

UNIVERSIDADE ESTADUAL DE PONTA GROSSA
SETOR DE ENGENHARIAS, CIÊNCIAS AGRÁRIAS E DE TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE SOFTWARE

JOÃO VITOR MARTINS DOS SANTOS

**TESTES AUTOMATIZADOS APLICADOS NO CICLO DE VIDA DE
DESENVOLVIMENTO DE SOFTWARE: ESTUDO DE CASO DA
IMPLEMENTAÇÃO EM UMA APLICAÇÃO WEB**

PONTA GROSSA
2022

JOÃO VITOR MARTINS DOS SANTOS

**TESTES AUTOMATIZADOS APLICADOS NO CICLO DE VIDA DE
DESENVOLVIMENTO DE SOFTWARE: ESTUDO DE CASO DA
IMPLEMENTAÇÃO EM UMA APLICAÇÃO WEB**

Trabalho de Conclusão de Curso no formato de monografia
apresentado ao Departamento de Informática - Universidade
Estadual de Ponta Grossa, como requisito para obtenção do
título de Bacharel em Engenharia de Software.

Orientador: Márcio Augusto de Souza
Universidade Estadual de Ponta Grossa

Coorientador: Luiz Pedro Petroski
Pontifícia Universidade Católica do Paraná

PONTA GROSSA
2022

RESUMO

Esta monografia apresenta uma proposta de um método para implementação de testes automatizados em projetos de software, utilizando como estudo de caso uma aplicação *web* já desenvolvida e implementada utilizando o *Framework web Laravel*, a qual foi desenvolvida sem a utilização de testes automatizados. Utilizaram-se procedimentos e técnicas encontrados na literatura para identificação, planejamento e criação de cenários de testes de integração necessários para testar casos de uso da aplicação *web* de forma eficaz. Foi proposta uma representação do sistema usando mapas mentais que permite visualizar os testes necessários mais facilmente. Esses testes foram automatizados utilizando a ferramenta *PHPUnit*, configurada por padrão em projetos que utilizam o *framework Laravel*. Os testes foram executados, e seus resultados analisados. A partir da análise dos resultados e das práticas utilizadas no planejamento dos testes e implementação dos *scripts* propôs-se um conjunto de recomendações para implementação de testes automatizados.

Palavras-chave: Testes Automatizados. Testes de Integração. Qualidade de Software. *Laravel*. *PHPUnit*.

LISTA DE FIGURAS

Figura 1 – Fases do Modelo Cascata	4
Figura 2 – Modelo V	5
Figura 3 – Ciclo de Vida XP	10
Figura 4 – Desenvolvimento do Trabalho	19
Figura 5 – Página <i>home</i> do Sistema Admink.	20
Figura 6 – Esquema representando o diagrama proposto pelo método para representação de cenário de testes utilizando diagramas baseados em mapas mentais. . .	22
Figura 7 – Diagrama representando os cenários de teste da funcionalidade <i>Efetuar Login</i>	25
Figura 8 – Diagrama representando os cenários de teste da funcionalidade <i>Cadastro de Orçamentos</i>	26
Figura 9 – Diagrama representando os cenários de teste das funcionalidades <i>Listagem e Deleção de Orçamentos</i>	27
Figura 10 – Diagrama representando os cenários de teste das funcionalidades <i>Edição de Orçamentos</i>	28
Figura 11 – Comparação do diagrama com o script de teste	30
Figura 12 – Relatório da execução dos testes automatizados destacando em vermelho o tempo de execução e a quantidade de testes e asserções realizadas com sucesso	31
Figura 13 – Exemplo de um teste cuja asserção falhou	33

Lista de Algoritmos

5.1	Cenários de Testes Automatizados de Integração do Login	29
5.2	Script do teste teste1ListagemDeOrcamentosComSucesso	34
5.3	Script de Teste exemplificando asserções baseadas em campos de documentos JSON	36
5.4	Scripts de teste de criação de orçamentos	36

SUMÁRIO

1 – INTRODUÇÃO	1
1.1 OBJETIVO GERAL	2
1.2 OBJETIVOS ESPECÍFICOS	2
1.3 ORGANIZAÇÃO DO TEXTO	2
2 – PROCESSO DE DESENVOLVIMENTO DE SOFTWARE	4
2.1 PROCESSOS SEQUENCIAIS	4
2.2 PROCESSOS ÁGEIS	6
2.2.1 Programação Extrema (XP)	8
2.2.1.1 Práticas orientadas para os negócios	10
2.2.1.2 Práticas orientadas à equipe	11
2.2.1.3 Práticas técnicas	11
3 – TESTE DE SOFTWARE	12
3.1 TESTES AUTOMATIZADOS	12
3.2 NÍVEIS DE TESTE	13
3.2.1 Teste unitário	13
3.2.2 Testes de integração	15
3.2.3 Testes de sistema	15
3.3 TIPOS DE TESTE	15
3.3.1 Testes funcionais	15
3.3.2 Testes não funcionais	16
3.3.3 Testes de confirmação	16
3.3.4 Testes de Regressão	16
3.4 TÉCNICAS DE TESTE	17
3.4.1 Teste caixa-branca	17
3.4.2 Teste caixa-preta	17
3.5 FACTORIES	17
4 – MATERIAL E MÉTODOS	19
4.1 LEVANTAMENTO BIBLIOGRÁFICO	19
4.2 DESCRIÇÃO DO SOFTWARE ESTUDADO	19
4.3 PLANEJAMENTO DE TESTES	21
4.3.1 Representação dos cenários de teste baseada em mapas mentais	21
4.4 IMPLEMENTAÇÃO E EXECUÇÃO DOS TESTES AUTOMATIZADOS	23
5 – RESULTADOS E DISCUSSÕES	25

6 – CONCLUSÕES	39
---------------------------------	-----------

Referências	41
------------------------------	-----------

Apêndices	42
------------------	-----------

APÊNDICE A–Descrição dos casos de uso do Sistema Admink	43
--	-----------

APÊNDICE B–Plano de Testes do Sistema Admink	45
---	-----------

APÊNDICE C–Scripts de Teste de Integração do Sistema Admink	48
--	-----------

1 INTRODUÇÃO

Softwares são suscetíveis a falhas em seu funcionamento, as quais são inconsistências entre o comportamento desejado e o comportamento apresentado após o desenvolvimento. Essas falhas e inconsistências podem chegar até o usuário final, tendo potencial de gerar prejuízos à imagem do software e de seus responsáveis e até mesmo prejuízos financeiros.

A baixa incidência de falhas é uma das características de um software de qualidade, além de questões como a conformidade do comportamento do software com o que se propõe a fazer e até mesmo quão bem o software se comporta. Ou seja, além de estar livre de falhas o software deve atender as necessidades e expectativas de seus usuários. Dada a importância de se desenvolver software com qualidade, é fundamental que existam atividades de teste em projetos de desenvolvimento de software como forma de evitar que um software com baixa qualidade chegue ao usuário final.

Nos processos tradicionais de desenvolvimento de software as fases são sequenciais, então é comum que as atividades de teste de *software* sejam realizadas após o desenvolvimento do software ter sido finalizado. Além disso, geralmente a execução dos testes é realizada de forma manual por equipes especializadas e dedicadas a essas atividades.

Como alternativa aos processos tradicionais foram criados métodos ágeis que buscam entregar softwares em funcionamento em ciclos curtos de desenvolvimento durante os quais ocorrem todas as atividades de desenvolvimento de software e não mais em longas fases sequenciais.

Após o surgimento desses métodos ágeis, as atividades de teste passaram a ser mais valorizadas em qualquer tipo de software, e sofreram mudanças na forma como são executadas. Muitos testes passaram a ser automatizados, possibilitando que, além da detecção de erros, seja assegurado o correto funcionamento do software após alterações e também que os testes sirvam como documentação para o código desenvolvido. (VALENTE, 2020). Também é comum que as atividades de teste sejam realizadas pela própria equipe que realiza o desenvolvimento do software, não mais existindo fases e equipes dedicadas exclusivamente a realização de testes.

Para Crispin (2009), os testes manuais levam muito tempo e tendem a levar mais tempo a cada nova iteração, demandando que cada vez mais pessoas precisem dedicar tempo à realização de testes manuais, aumentando o débito técnico e a frustração. Os testes automatizados fornecem *feedback* rapidamente e frequentemente, através da execução de testes sempre que há escrita de código novo. Erros causados por alterações no comportamento do software devido às mudanças do código podem ser detectados, possibilitando que as correções ocorram rapidamente. A automação de testes proporciona a redução do tempo necessário para execução dos testes de um software quando ele sofre atualizações, além da criação automática de relatórios sobre a execução dos testes automatizados.

É consenso entre os profissionais que desenvolvem software que é necessário testar

o código desenvolvido, mas as atividades de escrita de código de testes são frequentemente deixadas de lado devido à urgência e à pressão para terminar um projeto. Maior urgência e pressão pode levar a menor escrita de testes. Escrevendo menos testes a produtividade diminui, pois o código torna-se menos estável. Consequentemente, com menor produtividade, a urgência e a pressão aumentam, alimentando um ciclo que leva os profissionais a exaustão. A melhor forma de persuasão sobre o valor da criação de *scripts* de teste é implementando os testes em um projeto de desenvolvimento real, mostrando na prática o real valor do *feedback* rápido fornecido pela execução dos testes e os benefícios que oferecem para a refatoração de código (BECK; GAMMA, 1998).

A criação dos *scripts* de testes automatizados demanda tempo, mas os benefícios de sua implementação geram redução de tempo em outras atividades do projeto, como a execução dos testes, economizando cada vez mais tempo e esforço a longo prazo. Entretanto para implementar testes automatizados são necessários conhecimentos sobre planejamento de testes, codificação e ferramentas para criação e execução de *scripts* de testes automatizados.

Dito isso é de importante estudar a implementação de testes automatizados em projetos reais de desenvolvimento de software, buscando explorar as práticas necessárias para implementação dos mesmos, além dos benefícios e as dificuldades enfrentadas para sua implementação. Este trabalho se propõe a realizar tal estudo com intuito de fornecer recomendações para a implementação de testes automatizados, dados seus benefícios e importância.

1.1 OBJETIVO GERAL

O objetivo deste trabalho é propor um método para implementação de testes automatizados em projetos de software.

1.2 OBJETIVOS ESPECÍFICOS

Com base no objetivo geral, foram definidos objetivos específicos:

- Definir uma forma de representação dos cenários de teste baseada em mapas mentais para facilitar a definição dos testes a serem feitos;
- Inserir o processo de testes no fluxo de trabalho de métodos ágeis;
- Apresentar um estudo de caso do uso de testes em um software desenvolvido anteriormente chamado Sistema Admink;
- Definir um conjunto de recomendações para aplicação de testes automatizados de integração.

1.3 ORGANIZAÇÃO DO TEXTO

Nos primeiros capítulos pode ser encontrada a fundamentação teórica do trabalho. O Capítulo 2 aborda os processos de produção de software, suas características e evolução ao

longo do tempo. Já o Capítulo 3 descreve conceitos e práticas de teste de software e testes automatizados.

Os capítulos seguintes abordam a execução do trabalho e seu desfecho. O Capítulo 4 detalha o material e os métodos utilizados na implementação de testes automatizados. O Capítulo 5 mostra os resultados obtidos após a implementação e execução dos testes automatizados e as recomendações propostas. Por fim o Capítulo 6 explora as conclusões alcançadas.

2 PROCESSO DE DESENVOLVIMENTO DE SOFTWARE

Para que equipes produzam software com qualidade e de forma produtiva é necessário um ordenamento mesmo que mínimo. O processo de desenvolvimento de software determina um conjunto de ações que devem ser seguidas. Seu uso é importante para que as empresas possam gerenciar o trabalho de produção de software garantindo a produtividade e o alinhamento com os objetivos da empresa. Também é importante para os desenvolvedores, para que estejam cientes de suas tarefas e dos resultados esperados e para que não se sintam perdidos, desalinhados com os demais envolvidos ou para que não trabalhem sem previsibilidade (VALENTE, 2020).

Neste capítulo serão abordadas duas alternativas distintas de processos de desenvolvimento de software, começando pelos processos tradicionais, com foco nas características do Modelo Cascata, um dos primeiros processos criados para desenvolvimento de software, o qual é baseado em processo da Engenharia tradicional. Em seguida serão abordados os processos ágeis, que foram propostos como alternativa aos processos tradicionais por um grupo de profissionais insatisfeitos que buscavam resolver os principais problemas dos processos utilizados até então, com foco nas características do método de Programação Extrema, conhecido como XP.

2.1 PROCESSOS SEQUENCIAIS

Os processos sequenciais de desenvolvimento de software surgiram baseados em projetos de Engenharia tradicional, como a engenharia eletrônica e engenharia civil, nos quais existem planejamentos detalhados previamente e etapas sequenciais que geram documentações detalhadas em cada etapa do processo (VALENTE, 2020).

O ciclo de vida clássico de desenvolvimento de software, conhecido como Modelo Cascata (PRESSMAN, 2015), propõe uma abordagem de fases sequenciais (Figura 1).

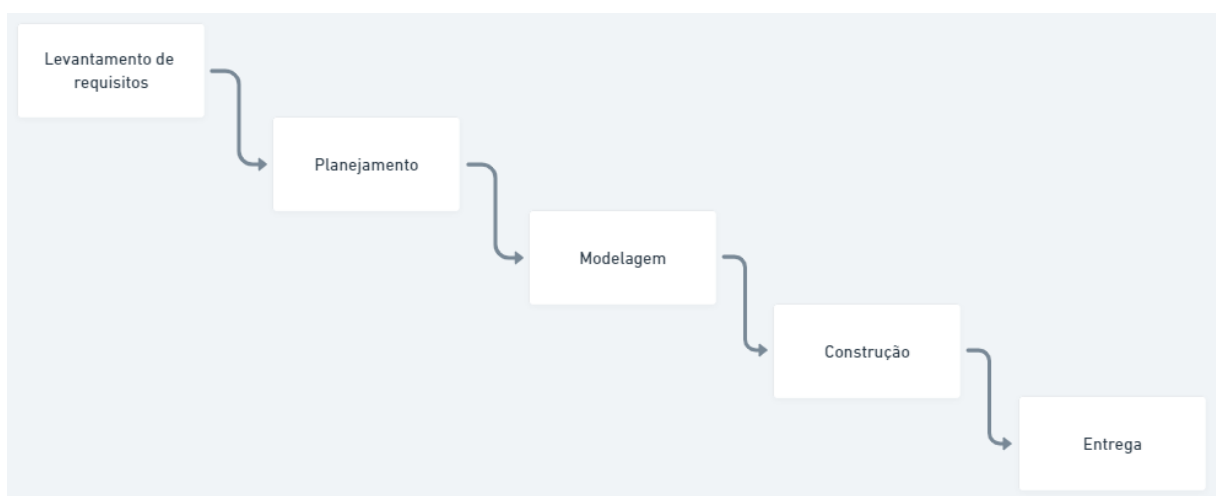


Figura 1 – Fases do Modelo Cascata

Fonte: Pressman (2015)

As fases podem durar semanas ou meses e a fase seguinte só é iniciada após a fase anterior ter sido finalizada. Cada fase possui foco e atividades específicas:

- O projeto inicia com a fase de levantamento dos requisitos, na qual são realizadas atividades para identificação das necessidades do cliente do projeto;
- Em seguida ocorre a fase de planejamento, incluindo atividades de estimativas, cronograma e acompanhamento;
- O projeto avança pela fase de modelagem, na qual são realizadas a análise, o projeto do software e a construção de diagramas e modelos;
- Na sequência ocorre a fase de construção do software, a qual engloba as atividades de codificação e em seguida os testes do software;
- Por fim o projeto chega na fase final, na qual é realizada a entrega do software, é prestado suporte e são obtidos os *feedbacks* do cliente.

As atividades de garantia de qualidade e de testes executadas na fase de construção se relacionam com o que é produzido nas fases anteriores. O modelo V (Figura 2) é uma representação de quais atividades de teste estão relacionadas com os trabalhos de cada uma das fases do projeto. (PRESSMAN, 2015)

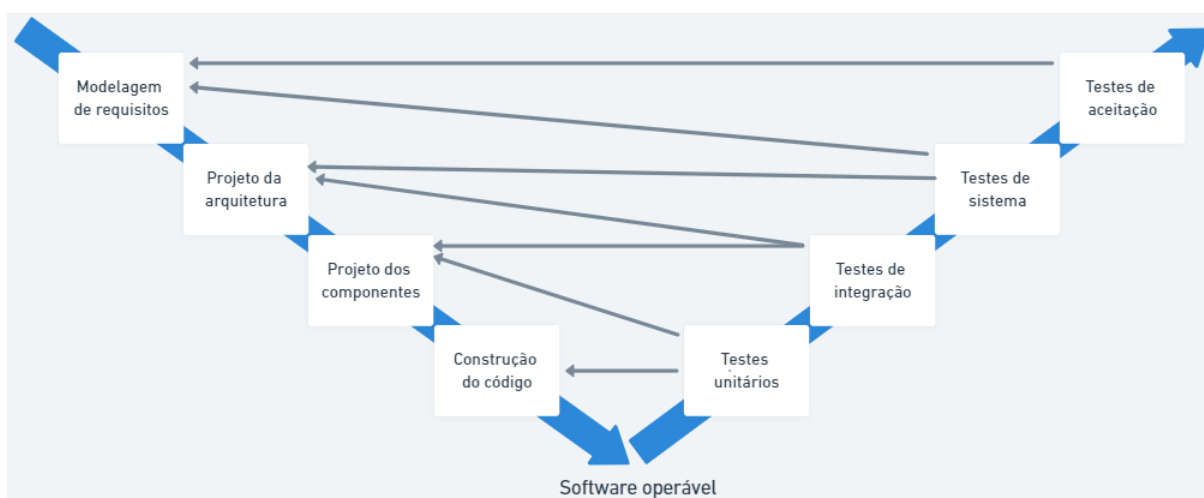


Figura 2 – Modelo V

Fonte: Pressman (2015)

As atividades de testes em processos sequenciais como o Cascata geralmente são realizadas manualmente por equipes dedicadas à execução de testes e ocorrem após finalizado o desenvolvimento do software. O software, que já está maduro o suficiente para ser executado, é utilizado nos testes de forma semelhante a que será utilizado pelos usuários finais, mas são informados dados de entrada específicos para teste. Os testes obtêm sucesso quando os comportamentos apresentados pelo software após a realização dos testes estiverem de acordo com os comportamentos que são esperados que sejam apresentados, com base nas especificações produzidas nas fases anteriores e atendendo às necessidades dos usuários.

Como um dos objetivos das atividades de teste é o de revelar defeitos do software, a realização dos testes traz à tona erros cometidos em fases anteriores à de testes, como codifica-

ções incorretas, as quais causam falhas durante a execução ou comportamentos inconsistentes. Quanto mais tarde os defeitos forem encontrados, mais onerosa será a correção, pois além da correção do código do software para que não apresente mais tal defeito, poderá ser necessária a adequação da documentação produzida para que corresponda às alterações realizadas no software para corrigir o defeito.

Testes de software tem relação direta com o processo de desenvolvimento de software, apresentando diversos conceitos e definições que serão exploradas no Capítulo 3 com maior profundidade.

O modelo cascata pode ser considerado o primeiro processo de desenvolvimento de software, mas ao longo dos anos sua eficácia passou a ser questionada. Alguns dos problemas encontrados são (PRESSMAN, 2015):

- Os projetos reais raramente seguem um fluxo linear e sequencial, então as mudanças podem ocasionar confusão à medida que o projeto avança;
- Geralmente o cliente tem dificuldade de explicitar todas as suas necessidades no início do projeto, como é requerido pelo modelo cascata;
- As versões operáveis do software só estarão disponíveis próximos ao final do projeto, ocasionando em uma tardia detecção de erros e em descontentamentos do cliente.

Os processos sequenciais podem funcionar bem para projetos que possuem requisitos fixos e nos quais o projeto pode ser executado do começo ao fim de forma linear sem muitas mudanças, de maneira que são recomendados para projetos com incertezas ou mudanças frequentes o uso de processos ágeis.

2.2 PROCESSOS ÁGEIS

Devido aos problemas dos processos tradicionais, um grupo de profissionais se reuniu para propor uma alternativa aos processos sequenciais para desenvolvimento de software, defendendo que software é diferente de produtos tradicionais da Engenharia, portanto necessita de um processo de desenvolvimento diferente. Esse encontro resultou em um documento que é conhecido como manifesto ágil. Com o manifesto ágil passam a ser valorizados (VALENTE, 2020):

- Indivíduos e interações, mais do que processos e ferramentas;
- Validação do software mais do que uma documentação abrangente;
- Colaboração com o cliente mais do que negociação de contratos;
- Responder a mudanças mais do que seguir um plano.

O que caracteriza os processos ágeis são os ciclos de desenvolvimento iterativos e de curta duração, de uma a quatro semanas. O sistema é implementado gradualmente, começando pela implementação de uma versão com as funcionalidades que são mais prioritárias para o cliente. Após o cliente validar e aprovar essa versão, um novo ciclo, também chamado de iteração, é iniciado dando sequência ao desenvolvimento das demais funcionalidades também priorizadas pelo cliente, de maneira que todo ciclo produz uma versão funcional do software.

O desenvolvimento é finalizado quando o cliente entende que todas as funcionalidades estão implementadas (VALENTE, 2020).

Além dos valores do manifesto ágil, há 12 princípios de agilidade (PRESSMAN, 2015):

- A maior prioridade é satisfazer o cliente através da entrega antecipada e contínua de software de valor;
- Mudanças nos requisitos são bem-vindas mesmo no fim do desenvolvimento, pois os processos ágeis aproveitam as mudanças como vantagem competitiva para o cliente;
- Deve-se entregar software funcionando frequentemente, de algumas semanas a alguns meses, preferindo-se os intervalos mais curtos;
- Pessoas de negócio e desenvolvedores devem trabalhar juntos diariamente ao longo do projeto;
- Deve-se construir projetos em torno de indivíduos motivados, dando a eles o ambiente e o suporte que precisarem, confiando neles para fazer o trabalho;
- O método mais eficiente e efetivo de transmitir informações com a equipe de desenvolvimento é a conversa cara a cara;
- Software funcionando é a principal medida de progresso;
- Processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capaz de manter um ritmo constante indefinidamente;
- Atenção contínua para excelência técnica e bom projeto (*design*) eleva a agilidade;
- Simplicidade, a arte da maximização do trabalho economizado, é essencial;
- As melhores arquiteturas, requisitos e projetos emergem de equipes auto-organizadas.
- Regularmente a equipe reflete sobre como se tornar mais efetiva, então sintoniza e ajusta seu comportamento de acordo.

Os processos que aplicam os valores e princípios do manifesto ágil mudam a visão sobre a responsabilidade pelas atividades de validação e testes. Todo time torna-se responsável por tais atividades e pela qualidade do software produzido, não sendo mais uma responsabilidade exclusiva de uma equipe de testes diferente da que realiza o desenvolvimento.

Algumas outras características dos processos ágeis são: documentação apenas do que é essencial, sendo mais importante conseguir avançar mesmo com incertezas e possibilidade de mudança do que fazer um plano detalhado; não existência de uma fase dedicada ao projeto (*design*), o qual também é feito de forma incremental; organização de times pequenos de desenvolvimento; e ênfase em novas práticas de desenvolvimento como programação em pares, testes automatizados e integração contínua. Devido a essas características, os processos ágeis são considerados leves, com pouca documentação. Essas características são genéricas e abrangentes, então foram criados métodos para auxiliar na adoção dos valores e princípios ágeis de forma concreta. (VALENTE, 2020).

Apesar de haver diversos métodos que aplicam os valores e princípios do manifesto ágil, neste trabalho será abordado o método intitulado Programação Extrema (*Extreme Programming*), também conhecido como XP. O método XP é um dos mais bem definidos e

completos e é o mais recomendado para entender a metodologia ágil, pois outros métodos ágeis empregam subconjuntos ou variações do XP, além de empregar práticas de testes automatizados em seu ciclo de vida (MARTIN, 2020), que são o foco deste trabalho.

2.2.1 Programação Extrema (XP)

O XP é um método leve recomendado para o desenvolvimento de software com requisitos imprecisos ou sujeitos a mudança e aplica as características dos processos ágeis. Entretanto o XP não define uma sequência detalhada de passos para produzir software, e sim um conjunto de valores e princípios que devem fazer parte da cultura e do dia a dia dos times de desenvolvimento, então esses valores e princípios são concretizados em práticas de desenvolvimento (VALENTE, 2020).

O XP é conduzido pelos valores da comunicação, simplicidade, *feedback*, coragem e respeito. Boa comunicação é essencial para evitar erros e também aprender com eles. A simplicidade é importante para focar em projetar mais as necessidades atuais que as futuras, produzindo projetos fáceis de serem implementados e que podem ser evoluídos futuramente, caso necessário. O rápido *feedback* do cliente, dos membros da equipe e do próprio software através de uma estratégia de testes eficaz são essenciais para controlar riscos e diminuir prejuízos e retrabalhos. É necessário ter coragem e disciplina para projetar para as necessidades atuais, tendo em mente que as necessidades futuras podem mudar e exigir mudanças no projeto e no código. Os integrantes de time ágeis devem respeitar os demais membros da equipe, os demais envolvidos no projeto, o próprio software, além de ter respeito pelo processo XP (PRESSMAN, 2015).

A aplicação dos valores de simplicidade, comunicação e o *feedback* representado pelos testes proporcionam um avanço mútuo, pois um valor beneficia o outro: Os testes melhoram a comunicação através do registro das interfaces do programa; A comunicação melhora os testes aumentando as chances de defeitos serem revelados; A simplicidade é aumentada pelos testes, pois aumentam a possibilidade de refatorar código facilmente; E a simplicidade melhora os testes, pois um código mais simples é mais simples de se testar ((BECK, 1998)).

Além desses valores, o XP também possui alguns princípios que são procedimentos mais concretos e pragmáticos (VALENTE, 2020):

- Humanidade: O principal recurso de uma empresa são seus colaboradores. Uma boa gestão de pessoas é a chave para o sucesso de projetos de software;
- Economicidade: Software é caro e quem está pagando por ele espera resultados econômicos e financeiros, então software não é uma obra de arte, mas uma produção que precisa de resultados econômicos;
- Benefícios Mútuos: As decisões tomadas nos projetos de software devem beneficiar a múltiplos envolvidos;
- Melhorias Contínuas: Como nenhum processo de desenvolvimento de software é perfeito, trabalhar em um software e em práticas de desenvolvimento que são aprimoradas a cada

iteração a partir *feedback* do cliente e da equipe é mais seguro;

- Falhas Acontecem: Software é uma das construções humanas mais complexas, então é esperado que softwares apresentem falhas e inconsistências. Elas não devem ser encobertas, mas também não devem ser usadas para punição do time;
- Passos de bebê (*baby steps*): Pequenos progressos contínuos na direção certa são melhores que grandes revoluções, pois estas costumam gerar resultados negativos. Ou seja, é melhor concluir pequenas atividades constantemente que acumular uma grande quantidade de trabalho realizado aguardando ser concluído;
- Responsabilidade Pessoal: Os desenvolvedores devem ter clareza de seu papel e responsabilidade na equipe. O engenheiro de software que realiza a implementação também é responsável pelos testes e manutenção.

Além dos valores e princípios, o XP adota um conjunto de práticas para desenvolvimento ágil de software conhecidas como Ciclo de Vida. Muitas dessas práticas podem ser aplicadas inclusive em projetos que utilizam outros métodos ágeis. Algumas das práticas são orientadas para os negócios, outras para a equipe e outras são práticas técnicas (Figura 3).



Figura 3 – Ciclo de Vida XP

Fonte: Martin (2020)

2.2.1.1 Práticas orientadas para os negócios

As práticas orientadas para os negócios do XP definem como a equipe de desenvolvimento de software se comunica com os negócios e como o projeto é gerenciado (MARTIN, 2020):

- Planejamento do Jogo: Atividade que resulta na divisão do projeto em funcionalidades, histórias de usuário¹ e tarefas;
- Pequenas Versões: A equipe trabalha em pedaços pequenos do software;
- Testes de Aceitação: São testes realizados para verificar que o software está pronto e pode ser utilizado pelos usuários. Esses testes usam critérios de conclusão claros definidos para cada história ou tarefa individualmente, e também critérios válidos para todas as histórias ou tarefas desenvolvidas pelo time para determinar quando estão concluídas.

¹É uma forma simples e leve de descrever requisitos de maneira concisa e sob a perspectiva do usuário.

- Equipe como um todo: Uma equipe de desenvolvimento de software possui muitas atribuições e responsabilidades diferentes como codificação, testes e gerenciamento, todas convergindo para o mesmo objetivo.

2.2.1.2 Práticas orientadas à equipe

As práticas orientadas à equipe determinam como a equipe de desenvolvimento se comunica e se gerencia (MARTIN, 2020):

- Ritmo Sustentável: Permite que o time desenvolva o projeto em um ritmo constante sem se esgotar no meio do projeto;
- Propriedade Coletiva: Evita que membros isolados da equipe detenham as informações do projeto apenas para si;
- Integração Contínua: o time de desenvolvimento constantemente deve integrar seu código a um repositório central. Feito isso, uma série de ações de construção (*build*) da aplicação e de verificação, como a execução dos testes automatizados existentes, são realizadas de forma automática, oferecendo um *feedback* rápido sobre o código integrado, permitindo identificar problemas rapidamente e mantendo a equipe ciente do andamento das atividades;
- Uso de Metáforas: Metáforas são utilizadas para que todos os envolvidos estabeleçam um vocabulário apropriado para o projeto e para que todos se entendam ao se comunicar sobre o software.

2.2.1.3 Práticas técnicas

As práticas técnicas direcionam os programadores para atingirem a mais alta qualidade técnica (MARTIN, 2020):

- Programação em pares: Possibilita o compartilhamento de conhecimento entre a equipe, resultando em inovação e precisão;
- Projeto (*design*) Simples: Garante que a equipe evite desperdiçar esforços;
- Refatoração: Proporciona a melhoria contínua e o aperfeiçoamento do que é produzido pela equipe;
- Desenvolvimento Guiado por Testes (*Test Driven Development, TDD*): Consiste em um conjunto de práticas por meio das quais, para cada pequena parte do software que será desenvolvida, os testes automatizados são implementados antes do código do software. Após criados, os testes são executados, porém falham inicialmente pois o código do software ainda não existe. Em seguida o desenvolvimento do código é iniciado e os testes são executados até que obtenham sucesso em sua execução. Por fim o código e os testes são evoluídos mantendo-se os testes alcançando sucesso em suas execuções.

A maioria das práticas do XP são utilizadas até mesmo por equipes que utilizam outro método ágil devido aos benefícios que essas práticas trouxeram para a produção ágil de software.

3 TESTE DE SOFTWARE

As atividades de teste de software estão presentes tanto nos processos tradicionais de desenvolvimento de software como nos processos ágeis, apesar das diferentes formas que as atividades são realizadas em cada um dos processos. Entretanto, em ambos os processos é necessário conhecimento dos fundamentos e técnicas de teste de software para que as atividades possam ser realizadas corretamente.

Para Myers (2012), testes de software são processos projetados para assegurar que códigos computacionais façam apenas o que propõem, não fazendo nada inesperado. Softwares devem ser previsíveis e consistentes, não apresentando surpresas para os usuários.

Segundo Graham et al. (2019), o teste de software pode ter diferentes objetivos dependendo de fatores como: pontos de vista, níveis de teste, partes interessadas no teste, contextos dos componentes, softwares testados e ciclos de vida de desenvolvimento. Alguns desses objetivos são:

- Avaliar produtos de trabalho;
- Verificar o cumprimento de requisitos especificados;
- Validar a completude de componentes ou softwares e sua conformidade com as expectativas dos usuários e outras partes interessadas;
- Gerar confiança no nível de qualidade de componentes ou softwares;
- Prevenir defeitos;
- Revelar falhas e defeitos;
- Prover informações para tomada de decisão;
- Reduzir o nível de risco de qualidade de software inadequada;
- Cumprir com requisitos e padrões regulatórios ou contratuais.

Independente do objetivo as atividades de teste não podem aumentar a qualidade do software, mas ajudam a revelar se o software desenvolvido possui ou não qualidade.

3.1 TESTES AUTOMATIZADOS

Testes automatizados são códigos computacionais que exercitam funcionalidades do software testado e verificam automaticamente os resultados obtidos. Essa abordagem traz algumas vantagens (BERNARDO; KON, 2008):

- A rapidez e facilidade de executar os testes a qualquer momento, mesmo repetidas vezes;
- A reprodução do mesmo conjunto de ações em todas as execuções, permitindo simular precisamente todos os passos que geram comportamentos específicos, evitando falhas humanas e contribuindo para a identificação de comportamentos indesejados;
- A possibilidade de executar diferentes testes simultaneamente;
- A possibilidade de realizar testes em condições em que há dificuldade de realizá-los

manualmente, como situações elaboradas e complexas envolvendo combinações de comandos e operações, ou então a simulação de utilizações simultâneas do software ou da parte dele que está sendo testada.

Sommerville (2019) descreve que os métodos de teste automatizados são compostos por três partes:

- **Configuração:** o software a ser testado é iniciado com as entradas e saídas esperadas. Por exemplo, são instanciados e inicializados os objetos que pretende-se testar.
- **Chamada:** são realizadas as chamadas aos métodos ou objetos que estão sendo testados.
- **Asserção:** é realizada a comparação dos resultados obtidos nas chamadas com os resultados esperados, sendo a asserção verdadeira quando os resultados obtidos e esperados são iguais. Os testes obtêm sucesso quando a asserção é verdadeira e falham quando a asserção é falsa.

As atividades de teste automatizado demandam tempo para serem realizadas. Apesar disso colaboram para a redução de tempo em outras atividades de teste do projeto devido às suas vantagens.

3.2 NÍVEIS DE TESTE

Para Pressman (2015), uma estratégia na qual os testes são realizados apenas após o sistema estar finalizado não funciona e resulta em um software com defeitos. Uma abordagem preferencial é seguir uma estratégia que emprega testes de forma incremental, começando por testes de unidades individuais do software, seguido por testes de integração de unidades e finalizando com testes utilizando o sistema completo. Graham et al. (2019) nomeia essas diferentes instâncias do processo de testes como níveis de teste.

3.2.1 Teste unitário

Teste unitário, também conhecido como teste de unidade ou teste de componente, tem foco na verificação da menor unidade do software, como um componente ou um módulo de software, e tem enfoque na lógica interna de processamento e nas estruturas de dados dos componentes (PRESSMAN, 2015). A complexidade dos testes e os erros revelados por eles são restritos ao escopo do teste unitário. Esse teste pode ser realizado em paralelo para vários componentes.

Para Sommerville (2019), os testes unitários devem ser automatizados sempre que houver a possibilidade. Segundo Valente (2020), os testes unitários são aplicados de forma automatizada a pequenas unidades do código, normalmente classes, e são isolados do restante do software.

Os testes unitários não dependem que todo o software esteja finalizado para serem realizados e obterem sucesso, basta que o componente testado tenha sido implementado. O componente testado, entretanto, pode ter dependência de outros componentes ainda não

implementados, o que pode tornar o processo de testes mais lento (SOMMERVILLE, 2019). Por exemplo, componentes que realizam comunicação com banco de dados podem precisar de uma etapa prévia de configuração para que o banco possa ser utilizado, podendo atrasar a realização dos testes.

Os testes unitários trazem algumas vantagens para o desenvolvimento do software (VALENTE, 2020):

- Revelar defeitos ainda durante o desenvolvimento do software em uma fase em que as correções são menos custosas, evitando que os defeitos cheguem até o usuário final;
- Adicionam proteção contra defeitos causados por modificações no código, pois se um defeito for introduzido, os testes que exercitam a parte do software afetada pelo defeito deixarão de obter sucesso;
- Auxiliam na documentação e especificação do código, pois revelam detalhes do comportamentos do código testado.

Como o teste unitário foca nas partes internas do componente testado e em sua verificação de forma isolada, podem ser utilizados objetos simulados (*mock objects*) no lugar das dependências do componente testado. Esses objetos simulam a funcionalidade e possuem a mesma interface do componente testado (SOMMERVILLE, 2019). Por exemplo, é possível utilizar um objeto simulando um banco de dados com poucos itens que podem ser acessados rapidamente, eliminando a sobrecarga de utilizar um banco de dados real. Também é possível utilizar objetos para simular a ocorrência de operações anormais ou que não ocorrem com frequência.

Para que os testes unitários atinjam seus objetivos eles devem atender a algumas propriedades. Os testes devem ser rápidos, independentes, determinísticos, auto-verificáveis e escritos a tempo. Essas propriedades são conhecidas como princípios FIRST, acrônimo formado pelas iniciais das palavras em inglês referentes a cada um dos princípios (*fast, independent, repeatable, self-checking e timely*).

(VALENTE, 2020) descreve cada um desses princípios como:

- Rápidos (*fast*): os testes unitários devem executar rapidamente, ou seja, em um tempo de execução de alguns milissegundos, pois serão executados frequentemente e devem fornecer *feedback* rápido sobre o funcionamento do código. Testes que não possam ser executados rapidamente podem ser separados em um conjunto de testes que têm maior tempo de execução e que serão executados com menor frequência;
- Independentes (*independent*): o resultado da execução de cada teste não deve ser afetado pela ordem de execução dos mesmos, podendo até mesmo serem executados de forma paralela, ou seja, os testes não devem depender de ações realizadas por outros testes;
- Determinísticos (*repeatable*): Os testes devem ter o mesmo resultado todas as vezes que executados nas mesmas condições, ou seja, apenas alterações no código devem ser capazes de alterar o resultado dos testes. Testes com resultados não determinísticos são chamados de testes erráticos (*flaky*);

- Auto-verificáveis (*Self-checking*): Os resultados de execução dos testes devem ser verificados com facilidade, sendo autoexplicativos e não dependendo de ações manuais para que sejam interpretados. Os resultados devem mostrar claramente quais testes obtiveram sucesso e quais falharam, e caso o teste falhe, devem possibilitar identificar rapidamente as asserções falsas;
- Feitos a tempo (*timely*): Os testes devem ser escritos o quanto antes, se possível até mesmo antes do código que será testado, como é feito no desenvolvimento guiado por testes.

Além dos princípios adotados para que os testes sejam efetivos, algumas características, chamadas de *test smells*, devem ser evitadas sempre que possível (VALENTE, 2020). Por exemplo:

- Testes obscuros, compreendidos por testes longos, complexos e difíceis de entender. O ideal é testar apenas um requisito de sistema em cada teste;
- Uso de estruturas condicionais e laços de repetição no código de teste;
- Duplicação de código em testes.

3.2.2 Testes de integração

Teste de integração, também chamado de teste de serviço, verifica a integração entre diferentes componentes do software ou entre diferentes sistemas. Para Valente (2020), o teste de integração executa serviços ou funcionalidades completas, engloba um maior número de componentes do software do que os testes unitários e não utiliza objetos simulados, sendo um teste maior, mais demorado e, portanto, executado com menos frequência que o teste unitário.

3.2.3 Testes de sistema

Testes de sistema, também chamados de testes de ponta a ponta ou testes de interface, são testes que manipulam todos os componentes e dependências do software de forma integrada. Esses testes simulam o uso real do sistema e são os testes que necessitam de maior esforço para serem implementados, que têm uma maior dificuldade para serem escritos e que levam mais tempo para serem executados. (VALENTE, 2020).

3.3 TIPOS DE TESTE

Testes de software podem ser categorizados de acordo com os diferentes objetivos, os quais são relacionados às diferentes características do software testado. Os diferentes tipos de teste podem ser aplicados nos diferentes níveis de teste.

3.3.1 Testes funcionais

Os testes funcionais verificam os requisitos funcionais do sistema ou componente testado, ou seja, o que o software deve ser capaz de fazer, as ações e funcionalidades que deve

ser capaz de operar. Os testes podem se basear nas documentações criadas para especificar os requisitos funcionais, mas também podem se basear em requisitos funcionais não documentados, que nesse caso são revelados por necessidades do processo do negócio ou pela comparação do software testado com softwares com propósito semelhante. Esses testes devem ser realizados em todos os níveis de teste, podendo ter diferentes focos em cada nível. A porcentagem de requisitos funcionais que são exercitados pelos testes é uma forma de medir a eficácia dos testes funcionais, medida que é chamada de cobertura funcional (GRAHAM et al., 2019).

3.3.2 Testes não funcionais

Testes não funcionais têm o objetivo de verificar requisitos não funcionais do software testado, ou seja, as características do software que não estão relacionadas às funcionalidades, mas a quão bem o software ou componente testado se comporta e se ele possui as características que são esperadas que possua, mas focando em questões como desempenho, escalabilidade, usabilidade e outras características que não são requisitos funcionais

Testes não funcionais verificam as características de qualidade ou os atributos não funcionais do software ou componente e utilizam critérios que podem ser medidos, como por exemplo, o tempo de resposta (GRAHAM et al., 2019).

3.3.3 Testes de confirmação

Após corrigir um defeito no software é necessário exercitar novamente o software reproduzindo as ações que revelaram o defeito.

O teste de confirmação verifica se as correções realizadas no software realmente corrigiram com sucesso os defeitos que pretendiam corrigir. O software pode ser testado realizando-se novamente os testes que falharam devido ao defeito, além de novos testes que podem ser identificados a partir da correção implementada. O teste de confirmação é realizado em todos os níveis de teste. (GRAHAM et al., 2019)

3.3.4 Testes de Regressão

Após modificar o código de um software, seja para correções em funcionalidades ou em partes do software já em funcionamento ou para adição de novas funcionalidades ao software, é comum que aconteçam erros de regressão, ou seja, defeitos que foram inseridos em partes do software que anteriormente funcionavam corretamente.

Testes de regressão têm o objetivo de detectar os comportamentos indesejados que podem ter sido inseridos acidentalmente em qualquer parte do software pelas modificações realizadas no código, sendo muito importante a realização de testes de regressão especialmente em ciclos de vida de desenvolvimento iterativo e incremental, nos quais há frequente alteração no código. Como os conjuntos de teste de regressão são executados com frequência e geralmente evoluem lentamente, são fortes candidatos à automação, que deve começar já no início do

projeto, e assim como os testes de confirmação, os testes de regressão são realizados em todos os níveis de teste. (GRAHAM et al., 2019).

O teste de regressão pode ser realizado através da execução de todo o conjunto de testes criados para o software ou parte desse conjunto.

3.4 TÉCNICAS DE TESTE

As técnicas de teste são formas sistemáticas de identificar quais testes devem ser executados, suas condições e os dados que devem ser utilizados nos testes.

3.4.1 Teste caixa-branca

As técnicas de teste de caixa-branca baseiam-se em análises da estrutura interna do software ou componente testado para identificar os testes necessários, concentrando-se na estrutura e processamento. Podem ser aplicados em todos os níveis de teste, mas são mais utilizados em testes unitários e de integração. Os testes de caixa-branca são frequentemente usados como forma de medir a eficácia dos testes através da cobertura de um conjunto de elementos estruturais, ou seja, a quantidade desses elementos que está sendo exercitada durante a execução dos testes. No nível de testes unitários há ferramentas que medem a cobertura do código calculando a porcentagem de elementos executáveis que foram exercitados pelo conjunto de testes executados (GRAHAM et al., 2019).

3.4.2 Teste caixa-preta

As técnicas de teste de caixa-preta baseiam-se na análise das especificações do software e dos requisitos para identificar os testes necessários, incluindo testes funcionais e não funcionais. Concentram-se na identificação das condições e dados de testes adequados e no que deve ser retornado pelo software ou componente testado, mas sem envolver a análise das estruturas internas do software, (GRAHAM et al., 2019).

3.5 FACTORIES

Para realização dos testes são necessários dados específicos para serem informados como entradas no software durante a execução dos testes. *Factories* permitem que seja definido um conjunto de atributos para popular o banco de dados. As entidades do banco de dados que possuem classes que as representam dentro do código do software podem ter um conjunto de dados definidos com auxílio de *factories* inseridos em suas tabela. *Fakers* podem ser utilizados para geração de valores aleatórios. Os *fakers* podem ser usados em conjunto com as *factories* para inserir no banco de dados um conjunto de dados com valores gerados aleatoriamente (LARAVEL, 2021). Então as *factories* podem ser utilizadas para criação de dados para teste

facilitando essa atividade e quando utilizadas em conjunto com *fakers* garante que os dados utilizados nos testes não serão um conjunto fixo, mas irão variar de forma aleatória.

As atividades de teste podem ser realizadas de diversas formas nos projetos de desenvolvimento de software. É necessário entender o contexto de cada projeto e quais abordagens mais atendem suas necessidades para adotar as práticas mais adequadas para execução de suas atividades de teste.

4 MATERIAL E MÉTODOS

O objetivo deste trabalho é propor um método para implementação de testes automatizados em conjunto com métodos ágeis no projetos de software. Buscando atingir tal objetivo o desenvolvimento do trabalho consistiu de algumas etapas (Figura 4):

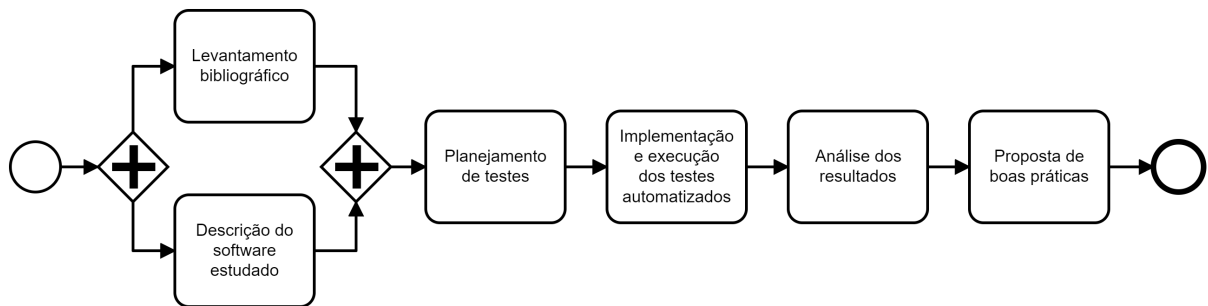


Figura 4 – Desenvolvimento do Trabalho

Fonte: o Autor

4.1 LEVANTAMENTO BIBLIOGRÁFICO

Nesta etapa foi realizado o levantamento bibliográfico para auxílio na implementação dos testes e do entendimento do fluxo de trabalho proposto por métodos ágeis. Foram realizadas buscas na internet utilizando a ferramenta Google Acadêmico, buscando em todos os períodos de publicação, utilizando strings como 'engenharia de software', 'engenharia de software moderna', 'clean agile', 'software testing', 'testes automatizados' e 'software testing foundations'. A partir de alguns dos trabalhos encontrados descreveu-se a relação das atividades de teste de software com os processos de desenvolvimento de software e a evolução de ambos com o passar dos anos. Também foram descritas as atividades de teste de software e de testes automatizados, incluindo as técnicas utilizadas para identificação dos testes necessários para verificar aplicações de forma eficaz.

4.2 DESCRIÇÃO DO SOFTWARE ESTUDADO

O conjunto de práticas de teste de software que pode ser aplicado em um software depende do contexto, ou seja, dependendo do processo, do projeto ou mesmo do software que é testado, algumas práticas podem ser mais apropriadas que outras. A fim de explorar a aplicação das práticas apropriadas, será realizado um estudo de caso da aplicação das práticas de teste de software nas atividades de teste do Sistema Admink.

O Sistema Admink (Figura 5) consiste em uma aplicação web, ou seja, um sistema de informação projetado para ser utilizado através de navegadores na internet, que foi desenvolvido há alguns meses pelo autor desta monografia e pelo estudante Thiago Teixeira durante a

disciplina de Projeto do curso de Engenharia de Software da Universidade Estadual de Ponta Grossa.

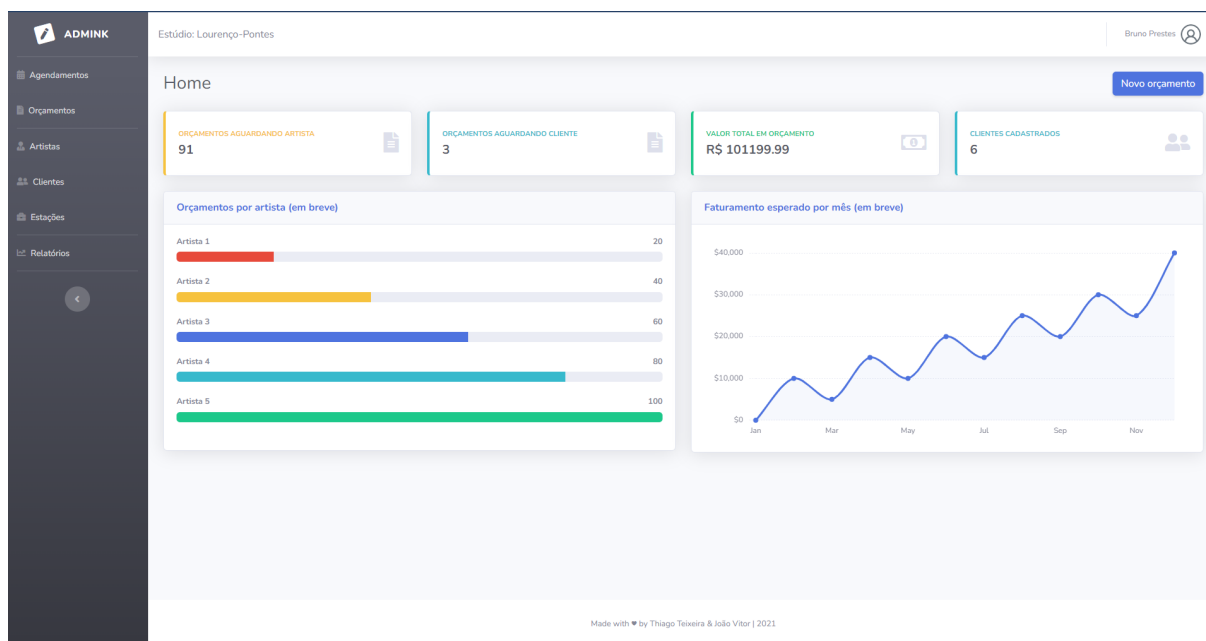


Figura 5 – Página *home* do Sistema AdminK.

Fonte: o Autor

O AdminK foi projetado para apoiar a administração de estúdios de tatuagem através do gerenciamento de informações importantes para o dia a dia do estúdio como dados de clientes, tatuadores, orçamentos, espaços de trabalho, agendamentos de tatuagem e indicadores de apoio à gestão. Previamente ao desenvolvimento do Sistema AdminK foram produzidos documentos de casos de uso (Apêndice A) e casos de teste (Apêndice B).

O Sistema AdminK foi desenvolvido utilizando a linguagem de programação *PHP* na versão 7.3.33, o *framework PHP Laravel* na versão 6.20.44, e o sistema gerenciador de banco de dados relacionais de código aberto MariaDB na versão 10.4.13. O código do sistema está armazenado e versionado em um repositório no *GitHub*.

PHP é uma linguagem de programação interpretada é direcionada para desenvolvimento para a *web* com o principal objetivo de permitir a criação de páginas geradas dinamicamente rapidamente (PHP, 2021).

Laravel é um *framework* para aplicações *web* com sintaxe elegante e expressiva. Um *framework* para aplicações *web* fornece a estrutura e ponto inicial para criação das aplicações, permitindo que desenvolvedores foquem na criação da aplicação sem se preocupar com todos os detalhes. O *Laravel* fornece recursos para a criação, configuração e execução de testes unitários e de integração, pois adota por padrão o *PHPUnit*, que é um *framework* de testes unitários para PHP (LARAVEL, 2021).

O *PHPUnit* é um *framework* de testes orientado ao programador, baseado na arquitetura *xUnit* para *frameworks* de testes de unidade, suportando a realização de testes automatizados. Assim como outros *frameworks* de testes de unidade, o *PHPUnit* utiliza

asserções (*assertions*) para verificar a conformidade do comportamento apresentado pelo código-fonte com o comportamento esperado. As asserções são métodos criados para garantir que o valor recebido em um teste é o valor esperado e no *PHPUnit* há pelo menos 46 métodos de asserção disponíveis (PHP.COM.BR, 2019).

O Sistema Admink também conta com *factories* que foram criadas durante a implementação para apoiar o desenvolvimento e a realização dos testes.

O *GitHub* fornece serviços de hospedagem de repositórios baseados em nuvem, permitindo controle de versão e colaboração de forma remota.

4.3 PLANEJAMENTO DE TESTES

Após entender quão amplos são os estudos de teste de software e mesmo de testes automatizados, manifestou-se a necessidade de delimitar um escopo específico como foco deste trabalho. Optou-se por estudar a implementação de testes funcionais em um dos níveis de teste automatizado para dois casos de uso do Sistema Admink, cobrindo nos testes as funcionalidades completas desses casos de uso. O nível de teste de integração foi escolhido por testar funcionalidades completas do sistema, mas não ser tão oneroso como os testes de sistema. Os casos de uso *Efetuar Login* e *Manter Orçamentos* foram escolhidos por abrangerem as funcionalidades mais importantes para utilização do Sistema Admink.

Com o escopo delimitado foi realizada a análise da documentação previamente criada do Sistema Admink referente aos casos de uso estudados, além da utilização de técnicas de teste de caixa preta para identificação dos cenários de testes. Em seguida os cenários de testes identificados foram representados em diagramas baseados em mapas mentais.

4.3.1 Representação dos cenários de teste baseada em mapas mentais

Buscando atender às necessidades de equipes que aplicam métodos ágeis com menor foco em documentações detalhadas, evitou-se a utilização de documentos extensos e que necessitem de muito tempo para serem interpretados para representação dos cenários de teste, então, propôs-se um método de representação de cenários de teste utilizando diagramas baseados em mapas mentais, contendo as principais informações necessárias para apoiar a implementação dos *scripts* de testes automatizados.

Mapas mentais são um método de representar informações de forma organizada e priorizada, baseado no uso de palavras-chave e imagens-chave capazes de resgatar memórias específicas, além de estimular novos pensamentos (BUZAN, 2009).

Para criação dos diagramas baseados em mapas mentais foi utilizada a aplicação web *XMind*, ferramenta destinada para criação de mapas mentais e realização de *brainstorm* (XMIND, 2022). Entretanto, outras aplicações ou ferramentas voltadas para criação de diagramas baseados em mapas mentais também poderiam ser utilizadas sem prejuízos à aplicação do método. O método proposto consiste nos seguintes passos (Figura 6):



Figura 6 – Esquema representando o diagrama proposto pelo método para representação de cenário de testes utilizando diagramas baseados em mapas mentais.

Fonte: o Autor

- Inicia-se o digrama preenchendo a informação central do diagrama com o nome do caso de uso, história de usuário ou funcionalidade para a qual os cenários de teste serão representados;
- Em seguida são criados os ramos *Cenários* e *Massa de dados*;
- A seguir são inseridos novos ramos ligados diretamente aos ramos *Cenários* e *Massa de dados*, os quais são preenchidos com as informações de cenários de testes de integração e os dados necessários para realização dos testes, respectivamente. Buscando facilitar a identificação de qual parte do código de *scripts* de testes de integração corresponde a cada um dos cenários mapeados, utilizou-se a mesma nomenclatura para os ramos que representam os cenários e para o título dos métodos de teste nos *scripts*;
- Em cada um dos ramos de cenários criados são adicionados os ramos *Passos* e *Resultados*. No ramo *Passos* são descritos os passos que devem ser executados durante a realização do teste. No ramo *Resultados* é descrito o resultado que é esperado que a aplicação apresente após a execução dos passos descritos no ramo *Passos*.

Cada um dos itens delimitados por chaves podem ser editados conforme as informações

correspondente dos cenários de teste, já os itens exibidos em negrito são fixos e não são editados independente das informações dos cenários de teste.

Após definido o método de representação, os cenários de teste planejados para os casos de uso *Efetuar Login* e *Manter Orçamentos* foram representados utilizando o método de representação proposto. Os cenários de teste para o caso de uso *Efetuar Login* foram representados em um único diagrama, entretanto, buscando facilitar a visualização e evitando um diagrama muito extenso, para o caso de uso *Manter Orçamentos* foi criado um diagrama para cada funcionalidade, sendo elas *Cadastro de Orçamentos*, *Listagem* e *Deleção de Orçamentos* e *Edição de Orçamentos*.

4.4 IMPLEMENTAÇÃO E EXECUÇÃO DOS TESTES AUTOMATIZADOS

A partir da delimitação do escopo deste trabalho, consultou-se a seção *Testing* da documentação da versão do *Framework Laravel* utilizada no desenvolvimento do Sistema Admink a fim de descobrir as recomendações para realização de testes de integração. Descobriu-se que projetos criados a partir do *Framework Laravel* já são pré-configurados por padrão com os recursos para criação de testes unitários e de integração, utilizando *PHPUnit*, e que é recomendado utilizar esses recursos para a criação dos *scripts* de testes automatizados.

Após os cenários de testes estarem representados nos diagramas baseados em mapas mentais, iniciou-se a codificação dos *scripts* de testes de integração automatizados utilizando os diagramas como base e seguindo as instruções da documentação do *Framework Laravel* e do *PHPUnit*. Além dos diagramas, o código foi analisado para entender como a aplicação está estruturada e quais partes do código deveriam ser utilizadas para realização dos testes.

Baseando-se nas informações fornecidas pelos diagramas cada um dos *scripts* de teste foi criado utilizando o framework para testes de unidade *PHPUnit* na versão 9.5.17. Cada um dos itens de cada diagrama foi analisado para criar os *scripts* de teste:

- Inicialmente foi criada uma classe por digrama para conter os métodos referentes aos cenários de teste representados em cada diagrama;
- Em seguida analisou-se as informações presentes nos itens referentes a *Massa de dados* para identificar os dados necessários para realização dos testes e como prepará-los para serem utilizados nos *scripts*;
- A partir dos itens referentes aos *Cenários* são criados métodos de teste com os mesmos nomes mostrados nos diagramas;
- A partir dos itens referentes às *Pré-condições* são implementados nos métodos de teste as ações que devem ser realizadas antes dos passos de cada teste;
- A partir dos itens referentes aos *Passos* são implementadas nos métodos as ações que devem causar os resultados esperados na aplicação;
- Por fim, as asserções são criadas nos métodos de teste esperando como retorno da aplicação as mesmas informações presentes nos itens referentes ao *Resultado* de cada cenário de teste presente no diagrama.

Após criados os *scripts* de teste foi realizada a execução dos mesmos em um computador equipado com o Sistema Operacional Windows 10 Pro na versão 21H1, com processador Intel(R) Core(TM) i5-10210U de 1.60GHz, com 16 GB de memória principal e com uma unidade de estado sólido de 256 GB. Foram realizados 30 execuções sucessivas dos *scripts* de teste criados e coletados os dados de tempo de execução para calcular-se o tempo médio gasto nessas execuções.

Os resultados obtidos após a realização dos métodos e experimentos acima descritos serão explorados no Capítulo 5.

5 RESULTADOS E DISCUSSÕES

Utilizando o método de representação de cenários de teste utilizando diagramas baseados em mapas mentais foram criados diagramas para representar cada um dos cenários de teste automatizados para as funcionalidades *Efetuar Login* (Figura 7), *Cadastro de Orçamentos* (Figura 8), *Listagem e Deleção de Orçamentos* (Figura 9) e *Edição de Orçamentos* (Figura 10).

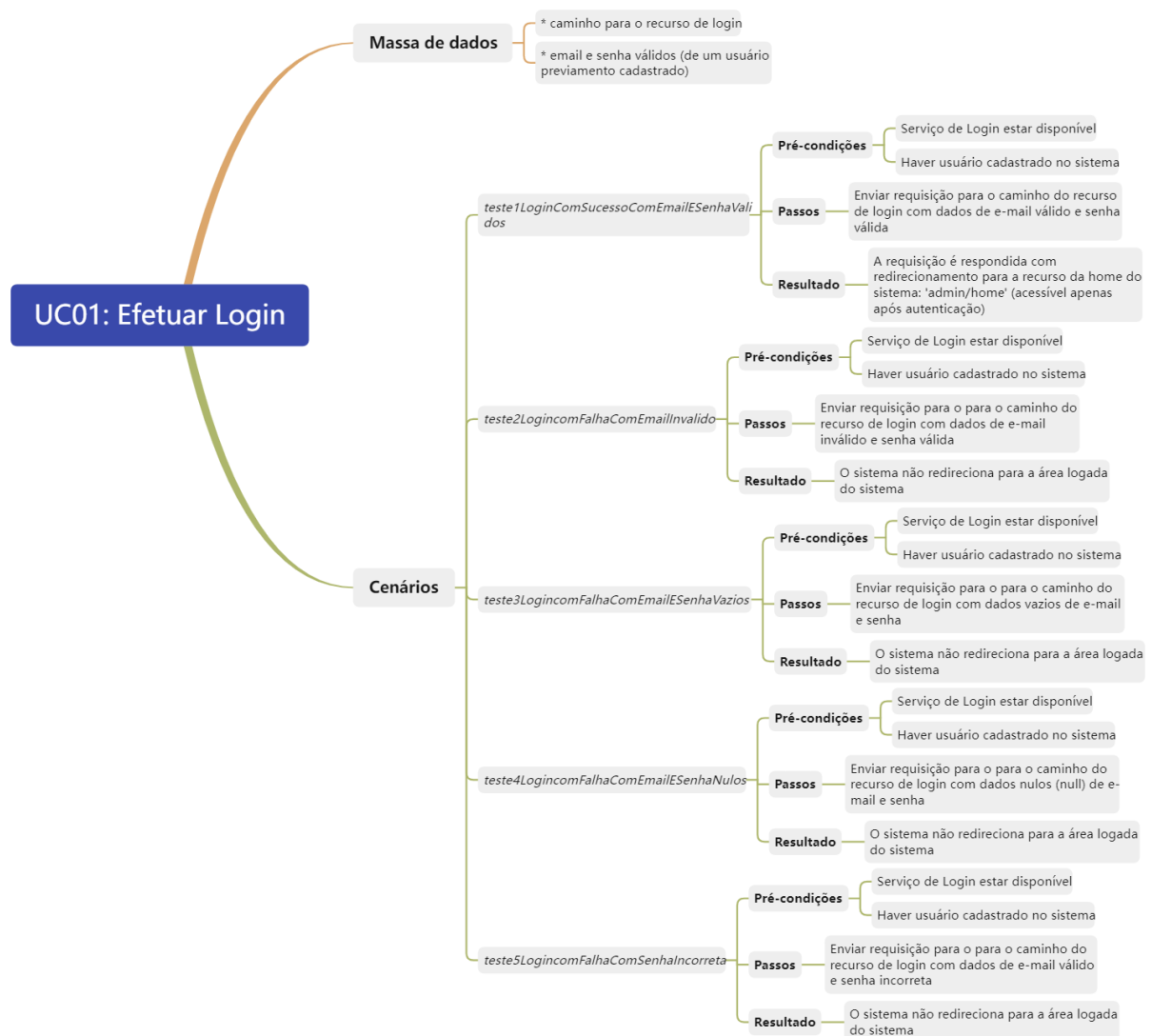


Figura 7 – Diagrama representando os cenários de teste da funcionalidade *Efetuar Login*

Fonte: o Autor

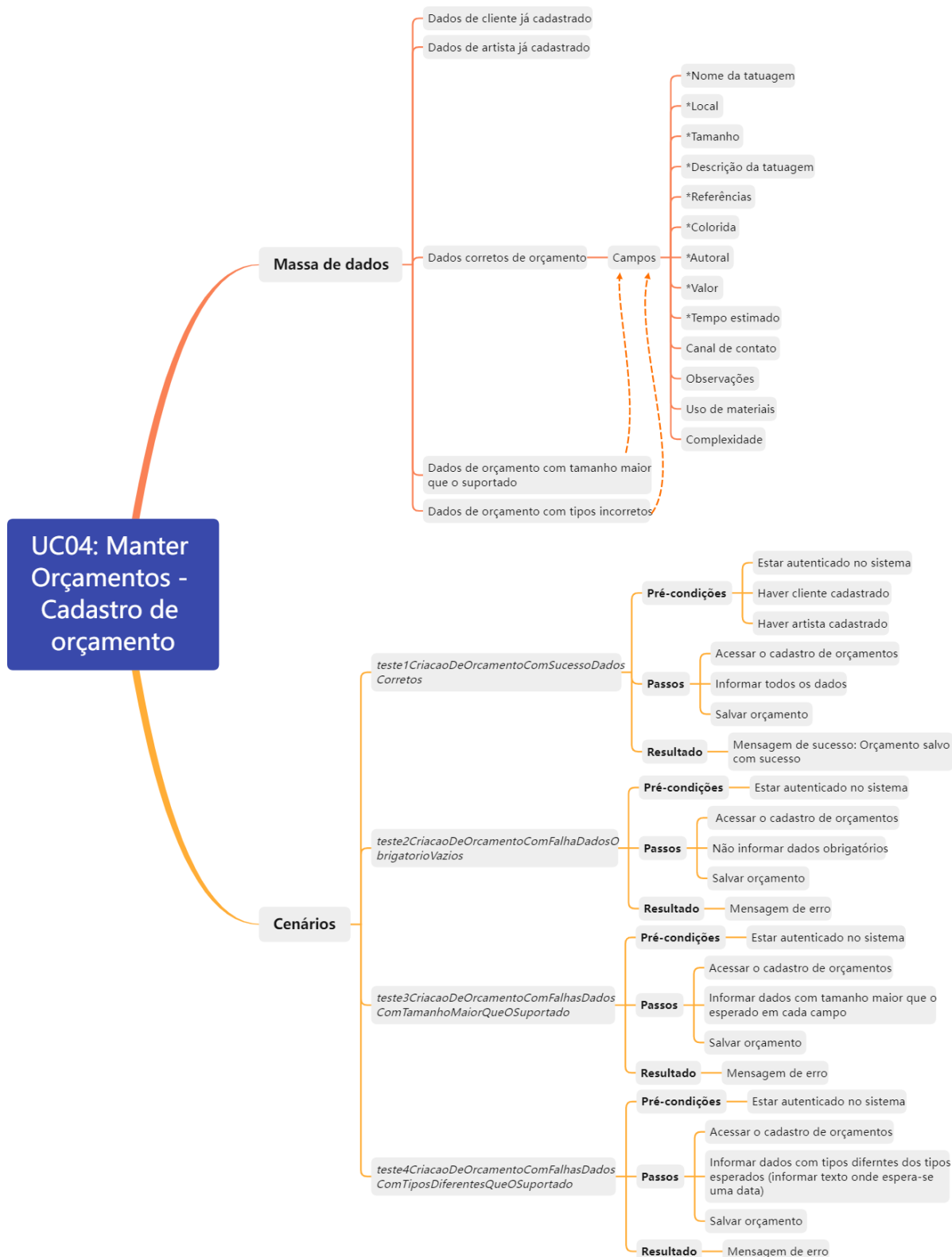


Figura 8 – Diagrama representando os cenários de teste da funcionalidade *Cadastro de Orçamentos*

Fonte: o Autor

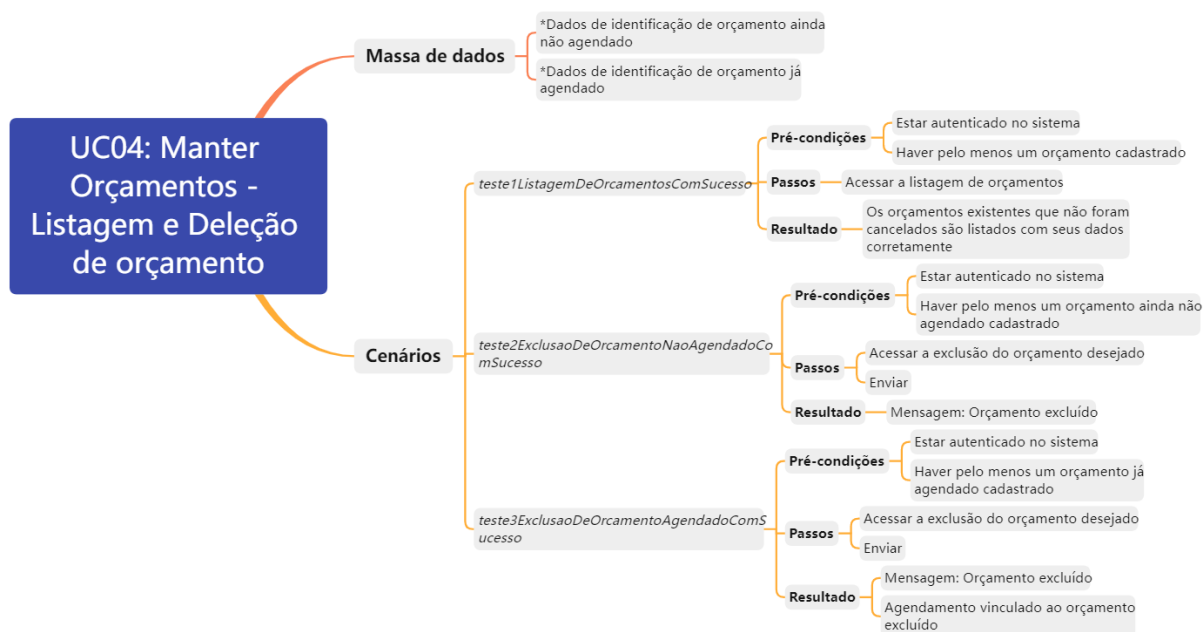


Figura 9 – Diagrama representando os cenários de teste das funcionalidades *Listagem e Deleção de Orçamentos*

Fonte: o Autor

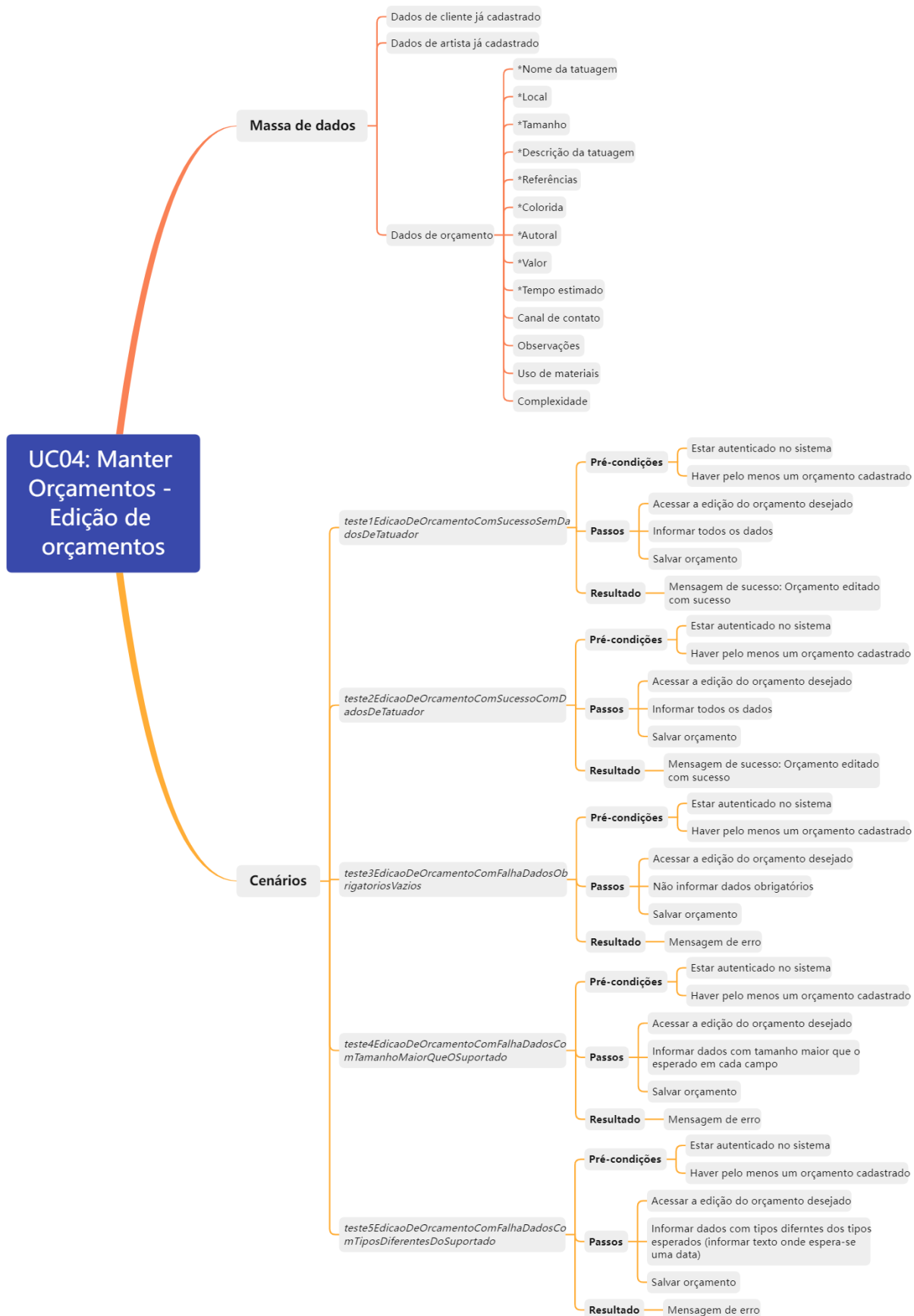


Figura 10 – Diagrama representando os cenários de teste das funcionalidades *Edição de Orçamentos*

Fonte: o Autor

Em seguida foram criados os *scripts* de teste de integração para os cenários de teste mapeados. O primeiro *script* criado, contém os testes do caso de uso *Efetuar Login* (Algoritmo 5.1):

```
1 <?php
2
3 namespace Tests\Integration\Login;
4
5 use Tests\TestCase;
6 use App\User;
7
8 class LoginTest extends TestCase
9 {
10
11     public function teste1LoginComSucessoComEmailESenhaValidos()
12     {
13         $user = factory(User::class)->create();
14         $response = $this->call('POST', '/login', ['email' => $user->
email, 'password' => 'teste@123']);
15         $response->assertStatus(302);
16         $response->assertRedirect('/admin/home');
17     }
18 }
```

Algoritmo 5.1 – Cenários de Testes Automatizados de Integração do Login

Fonte: o Autor

- O *script* de teste começa com a declaração da classe de teste e as importações utilizadas nos *scripts* de testes (linhas 1 a 9);
- Os métodos de teste são declarados, com o título iniciando sempre pela palavra teste, seguindo a mesma nomenclatura dada aos cenários nos diagramas de planejamento de testes (linha 11);
- Abaixo da declaração do método de teste, os objetos e variáveis utilizados no testes são declaradas e instanciadas para serem utilizados nas chamadas (linha 13);
- É feita a requisição para o recurso correspondente à funcionalidade testada com os dados de envio necessários, conforme a aplicação foi desenvolvida. A resposta da requisição é armazenada na variável *response* (linha 14);
- A asserção verifica o *status code* do retorno da requisição (linha 15) e o redirecionamento para a área do Sistema Admink que é acessível apenas após estar autenticado (linha 16).

O código realiza a requisição enviando os dados presentes no *script* de teste e por meio das asserções verifica o que foi retornado pelo sistema. Caso o retorno dado pelo sistema seja o esperado na asserção, o teste obtém sucesso, caso contrário o teste falha.

Os objetos instanciados a partir de classes do Sistema Admink foram criados dentro dos *scripts* de teste através das *factories* criadas durante o desenvolvimento do Sistema Admink. As *factories* permitem que os objetos sejam criados com dados aleatórios, possibilitando que diferentes execuções dos testes utilizem diferentes dados, além de permitirem que os comandos

para criação dos dados estejam dentro do *script* de teste, removendo a necessidade de que outros *scripts* ou instruções sejam executadas antes dos testes. O uso de *factories* também possibilitou a utilização de um banco de dados *SQLite* específico para os testes.

Comparando o *script* de teste do cenário *teste1LoginComSucessoComEmailESenhaValidos* com o diagrama de planejamento de testes é possível visualizar qual parte do *script* cada ramo representa (Figura 11).



Figura 11 – Comparação do diagrama com o script de teste

Fonte: o Autor

Para cada cenário do caso de uso *Efetuar Login* deve ser elaborado os *scripts* de teste seguindo uma estrutura semelhante ao primeiro cenário descrito, com as respectivas instâncias de dados e asserções que representam o cenário e sua regra de negócio, pois alterando-se os dados de entrada alteraram-se também o comportamento e os resultados esperados.

Os *scripts* de teste para os cenários de teste identificados para as funcionalidades do caso de uso *Manter Orçamentos* foram criados seguindo uma estrutura similar à dos testes do caso de uso *Efetuar Login*. O código completo com todos os *scripts* de teste criados pode ser encontrado no (Apêndice C).

Utilizando a estrutura definida pelo framework *Laravel*, foi realizada a configuração necessária para a execução dos *scripts* de teste nos arquivos de configuração do *PHPUnit* e de variáveis de ambiente.

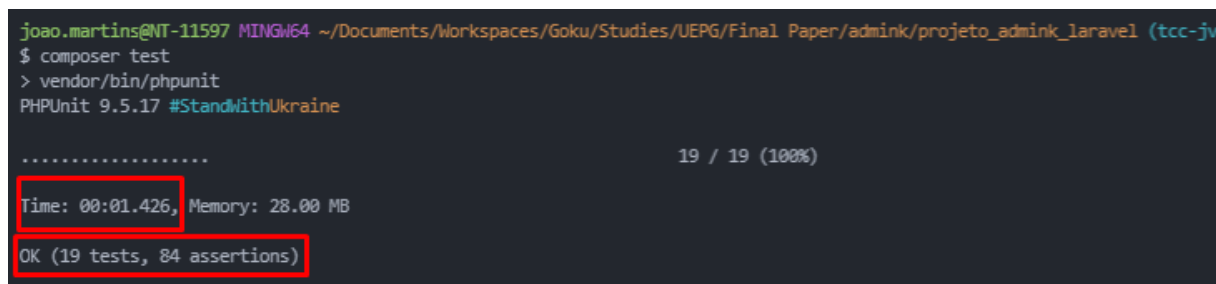
Em seguida os testes criados foram executados e os resultados analisados.

Todos os *scripts* criados foram armazenados no repositório privado do projeto *Admink* no *GitHub*, mas também foi criado um repositório público para armazenar e compartilhar o que foi desenvolvido neste trabalho ¹.

Após os testes criados terem sido executados, analisaram-se os resultados das execuções através dos relatório de execução fornecidos pelo *PHPUnit* após o término de cada execução (Figura 12). Foram realizadas 30 execuções sucessivas, e em todas elas os 19 testes e 84

¹<https://github.com/QAkarotto/tcc-software-2022>

asserções criadas nos *scripts* de teste obtiveram sucesso em suas execuções, não apresentando falhas. O tempo de execução em todas as execuções não ultrapassou 3 segundos e em média levou cerca de 1,6132 segundos (Tabela 1).



```
joao.martins@NT-11597 MINGW64 ~/Documents/Workspaces/Goku/Studies/UEPG/Final Paper/admink/projeto_admink_laravel (tcc-jv
$ composer test
> vendor/bin/phpunit
PHPUnit 9.5.17 #StandWithUkraine

..... 19 / 19 (100%)

Time: 00:01.426, Memory: 28.00 MB

OK (19 tests, 84 assertions)
```

Figura 12 – Relatório da execução dos testes automatizados destacando em vermelho o tempo de execução e a quantidade de testes e asserções realizadas com sucesso

Fonte: o Autor

Nº da Execução	Tempo de execução (segundos)
1ª execução	1.426
2ª execução	1.312
3ª execução	1.331
4ª execução	1.35
5ª execução	1.383
6ª execução	1.347
7ª execução	1.402
8ª execução	1.365
9ª execução	1.342
10ª execução	1.348
11ª execução	1.313
12ª execução	1.409
13ª execução	1.337
14ª execução	1.364
15ª execução	1.443
16ª execução	1.449
17ª execução	1.384
18ª execução	1.322
19ª execução	2.359
20ª execução	2.516
21ª execução	2.171
22ª execução	1.814
23ª execução	1.955
24ª execução	1.968
25ª execução	1.861
26ª execução	1.816
27ª execução	1.837
28ª execução	1.989
29ª execução	1.86
30ª execução	1.624
Média	1.613233333

Tabela 1 – Resultados das execuções *scripts* de testes.

Fonte: o Autor.

Foram inseridas alterações nos dados de envio de um dos testes para verificar o que é retornado no relatório de execução em caso de testes falharem. No relatório são exibidas informações indicando quantas falhas foram encontradas, em qual arquivo estão as asserções que falharam, quais asserções falharam e quais foram os valores obtidos comparados aos valores esperados nas asserções, evidenciando o motivo do teste falhar (Figura 13).

```
joao.martins@NT-11597 MINGW64 ~/Documents/Workspaces/Goku/Studies/UEPG/Final Paper/admink/projeto_admink_laravel (tcc-jv)
$ composer test
> vendor/bin/phpunit
PHPUnit 9.5.10 by Sebastian Bergmann and contributors.

.F..... 19 / 19 (100%)

Time: 00:01.314, Memory: 30.00 MB

There was 1 failure:

1) Tests\Integration\Login\LoginTest::testLoginComSucessoComEmailESenhaValidos
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'http://localhost:8000/admin/home'
+'http://localhost:8000'

C:\Users\joao.martins\Documents\Workspaces\Goku\Studies\UEPG\Final Paper\admink\projeto_admink_laravel\vendor\laravel\framework\src\Illuminate\Foundation\Testing\TestResponse.php:260
C:\Users\joao.martins\Documents\Workspaces\Goku\Studies\UEPG\Final Paper\admink\projeto_admink_laravel\vendor\laravel\framework\src\Illuminate\Foundation\Testing\TestResponse.php:285
C:\Users\joao.martins\Documents\Workspaces\Goku\Studies\UEPG\Final Paper\admink\projeto_admink_laravel\tests\Integration\Login\LoginTest.php:16

FAILURES!
Tests: 19, Assertions: 84, Failures: 1.
Script vendor/bin/phpunit handling the test event returned with error code 1
```

Figura 13 – Exemplo de um teste cuja asserção falhou

Fonte: o Autor

Com os *scripts* criados e suas execuções obtendo sucesso para todos os testes e asserções em poucos segundos, entende-se que a metodologia aplicada na implementação dos testes de integração automatizados foi bem-sucedida e as práticas utilizadas podem ser consideradas para a implementação de testes automatizados de integração.

Como recomendações propostas por este trabalho, para desempenhar atividades de teste de maneira eficiente deve-se:

- Realizar o planejamento dos testes, baseando-se na especificação dos requisitos do software e na aplicação de técnicas de caixa preta, visando identificar os cenários de teste que serão automatizados e quais são os conjuntos de dados que devem ser utilizados nos testes. Caso os *scripts* de teste sejam definidos após a criação do código fonte do software testado, o próprio código fonte somado à aplicação de técnicas de caixa branca pode ser utilizado para auxiliar na identificação dos cenários de teste. Recomenda-se também o uso do método para construção de diagramas de planejamento de testes baseados em mapas mentais como forma de representação dos cenários de teste, evitando a criação de documentos extensos e detalhados;
- Consultar a documentação da linguagem de programação ou *framework* utilizado no desenvolvimento do software que será testado, e seguir as recomendações dadas pela documentação a respeito dos *scripts* de teste e configurações necessárias para criação e execução dos mesmos;
- Nomear os métodos de teste de acordo com as recomendações da documentação da ferramenta, além de usar a mesma nomenclatura utilizada nos nomes dos cenários de teste identificados. Os nomes devem transmitir claramente o que está sendo verificado pelo método de teste. Desta forma, apesar da nomenclatura dos métodos se tornar verbosa, ela fornece sentido semântico mais rico, e possibilita que os próprios *scripts* de teste sejam parte da documentação do software;
- Evitar criar dependências entre diferentes métodos de teste, possibilitando que os métodos possam ser executados em qualquer ordem sem que os testes falhem. Ou seja, os dados

e pré-condições necessárias para a execução de um método de teste devem ser providos pelo próprio método, ou então, por um método auxiliar de configuração que é executado antes de todo método de teste ser executado;

- Utilizar bancos de dados criados especificamente para os testes, evitando que possua dados que não foram inseridos pelos testes ou como preparação para a execução dos testes. Com o apoio de *migrations*² e *factories*, um banco de dados pode ser facilmente criado e populado a cada execução dos testes. Caso não seja possível criar um banco de dados específico para execução dos testes, podem ser utilizados *mock objects*. Entretanto, dessa última forma, não haverá comunicação com um banco de dados real;
- Utilizar asserções que obtenham sucesso apenas quando o comportamento esperado realmente foi apresentado pela aplicação. Ou seja, as asserções devem ser precisas e devem evidenciar que o resultado esperado foi obtido.
- Armazenar o código criado em um repositório;

As recomendações propostas colaboram para que os *scripts* de teste atendam aos princípios *FISRT* e não possuam *test smells* (Capítulo 3) e podem ser inseridas no fluxo de times que utilizam o XP como método ágil, podendo ser realizadas durante a prática do Desenvolvimento Guiado por Testes. Mas também podem ser utilizadas por outros times que também desejem implementar testes automatizados sem ter que criar documentos extensos e detalhados.

Durante a implementação dos testes algumas dificuldades foram encontradas para realizar as asserções devido a forma que a aplicação foi desenvolvida. Nas respostas das requisições realizadas pelos *scripts* de teste sempre é retornado um documento *HTML* como resposta. Dessa forma as asserções devem ser realizadas comparando o conteúdo de campos do *HTML* retornado com os valores esperados. Foi necessário analisar o *HTML* retornado para identificar quais campos continham os dados esperados. Por exemplo no *script* do teste *teste1ListagemDeOrcamentosComSucesso* (Algoritmo 5.2):

- Nas linhas 18 a 22 é possível visualizar que as asserções estão verificando se o documento *HTML* contem os elementos esperados.

```
1 public function teste1ListagemDeOrcamentosComSucesso()  
2 {  
3     $user = factory(User::class)->create();  
4     $estudio = factory(Estudio::class)->create();  
5     factory(EstudioUsers::class)->create(['fk_users_id_users' =>  
6     $user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);  
7     $artista = factory(Artista::class)->create();  
8     $cliente = factory(Cliente::class)->create();  
9     factory(ArtistaEstudio::class)->create(['fk_artista_id_artista'  
10    => $artista->id_artista, 'fk_estudio_id_estudio' => $estudio->
```

²permite que o esquema (*schema*) do banco de dados seja definido e possa ser compartilhado, bastando executar a *migration* para atualizar o esquema, além de serem agnósticas de banco de dados, possibilitando a utilização do mesmo código de *migrations* em qualquer sistema gerenciador de banco de dados utilizado

```

    id_estudio]);
9      factory(ClienteEstudio::class)->create(['fk_cliente_id_cliente'
=> $cliente->id_cliente, 'fk_estudio_id_estudio' => $estudio->
id_estudio]);
10     $orcamento = factory(Orcamento::class)->create(['
fk_cliente_id_cliente' => $cliente->id_cliente, '
fk_artista_id_artista' => $artista->id_artista, '
fk_estudio_id_estudio' => $estudio->id_estudio, '
fk_orcamento_status_id_orcamento_status' => 1]);
11     $orcamento2 = factory(Orcamento::class)->create(['
fk_cliente_id_cliente' => $cliente->id_cliente, '
fk_artista_id_artista' => $artista->id_artista, '
fk_estudio_id_estudio' => $estudio->id_estudio, '
fk_orcamento_status_id_orcamento_status' => 1]);
12
13     $response = $this->actingAs($user)
14         ->get('/admin/orcamentos');
15
16     $response->assertStatus(200);
17     $response->assertSee('<td class="align-middle">' . $orcamento->
tatuagem_nome . '</td>', $escaped = false);
18     $response->assertSee('<td class="align-middle">#' . str_pad(
$orcamento->id_orcamento, 5, '0', STR_PAD_LEFT) . '</td>', $escaped =
false);
19     $response->assertSee('<td class="align-middle">' . $orcamento2->
tatuagem_nome . '</td>', $escaped = false);
20     $response->assertSee('<td class="align-middle">#' . str_pad(
$orcamento2->id_orcamento, 5, '0', STR_PAD_LEFT) . '</td>', $escaped
= false);
21 }

```

Algoritmo 5.2 – Script do teste teste1ListagemDeOrcamentosComSucesso

Fonte: o Autor

Caso o Sistema Admink tivesse sido desenvolvido para responder as requisições retornando os dados em um documento *JSON* ao invés de diretamente no *HTML* da interface gráfica, a identificação dos campos contendo os dados esperados seria facilitada, assim como a criação das asserções, pois seria necessário verificar apenas os dados do *JSON* e não todo o *HTML*. nas linhas 8 a 10 do *script* de exemplo extraído da documentação do *Framework Laravel* é possível ver como são realizadas asserções para campos de documentos *JSON* (Algoritmo 5.3).

Fonte: o Autor

Também foram encontradas dificuldades com asserções utilizando os *status codes* retornados nas respostas das requisições. A maioria das requisições realizadas para o Sistema Admink retornam o mesmo *status code* tanto quando se obtém sucesso como quando falha.

```

1 public function test_making_an_api_request()
2     {
3         $response = $this->postJson('/api/user', ['name' => 'Sally']);
4
5         $response
6             ->assertStatus(201)
7             ->assertJson([
8                 'created' => true,
9             ]);
10    }

```

Algoritmo 5.3 – Script de Teste exemplificando asserções baseadas em campos de documentos *JSON*

Para diferenciar sucessos de erros foi necessário analisar as mensagens de sucesso e de erro retornadas nas respostas das requisições, além de utilizar tais mensagens na criação das asserções. Por exemplo nos *scripts* *teste1CriacaoDeOrcamentoComSucessoDadosCorretos* e *teste2CriacaoDeOrcamentoComFalhaDadosObrigatorioVazios* (Algoritmo 5.4):

```

1 public function teste1CriacaoDeOrcamentoComSucessoDadosCorretos()
2     {
3         $user = factory(User::class)->create();
4         $estudio = factory(Estudio::class)->create();
5         factory(EstudioUsers::class)->create(['fk_users_id_users' =>
6             $user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
7         $artista = factory(Artista::class)->create();
8         $cliente = factory(Cliente::class)->create();
9         factory(ArtistaEstudio::class)->create(['fk_artista_id_artista'
10             => $artista->id_artista, 'fk_estudio_id_estudio' => $estudio->
11             id_estudio]);
12         factory(ClienteEstudio::class)->create(['fk_cliente_id_cliente'
13             => $cliente->id_cliente, 'fk_estudio_id_estudio' => $estudio->
14             id_estudio]);
15
16         $response = $this->actingAs($user)
17             ->post('/admin/orçamentos', [
18                 'cliente' => $cliente->id_cliente,
19                 'artista' => $artista->id_artista,
20                 'tatuagem_nome' => 'Teste Orçamento',
21                 'tatuagem_local' => 'Teste',
22                 'tatuagem_comprimento' => 10,
23                 'tatuagem_largura' => 10,
24                 'tatuagem_descricao' => 'Teste Orçamento Criado com
25                 Sucesso!',
26                 'tatuagem_referencias' => null,
27                 'canal_contato' => null,
28                 'tempo_estimado' => null,
29                 'valor' => null,

```

```

25         'uso_materiais'           => null,
26         'complexidade'           => null,
27         'observacao'             => null
28     ];
29     $response->assertStatus(302);
30     $response->assertSessionHas('success_toastr', '0 orçamento foi
cadastrado com sucesso!');
31 }
32
33 public function
teste2CriacaoDeOrcamentoComFalhaDadosObrigatorioVazios()
34 {
35     $user = factory(User::class)->create();
36     $estudio = factory(Estudio::class)->create();
37     factory(EstudioUsers::class)->create(['fk_users_id_users' =>
$user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
38
39     $response = $this->actingAs($user)
40         ->post('/admin/orcamentos', [
41         'cliente'                 => null,
42         'artista'                 => null,
43         'tatuagem_nome'           => '',
44         'tatuagem_local'          => '',
45         'tatuagem_comprimento'    => null,
46         'tatuagem_largura'        => null,
47         'tatuagem_descricao'      => '',
48         'tatuagem_referencias'    => null,
49         'canal_contato'           => null,
50         'tempo_estimado'          => null,
51         'valor'                   => null,
52         'uso_materiais'           => null,
53         'complexidade'           => null,
54         'observacao'             => null
55     ]);
56
57     $response->assertStatus(302);
58     $this->assertEquals(session('errors')->get('cliente')[0], '0
campo cliente é obrigatório.');
```

```

59     $this->assertEquals(session('errors')->get('artista')[0], '0
campo artista é obrigatório.');
```

```

60     $this->assertEquals(session('errors')->get('tatuagem_nome')[0],
'0 campo tatuagem nome é obrigatório.');
```

```

61     $this->assertEquals(session('errors')->get('tatuagem_local')[0],
'0 campo tatuagem local é obrigatório.');
```

```

62     $this->assertEquals(session('errors')->get('tatuagem_comprimento
')[0], '0 campo tatuagem comprimento é obrigatório.');
```

```

63     $this->assertEquals(session('errors')->get('tatuagem_largura'))

```

```
64     [0], '0 campo tatuagem largura é obrigatório.');
```

```
65     $this->assertEquals(session('errors')->get('tatuagem_descricao'))
```

```
66     [0], '0 campo tatuagem descricao é obrigatório.');
```

```
67 }
```

Algoritmo 5.4 – Scripts de teste de criação de orçamentos

Fonte: o Autor

- Na linha 30 é possível ver que a asserção está sendo realizada para o *status code* 302;
- Na linha 31 é realizada uma asserção para o texto da mensagem de sucesso na criação de orçamento;
- Na linha 58 é realizada asserção também para o *status code* 302, entretanto nesse *script* de teste, obtém-se falha na criação de orçamentos.
- Nas linhas 59 a 66 são realizadas asserções para cada uma das mensagens de erros retornadas para cada um dos campos enviados com valores não aceitos. Como o *status code* 302 foi retornado tanto quando se obteve sucesso como quando houve erro na criação de orçamento, essas asserções foram necessárias para diferenciar respostas de sucesso das de erro.

Caso o Sistema Admink tivesse sido desenvolvido para responder às requisições retornando *status codes* apropriados dependendo do resultado do processamento de cada requisição (por exemplo 201 para sucesso na criação e 400 para dados com valores não aceitos), o *status code* já seria suficiente para diferenciar retornos de sucesso dos de erro, podendo ser usado nas asserções ao invés de apenas o *status code* 302.

A implementação e execução dos testes automatizados bem-sucedidas dependem de um bom planejamento de testes, além da escrita de *scripts* de testes automatizados utilizando o *framework* apropriado. Entretanto o código do software testado e sua testabilidade³ também impactam a criação e execução dos testes automatizados.

³característica que diz respeito a facilidade de se testar um software

6 CONCLUSÕES

O trabalho apresentou um estudo de caso da implementação de testes funcionais de integração automatizados em uma aplicação web, a partir do processo de planejamento dos testes, do processo de implementação dos *scripts* e do conjunto de recomendações propostas e sua aplicação em um projeto real.

Utilizando a documentação de casos de uso existente e a aplicação de técnicas de teste e o método de representação de cenários de teste utilizando diagramas baseados em mapas mentais, os testes foram planejados e documentados nos diagramas. Através da realização dessas atividades, percebeu-se os benefícios para a identificação dos cenários de teste que devem ser automatizados, além da representação dos mesmos em uma forma de documentação pouco extensa, mas que transmite as informações necessárias.

Baseado nos diagramas produzidos e utilizando as informações por eles fornecidas, foi realizada a implementação dos *scripts* de teste de integração automatizados. Os diagramas auxiliaram na construção dos *scripts*, possibilitando identificar quais informações presentes no diagrama deveriam compor cada parte dos *scripts*, além da quantidade de cenários ser a mesma da quantidade de métodos de teste criados.

A execução dos testes automatizados mostrou-se rápida, levando poucos segundos para executar uma grande quantidade de testes e asserções, mostrando os testes automatizados como bons candidatos para serem utilizados em testes de regressão que executam os mesmos cenários de teste frequentemente.

O conjunto de recomendações produzido mostrou-se benéfico, auxiliando nas atividades de implementação e execução dos testes automatizados, podendo ser aplicado em novos projetos.

Algumas dificuldades para identificação de como realizar as asserções foram encontradas durante a implementação dos *scripts*. Essas dificuldades foram ocasionadas principalmente pela forma que o sistema foi implementado sem considerar a testabilidade do software durante o desenvolvimento.

Portanto é possível concluir que a implementação de testes automatizados é um processo capaz de trazer benefícios para as atividades de teste de um projeto, através da execução rápida, possibilitando execuções frequentes. Entretanto é um processo que demanda conhecimentos técnicos de níveis, tipos e planejamento de testes, de codificação utilizando a linguagem de programação do software e das ferramentas de automação de testes adequadas para os tipos e níveis dos testes realizados. Além dos conhecimentos, devem ser empregadas as recomendações das documentações das ferramentas para automação de testes. Por fim, as recomendações propostas por esse trabalho têm potencial para auxiliar na realização desse processo.

Como trabalhos futuros, existe a possibilidade de estender o processo realizado na criação de cenários de testes de integração automatizados para outros casos de uso do sistema.

Além disso, é possível estender a aplicação dos testes automatizados para outros níveis e tipos de testes. Também é possível estudar testabilidade de software e sua relação com padrões e boas práticas de desenvolvimento.

Referências

- BECK, K. *Extreme programming: A humanistic discipline of software development*. Springer Berlin Heidelberg, Berlin, Heidelberg, p. 1–6, 1998. Citado na página 8.
- BECK, K.; GAMMA, E. Test infected: Programmers love writing tests. **Java Report**, Citeseer, v. 3, n. 7, p. 37–50, 1998. Citado na página 2.
- BERNARDO, P. C.; KON, F. A importância dos testes automatizados. *Engenharia de Software Magazine*, 2008. Citado na página 12.
- BUZAN, T. **Mapas mentais**. 1. ed. [S.l.]: Editora Sextante, 2009. ISBN 9788575424933. Citado na página 21.
- CRISPIN, J. G. L. **Agile Testing**. Addison Wesley, 2009. ISBN 9780321534460. Disponível em: <https://www.ebook.de/de/product/7779433/lisa_crispin_janet_gregory_agile_testing.html>. Citado na página 1.
- GRAHAM, D. S. T. C. et al. **Foundations of Software Testing**. Cengage Learning EMEA, 2019. ISBN 1473764793. Disponível em: <https://www.ebook.de/de/product/37327358/dorothy_software_testing_consultant_graham_rex_president_rex_black_consulting_services_rbcs_inc_black_erik_improve_quality_services_b_v_van_veenendaal_foundations_of_software_testing.html>. Citado 4 vezes nas páginas 12, 13, 16 e 17.
- LARAVEL. **Laravel Docs**. [S.l.], 2021. Disponível em: <<https://laravel.com/docs>>. Acesso em: 20 de setembro de 2021. Citado 2 vezes nas páginas 17 e 20.
- MARTIN, R. C. **Desenvolvimento ágil limpo: de volta às origens**. Rio de Janeiro, RJ: [s.n.], 2020. ISBN 9788550815008. Citado 3 vezes nas páginas 8, 10 e 11.
- MYERS, G. **The art of software testing**. Hoboken, N.J: John Wiley & Sons, 2012. ISBN 1118031962. Citado na página 12.
- PHP. **Manual do PHP**. [S.l.], 2021. Disponível em: <https://www.php.net/manual/pt_BR/preface.php>. Acesso em: 20 de setembro de 2021. Citado na página 20.
- PHP.COM.BR. **Testes unitários com a PHPUnit**. [S.l.], 2019. Disponível em: <<https://php.com.br/24?testes-unitarios-com-a-phpunit>>. Acesso em: 08 de março de 2022. Citado na página 21.
- PRESSMAN, R. **Software engineering : a practitioner's approach**. New York, NY: McGraw-Hill Education, 2015. ISBN 0078022126. Citado 6 vezes nas páginas 4, 5, 6, 7, 8 e 13.
- SOMMERVILLE, I. **Engenharia de software**. São Paulo: Pearson Prentice Hall, 2019. ISBN 8543024978. Citado 2 vezes nas páginas 13 e 14.
- VALENTE, M. T. **Engenharia de Software Moderna**. 1. ed. [S.l.]: Amazon, 2020. ISBN 6500019504. Citado 8 vezes nas páginas 1, 4, 6, 7, 8, 13, 14 e 15.
- XMIND. **XMind - Mind Mapping Software**. [S.l.], 2022. Disponível em: <<https://www.xmind.net/>>. Acesso em: 08 de março de 2022. Citado na página 21.

Apêndices

APÊNDICE A – Descrição dos casos de uso do Sistema Admink

UC01: Efetuar Login		
Objetivo	Efetuar login no sistema.	
Atores	Usuário.	
Pré-condições	Usuário ainda não ter feito login no sistema. Usuário deve ter um cadastro no sistema.	
Pós-condições	Usuário estar logado, com acesso ao sistema e todas as suas funcionalidades.	
Fluxo Principal	Ator	Sistema
	1. Usuário digita email e senha no formulário depois clica no botão para enviar as informações de login.	2. Valida as informações recebidas pelo formulário de login.
Fluxos Alternativos	Ocorrência	Tratativa
	2.a As informações recebidas não são válidas.	2.a Sistema informa que email ou senha estão inválidos e foca no campo de digitação de email.

Quadro 1 – UC01: Efetuar Login

Fonte: o Autor

UC04: Manter Orçamentos			
Objetivo		Realizar Cadastro, Consulta, Edição e Exclusão de orçamentos no sistema.	
Atores		Usuário.	
Pré-condições		Usuário estar logado no sistema.	
Pós-condições		Dados dos orçamentos mantidos no sistema.	
Cadastro	Fluxo Principal	Ator	Sistema
		1.Usuário seleciona no formulário de cadastro o Cliente , Artista tatuador, informa o Nome da tatuagem , Local , Tamanho , Descrição da tatuagem, Referências , se é Colorida ou não, se é Autoral ou não, Valor , Tempo estimado do procedimento, Canal de contato usado pelo cliente, Observações, Uso de materiais e Complexidade, e Aceite do termo de responsabilidade, depois clica no botão de 'Salvar' do formulário.	2. Valida os dados recebidos pelo formulário de cadastro e exibe mensagem de sucesso
	Fluxos Alternativos	Ocorrência	Tratativa
		1.a. Usuário clica em cancelar ou clica fora do modal.	1.a. Sistema exibe a tela por onde a funcionalidade foi acessada.
		1.b. Usuário clica para adicionar Cliente.	1.b. Exibir tela de cadastro de cliente.
		1.c. Usuário clica para adicionar Artista.	1.c. Exibir tela de cadastro de artista.
		1.d. Usuário clica em agendar.	1.d. Sistema salva os dados do orçamento e exibe tela de cadastro de agendamento com o orçamento recém inserido já selecionado
		2.a. Dados inseridos são inválidos.	2.a. Sistema informa que os dados são inválidos e foca no campo do formulário a ser corrigido.
		Ator	Sistema
	Fluxo Principal	1.Usuário insere id do orçamento, nome ou apelido do cliente, nome ou apelido do artista, nome da tatuagem, data ou status do orçamento no campo de pesquisa e clica para realizar a busca	2. Retorna todos os registros correspondentes ao id do orçamento, nome ou apelido do cliente, nome ou apelido do artista, nome da tatuagem, data ou status do orçamento recebido pelo campo de pesquisa.
		Ocorrência	Tratativa
	Fluxos Alternativos	2.a. Não existem registros na base correspondentes a busca realizada	2.a. Sistema informa que não há registros cadastrados correspondentes à busca.
		Ator	Sistema
Edição	Fluxo Principal	1.Usuário faz uma consulta do orçamento desejado, o seleciona e clica no botão de editar.	2. Exibe formulário com os dados cadastrados do orçamento e permite a edição dos campos pelo usuário.
		3. Usuário edita as informações desejadas e clica no botão de salvar.	4. Valida as informações recebidas pelo formulário de edição, atualiza a base e exibe mensagem de sucesso.
	Fluxos Alternativos	Ocorrência	Tratativa
		1.a. Usuário clica em cancelar ou clica fora do modal.	1.a. Sistema exibe a tela por onde a funcionalidade foi acessada.
		4.a. Dados inseridos são inválidos.	4.a. Sistema informa que os dados são inválidos e foca no campo do formulário a ser corrigido.
		Ator	Sistema
Exclusão	Fluxo Principal	1.Usuário faz uma consulta do orçamento desejado, o seleciona e clica no botão de excluir.	2. Exibe um alerta para confirmar a exclusão.
		3. Usuário clica no botão de confirmar.	4. Atualiza a base com a exclusão do registro e exibe uma mensagem de sucesso.
	Fluxos Alternativos	Ocorrência	Tratativa
		1.a. Usuário clica em cancelar ou clica fora do modal.	1.a. Sistema exibe a tela por onde a funcionalidade foi acessada.
		2.a. Usuário clica no botão de 'Cancelar' exclusão.	2.a. Sistema retorna à tela de consulta de orçamentos.

Quadro 2 – UC04: Manter Orçamentos

Fonte: o Autor

APÊNDICE B – Plano de Testes do Sistema Admink

Caso de Teste: CT01 - Validação padrão		
Objetivo do Teste	Validação padrão de todas as telas. Validação padrão dos CRUDs que não tenham dependência de dados cadastrados previamente (artistas, clientes, estação de trabalho). Validação padrão dos relatórios e indicadores	
Pré-condições	Acesso ao sistema. Login no sistema.	
Fluxo Principal	Ação	Resultado esperado
	1. Tipos de dados: Informar tipo de dado diferente do esperado para o campo ou entradas que não atendem à validação do campo (ex: cpf inválido)	Não deve permitir.
	2. Campos obrigatórios: Não informar dados em todos os campos obrigatórios e confirmar.	Não deve permitir.
	3. Estouro de campos: Inserir mais dados que o suportado pelos campos do banco.	Não deve permitir.
	4. Navegação e persistência: Telas e dados exibidos após acessar, voltar, cancelar, informar dados, confirmar, salvar, deletar, realizar buscas ou utilizar filtros e ordenações, paginação e quantidade de itens listados,	<ul style="list-style-type: none"> - Acessar: deve exibir a tela da funcionalidade acessada. - Voltar ou cancelar: deve exibir a tela anterior ou a tela de listagem dos dados e os dados não devem ser salvos. - Confirmar ou salvar: deve exibir a tela seguinte no fluxo ou a tela de listagem dos dados e os dados devem ser salvos e exibidos ao serem consultados. - Deletar: deve exibir mensagem de confirmação. Após confirmação deve exibir a tela por onde a funcionalidade foi acessada e os dados excluídos não devem ser exibidos. - Realizar buscas ou utilizar filtros: os dados exibidos devem ser correspondentes a entrada informada na busca e aos filtros e ordenações selecionadas. - A quantidade de itens listados deve obedecer às opções de paginação e quantidade de itens selecionada.
	5. Usabilidade: Identificação de botões de ação, identificação de campos obrigatórios, identificação das ações dos ícones	<ul style="list-style-type: none"> - Botões de ação devem ter o mesmo padrão em todas as telas e serem diferenciáveis dos demais - Campos obrigatórios devem estar sinalizados - Os ícones devem transmitir claramente a ação que estão associados e parar cursor do mouse em cima deve exibir descrição
	6. Compatibilidade: Utilizar interface em resoluções desktop e mobile	Os dados devem se adaptar sem distorcer o conteúdo

Quadro 1 – TC01: Validação Padrão

Fonte: o Autor

Caso de Teste: CT02 - Login		
Objetivo do Teste	Validar a realização de login no sistema.	
Pré-condições	Usuário com e-mail e senha cadastradas no sistema.	
Fluxo Principal	Ação	Resultado esperado
	1. Acesar a tela de login e inserir e-mail e senha corretos de um usuário cadastrado no sistema e clicar para enviar as informações de login.	Deve permitir acesso ao sistema.
	2. Acessar menus e telas do sistema.	Deve permitir acessar a todas as telas e funcionalidades disponíveis para o nível de acesso do usuário (com apenas um nível de acesso todas as telas e funcionalidades devem estar acessíveis).
	3. Informações do usuário exibidas na tela.	Deve exibir o nome do usuário logado conforme cadastrada no sistema.
	4. Acessar opção de sair do sistema.	Deve exibir a tela de login do sistema.
Fluxos alternativos	1.a - Inserir informações de login inválidas: email não cadastrado no banco, senha não correspondente a cadastrada para o e-mail informado, e-mail e senha correspondentes a cadastrada mas com todos os caracteres em maiúsculo, e-mail e senha correspondentes a cadastrada mas com todos os caracteres em minúsculo.	Não deve permitir.

Quadro 2 – TC02: Login

Fonte: o Autor

Caso de Teste: CT04 - Orçamento		
Objetivo do Teste	Validar a realização de orçamento e funcionalidades associadas no sistema.	
Pré-condições	Estar logado no sistema. Artista vinculado ao estúdio cadastrado no sistema. Cliente cadastrado no sistema.	
Fluxo Principal	Ação	Resultado esperado
	1. Acesar a tela de orçamentos, clicar em 'Novo Orçamento'	Deve exibir o modal de cadastro de orçamento.
	2. Informar valores para todos os campos obrigatórios e clicar para enviar os dados.	Deve exibir mensagem de sucesso. Orçamento deve ser cadastrado com sucesso.
	3. Localizar orçamento na listagem de orçamentos e visualizar detalhes do orçamento.	Os dados exibidos devem ser os mesmos inseridos na ação "2". Status do orçamento deve ser 'Pendente'.
	4. Editar Orçamento e repetir a ação "3"	Os dados exibidos devem ser os mesmos inseridos na edição.
	5. Editar Status do orçamento e repetir a ação "3"	O status exibido deve ser o mesmo inserido na ação "5".
	6. Excluir Orçamento.	Os dados do orçamento excluído não devem ser exibidos.
Fluxos alternativos	2.a. Clicar para adicionar Cliente	Exibir tela de cadastro de cliente.
	2.b. Clicar para adicionar Artista	Exibir tela de cadastro de artista.
	2.c. Clicar em Agendar.	Deve exibir o modal de cadastro de agendamento com o orçamento recém inserido já selecionado. Após inserido agendamento deve alterar status do orçamento para 'Finalizado'.

Quadro 3 – TC04: Orçamento

Fonte: o Autor

APÊNDICE C – Scripts de Teste de Integração do Sistema Admink

```

1
2 <?php
3
4 namespace Tests\Integration\Login;
5
6 use Tests\TestCase;
7 use App\User;
8
9 class LoginTest extends TestCase
10 {
11
12     public function teste1LoginComSucessoComEmailESenhaValidos()
13     {
14         $user = factory(User::class)->create();
15         $response = $this->call('POST', '/login', ['email' => $user->
email, 'password' => 'teste@123']);
16         $response->assertStatus(302);
17         $response->assertRedirect('/admin/home');
18     }
19
20     public function teste2LogincomFalhaComEmailInvalido()
21     {
22         $user = factory(User::class)->create();
23         $response = $this->call('POST', '/login', ['email' => '
invalido@email.com', 'password' => 'teste@123']);
24         $response->assertStatus(302);
25         $response->assertRedirect('/');
26     }
27
28     public function teste3LogincomFalhaComEmailESenhaVazios()
29     {
30         $user = factory(User::class)->create();
31         $response = $this->call('POST', '/login', ['email' => '', '
password' => '']);
32         $response->assertStatus(302);
33         $response->assertRedirect('/');
34     }
35     public function teste4LogincomFalhaComEmailESenhaNulos()
36     {
37         $user = factory(User::class)->create();
38         $response = $this->call('POST', '/login', ['email' => null, '
password' => null]);
39         $response->assertStatus(302);

```

```

40     $response->assertRedirect('/');
41 }
42
43 public function teste5LogincomFalhaComSenhaIncorreta()
44 {
45     $user = factory(User::class)->create();
46     $response = $this->call('POST', '/login', ['email' => $user->
email, 'password' => 'wrongpassowrd']);
47     $response->assertStatus(302);
48     $response->assertRedirect('/');
49 }
50 }

```

Algoritmo C.1 – Scripts de teste de Login

Fonte: o Autor

```

1 <?php
2
3 namespace Tests\Integration\Orcamentos;
4
5 use Tests\TestCase;
6 use App\User;
7 use App\EstudioUsers;
8 use App\Estudio;
9 use App\Cliente;
10 use App\Artista;
11 use App\ArtistaEstudio;
12 use App\ClienteEstudio;
13
14
15 class CriacaoDeOrcamentoTest extends TestCase
16 {
17     public function teste1CriacaoDeOrcamentoComSucessoDadosCorretos()
18     {
19         $user = factory(User::class)->create();
20         $estudio = factory(Estudio::class)->create();
21         factory(EstudioUsers::class)->create(['fk_users_id_users' =>
$user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
22         $artista = factory(Artista::class)->create();
23         $cliente = factory(Cliente::class)->create();
24         factory(ArtistaEstudio::class)->create(['fk_artista_id_artista'
=> $artista->id_artista, 'fk_estudio_id_estudio' => $estudio->
id_estudio]);
25         factory(ClienteEstudio::class)->create(['fk_cliente_id_cliente'
=> $cliente->id_cliente, 'fk_estudio_id_estudio' => $estudio->
id_estudio]);
26

```



```

27
28     $response = $this->actingAs($user)
29         ->post('/admin/orcamentos', [
30             'cliente'           => $cliente->id_cliente,
31             'artista'           => $artista->id_artista,
32             'tatuagem_nome'     => 'Teste Orçamento',
33             'tatuagem_local'    => 'Teste',
34             'tatuagem_comprimento' => 10,
35             'tatuagem_largura'   => 10,
36             'tatuagem_descricao' => 'Teste Orçamento Criado com
Sucesso!',
37             'tatuagem_referencias' => null,
38             'canal_contato'      => null,
39             'tempo_estimado'     => null,
40             'valor'              => null,
41             'uso_materiais'      => null,
42             'complexidade'       => null,
43             'observacao'         => null
44         ]);
45     $response->assertStatus(302);
46     $response->assertSessionHas('success_toastr', '0 orçamento foi
cadastrado com sucesso!');
47 }
48
49 public function
teste2CriacaoDeOrcamentoComFalhaDadosObrigatorioVazios()
50 {
51     $user = factory(User::class)->create();
52     $estudio = factory(Estudio::class)->create();
53     factory(EstudioUsers::class)->create(['fk_users_id_users' =>
$user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
54
55     $response = $this->actingAs($user)
56         ->post('/admin/orcamentos', [
57             'cliente'           => null,
58             'artista'           => null,
59             'tatuagem_nome'     => '',
60             'tatuagem_local'    => '',
61             'tatuagem_comprimento' => null,
62             'tatuagem_largura'   => null,
63             'tatuagem_descricao' => '',
64             'tatuagem_referencias' => null,
65             'canal_contato'      => null,
66             'tempo_estimado'     => null,
67             'valor'              => null,
68             'uso_materiais'      => null,
69             'complexidade'       => null,

```

```
70         'observacao' => null
71     ]);
72
73     $response->assertStatus(302);
74     $this->assertEquals(session('errors')->get('cliente')[0], 'O
campo cliente é obrigatório.');
```

```
75     $this->assertEquals(session('errors')->get('artista')[0], 'O
campo artista é obrigatório.');
```

```
76     $this->assertEquals(session('errors')->get('tatuagem_nome')[0],
'0 campo tatuagem nome é obrigatório.');
```

```
77     $this->assertEquals(session('errors')->get('tatuagem_local')[0],
'0 campo tatuagem local é obrigatório.');
```

```
78     $this->assertEquals(session('errors')->get('tatuagem_comprimento
')[0], '0 campo tatuagem comprimento é obrigatório.');
```

```
79     $this->assertEquals(session('errors')->get('tatuagem_largura')
[0], '0 campo tatuagem largura é obrigatório.');
```

```
80     $this->assertEquals(session('errors')->get('tatuagem_descricao')
[0], '0 campo tatuagem descricao é obrigatório.');
```

```
81 }
82
83 public function
teste3CriacaoDeOrçamentoComFalhaDadosComTamanhoMaiorQueOSuportado()
84 {
85     include 'TestStrings.php';
86     $user = factory(User::class)->create();
87     $estudio = factory(Estudio::class)->create();
88     factory(EstudioUsers::class)->create(['fk_users_id_users' =>
$user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
89
90     $response = $this->actingAs($user)
91         ->post('/admin/orcamentos', [
92             'cliente' => 2147483648,
93             'artista' => 2147483648,
94             'tatuagem_nome' => 'String com 61 caracteres
TETESTETESTETESTETESTETESTETESTETES',
95             'tatuagem_local' => 'String com 61 caracteres
TETESTETESTETESTETESTETESTETESTETES',
96             'tatuagem_comprimento' => 201,
97             'tatuagem_largura' => 201,
98             'tatuagem_descricao' => 'String com 256 caracteres
ESTETESTETESTETESTETESTETESTETESTETESTETESTETESTETESTETESTETEST
',
99             'tatuagem_referencias' => $stringCom65536caracteres ,
100             'canal_contato' => 2147483648,
101             'tempo_estimado' => '24:00:00',
102             'valor' => 100000,
103             'uso_materiais' => 2147483648,
```

```

104         'complexidade'           => 2147483648,
105         'observacao'             => 'String com 256 caracteres
ESTETESTETESTETESTETESTETESTETESTETESTETESTETESTETESTETESTETEST
,
106     ]));
107
108     $response->assertStatus(302);
109     $this->assertEquals(session('errors')->get('tatuagem_nome')[0],
'0 campo tatuagem nome não pode ser superior a 60 caracteres.');
```

```

110     $this->assertEquals(session('errors')->get('tatuagem_local')[0],
'0 campo tatuagem local não pode ser superior a 60 caracteres.');
```

```

111     $this->assertEquals(session('errors')->get('tatuagem_comprimento
')[0], '0 campo tatuagem comprimento não pode ser superior a 200.');
```

```

112     $this->assertEquals(session('errors')->get('tatuagem_largura')
[0], '0 campo tatuagem largura não pode ser superior a 200.');
```

```

113     $this->assertEquals(session('errors')->get('tatuagem_descricao')
[0], '0 campo tatuagem descricao não pode ser superior a 255
caracteres.');
```

```

114     $this->assertEquals(session('errors')->get('valor')[0], '0 valor
do orçamento deve estar entre R$ 0,00 e R$ 99.999,99');
```

```

115     $this->assertEquals(session('errors')->get('tempo_estimado')[0],
'0 tempo estimado deve estar entre 00:00 e 23:59');
```

```

116     $this->assertEquals(session('errors')->get('observacao')[0], '0
campo observacao não pode ser superior a 255 caracteres.');
```

```

117 }
118
119 public function
teste4CriacaoDeOrçamentoComFalhaDadosComTiposDiferentesDoSuportado()
120 {
121     $user = factory(User::class)->create();
122     $estudio = factory(Estudio::class)->create();
123     factory(EstudioUsers::class)->create(['fk_users_id_users' =>
$user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
124
125     $response = $this->actingAs($user)
126         ->post('/admin/orcamentos', [
127             'cliente'           => 'String ao invés de inteiro',
128             'artista'           => 'String ao invés de inteiro',
129             'tatuagem_nome'     => 73273,
130             'tatuagem_local'    => 73273,
131             'tatuagem_comprimento' => 'String ao invés de inteiro',
132             'tatuagem_largura'  => 'String ao invés de inteiro',
133             'tatuagem_descricao' => ['array ao invés', 'de string
'],
134             'tatuagem_referencias' => 73273,
135             'canal_contato'      => ['array ao invés', 'de string
'],

```

```

136         'tempo_estimado'      => 1643067873,
137         'valor'               => 'String ao invés de float',
138         'uso_materiais'       => 'String ao invés de inteiro',
139         'complexidade'        => 'String ao invés de inteiro',
140         'observacao'          => 'String com 256 caracteres
ESTETESTETESTETESTETESTETESTETESTETESTETESTETESTETESTETESTETESTETEST
',
141     ]);
142
143     $response->assertStatus(302);
144     $this->assertEquals(session('errors')->get('tatuagem_nome')[0],
'0 campo tatuagem nome deve ser uma string.');
```

```

145     $this->assertEquals(session('errors')->get('tatuagem_local')[0],
'0 campo tatuagem local deve ser uma string.');
```

```

146     $this->assertEquals(session('errors')->get('tatuagem_comprimento
')[0], '0 campo tatuagem comprimento deve ser um número.');
```

```

147     $this->assertEquals(session('errors')->get('tatuagem_largura')
[0], '0 campo tatuagem largura deve ser um número.');
```

```

148     $this->assertEquals(session('errors')->get('tatuagem_descricao')
[0], '0 campo tatuagem descricao deve ser uma string.');
```

```

149     $this->assertEquals(session('errors')->get('valor')[0], '0 valor
do orçamento deve estar entre R$ 0,00 e R$ 99.999,99');
```

```

150     $this->assertEquals(session('errors')->get('tempo_estimado')[0],
'0 tempo estimado deve estar entre 00:00 e 23:59');
```

```

151     $this->assertEquals(session('errors')->get('observacao')[0], '0
campo observacao não pode ser superior a 255 caracteres.');
```

```

152 }
153 }
```

Algoritmo C.2 – Scripts de teste de Criação de Orçamentos

Fonte: o Autor

```

1 <?php
2
3 namespace Tests\Integration\Orçamentos;
4
5 use Tests\TestCase;
6 use App\User;
7 use App\EstudioUsers;
8 use App\Orçamento;
9 use App\Estudio;
10 use App\Cliente;
11 use App\Artista;
12 use App\ArtistaEstudio;
13 use App\ClienteEstudio;
14
15 class EdicaoDeOrçamentosTest extends TestCase
```

```

16 {
17
18     public function teste1EdicaoDeOrcamentoComSucessoSemDadosDeTatuador
19     ()
20     {
21         $user = factory(User::class)->create();
22         $estudio = factory(Estudio::class)->create();
23         factory(EstudioUsers::class)->create(['fk_users_id_users' =>
24 $user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
25         $artista = factory(Artista::class)->create();
26         $cliente = factory(Cliente::class)->create();
27         factory(ArtistaEstudio::class)->create(['fk_artista_id_artista'
28 => $artista->id_artista, 'fk_estudio_id_estudio' => $estudio->
29 id_estudio]);
30         factory(ClienteEstudio::class)->create(['fk_cliente_id_cliente'
31 => $cliente->id_cliente, 'fk_estudio_id_estudio' => $estudio->
32 id_estudio]);
33         $orcamento = factory(Orcamento::class)->create(['
34 fk_cliente_id_cliente' => $cliente->id_cliente, '
35 fk_artista_id_artista' => $artista->id_artista, '
36 fk_estudio_id_estudio' => $estudio->id_estudio, '
37 fk_orcamento_status_id_orcamento_status' => 1]);
38
39         $response = $this->actingAs($user)
40         ->post('/admin/orcamentos/' . $orcamento->id_orcamento, [
41             '_method' => 'PUT',
42             'cliente' => $cliente->id_cliente,
43             'artista' => $artista->id_artista,
44             'tatuagem_nome' => 'Teste Edição Orçamento',
45             'tatuagem_local' => 'Teste Edição Orçamento',
46             'tatuagem_comprimento' => 13,
47             'tatuagem_largura' => 13,
48             'tatuagem_descricao' => 'Teste Orçamento Editado com
49 Sucesso!',
50             'tatuagem_referencias' => 'testeedicaoorcamento.com.br',
51             'canal_contato' => 'Teste Edição',
52             'tempo_estimado' => null,
53             'valor' => null,
54             'uso_materiais' => null,
55             'complexidade' => null,
56             'observacao' => null
57         ]);
58         $response->assertStatus(302);
59         $response->assertSessionHas('success_toastr', '0 orçamento foi
60 atualizado com sucesso!');
61     }

```

```

50
51     public function teste2EdicaoDeOrcamentoComSucessoComDadosDeTatuador
52     (
53         {
54             $user = factory(User::class)->create();
55             $estudio = factory(Estudio::class)->create();
56             factory(EstudioUsers::class)->create(['fk_users_id_users' =>
57             $user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
58             $artista = factory(Artista::class)->create();
59             $cliente = factory(Cliente::class)->create();
60             factory(ArtistaEstudio::class)->create(['fk_artista_id_artista'
61             => $artista->id_artista, 'fk_estudio_id_estudio' => $estudio->
62             id_estudio]);
63             factory(ClienteEstudio::class)->create(['fk_cliente_id_cliente'
64             => $cliente->id_cliente, 'fk_estudio_id_estudio' => $estudio->
65             id_estudio]);
66             $orcamento = factory(Orcamento::class)->create(['
67             fk_cliente_id_cliente' => $cliente->id_cliente, '
68             fk_artista_id_artista' => $artista->id_artista, '
69             fk_estudio_id_estudio' => $estudio->id_estudio, '
70             fk_orcamento_status_id_orcamento_status' => 1]);
71
72             $response = $this->actingAs($user)
73             ->post('/admin/orcamentos/' . $orcamento->id_orcamento, [
74                 '_method' => 'PUT',
75                 'cliente' => $cliente->id_cliente,
76                 'artista' => $artista->id_artista,
77                 'tatuagem_nome' => 'Teste Edição Orçamento',
78                 'tatuagem_local' => 'Teste Edição Orçamento',
79                 'tatuagem_comprimento' => 13,
80                 'tatuagem_largura' => 13,
81                 'tatuagem_descricao' => 'Teste Orçamento Editado com
82                 Sucesso!',
83                 'tatuagem_referencias' => 'testeedicaoorcamento.com.br'
84             ,
85                 'canal_contato' => 'Teste Edição',
86                 'tempo_estimado' => '03:00',
87                 'valor' => '13,00',
88                 'uso_materiais' => 1,
89                 'complexidade' => 1,
90                 'observacao' => 'Teste Edição de Orçamento
91                 com sucesso com dados de tatuador!'
92             ]);
93             $response->assertStatus(302);
94             $response->assertSessionHas('success_toastr', 'O orçamento foi
95             atualizado com sucesso!');
96         }

```

```

83
84     public function
teste3EdicaoDeOrcamentoComFalhaDadosObrigatoriosVazios()
85     {
86         $user = factory(User::class)->create();
87         $estudio = factory(Estudio::class)->create();
88         factory(EstudioUsers::class)->create(['fk_users_id_users' =>
$user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
89         $artista = factory(Artista::class)->create();
90         $cliente = factory(Cliente::class)->create();
91         factory(ArtistaEstudio::class)->create(['fk_artista_id_artista'
=> $artista->id_artista, 'fk_estudio_id_estudio' => $estudio->
id_estudio]);
92         factory(ClienteEstudio::class)->create(['fk_cliente_id_cliente'
=> $cliente->id_cliente, 'fk_estudio_id_estudio' => $estudio->
id_estudio]);
93         $orcamento = factory(Orcamento::class)->create(['
fk_cliente_id_cliente' => $cliente->id_cliente, '
fk_artista_id_artista' => $artista->id_artista, '
fk_estudio_id_estudio' => $estudio->id_estudio, '
fk_orcamento_status_id_orcamento_status' => 1]);
94
95         $response = $this->actingAs($user)
96             ->post('/admin/orcamentos/' . $orcamento->id_orcamento, [
97             '_method' => 'PUT',
98             'cliente' => null,
99             'artista' => null,
100             'tatuagem_nome' => '',
101             'tatuagem_local' => '',
102             'tatuagem_comprimento' => null,
103             'tatuagem_largura' => null,
104             'tatuagem_descricao' => '',
105             'tatuagem_referencias' => null,
106             'canal_contato' => null,
107             'tempo_estimado' => null,
108             'valor' => null,
109             'uso_materiais' => null,
110             'complexidade' => null,
111             'observacao' => null
112             ]);
113         $response->assertStatus(302);
114         $this->assertEquals(session('errors')->get('cliente')[0], '0
campo cliente é obrigatório.');
```

```

117     $this->assertEquals(session('errors')->get('tatuagem_local')[0],
118         '0 campo tatuagem local é obrigatório.');
```

```

118     $this->assertEquals(session('errors')->get('tatuagem_comprimento
119         ')[0], '0 campo tatuagem comprimento é obrigatório.');
```

```

119     $this->assertEquals(session('errors')->get('tatuagem_largura')
120         [0], '0 campo tatuagem largura é obrigatório.');
```

```

120     $this->assertEquals(session('errors')->get('tatuagem_descricao')
121         [0], '0 campo tatuagem descricao é obrigatório.');
```

```

121     }
122
123     public function
124     teste4EdicaoDeOrcamentoComFalhaDadosComTamanhoMaiorQueOSuportado()
125     {
126         include 'TestStrings.php';
127         $user = factory(User::class)->create();
128         $estudio = factory(Estudio::class)->create();
129         factory(EstudioUsers::class)->create(['fk_users_id_users' =>
130             $user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
131         $artista = factory(Artista::class)->create();
132         $cliente = factory(Cliente::class)->create();
133         factory(ArtistaEstudio::class)->create(['fk_artista_id_artista'
134             => $artista->id_artista, 'fk_estudio_id_estudio' => $estudio->
135             id_estudio]);
136         factory(ClienteEstudio::class)->create(['fk_cliente_id_cliente'
137             => $cliente->id_cliente, 'fk_estudio_id_estudio' => $estudio->
138             id_estudio]);
139         $orcamento = factory(Orcamento::class)->create(['
140             fk_cliente_id_cliente' => $cliente->id_cliente, '
141             fk_artista_id_artista' => $artista->id_artista, '
142             fk_estudio_id_estudio' => $estudio->id_estudio, '
143             fk_orcamento_status_id_orcamento_status' => 1]);
144
145         $response = $this->actingAs($user)
146             ->post('/admin/orcamentos/' . $orcamento->id_orcamento, [
147                 '_method' => 'PUT',
148                 'cliente' => 2147483648,
149                 'artista' => 2147483648,
150                 'tatuagem_nome' => $stringCom61Caracteres,
151                 'tatuagem_local' => $stringCom61Caracteres,
152                 'tatuagem_comprimento' => 201,
153                 'tatuagem_largura' => 201,
154                 'tatuagem_descricao' => $stringCom256Caracteres,
155                 'tatuagem_referencias' => $stringCom65536Caracteres,
156                 'canal_contato' => 2147483648,
157                 'tempo_estimado' => '24:00:00',
158                 'valor' => '100.000,00',
159                 'uso_materiais' => 2147483648,

```



```

150         'complexidade'           => 2147483648,
151         'observacao'             => $stringCom256Caracteres
152     ]);
153
154     $response->assertStatus(302);
155     $this->assertEquals(session('errors')->get('tatuagem_nome')[0],
156         '0 campo tatuagem nome não pode ser superior a 60 caracteres.');
```

```

156     $this->assertEquals(session('errors')->get('tatuagem_local')[0],
157         '0 campo tatuagem local não pode ser superior a 60 caracteres.');
```

```

157     $this->assertEquals(session('errors')->get('tatuagem_comprimento')
158         [0], '0 campo tatuagem comprimento não pode ser superior a 200.');
```

```

158     $this->assertEquals(session('errors')->get('tatuagem_largura')
159         [0], '0 campo tatuagem largura não pode ser superior a 200.');
```

```

159     $this->assertEquals(session('errors')->get('tatuagem_descricao')
160         [0], '0 campo tatuagem descricao não pode ser superior a 255
161         caracteres.');
```

```

160     $this->assertEquals(session('errors')->get('valor')[0], '0 campo
161         valor tem um formato inválido.');
```

```

161     $this->assertEquals(session('errors')->get('tempo_estimado')[0],
162         '0 campo tempo estimado tem um formato inválido.');
```

```

162     $this->assertEquals(session('errors')->get('observacao')[0], '0
163         campo observacao não pode ser superior a 255 caracteres.');
```

```

163 }
164
165 public function
166 teste5EdicaoDeOrcamentoComFalhaDadosComTiposDiferentesDoSuportado()
167 {
168     include 'TestStrings.php';
169     $user = factory(User::class)->create();
170     $estudio = factory(Estudio::class)->create();
171     factory(EstudioUsers::class)->create(['fk_users_id_users' =>
172         $user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
173     $artista = factory(Artista::class)->create();
174     $cliente = factory(Cliente::class)->create();
175     factory(ArtistaEstudio::class)->create(['fk_artista_id_artista'
176         => $artista->id_artista, 'fk_estudio_id_estudio' => $estudio->
177         id_estudio]);
178     factory(ClienteEstudio::class)->create(['fk_cliente_id_cliente'
179         => $cliente->id_cliente, 'fk_estudio_id_estudio' => $estudio->
180         id_estudio]);
181     $orcamento = factory(Orcamento::class)->create(['
182         fk_cliente_id_cliente' => $cliente->id_cliente, '
183         fk_artista_id_artista' => $artista->id_artista, '
184         fk_estudio_id_estudio' => $estudio->id_estudio, '
185         fk_orcamento_status_id_orcamento_status' => 1]);
186
187     $response = $this->actingAs($user)

```

```

178         ->post('/admin/orcamentos/' . $orcamento->id_orcamento, [
179             '_method'           => 'PUT',
180             'cliente'           => 'String ao invés de inteiro',
181             'artista'           => 'String ao invés de inteiro',
182             'tatuagem_nome'     => 73273,
183             'tatuagem_local'    => 73273,
184             'tatuagem_comprimento' => 'String ao invés de inteiro',
185             'tatuagem_largura'   => 'String ao invés de inteiro',
186             'tatuagem_descricao' => ['array ao invés', 'de string
187
188             ],
189             'tatuagem_referencias' => 73273,
190             'canal_contato'       => ['array ao invés', 'de string
191
192             ],
193             'tempo_estimado'     => 1643067873,
194             'valor'              => 100000.00,
195             'uso_materiais'      => 'String ao invés de inteiro',
196             'complexidade'       => 'String ao invés de inteiro',
197             'observacao'         => $stringCom256Caracteres
198         ]);
199
200     $response->assertStatus(302);
201     $this->assertEquals(session('errors')->get('tatuagem_nome')[0],
202         '0 campo tatuagem nome deve ser uma string.');
```

```

197     $this->assertEquals(session('errors')->get('tatuagem_local')[0],
198         '0 campo tatuagem local deve ser uma string.');
```

```

198     $this->assertEquals(session('errors')->get('tatuagem_comprimento')
199         [0], '0 campo tatuagem comprimento deve ser um número.');
```

```

199     $this->assertEquals(session('errors')->get('tatuagem_largura')
200         [0], '0 campo tatuagem largura deve ser um número.');
```

```

200     $this->assertEquals(session('errors')->get('tatuagem_descricao')
201         [0], '0 campo tatuagem descricao deve ser uma string.');
```

```

201     $this->assertEquals(session('errors')->get('valor')[0], '0 campo
202         valor tem um formato inválido.');
```

```

202     $this->assertEquals(session('errors')->get('tempo_estimado')[0],
203         '0 campo tempo estimado tem um formato inválido.');
```

```

203     $this->assertEquals(session('errors')->get('observacao')[0], '0
204         campo observacao não pode ser superior a 255 caracteres.');
```

```

204     }
205 }
```

Algoritmo C.3 – Scripts de teste de Edição de Orçamentos

Fonte: o Autor

```

1 <?php
2
3 namespace Tests\Integration\Orcamentos;
4
5 use Tests\TestCase;
```

```

6 use App\User;
7 use App\EstudioUsers;
8 use App\Orcamento;
9 use App\Estudio;
10 use App\Cliente;
11 use App\Artista;
12 use App\ArtistaEstudio;
13 use App\ClienteEstudio;
14 use App\Estacao;
15 use App\Agendamento;
16
17 class ListagemEDelecaoDeOrcamentosTest extends TestCase
18 {
19
20     public function teste1ListagemDeOrcamentosComSucesso()
21     {
22         $user = factory(User::class)->create();
23         $estudio = factory(Estudio::class)->create();
24         factory(EstudioUsers::class)->create(['fk_users_id_users' =>
25 $user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
26         $artista = factory(Artista::class)->create();
27         $cliente = factory(Cliente::class)->create();
28         factory(ArtistaEstudio::class)->create(['fk_artista_id_artista'
29 => $artista->id_artista, 'fk_estudio_id_estudio' => $estudio->
30 id_estudio]);
31         factory(ClienteEstudio::class)->create(['fk_cliente_id_cliente'
32 => $cliente->id_cliente, 'fk_estudio_id_estudio' => $estudio->
33 id_estudio]);
34
35         $orcamento = factory(Orcamento::class)->create(['
36 fk_cliente_id_cliente' => $cliente->id_cliente, '
37 fk_artista_id_artista' => $artista->id_artista, '
38 fk_estudio_id_estudio' => $estudio->id_estudio, '
39 fk_orcamento_status_id_orcamento_status' => 1]);
40
41         $orcamento2 = factory(Orcamento::class)->create(['
42 fk_cliente_id_cliente' => $cliente->id_cliente, '
43 fk_artista_id_artista' => $artista->id_artista, '
44 fk_estudio_id_estudio' => $estudio->id_estudio, '
45 fk_orcamento_status_id_orcamento_status' => 1]);
46
47         $response = $this->actingAs($user)
48             ->get('/admin/orcamentos');
49
50         $response->assertStatus(200);
51         $response->assertSee('<td class="align-middle">' . $orcamento->
52 tatuagem_nome . '</td>', $escaped = false);
53         $response->assertSee('<td class="align-middle">#' . str_pad(
54 $orcamento->id_orcamento, 5, '0', STR_PAD_LEFT) . '</td>', $escaped =

```

```

    false);
38     $response->assertSee('<td class="align-middle">' . $orcamento2->
tatuagem_nome . '</td>', $escaped = false);
39     $response->assertSee('<td class="align-middle">#' . str_pad(
$orcamento2->id_orcamento, 5, '0', STR_PAD_LEFT) . '</td>', $escaped
= false);
40 }
41
42 public function teste2ExclusaoDeOrcamentoNaoAgendadoComSucesso()
43 {
44     $user = factory(User::class)->create();
45     $estudio = factory(Estudio::class)->create();
46     factory(EstudioUsers::class)->create(['fk_users_id_users' =>
$user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
47     $artista = factory(Artista::class)->create();
48     $cliente = factory(Cliente::class)->create();
49     factory(ArtistaEstudio::class)->create(['fk_artista_id_artista'
=> $artista->id_artista, 'fk_estudio_id_estudio' => $estudio->
id_estudio]);
50     factory(ClienteEstudio::class)->create(['fk_cliente_id_cliente'
=> $cliente->id_cliente, 'fk_estudio_id_estudio' => $estudio->
id_estudio]);
51     $orcamento = factory(Orcamento::class)->create(['
fk_cliente_id_cliente' => $cliente->id_cliente, '
fk_artista_id_artista' => $artista->id_artista, '
fk_estudio_id_estudio' => $estudio->id_estudio, '
fk_orcamento_status_id_orcamento_status' => 1]);
52
53     $response = $this->actingAs($user)
54         ->post('/admin/orcamentos/' . $orcamento->id_orcamento, [
55             '_method' => 'DELETE'
56         ]);
57     $response->assertStatus(302);
58     $response->assertSessionHas('success_toastr', '0 orçamento foi
excluído com sucesso!');
59 }
60
61 public function teste3ExclusaoDeOrcamentoAgendadoComSucesso()
62 {
63     $user = factory(User::class)->create();
64     $estudio = factory(Estudio::class)->create();
65     factory(EstudioUsers::class)->create(['fk_users_id_users' =>
$user->id, 'fk_estudio_id_estudio' => $estudio->id_estudio]);
66     $artista = factory(Artista::class)->create();
67     $cliente = factory(Cliente::class)->create();
68     factory(ArtistaEstudio::class)->create(['fk_artista_id_artista'
=> $artista->id_artista, 'fk_estudio_id_estudio' => $estudio->

```

```

        id_estudio]);
69         factory(ClienteEstudio::class)->create(['fk_cliente_id_cliente'
=> $cliente->id_cliente, 'fk_estudio_id_estudio' => $estudio->
id_estudio]);
70         $orcamento = factory(Orcamento::class)->create(['
fk_cliente_id_cliente' => $cliente->id_cliente, '
fk_artista_id_artista' => $artista->id_artista, '
fk_estudio_id_estudio' => $estudio->id_estudio, '
fk_orcamento_status_id_orcamento_status' => 2, 'tempo_estimado' => '
02:00:00', 'valor' => 500.00, 'fk_uso_materiais_id_uso_materiais' =>
1, 'fk_complexidade_id_complexidade' => 1, 'observacao' => '
teste3ExclusaoDeOrcamentoAgendadoComSucesso']);
71         $estacao = factory(Estacao::class)->create(['
fk_estudio_id_estudio' => $estudio->id_estudio]);
72         factory(Agendamento::class)->create(['fk_orcamento_id_orcamento'
=> $orcamento->id_orcamento, 'fk_estacao_id_estacao' => $estacao->
id_estacao, 'fk_agendamento_status_id_agendamento_status' => 1]);
73         $orcamento->fk_orcamento_status_id_orcamento_status = 3;
74         $orcamento->save();
75
76         $response = $this->actingAs($user)
77             ->post('/admin/orcamentos/' . $orcamento->id_orcamento, [
78                 '_method' => 'DELETE'
79             ]);
80         $response->assertStatus(302);
81         $response->assertSessionHas('success_toastr', '0 orçamento foi
excluído com sucesso!');
82     }
83 }

```

Algoritmo C.4 – Scripts de teste de Listagem e Deleção de Orçamentos

Fonte: o Autor