A importância dos testes de software: O que testar, como testar e o porquê testar

Edson Rodrigo de Oliveira[1]

[1]Faculdade de Computação – Universidade Federal do Pará (UFPA)
Castanhal - PA - Brazil

rodrigocode4@gmail.com

Abstract. *This article addresses the importance of software testing, exploring what should be tested, how to conduct tests, and why they are essential in the software development process. The main aspects to be considered when testing software, different testing approaches and techniques, as well as the benefits that testing brings to software quality and reliability, will be discussed.*

Resumo. *Este artigo aborda a importância dos testes de software, explorando o que deve ser testado, como realizar os testes e por que eles são essenciais no processo de desenvolvimento de software. Serão discutidos os principais aspectos a serem considerados ao testar um software, as diferentes abordagens e técnicas de teste, bem como os benefícios que os testes proporcionam para a qualidade e confiabilidade do software.*

1. Introdução

Os testes de software desempenham um papel fundamental no desenvolvimento de produtos tecnológicos confiáveis e de qualidade. Ao garantir que um software atenda aos requisitos estabelecidos e funcione corretamente, os testes contribuem para a satisfação do cliente, a redução de custos e a reputação das organizações [Bookman; Offutt, 2022].

A crescente complexidade das aplicações de software e a demanda por soluções inovadoras aumentaram a importância dos testes na indústria de desenvolvimento de software. Os testes permitem que as equipes identifiquem e corrijam erros, defeitos e falhas, garantindo que o software funcione de maneira adequada, cumpra seu propósito e fazendo com que os erros corrigidos não se repitam [Pressman, 2016].

Durante o processo de teste de software, é essencial definir claramente o escopo dos testes, identificando as funcionalidades e requisitos críticos que precisam ser verificados [Silva, 2023]. Os testes de funcionalidade são realizados para garantir que o software atenda aos requisitos funcionais estabelecidos, validando se todas as funcionalidades estão operando corretamente, de acordo com as regras de negócio demandadas e estabelecidas e produzindo os resultados esperados.

Ao adotar as melhores práticas e técnicas de teste, as equipes de desenvolvimento podem obter uma visão abrangente da qualidade do software [Mosconi, 2022]. Os testes permitem que os problemas sejam identificados precocemente, evitando retrabalho e reduzindo os custos de correção. Além disso, os testes garantem a conformidade com os requisitos, a melhoria contínua do software e a satisfação do cliente.

1.1. Objetivos

O objetivo deste artigo é explorar a importância dos testes de software, abordando os principais aspectos a serem considerados ao testar um software, as diferentes abordagens e técnicas de teste, bem como os benefícios que os testes proporcionam para a qualidade e confiabilidade do software.

2. Definição de escopo e requisitos dos testes

2.1. Definição de escopo: identificação das funcionalidades e requisitos críticos

No processo de teste de software, é essencial estabelecer claramente o escopo dos testes. Isso envolve a identificação das funcionalidades e requisitos críticos que serão abordados durante o processo de testagem. Ao definir o escopo, os testes são concentrados nas áreas consideradas mais importantes e que possam impactar diretamente a qualidade do software [Juristo; Vegas; Apa, 2013].

Para Sommerville [2019], qualidade de software está intrinsecamente relacionada à capacidade de um sistema atender de maneira eficaz e eficiente aos requisitos definidos durante o processo de engenharia de software. Ele destaca que a qualidade do software é diretamente influenciada pela qualidade dos requisitos, enfatizando a importância da clareza, completude e compreensibilidade na documentação desses requisitos desde as fases iniciais do desenvolvimento. A qualidade, nesse contexto, é alcançada por meio de práticas rigorosas de engenharia de requisitos, validação, gestão de mudanças e rastreabilidade, assegurando que o produto final não apenas atenda às expectativas dos stakeholders, mas também mantenha a integridade e coerência ao longo do ciclo de vida do software.

A primeira etapa na definição de escopo é a compreensão das funcionalidades do software que serão testadas. Isso envolve identificar todas as principais características e módulos do software que precisam ser abordados durante os testes. As funcionalidades podem ser derivadas dos requisitos estabelecidos pelo cliente, especificações do projeto ou documentação técnica. É importante garantir que todas as funcionalidades sejam incluídas no escopo dos testes para uma cobertura abrangente.

Além disso, é crucial identificar os requisitos críticos que devem ser atendidos pelo software. Esses requisitos geralmente são relacionados à confiabilidade, qualidade ou outras áreas que são de extrema importância para o sucesso do software. Ao identificar os requisitos críticos, pode-se concentrar em testar especificamente esses aspectos essenciais para garantir que o software cumpra os padrões e as expectativas estabelecidas [Cordeiro; Freitas, 2012].

A definição de escopo também deve levar em consideração a complexidade do software. Se o software possui diferentes camadas, módulos ou componentes, é importante identificar quais partes serão testadas e como a interação entre elas será abordada. Isso pode envolver a priorização de áreas críticas, bem como a identificação de dependências e integrações que precisam ser testadas em conjunto. Além disso, a definição de escopo também deve considerar restrições de recursos, como limitações de tempo e orçamento, sendo necessário encontrar um equilíbrio entre a abrangência dos

testes e os recursos disponíveis, para garantir a eficácia, eficiência e a viabilidade dos esforços de teste [Oliveira, 2012].

Ao identificar claramente as funcionalidades e requisitos críticos e estabelecer o escopo adequado para os testes, as equipes de desenvolvimento devem direcionar seus esforços aos testes de maneira eficiente e eficaz. Isso garante uma cobertura adequada das áreas mais importantes do software e contribui para a identificação precoce de problemas e a garantia da qualidade do produto final.

2.2. Testes de Funcionalidade: Verificação dos Requisitos e Comportamentos esperados

Um dos aspectos cruciais no processo de teste de software é a realização de testes de funcionalidade. Esses testes têm como objetivo principal verificar se o software atende aos requisitos funcionais estabelecidos e se as funcionalidades operam corretamente, produzindo os resultados esperados [Sommerville, 2019]. Eles desempenham um papel fundamental na garantia da qualidade do software e na entrega de um produto confiável aos usuários finais.

Os testes de funcionalidade são baseados nos requisitos funcionais definidos para o software. Esses requisitos são especificações detalhadas das funcionalidades que o software deve oferecer, descrevendo o comportamento esperado em diferentes cenários de uso. Ao realizar os testes de funcionalidade, as equipes de desenvolvimento verificam se o software cumpre essas especificações e se as funcionalidades estão implementadas corretamente.

A verificação dos requisitos é uma parte fundamental dos testes de funcionalidade, pois envolve garantir que o software atenda aos requisitos funcionais estabelecidos. Estes, que por sua vez, descrevem o que o software deve fazer e as funcionalidades que devem ser implementadas para atender às necessidades dos usuários e aos objetivos do projeto [Sommerville, 2019]. Assim sendo, durante a verificação dos requisitos, as equipes de desenvolvimento analisam cuidadosamente os requisitos estabelecidos para o software e criam casos de teste específicos para cada um deles. Esses casos de teste são projetados para validar se o software está realmente cumprindo as especificações definidas nos requisitos.

Para cada requisito funcional, é necessário verificar se o software executa a funcionalidade esperada, se os resultados são corretos e se o comportamento é consistente com as expectativas. Isso pode envolver a verificação de várias condições, como entradas válidas e inválidas, valores limite, fluxos de trabalho complexos e possíveis cenários de erro.

Para Sommerville [2019], ao verificar os requisitos, é importante considerar não apenas as funcionalidades individualmente, mas também como elas interagem entre si. O software deve ser capaz de executar todas as funcionalidades em conjunto sem conflitos ou comportamentos inesperados. A verificação dos requisitos também pode envolver a validação dos requisitos não funcionais, como desempenho, confiabilidade e clareza. Esses requisitos complementam os requisitos funcionais, definindo critérios adicionais para a qualidade e o desempenho do software.

Durante a escrita dos testes, é importante garantir que tanto os requisitos funcionais quanto os não funcionais sejam adequadamente verificados. Uma abordagem comum para a verificação dos requisitos é a criação de uma matriz de rastreabilidade que segundo Espinha [2020], é uma ferramenta que explicita a relação direta dos requisitos entre si ou com os outros componentes do projeto. Assim, caso alguma alteração seja feita no projeto, sabe-se quais requisitos serão afetados com tal mudança. Isso permite um acompanhamento eficaz e rastreabilidade entre os requisitos e os testes realizados, garantindo que todos os requisitos sejam abordados e testados de forma adequada.

A verificação dos requisitos também pode incluir a revisão de documentação técnica, como especificações de requisitos, diagramas de casos de uso e documentos de design. Isso ajuda a garantir que todos os detalhes estejam corretamente implementados no software e que não haja lacunas, ruídos ou inconsistências entre as especificações e a funcionalidade real do sistema [Pinto, 2022].

Ao realizar a verificação dos requisitos, a escrita dos testes desempenham um papel importante e fundamental na garantia da qualidade, durabilidade e manutenibilidade do software, validando se o software atende aos requisitos estabelecidos, contribuindo para a entrega de um produto confiável, que atende às expectativas dos usuários e aos objetivos do projeto.

A primeira etapa nos testes de funcionalidade é a elaboração de casos de teste. Os casos de teste são cenários específicos que descrevem os passos a serem seguidos e os dados a serem utilizados para verificar o comportamento do software [Gomes, 2023]. Eles devem abranger diferentes situações, como entradas válidas e inválidas, valores limite e casos de uso complexos, garantindo uma cobertura adequada das funcionalidades.

Durante a execução dos casos de teste, as equipes de desenvolvimento verificam se o software opera de acordo com o esperado. Isso envolve a verificação das entradas fornecidas, o acompanhamento das saídas produzidas e a comparação dos resultados com os resultados esperados. Qualquer discrepância entre o comportamento real do software e o comportamento esperado é considerada um defeito (*bug*), que precisará ser corrigido.

Ao realizar os testes de funcionalidade, é essencial verificar se o software apresenta os comportamentos esperados, que referem-se às ações, resultados e interações que são previstos e definidos nos requisitos funcionais ou nas especificações do software. Além disso, durante os testes, as equipes de desenvolvimento avaliam cuidadosamente cada funcionalidade do software e verificam se ela se comporta conforme o esperado. Isso envolve executar os casos de teste e analisar se os resultados estão alinhados com as expectativas estabelecidas nos requisitos.

A verificação dos comportamentos esperados pode abranger diversos aspectos, como:

1. Entradas e saídas: Os testes devem verificar se as entradas fornecidas ao software produzem as saídas esperadas. Isso inclui a validação de entradas válidas e

inválidas, garantindo que o software responda corretamente a diferentes tipos de dados.

2. Fluxos de trabalho: Os testes devem abordar os diferentes fluxos de trabalho do software, verificando se as sequências de ações e etapas são executadas corretamente e produzem os resultados esperados em cada etapa do processo.

3. Respostas a eventos: O software pode ser projetado para responder a eventos específicos, como cliques em botões, seleções de menus ou notificações. Os testes devem validar se o software responde adequadamente a esses eventos e executa as ações esperadas em resposta a eles.

4. Comportamento em situações excepcionais: Os testes devem verificar como o software lida com situações excepcionais, como erros, condições de falha ou indisponibilidade de recursos. É importante garantir que o software exiba comportamentos apropriados nessas situações e forneça mensagens de erro claras e úteis para os usuários.

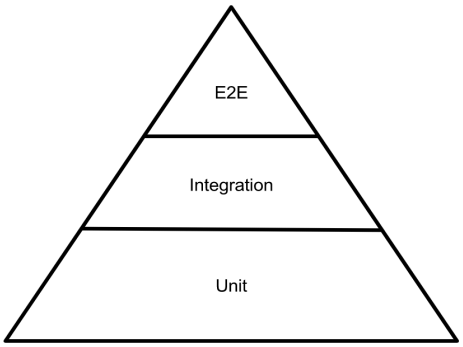
5. Integração com outros componentes: Se o software se integra a outros sistemas, bibliotecas ou serviços, os testes devem abordar a interação entre esses componentes e verificar se a integração ocorre conforme o esperado.

6. Conformidade com padrões e regulamentos: Em alguns casos, o software pode precisar cumprir padrões ou regulamentos específicos. Os testes devem validar se o software atende a esses requisitos e se comporta de acordo com as diretrizes estabelecidas.

Ao verificar os comportamentos esperados, é essencial ter um conhecimento aprofundado dos requisitos funcionais e das especificações do software [Sommerville, 2019]. Isso permite que as equipes de desenvolvimento identifiquem discrepâncias, erros lógicos ou comportamentos inesperados que podem ocorrer durante a execução do software. A documentação do software, desempenha um papel fundamental nessa verificação. Assim, ao comparar os resultados obtidos nos testes com as expectativas definidas na documentação de especificações, as equipes de desenvolvimento podem identificar e relatar qualquer desvio ou não conformidade.

Além disso, os testes de funcionalidade devem ser realizados em diferentes níveis de teste, como testes unitários, testes de integração e testes de ponta-a-ponta(*e2e*) [Figura 01]. Cada nível de teste tem sua própria abordagem e objetivo específico, contribuindo para uma verificação completa das funcionalidades em diferentes contextos.

Figura 01. Pirâmide de testes



Fonte: Adaptado de Reflect (2020)

Os testes de funcionalidade desempenham um papel essencial na validação do software e na garantia de que ele atenda às expectativas do cliente. Eles ajudam a identificar problemas de implementação, erros lógicos e comportamentos inesperados, permitindo que essas questões sejam corrigidas antes do lançamento do software e/ou de maneira mais assertiva dado que o software está em produção com comportamento inesperado que não tenha sido mapeado durante o processo de desenvolvimento do último lançamento.

3. Papéis e Estratégias

Estratégias de teste desempenham um papel crucial no processo de teste de um software. Elas envolvem a definição de abordagens e técnicas a serem utilizadas para garantir que o software seja testado de maneira adequada e eficiente. Duas estratégias comuns são os testes manuais e os testes automatizados [Sommerville, 2019].

3.1 Testes Manuais

Os testes manuais envolvem a execução de testes por seres humanos, que interagem diretamente com o software para verificar seu comportamento, funcionalidades e identificar possíveis problemas. Essa abordagem é geralmente utilizada no início do processo de teste e em cenários complexos que exigem a habilidade e o julgamento dos testadores. Os testes manuais são flexíveis, permitem a detecção de problemas de usabilidade e fornecem feedback imediato sobre o comportamento do software.

Contudo, os testes manuais têm algumas limitações e desafios que podem afetar a eficácia e eficiência do processo de teste [Cerruti, 2022]. Vamos discutir alguns problemas comuns relacionados aos testes manuais:

1. Suscetibilidade a erros humanos: Os testes manuais são realizados por seres humanos, o que significa que estão sujeitos a erros humanos. Os testadores podem cometer equívocos, perder detalhes importantes, esquecer cenários de teste ou ter lapsos de concentração. Isso pode resultar na falha em identificar problemas de software ou na falta de cobertura adequada durante os testes.

2. Repetitividade: Os testes manuais podem envolver tarefas repetitivas e monótonas, como preencher formulários, clicar em botões ou navegar em diferentes telas. A repetição constante dessas atividades pode levar à fadiga do testador e diminuir sua atenção, aumentando a probabilidade de erros. Além disso, o tédio decorrente da execução repetitiva de testes pode afetar a motivação e o desempenho dos testadores.

3. Limitações de tempo e recursos: Os testes manuais exigem tempo e recursos significativos. Dependendo do tamanho e complexidade do software, realizar testes manuais abrangentes pode ser demorado e exigir uma equipe de testadores dedicada. Além disso, os testes manuais podem ser custosos em termos de recursos humanos e financeiros, especialmente se forem necessárias várias iterações de teste ao longo do ciclo de desenvolvimento do software.

4. Escalabilidade limitada: Os testes manuais são difíceis de escalar quando o software cresce em tamanho e complexidade. Conforme o número de funcionalidades e casos de teste aumenta, torna-se desafiador para os testadores realizar manualmente todos os testes necessários dentro dos prazos e recursos disponíveis. Isso pode resultar em uma cobertura de teste insuficiente ou na necessidade de priorização seletiva dos casos de teste.

5. Dificuldade na simulação de cenários complexos: Alguns cenários de teste podem ser complexos ou difíceis de simular manualmente. Por exemplo, testar a interação do software com diferentes dispositivos, configurações de rede específicas ou volumes de dados massivos pode ser um desafio. Isso pode levar a lacunas na cobertura de teste ou à incapacidade de identificar problemas que ocorrem apenas nessas condições específicas.

3.2 Testes Automatizados

Os testes automatizados são uma abordagem em que os testes são executados por meio de ferramentas e scripts de automação [Aniche, 2015]. Eles são mais rápidos, repetíveis e escaláveis do que os testes manuais. Além disso, ajudam a reduzir a chance de erros humanos e permitem a execução de testes em larga escala. Os testes automatizados são particularmente úteis em cenários de desenvolvimento ágil, onde a entrega frequente de novas funcionalidades requer testes rápidos e confiáveis.

Em contraponto aos problemas dos testes manuais, os testes automatizados oferecem uma série de vantagens significativas. Vamos explorar algumas delas:

1. Precisão e Consistência: Os testes automatizados são executados por meio de scripts e ferramentas de automação, o que elimina a possibilidade de erros humanos. Uma vez que os testes são configurados corretamente, eles serão executados da mesma maneira todas as vezes, garantindo uma abordagem consistente de verificação do software.

2. Eficiência e Velocidade: Os testes automatizados são mais rápidos em comparação aos testes manuais. Eles podem ser executados em paralelo, em diferentes

ambientes ou configurações, permitindo uma cobertura de teste mais abrangente em um período de tempo mais curto. Além disso, os testes automatizados podem ser facilmente repetidos sempre que necessário, acelerando o processo de teste.

3. Cobertura Abrangente: Com os testes automatizados, é possível alcançar uma cobertura mais abrangente do software. Os casos de teste podem ser escritos para verificar diferentes cenários, fluxos de trabalho e funcionalidades em uma escala muito maior do que seria possível com os testes manuais. Isso resulta em uma detecção mais eficiente de bugs e problemas, proporcionando uma maior confiança na qualidade do software.

4. Escalabilidade: Os testes automatizados são altamente escaláveis. Eles podem ser facilmente estendidos para lidar com aumentos na complexidade do software, volume de dados e requisitos de teste. À medida que o software cresce, os testes automatizados podem acompanhar essa evolução, proporcionando uma cobertura abrangente e eficiente em diferentes estágios do desenvolvimento.

5. Reutilização de Testes: Os testes automatizados podem ser reutilizados em diferentes ciclos de desenvolvimento e versões do software. Uma vez que os testes são criados, eles podem ser executados repetidamente, garantindo que problemas já identificados não ocorram novamente. Isso economiza tempo e esforço, permitindo que a equipe de teste se concentre em novas funcionalidades e áreas críticas do software.

6. Integração Contínua: Os testes automatizados são um componente essencial da integração contínua e do processo de entrega contínua. Eles podem ser integrados aos pipelines de CI/CD (Integração Contínua/Entrega Contínua) para executar automaticamente os testes em cada mudança de código. Isso permite a detecção precoce de problemas e garante a estabilidade do software durante todo o ciclo de desenvolvimento.

7. Detecção de Regressões: Os testes automatizados são eficazes na detecção de regressões, ou seja, problemas que ocorrem quando uma nova alteração de software causa falhas em funcionalidades previamente funcionais. Com uma suíte de testes automatizados abrangente, é possível identificar rapidamente se uma alteração introduziu regressões no software, permitindo que os desenvolvedores corrijam os problemas imediatamente.

3.2.1 Tipos de Testes Automatizados

Dentro dos testes automatizados, existem três tipos principais que desempenham um papel fundamental no processo de teste [Sommerville, 2019]:

1. Testes de Unidade**:** Os testes de unidade focam na verificação do comportamento de pequenas partes individuais do software, como funções, métodos ou componentes isolados. Esses testes são escritos pelos desenvolvedores e são executados de forma independente para garantir que cada unidade do software esteja funcionando

corretamente. Os testes de unidade são rápidos, precisos e ajudam a identificar e corrigir problemas no nível do código.

2. Testes de Integração: Os testes de integração visam verificar a interação e a comunicação entre diferentes componentes ou módulos do software. Eles são projetados para identificar problemas que podem surgir quando várias partes do software são combinadas e testadas em conjunto. Os testes de integração podem ser executados em diferentes níveis, desde a integração de unidades individuais até a integração de subsistemas ou sistemas completos. Estes testes garantem a correta integração das partes do software e a interoperabilidade entre elas.

3. Testes *End-to-End (E2E)*: Os testes E2E simulam a interação completa de um usuário com o software, verificando se o fluxo completo de uma funcionalidade ou processo é executado corretamente. Esses testes são executados em um ambiente semelhante ao ambiente de produção e envolvem a simulação de ações do usuário, como cliques, preenchimento de formulários e navegação. Os testes E2E garantem que o software esteja funcionando corretamente em termos de fluxo de trabalho e comportamento esperado do usuário.

É importante ressaltar que a combinação de diferentes estratégias de teste, como testes manuais e automatizados, juntamente com uma seleção adequada dos tipos de testes, permite obter uma cobertura abrangente e eficaz durante o processo de teste do software. Cada estratégia e tipo de teste desempenha um papel complementar na identificação de problemas e na garantia da qualidade do software.

4. Ferramentas e Frameworks

Ferramentas de teste desempenham um papel fundamental na facilitação e automação dos processos de teste de software. Elas fornecem recursos e funcionalidades que ajudam os testadores a criar, executar e gerenciar testes de maneira eficiente.

Ao utilizar essas ferramentas de teste, os testadores podem aumentar sua eficiência, produtividade e precisão na execução dos testes de software. As ferramentas de teste automatizadas ajudam a acelerar o processo de teste, reduzir erros humanos e fornecer resultados mais consistentes. Além disso, elas permitem que os testadores se concentrem em atividades de teste mais complexas e críticas, como a análise de resultados e a exploração de cenários não triviais.

Podemos explorar o uso de frameworks e tecnologias como ferramentas de teste para aprimorar o processo de teste de software.

4.1 Frameworks

Frameworks de teste são estruturas que fornecem um conjunto de diretrizes, bibliotecas e funcionalidades pré-definidas para facilitar a criação e execução de testes automatizados. Eles oferecem uma abordagem padronizada e consistente para o desenvolvimento e manutenção de casos de teste. Alguns dos frameworks de testes de unidade e integração bem populares para são o JUnit para Java e o Jest para Javascript.

Esses frameworks simplificam a criação de casos de teste, a execução dos testes e a geração de relatórios de resultados.

4.1.1 Frameworks de Servidores Web

Os frameworks Spring, Nest.js e Fastify.js representam ferramentas poderosas e eficazes no cenário do desenvolvimento de software, cada um destacando-se em sua própria esfera tecnológica. O Spring, baseado em Java, é um ecossistema abrangente e robusto que oferece uma ampla gama de recursos para o desenvolvimento de aplicativos corporativos, desde a manipulação de injeção de dependência até a implementação de serviços web. Por sua vez, o Nest.js, construído sobre o ecossistema Node.js e TypeScript, adota uma abordagem modular e orientada a objetos, fornecendo uma estrutura escalável para o desenvolvimento de aplicativos server-side. Finalmente, o Fastify.js, centrado em performance, é um framework web leve para Node.js, projetado para fornecer velocidade e eficiência em operações intensivas de entrada/saída, tornando-se uma escolha ideal para aplicações que exigem alta performance e baixa latência. Cada um desses frameworks destaca-se por suas características distintas, oferecendo soluções especializadas para os desafios específicos encontrados no desenvolvimento moderno de software.

4.1.2 JUnit

O JUnit é um framework de teste para a linguagem de programação Java. Ele fornece um ambiente para escrever, organizar e executar testes automatizados para garantir a qualidade e a integridade do código-fonte durante o desenvolvimento de software. O JUnit facilita a criação de casos de teste, a execução controlada dos testes e a verificação automatizada dos resultados, tornando-o uma ferramenta essencial para práticas de desenvolvimento ágil e testes contínuos, Figura 02.

Figura 02. JUnit 5

Fonte: https://junit.org (2023)

4.1.2.1 Mockito

O Mockito, Figura 03 , em sinergia com o JUnit no ambiente Java, amplifica a eficiência dos testes automatizados. Funcionando como uma biblioteca de mocking, o Mockito destaca-se na criação de objetos simulados que permitem isolar e controlar o comportamento de dependências durante os testes. Essa capacidade é particularmente valiosa em cenários onde a simulação de interações com sistemas externos, como bancos de dados ou serviços, é essencial.

Figura 03. Mockito

Fonte: https://site.mockito.org (2023)

4.1.3 Jest

O Jest é um framework de teste de JavaScript amplamente utilizado para aplicações desenvolvidas em ambientes Node.js, React e outras tecnologias baseadas em JavaScript. Desenvolvido pelo Facebook, o Jest é conhecido por sua fácil configuração, desempenho rápido e uma série de recursos poderosos, Figura 04.

Figura 04. Jest

Fonte: https://jestjs.io (2023)

4.2 Ferramentas de Automação

As ferramentas de automação de testes têm um papel essencial na criação e execução automatizada de testes, reduzindo a intervenção manual repetitiva. Elas oferecem uma interface gráfica amigável e recursos poderosos para gravar, reproduzir ações do usuário, validar resultados esperados e reportar resultados de teste. Além disso, permitem a criação de planos de teste, gerenciamento de casos de teste, rastreamento de defeitos e geração de relatórios de status e cobertura de teste. Exemplos notáveis de ferramentas de automação de testes *end-to-end(E2E)* incluem o Playwright, especializado na automação de testes de interface do usuário em diferentes navegadores.

4.2.1 Playwright

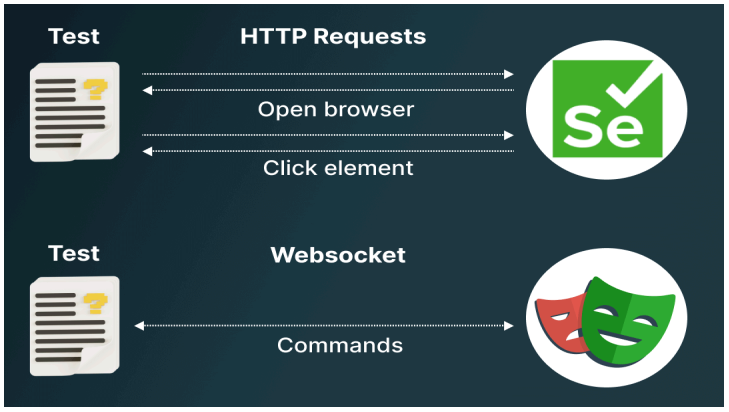
O Playwright é um framework de automação de testes moderno projetado para facilitar a automação de browsers em ambientes web. Criado pela Microsoft, o Playwright oferece uma abordagem abrangente para automação, permitindo a execução de testes em diferentes navegadores, interações complexas com páginas web e suporte para diversas linguagens de programação como JavaScript, TypeScript, Python e C#, Figura 05.

Figura 05. Playwright

Fonte: https://playwright.dev/ (2023)

É válido salientar que enquanto o Selenium, framework mais popular, utiliza uma arquitetura cliente-servidor, onde um WebDriver interage com os navegadores, o Playwright opera em um único processo para controlar o navegador, Figura 06. Essa diferença arquitetônica contribui para a eficiência notável do Playwright, especialmente em termos de execução paralela de testes e interações avançadas com páginas web.

Figura 06. Diferença de arquitetura: Selenium vs Playwright



Fonte: Adaptado de LAMBDATEST (2023)

Além disso, o Playwright é conhecido por suas características específicas, como gravação de vídeos de testes durante a execução, suporte nativo para dispositivos móveis e uma API robusta para interações complexas. Embora ambos os frameworks ofereçam suporte a diversos navegadores e linguagens de programação, a arquitetura e os recursos específicos do Playwright o tornam uma escolha atraente para projetos que demandam eficiência e funcionalidades avançadas de automação de testes.

5. Condução dos Testes Automatizados

No âmbito dos testes automatizados, uma prática fundamental para estruturar e conduzir cenários de teste é o emprego do conceito Given-When-Then. Segundo North [2006], esse paradigma fornece uma estrutura clara e consistente para a especificação de comportamento, contribuindo significativamente para a condução eficaz dos testes. Na seção "Given", estabelecemos as condições iniciais ou pré-requisitos necessários para o teste. A etapa "When" descreve a ação ou evento que desencadeia o comportamento a ser testado. Por fim, na etapa "Then", especificamos as expectativas e resultados

esperados após a execução da ação. A aplicação do Given-When-Then não apenas proporciona clareza na definição de testes, mas também se alinha perfeitamente com a automação de testes, facilitando a criação de scripts automatizados que seguem essa estrutura. Isso não só torna os testes mais compreensíveis para os membros da equipe, mas também favorece a manutenção e a evolução contínua dos testes automatizados, fortalecendo a condução eficaz do processo de teste automatizado.

Dado o conceito Given-When-Then no contexto de testes automatizados, torna-se relevante evidenciar a sua aplicação prática por meio de códigos. A seguir, vejamos exemplos concretos de testes de Unidade e Integração que ilustram como traduzir o Given-When-Then para scripts de teste automatizado, Figura 08 e 09:

Figura 08. Test de unidade com JUnit e Mockito

```java
// ... Rest code ... //

@RunWith(MockitoJUnitRunner.class)
public class CreateUserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private CreateUserService createUserService;

    @Test
    public void whenSaveUser_shouldReturnUser() {
      // Given
      User user = new User();
      user.setName("Test Name");

      // Mockando o método save do userRepository para simular a salvamento do usuário
      when(userRepository.save(ArgumentMatchers.any(User.class))).thenReturn(user);

      // When
      User created = createUserService.createNewUser(user);

      // Then
      assertThat(created.getName()).isSameAs(user.getName());
      verify(userRepository).save(user);
    }
}
```

Fonte: Adaptado de Medium (2020)

Na seção "Given", Figura 08, as condições iniciais do teste são estabelecidas. Um objeto User é criado e o método setName é chamado para definir o nome do usuário. Já na seção "When," ocorre a execução da função createUserService.createNewUser(user), que provavelmente envolve a chamada do método save do userRepository para criar o usuário. Por sua vez, na seção "Then" contém duas verificações. A primeira verifica se o nome do usuário criado (created) é o mesmo que foi definido nas condições iniciais. A segunda verifica se o método save do userRepository foi chamado corretamente com o objeto user como argumento.

A utilização do método when do Mockito cria um comportamento simulado para o método save do userRepository, retornando o objeto user quando chamado. O método verify é utilizado para garantir que o método save foi chamado corretamente durante a execução do teste.

Figura 09. Teste de unidade com Jest

```
// ... Rest code ... //

describe('UserService Tests', () => {
  let userService: UserService;

  beforeAll(async () => {

    const moduleFixture: TestingModule = await Test.createTestingModule({
      providers: [UserService]
    }).compile();

    userService = moduleFixture.get<UserService>(UserService);
  });

  it('Should add an user', async () => {
    // Given
    const user = {
      age: 35,
      name: "LuizTools",
      uf: "RS"
    } as users;

    const id: string = "abc123";

    // Mockando o método save do userRepository para simular a salvamento do usuário
    userService.users.create = jest.fn().mockReturnValueOnce({ id, ...user });

    //When
    const result = await userService.addUser(user);

    // Then
    expect(result.id).toEqual(id);
  });
});
```

Fonte: Adaptado de Github (2023)

Na seção "Given," as condições iniciais são estabelecidas, incluindo a definição de um objeto de usuário (user) com informações específicas, como idade, nome e estado, e a criação de um identificador (id). Já na seção "When," ocorre a execução da

função userService.addUser(user), simulando a adição de um usuário. Por fim, na seção "Then" inclui uma verificação (expect) para garantir que o identificador retornado após a adição corresponda ao identificador definido nas condições iniciais. Essa verificação valida se a função de adição de usuário está comportando-se conforme o esperado.

No contexto do Jest, a função jest.fn() é usada para criar um mock da função create de userService.users, e mockReturnValueOnce é utilizado para definir o valor retornado na primeira chamada da função. Este teste garante que a adição de usuário está ocorrendo corretamente, com o identificador esperado.

Figura 10. Test de integração com JUnit e Mockito

```java
// ... Rest Code ... //

@RunWith(SpringRunner.class)
@SpringBootTest(classes = SpringBootCrudRestApplication.class, webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class SpringBootCrudRestApplicationTests {

    @Autowired
    private TestRestTemplate restTemplate;

    @LocalServerPort
    private int port;

    private String getRootUrl() {
        return "http://localhost:" + port;
    }

    @Test
    public void contextLoads() {

    }


    @Test
    public void testCreateUser() {
        // Given
        User user = new User();
        user.setEmailId("admin@gmail.com");
        user.setFirstName("admin");
        user.setLastName("admin");
        user.setCreatedBy("admin");
        user.setUpdatedby("admin");

        // When
        ResponseEntity<User> postResponse = restTemplate.postForEntity(getRootUrl() + "/users", user, User.class);

        // Then
        assertNotNull(postResponse);
        assertNotNull(postResponse.getBody());
    }

}
```

Fonte: Adaptado de Medium (2020)

Na seção "Given", Figura 10, são estabelecidas as condições iniciais do teste. Um objeto User é criado e preenchido com dados fictícios. Na seção "When," ocorre a execução da requisição POST para a URL /users usando restTemplate.postForEntity. Esta chamada simula a criação de um usuário na API. E na seção "Then" contém duas verificações. A primeira verifica se a resposta da requisição (postResponse) não é nula. A segunda verifica se o corpo da resposta (postResponse.getBody()) não é nulo.

Figura 11. Test de integração com Jest

```
// ... Rest Code ... //

const BASE_URL = '/api/v1/user/'

describe('User', () => {

  describe('POST', () => {
    test('Deve criar um usuário e retornar CREATED', async () => {
      // Given
      const user = userBuilder().create()

      const expected: ExpectType = {
        data: {
          user,
        },
        errors: null,
      }

      // When
      const resp = await app.inject({
        method: 'POST',
        url: `${BASE_URL}new`,
        payload: {
          ...user,
        } as User,
      })

      // Then
      expect(resp.statusCode).toEqual(StatusCodes.CREATED)
      expect(resp.json<ExpectType>()).toEqual(expect.objectContaining({
        ...expected,
        data: {
          user: expect.objectContaining(user),
        },
      }))
    })
  })
})
```

Fonte: Adaptado de Github (2023)

Na seção "Given", Figura 11, é criado um usuário usando a função userBuilder().create(). Isso representa as condições iniciais para o teste. Já na seção "When," ocorre a execução da requisição POST para a URL ${BASE_URL}new utilizando o supertest (app.inject). O payload da requisição é o usuário criado. E por fim, a seção "Then" contém duas verificações. A primeira verifica se o código de status da resposta (resp.statusCode) é igual a StatusCodes.CREATED. A segunda verifica se o corpo da resposta (resp.json()) contém um objeto que é parcialmente igual ao objeto expected. Neste caso, está sendo verificado se o usuário retornado pela API está contido nos dados da resposta.

Figura 12. Teste End-to-End em Java

```java
// ... Rest Code ... //

public class RestfulBookerEndToEndTests extends BaseTest {

    private int bookingId;
    private BookingData bookingData;
    private String token;

    @BeforeClass
    public void setupTest() {
        // Given
        this.bookingData = getBookingData();
    }

    @Test
    public void createBookingTest() {
        // When
        final APIResponse response = this.manager.postRequest("/booking", RequestOptions.create()
                .setData(this.bookingData));
        // Then
        assertEquals(response.status(), 200);
        // When
        final JSONObject responseObject = new JSONObject(response.text());
        // Then
        assertNotNull(responseObject.get("bookingid"));
        // When
        final JSONObject bookingObject = responseObject.getJSONObject("booking");
        final JSONObject bookingDatesObject = bookingObject.getJSONObject("bookingdates");
        // Then
        assertEquals(this.bookingData.getFirstname(), bookingObject.get("firstname"));
        assertEquals(this.bookingData.getBookingdates()
                .getCheckin(), bookingDatesObject.get("checkin"));
        this.bookingId = responseObject.getInt("bookingid");
    }
}
```

Fonte: Adaptado de Medium (2022)

Na seção "Given", Figura 12, se refere à configuração inicial do estado do teste, temos a inicialização de dados da reserva no método setupTest() dentro da anotação @BeforeClass. O método getBookingData() é chamado para obter os dados necessários para criar uma reserva. Já ja seção "When," que representa a ação principal ou evento do teste, ocorre a execução da requisição POST para o endpoint /booking usando this.manager.postRequest(...). Aqui, estamos criando uma reserva com os dados configurados anteriormente. E por fim, na seção "Then" inclui as verificações que validam o comportamento esperado do sistema após a ação. Inicialmente, há uma verificação para garantir que o código de status da resposta seja 200 (OK). Em seguida, são realizadas verificações adicionais no conteúdo da resposta JSON para garantir a presença do campo "bookingid" e a correspondência dos dados da reserva com aqueles fornecidos durante a criação.

Figura 13. Teste End-to-End em Javascript/Typescript

```
// ... Rest Code ... //

test('should be able to create a booking', async ({ request }) => {
    // Given
    const requestData = {
        data: {
            "firstname": "Jim",
            "lastname": "Brown",
            "totalprice": 111,
            "depositpaid": true,
            "bookingdates": {
                "checkin": "2023-06-01",
                "checkout": "2023-06-15"
            },
            "additionalneeds": "Breakfast"
        }
    };

    // When
    const response = await request.post("/booking", requestData);
    const responseBody = await response.json();

    // Then
    expect(response.ok()).toBeTruthy();
    expect(response.status()).toBe(200);
    expect(responseBody.booking).toHaveProperty("firstname", "Jim");
    expect(responseBody.booking).toHaveProperty("lastname", "Brown");
    expect(responseBody.booking).toHaveProperty("totalprice", 111);
    expect(responseBody.booking).toHaveProperty("depositpaid", true);
});
```

Fonte: Adaptado de Github (2023)

Na seção "Given", Figura 13, a estrutura de dados (requestData) é configurada, representando as informações necessárias para criar uma reserva. Já na seção "When," a requisição POST é feita para o endpoint /booking com os dados configurados. Por fim, na seção "Then" inclui uma série de verificações para garantir que a resposta da API seja bem-sucedida (status 200) e que os detalhes da reserva na resposta correspondam aos dados fornecidos durante a criação. Cada expect representa uma verificação específica, verificando propriedades como "firstname," "lastname", "totalprice" e "depositpaid."

6. Conclusão e considerações finais

O processo de teste de software desempenha um papel crucial na garantia da qualidade, confiabilidade e desempenho de sistemas modernos. A definição clara de escopo e requisitos, aliada a estratégias de teste eficazes, estabelece a base para uma abordagem abrangente e sistemática. A realização de testes de funcionalidade, com foco na verificação rigorosa dos requisitos e comportamentos esperados, é crucial para

identificar e corrigir potenciais problemas, contribuindo para a entrega de um produto confiável.

A distinção entre testes manuais e automatizados revela as vantagens e desafios inerentes a cada abordagem. Enquanto os testes manuais oferecem flexibilidade e detecção imediata de problemas de usabilidade, os testes automatizados destacam-se pela precisão, eficiência e escalabilidade. A combinação equilibrada dessas abordagens, considerando as características específicas do software e os requisitos do projeto, é essencial para um processo de teste efetivo.

A introdução de frameworks e ferramentas de automação, como JUnit, Mockito e Playwright, desempenha um papel crucial na simplificação e aceleração do ciclo de vida dos testes. Essas tecnologias não apenas facilitam a criação e execução de testes automatizados, mas também promovem a reutilização, escalabilidade e integração contínua, essenciais em ambientes de desenvolvimento ágil.

É válido salientar que a relevância dos testes é substanciada pelo estudo da Undo [2020], que revela que falhas nos testes custam às empresas $61 bilhões anualmente. Segundo a pesquisa, 26% do tempo dos desenvolvedores é gasto reproduzindo e corrigindo falhas nos testes, totalizando 620 milhões de horas de desenvolvedor por ano. O custo total dessas horas de trabalho atinge $61 bilhões anualmente, resultando em uma perda de valor empresarial significativa.

Finalmente, a adoção de práticas estruturantes na condução de testes automatizados oferece uma abordagem consistente, clara e eficiente para especificar comportamentos, tornando os testes compreensíveis, manuteníveis e alinhados com as melhores práticas de automação. Ao incorporar esses princípios e práticas no ciclo de vida do desenvolvimento de software, as equipes podem aprimorar significativamente a qualidade do produto, acelerar a identificação de defeitos e, em última instância, oferecer soluções robustas e confiáveis aos usuários finais. Essa integração eficaz de estratégias, técnicas e ferramentas de teste é essencial para enfrentar os desafios dinâmicos do desenvolvimento de software contemporâneo.

7. Trabalhos futuros

Uma direção promissora para futuras pesquisas em testes de software é a integração mais profunda da Inteligência Artificial (IA). Algoritmos de machine learning podem automatizar a geração de casos de teste, identificar padrões complexos de defeitos e adaptar dinamicamente estratégias de teste. Essa abordagem não apenas otimiza processos, mas também vislumbra uma evolução para abordagens mais inteligentes, adaptáveis e proativas na garantia da qualidade de software.

8. Referências

Aniche, M. (2015). Testes automatizados de software: Um guia prático. Brasil: Casa do Código.

Booch, G. (2000). Software Testing and Quality Assurance. Addison-Wesley.

Bookman, A., Offutt, J. (2022). Software Testing: A Pragmatic Approach. CRC Press.

Cerruti, J. C. (2022). Superando os maiores desafios nos testes manuais de aplicativos mobile, https://imasters.com.br/mobile/superando-os-maiores-desafios-nos-testes-manuais-de-aplicativos-mobile.

Cordeiro, A., Freitas, A. L. (2012). Priorização de requisitos e avaliação da qualidade de software segundo a percepção dos usuários, http://eprints.rclis.org/17660.

Duarte, L. (2023). Como testar services em NestJS com Jest, https://www.luiztools.com.br/post/como-testar-services-em-nestjs-com-jest.

Espinha, R. G. (2020). Matriz de Rastreabilidade de Requisitos: saiba como gerenciar as mudanças no escopo, https://artia.com/blog/matriz-de-rastreabilidade.

Fadatare, R. (2018). Spring Boot CRUD REST APIs Integration Testing Example, https://www.javaguides.net/2018/09/spring-boot-2-rest-apis-integration-testing.html.

Fowler, M. (2013). GivenWhenThen, https://martinfowler.com/bliki/GivenWhenThen.html.

Gomes, J. (2023). Entenda a diferença: Caso de Teste & Cenário de Teste, https://www.dio.me/articles/entenda-a-diferenca-caso-de-teste-cenario-de-teste.

Juristo, N., Vegas, S., & Apa, C. (2013). Effectiveness for detecting faults within and outside the scope of testing techniques: a controlled experiment.

Khatri, M. F. How to perform End to End API Testing using Playwright with Java and TestNG, https://medium.com/@iamfaisalkhatri/how-to-perform-end-to-end-api-testing-using-playwright-with-java-and-testng-26b318927115.

Lambdatest (2023). What Is Playwright? Playwright Testing Tutorial - A Guide With Examples, https://www.lambdatest.com/playwright.

Mosconi, N. (2022). The Importance of Software Testing, https://www.devlane.com/blog/the-importance-of-software-testing.

North, D. T. (2006). Introducing BDD, https://dannorth.net/introducing-bdd.

Oliveira, G. S. (2012). Um framework para testes de software na nuvem. Dissertação (Mestrado em Informática) - Universidade Federal da Paraíba, João Pessoa, https://repositorio.ufpb.br/jspui/handle/tede/7818.

Oliveira, E. R. (2023). Template de Node com Fastify, https://github.com/rodrigocode4/node-fastify-template.

Silva, F. G. C. (2023). Levantamento de requisitos aplicado à pesquisa e desenvolvimento de produtos de software, https://ric.cps.sp.gov.br/handle/123456789/13619.

Pinto, G. (2022). Documentação de software: por que é tão importante e o que sabemos sobre ela?, https://www.zup.com.br/blog/documentacao-de-software.

Pressman, R. S. (2016). Engenharia de Software: uma abordagem profissional. 8ª ed. McGraw-Hill Education.

Pulga, G. (2020). A Beginners Guide to Unit Testing CRUD Endpoints of a Spring Boot Java Web Service/API, https://gabrielpulga.medium.com/a-beginners-guide-to-unit-testing-crud-endpoints-of-a-spring-boot-java-web-service-api-8ae342c9cbcd.

Reflect (2020). Mapping the Testing Pyramid to Automated Testing Tools, https://reflect.run/articles/automated-testing-tools.

Sommerville, I. (2019). Engenharia de Software. 10a edição. São Paulo: Pearson Prentice Hall.

Sharma, S. (2023). AI in Software Testing | Why it is Important In Software Test Automation?, https://testsigma.com/blog/is-ai-really-important-in-software-test-automation.

Undo (2020). What is the actual cost of software failures?, https://undo.io/solutions/developer-productivity/the-cost-of-software-failures.