Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Comparação das ferramentas Selenium e Cypress na automação de testes embasados em cenários no contexto de aplicações web

João Pedro Assunção Coutinho

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Fernando Albuquerque

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Comparação das ferramentas Selenium e Cypress na automação de testes embasados em cenários no contexto de aplicações web

João Pedro Assunção Coutinho

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Fernando Albuquerque (Orientador)
CIC/UnB

Prof.a Dr.a Fernanda Lima
CIC/UnB

Prof. Dr. Edison Ishikawa
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 25 de Julho de 2023

Dedicatória

Dedico este trabalho aos meus pais, Rosimeire e José, por proverem o necessário para que eu pudesse chegar ao final da graduação. Ao meu irmão, Lucas, pela companhia e por ser um excelente amigo. À minha namorada, Carolina, pelo apoio, carinho, e pelos bons momentos juntos. E aos amigos que fiz durante o curso.

Agradecimentos

Agradeço à Universidade de Brasília, ao Departamento de Ciência da Computação e seu corpo de docentes e demais funcionários, em especial, ao professor Fernando Albuquerque pela orientação e atenção providos durante a realização deste trabalho.

Resumo

Práticas de controle de qualidade têm se tornado cada vez mais importantes para aplicações de diversos tipos, em especial, para aplicações *web*, que são facilmente acessíveis via computadores e *smartphones*. Tendo isso em vista, torna-se cada vez mais comum utilizar ferramentas para facilitar, e automatizar a execução de testes, parte importante do controle de qualidade. Esse trabalho tem o objetivo de comparar duas ferramentas frequentemente utilizadas no contexto de automação de testes para aplicações *web*, o *Cypress* e o *Selenium*, seguindo uma abordagem embasada em cenários, onde testam-se cenários que representam fluxos completos de uso. Para atingir esse objetivo, parte de um sistema foi desenvolvido para a aplicação dos testes, cenários foram criados a partir dos requisitos do sistema, e a partir dos cenários, *scripts* de teste foram implementados e executados utilizando ambas ferramentas, para que possa ocorrer, por fim, a comparação segundo critérios selecionados.

Palavras-chave: Teste de aplicação web. Teste embasado em cenário. Ferramentas de automação de teste. Selenium. Cypress.

Abstract

Quality control practices become increasingly important for applications of various types, in particular, for web applications, which are easily accessible from personal computers and smartphones. Considering that, it becomes more common to use tools to facilitate and automate the execution of tests, an important part of quality control. This work aims to compare two popular tools used in the context of test automation for web applications, Cypress, and Selenium, following an approach based on scenarios, where scenarios that represent complete usage flows are tested. To achieve this goal, part of a system was developed for the application of tests, scenarios were created based on requirements of the system, scripts were implemented and executed using both tools, finally the tools were compared according to selected test criteria.

Keywords: Web application testing. Scenario based testing. Test automation tools. Selenium. Cypress.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nos dias de hoje, com a crescente facilidade de acesso à internet, cada vez mais pessoas passam a realizar as mais diversas tarefas *on-line*, desde socialização e comunicação, até consumo de conteúdos para entretenimento, e gerenciamento de contas bancárias, por exemplo. Nesse contexto, é importante que as aplicações apresentem a qualidade demandada pelos seus usuários

Para que as aplicações apresentem a qualidade demandada, isto é, se adéquem ao máximo à sua proposta [1], é relevante o controle de qualidade, particularmente o processo de teste de software.

Existem diversas ferramentas que facilitam e auxiliam o processo de teste de software. Em especial, existem ferramentas capazes de automatizar a execução, e até mesmo a criação de novos testes. Em especial, aplicar testes no contexto da *web* tem se tornado cada vez mais importante, visto que muitas das aplicações desenvolvidas são acessíveis via navegadores *web*. Para a automatização de testes no contexto da *web*, existem diversas ferramentas [2], sendo importante selecionar a que melhor atenda às necessidades das equipes de teste e desenvolvimento. Entre as ferramentas, das comumente utilizadas para esse propósito são o *Cypress* e o *Selenium*. Existem diferentes abordagens que podem ser utilizadas no desenvolvimento de testes, entre elas, existe a abordagem embasada em cenários, que propõe a execução de testes de fluxos de uso completos da aplicação sob teste, do ponto de vista do usuário. [3]

1.1 Objetivos

O objetivo geral deste trabalho é analisar e comparar as ferramentas denominadas *Cypress* e *Selenium* na automação de testes embasados em cenários aplicados a parte de uma aplicação *web*. A seguir, são relacionados objetivos específicos desse trabalho:

- Desenvolver parte de aplicação *web*, seguindo adaptação da Programação Extrema [4].
- Criar cenários de uso para a aplicação proposta, seguindo processo proposto para criação de cenários.
- Desenvolver casos de teste, a partir dos cenários criados, utilizando tanto o *Cypress*, como o *Selenium*.
- Comparar as ferramentas de teste segundo critérios selecionados.

1.2 Justificativa

A escolha de uma ferramenta para automação de testes pode não ser um processo trivial, considerando a diversidade de ferramentas que desempenham funções semelhantes, e de possíveis situações de uso. Portanto, é importante escolher a ferramenta para automação de testes que melhor atenda às necessidades.

1.3 Estrutura do documento

Este documento é composto por capítulos e apêndice que abordam a fundamentação teórica, a prática realizada e a conclusão. O Capítulo 2 apresenta definições e conceitos a respeito de qualidade de software; o Capítulo 3 apresenta processos de teste de software; o Capítulo 4 apresenta definições a respeito de automação de testes; o Capítulo 5 descreve métodos e processos usados na parte prática; o Capítulo 6 apresenta elementos do processo de desenvolvimento da aplicação proposta; o Capítulo 7 apresenta elementos do processo de teste e comparação das ferramentas; o Capítulo 8 apresenta considerações finais do trabalho; o Apêndice A apresenta descrições de cenários.

Capítulo 2

Qualidade de software

Neste capítulo são apresentadas definições e descrições relacionadas à qualidade no contexto da engenharia de software. Particularmente, é apresentada definição de qualidade e são abordados os processos de garantia de qualidade, controle de qualidade e teste de software.

2.1 Qualidade de software

Antes de definir qualidade no contexto de software, é interessante apresentar uma definição mais geral do termo "qualidade". Para Juran e De Feo (2010) [1], "qualidade" passou a significar "adequação à finalidade" ou "adequação ao propósito" (*fitness for purpose*), isto é, o que se produz, seja um bem ou serviço, deve se adequar à sua finalidade. Para ser adequado à finalidade, todo bem ou serviço deve ter as características e funcionalidades corretas para satisfazer às necessidades do cliente, e deve ser entregue com nenhuma ou poucas falhas. Juran e De Feo (2010) [1] trazem ainda dois significados, considerados de importância crítica ao se gerenciar qualidade, já que segundo os autores, nem sempre há um consenso quanto ao significado do termo e suas implicações. Os dois significados são:

- Características que atendem às necessidades dos clientes. O que permite: aumentar os níveis de satisfação do cliente, tornar produtos vendáveis, enfrentar a concorrência, aumentar a fatia de mercado, prover rendimento de vendas, assegurar maiores preços e reduzir riscos. Aqui, o maior efeito é na receita, e mais qualidade custa mais.
- Liberdade de falhas. O que permite: reduzir taxas de erro, reduzir retrabalho e desperdício, reduzir falhas de campo e cobranças de garantia, reduzir a insatisfação do cliente, reduzir inspeções e testes, diminuir tempo para lançar novos produtos no

mercado, aumentar rendimentos e capacidade e aumentar a performance de entrega. Aqui, o maior efeito é nos custos, e mais qualidade custa menos.

2.1.1 Definição de qualidade no contexto de software

Definir qualidade no contexto de engenharia de software, assim como a definição anterior, não é simples, e em geral, a definição varia a depender do autor e de seu contexto de pesquisa. Resumidamente, qualidade no contexto de software pode ser definido como:

> "A capacidade de um produto de software de satisfazer necessidades declaradas e implícitas, sob condições especificadas." (ISO/IEC, 2011 apud BOURQUE; FAIRLEY, 2014) [5]

> "O grau em que um produto de software atende aos requisitos estabelecidos; porém, qualidade depende do grau em que estes requisitos estabelecidos, representam necessidades, desejos, e expectativas das partes interessadas com precisão." (IEEE, 2012 apud BOURQUE; FAIRLEY, 2014) [5].

É possível notar que ambas as definições adotam a premissa de conformidade com requisitos, de forma que enfatizam a relação de dependência entre qualidade e requisitos.

2.1.2 Fatores de qualidade de software

Pressman (2010) [6] cita diversas definições aceitas de qualidade de software, que tentam, de certa forma, quantificar qualidade através da avaliação de múltiplos fatores; é citada, por exemplo, a definição dos fatores de qualidade segundo McCall et al. (1977 apud PRESSMAN, 2010) [6], sendo eles:

- Corretude: o quanto um programa satisfaz sua especificação, e atende aos objetivos do consumidor.
- Confiabilidade: o quanto pode-se esperar que um programa desempenhe sua funcionalidade esperada com a precisão desejada.
- Eficiência: a quantidade de recursos computacionais, e código-fonte são necessários para um programa desempenhar sua função.
- Integridade: quanto o acesso ao software ou dados por pessoas não autorizadas pode ser controlado.
- Usabilidade: esforço necessário para aprender, operar, preparar entradas e interpretar saídas de um programa.
- Manutenibilidade: esforço necessário para localizar e corrigir erros.

- Flexibilidade: esforço necessário para modificar um programa operante.
- Portabilidade: esforço necessário para transferir o programa de um ambiente de composto por hardware e software para outro.
- Reusabilidade: o quanto um programa, ou suas partes, podem ser reutilizadas em outras aplicações relacionadas ao pacote e escopo de funções que o programa desempenha.
- Interoperabilidade: esforço necessário para integrar um sistema a outro.

Pressman cita ainda os fatores definidos pelo padrão ISO 9126, um pouco mais concisos que os fatores listados anteriormente, sendo apenas seis: Funcionalidade, Confiabilidade, Usabilidade, Eficiência, Manutenibilidade e Portabilidade. Por fim, segundo Sommerville (2016) [7], avaliar a qualidade de um sistema, acaba sendo um processo subjetivo à equipe de gerenciamento de qualidade, sendo que deve haver um consenso, e deve ser decidido se um nível de qualidade considerado aceitável foi alcançado. Esta subjetividade ocorre pois muitas vezes, a qualidade de um sistema depende amplamente de fatores não-funcionais, e não somente da corretude das funcionalidades implementadas.

2.2 Garantia de qualidade de software

Segundo Pressman (2010) [6], a primeira função formal de garantia de qualidade foi introduzida na empresa Bell Labs, em 1916, e rapidamente se espalhou pelo mundo. Durante os anos 40, abordagens mais formais foram sugeridas, e dependiam de medidas e melhorias contínuas em processos.

Já a história de garantia de qualidade no contexto de software está ligada à história da qualidade na produção de hardware. Durante os primórdios da computação, qualidade era responsabilidade somente do programador. Padrões de garantia de qualidade em software foram introduzidos em desenvolvimento de software para contratos militares, na década de 70, e se espalharam rapidamente no desenvolvimento de software no mundo comercial.

Atualmente, garantia de qualidade (*quality assurance*), ao contrário de um mal-entendido difundido, não é "teste", mas um conjunto de atividades que definem e avaliam a adequação de processos de software para prover evidências que estabeleçam confiança que os processos de software produzam produtos de software com qualidade adequada para suas finalidades pretendidas. Garantia de qualidade de software possui dois aspectos: garantia de produto e garantia de processo. (BOURQUE; FAIRLEY, 2014) [5]

Um plano de qualidade de software (plano de garantia de qualidade de software) deve definir as atividades e tarefas empregadas para se assegurar que o software desenvolvido

para um produto específico satisfaz os requisitos especificados do projeto e necessidades dos usuários, obedecendo às restrições de custo e cronograma do projeto, bem como aos riscos. (BOURQUE; FAIRLEY, 2014) [5]

2.2.1 Tarefas e metas/atributos/métricas

Segundo Pressman (2010) [6], a equipe responsável pela garantia de qualidade de software deve prover assistência à equipe de software para alcançar um produto final de alta qualidade. A equipe de garantia de qualidade pode realizar algumas tarefas recomendadas, como:

- Preparar um plano de garantia de qualidade de software para o projeto: desenvolvido como parte do planejamento do projeto e revisado pelas partes interessadas, o plano identifica avaliações a serem realizadas, auditorias e revisões a serem conduzidas, padrões aplicáveis ao projeto, procedimentos para reporte e rastreamento de erros, produtos de trabalho que são produzidos pela equipe de garantia de qualidade de software, e *feedback* que será dado à equipe de software.
- Participar no desenvolvimento da descrição do processo de software do projeto: o grupo de garantia de qualidade revisa a descrição do processo selecionado para trabalho pela equipe de software, o processo deve obedecer a políticas organizacionais, padrões internos de software, padrões externos e outras partes do plano de projeto.
- Revisar atividades de engenharia de software a fim de verificar a conformidade com o processo de software definido.
- Auditar produtos de trabalho de software para verificar conformidade com os que foram definidos como parte do processo de software.
- Garantir que desvios no trabalho de software e produtos do trabalho estão documentados e manuseados de acordo com um procedimento documentado.
- Registrar qualquer desconformidade e relatar ao gerenciamento sênior.

Ainda segundo Pressman (2010) [6], as tarefas descritas acima são desempenhadas a fim de se alcançar alguns objetivos, tais quais:

- Qualidade de requisitos: corretude, completude e consistência do modelo de requisitos, têm uma forte influência na qualidade dos produtos do trabalho.
- Qualidade de projeto: todo elemento do modelo de projeto deve ser avaliado pela equipe de software para garantir que exibe alta qualidade, e que o próprio projeto está de acordo com os requisitos.

- Qualidade de código-fonte: código-fonte e outros produtos de trabalho relacionados devem estar de acordo com padrões de escrita de código-fonte e possui características que vão proporcionar melhor manutenibilidade.
- Eficácia no controle de qualidade: uma equipe de software deve aplicar recursos limitados de forma que se obtenha a maior probabilidade de se alcançar um resultado de alta qualidade.

Por fim, para Juran e De Feo (2010) [1], o foco primário de garantia de qualidade, não somente no contexto de software, é:

- Demonstrar que os requisitos de qualidade foram (e podem ser) alcançados.
- Ser motivada por partes interessadas, principalmente clientes.
- Objetivar a satisfação do usuário.
- Aumentar a confiança nos produtos da organização.
- Que o escopo de demonstração cubra atividades que afetam diretamente processos ou produtos relacionados à qualidade.

2.3 Controle de qualidade de software

Westfall (2016) [3] define controle de qualidade de software (*software quality control*) como o conjunto de ações e atividades necessárias para monitorar e realizar medições em projetos, processos e produtos de software, a fim de garantir que causas especiais não tenham introduzido variações indesejadas a esses mesmos projetos, processos e produtos.

Pressman (2010) [6] define controle de qualidade de forma semelhante à definição anterior. Para o autor, controle de qualidade (*quality control*) abrange o conjunto de ações de engenharia de software que ajudam a assegurar que cada produto do trabalho atinja seus objetivos. Modelos são revisados para garantir que sejam completos e consistentes. O código-fonte pode ser inspecionado a fim de se descobrir erros antes de serem realizados testes. Passos de testes são aplicados para descobrir erros na lógica de processamento, manipulação de dados, e comunicação de interfaces. Uma combinação de medidas e *feedbacks* permitem que uma equipe de desenvolvimento de software ajuste o processo quando qualquer dos produtos do trabalho falham em atender seus objetivos.

Tanto garantia como controle de qualidade de software integram o gerenciamento de qualidade de software, e enquanto garantia de qualidade é a definição de processos e padrões que devem levar a produtos de alta qualidade bem como à introdução de processos

de qualidade no processo de manufatura; controle de qualidade foca na aplicação desses processos com o intuito de eliminar produtos que não atinjam o nível desejado de qualidade. (BOURQUE; FAIRLEY, 2014; SOMMERVILLE, 2010) [5] [7]

Por fim, de acordo com o PMI e o IEEE (2013) [8], o melhor método para controlar e melhorar qualidade de software é focar em detecção e remoção precoce de defeitos, utilizando técnicas de verificação e validação contínua, e focar em realizar mudanças no processo de desenvolvimento de software para reduzir e prevenir falhas.

2.3.1 Entradas, ferramentas, técnicas e saídas para controle de qualidade em projetos de software

De acordo com o PMI e o IEEE (2013) [8], entradas para o controle de qualidade de software incluem: plano de gerenciamento de projeto, métricas de qualidade, listas de controle (*checklists*) de qualidade, dados sobre a performance de trabalho, solicitações de mudanças aprovadas, entregáveis, documentos do projeto e ativos do processo organizacional. Como ferramentas e técnicas para controle de qualidade, pode-se citar: (PMI, 2017) [9]

- Coleta de dados: podem ser utilizadas listas de verificação, folhas de verificação, amostragem estatística, e questionários/pesquisas.
- Análise de dados: podem ser utilizadas análise de desempenho, ou análise da causa raiz (usada para identificar a origem de defeitos).
- Inspeção: exame do produto resultante de um trabalho, a fim de se determinar se está de acordo com os padrões documentados.
- Testes/Avaliações dos produtos: podem ser realizados ao longo de todo o projeto, e têm como intuito encontrar defeitos ou outros problemas de não conformidade no produto ou serviço provido.
- Representação de dados: podem ser utilizados diagramas de causa e efeito, gráficos de controle, histogramas, e diagramas de dispersão.
- Reuniões: podem ser realizada análise das solicitações de mudança aprovadas, a fim de verificar se mudanças aprovadas foram implementadas, e se foram testadas, concluídas e certificadas de forma adequada; podem ser realizadas, também, retrospectivas, por exemplo, a fim de se discutir elementos bem-sucedidos, o que pode ser melhorado, etc.

No contexto de controle de qualidade em projetos de software, algumas considerações adicionais sobre as ferramentas e técnicas acima são pertinentes: (PMI; IEEE, 2013) [8]

- Inspeções realizadas precocemente no processo de desenvolvimento são mais efetivas para o controle de qualidade de software. Testes frequentes dos incrementos do produto é outra técnica que potencializa o controle de qualidade de software.
- Avaliações de usabilidade, na forma de demonstrações e passo-a-passo, são técnicas com boa relação custo-benefício, utilizadas para encontrar defeitos e discrepâncias que podem necessitar retrabalho. Testes de usabilidade gravados com representantes de usuários, utilizando a abordagem *think-aloud* (pensamento em voz alta) também podem ser úteis para encontrar defeitos antes do lançamento do produto ao usuário final.
- O desenvolvimento guiado por testes é conhecido por ser útil no controle de qualidade de software. Nessa abordagem, casos de teste são escritos antes do código-fonte. Os casos de teste são então executados e falharão. Então, novo código-fonte é adicionado, e os casos de teste são executados novamente com o intuito de demonstrar que não falham mais. Ferramentas são comumente utilizadas para automatizar esse processo.
- Testes de software incluem diversas fases/tipos e frequentemente equipes de desenvolvimento constroem módulos temporários que permitam a execução de testes precocemente simulando entradas e saídas de partes do software que ainda não foram construídas. Testes também podem focar em atributos específicos de qualidade, como desempenho, segurança e usabilidade.

Já como saídas do controle de qualidade em software, há (PMI; IEEE, 2013) [8]: medidas de controle de qualidade, mudanças validadas, entregáveis verificados, informações sobre o desempenho do trabalho, solicitações de mudança, atualização do plano de gerenciamento do projeto, atualizações na documentação do projeto e atualização nos ativos dos processos organizacionais.

Ainda segundo o PMI e o IEEE (2013) [8], algumas saídas adicionais referentes ao controle de qualidade de produtos e projetos de software incluem:

- Medidas dos atributos de qualidade especificados no plano de gerenciamento de qualidade e nos critérios de lançamento.
- Mudanças ao software e outros artefatos validados por testes ou inspeções.
- Entregáveis validados por testes/inspeções para obedecer ao escopo identificado no início do projeto/iteração
- Identificação de lacunas entre o desempenho planejado e o real, assim como as razões para estas discrepâncias.

2.4 Teste de software

Testes de software podem ser definidos como:

> "Verificações dinâmicas que um programa se comporta conforme o previsto, dado um conjunto finito de casos de teste, selecionados de um usualmente infinito domínio de execução." (BOURQUE; FAIRLEY, 2014) [5]

A escrita de casos de teste, nos dias atuais, tem se tornado uma prática cada vez mais comum e encorajada pela indústria, não somente após a construção dos artefatos, mas também durante, e antes do início do processo de desenvolvimento. De acordo com Sommerville (2016) [7], quando testamos um sistema, temos, sumariamente, dois objetivos: Mostrar ao desenvolvedor e ao cliente, que o software atende seus requisitos; e encontrar entradas ou sequências de entradas para os quais o software não se comporta da maneira esperada.

2.4.1 Fases de teste de software

De acordo com Sommerville (2016) [7] e Westfall (2016) [3], em geral, testes são organizados em três fases:

- Testes de desenvolvimento: realizados durante o desenvolvimento de software, ou escrita do próprio código-fonte, pelos próprios programadores e engenheiros de software, objetivando encontrar defeitos antes do lançamento do produto. Aqui, existem três subfases:
  - Testes unitários: foca no teste de classes e objetos individuais, e suas funcionalidades. Geralmente, o autor do módulo é responsável pela escrita dos casos de testes unitários.
  - Testes de componentes (ou testes de integração, segundo Westfall (2016) [3]): foca na integração das unidades para a criação de componentes, testa interfaces dos componentes. Dependendo da organização e tamanho do projeto, esse tipo de teste pode ser realizado pela própria equipe de desenvolvimento, ou por uma equipe separada.
  - Testes de sistema: o sistema é testado como um todo, foca na interação entre componentes.
- Testes de *release*: são realizados em uma *release* (versão de um sistema de software que é disponibilizada aos clientes do sistema, segundo Sommerville (2016) [7]) particular do sistema, difere de testes de sistema já que, geralmente não deve ser

realizado pelo time de desenvolvimento, e foca, ao invés de encontrar defeitos (*bugs*), em garantir que o produto atende aos requisitos do cliente. Geralmente ocorrem como um tipo de teste caixa preta (*black-box*), em que os casos de teste derivam da especificação do sistema., e que as particularidades estruturais do sistema não são conhecidas.

- Testes de usuário: fase em que clientes e usuários finais do sistema, dão sua opinião e recomendações em relação aos testes e ao sistema em si. Foca em garantir que o sistema desempenha satisfatoriamente no ambiente em que vai ser utilizado. Se divide em três tipos:

  - Testes alfa: um grupo selecionado de usuários trabalha com o time de desenvolvimento para o teste de *releases* antecipadas do sistema.
  - Testes beta: o sistema é disponibilizado a um número maior de usuários, a fim de experimentar e encontrar problemas antes da *release* final.
  - Testes de aceitação: clientes testam o sistema a fim de decidir se o mesmo está ou não apto a ser implantado no ambiente dos usuários finais.

2.4.2 Estratégias de teste de software

Segundo Westfall (2016) [3], existem diferentes estratégias que podem ser utilizadas quando testando software, sendo que cada estratégia possui seus benefícios e limitações. Entre essas estratégias, é possível destacar:

- Teste de caixa branca (*white-box testing*): testes em que se conhece a estrutura interna do software e procura-se por defeitos na construção ou lógica interna do sistema. Este tipo de teste é geralmente executado cedo no ciclo de testes, explorando cada módulo de código fonte internamente, e observando o fluxo de dados através de cada linha de código fonte.
- Teste de caixa cinza (*gray-box testing*): quando um grande número de módulos de código-fonte são integrados, pode se tornar difícil realizar puramente testes de caixa branca. Dessa forma, com o crescimento da complexidade e da quantidade de fluxos dentro de um sistema, os testes acabam progredindo para níveis de teste de caixa cinza, que representam uma mistura entre testes de caixa branca e preta. Nessa estratégia, o profissional responsável pelos testes adquire somente o conhecimento necessário da estrutura interna dos módulos, para que seja possível determinar de que forma módulos comunicam-se entre si.

- Teste de caixa preta (*black-box testing*): também conhecido como teste funcional (*functional testing*), nessa abordagem, a estrutura interna do sistema é ignorada e testa-se o comportamento do software da perspectiva do usuário. Teste de caixa preta foca em prover entradas ao sistema, sob circunstâncias conhecidas, e avaliar as saídas, comparando-as com resultados esperados. Já que, utilizando essa abordagem, conhecimento interno do sistema não é necessário, indivíduos externos, sem conhecimento sobre a estrutura interna do sistema, podem realizar esses testes.
- Desenvolvimento guiado por testes (*test-driven development*): implementam-se funcionalidades do software baseando-se na escrita de casos de teste pelos quais o código fonte deve passar. Estes casos de teste são executados com frequência a fim de que se verifique que mudanças ou incrementos não introduziram novos erros.
- Simulação: o ambiente de testes frequentemente não é capaz de reproduzir o ambiente real em que o sistema será executado. Tendo isso em vista, é possível contornar algumas limitações criando simulações que imitem condições do mundo real, como por exemplo, simulando o uso de um sistema por uma grande quantidade de usuários paralelos.
- Automação de teste: uso de software para automatizar atividades de projeto de testes, execução de testes, e captura e análise de resultados de testes. Testes automatizados podem, geralmente, ser executados mais rapidamente, e consequentemente, com mais frequência que testes manuais, provendo maior visibilidade aos níveis de qualidade do sistema. Por outro lado, automatizar testes requer um maior investimento inicial em ferramentas e recursos para criar a automatização, e um investimento de longo termo para manter as suítes automatizadas de teste, enquanto o sistema que está sendo desenvolvido muda e evolui ao longo do tempo. Por fim, automatizar testes também tem suas limitações: por exemplo, um teste automatizado somente verifica o que foi programado para verificar, e sob um conjunto finito de resultados esperados.

Capítulo 3

Teste de software

Este capítulo tem foco nos processos relacionados ao teste de software. Abordando, primeiramente, processos em ciclo de vida de software, processo de teste, projeto de teste, e por fim, teste baseado em cenários.

3.1 Processos em ciclo de vida de software

Antes de descrever processos pertencentes ao ciclo de vida de software, é interessante caracterizar o conceito de "processo" . Processo é definido pela ISO/IEC/IEEE (2017) [10] como o "conjunto de atividades inter-relacionadas ou que interagem entre si que transformam entradas em saídas". Já Sommerville (2016) [7] define processo no contexto de software como "conjunto de atividades relacionadas que levam à produção de um sistema de software".

Além disso, é importante apresentar alguns conceitos a respeito de ciclo de vida de software, como a divisão em estágios de um ciclo de vida, e os diferentes modelos existentes. O padrão definido pela ISO/IEC/IEEE (2017) [10] lista um conjunto de estágios comuns a ciclos de vida de sistemas, e esses incluem: conceitualização, desenvolvimento, produção, utilização, suporte e retirada. Já a respeito de modelos de ciclo de vida de software, e ainda segundo a ISO/IEC/IEEE (2017) [10], um ciclo de vida pode ser descrito utilizando um modelo funcional que representa a conceitualização da necessidade pelo sistema, sua realização, utilização, evolução e descarte. Um exemplo de modelo comumente citado, é o modelo em cascata, mas diferentes modelos surgiram, para atender diferentes necessidades e lidar com determinados problemas, como os modelos incremental, espiral, iterativo e evolucionário.

Agora, em posse desses conceitos, é possível começar a descrever processos pertencentes ao ciclo de vida de software. A ISO/IEC/IEEE (2017) [10] separa os processos e atividades

que podem ser realizadas durante o ciclo de vida de um sistema de software em quatro grupos de processos:

- **Processos de acordo**: Segundo a ISO/IEC/IEEE (2017) [10], acordos servem para que uma organização, agindo como um adquirente, solicite produtos ou serviços de outra organização, agindo como fornecedor. Acordos permitem que ambas as partes fiquem cientes de valores e suportem estratégias de negócio para suas organizações. Dessa forma, esse grupo engloba dois processos: processo de aquisição, e processo de fornecimento.

- **Processos organizacionais de facilitação para o projeto**: Segundo a ISO/IEC/IEEE (2017) [10], estes processos têm como pontos de atenção prover recursos para possibilitar que o projeto atenda às necessidades e expectativas das partes interessadas. Os processos organizacionais de facilitação para o projeto geralmente preocupam-se com o gerenciamento e melhora do negócio/empresa a nível estratégico, com provisão e mobilização de recursos e ativos, bem como com gerenciamento de risco em situações incertas. Em suma, estes processos são responsáveis por estabelecer o ambiente em que projetos serão conduzidos.

  Esse grupo engloba os seguintes processos: gerenciamento de modelo de ciclo de vida; processo de gerenciamento de infraestrutura; processo de gerenciamento de portfólio; processo de gerenciamento de recursos humanos; processo de gerenciamento de qualidade; e processo de gerenciamento de conhecimento. (ISO/IEC/IEEE, 2017) [10]

- **Processos de gerenciamento técnico**: Os processos pertencentes a este grupo têm como propósito gerenciar recursos e ativos alocados pelo gerenciamento da organização e aplicá-los para cumprir os acordos dos quais a organização participe. Os processos de gerenciamento técnico relacionam-se com o esforço técnico de projetos, em particular com planejamento de custos, prazos, resultados, bem como verificações de ações para ajudar a garantir que respeitam planos e critérios de desempenho. Este grupo engloba os seguintes processos: processo de planejamento de projeto; processo de análise e controle do projeto; processo de gerenciamento de tomada de decisão; processo de gerenciamento de riscos; processo de gerenciamento de configuração; processo de gerenciamento de informação; processo de mensuração; e processo de garantia de qualidade (ISO/IEC/IEEE, 2017) [10]

- **Processos técnicos**: Segundo a ISO/IEC/IEEE (2017) [10], os processos técnicos preocupam-se com ações técnicas realizadas durante o ciclo de vida. Processos técnicos têm como propósito transformar as necessidades das partes interessadas em

produtos/serviços. Estes processos devem auxiliar a prover desempenho sustentado, quando necessário, para que se atendam os requisitos das partes interessadas e se obtenha maiores níveis de satisfação do cliente. Processos técnicos são aplicados a fim de se criar e utilizar um sistema de software, seja na forma de um modelo, ou produto operacional.

Este grupo engloba os seguintes processos: processo de análise de negócio ou missão; processo de definição de necessidades e requisitos das partes interessadas; processo de definição de requisitos de sistemas/software; processo de definição de arquitetura; processo de definição de *design*; processo de análise de sistemas; processo de implementação; processo de integração; processo de verificação; processo de transição; processo de validação; processo de operação; processo de manutenção; e processo de descarte.

3.2 Processo de teste

Segundo Bourque e Fairley (2014) [5], conceitos, técnicas e medidas devem ser integradas em um processo bem definido e controlado, de forma que o processo de teste oriente testadores e equipes de teste a prover garantia de que os objetivos de teste foram cumpridos de forma a melhor utilizar os recursos disponíveis. O processo de teste é dividido segundo as seguintes etapas: [5]

- **Planejamento**: aspectos importantes a serem especificados em um plano de teste incluem coordenação de pessoal, disponibilidade de instalações e equipamentos de teste, criação/manutenção de documentos relacionados a testes, planejamento para resultados não desejados.
- **Geração de casos de teste**: essa atividade é baseada no nível de teste a ser realizado, bem como nas técnicas utilizadas.
- **Desenvolvimento de ambiente de teste**: o ambiente em que os testes serão executados deve ser construído de tal forma que facilite o desenvolvimento e controle dos casos de teste, bem como registro e recuperação de resultados esperados, *scripts*, e outros elementos presentes no processo.
- **Execução**: essa atividade deve ser realizada de forma a seguir procedimentos previamente documentados, utilizando uma versão especificada do sistema. Além disso, informações devem ser registradas de modo que outras pessoas possam reproduzir os resultados.

- **Avaliação de resultados**: resultados dos testes devem ser avaliados para que se determine se os testes executados obtiveram ou não sucesso.
- **Reporte de problemas/registro de testes**: atividades de teste podem ser incluídas em um registro que inclua dados como quando um teste foi executado, por quem, que configurações foram utilizadas, e outras informações de identificação relevantes. Resultados incorretos/inesperados podem ser registrados em um sistema de reporte de problemas, de forma que esses dados possam ser utilizados para, depois, realizar a correção dos erros.
- **Rastreamento de defeitos**: é possível rastrear e analisar defeitos encontrados em um sistema a fim de determinar quando foram introduzidos, o motivo de sua introdução e o momento provável em que foram observados pela primeira vez. Dados de rastreamento de defeitos são utilizados para determinar quais aspectos do processo de testes (e de outros processos) precisam de melhorias e podem ajudar a mensurar o quão efetivas abordagens anteriores foram.

A norma da ISO/IEC/IEEE (2013) [11] caracteriza o processo de teste de forma semelhante, porém mais concisa, dividindo-o em quatro etapas: projeto e implementação de teste; configuração de ambiente e manutenção de teste; execução de teste e reporte de incidentes dos testes.

3.3 Projeto de teste

Antes de melhor detalhar a etapa de projeto/implementação de teste, serão apresentadas definições dos seguintes termos: base de teste; condição de teste; e item de cobertura de teste. Base de teste (*test basis*) é o "corpo de conhecimento utilizado como base para o projeto de testes e casos de teste" (ISO/IEC/IEEE, 2013) [11]. Condição de teste (*test condition*) é definido como um "aspecto testável de um componente ou sistema, como uma função, transação, funcionalidade, atributo de qualidade ou elemento estrutural identificado como uma base para teste" (ISO/IEC/IEEE, 2013) [11]. E item de cobertura de teste é definido como "atributo ou combinação de atributos que são derivadas de uma ou mais condições de teste utilizando uma técnica de projeto de teste que possibilite mensuração do rigor da execução do teste" (ISO/IEC/IEEE, 2013) [11].

Através do processo de projeto/implementação de testes especificado na ISO/IEC/IEEE (2013) [11], é possível criar casos de teste e procedimentos de teste. Este processo pode ser interrompido e posteriormente retomado por diversas razões, por exemplo, no caso em que após a realização de um procedimento de teste, ou após o reporte de um incidente, percebe-se que casos de teste adicionais são necessários a fim de se atender os

critérios de conclusão de testes. O processo de projeto de testes requer que os indivíduos testadores apliquem uma ou mais técnicas de projeto de teste para criar casos e procedimentos de teste com a meta principal de alcançar os critérios de conclusão de testes, tipicamente descritos em termos de medidas de coberturas dos testes.

Ainda segundo a ISO/IEC/IEEE (2013) [11], o sucesso na implementação do processo de projeto e implementação de testes, tem como resultados esperados: análise da base de teste para cada item de teste; agrupamento das funcionalidades a serem testadas em conjuntos de funcionalidades; determinação das condições de teste; determinação dos itens de cobertura de teste; determinação dos casos de teste; montagem dos conjuntos de testes; determinação dos procedimentos de teste.

3.3.1 Atividades e tarefas

A seguir estão listadas as atividades, e tarefas associadas às atividades, que compõem, segundo a ISO/IEC/IEEE (2013) [11], o processo de teste de software:

- **Identificar os conjuntos de testes**:
  - Análise da base de teste para que sejam entendidos os requisitos para cada item de teste.
  - Combinação de funcionalidades a serem testadas em conjuntos.
  - Teste de uma funcionalidade deve ser priorizado de acordo com os níveis de exposição a risco identificados e analisados anteriormente.
  - Composição e priorização de conjuntos de funcionalidades devem ser acordadas com as partes interessadas.
  - Documentação dos conjuntos de funcionalidades na especificação do projeto de teste.
  - Rastreabilidade entre a base de teste e o conjunto de funcionalidades deve ser registrada.
- **Determinar condições de teste**:
  - Determinação das condições de teste, baseadas nos critérios de conclusão de teste especificados no plano de teste.
  - Condições de teste podem ser priorizadas segundo os níveis de exposição a risco identificados e analisados anteriormente.
  - Registro das condições de teste na especificação do projeto de teste.

  - Registro da rastreabilidade entre a base de teste, o conjunto de funcionalidades e as condições de teste.
  - Especificação do projeto de testes deve ser aprovada pelas partes interessadas.

- **Determinar itens de cobertura de teste**:

  - Itens de cobertura de teste a serem a exercidos pelos testes devem ser determinados através da aplicação de técnicas de projeto de teste para que as condições de teste atendam aos critérios de conclusão de cobertura de teste especificados no plano de teste.
  - Priorização dos itens de cobertura de teste segundo os níveis de exposição a risco identificados e analisados anteriormente.
  - Itens de cobertura de teste devem ser registrados na especificação do caso de teste.
  - Registro da rastreabilidade entre a base de teste, conjunto de funcionalidades, condições de teste, e itens de cobertura de teste.

- **Determinar casos de teste**:

  - Um ou mais casos de teste podem ser originados ao determinar pré-condições, selecionar valores de entrada e, quando necessário, agir para executar os itens de cobertura de teste, e ao determinar os resultados esperados correspondentes.
  - Priorização dos casos de teste segundo os níveis de exposição a risco documentados anteriormente.
  - Registro dos casos de teste na especificação do caso de teste.
  - Registro da rastreabilidade entre a base de teste, conjunto de funcionalidades, condições de teste, itens de cobertura de teste e casos de teste.
  - Especificação do caso de teste deve ser aprovada pelas partes interessadas.

- **Montar conjuntos de testes**:

  - Casos de teste podem ser distribuídos entre um ou mais conjuntos de teste baseados em restrições em sua execução.
  - Registro dos conjuntos de teste na especificação do procedimento de teste.
  - Registro da rastreabilidade entre a base de teste, conjunto de funcionalidades, condições de teste, itens de cobertura de teste, casos de teste e conjuntos de teste.

- **Determinar procedimentos de teste**:
  - Procedimentos de teste podem ser determinados ao se ordenar casos de teste dentro de um conjunto de testes segundo dependências descritas por pré ou pós condições e outros requisitos de teste.
  - Qualquer dado ou requisito de ambiente de teste que não esteja incluso no plano de teste deve ser identificado.
  - Priorização dos procedimentos de teste segundo os níveis de exposição a risco documentados anteriormente.
  - Registro dos procedimentos de teste na especificação do procedimento de teste.
  - Registro da rastreabilidade entre a base de teste, conjunto de funcionalidades, condições de teste, itens de cobertura de teste, casos de teste, conjuntos de teste, e procedimentos de teste.
  - Especificação do procedimento de teste deve ser aprovada pelas partes interessadas.

3.4 Projeto de teste caixa preta

Segundo Pressman (2010) [6] teste caixa preta (*black-box testing*), foca nos requisitos funcionais de um sistema de software. Isto é, técnicas de teste caixa preta permitem que sejam derivados conjuntos de condições de entrada que deverão exercitar, por completo, os requisitos funcionais para um programa. Para Sommerville (2016) [7], teste caixa preta é uma abordagem de teste em que os testadores não têm acesso ao código fonte de um sistema ou seus componentes, de forma que os testes são derivados da especificação do sistema.

Segundo Pressman (2010) [6], testes caixa preta tendem a ser aplicados em estágios mais tardios de teste, já que este tipo de teste, propositalmente desconsidera a estrutura de controle do sistema.

3.4.1 Métodos de projeto de teste caixa preta

Pressman (2010) [6], descreve, os seguintes métodos para projeto de teste caixa preta:

- **Teste baseado em grafo**: Segundo Pressman (2010) [6], o primeiro passo para o desenvolvimento de testes caixa preta, é o entendimento dos objetos que compõem o sistema, bem como os relacionamentos entre eles. Feito isso, o próximo passo consiste em verificar que todos os objetos possuem o relacionamento esperado entre si. Dessa forma, um teste pode ser conduzido através da criação de um grafo

com objetos e relacionamentos importantes e então elaborando uma série de testes que cubram o grafo, ou seja, cada objeto e relacionamento descrito pelo mesmo é exercitado, e erros podem ser descobertos.

Para cumprir esses passos, primeiramente, cria-se um grafo, onde os nós representam objetos, ligações representam relacionamentos entre objetos, e os pesos dos nós podem descrever propriedades de um nó, e pesos de ligações que descrevem características das ligações.

- **Particionamento por equivalência**: Segundo Pressman (2010) [6], outro método de desenvolvimento de teste caixa preta, consistente em dividir o domínio de entrada de um programa em classes de dados dos quais casos de teste podem ser derivados.

  Projeto de caso de teste por particionamento por equivalência é baseado na avaliação de classes de equivalência para uma condição de entrada. Utilizando conceitos do método anterior, se um conjunto de objetos podem ser ligados por relacionamentos simétricos, transitivos, e reflexivos, uma classe de equivalência está presente. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para condições de entrada. Tipicamente, uma condição de entrada é um valor específico, uma faixa de valores, um conjunto de valores relacionados ou uma condição booleana. Ainda segundo Pressman (2010) [6], neste método, casos de testes são selecionados a fim de que o maior número de atributos possíveis de uma classe de equivalência sejam testados ao mesmo tempo.

- **Análise de valores de limite**: De acordo com Pressman (2010) [6], grande parte dos erros, ocorrem nos limites do domínio de entrada, em vez de no centro. Por esse motivo, a técnica de teste de análise de valores de limite foi desenvolvida. A análise de valores de limite contribui para a seleção de casos de teste que utilizem valores de limite.

  Essa é uma técnica que complementa a técnica anterior, de particionamento por equivalência. Em vez de selecionar qualquer elemento de uma classe de equivalência, esta técnica leva à seleção de casos de teste de borda dentro das classes.

- **Teste de matriz ortogonal**: Em certas aplicações, o número de parâmetros de entrada é relativamente limitado e pequeno. Quando isso ocorre, pode ser possível considerar todas as permutações possíveis de entrada e testar exaustivamente o domínio de entrada. Porém, com o crescimento dos valores de entrada, e do número de parâmetros, pode se tornar impraticável ou impossível realizar um teste exaustivo. Tendo isso em vista, testes de matriz ortogonal são aplicados em situações em que

o domínio de entrada é relativamente pequeno, mas ainda muito grande para que o teste exaustivo torne-se praticável. (PRESSMAN, 2010) [6]

Quando o teste de matriz ortogonal ocorre, uma "matriz ortogonal L9" de casos de teste é criada. Esta matriz possui uma propriedade de balanceamento. E casos de teste são dispostos uniformemente pelo domínio. Em suma, uma abordagem de teste por matriz ortogonal permite que se alcance altos níveis de cobertura de teste, utilizando menos casos de testes do que a estratégia exaustiva de teste. (PRESSMAN, 2010) [6]

3.4.2 Projeto de teste embasado em cenário

Antes de caracterizar teste embasado em cenário, serão apresentadas algumas definições de cenário no contexto de teste de software: Sommerville (2016) [7] define cenário como uma descrição de uma forma em que um sistema é tipicamente utilizado, ou da forma com que realiza alguma atividade. De acordo com Pressman (2010) [6], cenários de uso, dentro de elicitação de requisitos, são uma forma de facilitar o entendimento de como diferentes funções e funcionalidades de um sistema serão utilizadas por diferentes usuários finais. Desenvolvedores e usuários podem criar cenários (também denominados, pelo autor, como casos de uso) que identifiquem fluxos de uso do sistema a ser construído, de forma a prover descrição de como o sistema será utilizado. Já segundo Sethi (2017) [12], um cenário é um tipo de narrativa personalizada de uma experiência fim a fim de um usuário primário do sistema, podendo prover, inclusive um contexto para o melhor entendimento de suas necessidades.

Para criar cenários de teste, Hamilton (2022)[13] sugere atividades em cinco etapas:

- Leitura de documentos de requisitos do sistema sob teste. Também pode ser útil utilizar casos de uso, livros manuais, ou outros documentos relevantes a respeito da aplicação a ser testada.
- Predição do uso do usuário final. Para cada requisito, determinar possíveis ações e objetivos dos usuários finais. Determinar os aspectos técnicos do requisito.
- Após a leitura e análise, listar os diferentes cenários de teste que verifiquem cada funcionalidade do sistema.
- Após a listagem dos possíveis cenários de teste, uma matriz de rastreabilidade é criada para verificar que cada requisito possui um cenário de teste correspondente.
- Os cenários criados são revisados por supervisores e partes interessadas do projeto.

Ryzer e Glinz (2000)[14] também sugerem etapas para a criação de cenários. As etapas sugeridas pelos autores são, resumidamente:

- Criação de cenário: aqui ocorre a elicitação de cenários, bem como a criação e refinamento dos mesmos.
- Descrição de cenário: pode ocorrer utilizando linguagem natural (organizada, por exemplo, utilizando um *template*), representação em tabela, ou representação gráfica.
- Validação, transformação e anotação de cenário.

Por fim, em posse da definição de cenário de teste, podemos definir testes embasados em cenários: Segundo Sommerville (2016) [7], teste embasado em cenário é uma abordagem de teste em que se criam cenários de uso típicos e utiliza-se esses cenários para derivar casos de teste para o sistema. Já para Westfall (2016)[3], teste de cenário de uso consiste no encadeamento de casos de teste juntos, de forma a passar do teste de funções/sub-funções individuais, para cenários de teste que avaliem as interações do usuário com o sistema do início ao final. Essa abordagem é utilizada já que cada funcionalidade pode apresentar o comportamento esperado quando isolada, mas pode não apresentar o comportamento esperado quando testada em conjunto com outras funcionalidades; em um ambiente no qual usuários utilizem o sistema para suas reais necessidades. Para Kaner (2013)[15], um teste embasado em cenário satisfatório deve possui as seguintes caracterísiticas:

- Ser baseado em uma história sobre como o programa é utilizado, incluindo motivações das pessoas envolvidas.
- A história em que se baseia deve ser motivadora.
- A história em que se baseia deve ser verossímil. Os acontecimentos descritos na história, não somente devem poder ocorrer no mundo real, como deve ser possível acreditar que irão ocorrer.
- A história em que se baseia envolve um tipo complexo de uso do sistema, um ambiente complexo ou um conjunto de dados complexos.
- Os resultados dos testes são facilmente avaliáveis. Isso é valioso para todo tipo de teste, mas é especialmente relevante em testes de cenários pois os mesmos tendem a ser complexos.

Kaner (2013) [15], lista, além disso, alguns motivos para o uso de teste embasado em cenário, entre eles:

- Aprendizado em relação ao produto: uma das melhores formas de adquirir conhecimento a respeito do sistema sob teste, é a realização de tarefas que necessitem investigação do produto.

- Conexão de testes com requisitos documentados, especialmente no caso de requisitos modelados com casos de uso.
- Exposição de falhas para entregar benefícios desejados: Um teste de cenário provê uma verificação de fim a fim de um benefício que o sistema deveria entregar.
- Exploração de casos de uso avançados do programa sob teste: indivíduos podem passar a utilizar um sistema de forma diferente conforme adquirem experiência com o mesmo, e da mesma forma que o *feedback* inicial dos usuários de um sistema é importante, a estabilidade do sistema, sob uso de um usuário mais avançado e com experiência também tem importância.
- Trazer problemas relacionados a requisitos à superfície, o que pode levar à reabertura de antigas discussões sobre requisitos, ou trazer à superfície requisitos ainda não identificados.

Capítulo 4

Automação de testes

Este capítulo apresenta definições a respeito de automação no contexto de testes de software, classes de ferramentas utilizadas para automação de testes, elementos da automação de testes caixa preta e da automação de testes embasados em cenários.

4.1 Automação de testes

Antes de definir e caracterizar automação no contexto de teste de software, convém e é importante definir o termo automação por si só, de forma mais geral. O SEVOCAB (2023)[16], define o termo como: "conversão de processos ou equipamentos para operação automática, ou os resultados da conversão".

Pode-se ainda, definir automação como sinônimo de "automatização", significando: "execução automática de tarefas industriais ou científicas sem intervenção humana intermediária" (PRIBERAM, 2023) [17].

4.1.1 Definição de automação de testes

De acordo com Garousi e Mäntylä (2016) [18], o trabalho de teste pode ser grosseiramente dividido em testes manuais e testes automatizados. Em testes manuais, um testador humano assume o papel de um usuário final de um dado sistema sob teste para assegurar que o comportamento do mesmo corresponde ao que se espera. Por sua vez, testes automatizados de software englobam a automatização de atividades de teste, ou mais especificamente:

> Automação de teste é o uso de softwares especiais (separados do software sob teste) para controlar a execução de testes e a comparação dos resultados obtidos, com os resultados esperados. (HUIZINGA; KOLAWA, 2007 apud GAROUSI; MÄNTYLÄ, 2016) [18]

Ainda segundo Garousi e Mäntylä (2016) [18], algumas das atividades de teste com maior potencial de ganhos através do uso de automação, são as seguintes:

- Projeto de casos de teste: designação de lista de casos de teste ou requisitos de testes para satisfazer critérios de cobertura, ou outros objetivos de engenharia.
- Criação de roteiros de teste: documentação de casos de teste em roteiros de teste manuais ou código-fonte de testes automatizados.
- Execução de teste: executar casos de teste no sistema sob teste, e registrar resultados.
- Avaliação de teste: avaliar resultados de testes.
- Criação de relatórios sobre os resultados de testes: relatar veredictos/resultados de testes a desenvolvedores, por meio de sistemas de rastreio de defeitos.
- Gerenciamento de testes e outras atividades de engenharia de testes: incluem atividades como planejar, controlar, monitorar, e estimar esforço. Além disso, podem incluir minimização de suíte de testes ou seleção de testes de regressão.

Westfall (2016) [3], define, de maneira similar, automação de testes como o uso de software para automatizar tarefas de projeto de teste, execução de teste, captura e análise dos resultados de testes. Ainda segundo Westfall (2016) [3], testes automatizados podem, geralmente, ser executados mais rapidamente, e logo, mais frequentemente, quando comparados aos testes manuais, e isso pode ser útil, por exemplo, durante testes de regressão, principalmente em ciclos de desenvolvimento iterativos ou ágeis. Automatizar atividades de teste pode também resultar na diminuição de taxas de erro humano ao comparar grandes quantidades de dados resultantes com os dados esperados.

Como um dos pontos de atenção quanto ao uso de automação em testes, Westfall (2016) [3] cita os investimentos iniciais necessários em ferramentas e recursos para automação, e um investimento a longo prazo para prover manutenção às suítes de teste automatizados com o crescimento e a evolução do sistema sob teste. Para itens de software que devem mudar frequentemente, mais tempo/recursos podem acabar vindo a ser gastos tentando manter os testes automatizados atualizados, do que testes manuais exigiriam. Além disso, automatizar testes pode requerer habilidades e qualificações diferentes de testar por si só, por esse motivo, profissionais adicionais ou treinamento adicional para profissionais presentes podem ser necessários para implementar testes automatizados. Por fim, de acordo com Westfall (2016) [3], testes automatizados possuem certas limitações, em especial no que diz respeito ao fato de que somente é possível comparar resultados de teste com conjuntos finitos de resultados esperados, e dado que testes automatizados somente verificam o que está programado, os mesmos podem falhar ao identificar anomalias

que não pertencem ao conjunto pré-estabelecido de resultados esperados (seja sucesso ou falha).

4.1.2 Ferramentas para automação de testes

Polo et al. (2013) [2] classifica as ferramentas para automatização de testes nas seguintes classes principais:

- **XUnit Frameworks**: são as mais comumente utilizadas para automatizar testes. Utilizando essas ferramentas, casos de teste devem ser escritos em uma linguagem executável e podem ser, assim, executados automaticamente. Alguns exemplos de ferramentas pertencentes a essa categoria são:
  - JUnit e JTest no contexto de Java.
  - DOH e QUnit no contexto de Javascript.
  - C++ test e Cantata++ no contexto de C++.
  - NUnit e DbUnit.NET no contexto de .NET.
  - PHPUnit e Apache-Test no contexto de PHP.
  - HtmUnit e Selenium no contexto da *web*.
- **Capture and replay tools**: funcionam registrando as ações realizadas pelo usuário com o sistema que se deseja testar, salvando essas ações como um tipo de *script* de teste, e por fim, executando esses *scripts* automaticamente. Alguns exemplos de ferramentas pertencentes a esta categoria são:
  - TestComplete, uma ferramenta multiplataforma/multitecnologia.
  - Abbot e Jacareto, no contexto de Java.
  - Selenium, no contexto da *web*.

Umar e Zhanfang (2019) [19] classificam as ferramentas para automação de testes nas seguintes classes:

- **Ferramentas de teste unitário**: envolve o teste das unidades mais básicas do código-fonte. As ferramentas prestam auxílio ao processo de teste unitário, e são, em geral, as ferramentas mais utilizadas para automatizar testes, sendo facilmente integráveis como arcabouços, dentro de ambientes de desenvolvimento. Testes unitários são utilizados para averiguar o correto funcionamento de uma unidade/método particular de um sistema. Exemplos de ferramentas nessa categoria: JUnit e NUnit.

- **Ferramentas de teste funcional**: envolvem a verificação de funcionalidade de acordo com requisitos de usuário. Ferramentas desta categoria testam funções provendo-as entradas e examinando as saídas resultantes de acordo com os resultados esperados. Exemplos de ferramentas nessa categoria: Selenium e TestComplete.
- **Ferramentas de cobertura de código**: ferramentas utilizadas para determinar partes do código-fonte sob cobertura de testes automatizados. Capazes de colher medidas como número de linhas, declarações, ou blocos de código-fonte testados utilizando suítes de teste automatizadas. Geralmente, essas medidas são dadas em porcentagem, e em geral, quanto mais próximo de 100% de cobertura, menor a chance de que hajam defeitos não detectados presentes no sistema. Exemplos de ferramentas nessa categoria: Cobertura e Atlassian Clover.
- **Ferramentas de gerenciamento de teste**: utilizadas para automatizar atividades do processo de teste, como criação de casos de teste, planos de teste, estratégias de teste, etc.) e auxiliar equipes a gerenciar projetos, provendo um espaço reservado para atividades de teste que suporte buscas e que seja de fácil manutenção. Exemplos de ferramentas nessa categoria: Test Manager e Test Link.
- **Ferramentas de teste de performance**: prestam auxílio ao processo de testar a performance, e ajudam a determinar como um sistema desempenha em termos de responsividade e estabilidade em condições diversas e cargas diversas. Pode também servir para verificar e mensurar outros atributos de qualidade do sistema sob teste, como escalabilidade, confiabilidade e uso de recursos. Exemplos de ferramentas nessa categoria: JMeter e HP LoadRunner.

Por fim, de acordo com Chemuturi (2011) [20], os benefícios da adoção de automação e de ferramentas de automação de testes de forma apropriada, podem incluir desde melhorias (diminuição) em termos de tempo e esforço, principalmente após a segunda iteração utilizando testes automatizados; até aumento na facilidade em executar testes de regressão a cada iteração, algo que pode ser dificultado com testes manuais, que podem demandar muito tempo e esforço; e possibilidade de realização de tipos de teste que não são possíveis, ou são muito dificilmente executáveis de realizar manualmente, como testes de carga, de estresse, testes em paralelo ou concorrentes.

4.2 Automação de testes caixa preta

Alguns estudos demonstram exemplos de aplicação de técnicas e ferramentas de automatização de testes caixa preta, em diferentes tipos de sistemas, o que pode servir para demonstrar a versatilidade e variedade de possibilidades de aplicação da abordagem.

Edwards (2001) [21], por exemplo, propõe um arcabouço para automatizar testes caixa preta de sistemas embasados em componentes, que inclui: geração automática de testes dos componentes, geração automática de dados para testes caixa preta e geração automática/semi-automática de wrappers que determinam a corretude do resultado de testes.

Tan et al. (2009) [22] propõem o uso de uma ferramenta para teste caixa preta de uma biblioteca para programação funcional. A ferramenta proposta, não somente gera testes unitários, mas também testes de performance, estresse, e testes de segurança. Além de geração de casos de teste, a ferramenta provê funcionalidades para executar e avaliar testes.

Já Acuri (2021) [23], demonstra a aplicação da ferramenta EvoMaster para a aplicação de testes automatizados, tanto caixa preta, como caixa branca, de APIs RESTful. O autor busca mostrar como a ferramenta pode ser utilizada para gerar casos de teste que podem encontrar diversos comportamentos anômalos. No contexto desse estudo, para a avaliação utilizando a abordagem caixa preta, a ferramenta EvoMaster utiliza testes randômicos, ou seja as entradas são geradas de forma randômica, porém ainda sendo entradas válidas, e o objetivo é maximizar a cobertura dos status HTTP retornados pelo sistema sob teste.

Por fim, no que diz respeito a ferramentas para automatizar o processo de teste caixa preta, várias das ferramentas citadas anteriormente podem ser utilizadas para esse propósito, tendo em vista, também, que testes caixa preta foram definidos, anteriormente, como um tipo de teste que foca nos requisitos de um sistema, sem considerar o código-fonte, e a estrutura interna do sistema sob teste, mas sim, as saídas produzidas.

4.2.1 Automação de testes baseados em cenário

Tendo sido expostas anteriormente as definições dos conceitos de cenários de teste, e de testes embasados em cenário, e sabendo como desenvolver cenários de testes, e casos/*scripts* de teste a partir desses cenários, assim como na seção anterior, pode ser possível utilizar uma das ferramentas de automação de teste citadas anteriormente para aplicar automação a testes embasados em cenários. Alguns estudos utilizam abordagens de automatização de testes embasados em cenários, também em diferentes tipos de sistemas.

Lima e Faria (2015) [24] salientam a importância de testes de software atualmente, bem como sua dificuldade, e propõem uma abordagem e um conjunto de ferramentas para automatizar testes de serviços em sistemas distribuídos e heterogêneos. Na abordagem proposta, o testador interage com um sistema de modelagem visual para descrever cenários de comportamentos chave, iniciar geração de casos de teste, e execução dos mesmos, e visualizar os resultados e métricas de cobertura.

Campos et al. (2016) [25] destacam o problema de usabilidade, e de garantir que uma aplicação permita aos usuários atingirem suas metas/objetivos. De acordo com o autor, uma das maneiras de atingir boas métricas de usabilidade e satisfação do usuário final, consiste em analisar tarefas, e prover compatibilidade entre a aplicação interativa, e os modelos de tarefas correspondentes. O autor propõe uma abordagem que recorre a técnicas de testes baseados em modelos para gerar automaticamente cenários de para execução sinérgica automatizada, que consiste em validar um sistema de acordo com modelos de tarefas, coexecutando o sistema e seus modelos de tarefas e comparando o comportamento do sistema contra o que é descrito nos modelos. Os cenários são gerados a partir dos modelos, e são coexecutados sobre o modelo de tarefas e o sistema. A automação da execução dos cenários finaliza o processo.

Capítulo 5

Elementos de métodos e processo

Neste capítulo, método e processo adotados serão descritos. Primeiramente, serão descritos elementos do método de desenvolvimento Programação Extrema (*eXtreme Programming*), então, será descrito processo para definição de cenários de teste, bem como técnica utilizada para derivação de casos de teste a partir de cenários. Por fim, serão apresentados critérios para avaliação e comparação de ferramentas de automação de testes no contexto de aplicações *web*.

5.1 Elementos da Programação Extrema

Antes de descrever elementos da Programação Extrema (*XP*) é importante apresentar características de métodos ágeis em geral. Segundo Bourque (2014) [5], os métodos de desenvolvimento ágeis começaram a surgir nos anos 90, a partir da necessidade de reduzir a sobrecarga de trabalho que outros métodos de desenvolvimento geravam em projetos desenvolvidas na época.

Métodos ágeis propõem um fluxo de trabalho "leve", sendo caracterizados por ciclos de desenvolvimento curtos e iterativos, equipes auto-organizadas, projetos (*design*) simplificados, refatoração de código, desenvolvimento guiado por testes, envolvimento frequente das partes interessadas, ênfase na criação de produtos funcionais e demonstráveis a cada ciclo de desenvolvimento, entre outras práticas. Alguns exemplos de métodos de desenvolvimento ágeis são: [5]

- *Rapid Application Development (RAD)*;
- *eXtreme Programming (XP)*;
- *Scrum*;
- *Feature-Driven Development (FDD)*.

A Programação Extrema é baseada em histórias de usuário ou cenários para especificação de requisitos, no desenvolvimento de testes antes do desenvolvimento, no envolvimento direto do cliente com a equipe, programação em pares, no refatoramento de código contínuo e na integração contínua. Histórias de usuárias são divididas em tarefas, priorizadas, estimadas, desenvolvidas e testadas. A cada incremento o sistema deve ser testado, tanto com testes manuais como automatizados. [5]

5.1.1 Fases e atividades

Pressman (2010) [6] destaca as seguintes fases e atividades na Programação Extrema:

- **Planejamento**: Se inicia com conversações com partes interessadas, atividade em que membros técnicos da equipe *XP* podem participar, o que os possibilita melhor entendimento, do contexto de negócio, saídas esperadas e funcionalidades essenciais.

  As conversações dão início à criação de histórias de usuário, que descrevem entradas, saídas, características e funcionalidades do sistema a ser construído. As histórias podem ser escritas pelos clientes, e a cada história é atribuído um valor pelo cliente, baseado no valor de negócio que a funcionalidade relacionada possui. Além disso a equipe *XP* avalia cada história e atribui um custo medido em tempo a elas. As histórias, se muito custosas, podem ser divididas em histórias menores e menos custosas.

  Clientes e desenvolvedores trabalham juntos para decidir a melhor forma de agrupar as histórias para a próxima iteração. Com a progressão do processo de desenvolvimento, clientes podem adicionar novas histórias, mudar o valor de uma história existente, dividir histórias em componentes menores ou removê-las.

- **Projeto (*Design*)**: O projeto (*design*) no contexto do *XP* foca no princípio da simplicidade, sendo que sempre se prefere um projeto mais simples ante um projeto mais complexo.

  A utilização de alguns mecanismos é encorajada durante a atividade de projeto, como cartões *CRC* (*class-responsibility-collaborator*), que identificam e organizam as classes, no contexto de orientação a objetos, que são relevantes para o incremento em desenvolvimento.

  Por produzir poucos produtos de trabalho (*work products*), o projeto (*design*) é considerado transitório, e pode ser constantemente modificado com o prosseguimento da construção do sistema. Dessa forma, refatorações podem também ser aplicadas aqui.

Uma prática em *XP* é o projeto (*design*) ocorrer tanto antes como após o início do desenvolvimento. Refatorar pode significar que o projeto (*design*) ocorre continuamente, conforme o sistema se estrutura, de forma que a própria atividade de construção ajuda a equipe a realizar melhorias de projeto.

- **Desenvolvimento**: Após a elicitação das histórias de usuário, e o esforço preliminar de projeto (*design*) é realizado, a equipe desenvolve testes unitários que exercitam as funcionalidades descritas nas histórias, e somente então a implementação ocorre, com foco no que deve ser implementado para passar nos testes.

  Outra prática em *XP* é a programação em pares. É recomendado que uma dupla de desenvolvedores trabalhe juntos para a criação de código-fonte para uma história. Isso provê mecanismos de melhora na soluções de problema, e de garantia de qualidade, já que um desenvolvedor pode avaliar o código-fonte sendo desenvolvido por outro.

  Ao ser finalizado, o código-fonte é integrado com o trabalho de outros desenvolvedores, seja por uma equipe de integração, ou pelo próprio par responsável pela funcionalidade.

- **Teste**: Os testes, como exposto anteriormente, são geralmente criados antes da implementação. Os casos de teste podem ser implementados utilizando algum *framework* que possibilite automatizá-los, de forma que seja possível executá-los sempre que código-fonte é modificado, ou adicionado.

5.1.2 Equipes com um desenvolvedor

Para equipes com um único desenvolvedor, Agarwal e Umphress (2008) [4] propõem adaptações à Programação Extrema (*XP*) e ao *Personal Software Process*, para que seja criado um novo método, o qual denominam *Personal Extreme Programming (PXP)*.

Várias das práticas do *XP* se baseiam no trabalho em equipe, como é o caso da programação em pares. O *PXP* adequa o *XP* de forma que as práticas possam ser adotadas em equipes de um único desenvolvedor. [4]

O *PXP* sugere um *script* constituído pelas seguintes fases e atividades: [4]

Entrada: Padrão de desenvolvimento/programação.

1. **Planejamento**:

   (a) Obter declarações de requisitos.

      i. Escrever Metáfora de Sistema (*System Metaphor*).

ii. Escrever Histórias de Usuário.

(b) Dividir cada História de Usuário em funcionalidades.

(c) Agrupar funcionalidades em Conjuntos de Funcionalidades de acordo com suas características.

(d) Escrever testes de aceitação para os Conjuntos de Funcionalidades.

(e) Ordenar Conjuntos de Funcionalidades de acordo com prioridade para criar uma Lista de Prioridades de Conjuntos de Funcionalidades.

(f) Estabelecer um planejamento de iteração.

2. **Desenvolvimento**:

(a) Até que a Lista de Prioridades de Conjuntos de Funcionalidades esteja vazia, escolher o Conjunto de Funcionalidades no topo.

i. Se uma mudança foi introduzida nesse Conjunto de Funcionalidades, atualizar Conjunto de Funcionalidades, e reordenar Lista de Prioridades de Conjuntos.

(b) Dividir cada Funcionalidade do Conjunto de Funcionalidades escolhido em tarefas.

(c) Ordenar tarefas de acordo com suas prioridades, e criar uma Lista de Prioridades de Tarefas.

(d) Até que a Lista de Prioridades de Tarefas esteja vazia, escolher a tarefa no topo.

i. Escrever testes unitários para a tarefa.

ii. Escrever/Modificar código para implementar a funcionalidade requerida pela tarefa.

iii. Compilar e executar testes unitários para o código-fonte desenvolvido.

iv. Se os testes unitários apontarem sucesso, realizar testes de aceitação.

A. Se os testes de aceitação apontarem sucesso, integrar código-fonte desenvolvido, à base de código a refatorar.

B. Se não, criar nova tarefa de correção de código, com prioridade 1.

v. Se não, criar nova tarefa de correção de código, com prioridade 1.

vi. Realizar testes de integração na base de código a refatorar.

A. Se os testes de integração apontarem sucesso, refatorar o código na base de código a refatorar, e integrar código à base de código de produção.

B. Se não, criar nova tarefa de correção da base de código a refatorar, com prioridade 1.

vii. Realizar testes de aceitação na base de código de produção.

A. Se os testes de aceitação apontarem sucesso, realizar *release* da iteração, e se há um novo Conjunto de Funcionalidades, atualizar Lista de Prioridades de Conjuntos de Funcionalidades.

B. Se não, criar nova tarefa de correção da base de código de produção, com prioridade 1.

3. ***Post Mortem***:

(a) Completar testes de aceitação para base de código de produção.

Saída: Programa/sistema testado.

5.1.3 Histórias de usuário

É possível notar através das seções anteriores, que histórias de usuário (*user stories*) são elementos relevantes na Programação Extrema. Além disso, serão importantes no projeto sendo realizado, tanto para a especificação dos requisitos do sistema a ser desenvolvido, como para a especificação dos cenários de teste. Por esses motivos, histórias de usuário, bem como seu processo de criação, serão melhor descritos.

Segundo Bourque e Fairley (2014) [5] histórias de usuário são comumente utilizadas em abordagens adaptativas, e referem-se a descrições curtas e de alto nível das funcionalidades requeridas expressas em linguagem compreensível pelo cliente. Uma história de usuário deve conter informação para que desenvolvedores possam produzir estimativas do esforço a ser empregado para implementar as funcionalidades descritas.

Histórias de usuário são tipicamente criadas durante reuniões entre desenvolvedores e partes interessadas do sistema; e identificam o usuário, a necessidade do usuário, e a justificativa para a existência dessa necessidade.[3] Desse modo, pode-se escrever uma história de usuário seguindo o formato geral da forma: "Como um <**papel**>, eu quero que <**meta/desejo**> para que <**benefício**>". [5]

A utilização de histórias de usuário pode reduzir o desperdício que frequentemente ocorre quando requisitos detalhados são elicitados em fases iniciais, mas tornam-se inválidos antes do trabalho efetivamente ser iniciado. [5]

Para Sommerville (2016) [7] o principal ponto negativo do uso de histórias de usuário está na completude, já que pode ser difícil julgar se histórias suficientes foram criadas para abranger todos os requisitos essenciais de um sistema. Também pode ser difícil julgar se

uma única história provê uma descrição “verdadeira” ou precisa de uma determinada atividade.

5.2 Elementos do processo de criação de cenários

Cenários de teste foram previamente definidos, e nesta seção, devido à sua importância para o projeto proposto, será formalizado um processo para a criação/derivação dos mesmos a partir da análise dos requisitos do sistema.

5.2.1 Elicitação/criação de cenários

Ryser e Glinz (2000) [14] propõem um processo para elicitação/criação de cenários, dividido nas seguintes etapas:

1. Listar todos atores que interagem com o sistema.
    - **Resultados**: Lista de atores.
2. Listar todos eventos externos relevantes ao sistema.
    - **Resultados**: Lista de eventos e quem ou o quê os causam.
3. Determinar resultados que se espera que o sistema produza, bem como saídas e entradas relacionadas.
    - **Resultados**: Lista de entradas/saídas do sistema.
4. Determinar limites do sistema, o que faz, e o que não faz parte do sistema.
    - **Resultados**: Limites do sistema/diagrama de contexto.
5. Criar cenários preliminares de alto nível, contendo somente nome e descrição curta. Cenários devem responder perguntas tais quais: Como cada ator interage com o sistema? Que invariantes e restrições existem?
    - **Resultados**: Lista de cenários preliminares.
6. Determinar todos cenários e priorizá-los de acordo com sua importância; verificar que há cenários criados para todas funcionalidades do sistema.
    - **Resultados**: Lista de cenários priorizados. Ligação entre cenários e atores.

7. Criar descrição passo-a-passo para os eventos e ações de cada cenário, estruturando cenários de acordo com um *template* dado.

    - **Resultados**: Lista de cenários priorizados. Ligação entre cenários e atores.

8. Criar um gráfico de dependências, que demostre dependências entre cenários. Criar um diagrama de visão geral.

    - **Resultados**: Gráfico de dependências e diagrama de visão geral.

9. Fazer com que usuários revisem e comentem os cenários e diagramas já produzidos.

    - **Resultados**: Comentários sobre os cenários.

10. Refinar cenários: estender cenários refinando a descrição do fluxo normal de ações; especificar comportamento normal em etapas únicas; descrever a sequência de estímulos e respostas do sistema detalhadamente.

    - **Resultados**: Descrição do fluxo normal de ações em cada cenário. Sugestões, instruções ou informações para derivação de casos de teste.

11. Modelar fluxos alternativos de ações, especificar exceções e como reagir a elas.

    - **Resultados**: Fluxos alternativos de ações, exceções e seu manuseio em cenários.

12. Retrabalhar cenários a fim de encontrar etapas de trabalho em comum. Fatorar sequências parecidas em "cenários abstratos" que possam ser utilizados em vários cenários normais.

    - **Resultados**: Cenários abstratos.

13. Incluir requisitos não-funcionais aos cenários.

    - **Resultados**: Cenários com requisitos não funcionais.

14. Revisar e ajustar o gráfico de dependências e o diagrama de visão geral de acordo com os cenários refinados.

    - **Resultados**: Gráfico de dependências e diagrama de visão geral estendidos e revisados.

15. Fazer com que usuários façam uma revisão e validação formal dos cenários.

    - **Resultados**: Cenários validados.

16. Estruturar cenários de acordo com *template* dado. Criar casos de teste preliminares; escrever plano e especificação de teste.

    - **Resultados**: Documento de especificação de cenários.

5.2.2 Descrição e *template* de cenário

Segundo Ryser e Glinz (2000) [14], há três principais formas de representação que podem servir para documentar cenários, sendo elas:

- Linguagem natural estruturada (utilizando um *template*);
- Representação tabular;
- Representação gráfica.

No trabalho proposto pelos autores, é sugerido o uso da primeira alternativa, já que, segundo os mesmos, em cenários representamos, principalmente, sequências de ações; e isso pode ser feito com conveniência utilizando linguagem natural e um esquema de numeração, por exemplo. [14]

A representação textual, no entanto, carrega consigo os pontos negativos da linguagem natural, caso não seja utilizada de forma restrita e precisa. Linguagem natural, por si só, pode não ser adequada para dar uma visão geral do sistema a um usuário, por exemplo, além de não ser ideal para demonstrar, de forma clara, conexões, relações e dependências entre estruturas e elementos. Mesmo assim, linguagem natural é entendida e falada por todos sem treinamento, além de ser fácil e simples de ser utilizada. Essas são vantagens de muito valor, já que isso possibilita que usuários e clientes se envolvam no processo de desenvolvimento sem ter que aprender notações ou linguagens especiais.

Tendo em vista que a representação textual será utilizada, é apresentado um *template* para cenários composto dos seguintes elementos: [14]

- **Identificador:** um rótulo único que identifique um dado cenário. Geralmente é um número ou uma palavra contendo elementos numéricos.
- **Nome:** responsável por prover uma concepção/visão/ideia a respeito do que trata o cenário. Transmite a essência do que um dado cenário faz, e é geralmente escrito como uma frase verbal curta.
- **Descrição/propósito/objetivo:** parágrafo curto onde a intenção do cenário é sumarizada. Objetivo do cenário, e resultado esperado após um ator interagir com o sistema.

- **Atores:** quem interage com o sistema no cenário. Quem aciona o cenário, e a quem o cenário entrega os resultados.
- **Pré-condições:** condições que devem ser atendidas antes do cenário ser executado.
- **Pós-condições:** resultados esperados ao ator que acionou o cenário, e ao estado do sistema quando a execução do cenário chega ao fim.
- **Gatilho:** nomeia o evento gatilho que leva à execução do cenário, geralmente iniciado por um ator.
- **Fluxo normal:** descrição de uma sequência passo a passo de ações, eventos e respostas do sistema geradas durante o comportamento normal e esperado do sistema.
- **Fluxos alternativos:** Lista os fluxos de ação que são gerados por comportamentos excepcionais e alternativos do sistema.
- **Requisitos não-funcionais:** os requisitos que não puderam ser incluídos nos fluxos anteriores são listados aqui.
- **Versão/Histórico de mudanças:** atribui um número de versão a cada *release* de um cenário a fim de ajudar a gerenciar versões de cenários.
- **Proprietário:** o desenvolvedor que escreveu o cenário e o mantém atualizado.
- **Tipo**: "normal" ou "abstrato".

5.2.2.1 Regras relativas ao uso de linguagem natural

Como exposto anteriormente, linguagem natural possui algumas desvantagens quando utilizada como linguagem de especificação. Tendo isso em vista, são apresentadas algumas orientações e regras gerais para a forma com que as descrições são escritas. Seguindo as orientações, certas palavras ou expressões que comumente levam a ambiguidade ou mal-entendidos, ou que exigem interpretação, palavras que indicam omissão ou imprecisão, e termos vagos, são evitados ou substituídos por alternativas definidas. Os principais termos são os seguintes: [14]

- **"E"**. Em linguagem natural pode ter três significados:
  - Concorrência de eventos ou ações, ou capturando o fato de que múltiplas condições devem ser cumpridas;
  - Ordenação temporal;
  - Enumeração, sem implicar concorrência ou ordenação temporal.

Sempre que possível, frases contendo “E” devem ser serializadas, sendo reescritas como uma sequência de ações, isso só não é possível no caso em que denotam concorrência. Nesse caso, a concorrência deve ser marcada adicionando-se “ao mesmo tempo”.

Enumerações e ordenações podem ser marcadas adicionando-se “em qualquer ordem”, e “nessa ordem em específico”, respectivamente, mas preferencialmente, devem ser escritas como uma sequência numerada de ações, mesmo que não necessário.

- **“OU”**. Pode ser exclusivo ou inclusivo.

  No caso do exclusivo, recomenda-se utilizar “OU [alternativa] OU [alternativa]”.

  No caso do inclusivo recomenda-se utilizar “OU [alternativa] OU [alternativa] OU AMBOS”.

- **Termos vagos**. “Vários, alguns, poucos, muitos...”. Recomenda-se especificar as quantidades e qualidades.

- **Indicadores de exceções**. “Embora, mas, às vezes, ainda, no entanto...”. Exceções devem ser capturadas por fluxos alternativos, portanto, esses termos devem ser evitados no fluxo normal.

- **Termos indicadores de incerteza**. “Pode, deveria, possivelmente...”. Em caso de incerteza a respeito dos requisitos, a incerteza deve ser indicada claramente.

- **Pronomes**. Frequentemente, a referência não é clara. Em lugar de utilizar pronomes, recomenda-se repetir explicitamente a palavra referenciada.

Além disso, é importante manter uma estrutura frasal simples, com um único sujeito e no máximo um objeto direto, e um indireto. O sujeito deve estar claramente definido e deve ser um ator do sistema. Deve-se evitar períodos compostos, cada passo em um cenário deve ser uma sentença simples sem orações subordinadas. [14]

5.3 Ferramentas para automação de testes selecionadas

As ferramentas selecionadas para comparação são os *frameworks* para automação de testes de aplicações *web* *Cypress* e *Selenium*. Ambas as ferramentas dispõem de funções para interagir com a página exibida no navegador, através de métodos localizadores. Dessa forma, é possível selecionar elementos *HTML* presentes na página através de atributos como *id*, *name*, *tag*, *class*, entre outras estratégias de localização. Sendo assim, espera-se

que ambos os *frameworks* sejam capazes de desempenhar funções semelhantes quanto ao teste de aplicações *web* através de interação automatizada com interface ,ou seja, através da interação com os elementos *HTML*; e utilizando uma estratégia de teste de caixa preta (*black-box testing*).

A escolha das ferramentas levou em conta, entre outros fatores, a popularidade das mesmas, sendo ambas frequentemente listadas entre *frameworks* para automação de testes no contexto da *web* populares. Os *frameworks* são listados entre os mais populares, por exemplo, por Rajora (2021) [26] e por Badkar (2023) [27]. Além disso, comumente são feitas comparações que incluem ambas ferramentas, por exemplo, por Pelivani et al. [28]. Por fim, é importante destacar que para a escolha, foram considerados, também, as experiências e interesses do próprio autor do trabalho.

5.3.1 *Framework Selenium*

O *framework Selenium*, segundo portal acerca da ferramenta [29], é um conjunto de ferramentas para automação de navegadores *web*. É utilizado, primariamente, para automatizar testes de aplicações *web*, mas é capaz de automatizar outras tarefas em interfaces *web*.

O *framework* provê extensões para emular a interação do usuário com interfaces *web*, simulando atividades comuns desempenhadas por usuários finais, como entrada de texto em campos de entrada, marcação de *checkboxes*, clique em *links*, etc. Provê também um servidor de distribuição para alocação escalável de navegadores; e infraestrutura para as implementações da especificação *W3C WebDriver* que permite a escrita de código-fonte intercambiável entre navegadores *web*. [29]

A ferramenta é composta de três componentes principais, que são os seguintes: [29]

- ***WebDriver:*** utiliza as interfaces (*APIs*) de automação providas pelos distribuidores de navegadores *web* para controlar o navegador e executar testes. Funciona como se um usuário real estivesse operando o navegador. Como o *WebDriver* não requer que sua *API* seja compilada junto à aplicação, não é intrusivo, e dessa forma, é possível testar a mesma aplicação que está em produção.
- ***IDE (Integrated Development Environment):*** ferramenta que pode ser utilizada para desenvolver casos de teste do *Selenium*. É uma extensão para os navegadores *Chrome* e *Firefox* capaz de gravar ações dos usuários e transformá-las em comandos do *Selenium*, com parâmetros definidos pelo contexto dos elementos.
- ***Grid:*** permite executar casos de teste em diferentes máquinas através de diferentes plataformas. O controle do acionamento dos casos de teste é feito localmente, e

quando casos de teste são acionados, são executados, de forma automática, remotamente. É utilizado quando se necessita executar testes em combinações de múltiplos navegadores e sistemas operacionais.

5.3.2 *Framework Cypress*

Segundo portal acerca da ferramenta, o *framework Cypress* pode facilitar a criação de testes para aplicações *web*; permite a depuração com auxílio de interface gráfica; e possibilita executar, automaticamente, os testes desenvolvidos em estruturas de integração contínua. [30]

A ferramenta é direcionada, especialmente, a desenvolvedores e profissionais de garantia de qualidade (*Quality Assurance*) que estejam desenvolvendo aplicações *web* utilizando *frameworks Javascript* modernos. O portal da ferramenta lista como principais vantagens e funcionalidades da ferramenta as seguintes: [30]

- **Facilidade de instalação.** É possível instalar via gerenciadores de pacotes de projetos *Node.js*.
- **Configuração guiada.** Ao iniciar o *Cypress* pela primeira vez, o desenvolvedor é apresentado um guia que o auxiliará com decisões e tarefas de configuração.
- **Testes legíveis.** A ferramenta propõe uma forma simplificada de escrever testes, de forma que seja simples ler e entendê-los. Escrever testes com *Cypress* deve ser como prover comandos descritivos para um usuário real executar.
- **Clique e grave.** Utilizando o *Cypress Studio* é possível gerar *scripts* de acordo com interações feitas com a aplicação. Adicionalmente, é possível, através de um seletor interativo, gerar comandos para localizar qualquer elemento na interface.
- **Tudo em um *framework*.** A ferramenta vem com tudo necessário para configurar uma suíte de testes, dessa forma, menos tempo é gasto gerenciando *drivers* e outras dependências.
- **"Viagem temporal".** É possível observar o comportamento da aplicação durante a execução dos passos dos testes. Pode-se verificar com quais elementos o *Cypress* interagiu, e como a aplicação reagiu.
- ***Reload* em tempo real.** É possível observar comandos sendo executados, e a aplicação sob teste, lado a lado, em tempo real. Testes são reexecutados automaticamente, ao salvar arquivos, para uma resposta instantânea para que seja possível guiar o desenvolvimento com testes.

- **Inspeção nativa do navegador.** É possível utilizando ferramentas já incluídas nos navegadores enquanto os testes são executados.
- **Vídeos e capturas de tela.** Vídeos e capturas de tela são produzidas automaticamente para facilitar a depuração de falhas.
- **Fluxos de *cloud* integrados.** A ferramenta permite gerenciar, localmente, a saúde de projetos e revisar, reexecutar e depurar testes gravados ao *Cypress Cloud*.
- ***Wait* automático.** Não deve ser necessário adicionar *waits* ou *sleeps* arbitrários, já que o *framework* é capaz de aguardar por comandos e asserções antes de continuar.
- ***Retries* de testes.** As consultas ao *DOM* são envolvidas por uma lógica de *timeout* e *retry*, de forma que quando um teste falha, são realizadas múltiplas tentativas de asserções. Dessa forma, minimizam-se falsos negativos e positivos.
- **Isolamento de testes.** O *framework* reduz resultados instáveis (*flaky*) de testes ao isolar o estado de cada teste, limpando o estado do navegador antes da execução de testes.
- **Detecção de instabilidade (*flake*).** O *Cypress* reexecuta, automaticamente, testes falhos, a fim de evitar que testes instáveis (*flaky*) ocasionem a falha de execuções por completas, ou a falha de *builds* de integração contínua.
- **Resultados consistentes.** A arquitetura da ferramenta não utiliza o *Selenium* ou o *WebDriver*. O *framework* foi construído por completo a fim de obter resultados mais estáveis.

5.4 Critérios para avaliação e comparação

Para a análise comparativa dos *frameworks* selecionados, é necessário selecionar critérios para avaliar e comparar as ferramentas. Para tal, foram reunidos critérios definidos e utilizados em múltiplas fontes para comparação de ferramentas de automação de testes.

Buscou-se priorizar critérios ou fatores objetivos e que possam ser, de alguma forma, quantificados e que independam da experiência ou opinião do desenvolvedor. Isso pois, desenvolvedores diferentes, podem vir a ter experiências discordantes mesmo trabalhando em condições similares. Mesmo assim, alguns critérios subjetivos serão trazidos e aplicados, apenas a título de complementação.

Illes et al. (2005) [31], utilizam uma abordagem orientada a atividades que podem ser, potencialmente, automatizadas, e a partir dessas atividades os seguintes critérios para avaliação de ferramentas de teste são derivados e agrupados:

1. **Planejamento e monitoramento de testes:**

   (a) Customização do processo de teste organizacional.

   (b) Linguagens/paradigmas de programação particulares, sistemas operacionais, navegadores e configurações de redes.

   (c) Características específicas da aplicação, que requerem técnicas específicas de teste.

   (d) Teste em domínios especiais.

   (e) Planejamento do processo de teste.

   (f) Monitoração de atividades de teste (monitorando tempo estimado e real por caso de teste; provendo métricas de cobertura do progresso das atividades de teste; provendo métricas de diferentes fontes).

   (g) Integração com outras ferramentas.

2. **Projeto de casos de teste:**

   (a) Projetar casos de teste para o nível desejado (unidade, integração, sistema).

   (b) Selecionar as técnicas de teste.

   (c) Definir condições de teste derivadas das técnicas de teste definidas.

   (d) Definir modelos para estruturação das informações especificando os casos de teste.

   (e) Geração de casos de teste lógicos a partir de modelos semi-formais.

   (f) Geração de casos de teste lógicos a partir de especificações formais.

   (g) Geração/derivação de disposição de dados de teste.

   (h) Otimização de conjuntos de casos de teste.

   (i) Projeto de casos de teste para testar critérios de qualidade da aplicação.

   (j) Restrição de conjuntos de casos de teste.

3. **Construção de casos de teste:**

   (a) Editar *scripts* de teste.

   (b) Desenvolver código-fonte de teste conforme práticas aceitas de engenharia de software.

   (c) Capturar casos de teste executáveis.

   (d) Geração de casos de teste concretos a partir de modelos.

   (e) Geração de dados de teste.

(f) Geração de *stubs*, *drivers* de teste, e objetos *mock*.

(g) Simulação de componentes faltantes/defeituosos.

4. **Execução de casos de teste:**

   (a) Configurar ambiente de teste, pré-condições e pós-condições para um conjunto de casos de teste.

   (b) Voltar ao início em caso de erros inesperados.

   (c) Executar casos de teste capturados, capturados e editados ou implementados manualmente para testes funcionais.

   (d) Executar casos de teste capturados para testar critérios de qualidade.

   (e) Parar e continuar a execução de um caso de teste.

5. **Captura e comparação de resultados de testes:**

   (a) Registrar/exibir informações a respeito dos casos de teste executados.

   (b) Facilidade de comparação entre os resultados esperados e obtidos.

6. **Reporte dos resultados de testes:**

   (a) Agregação dos resultados de testes exibidos.

   (b) Quantidade de informação customizável, de acordo com funções específicas.

7. **Monitoramento de problemas e defeitos reportados:**

   (a) Especificar reportes de problemas utilizando modelos predefinidos.

   (b) Geração de entradas para defeitos registrados.

   (c) Priorizar defeitos.

   (d) Monitorar pedidos de mudanças e defeitos e sua situação atual.

   (e) Geração de informação estatística.

   (f) Testes de regressão.

8. **Gerenciamento de artigos de teste:**

   (a) Gerenciar artigos de teste.

   (b) Rastreabilidade entre elementos dos artigos de teste.

   (c) Rastrear modificações em um objeto de teste e comunicar mudanças.

   (d) Manutenção de dados de testes.

   (e) Reutilização de testes automatizados em testes de regressão ou outros projetos.

(f) Facilidade de realização de *snapshots* (congelando um estado dos artigos de teste).

Gamido e Gamido (2019) [32] realizam análise comparativa de diversas ferramentas para automação de testes, utilizando os seguintes parâmetros de avaliação:

- Multiplataforma;
- Multinavegador;
- *Record Playback* (Habilidade da ferramenta de gravar ações);
- Facilidade de aprendizado;
- Guiado por dados (Habilidade da ferramenta de acessar dados de diferentes fontes externas);
- Habilidade de programação;
- Geração de reportes;
- Custo;
- Funcionamento (Tipos de teste suportados).

Por fim, Singh e Tarika (2014) [33] também realizam análise comparativa de ferramentas de automação de testes de software, utilizando critérios semelhantes ao citados anteriormente:

- Capacidade de gravação;
- Tempo de execução;
- Geração de *scripts*;
- Testes guiados por dados;
- Facilidade de aprendizagem;
- Reportes/saídas de testes bem formatados.

Capítulo 6

Elementos do processo de desenvolvimento

Neste capítulo, primeiramente, a aplicação proposta e suas partes são descritas, explicitando-se a ideia geral e forma de funcionamento da mesma. Além disso, as tecnologias e *frameworks* utilizados durante o desenvolvimento serão apresentados. Posteriormente, as etapas do método de desenvolvimento adotado são aplicadas conforme especificado no capítulo anterior, e elementos relevantes produzidos durante esse processo são documentados.

6.1 A aplicação proposta

A aplicação proposta foi criada no contexto deste trabalho com o intuito de possibilitar a criação e execução de testes. A aplicação é chamada, arbitrariamente, de *Lend Me A Book* (em português, Me Empreste Um Livro), e consiste em uma plataforma fictícia de empréstimo, *reviews* e discussões gerais a respeito de livros. O sistema proposto inspira-se, ao mesmo tempo, em redes sociais e em plataformas de comércio eletrônico (*e-commerce*).

A ideia da plataforma proposta baseia-se em uma abordagem descentralizada, ou seja, os próprios usuários criam e gerenciam seu conteúdo, e se comunicam com outros usuários, aos moldes do que ocorre em plataformas como *OLX*, *Mercado Livre*, e em redes sociais, em geral.

Através da aplicação proposta, usuários interessados em conhecer novos livros, emprestar livros que possuem, ou participar de discussões a respeito de livros já lidos, devem ser capazes de buscar na plataforma por livros que possam pegar emprestado, oferecer seus próprios livros para empréstimo, escrever e postar análises sobre livros que já leram, e participar de discussões iniciadas em análises de outros usuários.

As funcionalidades principais da aplicação foram propostas pensando-se em casos de uso comuns de aplicações *web* e *mobile* atuais e populares, como autenticação, criação de *posts* (ou anúncios), interação com outros usuários, e recebimento de notificações. Além disso, as funcionalidades foram propostas já com o intuito de testá-las posteriormente.

6.1.1 Tecnologias utilizadas e ambiente de desenvolvimento

Para o desenvolvimento da aplicação proposta, foram utilizadas algumas ferramentas visando, em especial, a diminuição do *overhead* de desenvolvimento, já que esse não é, necessariamente, o foco do projeto.

Para o desenvolvimento do *front-end* da aplicação, foi utilizado o *framework Next.js*. Segundo o portal na *web* da ferramenta, com *Next.js* é possível desenvolver interfaces de usuário utilizando componentes *React* (biblioteca *front-end Javascript*). Segundo o portal, algumas das principais funcionalidades do *framework* são as seguintes: [34]

- Roteamento baseado em sistema de arquivos construído sobre *Server Components* e que suporta *layouts*, roteamento aninhado, estados de carregamento e tratamento de erros;
- Renderização no lado do cliente (*client-side*) e no lado do servidor (*server-side*) com componentes de servidor (*Server Components*) e do cliente (*Client Components*). Renderização estática e dinâmica no servidor;
- Busca de dados simplificada com suporte a *async/await* nos componentes *React* e uma *API* de *fetch* que se alinha com as plataformas *React* e *Web*;
- Estilização utilizando métodos de estilização populares, como *CSS Modules*, *Tailwind CSS* e *CSS-in-JS*;
- Otimizações em imagens, fontes e *scripts* a fim de melhorar a saúde de aplicações e a experiência do usuário;
- Suporte melhorado ao *Typescript*, com melhorias na checagem de tipos e compilação mais eficiente.

Para o *back-end* da aplicação, foi utilizada a plataforma *Firebase* provida pelo *Google*. Segundo o portal na *web* da plataforma, o *Firebase* auxilia no desenvolvimento e crescimento de aplicações e jogos, provendo soluções para: [35]

- **Criação**: Acelerando o desenvolvimento de aplicativos com uma infraestrutura *back-end* gerenciada;

- **Liberar e monitorar**: Auxiliando na realização de lançamentos e monitoração do desempenho e estabilidade;
- **Engajamento**: Auxiliando no aumento do engajamento de usuários com análises avançadas, testes A/B e campanhas de mensagens.

Além disso, foram utilizadas algumas bibliotecas com o intuito de agilizar o desenvolvimento. Em especial foi utilizada a biblioteca de componentes de interface *Chakra UI* [36], que provê uma série de elementos *HTML* pré-desenvolvidos e customizáveis; e a biblioteca de funções *React Firebase Hooks* [37], que facilita a integração dos serviços providos pelo *Firebase*, com o *front-end* desenvolvido em *React*, ou *Next.js*, nesse caso.

Por fim, o ambiente de desenvolvimento, consiste, em suma, no sistema operacional *Windows 11*, com o editor de texto *Visual Studio Code*, que permite instalação de extensões para facilitar o processo de desenvolvimento e auxiliar o desenvolvedor.

6.2 Aplicando elementos do processo de desenvolvimento proposto

Nesta seção, elementos do método de desenvolvimento proposto anteriormente (*Personal eXtreme Programming*) serão aplicados, segundo o contexto da aplicação proposta. É importante notar que o desenvolvimento foi realizado em um período aproximado de duas semanas, e por tanto, uma única iteração.

6.2.1 Levantamento de requisitos

Inicialmente, deve ser feito o levantamento de requisitos, que na Programação Extrema é feita através da elicitação e criação de histórias de usuário junto a partes interessadas, as quais podem ser construídas segundo o formato geral especificado anteriormente. Aqui, devido ao objetivo do sistema, as histórias de usuário não são criadas junto a partes interessadas. Tendo isso em vista, as histórias de usuário criadas para a plataforma proposta são listadas a seguir:

1. Como **usuário**, quero ser capaz de **realizar registro na plataforma**, para que possa **ter acesso imediato e futuro**.
2. Como **usuário**, quero ser capaz de **entrar na plataforma, utilizando credenciais registradas anteriormente**, para que possa **acessar e utilizar os recursos da plataforma**.

3. Como **mutuário (aquele que recebe o empréstimo)**, quero ser capaz de **buscar livros disponíveis para empréstimo**, para que possa **fazer um pedido de empréstimo**.

4. Como **mutuante (aquele que empresta)**, quero ser capaz de **registrar livros para empréstimo**, para que outros usuários possam **buscá-los e realizar pedidos de empréstimo**.

5. Como **usuário**, quero ser capaz de **agir como mutuante e como mutuário**, para que possa **tanto emprestar, como pegar livros emprestados**.

6. Como **usuário**, quero ser capaz de **criar postagens de análise sobre determinado livro**, para que possa **divulgar minha opinião, e interagir com outros leitores**.

7. Como **usuário**, quero ser capaz de **interagir, adicionando observações, críticas ou comentários gerais às postagens de outros usuários**, para que possa **divulgar minha opinião, e interagir com outros leitores**.

8. Como **usuário**, quero ser capaz de **buscar entre pedidos de empréstimo, livros disponíveis, e análises, utilizando filtros**, para que possa **encontrar, facilmente, informações relevantes**.

9. Como **usuário**, quero ser capaz de **excluir conteúdo adicionado por mim (pedidos de empréstimo, livros cadastrados para empréstimo, análises, e comentários em análises)**, para que possa **controlar minhas postagens disponíveis na plataforma**.

10. Como **usuário**, quero ser capaz de **editar conteúdo adicionado por mim**, para que possa **controlar minhas postagens disponíveis na plataforma**.

11. Como **usuário**, quero ser capaz de **excluir minha conta**, para que possa **remover meu perfil e conteúdo da plataforma**.

12. Como **usuário**, quero ser capaz de **receber notificações a respeito de atividades em minhas postagens**, para que possa **ser informado, de forma prática, sobre a situação das minhas postagens**.

6.2.2 Divisão em funcionalidades

A próxima etapa recomendada pelo *Personal eXtreme Programming* consiste na divisão das histórias de usuário em funcionalidades, de acordo com as funcionalidades necessárias

para completar a história. Aqui, não será criada uma lista de prioridades para as funcionalidades, de forma que assume-se que é atribuída a mesma prioridade a cada uma. Tendo isso em vista, e realizando a divisão, tem-se as seguintes funcionalidades, separadas em dois grupos principais apresentados nas tabelas 6.1 e 6.2:

Gerenciamento de conta e autenticação
Criação de conta
Login
Exclusão de conta

Tabela 6.1: Primeiro grupo de funcionalidades.

Criação e gerenciamento de postagens
Realização de pedido de empréstimo
Registro de livro para empréstimo
Criação de postagens de análise
Criação de comentários em postagens de análises
Exclusão de postagens de todos os tipos
Edição de postagens de todos os tipos
Recebimento de notificações

Tabela 6.2: Segundo grupo de funcionalidades.

6.2.3 Divisão em tarefas

A próxima etapa recomendada pelo método adotado consiste na divisão das funcionalidades em tarefas. Aqui também será atribuída a mesma prioridade a todas as tarefas, que poderão ser realizadas em ordem arbitrária, de forma que não será necessário a criação de uma lista de prioridades. Tendo isso em vista, e realizando a divisão, tem-se as seguintes tarefas:

- Criar funções para criação de conta.
- Criar formulário de criação de conta.
- Criar funções para *login*.
- Criar formulário de *login*.
- Criar funções para exclusão de conta.
- Criar interface de exclusão de conta.

- Criar funções para criação/edição/exclusão de postagens de empréstimo (pedidos e ofertas).
- Criar interface para criação/edição/exclusão de postagens de empréstimo (pedidos e ofertas).
- Criar funções para criação/edição/exclusão de postagens de análise.
- Criar interface de criação/edição/exclusão de postagens de análise.
- Criar funções para adição/edição/exclusão de comentários em análises.
- Criar interface para adição/edição/exclusão de comentários em análises.
- Criar funções para envio de notificações.
- Criar interface para recebimento de notificações.

6.2.4 Elementos da implementação e da aplicação resultante

Na página inicial, conforme apresenta a figura 6.1, o usuário tem acesso a uma listagem dos *posts* já feitos na plataforma, ordenados do mais recente até o mais antigo. Cada *post* é representado como um *card* que contém informações como o autor da postagem, data de postagem, imagem de pré-visualização, tipo de postagem, título e autor do livro, quantidade de comentários e primeiras linhas da descrição adicionada. É apresentada, também, uma barra de busca, que suporta busca por autor e título de livro, ou a limpeza dos critérios de busca.

Além disso, é possível notar a barra de navegação, conforme apresenta a figura 6.1, que estará sempre presente no topo da aplicação, e apresenta, quando o usuário não está logado: o logo da aplicação, e botões de *log in* e cadastro; e quando o usuário está logado: o logo da aplicação, e botões de adição de nova postagem, botão de notificações e botão de acesso a funções de perfil.

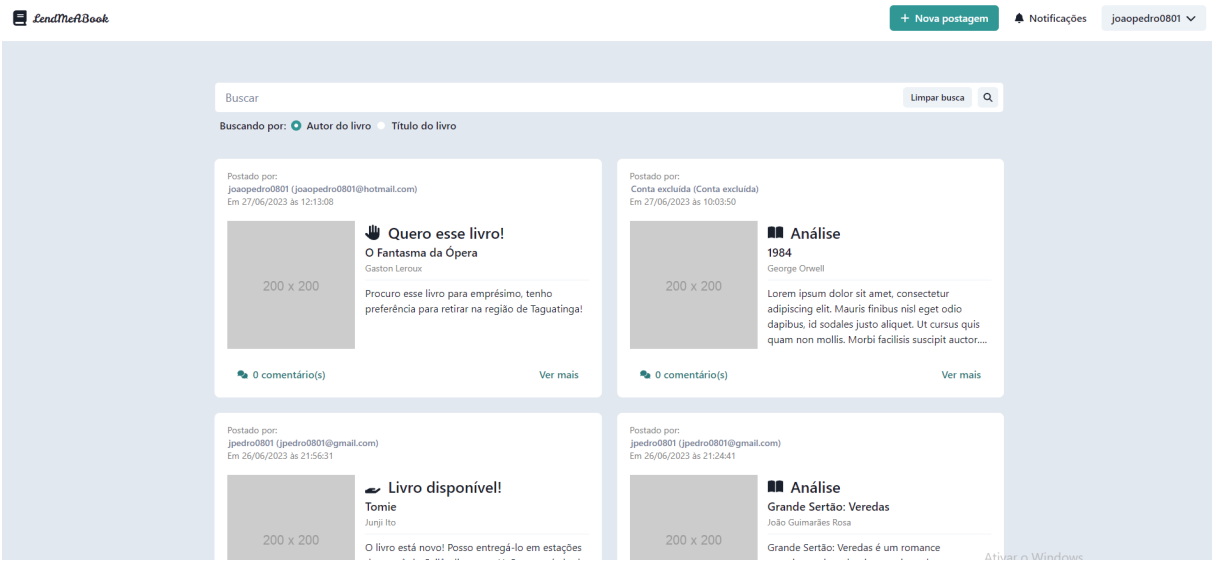

Figura 6.1: Captura de tela da página inicial da aplicação.

Ao clicar no botão “ver mais”, ou no botão “comentários”, contidos no *card* de *post*, o usuário é levado para a página de detalhes da postagem, onde o texto de descrição é exibido na íntegra, mais imagens adicionadas pelo autor da postagem são apresentadas, e os comentários do *post* são listados, bem como o autor dos comentários e o momento em que o comentário foi feito, ou editado pela última vez, conforme apresentam as figuras 6.2 e 6.3. Nessa página, apesar de não mostrada nas figuras, a barra de navegação no topo também é exibida, assim como ocorre na página inicial.

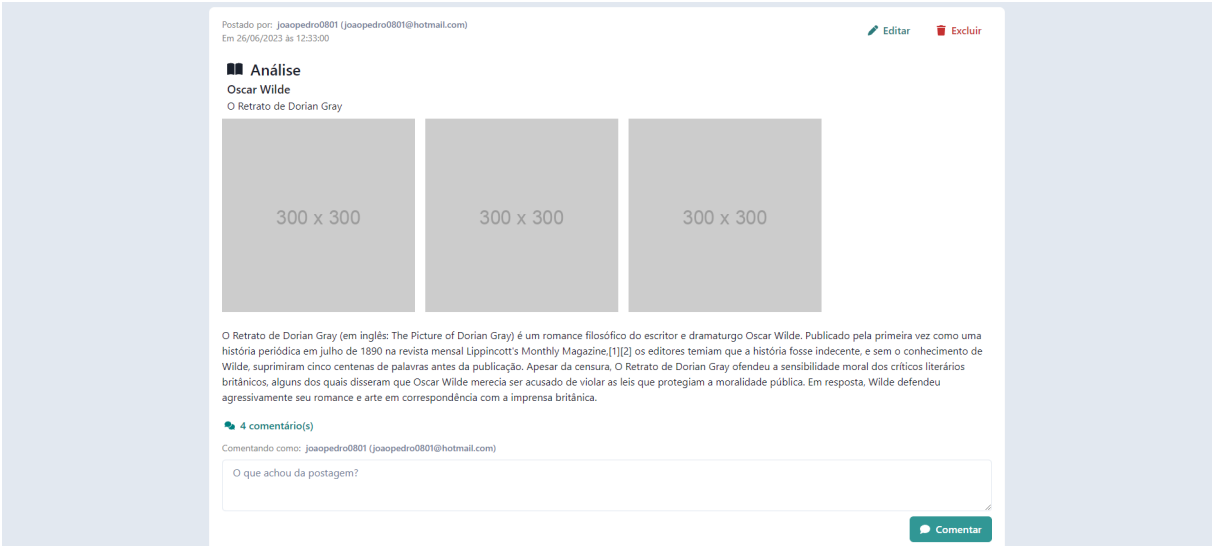

Figura 6.2: Primeira captura de tela da página de um *post*.

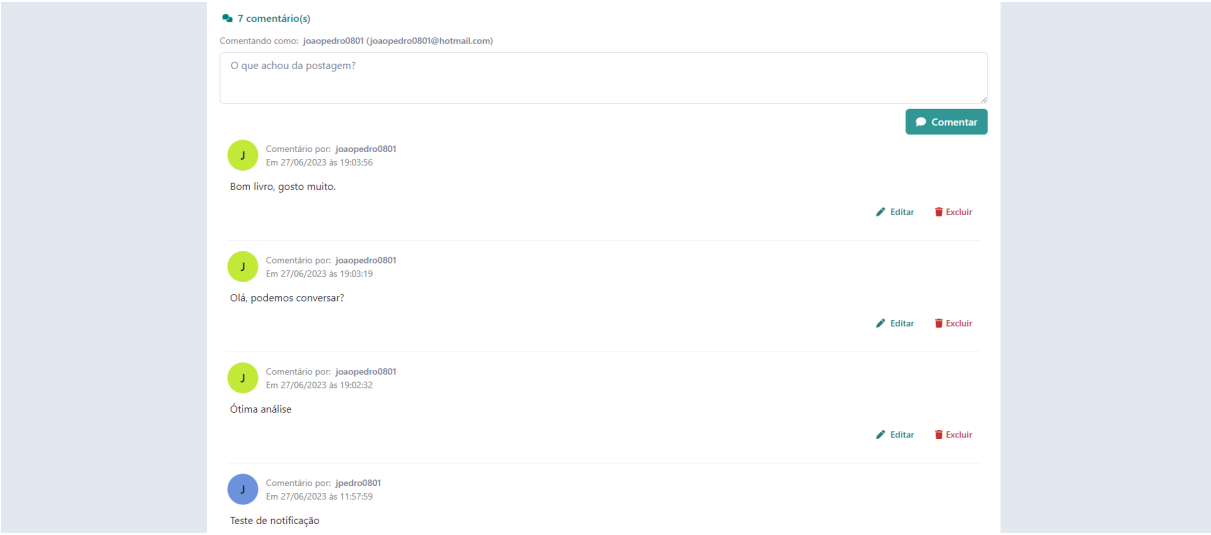

Figura 6.3: Segunda captura de tela da página de um *post*.

Utilizando a caixa de texto e o botão "comentar" apresentados na figura 6.3, é possível adicionar comentários às postagens de outros usuários, ou às postagens do usuário logado, sendo que comentários de autoria própria podem ser editados ou excluídos. Similarmente, postagens de autoria própria podem ser editadas ou excluídas através dos botões "editar" e "excluir", conforme apresenta a figura 6.2.

Parte das informações e funcionalidades da aplicação são exibidas como modais, de forma que a exibição das mesmas depende de eventos iniciados pelo usuário, como pressionar botões. É o caso das seguintes funcionalidades, exibidas nas figuras 6.4, 6.5, 6.6, 6.7 e 6.8:

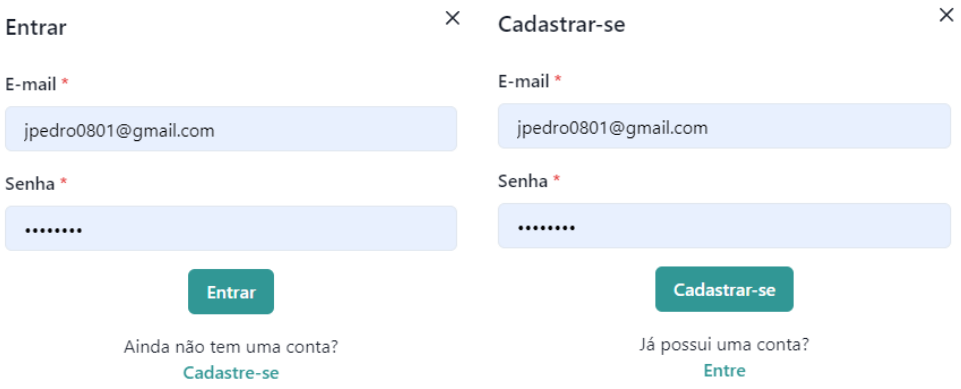

Figura 6.4: Captura de tela das modais de *login* e cadastro.

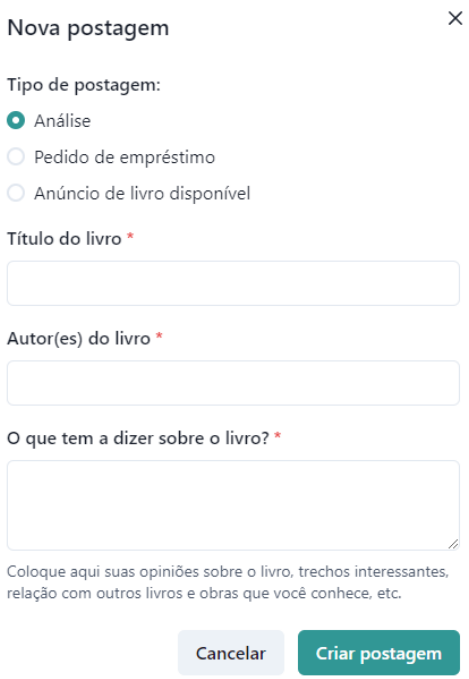

Figura 6.5: Captura de tela da modal de criação de nova postagem.

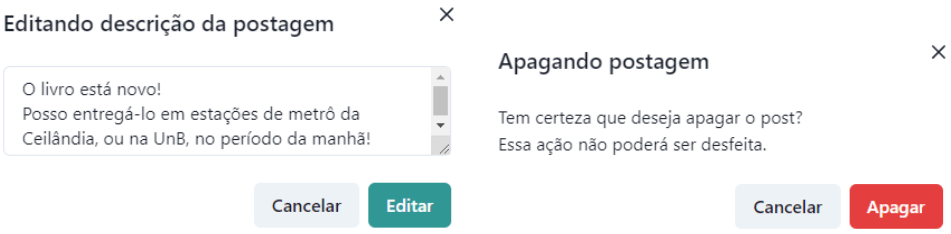

Figura 6.6: Captura de tela das modais de edição e exclusão de postagem.

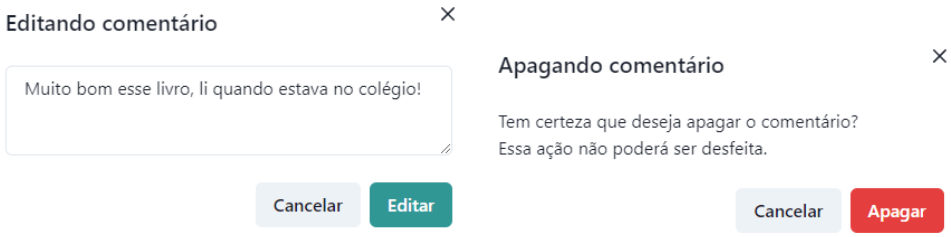

Figura 6.7: Captura de tela das modais de edição e exclusão de comentário.

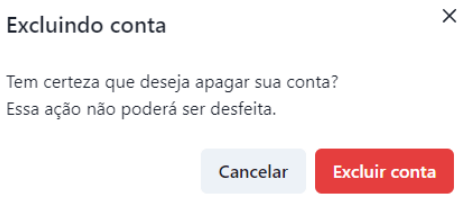

Figura 6.8: Captura de tela da modal de exclusão de conta.

Além disso, algumas informações são exibidas em forma de menus suspensos (menus *drop-down*), que exibem opções em lista, quando pressionados. Como é o caso das funcionalidades a seguir, apresentadas na figura 6.9:

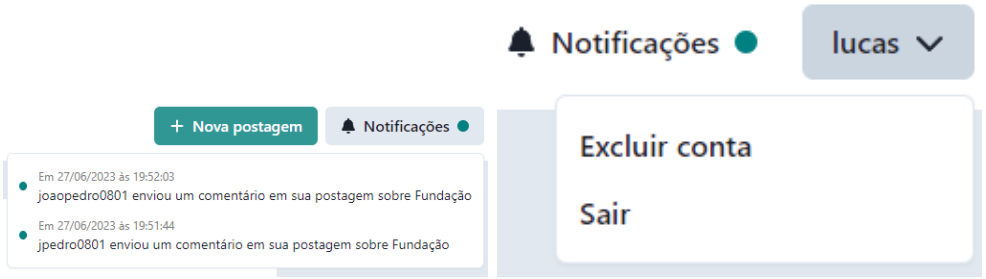

Figura 6.9: À esquerda: menu de gerenciamento de conta, por meio do qual é possível acessar as funcionalidades de *log out* e exclusão de conta. À direita, menu de notificações recebidas, através do qual o usuário é notificado a respeito de interações de usuários em suas postagens.

Por fim, um tratamento de erros foi feito, principalmente em áreas em que o usuário deve informar valores textuais. Mensagens de erro informativas são exibidas quando o usuário deixa campos obrigatórios vazios, por exemplo.

6.2.4.1 Elementos do projeto do banco de dados com *Firebase*

Como solução de banco de dados, é provido, pela plataforma *Firebase*, o *Firestore*, que é um sistema de banco de dados em nuvem, *NoSQL* e orientado a documentos.

O *Firestore* permite a criação de coleções, dentro das quais podem ser adicionados documentos, estruturas de dados que se assemelham a objetos *JSON*. Documentos são identificados por *strings* identificadoras únicas e dentro de documentos podem ser criadas novas coleções.

Para o desenvolvimento da aplicação, foram criadas as seguintes coleções:

- ***users***: coleção que armazena informações adicionais de cada usuário, em especial: ***email*** e ***username***. A autenticação é feita utilizando o módulo de autenticação do

Firebase, logo, os dados dessa coleção não são utilizados para autenticação, e sim, para exibição dos dados dos usuários autores de cada postagem. Além disso, para cada *user* é criada a seguinte subcoleção:

  - ***notifications***: coleção que armazena dados relevantes para o envio de cada notificação, como: ***createdAt***, data e hora de criação; ***postID***, identificador da postagem onde a notificação se originou; ***seen***, estado indicando se a notificação foi visualizada; ***text***, texto exibido na mensagem de notificação.

- ***posts***: coleção que armazena dados de cada postagem feita, em especial: ***bookAuthor***, nome do autor do livro; ***bookTitle***, título do livro; ***closed***, estado indicando se a postagem foi fechada; ***createdAt***, data e hora de criação; ***createdBy***, referência ao usuário que criou a postagem; ***numberOfComments***, número de comentários feitos na postagem; ***text***, texto de descrição ou análise do livro; ***type***, tipo de postagem.
- ***comments***: coleção que armazena dados a respeito de comentários adicionados em postagens, como: ***createdAt***, dia e hora de criação do comentário; ***createdBy***, referência ao usuário que fez o comentário; ***postedIn***, referência à postagem em que o comentário foi feito; ***text***, o conteúdo do comentário em si.

Capítulo 7

Elementos do processo de teste e comparação das ferramentas

Neste capítulo, os cenários de teste são criados seguindo processo proposto anteriormente, e depois, estes cenários são transformados em casos de teste e codificados para execução utilizando tanto o *Selenium* como o *Cypress*. Após a execução dos casos de teste utilizando ambos os *frameworks*, uma comparação é realizada, segundo critérios de avaliação também descritos anteriormente.

7.1 Aplicando elementos do processo de criação de cenários de teste

Para a criação dos cenários, são empregados elementos do método descrito por Ryser e Glinz (2000) [14], assim como exposto no Capítulo 5. No total, foram criados 13 cenários, que estão listados a seguir:

1. Usuário realiza cadastro.
2. Usuário realiza *login*.
3. Usuário acessa postagens.
4. Usuário realiza busca.
5. Usuário cria nova postagem.
6. Usuário edita postagem.
7. Usuário exclui postagem.
8. Usuário comenta em postagem.

9. Usuário edita comentário.
10. Usuário exclui comentário.
11. Sistema envia notificações.
12. Usuário realiza *logout*.
13. Usuário exclui sua conta.

No Apêndice A os cenários listados são apresentados de forma mais detalhada, representados de acordo com o *template* para cenários especificado anteriormente.

7.2 Codificação e execução dos casos de teste

Os cenários criados, e em especial, cada um dos fluxos, tanto normais (de sucesso), como alternativos (de falha), podem ser transformados em casos de teste individuais e de forma relativamente simples, já que as ações realizadas nos fluxos já estão descritas e dispostas em forma de lista, e entradas (pré-condições) e saídas esperadas (pós-condições) já foram especificadas.

7.2.1 Elementos da codificação com *Cypress*

O *framework Cypress* suporta codificação dos *scripts* de teste utilizando *Javascript* e *Typescript*, que foi utilizado nesse projeto. O *Typescript* adiciona tipagem estática opcional ao *Javascript*, o que pode melhorar a experiência de desenvolvimento, com melhores sugestões apresentadas por editores de texto, por exemplo. Mesmo assim, recursos do *Typescript* foram pouco utilizados, com uso mais notável na tipagem dos parâmetros de comandos personalizados, que serão apresentados a seguir, e dessa forma, o código-fonte produzido é quase idêntico ao que seria produzido caso se utilizasse *Javascript.*

Ao codificar os *scripts* de teste, cada cenário foi implementado em um arquivo, e no arquivo, foi implementado um caso de teste para cada fluxo, ou variações de fluxo, do cenário correspondente. A figura 7.1 apresenta a forma com que os arquivos resultantes foram organizados.

Após a instalação e execução inicial, o *Cypress* cria uma estrutura de arquivos inicial, que foi seguida na implementação dos casos de teste. Para organização dos *scripts* de teste, o *framework* disponibiliza funções como ***beforeEach***, que permite executar comandos antes de cada caso de teste em um arquivo; ***before***, que permite executar comandos antes de todos os casos de teste em um arquivo; ***describe***, que permite agrupar casos de teste, e informar uma *string* de descrição, aqui utilizada para agrupar os casos de teste de cada cenário; ***specify***, que permite criar casos de teste, e também informar *string* de descrição.

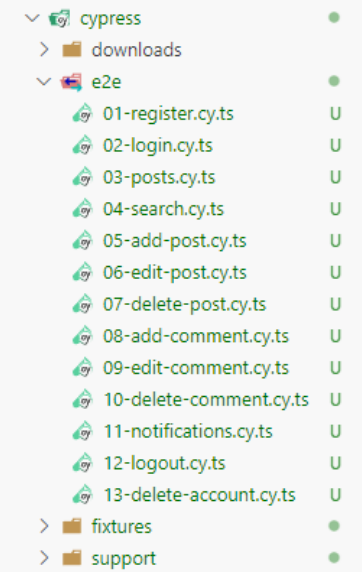

Figura 7.1: Captura de tela com a estrutura dos arquivos de teste implementados com *Cypress*.

A localização de elementos *HTML* com *Cypress* foi feita utilizando as funções ***cy.get*** e ***cy.contains***. A função ***cy.get*** é capaz de localizar elementos por meio de atributos, como *id*, *role*, *type* ou qualquer outro atributo. A função ***cy.contains***, por sua vez, localiza elementos por meio de seu conteúdo textual. Além disso, é possível realizar asserções utilizando a função ***cy.should***, e interações com os elementos localizados utilizando ***cy.click***, para clicar em elementos, ***cy.type***, para digitar em campos de entrada de texto, ***cy.check*** para marcar entradas do tipo *radio button*, entre outras.

A figura 7.2 exemplifica como um arquivo de *script* de teste do *Cypress* foi estruturado, e como as funções de organização, bem como as funções de localização de elementos, de interação com elementos, e de asserção são utilizadas. No caso do *script* apresentado, é possível observar que a função ***describe*** foi utilizada para agrupar todos os fluxos e testes referentes ao Cenário 02, atribuindo uma descrição que identifica esse cenário; as funções ***specify*** foram utilizadas para separar cada um dos fluxos, o normal e os 3 alternativos; e a função ***beforeEach*** está realizando limpeza dos dados do navegador e retorno à página inicial da aplicação antes da execução de cada fluxo.

Por fim, o *Cypress* permite a criação de comandos personalizados para utilização durante os testes. Ações realizadas em múltiplos cenários foram transformadas em funções desse tipo, de forma a evitar repetições de código-fonte. As funções criadas são: ***login***, ***newPost***, ***newComment*** e ***logout***, que realizam, respectivamente, o *login* na plataforma, a criação de uma nova postagem, a criação de um novo comentário e a realização de *logout* da plataforma.

Os comandos personalizados criados foram utilizados principalmente dentro de funções ***before***, e ***beforeEach***, executados antes do cenário, e antes de cada fluxo do cenário, respectivamente. Isso pois, alguns dos cenários têm como pré-condições a execução de fluxos de outros cenários, como por exemplo, o cenário de *logout* tem como pré-condição estar logado, o que é realizado no cenário de *login*. A implementação dos comandos pode ser observada por meio da figura 7.3.

```
describe("Cenário 02 - Usuário realiza login", () => {
  beforeEach(() => {
    indexedDB.deleteDatabase("firebaseLocalStorageDb");
    localStorage.clear();
    cy.visit("localhost:3000");
  });

  specify("Fluxo normal", () => {
    cy.contains("Entrar").click();

    cy.get("input[type=email]").type("pedro@gmail.com");
    cy.get("input[type=password]").type("12345678");
    cy.get("button[type=submit]").click();
    cy.get("span").contains("pedro").should("be.visible");
  });

  specify("Fluxo alternativo 1.1 - Email inválido", () => {
    cy.contains("Entrar").click();

    cy.get("input[type=email]").type("pedro");
    cy.get("input[type=password]").type("12345678");
    cy.get("button[type=submit]").click();
    cy.get("p").contains("O e-mail informado é inválido").should("be.visible");
  });

  specify("Fluxo alternativo 1.2 - Email não cadastrado", () => {
    cy.contains("Entrar").click();

    cy.get("input[type=email]").type("pedro@gmail.br");
    cy.get("input[type=password]").type("12345678");
    cy.get("button[type=submit]").click();
    cy.get("p")
      .contains("Usuário inexistente ou senha incorreta")
      .should("be.visible");
  });

  specify("Fluxo alternativo 2 - Senha inválida", () => {
    cy.contains("Entrar").click();

    cy.get("input[type=email]").type("pedro@gmail.com");
    cy.get("input[type=password]").type("1234");
    cy.get("button[type=submit]").click();
    cy.get("p")
      .contains("Usuário inexistente ou senha incorreta")
      .should("be.visible");
  });
});
```

Figura 7.2: Captura de tela do arquivo *02-login.cy.ts*, contendo os casos de teste referentes ao cenário 2, implementados com *Cypress*. É possível observar as funções de organização, e os comandos para localização de elementos *HTML*.

```
Cypress.Commands.add("login", (email: string, pwd: string) => {
  cy.contains("Entrar").click();
  cy.get("input[type=email]").type(email);
  cy.get("input[type=password]").type(pwd);
  cy.get("button[type=submit]").click();
});

Cypress.Commands.add(
  "newPost",
  (title: string, author: string, text: string) => {
    cy.contains("Nova postagem").click();
    cy.get("input[value=review]").check({ force: true });
    cy.get("input[name=bookTitle]").type(title);
    cy.get("input[name=bookAuthor]").type(author);
    cy.get("textarea[name=text]").type(text);
    cy.contains("Criar postagem").click();
  }
);

Cypress.Commands.add("newComment", (comment: string) => {
  cy.get("[id=post-card]")
    .first()
    .within(() => {
      cy.contains("Ver mais").click();
    });

  cy.get("textarea[name=commentText]").type(comment);
  cy.contains("button", "Comentar").click();
});

Cypress.Commands.add("logout", () => {
  cy.get("[id=menu-button-manage-account]").click();
  cy.contains("button[role=menuitem]", "Sair").click();
});
```

Figura 7.3: Captura de tela com os comandos personalizados adicionados, implementados com *Cypress*.

7.2.2 Elementos da codificação com Selenium

A utilização do *Selenium* é suportada em *C#*, *Ruby*, *Java*, *Javascript* e *Python* [29]. Aqui, a fim de tentar manter a experiência de desenvolvimento similar a que se tem com *Cypress*, e facilitar a comparação, os casos de teste foram desenvolvidos em *Javascript*.

O portal na *web* acerca da ferramenta [29] recomenda a utilização de um *test runner*, para organizar e executar os casos de teste desenvolvidos com comandos do *Selenium Webdriver*. Para a linguagem *Javascript*, o portal provê exemplo utilizando o *test runner Mocha* [38], que coincidentemente, é uma das bibliotecas utilizadas pelo *Cypress* para a organização dos testes, e essa foi a biblioteca utilizada como *test runner* nesse trabalho.

Dessa forma, nos casos de teste desenvolvidos com *Selenium*, serão encontradas funções de organização semelhantes às encontradas nos *scripts Cypress*, entre elas: ***describe***, para agrupar casos de teste; ***it***, para especificar casos de teste; ***before***, para executar comandos antes de todos testes de um arquivo; ***beforeEach***, para executar comandos

antes de cada caso de teste; ***after***, para executar comandos após todos os testes de um arquivo; ***afterEach***, para executar comandos após cada caso de teste.

Para a localização dos elementos *HTML*, foi utilizada a função ***findElement***, provida pelo próprio *framework*, que permite a localização de elementos por meio de diversos localizadores. Para o desenvolvimento dos testes foram utilizados os localizadores ***By.id***, para localizar por meio do atributo *id*, ***By.name***, para localizar por meio do atributo *name* e ***By.xpath***, para localizar elementos utilizando *XPath*, uma forma mais geral e flexível de localizar elementos. Além disso, a interação com os elementos localizados foi realizada por meio de funções como ***click***, para clicar em elementos, e ***sendKeys*** para digitar em campos de entrada. Asserções puderam ser realizadas utilizando a função ***assert*** do próprio *Javascript*.

Buscou-se seguir uma estruturação de arquivos, e estruturação dos testes em cada arquivo, de forma similar ao que foi desenvolvido com *Cypress*. A figura 7.4 apresenta a forma com que os arquivos de teste resultantes foram organizados.

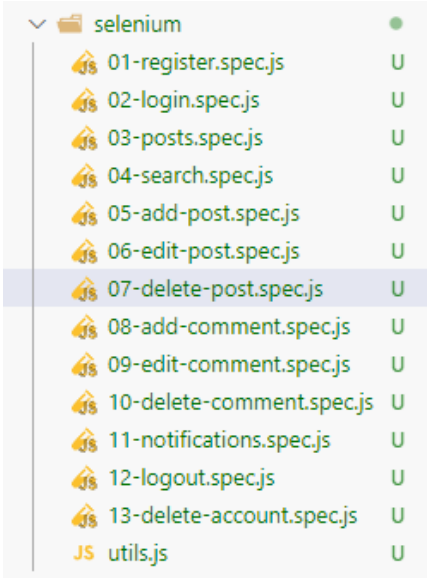

Figura 7.4: Captura de tela com a estrutura dos arquivos de teste implementados com *Selenium*.

A figura 7.5 exemplifica como um arquivo de *script* de teste com *Selenium* foi estruturado, sendo possível observar algumas funções semelhantes às ocorrentes anteriormente. No caso do *script* apresentado, a função ***describe*** agrupa os fluxos do Cenário 07; a função ***it*** foi utilizada para separar o único fluxo do cenário; na função ***before***, uma instância do *ChromeDriver* é criada, sua janela é maximizada, visita-se a página inicial da aplicação, e executam-se as funções personalizadas de ***login*** e de ***newPost***, que serão melhor descritas a seguir; a função ***after*** está limpando os dados do navegador, para que

o próximo cenário possa ser executado em um ambiente limpo; e a função ***beforeEach*** volta à página inicial antes da execução do fluxo. Além disso, é possível observar o uso da função de localização de elementos ***findElement***, das estratégias de localização ***By.id*** e ***By.xpath***, e da função de interação ***click***.

```
suite(function () {
  describe("Cenário 07 - Usuário exclui postagem", async function () {
    let driver;

    before(async function () {
      driver = await new Builder().forBrowser("chrome").build();
      await driver.manage().setTimeouts({ implicit: 5000 });
      await driver.manage().window().maximize();
      await driver.get("http://localhost:3000");
      await login(driver, "pedro@gmail.com", "12345678");
      await newPost(
        driver,
        "Livro teste",
        "Autor teste",
        "Testando a aplicação"
      );
    });

    beforeEach(() => {
      driver.get("http://localhost:3000");
    });

    after(async function () {
      driver.executeScript(
        'window.indexedDB.deleteDatabase("firebaseLocalStorageDb")'
      );
      await driver.quit();
    });

    it("Fluxo normal", async function () {
      let see_more_btn = await driver.findElement(
        By.xpath('(//*[contains(text(), "Ver mais")])[1]')
      );
      await see_more_btn.click();

      let delete_post_btn = await driver.findElement(By.id("delete-post-btn"));
      await delete_post_btn.click();

      let confirm_btn = await driver.findElement(
        By.id("post-op-confirmation-button")
      );
      await confirm_btn.click();

      return await driver.findElement(
        By.xpath('//*[contains(text(), "O seu post foi excluído!")]')
      );
    });
  });
});
```

Figura 7.5: Captura de tela do arquivo *07-delete-post.spec.js*, contendo os casos de teste referentes ao cenário 7, implementados com *Selenium*.

Por fim, durante os testes desenvolvidos com *Selenium* também foram criadas funções para evitar repetição de alguns fluxos. As funções desenvolvidas são similares aos comandos personalizados desenvolvidos durante a implementação de testes com *Cypress*, sendo elas ***login***, ***newPost***, ***newComment*** e ***logout***. A implementação dessas funções pode ser observada por meio das figuras 7.6, 7.7, 7.8 e 7.9.

```
async function login(driver, user, pwd) {
  let enter_btn = await driver.findElement(
    By.xpath('//*[contains(text(), "Entrar")]')
  );
  await enter_btn.click();

  let email_input = await driver.findElement(By.name("email"));
  await email_input.sendKeys(user);

  let password_input = await driver.findElement(By.name("password"));
  await password_input.sendKeys(pwd);

  let submit_btn = await driver.findElement(
    By.xpath('//button[@type="submit"]')
  );
  await submit_btn.click();

  return await driver.findElement(
    By.xpath(`//span[contains(text(), "${user.split("@")[0]}")]`)
  );
}
```

Figura 7.6: Captura de tela com a função *login*, implementada com *Selenium*.

```
async function newPost(driver, title, author, text) {
  let new_post_btn = await driver.findElement(
    By.xpath('//*[contains(text(), "Nova postagem")]')
  );
  await new_post_btn.click();

  let review_option = await driver.findElement(
    By.xpath('//input[@value="review"]/parent::label')
  );
  await review_option.click();

  let title_input = await driver.findElement(By.name("bookTitle"));
  await title_input.sendKeys(title);

  let author_input = await driver.findElement(By.name("bookAuthor"));
  await author_input.sendKeys(author);

  let description_input = await driver.findElement(By.name("text"));
  await description_input.sendKeys(text);

  let submit_btn = await driver.findElement(
    By.xpath("//*[contains(text(), 'Criar postagem')]")
  );
  await submit_btn.click();

  let toast = await driver.findElement(
    By.xpath('//*[contains(text(), "Seu post foi criado")]')
  );

  return await toast.click();
}
```

Figura 7.7: Captura de tela com a função *newPost*, implementada com *Selenium*.

```
async function newComment(driver, commentText) {
  let see_more_btn = await driver.findElement(
    By.xpath('(//*[contains(text(), "Ver mais")])[1]')
  );
  await see_more_btn.click();

  let textarea = await driver.findElement(
    By.xpath('//textarea[@name="commentText"]')
  );
  await textarea.sendKeys(commentText);

  let comment_btn = await driver.findElement(
    By.xpath('//*[contains(text(), "Comentar")]')
  );
  await comment_btn.click();

  return await driver.findElement(
    By.xpath(`(//p[contains(text(), "${commentText}")])[1]`)
  );
}
```

Figura 7.8: Captura de tela com a função *newComment*, implementada com *Selenium*.

```
async function logout(driver) {
  let account_btn = await driver.findElement(
    By.id("menu-button-manage-account")
  );
  await account_btn.click();

  let logout_btn = await driver.findElement(
    By.xpath('(//button[@role="menuitem"])[contains(text(), "Sair")]')
  );
  await logout_btn.click();

  return await driver.findElement(By.xpath('//*[contains(text(), "Entrar")]'));
}
```

Figura 7.9: Captura de tela com a função *logout*, implementada com *Selenium*.

7.3 Comparação das ferramentas

Após a implementação e execução dos cenários de teste, a comparação será realizada, segundo critérios reunidos anteriormente. Será utilizada uma mescla de critérios utilizados por Gamido e Gamido (2019), [32] Singh e Tarika (2014) [33], e alguns dos critérios especificados por Illes et al. (2005) [31]:

Critério: **A ferramenta oferece suporte a múltiplas plataformas**:
Para avaliação segundo esse critério, os portais na *web* acerca das ferramentas foram acessados, e buscou-se informações a respeito do suporte oficial a diferentes sistemas operacionais, já que, para o desenvolvimento e execução dos testes nesse trabalho, somente foi utilizado o sistema operacional *Windows*.

- **Cypress**: É suportado nos sistemas operacionais *Linux*, *Windows*, e *Mac* [30].
- **Selenium**: É suportado nos sistemas operacionais *Linux*, *Windows*, e *Mac* [29].

* * *

Critério: **A ferramenta oferece suporte a múltiplos navegadores**:
Para avaliação segundo esse critério, os portais na *web* acerca das ferramentas foram acessados, e buscou-se informações a respeito do suporte oficial a diferentes navegadores *web*, já que, no contexto desse projeto, todos os testes foram executados no navegador *Chrome*.

- **Cypress**: Suporta versões dos navegadores *Chrome*, *Edge*, *Firefox*, *Chromium*, *Electron* e *WebKit* (Experimental) [30].
- **Selenium**: Suporta os navegadores *Chrome*, *Edge*, *Firefox*, *Internet Explorer* e *Safari* [29].

* * *

Critério: **A ferramenta oferece suporte a múltiplas linguagens de programação**:
Para avaliação segundo esse critério, os portais na *web* acerca das ferramentas foram acessados, e buscou-se informações a respeito do suporte oficial a diferentes linguagens de programção.

- **Cypress**: Suporta oficialmente somente *Javascript* e *Typescript* [30].
- **Selenium**: Suporta oficialmente *C#*, *Ruby*, *Java*, *Python* e *Javascript* [29].

* * *

Critério: **A ferramenta oferece suporte a *Record Playback* e geração de *scripts***:
Para avaliação segundo esse critério, os portais na *web* das ferramentas são acessados, a fim de se confirmar a existência de suporte oficial a geração de *scripts* através da abordagem *Record Playback*, por ambas ferramentas.

- **Cypress**: Provê suporte a gravação de interações e geração de *scripts* através do *Cypress Studio*, um recurso em fase experimental, no momento da realização dessa comparação. [30].
- **Selenium**: Provê suporte a gravação de interações e geração de *scripts* através da extensão *Selenium IDE*, suportada nos navegadores *Chrome* (e derivados) e *Firefox* [29].

* * *

Critério: **A ferramenta produz reportes e saídas bem formatadas**:
Para avaliação segundo esse critério, observam-se as saídas produzidas após a execução de um mesmo cenário, implementado com ambos *frameworks*, a fim de avaliar a clareza, formatação e quais informações são apresentadas. Além disso, os portais na *web* das ferramentas são acessados, com o intuito de confirmar a possibilidade de gerar e customizar reportes.

- **Cypress**: Apresenta interface gráfica, por meio da qual é possível observar a execução dos testes, é possível observar após a finalização, cada etapa realizada em um fluxo; além disso, os resultados são apresentados, por padrão, em interface gráfica de fácil visualização, conforme apresenta a figura 7.10. Além disso, por utilizar o *Mocha*, é possível utilizar ferramentas de reporte customizáveis, feitas para utilização com *Mocha* [30].
- **Selenium**: Segundo o portal acerca da ferramenta [29], o Selenium não é construído para reportar resultados de execuções de testes, dessa forma, recomenda-se a utilização conjunta das capacidades de *frameworks* para execução de testes e reporte de resultados. No âmbito deste projeto, o *Mocha* foi utilizado e o mesmo apresenta, por padrão, os resultados dos testes executados na própria linha de comando, conforme mostra a figura 7.11. Além disso, similarmente ao que ocorre com a outra ferramenta, é possível a instalação de ferramentas de geração de reporte, para utilização com o *Mocha*.

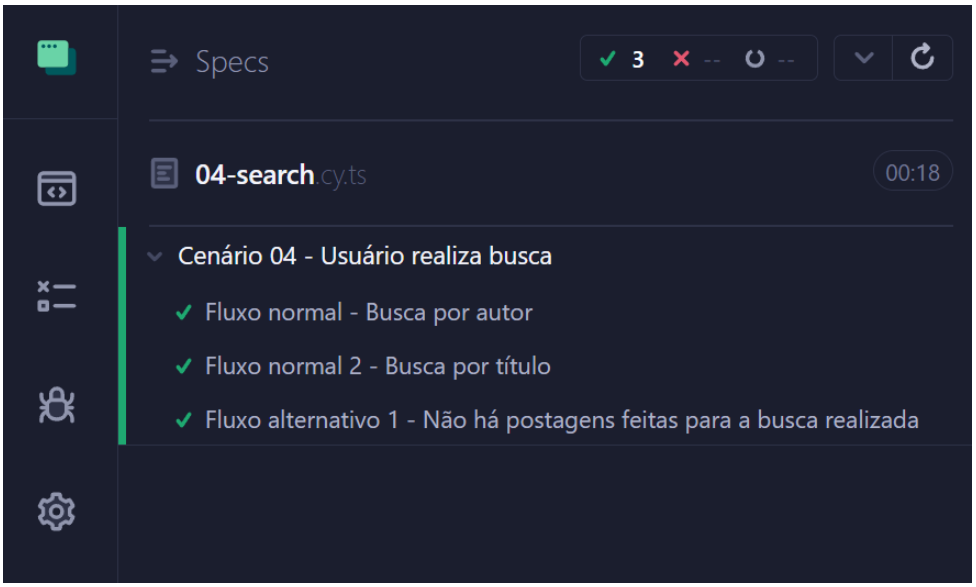

Figura 7.10: Captura de tela com os resultados apresentados pelo *Cypress* após a execução do *script* de teste implementado para o cenário 4.

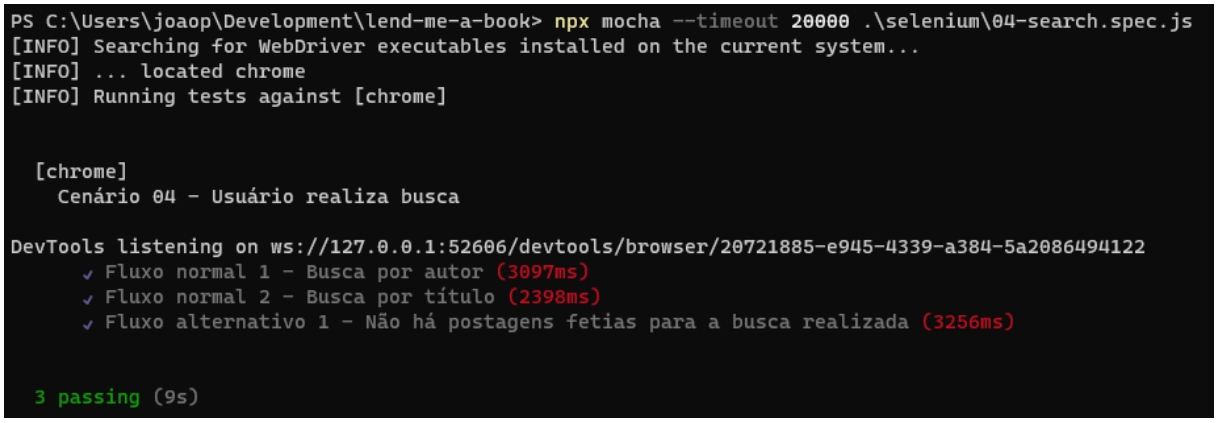

Figura 7.11: Captura de tela com os resultados apresentados pelo *Selenium* após a execução do *script* de teste implementado para o cenário 4.

* * *

Critério: **A ferramenta oferece suporte a vários tipos de teste**:
Para avaliação segundo esse critério, os portais e documentação na *web* das ferramentas são acessadas, para que seja possível aferir quais abordagens e tipos de testes são listadas como oficialmente suportadas por cada ferramenta.

- **Cypress**: Suporta, de acordo com portal acerca da ferramenta [30] e de acordo com a interface exibida ao executar a ferramenta, testes *end-to-end*, utilizados, segundo o portal, para simular interações do usuário com a aplicação, testando fluxos completos e a aplicação por inteiro; e testes de *componentes*, utilizados, segundo o portal, uma forma de testar componentes criados utilizando *frameworks* modernos para desenvolvimento *web*, e são testes especializados, que podem ser executados mais rapidamente e são mais facilmente configuráveis quando comparados à outra modalidade de teste suportada.
- **Selenium**: Suporta, de acordo com portal acerca da ferramenta [29], testes de aceitação, testes funcionais, testes de performance, testes de regressão, e abordagens como o Desenvolvimento guiado por testes (*Test driven development*), e o Desenvolvimento guiado por comportamento (*Behaviour driven development*).

* * *

Critério: **A ferramenta permite configuração de ambiente de teste, pré-condições e pós condições para um conjunto de casos de teste**:
Para avaliação segundo esse critério, observa-se, durante o desenvolvimento dos *scripts* de teste, se os *frameworks* oferecem suporte a funções para execução de comandos em

momentos definidos dos testes, como antes ou depois dos *scripts* de teste, ou antes e depois de cada fluxo de teste, a fim de configurar pré e pós condições, por exemplo.

- **Cypress**: Por padrão, a ferramenta provê funções para configuração de comandos a serem executados antes e/ou após *script*, e antes e/ou após cada caso de teste.
- **Selenium**: A ferramenta recomenda a utilização de *test runners* para organizar a execução dos testes. Este projeto utilizou o *Mocha*, que é o mesmo utilizado pelo *Cypress*, e portanto, tem-se capacidade semelhante de configuração de comandos e de ambiente de testes ao que é observado com *Cypress*.

* * *

Critério: **Tempo de execução de testes com a ferramenta**:
Para avaliação segundo esse critério, executou-se sequencialmente os *scripts* de teste desenvolvidos com *Cypress*, e então, executou-se os *scripts* desenvolvidos com *Selenium*, e assim sucessivamente, até completar duas execuções completas para cada ferramenta, de forma a ser possível calcular uma média entre os tempos de execução observados. Os *scripts* foram executados separadamente, a fim de facilitar a coleta de tempos de execução para cada cenário individual. Os testes foram executados em um mesmo ambiente e as configurações padrão das ferramentas foram utilizadas. Por fim, os tempos coletados foram os medidos e apresentados pelas próprias ferramentas ao fim da execução de cada *script*.

- **Cypress**: O tempo médio de execução pode ser consultado, ao fim da execução dos *scripts*, conforme mostra a figura 7.10. Os tempos médios observados para cada *script* de cenário executado com *Cypress* foram coletados e organizados na tabela 7.1.
- **Selenium**: O tempo médio de execução pode ser consultado, ao fim da execução dos *scripts*, conforme mostra a figura 7.11. Os tempos médios observados para cada *script* de cenário executado com *Selenium* foram coletados, e organizados na tabela 7.1.

#	Cenário	Tempo médio com Cypress (s)	Tempo médio com Selenium (s)
1	Cadastro	14.5	8.5
2	*Login*	11.5	7.5
3	Postagens	5	3
4	Busca	15	11.5
5	Nova postagem	18.5	11.5
6	Editar postagem	13.5	9
7	Excluir postagem	10	8
8	Adicionar comentário	10	9
9	Editar comentário	13.5	10.5
10	Excluir comentário	9.5	7.5
11	Notificações	18	11.5
12	*Logout*	4	3
13	Excluir conta	6	4.5

Tabela 7.1: Tempos médios de execução dos testes com *Cypress* e *Selenium*, em segundos.

* * *

Critério (subjetivo): **A ferramenta é de fácil utilização, e não exige habilidades avançadas de programação:**
Para avaliação segundo esse critério, por ser um critério subjetivo, considerou-se a experiência de desenvolvimento geral do ponto de vista do autor do trabalho, utilizando ambas ferramentas. Foram considerados, entre outros fatores, a facilidade de configuração da ferramenta, e a clareza da sintaxe do código-fonte produzido.

- **Cypress**: A ferramenta propõe uma sintaxe com comandos de simples utilização e o resultado são *scripts* curtos e de fácil entendimento, com maior proximidade à linguagem natural. Além disso, a instalação é feita com apenas um comando e guiada por interface gráfica, o que facilita o processo. A execução dos casos de teste também pode ser feita, por padrão, por meio de interface gráfica.
- **Selenium**: Se comparada ao *Cypress*, a instalação e utilização do *Selenium* não é tão simplificada, já que é necessário realizar alguns passos manuais, como a instalação do *WebDriver* a ser utilizado, e de um *test runner* para melhor estruturar os testes. Além disso, as funções disponibilizadas pelo *Selenium* para utilização com *Javascript*, retornam *Promises*, e dessa forma, é necessário encadeá-las com *callbacks .then()*, ou envolvê-las em blocos *async/await*, o que torna o código produzido

mais verboso e extenso. Apesar disso, as funções disponibilizadas são nomeadas de forma clara, o que torna simples entender qual ação cada função realiza; e por ser uma ferramenta presente na indústria há anos, é fácil encontrar ajuda em fóruns, por exemplo.

Considerações finais

Capítulo 8

Considerações finais

Este capítulo apresenta alcance dos objetivos, limitações do trabalho e sugestões para trabalhos futuros.

8.1 Alcance dos objetivos

O trabalho teve como objetivo geral a comparação dos *frameworks* para automação de testes *Cypress* e *Selenium*, no contexto de aplicações *web* e utilizando uma abordagem embasada em cenários; e como objetivos específicos, o desenvolvimento de parte de aplicação *web*, criação de cenários, desenvolvimento de casos de teste a partir dos cenários criados, e por fim, a comparação de ferramentas.

O trabalho passou, primeiramente, por capítulos conceituais, para definir termos e conceitos importantes para a compreensão do contexto em que o projeto se insere, e para a compreensão das atividades realizadas durante a parte prática. Posteriormente, foram descritos processos e métodos utilizados na parte prática. E por fim, a parte prática foi realizada, com o desenvolvimento de parte de uma aplicação *web*, realização de testes da aplicação, e comparação das ferramentas de teste.

Os objetivos propostos foram atingidos, já que as ferramentas foram comparadas segundo critérios definidos, testes foram desenvolvidos segundo um processo descrito, e parte de um sistema foi desenvolvido segundo a abordagem proposta.

8.2 Contribuições do trabalho

A abordagem baseada em cenários para o desenvolvimento de testes permitiu melhor entendimento dos requisitos, e dos fluxos potencialmente realizados na plataforma desenvolvida. Essa abordagem, em conjunto com outras abordagens de teste, pode ser de valor em projetos reais, dada a proximidade dos cenários testados com os requisitos produzidos

em conjunto com usuários; além de que o próprio processo de criação de cenários utilizado recomenda a participação de partes interessadas para correção e melhoria dos cenários.

O desenvolvimento dos testes, e a comparação das ferramentas, mostrou que, apesar de serem capazes de cumprir funções semelhantes, cada ferramenta possui suas particularidades. O *Cypress* facilita o processo de desenvolvimento de testes, simplificando processo de instalação, provendo sintaxe simplificada, e disponibilizando as funcionalidades mais comumente utilizadas quando testando interfaces *web*; além de prover suporte a *frameworks* populares para desenvolvimento *web*, como *React*, *Angular*, *Svelte* e *Vue*. Já o *Selenium*, é uma ferramenta de automação geral do navegador, sendo que para a execução e organização de testes, o recomendado é que se utilizem *test runners*; o *framework* possui funcionalidades adicionais que podem ser úteis, em especial, em sistemas de grande porte, como a possibilidade de manusear múltiplas janelas/abas do navegador, e o *Selenium Grid*, que permite a execução do *Selenium WebDriver* em máquinas remotas.

8.3 Limitações do trabalho

O trabalho é limitado, principalmente, pelo pequeno porte da aplicação desenvolvida. Uma aplicação de maior porte, como uma aplicação real, e em ambiente de produção, permitiria explorar ainda mais as capacidades das ferramentas de teste utilizadas, possibilitando, por exemplo, a criação de mais cenários, e cenários mais complexos, com maior quantidade de fluxos possíveis, além de fluxos mais longos.

Além disso, é limitado pela quantidade de ferramentas testadas, já que seria possível utilizar outras ferramentas que desempenham funções semelhantes às utilizadas neste trabalho, de forma a compará-las utilizando o mesmo processo e os mesmos critérios de comparação.

8.4 Sugestões para trabalhos futuros

Para trabalhos futuros, a primeira sugestão está relacionada a uma das limitações do trabalho: o porte da aplicação. Pode ser interessante buscar aplicar testes em aplicações já existentes, e de grande porte, como aplicações de *e-commerce* e redes sociais. Ou até mesmo, caso possível, desenvolver uma aplicação de maior porte, com usuários reais participando da elicitação de requisitos e da criação de cenários de teste, de forma a tornar o processo mais fiel ao que ocorreria em um ambiente comercial. Em uma aplicação de maior porte, pode-se explorar alguns dos recursos adicionais oferecidos pelos *frameworks* como o *Selenium Grid*.

Pode-se, também, utilizar critérios subjetivos para comparação, múltiplos desenvolvedores podem participar do processo de desenvolvimento de testes de uma aplicação, por exemplo, de forma que seja possível reunir opiniões, e dados de experiência por meio de formulários ou de outros meios.

Neste trabalho, não foram utilizadas as ferramentas para geração de *scripts* de teste (*Record and Playback*) providas por ambos *frameworks*, e pode ser interessante, utilizá-las e avaliá-las, de forma a enriquecer a comparação realizada.

Por fim, considerando-se a quantidade de ferramentas testadas, em trabalho futuro é possível, utilizando método semelhante ao seguido neste trabalho, desenvolver testes e comparar uma maior quantidade de ferramentas para automação de testes no contexto da *web*, já que várias estão atualmente disponíveis, mas somente duas foram selecionadas para comparação no trabalho cujos resultados são descritos neste documento.

Referências

[1] Juran, Joseph M. e Joseph A. De Feo: *Juran's Quality Hanbook: The Complete Guide to Performance Excellence.* McGraw-Hill, New York, USA, 6ª edição, 2010. 1, 3, 7

[2] Polo, Macario, Pedro Reales Mateo, Mario Piattini e Christof Ebert: *Test automation.* IEEE Software, 30:84–89, janeiro 2013. 1, 26

[3] Westfall, Linda: *The Certified Software Quality Engineer Handbook.* ASQ Quality Press, Wisconsin, USA, 2ª edição, 2016. 1, 7, 10, 11, 22, 25, 34

[4] Agarwal, Ravikant e David Umphress: *Extreme programming for a single person team.* Em *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, página 82–87, New York, NY, USA, 2008. Association for Computing Machinery, ISBN 9781605581057. https://doi.org/10.1145/1593105.1593127. 2, 32

[5] Bourque, Pierre e Richard E. Fairley: *Guide to the Software Engineering Body of Knowledge: SWEBOK.* IEEE Computer Society, USA, 3ª edição, 2014. 4, 5, 6, 8, 10, 15, 30, 31, 34

[6] Pressman, Roger S.: *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New York, USA, 7ª edição, 2010. 4, 5, 6, 7, 19, 20, 21, 31

[7] Sommerville, Ian: *Software Engineering.* Pearson, London, United Kingdom, 10ª edição, 2016. 5, 8, 10, 13, 19, 21, 22, 34

[8] Project Management Institute e IEEE Computer Society: *Software Extension to the PMBOK Guide.* Project Management Institute, Pennsylvania, USA, 5ª edição, 2013. 8, 9

[9] Project Management Institute: *Um Guia do Conhecimento em Gerenciamento de Projetos: Guia PMBOK.* Project Management Institute, Pennsylvania, USA, 6ª edição, 2017. 8

[10] *Iso/iec/ieee international standard - systems and software engineering – software life cycle processes.* ISO/IEC/IEEE 12207:2017(E) First edition 2017-11, páginas 1–157, 2017. 13, 14

[11] *Iso/iec/ieee international standard - software and systems engineering —software testing —part 2:test processes.* ISO/IEC/IEEE 29119-2:2013(E), páginas 1–68, 2013. 16, 17

[12] Sethi, Ravi: *Software Engineering*. University of Arizona, USA, 2017. 21

[13] Hamilton, Thomas: *What is test scenario in software testing (examples).*, 2022. https://www.guru99.com/test-scenario.html. 21

[14] Ryser, Johannes e Glinz Martin: *SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test*. University of Zurich, Zurich, Switzerland, 2000. 21, 35, 37, 38, 39, 57

[15] Kaner, Cem: *An Introduction to Scenario Testing*. Florida Institute of Technology, Florida, USA, 2013. 22

[16] SEVOCAB: *Software and systems engineering vocabulary*. https://pascal.computer.org/. 24

[17] *AUTOMATIZAÇÃO*. Em *Dicionário Priberam da Língua Portuguesa*. https://dicionario.priberam.org/automatiza%C3%A7%C3%A3o. 24

[18] Garousi, Vahid e Mika V. Mäntylä: *When and what to automate in software testing? a multi-vocal literature review*. Information and Software Technology, 76:92–117, 2016, ISSN 0950-5849. https://www.sciencedirect.com/science/article/pii/S0950584916300702. 24, 25

[19] Umar, Mubarak Albarka e Zhanfang Chen: *A study of automated software testing: Automation tools and frameworks*. International Journal of Computer Science Engineering, 8:217–225, dezembro 2019. 26

[20] Chemuturi, Murali: *Mastering Software Quality Assurance: Best Practices, Tools and Techniques for Software Developers*. J.Ross Publishing, USA, 2011. 27

[21] Edwards, Stephen: *A framework for practical, automated black-box testing of component-based software*. Software Testing, Verification and Reliability, 11, junho 2001. 28

[22] Tan, Roy Patrick, Pooja Nagpal e Shaun Miller: *Automated black box testing tool for a parallel programming library*. Em *2009 International Conference on Software Testing Verification and Validation*, páginas 307–316, 2009. 28

[23] Arcuri, Andrea: *Automated black- and white-box testing of restful apis with evomaster*. IEEE Software, 38(3):72–78, 2021. 28

[24] Lima, Bruno e João Pascoal Faria: *An approach for automated scenario-based testing of distributed and heterogeneous systems*. Em *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, volume 1, páginas 1–10, 2015. 28

[25] Campos, José C., Camille Fayollas, Célia Martinie, David Navarre, Philippe Palanque e Miguel Pinto: *Systematic automation of scenario-based testing of user interfaces*. Em *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '16, página 138–148, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 9781450343220. https://doi.org/10.1145/2933242.2948735. 29

[26] Rajora, Harish: *13 best test automation frameworks: The 2021 list*, 2021. https://www.lambdatest.com/blog/best-test-automation-frameworks-2021/. 40

[27] Badkar, Akshay: *Popular test automation frameworks*, 2023. https://www.browserstack.com/guide/best-test-automation-frameworks. 40

[28] Pelivani, Elis, Adrian Besimi e Betim Cico: *A comparative study of ui testing framework*. Em *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, páginas 1–5, 2022. 40

[29] Selenium: *Selenium*, 2023. https://www.selenium.dev/. 40, 61, 66, 67, 68

[30] Cypress: *Javascript Component Testing and E2E Testing Framework | Cypress*, 2023. https://www.cypress.io/. 41, 66, 67, 68

[31] Illes, Timea, Agnès Herrmann, Barbara Paech e Julius Rückert: *Criteria for software testing tool evaluation–a task oriented view*. Em *Proceedings of the 3rd World Congress for Software Quality*, volume 2, páginas 213–222, 2005. 42, 65

[32] Gamido, Heidilyn e Marlon Gamido: *Comparative review of the features of automated software testing tools*. International Journal of Electrical and Computer Engineering, 9:4473–4478, outubro 2019. 45, 65

[33] Singh, Inderjeet e Bindia Tarika: *Comparative analysis of open source automated software testing tools: Selenium, sikuli and watir*. junho 2014. 45, 65

[34] Vercel: *Next.js by Vercel - The React Framework*, 2023. https://nextjs.org/. 47

[35] Google: *Firebase*, 2023. https://firebase.google.com/. 47

[36] Chakra UI: *Chakra UI - A simple, modular and accessible component library that gives you the building blocks you need to build your React applications.*, 2023. https://chakra-ui.com/. 48

[37] CS Frequency: *React firebase hooks*, 2023. https://github.com/csfrequency/react-firebase-hooks. 48

[38] Mocha: *Mocha - the fun, simple, flexible JavaScript test framework*, 2023. https://mochajs.org/. 61

Apêndice A

Descrições de cenários

Cenário 01 - Usuário realiza cadastro

- **Descrição**: O usuário informa e-mail e senha válidos, e realiza cadastro para que possa acessar a plataforma.
- **Atores**: Novo usuário.
- **Pré-condições**: A plataforma é acessível.
- **Pós-condições**:
  - Sucesso: O usuário passa a ter acesso ao sistema. O usuário é autenticado no sistema.
  - Falha: O usuário recebe mensagem com descrição do erro.
- **Gatilho**: Usuário tenta se cadastrar.
- **Fluxo normal**:
  1. O usuário clica em "cadastrar-se" na barra de navegação.
  2. O usuário digita um e-mail válido.
  3. O usuário digita uma senha válida.
  4. O usuário clica no botão "cadastrar-se".
  5. O sistema autentica o usuário, e exibe seu nome de usuário na barra de navegação.
- **Fluxos alternativos**:
  1. O usuário informa e-mail inválido.
     (a) O usuário acessa o modal de cadastro.

(b) O usuário digita e-mail inválido/vazio ou que já está cadastrado.
(c) O usuário digita senha válida.
(d) O usuário clica no botão “cadastrar-se”.
(e) O sistema indica erro descritivo.

2. O usuário informa senha inválida.
   (a) O usuário acessa o modal de cadastro.
   (b) O usuário digita e-mail válido.
   (c) O usuário deixa a senha vazia ou com menos de 6 caracteres.
   (d) O usuário clica no botão “cadastrar-se”.
   (e) O sistema indica erro descritivo.

* * *

Cenário 02 - Usuário realiza *login*

- **Descrição**: O usuário informa e-mail e senha válidos, e realiza *login* utilizando credenciais cadastradas previamente.
- **Atores**: Usuário cadastrado.
- **Pré-condições**: Usuário deve estar cadastrado (Cenário 01).
- **Pós-condições**:
  - Sucesso: O usuário é autenticado no sistema.
  - Falha: O usuário recebe mensagem com descrição do erro.
- **Gatilho**: Usuário tenta realizar *login*.
- **Fluxo normal**:
  1. O usuário clica em “entrar” na barra de navegação.
  2. O usuário digita um e-mail válido.
  3. O usuário digita uma senha válida.
  4. O usuário clica no botão “entrar”.
  5. O sistema autentica o usuário, e exibe seu nome de usuário na barra de navegação.
- **Fluxos alternativos**:
  1. O usuário informa e-mail inválido.

(a) O usuário acessa o modal de *login*.
(b) O usuário digita e-mail inválido/vazio ou que não está cadastrado.
(c) O usuário digita senha válida.
(d) O usuário clica no botão “entrar”.
(e) O sistema indica erro descritivo.

2. O usuário informa senha inválida/incorreta.
   (a) O usuário acessa o modal de *login*.
   (b) O usuário digita e-mail válido.
   (c) O usuário deixa a senha vazia, com menos de 6 caracteres, ou incorreta.
   (d) O usuário clica no botão “entrar”.
   (e) O sistema indica erro descritivo.

* * *

Cenário 03 - Usuário acessa postagens

- **Descrição**: O usuário é capaz de acessar a lista de postagens.
- **Atores**: Usuário autenticado.
- **Pré-condições**: Usuário deve estar autenticado (Cenário 02).
- **Pós-condições**:
  - Sucesso: O sistema carrega e exibe as postagens feitas na plataforma.
  - Falha: O sistema exibe mensagem orientando o usuário a fazer novas postagens ou mudar os critérios de busca.
- **Gatilho**: Usuário acessa a página inicial.
- **Fluxo normal**:
  1. O usuário acessa a página inicial.
  2. O sistema exibe uma listagem com todas as postagens feitas na plataforma, ordenadas segundo data e hora.
- **Fluxos alternativos**: Não há.

* * *

Cenário 04 - Usuário realiza busca

- **Descrição**: O usuário é capaz de realizar buscas.

- **Atores**: Usuário autenticado.
- **Pré-condições**: Usuário deve estar autenticado (Cenário 02).
- **Pós-condições**:
  - Sucesso: O sistema carrega e exibe as postagens feitas na plataforma segundo o critério buscado.
  - Falha: O sistema exibe mensagem orientando o usuário a fazer novas postagens ou mudar os critérios de busca.
- **Gatilho**: Usuário realiza busca através da caixa de busca.
- **Fluxo normal**:
  1. O usuário acessa a página inicial.
  2. O usuário seleciona o critério de busca (autor ou título).
  3. O usuário digita a *string* a ser buscada.
  4. O usuário clica no ícone de busca.
  5. O sistema exibe uma lista de postagens feitas na plataforma, ordenadas segundo data e hora, e filtradas de acordo com os critérios de busca.
- **Fluxos alternativos**:
  1. Não há postagens feitas para a busca realizada.
     (a) O usuário acessa a página inicial.
     (b) O sistema não exibe lista de postagens.
     (c) O usuário seleciona o critério de busca (autor ou título).
     (d) O usuário digita a *string* a ser buscada.
     (e) O usuário clica no ícone de busca.
     (f) O sistema exibe mensagem orientando o usuário a fazer novas postagens ou mudar os critérios de busca.

* * *

Cenário 05 - Usuário cria nova postagem

- **Descrição**: O usuário é capaz de criar novas postagens.
- **Atores**: Usuário autenticado.
- **Pré-condições**: Usuário deve estar autenticado (Cenário 02).

- **Pós-condições**:
  - Sucesso: O sistema recarrega e mostra a nova postagem, no início da lista.
  - Falha: O sistema exibe mensagem de erro informativa.
- **Gatilho**: Usuário tenta adicionar nova postagem através do botão de "nova postagem".
- **Fluxo normal**:
  1. O usuário clica em "nova postagem".
  2. O sistema exibe o modal de criação de nova postagem.
  3. O usuário escolhe entre um dos tipos de postagem.
  4. O usuário digita um título válido de livro.
  5. O usuário digita um autor válido.
  6. O usuário informa uma descrição ou sua análise, a depender do tipo de postagem escolhida.
  7. O usuário clica em "criar postagem".
  8. O sistema fecha o modal.
  9. O sistema navega para a tela inicial, caso já não esteja.
  10. O sistema atualiza a lista de postagens, com a nova postagem no início.
- **Fluxos alternativos**:
  1. O usuário deixa um dos campos vazio.
     (a) O usuário clica em "nova postagem".
     (b) O sistema exibe o modal de criação de nova postagem.
     (c) O usuário escolhe entre um dos tipos de postagem.
     (d) O usuário deixa o título, ou autor, ou descrição/análise vazios, ou todos.
     (e) O sistema exibe erro descritivo, informando sobre a obrigatoriedade dos campos.

* * *

Cenário 06 - Usuário edita postagem

- **Descrição**: O usuário é capaz de editar postagens de autoria própria.
- **Atores**: Usuário autenticado.

- **Pré-condições**: Usuário deve ter feito uma postagem (Cenário 05).
- **Pós-condições**:
  - Sucesso: O sistema atualiza a postagem, com as novas informações.
  - Falha: O sistema exibe mensagem de erro informativa.
- **Gatilho**: Usuário tenta editar postagem através do botão "editar".
- **Fluxo normal**:
  1. O usuário acessa a lista de postagens.
  2. O usuário clica em "ver mais" em uma postagem feita por ele.
  3. O usuário clica em "editar" no componente de postagem.
  4. O sistema exibe modal de edição de postagem.
  5. O usuário digita uma nova descrição/análise para a postagem.
  6. O usuário clica em "editar".
  7. O sistema atualiza a exibição atual, atualizando a data da postagem, e a descrição/análise.
- **Fluxos alternativos**:
  1. O usuário deixa a nova descrição vazia.
     (a) O usuário acessa a lista de postagens.
     (b) O usuário clica em "ver mais" em uma postagem feita por ele.
     (c) O usuário clica em "editar" no componente de postagem.
     (d) O sistema exibe modal de edição de postagem.
     (e) O usuário deixa a nova descrição vazia.
     (f) O usuário clica em "editar".
     (g) O sistema exibe mensagem informativa, alertando sobre a obrigatoriedade do campo vazio.

* * *

Cenário 07 - Usuário exclui postagem

- **Descrição**: O usuário é capaz de excluir postagens de autoria própria.
- **Atores**: Usuário autenticado.
- **Pré-condições**: Usuário deve ter feito uma postagem (Cenário 05).

- **Pós-condições**:
  - Sucesso: O sistema exclui a postagem.
- **Gatilho**: Usuário tenta excluir postagem através do botão “excluir”.
- **Fluxo normal**:
  1. O usuário acessa a lista de postagens.
  2. O usuário clica em “ver mais” em uma postagem feita por ele.
  3. O usuário clica em “excluir” no componente de postagem.
  4. O sistema exibe modal de confirmação de exclusão de postagem.
  5. O usuário confirma, clicando em “apagar”.
  6. O sistema exclui a postagem.
  7. O sistema navega para a página inicial, com a listagem atualizada.
- **Fluxos alternativos**: Não há.

* * *

Cenário 08 - Usuário comenta em postagem

- **Descrição**: O usuário é capaz de adicionar comentários em postagens da autoria de qualquer usuário.
- **Atores**: Usuário autenticado.
- **Pré-condições**: Algum usuário deve ter feito uma postagem (Cenário 05).
- **Pós-condições**:
  - Sucesso: O sistema atualiza a exibição da postagem, com o novo comentário.
  - Falha: O sistema exibe mensagem de erro descritiva.
- **Gatilho**: Usuário tenta comentar postagem através da área de texto, e o botão “comentar”, presentes na tela de postagem.
- **Fluxo normal**:
  1. O usuário acessa a lista de postagens.
  2. O usuário clica em “ver mais” em uma postagem.
  3. O usuário digita um comentário na caixa de texto presente após as informações da postagem.

4. O usuário clica em “comentar”.
5. O sistema atualiza a visualização, adicionando o novo comentário no topo da lista de comentários da postagem.
6. O sistema atualiza a visualização, incrementando o número de comentários da postagem.

- **Fluxos alternativos**:

1. O usuário deixa o comentário vazio.
   (a) O usuário acessa a lista de postagens.
   (b) O usuário clica em “ver mais” em uma postagem.
   (c) O usuário deixa a caixa de comentário vazia.
   (d) O usuário clica em “comentar”.
   (e) O sistema exibe mensagem de erro, informando que um novo comentário não pode ser vazio.

* * *

Cenário 09 - Usuário edita comentário

- **Descrição**: O usuário é capaz de editar comentários de autoria própria.
- **Atores**: Usuário autenticado.
- **Pré-condições**: Usuário deve ter feito um comentário (Cenário 08).
- **Pós-condições**:
  - Sucesso: O sistema atualiza o comentário.
  - Falha: O sistema exibe mensagem de erro descritiva.
- **Gatilho**: Usuário tenta editar comentário através do botão “editar”, presente na listagem de comentários, ao lado de comentários de autoria própria.
- **Fluxo normal**:

1. O usuário acessa a lista de postagens.
2. O usuário clica em “ver mais” em uma postagem.
3. O usuário visualiza a listagem de comentários, e clica em “editar”, à direita de um comentário de autoria própria.
4. O sistema exibe modal de edição de comentário.

5. O usuário digita o novo comentário.
6. O usuário clica em “editar” para confirmar a edição.
7. O sistema atualiza a visualização, atualizando o conteúdo do comentário, e a data atribuída.

- **Fluxos alternativos**:

1. O usuário deixa o novo comentário vazio.
   (a) O usuário acessa a lista de postagens.
   (b) O usuário clica em “ver mais” em uma postagem.
   (c) O usuário visualiza a listagem de comentários, e clica em “editar”, à direita de um comentário de autoria própria.
   (d) O sistema exibe modal de edição de comentário.
   (e) O usuário deixa o novo comentário vazio.
   (f) O usuário clica em “editar” para confirmar a edição.
   (g) O sistema atualiza a visualização, atualizando o conteúdo do comentário, e a data atribuída.

* * *

Cenário 10 - Usuário exclui comentário

- **Descrição**: O usuário é capaz de excluir comentários de autoria própria.
- **Atores**: Usuário autenticado.
- **Pré-condições**: Usuário deve ter feito um comentário (Cenário 08).
- **Pós-condições**:
  - Sucesso: O sistema exclui o comentário.
- **Gatilho**: Usuário tenta excluir postagem através do botão “excluir”, presente na listagem de comentários, ao lado de comentários de autoria própria.
- **Fluxo normal**:

1. O usuário acessa a lista de postagens.
2. O usuário clica em “ver mais” em uma postagem.
3. O usuário visualiza a listagem de comentários, e clica em “excluir”, à direita de um comentário de autoria própria.

4. O sistema exibe modal de exclusão de comentário.
5. O usuário confirma exclusão, clicando em “apagar”.
6. O sistema atualiza a visualização, apagando o comentário.

- **Fluxos alternativos**: Não há.

* * *

Cenário 11 - Sistema envia notificações

- **Descrição**: O sistema é capaz de enviar notificações aos usuários.
- **Atores**: Usuário A e Usuário B, ambos autenticados.
- **Pré-condições**: Usuário A deve ter feito postagem (Cenário 05), e Usuário B deve ter feito comentário na postagem adicionada pelo Usuário A (Cenário 08).
- **Pós-condições**:
  - Sucesso: O sistema envia notificação de comentário ao Usuário A, autor da postagem.
- **Gatilho**: Usuário adiciona comentário na postagem de outro usuário.
- **Fluxo normal**:
  1. O Usuário B, autor de comentário, comenta postagem do usuário A, autor da postagem.
  2. O sistema envia, ao usuário A, nova notificação.
  3. O sistema exibe, no botão “notificações”, presente na barra de navegação, indicador de nova notificação, ao usuário A.
  4. O usuário clica no botão “notificações”.
  5. O sistema exibe lista de últimas notificações recebidas, com indicador em notificações ainda não vistas.
  6. O usuário clica em uma das notificações.
  7. O sistema leva o usuário à página da postagem em que o comentário ocorreu.
  8. O sistema atualiza a visualização, removendo indicador na notificação vista.
- **Fluxos alternativos**: Não há.

* * *

Cenário 12 - Usuário realiza *logout*

- **Descrição**: O usuário é capaz realizar *logout* da plataforma.
- **Atores**: Usuário autenticado.
- **Pré-condições**: Usuário deve ter feito *login* (Cenário 02).
- **Pós-condições**:
  - Sucesso: O sistema remove o estado de "autenticado" do usuário.
- **Gatilho**: Usuário tenta realizar *logout* através de menu de gerenciamento de conta, presente na barra de navegação ao topo.
- **Fluxo normal**:
  1. O usuário clica no botão de menu com seu nome de usuário, presente na barra de navegação.
  2. O sistema exibe opções de *logout* e exclusão de conta.
  3. O usuário clica em "sair".
  4. O sistema remove o estado de autenticação do usuário.
  5. O sistema realiza as mudanças necessárias na interface, para refletir o novo estado de autenticação.
- **Fluxos alternativos**: Não há.

* * *

Cenário 13 - Usuário exclui sua conta

- **Descrição**: O usuário é capaz realizar exclusão de conta da plataforma.
- **Atores**: Usuário autenticado.
- **Pré-condições**: Usuário deve ter feito *login* (Cenário 02).
- **Pós-condições**:
  - Sucesso: O sistema remove a conta do usuário. O sistema remove referências ao usuário, substituindo-as por "Conta excluída". O usuário não deve ser capaz de realizar *login* com as mesmas credenciais.
- **Gatilho**: Usuário tenta realizar exclusão de conta através de menu de gerenciamento de conta, presente na barra de navegação ao topo.

- **Fluxo normal**:

    1. O usuário clica no botão de menu com seu nome de usuário, presente na barra de navegação.
    2. O sistema exibe opções de *logout* e exclusão de conta.
    3. O usuário clica em “excluir conta”.
    4. O sistema exibe modal de confirmação de exclusão de conta.
    5. O usuário confirma a exclusão, clicando em “Excluir conta”.
    6. O sistema remove o estado de autenticação do usuário.
    7. O sistema remove a conta do usuário.

- **Fluxos alternativos**: Não há.