UNIVERSIDADE DO EXTREMO SUL CATARINENSE - UNESC
CURSO DE CIÊNCIA DA COMPUTAÇÃO

CÁSSIO BENINCÁ DE JESUS

TESTE AUTOMATIZADO PARA VALIDAÇÃO DE DADOS DE RELATÓRIOS APLICANDO CONCEITOS DE COMPARAÇÃO BYTE A BYTE EM ARQUIVOS PDF

CRICIÚMA
2019

CÁSSIO BENINCÁ DE JESUS

TESTE AUTOMATIZADO PARA VALIDAÇÃO DE DADOS DE RELATÓRIOS APLICANDO CONCEITOS DE COMPARAÇÃO BYTE A BYTE EM ARQUIVOS PDF

Projeto de Pesquisa do Trabalho de Conclusão de Curso em Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC.

Orientador: Prof. Esp. Gilberto Vieira da Silva

CRICIÚMA

2019

CÁSSIO BENINCÁ DE JESUS

TESTE AUTOMATIZADO PARA VALIDAÇÃO DE DADOS DE RELATÓRIOS APLICANDO CONCEITOS DE COMPARAÇÃO BYTE A BYTE EM ARQUIVOS PDF

Trabalho de Conclusão de Curso aprovado pela Banca Examinadora para obtenção do Grau de Bacharel no Curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC, com Linha de Pesquisa em Testes de Software

Criciúma, 27 de junho de 2019.

BANCA EXAMINADORA

Prof. Esp. Gilberto Vieira da Silva - (UNESC) - Orientador

Prof. Me. Gustavo Bisognin - (UNESC)

Prof. Esp. Anderson Rodrigo Farias - (Faculdade SATC/UNIBAVE)

AGRADECIMENTOS

Aos familiares e amigos, que acompanharam e auxiliaram neste percurso, com incentivos para prosseguir e superar os desafios.

Ao Professor Especialista Gilberto Vieira da Silva, orientador deste trabalho, pelo apoio e auxilio no desenvolvimento do projeto.

Aos colegas de classe que sempre apoiaram nos momentos de dificuldade, contribuindo com ideias, questionamentos, sugestões e críticas que possibilitaram o amadurecimento e desenvolvimento do projeto.

RESUMO

Os sistemas modernos consomem e geram quantidades de informação cada vez maiores. As informações geradas por esses sistemas precisam estar corretas pois pessoas irão tomar decisões baseadas nos dados que estão recebendo. Existem ferramentas que fazem a validação dos dados no sistema, mas grande parte dos dados gerados pelo sistema na forma de documento digital, ainda precisa ser validada pelo profissional de testes manualmente, lendo páginas de documentos e fazendo cálculos para garantir que os valores ali presentes estejam corretos. Esse processo é cansativo e deixa grande margem para que dados incorretos passem despercebidos. Esse projeto visa solucionar esse problema com a automação de testes aplicada a documentação digital, criando um teste que possa ler as informações do documento e destacar graficamente possíveis diferenças encontradas.

Palavras-chave: Automação de teste, Documentos digitais, Comparação de imagens, Selenium

ABSTRACT

Modern systems consume and generate ever-increasing amounts of information. The information generated by these systems must be correct because people will make decisions based on the data they are receiving. There are tools that validate data in these systems, but much of the data generated by the system in the form of digital documents, still needs test professionals manually validating them, reading document pages and making calculations to ensure that the values present are correct. This process is tiring and leaves a lot of room for incorrect data to go undetected. This project aims to solve this problem with test automation applied to digital documentation, creating a test that can read the information in the document and graphically highlight possible differences found.

Key words: Test automation, Digital documents, Image comparison, Selenium

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ISO	International Organization for Standardization
JDK	Java Development Kit
OCR	Optical Character Recognition
PDF	Portable Document Format
POM	Page Object Model

SUMÁRIO

1 INTRODUÇÃO

No mundo atual as empresas estão lidando com quantidades cada vez maiores de dados, é muito difícil gerenciar tudo isso sem a ajuda de softwares. Eles estão presentes na maioria das empresas e precisam crescer junto com elas para acompanhar a demanda de funções e a capacidade de processamento de dados. Criar tais sistema exige bastante planejamento por parte das equipes de desenvolvimento.

Para Parreira (2010) engenharia de software é um conjunto integrado de métodos e ferramentas utilizadas para especificar, projetar, implementar e manter um sistema.

Um dos desafios da engenharia de software é o desafio do legado, que é fazer a manutenção e a atualização dos softwares atuais, sem apresentar grandes custos e ao mesmo tempo prosseguir com a prestação dos serviços corporativos essenciais.

Parte do trabalho necessário para manter um sistema são os testes de software. E testar softwares cada vez maiores exige um grande esforço das equipes de teste para o planejamento e execução dos casos de teste.

É comum que em testes mais repetitivos e dispendiosos os testadores deixem de verificar todos os casos de teste a cada alteração do sistema, isso pode acarretar problemas que geram atrasos na entrega ou que só são encontrados mais tarde já no cliente final. É aí que os testes automatizados se fazem necessários, pois eles podem verificar todos os casos de teste, quantas vezes forem necessárias e em um curto espaço de tempo (BERNARDO; KON, 2008).

Bernardo e Kon (2008) apontam que testes automatizados são programas ou scripts que exercitam funcionalidades do sistema sendo testado e fazem verificações automáticas nos efeitos colaterais obtidos.

Mas os sistemas modernos tem vários aspectos para serem analisados. Um deles é a emissão de relatórios. Vários sistemas tem funcionalidades que imprimem listas com uma grande quantidade de dados, essas listas frequentemente saem do escopo do sistema para exibir seus dados através de arquivos de texto externos, sendo bastante comum a apresentação desses dados em arquivos no formato PDF.

Um teste automatizado consegue simular as ações do usuário interagindo com o sistema (CARDEAL; PARREIRA, 2015), porém quando o sistema emite um relatório em um arquivo de texto externo, o teste automatizado não consegue simular o usuário lendo este arquivo em busca de falhas. Para que o teste consiga fazer tais validações é necessário que outras ferramentas ou técnicas sejam mescladas na automação.

Diante do exposto, esse projeto propõe a implementação de um teste automatizado capaz de validar dados de relatórios através de uma biblioteca que faça a comparação, utilizando dados obtidos através de ferramentas de manipulação de arquivos no formato PDF, apresentando o resultado do teste, e em caso de falha, destacando as diferenças encontradas entre o arquivo de controle e o arquivo de teste.

1.1 OBJETIVO GERAL

Desenvolver uma biblioteca para integração de um teste automatizado utilizando Selenium, capaz de validar dados de relatórios emitidos por um sistema, através de ferramentas que fazem manipulação de arquivos PDF, retornando erro no teste e apontando graficamente as diferenças encontradas.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos consistem em:

a) estudar os conceitos de teste de software e automação de testes incluindo suas ferramentas;
b) compreender as dificuldades envolvidas na automação de testes para os sistemas modernos;
c) estudar os conceitos de comparação de PDF e analisar as principais ferramentas de comparação de PDF;
d) desenvolver uma biblioteca que faça comparação byte a byte e permita a integração da ferramenta de teste automatizado com o manipulador de PDF;
e) aplicar os objetos de estudo em um teste automatizado capaz de validar relatórios.

1.3 JUSTIFICATIVA

O processo de teste de software pode se estender por várias etapas no desenvolvimento de um sistema. E são dois os objetivos principais do teste de software. O primeiro chama-se teste de validação, e baseia-se em validar os requisitos estabelecidos. Ou seja, certificar-se de que o software cumpre todas as suas funções adequadamente conforme o esperado pelo cliente e pelos analistas.

O segundo chama-se teste de defeitos, e sua principal função é encontrar pontos de falha no sistema, fazendo isso através da criação de cenários de utilização diferentes do esperado para forçar a aparição dos problemas (SOMMERVILLE, 2012).

O principal problema encontrado em equipes de teste é o cumprimento dos prazos. Os prazos são combinados com o cliente solicitante do sistema baseando-se em experiências prévias para fazer uma estimativa do tempo de desenvolvimento.

Frequentemente os tempos calculados para teste e correções de problemas são menosprezados, e como os problemas sempre ocorrem, os atrasos acabam sendo inevitáveis.

Nos dias de hoje não é possível falar sobre teste de software sem mencionar automação de testes. Um profissional que deseja ser produtivo na área de testes não é aquele que consegue executar o mesmo teste várias vezes em um curto espaço de tempo, e sim aquele que gasta o tempo apenas uma vez para desenvolver um teste automatizado capaz de rodar em segundos centenas de validações diferentes (ANICHE, 2015).

Automatizar um teste é mais custoso a curto prazo se comparado ao teste manual. Pois exige um grande esforço inicial para o desenvolvimento de todos os cenários, e exige também um conhecimento de programação por parte dos profissionais envolvidos no teste. Porém o ganho a longo prazo é notável (FEWSTER; GRAHAM, 1999, tradução nossa).

Com toda uma estrutura de testes automatizados agindo em conjunto com os testadores manuais, o sistema pode ser validado por vários lados ao mesmo tempo. Garantindo mais qualidade, menor quantidade de problemas, e uma maior chance de resolver os problemas de prazo mencionados anteriormente.

Para Ian, O objetivo final dos processos teste é estabelecer a confiança de que o software está pronto para o seu propósito. Isso significa que o sistema deve ser

bom o suficiente para seu intuito. E para ele, o mais importante quando se fala sobre software é que ele seja confiável.

Softwares que geram grandes quantidades de dados através de relatórios em arquivos de texto precisam de muita confiabilidade. Pois esses arquivos serão lidos por pessoas que tomarão decisões baseadas nos dados que ali estão presentes. Portanto não basta apenas a realização de testes para validar dados dentro do sistema. Os dados devem ser validados em tudo aquilo que o sistema gera. Especialmente quando nos atentamos para o fato de que em alguns casos, as próprias ferramentas para criação de relatórios assumem certas responsabilidades do sistema, como, fazer consultas no banco de dados, fazer comparações de dados, realizar operações aritméticas, entre outros, antes de emitir os dados.

Isso pode fazer com o que os testes aplicados no sistema demonstrem um falso positivo. Onde o sistema apresentou os dados da forma correta, porém os arquivos de texto emitidos apresentaram falhas que não foram interceptadas por não existir um teste validando essa ponta.

Além disso, os relatórios podem apresentar problemas não somente nos textos, mas também no layout, posicionamento das informações, linhas de fechamento de colunas, imagens e logotipos quem podem não estar sendo exibidos, e várias outras informações não textuais quem podem apresentar algum tipo de problema.

Por isso, um teste automatizado que seja capaz de validar não somente o sistema, como também os documentos gerados por ele, incluindo texto, layout e demais informações visuais, se torna o cenário ideal. Tal validação será possível mesclando ferramentas que aplicam os conceitos de comparação byte a byte de arquivos PDF com as ferramentas de automação como o Selenium. Para que o resultado do teste possa apresentar as possíveis falhas, demonstrando graficamente em quais pontos do arquivo as diferenças estão presentes.

2 TESTES DE SOFTWARE

É um fato bem estabelecido que em um projeto de desenvolvimento de software, cerca de cinquenta por cento do tempo investido e mais da metade de todo o custo do projeto são aplicados apenas nos testes e correções de software. Tem sido dessa forma já a muito tempo (MYERS; BADGETT; SANDLER, 2012, tradução nossa).

Uma das palavras mais conhecidas no meio do teste de software vem do inglês, “Bug”, que significa “Inseto”. A história mais difundida conta que essa expressão surgiu quando o computador Harvard Mark II de 1947 teve problemas por conta de um inseto preso entre alguns de seus relés, o que impedia o acionamento dos mesmos, e com isso gerava resultados inesperados ou mesmo não executava sua função. Com o tempo, a expressão ganhou força, e hoje é associada a problemas no código dos softwares (SHAPIRO, 1987, tradução nossa).

Mas não é só em encontrar “Bugs” ao acaso que o teste de software se baseia. O conceito de testes de software é, um processo, ou uma série de processos, planejados para garantir que o código faça aquilo que foi desenvolvido para fazer, e que além disso, não faça nada inesperado. O software deve ser previsível e consistente, sem apresentar surpresas ao usuário (MYERS; BADGETT; SANDLER, 2012, tradução nossa).

Myers, Badgett e Sandler (2012, tradução nossa) fazem um apontamento interessante sobre este conceito. Eles contam que algumas expressões bastante utilizadas por leigos e que são originárias deste conceito, não trazem benefícios para o profissional que busca trabalhar com testes. Algumas delas dizem que, testar é o processo de demonstrar que o sistema não tem erros, ou que o propósito do teste é mostrar que o sistema executa suas funções corretamente.

Numa primeira observação não há nada de errado com essas afirmações, porém existe todo um lado psicológico por trás disso tudo. São seres humanos que realizam os testes, e seres humanos são bastante orientados a metas. Se a meta é demonstrar que o sistema não tem erros, o subconsciente inevitavelmente trabalha para atingir esse objetivo. Com isso, existe um menor esforço para criar cenários de teste que possam apresentar falhas. Se a meta é demonstrar que o sistema executa suas funções corretamente, os cenários de teste serão voltados para validar apenas as

regras de negócio. Se estiver tudo funcionando, o testador pode ser induzido a achar que seu trabalho ali acabou.

Por outro lado, se a meta é mostrar que o sistema tem erros, ou que é possível seguir um caminho que não foi previsto pelos desenvolvedores e analistas, os cenários de teste serão voltados para encontrar qualquer possível falha, pois a meta agora é mostrar essas falhas.

Observa-se assim, que apenas mudando a semântica da frase, *"Garanta que esse sistema não apresenta erros"*, para *"Mostre os erros desse sistema para que possamos corrigir"*, a motivação dos profissionais de teste muda, e os resultados obtidos podem ser bem diferentes, agregando mais valor ao software.

Os testes de software também tem algumas palavras e conceitos que são utilizados diariamente por aqueles que trabalham na área, o quadro 1 vai listar alguns dos mais comuns.

Quadro 1 – Conceitos gerais de teste.

Plano de testes	É o planejamento dos testes, descreve de que forma e quais tipos de teste serão utilizados para a validação do sistema.
Requisitos de testes	São as metas a serem atingidas pelo teste. Podem ser requisitos do cliente, como também dos analistas.
Caso de teste	É a forma como devemos executar o teste. Um passo a passo das etapas que devem ser executadas para cumprir os requisitos de teste. Um caso de teste pode atender a mais de um requisito de testes.
Caso de uso	É um cenário de testes, um conjunto de casos de testes dispostos em uma ordem que representam alguma utilização real do sistema na perspectiva do cliente.
Ambiente de teste	É o conjunto de ferramentas, software e hardware, que compõem todo o ambiente onde os testes serão executados.

Fonte: MOLINARI (2014).

2.1 IMPORTÂNCIA DO TESTE DE SOFTWARE

Mas, porque é importante investir tanto tempo e dinheiro em testes?
A resposta para esta pergunta está justamente no custo. Corrigir falhas encontradas ainda durante o desenvolvimento é muito mais barato do que corrigi-las já em um ambiente de produção, ou mesmo no cliente final.

Mcpeak (2017, tradução nossa) afirma que a correção de um problema encontrado em fase de produção pode ser até cinco vezes maior do que se o mesmo problema tivesse sido encontrado durante a fase de design.

A falta ou a negligência nos testes pode trazer grandes prejuízos, especialmente em projetos de grande porte. A lista abaixo exemplifica isso através de três falhas em software que por não terem sido descobertas durante o desenvolvimento, resultaram em grandes prejuízos financeiros.

- Orbitador climático de Marte:

  Foi uma sonda espacial americana lançada em 1998 com o objetivo de estudar o clima marciano. O projeto de mais de U$: 125 milhões foi perdido por conta de um simples erro de conversão de unidades para o sistema métrico, que fez com que a sonda saísse de sua trajetória e se chocasse contra a atmosfera do planeta (HARLEY, 2018, tradução nossa).

- Terminal 5 do Aeroporto de Heathrow:

  Pouco antes de ser inaugurado no Reino Unido, o sistema de transporte de bagagens do aeroporto havia passado por extensos testes com mais de doze mil bagagens. Nenhum problema foi encontrado. Porém no dia da inauguração se descobriu que o sistema não conseguia lidar com situações do mundo real, como por exemplo, quando um passageiro remove manualmente sua bagagem da esteira para pegar algum item esquecido. Essa ação fazia todo o sistema ficar confuso e travar. Durante os 10 dias seguintes mais de 42mil bagagens deixaram de viajar com seus donos, e mais de 500 voos foram cancelados (HARLEY, 2018, tradução nossa).

- Nave espacial Mariner 1:

  Lançada em 1962 com o objetivo de levar uma sonda espacial ao planeta Vênus, a nave precisou ser autodestruída poucos segundos após o lançamento. Um desvio da rota a fez virar de volta para a terra com risco de colisão. Análises posteriores mostraram que a omissão de um hífen nas instruções do código causou o desvio da rota. Um prejuízo de mais de U$: 18 milhões na época (HARLEY, 2018, tradução nossa).

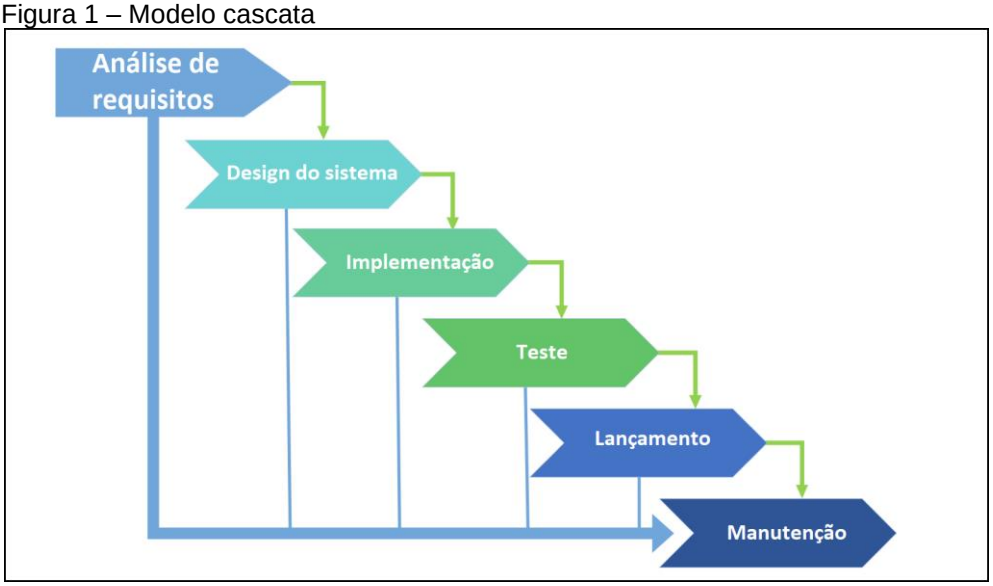
2.2 MODELOS DE DESENVOLVIMENTO

Antes de falarmos sobre os tipos de teste, é importante apontar em que momento do processo de desenvolvimento acontecem os testes. O processo de desenvolvimento de software é descrito por Sommerville (2012) como um conjunto de atividades relacionadas que levam à produção de um produto de software. Existem vários modelos para os processos de desenvolvimento, e todos eles acabam passando pelas mesmas etapas. Análise de planejamento e requerimentos, design da arquitetura do projeto, desenvolvimento e codificação, testes e lançamento (PRESSMAN, 2010). As mudanças entre os modelos ocorre na forma como cada modelo avança entre as etapas e o nível de interação que uma etapa tem com a outra.

2.2.1 Modelo cascata

É um modelo no qual o processo de desenvolvimento segue um fluxo, passando por todas as fases de análise, design, implementação, teste, lançamento e manutenção. Este modelo prevê a execução gradual de todos os estágios por completo antes de passar para a próxima etapa. Também é estritamente documentado e predefinido com os requisitos para cada fase do ciclo (PRESSMAN, 2010).

Figura 1 – Modelo cascata

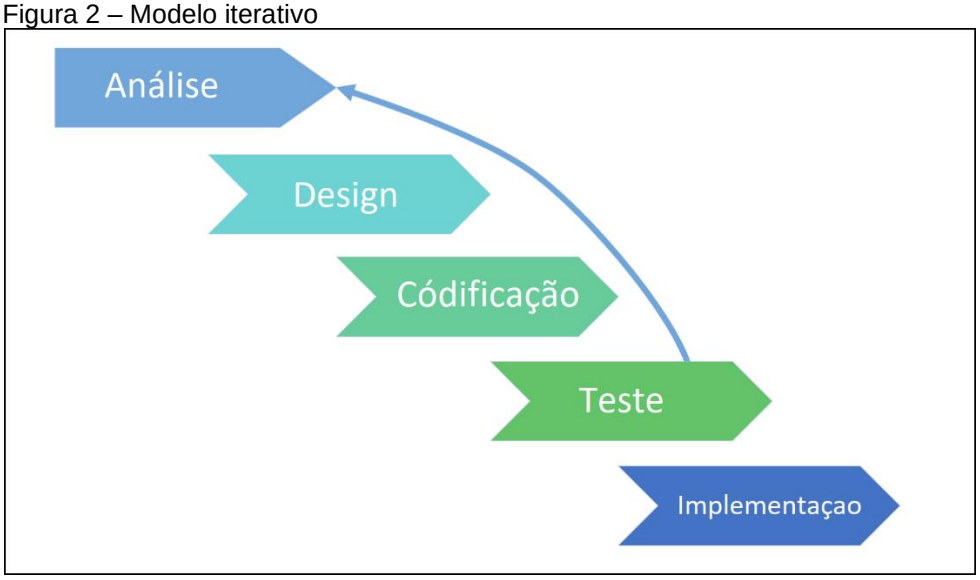


Fonte: Pressman (2010).

2.2.2 Modelo iterativo

O modelo Iterativo não precisa da lista de requisitos antes do início do projeto, com isso o desenvolvimento pode começar diretamente na parte funcional, podendo ser expandida posteriormente. O processo é repetitivo, permitindo fazer novas versões do produto a cada ciclo. Cada iteração (que dura de duas a seis semanas) inclui o desenvolvimento de um componente separado do sistema e, depois disso, esse componente é adicionado a versão desenvolvida anteriormente (PRESSMAN, 2010).

Figura 2 – Modelo iterativo

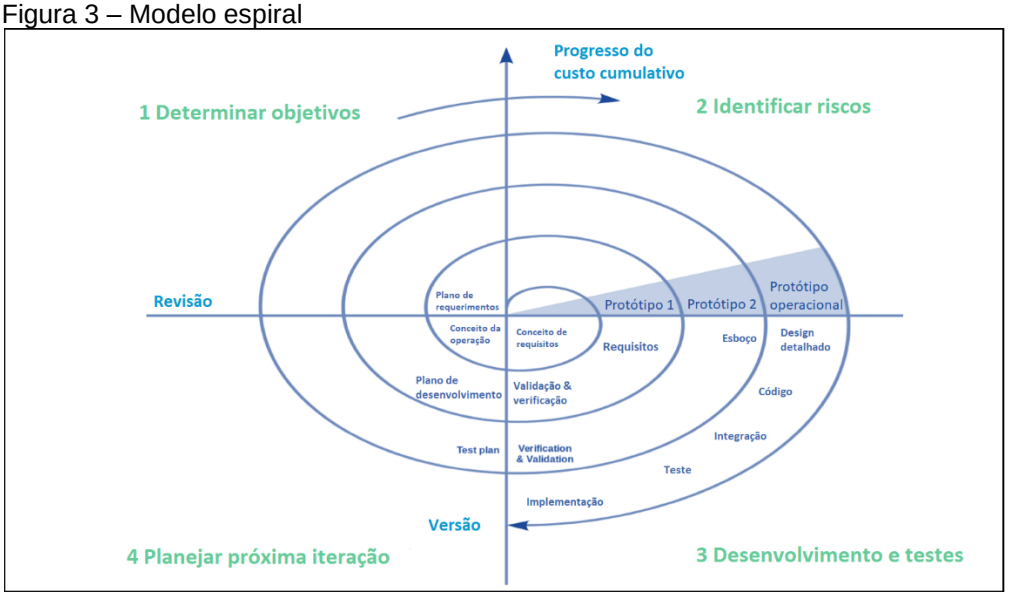


Fonte: Pressman (2010).

2.2.3 Modelo espiral

Combina arquitetura e prototipagem por etapas. É uma combinação dos modelos Iterativo e cascata com ênfase na análise de risco. O principal ponto do modelo espiral é definir o momento certo para avançar rumo ao próximo estágio. Os quadros de tempo preliminares são utilizados como solução para este problema. A mudança para o próximo estágio é feita de acordo com o plano, mesmo que o trabalho no estágio anterior ainda não tenha sido concluído. O plano é apresentado com base nos dados estatísticos recebidos durante os estágios anteriores (PRESSMAN, 2010).

Figura 3 – Modelo espiral



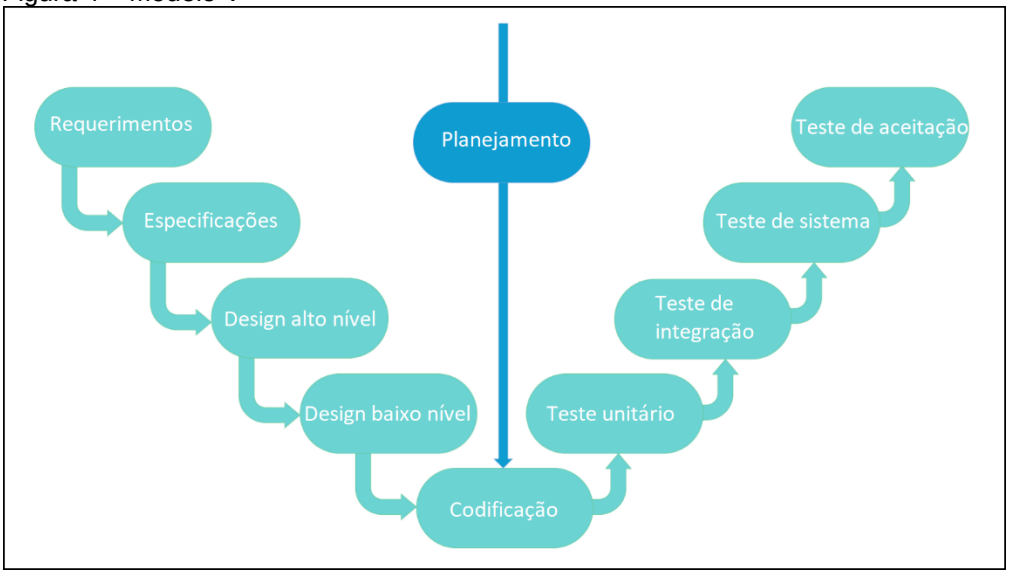
Fonte: Pressman (2010).

2.2.4 Modelo V

É a expansão do modelo cascata e é baseado no princípio de que para cada estágio de desenvolvimento, pode haver um estágio de teste. Este modelo é muito rigoroso e o próximo estágio é iniciado somente após a fase anterior ter passado pelos seus testes. Esse processo também é chamado de “validação e verificação”.

Cada estágio tem o controle do processo atual, para garantir que a conversão para a próxima etapa seja possível (PRESSMAN, 2010).

Figura 4 – Modelo V



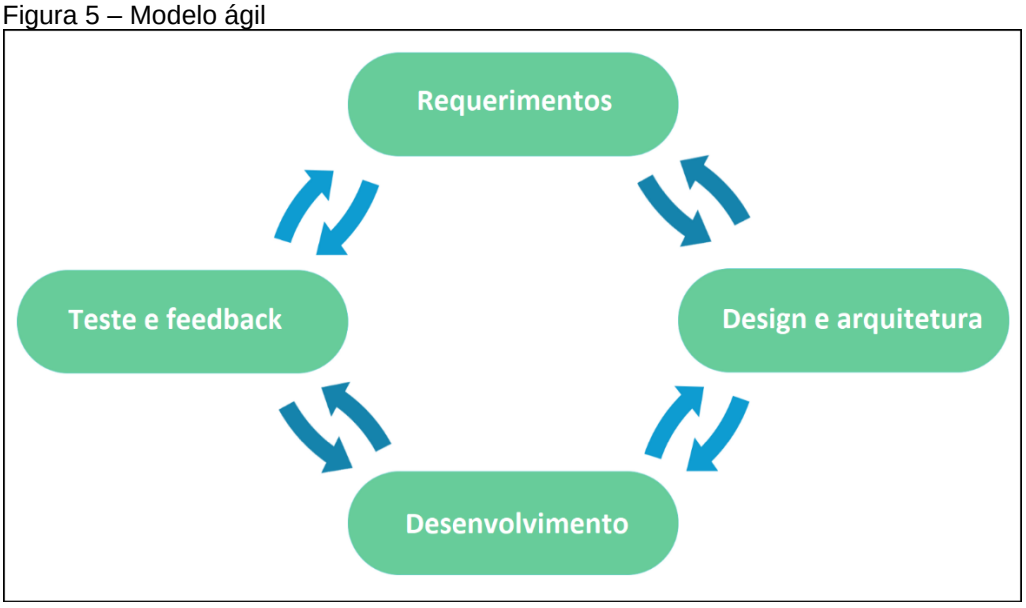
Fonte: Pressman (2010).

2.2.5 Modelo ágil

Na metodologia ágil, após cada iteração de desenvolvimento, o cliente é capaz de ver o resultado e avaliar se está de acordo com suas expectativas ou não. Essa é uma das principais deste modelo, pois é possível garantir que o sistema está atendendo as necessidades do cliente após cada etapa de desenvolvimento, e não apenas na entrega final do produto, onde já seria custoso de mais realizar mudanças.

Uma de suas desvantagens é que, com a ausência de requisitos definidos, é difícil estimar os recursos e o custo de desenvolvimento. A base desse modelo consiste em pequenas reuniões semanais chamadas de Sprints, para definir os passos seguintes baseados nos feedbacks do cliente (PRESSMAN, 2010).

Figura 5 – Modelo ágil



Fonte: Pressman (2010).

2.3 TIPOS DE TESTES

Os testes podem ser divididos em dois grandes grupos, testes funcionais, e testes estruturais, respectivamente conhecidos como testes de caixa preta e testes de caixa branca. Por terem princípios bastante diferentes, é comum que empresas adotem apenas um ou outro modelo de testes, porém, muitos especialistas afirmam que a maneira mais correta de testar é aplicando os dois conceitos, um software de

qualidade precisa ter passado pelos dois cenários de teste (NIDHRA; DONDETI, 2012, tradução nossa).

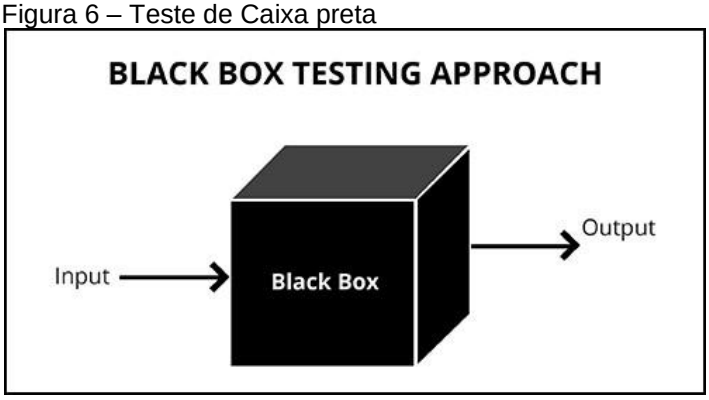
Dentro destes dois grandes grupos, existem outras ramificações dos testes, quem podem começar cedo, já na etapa de análise e desenvolvimento, e se estender até após a entrega do sistema, contando com auxílio do próprio cliente para alguns dos testes finais. Mais abaixo estarão listados as principais diferenças entre os testes de caixa preta e caixa branca, e os principais subgrupos de teste.

2.3.1 Testes de Caixa preta

Testes de caixa preta como são popularmente conhecidos, são teste funcionais. Eles enxergam o sistema como uma caixa preta fechada, tudo que é observado são as entradas e as saídas, não importando de que maneira o código funciona para entregar tais resultados (NIDHRA; DONDETI, 2012, tradução nossa).

Na figura 1 pode-se perceber que o teste consiste em uma entrada de dados, um processamento desconhecido ocorrendo dentro da caixa preta, e uma saída de dados. A validação ocorre apenas comparando a entrada e a saída de dados.

Figura 6 – Teste de Caixa preta



Fonte: Rongala (2015).

É um tipo de teste geralmente feito por uma equipe de testes separada, que não toma conhecimento do funcionamento interno do sistema, essa equipe está mais ligada aos requisitos passados pelo cliente, e a análise feita pelos analistas do sistema. Os testes de caixa preta podem começar a ser montados já durante o início do desenvolvimento das análises de sistema, e podem seguir até o final do projeto.

A vantagem desse teste é que os profissionais envolvidos não precisam ter conhecimento da linguagem de programação, e podem ser uma equipe separada do

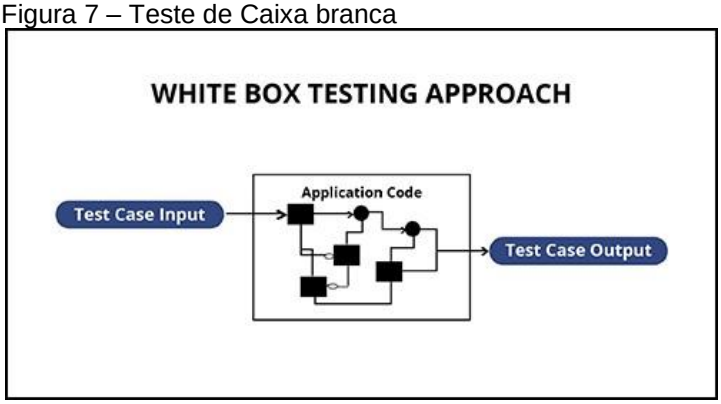
desenvolvimento. É bastante útil para pegar falhas nas especificações e nos requisitos passados pelo cliente ou pelos analistas. E como não estão preocupados com o código, podem focar também em usabilidade, propondo melhorias ou mudanças que tragam mais simplicidade a interface (NIDHRA; DONDETI, 2012, tradução nossa).

2.3.2 Testes de Caixa branca

Testes de caixa branca, também conhecidos como testes estruturais, são aqueles onde a equipe de testes tem participação ativa no desenvolvimento. Os testes são feitos levando-se em conta todas as funcionalidades internas do código do sistema, a lógica aplicada ao código e individualmente, cada umas das funções com suas tomadas de decisão (NIDHRA; DONDETI, 2012, tradução nossa).

A figura 2 exemplifica mostrando a entrada de dados, um processamento conhecido dentro da baixa branca, e a saída dos dados. A validação nesse caso considera a entrada de dados, o caminho percorrido por ela em cada método e as saídas geradas por cada método.

Figura 7 – Teste de Caixa branca



Fonte: Rongala (2015).

Este teste é utilizado durante todo o estágio de desenvolvimento do código do software. E tem como funções principais encontrar erros de lógica, tomadas de decisão incorretas, erros de ortografia e padrões de código. Geralmente é feito pela própria equipe de desenvolvimento ou por uma equipe de testes que tenha conhecimento da linguagem de programação e da estrutura de desenvolvimento do sistema.

A principal vantagem desse tipo de teste está no fato de ele ser uma forma mais eficiente de se encontrar e corrigir os problemas. Apenas “varrendo” o código é

possível identificar uma série de erros básicos que podem ser difíceis de encontrar, ou mesmo passarem despercebidos pelos testes funcionais.

Além disso, as correções são aplicadas imediatamente, poupando toda a etapa burocrática onde os problemas são reportados, corrigidos, uma versão é liberada e um novo teste é feito para validar as correções. Dessa forma, o erro é encontrado e corrigido na mesma etapa, poupando tempo e recursos de desenvolvimento (NIDHRA; DONDETI, 2012, tradução nossa).

2.3.3 Testes de desenvolvimento

Sommerville (2012) explica que são processos de teste executados ainda na fase de desenvolvimento do sistema, pelos próprios programadores, ou mesmo por testadores mais ligados ao código. São testes diferentes daqueles realizados nas fases posteriores de validação e geralmente focam em três frentes de validações diferentes, todas ligadas ao código fonte.

2.3.3.1 Teste unitário

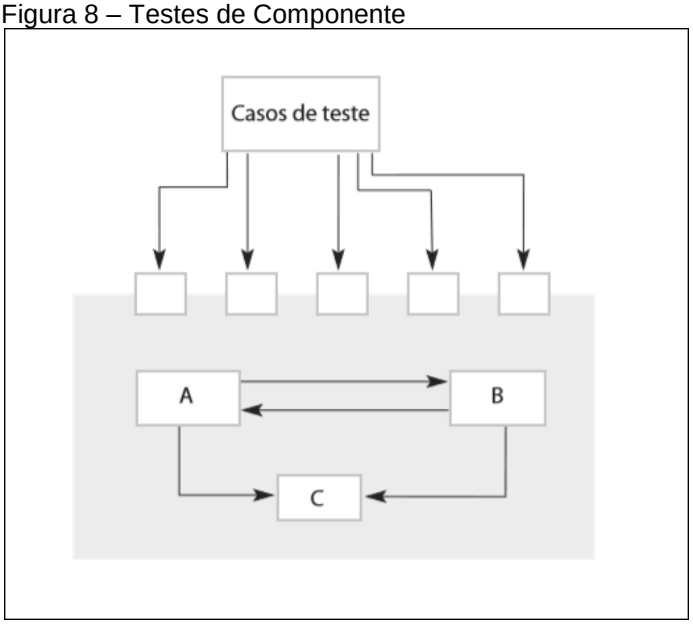
O primeiro dentre os testes de desenvolvimento tem por objetivo focar nas menores partes do código fonte, como as classes de objetos e os métodos. Essas validações são feitas não somente na classe onde as operações estão inseridas, mas também nas heranças que fazem uso dessas operações.

Todas as operações de um objeto devem ser verificadas, todas as possíveis entradas de dados devem ser atribuídas, e todas as mudanças de status devem ser simuladas. São feitos testes também com valores de entrada diferentes daquilo que é esperado. Os programadores geralmente desenvolvem as funções com as operações recebendo valores de entrada ideais para o cenário em questão. Então é interessante fazer uso de dados que estão nos limites inferiores ou superiores deste cenário, ou que forcem operações que irão gerar resultados muito pequenos ou muito grandes. Tudo aquilo que foge ao padrão ideal de operação pode apresentar problemas (SOMMERVILLE, 2012).

2.3.3.2 Teste de componente, ou integração.

No teste anterior foi visto que as validações são realizadas individualmente em cada componente. Já neste modelo, a integração de vários componentes é o objeto de teste. Geralmente o nome dado a essa integração é interface, e é ela que será testada. Pode-se questionar a necessidade de realizar tal teste, visto que os componentes individuais já foram todos testados, mas para Sommerville (2012) essa afirmação não é válida, pois os erros que podem ser encontrados aqui, resultam da integração e comunicação feita pela interface com os componentes, tais erros podem não aparecer em testes individuais.

Figura 8 – Testes de Componente



Fonte: Sommerville (2012).

Na figura 3 logo acima pode-se ver de forma mais clara o que foi dito anteriormente. Os casos de teste estão sendo aplicados para validar a interface que faz a ligação entre os componentes individuais A, B e C. Fica claro perceber que o teste individual de cada componente deixaria áreas sem cobertura de teste.

“Alguns dos erros mais comuns em grandes sistemas são os encontrados nas interfaces...” (LUTZ, 1993, apud SOMMERVILLE, 2012). Bons exemplos são quando um componente faz uma chamada para outro, porém usa a sua interface de forma errada, passando tipos de dados errados, na ordenação errada, ou em quantidades incorretas.

Também é possível encontrar erros desse tipo ocasionados pela má sincronia nos tempos de execução. Supondo que um componente abasteça um repositório com dados, e o outro componente leia esses dados. Se os componentes não estiverem sincronizados, dados podem ser lidos no momento errado, causando falhas no sistema.

2.3.3.3 Teste de sistema

Aqui é onde todos os componentes e interfaces são integrados para formar uma versão do sistema. É um ponto importante da fase de testes pois muitos dos componentes podem ser desenvolvidos por equipes diferentes, e como já visto anteriormente, grande parte dos problemas encontrados são decorrentes de uma comunicação incorreta entre as partes do sistema. Especialmente quando existe a integração de componentes reutilizáveis, com novos recém desenvolvidos.

A melhor forma de testar um ambiente com tantas interações é através dos casos de uso, que diferentemente dos casos de teste, possuem alto nível de abstração de requisitos, e são situações mais reais de uso. Através deles é possível forçar a interação entre vários componentes, e fazer emergir os problemas.

Apesar de muito válido, esse tipo de teste muitas vezes traz consigo um problema. Que é quando definir a hora de parar de testar. Pois como os cenários aqui envolvem grandes quantidades interações e resultados possíveis, é inviável criar um teste para cada situação diferente. Fica a cargo da empresa definir se seguirá por uma estratégia de testar o maior número de resultados possíveis, ou planejar as situações mais comuns de uso e traçar testes apenas para estes casos (SOMMERVILLE, 2012).

2.3.4 Testes de release

Ocorrem quando todos os testes de caixa branca já foram executados. O sistema então passa a ter uma versão liberada para testes, ou seja, um “release” de testes. Sommerville (2012) diz que esses testes são aplicados por uma equipe separada, sem contato com os processos de desenvolvimento do sistema, ou seja, é um teste de caixa preta. E segundo ele, o principal objetivo nesta etapa é garantir que o sistema esteja bom o suficiente para ser distribuído aos clientes. O foco então, passa

a ser o desempenho, a segurança e as funcionalidades do software, inclusive, é daí que surge o nome mais comum desse tipo de teste, os testes funcionais.

2.3.4.1 Testes baseados em requisitos

Um dos princípios gerais das boas práticas de engenharia de requisitos é que os requisitos devem ser testáveis (SOMMERVILLE, 2012, p. 157) e para um requisito ser testável, deve ser possível planejar um teste para ele.

Nesse tipo de teste a abordagem deixa de ser encontrar erros e passa a ser validar requisitos, ou seja, garantir que o sistema irá cumprir aquilo que estava especificado. O teste pode parecer simples, utiliza-se uma entrada de dados e verifica-se o resultado apresentado na saída. Mas existem várias formas diferentes de fazer isso.

Por exemplo, imagine um sistema de compra e venda de ações na bolsa de valores. O sistema deve emitir uma mensagem de aviso quando o usuário tenta vender uma ação por um preço menor que o mínimo definido.

O teste de requisito seria então verificar se tal mensagem aparece ao vender uma ação por um preço muito baixo. Mas isso garantiu apenas uma parte do requisito. Ele não validou o cenário contrário, que seria vender a ação pelo preço normal. Nesse caso, a mensagem de validação não deve aparecer. Ou mesmo, vender duas ações ao mesmo tempo, uma abaixo e outra acima do valor mínimo. O que deveria acontecer então?

Dessa forma fica estabelecido que para validar um requisito, são necessários vários testes (SOMMERVILLE, 2012).

2.3.4.2 Testes de cenário

Para Kaner (2003), esse modelo de testes é bem parecido com o caso de uso. Ele deve contar uma história crível e complexa, que seja baseada em um cenário real de uso e que tenha princípio, meio e fim. Escrevendo um teste dessa forma é possível garantir a integração de vários requisitos diferentes para ver como vão reagir ao trabalhar em conjunto.

O testador deve se colocar no papel do personagem da história e seguir os passos dele, tomando a liberdade para tentar fazer as coisas erradas simulando um

usuário inexperiente. Dessa forma pode-se medir o comportamento do sistema frente a entradas de dados incorretas, tempos de processamento interrompidos por usuários impacientes, e ao mesmo tempo, checar o comportamento do sistema quando vários de seus recursos estão sendo utilizados simultaneamente (apud SOMMERVILLE, 2012).

Pode-se usar como exemplo aqui um sistema de envio de mensagens. O usuário fictício Felipe acessa o sistema através do seu login e senha, para ler uma mensagem da corretora de investimentos que lhe solicitava uma cópia da identidade, ele clica em responder, escreve um texto, anexa o documento, e aproveita para adicionar um amigo que tem mais experiência no assunto como cópia da mensagem. Nessa pequena história foi possível testar uma grande quantidade de requisitos e validar a interação de várias partes diferentes do código.

2.3.4.3 Testes de desempenho

Como o nome sugere são testes que buscam validar a capacidade de trabalho do sistema e a segurança do mesmo diante de cenários de grande estresse.

Este teste tem por objetivo garantir que o sistema atenda a um requisito mínimo de carga de trabalho, mas também busca expor erros que possam ocorrer nesses limites de trabalho. Sommerville (2012) frisa a importância de serem criados testes para além da carga máxima pretendida, pois assim, além de ter um melhor entendimento da real capacidade de trabalho do sistema, pode-se descobrir falhas que ocorrem apenas nestes extremos.

É importante garantir que o sistema trabalhe com um desempenho satisfatório até os limites definidos nos requisitos, mas também é importante garantir que quando as falhas ocorrerem além desses limites, os dados que estão em processamento não sejam corrompidos, e que o sistema não entre em uma falha geral que cause a queda do mesmo.

Por mais que estes cenários sejam difíceis de se encontrar e estejam além daquilo que foi definido nos requisitos, circunstâncias atípicas podem se combinar e culminar em uma situação real de estresse do sistema para além daquilo que ele fora projetado. E apesar de estar previsto que o sistema pare de responder, é importante garantir que informações sensíveis não sejam perdidas.

2.3.5 Testes de usuário

São aqueles onde o usuário final ou cliente, auxilia na fase de testes. Esses testes podem ocorrer em três momentos diferentes chamados de testes alfa, beta e testes de aceitação. E tem por objetivo lapidar o sistema para melhor atender o usuário.

Os desenvolvedores, por mais focados que estejam em criar um sistema estável e que funcione perfeitamente na realidade do cliente, não podem evitar que certas situações passem despercebidas. O uso que um software irá encontrar no cliente é muito diferente daquilo que ele encontra no seu ambiente de produção. Várias situações que fogem ao controle proporcionam formas de uso do software que irão acarretar em falhas. E para isso é imprescindível a implementação dos testes de usuário (SOMMERVILLE, 2012)

2.3.5.1 Teste alfa

Os testes alfa ocorrem ainda durante as fases iniciais de desenvolvimento do sistema. Os usuários finais trabalham direto com os desenvolvedores ou a equipe de testes, experimentando uma prévia do sistema e analisando se o que está sendo apresentado atende as suas necessidades. Isso promove sugestões de melhorias e até mesmo o revela problemas que sem isso só seriam percebidos muito depois, já nas fases de teste de release ou mesmo em testes de usuário (SOMMERVILLE, 2012)

2.3.5.2 Teste beta

Esse tipo de teste é frequentemente utilizado por desenvolvedores que não podem simular todos os ambientes onde o seu sistema vai ser utilizado. Nesse caso, o sistema não é necessariamente de um cliente específico, mas sim de vários clientes. Então, uma versão de release completa ou incompleta é liberada para alguns usuários finais, ou para qualquer pessoa que demonstre interesse em colaborar. Isso traz resultados importantes pois cada usuário tem um ambiente de uso diferente, com isso uma grande quantidade de ambientes são testados em um curto espaço de tempo, revelando possíveis problemas (SOMMERVILLE, 2012).

2.3.5.3 Teste de aceitação

Nesse último tipo o cliente realiza os testes no software para validar se o mesmo será ou não aceito. É claro que esse tipo de teste ocorre desde o início do desenvolvimento, do contrário, a empresa poderia desenvolver um sistema completo apenas para chegar na entrega e ter sua solução não aceita pelo cliente. O cliente vem participando desde as fases de planejamento para avaliar os requisitos e compartilhar suas situações de uso. Segundo Sommerville (2012), é fácil imaginar que se um teste de aceitação for reprovado, o contrato é cancelado. Mas a realidade é diferente, o cliente já teve custos de migração e investimentos em hardware para se adequar ao novo sistema, ele tem grande interesse em começar a utilizar o sistema o mais rápido possível, e quase sempre é menos custoso usar o sistema com defeitos, do que não usar sistema algum, por isso existe um consenso entre as partes. O cliente cumpre seu papel em utilizar e validar o sistema, e a empresa se compromete em solucionar qualquer problema existente o mais rápido possível.

2.4 AUTOMAÇÃO DE TESTES

Como visto anteriormente, testes são uma parte fundamental no desenvolvimento de software, um software não terá valor se não tiver passado por testes que garantem a sua funcionalidade. Para garantir isso uma grande equipe é necessária, desde os níveis mais iniciais do desenvolvimento, até o estágio final, já com o software nas mãos do cliente. Fazer isso, como já dito anteriormente por Myers, Badgett e Sandler (2012, tradução nossa), custa caro, cerca de 50% do custo total do desenvolvimento. Com tanto dinheiro em jogo, é natural que as empresas busquem formas de otimizar esse trabalho. E como em quase todas as áreas da indústria, seja ela de software ou não, uma palavra está sempre presente quando se fala sobre otimizar. É a Automação.

O conceito geral de automação segundo Wiklund (2015, tradução nossa) é uma operação automaticamente controlada de um aparato, processo ou sistema por dispositivos mecânicos ou eletrônicos em substituição ao trabalho manual humano.

Na maioria das vezes essa automação nada mais é do que robôs utilizados para executar processos repetitivos e dispendiosos para um ser humano. Isso acaba tendo uma aplicação muito boa também aos testes de software, pois ele nada mais é

do que um grande roteiro com vários passos a serem seguidos, e a cada passo, uma validação do resultado obtido. Seguir passos e comparar resultados é exatamente o trabalho de um sistema de automação.

Antes de falar sobre a automação aplicada aos testes de software, é preciso esclarecer alguns conceitos utilizados nessa área. O quadro 2 traz alguns dos exemplos mais comuns com suas explicações.

Quadro 2 – Conceitos gerais de automação de testes.

Gravação	Grande parte das ferramentas de automação permitem gravar as ações do usuário e armazená-las em uma linguagem de programação.
Script de teste	É o conjunto de comandos que serão executados no sistema. Eles são criados com base nos casos de teste.
Execução	É quando o script de teste é executado, exibindo visualmente ou não, todos os passos sendo seguidos.
Mapeamento de objetos	Muitas ferramentas permitem que elementos do sistema como botões e campos por exemplo, sejam armazenados para reutilização durante os scripts. Esse mapeamento permite que seja possível ver os atributos destes elementos, como nomes e valores por exemplo.

Fonte: MOLINARI (2014).

Existem algumas diferenças entre a automação na indústria e a automação nos testes de software. Uma delas é a evolução do objeto de teste.

Wiklund (2015, tradução nossa) diz que muitos setores da automação na indústria são estáticos, ou seja, nunca mudam. E com isso, quando um certo padrão de qualidade for atingido o processo de automação está pronto e pode ser utilizado da mesma forma até o fim da produção.

Na área de sistemas isso não é possível, pois o objeto de teste em questão é um software que está em constante estágio de evolução, crescimento, correções e melhorias. Logo, uma ferramenta estática que faz sempre as mesmas coisas não vai servir. A ferramenta precisa evoluir junto com o sistema para ser útil.

Essa visão mostra um pouco que o teste automatizado se comporta de forma bastante similar ao próprio sistema ao qual ele irá testar, pois precisa de manutenção, mudanças e atualizações constantes para que cumpra sua função. Tais manutenções, são feitas por uma equipe responsável por manter o teste automatizado (WIKLUND, 2015, tradução nossa).

Molinari (2014) mostra que nesse ponto surge uma grande dúvida na cabeças dos gerentes de projeto. Vale a pena investir em automação, ou é melhor

continuar com testes manuais? E é aqui que muitos projetos de automação já começam da maneira errada. Muitos dos gerentes se deixam levar pelas propagadas dos fabricantes de ferramentas de automação, e acreditam que a ferramenta trará uma solução completa para os testes de sua empresa. Pensam que a solução é de fácil implantação, baixo custo de manutenção e trará resultados ótimos. Isso de fato pode ser verdade, mas não para todos os tipos de projeto. Antes de mais nada é preciso fazer um estudo para validar qual tipo de ferramenta melhor se enquadra aos sistemas que serão testados, além disso, devem ser definidos de forma clara quais tipos de teste a ferramenta deverá fazer, pois eles é que estão diretamente ligados ao sucesso ou ao fracasso de um projeto de automação de testes de software.

Iniciar um projeto de automação geralmente é um processo demorado, e muitas vezes o projeto cai nas mãos de um tester sem experiência em desenvolvimento. Com esforço, essa pessoa poderá aprender, mas não terá o conhecimento necessário para avaliar se as decisões que ele está tomando são as mais indicadas para a realidade da empresa. O ideal é alocar pelo menos dois profissionais experientes para pesquisar e definir as tecnologias e métodos que serão utilizados, para só então iniciar os trabalhos.

Molinari (2014) aponta ainda que uma boa forma de validar se a automação é aplicável para a realidade de um projeto, é verificar em primeiro lugar, o tempo que será necessário para gravar os testes. E depois, de quanto em quanto tempo esses testes precisarão ser executados. Logicamente, se a gravação do teste for demorada, e a execução dele acontecer esporadicamente, esse não é o teste ideal para ser automatizado.

Por outro lado, se a gravação for rápida, e o teste é rodado com frequência. Este sim é um bom candidato para a automação. Um bom exemplo disso são os testes de login, onde basta gravar a ação de logar, checar os resultados possíveis, e armazenar uma lista com nomes de usuário e senha, válidos e inválidos. Pronto, o teste já pode ser executado inúmeras vezes, rapidamente.

Outro cenário onde a automação se faz bastante útil, se não quase obrigatória, é nos testes de desempenho ou carga. Imagine um botão que executa um processamento e gera um resultado. Agora imagine que centenas ou milhares de usuários podem tocar nesse botão ao mesmo tempo. Testar isso manualmente é inviável pois requer um grande esforço para poder coordenar uma equipe que clique nesse botão exatamente ao mesmo tempo.

Já com a automação, basta executar o mesmo script simultaneamente em ambientes diferentes. O custo para automação nesse caso, é muito inferior se comparado ao teste manual (MOLINARI, 2014).

Mas antes de mais nada, é preciso definir quais combinações de ferramentas que serão adotadas, pois existem diferentes testes a serem executados, e diferentes sistemas a serem testados, e cada um deles se dará melhor com uma ou outra ferramenta.

2.4.1 Ferramentas para automação

O mercado atual para automação de testes conta com uma ampla variedade de ferramentas, sejam elas pagas ou gratuitas, cada uma tem suas funções, particularidades e especialidades. Para atingir os objetivos propostos, algumas ferramentas foram selecionadas, todas elas possuem versões gratuitas para utilização, e foram essas as versões escolhidas para utilização neste trabalho. Este capítulo falará sobre cada uma delas.

2.4.1.1 Selenium

Selenium é um framework para testes de software Web que facilita a automatização de navegadores. (PEIXOTO, 2018, pg. 2)

O projeto teve um início em 2004, quando Jason Huggins estava testando um sistema na empresa ThoughtWorks. Ele desenvolveu uma pequena biblioteca em Javascrpit que permitia realizar interações com a página web. Essa ferramenta eventualmente se tornou o Core do Selenium (SELENIUM HQ, 2018, tradução nossa)

Em 2006, O Google já utilizava o Selenium nas suas rotinas de testes. Mas como a biblioteca original rodava em Javascript, ela era limitada pelo Sandbox.

Sandbox é um programa que restringe a utilização de softwares a uma área isolada do disco rígido. Dessa forma, toda e qualquer operação realizada pelo software não poderá afetar nada que esteja fora dessa área de contenção no disco (RIBEIRO, 2014).

Simon Stewart, que era engenheiro do Google começou a desenvolver um projeto chamado WebDriver, que visava eliminar essas limitações, fazendo as comunicações com o navegador diretamente, sem as restrições do sandbox. Esses

esforços acabaram por resultar em 2008 na fusão dos dois projetos, gerando assim o Selenium WebDriver, ou Selenium 2 (SELENIUM HQ, 2018, tradução nossa).

O Selenium possui ao todo 4 produtos:

- Selenium WebDriver (Ou Selenium 2): É o resultado da fusão dos dois projetos mencionados anteriormente. Ele combina as já conhecidas funcionalidades da biblioteca Javascript do projeto original, com as novas capacidades do WebDriver. Permite a execução dos testes nos principais navegadores disponíveis no mercado. Isso é possível através de plug-ins de terceiros que rodam uma instância do navegador, permitindo que o WebDriver tome controle das ações.

  Além disso, o WebDriver é uma API orientada a objetos e permite a utilização de múltiplas linguagens de programação, como Java, C#, Ruby, Python, Javascript (SELENIUM HQ, 2018, tradução nossa).

- Selenium "*Integrated Development Environment*" (IDE)[1]: É uma ferramenta de prototipagem que permite a criação e execução de scripts de teste. Conta com um gravador que armazena os comandos e entradas de dados inseridos pelo usuário em um script, que pode ser posteriormente exportado em alguma das linguagens de programação suportadas.

  Essa ferramenta é uma extensão disponível para os navegadores Google Chrome e Mozilla Firefox (SELENIUM HQ, 2018, tradução nossa).

- Selenium Server: É um servidor que permite a execução dos testes em uma máquina diferente da qual os testes foram escritos. Ele cria um hospedeiro no qual outras máquinas podem se conectar para serem utilizadas como ambientes de execução dos testes (SELENIUM HQ, 2018, tradução nossa).

[1] IDE – Ambiente de desenvolvimento integrado

- Selenium Grid: Através do uso do Selenium Server, o Selenium Grid permite a criação de um ambiente de execução de testes distribuídos, onde é possível rodar em paralelo, testes em diferentes máquinas, com diferentes navegadores.

  Sua utilização é bem aplicada quando a intenção do teste é validar o sistema em vários ambientes diferentes, com sistemas operacionais e versões de navegadores diferentes. Ao invés de executar um teste por vês, é possível executá-lo em todos os ambientes desejados ao mesmo tempo. Também é válido utilizar esse recurso quando se deseja reduzir o tempo total de execução dos testes. O ambiente pode ser configurado para que várias máquinas rodem uma parte do teste. Isso reduz o tempo total de execução pelo número de máquinas rodando em paralelo (SELENIUM HQ, 2018, tradução nossa).

O Selenium é gratuito e conta com uma grande comunidades de apoiadores, dispondo de muito material de suporte para a utilização da ferramenta (DAVE, 2016, tradução nossa).

Por esse motivo ele foi o framework de testes escolhido para o desenvolvimento deste projeto.

2.4.1.2 TestNG

Segundo Beust (2004, tradução nossa), o criador do TestNG, essa ferramenta foi desenvolvida através da inspiração no JUnit e no NUnit, mas promovendo algumas mudanças e melhorias que Beust sentia necessidade ao utilizar as ferramentas originárias. E em primeiro de setembro de 2004 a primeira versão oficinal do TestNG foi liberada ao público.

Ele é um framework de testes que utiliza anotações para gerenciar a execução e organização dos testes. A organização dos testes pode ser feita através das anotações e de um arquivo no formato XML que define, ordenação do teste, exclusão de métodos, parâmetros e outros fatores. E com isso, todo um cenário de testes pode ser reagrupado e alterado sem a necessidade de recompilar o código.

Em resumo, o TestNG permite que a maneira de testar fique separada do código Java. Ele pode ser utilizado em testes unitários, funcionais, regressão e integração (Beust, 2004, tradução nossa).

Alguns dos conceitos e anotações mais utilizados no TestNG podem ser vistos no quadro 3 exibido logo abaixo.

Quadro 3 – Conceitos e anotações mais comuns do TestNG

Suite	É um arquivo no formato XML contento todo o escopo de um ou mais testes. É representado dentro do arquivo XML pela tag <suite>.
Test	É um teste específico, e ele pode conter uma ou mais classes TestNG. É representado dentro do arquivo XML pela tag <test>.
Classe TestNG	É uma classe Java que contenha pelo menos uma anotação @Test. Pode ter um ou mais métodos, e é representada dentro do arquivo XML pela tag <class>.
Método TestNG	É um método Java que possua a anotação @Test antes de sua definição.
@BeforeSuite	Métodos com essa anotação serão executados antes do início da suite começar.
@AfterSuite	Métodos com essa anotação serão executados após o término da suite.
@BeforeTest	Métodos com essa anotação serão executados antes de cada tag <test> dentro da suite.
@AfterTest	Métodos com essa anotação serão executados após o término de cada tag <test> dentro da suite.
@BeforeClass	Métodos com essa anotação serão executados antes de classe TestNG.
@AfterClass	Métodos com essa anotação serão executados após o término de cada classe TestNG.
@BeforeMethod	Métodos com essa anotação serão executados antes de cada método TestNG.
@AfterMethod	Métodos com essa anotação serão executados após o término de cada método TestNG.

Fonte: TESTNG (2018).

O TestNG ainda permite a utilização de parâmetros que podem ser passados pela suite XML e utilizados por métodos anotados com @Test. Esses parâmetros em conjunto com outras funcionalidades de paralelismo, podem ser utilizados para facilitar o controle de testes executados nos ambientes distribuídos do Selenium Grid, permitindo definir os navegadores a serem utilizados, e quantas Threads de processamento devem ser alocadas para cada execução (TESTNG, 2018, tradução nossa).

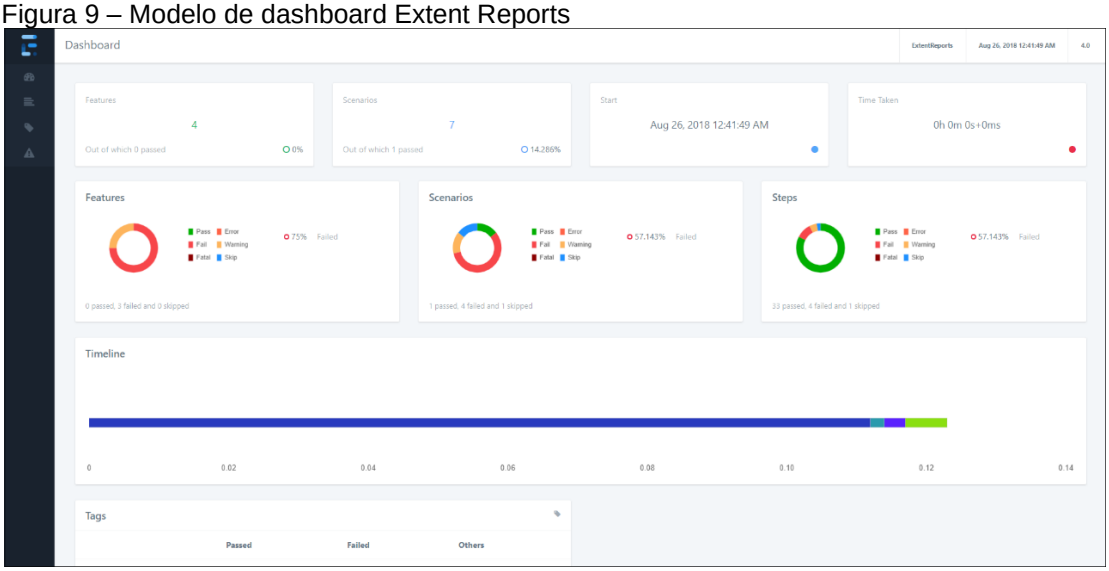
É possível também a geração de relatórios detalhados sobre a execução dos testes, porém neste trabalho será utilizado um framework próprio para isso, o Extent Reports, que possui mais possibilidades de customização e alguns recursos adicionais.

2.4.1.3 Extent Reports

Avilala (2018, tradução nossa) mostra que o framework Extent Reports permite a criação de relatórios mais elaborados sobre a execução dos testes. Ele gera Dashboards interativos e visualizações por gráficos que permitem analisar de forma mais clara em qual parte do teste houve falhas.

A figura 4 mostra um exemplo de relatório gerado pela ferramenta.

Figura 9 – Modelo de dashboard Extent Reports



Fonte: Extent Reports (2018).

O framework possui recursos para anexar capturas de tela do erro encontrado e possibilidades de enviar os resultados por e-mail (na sua versão comercial) para os responsáveis do projeto.

Pode ser configurado para rodar com as linguagens Java e .Net, além de possuir integração com os frameworks de teste JUnit, TestNG e NUnit.

A ferramenta funciona através da criação de testes e filhos de testes, o primeiro possui nível hierárquico 0, seu filho possui nível 1 e assim por diante, podendo haver vários níveis hierárquicos (EXTENT REPORTS, 2018, tradução nossa).

Cada teste ou filho pode possuir um método com um log que irá gravar no relatório se o teste resultou em sucesso, falha, ou foi pulado. A ferramenta sabe quando houve falha no teste através de sua integração com o framework de testes, como o TestNG por exemplo (OTWANI, 2015, tradução nossa).

O Extent Reports lê um método do TestNG que guarda o estado do teste. É feita uma comparação para verificar o estado do teste, e com base no resultado, o log apropriado é chamado. Nesse momento também é possível inserir algumas funções específicas, como a de anexar uma captura de tela junto com o log de falha. Para isso é chamado um método parar realizar a captura da imagem, e o método do Extent Reports que anexa essa imagem ao corpo do relatório (EXTENT REPORTS, 2018, tradução nossa).

Boa parte dos recursos mais avançados dessa ferramenta estão disponíveis apenas na versão comercial, porém, tudo o que foi descrito acima está disponível também na versão community que é gratuita.

2.4.2 Dificuldades para automação de testes

Como foi visto nos capítulos anteriores, a área de automação de testes possui nos dias de hoje bastante recursos disponíveis para auxiliar a equipe de testes na hora de validar um sistema. Existem a disposição várias ferramentas e muitos frameworks que podem ser combinados para transformar um projeto de teste automatizado numa ferramenta poderosa, capaz de realizar validações através de testes unitários, e até mesmo testes mais complexos como os funcionais, e ao final poder ainda enviar por e-mail os resultados dos testes de forma clara e interativa para os gestores do projeto.

Mas mesmo com tudo isso, a automação não resolve todos os problemas. E possui algumas dificuldades que são encontradas ao se iniciar um projeto de automação de testes.

Wiklund (2015, tradução nossa) fez um estudo onde mostra que existem dois conceitos para as dificuldades encontradas na automação. São os impedimentos, e as dívidas técnicas.

2.4.2.1 Impedimentos

São obstáculos encontrados durante o desenvolvimento da automação que geram atrasos. Eles não impossibilitam o desenvolvimento, mas geram mais trabalho pois a equipe responsável precisa buscar formas de contornar os problemas (WIKLUND, 2015, tradução nossa).

2.4.2.2 Dívida técnica

Esse é um termo que surgiu pra exemplificar a diferença entre o estado em que o projeto se encontra, e o estado ideal em que ele deveria estar. Geralmente essa diferença acontece quando ações são tomadas para cumprir prazos ou reduzir custos. Essa ações na maioria das vezes acabam por trazer problemas no futuro, sendo necessário um tempo muito maior para refazer o trabalho (WIKLUND, 2015, tradução nossa).

Essa diferença também pode acontecer quando as ferramentas escolhidas para o projeto não são as ideais. O time responsável não tinha conhecimento suficiente na área, e escolheu ferramentas que aparentavam atender suas necessidades iniciais, mas não previram cenários futuros. Com isso, ações tiveram que ser tomadas para contornar os problemas que as ferramentas não atendiam, e o resultado são os atrasos (WIKLUND, 2015, tradução nossa).

Mcpeak (2017, tradução nossa) mostra alguns dos problemas mais comuns na hora de automatizar testes. Dentre eles estão as Janelas Pop-Up.

Dependendo do tipo de janela, a ferramenta pode muitas vezes não conseguir interagir com aquele elemento, sendo necessários tomar outras abordagens para passar pelo problema.

Um problemas muito comum são as páginas dinâmicas que existem nos dias de hoje. Conteúdos aparecem o somem o tempo todo, e pode ser difícil fazer a ferramenta entender que precisa esperar determinada ação ocorrer antes de prosseguir com o teste. (MCPEAK, 2017, tradução nossa). Muitas vezes é adotada a solução fácil de gerar uma espera por um tempo programado, mas isso nunca é uma boa solução, visto que dependendo da velocidade do ambiente de testes, esse tempo pode variar e o teste pode quebrar. Outro problema também relacionado com o dinamismo das páginas, são os localizadores dos elementos. A melhor forma de localizar um elemento é pela propriedade ID, porém quando o elemento é dinâmico, essa propriedade possui um código que muda a cada teste, sendo necessário tomar outras abordagens pra guardar um localizador válido para esse elemento.

Outro problema é a instabilidade do teste. Geralmente relacionada aos impedimentos anteriores, por dificuldade de usar um localizador válido, ou realizar

uma espera estável, o teste pode resultar em falsos positivos. Hora apresenta um resultado, hora outro (MCPEAK, 2017, tradução nossa).

E referente aos problema de dívida técnica, tem-se como exemplo a escalabilidade do projeto. O que começa simples e traz bons resultados, pode ir crescendo exponencialmente junto com o sistema, e se não forem adotadas boas práticas de automação, somadas a uma constante pesquisa em busca de aperfeiçoamento por conta da equipe de testes, manter essa estrutura funcionando pode ficar complicado e acabar por trazer muitos transtornos ao projeto (MCPEAK, 2017, tradução nossa).

Por fim, tem-se ainda as expectativas irreais dos testers ou gerentes de projeto. Muitos acreditam que a ferramenta de automação pode realizar qualquer tarefa, e logo começam a escrever testes para tudo. Sendo que na realidade o melhor a se fazer é definir no início do projeto o que se espera da automação, e quais testes devem ser automatizados. Pois é fato que existem testes onde não é possível substituir o bom e velho teste manual (MCPEAK, 2017, tradução nossa).

Um bom exemplo de onde as ferramentas de teste automatizado encontram problemas são na hora de validar dados gráficos exibidos na tela. Eles conseguem interagir com elementos e tomar muitas ações, mas tem dificuldade na hora de interagir com informações visuais por não conseguir ainda simular um tester real olhando para a tela. Relacionado a isso temos os relatórios gerados pelos sistemas.

Boa parte dos sistemas atuais tem pelo menos um relatório que pode ser gerado. Esses relatórios normalmente contém uma massa de dados extraídas do sistema por comandos via banco de dados. E não é incomum encontrar dados com erro no relatório, enquanto esse mesmo dado está correto dentro do sistema. Isso acontece pois os selects feitos para emissão do relatório podem ter erros, ou até mesmo operações aritméticas que podem ser feitas na geração do relatório tenham algum erro de cálculo que passou despercebido aos programadores.

E como validar relatórios normalmente é uma tarefa onerosa e cansativa, é comum que os testers deixem passar alguma coisa, afinal a quantidade de dados pode ser bem grande dependendo do relatório extraído. Com isso, o erro acabou passando despercebido por todas as etapas de testes, os programadores não viram o select errado no relatório, a equipe de testes e os testes automatizados não viram o erro no sistema, pois o mesmo só está presente no relatório. E por fim, o tester

responsável por validar o relatório deixou passar o problema pois já estava cansado de validar relatórios.

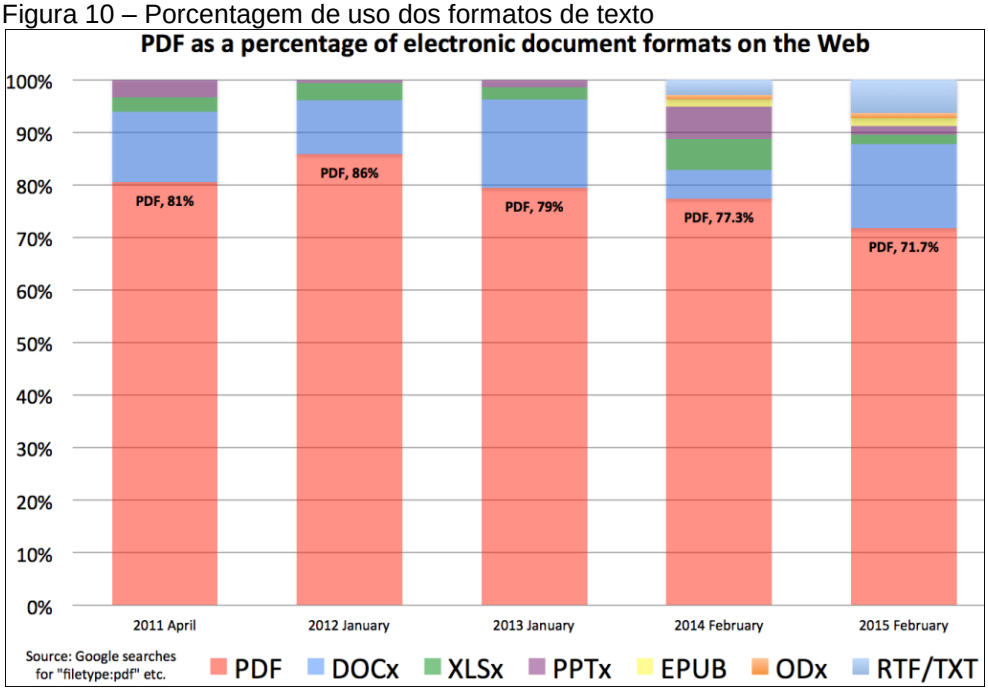
Nesse ponto surge a importância de criar um teste automatizado que seja capaz de ir além do escopo do sistema, e consiga validar os dados dentro do relatório.

3 COMPARAÇÃO DE ARQUIVOS PDF

Lazarte (2008, tradução nossa) mostra que o PDF é um dos formatos mais comuns para documentos eletrônicos, e tornou-se inclusive o padrão internacional da ISO para documentos.

A figura 5 destaca o percentual de uso dos formatos de texto mais comuns ao longo dos anos. PDF está em vermelho, com 71.7% em 2015.

Figura 10 – Porcentagem de uso dos formatos de texto



Fonte: Johnson (2015).

O formato da Adobe apresentado em 1992 tinha como intuito unificar toda a documentação das empresas em um formato que pudesse ser transmitido, visualizado e imprimido de qualquer computador rodando qualquer sistema operacional ou configuração distinta (SMITH, 2018, tradução nossa).

Ao longo dos anos o formato se espalhou cada vez mais e acabou por atingir os padrões que podem ser observados hoje, conforme visto na figura 5.

Sendo o formato mais comum de documento digital, muitas ferramentas surgiram para suprir necessidades relacionadas a este tipo de documento. E uma dessas necessidades é a comparação de documentos PDF.

3.1 MÉTODOS DE COMPARAÇÃO

Existem dois principais métodos para realizar a comparação de arquivos digitais, neste caso, arquivos no formato PDF. O primeiro busca a extração do texto para comparação textual, e o segundo varre o arquivo no nível dos bytes para encontrar diferenças.

3.1.1 Comparação textual

Ravichandran e Rekha (2013, tradução nossa) mostra que em arquivos no formato PDF existem objetos de string para armazenar os textos. Estes objetos por sua vez guardam informações referentes aos caracteres e seus estilos de fonte, tamanho, cor e etc. Ferramentas de extração conseguem extrair partes do texto e armazenar num formato editável.

Em conjunto com isso, existem algoritmos de reconhecimento ótico de caracteres, do inglês Optical character recognition (OCR), que de forma resumida, lê o arquivo como uma imagem em busca de padrões pré-estabelecidos, quando o padrão encontrado na imagem coincide com o que é conhecido, a ferramenta sabe que se trata de um caractere e grava esse valor. A combinação desses dois fatores permite que o texto seja extraído do arquivo, para então ser realizada uma comparação de textual, caractere por caractere (RAVICHANDRAN; REKHA, 2013, tradução nossa).

3.1.2 Comparação byte a byte

Para Lima (2016), esse tipo de comparação é bastante empregado quando se trabalha com imagens, nas quais não é possível extrair textos tão facilmente, ou quando se deseja comparar não somente o texto, mas também todo o conteúdo de um documento, como cores, formas, distribuição textual, etc.

Uma imagem é uma lista de bytes, e a comparação de duas imagens pode ser feita comparando os arquivos byte por byte. Se todos forem iguais, não existem diferenças entre as duas imagens.

É possível fazer isso com funções que leem todos os bytes de uma imagem. Esses bytes são então carregados em um array e o algoritmo realiza a comparação

de posição por posição para validar a igualdade entre eles. Porém, é importante lembrar que esse método não funciona quando as imagens são iguais, porém de formatos diferentes, pois os bytes não serão os mesmos nesse caso (LIMA, 2016).

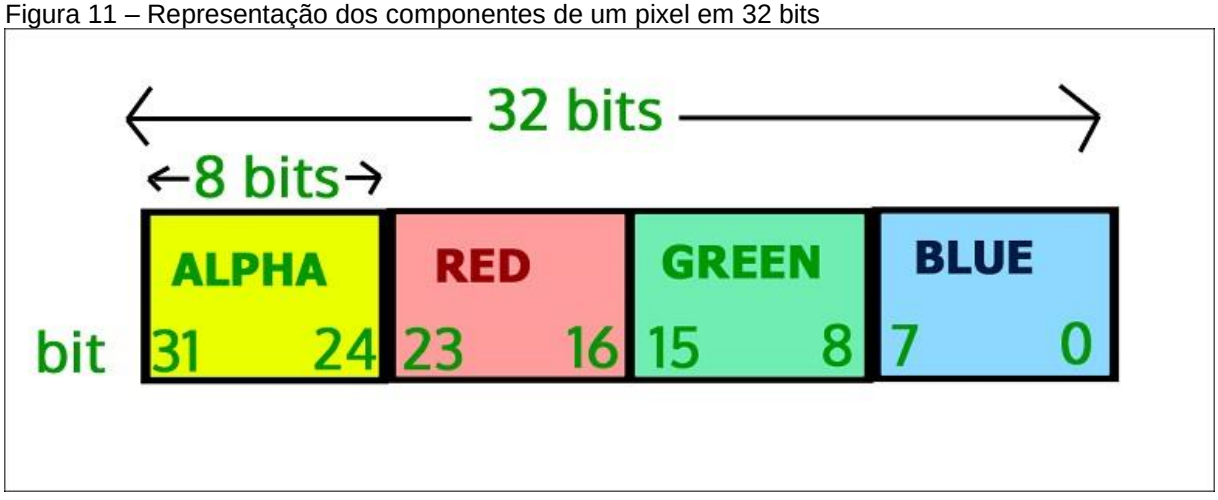
O método de comparação byte a byte se adequa ao que se espera atingir neste trabalho, pois além da comparação textual, é importante realizar a validação do relatório como um todo, incluindo as cores, layouts e posicionamento dos dados.

Neste método de comparação o processamento é feito através de imagens, então os arquivos PDF extraídos serão convertidos em imagem por uma biblioteca de manipulação de PDF. Mais abaixo serão discutidos algumas dessas ferramentas incluindo a que foi escolhida pra este projeto.

Então, como o a comparação se dará por imagens, são os pixels dessa imagem que serão analisados. Segundo Jain (2017), um pixel possui 4 componentes, Alpha, que representa a transparência e as três cores primárias aditivas, vermelho, verde e azul, ou em inglês, red, green e blue. Sendo daí a origem da sigla RGB. O valor de cada um destes componentes pode ficar entre 0 e 255. Se for 0, o componente está totalmente apagado, e se for 255 ele está totalmente aceso.

Considerando que 1 bit é um valor binário, ou seja, pode representar 0 ou 1. E que 1 byte possui 8 bits. Se aplicarmos a potência 2 elevado a 8 temos como resultado 256. Logo, 1 byte ou 8 bits pode guardar o valor de um dos quatro componentes. Para guardar todos os quatro componentes são necessários 4 bytes ou 32 bits (JAIN et al., 2017). A figura 11 representa em quais intervalos de bits estão alocados cada um dos componentes de um pixel.

Figura 11 – Representação dos componentes de um pixel em 32 bits



Fonte: Jain (2017).

Têm-se então que:

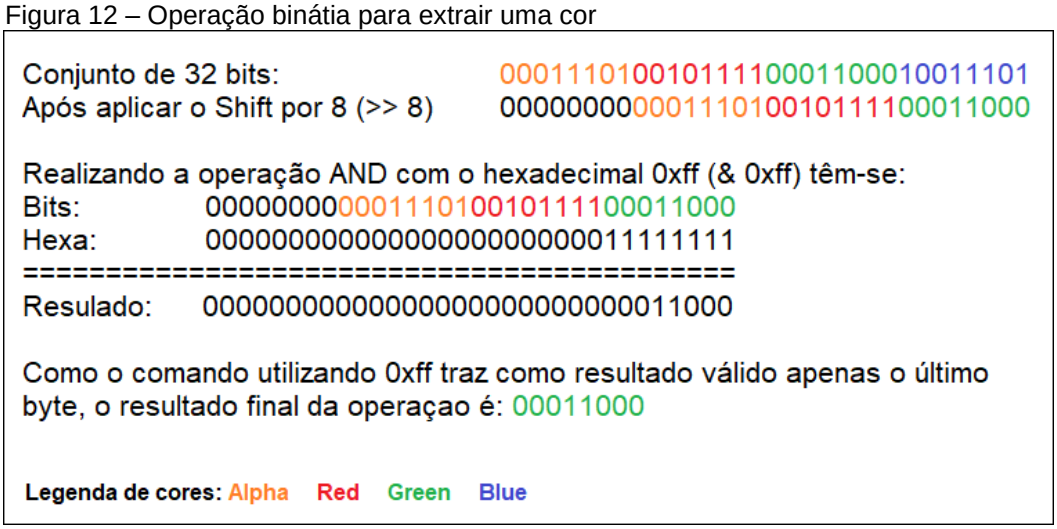
- Azul ocupa as posições de 0 a 7 bits;
- Verde ocupa as posições de 8 a 15 bits;
- Vermelho ocupa as posições de 16 a 23 bits;
- Transparência ocupa as posições de 24 a 31 bits;

Considerando a posição de cada componente do pixel dentro do conjunto de 32 bits, pode-se extrair apenas os 8 bits referentes a cada cor, e para fazer isso são utilizadas operações binárias (JAIN et al., 2017).

Primeiramente se aplica a lógica binária Shift para a direita (>>), que se refere a deslocar todos os bits para a direita um determinado número de vezes e preencher com bits zero os espaços que ficaram em branco. No exemplo será extraída a cor verde, então o número de Shifts necessários para colocar o verde na posição do último byte é 8, que como visto na figura 11, é onde começa o byte verde. Isso vai fazer com que os últimos bits desse inteiro não sejam mais a cor azul, agora a cor verde foi deslocada para essa posição. Após isso se aplica lógica binária com o operador AND (&) utilizando o valor hexadecimal 0xff, que representa o valor 255, ou seja, o componente todo aceso (JAIN et al., 2017).

A lógica END retorna o valor 1 sempre que ambos os bits em operação sejam 1. E ao ser utilizada com o hexadecimal 0xff, apenas o último byte, é levado em consideração, portanto, o resultado serão apenas os 8 bits referentes a cor verde. A figura 12 representa toda a operação.

Figura 12 – Operação binátia para extrair uma cor



Fonte: Do autor.

Dessa forma é possível extrair de cada pixel os valores individuais de cada componente e armazená-los para que a comparação possa ser feita individualmente pelas cores presentes dentro de cada pixel. Isso traz mais controle sobre as operações que podem ser executadas pois pode-se manipular individualmente cada cor de cada pixel.

3.2 FERRAMENTAS GRATUITAS PARA COMPARAÇÃO

A maioria das ferramentas gratuitas disponíveis não possui código aberto, e portanto não é possível integrar com o Selenium. Essas, normalmente funcionam através de websites, ou de programas locais instalados no computador. Algumas das mais comuns são:

- PDF24 Tools: Que é um website com ferramentas para trabalhar com arquivos PDF, através dele é possível executar tarefas como mesclar documentos, fazer buscas, extrair textos e informações, além de comparar arquivos.
- Draftable: Ferramenta hospedada em website, também gratuita para utilização, porém possui limitação para no máximo 300 páginas, ou arquivos de 10mb.
- Kiwi PDF Comparer: Ferramenta instalada localmente no computador, dispões de versão gratuita com algumas limitações, e possui recursos de comparação não apenas de PDF, como também de outros formatos de documentos digitais.

Partindo para as ferramentas Open source existem algumas opções disponíveis. Segundo Marx (2016, tradução nossa), o Apache PDFBox é uma delas. Tendo sua versão 2 lançada no início de 2016, e sendo mantido pela Fundação Apache, essa ferramenta possui código aberto, e permite a integração com linguagens como Java, Groovy, Kotlin, entre outros.

Através dela é possível realizar uma série de ações com arquivos PDF, como de criptografar, criptografar, criar documentos, extrair imagens ou texto, entre outros (APACHE PDFBOX, 2018, tradução nossa).

Outra ferramenta de código aberto é Haru Free PDF Library, que possui os mesmos recursos da ferramenta Apache, como extração de textos e imagens, concatenação ou divisão de arquivos, entre outros. Porém, essa biblioteca é compatível com compiladores da linguagem C, e por esse motivo fica inviável desenvolver uma integração com o Selenium.

Para o desenvolvimento da biblioteca de integração com Selenium, a ferramenta da Apache é a mais indicada. Pois roda na linguagem Java, e é mantida em código aberto pela Fundação Apache. Tendo assim, várias possibilidades para a realização do projeto.

3.3 FERRAMENTAS PAGAS PARA COMPARAÇÃO

Adobe Acrobat, é a ferramenta mais conhecida para manipulação de arquivos PDF. Desenvolvida pela própria criadora do PDF, ela conta com todas as possibilidades disponíveis para trabalhar com o formato.

Lançado em 1993, o Acrobat era o primeiro software capaz de ler o formato recém lançado pela Adobe, e ao longo dos anos foram lançadas várias versões que adicionavam cada vez mais funcionalidades a ferramenta.

O Adobe Acrobat possui uma versão gratuita que pode ser utilizada para leitura do de arquivos no formato PDF. Mas os recursos mais avançados que permitem a manipulação dos arquivos estão disponíveis somente nas versões proprietárias (ADOBE, 2018, tradução nossa).

Outra ferramenta proprietária disponível é o DiffPDF, que segundo a descrição do próprio desenvolvedor, é uma ferramenta de interface gráfica disponível para plataforma Windows (DIFFPDF, 2018, tradução nossa).

Segundo o fabricante, essa ferramenta permite realizar comparações entre as páginas do arquivo através métodos de comparação por texto e por gráfico. Destacando as divergências encontradas entre as páginas.

A ferramenta possui versão de avaliação por 20 dias, e após isso é necessário adquirir a licença de uso.

4 TRABALHOS CORRELATOS

Para a realização deste trabalho foram pesquisados alguns trabalhos semelhantes, adquirindo assim um maior embasamento do que já foi realizado na área.

4.1 PROPOSTA METODOLÓGICA DE AUTOMATIZAÇÃO DE TESTES DE SISTEMA UTILIZANDO SELENIUM WEBDRIVER

O propósito deste trabalho é demonstrar todo o processo de transformação de uma rotina de testes manual em uma aplicação web que são lentos e mais propensos a erros para uma versão automatizada do mesmo processo, esta, mais rápida e que permite ser executada sem necessitar de um recurso humano no processo. Esta modelagem será feita utilizando uma ferramenta de automação em navegadores, o Selenium WebDriver, aliada ao framework TestNG para construção dos casos. Os resultados de tal aplicação se mostraram bastante positivos e alinhados ao esperado, já que a também automação deste processo permite que seja possível integrar esta etapa da qualidade de software em um processo de integração contínua, utilizando o apoio de Jenkins CI e Apache Ant. Esta proposta se demonstrou um grande desafio, porém com resultados tão positivos, a mesma pode se considerar um sucesso na otimização dos testes de interface de usuário para softwares web.

4.2 ESTUDO DE AUTOMAÇÃO DE TESTES FUNCIONAIS E INTEGRAÇÃO CONTÍNUA PARA UM SISTEMA DE GESTÃO

Controlar e garantir a qualidade de um software é uma atividade extremamente complexa devido às variantes e problemas aos quais o processo de desenvolvimento está exposto, e cada vez mais as empresas são demandadas na criação de sistemas de qualidade em um curto espaço de tempo. A automação de testes tem sido uma alternativa para as empresas reduzirem custos e maximizarem a qualidade do produto entregue. Automação permite a execução de uma gama muito grande de testes em curto espaço de tempo, o que seria inviável se fossem executados de forma manual. O objetivo deste trabalho é propor um processo de automação com integração continua para a empresa alvo do estudo. Na primeira fase

deste trabalho foram levantadas as necessidades da empresa, definido um processo que contemple as atividades de automação e realizada a escolha de uma ferramenta para automatizar os testes. Na segunda etapa foi realizada a criação de um ambiente de protótipo para utilização dessa ferramenta, configurado processo de integração continua e realizada a integração entre as ferramentas Ranorex utilizada para automação e o Jenkins que auxilia nas atividades de integração contínua.

4.3 AUTOMAÇÃO DE TESTES FUNCIONAIS COM SELENIUM WEBDRIVER

Como a área de teste de software vem crescendo nos últimos anos, em especial a automação, está cada vez mais evidente no mercado a necessidade do teste de software, devido ao aumento da necessidade da qualidade para que o sistema tenha o mínimo possível de problemas em produção.

Este trabalho tem como objetivo apresentar as vantagens e desvantagens da implantação de testes automatizados utilizando como ferramenta de apoio o Selenium WebDriver. Além disso, busca-se comprovar que a utilização de testes automatizados garante maior qualidade do trabalho com menor esforço e otimizando o processo de teste da empresa alcançando assim maior produtividade.

Ao final do trabalho será apresentado a simulação do resultado alcançado com a implantação do Selenium WebDriver na empresa GS Group.

4.4 FERRAMENTAS DE SUPORTE A AUTOMAÇÃO DE TESTE FUNCIONAL: LEVANTAMENTO BIBLIOGRÁFICO

Técnicas e ferramentas de testes funcionais surgem no mercado para auxiliar na execução desses testes, de forma mais rápida e eficiente. Essas ferramentas, geralmente, simulam as ações de um usuário no sistema em teste e as transformam em scripts de teste que podem ser salvos e reexecutados sempre que necessário. Um problema enfrentado é o alto custo para a geração dos inúmeros casos de teste de entrada, necessários para se testar todas as funcionalidades do sistema. Embora os testes de software sejam grande aliados para a obtenção da qualidade, são poucos os documentos técnicos-científicos disponíveis acerca do tema. Geralmente, os livros de Engenharia de Software trazem uma breve descrição em uma seção ou capítulo de uma forma mais conceitual. Por essa razão, o presente

trabalho destina-se a um levantamento bibliográfico nas principais bases de pesquisa na área tecnológica, ACM, Science Direct e IEEE, e na revista Engenharia de Software Magazine, com o intuito de verificar as principais técnicas e ferramentas de automação de testes funcionais utilizadas, visando contribuir com a área de desenvolvimento de teste de software e fornecendo um apoio para fontes de consultas acadêmicas. A revisão proporcionou um levantamento de diversas estratégias e técnicas utilizadas atualmente, incluindo tanto as ferramentas de execução automatizada de testes funcionais, como o Abbot Framework, Marathon e Selenium IDE, quanto a geração de casos de testes para auxiliar no uso dessas ferramentas, como o AutoBlackTest, LBTest, e MoMuT::UML, dentre outras.

4.5 TESTES AUTOMATIZADOS NO PROCESSO DE DESENVOLVIMENTO DE SOFTWARES

O projeto consiste em um estudo sobre a automação de testes durante o desenvolvimento de um software. Para nos aprofundarmos nesse tema, realizamos um estudo sobre todos os principais conceitos e os diferentes tipos de testes.

Além disso, selecionamos algumas das mais utilizadas ferramentas do mercado de modo a entender um pouco mais seu funcionamento e suas aplicações na área de automação de testes.

Com o objetivo de ilustrar o funcionamento de um teste automatizado, exemplificamos, através de um estudo de caso, a construção e o funcionamento de um projeto automatizado.

5 TESTE AUTOMATIZADO PARA VALIDAÇÃO DE DADOS DE RELATÓRIOS APLICANDO CONCEITOS DE COMPARAÇÃO BYTE A BYTE EM ARQUIVOS PDF

O trabalho consiste em desenvolver um teste automatizado que irá simular um usuário logando num sistema de exemplo, inserindo dados e emitindo um relatório em PDF com esses dados. Após isso, o teste ficará responsável por comparar o arquivo PDF obtido com um modelo do que já era esperado. Será feita uma validação para garantir que o relatório foi de fato emitido, e em caso positivo, o serviço de comparação do PDF será inicializado.

Por fim, se houveram divergências entre os arquivos, os resultados devem ser apresentados destacando as diferenças entre o que era esperado e o que foi encontrado.

5.1 METODOLOGIA

O projeto teve início a partir da ideia de que validar páginas de relatórios faz parte do dia a dia de um tester, e seria de grande ajuda se esse processo pudesse ser automatizado assim como o sistema. Para por essa ideia em prática foi necessário levantar o material fonte através das pesquisas de levantamento bibliográfico, que tiveram início na parte de engenharia de software, especificamente na área de testes, englobando os principais modelos de testes existentes.

Tendo como base toda a teoria que define de que forma deve ser um teste, as pesquisas referentes a automação de testes se tornaram o foco. Foram vistas ferramentas para automação, bem como bibliotecas que auxiliam e agregam valor não só ao processo do teste, mas também a apresentação dos resultados encontrados pro usuário final.

O próximo passo foi estudar sobre modelo de documento eletrônico no formato PDF, que como foi visto na fundamentação teórica, é o principal formato empregado hoje no mundo para esse tipo de arquivo. O foco da pesquisa nesse ponto foi principalmente encontrar as formas existentes para fazer uma comparação de arquivos nesse formato.

De posse de todos os conhecimentos adquiridos com a pesquisa, os tópicos abaixo vão relatar sobre como foi desenvolvido o projeto, mostrando os

resultados obtidos através da simulações, e as conclusões que puderam ser tiradas deste projeto.

5.2 CRIAÇÃO DO SITE DE TESTE

O site criado para os testes (Apêndice A), consiste em uma página simples de uma biblioteca, onde é possível fazer login, inserir livros com autor e valor e guardar esses dados em uma tabela para posteriormente realizar a emissão do PDF.

O site foi criado com HTML, CSS e Javascript. Após digitar os dados desejados nos campos “Título”, “Autor” e “Valor”, o botão “Inserir” grava esses dados numa tabela HTML temporária, e limpa os campos do formulário.

O botão “Emitir” faz utilização da biblioteca jsPDF desenvolvida pela Agência Parallax que pega os dados presentes na tabela e monta um PDF (Apêndice B). A ferramenta também cria um cabeçalho e um rodapé pro relatório.

5.3 MONTAGEM DO AMBIENTE DE DESENVOLVIMENTO

O primeiro passo na montagem do ambiente é a instalação da IDE de desenvolvimento para criar o projeto. Como visto nos capítulos anteriores, a IDE escolhida é o IntelliJ, podendo ser utilizado para fins não lucrativos através da versão comunitária. Com a IDE já instalada, foi criado um projeto do tipo Maven com o nome “testerelatorio”. Como é um projeto do tipo Maven, as bibliotecas que foram utilizadas são instalas através do arquivo de configuração “pom.xml”.

Para realizar a instalação das bibliotecas foram inseridas as tags de dependências no arquivo pom.xml. A IDE então lê essas tags e faz a importação dos arquivos necessários automaticamente através dos caminhos informados nas dependências. Segue abaixo um modelo de como ficou a configuração.

Figura 13 – Dependências no arquivo pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>3.14.0</version>
    </dependency>
    <dependency>
        <groupId>com.aventstack</groupId>
        <artifactId>extentreports</artifactId>
        <version>3.1.5</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>6.14.3</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>commons-io</groupId>
        <artifactId>commons-io</artifactId>
        <version>2.6</version>
    </dependency>
    <dependency>
        <groupId>org.apache.pdfbox</groupId>
        <artifactId>pdfbox</artifactId>
        <version>2.0.15</version>
    </dependency>
</dependencies>
```

Fonte: Do autor.

Na imagem é possível ver que cada dependência tem um groupId, um artifactId e uma versão, essa informações estão disponíveis no repositório Maven. Basta buscar a biblioteca desejada, copiar as informações disponibilizadas no site e colar dentro da tag <dependencies>

Na imagem também é possível ver todas as bibliotecas necessárias para o projeto:

- Selenium para a criação do teste automatizado;
- TestNG para os métodos de teste;
- Commons-io para manipulação de arquivos físicos;
- PDFBox para manipulação dos arquivos PDF.
- ExtentReports para exibir os resultados do teste;

Após a importação das bibliotecas necessárias, o próximo passo é a criação da estrutura de pacotes e das classes do projeto. Neste caso foram criados três pacotes principais, “Infra”, “Page’ e “Test”.

O pacote Infra abriga todos os arquivos de configuração do teste e as bibliotecas de apoio ao teste. Já os pacotes Page e Test fazem parte do padrão

adotado no modelo de desenvolvimento chamado Page Objects Model (POM), no qual o pacote Page abriga as classes responsáveis pela localização e interação com os elementos da tela, e o pacote Test abriga apenas os casos de teste (SEKA, 2017). Isso traz benefícios pois separa de um lado toda a parte mais complexa do código como a localização e interação com os elementos, e do outro os casos de teste, que ficam separados numa classe própria deixando mais claro as ações e validações que serão feitas.

Dentro do pacote Infra foram criados mais três pacotes, “extentreport”, “testbase” e “utilitarios”.

Extentrport guarda os arquivos de configuração da biblioteca responsável por gerar a página com os resultados do teste. TestBase contém os arquivos de configuração do teste. E utilitários contém os arquivos que dão suporte ao teste, neste caso, toda a lógica criada para realizar a comparação dos arquivos PDF.

Dentro do projeto também foram criadas três pastas, Comparação, Config, Output. A pasta comparação abriga o arquivo modelo do PDF que está sendo testado, é com ele que o relatório emitido será comparado. Além disso, abriga também uma pasta com os arquivos temporários gerados pelo teste, como as imagens geradas através dos arquivos PDF. Esses arquivos temporários são apagados a cada nova execução do teste, para não interferir com testes anteriores, e não gerar um consumo de espaço desnecessário no disco.

A pasta config guarda os arquivos físicos utilizados no teste. Neste caso o driver do navegador Chrome utilizado pelo Selenium.

E a pasta Output que guarda os resultados do teste, como a página gerada pelo ExtentReports, e as imagens contendo as diferenças encontradas entre os dois arquivos PDF.

O apêndice C mostra a estrutura do projeto após a criação de todas as pastas necessárias.

5.4 CRIAÇÃO DOS ARQUIVOS DE CONFIGURAÇÃO

O primeiro e mais importante arquivo de configuração é chamado TestBase. Todas as classes de teste irão estender dessa classe principal. É nela que será definido toda a forma de execução do teste através de alguns métodos. Cada

método recebe uma anotação disponibilizada pela biblioteca TestNG, isso permite que seja definido uma ordem de execução para os métodos.

O primeiro método recebeu a anotação “@BeforeSuite”, que significa que será executado antes de cada suíte de testes. Esse é o método responsável por criar o arquivo HTML que por fim será populado com os resultados do teste.

Figura 14 – Método inicializaReporter

```
@BeforeSuite
public void inicializaReporter() {
    extent = ExtentManager.createInstance( fileName: "./Output/Report/ResultadoTeste.html");
}
```

Fonte: Do autor.

O segundo método recebeu a anotação “@BeforeTest”, e portanto irá ser executado uma vez antes de cada instancia de teste. Este método da início aos objetos necessários pro Selenium operar. São eles os objetos Driver e Wait.

Figura 15 – Método inicializaDriver

```
@BeforeTest
public static void inicializaDriver() throws MalformedURLException {
    InicializaDrivers.inicializaDriver();
    wait = new WebDriverWait(driver,  timeOutInSeconds: 30);

    driver.get(baseURL);
    Assert.assertTrue(driver.getTitle().equals("Login"));
}
```

Fonte: Do autor.

O objeto driver é inicializado em outra classe para melhor separação do código. Ele é responsável por todas as ações que o Selenium irá tomar sobre o navegador. É através dele também que é possível setar algumas configurações para o navegador, neste caso o chromedriver.

Após iniciar o driver, o objeto wait, responsável por todas as esperas entre as interações com os elementos também é inicializado. E após ele, o driver já pode receber o endereço do site de teste, e uma validação que irá garantir que o título do site seja “Login”.

A figura 16 mais abaixo mostra a inicialização do objetivo driver. Primeiro é informado o local do executável chromedriver.exe. Após isso são iniciados algumas configurações básicas, como iniciar maximizado, e ignorar alertas. Porém, duas

configurações são importantes nesse projeto. O Selenium não é capaz de manipular janela do Windows, e isso é necessário quando um download é realizado, pois o sistema solicita um local para salvar o arquivo. Para contornar esse problema foi utilizado a configuração "profile.default_content_settings.popups" que inicia o download automaticamente sem a abertura de popups do Windows. E outra configuração "download.default_directory" que define o local padrão para download de arquivos. Dessa forma, ao emitir o relatório no site, o mesmo será automaticamente baixado no diretório temporário do projeto, sem nenhuma interação necessária com janelas do Windows. Por fim, o driver é instanciado recebendo essas e as outras configurações por parâmetro.

Figura 16 – Preferências do driver

```
public static void inicializaDriver() throws MalformedURLException{
    try {
        System.setProperty("webdriver.chrome.driver",
                           "./Config/Drivers/chromedriver.exe");
        DesiredCapabilities dcChrome = DesiredCapabilities.chrome();
        ChromeOptions options = new ChromeOptions();
        HashMap<String, Object> chromePrefs = new HashMap<>();
        chromePrefs.put("profile.default_content_settings.popups", 0);
        chromePrefs.put("download.default_directory", "C:\\Users\\cassi\\IdeaProjects\\testerelatorio\\Comparacao\\ArquivosTemporarios");
        options.setExperimentalOption( name: "prefs", chromePrefs);
        options.addArguments("start-maximized");
        dcChrome.setCapability(CapabilityType.UNEXPECTED_ALERT_BEHAVIOUR, UnexpectedAlertBehaviour.IGNORE);
        dcChrome.setCapability(ChromeOptions.CAPABILITY, options);
        driver = new ChromeDriver(dcChrome);
    } catch (WebDriverException e) {
        System.out.println(e.getMessage());
    }
}
```

Fonte: Do autor.

O terceiro e quarto métodos receberam respectivamente a anotação "@BeforeClass" e "@BeforeMethod", eles serão executados uma vez antes de cada classe e método. Estes métodos são responsáveis por dizer para a ferramenta que grava os resultados do teste, o nome da classe e do método em execução atualmente.

Figura 17 – Métodos da ferramenta ExtentReport

```
@BeforeClass
public synchronized void reportCriaClasses() {
        ExtentTest parent = extent.createTest(getClass().getAnnotation(TestNomeEAutor.class).getTesteNome());
        parentTest.set(parent);
        parent.assignAuthor(getClass().getAnnotation(TestNomeEAutor.class).getAutor());
        parent.assignCategory();
}

@BeforeMethod
public synchronized void reportCriaMetodos(Method method) {
    if(!method.isAnnotationPresent(IgnoreReport.class)){
        ExtentTest child = parentTest.get().createNode(method.getName());
        test.set(child);
    }
}
```

Fonte: Do autor

Uma observação a ser feita são as anotações personalizadas “TestNomeEAutor” e “IgnoreReport”. Estas foram classes criadas para permitir que seja informado por anotações o nome do teste, e o autor. Essas informações serão vinculadas no resultado de teste (Apêndice D).

E a anotação IgnoreReport permite nem todos os métodos estejam visíveis nos resultados do teste. Alguns dos métodos presentes na classe do caso de teste são apenas configurações e etapas iniciais, não fazem parte do processo de validação, então não é interessante que tais métodos estejam presentes nos resultados.

O quinto método anotado com “@AfterMethod” é executado uma vez após cada método do teste. Ele verifica se o método em execução teve sucesso, foi pulado ou resultou em falha, e grava a informação na página HTML dos resultados.

Através da figura 18 é possível ver a implementação e as ações tomadas para cada resultado. Se houve falha, o sistema pode anexar imagens previamente guardadas, ou tirar capturas de tela no momento da falha para visualizar o erro. Nesse caso estão sendo inseridas duas imagens contendo as diferenças entre o relatório de controle e o relatório obtido.

Se o teste obteve sucesso ou foi pulado, o sistema apenas grava a informação e por fim vincula tudo no HTML com os resultados.

Figura 18 – Método reportSetaResultados

```
@AfterMethod
public synchronized void reportSetaResultados(ITestResult result, Method method) throws IOException {
    if(!method.isAnnotationPresent(IgnoreReport.class)){
        if (result.getStatus() == ITestResult.FAILURE) {
            test.get().addScreenCaptureFromPath("C:\\Users\\cassi\\IdeaProjects\\testerelatorio\\" +
                    "utput\\ResultadosComparacao\\imgControleDiff.png");
            test.get().addScreenCaptureFromPath("C:\\Users\\cassi\\IdeaProjects\\testerelatorio\\" +
                    "Output\\ResultadosComparacao\\imgTesteDiff.png");
            test.get().fail(result.getThrowable());
        }
        else if (result.getStatus() == ITestResult.SKIP)
            test.get().skip(result.getThrowable());
        else
            test.get().pass("Test passed");

        extent.flush();
    }
}
```

Fonte: Do autor

No penúltimo método a anotação “@AfterTest” indica que a execução se dará após cada instância de teste. Neste método será fechado o navegador através do comando driver.quit();

O último método, marcado com a anotação “@AfterSuite” irá executar uma vez após a suíte de testes. E ele será responsável por abrir a página HTML contendo os resultados obtidos com a suíte que acabou de ser executada.

Essa página HTML pode ser configurada através do arquivo ExtentManager como foi citado anteriormente. Através dela é possível definir temas, posição dos gráficos, codificação dos textos, formatação de data e hora, e também estilizar a página por comandos CSS.

Neste projeto foram utilizadas apenas as configurações básicas recomendadas pelos criadores da ferramenta, porém foi inserido através de CSS o logotipo da UNESC no canto superior esquerdo da página.

5.5 DESENVOLVIMENTO DOS ARQUIVOS DO TESTE

Conforme dito anteriormente, o modelo de desenvolvimento utilizado neste teste é o Page Objects Model em conjunto com o conceito Page Factory.

No modelo POM os localizadores e funções de interação com elementos ficam numa classe chamada Page. Enquanto as funções do caso de teste propriamente dito fica em outra classe. Essa classe de teste apenas chama os métodos da classe Page. Isso traz benefícios na hora de dar manutenção no teste pois reduz a verbosidade, mantendo as funções mais complexas de manipulação dos elementos fora da classe de teste.

Já o Page Factory é um conceito complementar ao POM. Ele permite separar previamente os localizadores dos elementos e inicializar todos de uma vez no método construtor da classe. Isso também ajuda a reduzir a verbosidade do código. E facilita a reutilização dos elementos posteriormente. O trecho de código abaixo simula uma ação de fazer login em um site sem a utilização do conceito Page Factory.

Figura 19 – Modelo código sem Page Factory

```
public void fazLogin() {
    driver.findElement(By.cssSelector("input[id='usuario']")).sendKeys( ...charSequences: "adm");
    driver.findElement(By.cssSelector("input[id='senha']")).sendKeys( ...charSequences: "admin");
    driver.findElement(By.cssSelector("button[id='botaoLogin']")).click();
    System.out.println("Login realizado");
}
```

Fonte: Do autor

Agora veja como fica o mesmo código quando se aplica o conceito Page Factory.

Figura 20 – Modelo código com Page Factory

```
@FindBy(how=How.CSS,using="input[id='usuario']")
private WebElement usuario;

@FindBy(how=How.CSS,using="input[id='senha']")
private WebElement senha;

@FindBy(how=How.CSS,using="input[id='botaoLogin']")
private WebElement botaoLogin;


public void fazLogin() {
    usuario.sendKeys( ...charSequences: "adm");
    senha.sendKeys( ...charSequences: "admin");
    botaoLogin.click();
    System.out.println("Login realizado");
}
```

Fonte: Do autor

Com os localizadores dos elementos separados das classes, é possível manter o código mais limpo e legível, além de eliminar a necessidade de chamar o driver para localizar os elementos cada vez que for interagir com eles. Dessa forma os elementos sempre estarão inicializados quando for necessário utilizá-los.

5.5.1 Classe TesteRelatorio

Essa é a classe principal, é através dela que o teste é executado. Ela pode ser vista por completo no apêndice E.

Os métodos presentes nessa classe estão utilizando as anotações @Test disponibilizadas pela biblioteca TestNG. Essas anotações declaram que o método em

questão é um método de teste. Isso faz com que a IDE saiba que a classe contendo esses métodos é uma classe de execução, e será por ela que o teste irá executar.

Em conjunto com a anotação @Test foi inserido o marcador (dependsOnMethods). Essa marcação cria uma dependência entre os métodos. O método marcado com essa opção só será executado após a conclusão do método ao qual ele é dependente. Caso o haja algum erro, o teste irá pular os métodos subsequentes e finalizar o teste com falha.

O primeiro dos métodos dessa classe faz a inicialização da classe TesteRelatorioPage, que contém todos os localizadores dos elementos juntamente com suas funções.

Após isso o método fazLogin() chama o método de login da classe TesteRelatorioPage, e o teste preenche os campos de usuário e senha e faz login na página da biblioteca.

O método seguinte é chamado insereLivros(). Ele passa por parâmetro um título, autor e valor, e a classe Page faz a inserção dos livros.

Os últimos dois métodos são os responsáveis por fazer a comparação do PDF. O primeiro deles vai emitir o PDF, e o segundo vai chamar as classes utilitárias criadas para fazer a comparação dos arquivos.

Se a comparação não mostrar diferenças, o teste será finalizado e a página com os resultados vai trazer todos os métodos dessa classe marcados como Ok. Caso contrário, as divergências entre os arquivos serão anexadas a página de resultados, e o método responsável por fazer a comparação irá ficar marcado com a falha.

5.5.2 Classe TesteRelatorioPage

Essa classe é chamada pela classe de teste principal quando é necessário fazer a manipulação dos elementos da página. Como esta classe foi desenvolvida utilizando os conceitos de POM com Page Factory, o primeiro passo é inicializar os elementos pelos seus localizadores. Isso é feito no método construtor da classe, como pode ser visto na figura 21 abaixo.

Figura 21 – Método construtor da classe Page

Fonte: Do autor

As classes Pdfbox e Pixels são instanciadas, elas serão usadas posteriormente para fazer a comparação dos arquivos.

A linha “PageFactory.initElements” é responsável por inicializar todos os elementos da página utilizando os localizadores que já haviam sido previamente declarados como foi visto anteriormente. A partir de agora, sempre que for necessário manipular um elemento, não é mais necessário chamar o driver do Selenium para encontrar o elemento. Basta chamar o elemento pelo nome que lhe foi atribuído, e tomar alguma ação.

Também pode ser visto na imagem uma chamada pro método limpaArquivosTemporarios. Essa chamada foi colocada ali para que a cada nova execução do teste, arquivos temporários do teste passado sejam removidos para prevenir que haja sobreposição de arquivos, bem como otimizar o espaço em disco eliminando arquivos que são usados apenas uma vez no escopo no teste.

Os métodos exibidos na figura 22 são todos os métodos de manipulação de elementos da página que foram chamados pela classe de testes.

Figura 22 – Métodos de manipulação da classe Page

```
public void fazLogin() {
    elUsuario.sendKeys( ...charSequences: "cassio");
    elSenha.sendKeys( ...charSequences: "123456");
    elLogin.click();
    Assert.assertTrue(driver.getTitle().equals("Livraria"));
}

public void insereLivro(String titulo, String autor, String valor) {
    eltitulo.sendKeys(titulo);
    elAutor.sendKeys(autor);
    elValor.sendKeys(valor);
    elInserir.click();
    Assert.assertEquals(colTabela.get(0).getText(), titulo);
    Assert.assertEquals(colTabela.get(1).getText(), autor);
    Assert.assertEquals(colTabela.get(2).getText(), s1: "R$: " + valor);
}

public void emitePDF() throws InterruptedException {
    elEmitir.click();
    aguardaDownloadEPegaNome();
}
```

Fonte: Do autor

A função faz login pega os elementos referentes ao campo de usuário e senha, que neste momento já estão inicializados, e envia uma string com os caracteres referentes aos dados necessários pro login. Depois, é tomada uma ação de clique no botão Login.

A linha Assert faz parte da biblioteca TestNG, ela traz métodos de validações para vários tipos de situações. No caso, foi usado o método assertTrue, que como o nome diz, verifica se algo é verdadeiro. Neste caso, verifica se após logar, o título da página mudou pra “Livraria”. Garantindo assim que o sistema de fato logou.

Na próxima função são inseridos os livros utilizando os dados que foram passados por parâmetros pela classe de teste. Após digitar título, autor e valor, o botão inserir recebe um clique e novamente os métodos Assert são utilizados. Dessa vez com o assertEquals, que verifica se duas condições estão iguais. Nesse caso, verifica se as colunas da tabela HTML estão sendo povoadas com os textos que foram passados por parâmetro pela classe de teste.

A última função que manipula elementos da página é de emitir o PDF. Ela envia a ação de clique pro elemento referente ao botão emitir, e chama o método que irá aguardar até que o arquivo tenha sido baixado, além de pegar o nome do arquivo gerado. Este é um ponto importante, para o teste prosseguir e realizar a comparação

dos PDF, foi necessário criar uma espera até que o download estivesse finalizado, do contrário, o teste retornaria erro por não ter encontrado o arquivo.

Essa espera consiste em ficar monitorando um diretório dentro da pasta temporária do projeto, para verificar se existem arquivos. Se existir, a função deve ficar monitorando até que o nome do arquivo existente não termine com a extensão “crdownload” que caracteriza um download em andamento no navegador Google Chrome. Quando isso ocorrer, o nome do arquivo encontrado deve ser armazenado em uma string, e uma variável booleana que representa o download finalizado deverá ser setada como verdadeira para permitir a saída do laço de repetição. Por fim o método retorna a string com o nome do arquivo encontrado.

Figura 23 – Método para aguardar o download do arquivo PDF

```
private String aguardaDownloadEPegaNome() throws InterruptedException {
    Thread.sleep( millis: 500);
    File path = new File( pathname: "./Comparacao/ArquivosTemporarios");
    File[] listaArquivos = path.listFiles();
    Boolean dlTerminou = false;

    if(listaArquivos.length > 0) {
        while (!dlTerminou) {
            for (int i = 0; i < listaArquivos.length; i++) {
                if (!listaArquivos[i].getName().endsWith(".crdownload")) {
                    dlTerminou = true;
                    nomeArquivo = listaArquivos[i].getName();
                }
            }
        }
    }
    return nomeArquivo;
}
```

Fonte: Do autor

Tendo a garantia de que o arquivo PDF já se encontra no disco, o teste pode prosseguir, e o próximo passo é enviar o documento para a classe de manipulação do PDF e para a classe de comparação. Isso é feito através do método comparaPDF.

Primeiramente a classe contendo a biblioteca do Pdfbox é chamada para gerar uma imagem dos documentos PDF passados por parâmetro. Um deles é o arquivo de controle que contém o resultado que era esperado. Esse documento já deve estar previamente salvo no diretório de controle. O outro é o documento que foi

baixado. A função que gera imagem recebe o diretório onde o documento foi salvo, e o nome do arquivo que havia sido gravado no método anterior.

Após isso é feita uma condição que chama a classe Pixels, responsável por manipular as imagens e encontrar possíveis diferenças. Caso essa classe retorne um valor verdadeiro, significa que o teste encontrou diferenças entre os arquivos. Então a validação Assert.fail é chamada para registrar a falha e finalizar o teste.

Figura 24 – Método comparaPDF

```
public void comparaPDF() throws IOException {
    pdfbox.geraImagemPdfControle( diretorioControle: "./Comparacao/Modelo", nomeControle: "modelo.pdf");
    pdfbox.geraImagemPdfRelatorio( diretorioPDF: "./Comparacao/ArquivosTemporarios", nomeArquivo);
    if (pixels.comparaImagens()){
        Assert.fail( s: "Relatórios contém diferenças");
    }
}
```

Fonte: Do autor

5.6 DESENVOLVIMENTO DOS ARQUIVOS DE COMPARAÇÃO DE PDF

Para permitir a comparação de documentos em PDF é necessário ter uma ferramenta que faça a manipulação desse tipo de arquivo. Conforme visto na fundamentação teórica, a ferramenta escolhida foi o PDFBox da Apache. Além de ser gratuito e oferecer uma grande quantidade de métodos, um dos mais importes é a conversão do arquivo em PDF para uma imagem. Seria possível realizar a comparação extraindo apenas os textos do PDF, mas o objetivo desse trabalho vai além de validar apenas os dados escritos. O intuito é validar o relatório como um todo, garantindo não somente os valores textuais, mas também todo o layout e a estrutura do documento. Isso pode ser feito através da manipulação de imagens, com algoritmos que comparam as imagens pixel por pixel em busca de diferenças. Sendo assim, o primeiro passo após o teste ter baixado documento do site de exemplo, é gerar imagens dos arquivos que serão analisados, para posteriormente os métodos de comparação da classe Pixels varrerem os documentos.

5.6.1 Classe Pdfbox

Na classe Pdfbox existem dois métodos que fazem isso. Um deles gera uma imagem do documento de controle, que contém os valores esperados. E o outro

gera uma imagem a partir do documento baixado pelo teste. Esse processo pode ser visto na figura 25.

Figura 25 – Método geraImagemPdfRelatorio

```
public void geraImagemPdfRelatorio(String diretorioPDF, String nomePDF) throws IOException{
    pdfRelatorio = new File( pathname: diretorioPDF + "/" + nomePDF);
    docRelatorio = PDDocument.load(pdfRelatorio);
    renderer = new PDFRenderer(docRelatorio);
    imgRelatorio = renderer.renderImageWithDPI( pageIndex: 0, dpi: 300);
    ImageIO.write(imgRelatorio, formatName: "PNG", new File( pathname: diretorioPDF + "/imgRelatorio.png"));
    System.out.println("Imagem relatorio gerada com sucesso!");
    docRelatorio.close();

}
```

Fonte: Do autor

É criado um objeto do tipo File, que recebe o caminho até o documento no formato PDF. Esse arquivo File por sua vez é carregado no objeto docRelatorio. Este é um objeto do tipo PDDocument pertencente a biblioteca PDFBox.

Também pertencente ao PDFBox temos o objeto PDFRenderer, que através do método “renderImageWithDPI” pode fazer a renderização do documento em uma imagem com a resolução desejada. Após isso a imagem é salva no disco e o objeto PDDocument é fechado pois não será mais utilizado.

Conforme pode ser visto na figura 25, o método que faz a renderização do PDF para imagem recebe através de parâmetros o número da página que será renderizada e a resolução desejada.

Como este projeto visa prototipar um teste que seja capaz de encontrar diferenças em arquivos PDF, as simulações foram feitas utilizando apenas documentos com uma página. Melhorias futuras podem ser aplicadas para realizar a comparação em documentos com várias páginas, especialmente considerando que o PDFBox já traz métodos que permitem a manipulação de várias páginas.

Após a renderização da imagem, ela é salva no disco, e o objetivo do tipo PDDocument já pode ser fechado pois não será mais utilizado.

A partir daqui é a vez dos métodos da classe Pixels realizarem a comparação através da imagens criadas.

5.6.2 Classe Pixels

Para manipular imagens o Java já traz incluído em sua JDK a biblioteca ImageIO, que permite fazer leitura, alteração e gravação da dados gráficos. E para começar são criados os objetos públicos utilizados ao longo do processo. São eles:

- BufferedImage imgControle: Representação em memória da imagem de controle;
- BufferedImage imgTeste = Representação em memória da imagem do teste;
- BufferedImage marcaDaguaControle = Marca d'água para a imagem de controle;
- BufferedImage marcaDaguaTeste = Marca d'água para a imagem de teste;
- Boolean resultado = Variável que será retornada pelo método de comparação.

O primeiro passo é ler na memória as imagens de controle e do teste. Para isso são chamados os métodos leImagemControle e leImagemTeste que utilizam o método ImageIO.read() passando por parâmetro objetos File contendo os diretórios das imagens. Antes da comparação iniciar é feita uma validação para verificar se ambas as imagens possuem a mesma resolução. Caso elas possuam tamanhos diferentes, o teste já é finalizado com uma falha.

De posse das duas imagens carregadas em memória, e considerando que ambas tenham o mesmo tamanho, o teste pode prosseguir iniciando os laços de repetição que vão varrer as imagens. A figura 26 mostra o código.

Figura 26 – Laço de repetição que faz a comparação

```
if (widthControle == widthTeste || heightControle == heightTeste) {
    for (int w = 0; w < widthControle; w++) {
        for (int h = 0; h < heightControle; h++) {
            int pixelControle = imgControle.getRGB(w, h);
            int pixelTeste = imgTeste.getRGB(w, h);

            int brilhoControle = pixelControle / 3;
            int brilhoTeste = pixelTeste / 3;

            if (pixelControle != pixelTeste) {
                if (brilhoControle < brilhoTeste){
                    pixelControle = corVermelho;
                    imgControle.setRGB(w, h, pixelControle);
                }else {
                    pixelTeste = corVermelho;
                    imgTeste.setRGB(w, h, pixelTeste);
                }
                resultado = true;
            }
        }
    }
```

Fonte: Do autor

Neste caso um laço fica dentro do outro, o primeiro vai varrer a largura da imagem armazenando a posição na variável “w”, e o segundo vai varrer a altura da imagem armazenando a posição na variável “h”. Após isso a função getRGB(w, h) armazena cada pixel da posição atual num inteiro referente ao pixel de controle e ao pixel de teste.

Os inteiros brilhoControle e brilhoTeste recebem uma média da somatória dos RGB. Que como visto na fundamentação teórica pode ir de 0 a 255. Esses inteiros serão utilizados posteriormente na comparação para poder criar a separação entre os destaques aplicados nas imagem de controle e de teste.

O próximo trecho de código é o if principal da comparação. Como visto na fundamentação teórica, após capturar o RGB através do método getRGB() é feita a separação das cores individuais do pixel uma em cada variável inteira contendo apenas seus 8 bits. Isso permite um controle maior sobre as cores que estão sendo manipuladas. Porém, como este projeto não se propõe a fazer manipulações gráficas avançadas, o intuito é apenas comparar os pixels e encontrar diferenças, não é necessário aplicar toda a lógica de separação dos componentes de um pixel. Por isso, a comparação ocorre com todo o conjunto de 32 bits retornados pela função.

Logo, se o pixelControle for diferente do pixelTeste, a lógica para destacar essa diferenças deve iniciar. A lógica desenvolvida envolve as médias armazenadas nas variáveis brilhoControle e brilhoTeste. Se houve uma diferença entre os pixels, então o código deve entrar em outra condição para avaliar se o brilhoControle é menor que o brilhoTeste. Se este for o caso, o pixelControle recebe a cor vermelha, e é setado na sua imagem. Se o brilhoControle for maior que o brilhoTeste é o pixelTeste que recebe a cor vermelha e é setado na sua imagem.

Após aplicar o destaque no pixel, o laço de repetição continua até ter varrido toda a imagem e encontrado todas as diferenças. A variável booleana que será retornada pela função já recebe o valor true desde a primeira diferença encontrada.

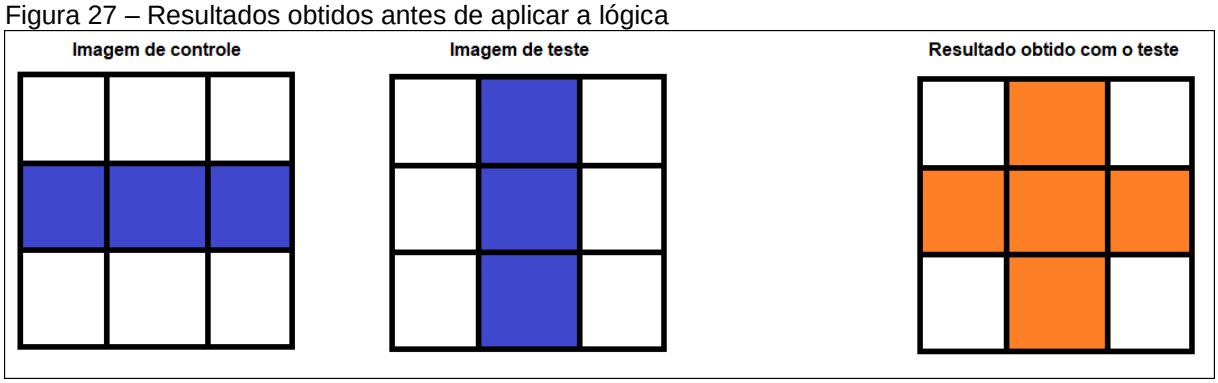
A partir daqui, é feita apenas uma validação para ver a variável boolenana está verdadeira, e com isso se iniciam as lógicas para gravar as imagens contendo as diferenças para que o resultado do teste possa exibi-las no final. Essas imagens também terão marcas d’água adicionadas para ser possível diferenciar qual imagem era o resultado esperado e qual imagem é o resultado encontrado.

5.7 RESULTADOS OBTIDOS

Para resolver o problema apontado pelo projeto foram levantados materiais de várias áreas do conhecimento em computação, como engenharia e teste de software, testes automatizados, documentos digitais, processamento de dados gráficos. Além de várias outras ferramentas auxiliares. Tudo isso deu suporte para o desenvolvimento do teste.

Existe pouca quantidade de material de qualidade falando especificamente sobre validação de dados em relatórios, o que fez com que o projeto durante sua fase inicial tivesse um futuro um pouco incerto quanto a possibilidade de buscar os resultados de maneira satisfatória. Já existiam ferramentas que faziam a diferenciação de imagens disponíveis, mas nenhuma das encontradas era aplicável no contexto de teste automatizado apresentado pelo projeto. Além disso, as ferramentas existentes apresentavam o mesmo problema encontrado durante o desenvolvimento deste projeto. Era possível fazer o teste encontrar diferenças nos arquivos. Porém os resultados obtidos mesclavam as diferenças das imagens, não sendo possível destacar claramente quais eram as diferenças. Veja a figura 27.

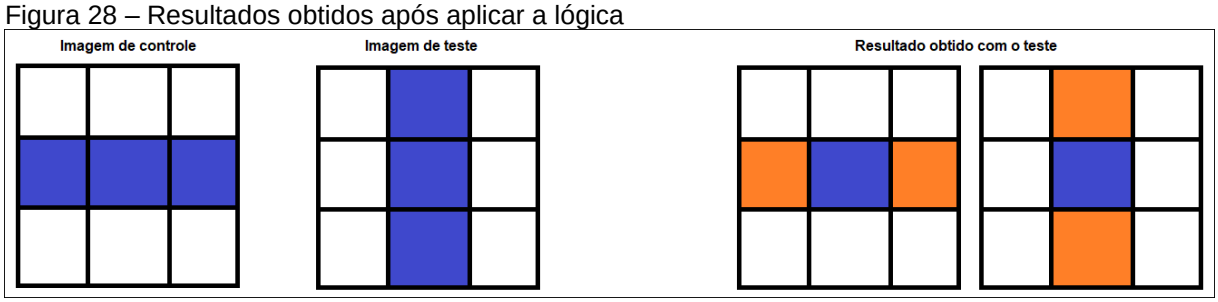
Figura 27 – Resultados obtidos antes de aplicar a lógica



Fonte: Do autor

Esse resultado servia para apontar que existem diferenças nos arquivos, mas não era satisfatório, o desejado é que o teste aponte claramente as diferenças encontradas destacando o que era esperado e o que foi encontrado. Para isso foi desenvolvida a lógica de comparação da média dos valores RGB apresentada no tópico anterior. Com isso os resultados começaram a ficar bem melhores, já sendo possível apresentar no final do teste duas imagens, uma com o resultado esperado e uma com o resultado encontrado, conforme pode ser visto na figura 28.

Figura 28 – Resultados obtidos após aplicar a lógica



Fonte: Do autor

As ferramentas utilizadas para auxiliar no teste como TestNG e ExtentReports se tornaram grandes aliados, pois com o TestNG é possível estabelecer de forma clara rotinas que são executadas antes ou depois de cada método, e isso foi fundamental para o projeto.

A forma de apresentar o resultado encontrado pelo teste ao usuário também não poder ser simplesmente um alerta de erro. Era necessário um documento mais completo que mostrasse tudo que aconteceu durante o teste, nesse ponto, o ExtentReports trouxe os resultados que eram esperados, os testes agora traziam resultados claros sobre os problemas encontrados, permitindo ao usuário saber exatamente em qual parte do teste houveram problemas.

Durante a prototipagem foram feitos testes com imagens simples contendo apenas 6 pixels como os modelos acima, quando os resultados obtidos foram suficientes, os testes começaram a ser feitos utilizando imagens reais dos documentos PDF gerados através do sistema de exemplo. E os resultados encontrados cumpriram aquilo que o objetivo geral do trabalho propunha.

6 CONCLUSÃO

Desenvolver um teste automatizado requer um tempo grande de desenvolvimento durante a criação do projeto, e o retorno só começa a vir mais tarde, quando o teste já pode ser aplicado e replicado em outros projetos. Isso foi levantado durante a fundamentação teórica e pode ser comprovado com a criação deste trabalho.

Criar um teste automatizado capaz de validar dados em relatórios emitidos em formato PDF exigiu a busca de conhecimentos variados e uma grande quantidade de tempo até fosse possível enxergar resultados satisfatórios.

As ferramentas auxiliares pesquisadas durante o projeto facilitaram a tarefa e agregaram valor aos resultados. A IDE de desenvolvimento IntelliJ se mostrou leve e inteligente, trazendo sugestões pertinentes de melhoria e otimização do código, além de rodar sem maiores problemas de travamentos mesmo quando os testes foram feitos em máquinas com menor capacidade de processamento e memória. Além disso a IDE ainda traz suporte nativo a várias ferramentas de teste, incluindo a biblioteca TestNG utilizada nesse projeto. Todos esses recursos podem ser habilitados durante a instalação da IDE.

A biblioteca TestNG juntamente com o ExtentReports permitiram controlar as rotinas que ocorrem antes, durante e depois de cada teste, como os métodos de configuração do ambiente, de validação do teste e de gravação dos resultados.

O PDFBox traz métodos para gerar imagens a partir de um arquivo PDF, e um ponto importante é a resolução personalizável dessas imagens. Foi importante ter controle sobre isso pois arquivos pequenos não permitem um resultado satisfatório na hora de ler as diferenças encontradas no teste, e arquivos muito grandes geram lentidão na hora que o método de comparação começa a varrer os pixels. Além da possibilidade de gerar imagens, essa biblioteca ainda traz recursos para manipular documentos com várias páginas, isso não foi abordado nesse projeto, mas é um ponto de melhoria para trabalhos futuros.

Uma vez pronto o teste automatizado ficou claro como ele pode auxiliar no dia a dia de um tester. Ter uma ferramenta que pode conferir não somente o sistema como também os relatórios gerados por ele permite ao tester garantir que a versão do sistema está pronta para liberação com um maior grau de segurança do que se comparado ao processo manual de inspecionar visualmente vários relatórios. Uma

tarefa como essa cansa rapidamente e é aí que os erros começam a passar despercebidos. Já o teste automatizado não tem o mesmo problema, e pode ser executado quantas vezes forem necessárias.

Alguns problemas foram encontrados durante o projeto, como a falta de clareza na diferenciação de imagem quando utilizada apenas a lógica de comparar pixel a pixel sem um tratamento para diferenciar qual pixel pertence a qual imagem na hora de gravar os resultados. Mas com a pesquisa de base realizada para a criação do projeto, especialmente na parte de processamento de dados gráficos, foi possível montar um teste de mesa para resolver o problema que deu resultados quando convertido para o código.

O projeto ainda tem vários pontos que podem ser melhorados ou aprofundados, esses pontos servem como gancho para a criação de trabalhos posteriores. São esses:

a) Implementar as lógicas para comparação de documentos com mais de uma página;
b) Incluir no trabalho uma opção para realizar a comparação textual além da comparação por pixels;
c) Anexar ferramentas como o Jenkins que pode disparar o teste quando uma nova versão do sistema for liberada, além de enviar os resultados obtidos com o teste por e-mail para os responsáveis;
d) Trabalhar no processamento de imagem para que pixels sobrepostos e com o mesmo valor RGB possam ser tratados e destacados como uma diferença. Como pode ser visto nos resultados obtidos da figura 28, o centro das imagens de mantiveram a cor azul, pois o código do teste não viu diferenças, então manteve a cor original dos arquivos. Uma sugestão para resolver esse problema seria analisar os pixels ao redor, se estes já tiverem a cor de destaque, muito provavelmente este pixel também deve ser destacado;
e) Incluir regiões de exclusão para o comparador. Alguns relatórios podem apresentar cabeçalhos com data e hora. Esses valores sempre serão diferentes, porém não caracterizam uma falha. Nesses casos é importante criar zonas de exclusão que não passam pelo comparador.

Contudo, pode-se concluir que os resultados atingidos estão de acordo com o que foi proposto no objetivo geral e nos objetivos específicos. Resultando em um teste que é capaz de sair do escopo do sistema e validar os dados emitidos no documento PDF, destacando as diferenças encontradas e trazendo os resultados de forma clara ao final do teste.

REFERÊNCIAS

ADOBE, Site da ferramenta *Adobe Acrobat*. Disponível em: < https://theblog.adobe.com/who-created-pdf/>. Acesso em: 26 nov. 2018.

ANICHE, Maurício. **Testes Automatizados de Software:** Um guia prático. São Paulo: Casa do Código, 2015.

APACHE PDFBOX, Site da ferramenta Apache PDFBox. Disponível em: <https://pdfbox.apache.org/index.html>. Acesso em: 26 nov. 2018.

AVILALA, Aswani Kumar. **Creating Extent Reports in Selenium Using the Extent API.** 2018. Disponível em: <https://dzone.com/articles/creating-extent-reports-in-selenium-using-extent-a>. Acesso em: 25 nov. 2018.

BERNARDO, Paulo Cheque; KON, Fabio. A Importância dos Testes Automatizados. **Engenharia de Software Magazine**, São Paulo, v. 3, n. 1, p.54-57, dez. 2008. Disponível em: <https://www.ime.usp.br/~kon/papers/EngSoftMagazine-IntroducaoTestes.pdf>. Acesso em: 19 ago. 2018.

BEUST, Cédric. **TestNG:** Testing, the Next Generation. 2004. Disponível em: <http://beust.com/weblog/>. Acesso em: 25 nov. 2018.

CARDEAL, Wisney; PARREIRA JÚNIOR, Walteno Martins. **Estudo sobre a Utilização de Testes Automatizados em Projeto de Software de Grande Porte**. In: Simpósio de Pós-Graduação do IFTM (Simpós), 2., 2015. Disponível em: <http://www.waltenomartins.com.br/Simpos2015_3.pdf>. Acesso em: 19 ago. 2018.

DAVE, Premal. **Top 10 Automated Software Testing Tools.** 2016. Disponível em: <https://dzone.com/articles/top-10-automated-software-testing-tools>. Acesso em: 25 nov. 2018.

DIFFPDF, Site da ferramenta DiffPDF. Disponível em:< http://www.qtrac.eu/diffpdf.html>. Acesso em: 26 nov. 2018

EXTENT REPORTS, Site da ferramenta *Extent Reports.* Disponível em: < http://www.extentreports.com>. Acesso em: 24 nov. 2018.

FEWSTER, Mark; GRAHAM, Dorothy. **Software Test Automation: Effective use of test execution tools**. London: ACM Press, 1999.

HARLEY, Nick. **11 of the most costly software errors in history.** 2018. Disponível em: <https://raygun.com/blog/costly-software-errors-history/>. Acesso em: 03 out. 2018.

HARU PDF LIBRARY, Site da ferramenta Haru Free PDF Library. Disponível em: < http://libharu.sourceforge.net/>. Acesso em: 26 nov. 2018.

JAIN, Sandeep et al. **Image Processing In Java:** Set 2 (Get and set Pixels). 2017. Disponível em: <https://www.geeksforgeeks.org/image-processing-java-set-2-get-set-pixels/>. Acesso em: 01 jun. 2019.

JOHNSON, Duff. The 8 most popular document formats on the Web in 2015. 2015. Disponível em: <http://duff-johnson.com/2015/02/12/the-8-most-popular-document-formats-on-the-web-in-2015/>. Acesso em: 25 nov. 2018.

jsPDF, Site da ferramenta jsPDF. Disponível em:< https://parall.ax/products/jspdf>. Acesso em: 10 abr. 2019.

KANER, C. **The power of 'What If...' and nine ways to fuel your imagination:** Cem Kaner on scenario testing. Software Testing and Quality Engineering, v. 5, n. 5, 2003, p. 16-22.

KOSCIANSKI, André; SOARES, Michel dos Santos. **Qualidade de Software:** Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software. 2. ed. São Paulo: Novatec, 2007. 395 p.

LAZARTE, Maria. PDF format becomes ISO standard. 2008. Disponível em: <https://www.iso.org/news/2008/07/Ref1141.html>. Acesso em: 25 nov. 2018.

LIMA, André Alves. **Como podemos comparar imagens no Windows Forms?** 2016. Disponível em: <http://www.andrealveslima.com.br/blog/index.php/2016/02/03/como-podemos-comparar-imagens-no-windows-forms/>. Acesso em: 26 nov. 2018.

LUTZ, R. R. **Analyzing Software Requirements Errors in Safety-Critical Embedded Systems.** RE'93, San Diego, Calif.: IEEE, 1993.

MARX, Dustin. Creating PDF Documents With Apache PDFBox 2. 2016. Disponível em: <https://dzone.com/articles/creating-pdf-documents-with-apache-pdfbox-2>. Acesso em: 26 nov. 2018.

Mvn repository, Site do repositório Maven. Disponível em: < https://mvnrepository.com/>. Acesso em: 29 abr. 2019.

MCPEAK, Alex. **The Most Common Selenium Challenges.** 2017. Disponível em: <https://crossbrowsertesting.com/blog/selenium/problems-selenium-webdriver/>. Acesso em: 25 nov. 2018.

MCPEAK, Alex. **What's the True Cost of a Software Bug?** 2017. Disponível em: <https://crossbrowsertesting.com/blog/development/software-bug-cost/>. Acesso em: 03 out. 2018.

MOLINARI, Leonardo. **Inovação e Automação de Testes de Software.** São Paulo: Érica, 2014. 140 p.

MYERS, Glenford J.; BADGETT, Tom; SANDLER, Corey. **THE ART OF SOFTWARE TESTING.** 3. ed. New Jersey: John Wiley & Sons, 2012. 240 p.

NIDHRA, Srinivas; DONDETI, Jagruthi. BLACK BOX AND WHITE BOX TESTING TECHNIQUES: A LITERATURE REVIEW. **International Journal Of Embedded Systems And Applications.** India, p. 29-50. jun. 2012. Disponível em: <https://www.researchgate.net/publication/276198111_Black_Box_and_White_Box_Testing_Techniques_-_A_Literature_Review>. Acesso em: 15 out. 2018.

OTWANI, Mukesh. **Advance Selenium Reporting with Screenshots.** 2015. Disponível em: <http://learn-automation.com/advance-selenium-reporting-with-screenshots-2/>. Acesso em: 25 nov. 2018.

PARREIRA JÚNIOR, Walteno Martins. **ENGENHARIA DE SOFTWARE.** 2010. 109 f. TCC (Graduação) - Curso de Engenharia da Computação, Universidade do Estado de Minas Gerais, Belo Horizonte, 2010. Disponível em: <http://www.waltenomartins.com.br/ap_es_v1.pdf>. Acesso em: 19 ago. 2018.

PEIXOTO, Rafael. **Selenium WebDriver:** Descomplicando testes automatizados com Java. São Paulo: Casa do Código, 2018. 274 p.

PRESSMAN, Roger S.. **Software Engineering:** A Practitioner's Approach. 7. ed. New York: Mcgraw-hill Education, 2010. 930 p. Disponível em: <http://dinus.ac.id/repository/docs/ajar/RPL-7th_ed_software_engineering_a_practitioners_approach_by_roger_s._pressman_.pdf>. Acesso em: 06 jul. 2019.

RAVICHANDRAN, Chandra; REKHA, Sasi. Text Extraction from PDF document. 2013. Disponível em: <http://dl.icdst.org/pdfs/files/c07aea28bea2cd04bb952ac9d6d4d263.pdf>. Acesso em: 26 nov. 2018.

RIBEIRO, Daniel. **O que é Sandbox e como usá-lo para proteger o seu computador.** 2014. Disponível em: <https://www.techtudo.com.br/dicas-e-tutoriais/noticia/2014/02/o-que-e-sandbox-e-como-usa-lo-para-proteger-o-seu-computador.html>. Acesso em: 25 nov. 2018.

RONGALA, Arvind. **What is Black Box Testing::** Advantages and Disadvantages. 2015. Disponível em: <https://www.invensis.net/blog/it/black-box-testing-advantages-disadvantages/>. Acesso em: 15 out. 2018.

RONGALA, Arvind. **What is White Box Software Testing::** Advantages and Disadvantages. 2015. Disponível em: <https://www.invensis.net/blog/it/white-box-software-testing-advantages-disadvantages/>. Acesso em: 15 out. 2018.

SEKA, Manikanta Rajkumar. **Page Object Model with Page Factory in Selenium:** Complete Guide | Software Testing Material. 2017. Disponível em: <https://www.softwaretestingmaterial.com/page-object-model/>. Acesso em: 31 maio 2019.

SELENIUM HQ, Site da ferramenta *Selenium.* Disponível em: < https://www.seleniumhq.org/>. Acesso em: 24 nov. 2018.

SHAPIRO, Fred R. **Etymology of the Computer Bug: History and Folklore**. 1987.

SMITH, Ernie. **Why the PDF Is Secretly the World's Most Important File Format.** 2018. Disponível em: <https://motherboard.vice.com/en_us/article/pam43n/why-the-pdf-is-secretly-the-worlds-most-important-file-format>. Acesso em: 25 nov. 2018.

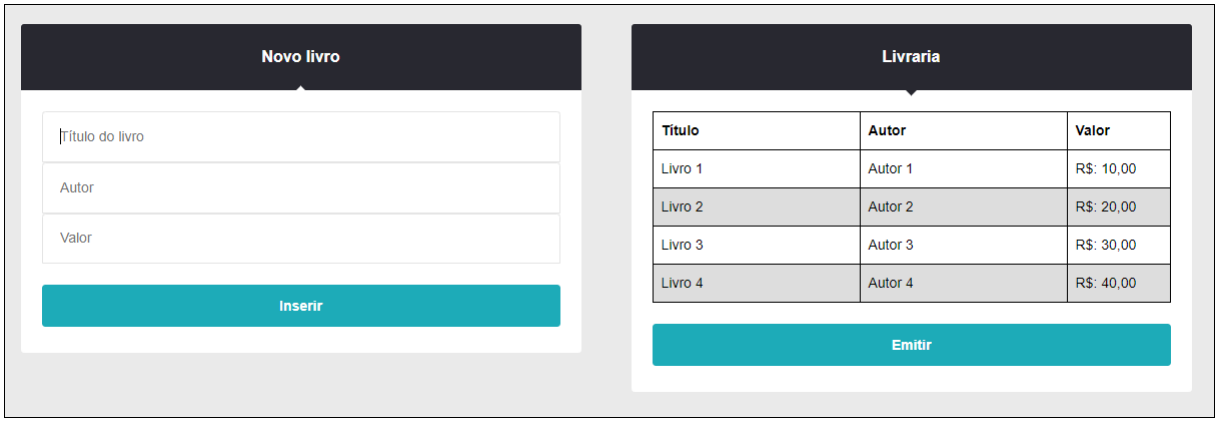
SOMMERVILLE, Ian. **Engenharia de Software.** 9. ed. São Paulo: Pearson, 2012.

TESTNG, Site da ferramenta *TestNG.* Disponível em: < https://testng.org/doc/index.html>. Acesso em: 24 nov. 2018.

WIKLUND, Kristian. **IMPEDIMENTS FOR AUTOMATED SOFTWARE TEST EXECUTION**. 2015. 200 f. Dissertação (Mestrado) - Curso de Engenharia de Software, Mälardalen University Press Dissertations, Suécia, 2015. Disponível em: <https://www.diva-portal.org/smash/get/diva2:808786/FULLTEXT01.pdf>. Acesso em: 22 nov. 2018.

APÊNDICES

Apêndice A – Site da biblioteca para o teste



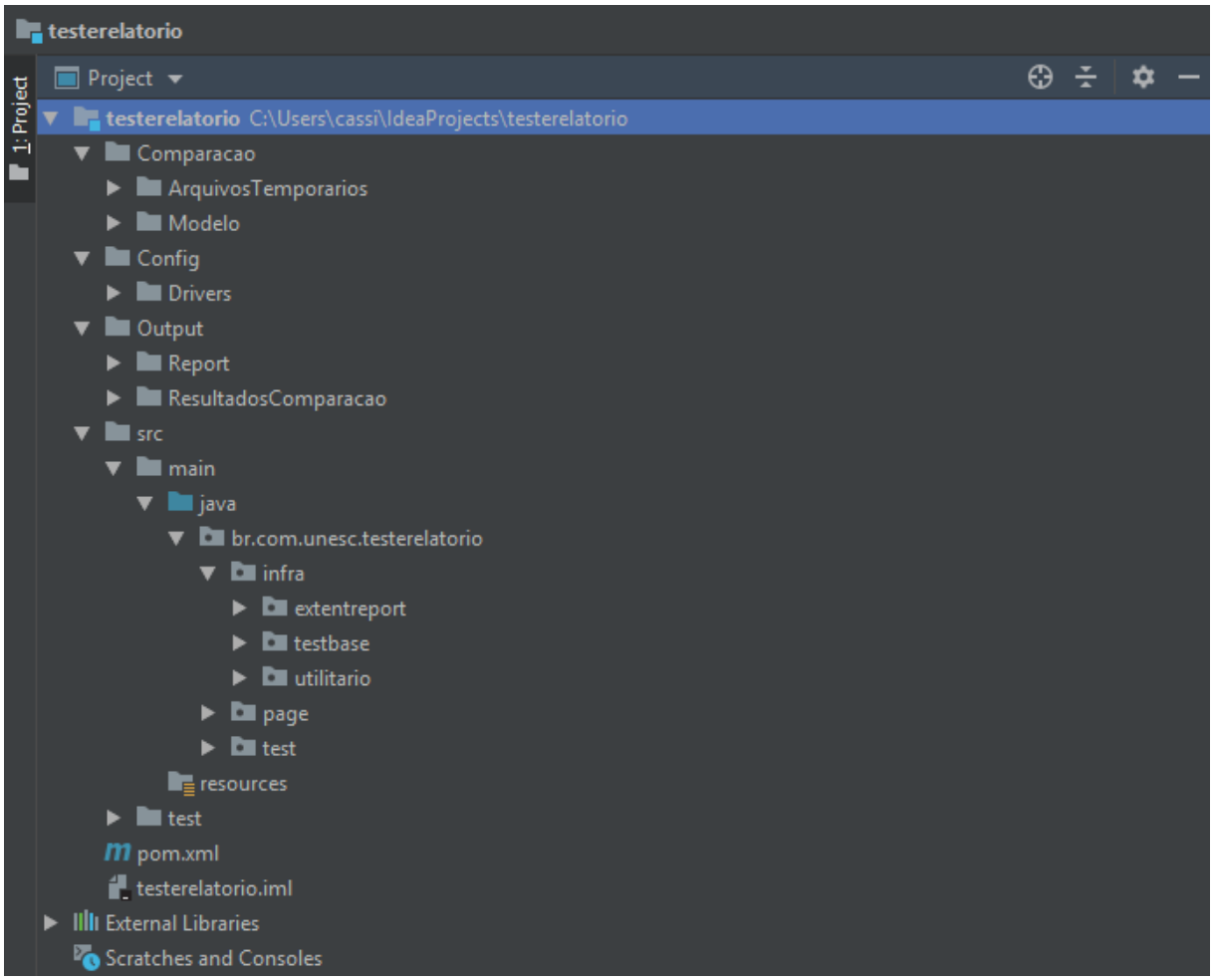
Apêndice B – Relatório em PDF gerado pelo jsPDF

Acervo da livraria

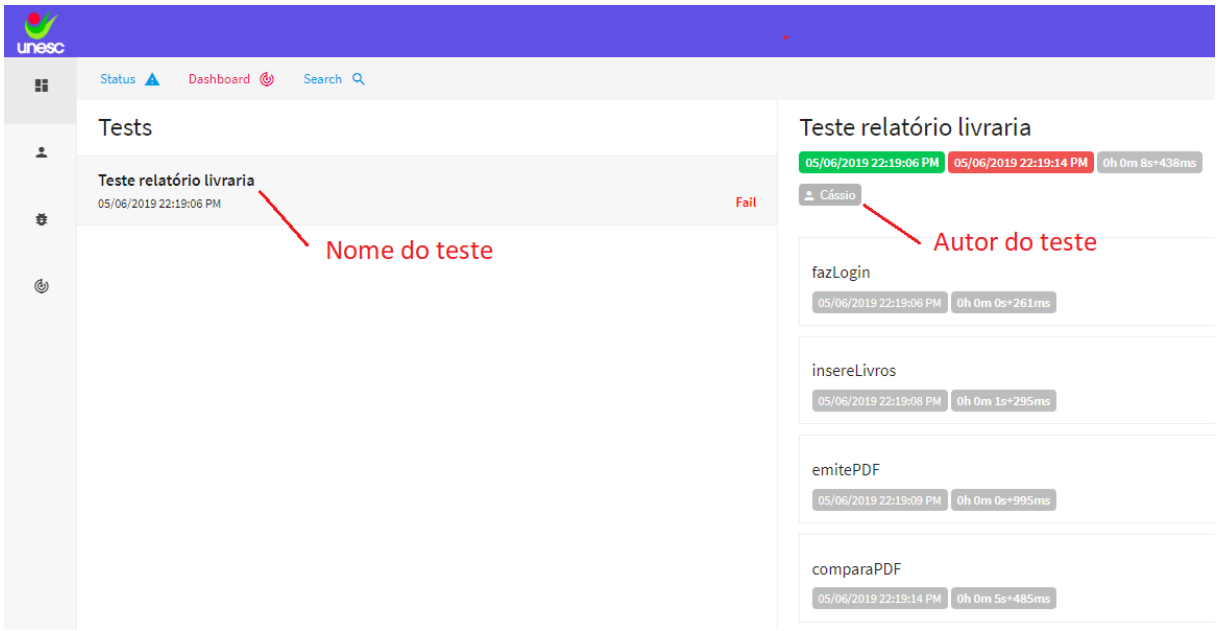
Criciúma - SC

Título	Autor	Valor
Contato	Carl Sagan	R$: 35,90
Bilhões e Bilhões	Carl Sagan	R$: 23,99
Guerra e Paz	Liev Tolstói	R$: 139,90
A Montanha Mágica	Thomas Mann	R$: 68,32
O Gene Egoísta	Richard Dawkins	R$: 58,32

Apêndice C – Estrutura de pastas do projeto



Apêndice D – Nome do teste e do autor exibidos nos resultados



Apêndice E – Classe TesteRelatorio

```java
package br.com.unesc.testerelatorio.test;

import br.com.unesc.testerelatorio.infra.extentreport.IgnoreReport;
import br.com.unesc.testerelatorio.infra.extentreport.TestNomeEAutor;
import br.com.unesc.testerelatorio.infra.testbase.TestBase;
import br.com.unesc.testerelatorio.page.TesteRelatorioPage;
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

import java.io.IOException;

@TestNomeEAutor(getTesteNome = "Teste relatório livraria", getAutor = "Cássio")
public class TesteRelatorio extends TestBase {

    private TesteRelatorioPage testRelPage;

    @Parameters({ "driver" })
    @Test
    @IgnoreReport
    public void iniciaPageFactorys() {
        testRelPage = new TesteRelatorioPage();
    }

    @Test(dependsOnMethods="iniciaPageFactorys")
    public void fazLogin() {
        testRelPage.fazLogin();
    }

    @Test(dependsOnMethods="fazLogin")
    public void insereLivros() {
        testRelPage.insereLivro( titulo: "Contato",  autor: "Carl Sagan",  valor: "35,90");
        testRelPage.insereLivro( titulo: "Bilhões e Bilhões",  autor: "Carl Sagan",  valor: "23,99");
        testRelPage.insereLivro( titulo: "Guerra e Paz",  autor: "Liev Tolstói",  valor: "139,90");
        testRelPage.insereLivro( titulo: "A Montanha Mágica",  autor: "Thomas Mann",  valor: "68,32");
        testRelPage.insereLivro( titulo: "O Gene Egoísta",  autor: "Carl Sagan",  valor: "51,32");
    }

    @Test(dependsOnMethods="insereLivros")
    public void emitePDF() throws InterruptedException {
        testRelPage.emitePDF();
    }

    @Test(dependsOnMethods="emitePDF")
    public void comparaPDF() throws IOException {
        testRelPage.comparaPDF();
    }
}
```

Apêndice F – Artigo

TESTE AUTOMATIZADO PARA VALIDAÇÃO DE DADOS DE RELATÓRIOS APLICANDO CONCEITOS DE COMPARAÇÃO BYTE A BYTE EM ARQUIVOS PDF

Cássio Benincá de Jesus[2]

Resumo: Os sistemas modernos consomem e geram quantidades de informação cada vez maiores. As informações geradas por esses sistemas precisam estar corretas pois pessoas irão tomar decisões baseadas nos dados que estão recebendo. Existem ferramentas que fazem a validação dos dados no sistema, mas grande parte dos dados gerados pelo sistema na forma de documento digital, ainda precisa ser validada pelo profissional de testes manualmente, lendo páginas de documentos e fazendo cálculos para garantir que os valores ali presentes estejam corretos. Esse processo é cansativo e deixa grande margem para que dados incorretos passem despercebidos. Esse projeto visa solucionar esse problema com a automação de testes aplicada a documentação digital, criando um teste que possa ler as informações do documento e destacar graficamente possíveis diferenças encontradas.

Palavras-chave: Automação de teste, Documentos digitais, Comparação de imagens, Selenium

ABSTRACT: Modern systems consume and generate ever-increasing amounts of information. The information generated by these systems must be correct because people will make decisions based on the data they are receiving. There are tools that validate data in these systems, but much of the data generated by the system in the form of digital documents, still needs test professionals manually validating them, reading document pages and making calculations to ensure that the values present are correct. This process is tiring and leaves a lot of room for incorrect data to go undetected. This project aims to solve this problem with test automation applied to digital documentation, creating a test that can read the information in the document and graphically highlight possible differences found.

Key words: Test automation, Digital documents, Image comparison, Selenium

[2] E-mail: cassiouru@gmail.com

1 INTRODUÇÃO

No mundo atual as empresas estão lidando com quantidades cada vez maiores de dados, é muito difícil gerenciar tudo isso sem a ajuda de softwares. Eles estão presentes na maioria das empresas e precisam crescer junto com elas para acompanhar a demanda de funções e a capacidade de processamento de dados. Criar tais sistema exige bastante planejamento por parte das equipes de desenvolvimento.

Para Parreira (2010) engenharia de software é um conjunto integrado de métodos e ferramentas utilizadas para especificar, projetar, implementar e manter um sistema.

Um dos desafios da engenharia de software é o desafio do legado, que é fazer a manutenção e a atualização dos softwares atuais, sem apresentar grandes custos e ao mesmo tempo prosseguir com a prestação dos serviços corporativos essenciais.

Parte do trabalho necessário para manter um sistema são os testes de software. E testar softwares cada vez maiores exige um grande esforço das equipes de teste para o planejamento e execução dos casos de teste.

É comum que em testes mais repetitivos e dispendiosos os testadores deixem de verificar todos os casos de teste a cada alteração do sistema, isso pode acarretar problemas que geram atrasos na entrega ou que só são encontrados mais tarde já no cliente final. É aí que os testes automatizados se fazem necessários, pois eles podem verificar todos os casos de teste, quantas vezes forem necessárias e em um curto espaço de tempo (BERNARDO; KON, 2008).

Bernardo e Kon (2008) apontam que testes automatizados são programas ou scripts que exercitam funcionalidades do sistema sendo testado e fazem verificações automáticas nos efeitos colaterais obtidos.

Mas os sistemas modernos tem vários aspectos para serem analisados. Um deles é a emissão de relatórios. Vários sistemas tem funcionalidades que imprimem listas com uma grande quantidade de dados, essas listas frequentemente saem do escopo do sistema para exibir seus dados através de arquivos de texto

externos, sendo bastante comum a apresentação desses dados em arquivos no formato PDF.

Um teste automatizado consegue simular as ações do usuário interagindo com o sistema (CARDEAL; PARREIRA, 2015), porém quando o sistema emite um relatório em um arquivo de texto externo, o teste automatizado não consegue simular o usuário lendo este arquivo em busca de falhas. Para que o teste consiga fazer tais validações é necessário que outras ferramentas ou técnicas sejam mescladas na automação.

Diante do exposto, esse projeto propõe a implementação de um teste automatizado capaz de validar dados de relatórios através de uma biblioteca que faça a comparação, utilizando dados obtidos através de ferramentas de manipulação de arquivos no formato PDF, apresentando o resultado do teste, e em caso de falha, destacando as diferenças encontradas entre o arquivo de controle e o arquivo de teste.

2 JUSTIFICATIVA

Nos dias de hoje não é possível falar sobre teste de software sem mencionar automação de testes. Um profissional que deseja ser produtivo na área de testes não é aquele que consegue executar o mesmo teste várias vezes em um curto espaço de tempo, e sim aquele que gasta o tempo apenas uma vez para desenvolver um teste automatizado capaz de rodar em segundos centenas de validações diferentes (ANICHE, 2015).

Com toda uma estrutura de testes automatizados agindo em conjunto com os testadores manuais, o sistema pode ser validado por vários lados ao mesmo tempo. Garantindo mais qualidade, menor quantidade de problemas, e uma maior chance de resolver os problemas de prazo mencionados anteriormente.

Softwares que geram grandes quantidades de dados através de relatórios em arquivos de texto precisam de muita confiabilidade. Pois esses arquivos serão lidos por pessoas que tomarão decisões baseadas nos dados que ali estão presentes. Portanto não basta apenas a realização de testes para validar dados dentro do sistema. Os dados devem ser validados em tudo aquilo que o sistema gera. Especialmente quando nos atentamos para o fato de que em alguns casos, as próprias ferramentas para criação de relatórios assumem certas responsabilidades do sistema,

como, fazer consultas no banco de dados, fazer comparações de dados, realizar operações aritméticas, entre outros, antes de emitir os dados.

Isso pode fazer com o que os testes aplicados no sistema demonstrem um falso positivo. Onde o sistema apresentou os dados da forma correta, porém os arquivos de texto emitidos apresentaram falhas que não foram interceptadas por não existir um teste validando essa ponta.

Além disso, os relatórios podem apresentar problemas não somente nos textos, mas também no layout, posicionamento das informações, linhas de fechamento de colunas, imagens e logotipos quem podem não estar sendo exibidos, e várias outras informações não textuais quem podem apresentar algum tipo de problema.

Por isso, um teste automatizado que seja capaz de validar não somente o sistema, como também os documentos gerados por ele, incluindo texto, layout e demais informações visuais, se torna o cenário ideal. Tal validação será possível mesclando ferramentas que aplicam os conceitos de comparação byte a byte de arquivos PDF com as ferramentas de automação como o Selenium. Para que o resultado do teste possa apresentar as possíveis falhas, demonstrando graficamente em quais pontos do arquivo as diferenças estão presentes.

3 METODOLOGIA

O projeto teve início a partir da ideia de que validar páginas de relatórios faz parte do dia a dia de um tester, e seria de grande ajuda se esse processo pudesse ser automatizado assim como o sistema. Para por essa ideia em prática foi necessário levantar o material fonte através das pesquisas de levantamento bibliográfico, que tiveram início na parte de engenharia de software, especificamente na área de testes, englobando os principais modelos de testes existentes.

Tendo como base toda a teoria que define de que forma deve ser um teste, as pesquisas referentes a automação de testes se tornaram o foco. Foram vistas ferramentas para automação, bem como bibliotecas que auxiliam e agregam valor não só ao processo do teste, mas também a apresentação dos resultados encontrados pro usuário final.

O próximo passo foi estudar sobre modelo de documento eletrônico no formato PDF, principalmente encontrar as formas existentes para fazer uma comparação de arquivos nesse formato.

De posse de todos os conhecimentos adquiridos com a pesquisa, os tópicos abaixo vão relatar sobre como foi desenvolvido o projeto, mostrando os resultados obtidos através da simulações, e as conclusões que puderam ser tiradas deste projeto.

A solução desenvolvida consiste em um teste automatizado que simula um usuário logando em um sistema de exemplo, inserindo dados e emitindo um relatório em PDF com esses dados. Após isso, o teste fica responsável por comparar o arquivo PDF obtido com um modelo do que já era esperado. É feita uma validação para garantir que o relatório foi de fato emitido, e em caso positivo, o serviço de comparação do PDF é inicializado.

Por fim, se houveram divergências entre os arquivos, os resultados são apresentados destacando as diferenças entre o que era esperado e o que foi encontrado.

3.1 MONTAGEM DO AMBIENTE DE DESENVOLVIMENTO

Em um primeiro momento foi desenvolvido o site de exemplo que será o objeto do teste. O qual consiste em uma página simples de uma biblioteca, com um formulário de 3 campos que solicita ao usuário as informações de título, autor e valor da obras. Os dados digitados nesses campos alimentam uma tabela HTML para posteriormente ser realizada a emissão de um arquivo PDF com os dados dessa tabela.

Após isso foi a vez da montagem do ambiente de desenvolvimento do teste. O primeiro passo é a instalação da IDE de desenvolvimento IntelliJ para criar o projeto. Com a IDE já instalada, foi criado um projeto do tipo Maven com o nome “testerelatorio”. Como é um projeto do tipo Maven, as bibliotecas que foram utilizadas são instalas através do arquivo de configuração “pom.xml”. E são elas, Selenium, TestNG, Commons-io, PDFBox e ExtentReports.

De posse de todas as bibliotecas instaladas e do ambiente pronto, foram criados os arquivos de configuração de browser do Selenium, além de uma classe base com toda a configuração do teste. Na qual estão inseridos métodos que são

executados antes ou depois de cada etapa com base nas anotações disponibilizadas pelo framework TestNG. Esses métodos são responsáveis por inicializar os drivers do Selenium antes de cada teste, capturar os nomes das classes e métodos antes de suas respectivas execuções para que essas informações possam alimentar a ferramenta ExtentReports que gerará um gráfico com todas as informações obtidas com o teste. Também são definidos métodos que farão a validação dos resultados do teste e métodos de conclusão, que finalizam os testes e apresentam os resultados obtidos.

3.2 MÉTODOS DE COMPARAÇÃO

Para permitir a comparação de documentos em PDF é necessário ter uma ferramenta que faça a manipulação desse tipo de arquivo. Essa ferramenta é o PDFBox da Apache. Além de ser gratuito e oferecer uma grande quantidade de métodos, possui entre eles um que permite a conversão do arquivo em PDF para uma imagem. É possível realizar a comparação extraindo apenas os textos do PDF, mas o objetivo desse trabalho vai além de validar apenas os dados escritos. O intuito é validar o relatório como um todo, garantindo não somente os valores textuais, mas também todo o layout e a estrutura do documento. Isso pode ser feito através da manipulação de imagens, com algoritmos que comparam as imagens pixel por pixel em busca de diferenças. Sendo assim, o primeiro passo após o teste ter baixado documento do site de exemplo, é gerar imagens dos arquivos que serão analisados, para posteriormente os métodos de comparação da classe de comparação varrerem os documentos.

Então, após o teste interagir com o site de exemplo e gerar um relatório PDF que é baixado em um diretório padrão, o PDFBox recebe os caminhos até o arquivo PDF baixado e até o arquivo PDF de controle. Com esses caminhos ele carrega os documentos em memória e renderiza em formato de imagens que são novamente salvas em disco.

A classe Pixels, responsável por fazer a comparação das imagens, recebe o local onde imagens geradas foram salvas, e inicia sua comparação.

Para manipular imagens o Java já traz incluído em sua JDK a biblioteca ImageIO, que permite fazer leitura, alteração e gravação da dados gráficos. E para começar são criados os objetos públicos utilizados ao longo do processo. São eles:

- BufferedImage imgControle: Representação em memória da imagem de controle;
- BufferedImage imgTeste = Representação em memória da imagem do teste;
- BufferedImage marcaDaguaControle = Marca d'água para a imagem de controle;
- BufferedImage marcaDaguaTeste = Marca d'água para a imagem de teste;
- Boolean resultado = Variável que será retornada pelo método de comparação.

O primeiro passo é ler na memória as imagens de controle e do teste. Para isso são chamados os métodos leImagemControle e leImagemTeste que utilizam o método ImageIO.read() passando por parâmetro objetos File contendo os diretórios das imagens. Antes da comparação iniciar é feita uma validação para verificar se ambas as imagens possuem a mesma resolução. Caso elas possuam tamanhos diferentes, o teste já é finalizado com uma falha.

De posse das duas imagens carregadas em memória, e considerando que ambas tenham o mesmo tamanho, o teste pode prosseguir iniciando os laços de repetição que vão varrer as imagens. A figura 1 mostra o código.

Figura 1 – Laço de repetição que faz a comparação

```
if (widthControle == widthTeste || heightControle == heightTeste) {
    for (int w = 0; w < widthControle; w++) {
        for (int h = 0; h < heightControle; h++) {
            int pixelControle = imgControle.getRGB(w, h);
            int pixelTeste = imgTeste.getRGB(w, h);

            int brilhoControle = pixelControle / 3;
            int brilhoTeste = pixelTeste / 3;

            if (pixelControle != pixelTeste) {
                if (brilhoControle < brilhoTeste){
                    pixelControle = corVermelho;
                    imgControle.setRGB(w, h, pixelControle);
                }else {
                    pixelTeste = corVermelho;
                    imgTeste.setRGB(w, h, pixelTeste);
                }
                resultado = true;
            }
        }
    }
```

Fonte: Do autor

Neste caso um laço fica dentro do outro, o primeiro vai varrer a largura da imagem armazenando a posição na variável "w", e o segundo vai varrer a altura da

imagem armazenando a posição na variável “h”. Após isso a função getRGB(w, h) armazena cada pixel da posição atual num inteiro referente ao pixel de controle e ao pixel de teste.

Os inteiros brilhoControle e brilhoTeste recebem uma média da somatória dos RGB. Que como visto na fundamentação teórica pode ir de 0 a 255. Esses inteiros serão utilizados posteriormente na comparação para poder criar a separação entre os destaques aplicados nas imagem de controle e de teste.

O próximo trecho de código é o if principal da comparação. Como visto na fundamentação teórica, após capturar o RGB através do método getRGB() é feita a separação das cores individuais do pixel uma em cada variável inteira contendo apenas seus 8 bits. Isso permite um controle maior sobre as cores que estão sendo manipuladas. Porém, como este projeto não se propõe a fazer manipulações gráficas avançadas, o intuito é apenas comparar os pixels e encontrar diferenças, não é necessário aplicar toda a lógica de separação dos componentes de um pixel. Por isso, a comparação ocorre com todo o conjunto de 32 bits retornados pela função.

Logo, se o pixelControle for diferente do pixelTeste, a lógica para destacar essa diferenças deve iniciar. A lógica desenvolvida envolve as médias armazenadas nas variáveis brilhoControle e brilhoTeste. Se houve uma diferença entre os pixels, então o código deve entrar em outra condição para avaliar se o brilhoControle é menor que o brilhoTeste. Se este for o caso, o pixelControle recebe a cor vermelha, e é setado na sua imagem. Se o brilhoControle for maior que o brilhoTeste é o pixelTeste que recebe a cor vermelha e é setado na sua imagem.

Após aplicar o destaque no pixel, o laço de repetição continua até ter varrido toda a imagem e encontrado todas as diferenças. A variável booleana que será retornada pela função já recebe o valor true desde a primeira diferença encontrada.

A partir daqui, é feita apenas uma validação para ver a variável boolenana está verdadeira, e com isso se iniciam as lógicas para gravar as imagens contendo as diferenças para que o resultado do teste possa exibi-las no final. Essas imagens também terão marcas d’água adicionadas para ser possível diferenciar qual imagem era o resultado esperado e qual imagem é o resultado encontrado.

4 CONCLUSÃO

Desenvolver um teste automatizado requer um tempo grande de desenvolvimento durante a criação do projeto, e o retorno só começa a vir mais tarde, quando o teste já pode ser aplicado e replicado em outros projetos. Isso foi levantado durante a fundamentação teórica e pode ser comprovado com a criação deste trabalho.

Criar um teste automatizado capaz de validar dados em relatórios emitidos em formato PDF exigiu a busca de conhecimentos variados e uma grande quantidade de tempo até fosse possível enxergar resultados satisfatórios.

As ferramentas auxiliares pesquisadas durante o projeto facilitaram a tarefa e agregaram valor aos resultados. A IDE de desenvolvimento IntelliJ se mostrou leve e inteligente, trazendo sugestões pertinentes de melhoria e otimização do código, além de rodar sem maiores problemas de travamentos mesmo quando os testes foram feitos em máquinas com menor capacidade de processamento e memória. Além disso a IDE ainda traz suporte nativo a várias ferramentas de teste, incluindo a biblioteca TestNG utilizada nesse projeto. Todos esses recursos podem ser habilitados durante a instalação da IDE.

A biblioteca TestNG juntamente com o ExtentReports permitiram controlar as rotinas que ocorrem antes, durante e depois de cada teste, como os métodos de configuração do ambiente, de validação do teste e de gravação dos resultados.

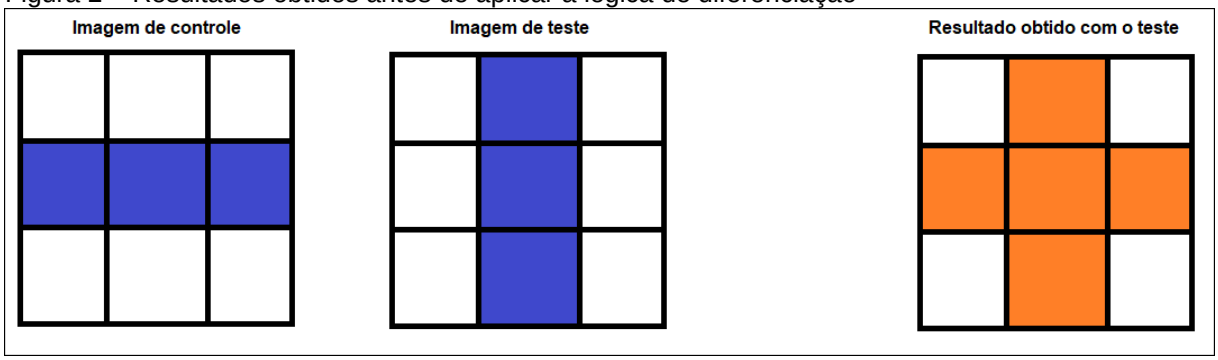
O PDFBox traz métodos para gerar imagens a partir de um arquivo PDF, e um ponto importante é a resolução personalizável dessas imagens. Foi importante ter controle sobre isso pois arquivos pequenos não permitem um resultado satisfatório na hora de ler as diferenças encontradas no teste, e arquivos muito grandes geram lentidão na hora que o método de comparação começa a varrer os pixels. Além da possibilidade de gerar imagens, essa biblioteca ainda traz recursos para manipular documentos com várias páginas, isso não foi abordado nesse projeto, mas é um ponto de melhoria para trabalhos futuros.

Uma vez pronto o teste automatizado ficou claro como ele pode auxiliar no dia a dia de um tester. Ter uma ferramenta que pode conferir não somente o sistema como também os relatórios gerados por ele permite ao tester garantir que a versão do sistema está pronta para liberação com um maior grau de segurança do que se comparado ao processo manual de inspecionar visualmente vários relatórios. Uma

tarefa como essa cansa rapidamente e é aí que os erros começam a passar despercebidos. Já o teste automatizado não tem o mesmo problema, e pode ser executado quantas vezes forem necessárias.

Um dos principais problemas encontrados durante o desenvolvimento foi a mesclagem das diferenças encontradas nas imagens, não sendo possível destacar claramente quais eram as diferenças. Veja a figura 2.

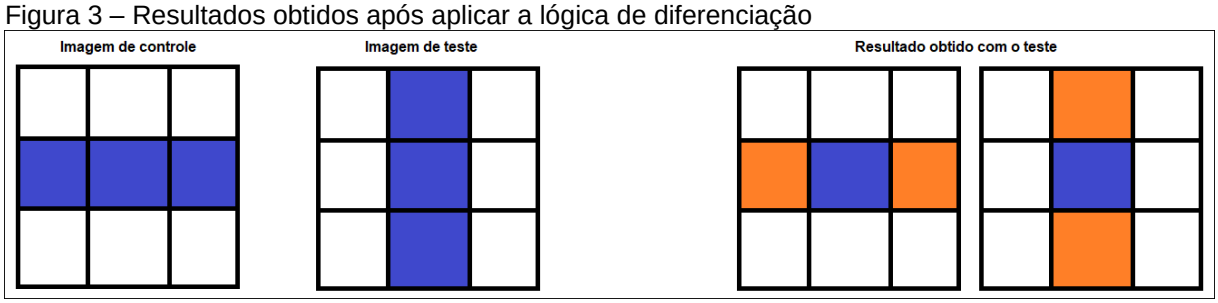
Figura 2 – Resultados obtidos antes de aplicar a lógica de diferenciação



Fonte: Do autor

Esse resultado servia para apontar que existem diferenças nos arquivos, mas não era satisfatório, o desejado é que o teste aponte claramente as diferenças encontradas destacando o que era esperado e o que foi encontrado. Para isso foi desenvolvida a lógica de comparação da média dos valores RGB. Com isso os resultados começaram a ficar bem melhores, já sendo possível apresentar no final do teste duas imagens, uma com o resultado esperado e uma com o resultado encontrado, conforme pode ser visto na figura 3.

Figura 3 – Resultados obtidos após aplicar a lógica de diferenciação



Fonte: Do autor

O projeto ainda tem vários pontos que podem ser melhorados ou aprofundados, esses pontos servem como gancho para a criação de trabalhos posteriores. São esses:

f) Implementar as lógicas para comparação de documentos com mais de uma página;

g) Incluir no trabalho uma opção para realizar a comparação textual além da comparação por pixels;

h) Anexar ferramentas como o Jenkins que pode disparar o teste quando uma nova versão do sistema for liberada, além de enviar os resultados obtidos com o teste por e-mail para os responsáveis;

i) Trabalhar no processamento de imagem para que pixels sobrepostos e com o mesmo valor RGB possam ser tratados e destacados como uma diferença. Como pode ser visto nos resultados obtidos da figura 28, o centro das imagens de mantiveram a cor azul, pois o código do teste não viu diferenças, então manteve a cor original dos arquivos. Uma sugestão para resolver esse problema seria analisar os pixels ao redor, se estes já tiverem a cor de destaque, muito provavelmente este pixel também deve ser destacado;

j) Incluir regiões de exclusão para o comparador. Alguns relatórios podem apresentar cabeçalhos com data e hora. Esses valores sempre serão diferentes, porém não caracterizam uma falha. Nesses casos é importante criar zonas de exclusão que não passam pelo comparador.

Contudo, pode-se concluir que os resultados atingidos estão de acordo com o que foi proposto no objetivo geral e nos objetivos específicos. Resultando em um teste que é capaz de sair do escopo do sistema e validar os dados emitidos no documento PDF, destacando as diferenças encontradas e trazendo os resultados de forma clara ao final do teste.

REFERÊNCIAS

APACHE PDFBOX, Site da ferramenta Apache PDFBox. Disponível em: <https://pdfbox.apache.org/index.html>. Acesso em: 26 nov. 2018.

BERNARDO, Paulo Cheque; KON, Fabio. A Importância dos Testes Automatizados. **Engenharia de Software Magazine**, São Paulo, v. 3, n. 1, p.54-57, dez. 2008. Disponível em: <https://www.ime.usp.br/~kon/papers/EngSoftMagazine-IntroducaoTestes.pdf>. Acesso em: 19 ago. 2018.

CARDEAL, Wisney; PARREIRA JÚNIOR, Walteno Martins. **Estudo sobre a Utilização de Testes Automatizados em Projeto de Software de Grande Porte**. In: Simpósio de Pós-Graduação do IFTM (Simpós), 2., 2015. Disponível em: <http://www.waltenomartins.com.br/Simpos2015_3.pdf>. Acesso em: 19 ago. 2018.

EXTENT REPORTS, Site da ferramenta *Extent Reports.* Disponível em: < http://www.extentreports.com>. Acesso em: 24 nov. 2018.

JAIN, Sandeep et al. **Image Processing In Java:** Set 2 (Get and set Pixels). 2017. Disponível em: <https://www.geeksforgeeks.org/image-processing-java-set-2-get-set-pixels/>. Acesso em: 01 jun. 2019.

Mvn repository, Site do repositório Maven. Disponível em: < https://mvnrepository.com/>. Acesso em: 29 abr. 2019.

PARREIRA JÚNIOR, Walteno Martins. **ENGENHARIA DE SOFTWARE.** 2010. 109 f. TCC (Graduação) - Curso de Engenharia da Computação, Universidade do Estado de Minas Gerais, Belo Horizonte, 2010. Disponível em: <http://www.waltenomartins.com.br/ap_es_v1.pdf>. Acesso em: 19 ago. 2018.

TESTNG, Site da ferramenta *TestNG.* Disponível em: < https://testng.org/doc/index.html>. Acesso em: 24 nov. 2018.