UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"
FACULDADE DE CIÊNCIAS - CAMPUS BAURU
DEPARTAMENTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GEOVANNA CAROLINA FAVILLA VIDAL TRIGO BRANDÃO

QUALIDADE DE SOFTWARE: ANÁLISE DE DESEMPENHO DE FERRAMENTAS DE AUTOMAÇÃO EM TESTES E2E

BAURU
Novembro/2023

GEOVANNA CAROLINA FAVILLA VIDAL TRIGO BRANDÃO

QUALIDADE DE SOFTWARE: ANÁLISE DE DESEMPENHO DE FERRAMENTAS DE AUTOMAÇÃO EM TESTES E2E

Trabalho de Conclusão de Curso do Curso de Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.
Orientador: Prof. Dr. Kleber Rocha de Oliveira

BAURU
Novembro/2023

Geovanna Carolina Favilla Vidal Trigo Brandão QUALIDADE DE SOFTWARE: ANÁLISE DE DESEMPENHO DE FERRAMENTAS DE AUTOMAÇÃO EM TESTES E2E/ Geovanna Carolina Favilla Vidal Trigo Brandão. – Bauru, Novembro/2023- 47 p. : il. (algumas color.) ; 30 cm.
Orientador: Prof. Dr. Kleber Rocha de Oliveira
Trabalho de Conclusão de Curso – Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Ciência da Computação, Novembro/2023.
1. Tags 2. Para 3. A 4. Ficha 5. Catalográfica

Geovanna Carolina Favilla Vidal Trigo Brandão

QUALIDADE DE SOFTWARE: ANÁLISE DE DESEMPENHO DE FERRAMENTAS DE AUTOMAÇÃO EM TESTES E2E

Trabalho de Conclusão de Curso do Curso de Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.

Banca Examinadora

Prof. Dr. Kleber Rocha de Oliveira
Orientador
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Ciência da Computação

Professor Convidado 1
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Ciência da Computação

Professor Convidado 2
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Ciência da Computação

Bauru, _____ de _____ de _____.

Dedico este trabalho à criança que já fui.

Agradecimentos

A realização deste projeto de conclusão de curso contou com a ajuda de diversas pessoas queridas, dentre as quais agradeço encarecidamente:

Aos professores da UNESP campus Bauru, por esses cinco anos em que me acompanharam pontualmente oferecendo auxílio necessário.

Ao meu orientador, pela generosidade e empatia demonstrada para comigo.

Ao meu parceiro de vida, que me incentivou em todos os momentos de dúvida e apreensão.

Aos meus familiares, pelo apoio prestado durante minha criação.

Aos meus amigos e colegas, pela compreensão das ausências e afastamento temporário.

E por último, aos médicos que me acompanharam durante o momento delicado em que me encontro, possibilitando a restauração parcial de minha saúde e por conseguinte a finalização deste trabalho.

"Good testing involves balancing the need to mitigate risk against the risk of trying to gather too much information."

—Gerald M. Weinberg

Resumo

O presente trabalho tem como tema a análise de desempenho de ferramentas de automação de testes de ponta-a-ponta. Onde o objetivo é abordar esse estudo de caso a partir do modelo ISO 25010, aliado aos critérios ortogonais de classificação da abordagem de localização de objetos em tela. Ademais, é realizada uma análise do parecer da indústria de tecnologia em relação a essa categoria de testes. A escolha deste tema é impulsionada pela crescente complexidade dos sistemas de software, a necessidade da redução de custos e aumento da dependência social na tecnologia da informação.

Palavras-chave: Automação.Ferramenta.Teste.

Abstract

The focus of this work is on the performance analysis of end-to-end test automation tools. The aim is to conduct this case study using the ISO 25010 model and the orthogonal classification criteria of the on-screen object location approach. Moreover, a review of the technology industry's perspective on this type of test is conducted. The increasing complexity of software systems drives the choice of this topic, the need to reduce costs, and the increasing social dependence on information technology.
Keywords: Automation.Tool.Test.

Lista de figuras

Lista de quadros

Sumário

1 Introdução

A partir de 1998, com a introdução do TMap e outras metodologias, *software tester* tornou-se uma profissão em seu próprio mérito. Os profissionais deveriam estar comprometidos apenas com a comprovação dos requisitos do sistema, possuindo uma postura liberada de especifidades da área técnica (DAM, 2020).

Nessa época, as atividades envolvidas na verificação, validação e teste de software (*VV&T*), se restringiam a uma *"caça aos defeitos"* por meio do escrutínio da documentação. A partir da década de 2010, qualidade de software passou a ter um foco maior no ciclo de desenvolvimento de software e esforço para garantir a entrega do melhor produto possível.

Durante os estágios finais desse ciclo, poucas atividades são mais problemáticas e sofríveis do que empurrar uma nova versão *"quebrada"* para produção. Exigindo esforço repetitivo por parte dos desenvolvedores e testers para a correção, o que por si só já é um ponto bastante relevante a favor do emprego de automação dentro das atividades de validação do sistema (ATESOGULLARI; MISHRA, 2020).

Entre os muitos testes que podem ser automatizados, os testes *end-to-end* (ou E2E) são usualmente empregados quando já existe uma versão funcional do projeto. Isso ocorro devido a própria natureza desses testes, que necessitam verificar o comportamento do sistema *"from start to finish"* em busca de erros na solução (BADAL; GRUNLER, 2021).

1.1 Problemática

A natureza dos testes E2E, aliada aos *frameworks* de automação de testes adicionam certo grau de confiabilidade e velocidade ao processo. Os objetos de teste devem ser selecionados por possuírem dados mensuráveis (tempo de carregamento, performance, tipo de entradas e etc) para que possam sofrer um teste automatizado. Usualmente, o emprego deste tipo de teste é mais voltado para sistemas WEB, mas podem também ser utilizados em aplicações mobile ou desktop (BADAL; GRUNLER, 2021).

Entretanto, ao contrário do que dita o senso comum, a criação e manutenção de testes automatizados exige rodadas de testes manuais para definição de possíveis cenários. A partir desse ponto, o desafio está na escolha da ferramenta de automação ideal (ATESOGULLARI; MISHRA, 2020).

1.2 Justificativa

Tendo em vista que testes automatizados podem propiciar tanto a escalabilidade nas atividades quanto a redução dos custos operacionais, além de serem a melhor forma de garantir que o software está executando de maneira apropriada. A escolha da ferramenta de automação torna-se uma questão relevante dentro do ciclo de desenvolvimento de software (BADAL; GRUNLER, 2021).

Nesse contexto, este trabalho visa proporcionar um estudo de caso comparativo entre algumas das principais ferramentas disponíveis, ao mesmo tempo que descreve o paradigma atual do mercado em relação ao emprego de testes de automatizados de ponta-a-ponta.

1.3 Objetivos

Este projeto tem como objetivo produzir uma análise comparativa entre ferramentas contemporâneas voltadas à automação de testes de software.

1.3.1 Objetivos Gerais

Realizar um estudo de caso sobre o desempenho de ferramentas de automação de software, em um contexto de testes de ponta-a-ponta para WEB. Sendo o objeto de testes um conjunto de sites desenvolvidos especificamente para o estudo de ferramentas e *frameworks* de automação, como o DEMOQA ou Swag Labs.

1.3.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

- Investigar diferentes ferramentas de automação de testes de software contemporâneas.
- Estudar o paradigma atual do mercado em relação ao emprego de testes de ponta-ponta.
- Desenvolver suítes de testes automatizados robustas para demonstração das ferramentas.
- Realizar o estudo de caso comparativo e análise do desempenho das ferramentas selecionadas.

2 Fundamentação Teórica

2.1 Engenharia de Software

A engenharia de software inclui processos, conjuntos de práticas e ferramentas que permitem aos profissionais desenvolver software de qualidade. A importância da engenharia de software reside no fato que a mesma nos permite que sistemas complexos sejam desenvolvidos no prazo e com alta qualidade (PRESSMAN; MAXIM, 2016).

Isso ocorre, pois a mesma consegue criar um nível de disciplina suficiente, que pode gerar certo atrito entre setores, mas também que permite aos desenvolvedores conceber softwares da maneira que melhor se adapte as exigências dos clientes (PRESSMAN; MAXIM, 2016).

O uso extensivo de software na sociedade atual — a presença de sistemas de computador em serviços essenciais e não essenciais (manufatura industrial, sistemas financeiros, distribuição de energia, entretenimento, etc.) ou produtos eletrônicos contendo algum tipo de computador. Evidencia a grande necessidade das associações nacionais e internacionais da engenharia de software (SOMMERVILLE, 2011).

2.1.1 O que é software?

Software é um transformador de informações. As produz, adquire, modifica, transmite ou exibe, de forma que podem ser simples como poucos bits ou tão complexas como a imagem de um buraco negro na região da constelação de sagitário. Possui ainda duplo papel, sendo tanto produto como meio de distribuição logístico de produtos (PRESSMAN; MAXIM, 2016).

Ao contrário do que muitas vezes se supõe, do ponto de vista da engenharia de software, o termo software refere-se não apenas ao programa em si, mas também a toda a documentação relacionada e dados de configuração necessários para o funcionamento adequado desse programa (SOMMERVILLE, 2011).

Esta é a principal diferença entre o desenvolvimento de software profissional e amador. Quando você desenvolve um programa que ninguém mais usa, não precisa se preocupar com documentação ou arquitetura. No entanto, ao desenvolver software para terceiros, no qual outros engenheiros farão alterações, é preciso fornecer informações adicionais, como o código do programa (SOMMERVILLE, 2011).

2.1.2 O Ciclo de vida do software e o teste de Software

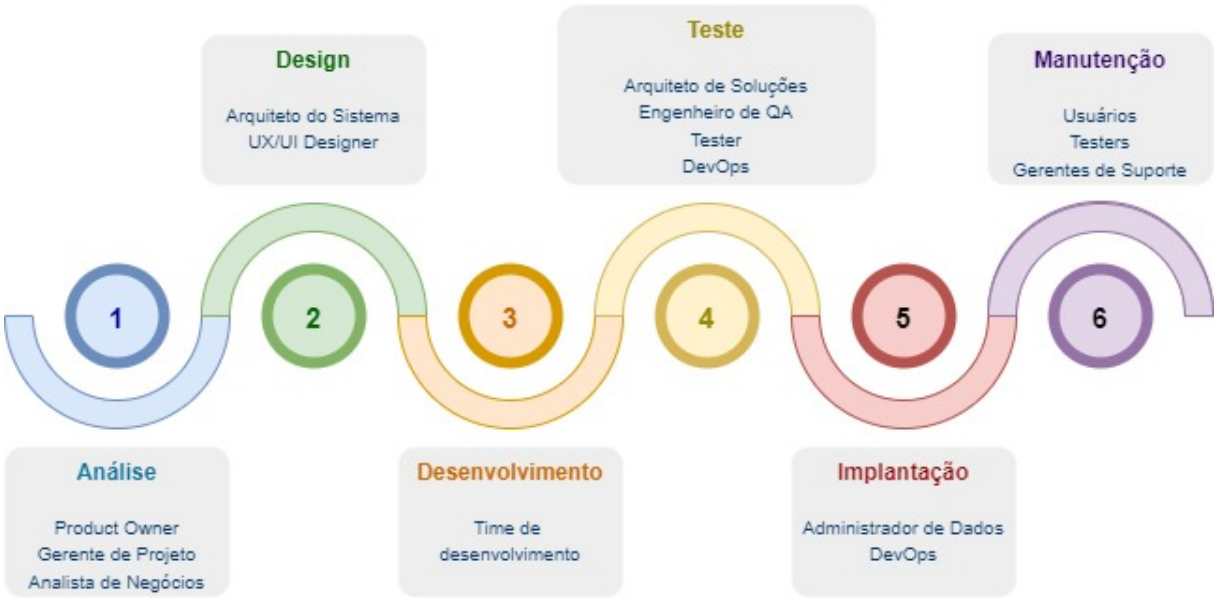
O ciclo de vida do software (sigla SDLC na língua inglesa) existe como um modelo abstrato, oferecendo uma representação em alto nível do processo de desenvolvimento. Define

como as diferentes fases (e tipos de atividade) interagem e se relacionam durante este processo. Alguns exemplos desses modelos são: desenvolvimento sequencial (cascata, V-model), interativo (espiral, protótipo), e incremental (processo unificado) (CERQUOZZI et al., 2023).

Parte das atividades existentes no processo de desenvolvimento podem também ser descritas em miúdos por métodos de desenvolvimento, bem como práticas ágeis: BDD (Behavior-driven development), DDD (Domain-driven development), programação extrema (XP), FDD (Feature-driven development), Kanban, Lean IT, Scrum, e TDD (Test-driven development) (CERQUOZZI et al., 2023).

Dito isto, o teste precisa se adaptar ao SDLC (ver Figura 1) para ser considerado bem-sucedido. A escolha do SDLC impacta tanto no escopo e tempo das atividades, como também no nível de detalhe da documentação dos testes, escolha de técnicas, abordagens, volume de automação e responsabilidades do testador de software (CERQUOZZI et al., 2023).

Figura 1 – O Ciclo de vida do software.



Fonte: Reprodução própria.

2.2 Controle de Qualidade

Na indústria de software, diferentes empresas e indústrias têm variadas interpretações de garantia e controle de qualidade. A garantia de qualidade pode significar simplesmente a definição de procedimentos, processos e padrões destinados a garantir a qualidade do software. Em outros casos, a garantia de qualidade também inclui todas as atividades de controle de configuração, verificação e validação realizadas pela equipe de desenvolvimento após a entrega do produto (SOMMERVILLE, 2011).

O controle de qualidade inclui várias atividades de desenvolvimento de software que garantem que o produto final atenda aos critérios de qualidade exigidos pelo projeto ao permitir

que o software sofra revisões constantes para assegurar sua robustez. Implementando várias etapas de teste e validação para detectar erros na lógica de processamento, manipulação de dados e comunicação de interface (PRESSMAN; MAXIM, 2016).

Garantia de Qualidade (QA, do inglês quality assurance) é a definição de processos e padrões que visam a produção de produtos de qualidade e a implementação de processos de qualidade na fabricação. O controle de qualidade é a aplicação desses processos de qualidade para selecionar produtos que não atendem ao nível de qualidade exigido (SOMMERVILLE, 2011).

2.2.1 Erros, defeitos, falhas e causas

De maneira geral, é possível encontrar defeitos em documentações, como requerimentos específicos ou script de teste, em código fonte, ou em artefatos de suporte como arquivos de build. Defeitos em artefatos produzidos no início do SDLC, se não detectados, costumam levar a artefatos defeituosos ao longo do ciclo de vida. Se um defeito em um código é executado, o sistema pode quebrar ou realizar algo que não deveria, causando uma falha. Alguns defeitos sempre resultam em uma falha quando executados, enquanto outros só chegariam a causar uma falha sob condições bastante específicas e alguns jamais irão causar qualquer falha (CERQUOZZI et al., 2023).

Pessoas cometem erros (enganos), que produzem defeitos (bugs), que por sua vez podem tornar-se falhas. Seres humanos podem cometer erros por uma miríade de motivos, pressão social, complexidade do trabalho, infraestrutura ou até mesmo cansaço e falta de treinamento apropriado. Erros e defeitos não são as únicas origens de falhas, essas podem ocorrer devido a intempéries climáticas, como radiação e campos eletromagnéticos que resultam em hardware defeituoso rodando o software (CERQUOZZI et al., 2023).

A causa raíz nada mais é do que a "razão fundamental"da ocorrência de uma falha, sendo identificada através de um processo de análise que costuma ser executado quando um defeito ou falha é identificado. Acredita-se que falhas similares ou defeitos futuros do mesmo tipo podem ser evitados ao se lidar com uma causa raíz identificada (CERQUOZZI et al., 2023).

2.3 Testes de software

O teste de software é um conjunto de atividades para identificar defeitos e avaliar a qualidade dos artefatos de software. No teste, esses artefatos são chamados de objetos de teste. Um equívoco comum sobre o teste é que se trata apenas de executar testes (ou seja, manter o software e verificar os resultados do teste) (CERQUOZZI et al., 2023).

O objetivo do teste é mostrar que o programa funciona conforme planejado e detectar

erros antes do uso. Os resultados do teste confirmam informações sobre erros de software, anomalias ou recursos não funcionais (SOMMERVILLE, 2011).

Entretanto, o teste de software inclui outras funções e deve estar alinhado com o ciclo de vida do desenvolvimento de software. Outro equívoco comum sobre o teste é que o teste se concentra inteiramente na verificação do objeto sob teste (CERQUOZZI et al., 2023).

O teste inclui verificação ou garantia de que o sistema atende a determinados requisitos, mas também inclui validação, o que significa garantir que o sistema atenda às necessidades dos usuários e outras partes interessadas (CERQUOZZI et al., 2023).

2.3.1 Princípios do teste de software

Segundo (CERQUOZZI et al., 2023), alguns princípios que oferecem guias gerais aplicáveis a todos os tipos de teste foram sugeridos através dos anos. Entre eles temos:

- ***Testes representam a presença, e não ausência de defeitos:*** Testes possibilitam verificar que defeitos estão presentes no objeto validado (BUXTON; RANDELL, 1970).Teste reduz a probabilidade de defeitos permanecerem não descobertos, entretanto, mesmo se nenhum defeito for encontrado, o teste não prova que defeitos inexistem no objeto avaliado.

- ***Testes exaustivos são impossíveis:*** Não é factível buscar testar exaustivamente um objeto. Ao invés disso, é preciso focar os esforços em avaliar a técnica de teste, priorização de caso de teste e testes baseados em riscos.

- ***Testes iniciais economizam tempo e dinheiro:***Defeitos removidos em estágios iniciais de um projeto não causarão subsequentes defeitos em produtos derivados do mesmo. Sendo assim, menos defeitos ocorrerão no decorrer do ciclo de desenvolvimento de software, de forma a reduzir o custo de qualidade no projeto.

- ***Defeitos se aglutinam:*** Segundo o princípio da escassez de fatores (ou princípio de Pareto), um número reduzido de componentes costumam conter a maior parte dos defeitos descobertos ou são responsáveis pela maioria das falhas operacionais. Previsão de núcleo de defeitos, e núcleos de defeitos observados durante a atividade de testes são uma entrada importante durante a avaliação de risco de testes.

- ***Testes se exaurem:***Testes repetidos muitas vezes tornam-se ineficientes em detectar novos defeitos. É possível que testes e dados de teste necessitem alterações de tempos em tempos, e novos testes podem necessitar ser escritos. Entretanto, vale ressaltar que em casos de testes automatizados para regressão é possível observar um retorno beneficial ao projeto.

- ***Testes dependem de contexto:*** Não há uma abordagem universalmente aplicável na área de testes. A atividade é realizada de acordo com as necessidades do contexto em que o projeto se encontra.
- ***A falácia da ausência de defeitos:*** É um equívoco pensar que a verificação de software através de testes irá garantir o sucesso de um sistema. Testar minuciosamente todos os requisitos e encontrar solução para todos os defeitos levantados, pode ainda sim levar a um sistema que não satisfaz completamente as necessidades do usuário final. Adicionalmente a verificação do software, é preciso que a validação por parte do usuário seja realizada a cada etapa do projeto.

Os objetivos da atividade de testes podem variar dependendo do contexto do projeto, produtos testados, nível de teste, riscos, o ciclo de desenvolvimento de software que está sendo seguido, e demais fatores relacionados a área de negócios.

2.3.2 Tipos de teste e níveis de teste

Níveis de teste são compostos por atividades de teste organizadas e gerenciadas ao mesmo tempo. São instâncias do processos de teste, executadas em relação ao estágio de desenvolvimento do software. São definidos de forma que o critério de saída de um nível, é o critério de entrada para o próximo nível de teste (ver **??**)(CERQUOZZI et al., 2023).

De acordo com Cerquozzi et al. (2023), é possível descrever os níveis de teste através das seguintes categorias:

- **Teste de componentes:** Também conhecido como "teste de unidades", o foco é testar componentes de maneira isolada. Geralmente é necessário suporte específico, como *frameworks* de testes. Usualmente, os testes de unidade são executados pelos desenvolvedores em seus ambientes.
- **Teste de integração de componentes:** Possui enfoque em testes de interface e interações entre componentes. São extremamente dependentes da estratégia de integração de componentes escolhida pelo projeto.
- **Teste de sistemas:** Seu foco reside em testar o comportamento e capacidade de sistemas ou produtos completos. Geralmente são compostos de testes funcionais de ponta-a-ponta, bem como testes não funcionais de características. Testes de sistemas costumam ser executados por times independentes, e estão vinculados a requisitos de sistema.
- **Teste de integração de sistemas:** Seu foco é testar as interfaces do sistema em teste e outros sistemas e serviços externos. Os testes de integração de sistemas requerem ambientes de teste adequados, de preferência semelhantes ao ambiente operacional.

- **Teste de Aceitação:** Seu foco é na validação e na demonstração de prontidão para implantação, o que significa que o sistema atende às necessidades de negócios do usuário. Idealmente, os testes de aceitação devem ser realizados pelos usuários finais. Sendo as principais formas de testes de aceitação: testes de aceitação do usuário (UAT), testes de aceitação operacional, testes de aceitação contratuais e regulatórios, testes alfa e testes beta.

Figura 2 – Estratégia de Teste.

Fonte: (PRESSMAN; MAXIM, 2016).

Diversos tipos de teste podem ser elencados. Neste trabalho, a atenção está voltada aos quatro tipos definidos por Cerquozzi et al. Sendo estes os que seguem:

- **Testes Funcionais:** Avalia as funções que componentes ou sistemas deveriam executar. O principal objetivo do teste funcional é verificar o funcionamento completude, correção funcional e adequação funcional.
- **Testes Não-Funcionais:** avalia atributos diferentes das características funcionais de um componente ou sistema. O teste não funcional é o teste de "quão bem o sistema se comporta". O principal objetivo dos testes não funcionais é verificar as características de qualidade do software não funcional. A norma ISO/IEC 25010 fornece a seguinte classificação das características de qualidade de software não funcionais:
  - Eficiência de desempenho;
  - Compatibilidade;
  - Usabilidade;
  - Confiabilidade;
  - Seguridade;
  - Capacidade de Manutenção;
  - Portabilidade;

- **Testes de Caixa Preta:** É baseado em comportamento e deriva da documentação do objeto de teste. Seu objetivo principal é testar o comportamento da solução em relação às suas especificações.
- **Testes de Caixa Branca:** É baseado na estrutura do projeto e implementação do sistema. Seu objetivo é cobrir o funcionamento da estrutura subjacente com testes de nível apropriado.

2.3.3 Teste e *Debugging*

Categoricamente, teste e *debugging* são atividades separadas. Os testes de um projeto podem dar visibilidade para falhas, que por sua vez são ocasionadas por defeitos no software (testes dinâmicos) ou podem diretamente encontrar defeitos no objeto de teste (testes estáticos) (CERQUOZZI et al., 2023).

Quando a execução de testes dinâmicos dá visibilidade a uma falha, a atividade de *debugging* é realizada de forma a encontrar as causas dessa falha, realizar uma análise e posteriormente eliminá-las. Subsequentemente, testes de confirmação verificam se o conserto realizado resolveu o problema (CERQUOZZI et al., 2023).

Quanto ao processo de *debugging*, realizado ao encontrar defeitos por meio de testes estáticos, seu intuito reside em buscar eliminar o defeito. Não há necessidade de reprodução e diagnose, afinal, testes estáticos encontram os defeitos diretamente (CERQUOZZI et al., 2023).

2.4 Behavior-driven development (BDD)

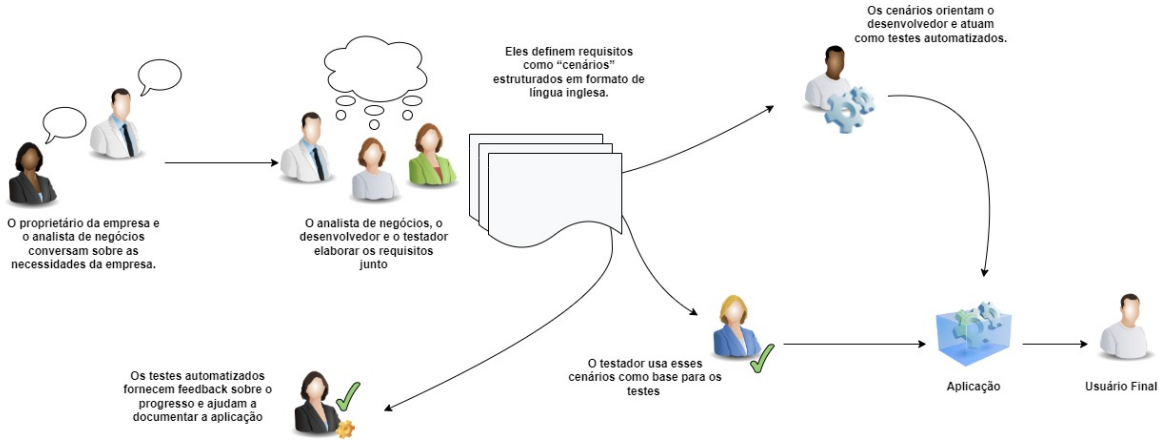
Segundo (SMART; MOLAK, 2023), é possível entender o BDD como um conjunto de práticas de engenharia de software desenhadas para auxiliar times a construir e entregar produtos mais valiosos, com maior qualidade de maneira mais rápida. É considerado uma prática ágil, com fortes influências do desenvolvimento baseado em testes (TDD), assim como do design baseado em domínio (DDD).O BDD provê de uma linguagem ubíqua, baseada em linguagem natural, a equipe responsável pelo projeto. Isso facilita a comunicação, como pode ser compreendido pela Figura 3, entre membros de equipe e os donos do negócio (*stakeholders*).

Originalmente, foi criado durante uma tentativa de ensinar TDD de maneira facilitada e prática por Dan North em meados dos anos 2000. Devido a tendência dos desenvolvedores perderem o conceito geral do projeto ao focar fortemente em testes unitários (como acontece durante o TDD), North observou que o simples ato de nomear os testes unitários como sentenças completas já reduzia esse efeito (SMART; MOLAK, 2023).

Testes escritos com essas características passam a se assemelhar a especificações de projeto. Tem um enfoque no comportamento da aplicação, utilizando os testes apenas como

ferramenta de expressão e validação deste comportamento. Além disso, testes escritos com essas características tornam-se fáceis de manter pois seu intento é explícito (SMART; MOLAK, 2023).

Figura 3 – SDLC utilizando BDD.

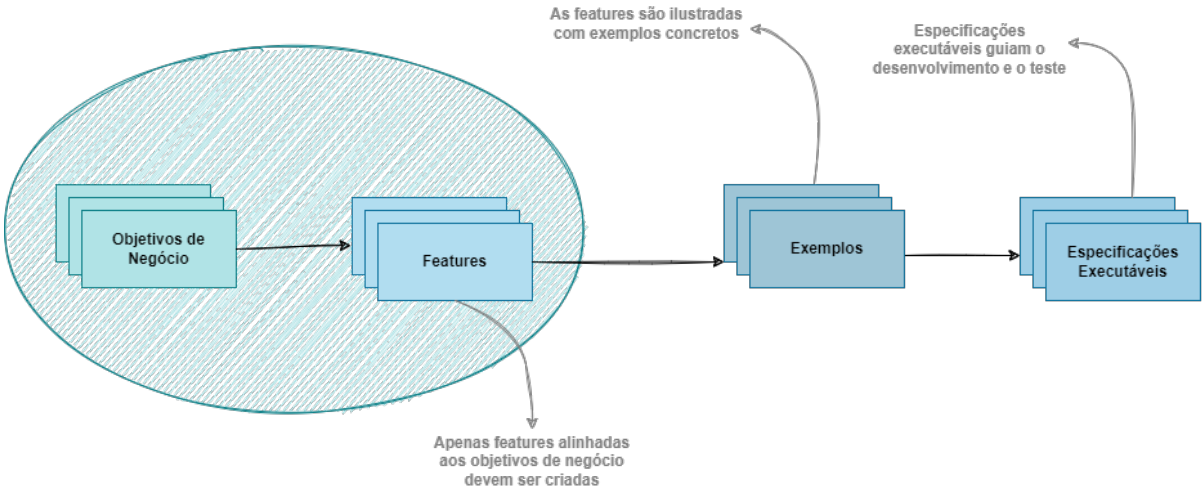


Fonte: (SMART; MOLAK, 2023) traduzido pela autora.

2.4.1 Análise de requisitos dentro do BDD

A partir da ideia de linguagem ubíqua pertencente ao Domain-Driven Design, foi criada uma linguagem que pessoas da área de negócios poderiam utilizar para definir requisitos inequivocamente; também sendo passível de transformação em testes de aceitação automatizados. Essas notações vagamente estruturadas e de fácil entendimento eventualmente evoluíram para a linguagem referida como *Gherkin*. Com a estrutura correta, cenários descritos em *Gherkin* podem ser executados automaticamente sempre que necessário ao projeto (SMART; MOLAK, 2023).

Figura 4 – Análise de requisitos utilizando BDD.



Fonte: (SMART; MOLAK, 2023) traduzido pela autora.

Times ágeis costumam escrever um pequeno esboço do requerimento, onde a ordem de escrita é bastante importante. Primeiro sempre é apresentado o valor que se pretende entregar, seguido pelo agente que será beneficiado, e finalmente a *feature* que irá gerar esse resultado esperado. Esse método é uma garantia que cada entrega converterá em valor agregado ao negócio (consultar Figura 4), também contribuindo para reduzir chances de um aumento no escopo de entregas. Do ponto de vista do time de desenvolvimento, essa técnica é como um lembrete da real necessidade dessa implementação (SMART; MOLAK, 2023).

Em alguns casos torna-se necessário a quebra de *features* em pequenas histórias de usuário. Nesse caso, o objetivo é que cada história explore uma faceta do problema de modo a ser entregue em uma única iteração. Essas histórias podem ser descritas com um pouco mais de detalhes no mesmo formato utilizado para as *features*. Esse processo todo acaba contribuindo para a criação dos "Critérios de Aceitação", consultar Figura 5, que ao ser atingidos servem como uma confirmação que o sistema está realizando de forma correta o que ele deveria (SMART; MOLAK, 2023).

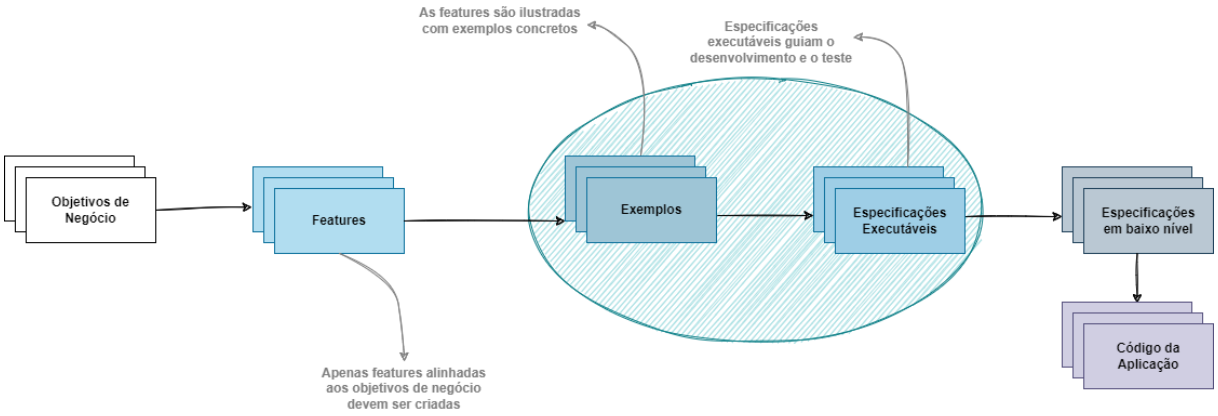
Figura 5 – Processo de desenvolvimento voltado aos objetivos de negócio.

Fonte: (SMART; MOLAK, 2023) traduzido pela autora.

2.4.2 Transformando histórias em cenários executáveis

Ao automatizar critérios de aceitação por meio das histórias de usuário descritas, é necessário escrevê-las de uma forma mais estruturada. A esse recurso é dado o nome *cenário*, cujo aspecto canônico ("Given...When...Then") foi descrito por Dan North em meados dos anos 2000 e continua a ser utilizado nos dias de hoje. Esse formato de cenário é de fácil leitura e automação, sendo parte integral da chamada "Documentação Viva"(fluxo descrito na Figura 6) do projeto (SMART; MOLAK, 2023)

Figura 6 – Transformando exemplos concretos em cenários executáveis.



Fonte: (SMART; MOLAK, 2023) traduzido pela autora.

A estrutura dos cenários, como mencionado anteriormente, é composta por três partes distintas: Um contexto inicial ("Given"), uma ação ou evento ("When"), e um resultado esperado ("Then"). É um formato simples e versátil, onde:

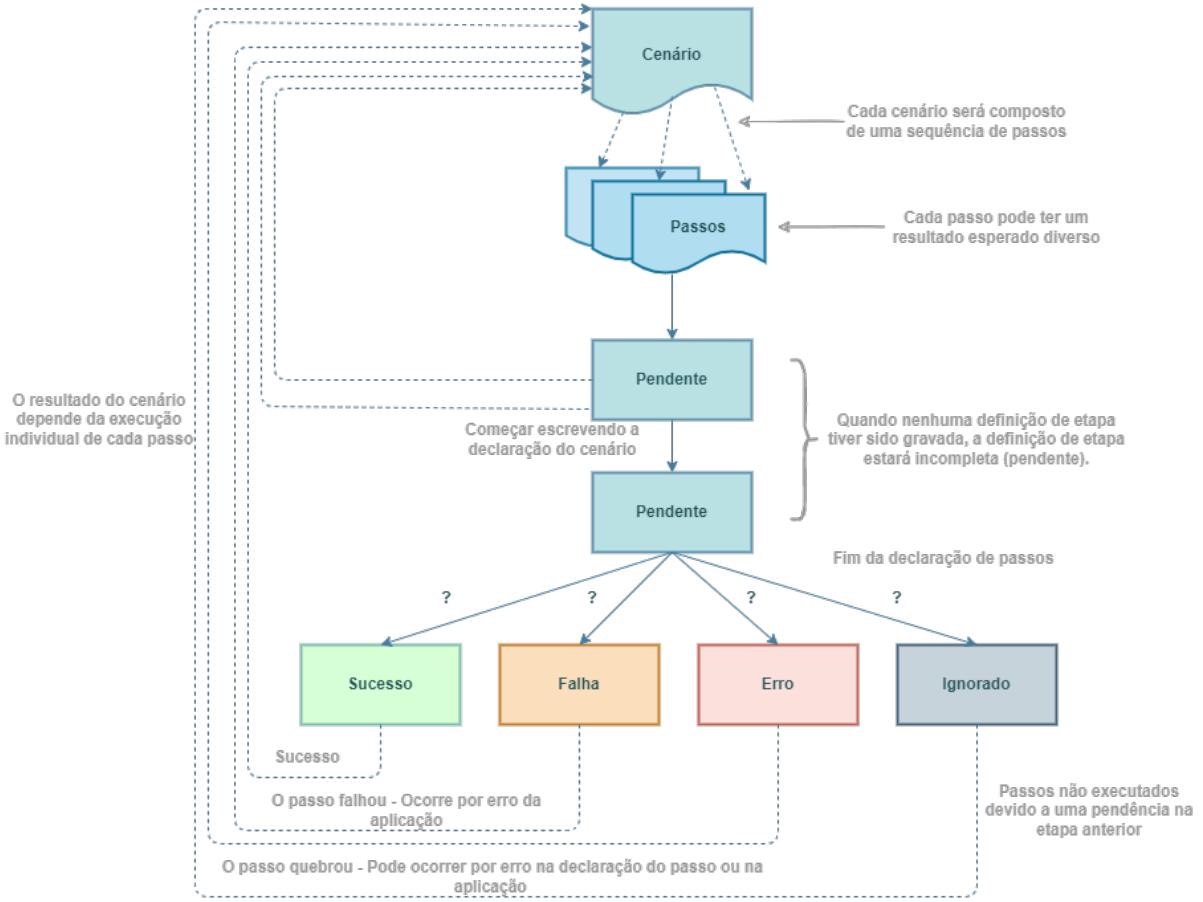
- A declaração **"Given"** descreve as pré-condições do cenário de teste e geralmente coloca a aplicação no estado correto. Usualmente, inclui a criação de qualquer dado de teste (ou no caso de aplicações web, logar e navegar entre telas). Há casos específicos em que essa etapa é apenas descritiva;
- A declaração **"When"** descreve a ação principal, ou evento, que é necessário testar. Como, por exemplo, um usuário performando uma ação dentro de página WEB, ou algum evento não relacionado a Interface de Usuário (UI, do inglês "User Interface"), como um processamento de transações ou tratamento de mensagem de evento;
- A declaração **"Then"** compara o resultado obtido durante a execução do cenário e o que era esperado.

Um artifício interessante apontando por J.F. Smart, é a utilização de dados dispostos em tabelas durante a escrita do teste que devem aparecer imediatamente após as declarações **"Given"** e **"Then"**. Incorporar dados tabelados é uma ótima maneira de expressar condições prévias e resultados esperados de forma clara e concisa, também é possível utilizar como exemplo de resultado esperado.

2.4.3 "Documentação Viva"no BDD

J.F. Smart enfatiza em seu livro o encorajamento da colaboração interdisciplinar para a criação dos critérios de aceitação durante a utilização do BDD. Isso se materializa durante o ciclo de desenvolvimento de software em forma de exemplos concretos, cenários, especificações executáveis, além da forma que esses artefatos guiam a codificação e entrega das *features*

Figura 7 – Implementando definições de etapas: princípios gerais.



Fonte: (SMART; MOLAK, 2023) traduzido pela autora.

necessárias ao projeto (consultar Figura 7). E no momento dessas entregas, a "Documentação Viva"faz o elo entre os requerimentos do projeto e as *features* desenvolvidas, confirmando que o código implementado corresponde com o que o time originalmente discutiu.

Este ciclo de *feedback* é uma excelente forma de obter a adesão dos *stakeholders*, que muitas vezes estão mais interessados em contribuir ativamente quando veem os resultados das suas sugestões literalmente na documentação viva. Além disso, como os relatórios são gerados automaticamente com base em critérios de aceitação automatizados, é uma forma rápida e eficaz de fornecer *feedback* depois de configurado (SMART; MOLAK, 2023).

A equipe de QA (do inglês, *"Quality Assurance"*) também usa a documentação viva para complementar suas próprias atividades de teste, para entender como os recursos foram implementados e para ter uma ideia melhor das áreas nas quais devem concentrar seus testes exploratórios (SMART; MOLAK, 2023).

Os benefícios da documentação viva não devem terminar quando um projeto é entregue. Quando organizada adequadamente, a documentação dinâmica também é uma ótima maneira de atualizar os novos membros da equipe não apenas sobre o que o aplicativo deve fazer, mas também como ele o faz. Para organizações que entregam projetos a uma equipe diferente

assim que entram em produção, os benefícios disso por si só podem valer o tempo investido na configuração do relatório de documentação viva (SMART; MOLAK, 2023).

Mas a documentação viva vai além de descrever e ilustrar os recursos que foram construídos. Muitas equipes também integram seus relatórios de BDD com sistemas ágeis de gerenciamento de projetos ou de rastreamento de problemas, tornando possível focar no estado dos recursos planejados para uma versão específica. As equipes que fazem isso normalmente contam com a documentação ativa para produzir seus relatórios de lançamento, caso eles não sejam gerados automaticamente a partir dos relatórios do BDD (SMART; MOLAK, 2023).

3 Automação de testes para Web

3.1 Evolução de ferramentas e estratégias

Em meados da década de 1940, o teste era uma atividade exclusiva de programadores sem muita distinção do processo de *debugging*. A partir do surgimento de sistemas mais vastos e complexos, a figura dos testadores de software tornou-se uma tendência de mercado. A primeira conferência a respeito do tópico foi organizada em 1972, em Chapel Hill na Carolina do Norte (EUA). Durante a mesma, a ideia da atividade de teste como algo distinto da programação foi apresentada (ANGMO; SHARMA, 2014).

A automação da tecnologia permite que a informação seja compartilhada de maneira ilimitada. Ferramentas de automação aplicadas de maneira independente, ou combinada, auxiliam equipes a superar desafios dentro da atividade de teste de software. Muitos pesquisadores utilizam ferramentas de automação como objeto de trabalho e pesquisa. Durante um estudo de caso industrial, Leotta et al. (2013a apud ANGMO; SHARMA, 2014, p. 909) pode identificar a significante redução no tempo e número de modificações para reparos em suítes de teste construídas com Selenium WebDriver quando o padrão de design PageObject é implementado nas mesmas.

Em um estudo similar a respeito de manutenção de suítes de teste automatizadas em um contexto de Sistema de Gerenciamento de Conteúdo de Aprendizagem (LCMS, do inglês "*Learning Content management System*"), Leotta et al. (2013b apud ANGMO; SHARMA, 2014, p. 909) propõe um esforço de realinhamento, em que se conclui que o método de localização de elementos de página web via ID gera um menor esforço de manutenção entre entregas em relação ao modelo de localização via Xpath.

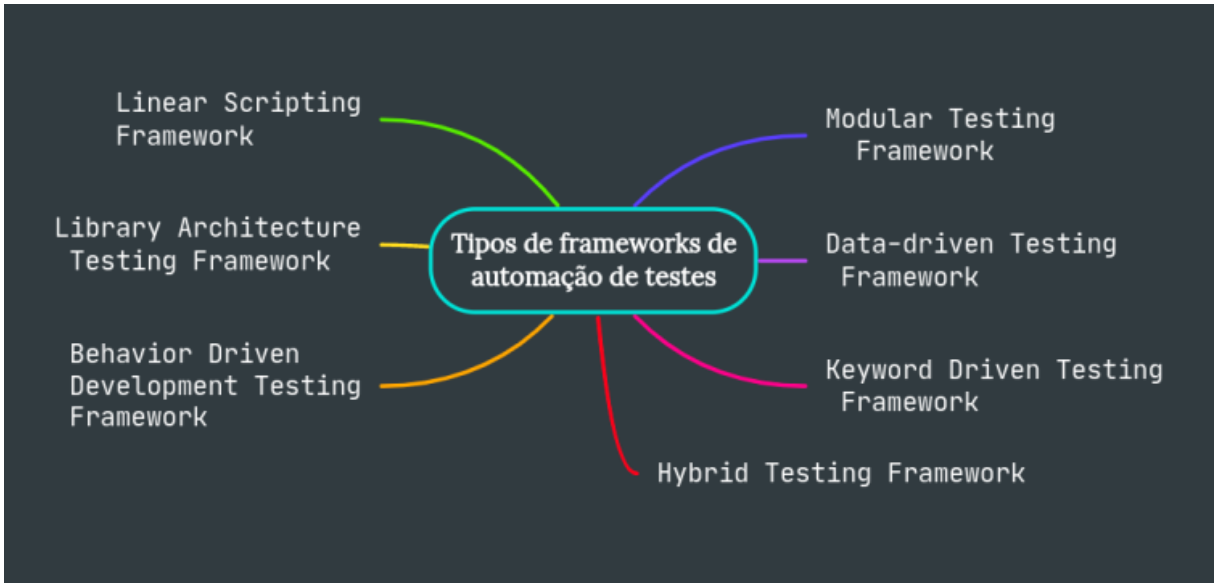
Cheluvaraju, Nagal e Pasala (2012 apud ANGMO; SHARMA, 2014, p. 909), propuseram um novo método de investigação de relações entre arquivos que sofreram controle de versão em conjunto. Esse método envolveu a aplicação avançada Análise de Redes Sociais (SNA, do inglês "*Social Network Analysis*") em uma rede. Neste trabalho, foi executada a análise empírica e revisão de histórias em Selenium -importante ferramenta de código aberto voltada a testes de sistemas-, extraindo parâmetros de resultado para alteração de dependências *cross-language*, mudança de propagação e análise de impacto.

Xu et al. (2012 apud ANGMO; SHARMA, 2014, p. 909), em seu estudo a respeito da mineração de especificações executáveis a partir de suítes de teste feitas para o Selenium IDE, demonstram a integração entre dois processos opostos de mineração de modelos e geração de testes (baseado em modelos) em um único *framework*.

3.1.1 *Frameworks* e ferramentas de automação

Uma estrutura de automação de teste (*framework*) é uma plataforma que inclui dados de teste, casos de teste, *scripts* de teste e ferramentas de automação de teste para fornecer relatórios e estratégias apropriadas para execução do controle de qualidade de um projeto com um nível de padronização das atividades executadas pelos *scripts* de teste (PINHEIRO, 2023).

Figura 8 – Diferentes categorias de *frameworks*.



Fonte: (PINHEIRO, 2023).

3.2 Exemplos de ferramentas

3.2.1 Ferramentas de testes para Web

Testes focados em aplicações web auxiliam a diminuir custos, minimizar o tempo de entrada no mercado e buscam a reutilização de cenários de teste. Existe uma gama de categorias de teste que podem ser utilizados para esse fim, entre eles: Testes Funcionais, Testes de Compatibilidade, Testes Localizados, Testes de Performance, Testes Funcionais de Serviços Web e Testes de Regressão (ANGMO; SHARMA, 2014).

Devido a natureza mutável e competitiva dos negócios baseados na web, há uma necessidade crítica de atividades de teste manual nesse seguimento. Dessa forma, testes automatizados devem garantir que esses serviços (portais, aplicações e etc.) tenham suas funções básicas funcionando corretamente, acomodando a necessidade reutilizar e estender os casos de teste para que sejam válidos independente do navegador, plataforma, tipo de base de dados ou linguagem de programação implementada (ANGMO; SHARMA, 2014).

3.2.2 Selenium IDE

Selenium IDE, onde IDE significa *Integrated Development Environment* (do inglês "Ambiente de Desenvolvimento Integrado") é uma ferramenta que permite gravar as ações do usuário a partir de comandos Selenium com parâmetros definidos para o contexto de cada elemento disposto em tela. É uma ferramenta que funciona como extensão do navegador web (SELENIUM, 2019).

Pode-se considerar o Selenium IDE um *framework* de automação do tipo linear, bastante antigo e simples de se utilizar. Não se adequando à execução de testes com conjuntos de dados diversificados devido a suas limitações quanto a reutilização de dados nas suítes de teste (PINHEIRO, 2023).

Devido a sua simplicidade, não necessita de um planejamento intenso ou consome muito tempo do engenheiro de teste. Além disso, isenta o usuário da necessidade de ter conhecimento em programação. Entretanto, a manutenção de conjuntos de teste criados por meio do Selenium IDE é alta, requerendo muito esforço para alterações mínimas (PINHEIRO, 2023).

Figura 9 – Características da ferramenta Selenium IDE.

Fonte: Reprodução própria.

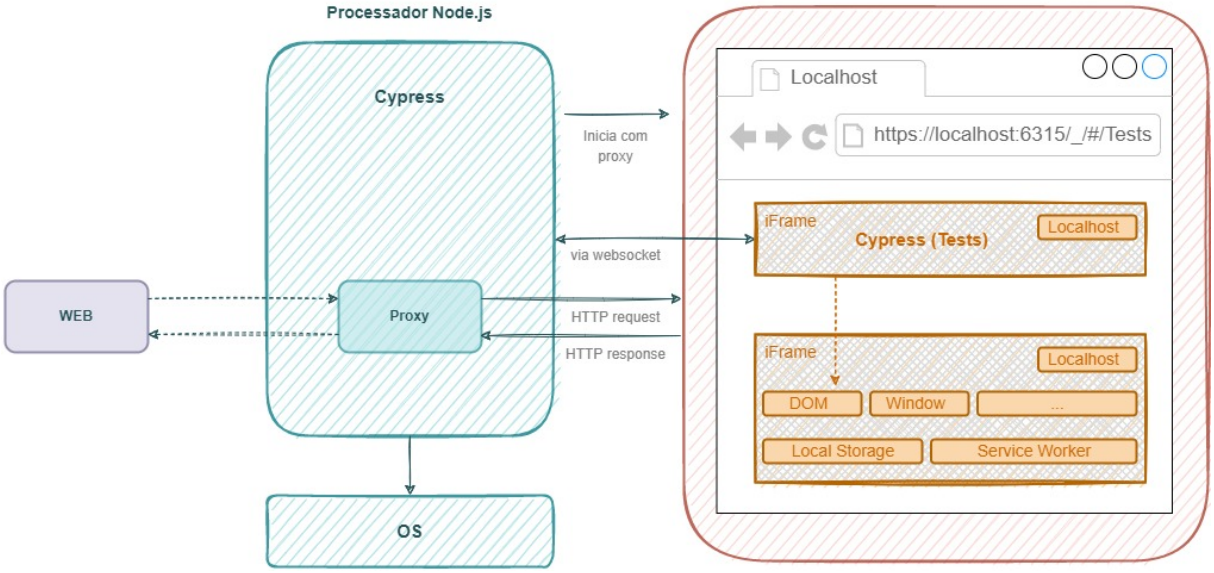
3.2.3 Cypress

Cypress é uma ferramenta de teste *front end* para web, construída pensando nas aplicações mais recentes da dita web 2 e 3.0, onde as páginas são desenvolvidas de maneira mais dinâmica e fluida (CYPRESS, 2023).

Nas versões mais recentes, é possível compreender o Cypress como um *framework* de testes voltado ao BDD. Isso se deve a adoção da syntax mocha pela ferramenta. Um *framework* desse tipo permite a diversificação de dados de teste com intervenção mínima no software. Esse detalhe é seu principal mecanismo de economia, visto que garante a reutilização do código (PINHEIRO, 2023).

A ferramenta consiste no Cypress Cloud (para gravar as execuções dos cenários de teste), e uma aplicação gratuita, *open source*, instalada localmente. Após a construção da suite de testes, há a possibilidade de utilizar a Cypress Cloud para um esforço de Integração Contínua (CI, do inglês "*Continuous Integration*") (CYPRESS, 2023).

Figura 10 – Diagrama arquitetural do Cypress.



Fonte: (ELM, 2019) traduzido pela autora.

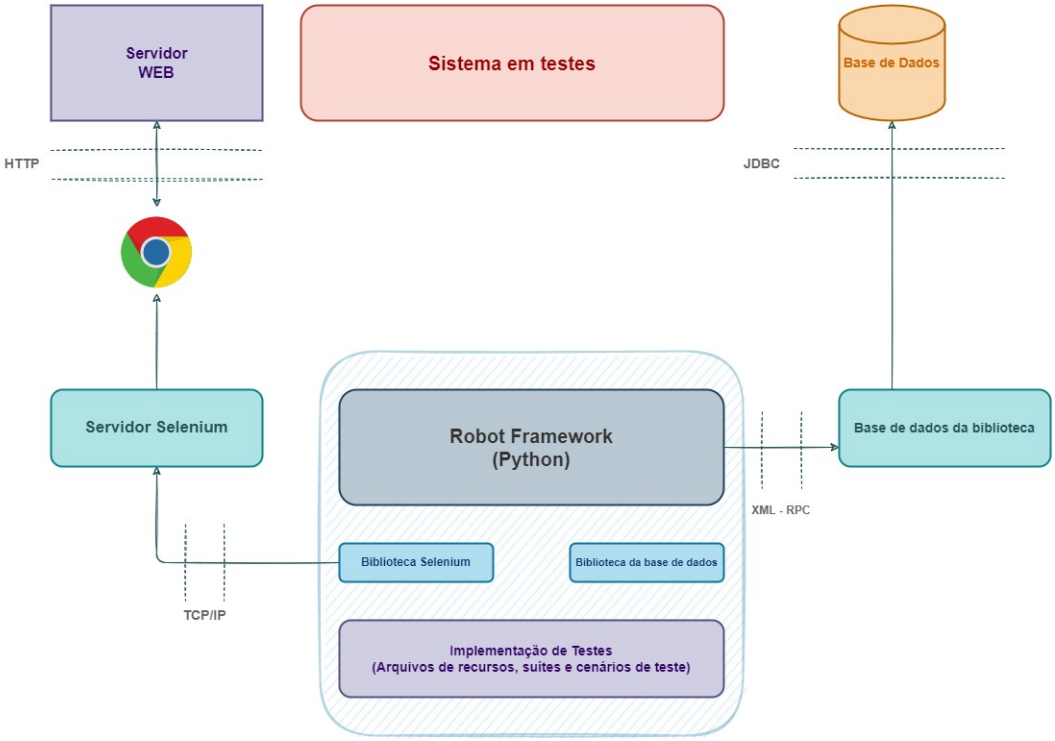
3.2.4 Robot *framework*

Robot é um *framework* de automação *open source* genérico, sendo utilizado tanto para automação de testes quanto para processos de automação robótico (RPA, do inglês "*Robotic Process Automation*"). É de uso gratuito, sem custos de licenciamento (ROBOT, 2023).

Com uma sintaxe simples, utilizando palavras-chave em linguagem humana. Suas capacidades podem ser estendidas por bibliotecas implementadas com Python, Java ou outras linguagens de programação. O *framework* possui um rico ecossistema, composto por bibliotecas e ferramentas que são desenvolvidas como projetos separados (ROBOT, 2023).

Segundo Pinheiro (2023), pode ser considerado um *framework* baseado em palavras-chave (do inglês, "*Keyword-Driven Framework*"), usando um formato de tabela para definição das mesmas. Cada palavra-chave é definida para uma função ou método que se executará, e é possível desenvolver qualquer script de automação de teste sem prejuízo para testadores sem robusto conhecimento em programação.

Figura 11 – Diagrama arquitetural do Robot Framework.

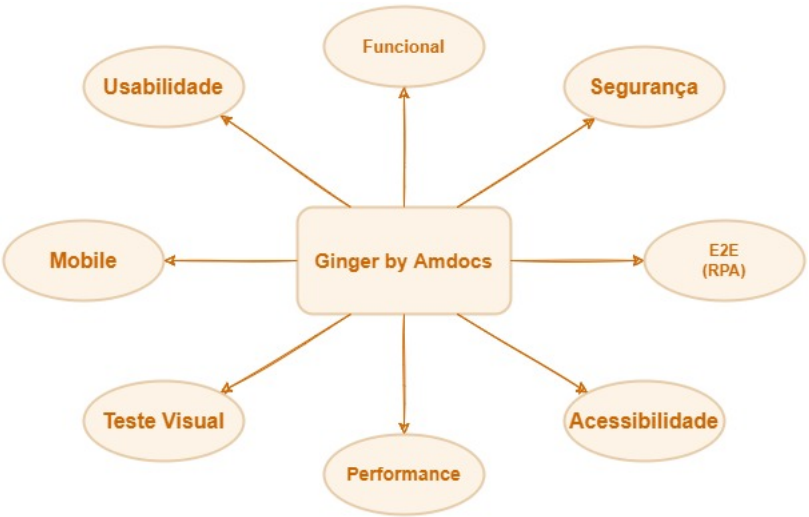


Fonte: (TECHGEEKNEXT, 2023) traduzido pela autora

3.2.5 Ginger

Ginger é um *framework* de automação de código aberto que maximiza os benefícios do DevOps, metodologias de desenvolvimento ágil e migração para nuvem. Permite testes contínuos com automação orientada por IA, orquestração de scripts (criados por ferramentas de qualquer linguagem de programação), criação de dados de teste e automação de gerenciamento de ambiente de teste (AMDOCS, 2023).

Figura 12 – Versatilidade do Ginger.



Fonte: Reprodução própria

O mecanismo de orquestração para ferramentas de execução de testes, mencionado

anteriormente, funciona na maioria dos sistemas operacionais (Windows, Unix, Linux, etc.), linguagens de programação (Java, .Net, aplicações web, aplicações para mainframe) e demais ferramentas de teste (SoapUI, Rest Assured e Cucumber) (RAVIART, 2019).

Essa versatilidade culmina em uma ampla capacidade de execução de testes de ponta-a-ponta (E2E, do inglês "*End To End*"), o que é particularmente relevante em indústrias que operam em processos de negócio padronizados. Dessa forma, tornou-se estado da arte para testes de integração no setor de telecomunicações (RAVIART, 2019).

3.3 Sobre Qualidade de Software

Segundo Amara e Rabai (2019), a qualidade de software é comumente definida através de algumas características, normalmente mencionados como atributos. Diversas definições desses atributos são encontradas, sendo as mais utilizadas:

- O padrão IEEE (IEEE 610.12,1990) define um atributo como "uma característica de um item que afeta sua qualidade";
- Lyu em seu trabalho (LYU,1996) define atributos como medidas usadas para compreender a performance de um software;

Dessa forma, os atributos são necessários para refletir a qualidade do software e também descrevê-la. Além disso, tanto a parte estrutural do produto como a forma que ele se relaciona com o ambiente é levada em conta. Isso acarreta na criação de atributos internos e externos de qualidade (AMARA; RABAI, 2019).

3.3.1 Atributos internos de qualidade

Atributos internos de qualidade são ligados as propriedades estruturais do código, cuja quantificação dependem apenas de sua representação no programa. Também podem ser definidos como propriedades estáticas do código como: tamanho, complexidade, coesão, e etc. Dentro do SDLC é possível quantificá-los em estágios iniciais do desenvolvimento (AMARA; RABAI, 2019).

- **Tamanho:** é um indicador de qualidade mensurado a partir da quantidade total de linhas de código executável (exclui-se aqui linhas em branco e comentários). Basicamente, é apresentado a quantidade de trabalho produzido durante o desenvolvimento podendo ser computado sem a necessidade de execução.
- **Complexidade:** Segundo o IEE 610.12 (1990), a complexidade aparece quando o design ou implementação de um componente (ou sistema) apresenta-se difícil de compreender

ou verificar. Também pode ser compreendida como a complexidade de um problema ou complexidade de algoritmo. Sendo a complexidade de problema a quantidade de recursos (armazenamento ou tempo) necessários para obter uma solução considerada satisfatória. Por sua vez, a complexidade de algoritmos é a complexidade do algoritmo implementado para obter tal solução.

- **Coesão:** É o grau de dependência que um módulo do software possui em relação a outros para executar uma tarefa (IEE 610.12,1990). Em adição, a coesão também descreve como elementos de um módulo interagem e reflete o grau de consistência entre eles.
- **Acoplamento:** Identifica a relação entre elementos de diferentes módulos do sistema. Quando os módulos possuem forte acoplamento a dificuldade de compreensão e manutenção dos mesmos aumenta e, dessa forma, a complexidade total do sistema.

Essa categoria de atributos é importante pois são informações que podem ser obtidas em estágios iniciais do desenvolvimento do produto, e tem natureza mais simples de captação do que informações a respeito de atributos externos (AMARA; RABAI, 2019).

3.3.2 Atributos externos de qualidade

São atributos que tornam-se disponíveis após a conclusão da implementação e execução do produto. Refletem a perspectiva do usuário, pois são afetados diretamente pela máquina, sistema operacional e padrão de utilização (AMARA; RABAI, 2019).

- **Capacidade de Suporte:** Uma medida da capacidade do sistema de suporte de software de atender as necessidades do usuário final.
- **Capacidade de Manutenção:** Uma medida da facilidade que o produto de software possui em ser modificado e adaptar a modificações do sistema (IEEE 610.12 1990).
- **Flexibilidade:** Uso eficientes e efetivos de um produto ou sistema em diferentes aplicações e ambientes (ISO/IEC 25010:2011; IEEE 610.12 1990).
- **Usabilidade:**Facilidade de operação, preparação de entradas de dados e interpretação das saídas de dados pelo usuário (IEEE 610.12 1990).
- **Confiabilidade:** Execução das funções definidas, por um período de tempo, com respeito as especificações do produto (IEEE 610.12 1990).
- **Portabilidade:** Possibilidade de uso do produto de software em diferentes ambientes (IEEE 610.12 1990).
- **Interoperabilidade:** A troca de informação entre diferentes componentes dos produtos do sistema, bem como a capacidade de utilização com outras aplicações.

- **Desempenho:**A capacidade de um sistema de realizar suas funções com relação a diferentes restrições, como velocidade e armazenamento de memória.

Por fim, o foco nos recursos mencionados acima existe pois são os mais importantes conhecidos na literatura. Acredita-se que as características internas e externas são importantes na medição da qualidade porque não são mutuamente exclusivas. Atributos internos podem ser usados como preditores externo (AMARA; RABAI, 2019).

3.3.3 Modelos de qualidade de software

Amara e Rabai (2019) descreve que os modelos de qualidade são propostos para discorrer sobre a interação entre os atributos de qualidade, enunciá-los e facilitar sua medição. Resultando na possibilidade de comparação entre produtos de software distintos. Os modelos decompõem os atributos de qualidade baseando-se, geralmente, em uma estrutura de árvore de três níveis.

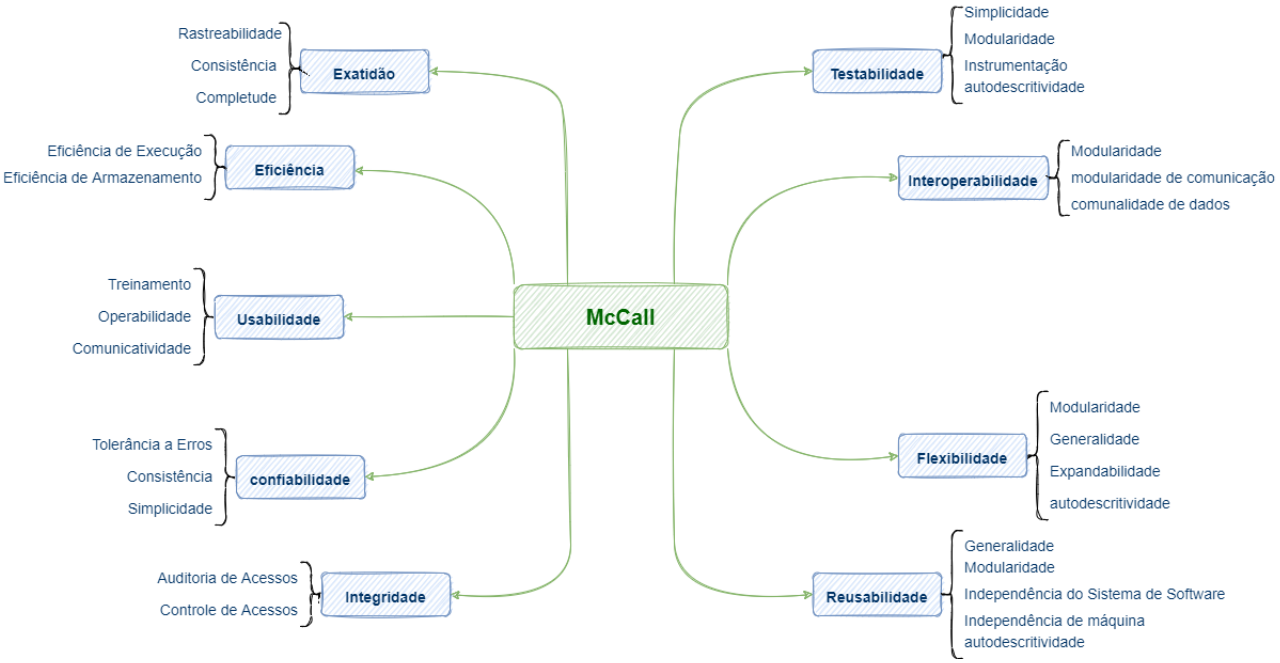
O ramo superior contém os atributos de qualidade mais importantes que se quer medir. Esses atributos podem ser externos, como confiabilidade e capacidade de manutenção, ou internos, como coesão. Além disso, cada atributo de qualidade do ramo superior é estruturado em outros atributos de nível inferior chamados de critérios, que são mais fáceis de medir do que os básicos. Esses critérios de qualidade, também chamados de subfatores, podem ser atributos internos. Finalmente, o terceiro nível envolve atributos diretamente mensuráveis chamados métricas e usados para medir os critérios de nível inferior (AMARA; RABAI, 2019).

3.3.3.1 Modelo de Qualidade de McCall

Proposto em 1977, este modelo busca considerar os atributos de qualidade de software a partir da perspectiva de desenvolvedores e usuários. É estruturado em três níveis principais, como observado na Figura 16, com 11 atributos de qualidade que descrevem o ponto de vista de usuários, e 23 critérios que refletem o ponto de vista de um desenvolvedor. Além disso, ainda conta com 43 métricas para medir esses critérios (AMARA; RABAI, 2019).

Uma vantagem apresentada por esse modelo é identificar as métricas apropriadas para diferentes características. Ele envolve a maior parte dos atributos de qualidade, como mencionado anteriormente, tanto a perspectiva do usuário como a do desenvolvedor. Entretanto, a característica de funcionalidade não é considerada. Isso também ocorre com características de hardware (AMARA; RABAI, 2019).

Figura 13 – Modelo de McCall.



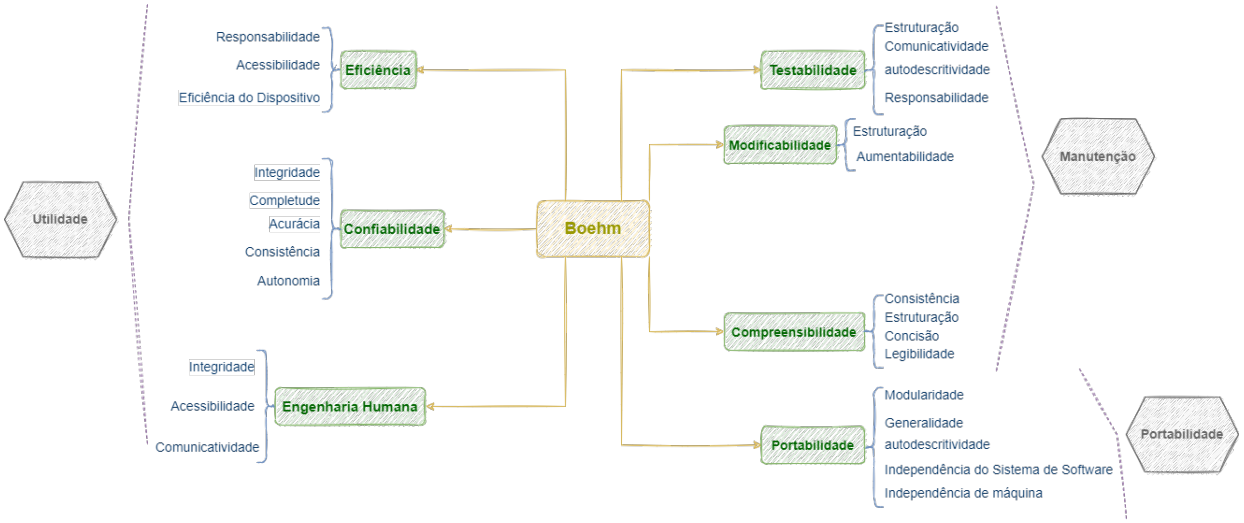
Fonte: Reprodução própria

3.3.3.2 Modelo de Qualidade de Boehm

Sugerido por Barry Boehm em 1978, propõe uma forma quantitativa de medição da qualidade de software e seus subfatores. Também definido em três níveis, temos: o nível superior composto por três atributos, o intermediário consistindo de 7 características e o nível inferior constituído de 15 características (AMARA; RABAI, 2019).

A adição de atributos como "engenharia humana"e "compreensibilidade"pode ser considerada uma vantagem. Assim como a inclusão da perspectiva do desenvolvedor e necessidade do usuário. Todavia, esse modelo (descrito na Figura 14) não identifica como a medição das característica de qualidade deve ocorrer (AMARA; RABAI, 2019).

Figura 14 – Modelo de Boehm.



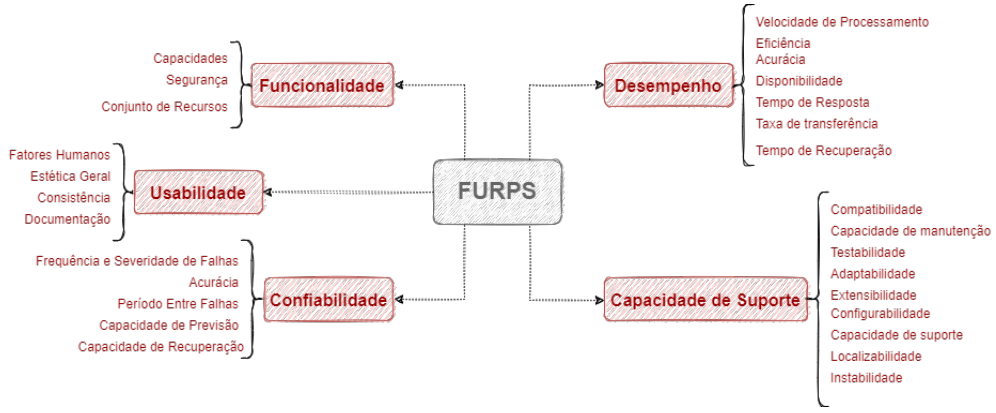
Fonte: Reprodução própria

3.3.3.3 Modelo de Qualidade FURPS

Foi sugerido pela primeira vez em 1987 por Hewlett Packard e Robert Grady. Seu nome é um acrônimo com as iniciais de cinco características: Funcionalidade, Usabilidade, Confiabilidade (*"Reliability"*), Desempenho (*"Performance"*) e Capacidade de Suporte (*"Supportability"*). Consiste em dois grupos de requerimentos, sendo o primeiro grupo composto por requerimentos funcionais e o segundo grupo composto por requisitos não-funcionais como pode ser observado a partir da Figura 15 (AMARA; RABAI, 2019).

Sua vantagem reside na adoção do atributo de desempenho, que não é mencionado nos modelos de McCall e Boehm. Entretanto, não considera a perspectiva do desenvolvedor por não incluir os atributos de capacidade de manutenção e reusabilidade de *features* (AMARA; RABAI, 2019).

Figura 15 – Modelo FURPS.



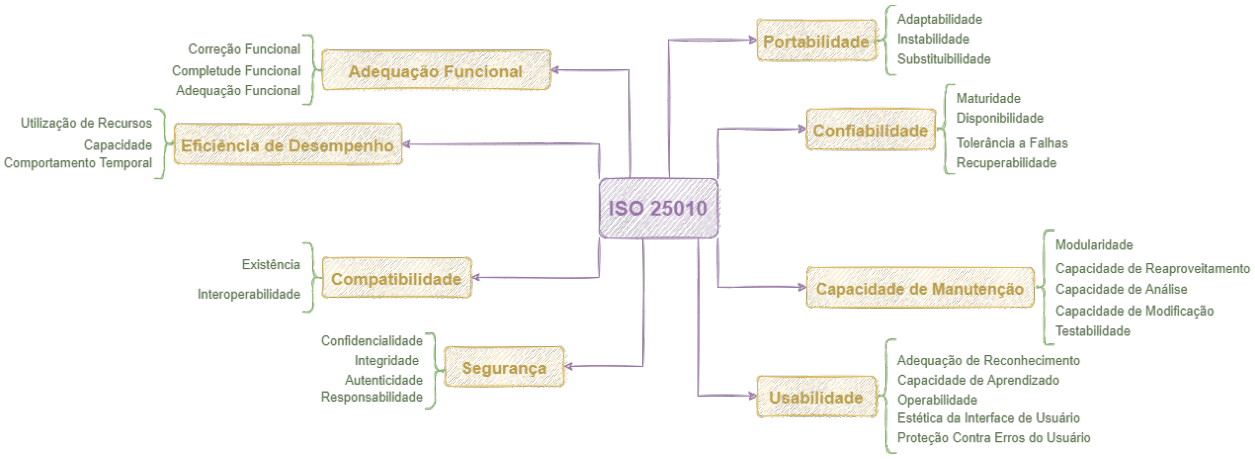
Fonte: Reprodução própria

3.3.3.4 Modelo de Qualidade ISO 25010 e ISO 9126

O modelo ISO 9126 foi publicado pela ISO em 1991 como o padrão de qualidade de terminologia de características. Em 2001, uma versão expandida foi lançada, contendo os modelos de qualidade ISO (ISO/IEC 9126-1 2003) e propôs sua medição. Após 2011, a ISO/IEC 25010 substituiu a ISO/IEC 9126-1 (AMARA; RABAI, 2019).

Dois diferentes modelos são propostos e classificados para objetivos específicos: um deles está ligado ao uso do software, enquanto o outro específica a sua produção. Neste trabalho apenas se trata deste último em que se enquadra o modelo de qualidade de produto ISO 25010 composto de oito características básicas conforme apresentado na Figura 4 (ISO/IEC 25010 2011). Este modelo (Figura 16) adiciona duas características à ISO 9126-1 (ISO/IEC 9126-1 2003) que são segurança e compatibilidade (AMARA; RABAI, 2019).

Figura 16 – Modelo ISO 25010.



Fonte: Reprodução própria

O modelo ISO 25010 considera características de segurança que o tornam o mais relevante entre os dois, além de distinguir compatibilidade de portabilidade. Dito isto, similar a ISO 9126-1, não apresenta uma descrição detalhada de medições e também não possui características conectadas as métricas (AMARA; RABAI, 2019).

3.4 Abordagens de automação para testes E2E em aplicações WEB

Segundo Leotta et al. (2016), para uma solução de automação de testes de ponta-a-ponta em aplicações web, existem basicamente dois critérios ortogonais para classificar a abordagem utilizada:

- Implementação de *scripts* de teste;
- Localização de elementos em páginas web.

O Quadro 1 a seguir expõe uma forma de classificação com os critérios mencionados acima, sendo possível aplicar a ferramentas de automação existentes:

Quadro 1 – Classificação de abordagens para testes E2E e ferramentas.

				Como desenvolver casos de teste	
				Captura-Replay	Programável
Como localizar elementos web	Geração	3º	Baseado em Visual		
		2º	Baseado em DOM		
		1º	Baseado em Coordenadas		

Fonte: (LEOTTA et al., 2016) traduzido pela autora.

3.4.1 Localização de itens e estratégias

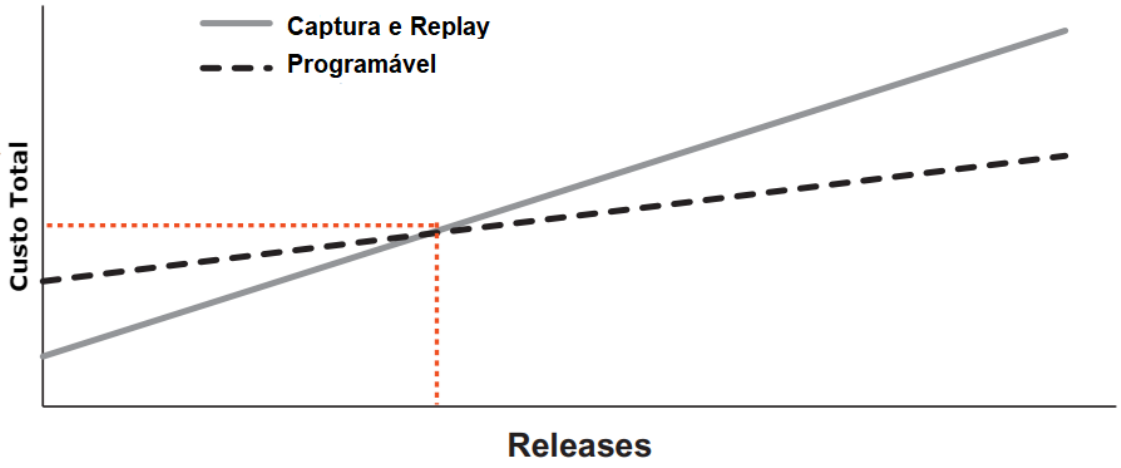
A estratégia de *"captura-replay"* consiste em gravar as ações performadas por um testador na GUI da aplicação, com posterior geração de script de teste que reproduz de maneira não assistida estas ações. Por outro lado, a abordagem *"programável"* tem como objetivo unificar o teste web às técnicas de teste tradicional (onde scripts são artefatos de software em si mesmos que desenvolvedores escrevem), tudo isso com a ajuda de*frameworks* de teste (LEOTTA et al., 2016).

A partir disso, Leotta et al. (2016) descreve três critérios utilizados para localização de componentes em uma página web:

- **Localização por coordenadas:** A ferramenta utilizada para implementar essa abordagem apenas grava as coordenadas dos elementos na página e utiliza essa informação para localizar os mesmos durante a execução não assistida do caso de teste. Por produzir scripts de teste extremamente frágeis, é considerada uma abordagem obsoleta.
- **Localização baseada em DOM:** Ferramentas que trabalham com esse tipo de localização de itens se utilizam da informação contida no *Document Object Model* (DOM). A partir disso, existem muitas formas de utilizar essa informação para encontrar os componentes: valores de atributo (id, nome e classe), o nome da tag do elemento, o texto apresentado em um hiperlink, expressões de CSS e XPath.
- **Localização por Visualização:** As ferramentas capazes dessa estratégia de localização emergiram em meados da década de 2010. Se utilizam de técnicas de reconhecimento de imagem para identificar componentes e controlar a GUI.

Em seu trabalho, Leotta et al. (2016) ainda verifica empiricamente o *trade-off* entre as estratégias de Captura-Replay (CR) e a abordagem programável no que toca a elevação do custo total de manutenção de suítes de teste utilizando as duas abordagens, como é possível de ser conferir a partir da Figura 17:

Figura 17 – Evolução de custos para casos de teste para estratégia C&R e programável.



Fonte: (LEOTTA et al., 2016) traduzido pela autora.

3.5 Análise das ferramentas selecionadas

O padrão internacional ISO 25010 fornece um estilo de avaliação de qualidade das características presentes em um produto de software. Fornece, como mencionado anteriormente, um conjunto de características e sub-características úteis para essa análise (ver Figura 16).

As características de "Adequação Funcional"tem seu foco na habilidade que a ferramenta de automação de testes possui de cumprir os requerimentos específicos de um projeto, executando seu papel de acelerador do processo de entrega de valor da melhor maneira possível. Sendo, dessa forma, essencial a validação que a ferramenta suporta a categoria de testes, a *stack* tecnológica do projeto, bem como interfaces e integrações com APIs e bases de dado.

Seguindo o modelo de avaliação ISO 25010, se faz necessária a análise das características relativas a "Usabilidade". Dentre elas: curva de aprendizagem, facilidade de uso devem ser observadas. Uma ferramenta considerada amigável precisa de uma interface que leva em conta os conceitos de experiência de usuário, assim como ser bem documentada e simplificar o processo de criação, gerenciamento e execução dos casos de teste.

No que toca a "Eficiência de Desempenho", o necessário é verificar o consumo de recursos pela ferramenta de automação. Levando em conta o fato que uma ferramenta eficiente deveria executar cenários de teste sem períodos de espera significantes, consumindo o mínimo de recursos, de maneira escalável para abranger suítes de teste complexas.

A "Capacidade de Manutenção"foca na facilidade de manter e evoluir um projeto realizado com a utilização de uma ferramenta de automação de testes. Uma ferramenta considerada boa nesse aspecto deve possuir capacidade de criação, customização, reutilização de scripts. Além disso, precisa prover o usuário com capacidades de integração a bibliotecas ou *frameworks* de teste.

Por sua vez, características voltadas a "Confiabilidade"são voltadas a estabilidade e robustez da ferramenta de software. Onde entende-se como uma ferramenta confiável aquela que: consegue suportar um volume auto de execuções sem quebrar, providenciando uma maneira adequada de lidar com falhas, mecanismos de recuperação. Tudo isso minimizando falsos positivos (ou negativos) nos resultados de teste.

Voltando o olhar para característica de "Compatibilidade", é possível compreendê-la como a capacidade de integração com outras ferramentas e sistemas. Outro ponto importante é a fácil adaptação a estratégias de *"Integração Contínua"* ou *"pipeline de DevOps"*. Por fim, é ideal que a ferramenta tenha uma boa cobertura de características relativas a "Portabilidade". Nessa categoria, é interessante que a ferramenta consiga suportar um conjunto de sistemas operacionais, navegadores.

Quadro 2 – Características das ferramentas selecionadas.

	Selenium IDE	**Cypress**	**Robot**	**Ginger**
Linguagem	Javascript	Javascript	Python	.NET
Sistemas operacionais	Windows Linux Mac OS	Windows Linux Mac OS	Windows Linux Mac OS Unix iOS Android	Windows Linux Unix Android Mac OS iOS DOS
Tipo	Código Aberto	Código Aberto	Código Aberto	Código Aberto
Versão inicial	2018	v0.3.3 (2015)	v2.0 (2008)	v3.7 (2017)
Licença	Apache License 2.0	MIT License	Apache License 2.0	Apache License 2.0
Custo	Gratuito	Gratuito	Gratuito	Gratuito
Status de desenvolvimento	Ativo	Ativo	Ativo	Ativo
Linguagens de script	Selenese	Javascript	Python Java .NET	Codeless
Navegadores suportados	Chrome Firefox Edge	Chrome Firefox Webkit	Chrome Firefox Safari Edge IE	Chrome Firefox Safari Edge IE
Aplicações passíveis de teste	Web	Web Mobile apps	Desktop Web	Desktop Web Mobile apps API/Web services Bases de dados
Habilidades em programação	Irrelevante	Necessário	Necessário	Irrelevante
Curva de aprendizagem	Baixa	Média	Média	Baixa
Facilidade de instalação e uso	Fácil	Média	Alta	Fácil
Tempo de criação de script	Baixo	Médio	Médio	Baixo
Integração DevOps/ALM	Não	Sim	Sim	Sim
Integração Contínua	Não	Sim	Sim	Sim
Localização de objetos	Ids Xpath	Ids Xpath	Ids Custom Locator	Xpath Id Custom Locator Shape
Estatísticas de teste	Não	Sim	Sim	Sim
Teste baseado em imagens	Não	Não	Não	Sim

Fonte: Reprodução própria.

A partir da análise do conteúdo do Quadro 2, bem como por meio do resgate do conceito de "Critérios Ortogonais de abordagem"desenvolvido por (LEOTTA et al., 2016). É possível verificar que, das ferramentas selecionadas, o Ginger pode ser considerado a ferramenta mais versátil e poderosa. Isso ocorre muito em função de sua capacidade de integração e

compatibilidade com diferentes sistemas operacionais e navegadores (além possibilidade de utilizar técnicas de localização de elementos visuais, consideradas de terceira geração) sem exigir habilidades em programação por parte do testador devido a sua natureza *codeless*.

4 Testes de Ponta-A-Ponta e o Mercado

Testes de ponta-a-ponta (E2E, do inglês *"End-to-End"*) são os mais simples de compreender, intuitivamente se deduz. Em uma aplicação *front-end*, um teste E2E automatiza a execução de um fluxo através de um navegador web para verificar se funções específicas estão funcionando corretamente. O teste deve simular a experiência de um usuário real: abrir o navegador, visitar a *url* da aplicação e navegar entre os componentes (EMMANUEL, 2019).

Dessa forma, é possível entender os testes E2E como uma garantia da sanidade e integridade de um sistema. Na atualidade, uma aplicação é composta de diversos componentes menores codificados com base em requerimentos próprios. O teste isolado dos mesmos pode garantir a estabilidade daquela *feature*, mas não garante que o sistema atende aos requisitos (ACKERSON; SAJJAD, 2022).

Considerando a pirâmide de testes, testes de ponta-a-ponta estão logo abaixo dos testes de interface de usuário (UI, do inglês *"User Interface"*), pois são mais custosos no tocante a recursos econômicos e humanos, bem como mais sensíveis a alterações no sistema. Em contrapartida, oferecem muito mais confiabilidade pois refletem - como mencionado anteriormente - o comportamento do sistema (ACKERSON; SAJJAD, 2022).

De certa forma, os testes E2E exemplificam o papel do teste de software em um mercado em que a utilização de estratégias de *DevOps* prevalece de maneira marcante. Isso ocorre por levarem em conta a conexão inerente entre como software é construído e utilizado. Ao garantir aos desenvolvedores uma visão de como alterações impactam a jornada do usuário final, os mesmos tornam-se melhores equipados para entregar valor a esses usuários (HUGHES, 2023).

4.1 O lado vantajoso dos testes E2E

Softwares comerciais sofrem atualizações periodicamente, portanto, testes contínuos devem ocorrer para garantir a estabilidade do produto final. Cada categoria de testes possui seu propósito intrínseco, sendo considerado um bom ponto de partida seguir a pirâmide de testes (GEORGIAN, 2021).

Os testes de ponta-a-ponta se encontram no topo da pirâmide de testes. São capazes de cobrir a maior parte das histórias de usuários com um conjunto reduzido de casos de teste. Isso devido ao aspecto de integração de componentes presente nessa estratégia de controle de qualidade (ACKERSON; SAJJAD, 2022).

Cada camada da pirâmide de testes (indo da base até o topo) representa: uma redução na velocidade, potencialização do escopo, complexidade e custo de manutenção. Por isso

é importante ressaltar que os testes E2E não tornam outras etapas de teste dispensáveis (GEORGIAN, 2021).

Idealmente, os defeitos precisam ser localizados o mais próximo da base da pirâmide de teste possível. Permitindo que os testes de ponta-a-ponta cuidem da validação de fluxo de trabalho entre os componentes do sistema (GEORGIAN, 2021).

4.2 O que leva ao declínio de suítes de testes E2E na indústria

Segundo Fogg (2020), é possível dividir as histórias comerciais de fracasso de suítes E2E em duas grandes categorias:

- **Suítes de teste com crescimento desordenado:** tornaram-se tão grandes, lentas e com tantos falsos positivos a cada execução que seus resultados são simplesmente irrelevantes as entregas do projeto. Tornam-se um fardo, um artefato indesejado que só traz transtornos aos times de qualidade e desenvolvimento.
- **Suítes de teste com manutenção defasada:** não são atualizadas entre as entregas em produção devido a uma falta de recursos humanos, tornando-se obsoletas. Os fluxos codificados na suíte original já não são capazes de refletir a experiência de usuário.

Na indústria, ainda se compreende as suítes de testes E2E como uma estratégia viável que agrega valor no produto final. Entretanto, não é uma experiência universal como demonstrado pela *fintech* brasileira Nubank. Dentro do referido estudo de caso, Freire (2022) identificou alguns pontos de dor que reduzem o valor entregue por esse tipo de conjunto de teste para essa organização:

- **Tempo de espera:** o tempo de execução da suíte aumenta a cada iteração, levando a um prolongado tempo de espera pelos resultados.
- **Falta de confiança:** a instabilidade dos casos de teste leva uma necessidade de reexecução da suíte de testes, para garantir que algo não é um falso negativo.
- **Custo de manutenção:** a corrupção de dados de teste devido a mudanças manuais na suíte de testes criaram um ambiente desafiador para garantir a manutenção da suíte;
- ***Debug* nebuloso:** devido a dependência de comunicação assíncrona entre componentes, as falhas encontradas pela suíte de teste não apontavam corretamente a origem do problema. Aumentando, dessa forma, o tempo de *debugging*.
- **Entrega lenta:** os enfileiramentos de commits para execução na suíte de testes E2E resultaram em períodos longos entre as entregas de valor.

- **Ineficiência e ineficácia:** de cada 1000 execuções da suíte de testes E2E, apenas 42 falhas e um bug eram encontrados. Ainda assim, após a entrega, bugs ocorriam em produção.

Dentro das operações da *Just Eat* (gigante estado-unidense do ramo de entrega a domicílio de alimentos), os pontos de dor foram localizados nas etapas anteriores a execução de uma suíte desse tipo. Segundo Johnston (2016), apesar de continuar empregando essa estratégia de testes, a *Just Eat* optou por reduzir significativamente a quantidade de testes de ponta-a-ponta em sua abordagem de controle de qualidade. Esse movimento ocorreu em função dos seguintes aspectos:

- O conhecimento do sistema necessário para que o engenheiro de testes seja capaz de desenhar e programar um caso de teste E2E de maneira eficaz;
- A complexidade existente em testar caminhos negativos quando existem integrações entre banco de dados ou APIs;
- Sua fragilidade, devido a dependência existente entre alterações de interface e alterações nos scripts da suíte de testes E2E

O trabalho investigativo de Johnston (2016) assim como análise da equipe de garantia de qualidade descrita por Freire (2022), levaram a alterações na cultura de controle de qualidade das referidas organizações. De forma que ambas adotaram novas abordagens de testes para assegurar continuidade de entregas robustas a seus respectivos clientes.

5 Considerações Finais

O desenvolvimento do presente trabalho possibilitou a análise de um conjunto de ferramentas de automação, em um contexto de testes de ponta-a-ponta para web. Além disso, também permitiu a investigação do paradigma atual da indústria em relação a essa categoria de testes.

Devido a crescente complexidade de sistemas de software modernos, a automação de testes aparece como um recurso indispensável para garantir a melhora da eficiência, acurácia e confiabilidade destes projetos. As vantagens resultantes da correta seleção de uma ferramenta de automação de testes se traduzem em redução de custos, ganho de desempenho e segurança.

Da mesma maneira, os testes de ponta-a-ponta se provam instrumentais para a verificação da integração entre os diferentes componentes destes complexos sistemas de software modernos. Auxiliando no trabalho de manter uma experiência de usuário consistente, além de identificar os desafios e fragilidades de um projeto de software.

A partir da análise das ferramentas realizada utilizando o modelo ISO 25010, foi possível constatar que (em um contexto de automação de testes E2E para web) a ferramenta *Ginger* mostrou-se mais versátil. Ademais, demonstrou-se a mais amigável e de fácil instalação em relação aos recursos disponíveis.

Em relação a posição da indústria no que toca aos testes de ponta-a-ponta, é possível argumentar que - apesar das vantagens citadas anteriormente que derivam dessa estratégia de teste -, os testes E2E costumam ser implementados de maneira errônea. Muitas vezes tendo um número exacerbado de cenários a cobrir, com uma manutenção mínima das suítes de automação. O que resulta no aumento do número de falhas (além de falsos negativos e positivos), e acarreta no abandono das mesmas.

A análise das ferramentas a partir do modelo ISO 25010 (aliado ao conceito de "Critérios Ortogonais"para classificação de ferramentas em relação ao estilo de localização de elementos em tela), atendeu as expectativas de maneira satisfatória. Da mesma maneira, foi possível compreender a partir da pesquisa realizada a profunda desarmonia existente em relação a visão dos testes de ponta-a-ponta por parte do mercado.

Referências

ACKERSON, Z.; SAJJAD, D. **Why End-to-End (E2E) Testing Is Often Good Enough**. 2022. Disponível em: <https://semaphoreci.com/blog/e2e-testing>.

AMARA, D.; RABAI, L. B. A. Software quality measurement: State of the art. In: _____. [S.l.: s.n.], 2019. p. 150–181. ISBN 9781522557944.

AMDOCS. **Ginger by Amdocs Help**. [S.l.], 2023. Disponível em: <https://ginger-automation.github.io/Ginger-Web-Help/assets/Ginger_By_Amdocs/Getting_Started/What_is_Ginger.htm>.

ANGMO, R.; SHARMA, M. **Performance evaluation of web based automation testing tools**. 2014. 731-735 p.

ATESOGULLARI, D.; MISHRA, A. Automation testing tools: A comparative view. **International Journal of Information and Computer Security**, v. 12, p. 63–76, 12 2020.

BADAL, L.; GRUNLER, D. Automated end-to-end testing: Useful practice or frustrating time sink? v. 1, p. 01–07, 04 2021.

BUXTON, J.; RANDELL, B. **Software engineering techniques - homepages.cs.ncl.ac.uk**. NATO Science Committee, 1970. Disponível em: <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF>.

CERQUOZZI, R.; DECOUTERE, W.; DUSSA-ZIEGER, K.; RIVERIN, J.-F.; HRYSZKO, A.; KLONK, M.; PILAETEN, M.; POSTHUMA, M.; REID, S.; COSQUER, E. Riou du; ROMAN, A.; STAPP, L.; ULRICH, S.; ZAKARIA, E. [s.n.], 2023. Disponível em: <https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB_CTFL_Syllabus-v4.0.pdf>.

CHELUVARAJU, B.; NAGAL, K.; PASALA, A. Mining software revision history using advanced social network analysis. **2012 19th Asia-Pacific Software Engineering Conference**, p. 717–720, 2012.

CYPRESS. **Cypress Documentation**. [S.l.], 2023. Disponível em: <https://docs.cypress.io/guides/overview/why-cypress>.

DAM, K. The future of testing: Digging in the past of software testing and unearthing the future. In: _____. [S.l.: s.n.], 2020. p. 197–205. ISBN 978-3-030-29508-0.

ELM, D. **Dominic Elm - Cypress: The future of E2E testing**. 2019. Disponível em: <https://www.youtube.com/watch?v=pXyBligMMr0>.

EMMANUEL, O. **The Problem with End-to-End Tests**. 2019. Disponível em: <https://levelup.gitconnected.com/the-problem-with-end-to-end-tests-65509df4bc7a>.

FOGG, E. **Why E2E Test Suites Fail—And How to Avoid Breakdowns**. 2020. Disponível em: <https://prodperfect.com/blog/end-to-end-testing/why-test-suites-fail-and-how-to-avoid-breakdowns/>.

FREIRE, A. **Why we killed our end-to-end test suite**. Nubank, 2022. Disponível em: <https://building.nubank.com.br/why-we-killed-our-end-to-end-test-suite/>.

GEORGIAN, S. **What is end-to-end testing and when should you use it?** freeCodeCamp.org, 2021. Disponível em: <https://www.freecodecamp.org/news/end-to-end-testing-tutorial/>.

HUGHES, B. **In Defense of Comprehensive End-to-End Testing**. 2023. Disponível em: <https://www.mabl.com/blog/in-defense-of-comprehensive-end-to-end-testing>.

JOHNSTON, B. **Tech blog**. 2016. Disponível em: <https://tech.justeattakeaway.com/2016/10/04/why-fewer-end-to-end-tests/>.

LEOTTA, M.; CLERISSI, D.; RICCA, F.; TONELLA, P. Chapter five - approaches and tools for automated end-to-end web testing. In: MEMON, A. (Ed.). Elsevier, 2016, (Advances in Computers, v. 101). p. 193–237. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0065245815000686>.

PINHEIRO, P. Tipos de frameworks de automação de testes. **LinkedIn**, Feb 2023. Disponível em: <https://www.linkedin.com/pulse/tipos-de-frameworks-automa%C3%A7%C3%A3o-testes-paulo-pinheiro>.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software - 8ª Edição**. [s.n.], 2016. ISBN 9788580555349. Disponível em: <https://books.google.com.br/books?id=wexzCwAAQBAJ>.

RAVIART, D. **Amdocs Simplifies the Test Case-to-Test Script Process with Ginger**. NelsonHall, 2019. Disponível em: <https://research.nelson-hall.com/blogs/?avpage-views=blog&type=post&post_id=976>.

ROBOT. **Robot framework user guide**. [S.l.], 2023. Disponível em: <https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html>.

SELENIUM. **Selenium - Getting Started**. [S.l.], 2019. Disponível em: <https://www.selenium.dev/selenium-ide/docs/en/introduction/getting-started>.

SMART, J.; MOLAK, J. **BDD in Action, Second Edition: Behavior-Driven Development for the Whole Software Lifecycle**. Manning, 2023. ISBN 9781617297533. Disponível em: <https://books.google.com.br/books?id=hIK3EAAAQBAJ>.

SOMMERVILLE, I. **Engenharia de software**. Pearson Prentice Hall, 2011. ISBN 9788579361081. Disponível em: <https://books.google.com.br/books?id=H4u5ygAACAAJ>.

TECHGEEKNEXT. **RobotFramework tutorial - robotframework architecture [updated] 2023**. TechGeekNext, 2023. Disponível em: <https://www.techgeeknext.com/robot-framework-architecture>.