UNIVERSIDADE FEDERAL DO ABC

Ana Paula Magalhães Silva

UTILIZANDO FERRAMENTAS GRATUITAS PARA AUTOMATIZAR TESTES DE SOFTWARE EM PORTAIS WEB

Santo André

2019

Ana Paula Magalhães Silva

UTILIZANDO FERRAMENTAS GRATUITAS PARA AUTOMATIZAR TESTES DE SOFTWARE EM PORTAIS WEB

Trabalho apresentado como requisito parcial para a Conclusão do Curso de Engenharia de Informação da Universidade Federal do ABC.

Orientador: Prof. Dr. Francisco de Assis Zampirolli

Santo André

2019

DEDICATÓRIA

Dedico este trabalho principalmente aos meus pais que mesmo com as dificuldades lutaram para que eu tivesse a melhor educação e me dão forças diariamente para seguir em frente. Dedico também aos meus companheiros de graduação Rodrigo e Patrícia, que compartilharam comigo essa longa jornada na UFABC e fizeram o meu caminho muito menos árduo.

AGRADECIMENTOS

Agradeço a Deus por me dar forças em todos os momentos que precisei; a minha família por estar sempre ao meu lado dando todo o suporte necessário; aos meus colegas de curso que compartilharam comigo momentos bons e difíceis; ao meu antigo chefe Jean que me incentivou a continuar no momento em que mais cheguei perto de desistir e finalmente ao Prof. Dr Francisco Zampirolli por toda paciência e orientação provida para que esse trabalho pudesse ser concluído.

RESUMO

A busca por sistemas com maior qualidade de software se tornou cada vez mais importante na medida em que software passou a estar presente no dia-a-dia das pessoas. As empresas envolvidas com desenvolvimento de software, apesar de desejarem sistemas de alta qualidade, acabam descobrindo que o tempo e esforço necessários para isso, podem se tornar um empecilho, visto que, a maioria dos projetos possuem prazos apertados de entrega e visto que o teste de software é uma fase extremamente custosa do processo de desenvolvimento. Diante desse impasse, a automação de testes de software vem ganhando popularidade entre os profissionais e empresas de desenvolvimento, já que o seu objetivo é justamente reduzir o custo e esforços do teste manual de software. Apesar de também ter custos, automatizar cenários de testes para homologação de funcionalidades, pode ser extremamente vantajoso para testes de natureza repetitiva, economizando tempo e esforço significativo. Neste estudo de caso é construído um projeto de automação de teste de software para as funcionalidades de uma plataforma educacional, desenvolvida por professores da UFABC, para auxiliar na elaboração e correção de exames. O estudo mostra que é possível automatizar testes de software para um dado sistema utilizando ferramentas gratuitas, minimizando significativamente tempo e esforço gastos na prática de testes manuais toda vez que uma nova versão de software é lançada.

Palavras-chave: testes de software; automação de teste; qualidade de software.

ABSTRACT

The search for higher quality software systems is becoming more important as softwares have become part of people's lives. The companies involved in software development, despite their wishes for high quality systems, have realized that the time and effort required might become an obstacle given that most projects require tight deadlines and an extremely expensive testing phase. Given these issues, the software testing automation has an increasing popularity among professionals and development companies whose goals are exactly to reduce time and manual testing. In spite of its costs, automate scenario testing during features approval can be extremely beneficial for repetitive tests, saving time and efforts. In this study an automation software project is developed for an educational platform to be used by the professors at the Federal University of ABC, specifically to assist in exams grading. This study shows that it is possible to automatize software testing to a given system using available free tools, minimizing significantly the time and efforts spent during manual testing phases each time a new software version is launched.

Key-words: Tests, Software, Automation, Quality.

SUMÁRIO

1. INTRODUÇÃO

A necessidade por sistemas com maior qualidade de software se tornou cada vez mais importante na medida em que software passou a estar presente no dia-a-dia das pessoas. Nos anos 90, as principais companhias tinham conhecimento e reconheciam que bilhões de dólares por ano estavam sendo gastos em software que não cumpriam a especificação de requisitos, ou seja, não apresentavam as funcionalidades esperadas (PRESSMAN, 2011). As empresas envolvidas com desenvolvimento de software, apesar de desejarem sistemas de alta qualidade, acabam descobrindo que o tempo e esforço necessários para isso, podem se tornar um empecilho, visto que, a maioria dos projetos possuem prazos apertados de entrega.

O teste de software é um mecanismo da qualidade, com o objetivo de descobrir defeitos. O papel de alguém que trabalha como testador, é garantir que a homologação de um sistema, seja conduzida sob planejamento apropriado de cenários de testes de forma eficiente, para que o resultado final seja um produto livre de bugs em produção. No entanto, o teste de software é uma fase extremamente custosa do processo de desenvolvimento, estima-se que os testes podem consumir 50% dos custos de todo o processo (BERTOLINO, 2007).

A automação de testes é uma abordagem usada para reduzir o custo e esforços do teste manual de software. Apesar de também não estar isento de custos, automatizar cenários para homologação de funcionalidades pode ser extremamente vantajoso para testes de regressão[1] e/ou natureza repetitiva, economizando tempo e esforço significativo. Além disso, elimina-se a propensão de erros que existiria caso a tarefa de homologação fosse realizada exaustivamente e frequentemente de forma manual por uma pessoa. Mesmo com isso, raramente automação de teste de software é aplicada na prática em empresas, na verdade, apenas uma pequena parcela dos profissionais de desenvolvimento, cerca de 6% (DOBSLAW et al., 2019) utilizam alguma ferramenta para automatizar testes. Isso se deve ao fato de que há muitos desafios na prática de automação, dado que, as ferramentas para tal tarefa ainda não são muito difundidas no mercado; os profissionais de testes teriam que ter conhecimentos de programação; e ainda há

1 https://www.testingcompany.com.br/blog/testes-de-regressao-como-onde-e-quando-utilizar/

muita dificuldade em estudar e estimar os custos diretos *versus* benefícios ao automatizar um conjunto de testes.

1.1. Objetivo

Este trabalho é um estudo de caso e tem como objetivo mostrar que é possível automatizar testes de software para um dado sistema utilizando ferramentas gratuitas, minimizando o esforço e tempo gastos em testes manuais toda vez que uma funcionalidade nova é implementada.

O objeto do presente estudo, para qual será aplicado o processo de garantia de qualidade de software, é uma plataforma para a geração e correção de exames chamado webMCTest.

2. PLATAFORMA PARA GERAÇÃO E CORREÇÃO DE EXAMES

A plataforma na qual serão aplicados os conceitos de automação de testes desenvolvidos ao longo deste trabalho, é o sistema webMCTest, desenvolvido por professores da Universidade Federal do ABC. O webMCTest é parte de uma pesquisa de uma década para atender demandas internas da instituição, tendo como principal problema a resolver, a grande quantidade de alunos para avaliar. O relatório mais recente mostra que até 4 de dezembro de 2018, a plataforma já havia gerado 21.359 exames e corrigido automaticamente cerca de 6.370, com um total de 138.452 questões (ZAMPIROLLI; TEUBL; BATISTA, 2019).

O sistema é gratuito e oferece aos professores a possibilidade de gerar e corrigir testes de forma automática e confortável mesmo se tratando de centenas ou milhares de alunos (ZAMPIROLLI; TEUBL; BATISTA, 2019). Além disso, ajuda no desafio de avaliá-los de maneira justa, já que fica fácil gerar provas com questões e alternativas diferentes, porém, abordando o mesmo assunto com o mesmo nível de dificuldade.

O sistema webMCTest armazena estruturas de instituições de ensino, com as principais entidades: Instituição, que possui Cursos, que possui Disciplinas. Cada Disciplina possui professores, coordenadores, Tópicos e Turmas. Um Tópico tem Questões. Uma Turma tem professores, estudantes e Exames. Os exames têm questões, que podem ter respostas (se de múltipla escolha). Considere que o usuário com perfil de administrador do webMCTest já criou Instituição (ex. UFABC),

Curso (ex. BC&T) e Disciplina (ex. Processamento da Informação), além de definir um professor coordenador para esta disciplina. O coordenador pode importar arquivos no formato CSV preenchendo os Professores da disciplina, Turmas e Estudantes.

A plataforma foi implementada utilizando Python 3.6 com Django 2.0 e utiliza um servidor de banco de dados MySQL. O servidor é executado em um sistema operacional Ubuntu Linux 18.04 (ZAMPIROLLI; TEUBL; BATISTA, 2019).

3. TEORIA

3.1. Software

Quando um software é bem desenvolvido, os usuários têm suas necessidades atendidas, funciona sem problemas por um longo tempo, tem fácil manutenção e é simples de operar (PRESSMAN, 2011). Entretanto, quando um software não é bem desenvolvido, ele falha, é muito difícil modificá-lo para dar manutenção e os usuários ficam descontentes, gerando problemas na relação entre uma empresa e um cliente. Atualmente, o software não é somente um produto, mas assume também o papel de ser um veículo para distribuir um produto. Sendo um produto, fornece o potencial computacional representado pelo hardware. O software distribui, transforma e gerencia informação, e em consequência disso, as indústrias que lidam com desenvolvimento se tornam extremamente importantes para a economia. O software segundo o livro do Pressman (2011), consiste em três pilares:

> "(1) instruções (programas de computador) que quando executadas, fornecem características, funções e desempenho desejados; (2) estruturas de dados que possibilitam aos programas manipular informações adequadamente; e (3) informação descritiva, tanto na forma impressa como na virtual, descrevendo a operação e o uso dos programas."

Diferente do hardware, o software não sofre desgastes causados por fatores físicos e/ou ambientais, porém, não está isento de defeitos. É normal que um sistema no início de sua utilização, apresente defeitos que não foram descobertos no processo de desenvolvimento. Mas a tendência é que, esses problemas sejam corrigidos e o aparecimento de novos problemas no sistema aconteça com pouquíssima frequência, isso é, se o programa foi bem escrito. Entretanto, durante o

ciclo de vida de um sistema, desde o desenvolvimento até a primeira versão entregue, o software pode passar por alterações, que são necessárias por conta de mudanças nas regras de negócio e/ou interesses do usuário final, e na medida que essas alterações acontecem, é possível que erros novos sejam introduzidos ao sistema, afetando possíveis novas funcionalidades ou até mesmo funcionalidades que já existiam. Por isso é extremamente importante adotar processos de engenharia de software e garantia de qualidade no desenvolvimento, para evitar que um sistema se torne um fardo cheio de problemas para quem os opera.

3.2. Engenharia de Software

Nas palavras de Boehm (1976), numa tradução livre para o português, Engenharia de Software pode ser definido como:

> "A aplicação prática de conhecimento científico no design e na construção de programas computacionais e a documentação associada necessária para desenvolvê-las, operá-las e mantê-las."

O termo "design" diz respeito a todas as atividades envolvidas em elaborar requisitos de software. A definição deve também levar em conta a cobertura de todo ciclo de vida do software, incluindo "redesign" e alterações no sistema, chamadas de manutenção de software.

A Engenharia de Software, pode ser considerada uma tecnologia em camadas, assim como é ilustrado na Figura 1. Como qualquer abordagem em engenharia, baseia-se num compromisso com foco na Qualidade.

Figura 1 - Camadas da Engenharia de Software

Fonte: Disponível em: <https://bit.ly/2L0HEMG>. Acesso em 20 Ago 2019

A gestão de qualidade promove uma cultura de melhoria constante de processo, e é justamente essa cultura que faz com o que o desenvolvimento seja feito a partir de abordagens eficazes para a Engenharia de Software.

A base da Engenharia de Software é a camada de Processos, pois, é o que une as camadas de tecnologia e possibilita o desenvolvimento inteligente e direcionado de software. Processo determina um *framework* que deve ser definido para entregas de sistemas.

A camada de Métodos, por sua vez, é responsável por tarefas que incluem requisitos, comunicação, modelagem de projeto, análise, programação, teste e suporte.

As Ferramentas de Engenharia de Software são quem fornecem apoio ou suporte para Processos e Métodos.

Neste trabalho, a camada que será mais estudada é a camada que diz respeito a qualidade de software, afinal, a qualidade do produto de software é fundamental para o seu sucesso.

3.3. Qualidade de Software

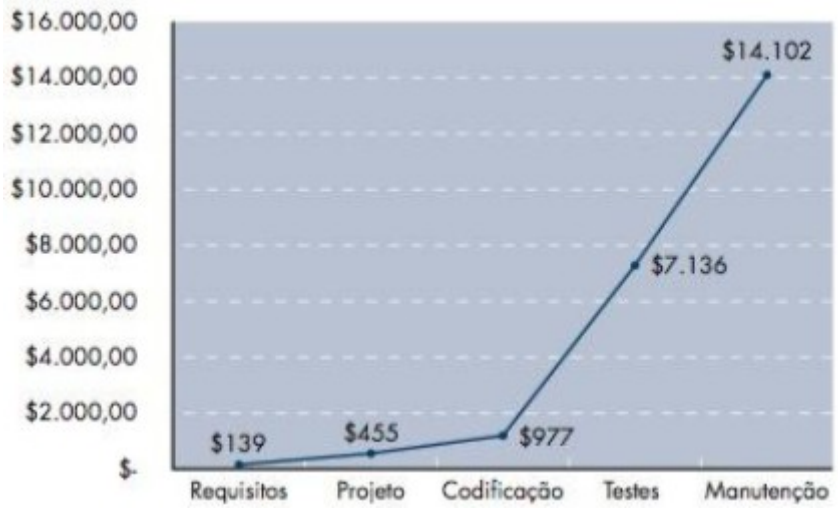
Um sistema de alta qualidade certamente foi construído observando processos e práticas comprovadas de Engenharia de Software, além de um gerenciamento impecável de projetos. No desenvolvimento de software, a qualidade de projetos está diretamente ligada com o cumprimento das especificações no modelo de requisitos. A definição de qualidade de software dada por Pressman (2011) é "uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam".

Sabe-se então que qualidade é fundamental para o sucesso do produto, mas as companhias acabam entrando num dilema já que o processo de garantia de qualidade possui também um preço. Dessa forma, algumas companhias acabam não entendendo que a falta de qualidade tem um custo maior, não só porque os usuários finais vão ter que operar e lidar diariamente com um sistema cheio de defeitos, mas também porque a manutenção de um software mal construído e mal

validado é mais complicada e leva mais tempo. Os custos gerados a partir dessas questões são chamados de "custos de falhas externas", que são decorrentes de defeitos encontrados após o sistema ter sido lançado. Exemplos de gastos com falhas externas são resolução de chamados (reclamações), suporte remoto ou presencial e correção de erros encontrados pelo usuário final.

Estudos mostram que quanto mais se demora para descobrir um falha/bug, mais caro será para corrigi-lo. O custo médio para corrigir um problema caso ele seja descoberto na fase de testes é de US$ 7.136, em contrapartida, caso esse mesmo erro seja descoberto após a entrega, ele custará praticamente o dobro US$14.102 (PRESSMAN, 2011).

Figura 2 - Tempo *versus* Custo de correção de um defeito



Fonte: Pressman, 2011

Assim, a fase de testes é essencial em qualquer processo adotado no desenvolvimento de software. Apesar de não existir um sistema 100% livre de defeitos ou inconformidades com os requisitos, se as tarefas relevantes à fase de testes forem bem conduzidas, pode-se garantir um produto estável, limitando as alterações no produto após a entrega apenas a adaptações solicitadas e mudanças em regras de negócio do cliente.

3.3.1. Testes de Software

Teste de software inclui não somente criação, avaliação e execução de casos

de teste, como também inclui o processo de comunicação entre as pessoas envolvidas no desenvolvimento de um sistema. Isso engloba discussões com pares, elaboração de relatórios de teste, leitura de *logs* (ou relatórios) do sistema, tal como tomar decisões mediante análise de resultados de teste.

No dia-a-dia de profissionais de desenvolvimento de software, como desenvolvedores e testadores, é constantemente necessário tomar decisões a respeito do que implementar, corrigir ou testar primeiro, priorizando tarefas da forma mais conveniente. Essas decisões podem ser também a nível de projeto, decidir se o software foi suficientemente validado, se tem uma boa qualidade e se pode ser entregue em breve.

O teste é o método padrão para detectar imperfeições de qualidade. A maior parte de testes de software na indústria são conduzidos de forma manual, desde a criação de testes até a execução para avaliação de resultados. Segundo Kasurinen (2010), citado por Strandberg e Afzal (2019), as práticas de testes manuais ultrapassaram 90% em um estudo na Finlândia em 2010 e um relatório de 2015 feito pela Capgemini, Sogeti e HP relatam que apenas 28% dos casos de teste são automatizados. O teste de software pode compor entre 30% e 80% de todo o custo de desenvolvimento de software (STRANDBERG, AFZAL, et al., 2019). A automação de testes poderia reduzir substancialmente esse custo, melhorando inclusive o tempo de entrega de um projeto ou de uma nova funcionalidade de um sistema.

3.3.2. Automação de Testes de Software

No teste manual, o testador assume o papel de um usuário operando o sistema para validar o comportamento e encontrar qualquer defeito que possa existir. Em testes automatizados, o profissional de qualidade deve também assumir o papel de desenvolvedor para implementar *scripts* de código de teste, na maioria das vezes utilizando *frameworks*, como por exemplo *Junit* (junit.org) para testes unitários e Selenium (seleniumhq.org) para testes direto na interface gráfica. Se for bem planejado e bem implementado, o teste automatizado se mostrará mais eficiente e com mais benefícios em relação ao teste manual, se mostrando útil principalmente nos casos em que existe a repetibilidade de execução de certos

testes, como os de regressão. O esforço para a prática de testes então será reduzido e portanto, o custo também.

No entanto, a automação de testes nem sempre é necessariamente eficaz. Definir quais partes de um sistema devem ser testadas de forma automática, pode ser um tanto complicado; um testador de software inexperiente poderia afirmar que testes manuais podem ser totalmente substituídos por automação, mas pode ser que isso não seja viável. Se não se sabe definir quais testes são importantes e quais testes são aplicáveis para automação, então, a ferramenta só ajudará a fazer um teste ruim mais rápido (RICE, 2003).

Geralmente, um processo de teste, automatizado ou não, compreende várias etapas para testar especificação, execução e relatórios. Essas etapas podem ser compreendidas em seis principais atividades (GAROUSI, ELBERZHAGER, 2016):

1. Design de caso de teste;
2. Teste de *script*;
3. Execução de teste;
4. Avaliação do teste;
5. Relatório de resultados de testes;
6. Gerenciamento de testes e outras atividades relacionadas a engenharia de software.

3.3.2.1. Design de caso de teste

Nesta atividade é gerado um conjunto de casos de testes ou requisitos de testes, nada mais é do que planejar e escrever os cenários de teste que devem ser validados, seja do ponto de vista de cumprimento de especificação de requisitos, seja do ponto de vista funcional do sistema.

A elaboração de casos de teste é baseada no conhecimento humano, isso é, na especificação de requisitos, e é manual. Para utilizar essa documentação de cenários de testes na automação, podem ser utilizadas diversas ferramentas, que possibilitarão escrever esses casos em linguagem natural mas utilizando palavras-chaves a partir das quais serão implementados os *scripts* para execução automática dos testes.

3.3.2.2. Teste de *script*

Nesta fase é que são implementados de fato os comandos via alguma linguagem de programação, para executar o que foi escrito na fase de design dos casos de teste. Testadores executam manualmente esses *scripts* há muitos anos, e inicialmente, surgiram muitas ferramentas de automação *record-and-playback* (por exemplo *HP QuickTest Professional*, que permitiam aos profissionais de teste registrar um cenário de teste interagindo com a interface gráfica do sistema, enquanto a ferramenta gravava automaticamente o teste como um *script* em segundo plano. Posteriormente, o código de testes gerado poderia ser executado quantas vezes fossem necessárias.

Entretanto, essa técnica de automação pode consumir muito tempo e pode acabar produzindo *scripts* frágeis. Diante dessas limitações, foram surgindo diversos outros *frameworks* em que é possível de fato programar um "robozinho" para fazer o que um humano teria que fazer lendo os casos de teste.

Um *script* de teste, é um conjunto detalhado de "instruções" para executar um fluxo particular no sistema a ser homologado.

A automação de testes tem como objetivo executar os *scripts* desenvolvidos sem intervenção manual. Existem ferramentas para isso como o Selenium e o Sikuli (sikulix.com), mas como dito, para automatizar testes utilizando essas ferramentas, códigos precisam ser escritos.

Para o testador desenvolvedor criar esses *scripts*, ele precisa conhecer a linguagem de programação com a qual está trabalhando, e precisa saber olhar através do HTML da página (caso esteja sendo testado um sistema Web) para identificar os elementos com os quais precisará fazer o *script* interagir.

3.3.2.3. Execução de teste

Os testes serão executados no sistema a ser homologado, e o resultados deverão ser devidamente registrados observando saídas e comportamento do software. As decisões tomadas durante as fases anteriores afetam a execução do teste. Se o testador desenvolver a automação de todos os testes de uma suíte de casos de teste, a execução será totalmente automatizada e existem artifícios (*plugins*) de *frameworks* para gerar relatórios da execução. Vale lembrar que uma

suíte de testes é o mesmo que conjunto de testes.

3.3.2.4. Avaliação do teste

Existem algumas formas para avaliar os resultados dos testes, se estes passaram ou se falharam. Quando o teste é feito manualmente, essa avaliação vai ser feita por uma pessoa. Quando o teste é automatizado, no desenvolvimento geralmente é colocado *asserts* para verificar se o comportamento do sistema é o esperado.

3.3.2.5. Relatório de resultados de testes

A fase de geração de relatórios é geralmente a última do processo de qualidade. Nesta fase são relatados os defeitos ou inconformidades encontrados. No teste manual, o próprio testador elabora esse relatório, enquanto que no teste automatizado, existem diversas ferramentas integradas aos *frameworks* de automação que geram relatórios automaticamente após a execução de uma *suíte* de testes.

3.3.2.6. Gerenciamento de testes e atividades relacionadas a engenharia de software

Essas atividades incluem a revisão de uma *suíte* de testes, identificação de testes redundantes e a seleção de testes convenientes para uma *suíte* de regressão. Essa *suíte* de testes de regressão inclui testes fundamentais para garantir as principais funcionalidades já implementadas de um sistema, e será executada toda vez que uma funcionalidade nova ou uma correção for implementada para esse sistema, isso é necessário para garantir que o funcionamento do sistema não foi afetado com novas alterações de código.

4. METODOLOGIA

Para atingir os resultados esperados, que se baseiam na automação de teste de software das principais funcionalidades da plataforma webMCTest (abordado na Seção 2 deste trabalho), as seguintes estratégias são apresentadas:

- Definição de quais funcionalidades devem ter os testes automatizados;
- Definição das ferramentas gratuitas a serem utilizadas para automação dos

testes;

- Design dos cenários de teste;
- Programação dos testes automatizados a partir de uma linguagem escolhida;
- Apresentar resultados consolidados num relatório legível a qualquer papel do negócio, seja o profissional de qualidade, seja o desenvolvedor da plataforma, o dono do produto ou o usuário.

4.1. Métodos e ferramentas utilizados

4.1.1. BDD - *Behavior Driven Development*

O desenvolvimento orientado a comportamento foi desenvolvido por Dan North. O BDD é um processo de desenvolvimento de software que se concentra na elaboração detalhada do comportamento do sistema, de forma que possa ser automatizado. Os comportamentos das funcionalidades do sistema são descritos e observados como especificações executáveis e tem o foco em como os usuários finais interagem com a plataforma (ALI, AWWAD, SLANY, 2019).

Inicialmente, os cenários em BDD podem ser usados como critérios de aceitação, escritos com a ajuda da linguagem Gherkin (cucumber.io/docs/gherkin). Os cenários descrevem como um recurso deve agir em diferentes situações com diferentes parâmetros de entrada. Os casos de teste em BDD devem ser claramente escritos e facilmente compreensíveis para todas as partes interessadas de um projeto, pois, é possível que sejam escritos em idiomas naturais (como inglês e português) que ajudam na especificação dos testes.

Para o processo de Dan North então, especificação executável é o mesmo que teste automatizado, que mostra e verifica como uma determinada funcionalidade atende a um requisito. A intenção é que sempre que houver uma alteração no sistema, essa especificação executável seja executada como parte do processo de desenvolvimento. Sendo assim, essa documentação serve tanto como guia para testes de aceitação (determinando quais novos recursos estão completos), como para testes de regressão, garantindo que alterações no código não danifiquem nenhuma funcionalidade que já existe no sistema. Utilizando uma ferramenta chamada *Cucumber* (cucumber.io), o profissional de qualidade poderá automatizar uma especificação relacionando cada etapa de um cenário executável com o código

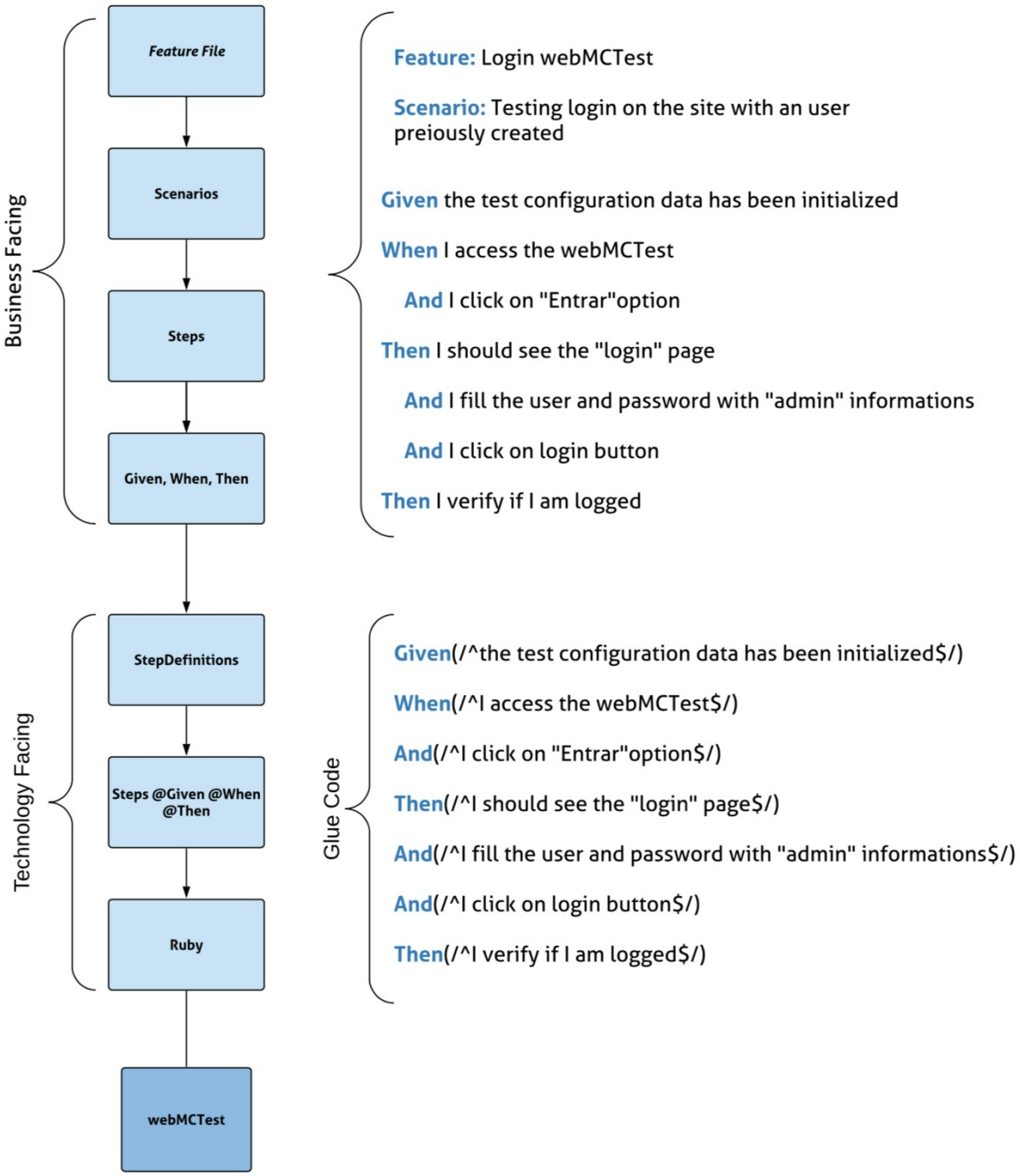
de teste apropriado usando uma linguagem de programação.

4.1.2. *Cucumber*

Cucumber (cucumber.io) é um *framework* de teste *open source* usado para executar testes de aceitação automatizados que foram criados no formato BDD. Um de seus principais recursos é a capacidade de realizar descrições funcionais de um texto simples - escritos na linguagem *Gherkin* - como testes automatizados (INFOQ, 2018). O *Cucumber* então executa especificações, escritas em linguagens naturais chamadas de *features.*

A Figura 3 mostra a estrutura em que os testes são implementados para uma aplicação quando se usa *Cucumber*: Cada *feature* possui diversos *scenarios* e cada *scenario* possui uma lista de *steps.* Nesse exemplo, pode-se observar as palavras chaves *Given, When* e *Then* para verificar os testes no sistema exemplo. Portanto, o *framework Cucumber* pode interpretar esses arquivos *".features"* desde que eles sigam as regras básicas de sintaxe do *Gherkin*. A partir dos arquivos *.features,* quando o *Cucumber* é executado, ele procura pelos *steps* correspondentes num arquivo *step definition* que mapeia a linguagem natural de cada etapa em um código específico utilizando uma linguagem de programação. Em cada passo do *step definition* devem estar presentes códigos ou chamadas para funções para que o *framework* saiba o que testar. Se o código for executado sem erros, o *Cucumber* prosseguirá para o próximo passo do cenário de teste e se chegar ao fim sem nenhum problema, dará o cenário como aprovado. Se algum *step* do cenário falhar, o *Cucumber* considera que o cenário apresentou problemas e passa para a execução do cenário de teste seguinte (Ali, Awwad e Slany, 2019).

Figura 3 - Estrutura dos testes com *Cucumber*



4.1.3. *Gherkin*

Gherkin (cucumber.io/docs/gherkin) é uma linguagem de programação que o *framework Cucumber* é capaz de interpretar e usa para definir casos de teste. É uma linguagem legível por humanos e de fácil compreensão mesmo para não programadores. Na prática, o *Gherkin* tem dois objetivos: Documentar e automatizar

testes, com ferramentas como o *Cucumber*. A gramática do *Gherkin*, segundo Ehrenfried (2019, p.23) deriva das histórias de usuário do estilo *Given - When - Then,* e portanto ela pode ser utilizada como única forma de formalizar requisitos. O *Gherkin* define apenas algumas palavras-chave obrigatórias e o restante das palavras para descrição das *features* pode ser usada de forma livre de acordo com o idioma escolhido. Abaixo segue a definição das principais palavras-chave utilizadas no contexto de automação de teste:

- *Feature* - Uma história de usuário escrita na estrutura legível do *Gherkin*. A palavra *Feature* é utilizada para documentar o nome da funcionalidade que será descrita. Um arquivo com extensão *.feature* deve ter relação com apenas uma funcionalidade, que contém um ou mais cenários de teste;
- *Scenario* - Um arquivo *.feature* contém diversos cenários e todo cenário possui um único caso de teste que, por sua vez, é composto por um ou mais passos (*steps*). Cada passo do cenário terá uma palavra-chave de contexto, que pode ser: ***Given*** *(*pré-condição do teste*),* ***When*** (ação do usuário) e ***Then*** (resultado esperado). ***And*** e ***But*** também são consideradas palavras chaves do *Gherkin*, utilizadas para melhorar a escrita quando existe mais de um passo com as palavras chaves principais;
- *Step definition* - Esta é a parte do código responsável por relacionar os testes escritos utilizando *Cucumber* com o código responsável por executar os *scripts* para que os testes de fato sejam feitos. O *Cucumber* é usado para converter o arquivo *.feature* nos passos contidos no arquivo de *step definitions.* A responsabilidade do *step definition* é traduzir os *steps* dos cenários escritos em *Gherkin* para código de programação efetivamente, utilizando linguagens como Java, Ruby ou Python. Uma expressão regular é usada para combinar os passos entre aspas duplas ou entre os símbolos "/^" e "$/". O *framework* usa essas expressões regulares para combinar os cenários com os nomes dos métodos gerados para fazer a chamada de funções para executar os testes.

4.1.4. *Ruby*

A linguagem *Ruby* (ruby-lang.org) nasceu em 1993 mas somente foi

apresentada ao público em 1995. Foi criada pelo japonês Yuri Matsumoto e é considerada uma linguagem limpa e direta, completamente orientada a objetos, simples de se aprender e trabalhar, parecida com Python (SOUZA, 2008).

O *Ruby* possui um gerenciador de pacotes chamado *RubyGems*. As *gems* são bibliotecas reutilizáveis de código *Ruby*, análogos aos *jars* no ambiente Java. Uma vantagem dessa linguagem que a diferencia das demais é o fato de que o *Ruby* é uma linguagem dinamicamente tipada, onde variáveis são atribuídas dinamicamente, sem a necessidade de especificar o tipo na criação. Isso diminui a verbosidade da linguagem, de forma que, se torna fácil definir variáveis e comportamento de objetos (CORBUCCI e ANICHE, 2014) .

4.1.5. *Capybara*

Capybara[2] é um *framework* de testes de aceitação para aplicações *web*, e é uma ferramenta excelente para auxiliar na interação entre uma funcionalidade desenvolvida e um navegador, seja para realizar testes ou apenas para coletar dados de um site. O *Capybara* não é o que realmente interage com uma plataforma, é uma camada que fica entre o usuário e o *webdriver* que pode ser o *Selenium*, *PhantomJS (phantomjs.org)*, entre outros. Esse *framework* fornece uma interface comum e um grande número de métodos auxiliares para extrair informações, inserir dados, testar ou clicar em elementos do sistema.

Essa ferramenta é usada principalmente para testes de integração, que validam não apenas um pedaço de código, mas sim um fluxo inteiro de história de usuário. Pode-se testar se um determinado conteúdo existe na página, pode-se inserir dados em um formulário e enviá-lo, interagir com botões ou *links*, etc.

4.1.6. *Selenium WebDriver*

Selenium WebDriver (seleniumhq.org/projects/webdriver) é uma ferramenta que oferece uma API que permite a escrita de forma mais produtiva e organizada de *script* de testes. Essa ferramenta realiza chamadas de forma direta ao navegador, fazendo uso do suporte à automação nativo de cada *browser* (GOES, 2017).

O *Selenium WebDriver* não é uma ferramenta de teste independente, é uma API que permite através de programação interagir com elementos de uma página

2 https://rubygems.org/gems/capybara/versions/2.7.1?locale=pt-BR

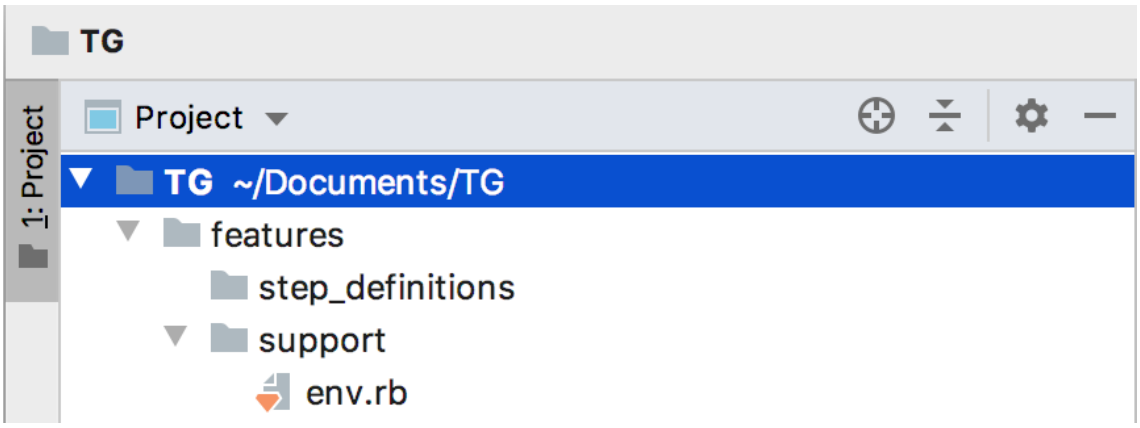
em um *browser*. Embora seja usado na maioria dos casos para realizar testes de um aplicativo *web*, também pode ser usado para qualquer tarefa em que seja necessário automação do navegador.

4.2. Procedimento adotado

4.2.1. Instalação de ferramentas e inicialização de repositório

Iniciar um projeto de automação de testes com as ferramentas estudadas até agora não é uma tarefa complexa. A primeira tarefa foi realizar a instalação do Ruby (ex. *apt install ruby-full,* no *ubuntu*) e a do cucumber (*apt install cucumber*), em versões estáveis. Em seguida foi criada uma pasta com o nome do projeto e após executar o comando *cucumber --init* no terminal dentro do diretório do projeto, foi criada a estrutura que pode ser observada na Figura 4.

Figura 4 - Estrutura dos testes com Cucumber após inicialização



Conforme explicado na Seção 4.2, dentro do repositório *features* devem ser criados os arquivos com a extensão *.feature* para cada funcionalidade do sistema a ser testada. Na pasta *step_definitions* será criado um *script ruby*, com extensão ".rb" onde serão feitas as chamadas para as funções responsáveis por realizar os testes automaticamente. A pasta suporte a princípio contém apenas o arquivo *env.rb* com configurações necessárias para a execução do projeto. Conforme o desenvolvimento dos testes forem progredindo, alguns outros arquivos de extensão ".rb" com funções dos testes automatizados serão também armazenados nesta pasta.

Os cenários mostrados nas próximas seções são alguns exemplos dos testes

desenvolvidos neste projeto para facilitar a compreensão da estrutura do repositório. Todos os cenários e todo o programa implementado para a execução dos cenários podem ser consultados nos apêndices desta monografia ou na página da autora no GitHub.

4.2.2. Definição de funcionalidades a serem automatizadas

A primeira tarefa foi estudar a plataforma webMCTest e mapear os principais cenários a serem automatizadas. Essa análise foi feita considerando a importância de cada funcionalidade da plataforma e considerando a repetibilidade de execução de um cenário cada vez que algo for alterado no código ou uma funcionalidade nova for implementada.

Para exemplificar, pode-se pensar no *Login* da plataforma, que é uma funcionalidade sensível e que é um dos primeiros passos que um usuário da plataforma precisa executar para fazer qualquer outra coisa. O teste fundamental quando se pensa nisso é realizar o *login* com um usuário pré-existente e verificar se o *login* foi de fato realizado. Essa é uma validação que precisa ser feita sempre antes de lançar uma nova versão de produção. Baseando-se na metodologia BDD e usando a linguagem *Gherkin*, foram desenvolvidos cenários de teste básicos para o *Login*, que podem ser vistos na imagem abaixo.

Figura 5 - Cenários de teste elaborados para a funcionalidade de login

Login.feature

```
1  Feature: Login webMCTest
2
3  @Login
4  Scenario: Testing login on the site with an user previously created
5    Given the test configuration data has been intialized
6      When I access the webMCTest
7          And I click on "Entrar" option
8      Then I should see the "login" page
9          And I fill the user and password with "coordenador" informations
10         And I click on login button
11     Then I verify if I am logged
12
13 @Logout
14 Scenario: Testing logout on the site with an user previously created
15     Given the test configuration data has been intialized
16     When I access the webMCTest
17         And I click on "Entrar" option
18     Then I should see the "login" page
19         And I fill the user and password with "coordenador" informations
20         And I click on login button
21     Then I verify if I am logged
22     Then I click on logout button
23         And I verify I am unloged
```

Neste exemplo, pode-se observar várias características citadas na descrição das ferramentas. A palavra chave *feature* logo na primeira linha do projeto descreve o nome da funcionalidade a ser validada. Em seguida tem a diretiva "@Login", que atribui um nome para um caso de teste. *Scenario* por sua vez permite fazer uma breve descrição do que vai ser validado no presente caso de teste, que neste exemplo, é o teste do login na plataforma utilizando um usuário previamente criado. *Given* é a palavra chave que permite determinar pré-condições para possibilitar a execução dos testes; no caso deste cenário, as classes que vão ser acessadas nos testes são instanciadas. No passo seguinte, a palavra *When* indica a primeira ação do teste para acessar a plataforma webMCTest; *AND* é a continuação da descrição da ação e significa clicar na opção "Entrar" exibida na plataforma. A partir daí começa-se a validar os resultados das ações realizadas anteriormente, já que a palavra *Then* indica que algo deverá ter acontecido, que neste caso é a exibição da página de *login*. Os passos seguintes são os responsáveis por preencher os campos

de usuário e senha, clicar no botão de *login* e verificar se o usuário realmente foi logado.

Da mesma forma como esse cenário de teste foi construído para validar uma funcionalidade do sistema utilizando o "caminho feliz" - a forma ideal como um usuário que conhece as regras de negócio operaria o sistema - podem ser construídos cenários para validar exceções do sistema. Utilizando como exemplo a funcionalidade de cadastro de usuário do webMCTest, podemos observar na Figura 6, dois dos cenários de teste desenvolvidos que validam regras para a criação de novos usuários. No primeiro teste, *@Register_With_Incorrect_Domain*, valida-se a regra de negócio que define que somente será possível criar um usuário novo no sistema, utilizando um email que tenha dominio *@ufabc.edu.br* (assume-se que o administrador já criou uma instituição com esse domínio). É necessário observar porém que o teste não valida o cenário em que o usuário segue a regra de negócio, e sim o cenário em que o usuário tenta criar uma conta com um email de domínio desconhecido. No step *Then I fill the email field with a user that not contains ufabc domain* a instrução de teste é para que seja utilizado justamente um email sem domínio da instituição de ensino UFABC para que no último *step Then I see a message error that requires a email with ufabc domain* seja testado se uma mensagem de erro é exibida e se o usuário é impedido de criar uma conta.

Figura 6 - Cenários de teste elaborados para a funcionalidade de cadastro de usuário

Register.feature

```
15    @Register_With_Incorrec_Domain
16    Scenario: Testing the message error when the email doesn't contain ufabc domain
17        Given the test configuration data has been intialized
18        When I access the webMCTest
19            And I click on "Inscrever" option
20        Then I should see the "Inscrever" page
21        Then I fill the email field with a user that not contains ufabc domain
22            And I fill the first and last name
23            And I fill the password and his confirmation "following" requirements with "same" keys
24            And I click on signup button
25        Then I see a message error that requires a email with ufabc domain
26
27    @Register_With_Too_Short_Password
28    Scenario: Testing the password requirements
29        Given the test configuration data has been intialized
30        When I access the webMCTest
31            And I click on "Inscrever" option
32        Then I should see the "Inscrever" page
33            And I fill the email field
34            And I fill the first and last name
35            And I fill the password and his confirmation "unfollowing" requirements with "same" keys
36            And I click on signup button
37        Then I see a message error that requires a longer password
```

No cenário seguinte *@Register_With_Too_Short_Password*, o caso de teste descrito valida as regras definidas para a criação da senha no momento do cadastro. Observando o *step And I fill the password and his confirmation "unfollowing" requirements with "same" keys,* percebe-se que será dado o *input* no campo de senha e confirmação de senha, uma senha que não segue as regras do sistema e portanto, quando o usuário tentar concluir o cadastro, será exibida uma mensagem de erro solicitando uma senha que cumpra os requisitos exigidos.

4.2.3. Definição dos *steps* de teste e conversão para linguagem de programação

Os passos de um cenário de teste escrito conforme mostrado nos exemplos da sessão anterior devem ser definidos em um arquivo de extensão ".rb", utilizando expressão regular para relacionar os casos de teste escritos em linguagem natural utilizando o *framework Cucumber*, com o código responsável por executar os *scripts* que no caso deste projeto será *Ruby*.

Um arquivo chamado "*StepDefinitions.rb*" foi criado dentro do diretório

step_definitions exibido na Figura 4. Neste arquivo, cada *step* é separado a partir da palavra-chave utilizada para se relacionar com a chamada de uma função adequada para a execução do teste.

Figura 7 - Estrutura do arquivo StepDefinitions.rb

StepDefinitions.rb

```
1   Given(/^the test configuration data has been intialized$/) do
2     $poHome = Home.new
3     $poLogin = Login.new
4     $poRegister = Register.new
5     $poAuxiliar = Auxiliar.new
6     $poAcesso = ControleDeAcesso.new
7   end
8
9   When(/^I access the webMCTest$/) do
10    visit 'http://177.104.60.16:8000'
11    puts page.current_url
12  end
13
14  Then(/^I check if every menus are present$/) do
15    $poHome.checkHeader
16  end
17
18  And(/^I click on "([^"]*)" option$/) do |option|
19    $poHome.clickMenu(option)
20  end
21
22  Then(/^I should see the "([^"]*)" page$/) do |option|
23    $poHome.checkPage(option)
24  end
25
26  And(/^I fill the user and password with "([^"]*)" informations$/) do |option|
27    $poLogin.loginUser(option)
28  end
29
30  And(/^I click on login button$/) do
31    $poLogin.clickLogin
32  end
33
34  Then(/^I click on logout button$/) do
35    $poLogin.clickLogout
36  end
37
38  Then(/^I verify if I am logged$/) do
39    $poLogin.checkIfIamlogged
40  end
```

Utilizando como exemplo o primeiro *step* exibido no cenário básico de *login*, temos a palavra chave *Given* que é seguida da sentença "*the test configuration data has been initialized*", que garante que todas as pré-condições necessárias para

execução do teste sejam respeitadas, que neste caso, é apenas instanciar todas as classes que serão utilizadas na execução dos cenários automatizados.

No segundo *step* separado por expressão regular temos descrita a ação que o usuário executa para acessar a plataforma webMCTest.

```
When(/^I access the webMCTest$/) do
     visit 'http://177.104.60.16:8000'
     puts page.current_url
end
```

Neste trecho de código, observa-se que o passo descrito anteriormente no arquivo "*.feature*" é agora uma função onde são passadas instruções que o *framework* de teste deve fazer. Neste caso, a primeira instrução é dada por meio de um comando do *Capybara* que acessa a *url* passada em seguida. A segunda instrução é dada a partir de um comando também do *Capybara* para capturar a *url* da página atual no *browser*; o "*puts"* é o comando da linguagem *Ruby* que permite que seja impresso algo no console.

Existem casos para alguns passos em que são muitas instruções a serem dadas para executar o que o teste pede. Quando isso acontece, o ideal é realizar chamadas para funções que estão escritas em classes específicas para cada funcionalidade.

```
     And(/^I fill the user and password with "([^"]*)"
informations$/) do |option|
          $poLogin.loginUser(option)
     end
```

No trecho de código acima, é descrito o passo em que são preenchidos os campos de usuário e senha no momento do *login*. Observe que nesse caso, diferente de como é mostrado na descrição da *feature*, não é passado o perfil do usuário que fará *login*, é passado apenas os caracteres "([^"]*)", isso porque, esse passo em específico foi desenhado para atender o *login* de usuários de todos perfis,

seja ele, administrador, professor, coordenador, etc. O parâmetro que for passado na descrição da *feature* (no exemplo da Figura 5, "coordenador") será interpretado como o parâmetro *"option"* e passado para a função *loginUser* como argumento. No método *loginUser* contida na classe *Login*, esse argumento passado será tratado e indicará quais informações serão utilizadas para realizar *login*.

4.2.4. Implementação dos testes automatizados utilizando *Ruby*

Para cada funcionalidade automatizável foi criada uma classe, onde foram definidos os métodos que executam os testes descritos nos cenários a partir da linguagem *Ruby* e utilizando comandos do *framework Capybara*.

A estrutura desse arquivo é similar à estrutura de uma classe de qualquer projeto de programação, em que os métodos são escritos dentro de uma classe definida; logo após a classe ser definida, a biblioteca do *Capybara* é incluída para que os comandos do *framework* sejam reconhecidos.

Os métodos são definidos normalmente na linguagem *Ruby*, utilizando *def* que é a palavra-chave para a criação de funções que podem ou não receber parâmetros. O método é finalizado com a palavra *end.*

Outro recurso do *Ruby* que utilizamos bastante no desenvolvimento dos testes são métodos de exceção, que indicam quando algo deu errado. Isso porque por padrão, os programas *Ruby* terminam quando ocorre um problema, então precisamos declarar e implementar manipuladores de exceção. Um manipulador de exceção é um bloco de código que é executado caso ocorra um comportamento inesperado durante a execução de outro bloco.

Figura 8 - Alguns métodos da classe *Login*

Login.rb

```ruby
class Login
  include Capybara::DSL

  def clickEntrar
    within('#navbarSupportedContent') do
      login = find('.nav-link', :text => 'Entrar')
      login.click
    end
  end

  def checkFields
    within('div.card-body') do
      raise "Missing mail's label! " unless find('.input-group-prepend', text: 'Endereço de email:')
      raise "Missing mail's field! " unless find('input#id_username')
      raise "Missing password's label! " unless find('label', text: 'Senha:')
      raise "Missing password's field! " unless find('input#id_password')
      raise "Missing login button!  " unless find('.btn.btn-primary', text: 'Entrar')
      raise "Missing forgot password button! " unless find('.btn.btn-warning', text: 'Perdeu a senha?')
    end
  end

  def clickLogin
    within('div.card-body') do
      find('.btn.btn-primary').click
    end
  end

  def clickLogout
    within('#navbarSupportedContent') do
      find('.nav-link', text: 'Sair').click
    end
  end
```

Nos métodos apresentados na Figura 8 é possível observar também a utilização de comandos do *framework Capybara* para interagir com o *browser*. São exemplos disso os comandos *find* que encontra um elemento da página baseado nos argumentos passados como *ids* ou classes de localização; *within* executa um bloco especificado no contexto de um elemento, funciona como o *find*, mas durante a execução do bloco, qualquer comando será tratado como se tivesse um escopo definido para o elemento especificado; *click* é o método utilizado para clicar em elementos clicáveis como botões e *links*.

O método mostrado na Figura 9, reúne vários dos artifícios utilizados na automação das *features* do webMCTest. Conforme o explicado na Seção 4.3.3., o método *loginUser* recebe um argumento *option* que indica qual perfil deverá ser utilizado para fazer *login* no sistema. Para isso, utiliza-se o *Ruby case statement* para que não seja necessário usar muitos *if* e *else* para implementar a lógica da

função. O parâmetro *option* é passado entre aspas no *step* do cenário executado a partir do arquivo "*.feature".*

Figura 9 - Método *loginUser(option)*

Login.rb

```
34    def loginUser(option)
35      case option
36      when 'coordenador'
37        userData = $poAuxiliar.readCredentials('coordenador')
38      when 'professor'
39        userData = $poAuxiliar.readCredentials('professor')
40      when 'estudante'
41        userData = $poAuxiliar.readCredentials('estudante')
42      when 'administrador'
43        userData = $poAuxiliar.readCredentials('administrador')
44      when 'usuário sem privilégios'
45        userData = $poAuxiliar.readCredentials('usuário sem privilégios')
46      end
47      within('div.card-body') do
48        @user = userData['login']
49        password = userData['password']
50        find('input#id_username').send_keys @user
51        find('input#id_password').send_keys password
52      end
53    end
```

De acordo com o caso em que a condição ocorrer, a variável *userData* é atribuída com o retorno de função chamada *readCredentials* que foi implementada na classe *Auxiliar*, onde ficam os métodos implementados para fazer coisas que não estão diretamente relacionadas aos testes, mas que ajudam a executá-los.

O método readCredentials é responsável por *parser* de um arquivo *".json"* contido no diretório *data* do projeto para capturar as informações necessárias para logar com cada tipo de usuário.

Figura 10 - Método *readCredentials(userType)*

Auxiliar.rb

```
1   class Auxiliar
2     def readCredentials(userType)
3       filepath = "./features/step_definitions/data/credentials.json"
4       file = File.read(filepath)
5       data_hash = JSON.parse(file)
6       case userType
7       when 'coordenador'
8         data_hash['coordenador']
9       when 'professor'
10        data_hash['professor']
11      when 'estudante'
12        data_hash['estudante']
13      when 'administrador'
14        data_hash['administrador']
15      when 'usuário sem privilégios'
16        data_hash['usuário sem privilégios']
17      end
18    end
```

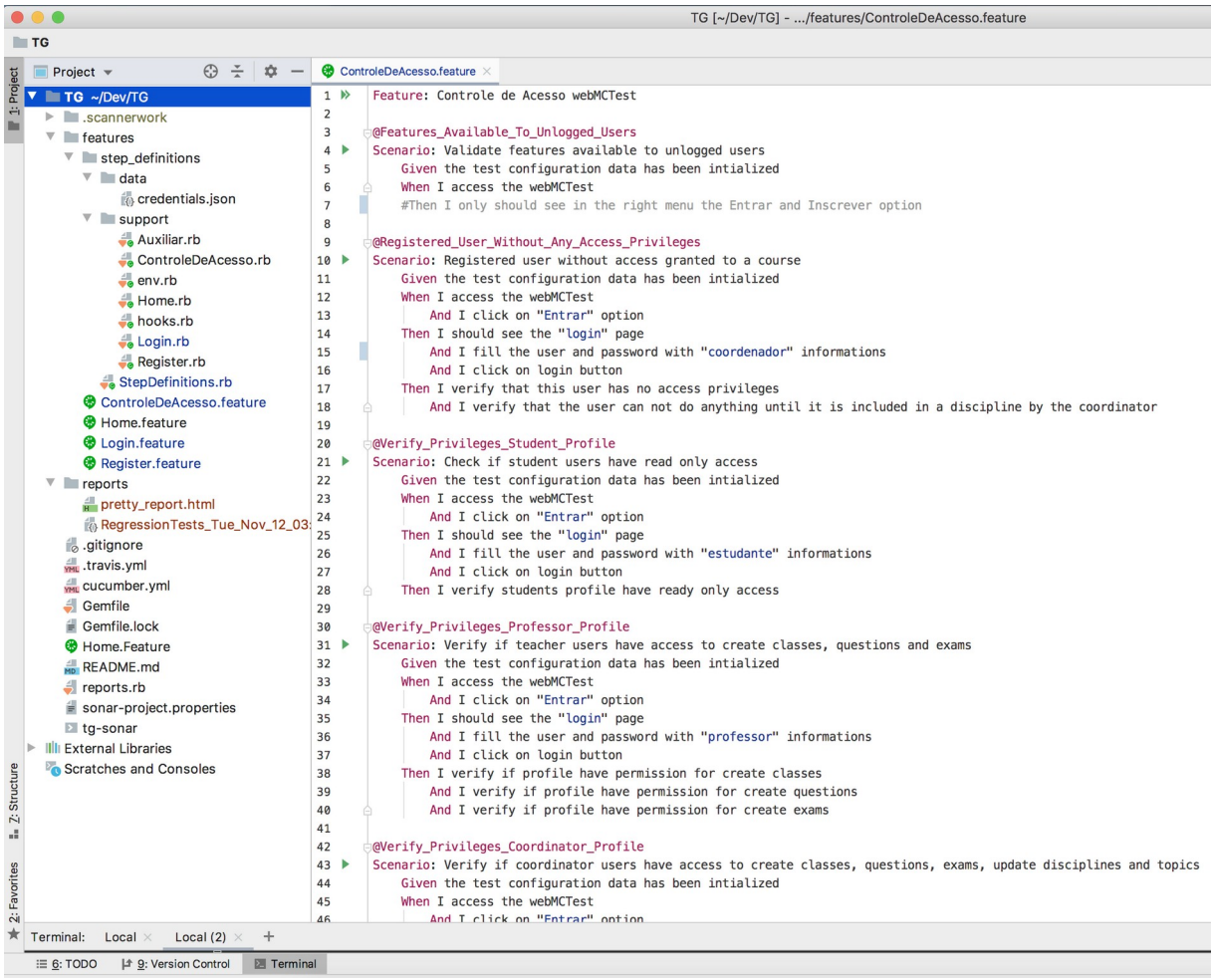
O comando *send_keys* pertence a biblioteca do *framework Capybara que* é utilizado para enviar uma informação a um determinado campo.

5. RESULTADOS E DISCUSSÕES

5.1. Execução dos Testes

Utilizando apenas ferramentas gratuitas, foram desenvolvidos diversos cenários de teste para a plataforma webMCTest. Até o momento da finalização desta monografia, a *suite* de testes contava com cinco classes de teste diferentes com inúmeros casos para funcionalidades da plataforma.

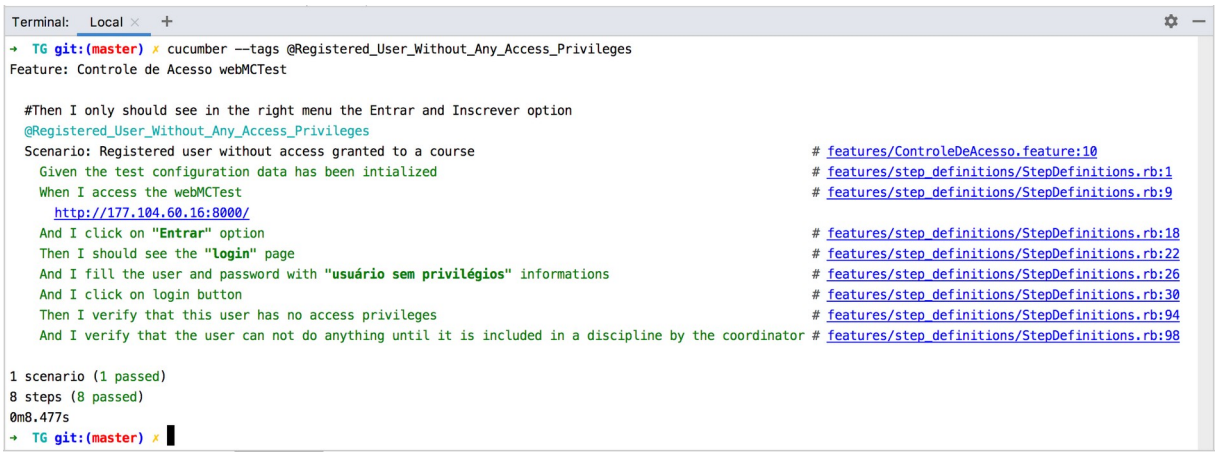
Figura 11 - Estrutura final do projeto de automação de testes para o webMCTest



Com essa estrutura, é possível executar cada cenário definido dentro de alguns instantes, evitando todo e qualquer esforço manual.

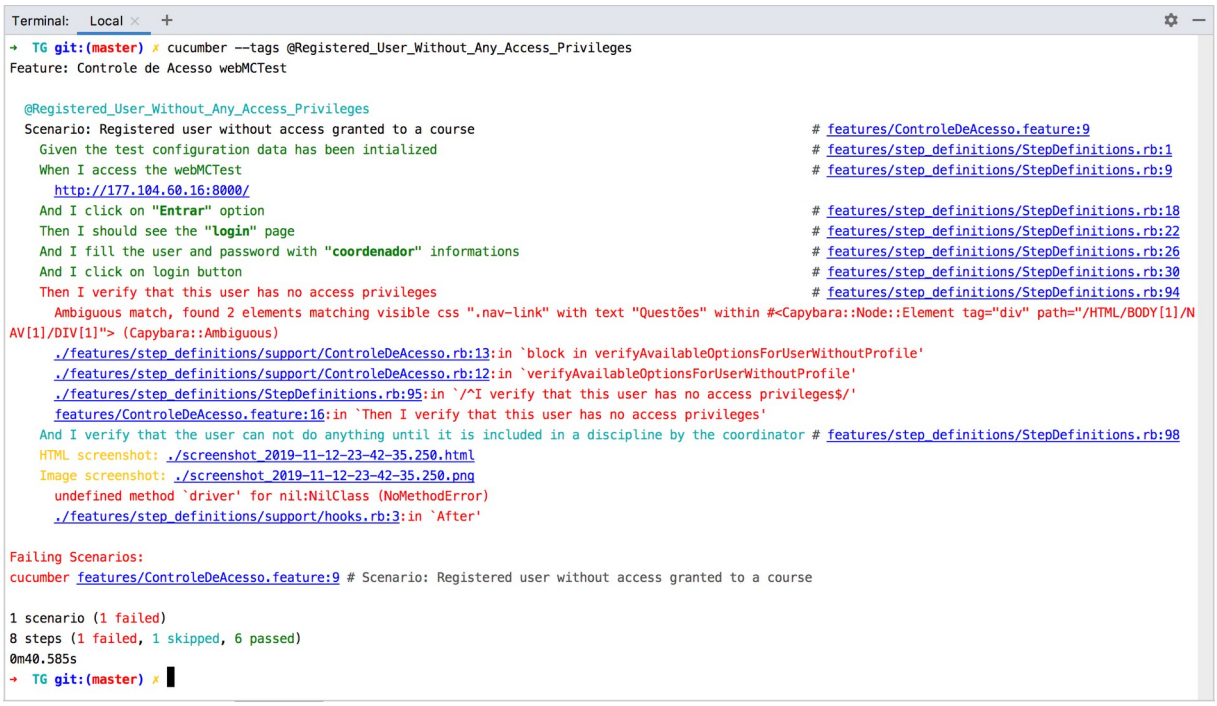
Por meio do comando `cucumber --tags @Feature_A_Ser_Executada` é possível executar separadamente cada cenário de teste, de modo que cada passo executado é exibido no console com a indicação de sucesso ou falha e um resumo no final, indicando quantos passos passaram, reprovaram ou foram pulados. Na Figura 12 pode-se observar a execução do cenário *@Registered_User_Without_Any_Access_Privileges* que verifica se um usuário que ainda não foi cadastrado em nenhuma disciplina tem acesso negado para fazer qualquer ação na plataforma até que seja incluído numa disciplina pelo coordenador da disciplina (atribuído pelo administrador).

Figura 12 - Execução do teste *@Registered_User_Without_Any_Access_Privileges*



Deve-se pensar também nos casos em que o teste pode falhar. E para exemplificar isto, será executado o mesmo cenário de teste, porém, alterando o parâmetro passado no quinto passo para "coordenador". O teste irá falhar, pois, terá logado no sistema com um perfil de acesso a diversas funcionalidades, sendo que o caso de teste quer justamente validar se o usuário não pode ter acesso enquanto não for cadastrado em uma disciplina.

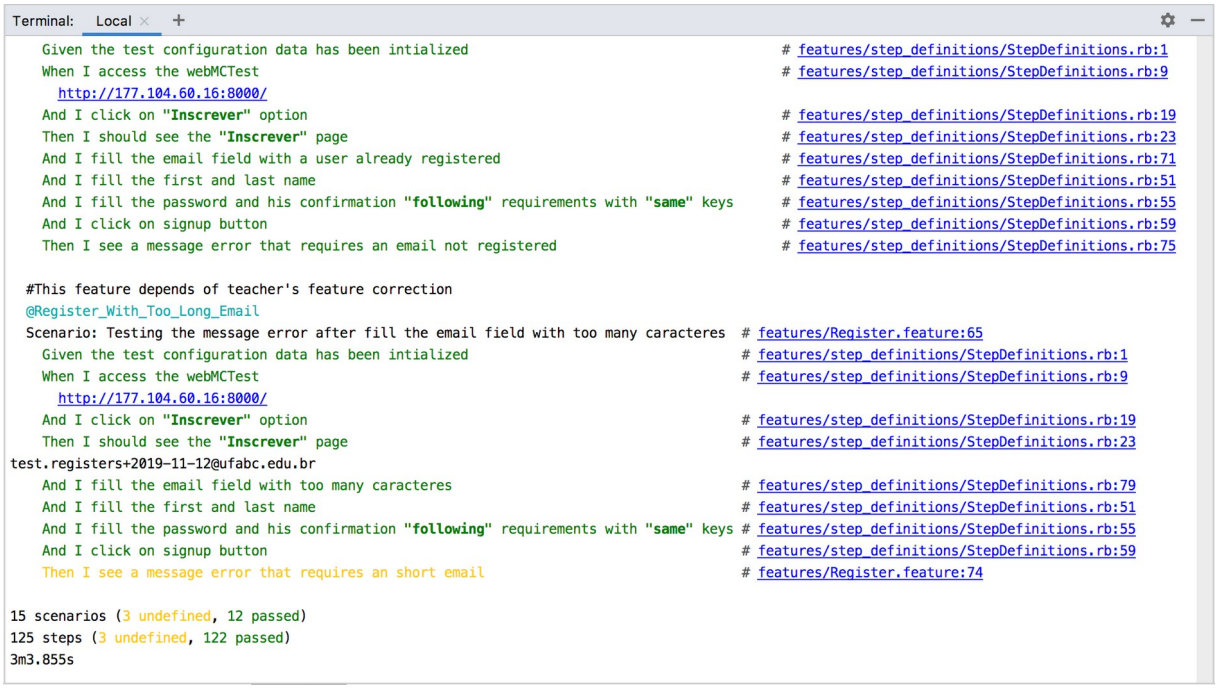
Figura 13 - Execução do teste modificado *@Registered_User_Without_Any_Access_Privileges*



O teste falha justamente no passo em que são verificados quais menus o usuário tem acesso, e falha porque encontra um menu a mais de questões que não deveria ter.

Executando o comando *cucumber* sem passar nenhuma *tag* de teste, todos os testes serão executados e no final será exibido um resumo pequeno de como foi a validação. No caso da execução mostrada na Figura 14, 125 *steps* de testes foram executados em 15 cenários dentro de alguns minutos, sendo que 122 *steps* passaram e 3 *steps* ainda não haviam sido programados.

Figura 14 - Execução e resultado de todos os cenários de teste automatizados



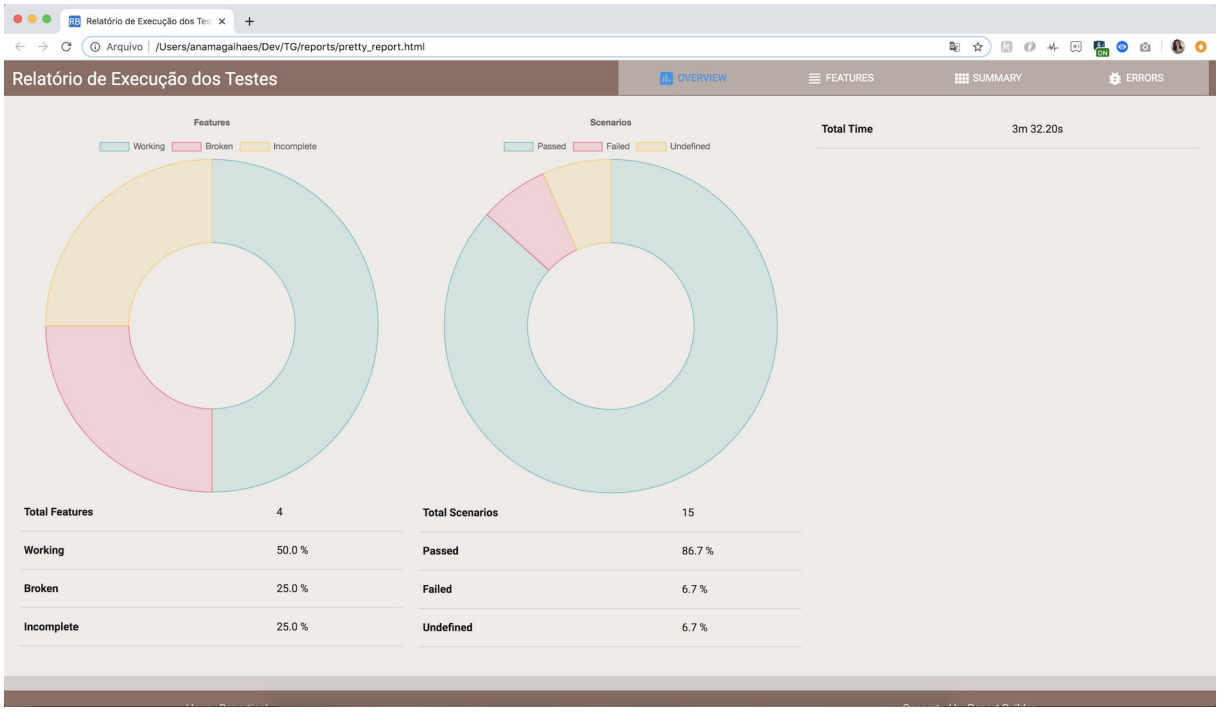
5.2. Consolidação de resultados

Apesar do *Cucumber* apresentar um resumo simplificado da execução dos testes, não se considera que isso seja um relatório apresentável a um cliente, desenvolvedor ou qualquer pessoa que cumpra um papel num projeto. Tendo isso em vista, encontrou-se uma *gem* do *Ruby* chamada *ReportBuilder* que possibilita gerar um relatório HTML único a partir de relatórios "*.json*" individuais que podem ser gerados pelo *Cucumber* a cada teste.

Para usar *ReportBuilder*, basta realizar a instalação conforme documentação (RUBYDOC, 2016) e executar os testes utilizando o comando `cucumber -p json`

que irá gerar um arquivo *json* a cada *feature* executada. Em seguida, para consolidar todos os resultados num relatório HTML, deve ser executado o comando `ruby reports.rb`, ver Figura 15.

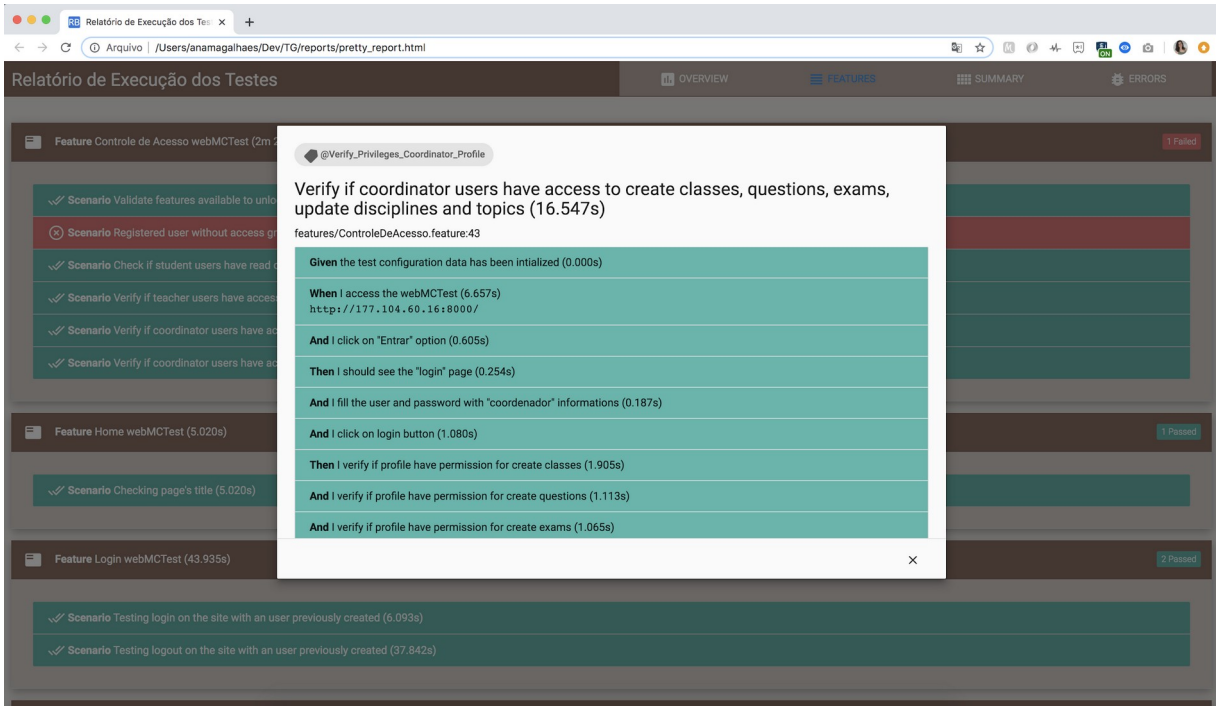
Figura 15 - Relatório gerado após a execução da *suite* completa de testes



Na página *Overview*, Figura 15, são mostrados gráficos e informações da porcentagem de testes que falharam, que passaram e os que ainda não foram programados.

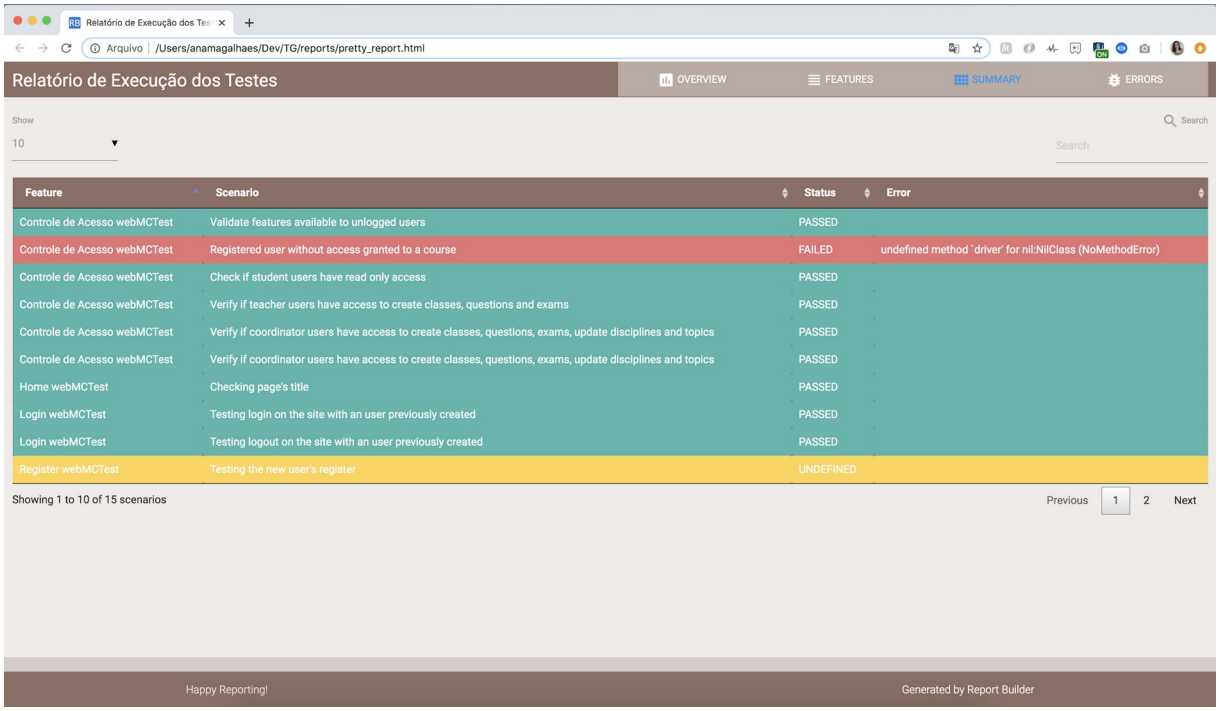
Na aba *Features* são mostrados todos os cenários executados com detalhes de cada *step*, incluindo tempo de execução.

Figura 16 - Página *Features* no relatório gerado após execução de testes



Na página *Summary* é exibido um resumo sem detalhes dos cenários, ver Figura 17.

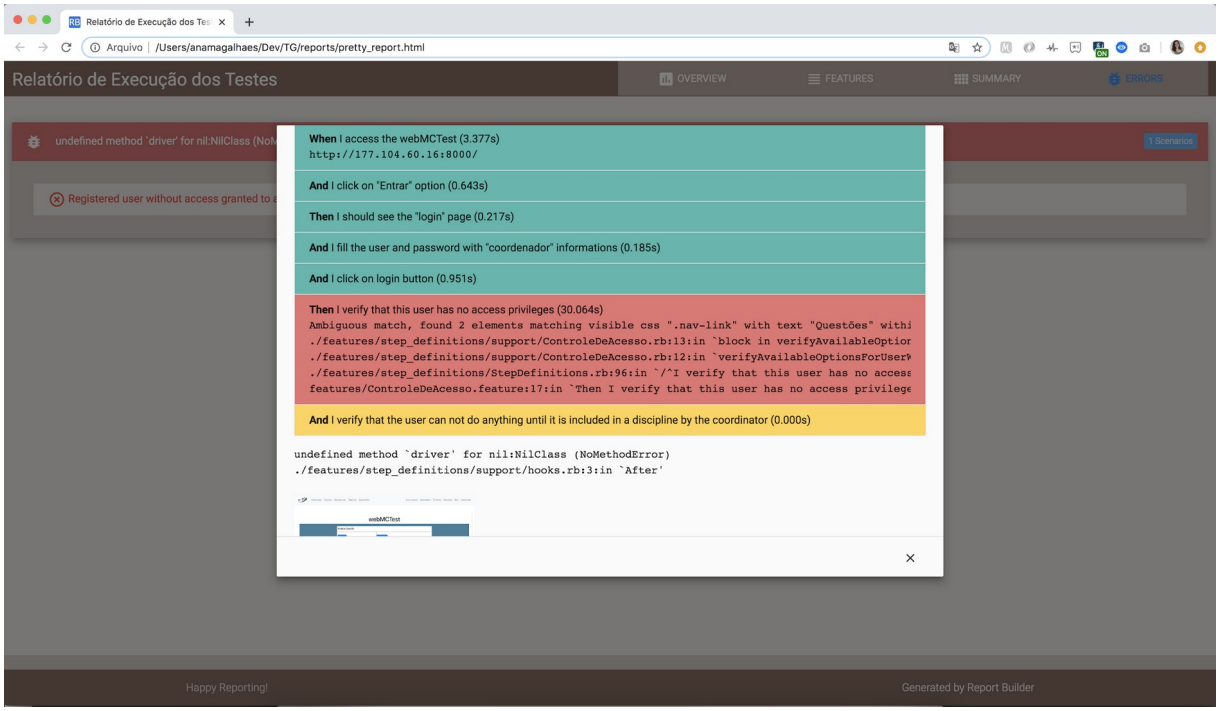
Figura 17 - Página *Summary* no relatório gerado após execução de testes



E finalmente, na aba *Errors* são mostrados em detalhes os *steps* que

falharam em cada teste junto com um *print screen* do exato momento em que o teste falhou, ver Figura 18.

Figura 18 - Página de Error no relatório gerado após execução de testes



Este relatório gerado em HTML traz informações valiosíssimas que podem ser analisadas sempre após a execução dos testes executados por conta de uma alteração e/ou melhoria em alguma funcionalidade da plataforma ou até mesmo na implementação de novas funcionalidades. É um artefato de teste interessante que pode ser arquivado documentando sempre todas as validações que foram feitas a cada versão de software.

6. CONSIDERAÇÕES FINAIS

A busca por garantia de qualidade de software vem se tornando cada vez mais importante para as empresas, na medida em que surgem diversos estudos mostrando que muito dinheiro se gasta em sistemas que não apresentam as funcionalidades esperadas. O papel de garantir a qualidade de um sistema é atribuído a fase de teste de software, que é um mecanismo para tentar descobrir defeitos, que deve ser conduzido sob planejamento apropriado de cenários de testes para que o resultado final seja um produto livre de *bugs* em produção. Em

contrapartida, o teste de software é uma fase custosa do processo de desenvolvimento, chegando a consumir 50% dos custos de todo o ciclo de desenvolvimento de uma plataforma, isso porque, na maioria das vezes demanda esforço manual de um profissional para realizar testes manuais a cada nova versão do sistema. É até complicado estimar quantas horas são necessárias para revalidar todas as funcionalidades de um sistema pronto e ainda validar possíveis novas funcionalidades.

A automação de testes é uma abordagem que visa reduzir custo e esforços do teste manual de software, dando agilidade na execução de todo o processo. Mesmo com isso, raramente automação de teste de software é aplicado na prática em empresas e isso se deve ao fato de que há muitos desafios na prática de automação, visto que, a maioria das ferramentas ainda não são tão conhecidas no mercado e visto que os profissionais de testes teriam que adquirir conhecimentos de programação e levariam um tempo considerável de aprendizado, ou seja, automação também tem custos.

Neste estudo de caso foi mostrado que com o uso de boas ferramentas e metodologias, é possível otimizar esforço e tempo, além de contribuir para a documentação de um sistema, utilizando uma linguagem que seja compreendida por qualquer pessoa envolvida num projeto e até mesmo o usuário final.

Utilizando apenas ferramentas gratuitas e trabalhando poucas horas por semana num curto período de tempo, foi possível desenvolver um projeto de automação de software de um sistema real que já está em produção. A principal vantagem a ser notada foi o ganho de tempo na execução de testes, que conforme foi mostrado ao longo deste trabalho, possibilitou executar cerca de 15 cenários de teste com 122 *steps* em cerca de 4 minutos. Além disso, uma vez que os testes foram desenvolvidos, podem ser executados a qualquer momento com apenas alguns comandos.

Com o uso de BDD, foi possível conseguir um ganho notável na reutilização de *steps,* ou seja, de código, para implementação de diversos testes, contribuindo também para a fácil e contínua manutenção do código e dos requisitos. Além disso, os cenários de testes escritos se tornam também parte da documentação do sistema, atrelando requisito ao código.

Apesar de os resultados terem sido satisfatórios, acredita-se que seriam ainda melhores caso os cenários de testes tivessem sido desenvolvidos conforme recomenda a literatura, antes do desenvolvimento de determinada *feature* e usados como requisitos para desenvolver o sistema. Ou seja, utilizar um processo de desenvolvimento de software orientado a testes. Isso facilitaria o entendimento de negócio da plataforma, que teve que ser realizado depois que a mesma já estava em operação.

Com tudo isso, pode-se afirmar que o custo-benefício de se automatizar testes se mostra relevante em relação a prática de testes manuais. Pode ser que a princípio, uma organização precise investir tempo para capacitação dos profissionais de qualidade ou dinheiro para a contratação de novos. Mas o que esse trabalho mostra, é que passado o tempo de aprendizado inicial de linguagem de programação e ferramentas, é possível desenvolver com facilidade *scripts* que façam automaticamente em minutos o que uma pessoa levaria horas para fazer manualmente.

REFERÊNCIAS BIBLIOGRÁFICAS

ALI, Z.; AWWAD, A.; SLANY, W. Using Executable Specification and Regression Testing for Broadcast Mechanism of Visual Programming Language on Smartphones, 2019.

BERTOLINO, Antonia. Software testing research: Achievements, challenges, dreams, in Future of Software Engineering, IEEE, 2007.

BOEHM, Barry W. Software Engineering, IEEE TRANSACTIONS ON COMPUTERS, VOL. C-25, NO. 12, DECEMBER 1976.

CORBUCCI, Hugo; ANICHE, Mauricio. Test-driven Development: Teste e Design no Mundo Real com Ruby. 1 ed. São Paulo: Casa do Código, 2014.

DE SOUZA, Thiago Cruz. Conhecendo a Linguagem Ruby. Medium, [S.L], jun./jul. 2008. Disponível em: . Acesso em: 11 nov. 2019.

DOBSLAW, F.; FELDT, R.; MICHAELSSON, D. Estimating Return on Investment for GUI Test Automation Tools, e-Informatica Software Engineering Journal, Volume 10, 2016, pages: 69–87.

EHRENFRIED, Henrique V. GHERKIN SPECIFICATION EXTENSION - UMA LINGUAGEM DE ESPECIFICAÇÃO DE REQUISITOS BASEADA EM GHERKIN, 2019.

GAROUSI, V.; ELBERZHAGER, F. Test Automation Not Just for Test Execution, 2016.

GOES, Ricardo. Introdução ao Selenium WebDriver, Testes Automatizados, 2017. Disponível em: www.testesautomatizados.com.br/introducao-selenium-webdriver/. Acesso em: 09 de nov. 2019.

INFOQ. Testes de Automação com Cucumber BDD em times Ágeis. Disponível em: https://www.infoq.com/br/articles/cucumber-bdd-automation-testing/. Acesso em: 11 de nov. 2019.

PRESSMAN, Roger S. Engenharia de Software - Uma Abordagem Profissional.

7ª.ed.[S.l.]: AMGH Editora Ltda., 2011.

RICE, Randall W., Surviving the Top Ten Challenges of Software Test Automation, Rice Consulting Solutions, LLC, 2003.

RUBYDOC, ReportBuilder, Ruby gem to merge Cucumber JSON reports and build single HTML Test Report. Disponível em: https://www.rubydoc.info/gems/report_builder/0.0.9. Acesso em: 11 de nov. 2019.

ZAMPIROLLI, F. A.; TEUBL, F.; BATISTA, V. R. Online Generator and Corrector of Parametric Questions in Hard Copy Useful for the Elaboration of Thousands of Individualized Exams, Maio 2019.

APÊNDICE A - CÓDIGO FONTE DO PROJETO DE AUTOMAÇÃO DE TESTE [3]

```
Feature: Home webMCTest

@CheckHeader
Scenario: Checking page's title
   Given the test configuration data has been intialized
   When I access the webMCTest
   Then I check if every menus are present
```

```
Feature: Login webMCTest

@Login
Scenario: Testing login on the site with an user previously created
  Given the test configuration data has been intialized
  When I access the webMCTest
     And I click on "Entrar" option
  Then I should see the "login" page
     And I fill the user and password with "coordenador" informations
     And I click on login button
  Then I verify if I am logged

@Logout
Scenario: Testing logout on the site with an user previously created
  Given the test configuration data has been intialized
  When I access the webMCTest
     And I click on "Entrar" option
  Then I should see the "login" page
     And I fill the user and password with "coordenador" informations
     And I click on login button
  Then I verify if I am logged
  Then I click on logout button
     And I verify I am unloged
```

```
Feature: Register webMCTest

@Register
Scenario: Testing the new user's register
  Given the test configuration data has been intialized
  When I access the webMCTest
     And I click on "Inscrever" option
  Then I should see the "Inscrever" page
     And I fill the email field
     And I fill the first and last name
     And I fill the password and his confirmation "following" requirements with "same" keys
     And I click on signup button
  Then I verify if the user was correctly created

@Register_With_Incorrec_Domain
Scenario: Testing the message error when the email doesn't contain ufabc domain
  Given the test configuration data has been intialized
  When I access the webMCTest
     And I click on "Inscrever" option
  Then I should see the "Inscrever" page
  Then I fill the email field with a user that not contains ufabc domain
     And I fill the first and last name
     And I fill the password and his confirmation "following" requirements with "same" keys
     And I click on signup button
  Then I see a message error that requires a email with ufabc domain

@Register_With_Too_Short_Password
Scenario: Testing the password requirements
  Given the test configuration data has been intialized
```

3 https://github.com/AnaPaulaMagalhaesSilva/TG

```
  When I access the webMCTest
     And I click on "Inscrever" option
  Then I should see the "Inscrever" page
     And I fill the email field
     And I fill the first and last name
     And I fill the password and his confirmation "unfollowing" requirements with "same"
keys
     And I click on signup button
  Then I see a message error that requires a longer password

@Register_With_Differents_Password
Scenario: Testing the message error after fill two differents password
  Given the test configuration data has been intialized
  When I access the webMCTest
     And I click on "Inscrever" option
  Then I should see the "Inscrever" page
     And I fill the email field
     And I fill the first and last name
     And I fill the password and his confirmation "following" requirements with "differents"
keys
     And I click on signup button
  Then I see a message error that requires two same passwords

@Register_With_Existent_Email
Scenario: Testing the message error after fill the email field with a user already
registered
  Given the test configuration data has been intialized
  When I access the webMCTest
     And I click on "Inscrever" option
  Then I should see the "Inscrever" page
     And I fill the email field with a user already registered
     And I fill the first and last name
     And I fill the password and his confirmation "following" requirements with "same" keys
     And I click on signup button
  Then I see a message error that requires an email not registered

@Register_With_Too_Long_Email
Scenario: Testing the message error after fill the email field with too many caracteres
  Given the test configuration data has been intialized
  When I access the webMCTest
     And I click on "Inscrever" option
  Then I should see the "Inscrever" page
     And I fill the email field with too many caracteres
     And I fill the first and last name
     And I fill the password and his confirmation "following" requirements with "same" keys
     And I click on signup button
  Then I see a message error that requires an short email
```

```
Feature: Controle de Acesso webMCTest
@Features_Available_To_Unlogged_Users
Scenario: Validate features available to unlogged users
  Given the test configuration data has been intialized
  When I access the webMCTest
  Then I only should see in the right menu the Entrar and Inscrever option

@Registered_User_Without_Any_Access_Privileges
Scenario: Registered user without access granted to a course
  Given the test configuration data has been intialized
  When I access the webMCTest
     And I click on "Entrar" option
  Then I should see the "login" page
     And I fill the user and password with "usuário sem privilégios" informations
     And I click on login button
  Then I verify that this user has no access privileges
     And I verify that the user can not do anything until it is included in a discipline by
the coordinator

@Verify_Privileges_Student_Profile
Scenario: Check if student users have read only access
```

```
  Given the test configuration data has been intialized
  When I access the webMCTest
     And I click on "Entrar" option
  Then I should see the "login" page
     And I fill the user and password with "estudante" informations
     And I click on login button
  Then I verify students profile have ready only access

@Verify_Privileges_Professor_Profile
Scenario: Verify if teacher users have access to create classes, questions and exams
  Given the test configuration data has been intialized
  When I access the webMCTest
     And I click on "Entrar" option
  Then I should see the "login" page
     And I fill the user and password with "professor" informations
     And I click on login button
  Then I verify if profile have permission for create classes
     And I verify if profile have permission for create questions
     And I verify if profile have permission for create exams

@Verify_Privileges_Coordinator_Profile
Scenario: Verify if coordinator users have access to create classes, questions, exams,
update disciplines and topics
  Given the test configuration data has been intialized
  When I access the webMCTest
     And I click on "Entrar" option
  Then I should see the "login" page
     And I fill the user and password with "coordenador" informations
     And I click on login button
  Then I verify if profile have permission for create classes
     And I verify if profile have permission for create questions
     And I verify if profile have permission for create exams
     And I verify if profile have permission for update disciplines
     And I verify if profile have permission for create and update topics

@Verify_Privileges_Admin_Profile
Scenario: Verify if coordinator users have access to create classes, questions, exams,
update disciplines and topics
  Given the test configuration data has been intialized
  When I access the webMCTest
     And I click on "Entrar" option
  Then I should see the "login" page
     And I fill the user and password with "administrador" informations
     And I click on login button
  Then I verify if profile have permission for create classes
     And I verify if profile have permission for create questions
     And I verify if profile have permission for create exams
     And I verify if profile have permission for update disciplines
     And I verify if profile have permission for create and update topics
     And I verify if profile have permission for create and update institutes
     And I verify if profile have permission for create and update courses
     And I verify if profile have permission for access admin menu
```

StepDefinitions.rb

```
Given(/^the test configuration data has been intialized$/) do
 $poHome = Home.new
 $poLogin = Login.new
 $poRegister = Register.new
 $poAuxiliar = Auxiliar.new
 $poAcesso = ControleDeAcesso.new
end

When(/^I access the webMCTest$/) do
 visit 'http://177.104.60.16:8000'
 puts page.current_url
 page.save_screenshot('screenshot.png')
end
```

```
Then(/^I check if every menus are present$/) do
 $poHome.checkHeader
end

And(/^I click on "([^"]*)" option$/) do |option|
 $poHome.clickMenu(option)
end

Then(/^I should see the "([^"]*)" page$/) do |option|
 $poHome.checkPage(option)
end

And(/^I fill the user and password with "([^"]*)" informations$/) do |option|
 $poLogin.loginUser(option)
end

And(/^I click on login button$/) do
 $poLogin.clickLogin
end

Then(/^I click on logout button$/) do
 $poLogin.clickLogout
end

Then(/^I verify if I am logged$/) do
 $poLogin.checkIfIamlogged
end

And(/^I verify I am unloged$/) do
 $poLogin.checkIfIamUnloged
end

And(/^I fill the email field$/) do
 $poRegister.fillMail
end

And(/^I fill the first and last name$/) do
 $poRegister.fillName
end

And(/^I fill the password and his confirmation "([^"]*)" requirements with "([^"]*)" keys$/)
do |option, keys|
 $poRegister.fillPass(option, keys)
end

And(/^I click on signup button$/) do
 $poRegister.clickBtnRegister
end

Then(/^I see a message error that requires a longer password$/) do
 $poRegister.verifyShortPassErrorMessage
end

Then(/^I see a message error that requires two same passwords$/) do
 $poRegister.verifyDifferentsPassErrorMessage
end

And(/^I fill the email field with a user already registered$/) do
$poRegister.fillUserAlreadyRegistered
end

Then(/^I see a message error that requires an email not registered$/) do
 $poRegister.verifyUserAlreadyRegistered
end

And(/^I fill the email field with too many caracteres$/) do
 $poRegister.fillUserLongEmail
end

Then(/^I fill the email field with a user that not contains ufabc domain$/) do
 $poRegister.fillUserNotUfabcDomain
end
```

```
Then(/^I see a message error that requires a email with ufabc domain$/) do
 $poRegister.verifyUserNotUfabcDomain
end

Then(/^I only should see in the right menu the Entrar and Inscrever options$/) do
 $poAcesso.verifyAvailableOptionsForUnloggedUsers
end

Then(/^I verify that this user has no access privileges$/) do
 $poAcesso.verifyAvailableOptionsForUserWithoutProfile
end

And(/^I verify that the user can not do anything until it is included in a discipline by the
coordinator$/) do
 $poAcesso.verifyRestrictedPermission
end

Then(/^I verify students profile have ready only access$/) do
 $poAcesso.verifyRestrictedPermission
end

Then(/^I verify if profile have permission for create classes$/) do
 $poAcesso.verifyIfProfileCanCreateClasses
end

And(/^I verify if profile have permission for create questions$/) do
 $poAcesso.verifyIfProfileCanCreateQuestions
end

And(/^I verify if profile have permission for create exams$/) do
 $poAcesso.verifyIfProfileCanCreateExams
end

And(/^I verify if profile have permission for update disciplines$/) do
 $poAcesso.verifyIfProfileCanUpdateDisciplines
end

And(/^I verify if profile have permission for create and update topics$/) do
 $poAcesso.verifyIfProfileCanUpdateTopics
end

And(/^I verify if profile have permission for create and update institutes$/) do
 $poAcesso.verifyIfProfileCanUpdateAndCreateInstitutes
end

And(/^I verify if profile have permission for create and update courses$/) do
 $poAcesso.verifyIfProfileCanUpdateAndCreateCourses
end

And(/^I verify if profile have permission for access admin menu$/) do
 $poAcesso.verifyIfProfileCanAccessAdminMenu
end
```

```
class Home
include Capybara::DSL

def checkHeader
 within('#navbarSupportedContent') do
   raise "Missing Institutos option! " unless find('.nav-link', :text => 'Institutos')
   raise "Missing Institutos option! " unless find('.nav-link', :text => 'Cursos')
   raise "Missing Institutos option! " unless find('.nav-link', :text => 'Disciplinas')
   raise "Missing Institutos option! " unless find('.nav-link', :text => 'Entrar')
   raise "Missing Institutos option! " unless find('.nav-link', :text => 'Inscrever')
  end
 end

def clickMenu(option)
  case option
   when 'Entrar'
```

```
    $poLogin.clickEntrar
   when 'Inscrever'
    $poRegister.clickRegister
  end
 end

 def checkPage(option)
  case option
   when 'login'
    $poLogin.checkFields
   when 'register'
    $poRegister.checkFields
  end
 end
end
```

```
class Login
include Capybara::DSL

def clickEntrar
 within('#navbarSupportedContent') do
  login = find('.nav-link', :text => 'Entrar')
  login.click
 end
end

def checkFields
 within('div.card-body') do
  raise "Missing mail's label! " unless find('.input-group-prepend', text: 'Endereço de
email:')
  raise "Missing mail's field! " unless find('input#id_username')
  raise "Missing password's label! " unless find('label', text: 'Senha:')
  raise "Missing password's field! " unless find('input#id_password')
  raise "Missing login button!  " unless find('.btn.btn-primary', text: 'Entrar')
  raise "Missing forgot password button! " unless find('.btn.btn-warning', text: 'Perdeu a
senha?')
 end
end

def clickLogin
 within('div.card-body') do
  find('.btn.btn-primary').click
 end
end

def clickLogout
 within('#navbarSupportedContent') do
  find('.nav-link', text: 'Sair').click
 end
end

def loginUser(option)
 case option
 when 'coordenador'
  userData = $poAuxiliar.readCredentials('coordenador')
 when 'professor'
  userData = $poAuxiliar.readCredentials('professor')
 when 'estudante'
  userData = $poAuxiliar.readCredentials('estudante')
 when 'administrador'
  userData = $poAuxiliar.readCredentials('administrador')
 when 'usuário sem privilégios'
  userData = $poAuxiliar.readCredentials('usuário sem privilégios')
 end
 within('div.card-body') do
  @user = userData['login']
  password = userData['password']
  find('input#id_username').send_keys @user
```

```
  find('input#id_password').send_keys password
 end
end

def checkIfIamlogged
 user = @user.split('@')[0]
 within('#navbarSupportedContent') do
  raise unless find('a.nav-link', text: user)
 end
end

def checkIfIamUnloged
 user = @user.split('@')[0]
 begin
  within('#navbarSupportedContent') do
   raise "You are logged!"  if find('a.nav-link', text: user)
  end
 rescue
  puts "You are unloged!"
 end
end
end
```

```
class Register
include Capybara::DSL

def clickRegister
 within('#navbarSupportedContent') do
  register = find('.nav-link', text: 'Inscrever')
  register.click
 end
end

def checkFields
 within('div.col-sm-10') do
  raise "Could not find correct page's title! " if find('h2').text != 'Inscrever'
  raise unless find('label', text: 'Email')
  raise unless find('input#id_email')
  raise unless find('label', text: 'Primeiro nome:')
  raise unless find('input#id_first_name')
  raise unless find('label', text: 'Último nome:')
  raise unless find('input#id_last_name')
  raise unless find('label', text: 'Senha:')
  raise unless find('input#id_password1')
  raise unless find('label', text: 'Confirmação de senha:')
  raise unless find('input#id_password2')
 end
end

def checkExceptions
 within('div.col-sm-10') do
  find('button', text: 'Sign up').click
 end
end

def fillMail
 email = find('input#id_email')
 email.send_keys $poAuxiliar.generateNewMail
end

def fillName
 first_name = find('input#id_first_name')
 last_name = find('input#id_last_name')
 first_name.send_keys 'Teste'
 last_name.send_keys 'Register ' + $poAuxiliar.generateNewDate
end

def fillPass(option, keys)
```

```
 if keys == "same"
  case option
  when "unfollowing"
   password = 'ana1234'
   passConfirmation = 'ana1234'
  when "following"
   password = 'ana12345'
   passConfirmation = 'ana12345'
  end
 elsif keys == "differents"
  case option
  when "following"
   password = 'ana12346'
   passConfirmation = 'ana12345'
  when "unfollowing"
   password = 'ana123'
   passConfirmation = 'ana124'
  end
 end
 pass = find('input#id_password1')
 pass_confirmation = find('input#id_password2')
 pass.send_keys password
 pass_confirmation.send_keys passConfirmation
end

def clickBtnRegister
 button = find('.btn.btn-primary')
 button.click
end

def verifyShortPassErrorMessage
 begin
  find('p', text: "Ela precisa conter pelo menos 8 caracteres")
 rescue Exception => e
  raise "Could not find correct message error! " + e.to_s
 end
end

def verifyDifferentsPassErrorMessage
 begin
  find('p', text: "Os dois campos de senha não combinam.")
 rescue Exception => e
  raise "Could not find correct message error! " + e.to_s
 end
end

def fillUserAlreadyRegistered
 userData = $poAuxiliar.readCredentials('coordenador')
 email = find('input#id_email')
 @user = userData['login']
 email.send_keys @user
 sleep 5
end

def verifyUserAlreadyRegistered
 begin
  find('p', text: "Usuário com este Endereço de email já existe.")
 rescue Exception => e
  raise "Could not find correct message error! " + e.to_s
 end
end

def fillUserLongEmail
 email = find('input#id_email')
 email.send_keys $poAuxiliar.generateTooLongMail
end

def fillUserNotUfabcDomain
 email = find('input#id_email')
 email.send_keys 'ana@gmail.com'
end

def verifyUserNotUfabcDomain
```

```
 begin
  find('div#info', text: 'Make sure email is intitutional:')
 rescue Exception => e
  raise "Could not finde correct message error! " + e.to_s
 end
end
end
```

```
class ControleDeAcesso
include Capybara::DSL

def verifyAvailableOptionsForUnloggedUsers
 within('div#navbarSupportedContent') do
  find('.nav-link', text: 'Entrar')
  find('.nav-link', text: 'Inscrever')
 end
end

def verifyAvailableOptionsForUserWithoutProfile
 within('div#navbarSupportedContent') do
  find('.nav-link', text: 'Questões')
  find('.nav-link', text: 'Turmas')
  find('.nav-link', text: 'Exames')
  find('.nav-link', text: 'Sair')
  find('.nav-link', text: 'Inscrever')
 end
end

def verifyRestrictedPermission
 find('.nav-link', text: 'Questões').click
 find('.card-title', text: 'Minha Lista de Questões')
 find('.card-body', text: 'Entre em contato com o coordenador da disciplina')

 find('.nav-link', text: 'Turmas').click
 find('.card-title', text: 'Listar Turmas')
 find('.card-body', text: 'Somente professores cadastrados em disciplinas')
 find('.card-body', text: 'Entre em contato com o coordenador da disciplina')

 find('.nav-link', text: 'Exames').click
 find('.card-title', text: 'Minha lista de Exames')
 find('.card-body', text: 'Somente professores cadastrados em discipinas e em turmas')
 find('.card-body', text: 'Entre em contato com o coordenador da disciplina')
end

def verifyIfProfileCanCreateClasses
 find('.nav-link', text: 'Turmas').click
 find('a.btn.btn-primary').click
 raise "No permission to create classes" if page.current_url !=
'http://177.104.60.16:8000/course/classroom/create/'
end

def verifyIfProfileCanCreateQuestions
 all('.nav-link', text: 'Questões')[1].click
 find('a.btn.btn-primary').click
 raise "No permission to create questions" if page.current_url !=
'http://177.104.60.16:8000/topic/question/create/'
end

def verifyIfProfileCanCreateExams
 begin
  find('.nav-link', text: 'Exames').click
 rescue
  all('.nav-link', text: 'Exames')[0].click
 end
 find('a.btn.btn-primary').click
 raise "No permission to create questions" if page.current_url !=
'http://177.104.60.16:8000/exam/exam/create/'
end
```

```
def verifyIfProfileCanUpdateDisciplines
 find('.nav-link', text: 'Disciplinas').click
 find('.btn.btn-outline-primary.btn-sm').click
 raise "No permission to update disciplines" if page.current_url !=
'http://177.104.60.16:8000/course/discipline/1/update'
end

def verifyIfProfileCanUpdateTopics
 find('.nav-link', text: 'Tópicos').click
 find('.btn.btn-outline-primary.btn-sm').click
 raise "No permission to update topics" if page.current_url !=
'http://177.104.60.16:8000/topic/topic/1/update'
 find('.btn.btn-outline-primary').click
 find('.btn.btn-primary').click
 raise "No permission to create topics" if page.current_url !=
'http://177.104.60.16:8000/topic/topic/create/'
end

def verifyIfProfileCanUpdateAndCreateInstitutes
 find('.nav-link', text: 'Institutos').click
 find('.btn.btn-outline-primary.btn-sm').click
 raise "No permission to update institutes" if page.current_url !=
'http://177.104.60.16:8000/course/institute/1/update'
 find('.btn.btn-outline-primary').click
 find('.btn.btn-primary').click
 raise "No permission to create institutes" if page.current_url !=
'http://177.104.60.16:8000/course/institute/create/'
end

def verifyIfProfileCanUpdateAndCreateCourses
 find('.nav-link', text: 'Cursos').click
 find('.btn.btn-outline-primary.btn-sm').click
 raise "No permission to update courses" if page.current_url !=
'http://177.104.60.16:8000/course/course/1/update'
 find('.btn.btn-outline-primary').click
 find('.btn.btn-primary').click
 raise "No permission to create courses" if page.current_url !=
'http://177.104.60.16:8000/course/course/create/'
end

def verifyIfProfileCanAccessAdminMenu
 find('.nav-link', text: 'Admin').click
 raise "No permission to access Admin menu" if page.current_url !=
'http://177.104.60.16:8000/admin/'
end
end
```

```
class Auxiliar
def readCredentials(userType)
 filepath = "./features/step_definitions/data/credentials.json"
 file = File.read(filepath)
 data_hash = JSON.parse(file)
 case userType
 when 'coordenador'
  data_hash['coordenador']
 when 'professor'
  data_hash['professor']
 when 'estudante'
  data_hash['estudante']
 when 'administrador'
  data_hash['administrador']
 when 'usuário sem privilégios'
  data_hash['usuário sem privilégios']
 end
end

def generateNewDate
```

```
 tdaydate = DateTime.now.to_date.to_s
 return tdaydate
end

def generateNewTime
 time = Time.new
 timeNow = time.to_s.split(' ')[1].gsub(':', '')
 puts timeNow
 return timeNow
end

def generateNewMail
 emailpadrao = 'test+@ufabc.edu.br'
 newemail = emailpadrao.split('@')[0] + generateNewTime + '@' + emailpadrao.split('@')[1]
 puts newemail
 return newemail
end

def generateTooLongMail
 emailpadrao = 'test.registers+@ufabc.edu.br'
 newemail = emailpadrao.split('@')[0] + generateNewDate + '@' + emailpadrao.split('@')[1]
 puts newemail
 return newemail
end
end
```