e resenhas extraídas da *Google*

Play Store, métricas do código e *code smells* de 439 *apps*. Catolino (2018) utilizou 13 métricas da qualidade de código e 8 *code smells* e fez treinamento do modelo de *machine learning* com uma *Random Forest*. Após o término, o autor pode concluir que quatro métricas impactam no sucesso dos aplicativos, *Number of Children* (NOCH), *Depth of Inheritance Tree* (DIT), *Percent Public Instance Variables* (PPIV) e *Access to Public Data* (APD).

Uma definição prática que induz automaticamente na solução para o problema foi proposta por Michael C. Feathers, em seu livro *Working Effectively with Legacy Code*. Ele considera o risco de alterar alguma parte do código ao atribuí-lo como legado. Assim, mesmo que os desenvolvedores sigam as bons práticas já bem estabelecidas por diversos autores renomados como Martin Robert, a não implementação de testes o torna complexo, aumentando o tempo de manutenção deste. Dessa forma, ele afirma que um código sem testes já é legado, já que sem eles, não se sabe se o código está aprimorando ou degradando (FEATHERS, 2004).

A estratégia de modernização mais utilizadas pelas empresas é a refatoração e atualização do sistema. Dentre as opções, está a migração para nuvem, substituição de componentes, utilizar blockchain, ERP System e IoT. O motivo da adoção por migração se justifica por não paralisar o sistema de produção e por não ser necessário dedicar parte da operação em outra estrutura. Além disso, criar um novo sistema pode ser mais demorado e caro, pois haverá problemas, alguns já conhecidos pela equipe, mas que a resolução é ainda desconhecida e outros completamente diferentes (CHERVENSKI, 2019).

1.2 Objetivos gerais e específicos

O objetivo do presente trabalho é analisar de forma sistêmica o processo de modernização de código, identificando os fatores que podem influenciar na performance e qualidade.

Como objetivo especifico, tem-se:

- Revisar a literatura acerca de estratégias de migração de sistemas legado;
- Adequar o caso de estudo de um código legado
- Melhorar a capacidade de evolução do projeto
- Introduzir testes

1.3 Justificativa do trabalho

A manutenção de código é uma tarefa recorrente do programador. Muito se fala em criar uma aplicação, mas manter uma plataforma operante demanda refatoração e adaptação com novas tecnologias. A manutenção pode demandar mais tempo e recursos que a implementação de novas funcionalidades, o que, na ausência de uma documentação completa e um sistema de má qualidade, exonera os desenvolvedores em suas tentativas de *debugging* para entender

a lógica e as regras de negócio, o que pode ocasionalmente, gerar mais *bugs*. A modernização e evolução de um *software* possuem diversas abordagens e estudos na academia, sendo que cada motivo possui uma estratégia um pouco mais adequada. Não há uma receita definitiva para solucionar casos onde o programa precisa de ser modernizado. A proposta do presente trabalho é desmistificar a transição de um código considerado legado e aprimorar a manutenção de código. Entender como a falta de testes pode impactar no desenvolvimento e aplicar técnicas que visam a transformação do código legado em um código intuitivo.

1.4 Estrutura do trabalho

O presente trabalho está dividido em 5 capítulos. O capítulo 1 inicia o tema com informações atuais e apresentando o Estado da arte, Objetivos e a Justificativa do trabalho.

O capítulo 2 apresenta a revisão da Literatura, o mesmo aborda conhecimentos que envolvem o desenvolvimento de *software* de forma genérica e conhecimentos para o melhor entendimento para a realização de uma modernização.

No capitulo 3 aborda o desenvolvimento do trabalho, o estudo do caso, o método utilizado para a modernização, descrição das ferramentas utilizadas, bem como a execução.

No capítulo 4 apresenta discussões a cerca dos resultados apresentados, bem como os aprendizados com as dificuldades.

E por fim, conclusões no capítulo 5 e propostas para trabalhos futuros.

2 REVISÃO DA LITERATURA

2.1 Processo de desenvolvimento de um *Software*

O ciclo de vida de desenvolvimento de *software* (SDLC) é um modelo de gestão que estabelece etapas e processos envolvidos na construção, operação e manutenção de um sistema, indo desde da especificação dos requisitos que tornam a concepção válida, até a descontinuação. Dessa forma, é uma estrutura que provê organização para realizar entregas e reparticionar o sistema de forma a tornar o desenvolvimento menos complexo. A escolha do modelo é alinhada entre o cliente e a equipe de desenvolvimento, e para isto, deve-se avaliar as características do modelo, se estas combinam com o tempo, recursos e com as particularidades da aplicação. Podendo ser categorizados como linear, interativo e evolutivo, o SDLC abrange diversos estudos no ambiente profissional que visam o aumento da longevidade do *software*. A descontinuidade deste só se torna uma opção considerada quando o software se tornou legado, inconsistente e dispensável (RUPARELIA, 2010).

Mais especificamente, o processo de constituição de um *software* é elaborado por etapas detalhadas a seguir:

- Definição: etapa onde a empresa/cliente conclui que custo-benefício de um *software* compensa e reunirá objetivos e requisitos para a concepção do mesmo;
- Arquitetura: fase de pesquisa da composição do sistema, preparando tudo que será necessário para implementação;
- Implementação: aplicação da teoria na prática, garantindo que todos os requisitos sejam atendidos;
- Testes: para evitar que defeitos e comportamentos indesejáveis entrem em produção, ou seja, que os clientes recebam o produto com problema, há a realização de testes por uma equipe diferente daquela que implementou para estressar as funcionalidades e assim, encontrar pontos de melhoria;
- *Deployment*: *Release* para produção. Nessa etapa, o código é entregue para os clientes;
- Manutenção: o propósito aqui é garantir a longevidade, estabilidade, correção de *bugs* e a entrega de novas funcionalidades, mantendo o produto potencializado;

É válido destacar que o processo descrito é genérico para os tipos de aplicações existentes, sendo que Ruparelia (2010) considera que há três tipos. Assim, pertencem à classe 1 aqueles que

provêm a API[1] para outras aplicações utilizarem. A categoria 2, os que possuem funcionalidades mais intuitivas para o utilizador. E em subsequência, a categoria 3, abrangendo aquelas com interfaces gráficas para o usuário. Esta última pode utilizar as outras como bibliotecas ou serviços e assim, ter sua dificuldade reduzida.

2.2 Método ágil

Porém, mesmo apesar da reutilização de aplicações, o processo de desenvolvimento ainda pode ser custoso e desorganizado. Dessa forma, há estudos de metodologias que visam a melhor relação entre a qualidade e a agilidade, existindo, assim, algumas abordagens estabelecidas como forma de atender cada necessidade percebida nas empresas. Em suma, existe o processo de desenvolvimento dirigido por plano e o processo ágil, sendo que cada um possui metodologias diferentes que podem atender os diferentes sistemas existentes. No entanto, é válido destacar que não há um modelo universal aplicável para todos os casos, tão pouco um modelo que seja o melhor para a maioria. Diversas peculiaridades do sistema, da equipe e dos requisitos do cliente afetam na escolha do método a ser utilizado (SOMMERVILLE, 2019).

Geralmente, os processos que mais se enquadram em empresas são os ágeis. Estes são dotados de conceitos e métodos que tornam a produção de um *software* útil mais rápida.

Neles, o *software* é dividido em partes funcionais relevantes. Há uma discriminação dos requisitos e exploração das regras de negócio, em seguida, o projeto, implementação e testes. Depois, estes são lançados como versões do sistema para os clientes ou usuários finais. Cada versão do produto é avaliada e melhorias são solicitadas e adicionadas a posteriores lançamentos. Dessa forma, incrementa-se o sistema de forma gradual e intercala-se as etapas do processo.

A documentação, diferentemente do processo orientado por plano, é minimizada e resumida para conter apenas características essenciais, como regras de negócio, serviços e *flags* associadas.

As interfaces são constituídas através de ferramentas de desenvolvimento rápido e interativas. Dessa forma, testes de usabilidade são realizados e a própria ferramenta serve de documentação.

Como forma de evitar que *bugs* permaneçam durante muito tempo em produção, costuma-se associar as novas funcionalidades com *flags*. Assim, se houver necessidade de retirar a nova *feature* de produção, é necessário apenas desligar *flag* para então seguir para a correção do *bug*. Essa técnica traz dinamismo e agilidade à equipe, que pode responder rapidamente as questões, e evitar grandes frustrações do cliente.

[1] *Application Programming Interface*: Conjunto de rotinas e padrões que facilitam a comunicação e troca de informações entre sistemas. Através da API é possível requisitar dados, bem como inserir e atualizar os mesmos.

2.3 Manutenção

O ciclo de vida não se finda após o lançamento do sistema. É intangível pensar que não haverá nenhuma melhoria ou correção para o software permanecer útil. A manutenção fornece suporte econômico e envolve mudanças depois que o sistema está disponível em produção, ou seja, sendo utilizado pelos usuários. De acordo com Sommerville (2019) a evolução e manutenção constitui cerca de dois terços do orçamento de TI. E de fato, eles são operações fundamentais para que os negócios funcionem (ERDIL et al., 2003).

Na figura 3 demonstra-se, de forma genérica, o ciclo de identificação de novas possíveis mudanças. Após o reconhecimento, elas são são propostas para serem avaliadas. A importância para o cliente e para a empresa, além dos benefícios obtidos, são fatores quem influenciam na priorização da mudança. Após a preparação, a evolução será implementada, originando em uma nova versão do sistema.

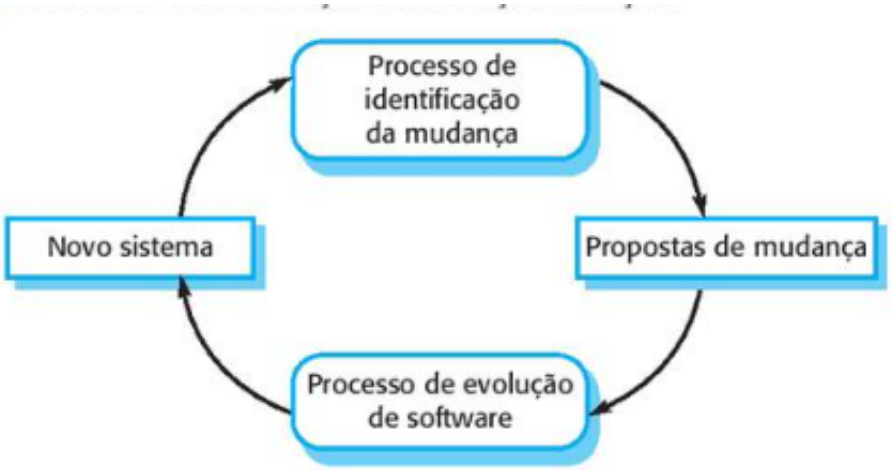

Figura 3 – Processos de identificação de mudanças e de evolução

Fonte: Sommerville (2019)

Malinova (2010) descreveu diferentes estratégias de evolução: A manutenção, modernização e substituição. Este trabalho dará mais ênfase aos dois primeiros. Tal escolha é justificada no subsequente parágrafo.

Apesar de se apresentar em uma forma mais moderna e de rápida inicialização, a substituição é a estratégia mais complexa, sua concretização se mostra exponencialmente custosa. A escolha desta implica na perda de aprendizados anteriores da equipe de desenvolvimento bem como o aparecimento de novos problemas ainda desconhecidos. Tais implicações subsequência uma série de adversidades que transtornam o desenvolvimento e pode causar prejuízos para a empresa. Além disso, o novo sistema pode se tornar legado facilmente, por falta de documentação, depreciação de bibliotecas e por falta de testes. Devido a todos os motivos supracitados, a estratégia de substituição não é viável, tão pouco escolhida pelas empresas. Geralmente, quando se toma essa decisão é porque o legado deixou de entregar agilidade ou performance. Tornando-se

caro demais para manter e com riscos para a operação. Ou, simplesmente não agrega, sendo, então, desnecessária. (RAKSI et al., 2017)

Há três tipos de manutenção, as adaptativas, corretivas e evolutivas. A primeira se refere a alguma modificação para se adequar a uma regra de negócio modificada ou nova. As corretivas estão relacionadas a correção de defeitos, *bugs*, seja na interface, usabilidade, incongruência para com as regras ou então, com alguma falha de segurança. Esses casos são normalmente emergenciais. Várias equipes podem parar o desenvolvimento de outras tarefas para resolver o problema.

Já a manutenção evolutiva se refere aquelas onde novas funcionalidades são inseridas. Porém, diferente da modernização, a manutenção é um processo geralmente mais simples com objetivos reduzidos. (RAJLICH, 2014)

Porém, é válido destacar que embora a manutenção esteja relacionada com pequenas alterações, deve-se fazê-la com cautela pois um código pode-se tornar legado por negligência. Ao realizar alguma manutenção indevida, a documentação bem como os testes por código podem ficar desatualizados. Além disso, pode-se introduzir *bugs* e dificultar o entendimento do código.

De acordo com OpusSoftware (2018), algumas técnicas podem auxiliar na operação de manutenção:

- Documentação: altamente recomendada principalmente no que tange ao repasse do conhecimento sobre o código e sobre a funcionalidade. Sem a documentação atualizada, todos os participantes do projeto que querem entender sobre algo, precisam procurar a pessoa que desenvolveu aquela funcionalidade ou alguém que participou do projeto na época. A falta torna custosos os casos de atendimento ao cliente, de manutenção e até mesmo em projetos de novas funcionalidades. Com a documentação, o processo de transferência de conhecimento sobre o sistema se torna mais eficaz, não dependendo de pessoas específicas que já passaram pelo projeto.

- Versionamento: cada lançamento, ou *deploy*, há uma nova versão do sistema. As versões se tornam um histórico, uma documentação automatizada e auxiliam quando o *rollback*(voltar a versão anterior) se faz necessário. Durante a manutenção ou evolução, utiliza-se, por segurança, ao menos duas versões do sistema. Sendo uma, a que está em produção, disponível para os clientes e a outra, alvo das modificações. Após a implementação, testes e analise do código, combina-se as duas versões (conhecido como *merge*) e faz o processo de *build* e *deploy* em produção, ou seja, torna as modificações disponíveis para o cliente.

- *Status Reporting*: é uma forma de alinhamento entre os envolvidos no projeto em relação à situação atual do desenvolvimento. São destacados pontos de dependência, dúvidas e de atenção. A documentação desse alinhamento é importante para que todos os pontos sejam lembrados e possuírem uma solução ou explicação.

- Codificação: há padrões e técnicas que auxiliam muito na organização e no entendimento do código. Boas práticas agregam a qualidade e torna o código legível e intuitivo. Além disso, contribui também para a performance da aplicação. O uso de pacotes, orientação a objetos e padrões de projeto também contribuem na organização e divisão das responsabilidades para reaproveitamento de código, aumentando a produtividade das equipes de desenvolvimento.

Realizando-o adequadamente, a manutenção evitará o envelhecimento precoce do sistema, o manterá por mais tempo e evitará prejuízos. E consequentemente, continuará agregando valor e benefícios para a empresa.

Há fatores que dificultam a manutenção, tornando-a demorada e custosa. Esta dificuldade se torna um impeditivo para o crescimento do sistema, ocasionando perda de produtividade e de competitividade, além de contribuir para a falta de bem estar dos desenvolvedores. Alguns desses fatores são: código não estruturado, de complexa lógica, programadores com pouco conhecimento do sistema ou de codificação, documentação insuficiente, suporte de hardware e software obsoleto,e além disso, dependência pela equipe que mantém a aplicação com àquela dos desenvolvedores que participaram e desenvolveram o software.

Assim, a modernização se torna imprescindível para evitar o definhamento da aplicação, das equipes, e consequentemente, dos negócios. O sucesso desse processo depende da prevenção e correção desses problemas desde o início do ciclo de vida.

2.4 Modernização

De acordo com Malinova (2010) e Majthoub, Qutqut e Odeh (2018), a modernização envolve modificações maiores, evitando grandes mudanças na arquitetura e nas regras de negócios, O cerne dessa pŕatica é aprimorar ou alterar o software existente para que ele possa ser compreendido, gerenciado e reutilizado como um novo software. No âmbito mais geral, há objetivos que diversificam dependendo do sistema e da necessidade, como quebra de sistemas monolíticos em serviços, migração de serviços em nuvem, melhoria da manutenção e portabilidade, aumento da eficiência ou adoção de novas tecnologias.

De acordo com Malinova (2010), Há duas técnicas para a realização, a *black-box* e a *white-box*. A diferença entre elas se resume na compreensão necessária do sistema. Porém, não são mutuamente exclusivas, os desenvolvedores podem mesclá-las, muitas vezes a técnica de *wrapping* é introduzida como uma das técnicas para realizar a reengenharia. Outrossim, maioria das vezes o *wrapping* também requer algum nível de engenharia reversa com o objetivo de melhorar a compreensão da relação entre objetos, de interfaces ou de hierarquia de classes.

A *black-box* considera o sistema uma caixa preta, onde há entradas e saídas, conseguindo, assim, abstrair seu comportamento e essência. O *wrapper*, ou seja, o empacotamento pode ser

realizado em diferentes níveis, dessa forma, é possível atingir a forma mais simples para produzir o efeito desejado.

Já a modernização *white-box* é mais complexa do que a abordagem anterior. Conhecida também como reengenharia, ela envolve compreensão dos componentes internos e análise do sistema a fim de reconstruí-lo sob uma nova forma.

A importância da reengenharia de software está em sua capacidade de recuperar e reutilizar componentes que já existem em um sistema desatualizado. Notoriamente, isso reduzirá o custo de manutenção do sistema e estabelecerá a base para o futuro desenvolvimento de software (MAJTHOUB; QUTQUT; ODEH, 2018).

Na figura 4 descreve-se um modelo geral de reengenharia, que não precisa necessariamente passar por todas as etapas demonstradas em questão. A primeira etapa, consiste na tradução do código para que se torne mais legível, sendo esta prática utilizada somente quando o ambiente de desenvolvimento da linguagem de programação não é mais usada.

Figura 4 – Modelo de reengenharia

Fonte: Sommerville (2019)

Depois, o código passa pela fase de engenharia reversa que consiste na análise do programa e extração das informações. A engenharia reversa envolve atividades como re-documentação e recuperação de projeto, é normalmente aplicada para entender os requisitos e ter base para projetar essas regras de negócios para um novo ambiente com base em uma arquitetura moderna. A engenharia reversa pode ser automatizada, o que facilita o processo (MALINOVA, 2010).

Em paralelo, a estrutura do código é aprimorada, podendo ser feita por refatoração. Refatorar é reestruturar dentro de um contexto orientado a objetos. É definido como mudar um sistema de software de tal forma que não altere o comportamento externo do código, mas melhore sua estrutura interna, como aplicar melhores prática de qualidade do software (MALINOVA, 2010).

Após, há a modularização de programa, onde partes relacionadas do programa são agrupadas, e onde redundâncias são removidas caso seja apropriado. Em alguns casos, esse

estágio pode envolver refatoração de arquitetura.

Caso haja modificação na estrutura de dados, a reengenharia de dados pode ser necessária. Esta consiste em redefinir os esquemas de banco de dados em uma nova estrutura. Os dados devem ser limpados, o que envolve encontrar e corrigir erros e remover registros duplicados (SOMMERVILLE, 2019).

As transformações de código reais durante a reengenharia são realizadas por meio de uma série de técnicas que envolvem reestruturação. A reestruturação é a transformação de uma forma de representação em outra no mesmo nível de abstração relativa, preservando o comportamento externo do sistema.

A abordagem de reengenharia utilizada no presente estudo é o incremental, onde o processo é dividido e adicionados incrementalmente como novas versões do sistema para satisfazer novos objetivos (ROSENBERG; HYATT, 1996).

Group et al. (2015) publicou um relatório baseado em seu *database*(banco de dados) o qual contém cerca de 50.000 perfis detalhados em 2011 até 2015, onde relaciona diversas métricas de forma a trazer *insights* sobre o efeito de cada no sucesso de um projeto. Como métrica de estudo, Group et al. (2015) define que um projeto obteve sucesso quando este foi entregue dentro de um tempo estimado razoável, permaneceu dentro do orçamento e proporcionou a satisfação do cliente e do usuário, independentemente do escopo original.

Em uma de suas análises, demonstrado na figura 5, pode-se constatar que grandes projetos não conseguem retornar valor, já que 53% presentes no estudo possuíram valor de retorno baixo, ou seja, custo-benefício ruim. Por outro lado, projetos com escopo reduzido possuem alta probabilidade de retorno. Como conclusão, Group et al. (2015) afirma que quanto mais rápido for entregue, mais rápido será o retorno do investimento.

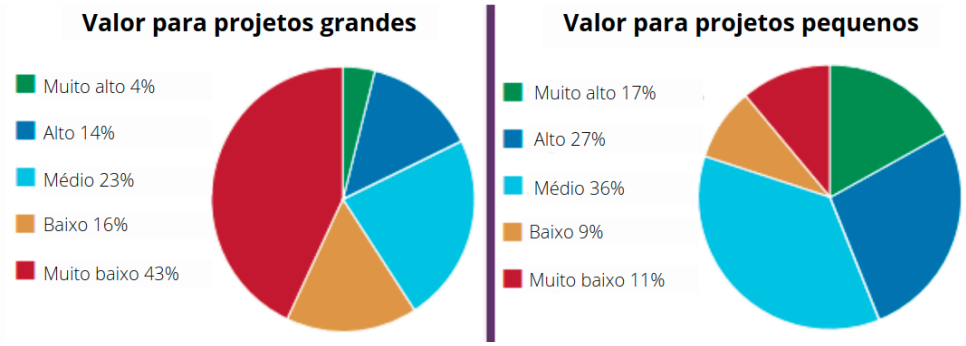

Figura 5 – Valor de retorno de projetos grandes e pequenos

Fonte: Group et al. (2015), traduzido pelo autor

O sucesso também é decrescido para grandes projetos, como mostra a tabela 1, apenas 6% possuem sucesso, em contraste com os de tamanho pequeno e moderado, 61% e 24%, respectivamente. Pode-se concluir, com ambas figuras que quanto maior o projeto, menor a taxa de sucesso e de valor. Induzindo, assim, a modernização com a quebra de sistemas monolíticos em microsserviços.

	Bem sucedido	Desafiado	Mal sucedido	Total
Grande	6%	51%	43%	100%
Largo	11%	59%	30%	100%
Médio	12%	62%	26%	100%
Moderado	24%	64%	12%	100%
Pequeno	61%	32%	7%	100%

Tabela 1 – Relação entre a taxa de sucesso e o tamanho do projeto

2.5 Testes

O processo de desenvolvimento e também de evolução de um software é cercado por testes que visam garantir a qualidade e confiabilidade da entrega, ou seja, certificar que o produto não possui erros e problemas. Muitos desses podem prejudicar a experiência do cliente, outros apresentar como falha de segurança. Portanto, a realização de testes tem por objetivo verificar se o comportamento é esperado. Apesar desse cerco, muitos erros ainda podem aparecer no ambiente de produção, chamando-o de *bug*. Por isso, o processo de testes deve ser bem planejado e conciso.

A minimização do custo de correção de defeitos está diretamente relacionada na qualidade do processo de testes e na concisão do mesmo. Estima-se que, através do cálculo de retorno de investimento, que um sistema pouco testado pode custar mais que o custo da realização de testes. Segundo Patton (2006), o custo para corrigir um defeito aumenta exponencialmente conforme a demora para encontrá-lo.

O *software* passa por ao menos três baterias de testes. Aconselha-se iniciar o primeiro desde do começo do desenvolvimento do código, assim a lógica é validada durante a construção (SOMMERVILLE, 2019).

Depois, é interessante que haja uma segunda fase antes do código ir para produção. Nesta, a versão pronta é sujeita a testes por uma equipe diferente daquela que o desenvolveu e tudo é verificado, desde a usabilidade, design até os requisitos do sistema. Isto favorece, inclusive, pontos de melhoria, e situações que ainda não foram contempladas. Esses testes são abordados neste trabalho como testes de QA(*Quality Assurance*).

O termo *Quality Assurance*, ou garantia de qualidade, trata de um processo importante para o ciclo de desenvolvimento de *software*, é uma etapa que visa garantir que o produto seja

entregue respeitando os requisitos e qualidades requeridas pelo cliente. É nele que o time vai testar e verificar tudo que esteja relacionado a funcionalidade.

Depois de tudo sinalizado, as funcionalidades são liberadas aos poucos para os clientes. Nesta fase, cabe estudos para ver como os clientes se comportam, realizando traqueamento de adesão e percepção de dificuldade de uso.

A introdução de testes no código não é crucial, mas é bastante importante para identificar defeitos enquanto se programa, antes do uso da funcionalidade. Pode-se perceber erros de lógica e até mesmo casos não percebidos das regras de negócio.

Nos dois primeiros estágios, os testes podem ser manuais e automatizados. No teste manual, um responsável segue o roteiro de teste, afim de encontrar erros. Após encontrados, reporta-os aos desenvolvedores. Assim, a correção é feita e os testes são realizados novamente.

Os automatizados, por outro lado, tem o intuito de agilizar e minimizar falhas humanas, dessa forma, testes são codificados com dados fictícios, e simulações de interação com o usuário. Assim, cada vez que o sistema em desenvolvimento é testado, o próprio sistema verifica se há alguma anomalia, comportamento indesejável e notifica, apontando qual teste falhou.

Porém, o processo de teste deve ser analisado e possuir critérios tanto quanto os demais processos. Caso contrário, será custoso e sem eficácia, pois muitos casos importantes não serão contemplados. Um processo de teste planejado se torna relevante e eficaz para provar o comportamento.

Um conjunto de dados de entrada é previamente selecionado através do critério de Particionamento de Equivalência. Para tal, particiona o domínio, transformando-os em classes) de dados passíveis de gerar casos de teste. Quando duas ou mais classes produzem o mesmo resultado, isto indicando que ambas são equivalentes.

Uma classe de equivalência representa um conjunto de estados válidos e inválidos para condições de entrada de um programa. Tal critério visa reduzir o domínio de entrada para um conjunto finito e ao mesmo tempo eficiente e pode ser complementado com a Análise do Valor Limite, este possui o intuito de verificar os limites das classes de equivalência, selecionando os elementos nas “fronteiras” da classe (GENTIL, 2020).

Na figura 6 demonstra-se um modelo de teste funcional, que pode ser pensando como uma caixa-preta, onde não é necessário entender como o sistema funciona, precisa apenas possuir um conjunto de dados para os casos e entender como deve ser os respectivos resultados. Dessa forma, o sistema reagirá e resultará em um conjunto de dados de saída. Se o resultado não for como esperado, o teste falha.

Outra técnica de teste é o estrutural, onde é proposto exercitar os caminhos lógicos do programa. Nessa técnica o programa é considerado uma caixa branca, uma vez que demanda a execução de partes do programa e os aspectos de implementação são fundamentais para gerar e

selecionar os casos de teste.

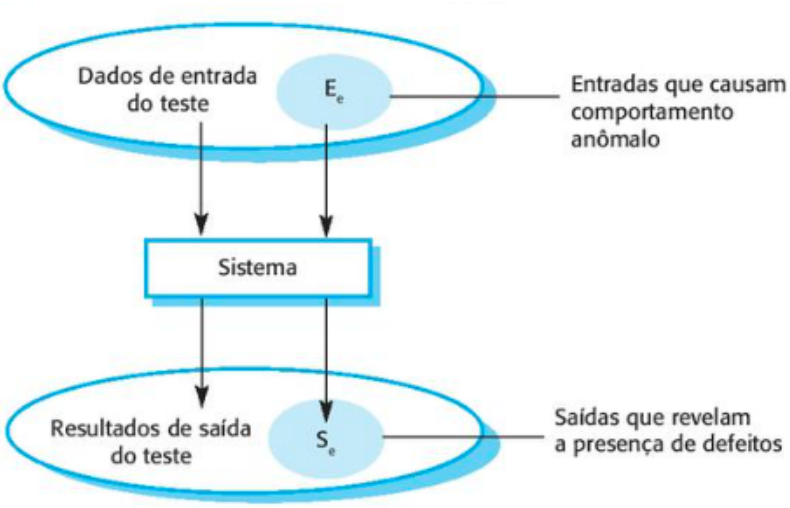

Figura 6 – Entrada e saída de teste

Fonte: Sommerville (2019)

Há também a técnica de introdução intencional de defeitos, onde o objetivo inserir testes para averiguar se a resposta será de fato um erro, derivando assim, os requisitos de teste que podem ser utilizados para revelar defeitos. Da mesma forma que um defeito pode ser introduzido no software de forma não intencional, os casos de testes podem ser ineficientes em não revelar um defeito.

Embora possam existir várias estratégias de teste de software, a abordagem incremental sido bem adotada pelas equipes. Nela, os testes são implementados em níveis, verificando desde as partes unitárias até o software como um todo (GENTIL, 2020).

Os níveis de teste são definidos em termos de unidade, de integração e de sistema. Normalmente considerado um processo auxiliar na etapa de codificação, o teste em nível de unidade tem o foco na verificação da menor unidade de um programa. (GENTIL, 2020).

Uma unidade é entendida por uma função, um método ou uma classe. O objetivo do teste unitário é revelar defeitos relacionados a algoritmos incorretos, estruturas de dados mal implementadas ou simples erros de programação. Como cada unidade é testada separadamente, a construção do teste pode ocorrer de forma incremental antes mesmo da conclusão do software. Uma vez que cada unidade funciona de forma adequada individualmente, é necessário garantir seu comportamento ao executar de forma simultânea, pois, quando combinadas, pode gerar equívocos (GENTIL, 2020).

À medida que as diversas partes do software são executadas de forma conjunta, é preciso verificar se a integração entre as partes funciona de acordo com o especificado e sem falhas. O teste em nível de integração é a forma sistemática para verificar essa combinação de unidades. O teste em nível de sistema pode ser aplicado quando o software está completo. Esse nível de teste

objetiva validar se os requisitos foram implementados corretamente e explora tanto aspectos não funcionais, tais como segurança e desempenho, quanto funcionais (GENTIL, 2020).

Na figura 7 ilustra-se o fluxo ideal para o processo de teste. inicialmente, os casos de testes são estudados e identificados. Caso haja necessidade de dados, os mesmos são simulados para executar o programa corretamente. Depois, os resultados do testes são avaliados e comparados com os resultados esperados. Um exemplo para esse fluxo é um teste de um botão de uma calculadora, espera-se que ao clicar em *Enter* o resultado seja apresentado corretamente na tela. Então, pode-se simular o clique do botão após a entrada de dois números e da operação e comparar o resultado obtido com o esperado, bem como garantir se de fato o número apareceu na tela.

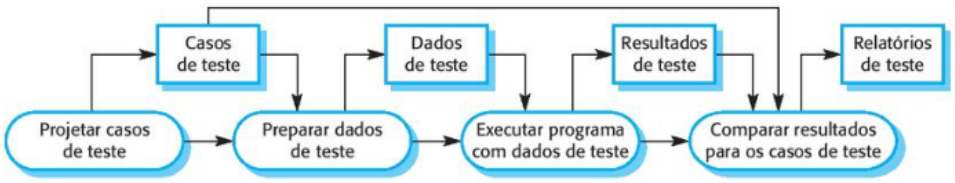

Figura 7 – Modelo do processo de teste de *software*

Fonte: Sommerville (2019)

Uma estratégia utilizada no presente trabalho é o TDD (Test Driven Development), em português, Desenvolvimento Orietado por Testes. Esta técnica se baseia em pequenos ciclos de desenvolvimento e de testes. Assim, a implementação se inicia com um primeiro teste. O mesmo deve falhar pois não há nenhuma funcionalidade implementada. Após essa falha, o código é feito e em seguida, o teste é executado novamente. Assim que o teste seja bem sucedido, ou seja, o código funciona, a refatoração do código é feita com intuito de torná-lo mais limpo e intuitivo. Após essas etapas, mais testes são implementados para assegurar o correto funcionamento. (ANICHE, 2014)

3 DESENVOLVIMENTO

Para a realização deste trabalho, foi feita um planejamento metódico e sistêmico, em seguida, a execução deste foi feita através de conceitos aplicáveis de metodologia ágil. A análise do estudo baseia-se na comparação das métricas de manutenibilidade e performance.

3.1 Avaliação do custo benefício da modernização

Antes da decisão a cerca da modernização, é preciso avaliar se a mesma compensa. Em algumas situações, o *software*, apesar de estar em uma linguagem de programação antiga, está bastante estável e o processo pode custar muito para a organização.

A avaliação de custo benefício leva em consideração todos os pontos que serão impactados durante a modernização, visto que as alterações podem comprometer todo um ecossistema que gira em torno desse serviço e é necessário tratar minuciosamente quanto aos detalhes que a envolvem. Levando em conta os benefícios que virão após a implementação e o tempo de trabalho gasto, é importante entender alguns pontos principais para que isso ocorra da melhor maneira:

- Tempo de demanda: Muitas vezes modernizações levam mais tempo que o desenvolvimento em si, isso impacta diretamente, dado que parte da força de trabalho da empresa fica destinado a atender essas atualizações, muitas vezes essas pessoas poderiam estar atacando outros problemas centralizados.
- Riscos de modernização: A atualização gera riscos na integração dos componentes novos com antigos, podendo gerar falta de comunicação o que acarreta mais demanda de trabalho para correção. Há riscos de má adaptação dos utilizadores caso não seja documentado corretamente, o que pode quebrar os componentes também, por isso enfatiza-se o fato de todos envolvidos estarem bem comunicados e informados.
- Problemas não mapeados: No decorrer da implementação, é comum ocorrer problemas de produção, isso acarreta novas demandas para correção que devem ser imediatas. Nesses casos, é preciso realizar outra análise, se é vantajoso o diagnóstico do caso ou voltar a versão anterior a migração. Por isso, a importância dos testes e liberação para usuário final a partir de *flags*, afim de não comprometer o projeto.
- Benefícios da modernização: Após implementação, a documentação e padronização dos componentes facilitará na utilização dos mesmos por novos e antigos membros da equipe de desenvolvimento. Para o usuário final, será evidente a melhoria na velocidade e qualidade de dados fornecidos na plataforma, tratando-os com mais assertividade. A atualização leva

em consideração o UX[1], trazendo mais conforto na usabilidade e facilitando a compreensão do usuário para as informações da plataforma.

A modernização traz benefícios óbvios, entretanto muitas vezes não é plausível, se tratando de sistemas que utilizam inúmeras ferramentas conectadas que podem ou não funcionar com os novos meios, ou demandariam muito tempo para alteração de algo que já atende as necessidades. Simplificando o ponto, a modernização deve sempre agregar algo mais do que a solução atual já atende, devido ao seu custo em mão de obra e riscos, muitas vezes a priorização deve ser mantida em atender as necessidades.

Tendo tudo isso em pauta, se for possível estimar os valores de gasto e de ganho, pode-se realizar o cálculo de retorno sobre investimento (ROI). O ROI é utilizado pelas empresas como uma forma simples de mensurar o retorno sob determinado investimento, classificar a atratividade de vários investimentos e realizar comparações entre eles. O cálculo é dado por:

$$ROI = \frac{Receita - Custo}{Custo} \tag{3.1}$$

Um ponto de atenção salientando por Galhardo (2018), é relativo ao tempo do investimento, é preciso considerá-lo e ponderar sob o cálculo. Nessa analise o tempo foi quantificado e considerado como custo. Dessa forma, o valor se aproxima da realidade.

3.2 Motivos da modernização

O primeiro propósito que leva a escolha da modernização do código é a utilização da versão 2.7 do Python que está em processo depreciação. A empresa *Python Software Foundation* lançou novas versões não compatíveis com as versões anteriores ao Python 2.7. Além disso, a empresa encerrou o suporte oferecido ao Python 2 em janeiro de 2020. Assim, muitas empresas e usuários estão realizando a migração do código. (PYTHON, 2020) E outras empresas estão removendo o uso dessa versão da linguagem, como por exemplo, Apple em Apple (2020).

A parte alvo engloba recursos e códigos importantes implementados na liguagem Python, na versão 2.7. Por isso, foi necessário encontrar novas formas de atingir o objetivo que antes era alcançado utilizando essa linguagem. Na funcionalidade abordada no presente trabalho, optou-se por remover a requisição feita em Python e chamá-la diretamente na linguagem JavaScript. .

Outro motivo é a utilização do *Design System* que, para isso, é preciso remover componentes feitos manualmente e importá-los desse projeto.

Visa-se também uma forma mais performática e intuitiva de implementação da funcionalidade. Assim, visa-se a reconstrução do código utilizando o webpack, que melhora a execução do código JS no navegador.

[1] *User Experience*, área que estuda a experiência do usuário e preocupa em entregar ao consumidor experiências com base em um design responsivo, agradável, organizado e intuitivo.

A utilização dessas ferramenta é mais explicada em seção 3.4.

Dentre as vantagens desta migração estão a organização dos componentes, velocidade para desenvolvimento, criação de testes unitários e também a oportunidade de resolver débitos técnicos e aplicar comunicação direta com o *back-end* sem a intermediação da GAE[2].

3.3 Metodologia

Essa seção contém o detalhamento da metodologia aplicada neste trabalho em prol de seu respectivo objetivo.

- Planejamento: Fase em que a parte alvo da sistema e o custo-benefício da modernização é avaliada. As características do mesmo, bem como as características da equipe devem ser consideradas para planejar o processo, definir como o processo será feito e estimar o tempo necessário para a entrega. Além disso, as boas práticas documentadas pela indústria e pela academia também foram consideradas para garantir a qualidade do novo código.
- Definição de um processo de migração: Esta fase está diretamente relacionada com a fase de planejamento, porém o estudo foi separado para restringir o escopo das fases.
- Estudo das tecnologias a serem utilizadas Nessa fase, houve um estudo das novas tecnologias em prol de aprender como as ferramentas escolhidas funcionam e conseguir aplicá-las corretamente.
- Engenharia reversa: Entendimento do código existente para aproveitar as funcionalidades e conseguir refatorar.
- Execução da migração: Adaptação do código existente na nova arquitetura. Além disso, implemento de testes a fim de garantir o correto funcionamento do código e evitar a inserção de *bugs*.
- Desenvolvimento das novas funcionalidades: Implementar funções relacionadas a novas boas práticas e adaptação da interface.
- Avaliação do processo e discussão de possíveis melhorias: O processo foi analisado para entender possíveis falhas e discutir como o processo poderia melhorar e ser agilizado. Além disso, uma discussão foi feita para demostrar como a nova aplicação facilitou o desenvolvimento e a manutenção e como as novas funcionalidades contribuíram para o clientes

[2] Google App Engine: Serviço de publicação de aplicativos em nuvem

3.4 A parte alvo

O projeto alvo se constitui de um grande sistema monolítico com mais de 4 anos de vida. Este sistema é de grande importância, tendo em vista que parte considerável das diretrizes do seu negócio e o planejamento estratégico da organização o envolve e muitas funcionam por ela. O sistema em si não foi desenvolvido completamente para depois ser entregue aos clientes. Ele foi incrementado a medida que as necessidades surgiam e oportunidades apareceriam. Características da metodologia ágil foram e são utilizadas nas entregas. Assim, gradualmente, cada funcionalidade é planejada e desenvolvida na história associada a ela.

Isso certamente influenciou diretamente na obtenção de sucesso e conseguiu agregar um alto valor de retorno, conforme visto no relatório Group et al. (2015), onde projetos grandes possuem baixa probabilidade de sucesso. Devido a isso e também à dificuldade de versionamento por ter muitas *releases*, adotou-se a decisão pela quebra do sistema em outras aplicações. Porém, a completa quebra é algo custoso e que demanda muito planejamento, por isso, é uma migração paralela também incremental. Atualmente, duas partes já foram refatoradas e separadas em novas aplicações.

O sistema estudado pertence a categoria 3, previamente mencionada em revisão de literatura pelo Rajlich (2014). Sendo assim, composta pela parte responsável pela construção das interfaces e pela parte que comunica com o servidor e com o banco de dados.

A hospedagem é feita pela Google App Engine (GAE) e a comunicação do projeto com o servidor utiliza a GAE como intermediário. Devido ao custo, visa-se retirá-lo, dessa forma, o projeto comunicará diretamente com os serviços e APIs. Tal retirada diminuirá os gastos de manutenibilidade da plataforma. Outro benefício é a consequente separação total das responsabilidades tidas como de *Back* e como de *Front*, e assim, deixar o projeto apenas com a construção visual da plataforma e seu correto funcionamento.

Para manter a consistência na programação, há uma forte cultura de padronização praticada. A identação é normalizada para haver códigos desestruturados, o que dificulta o entendimento no desenvolvimento, e aponta para modificações desnecessárias, complicando a análise das mudanças por outros profissionais. Boas práticas são disseminadas entre os desenvolvedores para assim, manter o código legível, performático e intuitivo. Além disso, há uma forte valorização de testes, seja manuais e automatizados.

Porém, como houve diversas mudanças tecnológicas durante o tempo de vida do *software*, é praticamente irracional pensar que não houve alguns percussos no caminho. A forma de implementar o código foi aprimorada conforme o aumento do *know-how* dos desenvolvedores. Além disso, as tecnologias utilizadas se atualizaram ou se depreciaram, implicando atualizações do sistema. Devido a esses pontos mencionados, o projeto possui muitas partes desatualizadas e/ou não possuem testes automatizados, apesar da padronização praticada. E assim, o projeto se tornou difícil de entender para os novos profissionais admitidos e consequentemente, sua

manutenção.

Além das ferramentas básicas utilizadas para construir interfaces, que são HTML, CSS e JavaScript, há também a linguagem de programação Python, que serve como uma camada intermediária (*Middleware*), capaz de fazer a mediação entre várias tecnologias de software, de modo que as informações, de diferentes fontes, são gerenciadas independendo das diferenças de protocolos, plataformas e arquiteturas. Assim, a camada de Python permite a comunicação e o gerenciamento de dados entre o banco de dados/frontend e frontend/backend.

Como o projeto de migração é incremental, a nova tarefa planejada foi designada para a migração. Esta tarefa consiste em aprimorar . A parte alvo da tarefa representa uma funcionalidade importante para o sistema, tendo em vista que interfere diretamente nos contratos dos clientes. Ela demonstra o consumo relativo ao limite contratado, seja mensal ou total, e o detalhamento desse consumo. Essa funcionalidade é montada em uma das páginas principais do sistema, e fica localizado na *dashboard* da plataforma. Porém, a exibição das informações é confusa e não é intuitiva, o que suscita dúvidas e alavanca mentorias para maior esclarecimento.

O consumo do limite é feita através de cálculos elaborados no *Back-end*, feito na linguagem de programação Java. O presente trabalho apenas consome esses dados, e não aborda mais detalhes.

Sendo um cálculo complexo, o resultado deste no sistema também causa confusão e contribuí para o aumento das dúvidas relativa ao consumo do limite. Devido a isso, os valores aparentam ser aleatórios, o que induz à pensarem que falta transparência nessa parte da plataforma. Isto põe em dúvida a credibilidade do cálculo e levanta muitas dúvidas sobre como o mesmo é feito e se está realmente certo.

Assim, a tarefa possui como objetivo:

- Aprimorar a visualização da interface, utilizando conceitos de usabilidade e acessibilidade.
- Reduzir e melhorar a quantidade de informações e de configurações.
- Tornar a funcionalidade mais intuitiva para os clientes possuírem rápida curva de aprendizado. Utilizar o novo serviço de cálculo do *Back-end*.

Alertar quando o consumo estiver próximo do limite. Orientar os clientes com consumo elevado à aumentá-lo. Refatorar o código, adicionar testes, introduzi-lo no *Webpack* e construir uma comunicação direta com os serviços do *Back-end*.

Além disso, visa-se a internacionalização dessa página, algo já utilizado em outras partes da plataforma. Em palavras mais simples, permitir a exibição da página em outros idiomas. Atualmente a plataforma tem suporte para o inglês e espanhol, o presente trabalho focou em ambas.

A interface alvo está construída em um arquivo HTML e outro em JS, os dados são provenientes de requisições utilizando Python para intermediar com GAE e com outros arquivos e há definições de variáveis em alguns *scripts*.

Uma das dificuldades desta migração está na requisição de dados, pois o novo serviço criado para realizar o cálculo ainda não tem todos os dados necessários disponíveis para uso. Portanto será preciso dividir o desenvolvimento da nova interface em dois épicos[3], a primeira usará os dados do serviço antigo para os dados que são inexistentes no novo serviço de cálculo.

Outra dificuldade desta migração está na criação dessa *View* sem ou com a mínima influência do Python, já que algumas variáveis são manipuladas em Python, como a identificação de sessão para o controle das mensagens. Além disso, os dados não são controlados e nem centralizados por uma *Store View*, dessa forma será necessário criar uma *Store*.

Para melhor planejamento, foram levantadas as necessidades e complexidades para facilitar a estimativa de entrega. A inserção de testes unitários também foi considerada.

3.5 Estratégia de migração

Para que esta mudança aconteça de modo gradativo foi criado uma *Feature Flag* que controle o aparacimento da nova funcionalidade para os clientes. Assim, será analisado se pode habilitar para determinado cliente, se sim, a nova funcionalidade aparecerá. Essa forma é a mais adequada para a grande maioria das novas *features*. Entre as razões estão: testes A/B onde será monitorado qual versão se sobressairá melhor, menos chamados em casos de erros ou *bugs*, maior controle sobre quem está usando.

Utilizando-se da engenharia reversa para entender o fluxo e o código que constrói a interface, um documento foi escrito contendo os dados necessários, como os arquivos, serviços, o que pode ser componentizado e onde o *Design System* poderia ser aplicado.

Após entender um pouco melhor o código, segue-se para o planejamento de como será o desenvolvimento. Separou-se o processo em quatro partes para serem realizadas em paralelo. A *Store Vuex* com dados inicialmente fictícios, comunicação direta com o serviço de cálculo do *Back*, implementação das novas funcionalidades e implementação dos componentes reusáveis no *Design System*.

Na figura 8 ilustra-se em forma de diagrama a estratégia utilizada.

Após a percepção da necessidade, há um processo de análise e *brainstorms* para avaliar a prioridade e comparar com outras demandas pelos agentes de mercado e produto. Depois de priorizado, há um estudo e exploração de como essa necessidade será satisfeita, como envolve interfaces, estudos de *Design* serão feitos para compor o épico.

[3] Corpo de trabalho composto por tarefas menores com o intuito de atender as necessidades e objetivos definidos em seu escopo

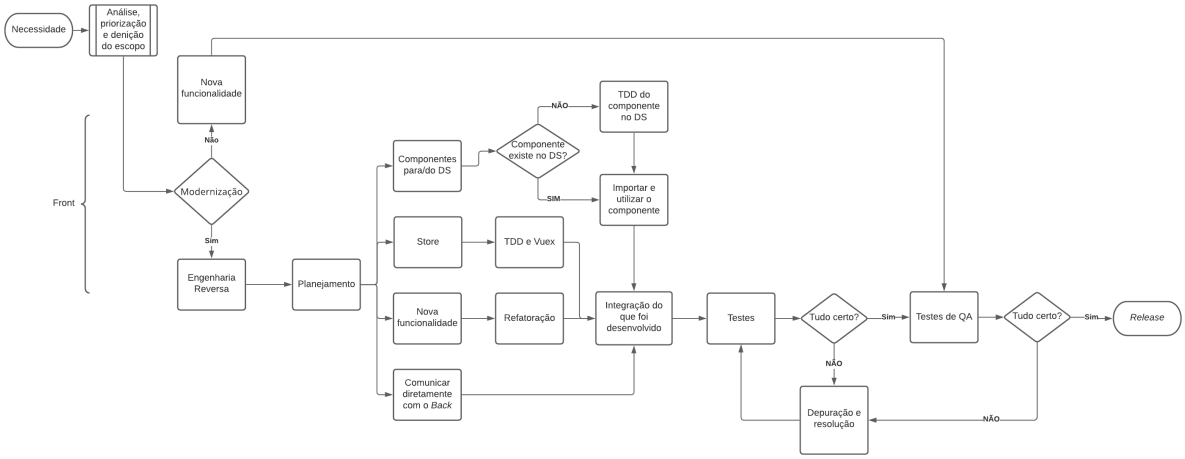

Figura 8 – *Diagrama*

Fonte: Autor

Quando um componente é desenvolvido para uso, a integração não é feita de imediato. Primeiramente, é necessário aprovação de partes distintas da equipe e de outras, afim de analisa-lo como um todo, se cumpre com os requisitos de boas práticas, como se pode ser reutilizado e se segue os padrões dos demais. Só então, após testado e aprovado ele sobe para aplicação principal.

Nesse segundo momento, os *containers* são substituídos por componentes desenvolvidos em Vue. Um ponto a ressaltar é que o foco da criação desta *View* é diminuir ou excluir a influência do Python e definir centralização dos dados em uma *Store*.

Com isso, após a o término da migração teremos uma *View* com os componentes e dependências necessárias para o carregamento da pesquisa e será mais fácil utilizá-lo em uma futura rota refatorada.

3.6 Nova aplicação

As tecnologias que são utilizadas, são pensadas de acordo com a necessidade e flexibilidade delas perante o projeto. Não se quer corrigir um problema para gerar outro, então tudo é construído com o intuito de facilitar o desenvolvimento como um todo.

As vantagens estão explanadas em cada subseção a seguir e relacionadas a cada ferramenta descrita.

3.6.1 Webpack

Algumas linguagens e *frameworks* podem não ser bem compreendidas pelo navegador, podendo acarretar em lentidão e afins. O Webpack é uma forma mais prática de executar o projeto no navegador, ele atua concatenando ou combinando com segurança os arquivos sem

se preocupar com a colisão do escopo.Sem o uso de uma ferramenta responsável por essa execução, seria necessário fazer um arquivo com todas as funcionalidades ou incluir um *script* para cada funcionalidade. Ambas formas são demasiadamente custosas para os desenvolvedores. Por isso utilizar um empacotador de arquivos que facilita no desenvolvimento. Utilizar essa ferramenta possibilita um processamento mais assertivo na virada de chave para produção, ele fica responsável pelo gerenciamento de pacotes e dependências, conseguindo definir padrões para o projeto em estado de desenvolvimento ou em produção por exemplo: ele interpreta arquivos sass, png, cjs entre outros e gerando arquivos estáticos já no formato ideal para virada (REIS, 2018).

3.6.2 Vue

A escolha do Vue implica em sua curva de aprendizado que é mais rápida do que outros *frameworks*, se tratando de um *Javascript open source*, onde se encontra a flexibilidade de trabalhar com componentes em seu projeto, dando chance de reutilizar mais facilmente em outros pontos, assim diminuindo o tempo de desenvolvimento. Esse *framework* possui o diferencial de ser projetado para a programação incrementada. A integração com outras bibliotecas e ferramentas também é facilitada (TECHNOLOGIES, 2020). Vue é um *framework* progressivo, isso quer dizer que pode ser integrado em apenas partes da aplicação. Vue foi projetado para ser possível a adoção de forma incremental. Seu foco principal é na camada visual da aplicação, facilitando integrações com outras bibliotecas e outras aplicações.

3.6.3 Vuex

Ao implementar uma aplicação em Vue, um dado pode ser requerido em vários arquivos. A atualização desse dado se torna cada vez mais complexo a medida do crescimento do projeto, onde a requisição e alteração pode ser feita em todos eles. Há o risco de uma alteração inadequada, e consequentemente um *bug*. Uma solução criada pelo Facebook é a arquitetura Flux, um esquema de fluxo de dados que tem por objetivo deixar a atualização dos dados o mais explicito possível. Assim, o desenvolvimento de código e a depuração de *bugs* se tornam mais simples (LIMA; PETRUCELLI; SANTO, 2019).

A arquitetura Flux mantém todos os dados centralizados em um único ponto. Nenhuma outra parte do código precisa conhecer como modificar o estado da aplicação pois os *stores* fazer isso internamente através das *actions* (MARIANO, 2018).

A *store* funciona como uma central de dados. Na figura 9, ilustra-se como o Vuex funciona. Quando uma ação requere uma modificação de um determinado dado, ela faz uma solicitação, um *commit*, só assim, haverá uma mudança do estado, uma *mutation*. Essa mutação ocorrerá de forma síncrona e atualizará todos os lugares em que o dado é utilizado. E quando acontece alguma operação assíncrona, como por exemplo uma requisição com alguma API,

a *action* é disparada e esta executa todo o fluxo previsto pela arquitetura. Isso garante que a alteração só ocorrerá de uma forma previsível, evitando algum conflito não mapeado.

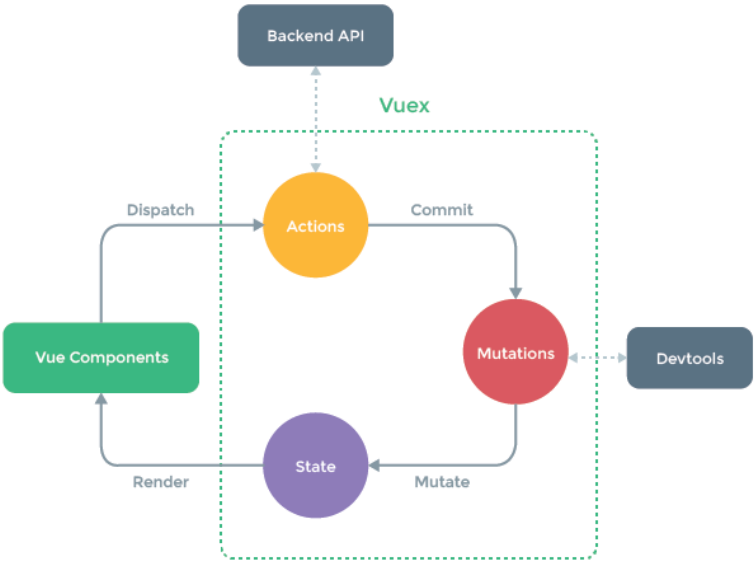

Figura 9 – *Vuex*

Fonte: (VUEX, 2020)

Assim, o Vuex funciona como um meio facilitador do armazenamento e uso de dados entre as páginas de uma aplicação Vue. Outra vantagem do uso do Vuex, são os recursos de depuração e mapeamento do estado nas ferramentas do navegador, importante para o entendimento de alguma falha dos testes (TECHNOLOGIES, 2020).

3.6.4 Jest e Vue tests

Os testes são um passo importante para garantir a qualidade dos componentes, testando-os para garantir o funcionamento e diminuir a chance de *bugs*. Essa tarefa é complexa e exige um certo recurso, dado isso, utilizar o Jest e *Vue Test Utils* é um facilitador para gerar mais velocidade tanto nos testes quanto na produção dos mesmo.

O Jest é um *framework* de JavaScript para a realização de testes unitários. O diferencial deste é a pouca configuração necessária para o funcionamento.

O *Vue Test Utils* complementa o Jest por conter funções que permitam uma melhor simulação dos componentes em Vue. De acordo com a documentação do Vue, *Vue Test Utils* é uma biblioteca oficial de testes do Vue. A abordagem adotada é o a orientada a contratos e interfaces públicas. Assim, cada função e interface tem testes separados que garantem o funcionamento de ambas. Além disso, técnica de *mock* também é utilizada afim de simular o comportamento de dados e objetos reais. (VUEJS, 2020)

Embora a cobertura de código seja ainda uma métrica importante e avaliada neste trabalho, uma visão focada nesse conceito geraria testes fracos, ou seja, testes que apresentam garantir algo, mas basta alguma mudança, supostamente coberta, que o mesmo falha. (NEVES, 2021)

3.6.5 *Flags*

A utilização de *flags* é importante para liberação de acesso as novas atualizações para cliente. Raramente uma nova funcionalidade ou alteração de estilo é liberada para todos, utiliza-se de *flags* para habilitar essas modificações aos poucos, afim de realizar um teste já em produção para amenizar os erros que podem surgir.

3.6.6 Comunicação sem intermédio da GAE

O Google App Engine(GAE), é um serviço de aplicações em nuvem criada e disponibilizada pelo Google, cujo principal objetivo é permitir a publicação de aplicativos com o mínimo de configuração possível. O tipo de computação em nuvem utilizada pela GAE é denominada como PaaS (Platform as a Service) e significa que ela oferece seus produtos através de uma plataforma, fornecendo infraestrutura para APIs, gerenciamento de recursos, acesso a banco de dados e um servidor Web. Assim, a GAE fica responsável por cuidar da ambiente, segurança deste, manter o sistema operacional usado internamente e o hardware onde o serviço está alocado. (CLOUD, 2020)

Atualmente, ela disponibiliza infraestrutura para a implantação de Apps que são feitas com as linguagens de programação mais conhecidas, como Python, Java e Go. Com ela é possível depurar os erros, avisos e outros tipos de logs para entender algum *bug* ocorrido em produção. (CLOUD, 2020)

Apesar das características benéficas, há também as desvantagens que envolvem principalmente custo e limitações. Pretende-se que a comunicação dos projetos das interfaces com os serviços seja direta, sem intermédios, o que permite uma configuração mais personalizada e também, a inutilização do código em Python, já que a versão Python está depreciada e em breve não haverá mais suporte em nenhuma ferramenta.

Dessa forma, um serviço para a comunicação foi previamente criado e disponibilizado como uma biblioteca e devidamente documentado. O presente trabalho inaugura a utilização deste serviço no projeto, importando-o corretamente para poder ser usado facilmente em qualquer parte dele.

3.6.7 Design System

O *Design System* consiste em um documento vivo que engloba os componentes e propriedades de um produto ou serviço com o objetivo de servir como consulta na hora do

planejamento de novas *features* e interfaces além de integrar todas as características relacionadas. (SERRADAS, 2018)

Os benefícios da construção e utilização do DS abrange todos os aspectos de uma organização, a comunicação por ser mais assertiva e empática, os negócios, por construir um identidade para a empresa. Os benefícios estão citados a seguir:

- Maior produtividade. Como as características estão definidas e de fácil acesso, assim como os componentes já estão prontos para ser reutilizados. A adequação não é necessária, promovendo mais rapidez no desenvolvimento.
- Conhecimento compartilhado e de fácil acesso.
- Discussão mais inclusive acerca de alternativa de *Design* e de melhores práticas. As pessoas conseguem participar mais ativamente e compartilhar conhecimento referente ao tema e estudos de caso
- Curva de aprendizado rápida. O DS é feito abstraindo qualquer informação redundante. Sendo um documento intuitivo e auto-explicativo.

O *Design System* faz uma coletânea dos seguintes itens:

- Componentes: Inclusão dos elementos atômicos e partículas
- Iconografia: Padronização dos ícones e inclusão de todos em que sua utilização é prevista.
- Estilos de texto e cores: Composição da identidade da marca

Além disso, opcionalmente pode conter *Motion Design*, para adequar comportamento de animação dos elementos na interface. E Comunicação e Linguagem, para explicar como o produto vai se comunicar, as expressões e limitações de escrita. Na figura 10 ilustra-se o sistema de *Design* (OTO, 2017).

O DS, nesse trabalho, foi construído recente e ainda está no estágio inicial de uso. A primeira integração com o projeto alvo também foi realizada previamente pela autora deste trabalho em uma tarefa própria e separada. Por ser recente, muitos componentes ainda não estão implementados, embora a definição e documentação do mesmo já esteja feita.

No estágio de engenharia reversa, todos os componentes identificados para a composição do DS foram salientados. Como sua implementação nesse sistema não existia, então a mesma ação entrou para o escopo da tarefa de modernização.

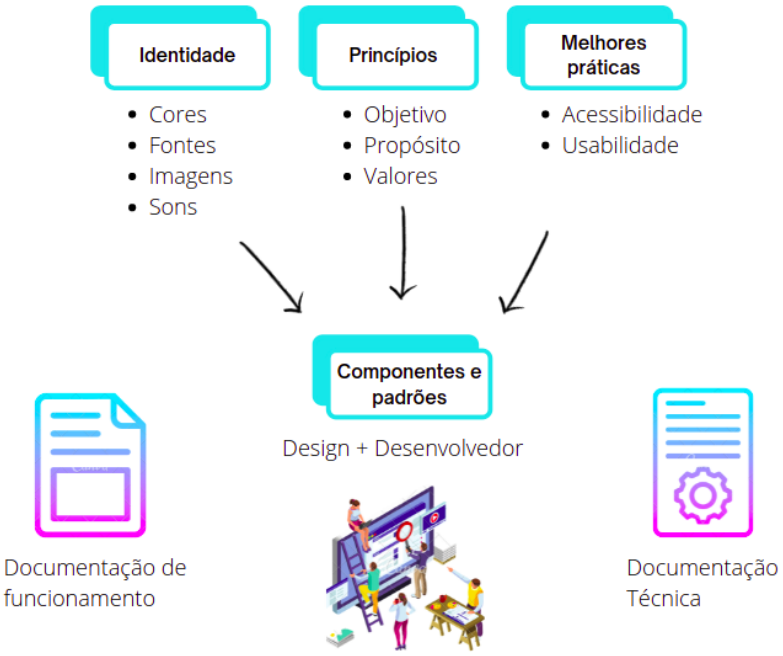

Figura 10 – *Design System*

Fonte: Autor

3.7 Implementação

Como o projeto alvo é muito extenso, foi definido que a migração fosse realizada incrementalmente. Cada equipe moderniza aos poucos as partes designadas a eles, seguindo as boas práticas de codificação.

Primeiro, uma parte do projeto foi escolhida e estudada para entender o fluxo do desenvolvimento. A engenharia reversa nessa fase é super importante para não reinventar a roda e reaproveitar todas as partes boas, como funções. A parte alvo foi subdividida em seções para simplificar as entregas, assim, cada seção ou mini seção é uma entrega. A nova arquitetura também foi estudada com o objetivo de entender o processo de migração. Dessa forma, o desenvolvimento foi iniciado.

Alguns componentes previstos para estar no DS foram devidamente implementados e testados, após o *release* do componente no DS, o mesmo estava pronto para uso e importado para o projeto alvo.

Após o importe, os mesmos foram testados na nova arquitetura. Em seguida, iniciou-se a refatoração do código antigo com os testes (TDD). Essa refatoração foi realizada em uma *branch* dita como *major*, a que conteve todas as modificações e incrementos desse processo.

Em paralelo, a *Store* foi criada, contendo seus testes e suas ações (funções). Após a implementação dessa parte, a mesma foi incorporada na *branch major*. Depois que a comunicação direta com a API foi bem sucedida, estes foram inseridos na *Store*.

Em primeira instância, até a que API terminasse de ser construída, dados *mockados*, ou seja, fictícios foram utilizados para a visualização dos mesmo na nova página.

Após a funcionamento correto de cada parte, iniciou-se a junção e após, testes.

Com tudo desenvolvido, foram realizados uma sequência de testes manuais de QA para assegurar o comportamento. Os mesmos não devem ser realizados por quem desenvolveu, pois acredita-se que a utilização já esteja treinada. Aconselha-se outras pessoas que não participaram ativamente do processo para experimentar, assim, falhas são encontradas mais facilmente e há mais sugestões de melhorias.

4 RESULTADOS E DISCUSSÕES

Para viés de comparação, dados da tarefa foram comparados com dados de uma tarefa cuja nova funcionalidade é de mesmo porte que a retratada no presente trabalho, porém não houve refatoração, testes unitários e modernização relacionada a ela. Foi possível observar que a implementação de testes reduziu em 43% os *bugs* encontrados no processo de teste de QA. Em contrapartida, o tempo de entrega aumentou em torno de 150%. Um fato observado é que as melhorias notificadas nessa fase final do processo reduziu em 56%, isto demonstra que o planejamento e o desenvolvimento conseguiu abranger todos os melhores aspectos da funcionalidade.

Outro ponto observado é a maior transparência que a nova funcionalidade trouxe para os clientes, que puderam entender que melhor o gasto e o tanto que o mesmo representa em porcentagem do limite contratual. Portanto, há casos que os clientes irão atualizar o contrato, aumentando ou diminuindo o limite.

Como o projeto do *Design System* é bem recente, é esperado que apareça alguns problemas ainda desconhecidos e imprevistos na hora de utilizá-lo no projeto alvo. A curva de adversidades tende a diminuir conforme o aprendizado e documentação, e após a queda da curva, a performance dos profissionais tende a melhorar, pois não precisarão se preocupar com tantos detalhes e padronização de *Design*. Assim, como o desenvolvimento se encontra no primeiro estágio, houve alguns contratempos que dificultaram e atrasaram a entrega do trabalho.

Poucos componentes estavam desenvolvidos nesse sistema, portanto, para complementá-lo, os identificados para essa transição foram implementados para depois ser importado no projeto alvo. Ao realizar esse processo, o desenvolvimento ficou entrelaçado em *releases* e correções de erros. Já que, apesar dos testes unitários no DS, ao utilizá-los na parte alvo, algumas características do comportamento não condiziam com o esperado, e outras não tinham sido mapeadas totalmente no planejamento, o que implicou em novos incrementos.

Como aprendizado, discute-se o melhor processo de utilização do DS para quando o componente não está ainda desenvolvido. Há identificada três formas, descritas a seguir:

1. Implementar o componente primeiro no DS, e após aprovação e *release*, utilizar no projeto. Essa foi a ação inicialmente adotada, porém esta se mostra bastante maçante, pois o funcionamento no DS não significa necessariamente em um comportamento ideal no projeto. Quando alguma falha acontece, é preciso passar pelo mesmo processo de aprovação e *release* mais que o desejado. Isso atrasa a entrega e enfada o time de desenvolvimento.

2. Implementar o componente primeiro no DS, e lançar uma versão alpha ou beta. Essa ação foi adotada após perceber que a medida anterior não era uma boa escolha.

3. Implementar o componente primeiro no projeto, e depois refatorar e levar para o DS após a entrega. Esta induz retrabalho e esforço a mais. Ao refatorar o código para o DS, adequações devem ser feitas como modificação de qualquer conflito de biblioteca. E depois de retrabalho de implementação, terá o de retirada do componente do projeto, de importe do componente para o projeto, para assim, posteriormente verificar se o comportamento permanece o mesmo. O fator que pode tornar essa medida interessante é a entrega que pode ser mais rápida, pois o retrabalho é feito após a entrega.

Outro contratempo identificado foi o uso de TypeScript no DS, linguagem JS com tipagem estática. Ao utilizá-lo no projeto alvo, erros na hora do *build* surgiam pelas características do TS não serem identificadas. O que consequentemente induziu a correção e o desenvolvimento dos componentes de forma que o TS não reclamasse a falta de tipagem. Essa ação é um pouco controvérsia, pois põe em pauta a real necessidade do TS para o DS. Se a tipagem, maior beneficio induz erros nos outros projetos que não entendem TS, e portanto, o evita-se o seu uso, porque, então, é utilizado?

Outra dificuldade foi a falta de experiência e conhecimento do código existente e com as novas tecnologias a serem utilizadas, como a de testes e com a biblioteca de testes, *Vue Tets Utils*.

Apesar da dificuldade inicial do aprendizado sobre testes, o que também delongou as entregas, o uso de TDD se mostrou bastante promissor. Iniciar o desenvolvimento com testes fomentou melhor qualidade do código e discussão a cerca do comportamento da funcionalidade para cada variação do mesmo. Acredita-se que em próximos desenvolvimentos, a implementação de testes será mais rápida devido ao conhecimento agregado.

Sabe-se que a modernização incluída no desenvolvimento da funcionalidade em questão atrasou um pouco a entrega. Surge então o questionamento sobre a repartição em duas histórias, uma de modernização e outra apenas da funcionalidade. Se o planejamento fosse esse, primeiro a nova função seria implementada, usando a arquitetura anterior e após a entrega, a migração seria realizada. Esse planejamento é interessante quando o tempo de desenvolvimento é curto ou quando há urgência para que a funcionalidade entre em produção. No entanto, como não é o caso abordado, o processo de modernizar enquanto uma novas funcionalidades são construídas se mostrou complementar e mais rápido, pois a separação em duas entregas implicaria em mais sessões de testes de QA, sendo que os primeiros não seriam proveitosos.

5 CONCLUSÃO

Em geral, presume-se que um projeto terá começo, meio e fim. Contudo, o mesmo não acontece em projetos de desenvolvimento de *software*. A manutenção será necessária até o fim do ciclo de vida, onde o projeto será encerrado e depreciado.

Apesar de inicialmente custoso devido ao tempo investido, o processo de modernização é necessário e crucial para estender a vida do *software* e evitar possíveis quebras por ferramentas depreciadas.

Porém, o processo de modernização necessita ser avaliado primeiro, não deve ser iniciado simplesmente por ter ferramentas novas no mercado e sem uma análise prévia do custo-beneficio. Sempre haverá novas tecnologias, e isto não implica que o *software* se tornou legado ou atrasado.

Um programa se tornou legado quando o mesmo se demonstra difícil de ser entendido, a manutenção é custosa, linguagem de programação antiga e quando não há testes implementados, seja unitários ou outros.

Como sugestão de melhoria para usos futuros do DS, utilizar versionamento e assim poder realizar testes no projeto sem os passos relacionados a *release*.

Por fim, um *software* estagnado e sem manutenção dificilmente trará benefícios a longo prazo, sendo fadado à inutilização. O único fator que estenderá seu ciclo de vida é a manutenção e modernização planejada e concisa.

REFERÊNCIAS

ANICHE, M. *Test-Driven Development*. 2014. Disponível em: https://tdd.caelum.com.br/. Acesso em: 03/12/2021. Citado na página 29.

APPLE. *macOS Monterey 12.3 Beta Release Notes*. 2020. Disponível em: https://developer.apple.com/documentation/macos-release-notes/macos-12_3-release-notes#Python. Acesso em: 03/12/2021. Citado na página 31.

CATOLINO, G. Does source code quality reflect the ratings of apps? In: *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. [S.l.: s.n.], 2018. p. 43–44. Citado 2 vezes nas páginas 16 e 17.

CHERVENSKI, A. S. Entendimento sobre sistemas legados à luz da teoria fundamentada em dados. 2019. Universidade Federal do Pampa, 2019. Citado 2 vezes nas páginas 16 e 17.

CLOUD, G. *App Engine*. 2020. Disponível em: https://cloud.google.com/appengine. Acesso em: 03/11/2021. Citado na página 39.

ERDIL, K. et al. Software maintenance as part of the software life cycle. *Comp180: Software Engineering Project*, 2003. v. 1, p. 1–49, 2003. Citado na página 21.

FEATHERS, M. *Working Effectively with Legacy Code: WORK EFFECT LEG CODE _p1*. [S.l.]: Prentice Hall Professional, 2004. Citado na página 17.

GALHARDO, A. *Entenda o que é Roi e como calcular o da sua empresa*. 2018. Disponível em: https://bizcapital.com.br/blog/roi-como-mensurar/?utm_campaign=AW-20210519-DYNAMIC-SEARCH-ADS&utm_source=adwords&utm_medium=all_pages&gclid=CjwKCAiAhreNBhAYEiwAFGGKPOPFhCu47LVpTmgpezu5AP4O9rX8r_RYwiyt_y1kTBe2x-XzvmYovhoC5tAQAvD_BwE. Acesso em: 03/11/2021. Citado na página 31.

GENTIL, R. J. Um estudo de caso sobre como a introdução de comportamentos adaptativos em uma aplicação web legada impacta a cobertura de código. 2020. Universidade Federal de São Carlos, 2020. Citado 3 vezes nas páginas 27, 28 e 29.

GREIF, S.; BENITTE, R. *Changes Over Time*. 2020. Disponível em: https://2020.stateofjs.com/en-US/technologies/. Acesso em: 13/06/2021. Citado 2 vezes nas páginas 14 e 15.

GROUP, S. et al. Chaos manifesto 2015. *The Standish Group International*, 2015. 2015. Citado 2 vezes nas páginas 25 e 33.

IBM. *COVID-19 and the future of business*. 2020. Disponível em: https://www.ibm.com/thought-leadership/institute-business-value/report/covid-19-future-business. Acesso em: 15/06/2021. Citado na página 13.

LIMA, L. G.; PETRUCELLI, E. E.; SANTO, F. do E. Visão geral sobre o gerenciamento de estado no vue. js com a biblioteca vuex. *Revista Interface Tecnológica*, 2019. v. 16, n. 1, p. 56–66, 2019. Citado na página 37.

MAJTHOUB, M.; QUTQUT, M. H.; ODEH, Y. Software re-engineering: an overview. In: IEEE. *2018 8th International Conference on Computer Science and Information Technology (CSIT)*. [S.l.], 2018. p. 266–270. Citado 2 vezes nas páginas 23 e 24.

MALINOVA, A. Approaches and techniques for legacy software modernization. *Scientific Works, Plovdiv University*, 2010. v. 37, p. 77–85, 2010. Citado 3 vezes nas páginas 21, 23 e 24.

MARIANO, R. *Arquitetura Flux*. 2018. Disponível em: https://medium.com/engenharia-arquivei/arquitetura-flux-26a419871ade. Acesso em: 03/12/2021. Citado na página 37.

NEVES, M. V. da S. *Desmistificando testes de unidade no Vue*. 2021. Disponível em: https://www.alura.com.br/artigos/desmistificando-testes-de-unidade-no-vue. Acesso em: 03/11/2021. Citado na página 39.

OPUSSOFTWARE. *Manutenção de Software – Definição e melhores práticas*. 2018. Disponível em: https://www.opus-software.com.br/manutencao-de-software-definicao/. Acesso em: 03/11/2021. Citado na página 22.

OTO, O. *Build and share a world-class design system*. 2017. Disponível em: https://medium.com/sketch-app-sources/build-and-share-a-world-class-design-system-with-sketch-45d1104420f1. Acesso em: 03/11/2021. Citado na página 40.

PATTON, R. *Software testing*. [S.l.]: Pearson Education India, 2006. Citado na página 26.

PERITO, J. *[Guia completo] Como lidar com software legado?* 2019. Disponível em: https://blog.geekhunter.com.br/lidando-com-codigo-legado/. Acesso em: 13/06/2021. Citado na página 16.

PYTHON. *Sunsetting Python 2*. 2020. Disponível em: https://www.python.org/doc/sunset-python-2/. Acesso em: 03/12/2021. Citado na página 31.

RAJLICH, V. Software evolution and maintenance. In: *Future of Software Engineering Proceedings*. [S.l.: s.n.], 2014. p. 133–144. Citado 2 vezes nas páginas 22 e 33.

RAKSI, M. et al. Modernizing web application: case study. 2017. 2017. Citado na página 22.

REIS, T. *Entendendo os conceitos do webpack*. 2018. Disponível em: https://medium.com/rocketseat/entendendo-e-dominando-o-webpack-4b2e8b3e02da. Acesso em: 02/11/2021. Citado na página 37.

ROSENBERG, L. H.; HYATT, L. E. Software re-engineering. *Software Assurance Technology Center*, 1996. Citeseer, p. 2–3, 1996. Citado na página 25.

RUPARELIA, N. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 2010. v. 35, p. 8–13, 05 2010. Citado na página 19.

SERRADAS, V. *Entendendo Design Systems*. 2018. Disponível em: https://brasil.uxdesign.cc/entendendo-design-system-f375bbb6f704. Acesso em: 03/11/2021. Citado na página 40.

SOMMERVILLE, I. *Engenharia de Software*. [S.l.]: Pearson Universidades, 2019. 768 p. Citado 8 vezes nas páginas 16, 20, 21, 24, 25, 26, 28 e 29.

STATISTICS; DATA. *The Most Popular Programming Languages – 1965/2021 – New Update*. 2020. Disponível em: https://statisticsanddata.org/data/the-most-popular-programming-languages-1965-2021/. Acesso em: 15/06/2021. Citado na página 13.

TECHNOLOGIES, C. *VueJS + VueX: Vue na construção de interfaces de usuário*. 2020. Disponível em: https://blog.cedrotech.com/vuejs-vuex-vue-na-construcao-de-interfaces-de-usuario. Acesso em: 03/11/2021. Citado 2 vezes nas páginas 37 e 38.

VUEJS. *Recommendations*. 2020. Disponível em: https://vuejs.org/v2/guide/testing.html. Acesso em: 03/11/2021. Citado na página 38.

VUEX. *What is Vuex?* 2020. Disponível em: https://vuex.vuejs.org/#what-is-a-state-management-pattern. Acesso em: 03/12/2021. Citado na página 38.