UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Ana Gabriela de Abreu Campos

Proposta para Implantação de Automação de Testes de Software usando Behavior Driven Development (BDD) - Estudo de Caso

Uberlândia, Brasil

2024

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Ana Gabriela de Abreu Campos

Proposta para Implantação de Automação de Testes de Software usando Behavior Driven Development (BDD) - Estudo de Caso

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Orientador: Ronaldo Castro de Oliveira

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2024

Ana Gabriela de Abreu Campos

Proposta para Implantação de Automação de Testes de Software usando Behavior Driven Development (BDD) - Estudo de Caso

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Uberlândia, Brasil, 26 de abril de 2024:

Ronaldo Castro de Oliveira
Orientador

Renato Aparecido Pimentel da Silva

Rodrigo Sanches Miani

Uberlândia, Brasil
2024

Resumo

No cenário de desenvolvimento de *software*, a qualidade é um aspecto fundamental que impacta diretamente a satisfação do cliente, a reputação da empresa e o sucesso do produto no mercado. A garantia de que um *software* funcione conforme o esperado, sem *bugs* ou falhas, é crucial para proporcionar uma experiência positiva ao usuário e para assegurar a confiabilidade e eficácia das operações empresariais. No entanto, alcançar esse nível de qualidade pode ser um desafio, especialmente em projetos complexos que demandam uma abordagem meticulosa e abrangente para o teste de *software*. Na empresa-alvo deste estudo, uma problemática significativa surge em relação à dependência excessiva de testes manuais. Esta dependência pode resultar em atrasos consideráveis no ciclo de desenvolvimento e aumento dos risco de erros humanos. Embora os testes manuais sejam essenciais para garantir a qualidade do produto, eles apresentam limitações de cobertura e são propensos a inconsistências, especialmente em projetos como o *Pricing* (sistema de precificação de produtos) objeto deste estudo. Diante desse cenário, surge a necessidade de explorar abordagens inovadoras e eficazes para otimizar o processo de teste de *software*. Este trabalho propõe a implantação de um processo de testes automatizados de *software* utilizando *Behavior Driven Development* (BDD) como uma abordagem para superar os desafios identificados na empresa-alvo. O BDD oferece uma metodologia centrada no comportamento do usuário, promovendo uma compreensão comum entre as equipes de desenvolvimento e teste, e proporciona uma estrutura clara e organizada para a automação de testes. Espera-se que a implementação da proposta de solução proporcione benefícios como: redução no tempo de execução dos testes, consistência na identificação de *bugs* e ganho de tempo nos retestes, demonstrando a eficiência da execução automatizada e ressaltando as vantagens da automação, unida ao BDD, no processo de teste. Dessa forma, este trabalho estabelece uma base sólida para pesquisas futuras no campo de teste de *software*, destacando o potencial do BDD aliado à automação como uma estratégia eficaz para garantir a qualidade de *software*.

Palavras-chave: Teste de *software*, automação, processo, BDD, QA.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

BDD	Behavior Driven Development
TDD	Test Driven Development
QA	Quality Assurance
TI	Tecnologia da Informação
SM	Scrum Master
PO	Product Owner
E2E	End-to-End
UAT	Testes de Aceitação do Usuário
DDD	Design Orientado a Domínio
CI	Continuous Integration
CD	Continuous Delivery
CEO	Chief Executive Officer
DOI	DevOps Institute
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
UI	Interface do Usuário
ID	Identificador de Dispositivo
CSS	Cascading Style Sheet

Sumário

1 Introdução

A indústria de desenvolvimento de *software* está em constante evolução, impulsionada pela demanda por produtos de alta qualidade, segurança e desempenho. Desde pessoas, organizações, empresas e até mesmo o governo, todos dependem de diversos sistemas de informação para automatizar processos e se comunicarem efetivamente por meio de aplicações. Dispositivos e produtos técnicos, como os próprios automóveis, aviões, robôs, satélites, entre outros, também possuem o *software* inserido em suas construções como componente fundamental.

Devido a sua relevância no mundo, não é surpresa que exista uma área da Computação destinada a investigar os desafios e propor soluções que permitam desenvolver sistemas de *software* de forma produtiva e com qualidade. Essa área é chamada de Engenharia de *Software* (VALENTE, 2020).

Dentro das diversas áreas estudadas em Engenharia de *Software*, neste trabalho a área apresentada como objeto de estudo é a de Testes de *Software*. A execução de testes de *software* desempenha um papel fundamental na garantia da qualidade e confiabilidade de aplicações digitais. O teste de *software* é um processo utilizado para determinar se um produto atingiu o resultado esperado para o qual foi criado (PRESSMAN; MAXIM, 2021). Em outras palavras, é através dos testes que identificamos se o desenvolvimento do produto atendeu às expectativas do cliente, funcionando corretamente no ambiente para o qual foi projetado.

A importância de se ter um *software* testado e que ofereça garantia de qualidade é de grande valia. Quando um *software* falha, os potenciais problemas podem ser vastos e, em muitos casos, extremamente graves. Desde perdas financeiras significativas até situações que envolvem riscos à saúde e à vida, a confiabilidade do *software* é essencial para evitar consequências adversas. Portanto, um dos principais objetivos dos testes é revelar falhas em um produto, para que as causas dessas falhas sejam identificadas e possam ser corrigidas pela equipe de desenvolvimento antes da entrega final (CLAUDIO; NETO, 2017). No entanto, à medida que os projetos de *software* se tornam mais complexos e as equipes de desenvolvimento crescem, a necessidade de abordagens metodológicas eficazes e colaborativas para os testes se torna cada vez mais evidente.

Em 2002, Beck e Cheiran (2010) através do seu livro “TDD - Desenvolvimento Guiado por Testes”, em inglês *Test-Driven Development*, apresentou uma nova técnica para criar sistemas baseados em testes, visando garantir a qualidade e a funcionalidade do *software* durante este ciclo. Embora seja uma técnica aprovada e testada por profissionais, uma de suas características é a falta de foco no comportamento do sistema

(BECK; CHEIRAN, 2010). O TDD se concentra principalmente na implementação de funcionalidades do ponto de vista técnico, com testes que podem ser altamente vinculados à estrutura interna do código. Isso pode levar a uma falta de clareza sobre como as funcionalidades devem se comportar do ponto de vista do usuário final ou do negócio. Essa dificuldade na compreensão entre partes não técnicas, acaba se tornando um obstáculo em adquirir uma comunicação eficaz sobre os requisitos do *software* entre todas as partes interessadas.

Visando solucionar os problemas mencionados, North e Ltd (2020), em 2003, criou o BDD - *Behavior Driven Development* - em português: Desenvolvimento Orientado ao Comportamento. Sua intenção era fazer com que pessoas não técnicas também entendessem os testes descritos e as funcionalidades dos programas, fornecendo um vocabulário comum (linguagem ubíqua/natural) que abrange a divisão entre negócios e tecnologia (NORTH; LTD, 2020). Em outras palavras, trata-se de uma técnica de desenvolvimento ágil derivada do TDD que emergiu como uma proposta efetiva para abordar os desafios enfrentados na execução de testes de *software*, não apenas focando nos aspectos técnicos do teste, mas também coloca ênfase na comunicação e colaboração entre as partes interessadas, incluindo desenvolvedores, testadores e partes não técnicas, como os *stakeholders* do projeto.

Nesse contexto, é importante abordar também a Automação de Testes, por ser por meio dela que este estudo de caso é aplicado. A automação de testes refere-se à prática de supervisionar a execução de testes de *software* por meio de ferramentas ou *frameworks* especializados. Essa prática é realizada mediante o uso de *scripts* que reproduzem as ações manuais, testando as funcionalidades do *software* e verificando se os resultados correspondem aos esperados. A automação visa agregar valor ao processo de teste de *software*. No entanto, é importante ressaltar que os testes manuais não devem ser descartados, já que cada abordagem possui suas próprias vantagens. Cabe à equipe decidir quais tarefas devem ser automatizadas, uma vez que é desafiador alcançar uma cobertura completa de todas as funcionalidades de um sistema (LOURENçO, 2022).

Por fim, este trabalho de conclusão de curso adota a classificação de estudo de caso qualitativo. São levantadas características específicas de uma empresa, com foco na análise do seu processo de teste de *software* atual, que utiliza métodos manuais. O estudo visa propor um modelo de aplicação do BDD para essa empresa, visando aprimorar a eficiência do processo de teste por meio da automação. A abordagem qualitativa permite uma análise detalhada das práticas existentes, identificando desafios, oportunidades e lacunas no processo atual. As conclusões derivadas do levantamento inicial fornecem percepções fundamentais para a elaboração da solução proposta, que aspira promover uma transição eficaz para os testes automatizados, otimizando a qualidade e a eficiência do desenvolvimento de *software* na empresa em questão.

1.1 Objetivos Geral e Específicos

O presente trabalho expõe conceitos relacionados à técnica do BDD, e a sugere como uma abordagem de auxílio ao profissional testador, também conhecido como QA - *Quality Assurance* ou Garantia de Qualidade, durante o desenvolvimento de um *software*. Um estudo de caso é conduzido com o objetivo de formular uma proposta para a implementação de testes automatizados utilizando o BDD na empresa escolhida como objeto deste estudo.

O trabalho apresenta os princípios fundamentais do BDD, sua relação com o desenvolvimento ágil, a aplicação do BDD em processos de testes automatizados e como ele promove uma compreensão comum entre as equipes de desenvolvimento e teste, apresentando os impactos positivos ou negativos resultantes dos casos avaliados. Os seguintes objetivos específicos foram definidos para este estudo:

1. Apresentar os conceitos fundamentais do BDD, junto a outros referenciais teóricos importantes para o processo de *software*;
2. Investigar a aplicação prática do BDD em processos de testes automatizados, fornecendo *insights* sobre como essa metodologia pode ser implementada eficazmente;
3. Analisar detalhadamente o processo de desenvolvimento de *software* da empresa escolhida como foco deste estudo, com o intuito de realizar uma avaliação crítica deste procedimento;
4. Propor uma nova abordagem para a implementação de testes, integrando a automação com a metodologia BDD;
5. Apresentar os resultados obtidos com a implantação da proposta de solução, comparando esses resultados com o processo de testes de *software* manual.

1.2 Motivação

Diante do contexto acima, apresentar como a metodologia BDD pode superar os desafios mais comuns enfrentados na execução de Testes de *Software* tornou-se um importante e relevante tema de estudo e pesquisa. Não existe uma contagem exata de organizações que adotam o BDD, mas essa metodologia ganhou popularidade nos últimos anos, especialmente entre empresas que buscam melhorar a qualidade e a colaboração em seus projetos de *software*, em comparação com abordagens de testes mais tradicionais.

No enredo deste trabalho, é crucial analisar e contextualizar os trabalhos correlatos que abordaram questões similares ou relacionadas ao objeto de estudo proposto. A pesquisa realizada revelou uma variedade de estudos que exploraram aspectos relevantes

para o estudo de caso. Por exemplo, Albiero (2017) em seu trabalho de mestrado investigou a influência dos cenários escritos em BDD como uma abordagem de teste para aplicativos *Android*, em um contexto semelhante. Enquanto o trabalho de Paula (2019) focou em acompanhar a aplicação da técnica de BDD junto a histórias de usuário *(*User Stories), como uma alternativa de melhoria em um projeto ágil. Já a monografia de graduação de Lourenço (2022), focou na implementação da automação de testes em uma empresa, utilizando o *framework Cypress* e o padrão *Page Object*, concluindo que os testes automatizados são bem mais rápidos e possuem melhor cobertura, porém complementa mutuamente os testes manuais. Além disso, o artigo de Carvalho e Marques (2019) propõe a utilização da técnica BDD para otimizar a escrita e automação de testes no *framework Scrum*, utilizando a linguagem natural e ubíqua.

Ao considerar esses trabalhos correlatos e muitos outros como uma motivação para o desenvolvimento deste estudo, o objetivo é identificar lacunas de pesquisa e construir uma base sólida para análise, aprofundando o entendimento do tópico em questão e contribuindo para o avanço do conhecimento nessa área específica. E, mediante um estudo de caso, é possível analisar criticamente o processo de desenvolvimento de *software* existente, identificando oportunidades de melhoria e propondo uma solução que integre a automação com a metodologia BDD. Essa proposta visa contribuir para aprimorar a qualidade e eficiência dos processos de teste de *software*, alinhando-os com as melhores práticas da indústria e promovendo uma cultura de colaboração entre equipes de desenvolvimento e teste.

1.3 Organização do Trabalho

A estrutura deste trabalho consiste em cinco capítulos organizados conforme descrito abaixo:

- Capítulo 2: fundamentação teórica, abordando diversos assuntos relevantes para a clareza e compreensão do estudo, e incluindo também trabalhos correlatos para enriquecer os princípios e referências;
- Capítulo 3: visão detalhada sobre o tema central do trabalho, o qual é automação do processo de testes de *software* integrado ao BDD. Esta seção explica a junção das ferramentas *Cypress*, *Page Objects* e *Cucumber* com a sintaxe *Gherkin* escolhida para programar os scripts de automação;
- Capítulo 4: desenvolvimento da proposta de implantação do processo de testes de *software* utilizando a abordagem BDD, apresentando os resultados e comparando-os entre a automação de testes e os testes manuais;

- Capítulo 5: conclusão do trabalho, destacando as considerações finais, a validade do estudo de caso e sua contribuição para a academia, além de recomendações e sugestões para trabalhos futuros;
- Apêndice: passo a passo para a configuração do ambiente necessário para iniciar o processo de automação.

2 Fundamentação Teórica

Neste capítulo são apresentados e discutidos os principais conceitos e fundamentos relacionados ao BDD, à área de testes de *software* e à automação. São abordados aspectos teóricos de vários assuntos relevantes, fornecendo uma base sólida para a compreensão do tema e percepção sobre a escolha do BDD como metodologia para a proposta de implantação de processo de testes de *software* no estudo de caso apresentado.

2.1 Processo de Desenvolvimento de Software

Um processo de desenvolvimento define quais atividades e etapas devem ser seguidas para construir e entregar um sistema de *software* (VALENTE, 2020). Esse processo é uma sequência de etapas e atividades organizadas que visa a especificação, criação do projeto, desenvolvimento (codificação), testes, implantação e manutenção de um sistema de *software*. Geralmente é implementado de maneira sistemática para garantir a entrega de *software* de alta qualidade que atenda aos requisitos especificados.

Os modelos de processos de desenvolvimento de *software* utilizados na indústria de Tecnologia da Informação - TI, seguem uma estrutura comum usada para planejar e controlar o processo de desenvolvimento de um sistema de informação de forma eficiente e produtiva. Existem vários modelos de ciclo de vida de desenvolvimento de *software* definidos e projetados para serem praticados durante seu processo de desenvolvimento (PERERA; SILVA; PERERA, 2017). Neste trabalho é contextualizado os modelos Ágeis de Processo.

2.1.1 Metodologias Ágeis

Metodologias ágeis são abordagens de desenvolvimento de *software* que enfatizam a flexibilidade, a colaboração, a entrega contínua e a resposta rápida às mudanças. De acordo com Ambrosio e Faria (2021), os métodos ágeis se destacam por contraporem métodos tradicionais de desenvolvimento de *software*, como os modelos de Cascata e Espiral, que apesar de terem uso mais abrangente, eram considerados lentos e burocráticos. Os novos métodos levaram o apelido de “leves” em relação aos anteriores, considerados “pesados”.

Os processos ágeis tiveram um profundo impacto na indústria de *software*. Eles são utilizados pelas mais diferentes organizações que produzem *software*, desde pequenas empresas até as grandes companhias da *internet*. Diversos métodos que concretizam os princípios ágeis foram propostos, tais como *XP*, *Scrum*, *Kanban* e *Lean Development*. Esses métodos também ajudaram a disseminar diversas práticas de desenvolvimento de

software, como testes automatizados, *Test-Driven Development* (TDD, isto é, escrever os testes primeiro, antes do próprio código) e integração contínua (*continuous integration*) (VALENTE, 2020).

2.1.2 Scrum

O *Scrum* é uma metodologia ágil bastante popular e amplamente utilizada no desenvolvimento de *software*, e em projetos que envolvem colaboração e gerenciamento de equipe (PRESSMAN; MAXIM, 2021). Ele é um *framework* ágil de gerenciamento de projetos e desenvolvimento de *software*, que se concentra na entrega incremental e na colaboração em equipe.

Foi definido como o arcabouço de desenvolvimento de *software* entre os anos 1980 e 1990, por Schwaber e Sutherland (2020), signatários do Manifesto Ágil, visando lidar mais com os aspectos gerenciais de um projeto do que com os aspectos técnicos (AMBROSIO; FARIA, 2021).

Conforme destacado por Pham e Pham (2011) em seu livro "*Scrum* em Ação: Gerenciamento e Desenvolvimento Ágil de Projetos de *Software*", enfatiza-se a importância de uma equipe *Scrum* ser multifuncional, composta por cinco a nove membros e organizada em apenas três papéis:

- O *Scrum Master* (SM) - um facilitador que assegura que o time respeite e siga os valores e as práticas do método;
- O *Product Owner* (PO) - representante do cliente que ajudará a identificar os requisitos e tomará as decisões referentes ao negócio;
- A Equipe de Desenvolvimento (Time) - equipe com todas as aptidões necessárias para desenvolver o produto solicitado.

Esses membros participam de cerimônias que são reuniões e eventos regulares que fazem parte do processo e que permitem a colaboração, a inspeção e a adaptação contínuas. São elas: a) Reunião de Planejamento (*Planning*) - reunião que marca o início de cada *sprint* e nela são selecionadas as tarefas do *backlog* do produto que são implementadas durante o *sprint*; b) Reunião Diária (*Daily Scrum*) - reunião diária curta, com duração máxima de 15 minutos, realizada pela equipe de desenvolvimento. Cada membro compartilha o que fez desde a última reunião; c) Revisão de *Sprint* (*Review*) - ao final de cada *sprint*, a equipe realiza uma reunião de revisão para demonstrar o trabalho concluído ao PO e outros *stakeholders*; d) Retrospectiva de *Sprint* (*Retrospective*) - a equipe realiza uma retrospectiva para avaliar o próprio processo de trabalho, discutindo o que funcionou bem e o que precisa ser melhorado; e) *Grooming* ou Refinamento do *Backlog* (*Product*

Backlog Refinement) - embora não seja uma reunião formal do *Scrum*, o *Grooming* do *Backlog* é uma atividade contínua na qual o PO e a equipe de desenvolvimento trabalham juntos para refinar e priorizar o *backlog* do produto (PHAM; PHAM, 2011).

O *Scrum* alinha seus princípios com o manifesto ágil, segundo Pressman e Maxim (2021), estabelecendo uma base sólida que engloba atividades estruturais, incluindo requisitos, análise, projeto, evolução e entrega de produtos/funcionalidades. Essas atividades ocorrem em um ciclo denominado *Sprint*, cuja duração pode variar de 2 a 4 semanas, dependendo da complexidade do produto, e abrange uma seleção de requisitos/histórias a serem implementadas. Essas histórias de usuários, denominadas *User Stories*, são definidas pelo cliente durante a fase de levantamento de requisitos e fazem parte da lista do *Product Backlog* (Backlog do produto) (ANDERLE, 2015). O ciclo do *Scrum* pode ser representado na Figura 1.

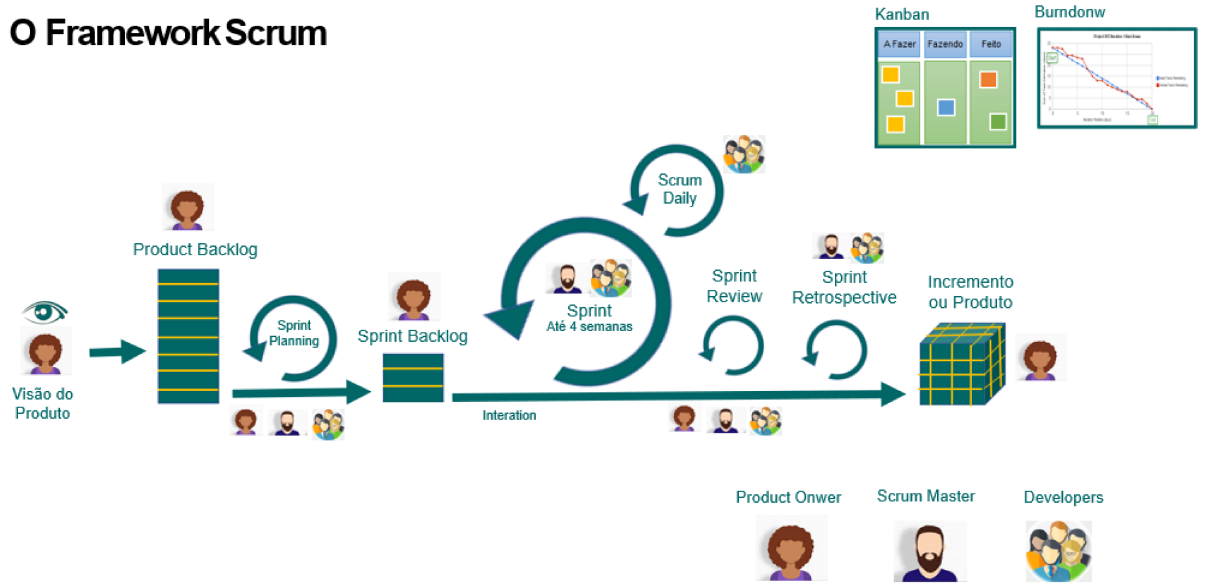

Figura 1 – Ciclo de desenvolvimento do *Scrum*. Fonte: Extraído de (SILVA, 2020).

2.2 Testes de Software

Teste consiste na execução de um programa com um conjunto finito de casos, com o intuito de verificar se ele possui o comportamento esperado (VALENTE, 2020). Ou seja, o ato de testar se refere à prática de verificar e avaliar o *software* para garantir que ele funcione corretamente e atenda aos requisitos estabelecidos. Além de possuir um papel fundamental na avaliação da qualidade do *software*, reduzindo o risco de falhas em operação.

O processo do teste, em poucas palavras, envolve a criação de casos de testes, que são cenários ou situações em que o *software* é submetido a diferentes entradas e condi-

ções. Os resultados desses testes são comparados com o comportamento esperado para determinar se o *software* está funcionando conforme o esperado. Quando problemas são identificados, eles são relatados e, em seguida, corrigidos pela equipe de desenvolvimento.

Garantir que um *software* seja testado é de suma importância por diversos motivos (OLSEN; ULRICH, 2018). Em primeiro lugar, os testes são essenciais para garantir a qualidade do produto, por ajudarem a identificar e corrigir eventuais erros ou falhas no sistema. Em segundo, um *software* testado proporciona maior confiança aos usuários, uma vez que reduz a probabilidade de falhas inesperadas ou comportamentos inadequados durante o uso. E, em terceiro, do ponto de vista econômico, investir em testes durante o desenvolvimento é mais vantajoso, pois identificar e corrigir problemas nessa fase é mais eficiente e econômico do que lidar com falhas após o lançamento, quando o *software* já está em produção (CLAUDIO; NETO, 2017).

O teste de *software* abrange uma variedade de atividades, das quais a execução dos testes e a verificação dos resultados são apenas uma parte. O processo de teste também engloba etapas como o planejamento dos testes, análise, *design*, implementação dos testes, avaliação da qualidade de um objeto de teste e relatórios de progresso e resultados de teste. Alguns testes envolvem a execução do componente ou sistema em análise, sendo denominados testes dinâmicos. Outros tipos de testes não requerem a execução do componente ou sistema, sendo conhecidos como testes estáticos (OLSEN; ULRICH, 2018).

2.2.1 Tipos de Testes

Consoante Olsen e Ulrich (2018), um tipo de teste é um grupo de atividades de teste destinadas a testar características específicas de um sistema de *software*, ou parte de um sistema, com base em objetivos de teste específicos. Tais objetivos podem incluir:

- Avaliar as características de qualidade funcional, como integridade, exatidão e adequação. Os chamados testes funcionais (aqui técnicas de caixa preta podem ser usadas);
- Avaliar características de qualidade não funcionais, como confiabilidade, eficiência de desempenho, segurança, compatibilidade e usabilidade. Os chamados testes não funcionais;
- Avaliar se a estrutura ou arquitetura do componente, ou sistema, está correta, completa e conforme o esperado. Os chamados testes de caixa branca;
- Avaliar os efeitos das mudanças, como confirmar que os defeitos foram corrigidos (teste de confirmação), e procurar mudanças não intencionais no comportamento

resultantes de mudanças de *software* ou ambiente (teste de regressão). Os chamados testes relacionados à mudança.

Segundo as palavras do autor Badgett et al. (2011) em seu livro "*The Art of Software Testing*", ele menciona que "O desenvolvimento de *software* é basicamente um processo de comunicação de informações sobre o programa final e a tradução dessas informações de uma forma para outra. Por esse motivo, a grande maioria dos erros de *software* pode ser atribuída a falhas, erros e ruídos durante a comunicação e tradução de informações.". E, para mitigar esses erros, ele diz que uma das alternativas é direcionar diferentes processos de teste para diferentes etapas de desenvolvimento. Pois, cada fase do teste é focada em uma etapa específica do ciclo de desenvolvimento, permitindo uma atenção especial a uma classe particular de erros (BADGETT et al., 2011).

Dessa forma, para cada avaliação citada mais acima, existem diversos tipos de testes, cada um com seu próprio objetivo e abordagem específica. Fornecendo uma estrutura sistemática para identificar defeitos, validar funcionalidades e garantir a conformidade com os requisitos do cliente. Abaixo, são explorados alguns dos tipos de testes de *software* mais conhecidos segundo Claudio e Neto (2017):

1. Testes de unidade: esses testes verificam unidades individuais de código, como funções ou métodos, isoladamente. Eles são frequentemente automatizados e auxiliam os desenvolvedores a identificar rapidamente erros em pequenas partes do código. Geralmente são realizados pelo próprio desenvolvedor;

2. Testes de integração: nesse tipo de teste, as unidades individuais já testadas são combinadas e testadas em conjunto. O objetivo é garantir que essas unidades funcionem corretamente em conjunto;

3. Testes de sistema: os testes de sistema verificam todo o sistema de *software* como uma entidade única. Eles são usados para validar se o sistema atende aos requisitos especificados e se está pronto para ser entregue ao cliente. Os testes E2E (*End-to-End*) fazem parte da categoria de testes de sistema, e visam avaliar o fluxo completo de uma aplicação (ponta-a-ponta), simulando a interação de um usuário com o sistema (FIGUEIREDO et al., 2022);

4. Testes de aceitação do usuário (UAT): esses testes são realizados pelos usuários finais para validar se o *software* atende aos seus requisitos e expectativas. Eles são cruciais para garantir a usabilidade e a satisfação do cliente;

5. Testes de regressão: com o desenvolvimento contínuo do *software*, é importante garantir que as novas alterações não causem regressões em funcionalidades existentes. Os testes de regressão são realizados para garantir que as atualizações não quebrem o que já estava funcionando;

6. Testes de desempenho: esses testes avaliam como o sistema se comporta em termos de velocidade, escalabilidade e estabilidade sob diferentes condições de carga. Eles ajudam a identificar gargalos de desempenho e otimizar o *software* (FERREIRA et al., 2022).

Esses são apenas alguns dos muitos tipos de testes de *software* disponíveis, cada um desempenhando um papel importante na garantia da qualidade e no sucesso de um produto de *software*. Vale ressaltar que os testes de *software* manuais sempre manterão sua relevância, por oferecem uma abordagem flexível e criativa para explorar o sistema. Em cenários altamente complexos, variáveis ou não repetitivos, eles podem ser mais eficazes, já que os testadores podem ajustar suas abordagens conforme necessário. Além da usabilidade e da experiência do usuário, ao envolverem a interação direta com o sistema para avaliar aspectos subjetivos como design e fluxo de trabalho (LOURENçO, 2022).

2.2.2 Garantia de Qualidade (QA)

Embora as pessoas empreguem frequentemente a expressão "Garantia de Qualidade", ou simplesmente "Controle de Qualidade", para se referir a testes, é importante ressaltar que garantia de qualidade e testes não são sinônimos, mas sim conceitos interligados. Um conceito mais amplo, o gerenciamento de qualidade, age como um elo entre eles. A garantia de qualidade concentra-se, em sua maioria, na conformidade com processos adequados, visando assegurar que os padrões de qualidade apropriados sejam alcançados. Quando os processos são executados de maneira apropriada, os produtos resultantes tendem a possuir uma qualidade superior, contribuindo, assim, para a prevenção de defeitos (OLSEN; ULRICH, 2018).

A Garantia de Qualidade de *Software*, ou *Quality Assurance* (QA), é um pilar essencial no desenvolvimento de *software*, focada em garantir que os produtos de *software* atendam aos padrões de qualidade e requisitos definidos. Segundo Pressman e Maxim (2021) "No desenvolvimento de *software*, a qualidade de um projeto engloba o grau de atendimento às funções e características especificadas no modelo de requisitos. A qualidade de consistência focaliza o grau em que a implementação segue o projeto e que o sistema resultante atende às suas necessidades e às metas de desempenho.".

O profissional que atua na área da qualidade, mais conhecido como *Quality Assurance Analyst*, ou Analista de Qualidade, é parte integrante da gestão de qualidade de *software* e suas funções envolvem uma série de processos, atividades e práticas destinadas a prevenir defeitos, melhorar a qualidade e aumentar a confiabilidade do *software*.

As práticas deste profissional apresentadas neste trabalho são: os Testes de *Software* - que desempenham um papel fundamental na qualidade do *software* para garantir que este funcione conforme o esperado e atenda aos requisitos de qualidade; a Automa-

ção de Testes - utilizada para melhorar a eficiência e a cobertura dos testes, reduzindo a chance de erros humanos e permitindo a execução de testes repetidamente; e a Técnica BDD - como artefato nas escritas dos cenários de testes de *software*, contribuindo para a comunicação e compreensão entre as partes envolvidas no desenvolvimento.

2.3 Behavior Driven Development (BDD)

Como citado na Introdução, a técnica BDD foi concebida por North e Ltd (2020), em 2003, em resposta aos problemas encontrados por ele enquanto utilizava e lecionava sobre as práticas da técnica TDD do autor *Kent Beck*. Tais problemas partiam do desentendimento na comunicação e compreensão entre as partes envolvidas no desenvolvimento de *software*, bem como a dificuldade de definir "o que", "quando"e "como"deve ser testado (CARVALHO; MARQUES, 2019). Em outras palavras, o BDD é uma evolução do TDD e enfatiza a comunicação e colaboração entre desenvolvedores, QAs, POs (Product Owners) e outras partes interessadas, utilizando linguagem natural para descrever o comportamento desejado do *software*.

Conforme o autor Smart (2023) menciona em sua obra "BDD *in Action*", o BDD é uma abordagem no campo da engenharia de *software* elaborada para auxiliar equipes no desenvolvimento e entrega de *software* mais valioso e de maior qualidade, em prazos reduzidos. Fundamenta-se em práticas ágeis e enxutas, como o TDD e o Design Orientado a Domínio (DDD). No entanto, sua característica mais distintiva é a utilização de uma linguagem comum, baseada em frases simples e estruturadas em inglês (ou na língua nativa das partes interessadas), facilitando a comunicação entre todos os envolvidos no projeto.

Em BDD, a linguagem de negócios é extraída das histórias ou especificações fornecidas pelo cliente durante a fase de levantamento de requisitos. Quando *Dan North* introduziu esse conceito, ele propôs um modelo para a escrita desses documentos. Apesar de ser apenas um modelo, ou seja, não é obrigatório, ele enfatiza a importância da equipe aderir a um padrão específico, pois isso simplifica a comunicação entre todos os participantes do projeto (SOARES, 2011).

Em relação à formatação para especificar esse modelo de escrita, é geralmente aplicada uma linguagem estruturada, conhecida como *Gherkin*. Segundo Utermohl (2023) "O *Gherkin* é uma linguagem simples e legível que descreve o comportamento do *software*. Enquanto o BDD é uma abordagem que se concentra no comportamento do usuário ao desenvolver o *software*.". Ou seja, um complementa o outro, sendo o *Gherkin* uma espécie de sintaxe que apoia a lógica da escrita no processo de BDD.

Embora o processo de BDD não se limite à sintaxe *Gherkin*, pois as equipes têm a liberdade de adotar abordagens como: especificação por exemplos; cenários com critérios

de aceitação bem definidos; fluxogramas. O *Gherkin* surgiu com o propósito de facilitar a incorporação do BDD e dessas especificações de negócio (SW, 2019). Como salientado por Smart (2023) em seu livro que aborda o BDD em prática, é possível observar que a maioria das ferramentas BDD destacadas no livro adotaram o formato *Gherkin*. Em outras palavras, a prática do BDD gera o *Gherkin* que por sua vez é utilizado por algumas ferramentas, por exemplo o *Cucumber* (CHAVES, 2021). A sintaxe *Gherkin* e ferramenta de automação *Cucumber* são tratadas detalhadamente mais adiante.

2.3.1 Escrita

A composição da escrita de cenários no processo do BDD pode ser divida em 3 etapas:

1. *Features* - São as funcionalidades que serão desenvolvidas no sistema.
2. *User Stories* - São as descrições simples de uma funcionalidade. Geralmente escritas utilizando o formato:
   - Como um (*Like a*) <PAPEL>
   - Eu posso/quero/devo (*I want/can*) <AÇÃO/FUNÇÃO>
   - Para/de (*For/to*) <RESULTADO>
3. Critérios de Aceitação ou Cenário (Scenario) - São apresentados como os próprios cenários, pois eles descrevem as ações que serão conferidas e testadas. Os cenários podem ser divididos em três elementos principais:
   - <DADO> (*GIVEN*) - pré-condições para executar o cenário;
   - <QUANDO> (*WHEN*) - o que eu quero realizar, passos do cenário;
   - <ENTÃO> (*THEN*) - resultado esperado pela execução do cenário.

Além disso, cada elemento pode ter um contexto adicional expresso no modelo pela palavra "E"(*AND*). A Figura 2 mostra o modelo padrão proposto e a Figura 3 um exemplo, na prática.

```
Título (uma linha descrevendo a história)

Narrativa:
Como [o papel]
Eu quero [recurso]
Assim que [benefício]

Critérios de Aceitação: (apresentado como Cenários)

Cenário 1: Título
Dado contexto []
  E [um pouco mais de contexto] ...
Quando [eventos]
Então [resultado]
  E [outro resultado ...]

Cenário 2: ...
```

Figura 2 – Modelo de escrita de Estória de Usuário em arquivo texto. Fonte: Extraído de (MEDIUM.COM, 2018).

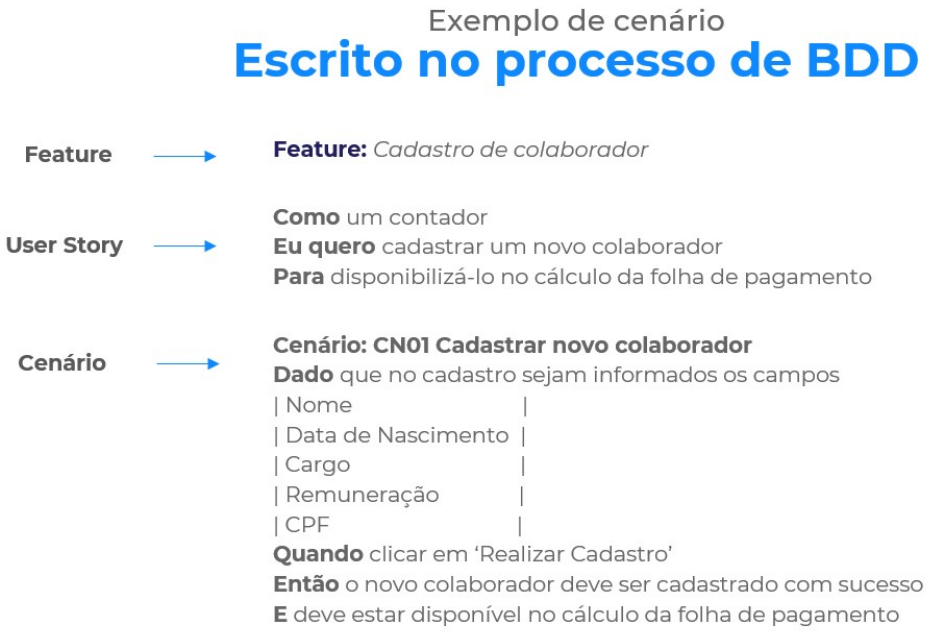

Figura 3 – Exemplo da escrita BDD, na prática. Fonte: Da Autora.

Devido o BDD se concentrar no comportamento do *software* a partir da perspectiva do usuário ou do negócio, ajuda a garantir que o *software* atenda aos requisitos reais e entregue valor aos usuários, em vez de apenas cumprir especificações técnicas. Como Anderle (2015) cita em seu trabalho, o BDD baseia-se no emprego de um vocabulário restrito e bem específico, a fim de reduzir ao mínimo qualquer interferência na comunicação, assegurando, assim, que todas as partes envolvidas, tanto da área de TI quanto do setor de negócios, estejam em perfeita sintonia.

2.4 Testes Automatizados

Conscientes da crescente relevância atribuída aos testes no contexto da garantia de qualidade, surgem diversos conceitos com o intuito de simplificar ainda mais essa função, um deles sendo a automação de testes (RIBEIRO, 2019).

A automação de testes é um processo no desenvolvimento de *software* que envolve o uso de ferramentas e *scripts* de *software* para executar testes de forma automatizada. Em vez de realizar testes manualmente, um testador ou equipe de teste escreve *scripts* que instruem o *software* a realizar ações específicas, interagir com componentes ou interfaces do sistema e avaliar os resultados automaticamente.

Automatizar o processo de *software* oferece inúmeras vantagens para garantir a qualidade e eficiência dos produtos. Ao realizar a automação dos testes, as equipes de desenvolvimento podem executar repetidamente testes de regressão, funcionais e de desempenho, permitindo a identificação precoce de defeitos e a validação contínua das funcionalidades implementadas. Isso não apenas acelera o ciclo de desenvolvimento, permitindo implementações mais rápidas e frequentes, mas também melhora a confiabilidade do *software*, reduzindo a ocorrência de erros em produção (CHICANELLI et al., 2019).

No entanto, automatizar não implica na substituição total dos testes manuais. Pois, de modo geral, pesquisas indicam que os testes manuais têm a capacidade de identificar mais defeitos em comparação aos testes automatizados. Isso ocorre porque, ao interagir diretamente com o sistema, sem depender exclusivamente de uma interface de programa que automatiza o teste, consegue-se pensar em uma solução que, até então, não havia sido considerada apenas por meio da leitura dos requisitos (PAULA, 2019).

2.4.1 Automação + BDD

A automação de testes e o BDD compartilham uma abordagem centrada no comportamento do *software*, na colaboração entre equipes e na integração de cenários de teste em práticas de desenvolvimento ágil. A automação de testes é uma parte prática da implementação do BDD, permitindo que os cenários de teste escritos em linguagem natural sejam executados automaticamente para verificar o comportamento do *software*.

Diferente da automação de testes convencional, que focam na interação direta com os elementos da interface do usuário ou no acesso direto às funcionalidades do sistema, as práticas do BDD expressam os requisitos e comportamentos esperados em termos de cenários de negócios compreensíveis para todas as partes interessadas. Diante disso, segundo o resultado obtido no trabalho de Chiavegatto et al. (2013), a aplicação da metodologia BDD junto à técnica de automatização agregou os seguintes benefícios ao processo de testes: facilidade de elaboração e entendimento dos cenários de testes por todo o time, e redução de esforço com a execução de testes.

As práticas do BDD resultam em um conjunto completo de testes unitários e de aceitação automatizados, minimizando o risco de regressões decorrentes de modificações na aplicação. Esses testes automatizados detalhados também aceleram consideravelmente o ciclo de lançamento. Os testadores não são mais obrigados a realizar extensas sessões de testes manuais antes de cada novo lançamento. Em vez disso, podem usar os testes de aceitação automatizados como base e direcionar seus esforços para atividades mais produtivas e eficazes, como testes exploratórios e outras avaliações manuais mais complexas (SMART, 2023).

Além disso, no BDD, os cenários de teste são considerados especificações executáveis do comportamento do *software*, e a automação de testes transforma essas especificações em testes automatizados que podem ser executados repetidamente para validar o comportamento do *software*. Ainda, ambas abordagens se encaixam bem com a prática de Integração Contínua (CI). A automação de testes é fundamental para a CI, pois os testes automatizados podem ser executados continuamente para verificar a estabilidade do *software*. Enquanto o BDD também pode ser integrado ao *pipeline* de CI, permitindo a execução automatizada de cenários de teste.

2.5 Trabalhos Relacionados

Na revisão bibliográfica para esta monografia, destacam-se diversos trabalhos correlatos que abordam temas relacionados ao deste estudo. Um dos trabalhos relevantes é o artigo desenvolvido por Irshad, Britto e Petersen (2021), cujo título, traduzido para o português, é: Adaptando o Behavior Driven Development (BDD) para sistemas de *software* de grande escala, que discute a importância da colaboração entre as partes interessadas em projetos complexos e destaca a necessidade de abordagens eficazes para promover essa interação. Nesse sentido, o estudo propõe uma metodologia adaptada de BDD, demonstrando seu impacto positivo na colaboração entre *stakeholders*.

Os autores também exploram os desafios enfrentados em projetos de grande porte e destacam a importância de processos bem definidos para lidar com essas complexidades. O estudo concentra-se na reutilização de artefatos no contexto de desenvolvimento de *software*, identificando práticas eficazes para melhorar a eficiência do processo. Esse trabalho é particularmente relevante, uma vez que a reutilização de artefatos foi considerada um dos benefícios do BDD na presente análise.

Os métodos utilizados abordaram a adoção de novas ferramentas, versionamento de comportamentos, transferência de tecnologia para conseguir aplicar o processo baseado em BDD, além de seis sessões de *workshop* para compreender os desafios e benefícios do BDD.

A revisão desse artigo destaca a consistência e a relevância do presente trabalho, ao

proporcionar visões valiosas sobre a adaptação do BDD em projetos de *software* de grande escala. Essa análise recente e aprofundada da literatura, respalda a contribuição original deste trabalho, tendo o BDD como uma proposta efetiva e importante para aprimorar as etapas do desenvolvimento de *software*.

A monografia de graduação em Engenharia de Computação, elaborada por Lourenço (2022), cujo tema é: Automação de Testes para um Sistema de *E-commerce*, se concentra na automação de testes para o sistema de vendas *online* da Promofarma, desenvolvido pela *Codeby*. A empresa anteriormente realizava testes manualmente, o que resultava em atrasos na liberação de *software* e demandava muito esforço da equipe de testes. O objetivo do trabalho foi implementar a automação de testes utilizando o *framework Cypress* em combinação com o padrão *Page Object*, que inclusive são algumas das ferramentas que também são utilizadas neste estudo.

Como resultado, foram definidos 37 casos de teste automatizáveis e a execução automatizada desses testes mostrou-se significativamente mais rápida, eficiente e menos dispendiosa em comparação com os métodos manuais. Observou-se que a automação complementa os testes manuais, promovendo o reuso e cobrindo boa parte das funcionalidades do sistema.

As principais contribuições do trabalho foram a criação de um catálogo de casos de teste para sistemas de *e-commerce* e um template para automação de casos de teste. O autor explica que para futuros trabalhos, pretende-se refatorar o código implementado para atender a outros projetos da empresa, além de implementar a automação para as versões *mobile* e *tablet* do sistema.

Dessa forma, é nítido que o trabalho de Lourenço (2022) e o trabalho em questão têm em comum o objetivo de implementar a automação de testes de *software* para melhorar a qualidade e eficiência dos processos de teste. No entanto, este estudo, além de focar na automação, também propôs uma mudança no processo de testes, adotando a abordagem BDD, que enfatiza a colaboração entre as equipes de desenvolvimento e teste, utilizando uma linguagem comum para definir requisitos e comportamentos do sistema.

Outro trabalho relacionado ao tema desta monografia é o livro recém-publicado "*BDD in Action - Behavior-Driven Development for the whole software lifecycle*", em português: BDD em Ação - Desenvolvimento Orientado a Comportamento para todo o ciclo de vida do *software*, escrito por Smart (2023), cujo prefácio foi escrito pelo próprio Dan North, criador do BDD. O livro proporciona uma visão abrangente de como as práticas de BDD se aplicam em todas as fases do processo de desenvolvimento de *software*, desde a descoberta e definição de requisitos até a implementação dos recursos do sistema e a criação de especificações executáveis por meio de testes automatizados e unitários.

O livro destina-se a equipes que enfrentam desafios como requisitos desalinhados,

mudanças frequentes, tempo perdido devido a defeitos e retrabalho, visando melhorar a qualidade do produto. O autor reconhece que diferentes profissionais, como analistas de negócios, testadores, desenvolvedores, gerentes de projeto e partes interessadas do negócio, podem extrair benefícios específicos do livro.

Sua estrutura apresenta inicialmente motivações, origens e a filosofia geral do BDD, com uma introdução prática do BDD no mundo real. Depois explora como as práticas de BDD podem auxiliar as equipes na análise eficaz de requisitos, apresentando técnicas importantes, e estabelecendo a base conceitual para o restante do livro. Também oferece uma cobertura técnica das práticas de BDD, abordando técnicas para automatizar testes de aceitação de forma robusta, ferramentas BDD para diferentes linguagens e como o BDD contribui para códigos mais limpos e bem projetados.

Além disso, o livro analisa o BDD no contexto mais amplo do gerenciamento de projetos, documentação de produtos, relatórios e integração no processo de construção, com exemplos práticos em várias linguagens. Essa abordagem prática e abrangente torna essa obra uma valiosa fonte tanto para profissionais em busca de aprimoramento no desenvolvimento de *software* através do BDD, quanto para o tema deste trabalho, como uma importante ferramenta e fonte de informações.

Por fim, o artigo desenvolvido por Carvalho e Marques (2019), que tem como tema: Proposta de uso da técnica BDD para otimizar a escrita e automação de testes no *framework Scrum*, mostra um enfoque maior no processo de automação de testes. O artigo destaca a importância dos testes no desenvolvimento de *software* para garantir a qualidade do produto final, e sua proposta central é integrar a técnica BDD ao *framework Scrum*, utilizando a linguagem natural "Dado-Quando-Então". Um estudo de caso é apresentado, demonstrando a viabilidade da proposta com o uso do *framework Spock*.

Os resultados do estudo indicam que a aplicação da linguagem natural, especificamente o padrão "Dado-Quando-Então"do BDD, é aplicável para a criação e execução de testes em *frameworks* que suportam essa técnica no ciclo de desenvolvimento do *Scrum*. A abordagem proporciona uma fácil interpretação e leitura dos testes, especialmente quando comparada às abordagens mais tradicionais, como *JUnit*. O uso da linguagem natural no BDD facilita a comunicação e compreensão dos testes, aproximando-se da linguagem humana e tornando o entendimento mais acessível, o que pode ser particularmente relevante para novos desenvolvedores em um projeto.

No entanto, o artigo mostra que, embora o BDD promova a colaboração e comunicação, a validação desses aspectos em um contexto real de *stakeholders* não foi possível devido à natureza do estudo de caso isolado. Mesmo assim, acredita-se que a linguagem natural pode contribuir para a abordagem iterativa e centrada nas pessoas do *Scrum*. Além disso, a pesquisa observou que o grau de detalhamento dos cenários de funcionalidades varia consoante o nível de teste, sendo menor para testes de unidade e maior para

testes de integração e aceitação.

Assim, o artigo oferece uma proposta concreta e demonstra benefícios potenciais na combinação de BDD com o *framework Scrum*, destacando a clareza na expressão dos testes e o potencial impacto positivo na comunicação e compreensão da equipe de desenvolvimento.

2.6 Conceitos Relevantes

Os conceitos citados nesta seção foram adicionados como assuntos relevantes para o tema deste trabalho, uma vez que estão intimamente relacionados à automação com BDD. Tanto o *DevOps* quanto o *ScrumOps* representam abordagens que visam a integração contínua, a entrega contínua e a automação de processos em todo o ciclo de vida do desenvolvimento de *software*. A implementação eficaz dessas práticas proporciona uma base sólida para a aplicação bem-sucedida do BDD, permitindo a criação de testes automatizados que refletem os comportamentos esperados do *software*.

2.6.1 DevOps

DevOps é um conjunto de métodos nos quais desenvolvedores e operações se comunicam e colaboram para fornecer *software* e serviços de forma rápida, confiável e com maior qualidade. A palavra *"DevOps"* surgiu com Dev de Desenvolvedores e Ops de Operação (PERERA; SILVA; PERERA, 2017). É importante entender que o *DevOps* é uma filosofia e um conjunto de práticas, e não algo que possa ser atribuído a uma única pessoa ou entidade, pois não é uma tecnologia ou uma ferramenta, mas sim uma cultura e uma abordagem de colaboração entre equipes de desenvolvimento.

A abordagem do *DevOps* tem evoluído ao longo do tempo com contribuições de várias pessoas e organizações. No entanto, *Patrick Debois* é frequentemente creditado como um dos pioneiros do movimento *DevOps*. Ele organizou a primeira conferência *DevOpsDay* em 2009, que foi um marco importante na disseminação das práticas e princípios do *DevOps* (RAFAEL, 2019).

DevOps é comumente considerado a interseção entre os escopos de Desenvolvimento de *Software* (Dev), Operação (Ops) e Quality Assurance (QA). Apesar de sua disseminação, não existe uma definição universalmente aceita para *DevOps* (PIETRANTUONO et al., 2019). Além disso, é importante ressaltar que não há uma fórmula precisa para a implementação dessa metodologia. Na verdade, ela deve ser adicionada no cotidiano de uma empresa de uma maneira que faça sentido, sendo significativa e benéfica às operações das equipes.

Figura 4 – Escopo *DevOps*. Fonte: Extraído de (PIETRANTUONO et al., 2019).

Os principais pilares do *DevOps* de acordo com o artigo de Soares e Castelli (2023) incluem:

1. Cultura colaborativa: criação de uma cultura nas equipes de desenvolvimento para que todos trabalhem juntos, incentivando a multidisciplinaridade dos profissionais;
2. Automação: automação das tarefas manuais executadas pelos times. A ideia é automatizar tarefas repetitivas e propensas a erros, como compilação de código, testes, implantação, provisionamento de infraestrutura e monitoramento;
3. Métricas: criação de métricas para avaliação se a implantação da metodologia foi efetiva. Isso envolve monitorar o desempenho de sistemas, aplicações e processos ao longo de todo o ciclo de vida de desenvolvimento e implantação;
4. Compartilhamento: compartilhamento de informações e conhecimento entre os times. Incentivando a divulgação de conhecimento, boas práticas e lições aprendidas na organização. De modo a reduzir a dependência de indivíduos, promovendo a colaboração e a melhoria contínua.

Dentro desses pilares, existem a Integração Contínua (CI) e a Entrega Contínua (CD), que são práticas essenciais no contexto do *DevOps*, e que visam melhorar o processo de desenvolvimento de *software*. A Integração Contínua, ou *"Continuous Integration"*, é o pilar de *DevOps* responsável por integrar, testar e conferir alterações no código-fonte de um projeto regularmente, ela tem em vista identificar problemas de integração o mais cedo possível no ciclo de vida do *software*, para poderem ser corrigidos de forma rápida e eficaz (SOARES; CASTELLI, 2023) (BRAGA, 2015).

Já a Entrega Contínua, ou *Continuous Delivery*, é a técnica de desenvolvimento de *software* que automatiza todo o processo de criação, teste e implantação de *software* para fornecer novos recursos de maneira eficiente e consistente. O principal objetivo da entrega contínua é automatizar todo o ciclo de entrega de *software*, do desenvolvimento

à implantação, para permitir que as equipes de desenvolvimento forneçam novos recursos de maneira rápida e eficaz, sem sacrificar a qualidade (SOARES; CASTELLI, 2023) (BRAGA, 2015).

2.6.2 ScrumOps

No âmbito do desenvolvimento de *software*, um conceito em ascensão é o chamado *"ScrumOps"*. A integração da prática *DevOps* com um *framework Scrum* pode levar a uma entrega mais rápida, eficiente e confiável de *software*, alinhando o desenvolvimento com as necessidades operacionais e de negócios. O *Scrum* pode ser usado para gerenciar o desenvolvimento de *software*, enquanto o *DevOps* se concentra na entrega e operação contínuas. Como Ambrosio e Faria (2021) demonstraram em seu trabalho, o processo de criação de um produto de *software* pode se beneficiar de diversas formas caso métodos ágeis sejam utilizados em conjunto com as práticas *DevOps*.

Diante desses dois conceitos, o termo *ScrumOps* surgiu e pode ser considerado um avanço recente nesta área. Conforme o artigo escrito por Waters (2017), *Dave West* - CEO (Chief Executive Officer) da *Scrum.org* e *Jayne Groll* - dirigente do *DevOps Institute* (DOI), anunciaram uma nova parceria estratégica entre as suas respectivas organizações e concordaram em colaborar em conteúdo, treinamento e recursos contínuos. E, devido a isso, eles criaram o novo termo para descrever sua missão conjunta: *"ScrumOps"*, como mostra a Figura 5.

ScrumOps refere-se a uma nova abordagem para entrega de *software*, unida pelo *Scrum.org* e pelo instituto *DevOps*. Ele foi criado para transformar as organizações de TI na era ágil e de trabalho da próxima geração, reunindo *Scrum* e *DevOps*. O *ScrumOps* visa oferecer um caminho para as equipes entregarem continuamente *software* seguro e funcional enquanto medem o sucesso. Em outras palavras, *Scrum* fornece uma estrutura leve para a entrega do *software*, enquanto o *DevOps* fornece cultura, automação e práticas para apoiar a entrega do *software* (JENA; MOH, 2023).

Figura 5 – *Scrum.org* e *DevOps Institute*. Fonte: Extraído de (BESTDEVOPS, 2017).

3 Automação do Processo de Testes de Software

A crescente complexidade dos sistemas de *software* modernos e a demanda por implantações rápidas e confiáveis impõem desafios significativos às equipes de desenvolvimento. Nesse contexto, a automação de testes emerge como uma solução vital para mitigar os riscos associados à manutenção de qualidade, eficiência e agilidade no ciclo de desenvolvimento de *software*. Diante disso, neste capítulo será abordado de forma mais detalhada o processo de automação de testes de *software* escolhido para este trabalho.

3.1 Contexto da Automação de Teste de Software

A automação é uma prática essencial na indústria de desenvolvimento de *software*, onde ferramentas e *scripts* são utilizados para executar testes automaticamente, em vez de depender exclusivamente de intervenção manual (CHICANELLI et al., 2019). Isso traz várias vantagens significativas para as equipes de desenvolvimento, descritas na Figura 6:

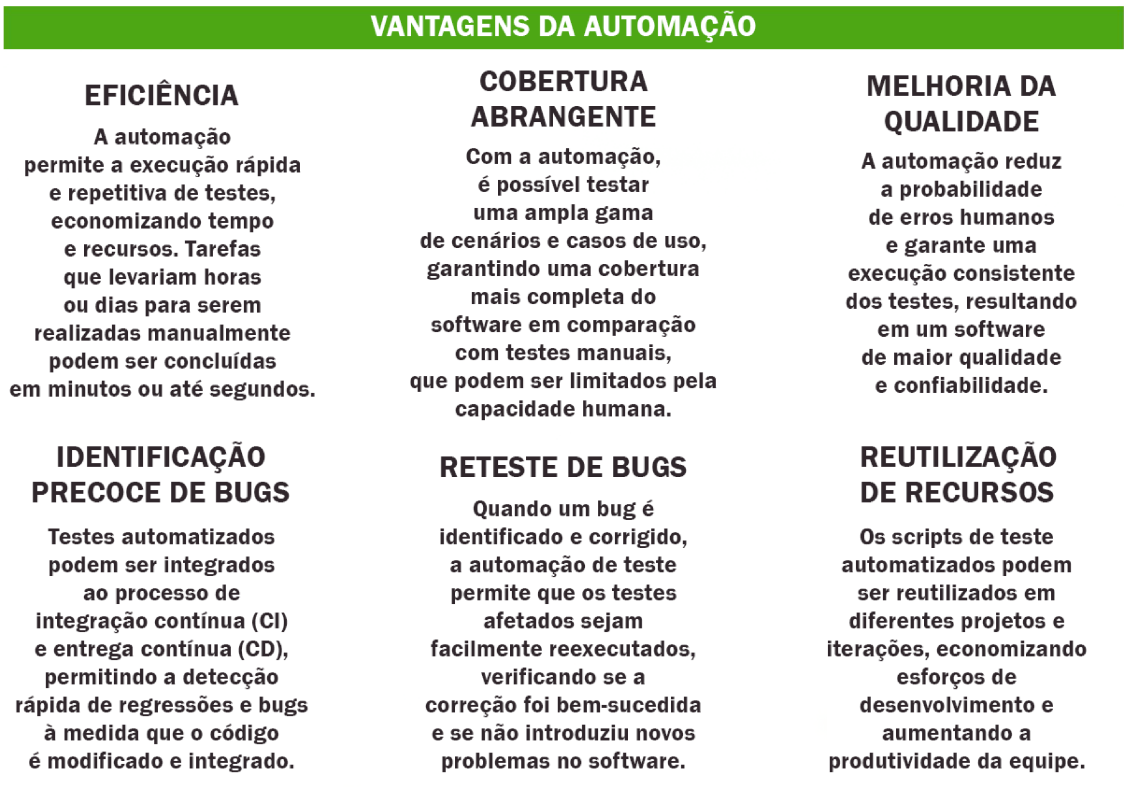

Figura 6 – Vantagens da Automação. Fonte: Da autora.

No entanto, por mais que a automação de testes ofereça uma série de vantagens incontestáveis, é importante reconhecer que há casos em que os testes manuais continuam sendo a melhor abordagem. Pois, é possível realizar testes exploratórios que se aproximam de usuários reais. Segundo o autor Smart (2023) em seu livro, ressalta "Nem todos os cenários precisam ser automatizados; alguns podem ser muito complicados para serem automatizados de maneira econômica e podem ser deixados para testes manuais (...). Mas quando um cenário pode ser automatizado, quando faz sentido fazê-lo e quando é bem feito, automatizar o cenário traz o seu próprio conjunto de benefícios inegáveis.".

Os tipos de testes mais frequentemente automatizados são os testes de unidade e os testes de aceitação. Esses testes desempenham um papel primordial na garantia da qualidade do *software*, sendo considerados atividades importantes para garantir que o projeto seja entregue ao cliente com o mínimo de ocorrência de erros, ou *"bugs"*, que poderiam causar transtornos ou aumentar os custos durante o desenvolvimento (SANTOS, 2009).

Visando aprimorar a qualidade e a autonomia dos testes, surgiram ferramentas desenvolvidas para auxiliar nesse processo. Cada uma com suas características específicas, destinadas a diferentes tipos de teste (SANTOS, 2009). Algumas ferramentas para automação de testes de *software* incluem:

- *Selenium*: uma ferramenta de código aberto para automação de testes de aplicativos da *web*, que oferece suporte a várias linguagens de programação, como *Java*, *Python*, e C# (FERREIRA et al., 2022);
- *Cypress*: uma ferramenta moderna e de código aberto para automação de testes de aplicativos da *web*, conhecida por sua facilidade de uso e rapidez de execução, permitindo a execução de testes em vários navegadores de forma rápida e confiável (CYPRESS.IO, Acesso em abril de 2024);
- *Appium*: uma ferramenta de automação de testes móveis de código aberto que suporta testes em dispositivos *iOS*, *Android* e *Windows* (SANTOS, 2022);
- *JUnit* e *TestNG*: estruturas de teste de código aberto para *Java*, amplamente utilizadas para automação de testes unitários e de integração (SANTOS, 2009);
- *Robot*: uma ferramenta de automação de testes de código aberto que oferece suporte a testes de aceitação, teste de unidade, teste de interface de usuário e testes de API (*Application Programming Interface* ou Interface de Programação de Aplicação). Ele possui uma sintaxe fácil de aprender e suporta várias bibliotecas e extensões (FERREIRA et al., 2022);
- *Postman*: uma ferramenta amplamente utilizada para automação de testes de API, possibilitando não apenas a criação e envio de solicitações HTTP (*Hypertext Trans-*

fer Protocol ou Protocolo de Transferência de Hipertexto) para interagir com APIs, mas também a escrita e execução de testes automatizados para validar seu comportamento (CARVALHO, 2022).

Como no mercado várias ferramentas estão disponíveis, cada uma com suas características específicas, é essencial avaliar cuidadosamente as vantagens e desvantagens de cada uma delas. Para assim, escolher a melhor e mais apropriada ferramenta de automação que se encaixe com o projeto que se quer automatizar (LOURENçO, 2022).

De acordo com Silva (2019), estabelecer estratégias que definem critérios de avaliação para a escolha da ferramenta certa são importantes, porque conseguem otimizar o tempo, evitando desperdício tanto de tempo quanto de esforço. Aspectos como o grau de complexidade da ferramenta, seu custo, curva de aprendizagem, capacidade de reutilização de código e compatibilidade com diferentes plataformas podem ser considerados durante essa avaliação.

Dito isso, a ferramenta escolhida para realizar a automação deste projeto foi o *Cypress*, um *framework* que oferece várias vantagens distintas em relação a outras opções disponíveis no mercado. Uma das principais vantagens do *Cypress* é sua arquitetura moderna e eficiente, que permite a execução rápida e confiável dos testes *end-to-end* em aplicativos da *web*. Além disso, o *Cypress* é reconhecido por sua simplicidade e facilidade de uso, facilitando a criação e manutenção dos testes, mesmo para aqueles com pouca experiência em programação. Outro ponto forte do *Cypress* é sua capacidade de visualizar e depurar os testes em tempo real, simplificando o processo de identificação e correção de problemas.

3.2 Cypress

O *Cypress* é um *framework* de teste de *front-end* de código aberto que permite a escrita e execução de testes de maneira simples e eficiente. Ele oferece uma API intuitiva e robusta, possibilitando a interação direta com elementos da interface do usuário e simulando ações, como clicar, digitar e navegar, além de fornecer recursos avançados de asserção e *debug*. O *Cypress* é mais frequentemente comparado ao *Selenium*, no entanto, ele é fundamentalmente e arquitetonicamente diferente. *Cypress* não é limitado pelas mesmas restrições que o *Selenium*, isso permite que você escreva testes mais rápidos, fáceis e confiáveis (CYPRESS.IO, Acesso em abril de 2024).

Os usuários do *Cypress* normalmente são engenheiros de controle de qualidade (QA) ou desenvolvedores que criam aplicativos da *web* usando estruturas *JavaScript* modernas cypress.io (Acesso em abril de 2024). Além disso, o *Cypress* permite que o usuário escreva todos os tipos de testes, desde testes *end-to-end*, à testes de componentes, integra-

ção e testes unitários. Com ele também é possível configurar, escrever, executar e depurar os testes.

Uma grande vantagem do *Cypress* é sua aderência ao conceito *"All in one"*, significando que, com o uso único e exclusivo dele, o desenvolvedor tem à disposição uma ampla variedade de ferramentas para o desenvolvimento, execução e visualização de relatórios dos testes automatizados (STAFFEN, 2021).

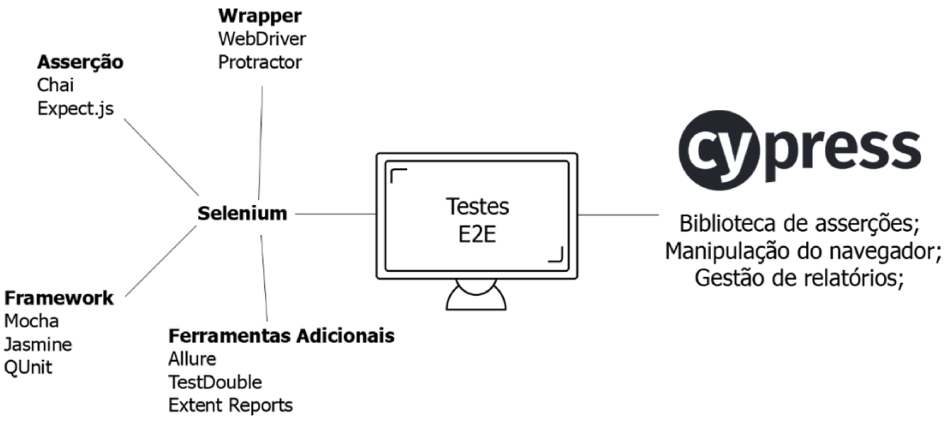

Figura 7 – Diferença entre *Cypress* e *Selenium*. Fonte: Extraído de *Testing Company* (STAFFEN, 2021).

Conforme ilustrado na Figura 7, o *Selenium* opera externamente ao navegador, requerendo o envio de comandos remotos para interagir com as páginas a serem testadas. Também é necessário selecionar um *framework* de teste, como *Mocha*, *Jasmine* ou *QUnit*, uma biblioteca de asserção, como *Chai* ou *Expect.js*, e um *wrapper*, uma função designada para chamar uma ou mais funções, como o *WebDriver* para *Java*. Em contrapartida, o *Cypress* executa comandos remotos dentro do mesmo ciclo de execução do navegador, permitindo acesso direto aos comandos nativos. Além disso, como o *Cypress* é instalado localmente na máquina em uso, a ferramenta tem a capacidade de acessar recursos específicos da máquina (STAFFEN, 2021).

3.3 Page Objects

O *Page Objects* é um padrão de design amplamente utilizado em automação de testes de *software* e também é utilizado neste projeto. Como diz o autor Nascimento (2021) "*Page Object* é um padrão de design que ajuda a aprimorar a manutenção de testes e reduzir a duplicação de código, também pode ser utilizado para descrever e documentar o fluxo de uma aplicação.". Em outras palavras, ele consiste em encapsular a interação com elementos da interface do usuário em classes ou módulos separados, chamados de *"Page Objects"*. Esses objetos representam páginas ou componentes específicos da aplicação.

A ideia por trás dos *Page Objects* é criar uma abstração dos elementos da interface do usuário, de modo que a lógica de interação com esses elementos seja encapsulada em métodos dentro dos *Page Objects*. Em vez de acessar diretamente os elementos da página em cada teste, os testes interagem com os métodos fornecidos pelos *Page Objects*, o que torna o código de teste mais legível, reutilizável e fácil de manter. Dessa forma, agrupamos todos os elementos e métodos específicos de uma determinada página em seu próprio arquivo, que é uma classe *JavaScript*, e os utilizamos diretamente nos *scripts* de teste (NASCIMENTO, 2021).

Alguns benefícios-chave dos *Page Objects*, de acordo com Marinheiro (2020), incluem:

- Facilidade de entendimento do código: a leitura e compreensão dos testes é facilitada significativamente. Os cenários são mais concisos, já que em vez de blocos de código com ações diretamente nos testes, teremos apenas a chamada de uma ou mais funções para executar determinada ação. Assim, o código de teste se torna mais legível e organizado, facilitando a compreensão e manutenção;
- Reutilização de código: ao separar os elementos e ações dos testes, elimina-se a necessidade de duplicação de código para realizar ações repetidas dentro dos cenários. Em vez disso, é simplesmente chamado a referência já existente de um elemento ou ação específica daquela página;
- Manutenção simplificada: qualquer alteração na interface do usuário pode ser feita em um único lugar, dentro do *Page Object* correspondente, em vez de modificar vários testes. Ou seja, é necessário alterar apenas o arquivo onde se encontra a referência para aquela ação/elemento.

Em resumo, os *Page Objects* são uma prática recomendada para desenvolver testes automatizados robustos e escaláveis, especialmente em ambientes onde a interface do usuário está sujeita a mudanças frequentes. Por conta disso, para deixar este trabalho o mais vigoroso e claro possível, foi decidido utilizar o *Page Objects* nesta automação.

3.4 Cucumber e sintaxe Gherkin

O *Cucumber* é uma ferramenta de automação de testes de *software* que facilita a implementação do BDD. Segundo os autores e desenvolvedores do *Cucumber* Rose, Wynne e HellesØy (2015) "O *Cucumber* foi projetado para ajudar a construir pontes entre os membros técnicos e não técnicos de uma equipe de *software*.". Ele permite que equipes de desenvolvimento escrevam testes automatizados em uma linguagem natural

compreensível por todos os envolvidos no projeto, desde desenvolvedores até *stakeholders* não técnicos.

A ferramenta *Cucumber* utiliza uma sintaxe chamada *Gherkin* para descrever o comportamento esperado do sistema em cenários de teste. Esses cenários são escritos em formato de especificação, usando palavras-chave como "Dado"(Given), "Quando"(When) e "Então"(Then) para descrever o estado inicial, a ação realizada e o resultado esperado, respectivamente. Isso torna os testes mais compreensíveis e colaborativos, facilitando a comunicação entre os membros da equipe e garantindo que todos tenham uma compreensão clara dos requisitos e expectativas do *software*.

A sintaxe *Gherkin* proporciona uma estrutura simples para registrar casos de comportamento desejados pelos *stakeholders*, de modo que tanto eles quanto o *Cucumber* possam entender facilmente. Embora seja chamado de linguagem de programação, o *Gherkin* foi projetado principalmente para ser legível para humanos, permitindo que você escreva testes automatizados que se assemelham a uma documentação (ROSE; WYNNE; HELLESØY, 2015).

Além disso, o *Cucumber* é altamente flexível e pode ser integrado com várias linguagens de programação e *frameworks* de automação de testes, como *Java*, *JavaScript*, *Cypress*, *Ruby* e outros. Isso o torna uma escolha popular para equipes que desejam adotar práticas de BDD em seus processos de desenvolvimento de *software*, promovendo uma abordagem mais colaborativa e orientada a comportamento na criação e execução de testes automatizados.

3.5 Automação com *Cypress*, *Page Objects* e *Cucumber* com a sintaxe *Gherkin* (BDD)

Como foi visto nas seções deste capítulo, entre as ferramentas amplamente reconhecidas por sua eficácia na automação de testes, destacam-se o *Cypress* e o *Cucumber*, notáveis por sua capacidade de implementar o *Behavior-Driven Development*. E, a principal ferramenta que implementa a abordagem BDD nos testes automatizados é o *Cucumber* fazendo uso da sintaxe *Gherkin* (ROSE; WYNNE; HELLESØY, 2015).

A combinação do *Cypress* com o *Cucumber* e sintaxe *Gherkin* proporciona uma estrutura poderosa para a automação de testes de *software*, integrando os benefícios do BDD com a confiabilidade e simplicidade do *Cypress*. Ao adotar essa abordagem, as equipes de desenvolvimento podem traduzir requisitos de negócios em cenários de teste claros e compreensíveis, possibilitando uma colaboração mais estreita entre todas as partes interessadas.

Além disso, ao incorporar o conceito do *Page Objects* à estrutura de automação, as

equipes podem colher uma série de vantagens adicionais. *Page Objects* ajudam a modularizar o código de automação, promovendo a reutilização e a manutenção simplificada. Isso significa que, ao atualizar ou modificar elementos da interface do usuário, as mudanças podem ser feitas em um único local, ao invés de modificar em vários *scripts* de teste, aumentando a eficiência e reduzindo a probabilidade de erros.

Com isso, a automação de testes com essas quatro ferramentas juntas oferece uma série de vantagens em comparação aos testes manuais. Em primeiro lugar, os testes automatizados garantem uma cobertura abrangente dos cenários críticos, reduzindo significativamente o risco de regressões e erros após a implementação de novas funcionalidades ou correções. Além de que a execução automatizada dos testes permite uma iteração mais rápida durante o processo de desenvolvimento, identificando falhas precocemente e facilitando a correção imediata, antes mesmo que se tornem problemas mais graves.

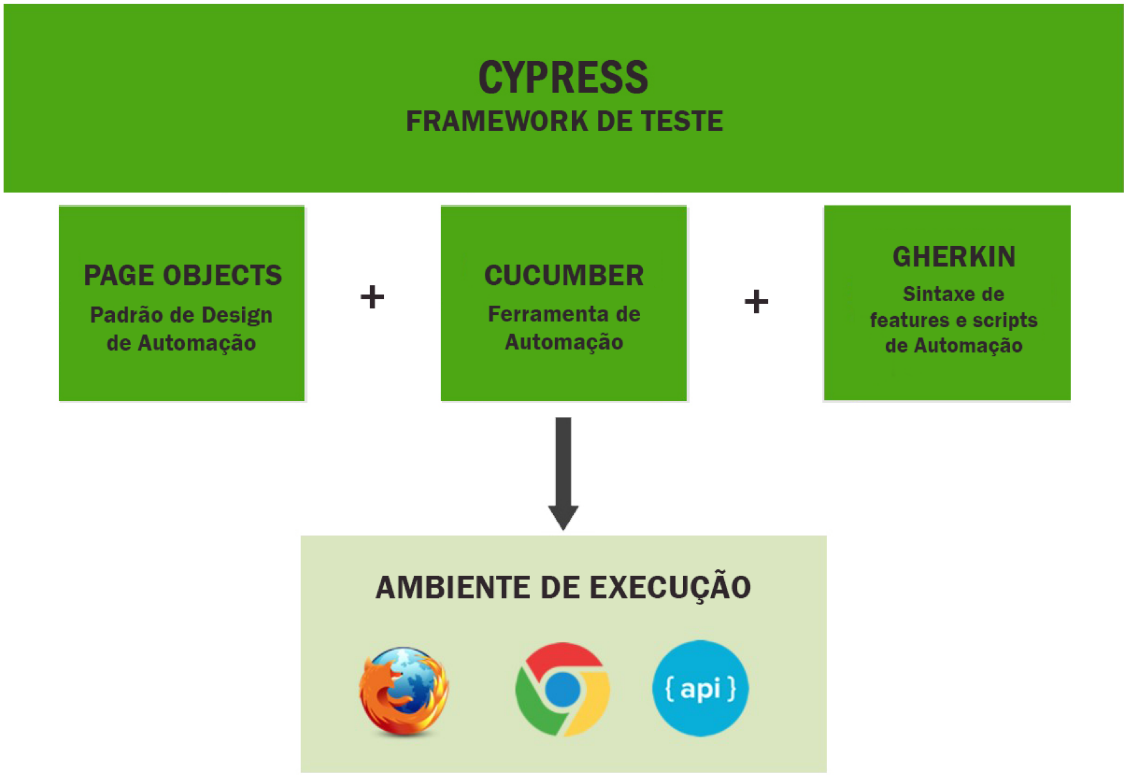

Figura 8 – Ambiente de Execução. Fonte: Adaptado de Taylor (2017).

4 Desenvolvimento

Este capítulo fornece uma visão detalhada do planejamento, implementação e avaliação do processo de teste de *software* proposto, destacando os passos e etapas envolvidos na sua execução. São apresentados os procedimentos adotados para a aplicação do BDD no contexto do projeto, incluindo a definição de cenários de teste, a escrita de especificações em linguagem natural, a implementação dos testes automatizados, os desafios enfrentados durante a implantação e a análise dos resultados obtidos. Ao final do capítulo, espera-se fornecer uma visão abrangente e detalhada de como foi o processo de implantação dos testes, demonstrando sua aplicabilidade e benefícios no contexto do estudo de caso em questão.

4.1 Contextualização do Estudo de Caso

A empresa escolhida como objeto desse estudo de caso terá seu nome ocultado por questões legais de proteção de dados. A empresa XYZ é uma empresa brasileira extremamente renomada. É da área de tecnologia e logística voltada para o varejo, especializada na venda de eletrodomésticos, móveis, eletrônicos, eletroportáteis, produtos de informática, telefonia, entre outros.

A empresa-alvo abrange diversas áreas e iniciativas, sendo uma delas o projeto selecionado para este estudo: um sistema de precificação de produtos. Este sistema é utilizado pelos administradores para estabelecer os preços dos produtos comercializados pela empresa. O *Pricing*, como é conhecido, serve como o ponto central onde os preços são definidos, permitindo que outros sistemas e ferramentas os utilizem. Por exemplo, se houver uma mudança na tabela de preços programada para setembro, em agosto são preparadas as variações de preço para que, no próximo mês, outras ferramentas utilizem as simulações criadas no *Pricing* e apliquem os preços cadastrados.

Este projeto adotou a metodologia *Scrum* para sua gestão, visando aumentar a eficiência e a agilidade no desenvolvimento. No contexto do *Scrum*, o papel do QA é fundamental. Durante o planejamento da *sprint*, o QA participa ativamente, colaborando na definição das tarefas e estimando suas horas de trabalho. Assim que a *sprint* é iniciada, o QA começa imediatamente a mapear os cenários de teste em uma ferramenta de gestão interna. À medida que os desenvolvedores implementam as histórias da *sprint* e sobem para o ambiente de testes, o QA entra em ação realizando os testes manuais.

Embora a equipe de desenvolvimento seja altamente talentosa e capaz de realizar todo esse processo de desenvolvimento do *software*, existe uma problemática em relação

à dependência excessiva de testes manuais. Isso pode resultar em atrasos significativos e potenciais erros humanos. Embora os testes manuais sejam essenciais para garantir a qualidade do produto, eles são propensos a inconsistências e limitações de cobertura, especialmente em projetos complexos como o *Pricing*.

Além disso, a falta de automação nos testes pode levar a uma cobertura insuficiente de casos de teste, especialmente à medida que o projeto cresce em escala e complexidade, gerando desafios significativos com testes repetitivos e regressivos manuais no sistema.

4.2 Processo de Testes de Software da Empresa

Visto e entendido o processo de *software* do projeto, agora é realizado um entendimento mais profundo sobre o processo de testes.

A Figura 9 descreve como são organizadas as atividades metodológicas, bem como as ações e tarefas que ocorrem dentro de cada atividade em um processo de teste de *software* manual. Esse processo ocorre após a conclusão do planejamento da *sprint*, ou seja, após a cerimônia de *planning*. Em outras palavras, o QA possui a documentação de requisitos, já especificada e revisada em equipe, pronta para ser utilizada.

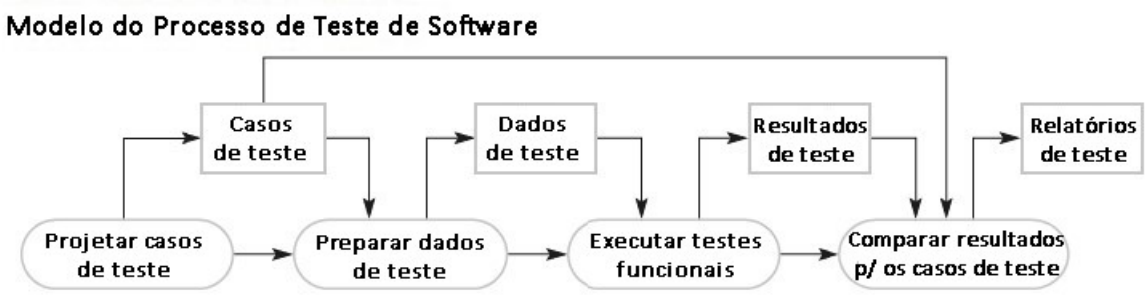

Figura 9 – Modelo de processo de teste funcional. Fonte: Adaptado de Revista Científica Multidisciplinar Núcleo do Conhecimento (SILVA, 2019).

Durante o processo de elaboração dos casos de teste, o QA da empresa utiliza o *Excel* como uma ferramenta para o mapeamento dos cenários, conforme ilustrado na Figura 10. Após o mapeamento, cada cenário é transferido para a ferramenta de gestão conhecida como *Kanban*, no formato de "*cards*".

É por meio do *Kanban* que o analista mantém o controle sobre o progresso dos testes, uma vez que o quadro possui quatro colunas para o acompanhamento dos *cards*: *backlog* - onde são colocados todos os *cards*/cenários mapeados para aquela *sprint*; em andamento - onde são posicionados os *cards* que estão sendo testados no momento; em correção - quando é detectado *bug* em um cenário, o *card* correspondente é movido para esta coluna enquanto aguarda correção pelos desenvolvedores; e, finalizado - todos os

cards que foram executados e concluídos são transferidos para esta coluna. Todas essas ações de movimentação dos *cards* dentro do *Kanban* são realizadas manualmente pelo QA, conforme exemplificado na Figura 11.

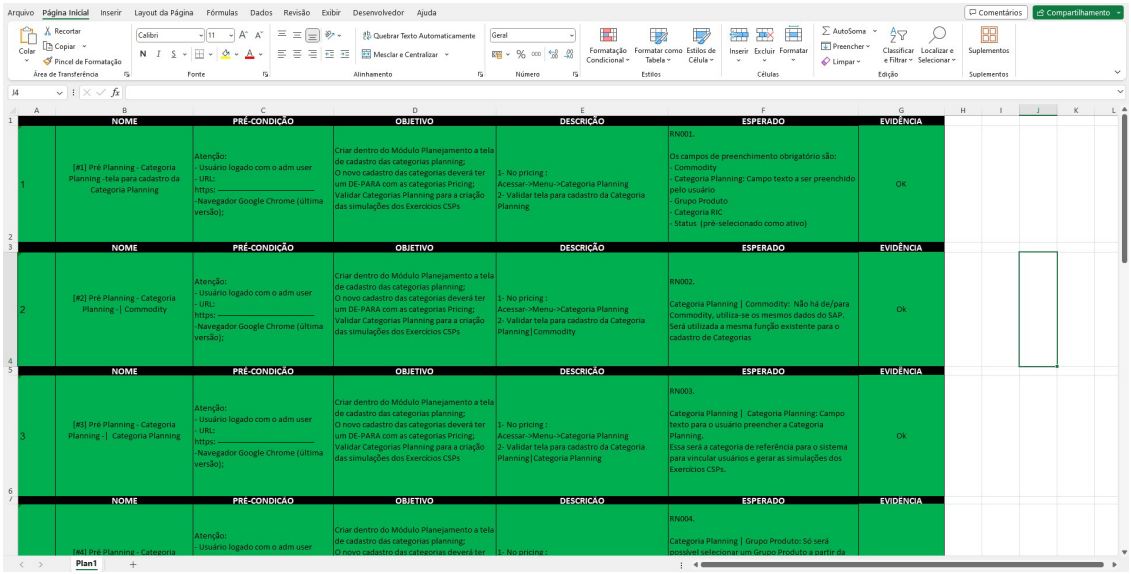

Figura 10 – Modelo de mapeamento de cenários no *Excel*. Fonte: Da Autora.

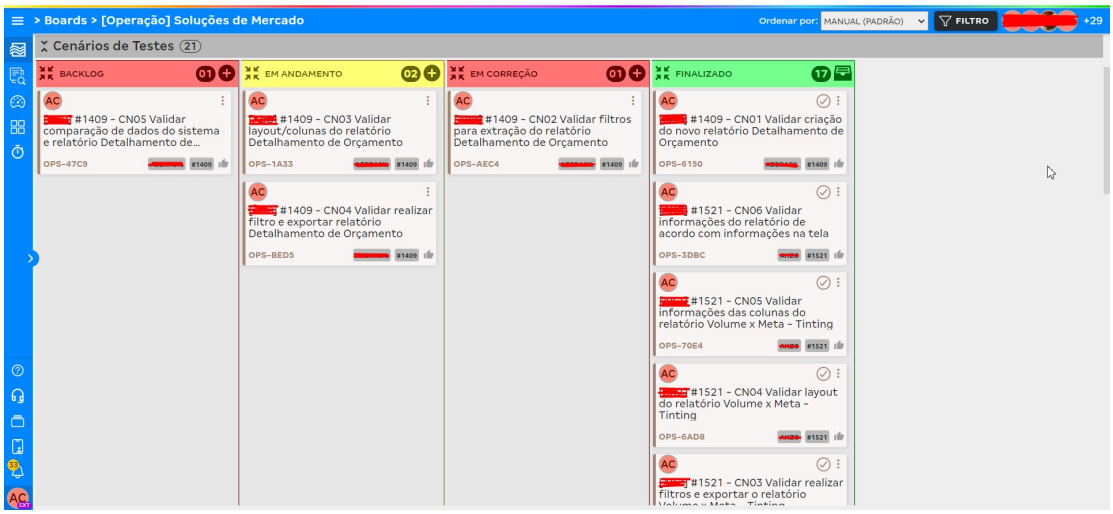

Figura 11 – Ferramenta de gestão *Kanban*. Fonte: Da Autora.

Quando os desenvolvedores disponibilizam a *sprint* para testes, significa que eles migraram todas as implementações solicitadas na *sprint* para o ambiente de testes. No contexto deste projeto, o *Pricing* contempla três ambientes distintos: o "Ambiente de QA", utilizado para os testes de garantia de qualidade na fase de validação da *sprint*, ou seja, todos os casos de testes são executados neste ambiente pelo analista de qualidade; o "Ambiente de HML", após a conclusão dos testes pelo QA no ambiente de testes, a equipe sobe a atualização para o ambiente de homologação. Este ambiente é utilizado pelo cliente

para ele validar o que foi requisitado na documentação (teste de aceitação); e o "Ambiente de Produção", após a conclusão das fases de teste no ambiente de QA e a validação do cliente no ambiente de HML, a demanda está pronta e validada para ser migrada para o ambiente de produção do cliente.

A Figura 12 ilustra precisamente a descrição anterior sobre a comunicação entre os ambientes:

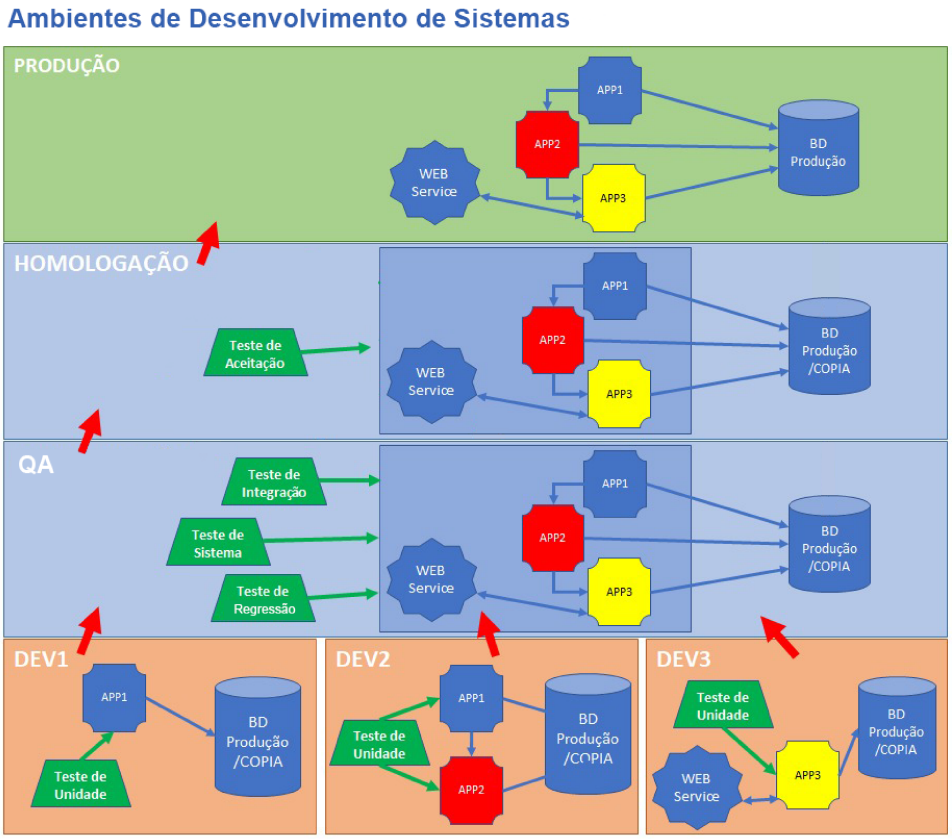

Figura 12 – Comunicação entre ambientes. Fonte: Da Autora.

Depois que o QA conclui a execução de seus cenários de testes no ambiente de QA, ele informa à equipe sobre a finalização dos testes e carrega todas as evidências de sucesso dos cenários para o *drive* da *sprint*, em uma pasta designada "Evidências QA", conforme exemplo na Figura 13. Com isso, a *sprint* é oficialmente encerrada e imediatamente se inicia o planejamento da próxima *sprint*, seguindo o mesmo procedimento.

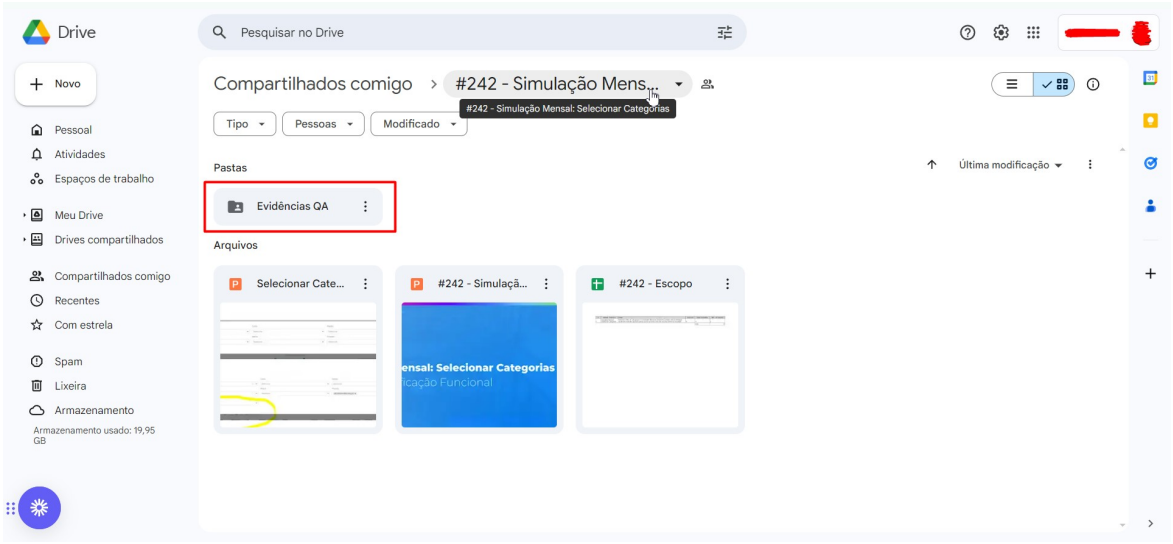

Figura 13 – Exemplo de *Drive* de *Sprint*. Fonte: Da Autora.

Por mais que o processo de testes exista, ele é extenso e muitas vezes custoso. Acaba que o projeto enfrenta desafios significativos com testes repetitivos e regressivos manuais no sistema. Devido à natureza dinâmica do desenvolvimento de *software* e às constantes atualizações neste sistema, o QA passa uma quantidade significativa de tempo manualmente testando cada funcionalidade, o que reduz a eficiência e limita sua capacidade de criar cenários improváveis. Isso resulta em atrasos nos prazos de entrega, aumento dos custos e, em última instância, insatisfação do cliente devido a *bugs* não detectados.

4.3 Proposta de Solução

Uma possível solução para o processo de testes de *software* atual da empresa, seria investir em automação de testes, permitindo a execução rápida e repetitiva de casos de teste, aumentando a cobertura e liberando recursos humanos para atividades mais estratégicas, como a identificação de novos cenários de teste e a análise de resultados. Isso não apenas reduziria os riscos associados à dependência de testes manuais, mas também melhoraria a eficiência e a confiabilidade do processo de desenvolvimento de *software*.

Além disso, a automação de testes aplicada à metodologia BDD ajuda a garantir que os testes automatizados estejam alinhados com as expectativas dos *stakeholders* e com os requisitos de negócios. Os cenários de teste escritos em BDD servem como uma forma de documentação viva do comportamento do sistema, além de serem facilmente traduzidos em testes automatizados. Resultando em testes automatizados mais legíveis, compreensíveis e mantíveis.

A Figura 14 mostra o processo de teste de *software* de maneira automatizada aplicada à metodologia BDD, para este estudo. Esse processo, assim como o manual, também

ocorre após a conclusão da cerimônia de *planning*. É possível observar que o fluxo começa com o levantamento dos cenários em BDD, alinhando-os com as expectativas de negócio. Em seguida, é elaborado o ambiente para automação, escolhendo as ferramentas: *Cypress*, *Page Objects*, *Cucumber*, *Gherkin* e linguagem *JavaScript*. A etapa seguinte envolve a implementação dos *scripts* dos cenários de testes, que incluem *features*, *elements*, *pages* e *steps* (os quais são explicados mais à frente). Uma vez desenvolvidos, os testes automatizados são executados para verificar o comportamento do sistema. Os resultados são então apresentados e comparados com os casos de testes originais. Por fim, a manutenção do código é realizada ao longo do tempo, para assegurar seu funcionamento eficiente.

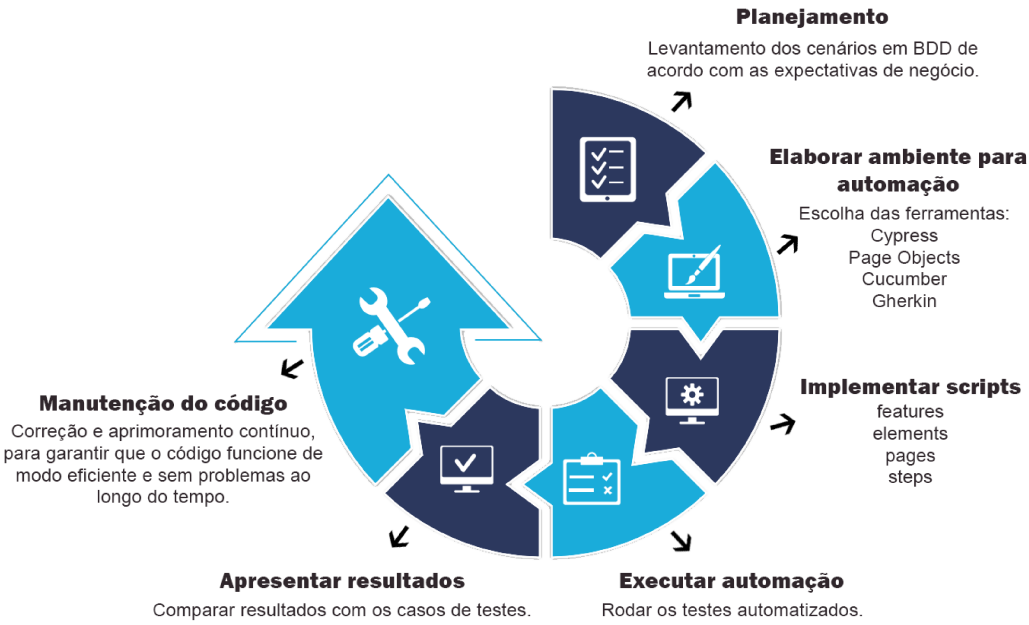

Figura 14 – Modelo de processo de teste automatizado. Fonte: Adaptado de (SLIDETEAM, 2023).

Conforme descrito na seção 4.1, o sistema selecionado para este estudo é um sistema de precificação de produtos. Dentro deste sistema, diversas interfaces gráficas estão disponíveis. A interface selecionada para a implementação de testes automatizados aplicados ao método BDD é conhecida como "Categoria *Planning*".

Esta interface consiste em um formulário destinado ao cadastramento de categorias, composto por uma tela principal de consulta e visualização de registros (Figura 15), e a subtela dedicada ao cadastro específico das categorias de planejamento (Figura 16). A implementação dos testes automatizados é realizada nas duas interfaces: consulta e cadastro.

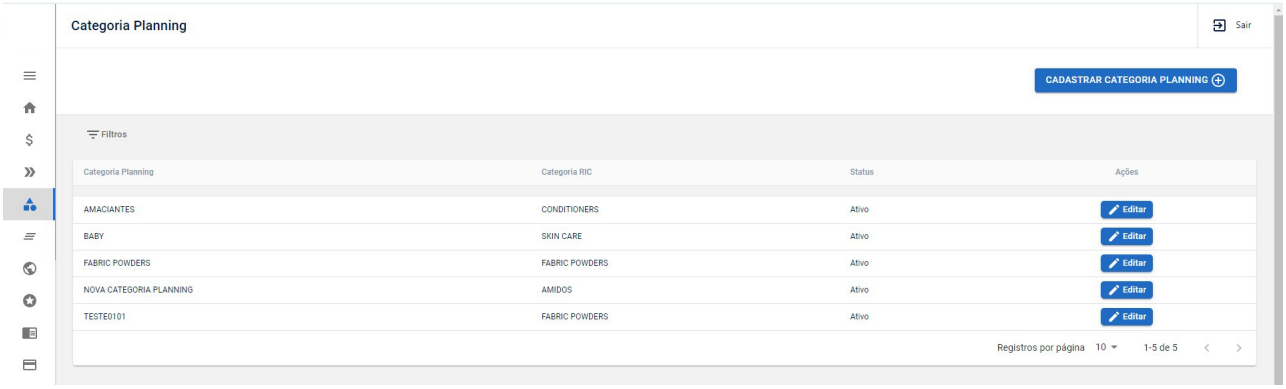

Figura 15 – Tela de consulta. Fonte: Sistema da empresa-alvo em estudo.

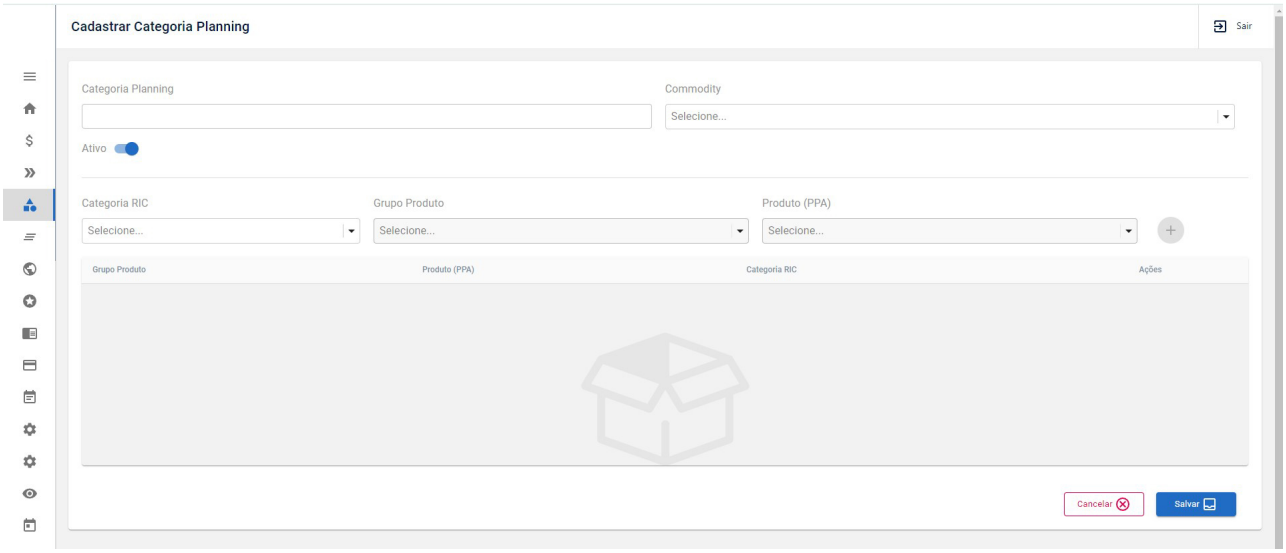

Figura 16 – Tela de cadastro. Fonte: Sistema da empresa-alvo em estudo.

4.3.1 Configurando o ambiente

A primeira coisa que se deve fazer ao iniciar um processo de automação de testes é configurar o ambiente. Para o estudo deste caso, toda a configuração e instalação de ferramentas são descritas e direcionadas para o *Windows* e o navegador *Google Chrome*. O ambiente de desenvolvimento escolhido para a implementação dos testes foi a IDE - *Integrated Development Environment*, ou Ambiente de Desenvolvimento Integrado, *Visual Studio Code*, utilizando a linguagem de programação *JavaScript*. Sendo assim, o primeiro passo é baixar a IDE.

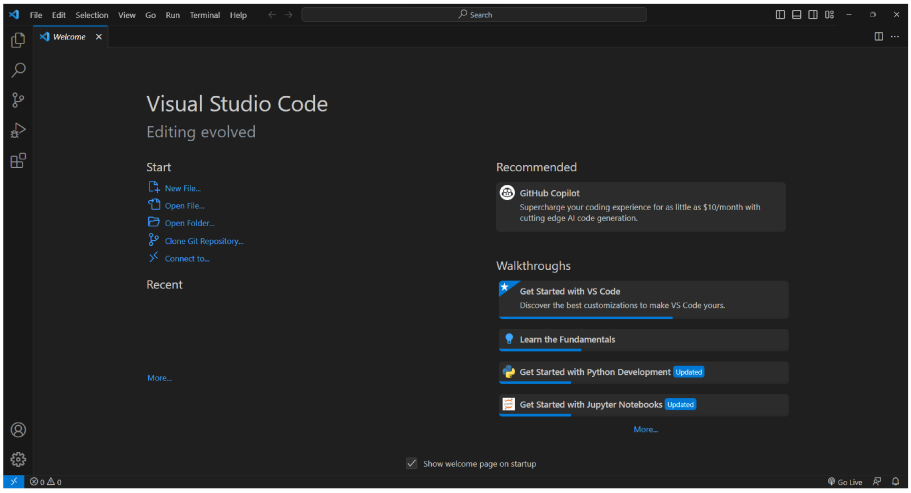

Figura 17 – IDE *Visual Studio Code*. Fonte: Extraído de Wikipédia.

O segundo passo é fazer o *download* do *Node.js*. O *Node.js* é um ambiente de tempo de execução de código aberto, construído sobre o motor *JavaScript* V8 do *Google Chrome*. De acordo com (NODE.JS, 2009) "Ele foi projetado para construir aplicativos de rede escalonáveis, ou seja, muitas conexões podem ser tratadas simultaneamente. A cada conexão, o retorno de chamada é acionado, mas se não houver trabalho a ser feito, o *Node.js* irá dormir.". Isso o torna eficiente em termos de recursos e ideal para sistemas de rede em tempo real. É necessário baixá-lo, uma vez que, para realizar testes automatizados, muitas ferramentas e *frameworks*, por exemplo, o *Cypress*, dependem do *Node.js* para serem executados.

Após finalizar os dois primeiros passos, o próximo é instalar o *Cypress* e o *Cucumber*. As instruções detalhadas sobre como realizar essa instalação estão fornecidas no Apêndice A, ao fim do trabalho.

4.3.2 Desenvolvendo os scripts de automação

Antes de iniciar a automação, o primeiro passo é identificar e compreender os cenários apropriados para testes (LOURENçO, 2022). Ou seja, mesmo ao implementar uma automação, o processo manual de mapeamento dos casos de testes continua sendo essencial, pois é por meio dele que os *scripts* de testes automatizados são elaborados. Com a diferença de que, nesse caso, os cenários já são escritos em BDD diretamente no ambiente de desenvolvimento da automação. Portanto, os cenários foram mapeados conforme a documentação do projeto *Princing*, focando especialmente na tela de Categoria *Planning*. A Figura 18 mostra como esse processo foi realizado.

Figura 18 – Processo de levantamento dos casos de teste. Fonte: Adaptado de (LOURENçO, 2022).

Depois de levantar os cenários e realizar a configuração do ambiente, o próximo passo é de fato programar os *scripts* de automação. O projeto foi organizado de acordo com a seguinte estrutura:

- *Features*: diretório que contém os arquivos de especificação de teste escritos em linguagem *Gherkin*. Cada arquivo de *feature* descreve um conjunto de cenários de teste em uma linguagem simples e legível, destacando o comportamento esperado do *software* em termos de funcionalidades e interações do usuário;
- *Elements*: diretório em que são armazenados os arquivos que definem os elementos da interface do usuário e suas respectivas interações com o *software*. Os elementos são representados como objetos, usando a abordagem de *Page Objects*, para facilitar a reutilização e manutenção do código;
- *Pages*: diretório em que são mantidos os arquivos que representam as diferentes telas da aplicação sob teste. Cada arquivo de página contém métodos e funcionalidades relacionadas a página específica, encapsulando a lógica de interação com os elementos da página e simplificando o acesso a eles em seus testes;
- *Steps*: diretório que armazena os arquivos que contêm a implementação dos passos definidos nos arquivos de *feature*. Cada passo definido em *Gherkin* é mapeado para um ou mais métodos nos arquivos de *steps*, onde a lógica de teste é executada usando os métodos e funcionalidades definidos nos arquivos *elements* e *pages*;
- *Cypress.config.js*: arquivo localizado no diretório raiz do projeto, contém as configurações específicas do *Cypress* e define a URL que está sendo testada.

O diretório *features* é a essência do BDD, pois nele são armazenados os arquivos que descrevem os cenários de teste na linguagem natural (pela sintaxe *Gherkin*) que tanto é citada neste trabalho.

Cada arquivo geralmente representa uma funcionalidade específica ou um conjunto de funcionalidades relacionadas. Cada funcionalidade é descrita em uma ou mais *features*,

e cada *feature* pode conter vários cenários de teste, representados por uma sequência de etapas (*steps*) definidas pelas palavras-chave do *Gherkin*. As Figuras 19 e 20 exemplificam como foi implementado na prática.

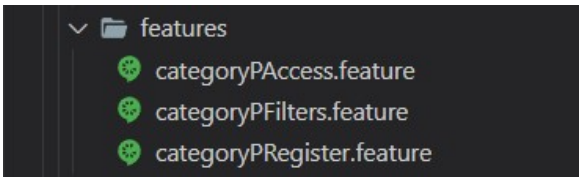

Figura 19 – *Features*. Fonte: Da autora.

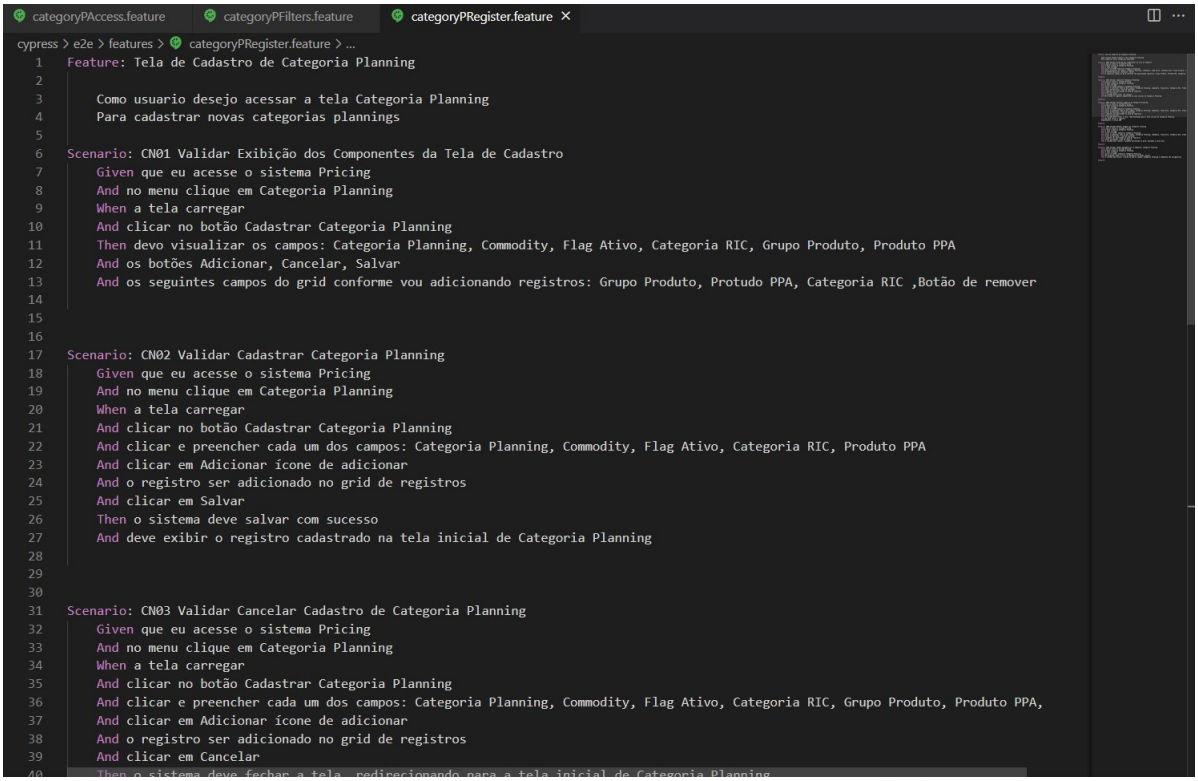

Figura 20 – Arquivo *categoryPRegister.feature*. Fonte: Da autora.

Foram criados três arquivos *.feature*: um para acessar a tela Categoria *Planning*, outro para validar os filtros e o terceiro para cadastrar categorias na tela de cadastro. É perceptível que as palavras-chave estão em inglês, enquanto a descrição dos passos está em português, porém essa diferença não apresenta dificuldades, sendo apenas uma questão de conveniência.

Ao organizar os cenários de teste em arquivos de *feature*, a equipe de desenvolvimento pode ter uma visão clara e estruturada dos requisitos e comportamentos esperados do sistema, facilitando a comunicação e colaboração entre os membros da equipe e garantindo uma cobertura abrangente dos testes.

Como dito anteriormente, o diretório *elements* é onde são armazenados os arquivos que definem os elementos da interface do usuário (UI) que serão interagidos nos testes automatizados. Esses elementos podem incluir botões, campos de texto, caixas de seleção, *links* e outros componentes de UI importantes para os testes. A estratégia utilizada para capturar os localizadores é dando preferência ao ID (identificador de dispositivo), visto que ele é um elemento único no código (LOURENçO, 2022).

Cada arquivo no diretório *elements* corresponde geralmente a uma página ou tela da aplicação, e dentro de cada arquivo estão definidos os elementos específicos daquela página. Para facilitar ainda mais a organização e reutilização, foi utilizado o padrão *Page Object*. Nesse padrão, cada página da aplicação é representada por uma classe, e os elementos daquela página são encapsulados como propriedades dessa classe. Isso permite que os elementos sejam facilmente acessados e manipulados nos testes, sem a necessidade de repetir o código em vários lugares. A Figura 21 mostra os arquivos dentro da pasta *elements* criada, enquanto a Figura 22 exemplifica os elementos da página de cadastro.

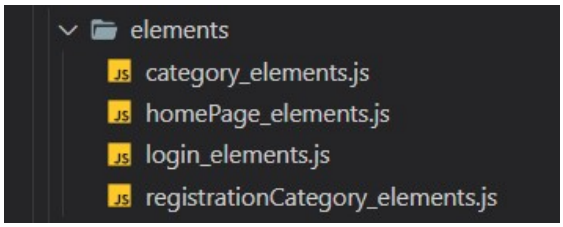

Figura 21 – *Elements*. Fonte: Da autora.

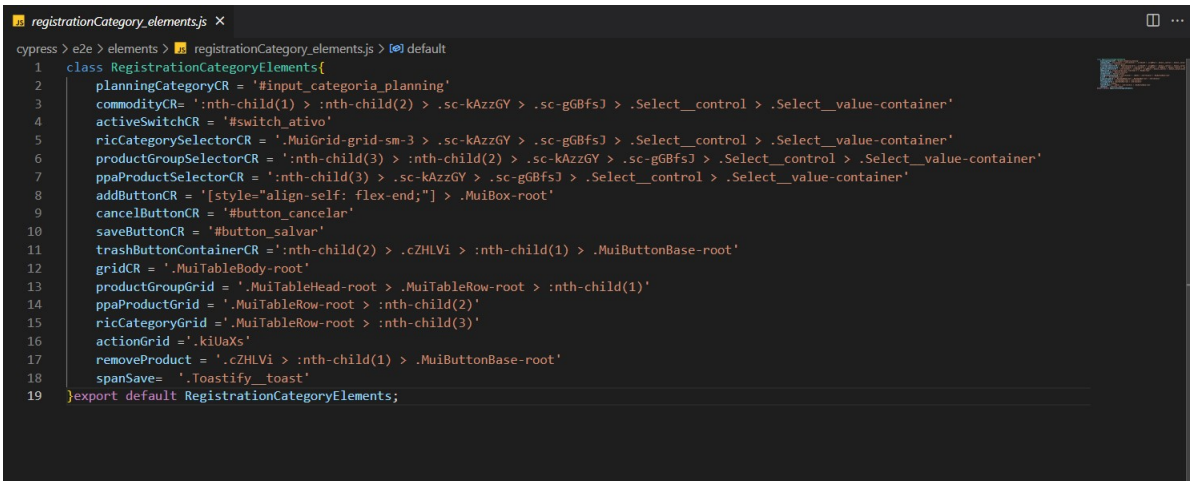

Figura 22 – Arquivo *registrationCategory_elements.js*. Fonte: Da autora.

Este arquivo define uma classe chamada *RegistrationCategoryElements*, que contém várias propriedades que representam elementos da interface de cadastro da Categoria *Planning*. Cada propriedade está associada a um seletor CSS (*Cascading Style Sheet* ou

ou Folha de Estilo em Cascatas) que identifica um elemento específico na página. Por exemplo:

- *planningCategoryCR* representa o elemento associado ao seletor CSS *#input_categoria_planning*;
- *commodityCR* representa o elemento associado ao seletor CSS *:nth-child(1) > :nth-child(2) > .sc-kAzzGY > .sc-gGBfsJ > .Select___control > .Select___value-container*;
- E assim por diante para os outros elementos listados.

Como nem todos os elementos possuiam ID, esses seletores CSS foram utilizados para localizar e interagir com os elementos em seu lugar. Por exemplo, ao preencher um campo de formulário ou clicar em um botão, o teste pode utilizar o seletor CSS definido neste arquivo para encontrar o elemento correto na página.

No entanto, uma observação digna de um QA neste quesito de seletores é: quando os elementos da interface do usuário são identificados por seletores CSS em vez de IDs, pode causar alguns problemas potenciais no futuro, como:

- Os seletores CSS podem ser mais propensos a quebras durante as atualizações da interface do usuário, pois são mais suscetíveis a alterações na estrutura ou estilo da página. Se os elementos forem movidos, renomeados ou alterados de alguma forma, os seletores CSS podem precisar ser atualizados manualmente nos arquivos de elementos, o que pode tornar a manutenção mais trabalhosa e propensa a erros;
- Em comparação com os IDs, os seletores CSS podem ser menos eficientes em termos de desempenho ao localizar elementos na página, especialmente em páginas com muitos elementos ou em navegadores mais antigos.

Enfim, embora seja possível e comum usar seletores CSS para identificar elementos da UI em testes automatizados, é importante estar ciente dos possíveis problemas e adotar práticas de desenvolvimento que minimizem esses riscos. Uma possível solução é alinhar com os desenvolvedores, desde o início do ciclo de desenvolvimento, a utilização de IDs em todos os elementos da página, sempre que possível. Caso isso não seja factível, ao menos priorizar o uso de seletores mais específicos e realizar revisões periódicas para garantir a precisão e a manutenibilidade dos seletores conforme necessário. Essas práticas contribuem para uma automação de teste mais robusta e resiliente às mudanças na interface do usuário.

Agora, abordando o diretório *pages*. Este pode ser considerado um dos diretórios mais abrangente e trabalhoso, dada a estrutura selecionada para essa automação. Aqui é

onde todas as funções e métodos são definidos para executar ações em cada página, tais como: preencher campos, clicar em botões, verificar mensagens de erro, entre outros.

O objetivo do *pages* é fornecer uma estrutura organizada para os *Page Objects*, de modo que cada página da aplicação tenha sua própria representação em um arquivo separado. Isso facilita a manutenção e a escalabilidade dos testes, ao permitir isolar as interações e verificações específicas de cada página.

Para este projeto foram criados quatro arquivos *pages.js*, como é possível visualizar na Figura 23. Cada arquivo criado no diretório *pages* está relacionado a um arquivo *feature* e *elements* correspondente. Em seguida, no arquivo de passos (*steps*), esses métodos das *pages* são chamados para interagir com a aplicação durante a execução dos testes. A Figura 24 mostra um pouco do arquivo *registrationCategory_page.js* que se refere a página de cadastro do sistema.

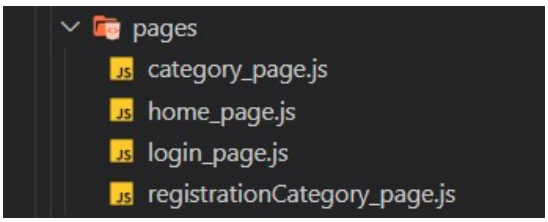

Figura 23 – *Pages*. Fonte: Da autora.

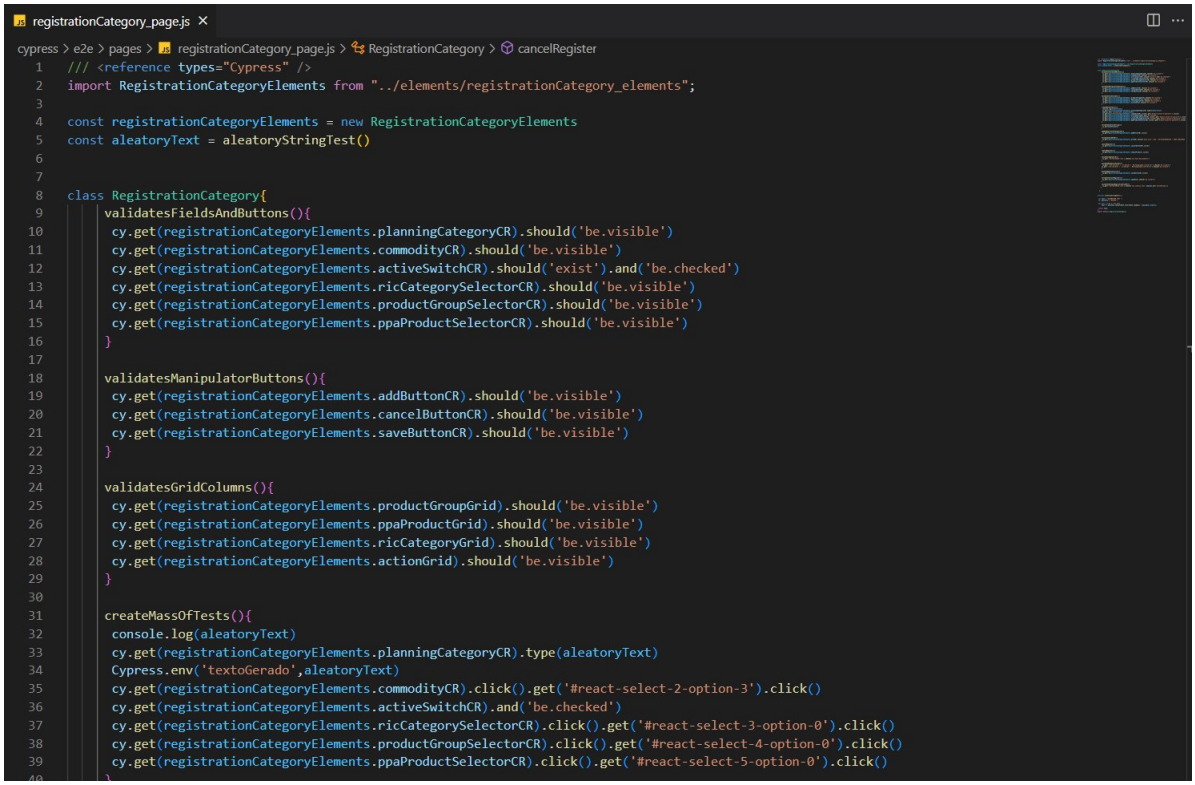

Figura 24 – Arquivo *registrationCategory_page.js*. Fonte: Da autora.

Embora a Figura 24 não mostre todas as funções criadas, abaixo é detalhadamente explicado um exemplo do que foi realizado para esta página específica e suas descrições. Isso evita a necessidade de explicar as outras três *pages* também, uma vez que o modelo segue o mesmo padrão de criação de funções, porém cada uma específica para a tela referenciada.

1. *validatesFieldsAndButtons()*: função que valida a visibilidade dos campos e botões na página de cadastro de categorias;
2. *validatesManipulatorButtons()*: função que valida a visibilidade dos botões de manipulação (adicionar, cancelar, salvar) na página;
3. *validatesGridColumns()*: função que valida a visibilidade das colunas na grade de dados da página;
4. *createMassOfTests()*: função que gera um texto aleatório (nome do registro que pretendo cadastrar) e preenche os outros campos necessários para realizar o cadastro de categorias;
5. *validateAleatoryString()*: função que valida a presença do texto aleatório gerado no momento de cadastrar um novo registro;
6. *addNewRegisterOfCategoryP()*: função que clica no botão para adicionar um novo registro de categoria;
7. *validatesGridOfCR()*: função que valida se um certo texto está presente na grade de dados;
8. *cancelRegister()*: função que clica no botão para cancelar o registro em andamento;
9. *removeRegister()*: função que clica no botão para remover um registro;
10. *validatesEmptyGrid()*: função que valida se a grade de dados está vazia;
11. *validateMandatoryFields()*: função que valida a visibilidade dos campos obrigatórios na página;
12. *btnSaveRegisterClick()*: função que clica no botão para salvar o registro em andamento;
13. *validatesSucessRegister()*: função que valida a visibilidade de um elemento indicando que o registro foi salvo com sucesso;
14. *validatesNotSaveRegisterInGridCP()*: função que valida se o texto gerado aleatoriamente não está presente na grade de dados, para o caso do cenário em que é clicado em 'Cancelar', ao invés de 'Salvar'.

Além das funções da classe *RegistrationCategory*, o código também inclui uma função externa chamada *aleatoryStringTest()*, que gera um texto aleatório para ser usado nos testes (Figura 25). Esta função desempenha um papel crucial nos testes, pois é responsável por cadastrar um novo registro, inserindo um nome aleatório toda vez que a automação é executada. Essa aleatoriedade do nome é necessária, porque o sistema não permite o cadastro de registros com nomes idênticos.

```
function aleatoryStringTest() {

 var text = "AUTOMATION TEST ";
 var possible = "012345";

 for (var i = 0; i < 4; i++)
     text += possible.charAt(Math.floor(Math.random() * possible.length));

 return text;
}
export default RegistrationCategory;
```

Figura 25 – Função *aleatoryStringTest()*. Fonte: Da autora.

A função *aleatoryStringTest()* retorna um texto aleatório composto pela *string* inicial *AUTOMATION TEST* seguida por quatro caracteres aleatórios selecionados da *string possible*, sendo chamada na função de cadastro do registro: *createMassOfTests()*, explicada anteriormente e que pode ser vista na Figura 24.

Após o cadastro, é realizada uma validação na grade de dados para verificar se o texto gerado aleatoriamente está presente. Essa validação é feita pela função *validatesRegisterInGridCP()*, localizada no arquivo *pages*: *category_page.js*, referente aos componentes da tela inicial da Categoria *Planning*, exibida na Figura 26. Ela busca na grade de dados por esse texto específico e verifica se ele foi cadastrado de fato.

```
registrationCategory_page.js   category_page.js ×
cypress > e2e > pages > category_page.js > CategoryPlanningPage
 6   class CategoryPlanningPage{
38       validatesRegisterInGridCP(){
39           cy.get('.MuiTableBody-root').should('contain.text',Cypress.env('textoGerado'))
40       }
41
```

Figura 26 – Função *validatesRegisterInGridCP()*. Fonte: Da autora.

Sobre o diretório *steps*, como mencionado anteriormente, é nele que os métodos das *pages* são invocados para interagir com a aplicação durante a execução dos testes, seguindo a lógica definida nos arquivos de *feature*. Foram criados três arquivos de *steps* correspondentes aos três arquivos *features*, exibidos na Figura 27.

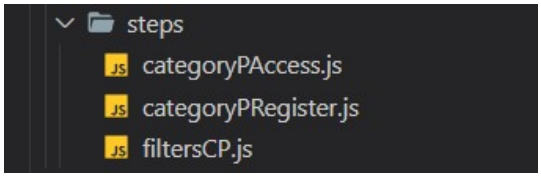

Figura 27 – *Steps*. Fonte: Da autora.

Basicamente, o arquivo de passos copia os passos criados nas *features* em sintaxe *Gherkin*, e acrescenta as referências às funções criadas no arquivo *page* correspondente. Ou seja, se na *feature* o cenário foi escrito assim:

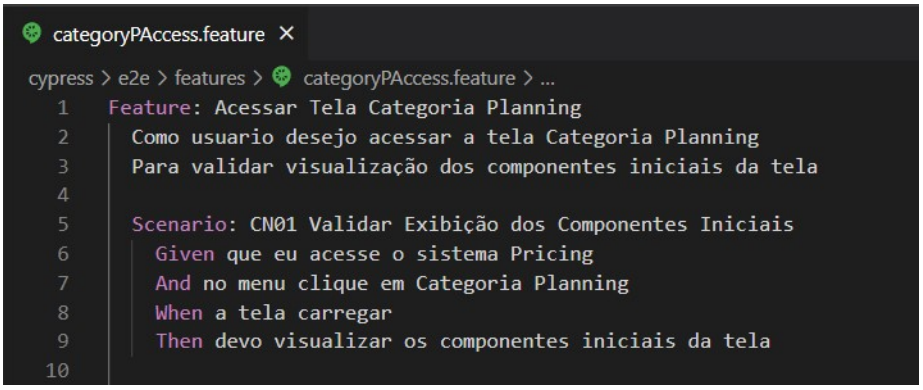

Figura 28 – Arquivo *categoryPAccess.feature*. Fonte: Da autora.

Agora, ele é implementado dessa forma:

Figura 29 – Arquivo *steps*: *categoryPAccess.js*. Fonte: Da autora.

No início do arquivo, há os *imports* das classes que são utilizadas, representando às páginas da aplicação (*LoginPage, HomePage, CategoryPlanningPage*). Em seguida, a descrição do cenário de teste definida no arquivo *feature*. Cada passo do cenário é implementado em *JavaScript*.

Os arquivos *steps* demonstram claramente a união entre o *Cucumber* (sintaxe *Gherkin*) e o padrão *Page Objects* (na linguagem *JavaScript*), deixando a automação dos testes mais modular, legível, fácil de manter e reutilizar os componentes em diferentes partes da aplicação.

Por fim, a Figura 30 mostra a configuração do arquivo *cypress.config.js*, cujo objetivo é configurar o ambiente de teste para usar o *plugin cypress-cucumber-preprocessor*, sendo quem permite escrever os testes usando a sintaxe do *Gherkin* em arquivos *.feature* e executá-los no *Cypress* (a implementação deste *plugin* foi exemplificada no Apêndice A ao fim do trabalho). Além de definir a URL base que é usada para acessar a aplicação durante os testes.

```
cypress.config.js
cypress.config.js > ...
1  const { defineConfig } = require("cypress");
2  const cucumber = require("cypress-cucumber-preprocessor").default
3
4  module.exports = defineConfig({
5    e2e: {
6
7      baseUrl: "_____",
8      specPattern: "**/*.feature",
9      setupNodeEvents(on, config) {on("file:preprocessor",cucumber());
10     },
11   },
12 });
13
```

Figura 30 – Arquivo *cypress.config.js*. Fonte: Da autora.

Em relação às métricas de código, o projeto envolveu a criação de 3 arquivos *features*, abrangendo um total de 9 cenários de teste, 4 arquivos *elements*, nos quais foram inspecionados ao todo 34 elementos, 4 arquivos *pages* com 1 classe em cada, contendo 30 funções em seu total, sendo uma delas uma função externa de gerador de texto aleatório. E, 3 arquivos *steps*, correspondendo aos 9 cenários das *features*, nos quais as funções das *pages* são chamadas.

4.4 Execução dos testes

Após concluir o desenvolvimento dos *scripts* de automação, é necessário executar o comando *cypress.open* que pode ser realizado através do botão *run*, localizado em *NPM Scripts*, como mostra na Figura 31. Depois, é escolhido o tipo do teste, que no caso é o *E2E Testing*, e o navegador, que no caso é o *Google Chrome* (Figuras 32 e 33).

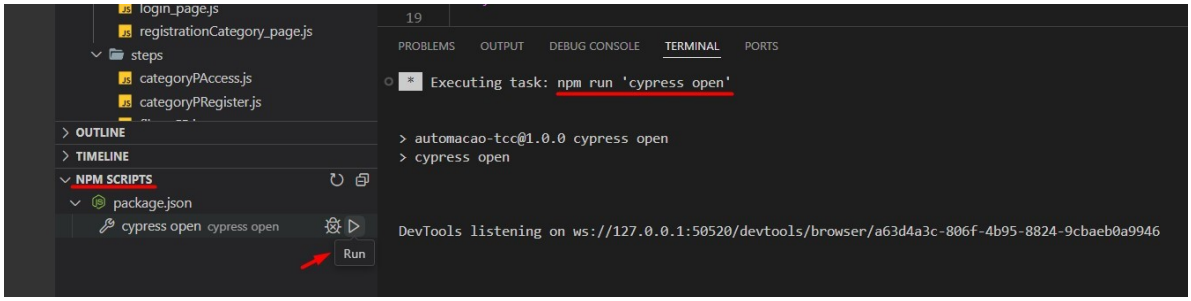

Figura 31 – Abrindo o *Cypress*. Fonte: Da autora.

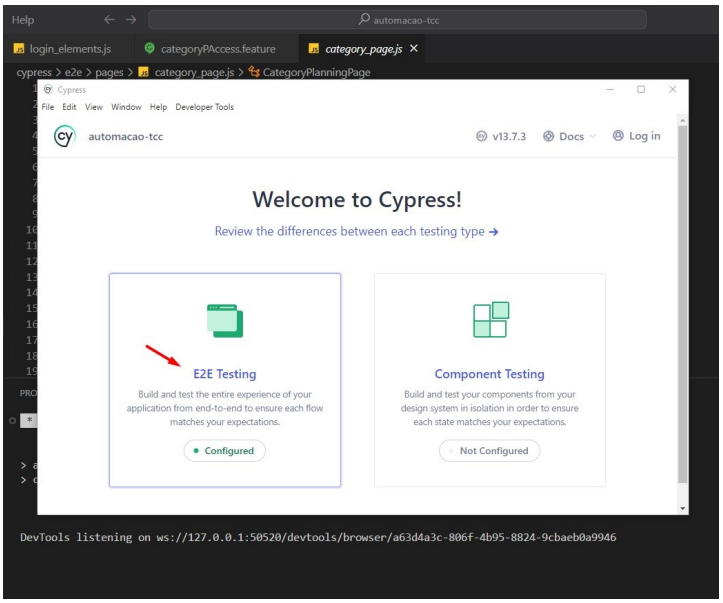

Figura 32 – *E2E Testing*. Fonte: Da autora.

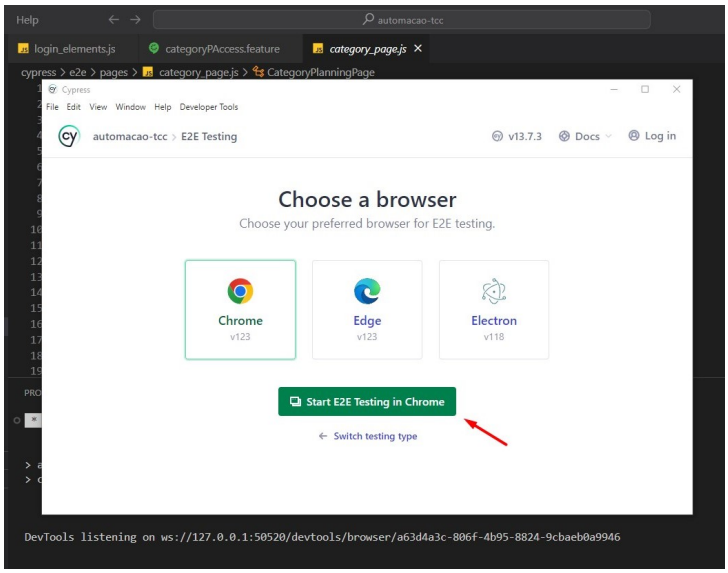

Figura 33 – Navegador *Google Chrome*. Fonte: Da autora.

Em seguida, o *Cypress* exibe as três *features*: *categoryPAccess*, *categoryPFilters* e *categoryPRegister* implementadas, onde ambas rodaram com sucesso e podem ser visualizadas em partes, nas Figuras 34, 35 e 36.

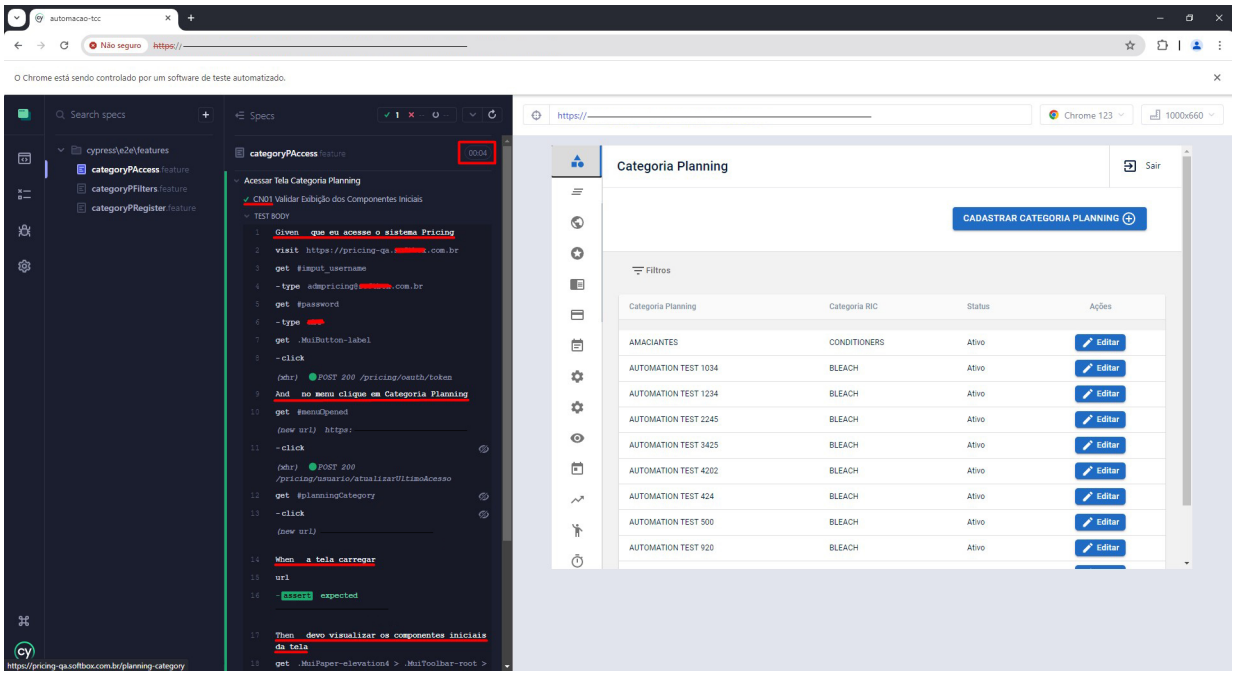

Figura 34 – *categoryPAccess*. Fonte: Da autora.

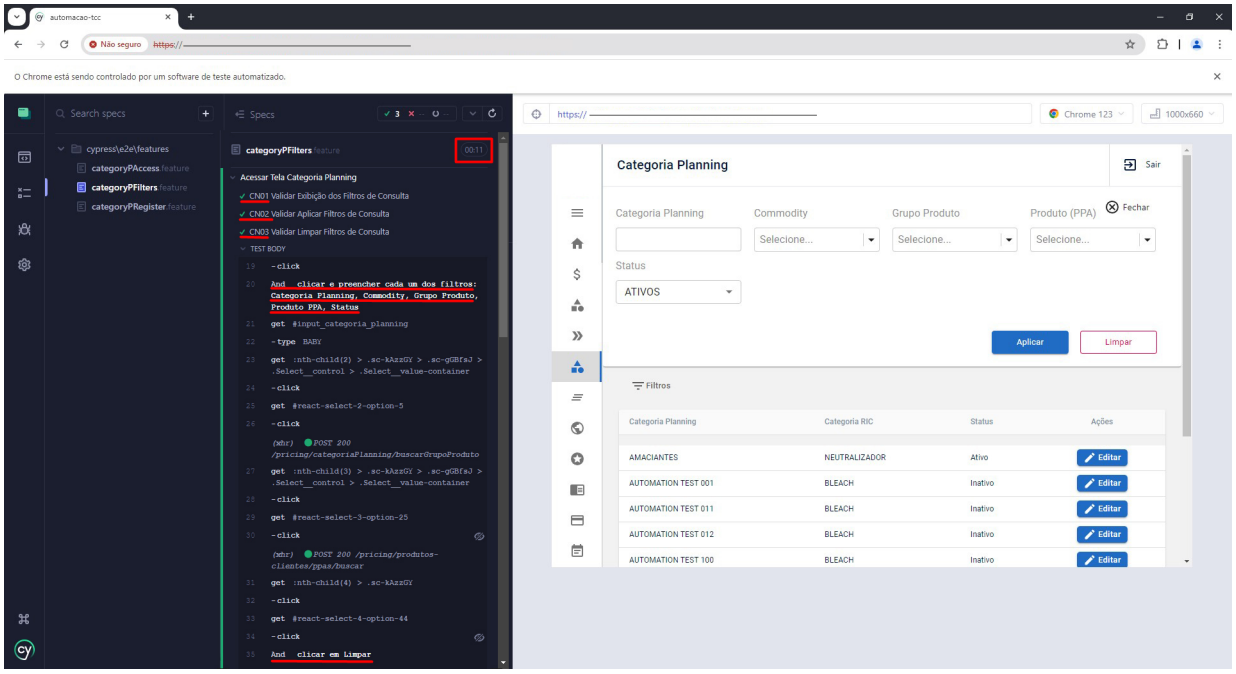

Figura 35 – *categoryPFilters*. Fonte: Da autora.

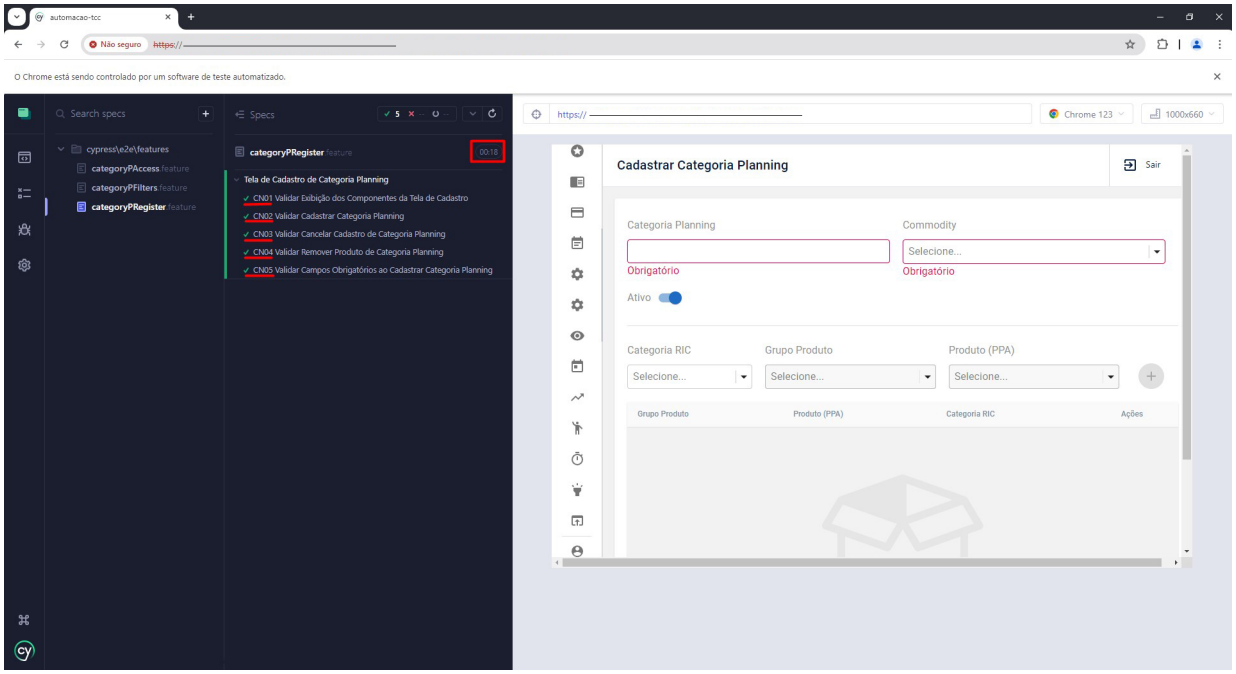

Figura 36 – *categoryPRegister*. Fonte: Da autora.

Um aspecto nítido e fundamental que se destacou foi o tempo de execução, calculado pelo próprio *Cypress* (como evidenciado nas figuras acima, indicado ao lado do nome da *feature*). A primeira *feature* (*categoryPAccess*) foi concluída em 4 segundos, a segunda (*categoryPFilters*) em 11 segundos e a terceira (*categoryPRegister*) em 18 segundos, totalizando 33 segundos. Embora o número de cenários testados seja relativamente baixo, esses dados já indicam a eficiência e rapidez da automação.

Além disso, visualizar a execução dos cenários de testes no *Cypress* utilizando o BDD, proporcionou uma clareza e organização notáveis em cada *step* do cenário, chegando a assemelhar uma forma de documentação em tempo real. Sendo assim, a execução dos testes automatizados foram realizados com vigor e os resultados são tratados na próxima seção.

4.5 Resultados

Tendo em vista que a interface Categoria *Planning* passou inicialmente por testes manuais, e durante este trabalho, foram implementados os testes automatizados para esta mesma interface, é realizado uma comparação entre os resultados obtidos pelas duas abordagens.

Considerando que os 9 cenários foram mapeados para ambos os formatos:

	Testes Manuais	**Testes Automatizados**
Mapeamento dos Cenários	1 hora	1 hora
Desenvolvendo a Automação	N/A	4 horas
Execução dos Testes	6 horas	33 segundos
Bugs Encontrados	2 erros	2 erros
Execução dos Retestes	3 horas	33 segundos
Tempo Total Gasto	**10 horas**	**5 horas, 1 minuto e 6 segundos**

Tabela 1 – Comparação dos dados de execução entre testes manuais e automatizados. Elaborado pela autora.

Realizando uma análise da comparação acima, entre a execução manual e automatizada de testes, observam-se importantes *insights* sobre eficiência, precisão e economia de tempo. Na abordagem manual, o mapeamento dos cenários consumiu uma hora, seguido por uma execução de testes que durou seis horas. Durante esse processo, foram identificados dois *bugs*, resultando em três horas adicionais de retrabalho para os retestes. O tempo total gasto em testes manuais foi de 10 horas.

Por outro lado, a execução automatizada apresentou o tempo de mapeamento de cenários também de uma hora. O desenvolvimento da automação demandou quatro horas, um investimento inicial de tempo. Entretanto, uma vez implementada, a execução dos testes automatizados foi notavelmente rápida, levando apenas 33 segundos. Curiosamente, os mesmos dois *bugs* foram encontrados, indicando consistência nos resultados entre as abordagens. O tempo total gasto em testes automatizados foi de 5 horas, 1 minuto e 6 segundos.

Ao comparar esses dados, pode-se entender que os testes automatizados foram significativamente mais eficientes em termos de tempo do que os testes manuais. Enquanto os testes manuais consumiram um total de 10 horas, os testes automatizados levaram apenas 5 horas, 1 minuto e 6 segundos para serem concluídos. Isso sugere que a automação dos testes não apenas reduziu pela metade o tempo necessário para realizar os testes, mas também aumentou a eficiência geral do processo de teste.

Uma das maiores vantagens da automação também ficou evidente nos retestes, onde não foi necessário um esforço adicional de execução manual. Ao invés disso, repetir os testes automatizados levou o mesmo tempo de 33 segundos, resultando em economia significativa de tempo e recursos.

Essa comparação demonstra que, embora a automação inicialmente exija um investimento de tempo para desenvolvimento, ela oferece benefícios consideráveis em termos de velocidade, precisão e eficiência nos ciclos de teste, além de reduzir significativamente o tempo necessário para realizar retestes.

Além disso, a automação dos testes junto ao BDD oferece uma série de vantagens adicionais. Ela possibilita a identificação rápida de regressões e problemas de integração, permitindo que as equipes de desenvolvimento ajam proativamente para corrigi-los antes que se tornem maiores obstáculos. A reutilização de *scripts* de teste para cenários similares é outro ponto crucial, pois não apenas economiza tempo na criação de novos testes, mas também garante consistência e confiabilidade nos resultados.

Outro benefício particularmente importante é a integração contínua, a qual pode ser relacionada intimamente com os aspectos fundamentais da cultura *DevOps*. Onde é possível configurar um gatilho (*trigger*) no *Cypress*, para os testes automatizados serem executados automaticamente sempre que houver uma atualização no código-fonte, como ao fazer um *push* na esteira de integração. Isso permite que a equipe de desenvolvimento identifique rapidamente quaisquer problemas ou regressões introduzidas pelo novo código, possibilitando correções imediatas antes que afetem o ambiente. Essa prática de integração contínua não só agiliza o processo de desenvolvimento, mas também ajuda a garantir a qualidade do *software* em todas as etapas do ciclo de vida do projeto.

5 Conclusão

Após uma análise aprofundada sobre a implantação de um processo de testes de *software* utilizando *Behavior Driven Development (BDD)*, fica claro que esta abordagem se mostra promissora para aprimorar a eficiência, precisão e economia de tempo no ciclo de desenvolvimento de *software*. Este estudo examinou os fundamentos do BDD, como eles puderam ser usados em processos de testes automatizados e como isso afetou o processo de desenvolvimento e teste. Inicialmente, foram apresentados os objetivos do trabalho, que incluíram apresentar as ideias e analisar criticamente o processo de desenvolvimento de *software* da empresa-alvo. Nesse contexto, foram identificados desafios significativos, como a dependência excessiva de testes manuais e os riscos associados a ela, especialmente em projetos complexos.

Como resposta a esses problemas, sugeriu-se a implantação de um modelo que integrou automação com a metodologia BDD. O objetivo era superar as limitações dos testes manuais e otimizar o processo de teste de *software*. Durante a implementação desse modelo, foi enfrentada a dificuldade de ajustar os processos para garantir que os testes fossem executados automaticamente. Contudo, os resultados obtidos foram notáveis.

Observou-se uma redução significativa no tempo de execução dos testes, mantendo a consistência na identificação de *bugs* entre as abordagens automatizadas e manuais. Destacou-se principalmente a economia de tempo nos retestes, onde a execução automatizada se mostrou extremamente eficaz, reforçando os benefícios da automação no processo de teste de *software*. Também é importante ressaltar que os testes automatizados não substituem os testes manuais por completo, uma vez que é inviável automatizar completamente todas as funcionalidades de um sistema.

Para trabalhos futuros, sugere-se explorar a cultura *DevOps* como uma extensão natural deste estudo, incorporando ferramentas e práticas *DevOps* para aprimorar ainda mais a automação e integração contínua no ciclo de desenvolvimento. Em resumo, este estudo estabeleceu um fundamento robusto para pesquisas e implementações subsequentes na área de testes de *software*, evidenciando o potencial transformador da combinação entre a abordagem BDD e a automação.

Referências

ALBIERO, F. W. Monografia de Pós-Graduação em Computação, *Uma abordagem de teste para aplicativos Android utilizando os cenários do Behavior Driven Development*. 2017. Citado na página 13.

AMBROSIO, B.; FARIA, C. C. Monografia de Graduação em Sistemas de Informação, *Proposição de uma abordagem para desenvolvimento de Software utilizando fatores do arcabouço Scrum e das práticas DevOps*. 2021. Citado 3 vezes nas páginas 15, 16 e 30.

ANDERLE, A. Monografia de Especialização em Qualidade de Software, *Introdução de BDD (Behavior Driven Development) como Melhoria de Processo no Desenvolvimento Ágil de Software*. 2015. Citado 2 vezes nas páginas 17 e 23.

BADGETT, T. et al. *The Art of Software Testing*. [S.l.]: John Wiley and Sons, Inc., 2011. 255 p. Citado na página 19.

BECK, K.; CHEIRAN, J. F. P. *TDD desenvolvimento guiado por testes*. [S.l.]: Bookman, 2010. 241 p. ISBN 9788577807475. Citado 2 vezes nas páginas 10 e 11.

BESTDEVOPS. *Scrum Ops*. 2017. Disponível em: <https://www.bestdevops.com/organizations-put-the-scrum-back-into-devops/>. Citado 2 vezes nas páginas 4 e 30.

BRAGA, F. A. M. Dissertação de Mestrado em Ciência da Computação, *Um panorama sobre o uso de práticas DevOps nas indústrias de software*. 2015. Citado 2 vezes nas páginas 29 e 30.

CARVALHO, D. M. D.; MARQUES, D. Proposta de uso da técnica bdd para otimizar a escrita e automação de testes no framework scrum. 2019. Citado 3 vezes nas páginas 13, 21 e 27.

CARVALHO, M. M. T. de. Monografia de Graduação em Sistemas de Informação, *Aplicação de Teste de Regressão na Qualidade de Software usando Postman e Newman*. 2022. Citado na página 33.

CHAVES, W. Desmistificando o uso do gherkin. 2021. Disponível em: <https://medium.com/revista-dtar/desmistificando-o-uso-do-gherkin-d1e56c592b80>. Citado na página 22.

CHIAVEGATTO, R. B. et al. Desenvolvimento orientado a comportamento com testes automatizados utilizando jbehave e selenium. Centro de Pós-Graduação e Extensão (CPGE) Faculdade FUCAPI, 2013. ISSN 2238-5096. Citado na página 24.

CHICANELLI, R. et al. *Aspectos sociais, humanos e econômicos da utilização de testes automatizados no desenvolvimento de sistemas*. 2019. Citado 2 vezes nas páginas 24 e 31.

CLAUDIO, A.; NETO, D. Introdução a teste de software. 2017. Disponível em: <http://www.projectcartoon.com/cartoon/611>. Citado 3 vezes nas páginas 10, 18 e 19.

CYPRESS.IO. *Cypress*. Acesso em abril de 2024. Disponível em: <https://docs.cypress.io/guides/overview/why-cypress>. Citado 4 vezes nas páginas 5, 32, 33 e 66.

FERREIRA, A. C. da S. et al. *Selenium, Robot e Cypress: Um estudo comparativo entre ferramentas de Automação de Teste*. 2022. Citado 2 vezes nas páginas 20 e 32.

FIGUEIREDO, R. M. de C. T. et al. Coordenadoria de Análise e Desenvolvimento de Sistemas Instituto Federal de Educação, Ciência e Tecnologia de Sergipe (IFS), *Um estudo comparativo de características das ferramentas de automação de teste end-to-end: Cypress vs QA Wolf vs TestCafé*. 2022. Citado na página 19.

IRSHAD, M.; BRITTO, R.; PETERSEN, K. Adapting behavior driven development (bdd) for large-scale software systems. *Journal of Systems and Software*, Elsevier Inc., v. 177, 7 2021. ISSN 01641212. Citado na página 25.

JENA, S.; MOH, C. Scrumops in software engineering. Geeks for Geeks, 2023. Disponível em: <https://www.geeksforgeeks.org/scrumops-in-software-engineering/>. Citado na página 30.

LOURENçO, R. de S. Monografia de Graduação em Engenharia de Computação, *Automação de Testes para um Sistema de E-commerce*. 2022. Citado 9 vezes nas páginas 4, 11, 13, 20, 26, 33, 45, 46 e 48.

MARINHEIRO, V. *Cypress + Page Objects = Sucesso*. 2020. Disponível em: <https://vitormarinheiroautomation.medium.com/cypress-page-object-sucesso-6841cb7c19a0>. Citado na página 35.

MEDIUM.COM. *BDD*. 2018. Disponível em: <https://medium.com/desenvolvimento-orientado-por-comportamento/desenvolvimento-conduzido-de-comportamento-bdd-b2c98daea331>. Citado 2 vezes nas páginas 4 e 23.

NASCIMENTO, F. Page object: o que é? Alura, 2021. Disponível em: <https://www.alura.com.br/artigos/page-object-o-que-e>. Citado 2 vezes nas páginas 34 e 35.

NODE.JS. *Sobre Node.js®*. 2009. Disponível em: <https://nodejs.org/en/about>. Citado na página 45.

NORTH, D.; LTD, A. *Introducing BDD - Dan North and Associates Ltd*. 2020. Disponível em: <https://dannorth.net/introducing-bdd/>. Citado 2 vezes nas páginas 11 e 21.

OLSEN, M. P. K.; ULRICH, S. *Certified Tester Foundation Level (CTFL) Syllabus*. [S.l.]: ISTQB - International Software Testing Qualifications Board, 2018. 93 p. Citado 2 vezes nas páginas 18 e 20.

PAULA, W. H. D. Monografia de Graduação em Engenharia de Controle e Automação, *Qualidade de Software e desenvolvimento dirigido por comportamento - BDD: Um estudo de caso*. 2019. Citado 2 vezes nas páginas 13 e 24.

PERERA, P.; SILVA, R.; PERERA, I. Improve software quality through practicing devops. International Conference on Advances in ICT for Emerging Regions (ICTer), 2017. Citado 2 vezes nas páginas 15 e 28.

PHAM, A.; PHAM, P.-V. *Scrum em Ação: Gerenciamento e Desenvolvimento Ágil de Projetos de Software*. Novatec Editora, 2011. ISBN 9788575222850. Disponível em: <https://s3.novatec.com.br/sumarios/sumario-9788575222850.pdf>. Citado 2 vezes nas páginas 16 e 17.

PIETRANTUONO, R. et al. Towards continuous software reliability testing in devops. *Proceedings - 2019 IEEE/ACM 14th International Workshop on Automation of Software Test, AST 2019*, Institute of Electrical and Electronics Engineers Inc., p. 21–27, 5 2019. Citado 3 vezes nas páginas 4, 28 e 29.

PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de Software - Uma abordagem profissional*. [S.l.]: MC Graw Hill Education - Bookman - AMGH Editora Ltda, 2021. 1305 p. ISBN 9781259872976. Citado 4 vezes nas páginas 10, 16, 17 e 20.

RAFAEL, B. Tudo sobre devops — para iniciantes. Geek Hunter, 2019. Disponível em: <https://blog.geekhunter.com.br/tudo-sobre-devops-iniciantes/>. Citado na página 28.

RIBEIRO, G. D. Monografia de Graduação em Sistemas de Informação, *Automação de Testes Aplicados ao DevOps*. 2019. Citado na página 24.

ROSE, S.; WYNNE, M.; HELLESØY, A. *The Cucumber Book: Behaviour-driven Development for Testers and Developers*. [S.l.]: The Pragmatic Bookshelf, 2015. 325 p. Citado 2 vezes nas páginas 35 e 36.

SANTOS, C. D. O. D. Monografia de Graduação em Ensino Superior de Assis, *Comparação de Ferramentas para Automatização de Teste em Desenvolvimento Ágil*. 2009. Citado na página 32.

SANTOS, E. D. dos. Monografia de Graduação em Sistemas de Informação, *Aplicação de Testes Automatizados para Aplicações Mobile Desenvolvidas em React Native*. 2022. Citado na página 32.

SCHWABER, K.; SUTHERLAND, J. *Objetivo do Guia do Scrum*. 2020. Disponível em: <http://creativecommons.org/licenses/by-sa/4.0/legalcodeandalsodescribedinsummaryformathttp://creativecommons.org/licenses/by-sa/4.0/.Byutilizing>. Citado na página 16.

SILVA, G. *Scrum*. 2020. Disponível em: <https://analistaexpert.com.br/scrum/>. Citado 2 vezes nas páginas 4 e 17.

SILVA, M. L. M. da. Uma visão geral sobre automação de testes. Revista Científica Multidisciplinar Núcleo do Conhecimento, 2019. ISSN 2448-0959. Disponível em: <https://www.nucleodoconhecimento.com.br/tecnologia/automacao-de-testes>. Citado 3 vezes nas páginas 4, 33 e 39.

SLIDETEAM. *Os 25 principais modelos de processo de implantação*. 2023. Disponível em: <https://www.slideteam.net/blog/25-principais-modelos-de-processo-de-implantacao?lang=Portuguese>. Citado 2 vezes nas páginas 4 e 43.

SMART, J. F. *BDD in Action: Behavior-driven development for the whole software lifecycle*. [S.l.]: Simon and Schuster, 2023. 385 p. Citado 5 vezes nas páginas 21, 22, 25, 26 e 32.

SOARES, I. Desenvolvimento orietado por comportamento (bdd) - um novo olhar sobre o tdd. Java Magazinne, 2011. Disponível em: <https://www.devmedia.com.br/desenvolvimento-orientado-por-comportamento-bdd/21127>. Citado na página 21.

SOARES, J. G. O.; CASTELLI, N. M. M. Desenvolvimento de software Ágil com devops: Benefícios e desafios. 2023. Citado 2 vezes nas páginas 29 e 30.

STAFFEN, G. *Conheça o Cypress e suas vantagens para automação de testes*. 2021. Disponível em: <https://testingcompany.com.br/blog/conheca-o-cypress-e-suas-vantagens-para-automacao-de-testes>. Citado 2 vezes nas páginas 4 e 34.

SW, C. *Por quê aplicar BDD?* 2019. Disponível em: <https://ciclosw.wordpress.com/2019/06/03/porque-bdd/>. Citado na página 22.

TAYLOR, E. *The 5 Step Guide for Selenium, Cucumber, and Gherkin*. 2017. Disponível em: <https://www.cirruslabs.io/blog1/the-5-step-guide-for-selenium-cucumber-and-gherkin>. Citado 2 vezes nas páginas 4 e 37.

UTERMOHL, L. Cucumber, gherkin e bdd. 2023. Disponível em: <https://pt.linkedin.com/pulse/cucumber-gherkin-e-bdd-larissa-uterm%C3%B6hl#:~:text=O%20Gherkin%20%C3%A9%20uma%20linguagem,do%20usu%C3%A1rio%20ao%20desenvolver%20software.> Citado na página 21.

VALENTE, M. T. *Engenharia de Software Moderna: Princípios e práticas para desenvolvimento de software com produtividade*. 2020. Disponível em: <https://ler.amazon.com.br/kp/embed?linkCode=kpd&asin=B086K5QJ9V&tag=lp-ler-20&reshareId=JKG1G14ENZX786E3GT1V&reshareChannel=system>. Citado 4 vezes nas páginas 10, 15, 16 e 17.

WATERS, J. K. Scrum + devops = scrumops. ADT Magazine - Application Development Trends, 2017. Disponível em: <https://www.scrum.org/resources/scrum-devops-scrumops>. Citado na página 30.

APÊNDICE A – Configuração do Ambiente

Abaixo, é exemplificado um guia sobre como realizar a instalação do *Cypress* com o *Cucumber*:

1. Criar um diretório para o projeto *Cypress*:
   - Escolha ou crie um diretório onde deseja iniciar o projeto *Cypress*.
2. Inicializar o projeto *Node.js*:
   - No *Prompt* de Comando, navegue até o diretório do projeto *Cypress*.
   - Execute o seguinte comando para inicializar um novo projeto *Node.js*: **npm init -y**.

     *Isso criará um arquivo '*package.json*' no diretório, que é necessário para gerenciar as dependências do projeto.
3. Instalar o *Cypress*:
   - No *Prompt* de Comando, execute o seguinte comando para instalar o *Cypress*: **npm install cypress –save-dev**.

     *Isso instalará o *Cypress* e suas dependências no diretório '*node_modules*' do seu projeto.
4. Instalar o *Cucumber Plugin*:
   - Execute o seguinte comando para instalar o *plugin* do *Cucumber* no *Cypress*: **npm install –save-dev cypress-cucumber-preprocessor**.

     *Isso instalará o *plugin* para que seja possível escrever os testes usando a sintaxe do *Cucumber* (linguagem ubíqua).
5. Abrir o VS *Code*:
   - Após a instalação, execute o comando: **code .** para abrir o *Visual Studo Code*;
   - E, continue com os passos abaixo adicionando trechos de código no ambiente, para configurar o *Cucumber*.
6. Configurar o *Cypress* para usar o *Cucumber*:
   - Adicione a configuração necessária ao arquivo .../plugins/index.js, como mostra na Figura 37, abaixo:

```
const cucumber = require('cypress-cucumber-preprocessor').default

module.exports = (on, config) => {
  on('file:preprocessor', cucumber())
}
```

Figura 37 – Código demonstrativo onde o *Cypress* configura o *plugin* do *Cucumber*. Fonte: Da autora.

7. Configurar o *Cucumber*:

   - Adicione a configuração abaixo, ao arquivo *cypress.json*. É necessário configurar o *Cucumber* para reconhecer os arquivos *.feature* nos testes:

```
{
  "testFiles": "**/*.feature"
}
```

Figura 38 – Código demonstrativo *testFiles*. Fonte: Da autora.

8. Escrever os testes:

   - Agora já é possível começar a escrever os testes usando a sintaxe do *Cucumber* em arquivos .feature. Os cenários de testes podem ser criados na pasta *cypress/integration* do projeto.

9. Abrir o *Cypress*:

   - Basta executar o seguinte comando para abrir o *Cypress*: **npx cypress open**. *Isso iniciará o *Cypress* e abrirá a interface do usuário, como mostra na Figura 39.

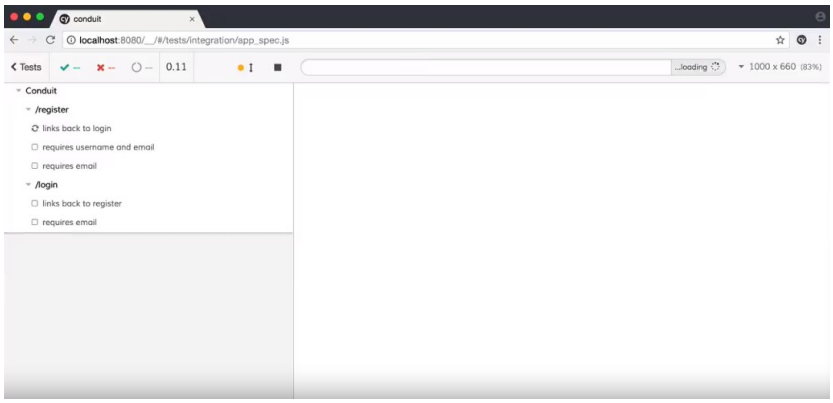

Figura 39 – Interface *Cypress*. Fonte: Extraído de (CYPRESS.IO, Acesso em abril de 2024).