

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO  
CÂMPUS CAMPINAS

VINÍCIUS CUNHA DE ALMEIDA

**IMPLEMENTAÇÃO DA PIRÂMIDE DE AUTOMAÇÃO DE TESTES EM UMA  
APLICAÇÃO WEB**

CAMPINAS

2021

VINÍCIUS CUNHA DE ALMEIDA

**IMPLEMENTAÇÃO DA PIRÂMIDE DE AUTOMAÇÃO DE TESTES EM UMA  
APLICAÇÃO WEB**

Trabalho de Conclusão de Curso apresentado como exigência parcial para obtenção do diploma do Curso de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia Câmpus Campinas.

Orientador: Prof. Dr. Andreiuid Sheffer Corrêa

CAMPINAS

2021

Ficha catalográfica  
Instituto Federal de São Paulo – Câmpus Campinas  
Biblioteca  
Rosângela Gomes – CRB 8/8461

Almeida, Vinícius Cunha de

A447i Implementação da pirâmide de automação de testes em uma aplicação web /  
Vinícius Cunha de Almeida. – Campinas, SP: [s.n.], 2021.

42 f. il. :

Orientador: Andreiuid Sheffer Corrêa

Trabalho de Conclusão de Curso (graduação) – Instituto Federal de  
Educação, Ciência e Tecnologia de São Paulo Câmpus Campinas. Curso de  
Tecnologia em Análise e Desenvolvimento de Sistemas, 2021.

- Software - Testes. 2. Software – Controle de qualidade. 3. Aplicações Web. I. Instituto Federal de Educação, Ciência e Tecnologia de São Paulo Câmpus Campinas. Curso de Tecnologia em Análise e Desenvolvimento de Sistemas. II. Título.

ATA N.º 26/2021 - TADS-CMP/DAE-CMP/DRG/CMP/IFSP

Ata de Defesa de Trabalho de Conclusão de Curso - Graduação

Na presente data, realizou-se a sessão pública de defesa do Trabalho de Conclusão de Curso intitulado **IMPLEMENTAÇÃO DA PIRÂMIDE AUTOMAÇÃO DE TESTES EM UMA APLICAÇÃO WEB** apresentado(a) pelo(a) aluno(a) **Vinicius Cunha de Almeida (CP3001784)** do Curso **SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS** (Câmpus Campinas). Os trabalhos foram iniciados às 20:00 pelo(a) Professor(a) presidente da banca examinadora, constituída pelos seguintes membros:

Membros	Instituição	Presença (Sim/Não)
Andreiwid Sheffer Corrêa (Presidente/Orientador)	IFSP	Sim
Rafael da Silva Pereira (Examinador 1)	BEEs	Sim
Fabio Feliciano de Oliveira (Examinador 2)	IFSP	Sim

Observações:

A banca examinadora, tendo terminado a apresentação do conteúdo da monografia, passou à arguição do(a) candidato(a). Em seguida, os examinadores reuniram-se para avaliação e deram o parecer final sobre o trabalho apresentado pelo(a) aluno(a), tendo sido atribuído o seguinte resultado:

☒ Aprovado(a)

☐ Reprovado(a)

Proclamados os resultados pelo presidente da banca examinadora, foram encerrados os trabalhos e, para constar, eu lavrei a presente ata que assino em nome dos demais membros da banca examinadora.

Câmpus Campinas, 12 de novembro de 2021

Documento assinado eletronicamente por:

- Andreiwid Sheffer Correa, PROFESSOR ENS BASICO TECN TECNOLOGICO, em 12/11/2021 21:12:30.
- Fabio Feliciano de Oliveira, PROFESSOR ENS BASICO TECN TECNOLOGICO, em 12/11/2021 21:12:47.

Este documento foi emitido pelo SUAP em 12/11/2021. Para comprovar sua autenticidade, faça a leitura do QRCode ao lado ou acesse <https://suap.ifsp.edu.br/autenticar-documento/> e forneça os dados abaixo:

Código Verificador: 257136  
Código de Autenticação: fdd2004bb0



ATA N.º 26/2021 - TADS-CMP/DAE-CMP/DRG/CMP/IFSP

## **AGRADECIMENTOS**

Agradeço a todos os professores e colaboradores do IFSP - Câmpus Campinas que fizeram a diferença em minha formação como profissional e cidadão. Em especial, gostaria de agradecer ao meu orientador Prof. Dr. Andreiuid Sheffer Corrêa pela confiança, liberdade e parceria na escrita desta monografia.

Aos meus pais Graciela e Valdecir, por me darem total suporte e incentivo em toda a minha jornada acadêmica. À Giovana por me apoiar de maneira integral em todas as minhas escolhas.

Aos meus colegas de trabalho, pela paciência e pelos ensinamentos.

## RESUMO

Atualmente, a adoção de metodologias ágeis vem se tornando recorrente na indústria de software. A partir disso, os ciclos para as entregas estão cada vez mais velozes, e as formas de validações em projetos ágeis passam a ser questionadas. Considerando a resistência do emprego de automação de testes por uma parcela da indústria e sabendo que a prevenção de erros em uma aplicação é menos custoso do que encontrá-los e corrigi-los, este trabalho tem como objetivo detalhar a implementação da pirâmide de automação de testes em uma aplicação web. Através de uma análise comparativa, foi desenvolvido uma aplicação do tipo *marketplace* para auxiliar na exemplificação e detalhamento da implementação das três camadas presentes na pirâmide de testes, sendo elas, testes unitários, testes de integração e testes *end-to-end*. Após a implementação das automações, foi coletado relatórios sobre a cobertura dos testes, tempo de execução de cada camada e como as automações auxiliam na prevenção de futuros erros na etapa do desenvolvimento.

**Palavras-chave:** Software - Testes; Software - Controle de qualidade; Aplicações Web.

## **ABSTRACT**

Currently, the adoption of agile methodologies in the software industry is becoming more usual. Because of that, the delivery cycles are faster, and the validation approaches in agile projects start to be questionable. Considering the resistance to implementing automation tests from a part of the software industry and knowing that error prevention is cheaper than finding and fixing an error later, this paper has a purpose to detailing the implementation of the pyramid of automated tests in a web application. Through comparative analysis, was developed a marketplace application for assisting on the exemplification and detail of the implementation of the three types of layers present in the pyramid test, namely, unit tests, integration tests, and end-to-end tests. After the implementation, reports were collected about the test coverage, execution time of each layer, and evidence about how the automation prevented future application errors in the development stage.

**Keywords:** Software - Tests; Software - Quality Control; Web Applications.

## LISTA DE FIGURAS

Figura 1 – Pirâmide de testes de Rouchon (1990).....	17
Figura 2 – Pirâmide de testes de Mike Cohn (2009) descrita por Vocke (2018) .....	18
Figura 3 – Pirâmide de testes adaptada pelo Google.....	19
Figura 4 – Componente de Header das aplicações analisadas.....	23
Figura 5 – Prototipação da aplicação no Figma.....	26
Figura 6 – Organização de pastas da aplicação.....	28
Figura 7 – Padrão da construção de páginas e componentes.....	27
Figura 8 – Telas da aplicação após o desenvolvimento.....	29
Figura 9 – Componente de barra de busca do <i>marketplace</i> .....	30
Figura 10 – Teste unitário para a validação do componente de header.....	31
Figura 11 – Teste unitário para simular ação de click no botão de pesquisa.....	31
Figura 12 – Validação da adição do mesmo item ao carrinho de compras.....	33
Figura 13 – Validação do fluxo de comprar um produto adicionando pelo carrinho.....	34
Figura 14 – Resultados da implementação da pirâmide de testes.....	36
Figura 15 – Falha no terminal de um teste de unidade.....	37
Figura 16 – Relatório de cobertura dos testes unitários.....	38
Figura 17 – Parte de código não coberta pelo teste de unidade.....	39



## LISTA DE TABELAS

Tabela 1 – Tipos de interações em um <i>marketplace</i> .....	21
Tabela 2 – Análise comparativa entre marketplaces.....	22
Tabela 3 – Componente do produto das aplicações analisadas.....	24
Tabela 4 – Ferramentas instaladas para a realização da aplicação e dos testes.....	25
Tabela 5 – Estruturas de pastas da aplicação.....	27

## LISTA DE SIGLAS

XP	Extreme Programming
E2E	End-to-end
B2B	Business-to-business
B2C	Business-to-consumer
G2G	Government-to-government
B2G	Business-to-government
C2G	Consumer-to-government
G2B	Government-to-business
C2B	Consumer-to-business
C2G	Consumer-to-government
B2C	Business-to-consumer
C2C	Consumer-to-consumer

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO</b>	<b>11</b>
<b>2</b>	<b>JUSTIFICATIVA</b>	<b>12</b>
<b>3</b>	<b>OBJETIVOS</b>	<b>13</b>
3.1	Objetivo Geral	13
3.2	Objetivos Específicos	13
<b>4</b>	<b>FUNDAMENTAÇÃO TEÓRICA</b>	<b>14</b>
4.1	Qualidade	14
4.2	Qualidade de software	15
4.3	Metodologias ágeis	16
4.4	Testes manuais e automatizados	16
4.5	Pirâmide de automação de testes	17
4.6	Marketplace	20
<b>5</b>	<b>METODOLOGIA</b>	<b>22</b>
5.1	Análise comparativa	22
5.1.1	Características e componentes	23
5.2	Ferramentas	25
5.3	Desenvolvimento da aplicação	26
5.3.1	Estrutura da aplicação	27
5.3.2	Etapas de desenvolvimento	28
5.4	Implementação da pirâmide de testes automatizados	30
5.4.1	Testes unitários	30
5.4.2	Testes de integração	32
5.4.3	Testes <i>end-to-end</i>	34
<b>6</b>	<b>RESULTADOS</b>	<b>36</b>
<b>7</b>	<b>CONCLUSÃO</b>	<b>40</b>
	<b>REFERÊNCIAS</b>	<b>41</b>

## 1 INTRODUÇÃO

Segundo um formulário de pesquisa feito pela QASymphony, com parceria da TechWell coletando respostas de funcionários de empresas de médio e grande porte, 76% das empresas estão adotando metodologias ágeis e 72% estão automatizando seus testes de software (YACKEL, 2018).

A maioria dos métodos ágeis sugerem que todos envolvidos em um projeto de software controlem a qualidade, sempre que possível, pois prevenir erros é menos custoso do que achá-los e corrigi-los. A partir disso, as metodologias ágeis não contestam nenhum tipo de ação que possa melhorar a validação da qualidade, sendo o caso de algumas em específico recomendarem fortemente o uso de testes automatizados (BERNARDO, KON, 2008) .

Porém, muitas equipes pensam que os testes automatizados são custosos de escrever. Mas essa afirmação se baseia devido às camadas serem automatizadas de forma errada. Para solucionar isso, uma estratégia chamada pirâmide de testes automatizados é apresentada, sendo essa, uma abordagem que permite focalizar as automações em três diferentes camadas, são elas: testes unitários, teste de serviços e testes de interface de usuário (COHN, 2009).

A partir das informações apresentadas, este trabalho tem como objetivo demonstrar a implementação da pirâmide de automação testes em um software de conhecimento popular. Dessa forma, a aplicação que será desenvolvida para a aplicabilidade desta estratégia de testes automatizados será do tipo marketplace.

Além disso, este projeto visa apresentar detalhes da implementação de cada camada da pirâmide de testes, evidenciando o número de cada tipo de teste, a porcentagem de cobertura de código da aplicação, os benefícios obtidos e os possíveis erros prevenidos.

## 2 JUSTIFICATIVA

Nos últimos anos houve o aumento da adoção de metodologias ágeis nos processos de desenvolvimento de software na indústria, este fato traz à tona a discussão de como executar testes em projetos ágeis. Uma das principais diferenças entre as metodologias ágeis e tradicionais é a velocidade dos ciclos e das interações realizadas. Para isso, a automação de testes ajuda a tornar esse ciclo mais rápido, economizando o tempo testando pequenos pontos de códigos do sistema rapidamente e repetidamente (SALEEM et al., 2014).

Muitas metodologias ágeis recomendam que todas as pessoas envolvidas no projeto validem a qualidade do produto a todo o momento, pois afirmam que é menos custoso prevenir um erro do que achá-lo e corrigi-lo posteriormente (BERNARDO, KON, 2008).

Segundo Cohn (2009), a realização dos testes de software após a etapa do desenvolvimento não funciona. Com essa afirmação, o autor apresenta cinco razões para a falha dessa abordagem:

- É difícil melhorar a qualidade de um produto já construído;
- Erros continuam não sendo percebidos;
- O estado do projeto se torna difícil de medir;
- Oportunidades de feedbacks são perdidas;
- O teste tem uma maior probabilidade de ser incompleto.

A partir disso, Cohn (2009) apresenta a pirâmide de testes automatizados, um modelo conceitual para checar como a qualidade é organizada para garantir a qualidade do sistema continuamente em todos os níveis. Esta estratégia é usada atualmente para definir a prioridade dos testes automatizados em diferentes níveis, partindo do nível mais baixo (testes unitários), seguindo para uma camada intermediária (testes de integração) e por último os mais lentos de executar (testes de interface) (FREEMAN, RADZIWIŁŁ, 2020).

Sabendo da importância da validação contínua de um software com automação de testes para a prevenção de futuros erros e do aumento da utilização de metodologias ágeis na indústria de software, a implementação de uma estratégia é fundamental para sua realização de forma bem sucedida. Este projeto tem como proposta apresentar a implementação da pirâmide de automação de testes em uma aplicação web. Para isto, será desenvolvido o *front-end* de uma aplicação do tipo *marketplace* que será a base dos exemplos a serem demonstrados.

### **3 OBJETIVOS**

Com intuito de compreender amplamente o que está sendo abordado e as etapas realizadas na monografia, nesta seção é contido os tópicos sobre o objetivo geral e os objetivos específicos do trabalho.

#### **3.1 Objetivo geral**

Implementar a pirâmide de automação de testes em uma aplicação front-end web do tipo *marketplace*.

#### **3.2 Objetivos específicos**

- a) Investigar aplicações de marketplace já existentes, identificando as regras de negócios e as principais jornadas.
- b) Desenvolver front-end do marketplace.
- c) Implementar a pirâmide de testes automatizados no contexto da aplicação.
- d) Coletar dados de cobertura e qualidade de código.

## **4 FUNDAMENTAÇÃO TEÓRICA**

Nesta seção é apresentado os conceitos teóricos para a concepção desta monografia e desenvolvimento do projeto.

### **4.1 Qualidade**

Segundo Garvin (1984), pode-se ter cinco abordagens para definir o conceito de qualidade, elas são: abordagem transcendente da filosofia; abordagem baseada em produtos da economia; abordagem baseada no usuário do marketing, gerenciamento de operações, e da economia; abordagem baseada em manufatura; abordagem baseada no valor do gerenciamento de operações. As cinco abordagens estão sendo explicadas a seguir, conforme os ensinamentos de Garvin (1984):

A visão transcendental da filosofia é que a qualidade é algo com “excelência nativa”, que é reconhecida universalmente, se tornando algo absoluto. Esta definição está fortemente baseada na discussão de Platão sobre beleza, em que é um termo que não pode ser definido, apenas ser reconhecido através da experiência.

A abordagem baseada em produtos é de que a qualidade é mensurável e precisa. Com esta visão, a qualidade de um produto está altamente atrelada com a quantidade ou atributos que um produto possui. As definições de qualidade baseadas em produtos apareceram primeiro na literatura relacionada a área de economia, sendo apresentados dois caminhos para a definição de qualidade. O primeiro é de que a qualidade somente pode ser obtida com um custo alto e o segundo, onde a qualidade é vista como uma característica específica dos bens, ao invés de ser algo que é atribuído a eles.

Clientes têm preferências e necessidades individuais, aqueles que cumprem suas expectativas são definidos de alta qualidade. A partir disso, a abordagem baseada no valor do usuário, tem base em três literaturas. A primeira delas é a do marketing, que trouxe a definição de “pontos ideais”, ou seja, um produto com combinação precisa de atributos traz uma grande satisfação ao cliente. Na área de gerenciamento de operações, traz o conceito de que um produto com qualidade é quando está apto para o uso. Já na área da economia, a qualidade de um produto pode ser notada na curva de mudança de sua demanda.

Na abordagem baseada na manufatura, as principais preocupações são relacionadas a engenharia e a prática de manufatura. Basicamente, todas as definições dessa abordagem trazem à tona a definição de que a qualidade está ligada a conformidade com os

requerimentos, isto significa que, a excelência é atingir as especificações. Nesta visão a qualidade é definida para simplificar o controle de engenharia e da produção. Além disso, a melhoria na qualidade resulta em uma redução de custos, pois previne defeitos que precisam ser reparados futuramente.

Por último, a abordagem baseada no valor define a qualidade em termos de custos e preços. Sendo assim, um produto com qualidade é aquele que fornece uma boa performance com um preço acessível ou conformidade com um custo aceitável.

## 4.2 Qualidade de software

A definição de qualidade de software pode ser determinada como uma gestão efetiva que constrói um produto útil que gera ganhos ao fabricante e aos usuários. A partir disso, três pontos importantes são apresentados: gestão de qualidade efetiva, produto útil e agregação de valor ao fabricante e ao usuário. A gestão de qualidade efetiva é um fator crucial para construir um software de qualidade, isto é, realizando um processo com aspectos administrativos que trazem controle e equilíbrio para evitar uma qualidade inadequada. Um produto útil carrega características que o usuário final deseja, além disso, satisfaz as exigências dos interessados com confiabilidade. Por último, a agregação de valor ao fabricante e ao usuário decorre dos benefícios gerados pelo produto em ambas as partes, para as empresas esse valor pode ser convertido em menos esforços de manutenção, permitindo assim, os engenheiros de software aprimorarem e criarem funcionalidades. Já para usuário, o valor se apresenta de forma que o produto proporciona benefícios de algum processo relacionado ao seu negócio (PRESSMAN, 2011).

Os fundamentos do gerenciamento da qualidade foram definidos pela indústria manufatureira para melhorar as qualidades dos produtos. Com isso, propuseram que a qualidade estava relacionada a conformidade com as especificações e a noção de tolerância. Porém, a qualidade de software não é diretamente comparável a essa definição manufatureira, pois a ideia de tolerância não é aplicável em sistemas digitais, devido a haver diversas interpretações das especificações por parte dos desenvolvedores e *stakeholders* (partes interessadas que influenciam nos requisitos do sistema), além de as especificações serem definidas por diversos *stakeholders* e o não cumprimento de sua totalidade pode ser um fator para um sistema de baixa qualidade. Por tanto, a qualidade de software é um processo subjetivo, e a equipe de qualidade deve ser capaz de decidir se foi alcançado um nível de qualidade aceitável ao longo do processo de desenvolvimento (SOMMERVILLE, 2011).



### 4.3 Metodologias ágeis

As metodologias ágeis são uma resposta a ineficiência por parte de metodologias tradicionais no desenvolvimento de software. No entanto, as metodologias ágeis, em sua maioria, não trazem nenhuma novidade do que já apresentado nas tradicionais, sendo diferente apenas no enfoque das abordagens e os valores. Dessa maneira, são adaptativas conforme o desenvolvimento do projeto, nelas ocorrem a priorização em pessoas e não em processos, e também visam gastar menos tempo com as documentações, com isso, priorizando a implementação. Se destacam entre as várias metodologias ágeis a Scrum e a Extreme Programming (DOS SANTOS SOARES, 2004).

O Scrum é um *framework* utilizado desde o início dos anos 90 com intuito de gerenciar projetos de produtos complexos. Esta metodologia propõe-se a realizar entregas de alto valor, para isso, os times são associados a eventos, papéis, artefatos e regras, sendo cada uma dessas integradas e consideradas essenciais para a utilização e sucesso do Scrum (SCHWABER, SUTHERLAND, 2013).

A metodologia Extreme Programming (XP) é ideal na adoção de pequenas e médias equipes para lidar com desenvolvimento de softwares com requisitos vagos e que se alteram rapidamente. A XP visa satisfazer o cliente e as estimativas, para isso, é ideal que a equipe que esteja utilizando a metodologia esteja em sinergia e tenha o constante feedback entre os pares (DOS SANTOS SOARES, 2004).

### 4.4 Testes manuais e automatizados

Os testes manuais são validações feitas por um testador com valores inseridos e comparado com resultados esperados, coletando então as evidências e caso haja alguma divergência, isto é reportado aos desenvolvedores (SOMMERVILLE, 2011). Em um contexto com uso de metodologias ágeis, este tipo teste é visto inicialmente como testes exploratórios, ou seja, testes executados com novas informações a fim de obter melhores validações para a aplicação. (BORTOLUCI, DUDUCHI, 2015).

Os testes automatizados são *scripts* ou programas que executam tarefas repetidas, que têm como objetivo a validação de funcionalidades e verificações dos resultados obtidos. Os benefícios da automação de testes são que os cenários podem ser reproduzidos repetidamente com pouco esforço e executados com facilidade quando necessário. A possibilidade da reprodutibilidade de cenários idênticos e repetidos, resulta em uma garantia de

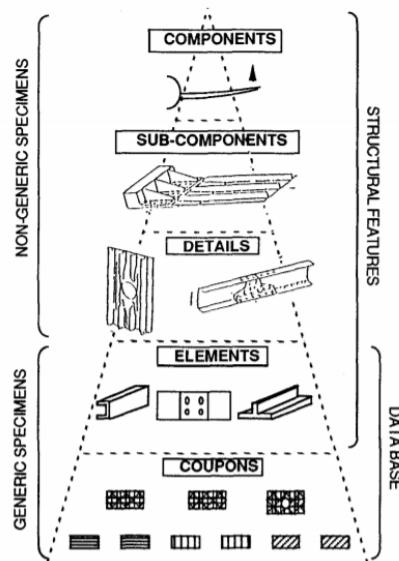
comportamentos não esperados. Isso se dá, devido a possibilidade de escrever cenários complexos que um teste manual não cobriria. A partir disso, a automação de testes permite solucionar os problemas dos testes manuais, garantindo uma maior qualidade de software e diminuindo a quantidade de erros na aplicação (BERNARDO, KON, 2008).

Geralmente, testes automatizados são mais rápidos do que os manuais, havendo benefícios na sua utilização em testes de regressão (reexecução de testes já realizados para verificar a confiabilidade do sistema). Porém, uma aplicação não pode ter somente testes automatizados, pois com este tipo de teste não é possível verificar o propósito de um sistema em sua totalidade (SOMMERVILLE, 2011). Em vista disso, os testes manuais podem ser eficazes na contribuição para a geração de novos casos de testes automatizados (BORTOLUCI, DUDUCHI, 2015).

#### 4.5 Pirâmide de automação de testes

A primeira aparição do conceito de pirâmide de testes foi em Rouchon (1990), porém não era relacionado à área de engenharia de software, mas sim uma abordagem para medir a qualidade na construção de estruturas compostas de aeronaves (Figura 1). A pirâmide em sua base apresenta os componentes fundamentais e conforme sobe ao topo da pirâmide, a complexidade aumenta, até chegar na combinação de sistema como um todo (FREEMAN, RADZIWIŁŁ, 2020).

Figura 1 - Pirâmide de testes de Rouchon (1990).

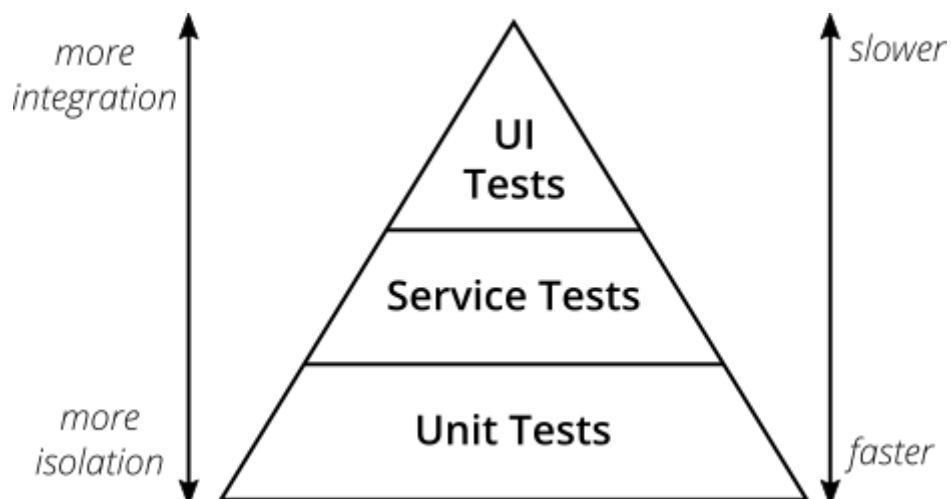


Fonte: Rouchon (1990).

Anos mais tarde, este padrão foi aplicado na engenharia de software. Cohn (2009) apresentou em seu livro *Succeeding with Agile* o conceito de pirâmide de testes automatizados (FREEMAN, RADZIWILL, 2020).

Testes automatizados são considerados custosos de escrever e geralmente são escritos muito tempo após a funcionalidade ser feita. Uma das dificuldades do time para não implementar os testes automatizados mais cedo, durante o processo de desenvolvimento, é porque são feitos em níveis inadequados. Para solucionar isto, uma estratégia de automação de testes em três níveis é apresentada, a pirâmide de automação de testes (COHN, 2009).

Figura 2 - Pirâmide de testes de Mike Cohn (2009) descrita por Vocke (2018).



Fonte: Vocke (2018).

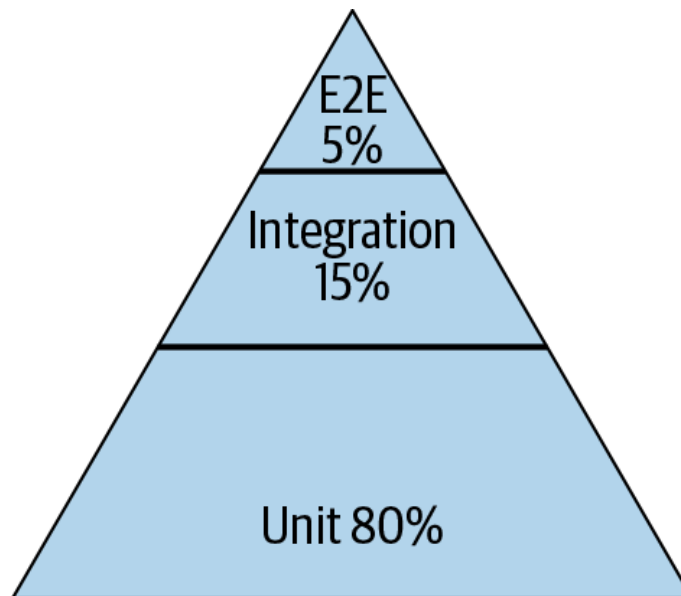
Como apresentado na Figura 2, a pirâmide demonstra de forma visual as camadas de cada tipo de testes e a quantidade que devem ser feitos. As camadas que foram idealizadas originalmente por Mike Cohn, são as de testes unitários, testes de serviços e testes de interfaces de usuário (VOCKE, 2018).

Nos dias de hoje alguns conceitos da pirâmide de testes são antigos e não são tão fáceis de entender, porém os fundamentos devem permanecer. Do ponto de vista moderno, dois princípios da pirâmide de testes podem ser evidenciados: a granularidade de testes na aplicação e quanto maior são os testes de alto nível, menor deverá ser a quantidade. A partir disso, não se deve seguir estritamente os nomes estipulados por Mike Cohn em cada camada da pirâmide, por exemplo, teste de serviço pode ser difícil de entender e testes de interface de

usuário, com *frameworks* atuais com possibilidade de desenvolvimento de aplicações de página única, este tipo de teste não necessariamente está no topo da pirâmide, mas pode ser realizado com testes unitários (VOCKE, 2018).

Assim como Vocke (2018) observa, a pirâmide de testes de Mike Cohn pode estar defasada e o nome de algumas camadas podem não fazer sentido. Desse modo, o Google fez sua adaptação, mantendo a granularidade dos testes e alterando o nome das camadas para nomes mais atuais que façam mais sentido, são eles: testes unitários, testes de integração e testes de *end-to-end*. Esta adaptação pode ser conferida na Figura 3.

Figura 3 - Pirâmide de testes adaptada pelo Google, contendo as porcentagens cada de teste por camada (podendo variar de equipe para equipe) (WRIGHT, WINTERS, MANSHREK, 2020).



Fonte: Software Engineering at Google (2020).

Os testes unitários são focalizados nos componentes e nos módulos, ou seja, na menor unidade de um projeto de software. São responsáveis por descobrir erros dentro dos limites do módulo e de um componente. Sendo assim, um bom projeto prevê as condições dos erros, criando então tratativas e soluções para eles. A escrita deste tipo de teste pode ser durante o período de desenvolvimento ou após, sempre havendo um caso de teste e um conjunto de respostas esperadas (PRESSMAN, 2011).

Testes de integração enfatizam a integração de módulos individuais e interfaces, ou seja, estes testes realizam a validação de quando são integrados para formar uma aplicação

funcional. Cerca de 40% dos erros em um software podem ser investigados no decorrer da integração dos componentes (LEUNG, WHITE, 1990). Há duas abordagens para se realizar os testes de integração, são elas de forma não incremental e incremental. Na abordagem não incremental, o sistema é testado como um todo, gerando muitos erros, e sua correção se torna complicada devido à falta de isolamento do que está sendo tratado, além disso, uma vez corrigidos podem gerar novas falhas. Por outro lado, a abordagem incremental é oposta, pois as integrações são feitas em pequenos incrementos, tornando assim, maior o nível de cobertura de testes e facilita a correção de erros (PRESSMAN, 2011).

Testes *end-to-end* ou E2E são testes realizados diante da perspectiva do usuário, sendo um tipo de validação parecida com a do teste de integração, porém não foca tanto nas particularidades dos subconjuntos do sistema. Com isso, para que os testes E2E ocorram é ideal que os testes de integração já estejam feitos, devido aos muitos níveis de integração que um teste E2E necessita (TSAI, et al., 2001).

#### **4.6 Marketplace**

Marketplaces são plataformas com múltiplos lados ou um mercado de dois lados, sendo esses lados, o comprador e o vendedor que por meio desta plataforma, podem realizar interações comerciais de venda entre eles (HAGIU, WRIGHT, 2015).

Os marketplaces estão historicamente envolvidos com a possibilidade de realizar encontros de vendedores e compradores em mesmo lugar com objetivo de comunicar os interesses de compra e venda de ambos. Porém, com o avanço da tecnologia esse ponto de encontro entre vendedores e compradores mudou para o ambiente virtual da internet. Sendo assim, a principal característica dos e-marketplaces é a de unir múltiplos compradores e vendedores em uma única plataforma (GRIEGER, 2003).

Um e-marketplace ou marketplace eletrônicos são sistemas que permitem a participação de compradores e vendedores na troca de informação sobre produtos e preços ofertados. Este tipo de mercado está causando um grande impacto no mundo dos negócios, tanto para os consumidores quanto para as empresa, sendo visto que, há dois segmentos principais utilizados em um e-marketplace, sendo eles, do tipo *business-to-business* (B2B) ou *business-to-consumer* (B2C). O tipo de marketplace B2B é a comercialização entre empresas, já o B2C é a negociação de empresas direto com o consumidor final (STANDING, et al., 2010).

Segundo Grieger (2003), a internet possibilitou uma ampla possibilidade de segmentos de e-marketplace, sendo eles demonstrados na Tabela 1.

Tabela 1 - Tipos de interações em um *marketplace*.

	Governo ( <i>Government</i> )	Negócio ( <i>Business</i> )	Consumidor ( <i>Consumer</i> )
Governo ( <i>Government</i> )	<b>G2G</b> Ex.: Coordenação	<b>G2B</b> Ex.: Informação	<b>C2G</b> Ex.: Informação
Negócio ( <i>Business</i> )	<b>B2G</b> Ex.: e-procurement	<b>B2B</b> Ex.: e-commerce, e-markets	<b>B2C</b> Ex.: e-commerce
Consumidor ( <i>Consumer</i> )	<b>C2G</b> Ex.: Conformidade fiscal	<b>C2B</b> Ex.: Preço, Comparação, e-markets	<b>C2C</b> Ex.: e-markets

Fonte: Adaptado de Coppel, 2000

## 5 METODOLOGIA

A metodologia utilizada segue três etapas principais, sendo elas, a análise comparativa, o desenvolvimento da aplicação e a implementação da pirâmide de testes automatizados. Cada etapa está descrita nas subseções a seguir.

### 5.1 Análise comparativa

Tabela 2 - Análise comparativa entre marketplaces.

Características/Componentes	Marketplaces				
	Amazon	MagazineLuiza	Americanas	Mercado Livre	Shopee
Header	✓	✓	✓	✓	✓
Barra de busca	✓	✓	✓	✓	✓
Filtros de busca por categorias	✓	✓	✓	✓	✓
Componente de notificações	✗	✗	✗	✓	✓
Carrinho de compras	✓	✓	✓	✓	✓
Produto: Imagem, Título, Preço e Avaliação	✓	✗	✓	✓	✓
Favoritar produto	✗	✓	✓	✓	✓
Quantidade de produtos na Grid	4	4	4	3	5
Tela de detalhes do produto	✓	✓	✓	✓	✓
Imagem à esquerda e detalhes à direita	✓	✓	✓	✓	✓
Opção para adicionar ao carrinho	✓	✓	✗	✓	✓
Opção para comprar direto	✗	✗	✓	✓	✓
Pagamento: Cartão de Crédito	✓	✓	✓	✓	✓
Pagamento: Cartão de Débito	✗	✓	✗	✓	✗
Pagamento: Boleto	✓	✓	✓	✓	✓
Pagamento: Pix	✗	✓	✓	✓	✗

Fonte: Elaborado pelo autor (2021).

Para maior entendimento da aplicação de suporte desenvolvida para a implementação da pirâmide de testes automatizados, foi realizada uma análise comparativa entre reconhecidos *marketplaces* (Tabela 2), sendo eles, Amazon, MagazineLuiza, Americanas, Mercado Livre e Shopee. Nesta análise, foi considerado fatores característicos e componentes similares entre as aplicações comparadas. Ademais, uma análise sobre regras de negócio, fluxos dentro da aplicação e comportamentos foram realizadas.

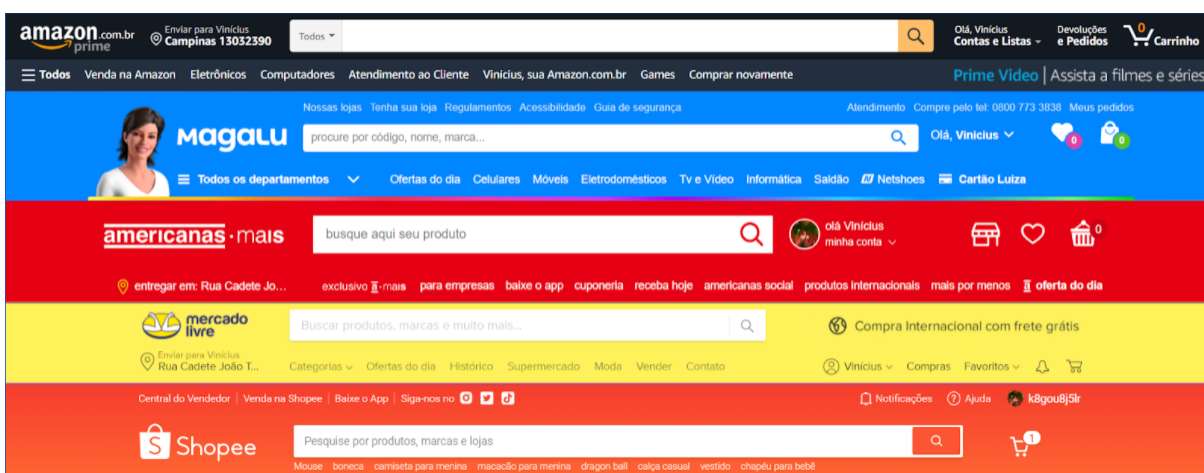
A partir disso, foi traçado um nível de prioridade entre as características abordadas. A prioridade definida é relativa à quantidade em que a característica aparece nas diferentes aplicações abordadas. Os atributos com maior prioridade comporam a aplicação, isto significa que, o processo de desenvolvimento foi baseado nesta triagem, com a tentativa de abordar de forma mais concreta a implementação de testes automatizados em diferentes aplicações com a mesma temática.

### 5.1.1 Características e componentes

Como resultado da análise comparativa dos *marketplaces*, pode-se notar um padrão desse tipo de aplicação, para isso será desenvolvido adiante o detalhamento de alguns dos atributos compostos na Tabela 2.

Ao acessar as aplicações analisadas pode-se notar um padrão na estrutura do site, em que neles sempre há a presença de um componente Header, com uma barra de busca e filtros para categorias específicas. Alguns contêm um ícone de notificações para o usuário ser alertado de novas ações realizadas em relação ao seu pedido. Esses padrões podem ser conferidos na Figura 4.

Figura 4 - Componente de Header das aplicações analisadas.








Fonte: Elaborado pelo autor (2021).



Além disso, nota-se também um padrão na estruturação do componente header, sendo o lado esquerdo sempre composto pela logomarca da aplicação, no meio a barra de busca e na direita o carrinho de compras. O componente carrinho de compras tem algumas variações de nomes entre as aplicações, por exemplo, no site Magazine Luiza é denominado como “Sacola de compras”, já na Americanas, o componente tem o nome de “Cesta de compras”.

Após a busca de um item desejado, os itens encontrados com a busca são exibidos em um formato de *grid* e com componentes semelhantes para pré-visualização do produto. O componente para a pré-visualização do produto geralmente é composto pela imagem, título, preço e avaliação do produto. Também há uma padronização deste componente, em que a imagem aparece ao topo, logo em seguida pelo título ou preço do produto. Os componentes de produto de cada marketplace são exibidos na Tabela 3.

Tabela 3 - Componente do produto das aplicações analisadas.

Amazon	MagazineLuiza	Americanas	Mercado Livre	Shopee
 <p>Mouse com fio USB Logitech M90 - Cinza</p> <p>★★★★★ ~ 3.582</p> <p>R\$24,90</p> <p>Exclusivo Prime: Cupom de R\$ 15,00 off no app</p> <p>prime Frete GRÁTIS: receba até Sexta-feira, 18 de jun</p>	 <p>Mouse Multilaser Sem Fio 2.4 Ghz 1200 Dpi Usb Preto Mo251</p> <p>R\$ 34,90</p>	 <p>indica</p> <p>mouse óptico wm126 preto wireless - dell</p> <p>★★★★★ 334 avaliações</p> <p>R\$99,99</p> <p><b>R\$ 89,99</b></p> <p>em 1x no cartão de crédito</p>	 <p>4 cores</p> <p>R\$33,92 10% OFF em 6x R\$ 6,25</p> <p>10% OFF</p> <p>Chegará hoje! FULL</p> <p>Mouse Multilaser Emborrachado Preto C Fio Usb 1200 Dpi Mo222</p>	 <p>39% OFF</p> <p>Frete GRÁTIS</p> <p>Mouse Sem Fio 2.4G Recarregável De Carregamen...</p> <p>Leve 5 e aproveite... R\$3 off</p> <p>R\$50,00 <b>R\$30,50</b></p> <p>★★★★★ 197 vendidos</p> <p>China Continental</p>

Fonte: Elaborado pelo autor (2021)

Em cada aplicação, a *grid* varia a quantidade de produtos por linha. Nos *marketplaces* Amazon, MagazineLuiza e Americanas, são exibidos quatro itens por linha, no Mercado Livre são três e no Shopee são cinco. Como a maioria das aplicações utilizam a disposição de quatro itens por linha, esse formato foi utilizado na aplicação.

Quando clicado em um desses produtos da *grid*, o usuário é redirecionado para a página de detalhes do produto, em que, há imagens do produto do lado esquerdo e ao lado direito, as informações. Nesta página, há também opções de adicionar o item ao carrinho de compras ou realizar a compra direta. Porém, a opção de adicionar ao carrinho e depois finalizar a compra é mais recorrente, isso foi levado em conta na construção da aplicação.

Por fim, outro ponto analisado foi os tipos de pagamento disponibilizados em cada *marketplace*, sendo predominantemente aceitos os pagamentos em cartão de crédito e boleto bancário. Os menos aceitos são cartões de débito e Pix.

## 5.2 Ferramentas

A construção da aplicação web foi com framework ReactJS utilizando a linguagem Typescript. Como apenas o *front-end* da aplicação foi desenvolvido, para simular os serviços e realizar os mocks de dados (dados falsos que simulam o comportamento real da aplicação) que serão visualizados no *marketplace*, foi utilizado o framework MirageJS.

Na implementação dos testes unitários, foi utilizado o framework Jest. Este framework atualmente é mantido pelo Facebook, e é projetado para garantir correções de qualquer código com base em Javascript, sendo um de suas principais vantagens, a alta velocidade de execução, linguagem simples e amplos recursos. Em conjunto com o Jest, foi utilizado o utilitário Enzyme, que foi desenvolvido inicialmente pela empresa Airbnb e depois foi destinado a uma outra empresa independente dedicada a continuar essa biblioteca. O utilitário Enzyme facilita os testes em aplicações React, pois é possível manipular e transferir componentes da aplicação para os arquivos de teste.

Para realização dos testes integrados e *end-to-end*, foi utilizado o framework Cypress. Este framework vem ganhando popularidade e sendo utilizado por empresas como Walt Disney Studios, PayPal, DHL e Johnson & Johnson. Segundo documentação do Cypress, o framework foi feito para validar tudo que é executado em um browser, possibilitando a realização de testes unitários, de integração e *end-to-end*.

Tabela 4 - Ferramentas instaladas para a realização da aplicação e dos testes.

Ferramenta	Versão
React (REACT, 2021)	17.0.2
TypeScript (TYPESCRIPT, 2021)	4.1.2
MirageJS (MIRAGEJS, 2021)	0.1.41
Jest (JEST, 2021)	26.0.15
Enzyme (ENZYME, 2021)	3.11.0
Cypress (CYPRESS, 2021)	8.5.0

Fonte: Elaborado pelo autor (2021).

### 5.3 Desenvolvimento da aplicação

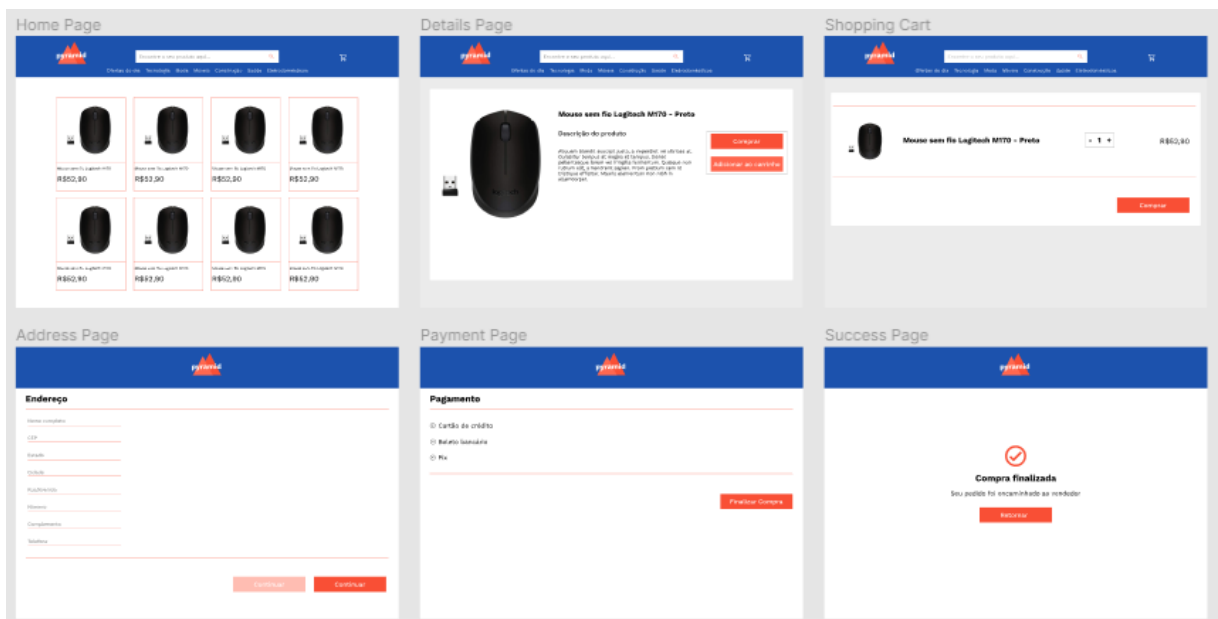
A aplicação desenvolvida é do tipo *marketplace* para plataforma web, sendo focalizada na parte do *front-end* com a utilização do framework ReactJS. O *marketplace* é voltado apenas para o lado do comprador, sendo enfatizado as principais características de acesso por parte deste tipo de usuário.

Esta etapa está totalmente atrelada à seção anterior de análise comparativa entre as plataformas, pois os levantamentos realizados fundamentaram as fases do desenvolvimento que são: a prototipação, as regras de negócio e, por fim, a própria aplicação.

O protótipo é uma versão inicial do software a ser desenvolvido, que auxilia o descobrimento de características de um software. Além disso, o processo de prototipação pode ajudar em outras áreas na construção de uma aplicação, como a validação dos requisitos, encontrar novas ideias, soluções específicas, e pontos fortes e fracos da aplicação (SOMMERVILLE, 2011).

A prototipação do *marketplace* foi feita no editor gráfico de prototipação web Figma (FIGMA, 2021), com objetivo de ter uma prévia de alta fidelidade da aplicação, sendo possível, ver a interface com os componentes requisitados, fluxos de navegação e resultados esperados.

Figura 5 - Prototipação da aplicação no Figma.



Fonte: Elaborado pelo autor (2021).

Por último, com a prototipação e os requisitos realizados (Figura 5), a etapa de desenvolvimento foi feita com o fundamento das etapas anteriores, sendo assim, cobrindo o que foi especificado.

### 5.3.2 Estrutura da aplicação

A estrutura da aplicação foi gerada inicialmente com o comando para a criação de aplicações React com a linguagem Typescript. Após isso, foram realizadas algumas alterações dessa criação inicial, adicionando pastas responsáveis para lidar com os componentes, páginas e utilitários presentes na aplicação. A estrutura de pastas da aplicação pode ser observada na Tabela 5.

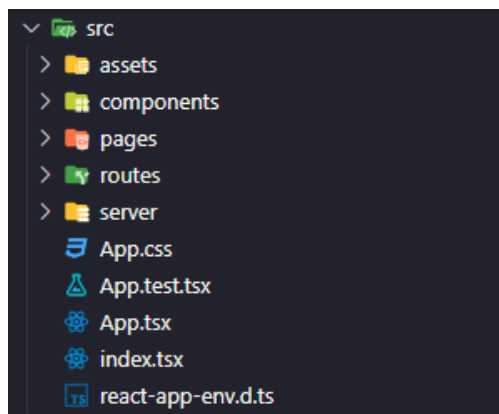
Tabela 5 - Estruturas de pastas da aplicação.

Pastas	Descrição
config	Configurações globais da aplicação
node_modules	Armazenamento de módulos requeridos para o funcionamento da aplicação
public	Arquivos públicos para a aplicação
src	Armazena o código fonte da aplicação, arquivos relacionados aos componentes e as páginas da aplicação
tests	Armazena os testes de integração e testes <i>end-to-end</i>

Fonte: Elaborado pelo autor (2021).

Dentro da pasta *src* contém a estruturação dos componentes e páginas da aplicação, havendo pastas específicas para organização desses elementos, sendo respectivamente as pastas *components* e *pages*. Também na pasta, estão presentes outros elementos importantes para o funcionamento da aplicação como mídias, rotas, e o servidor, como pode ser observado na Figura 6.

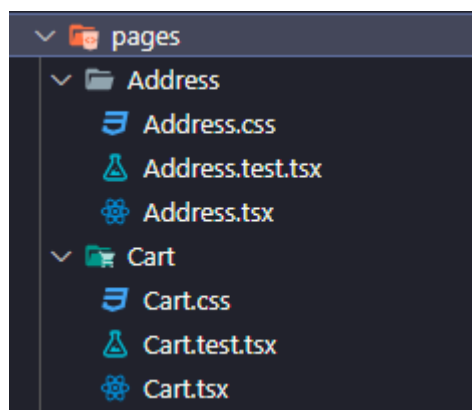
Figura 6 - Organização de pastas da aplicação.



Fonte: Elaborado pelo autor (2021).

Ao olhar de maneira específica aos componentes e páginas da aplicação, a estrutura segue um formato padronizado (Figura 7), havendo de maneira sistêmica a presença de um arquivo destinado ao estilo (*.css*), um relacionado aos testes unitários (*.test*) e um para o próprio elemento (*.tsx*).

Figura 7 - Padrão da construção de páginas e componentes.



Fonte: Elaborado pelo autor (2021).

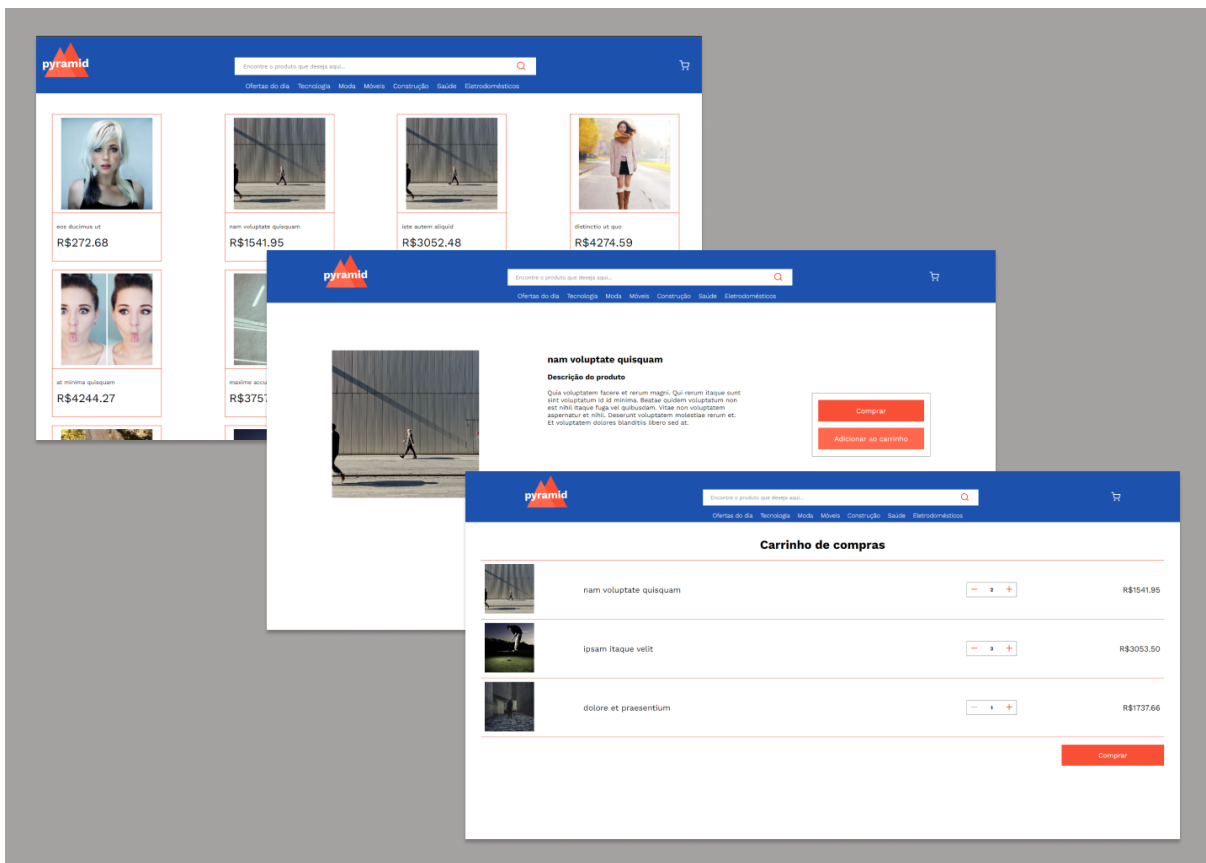
### 5.3.3 Etapas de desenvolvimento

Após a geração da estrutura inicial e feita as devidas alterações, as etapas de desenvolvimento seguiram primeiro com a das páginas e das rotas. As páginas foram construídas com base na prototipação realizada no framework Figma, seguindo assim proporções e estilizações previamente definidas.

Algumas páginas necessitavam de componentes específicos e alguns poderiam ser utilizados mais de uma vez em diferentes páginas, para isso, os componentes foram criados

em uma estrutura externa ao das páginas. Com isso, os componentes são centralizados e possíveis alterações se aplicam a todas páginas em que os componentes estão sendo utilizados, evitando duplicidade de código e mantendo o princípio de codificação limpa.

Figura 8 - Telas da aplicação após o desenvolvimento.



Fonte: Elaborado pelo autor (2021).

Com a parte visual realizada (Figura 8), foi então implementado os testes automatizados. De início foi implementado os testes unitários, logo após foram os testes de integração e por fim os testes *end-to-end*. Na próxima seção a implementação dos testes será explicada de forma mais detalhada.

## 5.4 Implementação da pirâmide de testes automatizados

A implementação da pirâmide de testes automatizados foi realizada com os fundamentos do livro *Succeeding with Agile* de Mike Cohn (2009), porém as camadas da pirâmide foram adaptadas conforme a abordagem do livro *Software Engineering at Google* (2020). Portanto, as camadas de testes automatizados implementadas, sendo de baixo para cima da pirâmide, são as de teste unitário, teste de integração e teste *end-to-end*.

### 5.4.1 Testes unitários

Os testes unitários são a base da pirâmide, ou seja, os testes predominantes em quantidade na aplicação e mais isolados. Estes testes foram realizados em conjunto com a fase de desenvolvimento das funcionalidades do sistema, validando assim a implementação de funções, métodos e lógicas no código.

Os testes unitários foram implementados para todos os arquivos que continham uma funcionalidade e um possível resultado esperado. Diferente dos demais testes, os unitários foram implementados no mesmo local dos arquivos relacionados às páginas e componentes com a nomenclatura *.test* no arquivo de teste. Além disso, foi utilizado o framework Jest em conjunto com Enzyme para a validação.

Para exemplificar as validações realizadas nesta etapa, a seguir será demonstrado a estrutura dos testes, mais detalhes sobre a codificação e tipos de validações realizadas. O componente de análise será a Barra de Pesquisa (*Search bar*) presente na *Header* da aplicação. Em vista de facilitar a explicação, será apresentado a Figura 9 que retrata o layout do componente, ao invés do código HTML e em seguida algumas validações realizadas.

Figura 9 - Componente de barra de busca do *marketplace*.



Fonte: Elaborado pelo autor (2021).

Em uma aplicação *front-end*, os testes unitários podem ser realizados de forma a aumentar a imutabilidade do componente construído, implementando então validações relacionadas a quantidade de elementos esperados, comportamentos e estados do componente.

Figura 10 - Teste unitário para a validação do componente de header.

```
it("should contains a search input", () => {
  expect(wrapper.find("input").length).toBe(1);
});

it("should contains a search button", () => {
  expect(wrapper.find("button").length).toBe(1);
});
```

Fonte: Elaborado pelo autor (2021).

A partir da Figura 10, é possível visualizar a estrutura dos testes unitários com o Jest. O framework possibilita a escrita dos testes em um formato legível e podem ser interpretados como frases. Em vista disso, as validações realizadas nos testes em questão são:

- Validar se a barra de busca contém apenas um input: Este teste avalia que será esperado apenas um input na barra de busca para o usuário inserir a busca.
- Validar se a barra de busca contém apenas um botão: Este teste avalia que será esperado apenas um botão na barra de busca para o usuário clicar para efetivar a busca desejada.

Além de validações da quantidade de elementos é possível simular ações e comportamentos nos componentes desejados, como cliques e inserção de texto. Por exemplo, ao validar a busca de um item no *marketplace* no componente de barra de busca, o usuário necessita inserir um texto no elemento de input, em seguida apertar o botão de busca, e então uma API responsável pela busca desse elemento será chamada. A exemplificação deste tipo de validação está presente na imagem abaixo.

Figura 11 - Teste unitário para simular ação de click no botão de pesquisa.

```
it("should click on search and call handleSearch", () => {
  axios.get = jest.fn().mockResolvedValue({});
  wrapper.find("input").simulate("change", { target: { value: "foo" } });
  wrapper.find("button").simulate("click");
  expect(axios.get).toHaveBeenCalledTimes(1);
  expect(axios.get).toHaveBeenCalledWith("/api/products/search?q=foo");
});
```

Fonte: Elaborado pelo autor (2021).

Na figura 11, foi realizado os seguintes passos:

- O mock do método GET presente na biblioteca Axios, a qual é responsável pelas chamadas das APIs existentes.



- Simulação da escrita no componente de input com o valor “foo”.
- Simulação do clique no botão de pesquisar, nesta ação a função *handleSearch* é chamada.

A partir destes passos, o resultado do teste espera que o método GET, que está contido dentro da função *handleSearch*, seja chamado uma vez e que tenha sido chamado com o endpoint correto (“/api/products/search?q=foo”).

Dessa maneira, os testes unitários seguem esse teor validando a maioria dos arquivos da aplicação de forma isolada e com rápida execução. É de boa prática na etapa de desenvolvimento, além de uma possível alteração no código da aplicação, sempre rodar os testes de unidade para validar se uma parte alterada não está sendo coberta pelos testes, e assim, incluir esta parte no arquivo de teste.

#### **5.4.2 Testes de integração**

Na próxima camada implementada, são os testes de integração, voltados para validar as funcionalidades de forma integrada, desta forma, havendo uma validação melhor do funcionamento em conjunto das unidades testadas na camada anterior. Nesta etapa, existem menos testes do que os unitários, devido a esse tipo de validação se encaixar em apenas em alguns contextos e também para os testes não serem frágeis e quebram com facilidade em uma possível futura alteração.

Nesta etapa, o importante é validar se componentes construídos separadamente funcionam de forma esperada quando estão em conjunto em uma funcionalidade específica. Como na seção anterior, será apresentado a seguir a exemplificação de testes de integração implementados. De maneira a expor o conteúdo de forma didática, será disponibilizado imagens do layout e do código, detalhando a implementação, a validação e características do framework utilizado.

O teste de integração utilizado para ser detalhado nesta etapa será a validação da adição do mesmo produto no carrinho de compras. Com este teste, é possível verificar diversos componentes da aplicação do tipo *marketplace*, por exemplo, validar a existência de produtos, ao clicar em um produto ir para página de detalhes, o botão de adicionar ao carrinho, a adição do item no carrinho e se adição está sendo feita de maneira esperada.

Figura 12 - Validação da adição do mesmo item ao carrinho de compras.

```
it("Add the same item in the Shopping Cart", () => {
  cy.get(":nth-child(1) > .product-container").click();
  cy.wait(400);
  cy.get('[href="/shopping-cart"] > .button-pyramid').click();
  cy.get("a > img").click();
  cy.get(":nth-child(1) > .product-container").click();
  cy.wait(400);
  cy.get('[href="/shopping-cart"] > .button-pyramid').click();
  cy.get(".product-cart").should("have.length", 1);
});
```

Fonte: Elaborado pelo autor (2021).

Na Figura 12, é possível visualizar a estrutura de um teste escrito com framework Cypress. Como na utilização do Jest, os testes com Cypress são legíveis e intuitivos, dessa forma, o framework contém métodos específicos, que por meio de seletores CSS é possível localizar um elemento da aplicação e realizar alguma ação ou validação. Com isso, na validação em questão há a adição de um mesmo produto no carrinho de compras, sendo assim, pode ser testado a integração dos componentes e se a adição ao carrinho de compras está sendo feita corretamente. Quando adicionado produtos diferentes, é esperado que contenha dois itens no carrinho, porém, quando adicionado o mesmo produto mais de uma vez, o carrinho deve conter apenas um produto.

Em vista de detalhar mais o que está sendo feito na imagem, as ações realizadas são:

1. Clicar em um produto na Home Page;
2. Adicionar este produto no carrinho de compras;
3. Voltar para a Home Page;
4. Clicar no mesmo produto anteriormente;
5. Adicionar o mesmo produto no carrinho de compras;
6. Acessar o carrinho de compras e checar se há apenas um produto na lista.

Uma vez que estes testes estão escritos, poderão ser executados de forma automatizada, veloz e repetidamente. Desta maneira, os testes de integração conseguem capturar em qualquer possível desenvolvimento ou alteração na aplicação, impactos na comunicação das páginas e componentes que divergem de seus comportamentos esperados.

### 5.4.3 Testes *end-to-end*

Na última camada, foram realizados os testes *end-to-end* (E2E), em que nesta etapa ocorreram as validações das jornadas principais, de ponta-a-ponta, que o usuário poderá realizar na aplicação. A menor quantidade de testes a serem desenvolvidos nesta camada, se dá pelo fato de que este tipo é o mais custoso para se automatizar, pois requer a interação com a interface e geralmente levam um maior tempo para serem executados. Portanto, será escrito quando a aplicação obtiver um nível de maturidade suficiente para receber este tipo de verificação.

Para a execução dos testes desta etapa, será utilizado o mesmo framework utilizado para os testes de integração, o Cypress. Por mais que esta última camada de teste seja custosa para automatizar e mais lenta para execução, o Cypress torna essa experiência diferente, pois permite que a execução destes cenários sejam velozes e de fácil escrita.

No contexto da aplicação do tipo *marketplace* é importante que nos testes *end-to-end* sejam cobertos os principais fluxos realizados na aplicação, desde a primeira ação até a última. Levando em conta a temática da aplicação, os fluxos principais automatizados foram as que permitem a validação de toda a experiência, ou seja, passar pela tela inicial, acessar os detalhes de um produto, comprar diretamente ou adicionar ao carrinho, inserir dados do usuário, escolher pagamento, e por fim a finalizar a compra.

Assim como nas seções anteriores, será feito uma análise detalhada da implementação de um teste *end-to-end* específico. Para esta seção, será analisado o fluxo de comprar um produto adicionando ao carrinho de compras.

Figura 13 - Validação do fluxo de comprar um produto adicionando pelo carrinho.

```

it("Buy a product adding in the shopping cart", () => {
  cy.visit("/");
  cy.get(":nth-child(1) > .product-container").click();
  cy.url().should("include", "/product-details");
  cy.wait(400);
  cy.get('[href="/shopping-cart"] > .button-pyramid').click();
  cy.checkUrl("/shopping-cart");
  cy.get(".button-pyramid").click();
  cy.checkUrl("/address");
  cy.fillAddressForm();
  cy.get(".button-pyramid").click();
  cy.checkUrl("/payment");
  cy.get("#creditCard").click();
  cy.get(".button-pyramid").click();
  cy.checkUrl("/success");
  cy.get(".button-pyramid").click();
  cy.checkUrl("/");
});

```

Fonte: Elaborado pelo autor (2021).

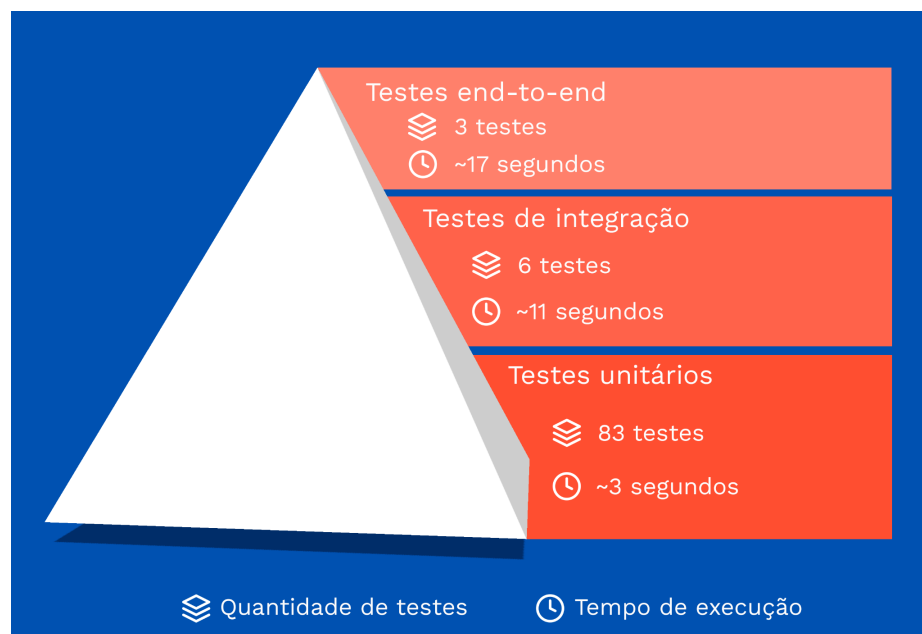
O script apresentado na Figura 13, apresenta o cenário de um teste fim-a-fim realizado na aplicação, nele é garantido que o fluxo de comprar um produto adicionando ele ao carrinho seja garantido. Neste script também é validado se as rotas de redirecionamento estão corretas, desde a página inicial até a ida da página de sucesso, para novamente retornar a página inicial e assim encerrar o fluxo.

O intuito deste tipo de teste não é realizar uma validação minuciosa de componentes e páginas da aplicação, pois isso já foi realizado nas etapas anteriores de testes unitários e testes de integração. Para esta etapa a automação é responsável por validar se fluxos cruciais realizados pelos usuários estão de acordo com o esperado, e também para verificar se o objetivo da aplicação está sendo atingido.

## 6 RESULTADOS

Os testes unitários, localizados na base da pirâmide e caracterizados por conterem os maiores números de testes e serem executados rapidamente, possuíram 83 testes ao total e foram executados em uma média de 3 segundos. Em seguida, a camada dos testes de integração, houveram o total de 6 testes e foram executados em uma média de 11 segundos. Por último, os testes *end-to-end* tiveram 3 testes e ocorreram em um tempo de em média 17 segundos. Estes dados podem ser verificados de forma visual na Figura 14.

Figura 14 - Resultados da implementação da pirâmide de testes.



Fonte: Elaborado pelo autor (2021).

Os frameworks utilizados para realização dos testes automatizados possuem relatórios de cobertura e alertam inconsistências ao encontrarem algo divergente do que está sendo esperado pela validação. Por exemplo, na etapa de teste unitário, quando uma validação espera um comportamento de um componente e isto não ocorre como previsto, o teste falhará e aparecerá uma mensagem detalhando o erro encontrado.

Figura 15 - Falha no terminal de um teste de unidade.

```
FAIL src/components/QuantityInput/QuantityInput.test.tsx
  ● Quantity Input Component > should be possible to increase a value

    expect(received).toEqual(expected) // deep equality

    Expected: 2
    Received: 3

    27 |     it("should be possible to increase a value", () => {
    28 |       wrapper.find(".quantity-input__increase-btn").simulate("click");
    > 29 |       expect(wrapper.find("input").props().value).toEqual(2);
        |                                                     ^
    30 |     });
    31 |
    32 |     it("should be possible to decrease a value", () => {

    at Object.<anonymous> (src/components/QuantityInput/QuantityInput.test.tsx:29:49)
```

Fonte: Elaborado pelo autor (2021).

Na Figura 15, há um caso em que o cenário de teste unitário falhou em sua validação do componente de manipulação da quantidade de um produto a ser comprado, já que em vez de adicionar uma quantidade por vez, está adicionando duas. Então, a falha relata o valor esperado e o recebido, sendo respectivamente, o valor 2 e 3. Desta maneira, o desenvolvedor consegue analisar o que está acontecendo de errado na aplicação e realizar a correção da funcionalidade antes de subir o código para produção.

Além de registros de falhas no terminal, o framework Jest de teste unitário permite a análise da cobertura do código validado. Este resultado de cobertura é obtido em diversos formatos, sendo pelo próprio terminal e até por uma página HTML. Na aplicação desenvolvida todos os arquivos passíveis de validação foram atingidos a cobertura de 100% (Figura 16).

Figura 16 - Relatório de cobertura dos testes unitários.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
src	100	100	100	100	
App.tsx	100	100	100	100	
src/assets	100	100	100	100	
constants.ts	100	100	100	100	
src/components/Categories	100	100	100	100	
Categories.tsx	100	100	100	100	
src/components/Header	100	100	100	100	
Header.tsx	100	100	100	100	
SimpleHeader.tsx	100	100	100	100	
src/components/Product	100	100	100	100	
Product.tsx	100	100	100	100	
src/components/ProductCart	100	100	100	100	
ProductCart.tsx	100	100	100	100	
src/components/QuantityInput	100	100	100	100	
QuantityInput.tsx	100	100	100	100	
src/components/SearchBar	100	100	100	100	
SearchBar.tsx	100	100	100	100	
src/pages/Address	100	100	100	100	
Address.tsx	100	100	100	100	
src/pages/Cart	100	100	100	100	
Cart.tsx	100	100	100	100	
src/pages/Home	100	100	100	100	
Home.tsx	100	100	100	100	
src/pages/Payment	100	100	100	100	
Payment.tsx	100	100	100	100	
src/pages/ProductDetails	100	100	100	100	
ProductDetails.tsx	100	100	100	100	
src/pages/SearchResult	100	100	100	100	
SearchResult.tsx	100	100	100	100	
src/pages/Success	100	100	100	100	
Success.tsx	100	100	100	100	
src/routes	100	100	100	100	
Routes.tsx	100	100	100	100	

Fonte: Elaborado pelo autor (2021).

A cobertura de código elevada compreende-se que todas as funcionalidades a nível de código estão sendo executadas. Porém, a cobertura não está relacionada à qualidade do teste, podem haver testes que cobrem o arquivo em sua totalidade e a validação pode não ser adequada, por isso, vale a atenção às boas práticas de testes e a utilização de frameworks para cooperar na qualidade deste tipo de validação. Além dessas observações, analisar a cobertura do código pode-se tornar um caminho para a realização dos testes, pois o framework permite visualizar os pontos específicos do código que não estão sendo validados, como demonstrado na Figura 17.

Figura 17 - Parte de código não coberta pelo teste de unidade.

```
const decreaseQuantity = (qty: number) => {
  if (qty > 1) {
    setQuantity(quantity - 1);
  }
};

const increaseQuantity = () => {
  setQuantity(quantity + 1);
};

return (
  <div className="quantity-input">
```

statement not covered

Fonte: Elaborado pelo autor (2021).

Além dos testes unitários, o framework responsável pelos testes de integração e *end-to-end* também alerta possíveis erros ocorridos no terminal e no próprio dashboard do Cypress. Neste dashboard é possível a visualização de qual etapa de validação não foi atingida como a esperada e assim como nas validações unitárias, é demonstrado o valor esperado e o valor recebido.

Para manter a periodicidade da execução dos testes automatizados, é comum que em todo processo de subida de versão ou de integração contínua, os testes automatizados rodem em uma *pipeline*. Se alguma falha for encontrada, esse processo deve alertar e impedir que a nova implementação realizada se junte ao código que já está em produção. Para realização dessas ações há diversas plataformas que são responsáveis por essas validações, como a Azure (Microsoft) e Jenkins.

Segundo Mike Cohn (2009), os times que acham custosos e trabalhosos o processo da automatização dos testes, é devido a não saberem as camadas que devem realizar essas validações. Porém, ao realizar os testes de forma automatizada nas camadas da pirâmide, esse esforço e trabalho não se tornou mais fácil, uma vez que, para realizar os scripts de automação e cenários de testes deve-se obter um conhecimento dos frameworks e dos tipos de validação.

Contudo, o custo para realizar as automações é proporcional ao nível de valor que é acrescentado a qualidade da aplicação, por isso, empresas que não implementam este processo, a abordagem da pirâmide pode servir de facilitação para traçar uma estratégia de qualidade de software e um plano que caiba nos prazos determinados de um projeto.

De modo geral, a suíte de testes idealizada pelo formato de pirâmide mostraram-se eficazes e trazem imutabilidade à aplicação, sendo assim, responsáveis pela garantia da cobertura da maioria das funcionalidades do *marketplace*. Com isso, o ato de se prevenir erros no momento do desenvolvimento se tornam mais perceptíveis e permitem que já nesta etapa o desenvolvedor consiga corrigir inconsistências de funcionalidades implementadas ou já existentes.



## 7 CONCLUSÃO

Ao analisar as aplicações de *marketplaces* já existentes, foi possível identificar que essas aplicações possuíam regras de negócio, jornadas e componentes semelhantes. Com isso, a análise comparativa foi realizada entre cinco aplicações reconhecidas no segmento de *marketplace*, sendo elas, Amazon, Americanas, Magalu, Mercado Livre e Shopee.

A partir da análise comparativa e levantamento de quais eram as características mais utilizadas entre as aplicações já existentes, foi realizado o desenvolvimento do *front-end* da aplicação. O desenvolvimento seguiu estritamente a análise realizada, com intuito de construir um software base para a implementação dos testes automatizados.

Com o desenvolvimento do *marketplace* e a implementação de todas as camadas da pirâmide de testes de forma automatizada, os resultados obtidos relacionados à quantidade de testes e ao tempo de execução foram como os esperados, tendo em vista as bibliografias.

Por fim, via funcionalidades providas dos frameworks de automação implementados, os dados de cobertura foram coletados. Com isso, foi possível realizar a verificação da efetividade da implementação da pirâmide de testes na qualidade de código da aplicação.

De maneira geral, uma das principais dificuldades encontradas na realização do trabalho foi a distinção dos tipos de testes presentes na pirâmide, com isso, para a implementação devem-se estar bem definidos os conceitos dos tipos de testes e o entendimento do escopo da aplicação. Em relação a parte de implementação das automações, o conhecimento das ferramentas e o domínio da linguagem de programação é essencial, para isso, as documentações auxiliam de maneira fundamental neste uso.

A partir do conteúdo apresentado, considera-se de forma geral que os objetivos deste trabalho foram alcançados. Além disso, esta monografia espera que a implementação de testes automatizados com a abordagem da pirâmide de testes seja acessível para estudantes, pesquisadores, desenvolvedores e analistas de qualidade.

Para trabalhos futuros, há algumas oportunidades de melhorias da aplicação base desenvolvida e a implementação de mais testes que poderiam complementar a pirâmide de testes, como os testes de componente, testes de mutação, e se a aplicação houvesse um *back-end*, testes de contrato. Além disso, seria possível empregar o vínculo do *marketplace* desenvolvido em softwares de entregas contínuas, que possibilitam a criação de pipelines responsáveis por executar as automações existentes na aplicação.

## REFERÊNCIAS

- BERNARDO, P.; KON, F. **A importância dos testes automatizados**. Engenharia de Software Magazine, 3:54–57, 2008.
- BORTOLUCI, Raquel ; DUDUCHI, M. . **Um estudo de caso do processo de testes automáticos e manuais de software no desenvolvimento ágil**. In: X WORKSHOP DE PÓS-GRADUAÇÃO E PESQUISA DO CENTRO PAULA SOUZA, 2015, São Paulo. Sistemas Produtivos e Desenvolvimento Profissional: Desafios e Perspectivas, 2015. p. 805-814.
- COHN, M. **Succeeding with Agile: Software Development Using Scrum**. Addison-Wesley Professional, 2009.
- CYPRESS. **Cypress**. 2021. Disponível em: <<https://www.cypress.io/>>. Acesso em: 20 nov. 2021
- DOS SANTOS SOARES, Michel. Metodologias ágeis extreme programming e scrum para o desenvolvimento de software. **Revista Eletrônica de Sistemas de Informação**, v. 3, n. 1, 2004.
- ENZYME. **Enzyme**. 2021. Disponível em: <<https://enzymejs.github.io/enzyme/>>. Acesso em: 20 nov. 2021
- FIGMA. **Figma**. 2021. Disponível em: <<https://www.figma.com/>>. Acesso em: 20 nov. 2021
- GARVIN, D.A.: "What does product quality really mean?". **Sloan Management Review**, p.25-43, Fall/1984.
- GRIEGER, Martin. Electronic marketplaces: A literature review and a call for supply chain management research. **European journal of operational research**, v. 144, n. 2, p. 280-294, 2003.
- HAGIU, Andrei; WRIGHT, Julian. Multi-sided platforms. **International Journal of Industrial Organization**, v. 43, p. 162-174, 2015.
- JEST. **Jest**. 2021. Disponível em: <<https://jestjs.io/>>. Acesso em: 20 nov. 2021
- LEUNG, Hareton KN; WHITE, Lee. A study of integration testing and software regression at the integration level. In: **Proceedings. Conference on Software Maintenance 1990**. IEEE, 1990. p. 290-301.
- MIRAGEJS. **MirageJS**. 2021. Disponível em: <<https://miragejs.com/>>. Acesso em: 20 nov. 2021
- PRESSMAN, R. S. **Engenharia de software: uma abordagem profissional (7a ed.)**. Porto Alegre: AMGH, 2011.

RADZIWILL, Nicole; FREEMAN, Graham. Reframing the Test Pyramid for Digitally Transformed Organizations. **arXiv preprint arXiv:2011.00655**, 2020.

REACT. **React**. 2021. Disponível em: <<https://pt-br.reactjs.org/>>. Acesso em: 20 nov. 2021

SALEEM, Rana Muhammad et al. Testing Automation in Agile Software Development. **International Journal of Innovation and Applied Studies**, v. 9, n. 2, p. 541, 2014.

SCHWABER, Ken; SUTHERLAND, Jeff. Guia do Scrum. **Scrumguides. Org**, v. 1, p. 21, 2013.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: PEARSON, v. I, 2011.

STANDING, Susan; STANDING, Craig; LOVE, Peter ED. A review of research on e-marketplaces 1997–2008. **Decision Support Systems**, v. 49, n. 1, p. 41-51, 2010.

TSAL, Wei-Tek et al. End-to-end integration testing design. In: **25th Annual International Computer Software and Applications Conference. COMPSAC 2001**. IEEE, 2001. p. 166-171.

TYPESCRIPT. **TypeScript**. 2021. Disponível em: <<https://www.typescriptlang.org/>>. Acesso em: 20 nov. 2021

VOCKE, H. The Practical Test Pyramid. **Martin Fowler**, 26 de fevereiro de 2018. Disponível em: <https://martinfowler.com/articles/practical-test-pyramid.html>. Acesso em: 15 mai. 2021.

WRIGHT, Hyrum; WINTERS, Titus Delafayette; MANSHRECK, Tom. Software Engineering at Google. 2020.

YACKEL, R. Test automation comes of age. **InfoWorld**, 2 de julho de 2018. Disponível em: <https://www.infoworld.com/article/3286529/test-automation-comes-of-age.html>. Acesso em: 22 mai. 2021.