Universidade Federal
do Rio de Janeiro
Escola Politécnica

AMBIENTE ÁGIL DE TESTES AUTOMÁTICOS EM NUVEM PARA OS SISTEMAS WEB FENCE DOS EXPERIMENTOS DO CERN

Leandro Domingues Macedo Alves

Projeto de Graduação apresentado ao Curso de Engenharia de Controle e Automação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientadores: Carmen Lucia Lodi Maidantchik
Flávio Luis de Mello

Rio de Janeiro
Outubro de 2019

AMBIENTE ÁGIL DE TESTES AUTOMÁTICOS EM NUVEM PARA OS SISTEMAS WEB FENCE DOS EXPERIMENTOS DO CERN

Leandro Domingues Macedo Alves

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE CONTROLE E AUTOMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE CONTROLE E AUTOMAÇÃO.

Examinado por:

Carmen Lucia Lodi Maidantchik, D.Sc. - Orientadora

Flávio Luis de Mello, D.Sc. - Orientador

Jomar Gozzi, D.Sc.

Carlos José Ribas D'Avila, M.Sc.

RIO DE JANEIRO, RJ – BRASIL
OUTUBRO DE 2019

Domingues Macedo Alves, Leandro

Ambiente ágil de testes automáticos em nuvem para os sistemas web FENCE dos experimentos do CERN/Leandro Domingues Macedo Alves. – Rio de Janeiro: UFRJ/ Escola Politécnica, 2019.

X, 77 p.: il.; 29, 7cm.

Orientadores: Carmen Lucia Lodi Maidantchik
Flávio Luis de Mello

Projeto de Graduação – UFRJ/ Escola Politécnica/ Curso de Engenharia de Controle e Automação, 2019.

Referências Bibliográficas: p. 73 – 77.

1. Sistemas *Web*. 2. Testes automáticos. 3. Integração Contínua. 4. *CERN*. I. Lucia Lodi Maidantchik, Carmen *et al.* II. Universidade Federal do Rio de Janeiro, Escola Politécnica, Curso de Engenharia de Controle e Automação. III. Título.

Agradecimentos

Ao achado da minha caminhada, Bárbara, por ter mudado a minha vida. Obrigado por ser uma pessoa tão especial e inspiradora, pelas aventuras inesquecíveis, pelo conforto nas dificuldades, pelo sorriso maravilhoso e por me fazer uma pessoa tão feliz. Este trabalho não teria sido possível sem a sua participação e paciência.

Aos meus pais, José e Leila, por todos os sacrifícios e por sempre me priorizarem em absolutamente todos os momentos. Me sinto extremamente privilegiado em tê-los como pais, e palavras não são capazes de exprimir o quanto sou agradecido pelo amor provido.

Aos meus grandes amigos do Croácia, o grupo que me acolheu e que levarei para o resto da minha vida. Obrigado por estarem sempre presentes nos melhores e nos piores momentos.

Ao Breno, Bruno, Gabriel F., Gabriel L., Gabriela, Gustavo, Marcelo, Mario, Michelly e Varlen, obrigado por serem a família que tive e que tenho em Genebra. Obrigado pela imensa contribuições de cada um de vocês neste trabalho, direta ou indiretamente. Certamente não teria sido o mesmo sem vocês.

Aos colegas de turma de ECA, pelo companheirismo e amizade que foram fundamentais durante a faculdade.

A todos que contribuíram e incentivaram a sempre buscar o meu melhor, desde o Santo Agostinho até o CERN. Agradeço à Carmen por acreditar em mim quando poucos acreditavam, pelo apoio constante e por ter proporcionado oportunidades as quais nunca esquecerei.

Resumo do Projeto de Graduação apresentado à Escola Politécnica/ UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Controle e Automação.

AMBIENTE ÁGIL DE TESTES AUTOMÁTICOS EM NUVEM PARA OS SISTEMAS WEB FENCE DOS EXPERIMENTOS DO CERN

Leandro Domingues Macedo Alves

Outubro/2019

Orientadores: Carmen Lucia Lodi Maidantchik
Flávio Luis de Mello

Curso: Engenharia de Controle e Automação

Atualmente o grupo de desenvolvedores da colaboração entre UFRJ e *CERN* disponibiliza mais de 30 sistemas web para atender mais de 2 mil usuários por mês. É necessária a garantia do funcionamento correto destes sistemas, e o tema deste projeto de conclusão de curso está relacionado à implementação de uma plataforma de testes automáticos para a verificação destes sistemas.

Em linhas gerais, objetiva-se o desenvolvimento de abstrações para facilitar a escrita de testes automáticos, juntamente com uma solução em integração contínua para administrar a execução destes testes. Estas abstrações por sua vez foram desenvolvidas para escrita de testes de unidade e testes de ponta a ponta, oferecendo funcionalidades para verificar os sistemas que compartilham o *framework Fence*, desenvolvido pelo grupo para facilitar a implementação de interfaces web. A implementação da integração contínua para estes testes permitiu a execução automática de mais de 150 mil testes e mais de 800 inconsistências foram encontradas antecipadamente, motivando a escrita de mais testes utilizando a plataforma.

Abstract of Undergraduate Project presented to POLI/UFRJ as a partial fulfillment of the requirements for the degree of Engineer.

Leandro Domingues Macedo Alves

October/2019

Advisors: Carmen Lucia Lodi Maidantchik
Flávio Luis de Mello

Course: Automation and Control Engineering

Currently, the group of developers originated from a collaboration between UFRJ and CERN provides more than 30 web systems, which receive over 2 thousand users per month. It's important to ensure the proper behavior of these systems and the subject of this undergraduate thesis is related to the development of an automated testing platform for verification of these systems.

In general, the goal is to develop utilities that facilitate the coding of automatic tests along with a continuously integrated solution to manage their execution. These utilities were developed to help writing unit and end-to-end tests, offering built-in tools to verify systems built using the Fence framework, a solution created by the group for high configurable web interface generation. The development of continuous integration for these tests allowed the automatic execution of more than 150 thousand tests and more than 800 inconsistencies were found in advance, motivating the writing of more tests using the platform.

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 Tema

Este trabalho de conclusão de curso está relacionado ao projeto e desenvolvimento de uma plataforma ágil para implementação e gerenciamento de testes automáticos para sistemas web que fazem uso do *framework Fence*, implementado pelo grupo de desenvolvedores da UFRJ no *CERN*. O problema a ser resolvido é garantir a confiabilidade destes sistemas, tornando-os mais estáveis, seguros, robustos e isentos de falhas.

Esta proposta também otimiza o processo de trabalho do grupo, uma vez que as tarefas que até então eram feitas de forma manual, passaram a ser padronizadas e automáticas. O processo de trabalho também é revisto sob a perspectiva de metodologias ágeis.

1.2 Justificativa

Até o início deste trabalho, todos os testes em sistemas eram realizados de forma manual pelo grupo de desenvolvedores da UFRJ. Estes sistemas web fazem parte da colaboração internacional entre a UFRJ e o *CERN* para gerenciar publicações, equipamentos e pessoas. A validação e a verificação destes softwares fazem uso de planos de testes, que contêm as regras dos sistemas para serem testadas pelos desenvolvedores. Para a garantia de qualidade do software, é necessário que estas verificações sejam executadas frequentemente, o que pode se tornar algo exaustivo e consumir uma quantidade significativa de tempo. Dessa forma, almejou-se otimizar o processo de verificação, com a implementação de testes automáticos para estes sistemas. O principal objetivo destes testes é garantir o comportamento esperado minimizando a necessidade de intervenção humana para verificar cada etapa. Assim que codificados, estes testes podem ser executados de maneira contínua e bem

estruturada, comparando os resultados esperados com os obtidos. Eles podem periodicamente verificar a condição de um sistema web assegurando suas funções, desde que cobertas pelos próprios testes.

1.3 Objetivo

O propósito geral deste trabalho é projetar e desenvolver uma plataforma de testes que garanta o funcionamento correto de um sistema web gerado através do *framework Fence*, conforme os requisitos do software. De forma mais detalhada, objetiva-se a disponibilização de dois utilitários para auxiliar na escrita de testes de unidade de código e testes amplos de interface de usuário, e a implementação de um ambiente automático para controle dos testes implantados.

1.4 Metodologia

O início deste trabalho se deu com um embasamento teórico a partir de artigos e livros no assunto de testes de software. Depois foi proposta a implementação de um protótipo baseado no conceito de testes de interface. O objetivo deste protótipo estava na realização de testes ponta a ponta básicos, popularmente conhecidos de testes *end-to-end*, os quais realizavam comandos simples no navegador web, como por exemplo a verificação se uma interface de busca retorna o que foi requisitado. Tais testes foram escritos utilizando um interpretador de código capaz de realizar as respectivas ações no navegador.

Após esta prova de conceito, foi analisado o impacto que os testes automáticos teriam na qualidade dos sistemas providos pelo grupo, e quais seriam os desafios em acomodá-los no processo de trabalho dos desenvolvedores. Dessa forma, a proposta do projeto foi expandida para a implementação de uma plataforma capaz de fornecer ferramentas necessárias para o desenvolvedor implantar testes unitários e testes *end-to-end* com mais facilidade. Esta plataforma também teria que estar apta a administrar estes testes de forma automática, executando-os conforme o ciclo de desenvolvimento.

Aperfeiçoando o protótipo inicial para incrementar mais funcionalidade, foi desenvolvido um *framework* voltado para a construção e compartilhamento de testes *end-to-end*. A ideia principal destes testes é verificar o sistema sob a perspectiva dos usuários, comparando os resultados obtidos diretamente com os esperados pelos requisitos. Com estes resultados, os desenvolvedores podem avaliar a necessidade de alterações antes de entregar uma nova versão do sistema em produção.

Posteriormente foi desenvolvido um segundo *framework* que se torna o responsável pela construção de testes unitários de código contido no servidor. Diferentemente

dos testes *end-to-end*, os testes unitários realizam verificações a um nível modular e isolado, conferindo o comportamento de pequenos trechos de código. Este tipo de teste auxilia em encontrar problemas logo no início do ciclo de desenvolvimento e também permite ao programador reescrever o código do sistema posteriormente, garantindo que o módulo ainda funcione corretamente.

A terceira parte do trabalho foca no desenvolvimento de um ambiente baseado em sistemas distribuídos para execução destes testes. Fazendo uso de máquinas virtuais disponibilizadas pelo *CERN*, foi implementado um ambiente de integração e entrega contínua de software. Quando o desenvolvedor prepara uma nova versão de um sistema, uma série de tarefas são automaticamente executadas na tentativa de certificar que problemas não ocorrerão no ambiente de produção. Estas tarefas incluem compilação, testes unitários, testes *end-to-end* e simulação de instalação do software em ambiente de produção.

Durante a evolução deste trabalho foram agregados conhecimentos sobre práticas ágeis que poderiam contribuir para o processo de desenvolvimento de software do grupo de desenvolvedores. Ferramentas e rotinas foram adotadas, com o intuito de melhorar a comunicação interna, aproximar os usuários e prover ciclos curtos de entrega.

É importante citar também a colaboração de outros integrantes do grupo que possibilitaram a expansão deste trabalho. O processo de entendimento dos problemas contou principalmente com a participação de programadores que desenvolvem sistemas para as colaborações *ATLAS*, *LHCb* e *ALICE*. Reuniões recorrentes foram realizadas a fim de entender com profundidade o problema e discutir as diferentes estratégias disponíveis para tentar solucioná-lo. Estas pessoas também contribuíram para o desenvolvimento da plataforma deste trabalho, assim como na implantação de testes automáticos.

O projeto também envolveu profissionais e estudantes da UFRJ sendo utilizada uma comunicação principalmente através de e-mails, reuniões remotas por conferência, ferramenta de mensagens *Slack* e pelo sistema de gerenciamento de software, *Jira*.

1.5 Descrição

O conteúdo deste documento está dividido em 6 capítulos. O capítulo atual descreve o tema do trabalho, o problema no qual ele se aplica, o seu objetivo e a metodologia por trás dele.

Uma breve contextualização é feita no capítulo 2, explicando a conjuntura da colaboração com o *CERN* e as características específicas deste ambiente que motivaram esta dissertação.

O capítulo 3 trata da fundamentação teórica, apresentando principalmente a teoria de testes e os autores que influenciaram a elaboração desta atividade.

As especificações da solução proposta são definidas no capítulo 4. É aprofundada a explicação da arquitetura da plataforma desenvolvida, explicando o funcionamento, suas tecnologias e casos de uso. Por fim é falado sobre a solução de integração e de entrega contínua adotada.

No capítulo 5 são apresentados os resultados obtidos com a implantação da plataforma para projetar, implementar e integrar testes automáticos no processo de trabalho. O capítulo 6 é reservado à conclusão.

Capítulo 2

Contextualização

Este capítulo aborda o ambiente no qual este projeto foi desenvolvido. Será apresentado o centro de pesquisas *CERN*, explicando sobre sua estrutura, história e objetivos. Após isso será detalhado sobre o grupo de estudantes da UFRJ que desenvolvem sistemas web para a colaboração, e sobre o *framework Fence*, que é utilizado para implementação destes sistemas.

2.1 Centro de pesquisa CERN

Levando-se em consideração a análise retrospectiva feita por James Gillies [4], no fim da década de 40 a Europa passava por um período de reestruturação. A necessidade de se recuperar em um cenário pós segunda guerra mundial fez com que crescesse o interesse colaborativo entre as nações europeias em diversas áreas, como a ciência. A retomada da pesquisa fundamental era de enorme importância para a comunidade científica, e com isso, em 1949 foi feita a primeira proposta oficial da formação de uma organização científica europeia. Discussões subsequentes se deram nos anos seguintes, até que em 1951 ocorre a primeira resolução sobre o estabelecimento de um conselho europeu voltado a pesquisa nuclear, ou abreviando-se, *CERN* (*Conseil Européen pour la Recherche Nucléaire*). Em 1954 o laboratório é oficialmente fundado próximo de Genebra, na Suíça.

De acordo com o relatório anual do *CERN* [5], o centro de pesquisa conta com a participação de mais de 17.900 pessoas ao redor do mundo. Mais de 12.500 cientistas contribuem dando assistência aos experimentos e na análise de dados, representando 110 nacionalidades presentes em institutos de mais de 70 países.

O objetivo principal do centro de pesquisa é compreender melhor o universo a partir do estudo da estrutura fundamental das partículas [6]. Partículas subatômicas colidem entre si próximas a velocidade da luz, gerando informações sobre como as partículas interagem e ajudando a melhor compreender as leis fundamentais da natureza. Estes experimentos se tornam possíveis graças à presença de aceleradores

de partículas e detectores, responsáveis respectivamente por impulsionar feixes de partículas a altas energias e por observar e coletar o resultado das colisões.

O Grande Colisor de Hádrons (*Large Hadron Collider*) é definido como o principal acelerador do *CERN* e o mais poderoso acelerador do mundo [6] [7]. O *LHC* possui dois tubos mantidos a vácuo em uma circunferência de 27 km nos quais os feixes de partículas viajam em direções opostas momentos antes da colisão. Tais feixes são então conduzidos a colidir em 4 pontos, onde ficam localizados os 4 detectores principais do *LHC*: *ATLAS*, *CMS*, *ALICE* e *LHCb*.

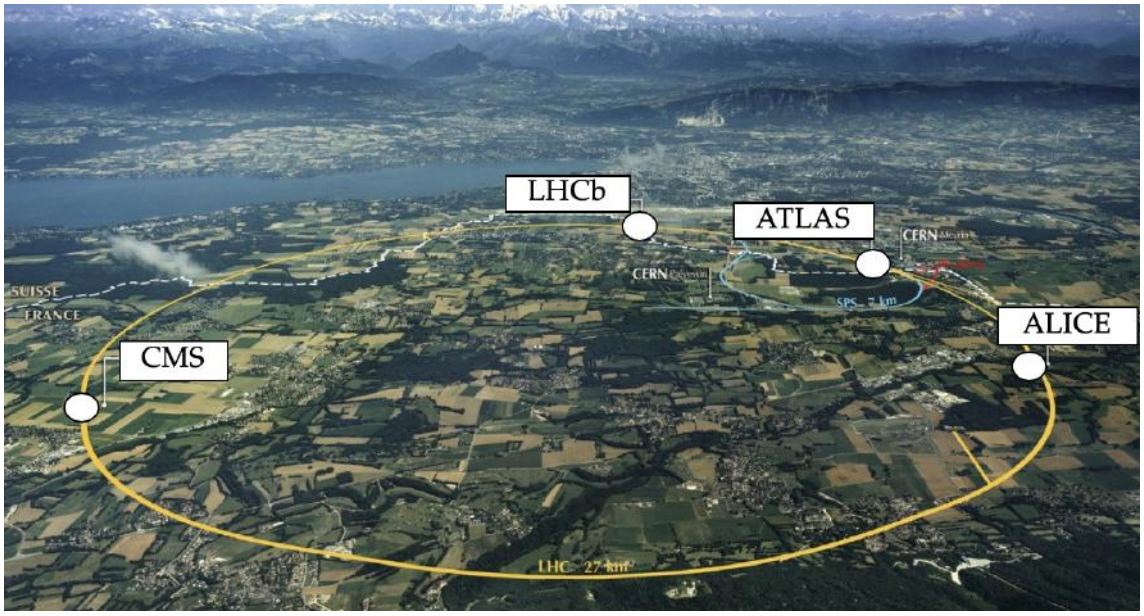

Figura 2.1: Visão topográfica do *CERN*.

O experimento *LHCb* (*Large Hadron Collider beauty*) tem como um dos principais objetivos tentar justificar o fato do universo ser aparentemente composto em quase sua totalidade de matéria, mas não de antimatéria [6]. O experimento pretende encontrar respostas com a investigação de pequenas diferenças entre as duas, a partir do estudo de uma partícula chamada *beauty quark*. O detector envolve cerca de 850 cientistas de 79 institutos de 18 países [8], e sua estrutura está ilustrada na 2.2. Foi implementado para este experimento o sistema *LHCb Membership*, para gestão de pessoas, institutos e listas de autores. Este sistema foi utilizado como referência para o primeiro protótipo de testes automáticos desenvolvido neste trabalho.

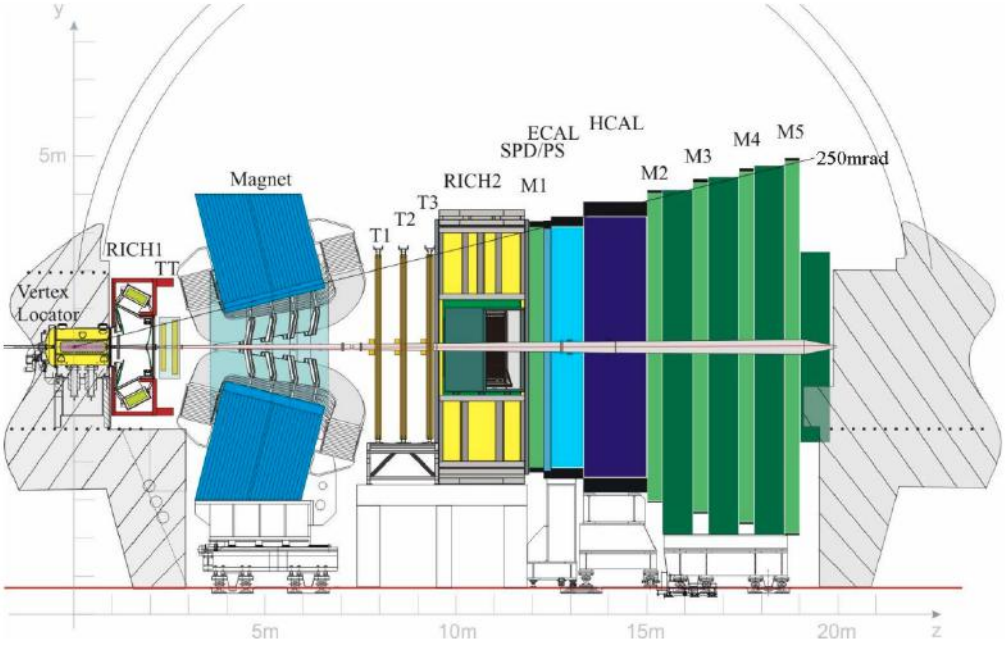

Figura 2.2: Estrutura do detector *LHCb* [1].

O foco do detector ATLAS (*A Toroidal LHC ApparatuS*) se concentra na tentativa de melhor entender os componentes fundamentais da matéria [9]. O detector investiga um vasto campo da física de partículas, desde o estudo das partículas subatômicas descritas no modelo padrão, até a busca por outras partículas, como por exemplo o bóson de Higgs e partículas que possam formar matéria escura. O experimento conta com 3000 autores científicos de 183 instituições presentes em 38 países [6], e foi constituído como apresentado na 2.3. Dos 22 sistemas desenvolvidos para este experimento, é valido destacar o sistema *ACES*, *Atlas Central Equipment System*, desenvolvido para gerenciamento de equipamentos e cabos, e planejamento de expansões no detector. Contando com mais de 150 mil equipamentos e 80 mil cabos, a complexidade deste sistema foi um motivacional para usá-lo como objeto de teste durante o desenvolvimento da plataforma, e atualmente este é o sistema que possui mais testes *end-to-end*.

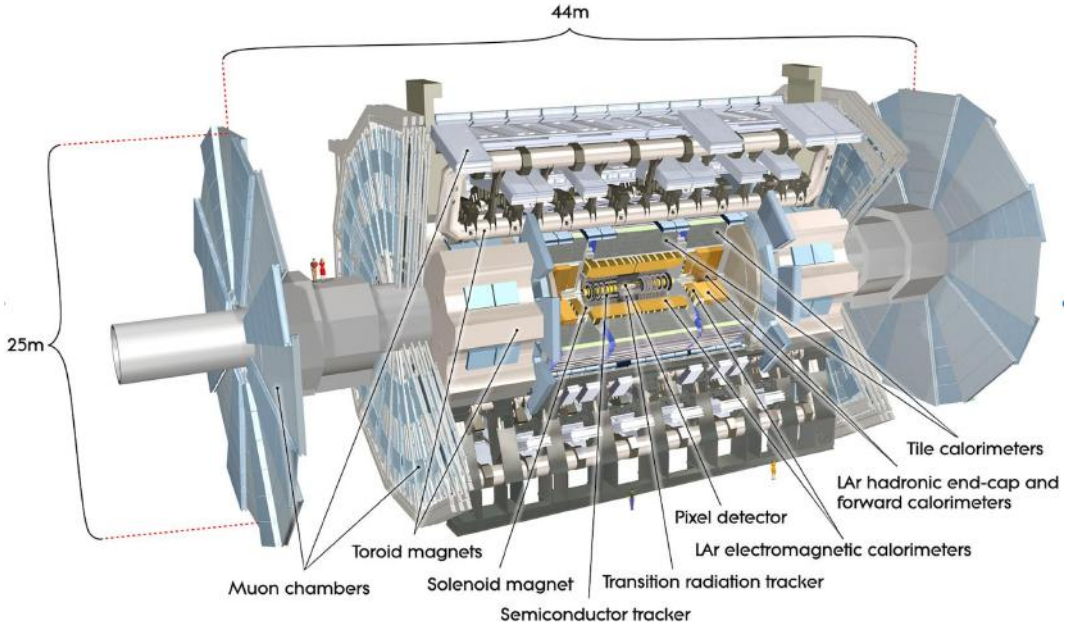

Figura 2.3: Estrutura do detector *ATLAS* [2].

Com a longevidade dos experimentos, as numerosas pesquisas realizadas pelo laboratório e a grande quantidade de pessoas envolvidas, um grande volume de dados vem sendo produzido desde as últimas décadas, e a complexidade exigida para o controle destes dados requer a assistência de softwares. É essencial para estes softwares possuírem fácil adaptação a mudanças, pois com os constantes aprimoramentos realizados nos experimentos, a grande rotatividade de contribuidores e a própria evolução do software, as necessidades de funcionalidades mudam com frequência. Isso se torna um forte motivacional para o investimento em testes de software, já que o dinamismo destes softwares exige verificações constantes quanto à confiabilidade oferecida.

Dentro do *CERN*, diversos grupos atualmente desenvolvem soluções automatizadas para testes. Um exemplo é o grupo *Joint Controls Project* que desenvolve softwares de sistemas de controle comuns aos 4 principais detectores. Juntamente com estes softwares, o grupo implementou um *framework* próprio de escrita de testes unitários para verificá-los. Outro exemplo é o desenvolvimento de testes ponta a ponta para a plataforma *EDH* que gerencia documentos eletrônicos do *CERN*, feito pelo grupo *Advanced Information Systems*. Ambos os casos as soluções apresentadas atingem propósitos específicos e em linguagens de programação distintas, não sendo soluções genéricas que atendam as necessidades de todos os grupos presentes no centro de pesquisa.

2.2 Grupo UFRJ e o framework Fence

A construção do acelerador *LHC* e respectivos detectores envolveu centenas de institutos ao redor do mundo, constituindo um importante trabalho cooperativo. Com o crescimento desta cooperação, diversos desafios passaram a existir a fim de garantir a qualidade da pesquisa científica produzida. O longo período de tempo levado para construção, teste e aprimoramento dos detectores e aceleradores ocasionou o armazenamento de informações utilizando diferentes tecnologias, métodos e terminologias. Acrescentando isso à descentralização de dados com a participação global de pesquisadores, fez com que a recuperação e compartilhamento de dados se tornasse gradativamente mais difícil com o tempo. Cada solução implementada atendia apenas necessidades específicas do próprio grupo na maior parte dos casos, dificultando a recuperação de dados heterogêneos e a integração de tais fontes de informação. [10].

No contexto da colaboração entre UFRJ e *ATLAS*, em 2003 um grupo de brasileiros propuseram um modelo para recuperar dados armazenados em repositórios independentemente de tecnologia, modelagem ou terminologia. Nomeado *Glance*, esse modelo tinha como objetivo proporcionar uma ferramenta amigável para o próprio usuário poder customizar as suas interfaces de busca e inserção de dados [11]. Esta ferramenta abstraía a complexidade do armazenamento de dados, tornando mais fácil para o usuário conseguir procurar e modificar a informação desejada. O paradigma *procedural* foi adotado na arquitetura desta ferramenta, que foi escrita nas linguagens *C++*, *php* e *Javascript*. Na época, 21 sistemas foram implementados a partir dele, atendendo além do próprio *ATLAS*, os experimentos *LHCb* e *ALICE*.

Com a expansão do uso do *Glance* pela crescente demanda de sistemas, novas oportunidades passaram a existir para os desenvolvedores [12]. O modelo de administração empregado no *CERN* e seus experimentos se baseia em uma alta rotatividade nos cargos gerenciais, e somando-se isso com os constantes avanços tecnológicos dos experimentos, a alteração de funcionalidades nos sistemas *Glance* passou a se tornar algo frequente. Uma nova administração pode ter uma expectativa distinta da anterior, assim como novos usuários podem necessitar novas soluções para atender demandas tecnológicas mais recentes.

Outra oportunidade observada foi a possibilidade de melhorar o compartilhamento de código entre os sistemas. Uma proposta como a adoção do paradigma de orientação a objetos e utilização de classes bases abstratas, contendo a funcionalidade comum, poderia reduzir a base de código total dos sistemas e facilitar a manutenção.

Tendo em vista a frequente mudança de requisitos pelos representantes da base de usuários e a evolução das tecnologias web, em 2014 foi proposto e implementado

pela UFRJ o *framework Frontend Engine for Glance*, ou simplesmente *Fence* [13]. O *Fence* tem como objetivo reunir um conjunto de classes e ferramentas para implementar e reutilizar as interfaces entre os sistemas. Sua proposta inicial era projetar uma camada entre a recuperação dos dados provenientes do *Glance* e a visualização destas informações, indo de acordo com uma série de regras estabelecidas por cada experimento. Escrito em *php* e fazendo uso extensivo de herança e polimorfismo com a orientação a objetos, o comportamento das interfaces geradas pelo *framework* é condicionado a arquivos de configuração *JSON*. Estes arquivos funcionam como agentes de controle, podendo especificar as regras de inserção de um formulário ou os critérios disponíveis de busca. Esta abordagem permite que os sistemas possam ser projetados ou adaptados às variações de seus requisitos com menor necessidade de escrita de código.

Atualmente existem 20 sistemas e subsistemas web que foram desenvolvidos a partir do *framework Fence* [14], conforme pode ser visto na figura 2.4. Tais sistemas são responsáveis por apoiar os principais aspectos da colaboração, como os membros dos experimentos, contratos, institutos, publicações, equipamentos e configurações dos detectores. Na figura, o sistema *ACES* para administração de equipamentos, criado a partir do *framework Fence*.

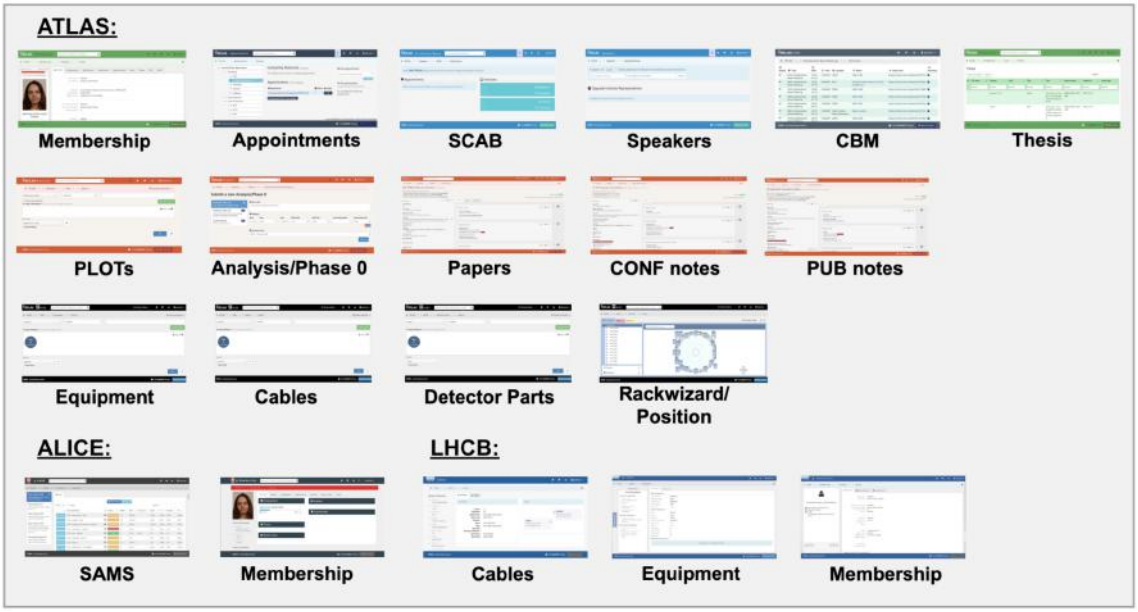

Figura 2.4: Sistemas *FENCE* atualmente desenvolvidos [3].

Capítulo 3

Fundamentação

Com a constante procura na melhoria no processo de desenvolvimento de software, o grupo se esforça em adotar gradativamente valores ágeis, com ciclos de entrega mais curtos e garantia da qualidade de entrega. Esse esforço foi iniciado com uma pesquisa na literatura, principalmente na teoria de testes. Uma visão geral do assunto é apresentada ao longo desta fundamentação, analisando o planejamento de testes e suas diversas categorias. A partir disso é feita uma divisão entre testes manuais e automáticos, enfatizando os testes automáticos. São discutidos com mais profundidade os testes de unidade e *end-to-end*, e por fim são estudadas as possibilidades existentes de ter um ambiente distribuído de execução de testes com integração contínua.

3.1 Aplicações web e metodologia ágil

Com o início da popularização da internet a partir da concepção do *World Wide Web* por Berners-Lee [15] no próprio *CERN*, o cenário de desenvolvimento e consumo de software passou por uma grande transformação. Navegadores web passaram a existir, cada vez mais poderosos e com cada vez mais capacidade de exibir conteúdo remoto prontamente. Dessa forma, tornou-se favorável o desenvolvimento de aplicações web.

Como salientado por Vora [16], aplicações web tornam extremamente fácil para usuários fazerem uso do software desenvolvido, sendo necessário apenas que tal software seja hospedado em um servidor capaz de suprir a demanda. A atualização do software também se torna mais rápida, pois uma vez implementada a atualização no servidor, é possível garantir uniformidade para todos que o acessam. Do lado dos desenvolvedores também se torna uma tarefa mais simples, já que na maior parte dos casos, os navegadores são os que assumem a responsabilidade de tornar compatível o conteúdo remoto para diferentes plataformas. Deste modo, o programador não precisa se preocupar se sua aplicação irá funcionar para um cliente com um am-

biente computacional específico, contanto que seja utilizado um navegador com uma versão suportada pelo software em questão. As aplicações web também permitem a redução em custos computacionais pois permite que a base de código seja dividida entre servidor e cliente, com o *back-end* sendo o código executado pelo servidor e o *front-end* possui o código que será processado nos navegadores dos usuários.

Por conta destas características, e do dinamismo presente no *CERN*, desde o início do desenvolvimento de sistemas *Glance* optou-se pela implementação de softwares em web. O resultado obtido com esta implementação foi positivo, e o crescimento sucessivo do número de sistemas proporcionou o aumento correspondente no número de desenvolvedores do grupo. A ampliação do grupo passou a motivar reflexões recentes sobre a forma na qual vem sendo desenvolvido software, e como aprimorar a comunicação. Estas reflexões levam em consideração o ambiente de desenvolvimento do grupo, marcado por alterações recorrentes de requisitos e urgência nas entregas. Isso evidencia a importância em adaptar continuamente o software a mudanças e com ciclos curtos de desenvolvimento, sendo estas características semelhantes com os valores e princípios empregados pelas metodologias ágeis existentes.

Apesar do destaque recente recebido pelo movimento ágil, este pensamento possui décadas de existência. Baseando-se na análise feita em Myers *et al.* [17], com o aumento da competição e interconectividade em todos os setores da economia durante os anos 90, as estratégias de negócios passaram a adotar soluções que reduzissem o tempo de lançamento de um produto sem afetar sua qualidade. Na área de software não foi diferente, e o processo de desenvolvimento tradicional não se mostrava tão satisfatório nesse novo ambiente competitivo, principalmente com a popularização da internet.

No início dos anos 2000, um grupo de desenvolvedores propôs um novo processo de trabalho, chamado de manifesto ágil. Esse documento continha uma nova filosofia, focada em clientes e funcionários ao invés de processos e hierarquias. O desenvolvimento ágil promove um processo interativo e incremental, centrado no cliente, incluindo mudanças durante o processo, e possuindo ciclos curtos de entrega. O método não é considerado eficiente se não obtiver a participação constante dos clientes durante o desenvolvimento. É importante a validação contínua destes clientes sobre a implementação progressiva do produto e se atende as expectativas.

Uma das abordagens de metodologia ágil mais adotadas na comunidade é a programação extrema, conhecida como *Extreme Programming* ou *XP*, e desenvolvida por Beck *et al.* [18]. Além da participação do usuário, o modelo *XP* depende fortemente de testes unitários e de aceitação, e sua adoção é recomendada principalmente em cenários sujeitos a mudanças frequentes de especificações e que possuem um canal acessível para comunicação constante. O planejamento foca em entregáveis, chamados *user stories* que definem os requisitos dos usuários. Estas *user stories*

podem então ser verificadas no final de um ciclo de entrega, por meio dos testes de aceitação do usuário. Dessa forma, se obtém retorno da opinião dos usuários mesmo durante o desenvolvimento, reduzindo as chances de um produto que não atenda as expectativas finais.

O desenvolvimento ágil também se estende a nível de código, pois é fundamental o esforço em escrever código que seja receptivo a mudanças, extensível, robusto e manutenível. A fim de alcançar estas metas um conjunto de princípios de *design*, abreviado *SOLID*, foi desenvolvido por Martin [19]. São cinco princípios teoricamente capazes de aprimorar a compreensão, manutenção e escalabilidade de código. Citando-os em ordem, o princípio da responsabilidade única define que um módulo deve ter apenas um motivo para mudar; o princípio do aberto-fechado determina que as entidades de software devem ser abertas para ampliação mas fechadas para modificação; o princípio da substituição de *Liskov* diz que os subtipos devem ser substituíveis pelos seus tipos de base; o princípio da segregação de interfaces defende o desenvolvimento de interfaces refinadas que são específicas para o cliente; e finalmente o princípio da inversão de dependências estabelece que só se deve depender de abstrações, se desvencilhando de classes concretas. Estes princípios são amplamente utilizados pela comunidade no desenvolvimento de aplicações em orientação a objetos.

Outro conceito importante é o do teste ágil, mais conhecido como *agile testing* [17]. O teste ágil é uma forma colaborativa de teste, na qual desenvolvedores, testadores e clientes fazem parte do planejamento, implementação e execução do plano de testes. Os clientes contribuem para os testes de aceitação com casos de uso e expectativas de funcionalidade, enquanto os desenvolvedores colaboram com os testadores implementando o ambiente de testes e automatizando-os quando possível. Como nas metodologias ágeis os ciclos de desenvolvimento são curtos, o tempo se torna um recurso valioso, e por este motivo a filosofia do teste ágil apoia a implantação de testes automáticos. Este foi um dos motivacionais para o aprofundamento deste trabalho em testes automáticos.

3.2 Testes de software

Este subcapítulo foca na teoria de testes. Testes de software fazem parte do processo de qualidade de software, e avaliam atributos como funcionalidade, confiabilidade e usabilidade. Para minimizar o risco de usuários experienciarem dificuldades em um sistema, é fundamental a realização de testes.

Inicialmente são discutidos os principais conceitos de testes de software, assim como a motivação para implementá-los. Em seguida, é abordado o conceito do plano de testes e as principais classificações existentes, dissertando sobre as diferenças entre

testes manuais e automáticos.

3.2.1 Conceito de testes

"Teste de Software é um processo ou uma série de processos feitos a fim de garantir que o programa está se comportando da forma esperada quando foi projetado, e nada além disso". Esta é a definição utilizada por Myers *et al.* [17]. De acordo com Myers, o software tem que ser previsível e consistente, não apresentando nenhuma surpresa aos usuários. O ato de testar consiste então em "executar um programa com o propósito de encontrar erros".

Idealmente seria possível testar todas as possíveis permutações de um programa, porém isto é inviável na prática. Até mesmo um simples programa pode conter milhares de combinações de entradas e saídas, e testar cada caso exigiria um número excessivo de recursos. É necessário que o desenvolvedor de testes tenha uma visão crítica sobre o que é prioritário a ser testado em seu programa.

Também mencionado por Myers e reforçado por Leventhal *et al.* [20], há o fator psicológico envolvido no processo de testes. O objetivo do teste é demonstrar que o programa não possua erros, logo pode ocorrer um viés durante a implementação dos testes a fim de alcançá-lo. Isso pode ocorrer por exemplo na escolha de dados que serão utilizados para o teste, de forma que tais dados possuam menor probabilidade de causar erros durante a execução do programa. Portanto um teste que encontra um erro deve ser visto de forma positiva.

Outro aspecto também importante para o teste de software é o econômico. Desde décadas atrás, conforme pode ser visto em Tarek Abdel-Hamid [21], há a preocupação em encontrar o equilíbrio entre os custos de controle de qualidade e os benefícios econômicos trazidos por ela. Estratégias como testes de exaustão de entrada de dados ou de trajetórias em um software já se provaram pouco eficazes visto o consumo de tempo necessário em implementá-las. A ideia se torna, portanto, avaliar qual subconjunto de todos os casos de teste possíveis tem a maior probabilidade de detectar a maioria dos erros. Sob esta avaliação se torna economicamente mais viável a realização de testes de software.

De forma resumida, Myers *et al.* [17] sugerem os princípios a seguir para o desenvolvimento de testes:

1. Uma parte fundamental para o caso de teste é a definição do resultado esperado.

2. Testes manuais devem ser realizados preferencialmente por uma outra pessoa que não seja o autor.

3. Da mesma forma que casos de testes devem conter condições de entradas válidas e esperadas, é importante também conter condições inválidas e inesperadas.

4. Não é suficiente apenas examinar se o programa está fazendo o que deveria ser feito. É fundamental também verificar o que está sendo feito a mais que não deveria ser feito.

5. Jamais planejar um teste sob a suposição de que nenhum erro será encontrado.

6. A probabilidade da existência de mais erros em uma seção de um programa é empiricamente proporcional ao número de erros já encontrados nessa seção.

Um conceito adicional para a teoria de testes são as definições de validação e verificação. A validação se preocupa em certificar que o sistema atenderá às necessidades reais do cliente, enquanto a verificação se responsabiliza em garantir que o sistema esteja bem projetado e isento de erros [22]. Tanto verificações como validações precisam ser apropriadamente documentadas, como em um plano de testes.

3.2.2 Motivação para testes

Com a implementação de sistemas utilizando o padrão de interfaces com o *Fence*, obteve-se a oportunidade de analisar outras etapas presentes no processo de desenvolvimento de software, na possibilidade de aprimorar alguma delas. Nesta análise, constatou-se que a realização de testes manuais pelos desenvolvedores consumia um tempo considerável durante a rotina de desenvolvimento, e a possibilidade de uso de testes automáticos poderia otimizar este processo.

Antes de realizar o teste, é necessário planejá-lo com a documentação em plano de testes, tornando possível organizar e priorizar o que será testado [23]. O desenvolvedor, em seu ambiente de desenvolvimento, segue uma série de instruções explicitadas neste plano de testes a fim de garantir a estabilidade do sistema após suas modificações. É comum estas verificações serem feitas apenas com testes manuais para sistemas pequenos ou de pouca longevidade, pois em certos casos não ocorre compensação com o esforço adicional da implementação de testes automáticos. Entretanto para projetos maiores ou de longa duração, a verificação recorrente com somente testes manuais torna-se onerosa e inviável. A exemplo de código orientado a objetos, uma pequena correção de poucos minutos de uma classe pai pode significar horas de testes manuais em diversos sistemas que estendem esta classe.

Como consequência da prática de testes manuais, acumulado com a comum urgência de entrega rápida requisitadas pelas partes interessadas, os *stakeholders*, e

os ciclos ágeis de curta duração, torna-se difícil na prática realizar todos os procedimentos de controle de qualidade necessários com tempo limitado sem uma estratégia adequada. Softwares estão sendo constantemente aprimorados e aumentando sua complexidade, o que torna testá-los um desafio cada vez maior. Isso leva ao aumento no risco de anomalias que podem ser sentidas pelos usuários. Um possível erro adicionado de forma involuntária pelo desenvolvedor exige uma subsequente submissão de código para corrigi-lo, que por sua vez também traz riscos se não for devidamente testada.

Outra motivação está no impacto causado pelas atualizações de software. À medida que um software está estável e em produção durante um longo período de tempo, um dos desafios do desenvolvedor passa a ser a implementação de novas funcionalidades e o conserto de anomalias [24]. Interações com o programa podem se tornar algo desafiador, visto que inserções ou modificações em linhas de código podem gerar comportamentos não intencionais, como o surgimento de novos erros. Isto por sua vez pode desencorajar o programador a fazer modificações, que em alguns casos, para evitar riscos, optam por estratégias de intervenção minimalista em código, alterando o mínimo necessário para implementar a mudança ao invés de adotar uma solução mais correta [25]. Essa estratégia por sua vez se mostra não escalável, podendo prejudicar a legibilidade do código. O desenvolvedor pode ter mais segurança em codificar se existe a garantia que uma mudança mais significativa em código não cause efeitos colaterais nos sistemas, e os testes automáticos de regressão podem contribuir para esta garantia.

É importante considerar também o efeito psicológico que a repetição de tarefas pode ocasionar no desempenho dos testes manuais [26]. Os testadores podem ficar fatigados em realizar a mesma sequência de verificações continuamente, o que pode prejudicar o foco durante a comparação dos resultados obtidos com os esperados. Isso por fim propicia à ocorrência de falhas humanas e podem comprometer a qualidade do teste.

Portanto, como forma de tentar solucionar tais problemas, a comunidade vem tentando encontrar alternativas para garantir a qualidade do código. Estudos como o de Taipale *et al.* [27] apontam que existem situações, como testes de regressão, em que a automatização de testes traria benefícios na redução de custos através do alívio da carga de trabalho manual. Já outros casos, como testes exploratórios e de usabilidade do usuário, mostram que testes manuais ainda tem sua importância dada a complexidade de sua análise.

3.2.3 Plano de testes

O plano de testes é considerado um passo essencial no processo de teste de software [28]. Ele define a estratégia que será utilizada para o desenvolvimento de testes, sendo o responsável em documentar as validações que precisam estar de acordo com as especificações do usuário, e as verificações necessárias para isenção de erros.

Sendo explicitado em um ou mais documentos, o plano de testes possui especificações sobre a natureza de cada teste, como o conteúdo a ser testado, o critério de aceitação e as suas condições de execução. Para todo teste desenvolvido, seja manual ou automático, é necessário o planejamento de seu funcionamento, e idealmente documenta-se este planejamento em um plano de testes.

Planos de testes podem variar conforme a área de atuação do teste. Testes modulares de código podem conter planos de testes mais simples, enquanto testes de validação pelo usuário possuem especificações mais complexas e detalhadas. Em todos os casos, entretanto, é possível obter uma visão geral do que é necessário ser testado e ter um registro dos resultados de cada teste especificado.

Um outro papel do plano de testes é quantificar os recursos necessários ao exercício do teste. É a partir dele que os administradores são capazes de obter uma visão geral da cobertura de testes sobre a aplicação e verificar que áreas estão com falta de testes e quais estão com redundância. Portanto, é possível observar por exemplo que uma aplicação pode possuir um alto número de testes de usabilidade em uma parte do sistema, mas ao mesmo tempo apresentar carência em especificações de testes de aceitação em outra. Este levantamento permite um melhor controle da implantação de testes.

O padrão *IEEE* 829 [29] para documentação de teste de software define em 8 passos principais o desenvolvimento de um plano de testes, como listados abaixo:

1. Analisar o produto para obter informações, compreendendo seus requisitos.

2. Implantar a estratégia de teste, levando em consideração o escopo e o tipo de teste.

3. Definir o objetivo do teste, listando as funcionalidades e estabelecendo metas.

4. Estipular os critérios de teste, como a continuidade ou suspensão do ciclo de testes de acordo com os resultados de uma etapa.

5. Planejar os recursos necessários para os testes, sejam humanos ou computacionais.

6. Preparar o ambiente de testes, onde serão realizados os testes.

7. Estimar um cronograma de desenvolvimento e execução de testes.

8. Catalogar todos os componentes presentes antes, durante e depois do ciclo de testes, como documentos, *scripts* e relatórios.

Para introduzir um plano de testes é necessário organizá-lo conforme os tipos de teste desejados, levando em consideração as diferentes categorias de testes existentes para implementação.

3.2.4 Categorias de teste

Diversos pontos de vista são encontrados na literatura em relação a classificação de testes. Por exemplo, na listagem feita por Rungta [30], mais de 100 categorizações existem para testes de software. Porém nesta fundamentação serão discutidas apenas as classificações de testes mais relevantes e que foram utilizadas neste trabalho.

Os testes podem ser classificados em relação a sua capacidade de observação interna. Myers [17] usa a caracterização de um teste podendo ser caixa preta ou caixa branca. Um teste é considerado caixa preta quando não há preocupação com o conhecimento do comportamento interno e da estrutura do programa. A partir da observação de suas entradas e saídas, o objetivo do teste é encontrar as circunstâncias nas quais o programa não se comporta de acordo com suas especificações. Já o teste de caixa branca permite examinar a estrutura interna do programa de forma que o teste possa ser planejado de acordo com a lógica do código.

Outra classificação importante encontra-se presente em Graham *et al.* [31], na qual se classifica os testes de acordo com os requisitos, podendo estes serem funcionais ou não funcionais. Testes funcionais verificam se o programa funciona de acordo com o comportamento esperado ao ser desenvolvido, levando-se em consideração os requisitos. Já os testes não funcionais avaliam critérios complementares e de caráter qualitativo como desempenho, segurança e usabilidade do software. Pode-se ter como exemplo um requisito funcional que especifica a realização de uma ação específica, enquanto um requisito não funcional especifica uma velocidade aceitável para a realização de tal ação.

Mais uma classificação pertinente na literatura é em relação a mudanças no software. Segundo Graham, quando o software recebe uma atualização e ocorre um erro durante o teste deste novo módulo, a inconsistência é reportada. Assim que uma nova versão é lançada contendo o conserto do erro, o mesmo teste, nas mesmas condições anteriores, precisa ser refeito. Tal teste passa então a ser classificado como teste de confirmação. De forma semelhante, na verificação da mudança também ocorrem os testes de regressão. Entretanto os testes de regressão contêm todos os testes que anteriormente não haviam reportado erros, garantindo que uma nova mudança não afetará algo que funcionava de forma correta previamente.

Após essa série de classificações, tem-se por fim, mais uma categorização de acordo com a área de atuação do teste. Levando em consideração a análise feita por Bourque *et al.* [32], os testes podem ser enquadrados em 4 níveis principais. Estes são os testes de unidade, testes de integração, testes de sistema e testes de aceitação. Nas próximas duas seções esses tipos de testes serão divididos entre manuais e automáticos. Esta divisão é feita de acordo com a aplicabilidade habitual feita pela comunidade.

3.2.5 Testes manuais

Apesar da grande vantagem apresentada pela realização de testes automáticos, em certos casos eles não podem substituir os testes manuais. Como sugerido por Myers *et al.* [17], é importante a realização de validações humanas em certas situações, tais como revisão de código, testes exploratórios, testes de usabilidade, e por fim, testes de aceitação de usuário. Estes testes são em geral testes de confirmação, e possuem características que os tornam difíceis de serem automatizados.

A revisão de código fornece uma avaliação a qualidade geral do código apresentado, focando nos níveis de unidade e integração. Por ser uma verificação de caixa branca, ao ler o código um outro desenvolvedor pode classificar quesitos como facilidade de manutenção, extensibilidade, usabilidade e clareza. O revisor pode também sugerir soluções preexistentes que o autor do código desconhece, assim como também aprender novas técnicas de programação ao avaliar um código de terceiro. A revisão de código permite um aprendizado colaborativo entre autor e revisor, e sua aplicação já se mostrou benéfica em diversos estudos [33].

Testes de sistemas são os responsáveis em verificar se o produto está consistente com os objetivos originais. Testes categorizados nesta área de atuação são realizados após testes de integração e antes dos testes de aceitação, analisando a coordenação dos diversos componentes em código que resultam na funcionalidade a ser utilizada pelo usuário final. São testes de caixa preta e que dependendo do tipo de teste podem ser tanto manuais como automáticos, avaliando tanto critérios funcionais como não funcionais. Os testes exploratórios e de usabilidade são considerados testes de sistema.

Testes exploratórios, termo inventado por Cem Kaner [34] em 1984, avaliam o produto desenvolvido sem possuir instruções e entendimento de como o próprio funciona. Sendo um teste de caixa preta, o testador explora o teste de forma livre em busca de comportamentos inesperados. O teste exploratório também contribui com o desenvolvimento do plano de testes, fomentando a descoberta de mais casos de teste. Possuindo uma característica de depuração, testes exploratórios também podem ser acrescentados durante a execução de um teste convencional, a fim de

tentar reproduzir um defeito aleatório ou investigar a causa raiz de um problema.

Os testes de usabilidade por sua vez avaliam como o produto é utilizado, relacionado principalmente com a sua intuitividade, clareza e qualidade. Podem ser usados como critérios por exemplo a navegação feita pelo usuário, sua facilidade em encontrar o que procura e se existem elementos que são ignorados por ele. Como forma de melhor qualificar os testes de usabilidade, em Nielsen *et al.* [35] uma fórmula empírica foi proposta a fim de estimar uma relação entre testadores e o número de problemas de usabilidade encontrados. Tal fórmula pode ser descrita a seguir:

$$E = 100(1 - (1 - L)^n)$$

Onde E é a porcentagem de erros encontrados na interface, n é o número de testadores e L é a porcentagem de problemas de usabilidade encontrados por um testador. Usando $L = 0,3$ na equação, que é um valor plausível segundo a pesquisa de Nielsen, é possível ver por exemplo que havendo 5 testadores, aproximadamente 83% dos erros de usabilidade podem ser encontrados.

Definido em Hambling *et al.* [36], testes de aceitação de usuário são realizados com o objetivo de determinar se um sistema atende aos requisitos originais do usuário. Em outras palavras, estes testes utilizam os próprios usuários como testadores, verificando se o desenvolvedor compreendeu corretamente as especificações ao desenvolver o software e validando se está de acordo com as expectativas. Apesar de se assemelhar ao teste de usabilidade, os testes de aceitação focam nos requisitos funcionais, enquanto os testes de usabilidade possuem uma visão geral, considerando os requisitos não funcionais.

3.2.6 Testes automáticos

Testes automáticos são em geral regressivos e focam nos requisitos funcionais. Myers *et al.* [17] destaca que para ciclos curtos e rápidos de desenvolvimento, como presente em ambiente ágeis, o tempo é um recurso valioso, e por isso a adoção de testes automatizados é incentivada. Os principais níveis de testes em que podem haver automatizações são os de unidade, de integração e de sistema.

Testes de unidade são os primeiros dos testes a serem executados e onde se detectam a maior parte dos erros de programação. Sendo testes de caixa branca, eles validam a nível modular de código, seja ele uma função, método ou classe.

Já os testes de integração avaliam a interface entre dois ou mais módulos a fim de encontrar inconsistências na associação dos componentes. Eles podem ser tanto caixa preta como caixa branca, de acordo com a motivação do teste.

Finalmente, há os testes ponta a ponta, que fazem parte dos testes de sistema. Estes são testes de caixa preta e focam na funcionalidade desenvolvida sob a perspec-

tiva do usuário final. São testes que simulam diferentes ambientes em que a aplicação será utilizada e validam se os resultados estão conforme os requisitos definidos.

O conteúdo deste trabalho foi realizado principalmente com o desenvolvimento de *frameworks* voltados para testes de unidade e testes ponta a ponta. Portanto, estes assuntos serão discutidos com mais profundidade nas próximas seções.

3.3 Testes unitários

É fundamental que as soluções automatizadas sejam flexíveis tanto para mudanças grandes de código, quanto para pequenas, como uma alteração mínima dentro de uma função. As verificações têm que ser independentes entre si e terem a capacidade de testar aquilo que foi mudado, sem a interferência de fatores externos. Uma das soluções concebidas para atender este objetivo de modularidade, e que vem sendo cada vez mais aplicada em softwares, é a implementação de testes unitários. Segundo o estudo feito por Yuan [37], "a maioria dos incidentes em ambiente de produção (77%) podem ser reproduzidos por um teste unitário".

Desde 1976, é possível ver o início do desenvolvimento de ferramentas com propósitos semelhantes a dos testes unitários atuais [38]. Testes eram separados em dois tipos: verificação e validação. Testes de verificação eram referentes a exatidão lógica do programa e funcionamento correto de acordo com as especificações, enquanto testes de validação eram voltados a performance e a eficiência do programa de acordo com o ambiente de usuário.

Testes de verificação eram pouco enfatizados durante o processo de desenvolvimento de software. Nenhuma metodologia era utilizada e as verificações na maior parte dos casos eram por meio de depuração manual apenas, ou seja, *debugging*. Dessa forma, Panzl e seu time da *General Electric* propuseram a *Test Procedure Language* para a linguagem *Fortran*, que se tratava de uma extensão para desenvolver testes que fossem completos, compactos e auto contidos. Definição que hoje em dia se assemelha com a de testes unitários.

Em 1989 Kent Beck menciona o *SUnit* em uma publicação, que se tratava e que ainda hoje é utilizado, de um *framework* de testes unitários para a linguagem orientada a objetos *SmallTalk* [39]. Nesta publicação, Beck proporciona uma série de diretrizes, padrões de design e ferramentas para facilitar o desenvolvimento e organização de testes unitários no novo paradigma da época, que era a orientação a objetos. A primeira versão de *SUnit* foi desenvolvida em 1994 e sua publicação em 1999.

Em 1997 Kent Beck e Erich Gamma começaram a migrar a filosofia da *SUnit* para a linguagem *Java* com o desenvolvimento do *JUnit*. A facilidade encontrada em realizar os testes unitários de forma rápida e eficiente fez com que a ferramenta se

tornasse popular, e com isso ramificações passaram a existir também para outras linguagens, marcando o início do design *xUnit*, que se trata da família de *frameworks* de testes unitários, atualmente disponíveis para mais de 80 linguagens. Como exemplo, o trabalho desenvolvido na área de testes unitários foi realizado utilizando *phpunit*, que se trata de um descendente do original *SUnit*.

Conforme significado presente em Hamill [40], testes unitários, ou testes de unidade, são definidos como testes que implementam uma metodologia na qual trechos individuais de código são testados de forma automática. Uma unidade é definida como a menor parte possível de um software que pode ser testada de forma independente, ou seja, sem a necessidade de interações externas. Por exemplo em programação *procedural*, uma unidade pode ser uma função, enquanto em programação orientada a objetos uma unidade é frequentemente caracterizada como um método de visibilidade pública ou até mesmo uma classe.

Em geral o funcionamento de um teste unitário é bem simples. Inicialmente, o teste unitário recebe uma ou mais entradas que são necessárias para o funcionamento da unidade a ser testada. A unidade é então executada e a sua saída é coletada, comparando-a com o valor de referência que era esperado com as entradas iniciais. Desta forma é garantido o funcionamento da unidade testada para este caso.

Dada a importância atual dos testes unitários no processo de desenvolvimento de software, hoje em dia a maior parte das linguagens possuem bibliotecas desenvolvidas para auxílio no desenvolvimento destes testes. Testes unitários são os primeiros testes a serem executados, e onde se detectam a maior parte dos erros de programação. Eles examinam o software a nível modular ou unitário, checando os métodos, fórmulas, caminhos lógicos e pontos de decisão encontrados no código, podendo ser executados pelo próprio desenvolvedor ao alterar o código.

A mérito quantitativo, à medida que novas linhas são escritas, ferramentas adicionais podem ser utilizadas para avaliar o quanto do código está sendo coberto e garantido por testes unitários. Pela literatura, tal verificação é chamada de cobertura de código, e levantamentos como o feito por Yang [41] mostram a disponibilidade de diversas ferramentas que permitem analisar o alcance dos testes unitários na base de código.

Testes de unidade também trazem benefícios em relação à documentação e à própria arquitetura de código. Desenvolvedores passam a entender melhor o comportamento de um módulo a partir da entrada proporcionada e da saída esperada pelo teste unitário, servindo como documentação adicional. Além disso, para um desenvolvimento mais simplificado de um teste unitário, o programador deve estar atento a qualidade de seu código, como por exemplo evitar a utilização direta de métodos externos, pois inviabilizam a simulação de objetos.

3.4 Testes ponta a ponta

Testes de unidade conseguem avaliar isoladamente diversos módulos, e testes de integração são capazes de avaliar a interação entre estes módulos. Entretanto, levando em consideração o cenário de aplicação web, com diversos navegadores usando diferentes extensões em múltiplos sistemas operacionais, as verificações apenas de módulos e suas integrações são insuficientes para garantir a compatibilidade em múltiplos ambientes. É necessário também reproduzir uma verificação que seja a mais fiel possível à experiência do usuário. Isso significa a necessidade de testar a interação com o sistema, a partir de sua interface gráfica, e a integração do sistema, desde a requisição das informações no banco de dados até ser corretamente disponibilizada ao usuário. Todos os setores se comportando da forma esperada a partir das ações do usuário na interface. A forma de verificação automatizada mais próxima de atender estes critérios é a realização de testes *end-to-end*, também chamados de testes ponta a ponta, testes de interface do usuário, *chain testing*, ou simplesmente testes *e2e*.

Em uma análise histórica, no fim da década de 80 ocorreu o surgimento dos *event-driven GUI software*, que nada mais eram do que interfaces que possuíam o funcionamento determinado pelas interações do usuário, como cliques e teclas. Com estes novos programas, novas ferramentas de testes também foram desenvolvidas, como *Segue* e *Mercury*. Essas ferramentas eram baseadas na metodologia do *capture and replay* na qual os testes manuais dos usuários eram inicialmente gravados e depois repetidos automaticamente como forma de teste de regressão.

A metodologia de testes de interface se manteve estacionária até 2004, quando Jason Huggins desenvolveu a ferramenta *Selenium* para teste de aplicações web [42]. *Selenium* é um *framework* de testes disponível em diversas linguagens que reproduz uma série de comandos diretamente no navegador, na tentativa de reproduzir o comportamento humano. Diferentemente do método de *capture and replay*, o *Selenium* permite que as instruções de comandos sejam feitas diretamente no código, tornando mais fácil sua manutenção e escalabilidade. A partir disso novas ferramentas passaram a existir, popularizando o desenvolvimento de testes *e2e*.

O objetivo dos testes *e2e* é assegurar que todas as partes integradas de uma aplicação funcionem como esperado, a partir de simulações de como um usuário real usaria a aplicação. Estas simulações têm que ser feitas contendo o maior número possível de ambientes diferentes em que será utilizada a aplicação, validando-as sob a perspectiva do usuário final. Para a realização destes testes, é fundamental o conhecimento da aplicação como um todo.

Durante o teste *e2e*, os diferentes componentes da sua aplicação são avaliados de forma unificada, de forma que banco de dados, *back-end* e *front-end* sejam avaliados

simultaneamente. Enquanto os testes de unidade e de integração identificam inconsistências de forma mais modular e a nível de código, os testes de *e2e* procuram por inconsistências em um panorama geral. De certa forma, pode-se dizer que testes *e2e* possuem um alcance maior do que testes de unidade e de integração, mas ao mesmo tempo não possuem a mesma precisão e objetividade em identificar a localização de um erro, consequentemente sendo menos simples o processo de depuração de erros.

Na literatura mais atual, pode-se encontrar em alguns casos em que testes *e2e* não são considerados testes de sistema [43]. Isso se deve ao fato de ainda haver discordância na comunidade sobre a área de atuação de testes de sistemas. Para alguns, os testes *e2e* são realizados posteriormente aos testes de sistema, fazendo verificações adicionais não funcionais que não são responsabilidade dos testes de sistema. Entretanto, baseando-se em Myers *et al.* [17] e o conceito de que o objetivo dos testes de sistema é, acima de tudo, apresentar que o produto está de acordo com seus objetivos iniciais, pode-se concluir que testes *e2e* possuem características de teste de sistema, já que esse é seu principal objetivo, apesar de verificar por consequência também critérios não funcionais.

O desenvolvimento de testes de *e2e* requer inicialmente a análise dos requisitos feitos pelos usuários, que pode ser especificado em um plano de testes. A partir daí são definidas *personas* que são identidades para caracterizar um grupo de usuários de acordo com suas características comuns, como permissões de acesso, ambiente de navegação e acessibilidade. Obtendo estes agrupamentos, condições são implementadas baseando-se em cada *persona*, que se transformam no fim em diferentes casos de testes. Estes casos de testes contêm informações sobre as ações permitidas aos usuários, que são então convertidas em comandos escritos em código. Esses comandos são então executados e passam a representar o usuário, realizando ações na interface como cliques de botões, escritas de texto e navegação de links.

Com o desenvolvimento de testes automáticos para sua aplicação, uma outra necessidade passa a surgir, que é como gerenciá-los. Se torna preciso a existência de um agente controlador que execute os testes com uma certa frequência ou à medida que ocorre uma atualização na aplicação, reportando os possíveis erros de regressão encontrados. A integração contínua é vista como uma opção para isso, e será discutido em mais detalhes na próxima seção.

3.5 Integração contínua

Conforme apresentado por Howard Deiner [44], tanto o desenvolvimento de software como o de seus testes automáticos são necessariamente realizados pelos programadores. Até o momento, não é possível o desenvolvimento de testes de unidade, de integração ou *end-to-end* por mecanismos automáticos. Entretanto sua execução

pode ser automatizada. Não é necessário que o programador seja responsável por realizar tarefas repetitivas como, preparar o ambiente para os testes, executá-los e verificar os resultados de modo individual. Além de poder ser caracterizado como um desaproveitamento de recursos humanos, pode ser visto também como arriscado, já que erros podem ocorrer nos procedimentos a serem seguidos ou nas observações dos resultados.

Uma solução automática de integração pode melhorar a produtividade do desenvolvedor, encontrar inconsistências de forma mais eficiente e distribuir novas versões da aplicação mais rapidamente.

Fowler *et al.* [45] define integração contínua como uma prática de desenvolvimento de software na qual membros de uma equipe integram seus desenvolvimentos frequentemente, podendo ocorrer diversas vezes ao dia. Cada integração é verificada por um processo automático de compilação e teste, detectando erros mais rapidamente. Isso por fim agiliza o desenvolvimento e o torna mais coeso, indo de acordo com os princípios do desenvolvimento ágil. Integração contínua também é conhecido como *continuous integration* ou apenas *CI*.

Como apresentado por Fowler e detalhado em Shanin *et al.* [46], a integração contínua é aplicada juntamente a uma plataforma de controle de versão, como o *Gitlab*. Assim que um pedido de novo código é submetido à base de código, por meio de um processo denominado de *merge request*, a integração contínua entra em ação antes deste código ser adicionado. Ao identificar esta requisição, a integração contínua executa uma canalização, mais conhecida como *pipeline*, que consiste em uma cadeia de tarefas a serem realizadas, chamadas *jobs*. Cada *job* fica responsável por uma ação específica, como compilação, testes de unidades ou testes *e2e*. Se em algum momento algum dos *jobs* falha, a *pipeline* é imediatamente interrompida, informando os autores da requisição e impedindo a adição do código.

Juntamente com a integração contínua, pode ocorrer uma prática complementar semelhante que é a distribuição contínua, ou *continuous delivery*. Após o processo de integração ter sido realizado com sucesso, a distribuição contínua se responsabiliza pelo *deploy*, ou seja, distribuir todas as alterações de código em um ambiente de teste ou ambiente de produção, realizando todas as etapas necessárias também de forma automatizada.

Para o funcionamento da integração contínua é fundamental a existência de um ambiente para execução das tarefas. Isso significa, na condição mais simples, a existência de um servidor que esteja devidamente configurado para realizar as tarefas da integração contínua. Para tarefas simples que exigem pouco processamento, como compilação da aplicação e testes unitários, apenas uma máquina pode ser suficiente para desempenhar tais obrigações. Entretanto, em certos casos como o deste trabalho, há tarefas mais complexas, como os testes *e2e*, que exigem a

necessidade de um processamento maior do que apenas uma instancia operacional pode oferecer.

Os testes *e2e* reproduzem o usuário utilizando o navegador web, realizando as ações humanas com tempo de duração semelhante. Por conta disso, se todos estes testes forem postos em série, o *job* de integração contínua pode demorar para ser concluído, inviabilizando o processo em ambientes que empregam metodologia ágil. Portanto é necessário o ato de paralelizar os testes *e2e*, tornando a execução mais rápida, e para isso um *cluster* de máquinas se torna uma solução viável. Para o correto funcionamento concomitante de cada máquina do *cluster*, uma comunicação entre elas é essencial, e a partir daí vem o conceito de sistemas distribuídos.

Segundo Tanenbaum *et al.* [47], um sistema distribuído é "uma coleção de computadores independentes que aparenta aos seus utilizadores como um sistema único e coerente". É importante a independência entre as máquinas para garantir a fácil escalabilidade quando necessário, mas ao mesmo tempo abstraindo a complexidade interna para tornar simples sua utilização, no caso pela integração contínua. Nestes computadores, uma camada de software adicional é implementada entre a camada do sistema operacional e a das aplicações em que serão executadas as tarefas, como os testes *e2e*. Esta camada possibilita a comunicação entre as aplicações e simplifica as possíveis diferenças de hardware e sistema operacional de cada instância.

Uma forma de estruturar os testes *e2e* em um ambiente de sistemas distribuídos é por meio de uma divisão entre primário e secundário, como presente em Nwana *et al.* [48]. As máquinas primárias são as responsáveis por atender o pedido do utilizador e gerenciar a execução dos testes para as máquinas secundárias. Estas máquinas primárias coletam os resultados de cada máquina secundária e enviam o resultado final para a integração contínua.

No próximo capítulo será apresentado o trabalho desenvolvido com base na fundamentação, com a reestruturação do grupo adotando práticas ágeis, desenvolvimento dos *frameworks* de testes automáticos e a implementação de uma solução de integração e entrega contínua de software.

Capítulo 4

Ambiente de testes automáticos em nuvem

Neste capítulo será apresentado o trabalho técnico realizado iniciando com os *frameworks* desenvolvidos, *Fence Unit Test* e *Fence Automated Testing Environment*. Após isso, será abordado o trabalho de infraestrutura, desenvolvimento e operações realizados com integração e entrega contínua, assim como a preparação de máquinas em nuvem. Por fim, será apresentado as melhorias gerais implementadas seguindo princípios de metodologias ágeis.

4.1 Testes unitários com FUnit

O *framework Fence Unit Test*, ou *FUnit*, foi implementado para auxiliar os desenvolvedores de sistemas *Fence* na escrita de testes unitários. Conforme mostrado de forma simplificada pela figura 4.1, após definir o método que deseja ser testado como entrada, o desenvolvedor escreve o teste utilizando as ferramentas disponibilizadas pelo *FUnit*, para no final ser fornecido um relatório com o resultado.

Figura 4.1: Simplificação do *FUnit*. Métodos que desejam ser testados são usados como entrada e um relatório é gerado no final.

Esta seção discute o *framework* desenvolvido, explicando primeiramente o *ph-*

punit e a possibilidade de aprimorar a ferramenta para as necessidades dos desenvolvedores do grupo. É discutido também a implementação do *framework*, com detalhamento de casos de uso e funcionalidades mais avançadas que foram desenvolvidas.

Framework phpunit

Para atender o objetivo de pequenas entregas, proposto pela metodologia ágil, é fundamental garantir a qualidade do código. Os testes unitários são essenciais para contribuir para esta garantia. Com isso, deu-se início a uma pesquisa por soluções disponíveis para escrita de testes unitários em *php*, a linguagem principal do *Fence*. Entre as opções encontradas, o *framework phpunit* foi a opção escolhida para este projeto. Esta ferramenta é atualmente a que possui mais recursos e está presente na comunidade desde 2001.

Antes de dar início a descrição do *phpunit*, é importante definir um conceito criado neste trabalho que é frequentemente utilizado ao longo desta seção. Para agrupar os parâmetros que são utilizados para implantação de testes unitários, como os valores de entrada ou os valores de saída esperados ao realizar o teste, é utilizado o termo *condições de controle*. Ao testar um método por exemplo, pode ser necessário declarar um valor que será o argumento de entrada, assim como o valor esperado de retorno. Estes valores são portanto as *condições de controle* para este teste unitário.

Para realizar o teste de um método de uma classe, ou da própria classe inteira, é preciso estender a classe abstrata base do *phpunit* por meio do princípio da herança. Para o caso de um método, por exemplo, é necessário desenvolver um método testador que contenha a execução do método original a ser testado e as nomeadas *condições de controle*. O retorno do método original é então comparado com o valor esperado, informando se o teste obteve alguma falha. Na ausência de falhas, o teste é classificado como aprovado. Na figura 4.2 pode ser visto um exemplo simples deste processo.

```php
/**
 * Classe contendo suíte de testes
 */
class AverageTest extends TestCase
{
    /**
     *  Um caso de teste do método "mean"
     */
    public function testCalculationOfMean()
    {
        // Valor de entrada
        $input = [3, 7, 6, 1, 5];

        // Valor esperado de saída
        $outputExpected = 4.4;

        // Valor obtido ao chamar o método a ser testado
        $outputObtained = $this->Average->mean($input);

        // Validaçao do resultado
        $this->assertEquals($outputExpected, $outputObtained);
    }
}
```

Figura 4.2: Exemplo simplificado de um teste unitário.

No contexto da teoria de testes, cada método testador é definido como *test case*, ou caso de teste, enquanto o agrupamento de *test cases* por sua vez é chamado de *test suite*, ou suíte de testes. É de costume um ou mais *test cases* validarem cada método de uma classe, e o conjunto de todos os *test cases* que percorrem a classe ser um *test suite*. Quando executado, o *phpunit* percorre os *test suites* desejados e exibe os resultados de todos os *test cases* presentes.

Busca por padronização e concisão

No passado, alguns testes de unidade foram experimentalmente desenvolvidos pelo grupo com o propósito de incorporar a prática no futuro junto ao processo de desenvolvimento. Com o crescimento do número de testes elaborados foi possível identificar pontos a serem aprimorados. Foi analisado que a escrita dos testes poderia ser padronizada e otimizada. Viu-se a oportunidade em adaptar setores do *framework* para o contexto do grupo, podendo tornar mais econômico e mais rápido para o programador desenvolver testes rotineiramente.

Notou-se a possibilidade de melhoria na organização nos *test cases* e suas *condições de controle*. No exemplo anterior e como era feito inicialmente pelos desenvolvedores do grupo, estas condições eram definidas juntamente com a lógica do

teste, ou seja, no mesmo código, podendo inclusive ocorrer repetições entre testes que possuíssem as mesmas condições. Esta implementação é satisfatória para uma certa ordem de grandeza de testes, porém quando se passa a ter milhares de testes unitários, dificulta-se a manutenção dos mesmos. O desenvolvedor precisa procurar em que linhas de código a *condição de controle* foi definida, sem indicativos claros de sua localização e sem uma padronização de como esta condição foi estabelecida. Idealmente as *condições de controle* deveriam estar desatreladas da lógica do teste, sem repetições e seguindo um padrão claro, que facilite a compreensão e manutenção.

Para facilitar o desenvolvimento dos testes pelos programadores, percebeu-se também a possibilidade de simplificar procedimentos mais complexos. Isso ocorre principalmente na simulação de objetos, conhecidos como *mocks*. Para testes unitários simples, em que não ocorre chamadas a outros métodos da classe, *mocks* são desnecessários. Contudo, na maioria dos casos, um método a ser testado possuirá chamadas a outros métodos da mesma classe, e levando em consideração os princípios da prática de teste unitário, o comportamento destas chamadas precisa ser controlado. Um teste unitário não pode possuir dependências externas, como a outros métodos que podem ter comportamentos imprevisíveis, pois isso prejudicaria a natureza modular do teste. Portanto conclui-se que *mocks* são necessários, e apesar do *phpunit* possuir funcionalidades nativas para criá-los, grande parte dos comandos podem ser simplificados, reduzindo a prolixidade da escrita.

Levando em consideração principalmente estes dois motivos, foi proposta a implementação de uma camada acima do *framework phpunit*, que passaria a ser a nova ferramenta para desenvolvimento de *test suites* do grupo. Esta camada tem como objetivo principal implementar um padrão conciso na escrita de testes unitários, guiando o desenvolvedor durante a escrita do teste. Pelo fato de ser uma abstração motivada para testar código relacionado com o *framework Fence*, esta ferramenta foi nomeada *Fence Unit Test*.

Desenvolvimento do *framework FUnit*

FUnit foi inspirado nos princípios seguidos pelo próprio *Fence*. Através do *Fence*, os parâmetros sujeitos a mudanças constantes são abstraídos em arquivos de configuração, de modo que até mesmo o próprio usuário possa ser capaz de alterá-los em certos casos. A premissa é semelhante ao *FUnit*, na medida que as *condições de controle* são exteriorizadas em um arquivo *JSON*. Dessa forma, torna-se possível para o desenvolvedor alterar o valor de entrada ou da saída esperada sem a necessidade de alterar o código do próprio teste.

Na figura 4.3 encontra-se o diagrama contendo as classes que serão referenciadas nesta seção. O *Fence Unit Test* é constituído apenas de duas classes. Sua classe principal, a *TestCase*, responsável pela maior parte da lógica do *framework*, e a

classe secundária, *Assert*, encarregado de asserções. Juntamente com a orientação a objetos, estas classes foram escritas baseando-se parcialmente no paradigma funcional, evitando mudanças de estado e dados mutáveis. Optou-se pela estruturação em pequenos métodos, dependentes apenas de seus argumentos, e gerar novas variáveis ao invés de reutilizá-las.

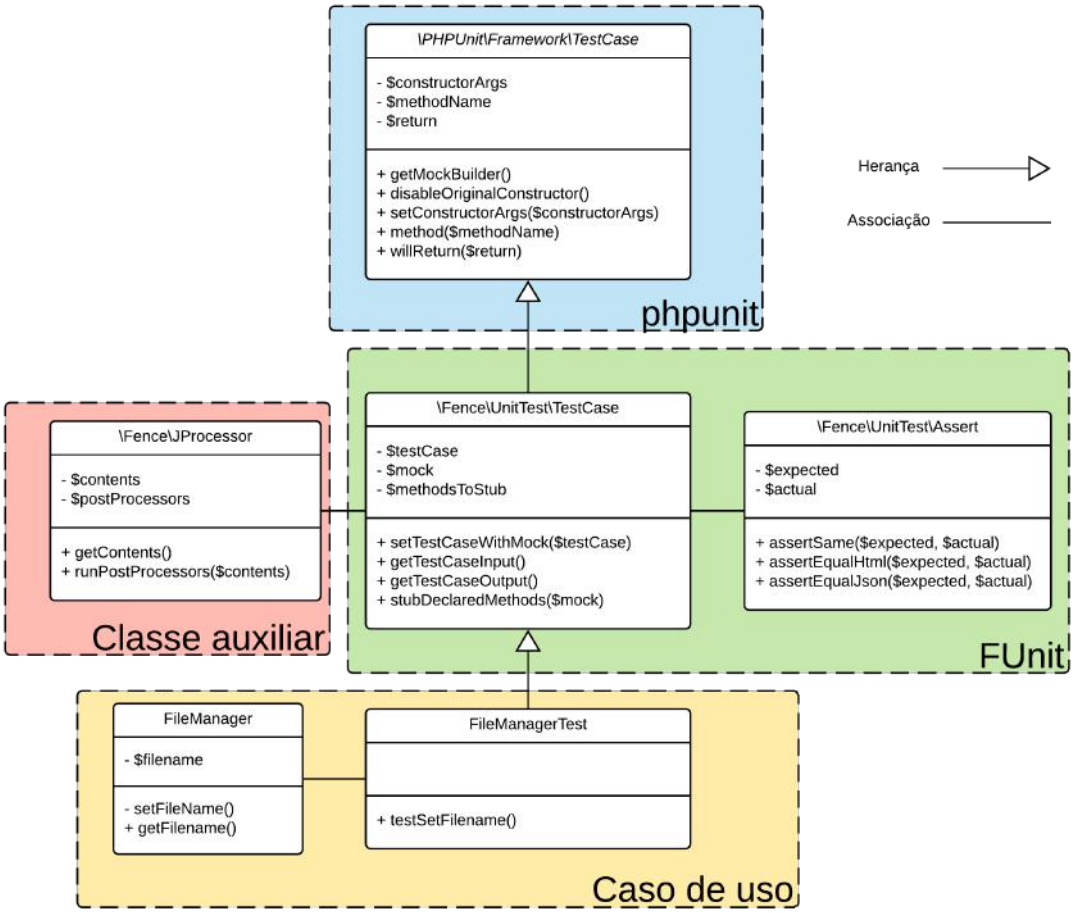

Figura 4.3: Diagrama de classes da *FUnit*, classes auxiliares e classes de uso.

Aplicaram-se também dois princípios da filosofia *SOLID* para o *FUnit*. Primeiramente foi utilizado o princípio da inversão de dependências, no qual uma classe de testes possui dependência apenas das classes abstratas da *FUnit*, *TestCase* e *Assert*. Também foi utilizado o princípio da responsabilidade única, de modo que os métodos foram planejados para cumprirem unicamente uma tarefa.

Um conceito importante para a compreensão da *FUnit* é o de arquivos de configuração em *JSON*. Arquivos *JSON* se comportam como dicionários estruturados em chave e valor. A flexibilidade do *JSON* em aceitar diversos tipos de valores permite a representação de várias estruturas de dados, como listas e objetos. Por este motivo foi decidido em parametrizar as *condições de controle* nestes arquivos.

Com o objetivo de dar mais opções na escrita de arquivos de configuração, foi

implementado uma classe auxiliar de processamento de arquivos *JSON* chamada *JProcessor*. Esta classe interpreta o arquivo de configuração realizando mudanças de acordo com certas regras. Ações como capacitação de comentários, substituição de variáveis globais, concatenação de arquivos de configuração e execução de funções *callback* são possíveis por meio da classe *JProcessor*, se tornando um utilitário para os desenvolvedores para além do desenvolvimento de testes.

Como desenvolver um teste unitário: casos de uso

Observando o funcionamento interno da *FUnit* é possível identificar as etapas bem definidas ao longo do processo de desenvolvimento de testes, como presente na figura 4.4. Definido o método a ser testado, considerado como requisito, é possível dar início ao processo de desenvolvimento do teste. Em resumo, *condições de controle* são declaradas em arquivos de configuração e interpretados por uma classe auxiliar que as disponibiliza para a escrita da lógica do teste. Os testes são então executados e validados, gerando um relatório final com os resultados.

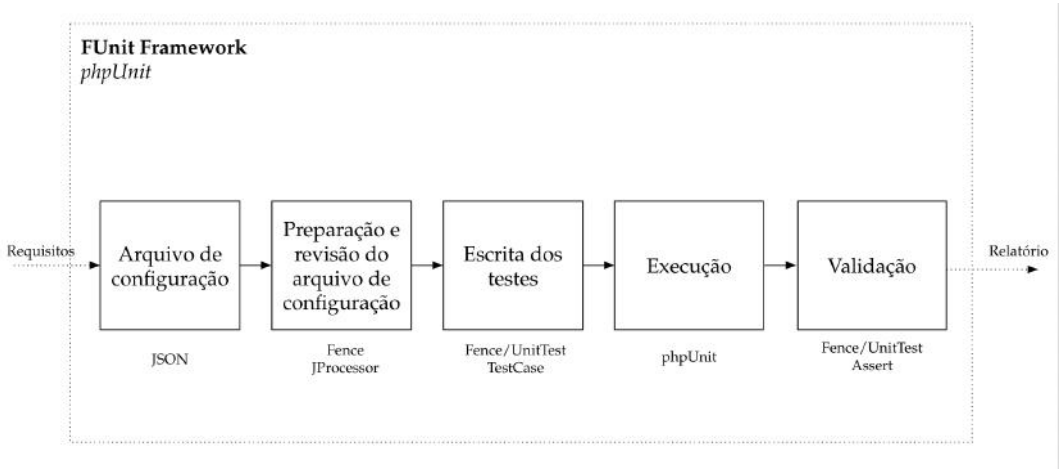

Figura 4.4: Funcionamento interno do *FUnit*.

A figura 4.5 exemplifica um método de classe a ser testado. Este exemplo se trata apenas de um atribuidor de propriedades que armazena o nome de um arquivo na instância da classe. Chamado *setFilename* este método faz parte da classe *FileManager* responsável por gerenciar arquivos. Ele possui dois comportamentos distintos, um para o caso de sucesso, e outro para o caso de falha. Na eventualidade do argumento recebido não ser um arquivo, uma exceção deve ser executada, notificando o desenvolvedor do erro. Caso contrário, o nome do arquivo é armazenado como uma propriedade da classe. Portanto, pode-se concluir que dois testes unitários são necessários para esse método, validando cada comportamento possível.

```php
public function setFilename($filename)
{
    if (!is_file($filename)) {
        throw new \Exception("Unable to open [$filename]");
    }
    $this->_filename = $filename;
}
```

Figura 4.5: Método a ser testado pelo teste unitário.

É necessário antes de tudo a preparação de um arquivo *JSON* de configuração que proporcione as *condições de controle* do teste, como presente na figura 4.6. Este arquivo inicialmente possui a propriedade *testSetFilename*, que contém as condições necessárias para o desenvolvimento do *test case* de *setFilename*. Dentro desta propriedade, pode-se ver a subpropriedade *class* apontando para o *namespace* da classe *FileManager*, com a condição de construtor desabilitada, denominado *constructor*. Isso significa que um *mock* da *FileManager* será gerado sem a exigência de executar o construtor da classe. Em seguida pode-se ver a subpropriedade *input* que por sua vez possui dois atributos. Estes atributos declaram a entrada para cada teste unitário deste método, sendo um para o comportamento em que se armazena o valor e o outro para quando ocorre a exceção. No fim do arquivo tem-se a propriedade *output*, que contém o valor a ser armazenado pelo método testado, no caso sendo o mesmo valor de *input*.

```json
{
    "testSetFilename": {
        "class": "\\Fence\\FileManager",
        "constructor": {
            "disabled": true
        },
        "input": {
            "filename": "{{FILEMANAGER_FOLDER}}/getContents.json",
            "invalidFilename": "invalidFilename.json"
        },
        "output":"{{FILEMANAGER_FOLDER}}/getContents.json"
    }
}
```

Figura 4.6: Arquivo de configuração com as condições de controle para teste.

Finalizado o arquivo, resta escrever a classe de teste *FileManagerTest*, que estende a classe *TestCase* do *FUnit*. A *TestCase* irá automaticamente invocar a classe *JProcessor* para realizar um pré-processamento do arquivo de configuração. A *JPro-*

cessor irá realizar as traduções necessárias e decodificar o arquivo da estrutura *JSON* para a estrutura de dados compatível com a linguagem *php*.

Dentro da classe *FileManagerTest*, ilustrada na figura 4.7, é desenvolvido um dos métodos de teste, o *testSetFilename*, que valida o valor armazenado. Este método por sua vez executa o principal método da *TestCase*, chamado *setTestCaseWithMock*, que é o responsável por tornar disponíveis as informações do arquivo de configuração. A partir disso se obtém o valor do argumento a partir do *input* que será usado pelo método *setFilename*. O retorno do método é por fim comparado com o valor esperado, por meio do método *assertSame* que é disponibilizado por intermédio da classe *Assert*. Quando o teste é executado pela linha de comando *phpunit*, é acusado o erro caso haja divergência entre os valores.

```php
class FileManagerTest extends \Fence\UnitTest\TestCase
{
    public function testSetFilename()
    {
        $fileManager = $this->setTestCaseWithMock();

        $filename = $this->getTestCaseInput('filename');
        $fileManager->setFilename($filename);

        $expected = $this->getTestCaseOutput();
        $actual = $fileManager->getFilename();

        $this->assertSame($expected, $actual);
    }
}
```

Figura 4.7: Lógica do teste do método.

Utilizando a mesma classe e o mesmo arquivo de configuração, o programador pode repetir o processo e desenvolver mais testes até que seja atingida a cobertura de código desejada para a classe a ser testada.

Funcionalidades adicionais

Com o início do uso do *FUnit*, observou-se a necessidade de funcionalidades adicionais, que estão exemplificadas na figura 4.8. Primeiramente foi desenvolvido o conceito de *Mocked Methods*, no qual é possível controlar a resposta de um método pertencente a um *mock*. Isso se torna essencial para testar métodos que fazem chamadas a outros métodos de mesma classe.

Como visto no exemplo anterior, o *FUnit* permite também ter múltiplas condições de entrada ou de saída no arquivo de configuração. Isto se torna extremamente útil quando é necessário verificar diferentes comportamentos para um mesmo método a ser testado.

Tornou-se possível também a possibilidade de passar argumentos para o construtor do *mock* quando habilitado. Na maior parte dos casos, não é necessário executar o construtor, já que dado o princípio do teste unitário ser isolado, o próprio independe da execução do construtor. Entretanto, para testar múltiplos módulos, como é o caso dos testes de integração, esta ferramenta pode ser útil.

Apesar do *phpunit* possuir uma biblioteca nativa com asserções disponíveis para diversas estruturas de dados, se tornou essencial a possibilidade de gerar asserções adicionais, personalizadas para os *test cases* do projeto. Para esse intuito a classe auxiliar *Assert* foi desenvolvida, a qual estende as asserções nativas do *phpunit* e permite a escrita de novas asserções conforme a necessidade.

Por fim tem-se no *FUnit* a possibilidade de compartilhar propriedades em comum dentro do próprio arquivo de configuração. Foram desenvolvidos métodos híbridos no *FUnit* que, na ausência de uma *condição de controle* dentro da propriedade do *test case*, procura automaticamente na raiz do arquivo. Esta funcionalidade foi gerada principalmente para o compartilhamento de configurações de *mock*, já que geralmente os *test cases* de uma *test suite* utilizam a mesma classe como referência de *mock*.

Figura 4.8: Exemplo mais sofisticado de um arquivo de configuração provido à *FUnit*.

Discutida as principais características do *framework FUnit*, a próxima seção discute o framework *Fate*, voltado para testes *end-to-end*.

4.2 Fate e desenvolvimento de testes ponta a ponta

O *Fence Automated Testing Environment*, ou apenas *Fate*, possui o objetivo de testar as interfaces de um sistema web de acordo com os seus requisitos. Como presente de forma simplificada na figura 4.9, o *Fate* é um *framework* para testes *end-to-end*, ou *e2e*, e utiliza como entrada os requisitos expressos no plano de testes, e com os testes desenvolvidos a partir dele, o relatório de erros é gerado.

Figura 4.9: Simplificação do *Fate*. A partir de um plano de testes, um relatório final com *test cases* é gerado.

Esta seção foca desde as experiências realizadas inicialmente com diversas ferramentas *end-to-end* até a escolha final. Após isso é detalhado as características da ferramenta escolhida e o planejamento do *framework Fate*, descrevendo as tecnologias utilizadas e casos de uso.

Escolha de ferramenta *end-to-end*

Diferentemente do *phpunit*, não existe a consolidação na comunidade de uma ferramenta para *end-to-end testing*. Mais de dez diferentes tecnologias foram avaliadas durante a realização deste trabalho.

Inicialmente para o desenvolvimento de um protótipo, foram experimentadas duas ferramentas simultaneamente, *Sikuli* e *Selenium IDE*, com exemplos de código na figura 4.10. *Sikuli*, atualmente chamado de *SikuliX*, se trata de um *framework* no qual é possível utilizar capturas de tela, ou *screenshots*, de elementos da página juntamente com sua lógica, em uma abordagem diferente das outras ferramentas. Já o *Selenium IDE*, que faz parte da coleção *Selenium* de ferramentas de automação de navegadores, se baseia no conceito de *capture and replay*, no qual o testador realiza os comandos no navegador e o *Selenium IDE* captura estas iterações, gerando o código referente aos eventos e aos elementos da página web. Na imagem 4.10 é possível ver exemplo destas ferramentas.

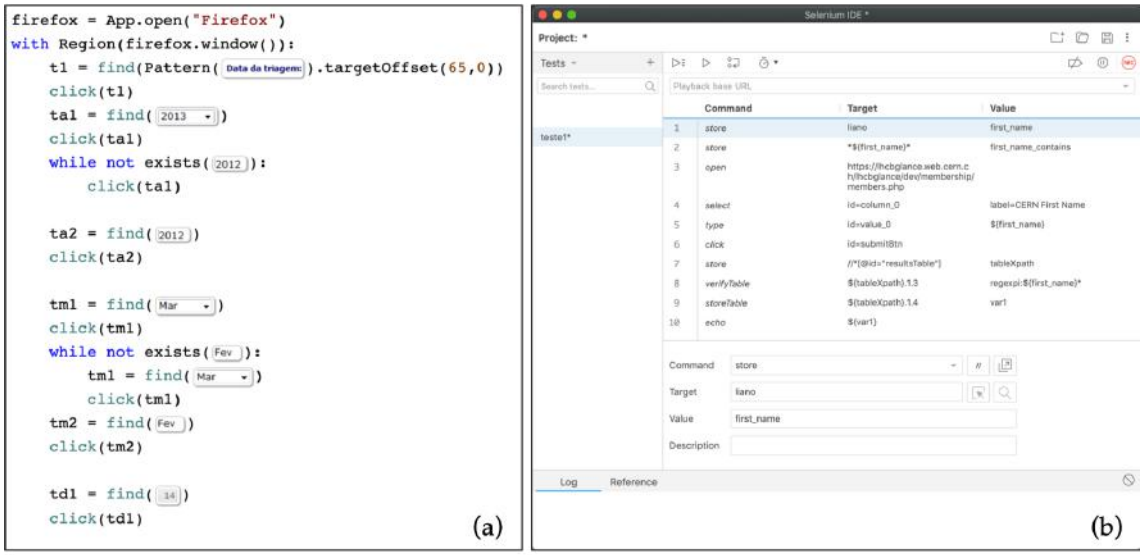

Figura 4.10: Exemplo de código *Sikuli* em (a), com capturas de tela usadas na lógica do código. Em (b), exemplo de uma instrução de comandos no *Selenium IDE*.

Apesar de ambas funcionarem corretamente, nenhuma delas atendia todas as necessidades dos sistemas. O *Sikuli* realiza as asserções de teste por meio de regressão visual com comparação direta de imagens, o que é algo volátil a acusação falsa de erros. Juntando-se isso ao fato de a ferramenta ter perdido grande parte de seus mantenedores, foi desistido do uso da mesma. Já o *Selenium IDE*, apesar de sua praticidade e da sua geração automática de código, não permite lógicas mais complexas, como iterações e condicionais. Concluiu-se que seria preferível escrever os testes diretamente em código, sem captura direta de imagens ou gravação de movimentos, e dessa maneira foi decidido experimentar a ferramenta *Selenium Webdriver*.

O *Selenium Webdriver* é considerado um dos mais famosos *frameworks* de *end-to-end testing* para navegadores. Com ele se torna possível desenvolver testes sofisticados, com lógica de linguagens imperativas, capazes de realizar testes em diferentes sistemas operacionais com navegadores distintos. Disponível em diferentes linguagens, os primeiros protótipos foram feitos na linguagem *Python*. Um exemplo de código de teste realizado se encontra na figura 4.11, em que é verificado se existe um contrato na interface com um nome específico.

```
def test_12(self):
    driver = self.driver
    first_name = "Leandro"
    driver.get("https://lhcbglance.web.cern.ch/")
    driver.find_element_by_id("txtFormsLogin").send_keys("lhcbglan")
    driver.find_element_by_id("btnFormsLogin").click()
    driver.get("https://lhcbglance.web.cern.ch")

    driver.find_element_by_xpath("/option='LHCb First Name']").click()

    driver.find_element_by_id("value_0").clear()
    driver.find_element_by_id("value_0").send_keys(first_name)
    driver.find_element_by_id("submitBtn").click()
    tableXpath = "//*[@id=\"resultsTable\"]"

    for i in range(60):
        try:
            content = driver.find_element_by_css_selector("BODY").text
            if re.search(r"^[\s\S]*Details[\s\S]*$", content): break
        except: pass
        time.sleep(1)
    else: self.fail("time out")
```

Figura 4.11: Exemplo de código *Python* utilizando o *Selenium Webdriver*.

Selenium Webdriver foi a ferramenta utilizada durante a parte inicial deste trabalho, entretanto não oferece uma solução completa para utilização em ecossistemas mais complexos. Diversos utilitários adicionais não estão presentes no *framework*, como registro de eventos *log*, biblioteca de validações, gerador de relatório, depuração e facilidades a integração contínua. Havia a possibilidade de desenvolver esses complementos, entretanto seria algo que consumiria tempo. Foi então realizada uma nova pesquisa e um novo *framework* foi encontrado, desenvolvido a partir do *Selenium Webdriver*. Chamado de *WebdriverIO*, este foi a escolha final para desenvolvimento deste trabalho.

Implantação do *framework WebdriverIO*

Este *framework* é uma implementação feita a partir do *Selenium Webdriver*, escrito na linguagem *Javascript* e interpretado pelo mecanismo *Node.js*. Como esta é uma das principais linguagens utilizada pelo grupo de desenvolvedores, a facilidade em escrever código se torna maior do que era com o *Selenium Webdriver* em *Python*. A abstração existente no *WebdriverIO* permite que comandos mais complexos, que implementados em *Selenium* precisam de múltiplas linhas e diversas condicionais para considerar diferentes ambientes, sejam realizados em apenas uma linha. Isso é perceptível principalmente em *watchers*, que são implementações de observadores que aguardam um evento ser realizado antes de dar continuidade com o *script*. *Watchers* são fundamentais para trazer estabilidade aos testes, uma vez que eventos

podem ser mais lentos do que esperado, quando por exemplo ocorrem problemas em rede ou no servidor que hospeda a aplicação a ser testada.

Logo no início da configuração do *WebdriverIO*, uma série de ferramentas são oferecidas de acordo com a necessidade. Por meio de um arquivo de configuração, chamado *wdio.conf.js*, é possível definir os parâmetros de execução no teste, como por exemplo na figura 4.12.

```
exports.config = {
    // Runner Configuration
    runner: 'local',
    hostname: hostname,
    port: 4444,

    // Specify Test Files
    specs: ['./test/**/specs/**/*.js'],
    exclude: ['./test/atlas/aces/specs/equipmentDetails.js'],

    // Capabilities
    capabilities: [{
            maxInstances: 5,
            browserName: 'chrome'
        },{
            maxInstances: 5,
            browserName: 'firefox',
    }],

    // Test Configurations
    logLevel: 'warn',
    outputDir: './logs',
    bail: debug ? 2 : 0,
    waitforTimeout: debug ? 30000 : 10000,
    framework: 'mocha',
    reporters: ['spec'],

    // Hooks
    // Saves a screenshot to a given path if a command fails.
    screenshotPath: './errorShots/',
    afterTest: function takeScreenShot(test) {…
    },
};
```

Figura 4.12: Exemplo de arquivo de configuração do *WebdriverIO*.

Analisando a imagem do arquivo, inicialmente em *Runner Configuration* é possível explicitar em que máquina os testes serão rodados, podendo ser local ou remotamente. *Specify Test Files* permite especificar quais testes executar e quais ignorar. Por *Capabilities* é viabilizado balancear em quantas instâncias de quais navegadores os testes executarão em paralelo. *Test Configurations* define configu-

rações gerais, como verbosidade de *log*, tempo máximo de duração do teste com os *timeouts*, tolerância com testes falhos em *bail*, e escolha de ferramentas adicionais, como *framework* auxiliares de testes e *reporters* que geram os relatórios finais. Por fim, há os *hooks*, que permitem realizar tarefas adicionais durante o ciclo de execução, como por exemplo tirar uma *screenshot* da página ao fim de um teste quando ocorre um erro.

WebdriverIO possui embutido recursos avançados de escrita de código. Por exemplo o *tradutor Babel* para conversão de código, permitindo o uso das funcionalidades mais recentes do *Javascript*, e o validador *typescript*, que permite tipificar a linguagem. O *framework* também é compatível com serviços de execução de testes em nuvem, como o *Sauce Labs*, e depuração avançada, com a interface *REPL* que permite inspecionar o teste no momento de execução.

Com essa alta capacidade de customização e integrações disponíveis, foi preparado um ambiente inicial para uso da ferramenta, e testes passaram a ser escritos. O crescimento do número de testes e de funcionalidades compartilhadas, deu início a um novo *framework*, que será mencionado adiante, construído a partir do *WebdriverIO* e voltado para interfaces em *Fence*.

Implementação e *design* do *Fate*

O *framework Fate*, segue princípios próximos dos utilizados no *FUnit*. Um híbrido de paradigmas funcional e orientado a objetos é utilizado, com também aplicação dos princípios *SOLID* e abstração em arquivos de configuração.

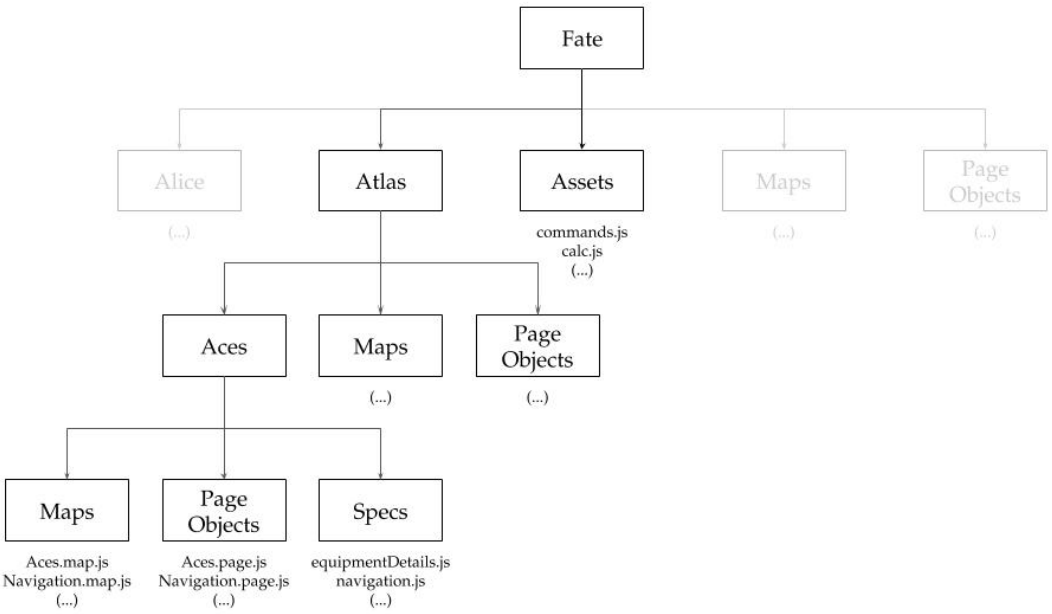

Figura 4.13: Estrutura de arquivos do *Fate*, categorizados basicamente em *Specs*, *Page Objects* e *Maps*. À medida que se ramifica, os arquivos abaixo possuem acesso às propriedades dos acima.

Como ilustrado na figura 4.13, a estrutura que o *Fate* possui é hierarquizada em diretórios. No topo, há a divisão em experimentos, *Atlas* e *Alice*, e a existência do diretório *Assets*, onde se encontram bibliotecas globais com funções que auxiliam a escrita de testes, como comandos adicionais para simular ações comuns de usuários nos sistemas *Fence* e operações matemáticas regulares nos testes.

Dentro do diretório *Atlas*, é possível ver uma segunda divisão baseada em sistemas, como o *ACES*. No interior do diretório do sistema, é possível ver os diretórios *Maps*, *Page Objects* e *Specs*. Em *Specs*, se encontram os testes que serão executados, executando as interações desejadas com a interface a ser testada. Estas interações são disponibilizadas pelos *Page Objects*, que se responsabilizam em conter a lógica e as interações disponíveis para cada interface disponibilizada para teste. Os arquivos *Maps* armazenam informações adicionais de configuração do objeto, como localizadores, que serão utilizados pelos *Page Objects*.

Essa organização se baseia em um padrão de *design* chamado de *page object pattern*. Esse *design* defende a ideia de cadeia de responsabilidades, na qual se separa o que precisa ser testado de como precisa ser testado. Isso torna mais simples o reaproveitamento de código entre testes, principalmente quando possuem páginas web em comum. Portanto o *page object pattern* recomenda a implementação de um objeto que represente a sua página, possuindo propriedades e métodos próprios que definem as possibilidades de interação pelos testes. Dessa forma, não é de responsabilidade do teste em implementar a lógica de interação, cabendo apenas importar o objeto *page* o qual deseja testar e utilizar as interações disponíveis. Na próxima seção será ilustrado um caso de uso desta estratégia.

É frequentemente utilizado no *Fate* o princípio da herança. *Page Objects* e *Maps* são definidos ao longo de toda a hierarquia do *framework*, de forma que à medida que se desce um nível de diretório, *Page Objects* e *Maps* deste nível herdam as funcionalidades dos respectivos diretórios pais. Este é um princípio inspirado no *framework Fence*, que também realiza a mesma prática. Portanto, como visto no exemplo da figura 4.13, *Page Objects* e *Maps* possuem acesso as funcionalidades adicionais dos diretórios de mesmo nome presentes em *Atlas* e na raiz do *Fate*.

Tecnologias e caso de uso

Em relação às tecnologias utilizadas, o *Fate* possui um conjunto abrangente. Além do *WebdiverIO*, sua arquitetura faz uso também de *frameworks* e bibliotecas adicionais, como presente no fluxograma da figura 4.14. Diferentemente do *FUnit* que depende unicamente do *phpunit*, na linguagem *Javascript* há um número maior de opções que permitem uma maior customização da aplicação.

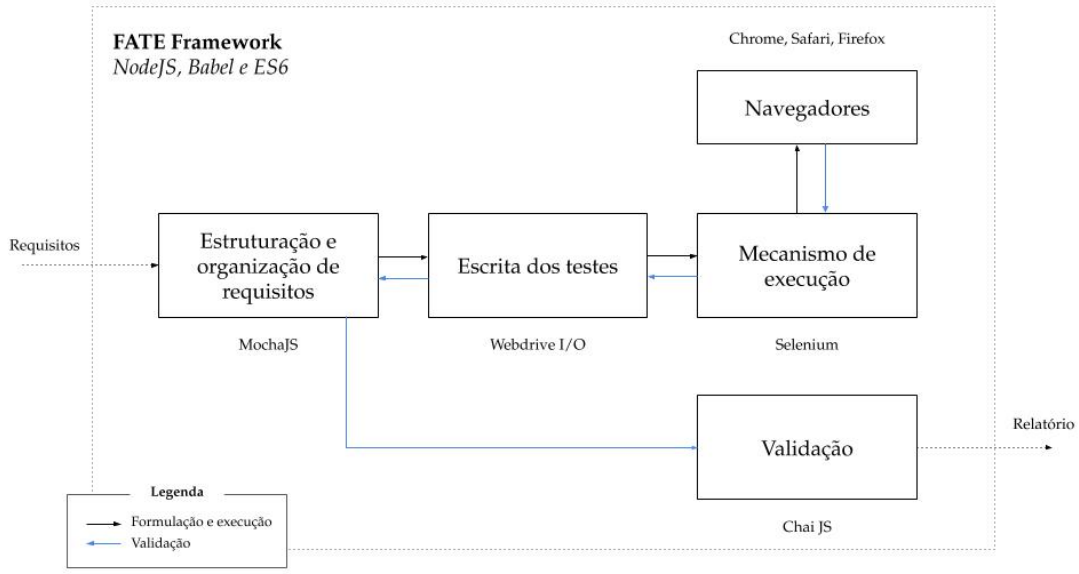

Figura 4.14: Funcionamento detalhado do *Fate*. Os testes são escritos, executados e validados após a estruturação dos requisitos.

Inicialmente os requisitos são expressados no formato exigido pelo *framework Mochajs*, que tem como responsabilidade estruturar e administrar a execução de testes. Ele associa os requisitos em *test cases*, aciona a rotina em *WebdriverIO* respectiva a cada *test case*, compara o resultado obtido com a biblioteca de validação, e finalmente invoca o gerador de relatórios escolhido. Na figura 4.15, há a página web *navigation* do sistema *ACES*, que possui a esquerda uma lista de itens utilizados no exemplo de teste a seguir.

Figura 4.15: Captura de tela do sistema *ACES*, com a lista de itens à esquerda, que são o objeto de exemplo do teste.

O requisito extraído do plano de testes é descrito no *test case* da *Spec navigation*, como no exemplo da figura 4.16, em que quando se clica na página em um item aleatório da lista, o título da página deve ser atualizado com o nome do item. Em seguida, se dá início a escrita do teste, onde é utilizado o *Page Object Navigation* para requisitar as interações necessárias para o prosseguimento do teste.

```
it('Should display zone title when clicked in list', () => {
    Navigation.collapseDefaultTree();
    const zoneName = Navigation.goToRandomZone();

    const title = browser.getTitle();
    assert.equal(title, `ACES | ${zoneName}`);
});
```

Figura 4.16: *Spec* realizando os pedidos ao *Page Object* e validando.

No conteúdo do *Page Object*, na figura 4.17, é possível ver a definição da série de passos necessários para atender o pedido do *Spec*. No método *collapsedDefaultTree* é pedido ao *WebdriverIO* que aguarde a existência do elemento *htCollapsedNode*, que contém a lista de itens, até o limite de tempo *timeout*, acusando erro se ultrapassado. Após isso, é enviado o comando *click*, que representa o clique de um *mouse*, para colapsar a lista. O *WebdriverIO* recebe estes comandos, traduz e envia para o *Selenium* por meio da *API*, que realiza as respectivas ações no navegador, como por exemplo *Chrome*, *Firefox* ou *Safari*.

```
class Navigation extends Aces {

    collapseDefaultTree() {
        $(navigation.ui.htCollapsedNode).waitForExist(
            timeout.loadHttps,
            false,
            'Navigation Page was not loaded'
        );

        $(navigation.ui.treeActiveMinus).click();
    }
    goToRandomZone() {
        const randomIndex = Calc.getRandomByCeil(this.countAllZones);
        this.zoneByIndex(randomIndex).click();

        return $(navigation.ui.activeZoneAnchor).getText();
    }
}
```

Figura 4.17: *Page Object* contendo a lógica de ações disponíveis na interface.

Realizadas as ações, a execução é retomada no *Page Object*, que retorna o fim da interação para o *Spec* original. É válido notar também que ao longo do código do *Page Object Navigation*, diversas propriedades do objeto *navigation.ui* são utilizadas, as quais pertencem ao arquivo *Maps*. Este arquivo, ilustrado na figura 4.18, funciona como um dicionário, fornecendo informações de localização de elementos na página e seus atributos.

```
export default {
    ui: {
        htCollapsedNode: 'ul.node-collapsed',
        treeActiveMinus: '#tree-fa-minus'
    }
};
```

Figura 4.18: *Maps* com a localização dos elementos usados pelo *Page Object*.

Finalizadas as interações desejadas com a página, tem-se a validação na última linha da *Spec*. No exemplo mostrado, é utilizado o método *assert.equal*, disponibilizado pela biblioteca *Chaijs*. Esta biblioteca permite diversos tipos de validações para diferentes estruturas de dados, facilitando as verificações do teste. No caso da *FUnit*, as validações disponíveis no *phpunit* são mais limitadas, tendo sido necessária a implementação de mais opções de validação.

Este processo descrito é repetido para cada *test case* do arquivo *Spec*, e a partir do momento em que todos os arquivos são executados ou interrompidos em caso de erro, um relatório é então gerado. Atualmente são utilizados três tipos diferentes de relatórios, um para cada situação, exemplificadas na figura 4.19. O relatório *Dot Reporter* é compacto e usado principalmente para verificações rápidas de regressão enquanto se desenvolvem novos testes. No caso do relatório *Spec Reporter*, ele é utilizado principalmente durante as *pipelines* da integração contínua, e fornece informações mais detalhadas sobre a execução dos testes também em linha de comando. E finalmente há o relatório *Allure Reporter*, que se trata de uma página web e possui uma análise gráfica criteriosa dos testes, sendo utilizado principalmente para situações mais formais, como apresentações e auditorias.

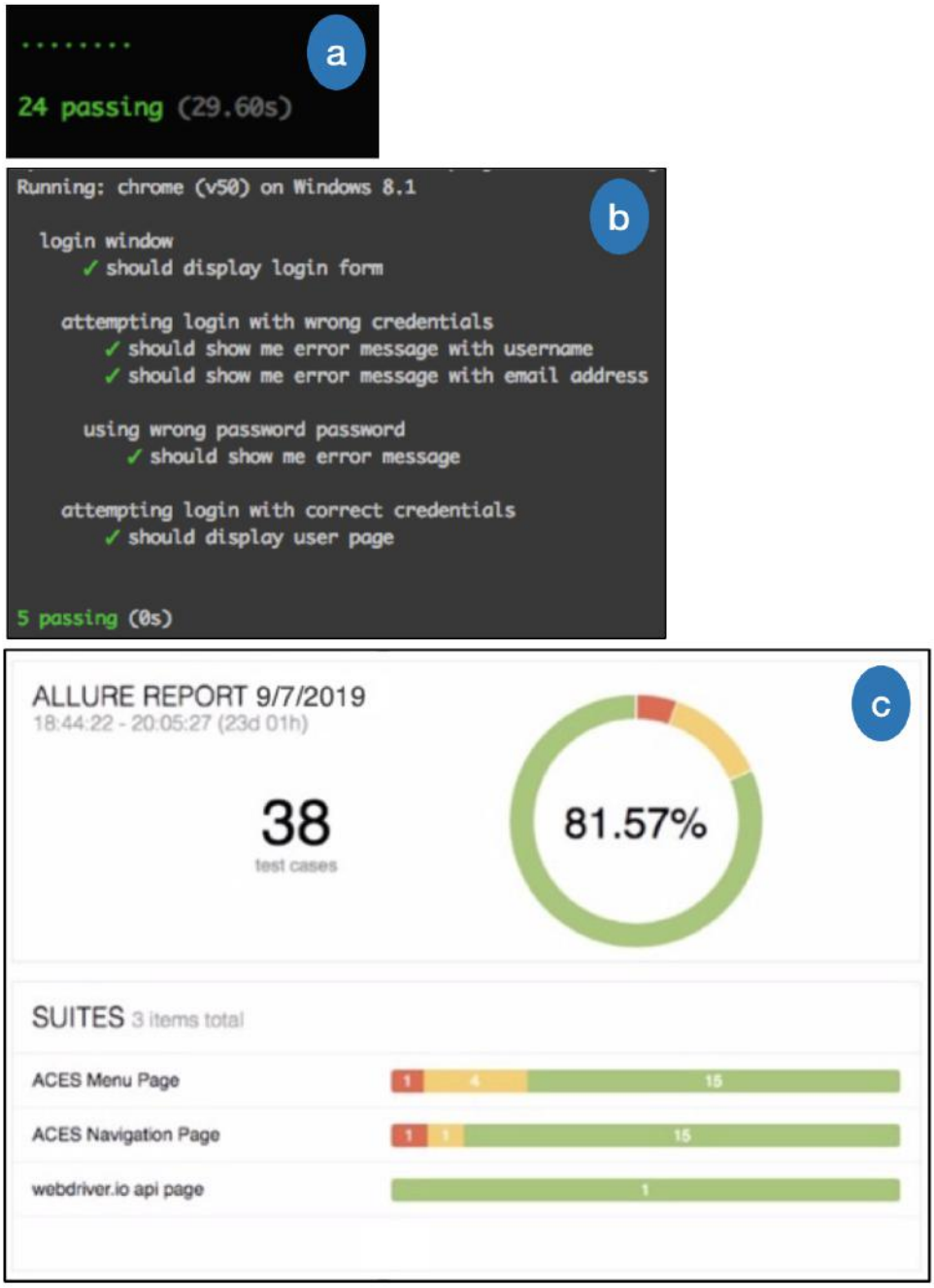

Figura 4.19: Exemplos de relatórios gerados pelo *Fate*. *Dot* em (*a*), *Spec* em (*b*) e *Allure* em (*c*).

Teste de regressão visual

Com a disponibilização recente da nova ferramenta *ResembleJS* pelo *Webdriver-rIO*, se tornou possível também a realização de testes de regressão visual. Estes testes realizam comparação direta de imagens, o que é útil em certos casos, apesar de serem testes que exigem mais processamento. O comportamento é semelhante ao da ferramenta *Sikuli* mencionada anteriormente. Entretanto, diferentemente do *Sikuli*, a integração com o *ResembleJS* já é nativa e compatível com o ambiente de

desenvolvimento do *Fate*, exigindo menos esforço para ser utilizada.

A figura 4.20 disponibiliza um exemplo de seu funcionamento. Na imagem extraída da página de *navigation* do *Aces*, é possível observar um desenho com pequenos retângulos posicionados ao seu redor. Estes retângulos são chamados de *racks* e representam painéis de eletrônicos posicionados fisicamente em uma seção transversal do detector *Atlas*, no subsolo do *CERN*. Estes *racks* não podem ser movimentados ou removidos, portanto, um teste de regressão visual foi desenvolvido para eles. Ao remover o *rack Y.M1-05.XA* e executar o teste, é possível ver que o teste destaca a diferença entre a imagem de referência e a obtida, acusando o erro. Assim é possível garantir a confiabilidade do mapa em questão.

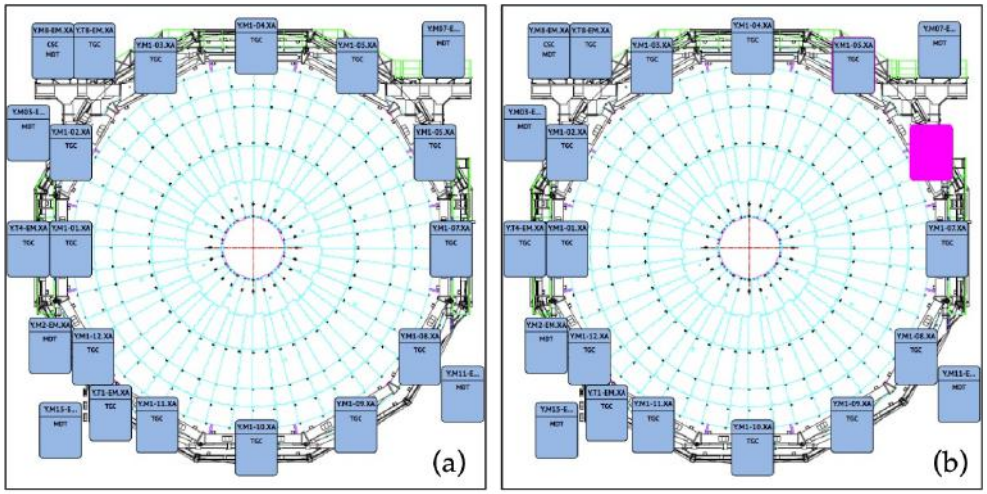

Figura 4.20: Exemplo de teste de regressão visual. O teste identificou a ausência de um *rack* na área em rosa.

O desenvolvimento dos testes unitários utilizando *FUnit* e dos testes *e2e* com o *Fate* mostrou a importância da elaboração de um ambiente de execução automática conforme a demanda. Isso motivou o desenvolvimento da integração contínua para os *frameworks*, que será discutida na próxima seção.

4.3 Ambiente Fence CID para integração de código

Com o desenvolvimento dos *frameworks FUnit* e *Fate* para escrita de testes automáticos, constatou-se a necessidade de um ambiente de gerência automática destes testes. Nesta seção será discutida a solução implementada com integração e entrega contínua, descrevendo de que formas poderiam ajudar no processo de trabalho dos desenvolvedores do grupo. A seção também discute como esta solução foi implementada, incluindo seus casos de uso, e a necessidade de aprimoramento para escalabilidade em sistemas distribuídos.

Processo de entrega de software

Ao realizar a entrega de uma nova versão para produção, chamada de *deploy*, uma série de processos são necessários seguindo o processo de trabalho do grupo. Ao estar satisfeito com a nova versão do código desenvolvido, o desenvolvedor realiza um *merge request* para a *branch stage* pelo *git*, e quando aprovado, se torna necessário realizar um *deploy* para o ambiente de mesmo nome *stage*.

O ambiente *stage* possui a responsabilidade de reproduzir o ambiente de produção, com a motivação de realizar uma verificação final da nova versão antes de ser colocada em produção. Este processo inclui realizar a compilação da aplicação, chamado de *build*, que consiste em vários comandos, como instalação de dependências.

No processo de trabalho antigo, esta verificação em *stage* era feita de forma manual pelo desenvolvedor. Além disso, caso o desenvolvedor quisesse realizar testes automáticos, o próprio tinha que verificar se os ambientes de execução da *FUnit* e *Fate* estavam adequados e atualizados. No fim, o processo poderia levar uma quantidade significativa de tempo que em vários casos o desenvolvedor não possuía disponível, especialmente em situações de correções críticas de erros em produção.

A fim de atender a necessidade de uma verificação geral e rápida, que realizasse o processo de *build* da aplicação, execução de testes a partir deste *build* e finalmente o *deploy* para o ambiente de testes *stage*, foi proposta o desenvolvimento de um ambiente de integração contínua.

Integração contínua de código

Nomeado de *Fence Continuous Integration & Delivery*, ou simplesmente *Fence CID*, esta solução atua na integração do código dos desenvolvedores do grupo, executando as ações necessárias no controle de qualidade quando ocorrem mudanças na plataforma *Gitlab*, o hospedeiro *git* escolhido pelo grupo para controle de versão.

O *Fence CID* foi implementado a partir do *Gitlab Continuous Integration & Delivery*, abreviado como *CI/CD*, que é disponível automaticamente para qualquer repositório desenvolvido usando *Gitlab*. Com o *Gitlab CI/CD* é possível monitorar diferentes etapas no controle de versão do código, podendo atribuir tarefas de acordo com cada etapa, e no final realizar o processo de *deploy*. No caso dos repositórios *git* do grupo, dois fluxos de execução principais, *pipelines*, foram desenvolvidos, sendo uma para o repositório do *framework Fence* e a outra para o repositório *Atlas*, que contém a aplicação com os sistemas que fazem uso do *Fence*. Cada *pipeline* executa os respectivos *jobs* necessários para seu cumprimento, podendo estes serem executados em série ou em paralelo, de acordo com a etapa definida em cada *job*. Na ausência de erros após a execução dos *jobs*, a *pipeline* é qualificada como bem-sucedida.

A descrição de *jobs* e da própria *pipeline* é definida por meio de arquivos de configuração *YAML*, que serão utilizados pelo *Gitlab CI/CD*. Semelhante aos arquivos *JSON*, arquivos *YAML* permitem a serialização de dados por meio de conceitos como escalares, sequências, mapeamentos e âncoras.

Um ponto importante no uso do *Gitlab CI/CD* é a alocação de máquinas disponíveis para executar as *pipelines*. Para uma máquina se tornar disponível é necessário que possua o *Gitlab Runner* instalado, que se trata de um binário capaz de executar as tarefas recebidas pelo servidor principal do *Gitlab CI*. Atribuindo *tags* aos *Gitlab Runner* é possível diferenciar quais máquinas se responsabilizarão por quais tarefas.

Funcionamento de uma *pipeline*

Ao abrir ou concluir um *merge request* no repositório *Fence*, se inicia a execução da *pipeline*, como exposta na figura 4.21. Pode-se observar a existência da *pipeline* com três estágios definidos, sendo eles o *Build*, o *Test* e o *Deploy*. Cada estágio possui apenas um *job*, significando que toda a *pipeline* é executada em série, sem paralelismo. Por exemplo o *job unit test*, responsável pelos testes unitários da *FUnit*, será executado apenas se o *job* anterior, *build*, tiver sido realizado com sucesso, notificando os desenvolvedores por e-mail e *Slack* caso contrário.

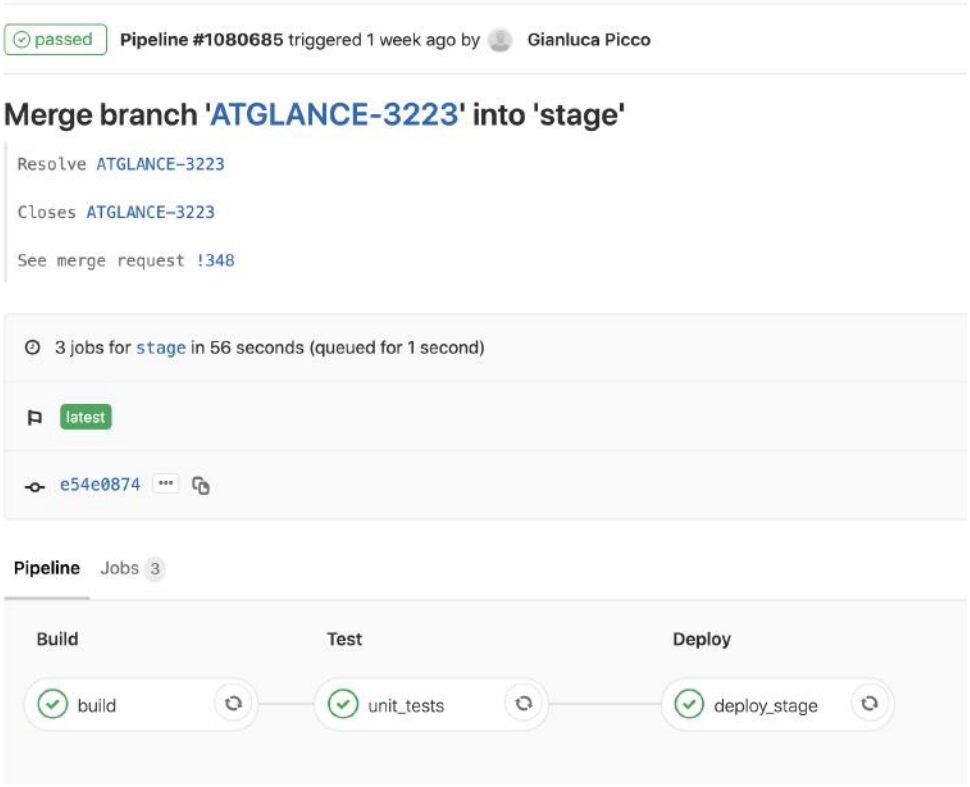

Figura 4.21: Exemplo de execução de uma *pipeline*, contendo as etapas *Build*, *Test* e *Deploy*, com um *job* em cada.

Entrando no universo do código, na figura 4.22(a) tem-se o detalhamento do *job build* no arquivo *YAML*. Decompondo o arquivo de acordo com seus parâmetros, é possível compreender seu funcionamento. *Stage* define a etapa em que o *job* será executado na *pipeline*, como na etapa *Build* de mesmo nome. Em *only* são definidas as condições para execução do *job*, que neste caso ocorre apenas em *merge requests*. O parâmetro *tags* se responsabiliza em definir quais máquinas podem executar o *job*, que neste caso é o servidor *stage* dos sistemas. Por fim têm-se *before_script* e *script*, os quais contêm a lógica a ser executada pelo *job*, sendo neste exemplo os passos necessários para compilação.

Da mesma forma que feita em *build*, um arquivo de configuração é escrito para cada *job* a ser executado. *Gitlab CI/CD* permite a utilização de configurações avançadas nestes arquivos, que foram amplamente utilizadas na construção das *pipelines* do projeto, como extensão de parâmetros, disponibilidade de variáveis globais durante a execução da *pipeline* e repetição do *job* em caso de falha. Por fim, os arquivos de configuração são unificados em um arquivo único, chamado *gitlab-ci.yml* e ilustrado na figura 4.22(b), que será automaticamente lido pelo *Gitlab CI/CD*.

(a)

```
build:
  stage: build
  only:
    refs:
      - merge_requests
  tags:
    - glance
  before_script:
    - cd /srv/release/ci-builds/$CI_PROJECT_NAME
    - git checkout $CI_COMMIT_REF_NAME
    - git pull origin $CI_COMMIT_REF_NAME
  script:
    - npm ci
    - composer validate --no-check-all --strict
    - composer install
    - npm run build
```

(b)

```
stages:
  - build
  - test
  - deploy

include:
  - local: '/source/ci-jobs/compile.yml'
  - local: '/source/ci-jobs/build.yml'
  - local: '/source/ci-jobs/unit_tests.yml'
  - local: '/source/ci-jobs/e2e_tests.yml'
  - local: '/source/ci-jobs/deploy_stage.yml'
```

Figura 4.22: Arquivo de configuração do *job build* em (*a*) e o arquivo principal, da *pipeline*, em (*b*).

No caso do repositório *Atlas* a *pipeline* é bem similar, com algumas ressalvas. Como a aplicação depende do *Fence* para o seu funcionamento, é interessante que a *pipeline* do *framework* seja executada juntamente com a do *Atlas*, podendo inclusive ser paralelizado. Como pode ser visto na figura 4.23, a *pipeline* do *Atlas* invoca a *pipeline* do *Fence* e ambas são executadas em paralelo, rodando os respectivos *jobs* das etapas de *Build* e *Test*. Pode-se notar também a existência do *job e2e_tests*, responsável pela execução do *Fate*.

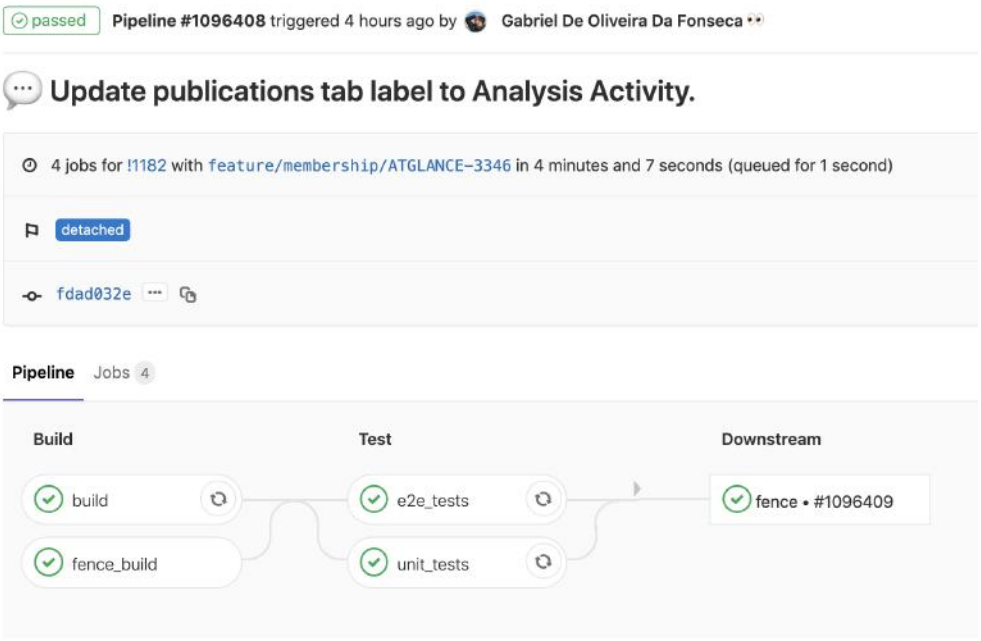

Figura 4.23: Ilustração da integração entre as *pipelines* do *Atlas* e *Fence*.

Implementada a integração e entrega contínua, a atenção voltou-se para outra questão. Como visto anteriormente nesta dissertação, enquanto testes unitários são rapidamente executados dada a sua simplicidade, o acúmulo de testes *e2e* pode gerar lentidão na execução de testes. Por conta disso, um esforço foi feito a fim de paralelizar estes testes, sendo abordado na próxima seção.

Orquestração em *containers*

Ao buscar estratégias para execução de testes em paralelo do *Fate*, alguns desafios foram considerados. Primeiramente, para a execução em paralelo ser possível, seria necessário a disponibilidade de múltiplas instâncias de sistemas operacionais, como as presentes em um *cluster*. Em seguida, estas instâncias precisariam ser capazes de executar os testes, contendo todas as dependências necessárias. Isso inclui softwares, como *git* e *Node.js*, e pacotes *npm*, como o *WebdriverIO* e o *Mochajs*. Por fim, estas instâncias precisariam se comunicar, de forma que fosse balanceada igualmente a execução dos testes. Assim que uma instância finalizasse a execução de um teste, ela se tornaria disponível para receber outro.

Para a solução da disponibilidade de instâncias e máquinas, foi encontrada a possibilidade de utilizar recursos disponibilizados pela infraestrutura do *CERN*. Por ser um centro de pesquisas de alto porte, dispõe-se de máquinas já integradas com o *Gitlab CI/CD* para serem utilizadas em projetos. Estas máquinas são chamadas de *Runners*, e a figura 4.24(a) mostra um exemplo exibindo-as na interface do *Gitlab*.

Em relação a questão do ambiente das máquinas, a solução encontrada foi a adoção do *Docker*. *Docker* é uma plataforma famosa na comunidade, onde se torna

possível a hospedagem de aplicações em um ambiente isolado, chamado de *container*, em que o desenvolvedor consegue preparar o ambiente de execução do software de maneira padronizada. A plataforma possui uma performance superior à virtualização convencional e permite configurar diferentes ambientes com facilidade a partir do conceito de imagens, no qual é possível gerar um *container* a partir da representação de um outro *container* já existente. Isso permite o ambiente de uma aplicação ser mais modularizado e escalável.

Para começar a utilizar *Docker* um arquivo *Dockerfile* foi gerado para o repositório *Fate*, como ilustrado na figura 4.24(b), contendo os comandos de instalação para as dependências do *framework*. É então construída uma imagem a partir destes comandos, que é enviada para o repositório central de imagens *docker* do *CERN*. Esta imagem fica então disponível para outros repositórios hospedados no *Gitlab* poderem utilizar.

Uma *pipeline* foi implementada no caso do *Fate* , como pode ser visto na figura 4.24(c), em que é realizada uma verificação do próprio funcionamento do *framework* quando são desenvolvidos novos testes *e2e*. Estes testes, juntamente com os anteriores, são compilados e testados diversas vezes para garantir sua robustez. Sendo o resultado bem-sucedido, a imagem *docker* do *Fate* é atualizada e seu *upload* feito, podendo ser utilizada nos futuros *jobs* de *e2e tests* da *pipeline* do repositório *Atlas*. Pode-se dizer que o *Fate* testa os seus próprios testes.

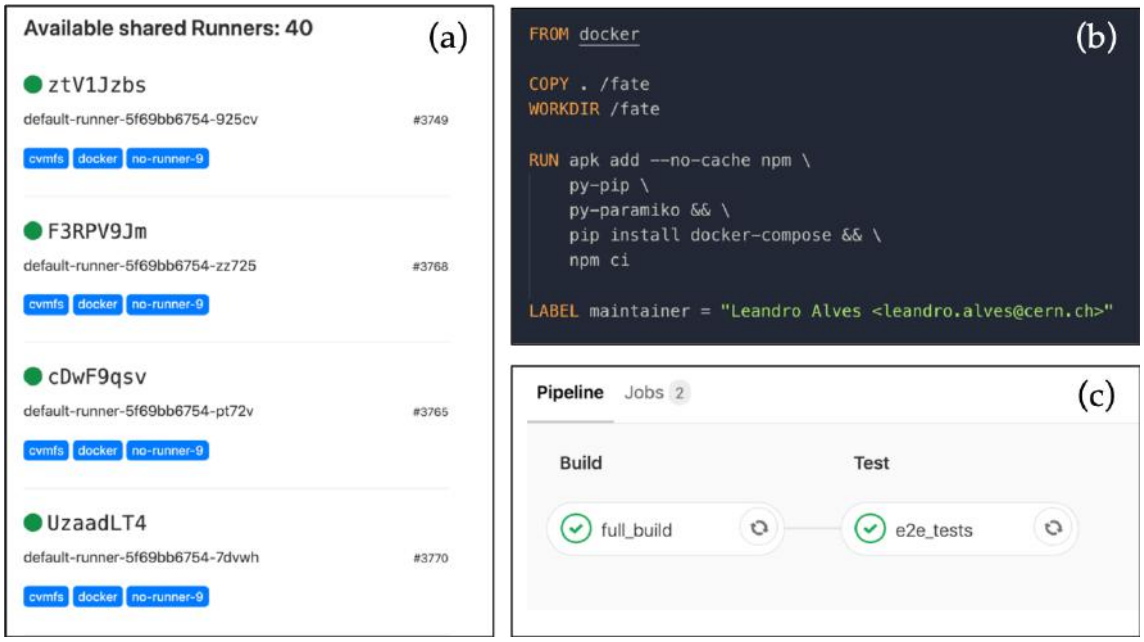

Figura 4.24: Em (*a*), lista de instâncias disponíveis. Em (*b*), exemplo de arquivo *Dockerfile* para geração de uma imagem. Em (*c*), a *pipeline* do *Fate*.

Obtendo as instâncias necessárias e um método eficiente para prepará-las, o desafio final passa a ser a intercomunicação e administração destes containers. Para

solucioná-lo, foi necessário utilizar em conjunto a ferramenta de orquestração *docker swarm*, o utilitário *docker compose* de configuração *multi-container*, e por fim o *Selenium Grid* para alocação dos testes entre os *containers*.

O *Selenium Grid* faz parte da família *Selenium*, juntamente com o *Selenium Webdriver* e o *Selenium IDE*, e permite a organização e preparação de diversas máquinas para execução testes. Por meio do conceito de *hub* e *nodes*, o *Selenium Grid* define uma máquina central, o *hub*, que será a responsável por gerenciar as máquinas que irão executar os testes, os *nodes*. O *hub* recebe o conjunto de testes a serem executados, acompanhado das informações de execução de cada teste, como por exemplo o sistema operacional ou navegador. A partir disso, o *hub* avalia a configuração de cada *node* presente na malha e aloca os testes conforme a disponibilidade dos *nodes* compatíveis com cada teste. Os comandos provenientes do *Fate* com o *WebdriverIO* são então enviados do *hub* para o *node*, que os executa e informa de volta para o *hub*. Finalizados todos os testes, o *hub* sintetiza os resultados para o *Fate*, o qual gera o relatório para a *pipeline*.

Para implementar essa arquitetura no projeto, foram utilizadas imagens *docker* oficiais do *Selenium Grid*. Estas imagens são *selenium/hub*, *selenium/node-chrome* e *selenium/node-firefox*, tornando disponível para teste os navegadores *Chrome* e *Firefox*, que são os mais populares pelos usuários dos sistemas. *Docker compose* é utilizado para dar as instruções de geração destas imagens, como por exemplo que portas serão utilizadas por cada *container*, sua ordem de construção, variáveis de ambiente, e o número de *containers* que serão implementados a partir de cada imagem. Todas estas instruções são definidas no arquivo *docker-compose.yml*, como ilustrado na figura 4.25, que é então utilizado pelo *docker swarm* para finalmente implementar múltiplos *containers* a partir dos *Runners* disponíveis no *Gitlab CI/CD* do *CERN*.

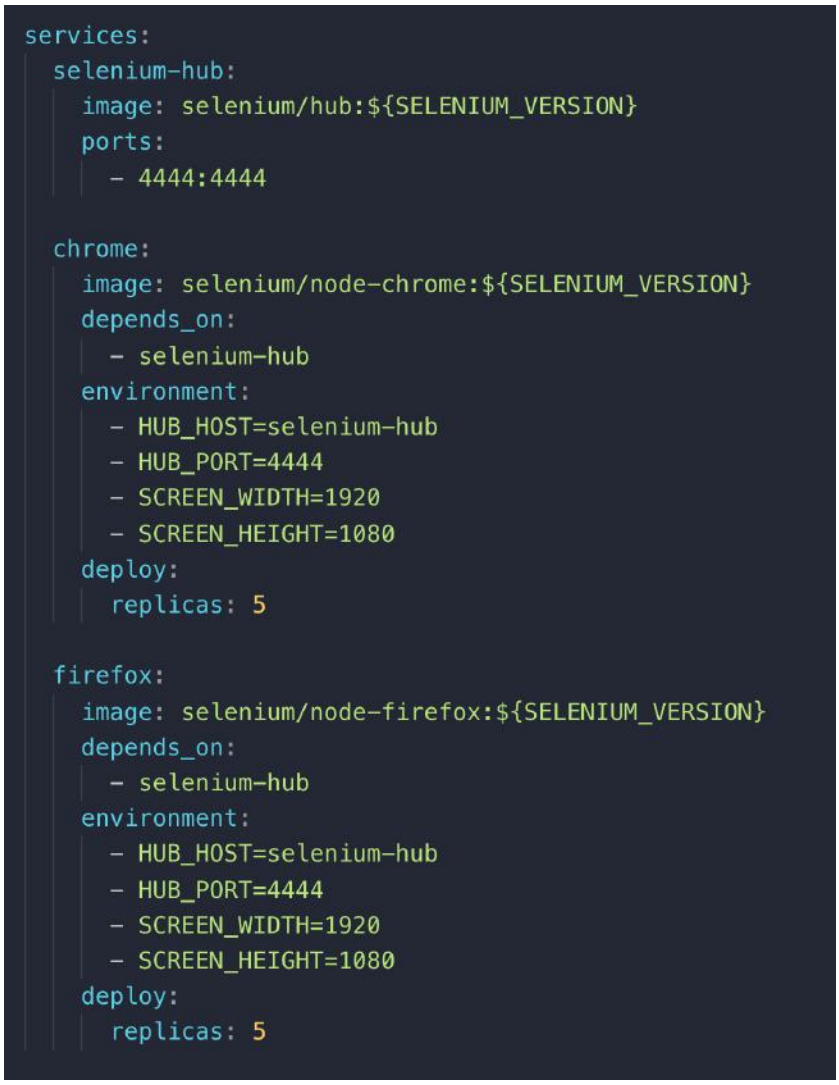

Figura 4.25: Arquivo de configuração do *docker compose*, contendo o balanceamento ente *hub* e *nodes*.

De forma resumida, no *job e2e tests* é utilizada a imagem *docker* do *Fate* que possui as instruções do orquestrador *docker compose*. Estas instruções são interpretadas e múltiplos *containers* são implementados a partir da disponibilidade de máquinas no *Gitlab CI/CD*. Estes *containers* fazem parte da rede do *Selenium Grid* que os dividem entre *hub* e *nodes*. O *container hub* gerência as execuções dos testes nos *nodes* e por fim coleta os resultados, que serão enviados no final para a *pipeline*.

4.4 Reflexão sobre o processo de trabalho

Ao longo do desenvolvimento da plataforma de testes, esforços adjacentes foram feitos com o objetivo de melhor enquadrar o grupo de desenvolvedores em relação a práticas ágeis. Apesar deste esforço ser contínuo e ainda existirem diversas práticas

ágeis a serem adotadas, é válido listar algumas já alcançadas.

Modelo Programação Extrema

Recentemente, o grupo de desenvolvedores da colaboração entre UFRJ e *CERN* vem tentando utilizar algumas práticas incentivadas pelos princípios ágeis. Condições existentes como mudança constante de requisitos e um grupo que trabalha constantemente próximo, viabilizam a adoção gradativa do modelo de *Extreme Programming*. Algumas práticas já são realizadas, como reuniões semanais que são feitas com o objetivo de priorizar as funcionalidades, pequenas versões do produto que são frequentemente liberadas, e testes de aceitação que são sempre incentivados.

Em relação a organização do grupo, reuniões em pé, conhecidas como *stand-up meetings*, são realizadas diariamente por todo os desenvolvedores, e a filosofia de posse coletiva é empregada internamente, na qual o código fonte não possui um único proprietário e todos têm os mesmos direitos em modificá-lo.

Práticas em pares também são encorajadas, com a recorrente *pair negotiation*, na qual o par discute a melhor solução para um problema, e a ocasional *pair programming*, onde a dupla implementa a solução juntos na escrita de código. O *pair programming* é particularmente importante na recepção de um novo desenvolvedor, na qual se apresenta a filosofia aplicada pelos desenvolvedores do grupo durante a escrita de código.

A implementação vem sendo constantemente aprimorada. A escrita de código vem sendo padronizada com os *coding standards* em conjunto com a adoção de ferramentas de verificação automática, os *linters*, que serão falados mais adiante. A *refatoração* é encorajada dentro do grupo, possibilitando a melhoria na clareza e na estruturação do código, e a integração contínua já é adotada nos principais repositórios. Recentemente tentativas têm sido feitas em implementar o desenvolvimento orientado a testes, conhecido como *TDD*, onde se elabora primeiramente os testes unitários para depois ser implementado o código referente a estes testes. Entretanto, dada a complexidade desta abordagem, esta prática tem sido exercida apenas de forma experimental.

Ferramenta de comunicação interna

Um dos quesitos que o grupo procura sempre melhorar é a comunicação entre os desenvolvedores. Apesar dos esforços colaborativos dos desenvolvedores do grupo em serem solícitos entre si, havia a falta de uma ferramenta destinada exclusivamente ao diálogo e notificação. Foram propostas duas ferramentas como tentativa de melhoria, o *Mattermost* e o *Slack*. Apesar do *Mattermost* ser a ferramenta recomendada no *CERN*, essa não possuía notificações para dispositivos móveis, inviabilizando-a para

o grupo. No fim foi decidida a adesão do *Slack*.

O *Slack* permite a distribuição de assuntos em diferentes canais que funcionam como *chats online*, como ilustrado na figura 4.26(a). Cada participante pode escolher os canais com os quais deseja se conectar e interagir por meio de mensagens normais ou *threads*. Os *threads* ajudam a discutir de modo mais aprofundado um tópico específico, sem prejudicar o fluxo principal do canal. Mensagens individuais também são possíveis, assim como chamadas por conferência, e formatações para escrita de código.

Uma importante característica presente no *Slack* é o suporte a comunicadores automáticos, conhecidos como *bots*, exemplificado na figura 4.26(b). Os *bots* são uma forma de trazer para o *Slack* comunicações automáticas de serviços externos. Comunicando-se por meio de *API*, se tornou possível a centralização das notificações de diversas ferramentas utilizadas pelo grupo em um só lugar. Estas notificações incluem o *Gitlab* para movimentações de código, o *Jira* para geração e rastreamento de entregáveis, e *Confluence* para documentos criados ou editados. Desta forma, torna-se mais fácil os desenvolvedores acompanharem as mudanças, mesmo estando trabalhando remotamente.

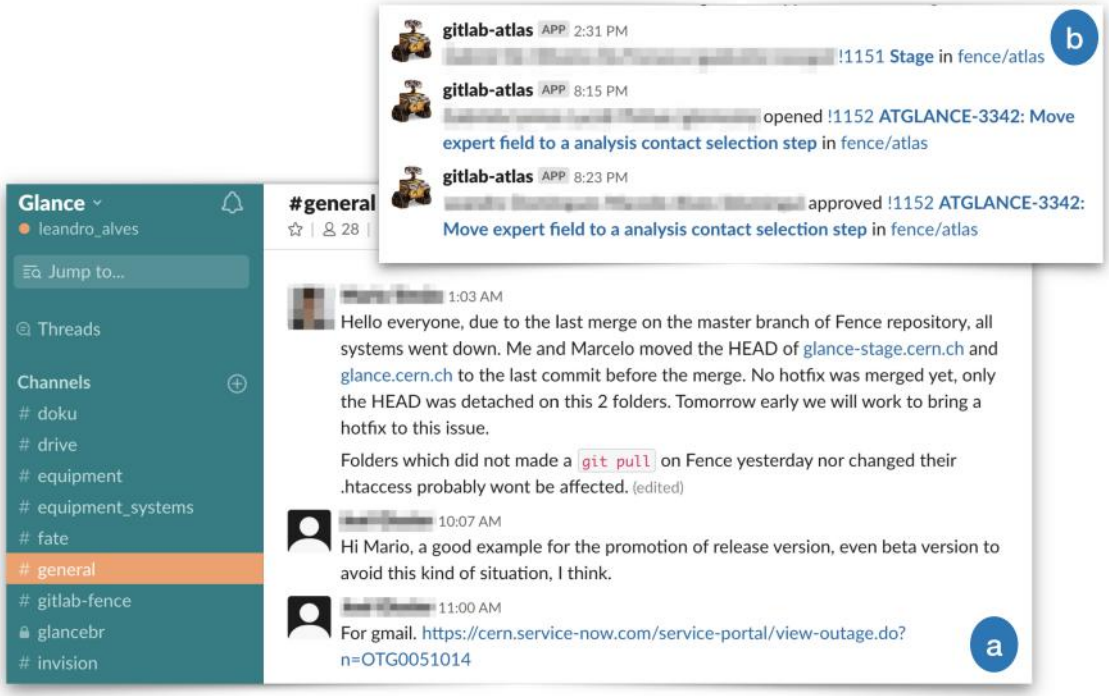

Figura 4.26: Exibição de um canal do *Slack* em (*a*). Exemplo de mensagens geradas por *bots* em (*b*).

Revisões de código

A contribuição mais relevante neste aspecto ágil foi a adoção da revisão de código pelo grupo, chamadas de *merge requests*. Os *merge requests* permitem a melhoria na

comunicação entre os desenvolvedores a nível de código. Eles foram implementados nos repositórios *git* do grupo, e se tratam de pedidos formais de adição de novo código às bases de código já existentes nestes repositórios. Ao realizar o pedido, o desenvolvedor deste novo código notifica por e-mail e *Slack* os outros desenvolvedores responsáveis, os quais podem avaliar o conteúdo deste novo código e aprovar para poder ser inserido.

É utilizada a ferramenta web *Gitlab* para gerenciamento em *git* dos repositórios. Esta ferramenta proporciona uma interface para revisão de *merge requests*, em que é possível realizar discussões sobre o novo código. Estas discussões podem inclusive ser apontadas em linhas de código, levantando dúvidas, sugerindo modificações ou alertando possíveis problemas, como presente na figura 4.27. Depois de abertas estas discussões, o autor do *merge request* pode solucioná-las ou transformá-las em *issues*, que são registros de tarefas a serem realizadas posteriormente.

Uma convenção foi também adotada para melhor organizar os novos códigos desenvolvidos. Antes de começar este trabalho, é esperado que o desenvolvedor crie antes uma *issue* no *Jira*, contendo uma descrição deste trabalho. O *Jira* é uma plataforma voltada a gerenciar os requisitos do grupo, e é a interface de contato com os usuários. Após a *issue* ser gerada e seguindo o padrão do *git flow*, o trabalho em código por um desenvolvedor é separado inicialmente em uma versão exclusiva, chamada de *branch*. Esta versão separada permite ao desenvolvedor não afetar o trabalho dos outros e não ser afetado, e a medida que o próprio avança por meio de *commits*, que são pequenas versões geradas durante a escrita de código, é possível que os observadores, ou *watchers*, possam acompanhar o andamento diretamente da *issue* no *Jira*. Essa integração traz o usuário para mais perto do desenvolvimento em si, na tentativa de estimular a sua contribuição.

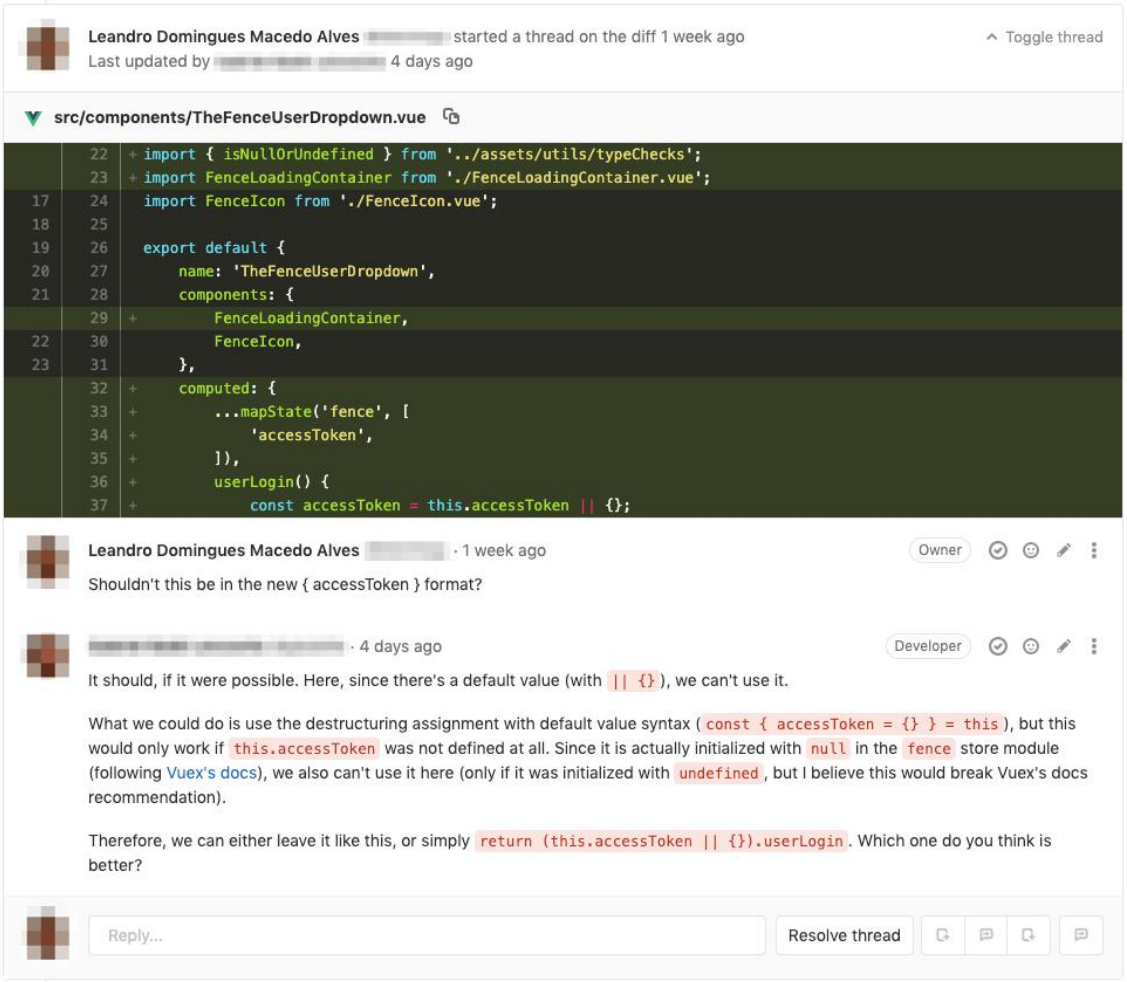

Figura 4.27: Exemplo de discussão de *merge request* a partir da interface web *Gitlab*.

Centralização da documentação

Sempre houve ao longo da existência do grupo de desenvolvedores a preocupação em documentar requisitos, funcionamento dos sistemas e o código fonte. Diversas ferramentas foram utilizadas, e migrações ocorreram com o passar dos anos, na tentativa constante de achar uma melhor opção, e isso consequentemente contribuiu para a dispersão da documentação entre diferentes plataformas. *Google Docs*, *Trac*, *DokuWiki* e *Twiki* estão entre elas, dificultando encontrar um documento específico, e ocorrendo a repetição de documentação. Se tornou necessário não somente achar uma ferramenta em que todo o grupo concordasse, como também realizar a migração para uma plataforma única.

Levando em consideração a satisfação do grupo com a utilização do *Jira*, optou-se experimentar pela ferramenta *Confluence*, ilustrada na figura 4.28(b). *Confluence* possui uma integração nativa com *Jira*, tornando mais fácil a referenciação a *issues* durante a documentação, como a elaboração automática de relatórios semanais do trabalho feito por cada desenvolvedor, ilustrado na figura 4.28(a). Além disso, a fer-

ramenta possui suporte a diversos *plugins*, permitindo a customização para atender as especificidades do grupo, como formatações para escrita de código e *templates* de atas de reunião. Após explorar a ferramenta, decidiu-se a sua adoção, e um esforço coletivo do grupo foi realizado em migrar a documentação preexistente e depreciar as antigas ferramentas.

No momento atual, o grupo se concentra em aperfeiçoar a documentação de código. Estão sendo experimentados utilitários de documentação automática para as principais linguagens usadas, como o *php*, com o *phpDox*, exemplificado na figura 4.28(b). O objetivo é integrar estas ferramentas no processo de trabalho com a integração contínua, e complementá-las ao *Confluence*.

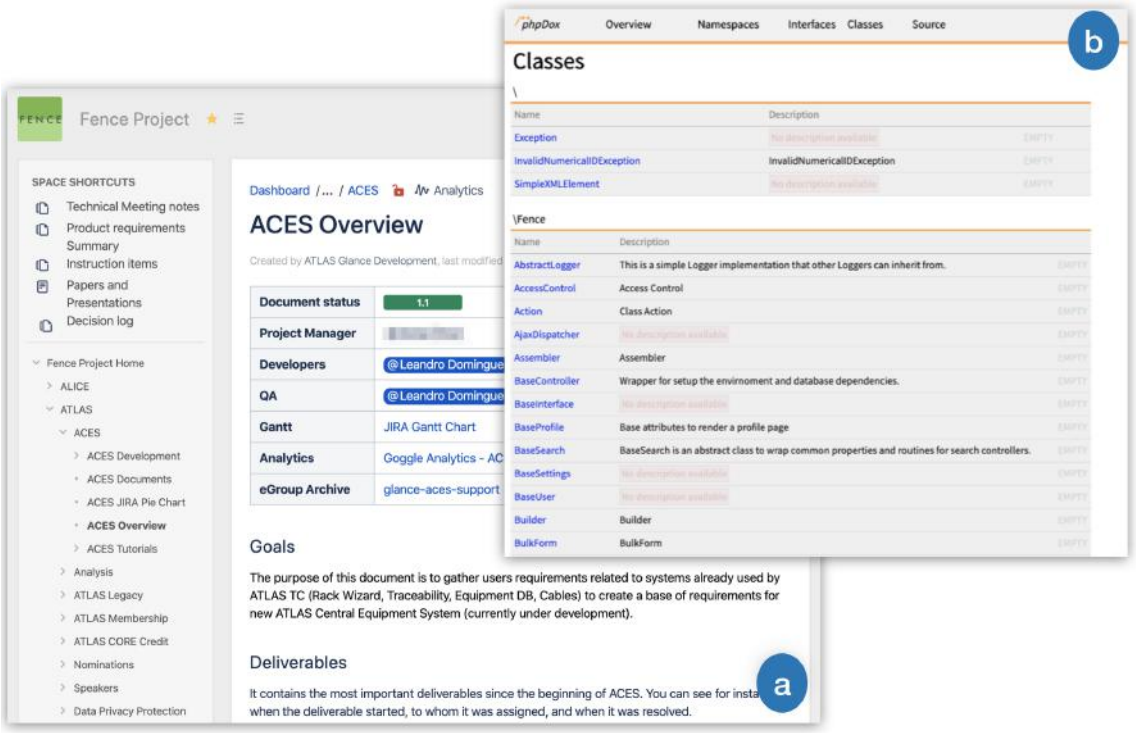

Figura 4.28: Exemplo de documentação web elaborada a partir do *Confluence* em (*a*). Documentação automática de código gerada pelo *phpDox* em (*b*).

Por fim, foi implementada uma integração automática entre as ferramentas de comunicação. Um documento elaborado no *Confluence* possui acesso livre a *issues* disponíveis na ferramenta de administração de requisitos e inconsistências *Jira*. Cada *issue* por sua vez possui referências ao trabalho feito em código para sua solução, com a integração à plataforma de controle de versão *Gitlab*. Todas estas ferramentas enviam notificações por meio de *bots* aos canais respectivos no *Slack*, aonde todos os desenvolvedores possuem acesso.

Validadores de sintaxe

A rotatividade de pessoas sempre foi uma característica marcante dentro do grupo de desenvolvedores. Por conta desta alternância, a possibilidade de padronização de práticas sempre foi algo desejado para o grupo, inclusive em relação ao processo de escrita de código. Em geral, ocorre uma transmissão de conhecimento entre os desenvolvedores em relação a boas práticas de código, porém isto pode ser insuficiente e consumir tempo.

Levando esta circunstância em consideração, foi proposta a elaboração de padrões de escrita de código, chamados de *code standards*. Estes padrões são definidos em arquivos de configuração presentes nos repositórios do grupo e foram gerados para as linguagens mais utilizadas pelo grupo, sendo estas o *php* e o *Javascript*.

Por meio dos *code standards* passaram a ser utilizados pelo grupo em seus editores de código as ferramentas adicionais de análise chamadas *linters*. Os *linters* são capazes de aplicar as regras de *code standards* durante a escrita de código, notificando o desenvolvedor de regras definidas pelo grupo e aconselhando-o de boas práticas. Desta forma, o programador ingressante tem a possibilidade de obter avaliações instantâneas sobre a qualidade de seu código, reduzindo a possibilidade de reescrita no futuro.

Atualmente são oficialmente utilizados quatro *linters* na análise de código, com mais duas em fase de experimentação. Para *Javascript*, somente o *eslint* é utilizado para avaliar erros de sintaxe e compatibilidade com os padrões decididos. Já para *php*, são utilizados o *phpcs*, também para regras do grupo, o *phpmd*, para principalmente análise de complexidade de métodos e classes, e o próprio validador do *php*, para identificar antecipadamente potenciais erros de sintaxe. Estes *linters* estão atualmente sendo utilizados nos editores de texto *Sublime Text* e *Visual Code*, mas também podem ser executados de forma autônoma pela linha de comando do terminal. Na figura 4.29 há um exemplo de código sob revisão de *linters*, de forma que as linhas que apresentam inconsistências são sublinhadas e estes erros são exibidos no compilador, contendo informações de como consertá-los.

Para garantir a consistência do uso dos *linters* pelo grupo, foi disponibilizada também uma ferramenta de verificação automática chamada *lint staged*. Independente se o desenvolvedor fez uso de *linters* ao programar, o *lint staged* é aplicado diretamente a um *git hook*, de forma que estes *linters* são executados durante o processo de *commit*. Caso haja algum problema, o *commit* é cancelado, informando as inconsistências ao desenvolvedor. Assim, evita-se a submissão por acidente de código que esteja fora do padrão do grupo ou até mesmo com erros de sintaxe.

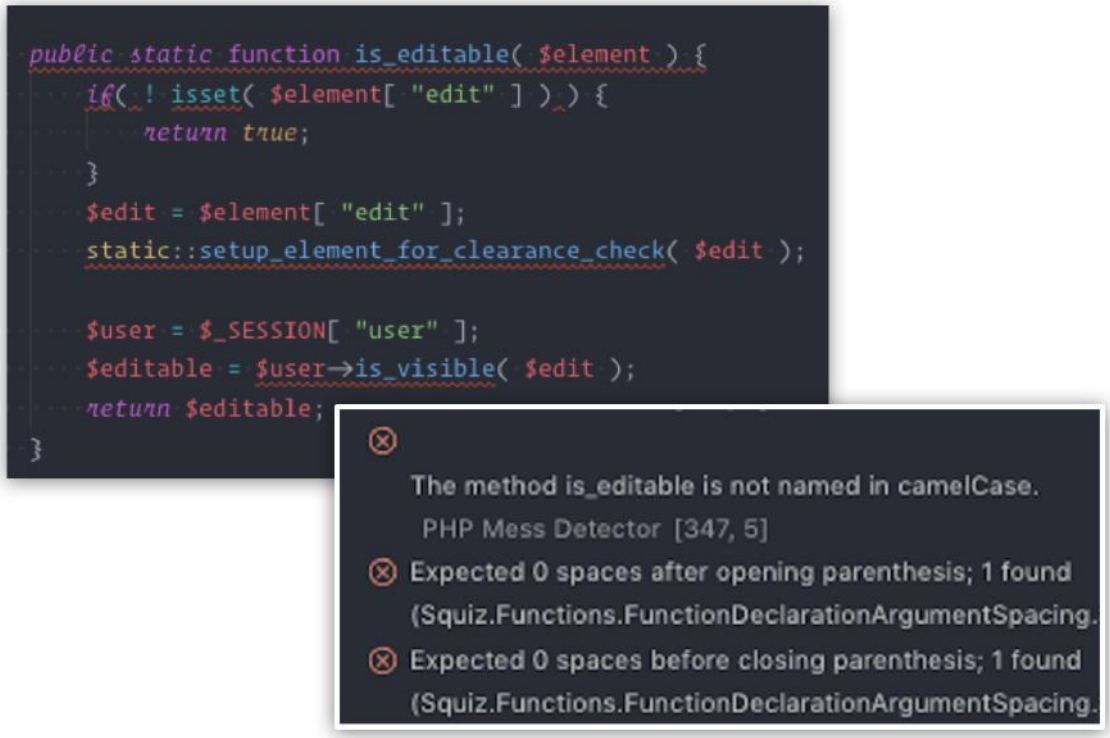

Figura 4.29: Exemplo de alertas gerados por *linters* no editor de código *Visual Code*.

Plano de Testes

Os desenvolvedores do grupo têm realizado a prática de planos de testes ao longo dos últimos anos, acumulando dezenas de documentos com milhares de testes. Além de contribuir para a clareza do desenvolvedor em saber o que tem que ser testado, o plano de testes também auxilia na documentação dos requisitos e na compreensão do sistema. É uma prática do grupo apresentar os planos de testes dos sistemas aos ingressantes, convidando-os a realizar testes exploratórios e de usabilidade no ambiente de desenvolvimento.

Considerando a importância do plano de testes, foi realizada uma pesquisa em busca de possíveis aprimoramentos em seu processo de desenvolvimento e manutenção, que no momento consistia em serem realizados em planilhas no *Google Sheets*. Nesta pesquisa, encontrou-se uma ferramenta alternativa que se enquadra nas condições do grupo, chamada *Lean Testing*. Semelhante ao *Google Sheets*, o *Lean Testing* permite o agrupamento de dados em diversas categorias, como *test cases* e *test suites*. Entretanto o *Lean Testing* possui uma interface focada para as necessidades dos testadores, como a disponibilidade de histórico de execuções de testes e a possibilidade de fornecer um modelo de relatório no caso de erros. Além disso, o *Lean Testing* fornece diversos gráficos sobre os resultados, facilitando uma análise qualitativa e quantitativa. Na figura 4.30 há a ilustração das duas ferramentas contendo a listagem dos *test cases*.

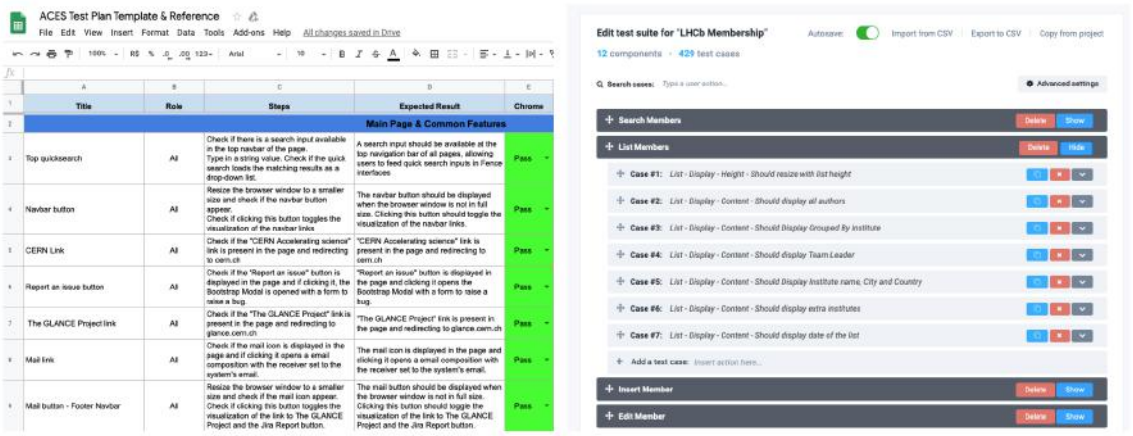

Figura 4.30: Exemplo de plano de testes feito através do *Google Sheets* à esquerda e *Lean Testing* à direita.

Este capítulo abordou as principais contribuições deste trabalho para o processo de desenvolvimento do grupo. Inicialmente foi falado sobre o desenvolvimento de duas ferramentas de testes orientadas a contribuir para a escrita de testes, com o *FUnit* direcionado para testes unitários e o *Fate* para testes *e2e*. Após isso foi abordada a estratégia adotada para administração automática destas ferramentas, aprimorando o processo de trabalho dos desenvolvedores. Por fim foi debatido o esforço na adoção de práticas ágeis dentro da cultura do grupo, com o objetivo de melhorar a qualidade das entregas. No próximo capítulo serão apresentados os resultados obtidos com o trabalho desenvolvido e uma análise será feita em relação a estes resultados.

Capítulo 5

Resultados e Avaliação

5.1 Verificação de código com testes unitários

Até o momento atual, 291 testes unitários foram desenvolvidos em 3 repositórios do grupo de programadores, sendo estes *Fence*, *Atlas* e *Alice*. Estes *test cases* contém 310 asserções que cobrem 64 métodos de 29 classes. O *FUnit* foi desenvolvido há 6 meses atrás, e desde então mais de 100 testes foram desenvolvidos utilizando o *framework*, em uma média de quase 20 testes por mês. A expectativa é que taxa de desenvolvimento cresça no futuro. Isso se deve principalmente ao tempo necessário para ambientação e aprendizado dos desenvolvedores com a nova proposta de testes.

Por causa da ausência da prática de testes unitários no passado, a cobertura atual de testes em toda a base de código é considerada baixa. Por exemplo no repositório *Fence*, há uma cobertura de apenas 9.2%, com apenas 2 mil linhas cobertas das 22 mil existentes. Entretanto este número vem crescendo, e os desenvolvimentos recentes no repositório costumam ter uma média de 60% de cobertura de testes. Fica clara a necessidade do grupo consolidar o hábito de desenvolver testes unitários em paralelo com o desenvolvimento em código. Isso poderia ser inclusive mais aperfeiçoado no futuro com a prática de *TDD*, escrevendo antecipadamente o teste antes do próprio código.

Em relação ao desempenho de execução destes testes, ao longo dos últimos 4 meses a integração contínua executou os testes unitários entre os diferentes repositórios em um total de mais de 900 vezes, com 87% de taxa de sucesso. Em 8,5% das vezes foram detectados erros, evitando em mais de 70 vezes a propagação de inconsistências para o ambiente de produção. Esse número pode ser ainda maior se for considerada as ocasiões em que os desenvolvedores executaram os testes unitários em suas próprias máquinas durante o desenvolvimento.

Em média cada execução de todos os testes unitários de um repositório leva 2,8 segundos para ser executada em apenas uma máquina, tendo sido no total exe-

cutados mais de 140 mil *test cases* ao longo de aproximadamente 45 minutos nas *pipelines*. Estes dados mostram o custo extremamente baixo exigido pelos testes unitários, de forma que não prejudicam o processo de entregas rápidas.

5.2 Validação de usuário com testes ponta a ponta

Dada a complexidade em realizar testes *e2e*, suas estatísticas possuem uma ordem de grandeza inferior a dos testes unitários. Por meio do *Fate*, nos últimos 10 meses foram desenvolvidos 38 *test cases* contidos em 5 *test suites* para os sistemas *Alice Membership* e *Atlas Central Equipment System*. Estes testes avaliam 5 interfaces entre os dois sistemas, levando atualmente em média 3 minutos para execução de todos os testes. Essa duração era consideravelmente maior no passado, podendo a chegar a mais de 15 minutos em uma execução, mas dado os esforços em paralelizar e otimizar as máquinas hospedeiras, este tempo foi reduzido. Atualmente 6 instâncias de máquinas são requisitadas, com uma instância *hub* administrando os testes para as 5 instâncias *node* os executarem no navegador *Chrome*. Este número de instâncias foi escolhido pois é proporcional ao número de testes atualmente existentes.

Como exemplos de testes *e2e* desenvolvidos pelo grupo, é válido mencionar verificações de interações como acoplamento e desacoplamento de equipamentos, submissão de novos contratos, busca de equipamentos, e movimentações de *racks*. Diferentemente da natureza atômica dos testes unitários, os testes *e2e* são capazes de avaliar o comportamento completo de uma funcionalidade requisitada.

Nos últimos 4 meses, com o início das *pipelines*, foram feitas mais de 500 execuções de testes *e2e*, acumulando um total de quase 15 mil *test cases* executados em uma duração de mais de 37 horas. Destas execuções, 79% obtiveram sucesso, e 13% indicaram pelo menos uma inconsistência. O motivo do valor mais elevado na taxa de erros em relação aos testes unitários é o fato da maior sensibilidade presente nos testes *e2e*. Como dito anteriormente, testes avaliam o software como um todo, tanto em quesitos funcionais como não funcionais, e, portanto, instabilidades na infraestrutura ou falta de robustez no próprio desenvolvimento do teste condicionam à falha. Entretanto, da mesma forma que a chance de falsos positivos de erro é maior do que a dos testes unitários, os testes *e2e* conseguem ter um alcance maior no encontro de inconsistências para uma mesma equivalência a ser verificada por testes unitários.

Como aspiração para os próximos passos, se mostra imprescindível o aumento do desenvolvimento de testes *e2e* dentro do grupo, assim como no caso dos testes unitários. As ferramentas disponíveis dentro do *Fate* facilitam o desenvolvimento dos testes, entretanto é necessário a familiarização com o *framework*. É de interesse também implementar abstrações no *framework* para aumentar a estabilidade dos

testes desenvolvidos pelo grupo. Por exemplo, se é de conhecimento que para uma mesma interação ocorrem comportamentos distintos para *Chrome* e *Firefox*, é interessante encapsular e tratar estes comportamentos em um módulo para ser usado pelo desenvolvedor do teste, evitando que o próprio tenha esta tarefa adicional.

5.3 Fluidez com integração contínua

Desde a concepção da integração contínua para os repositórios *Fate*, *Fence* e *Atlas* os *pipelines* foram executados respectivamente e aproximadamente 350, 1000 e 2400 vezes, totalizando mais de 3750 execuções. Destas, 84% obtiveram sucesso, 14% falharam e o restante foi manualmente cancelada. Essas *pipelines* foram responsáveis pela execução de mais 8 mil *jobs* em uma duração média de 46 segundos para cada *job* e mais de 100 horas totais. Isso tudo ocorreu nos últimos 4 meses, o que demonstra o quanto a integração contínua vem sendo utilizada pelos desenvolvedores do grupo.

O primeiro *job* realizado pela maior parte das *pipelines* do *Fence CID* é o de *build*. Este *job* se certifica que o processo de compilação foi feito com sucesso e que a aplicação possui todos os recursos necessários. Em 6,4% dos casos em que foi executado, foi identificado pelo menos um problema durante a compilação. Por conta dessas verificações, foi identificada a falta de padronização dentro do grupo em relação a preparação das dependências do software, assim como a não execução de alguns passos necessários para a compilação.

Geralmente presente seguidamente ao *job build*, existem os *jobs* de testes unitários e de *e2e*. Estes *jobs* são os que mais encontram problemas quando executados, apresentando uma taxa de 14,6% de identificação de falhas. Esta taxa corresponde a 70,4% da acusação total de erros por *jobs*, sendo o principal agente em alertar problemas com a nova versão da aplicação. Dada a eficiência destes testes, além do incentivo em aumentar a sua quantidade, há o interesse em investir também em outros tipos de testes e acrescentá-los à integração contínua, como os testes de integração.

É encontrado também em certos casos o *job* de *deploy*, que após a execução dos *jobs* de *build* e testes realiza a entrega da nova versão em uma máquina hospedeira. Fazendo parte da entrega contínua do *Fence CID*, poucas falhas são identificadas por esse *job*, sendo geralmente problemas existentes na máquina hospedeira. O maior benefício notado por esta implementação foi a automação e rapidez em disponibilizar uma nova versão em ambiente *stage*. Para trabalho futuro seria interessante implementar também um *deploy* automático para a máquina de produção, reduzindo as chances de falha humana.

A figura 5.1 possui as comparações mais relevantes em relação aos resultados dos *jobs*. A maior parte das tarefas são executadas para sistemas *Atlas*, pois atualmente

é o repositório com maior atividade dado ao maior número de desenvolvedores. Como existem tarefas para testes unitários e testes *e2e*, os *jobs* de testes são os mais executados pelo *Fence CID*, e a detecção de falhas pelas tarefas de testes é consideravelmente superior à de outras tarefas. É possível observar também na figura o total de resultados obtidos com a execução dos *jobs*.

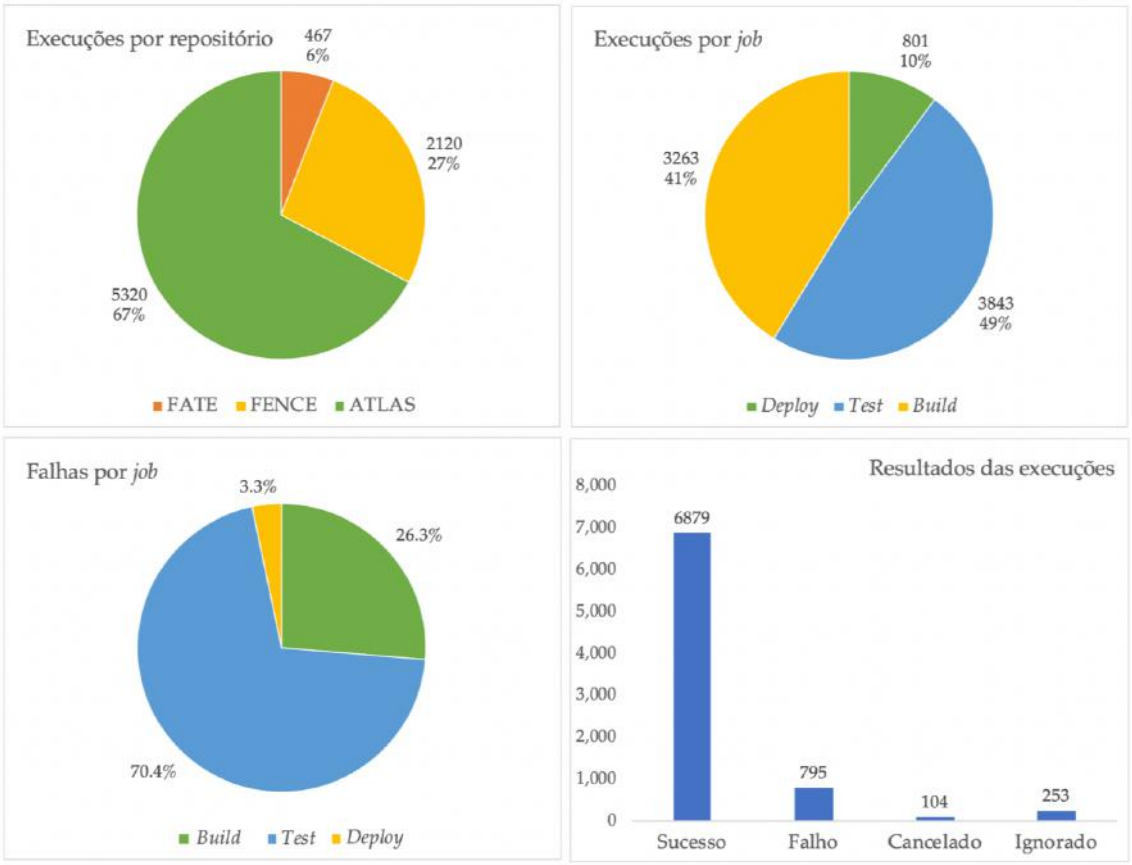

Figura 5.1: Diversas comparações entre as execuções de *job*.

As vantagens trazidas pelo *Fence CID* com a integração e entrega contínua justificam o esforço realizado em implementá-lo. O custo de manutenção é minimizado, sendo apenas necessário realizar alterações em arquivos de configuração. A ferramenta também provou estabilidade, executando a *pipeline* em todos os momentos esperados, e eficiência, mantendo um baixo tempo de execução das tarefas.

5.4 Comunicação com práticas ágeis

Comunicação com *Slack*

Desde o início da utilização da ferramenta *Slack* a cerca de 20 meses, foram enviadas mais de 26 mil mensagens entre 28 membros. 62% destas mensagens foram individuais, 32% foram nos 14 canais públicos disponíveis, e 6% em canais priva-

dos. É válido mencionar também que deste total, aproximadamente um terço foram mensagens de notificação emitidas por *bots*.

Fazendo uma avaliação destes números, é possível observar o uso diário da ferramenta, o que proporcionou uma melhoria significativa na comunicação interna do grupo, principalmente para a colaboração remota dos alunos que estão localizados fora do *CERN*. Algo interessante a melhorar seria a implementação de filtros mais eficientes a fim de reduzir o número de mensagens enviadas por *bots*. Atualmente o número de mensagens deste tipo é considerável e pode ser parcialmente irrelevante para a maior parte dos desenvolvedores.

Revisão com *Merge Requests*

A adoção da prática de *merge requests* foi uma das contribuições mais notórias em relação a melhorias ágeis. Tendo sido iniciada há aproximadamente dois anos atrás, atualmente mais de 1800 *merge requests* foram realizados entre os 23 repositórios do grupo. Deste número, 95% dos pedidos passaram por aprovações e foram acrescentados à base de código. No total mais de 2000 discussões ocorreram nesses pedidos, contribuindo na aproximação entre os desenvolvedores na forma de se escrever código. As discussões sobre boas práticas cresceram substancialmente no grupo com o início dos *merge requests*. Ao mesmo tempo, facilitou o compartilhamento de soluções já desenvolvidas para um problema identificado durante a revisão. Como uma possível melhoria, seria interessante o autor do código proporcionar mais informações ao revisor sobre a motivação e comportamento esperado do código a ser revisado, para facilitar sua análise integral pelos revisores.

Documentação com *Confluence*

Começando a ser utilizado há 10 meses, a documentação contida no software *Confluence* contém atualmente mais de 250 documentos em uma colaboração de mais de 30 membros e com um total de mais de 4200 acessos. Nos últimos 3 meses a plataforma chegou a possuir quase 500 acessos em uma semana, e documentos contendo apresentações e tutoriais foram os mais visualizados. Na figura 5.2 é possível analisar alguns números. Durante o mês de agosto o *Confluence* registrou quase 1,5 mil acessos, com a página principal possuindo 475 visualizações.

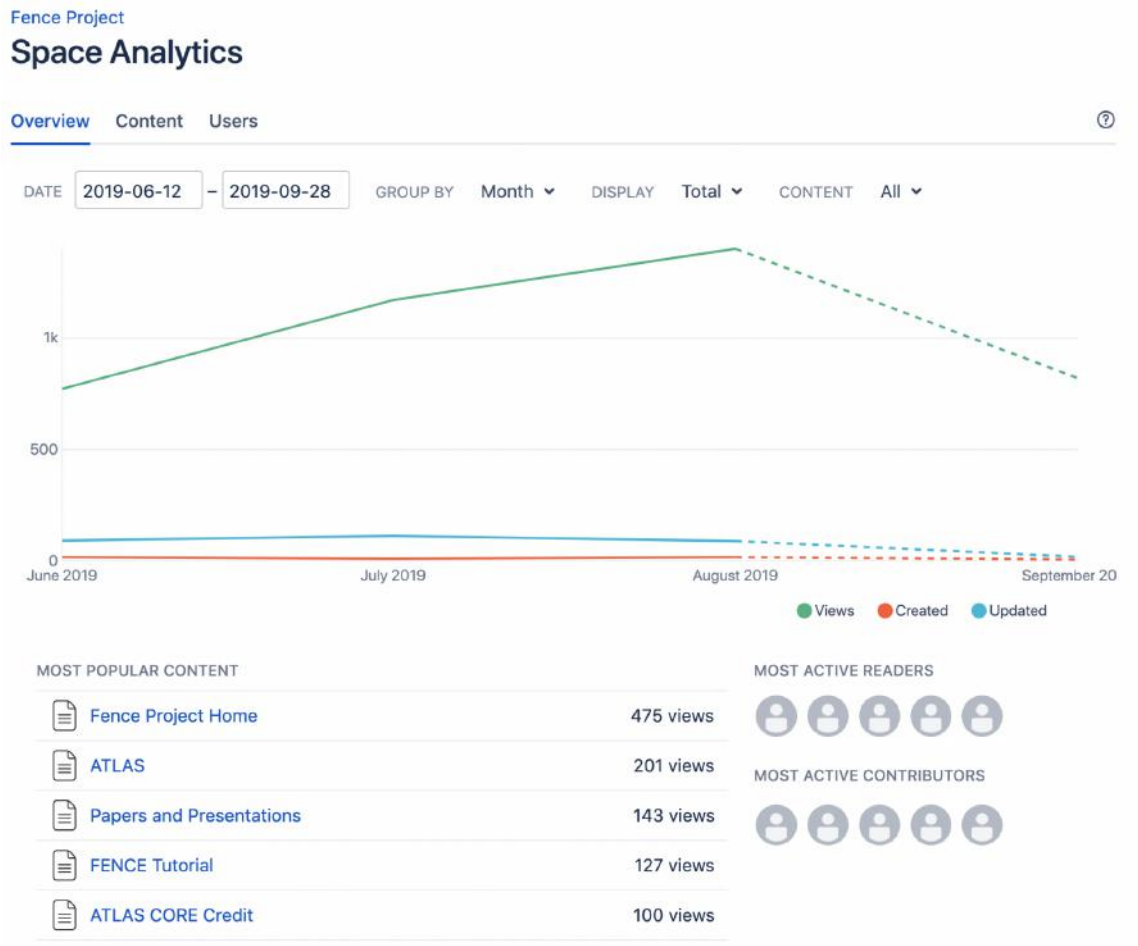

Figura 5.2: Análise de acessos à plataforma *Confluence* do grupo.

O principal benefício trazido pela plataforma foi a centralização da documentação que anteriormente se encontrava espalhada entre outras ferramentas. Apesar de possuir grande integração nativa com a plataforma *Jira*, há o desejo futuro em ter mais controle com esta integração, como a adoção do *plugin ScriptRunner* que permite uma flexibilização maior, a nível de código, na manipulação de dados entre as duas plataformas.

Validações com *Linters*

Começando a ser usado há cerca de um ano, a análise do uso de *linters* é mais subjetiva em relação a resultados. Conversando com os desenvolvedores do grupo, para um programador iniciante os avisos de *linters* podem ocorrer em até 50% de suas linhas escritas, enquanto para os mais experientes esta ocorrência diminui para uma taxa média de 10%. De toda forma, seu uso se mostra fundamental por todos, apesar de haver constantes dificuldades em manter todos os *linters* funcionais. Estas dificuldades ocorrem principalmente por conta dos diferentes editores de texto usado pelos desenvolvedores, assim como suas diferentes versões, configurações e condições

operacionais.

Plano de testes com *Lean Testing*

O uso da ferramenta *Lean Testing* se faz presente há pelo menos dois anos. Planos de testes para 6 sistemas foram desenvolvidos desde então, totalizando mais de 1500 *test cases*. Diversas validações foram realizadas a partir destes planos, feitas principalmente por alunos ingressantes.

A utilização desta ferramenta ainda é parcial pelo grupo. Apesar de seus benefícios, para alguns membros há ainda a preferência pelo software *Google Sheets*. O *Lean Testing* estabelece critérios bem definidos sobre como realizar o plano de testes, na tentativa de orientar o desenvolvedor de testes e o testador. Já o *Google Sheets*, como se trata de uma ferramenta genérica para edição de planilhas, possui suporte a mais customizações, como *macros* e *scripts*, mas ao mesmo tempo possui menos funcionalidades específicas de testes, como histórico de execuções.

É notável também a redução recente da atualização dos planos de testes, motivado principalmente por dificuldade de alocação de tempo nesta tarefa durante o processo de trabalho. Para um trabalho futuro há o interesse em revisitar esta questão, na tentativa de compreender como esta atividade pode ser otimizada.

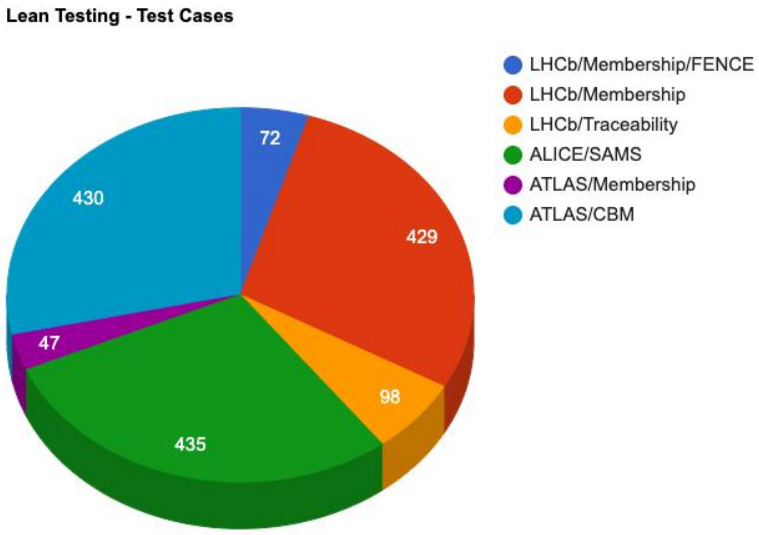

Figura 5.3: Divisão de *test cases* entre os sistemas utilizando o *Lean Testing*.

Capítulo 6

Conclusão

Presente no maior experimento científico já construído, este trabalho faz parte da colaboração entre UFRJ e *CERN*, com o desenvolvimento de aplicações web que vão desde o monitoramento de equipamentos até a gerência de membros, atendendo as necessidades dos experimentos *ATLAS*, *ALICE* e *LHCb*. Atualmente 20 sistemas com mais de 3 mil usuários foram implementados pelo grupo, fazendo uso do *framework Fence* para reutilização de código e facilidades em desenvolvimento de interfaces.

Com o aumento de sistemas, foi levada em consideração a possibilidade de tentar melhorar a qualidade do produto sem prejudicar o tempo de entrega. Utilizando a filosofia de metodologias ágeis como *Extreme Programming*, diversos processos e métodos foram implementados no grupo de desenvolvedores com os objetivos de melhorar a comunicação, aumentar a transparência e principalmente prover um software confiável. Nesse último ponto, que é o tema principal deste trabalho, o maior esforço foi realizado na área de testes automáticos e em *devops*, com o desenvolvimento de dois *frameworks*, *Fate* e *FUnit*, e de um ambiente de integração e entrega contínua, o *Fence CID*.

Para as metas de comunicação e transparência, diversas ferramentas e rotinas foram apresentadas aos desenvolvedores do grupo. *Merge requests*, *Slack*, *Confluence*, *Linters*, *Lean Testing* e *stand-up meetings* estão entre elas, destacando-se as discussões produtivas entre desenvolvedores nas revisões de código do *merge request* e a padronização e aprendizado de código proporcionado pelos *linters*. Todas essas ideias possuíram bons resultados, excetuando-se possivelmente o *Lean Testing*, de modo que uma nova estratégia para o desenvolvimento de plano de testes possa ser necessária.

Em relação à confiabilidade de software, os testes automáticos possuíram resultados significativos. Testes unitários são rápidos, modulares e fáceis de escrever, considerando-se também os ganhos com documentação e orientação a boas práticas de código. Já os testes *e2e* conseguem avaliar uma funcionalidade por completo, na

mesma perspectiva que o usuário final, e fazer um verificação complementar a dos testes unitários. Apesar de serem mais custosos e demorados, estes tipos de teste compensam para projetos de médio e longo prazo, com a capacidade de avaliar o comportamento do software em diferentes ambientes operacionais. Apesar das facilidades de ambos *frameworks Fate* e *FUnit* para o desenvolvimento destes testes, o número ainda é baixo em relação a base de código existente, e portanto há a urgência na adoção da prática de implantação de testes durante o desenvolvimento, com o uso do *FUnit* para testes unitários no decorrer da escrita do código, e o uso do *Fate* quando a funcionalidade estiver finalizada e for apresentada para o usuário.

O *Fence CID* tornou possível a orquestração da verificação e entrega de software pelo grupo. Compilando, realizando testes e executando o *deploy*, as *pipelines* concederam tempo e garantia para os desenvolvedores com a execução automática destas tarefas. A possibilidade de paralelizar tarefas com um ambiente de sistemas distribuídos em *Docker* também trouxe benefícios em relação a uma possível escalabilidade de sistemas no futuro.

Testes automáticos com apoio de metodologias ágeis vêm ganhando cada vez mais adesão pela comunidade de programadores. A proposta inicial deste trabalho era a implementação de uma plataforma de testes automáticos que trouxesse melhorias para a confiabilidade do software. Apesar de ser recente, a plataforma de testes implementada conseguiu atingir o objetivo, possuindo resultados significativos na localização antecipada de inconsistências. Adicionalmente, a plataforma possibilitou a redução de tempo gasto pelos desenvolvedores com a entrega de software, e as práticas ágeis adotadas pelo grupo contribuíram para a comunicação e organização dos desenvolvedores.

6.1 Trabalhos Futuros

Em relação a trabalhos futuros para o *FUnit*, há principalmente o interesse em aperfeiçoá-lo com um melhor suporte a testes de integração. Atualmente é possível a realização destes testes, mas há oportunidades para otimização, como a criação de abstrações para facilitar testes em *REST API*, que se trata de um padrão de arquitetura utilizado para implementação de serviços web. Em relação a testes unitários, algumas funcionalidades extras poderiam ser desenvolvidas no futuro. Por exemplo, é desejável ter a possibilidade de múltiplas condições de entrada para um único *test case*, de forma que possa ser executado repetidamente para cada condição. Seria interessante também o *framework* ter capacidade de desenvolver *mocks* de outras classes além do tradicional *mock* que representa a própria classe a ser testada, algo que é especialmente útil para métodos que fazem chamadas a outras instâncias de classe. Outra ideia é a extensão de testes unitários para além da linguagem *php*,

estendendo a implementação dos testes para a linguagem *javascript*.

Como aspiração para os próximos passos para o *Fate*, se mostra imprescindível o aumento do desenvolvimento de testes *end-to-end* dentro do grupo, assim como no caso dos testes unitários. As ferramentas disponíveis dentro do *Fate* facilitam o desenvolvimento dos testes, entretanto é necessária a familiarização com o *framework*. É de interesse também implementar abstrações no *framework* para aumentar a estabilidade dos testes desenvolvidos pelo grupo. Por exemplo, se é de conhecimento que para uma mesma interação ocorrem comportamentos distintos para *Chrome* e *Firefox*, é interessante encapsular e tratar estes comportamentos em um módulo para ser usado pelo desenvolvedor do teste, evitando que o próprio tenha esta tarefa adicional.

Finalmente para o *Fence CID* seria interessante incrementar a integração contínua com mais tarefas a serem realizadas, como testes de validação de sintaxe e verificação de vulnerabilidade em dependências. É almejada também a possibilidade de estender a entrega contínua para o ambiente de produção, tarefa que ainda é feita manualmente dada sua sensibilidade.

Referências Bibliográficas

[1] COLLABORATION, T. L. "The LHCb Detector at the LHC", *Journal of Instrumentation*, v. 3, n. 08, pp. S08005–S08005, 2008.

[2] COLLABORATION, T. A. "The ATLAS Experiment at the CERN Large Hadron Collider", *Journal of Instrumentation*, v. 3, n. 08, pp. S08003–S08003, 2008.

[3] PINHÃO, G. L. L. "Automação do processo de produção de publicações em colaborações científicas geograficamente distribuídas", *Poli Monografias UFRJ*, 2019.

[4] GILLIES, J. *CERN and the Higgs Boson*. Icon Books, 2018.

[5] *CERN Annual report 2018*. Relatório técnico, CERN, Geneva, 2019. Disponível em: <http://cds.cern.ch/record/2671714>. Acessado em 2019-09-12.

[6] CERN. "CERN Official Website - About section", . Disponível em: <https://home.cern/about>. Acessado em: 2019-08-23.

[7] EVANS, L., BRYANT, P. "LHC Machine", *Journal of Instrumentation*, v. 3, n. 08, pp. S08001–S08001, 2008.

[8] CERN. "CERN Official Website - LHCb experiment section", . Disponível em: <https://home.cern/science/experiments/lhcb>. Acessado em: 2019-08-23.

[9] CERN. "CERN Official Website - ATLAS experiment section", . Disponível em: <https://home.cern/science/experiments/atlas>. Acessado em: 2019-08-24.

[10] GRAEL, F. F. "Sistema Glance: recuperação e processamento de grandes volumes de dados", *Poli Monografias UFRJ*, 2009. Disponível em: <http://monografias.poli.ufrj.br/monografias/monopoli10001978.pdf>.

[11] MAIDANTCHIK, C., GRAEL, F., GALVAO, K., et al. "Glance project: a database retrieval mechanism for the ATLAS detector", v. Journal of Physics: Conference Series 119(4), pp. 042020, 2008.

[12] RAMOS, B. L. "Fence: uma abordagem orientada a objetos na concepção de sistemas web altamente configuráveis para os experimentos do CERN", *Poli Monografias UFRJ*, 2015. Disponível em: <http://monografias.poli.ufrj.br/monografias/monopoli10014830.pdf>.

[13] LANGE, B., MAIDANTCHIK, C., POMMES, K., et al. "An object-oriented approach to deploying highly configurable Web interfaces for the ATLAS experiment", *Journal of Physics: Conference Series*, v. 664, n. 6, pp. 062026, 2015.

[14] FENCE. "Fence Official Website - Main Page", Disponível em: <https://glance.cern.ch/fence/>. Acessado em: 2019-08-23.

[15] BERNERS-LEE, T. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. Harper San Francisco, 1999.

[16] VORA, P. *Web application design patterns*. Morgan Kaufmann, 2009.

[17] Myers, G. J., Badgett, T., Sandler, C. (Eds.). *The Art of Software Testing*. Hoboken, NJ, USA, John Wiley & Sons, Inc., 2012.

[18] BECK, K. *Kent Beck's Guide to Better Smalltalk*. Cambridge University Press, 1999.

[19] MARTIN, R. C. "Design principles and design patterns", *Object Mentor*, v. 1, n. 34, pp. 597, 2000.

[20] LEVENTHAL, L. M., TEASLEY, B. E., ROHLMAN, D. S. "Analyses of factors related to positive test bias in software testing", *International Journal of Human-Computer Studies*, v. 41, n. 5, pp. 717–749, 1994.

[21] ABDFEL-HAMID, T. K. "The Economics of Software Quality Assurance: A Simulation-Based Case Study", *MIS Quarterly*, v. 12, n. 3, pp. 395, 1988.

[22] PHAM, H. "Software reliability and cost models: Perspectives, comparison, and practice", *European Journal of Operational Research*, v. 149, n. 3, pp. 475–489, 2003.

[23] LEUNG, K. R. P. H., YEUNG, W. L. "Generating User Acceptance Test Plans from Test Cases", *IEEE Computer Society*, 2007.

[24] GUPTA, A., SHARMA, S. "Software Maintenance: Challenges and Issues", *Issues*, v. 1, pp. 23–25, 2015.

[25] MIKKONEN, T., TAIVALSAARI, A. "Web Applications: Spaghetti code for the 21st century", *Sun Microsystems Laboratories*, 2007.

[26] DESIKAN, S. *Software Testing*. Pearson Education India, 2006.

[27] TAIPALE, O., KASURINEN, J., KARHU, K., et al. "Trade-off between automated and manual software testing", *International Journal of System Assurance Engineering and Management*, v. 2, pp. 114–125, 2011.

[28] NGUYEN, H. Q. *Testing applications on the Web: Test planning for Internet-based systems*. John Wiley & Sons, 2001.

[29] OF THE IEEE COMPUTER SOCIETY, S. . S. E. S. C. *IEEE standard for software and system test documentation*. IEEE, 2008.

[30] RUNGTA, K. *Types of Software Testing*. Krishna, 2017.

[31] GRAHAM, D., VEENENDAAL, E. V., EVANS, I. *Foundations of Software Testing*. Cengage Learning EMEA, 2008.

[32] BOURQUE, P., FAIRLEY, R. E., OTHERS. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[33] BACCHELLI, A., BIRD, C. "Expectations, outcomes, and challenges of modern code review", *IEEE Press*, pp. 712–721, 2013.

[34] KANER, C. "A tutorial in exploratory testing", *Tutorial presented at QUEST*, 2008. Disponível em: <http://www.kaner.com/pdfs/QAIExploring.pdf>. Acessado em: 2019-09-12.

[35] NIELSEN, J., LANDAUER, T. K. "A mathematical model of the finding of usability problems", *Interchi*, 1993.

[36] HAMBLING, B., GOETHEM, P. V. *User Acceptance Testing*. BCS, The Chartered Institute for IT, 2013.

[37] YUAN, D., LUO, Y., ZHUANG, X., et al. "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems", v. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pp. 249–265, 2014.

[38] PANZL, D. J. "Test procedures: A new approach to software verification", v. Proceedings of the 2nd international conference on Software engineering, pp. 477–485, 1976.

[39] BECK, K., GAMMA, E. *Extreme programming explained: embrace change*. Addison-Wesley, 2000.

[40] HAMILL, P. *Unit Test Frameworks*. O'Reilly Media, Inc., 2004.

[41] YANG, Q., LI, J. J., WEISS, D. M. "A survey of coverage-based testing tools", *The Computer Journal*, v. 52, n. 5, pp. 589–597, 2009.

[42] BROWN, C. T., GHEORGHIU, G., HUGGINS, J. *An introduction to testing web applications with twill and selenium*. O'Reilly Media, Inc., 2007.

[43] CHOWDHURY, A. R. "All You Need to Know About End to End Testing", *LambdaTest*, 2018. Disponível em: <https://www.lambdatest.com/blog/all-you-need-to-know-about-end-to-end-testing/>. Acessado em: 2019-08-24.

[44] DEINER, H. "The motivation behind continuous integration in embedded software development", *TechTarget*, 2011. Disponível em: <https://searchsoftwarequality.techtarget.com>. Acessado em: 2019-09-12.

[45] FOWLER, M., FOEMMEL, M. "Continuous integration", *Martin Fowler*, 2006. Disponível em: <http://martinfowler.com/articles/continuousIntegration.html>. Acessado em: 2019-09-14.

[46] SHAHIN, M., BABAR, M. A., ZHU, L. "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices", *IEEE Access*, v. 5, pp. 3909–3943, 2017.

[47] TANENBAUM, A. S., STEEN, M. V. *Distributed Systems*. Createspace Independent Publishing Platform, 2016.

[48] NWANA, H. S., LEE, L. C., JENNINGS, N. R. "Coordination in software agent systems", *British Telecom Technical Journal*, v. 14, n. 4, pp. 79–88, 1996.

[49] CERN. "LHC section", *CERN Official Website*, . Disponível em: <https://home.cern/science/accelerators/large-hadron-collider>. Acessado em: 2019-08-23.

[50] ATLAS. “ATLAS Official Website - Collaboration section”, Disponível em: <https://atlas-collaboration.web.cern.ch/>. Acessado em: 2019-08-23.

[51] LHCB. “LHCb Official Website - Collaboration section”, Disponível em: <http://lhcb.web.cern.ch/lhcb/>. Acessado em: 2019-08-23.