UNIVERSIDADE FEDERAL DE ALFENAS

GLAUCO AMARAL GERALDINO

UMA PROPOSTA PARA AUTOMAÇÃO DE TESTES E MELHORIAS NO *DESIGN* DA WEBSTAMP

Alfenas/MG
2021

GLAUCO AMARAL GERALDINO

UMA PROPOSTA PARA AUTOMAÇÃO DE TESTES E MELHORIAS NO *DESIGN* DA WEBSTAMP

Trabalho apresentado como parte dos requisitos para a disciplina de Trabalho de Conclusão de Curso pelo curso de Ciência da Computação da Universidade Federal de Alfenas - UNIFAL-MG.
Orientador: Rodrigo Martins Pagliares

Alfenas/MG
2021

GLAUCO AMARAL GERALDINO

UMA PROPOSTA PARA AUTOMAÇÃO DE TESTES E MELHORIAS NO *DESIGN* DA WEBSTAMP

A Banca examinadora abaixo-assinada, aprova o Trabalho apresentado como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação pela Universidade Federal de Alfenas.

Aprovado em:

_____

Prof. Rodrigo Martins Pagliares
Universidade Federal de Alfenas

_____

Prof. Fellipe Guilherme Rey de Souza
Instituto Tecnológico de Aeronáutica

_____

Profa. Mariane Moreira de Souza
Universidade Federal de Alfenas

Alfenas/MG
2021

RESUMO

A análise de situações de perigo (*hazards*) em sistemas complexos é uma tarefa árdua que demanda a identificação de diversas situações que possam levar o sistema para um estado de alto risco e, consequentemente, possibilitando a ocorrência de perdas (*losses*). Analistas de situações de perigo se beneficiam de ferramentas de *software* na confecção de suas análises, tendo em vista que é inviável gerenciá-las de forma manual devido a questões de tamanho e complexidade. WebSTAMP é uma ferramenta *web* gratuita de código fechado que objetiva auxiliar analistas de situações de perigo de maneira mais guiada e automatizada. A versão atual da WebSTAMP apresenta uma série de limitações relacionadas ao seu desenho (*design*). Além disso, testes de unidade, integração e fim-a-fim são feitos de forma manual. Este trabalho tem como objetivos (i) propor melhorias no código fonte da WebSTAMP que objetivam reduzir as limitações de *design* da versão atual e (ii) facilitar a atividade de testes da WebSTAMP por meio do desenvolvimento de testes automatizados que permitem a execução de testes de regressão e facilitam a validação da corretude da implementação no que tange à aderência aos requisitos. A principal contribuição deste trabalho é uma proposta de um conjunto de melhorias no *design* do módulo servidor (*back-end*) da WebSTAMP juntamente com um conjunto (*suite*) de testes automatizados que facilita a execução dos testes e evolução da ferramenta. Baseado em revisão da literatura, o presente trabalho descreve as melhorias de *design* propostas para a WebSTAMP, além de apresentar testes que foram automatizados (unidade, integração e fim-a-fim). Nossos resultados indicam que o conjunto de melhorias proposto neste trabalho tem o potencial de facilitar o desenvolvimento de novas funcionalidades para a WebSTAMP, tendo em vista que é mais fácil realizar manutenções e acrescentar novas funcionalidades em código bem projetado. Desenvolver *software* livre de erros é uma tarefa complexa, ainda mais quando temos um alto grau de acoplamento entre os componentes do *software*, muitas vezes devido a um *design* ruim. Este trabalho sugere uma proposta para a automação de testes e melhorias no *design* da WebSTAMP. A proposta visa facilitar a evolução da WebSTAMP a partir de melhorias no seu *design* e minimizar esforços para encontrar falhas, reduzir tempo com testes e diminuir custos, visto que quanto mais tardio um erro (*bug*) for detectado, maior o custo de manutenção. Os resultados obtidos pela automação de testes possibilitam testar código existente de maneira rápida e automatizada, além de evitar que erros sejam introduzidos inadvertidamente no momento de manutenção ou evolução do código existente.

Palavras-Chave: WebSTAMP. *Design* de *software.* Testes. Teste de *Software*. Testes Automatizados.

ABSTRACT

The analysis of hazards in complex systems is a hard task, which demands the identification of many situations that may take the system to a state of higher risk and, consequently, losses. Hazard Analysts benefit themselves from software tools in the confection of their analyses, knowing that is impracticable to manage them in a manual way, considering the size and their complexity. WebSTAMP is a free web tool with closed source that aims to assist hazards analysts in an automatic and guided way. The current version of WebSTAMP presents a set of limitations related to its design. Moreover, unit, integration and end-to-end tests are performed manually. This research aims (i) to suggest improvements in the WebSTAMP source code that aim to reduce limitations of the current design and (ii) to facilitate the WebSTAMP testing activity by the development of automatic tests, allowing the execution of regression tests, and supporting the validation of the correctness of the implementations to verify the adherence to the requirements. The main contribution of this work is a proposal of a set of improvements in the design of the back-end module of WebSTAMP together with an automatic test suite that facilitates the execution of the tests and WebSTAMP evolution. Based on a literature review, the present work describes the proposed improvements in the design of WebSTAMP, besides presenting automated tests. Our results show that the set of improvements proposed in this work has the potential to facilitate the development of new functionalities for WebSTAMP, since it is easier to maintain and add new functionalities in a well-designed code. Developing software without errors is a complex task, even more, when we have a high degree of coupling between the software components, many times because of a bad design. This work suggests a proposal for test automation and improvements in the WebSTAMP design. The proposal aims to facilitate the evolution of WebSTAMP by conducting improvements in its design and minimizing efforts to find failures, to reduce time with tests and costs, since the late the bug is detected, the higher the maintenance costs. The results obtained by automation of tests make it possible to assess the existent code in a quick and automatized way besides avoiding errors that could be introduced inadvertently during maintenance or code evolution.

Keywords: WebSTAMP. Software design. Software testing. Automated Test.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIAÇÕES

API – Application Programming Interface

CRUD - Create, Read, Update, Delete.

DOC - Depend On Component

HTML - HyperText Markup Language

HTTP - Hypertext Transfer Protocol

MVC - Model View Controller

SGBD - Sistema Gerenciador De Banco De Dados.

SGBDR - Sistema De Gerenciador De Banco De Dados Relacional

SQL - Structured Query Language

STPA - Systems Theoretic Process Analysis

SUT - System Under Test

SUMÁRIO

1 INTRODUÇÃO

Um sistema complexo é composto por um amplo conjunto de partes que se inter-relacionam para um determinado propósito (LEVESON, 2018). A análise de situações de perigos (*hazards*) em sistemas complexos é uma tarefa árdua que demanda a identificação de diversas situações que possam levar o sistema para um estado de alto risco, possibilitando a ocorrência de perdas. *Safety Analysts* (Analistas de Segurança) se beneficiam de ferramentas de *software* já que é inviável gerenciar análises de perigos de forma manual.

Uma situação de perigo consiste em uma situação com potencial para provocar danos e perdas em um sistema, como por exemplo, em um sistema metroviário, caso uma porta não esteja fechada no início de um percurso com o trem/metrô em movimento, pode ocorrer lesões em passageiros e até casos mais graves de mortes.

WebSTAMP (Souza *et al.*, 2019), é um *software web*, de código fechado, que objetiva auxiliar analistas no processo de análise de perigos de maneira mais guiada e automatizada. Um *software web* típico possui ao menos duas camadas: cliente (*front-end*) e servidor (*back-end*). A camada cliente da WebSTAMP utiliza tecnologias HTML, CSS (*Cascading Style Sheets*) e JavaScript. A camada da WebSTAMP implantada no servidor utiliza a linguagem PHP por meio do *framework* Laravel (STAUFFER, 2019). A WebSTAMP armazena as análises de perigos em um SGBD (Sistema Gerenciador de Banco de Dados) MySQL.

A versão atual da WebSTAMP possui diversas limitações referentes ao *design* de seu *back-end*, além de não possuir testes automatizados. Dentre as limitações no *design*, destacamos a baixa coesão e alto acoplamento entre os componentes (GAMMA *et al.*, 2000). Todos os testes da WebSTAMP (testes de unidade, testes de integração e testes fim-a-fim) são realizados de forma manual. Testar a WebSTAMP é desgastante já que consome muito tempo, ainda mais quando levamos em consideração o processo de retestagem por meio de testes de regressão, que implicam na verificação de funcionalidades pré-existentes.

Este trabalho tem como objetivos (i) propor melhorias no código fonte da WebSTAMP que objetivam reduzir as limitações de *design* da versão atual e (ii) facilitar a atividade de testes da WebSTAMP por meio do desenvolvimento de testes automatizados. A automação dos testes auxilia a execução de testes de regressão e suporta a validação da corretude da implementação no que tange à aderência aos requisitos.

A principal contribuição deste trabalho é uma proposta de um conjunto de melhorias no *design* do módulo servidor (*back-end*) da WebSTAMP juntamente com um conjunto (*suite*) de testes automatizados que facilita a execução dos testes e evolução da ferramenta.

Utilizamos uma metodologia constituída de etapas. Na primeira etapa, identificamos trechos de código fonte passíveis de refatoração no *back-end* da WebSTAMP objetivando melhorias de *design*. Uma vez identificados os trechos de código, iniciamos a segunda etapa, executando refatorações no código fonte com o intuito de modificar a estrutura interna do código, sem alterar o seu comportamento observável (FOWLER *et al*, 1999). Os resultados desta etapa facilitaram a execução da terceira etapa, que consiste no planejamento e implementação de testes automatizados, já que é mais fácil criar testes para código com bom *design*. Implementamos testes automatizados que cobrem desde uma classe individual com testes de unidade, até classes que são combinadas e testadas em grupo com testes de integração e testes fim-a-fim (*end-to-end*). Os testes implementados visam garantir maior cobertura e qualidade do código sendo testado.

O Quadro 1.1 apresenta uma visão geral dos pontos de *design* e testes de como a ferramenta encontrava-se no momento em que se iniciou o presente trabalho e como está após a finalização deste.

WebSTAMP (Atualmente)	WebSTAMP (Proposta)
Alto acoplamento entre os componentes	Baixo acoplamento entre os componentes (facilidade de manutenções futuras)
Testes manuais	Testes automatizados (unidade, integração e fim-a-fim)
Somente camada de Autenticação	Criação de uma camada para Autorização
Ausência das camadas de Serviço (*Service*) e Repositório (*Repository*)	Implementação das camadas de Serviço e Repositório

Quadro 1.1 - Visão Geral dos pontos propostos no presente trabalho em comparação com a atual versão da ferramenta
Fonte: dados da pesquisa, 2021.

O restante deste trabalho está organizado da seguinte maneira. No Capítulo 2, apresentamos a revisão bibliográfica sobre técnicas, métodos, princípios e padrões direcionados à melhoria de *design* e testes de *software*. Apresentamos os trabalhos relacionados no Capítulo 3. No Capítulo 4 são apresentadas as tecnologias utilizadas no presente trabalho. A proposta de melhorias para a WebSTAMP é discutida nos Capítulos 5 e 6, que tratam de melhorias de *design*

e testes automatizados, respectivamente. Por fim, nos Capítulos 7 e 8 são apresentadas as discussões, os trabalhos futuros e conclusões.

2 REVISÃO BIBLIOGRAFICA

Este capítulo apresenta uma revisão bibliográfica sobre técnicas, métodos, princípios e padrões direcionados à melhoria de *design* e testes de *software*.

Fowler *et. al.* (1999) descrevem refatorar como sendo o processo de alteração de um sistema em *software*, melhorando sua estrutura interna, sem que ocorra alterações no comportamento externo (observável) do código. Refatorar é uma maneira disciplinada de limpar código, minimizando as chances de introdução de *bugs*. Além disso, ao refatorar, você está aperfeiçoando o *design* do código após ele ter sido escrito.

Fowler (2002) apresenta o pattern arquitetural *Service Layer* (Camada de Serviço), uma camada de serviços que estabelece um conjunto de operações disponíveis em uma aplicação de *software* e coordena a resposta da aplicação em cada operação. Uma camada de serviço encapsula a lógica de negócios da aplicação, controlando transações e coordenando respostas na implementação de suas operações.

Evans (2003) apresenta o padrão de projeto (*design pattern)* chamado *Repository*, que tem como objetivo encapsular a lógica de acesso aos dados presentes no banco de dados, por meio de objetos entidades (*Entities*) que representam as tuplas de uma tabela no banco de dados. Dentre as vantagens da utilização do *Repository*, destacam-se, por exemplo, a possibilidade de se usar *mocks* (objetos fictícios) para simulação em testes que envolvam banco de dados e uma visão mais orientada a objetos das interações com a camada de persistência.

As áreas de *design* e testes estão intimamente relacionadas. Segundo Cardoso e Aniche (2015), se o processo de testagem em um projeto torna-se difícil, é preciso reavaliar o *design*, visto que o SUT (*System Under Test*) pode apresentar baixa coesão e alto acoplamento. Dessa forma, torna-se necessário realizar mudanças em relação ao *design* do SUT a fim de que seja possível criar testes automatizados para o mesmo.

Myers *et. al.* (2004) argumentam que *software* deve ser previsível e consistente, não oferecendo surpresas aos usuários. Sendo assim, teste de *software* baseia-se em uma atividade ou um conjunto de atividades que determinam se a funcionalidade ocorre de forma correta no ambiente para o qual foi projetada. Posto isto, é por meio dos testes que os desenvolvedores podem pôr à prova o que foi produzido, e elevando a confiabilidade e qualidade do *software*.

Sob o mesmo ponto de vista, Bertolino (2007), descreve que testes são amplamente utilizados na indústria para garantia de qualidade, fornecendo um *feedback* real do comportamento do *software* conforme o planejado. Ainda, segundo o autor, teste é uma atividade crucial na engenharia de *software*. Os testes podem ser subdivididos em dois modos principais: manuais e automatizados.

Testes manuais são a forma mais primitiva de teste, na qual testadores e desenvolvedores executam ações guiadas por casos de testes sem o auxílio de ferramentas de testes. Segundo Bernardo e Kon (2008), realizar a execução de um caso de teste de maneira manual é uma tarefa rápida e efetiva, mas quando se trata de um conjunto grande de casos de testes, realizar manualmente acaba sendo dispendioso e cansativo.

Testes automatizados consistem em ferramentas e *frameworks* que executam o SUT em diversas circunstâncias e comparam o resultado das execuções com o resultado esperado do *software*. Dessa forma, testes automatizados possuem a vantagem de executar um conjunto de casos de teste, além de verificar os mesmos casos após mudanças consideráveis de código.

O presente trabalho utiliza os conceitos de SUT e DOC (*Depended On Component*), apresentados por Kaczanowski (2013). SUT, conforme já discutido, consiste na parte do sistema/*software* que está sendo testada e DOC trata-se dos componentes exigidos pelo SUT para que este possa realizar suas funções.

A Figura 2.1 refere-se aos conceitos de SUT e DOC's juntamente com as relações diretas e indiretas entre os componentes (classe de teste, SUT e DOC's). As relações diretas ocorrem entre a classe de teste e o SUT, havendo uma troca direta entre informações e controle da classe de teste sobre o SUT. Já as relações indiretas ocorrem entre o SUT e os DOC's, havendo uma troca indireta de informações na qual a classe de teste não possui controle.

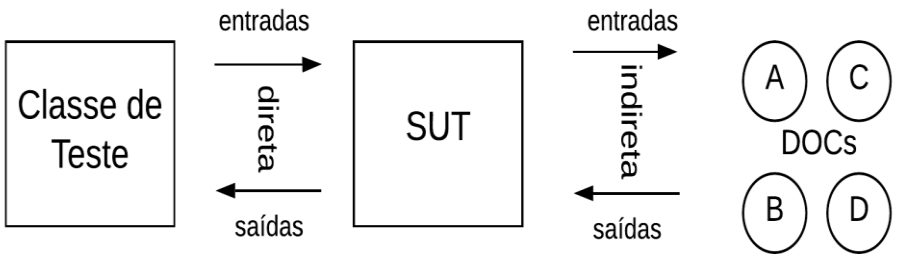

Figura 2.1 - Relações entre SUT e DOCs
Fonte: Adaptada de Kaczanowski, 2013.

Além de manuais ou automatizados, testes de *software* podem ser classificados como teste de unidade, teste de integração, e teste fim-a-fim (KACZANOWSKI; TOMEK, 2013). O Quadro 2.1 apresenta a definição para estes três tipos de testes, junto com alguns exemplos de ferramentas/*frameworks* de suporte que podem ser utilizados na confecção dos mesmos.

Tipo	**Definição**	**Ferramentas/*frameworks***
Teste de unidade	Testa as menores unidades de *software* (métodos, classes) que podem ser isoladas logicamente em um sistema. Dependências entre classes são substituídas por componentes *mocks*. Testes de unidade são rápidos e executados frequentemente pelos próprios desenvolvedores.	Tecnologias Java: - JUnit (JUnit, 1998) - JMock (JMock, 2007) Tecnologias PHP: - PHPUnit (PHPUnit, 2001) - Phockito (Phockito, 2013)
Teste de integração	Realiza a integração de módulos (partes) do sistema, incluindo também códigos que não temos controles (bibliotecas de terceiros). São demorados quando comparados aos testes de unidade por exigirem um contexto (ambiente) configurado e também por invocar recursos que possam ter um tempo maior de resposta, como chamadas a banco de dados.	Tecnologias Java: - JUnit - DBunit (DBunit, 2002) - Arquillian (Arquillian, 2010) Tecnologias PHP: - PhpUnit
Teste fim-a-fim	Tem como finalidade testar o sistema como um todo, simulando um usuário utilizando o sistema real. Estende-se, por exemplo, desde a interface gráfica com o usuário do sistema até os módulos mais internos de armazenamento dos dados.	Tecnologias Java e PHP: - Selenium WebDriver (Selenium WebDriver, 2004)

Quadro 2.1 - Tipos de testes de *software* e ferramentas de suporte
Fonte: dados da pesquisa, 2021.

Conforme pode ser visto no Quadro 2.1, testes de unidade são utilizados para testar uma classe de forma isolada, sem DOC's. Testes de integração são utilizados para testar as interações entre classes e módulos, sejam eles do próprio sistema ou de terceiros. Os testes fim-a-fim, testam a interação do usuário com o *software*, desde a chamada a uma operação do sistema feita via *interface* com o usuário, até passar por todas as camadas *do software*, chegando na camada de recursos onde reside, por exemplo, um SGBD. De maneira geral, os testes fim-a-fim verificam se as interações realizadas no *software* estão corretas de acordo com o que se espera da funcionalidade do sistema.

Além disso, a terceira coluna no Quadro 2.1 apresenta exemplos de ferramentas/*frameworks*, baseados nas tecnologias Java e PHP, para cada tipo de teste citado anteriormente. Tecnologias similares existem para outras linguagens de programação.

Tendo em vista que a WebSTAMP é implementada utilizando tecnologias PHP, usamos neste trabalho as tecnologias/*frameworks* de testes também voltados para esta linguagem. São eles: PHPUnit, para testes de unidade e integração e Selenium para testes fim-a-fim.

No próximo capítulo apresentamos os trabalhos relacionados. Focamos em trabalhos sobre métodos/técnicas para melhoria do *design* de *software*, além de fornecer uma breve apresentação da WebSTAMP, objeto de estudo deste trabalho.

3 TRABALHOS RELACIONADOS

Este capítulo tem como objetivo apresentar trabalhos relacionados ao apresentado nesta monografia.

WebSTAMP (Souza *et al.*, 2019) é uma aplicação *web* para análise de situações de perigo usando STPA (System Theoretic Process Analysis) (LEVESON, 2011). WebSTAMP auxilia engenheiros de *safety* fornecendo diretrizes e parcialmente automatizando a geração de trechos das análises. Conforme discutido, a WebSTAMP possui uma série de limitações no *design* de seu *back-end* e não possui testes automatizados.

Da Silva e Siqueira (2013) apresentam técnicas de automação de testes em um exemplo de um sistema PDV (Ponto de Venda), sistema onde ocorre uma transação de venda, descrito por Larman (2012). O trabalho foca em testes de unidade com e sem a utilização de *mocks* (MESZAROS, 2007). Os autores usam os *frameworks* JUnit e Mockito (Mockito, 2008) na automação dos testes. O JUnit é utilizado para criação dos testes automatizados de unidade e o Mockito para simular as dependências entre objetos com base nos conceitos de SUT e DOC's. Os autores concluem que sistemas com testes se tornam mais flexíveis, confiáveis e com maior segurança para alterações, pois pode se verificar rapidamente se alguma alteração realizada é a causa de problemas ou falhas no sistema presente.

Ratzinger *et al.*, (2005), apresentam um estudo sobre evolução de *software* por meio de refatoração. Os autores usam um sistema industrial com 500 mil linhas de código propondo refatorações em pontos específicos com base em observação de manutenções frequentes realizadas na mesma área de código e detecção de *bad smells* (indícios de que o código-fonte pode melhorar) no código. Dessa forma, os desenvolvedores responsáveis pelas manutenções concluem que as refatorações direcionadas por meio das análises são eficazes, e que as novas classes e *interfaces* propostas são claras e fáceis de serem utilizadas. Por fim, os autores afirmam que a abordagem de análise por meio da observação de manutenções e detecção de *bad smells*, podem ajudar a melhorar a capacidade de manutenção e evolução de um grande sistema de *software*.

Ampatzoglou *et al.*, (2015), apresentam um estudo sobre a estabilidade de um sistema de *software*, ou seja, a resistência ao efeito cascata na propagação de mudanças. No estudo, os autores investigaram a estabilidade por meio da observação de utilização de padrões de projeto GoF (GAMMA *et al*., 2000) (conjunto de 23 padrões de projeto - *design patterns*) nas classes do sistema de *software*, examinando se esta pode ser afetada pelo tipo de *design pattern*, papel

desempenhado pela classe no *design pattern*, número de *design patterns* em que a classe está envolvida e domínio do sistema. Com base nos resultados encontrados no estudo de impacto de mudança em classes que participam de zero, uma ou mais ocorrências de *design patterns* e com cerca de 65.000 classes Java de código aberto, os autores concluem que a aplicação de *design patterns* em sistemas pode oferecer uma "blindagem" em certas classes contra sofrer mudanças, visto que classes que estão associadas a *design patterns* possuem maior estabilidade. Além disso, pode-se utilizar dos resultados, de correlação entre estabilidade e utilização de *design patterns*, para testar e priorizar refatorações, pois classes menos estáveis exigem maior esforço durante a etapa de testes e a refatoração torna a classe mais resistente contra a propagação de mudanças.

Kumar e Mishra (2016) apresentam um estudo sobre o impacto da automação de testes no custo, qualidade e tempo para comercialização de um *software*. O estudo foi realizado por meio do cálculo de um modelo matemático que se baseia em fatores de esforço de teste, para que fosse possível quantificar os impactos da automação de testes em três sistemas de *software*. Dessa forma, os autores confirmam que os resultados da comparação entre testes manuais e automatizados revelam efeitos positivos da automação de testes no custo, qualidade e tempo de lançamento de um *software*.

4 TECNOLOGIAS UTILIZADAS

Este capítulo tem como objetivo descrever as tecnologias utilizadas para o desenvolvimento deste trabalho. De maneira geral, usamos as mesmas ferramentas utilizadas na WebSTAMP (versão de 2019), excetuando as ferramentas para automação de testes, tendo em vista que a WebSTAMP não possui testes automatizados.

Laravel (Laravel, 2011) é um *framework* PHP livre e de código aberto que utiliza o padrão arquitetural MVC (*Model*, *View*, *Controller*) para o desenvolvimento de aplicações *web*. Laravel utiliza um recurso conhecido como *Migration* (traduzido como migração neste trabalho) que possibilita a manipulação (criação, alteração e remoção) de tabelas em um banco de dados. Com a *migração*, é possível alterar o esquema de dados sem ser necessária a utilização da linguagem SQL (Structured Query Language), amplamente utilizada em SGBDRs (Sistema de Gerenciamento de Banco de Dados Relacionais).

MariaDB (MariaDB, 2009) é um SGBD gratuito que utiliza a linguagem SQL para consultas e criação de bancos de dados. MariaDB pode ser integrado com PHP e, consequentemente, com Laravel. Neste trabalho, usamos MariaDB, juntamente com PHPUnit, para realização dos testes de integração com o banco de dados.

PHPUnit é um *framework* de testes de unidade e integração para a linguagem de programação PHP. Usamos testes de unidade para testar de forma isolada os menores componentes possíveis presentes na WebSTAMP (i. e. classes). Já em relação aos testes de integração, o PHPUnit possui diversas asserções e *mocks* para os dublês de testes, não sendo necessária a utilização de outros *frameworks* como Mockery (2015) e Phockito. O Quadro 4.1 apresenta as tecnologias que a WebSTAMP utiliza e as quais foram utilizadas para realização deste trabalho, juntamente com seus nomes e respectivas versões.

Tecnologia	Versão (WebSTAMP)	Versão (este trabalho)
Php	5.6.37	7.1.30
Laravel	5.2.45	5.8.35
PhpUnit	-	4.8.27
MariaDB	-	10.1.38
MySql	5.7	-

Quadro 4.1 - Tecnologias e versões usadas no código original da WebSTAMP e neste trabalho
Fonte: autor da pesquisa, 2021.

Selenium Webdriver (Selenium Webdriver, 2004) é uma ferramenta utilizada para automação de testes fim-a-fim (*end-to-end*). O Selenium WebDriver faz chamadas diretamente ao navegador *web* utilizando o suporte à automação nativo de cada navegador. Dessa forma, torna-se possível simular um usuário realizando operações como abrir e fechar uma janela de navegação, inserir dados em um formulário, entre outras operações. A automação de testes com Selenium WebDriver permite validar fim-a-fim se a aplicação está funcionando conforme requisitos. No presente trabalho foi utilizado também o driver ChromeDriver (ChromeDriver, 2012), uma ferramenta de código aberto que possibilita a comunicação do Selenium WebDriver com o navegador *web* Google Chrome.

Postman (Postman, 2014) é uma ferramenta que possui como função principal realizar chamadas HTTP (HyperText Transfer Protocol) em APIs (*Application Programming Interface*) além de disponibilizar um ambiente para a documentação, execução de testes e requisições em geral. Neste trabalho, utilizamos o Postman para realizar chamadas HTTP na camada *back-end* da WebSTAMP no processo de refatoração sem a presença do *front-end*, pois refatoramos o *back-end* inserindo as camadas de *Service* e *Repository*, tornando possível visualizar e verificar o retorno dos dados da API após a refatoração.

5 PROPOSTA DE MELHORIAS DE *DESIGN* NO MÓDULO *BACK-END* DA WEBSTAMP

A WebSTAMP possui diversos problemas de *design* no seu módulo servidor (*back-end*). Dentre os problemas de *design*, destaca-se o alto acoplamento entre os componentes que constituem o módulo servidor, o que dificulta a criação de testes, manutenção de código, evolução da WebSTAMP. No intuito de endereçar o problema de acoplamento entre os componentes, além de outros problemas de *design* presentes na WebSTAMP, tais como baixa coesão e ausência de *patterns*, promovemos uma série de melhorias, discutidas neste capítulo.

5.1 PROPOSTA DE CRIAÇÃO DAS CAMADAS *SERVICE* E *REPOSITORY* NO *DESIGN* DA WEBSTAMP

Os controladores são classes responsáveis por encapsular os dados oriundos das requisições HTTP no *framework* Laravel, assim como em diversos outros *frameworks* que utilizam o padrão de arquitetura MVC.

Atualmente, os controladores da WebSTAMP, além de encapsular dados das requisições HTTP, também são responsáveis pela lógica de negócios relacionada à análise de situações de perigo e integração com a camada de persistência. Em outras palavras, os controladores atuais da WebSTAMP não são coesos e possuem alto acoplamento (LARMAN, 2012).

Uma das possíveis soluções para minimizarmos o acoplamento e aumentarmos a coesão dos *controladores* se dá por meio da criação de camadas lógicas adicionais na *WebSTAMP*, tais como uma camada com componentes que encapsulam a lógica de negócios e uma camada que engloba lógica necessária para persistência de dados em um SGBD.

Service Layer (*Camada de Serviço*) é um padrão arquitetural para *software* desenvolvido em camadas (Evans, 2003). Uma *Service Layer* contém componentes de *software* que recebem os dados (objetos) encapsulados pelos controladores MVC e executam as regras de negócios presente em uma aplicação.

Repository Layer (*Camada de Repositório*) é uma camada situada entre a *Camada de Serviço* e o SGBD, permitindo realizar ações de inserção, alteração, consulta e remoção de dados em um banco de dados (Fowler, 2002).

Com o uso da *Camada Repository* conforme descreve Fowler *(2004)* juntamente com o conceito de injeção de dependências presente no *framework* Laravel, é possível realizar a criação de testes de unidade. Isso se deve ao fato de ser possível injetar objetos *mocks* na classe em teste (SUT), por meio do conceito de injeção de dependência e deixá-la totalmente isolada. Além disso, é possível reduzir o acoplamento e aumentar a coesão dos objetos de *software* da WebSTAMP, deixando cada classe responsável por realizar ações que cabem somente a ela. Dessa forma, manutenções no código da WebSTAMP são realizadas com maior facilidade.

5.2 MODELO DE *DESIGN* EM CAMADAS DO *BACK-END* DA WEBSTAMP UTILIZANDO UML

Esta seção apresenta uma proposta de modelo de *design* para o módulo *back-end* da WebSTAMP visualizada a partir dos diagramas UML de Sequência e Implantação (*Deployment)*.

A Figura 5.1 ilustra o Diagrama de Sequência UML para a operação de sistema Larman, (2012) *"Criar novo projeto"*. Uma operação de sistema consiste em uma funcionalidade que o usuário pode realizar para criar/modificar um objeto no domínio do sistema, sendo no caso a criação de um novo projeto. Este exemplo foi escolhido, pois serve de base para a criação de um projeto inicial na WebSTAMP. Um projeto é equivalente a uma análise de situações de perigo e sua criação é normalmente a primeira funcionalidade executada por um *Safety Analyst* na WebSTAMP.

Conforme pode ser visto na Figura 5.1 o objeto do tipo `ProjectController` recebe uma requisição HTTP com os dados de um projeto, cria instâncias de um `Project` e delega para o objeto *service* do tipo `ProjectService`. O *service* por sua vez, delega os dados para o objeto *repository* do tipo `ProjectRepository`, que por fim realiza a inserção dos dados no banco de dados.

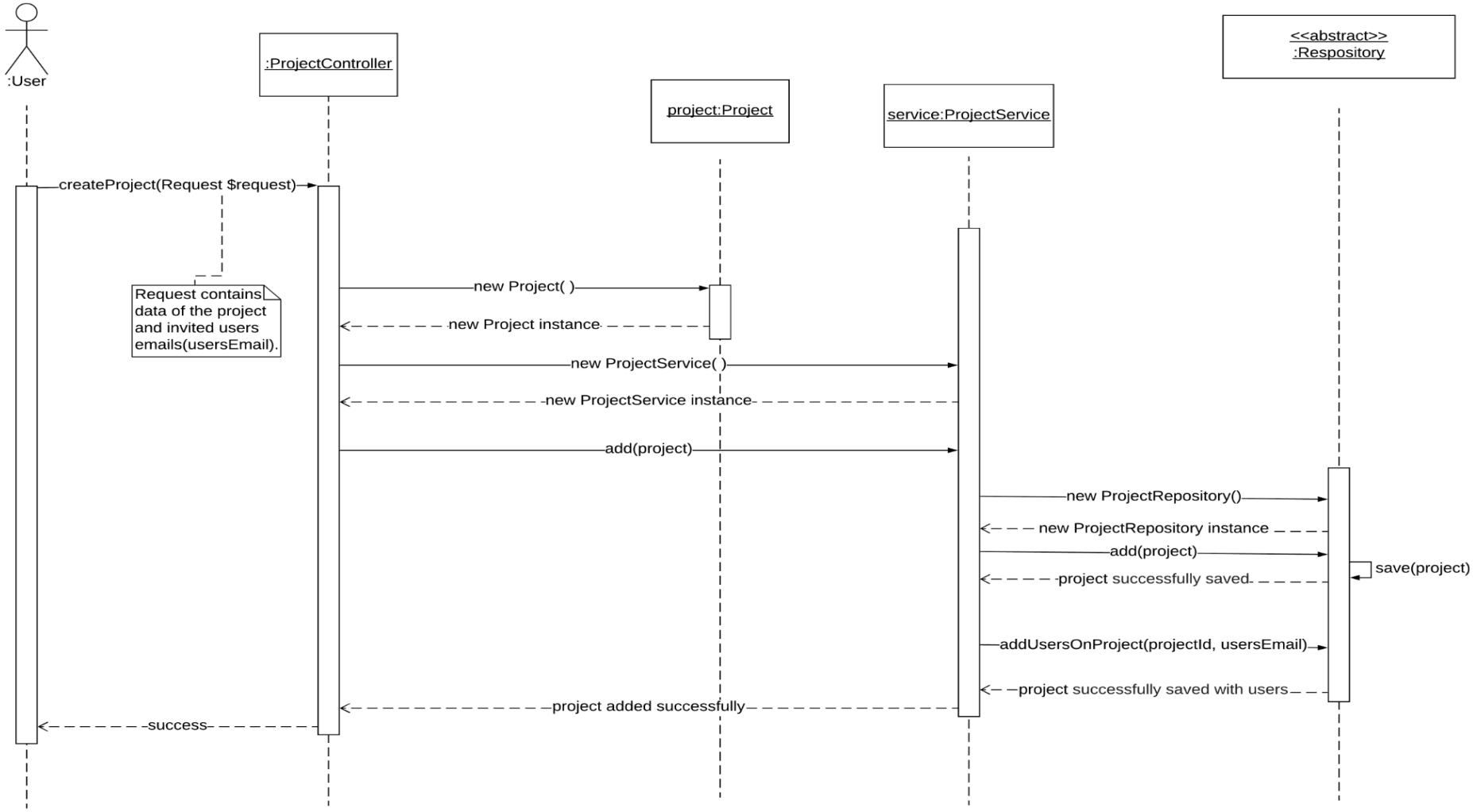

Figura 5.1 - Proposta de diagrama de sequência UML no caso particular da classe `Project` na operação de sistema “Criar novo projeto”
Fonte: autor da pesquisa, 2021.

A Figura 5.2 ilustra um Diagrama de Implantação (*deployment)* UML implementado para a proposta de melhoria do *design* da WebSTAMP. Considerando a operação de sistema "Criar novo projeto" (Figura 5.1), após a ferramenta receber informações nos controladores, via verbos HTTP, o componente `ProjectController` cria o objeto de modelo (`Project`) com base em parâmetros da requisição HTTP e o repassa para a camada *Service* (`ProjectService`). Os verbos HTTP são um conjunto de métodos de requisições (POST, GET, PUT, DELETE, etc.), disponibilizados pelo protocolo HTTP, responsáveis por indicar a ação a ser realizada para um dado recurso. Na camada de serviços, ocorre a execução da lógica de negócios e, após a execução, o fluxo de controle é redirecionado para a *Camada Repository* (classe `ProjectRepository`). Por fim, a classe `ProjectRepository` realiza a persistência dos dados no banco de dados.

Por fim, A Figura 5.3, apresenta um diagrama de implantação genérico que pode ser usado para qualquer classe de modelo da WebSTAMP substituindo a palavra *Entity* (em azul na figura) pela respectiva classe de modelo (`Loss`, `Hazard`, `SystemGoal`, etc).

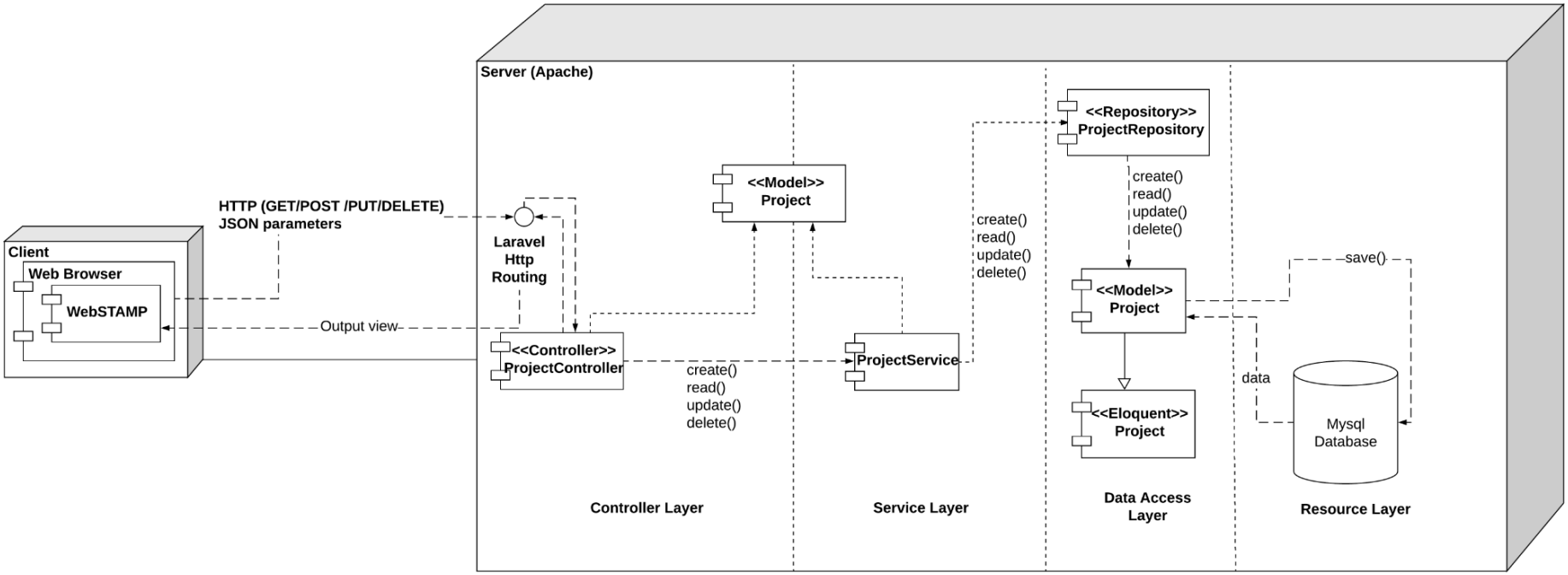

Figura 5.2 - Proposta de diagrama de *deployment* UML no caso particular da classe `Project`
Fonte: autor da pesquisa, 2021.

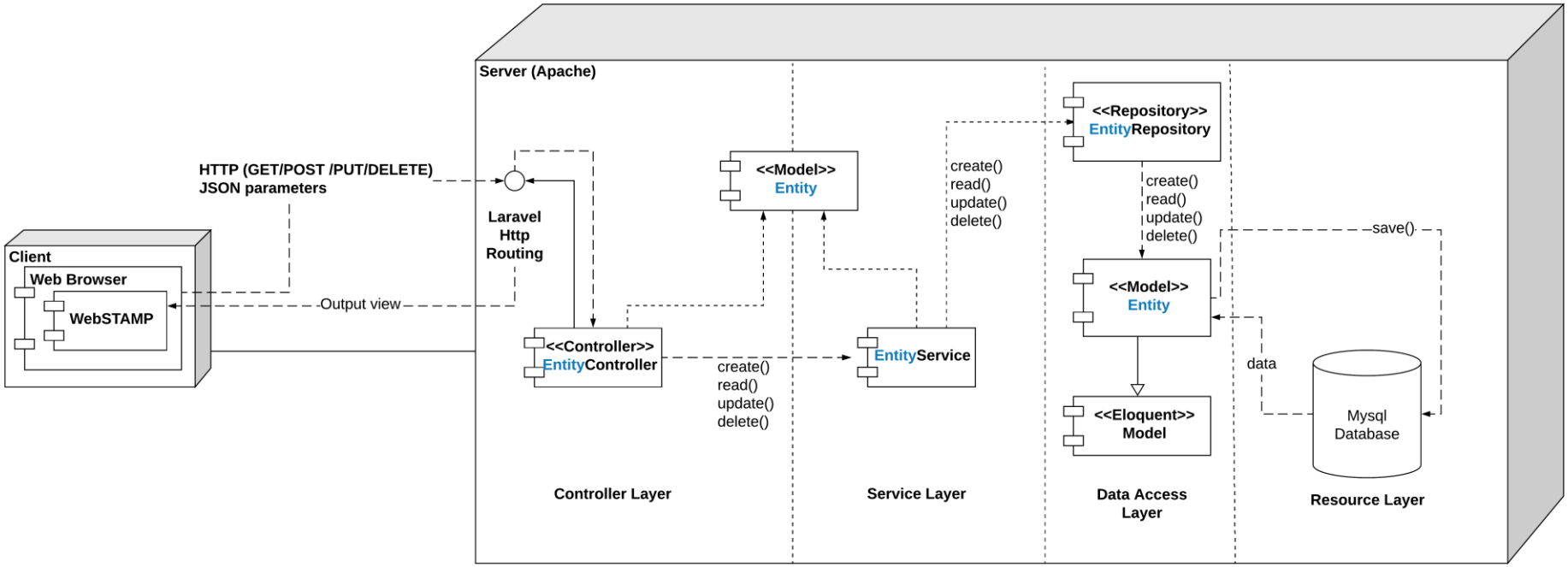

Figura 5.3 - Diagrama de *deployment* UML genérico para proposta de melhorias no design para a WebSTAMP.
Fonte: autor da pesquisa, 2021.

5.3 PROPOSTA DE UMA CAMADA EM *SOFTWARE* PARA AUTORIZAÇÃO

Implementamos uma camada de *software* que oferece serviço de autorização na WebSTAMP. O serviço de autorização é baseado em papéis. Um papel determina um conjunto de habilidades necessárias para o desempenho de determinada atividade. Definimos os papéis de *Safety Analyst*, *Especialista em Situações de Perigo* (*Safety Specialist*) e *Administrador* (*Administrator*) como exemplo para verificar a corretude da nova camada implementada. Dessa maneira, o *Administrador* é responsável por conceder privilégios a cada um dos papéis. Definimos dois privilégios: *escrita* e *leitura*. Um papel com privilégio de escrita consegue editar qualquer parte de uma análise de situação de perigos na WebSTAMP. O privilégio de leitura concedido a um papel, elimina a possibilidade de alterações nas análises com WebSTAMP.

A camada de autorização foi implementada para prover segurança aos usuários da WebSTAMP e evitar violação de integridade dos dados por meio da edição de projetos por integrantes que não possuem um perfil com permissão de escrita. Essa camada é responsável por proteger os dados contra alterações presentes em um projeto por quaisquer usuários. Dessa forma, um usuário poderá somente visualizar e realizar, por exemplo, alterações na camada de persistência, se e somente se possuir o papel que possui a autorização para estas ações em um projeto.

A Figura 5.4 ilustra as tabelas novas criadas no banco de dados (exceto a users), para que seja possível realizar a criação de papéis e permissões na camada de autorização.

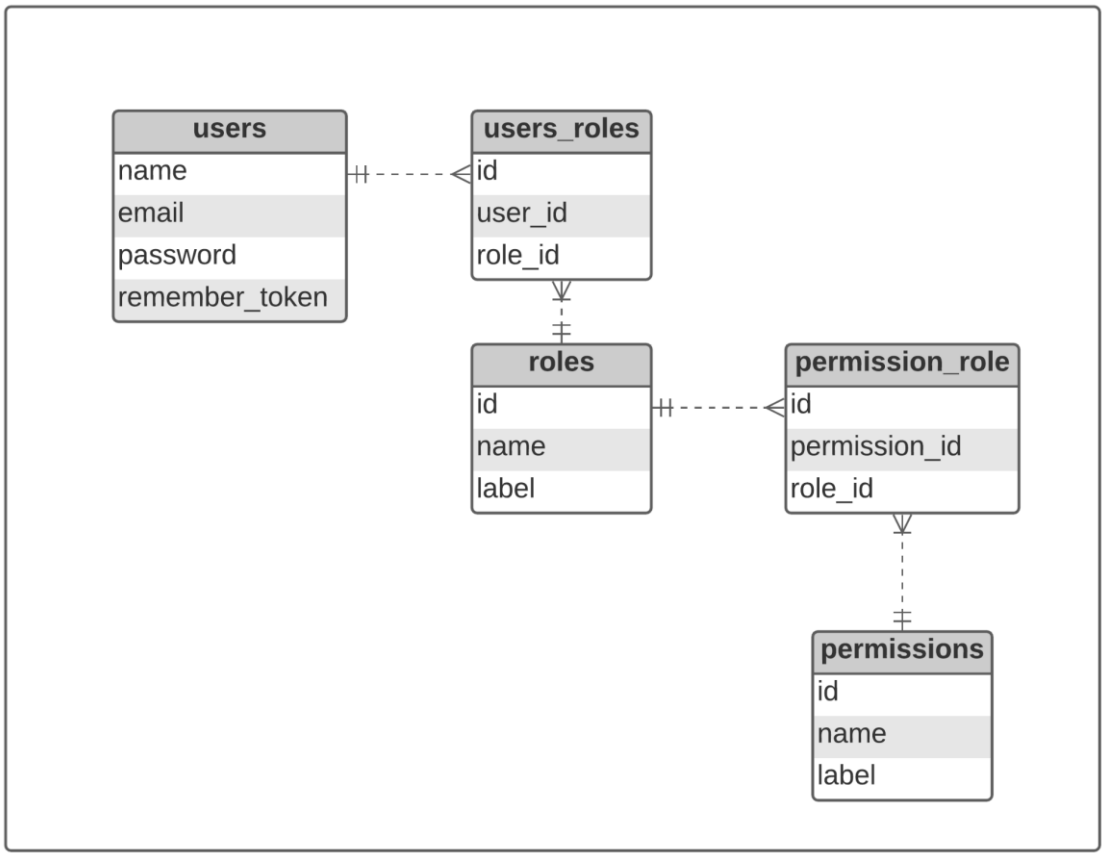

Figura 5.4 - Diagrama Entidade Relacionamento utilizado na camada de autorização
Fonte: autor da pesquisa, 2011.

A Figura 5.5 ilustra a utilização da camada de autorização, por meio de um recurso do Laravel chamado *Gate*, que são simplesmente fechamentos (*closures*) que determinam se um usuário está autorizado a executar uma determinada ação. No caso da Figura 5.5, pode-se ver a utilização da autorização entre as Linhas 02 e 04. Dessa forma, se o atual usuário autenticado na WebSTAMP não possuir permissão de escrita, ele será redirecionado a uma rota de erro (Linha 03). Caso o usuário possua permissão (Linhas 05 a 10), será executado o fluxo normal do método.

```
01  public function edit(Request $request, $id) {
02    if (Gate::denies('write', [auth()->user()])) {
03      return redirect('error');
04    }
05    $project = $this->read($id);
06    $project->name = $request->input("name");
07    $project->description = $request->input("description");
08    $project->URL = str_slug($project->name . " " . strval($project->id),
09                             '-');
10    return $this->projectService->edit($project);
11  }
```

Figura 5.5 - Exemplo da utilização do recurso de autorização
Fonte: autor da pesquisa, 2021

5.4 PROPOSTA DE MELHORIAS NO *DESIGN* DA WEBSTAMP

Partindo da premissa que um dos benefícios de refatoração consiste em alterações/modificações do sistema para melhorias de *design*, houve uma mudança significativa em relação aos códigos presentes atualmente na WebSTAMP. Em particular, o código presente na proposta deste trabalho possui maior proximidade a um modelo orientado a objetos (Larman, 2012) do que a um modelo relacional (Elmasri e Shamkant, 1994), como na implementação original. Código aderente a princípios da orientação a objetos, tais como alta coesão e baixo acoplamento, possui maior legibilidade e facilita a atividade de testes e inclusão de novos requisitos.

Em alguns casos realizamos refatorações para eliminar classes redundantes presentes na versão atual da WebSTAMP. Classes redundantes na WebSTAMP são utilizadas para realizar operações de interseção entre tabelas de um modelo relacional. Dessa forma, as classes redundantes foram eliminadas, pois alterações feitas no banco de dados já realizam as associações automaticamente. Além disso, exclusões de registros que eram feitos de maneira manual, foram eliminadas.

A seguir, é mostrado alguns exemplos de melhorias no código, comparando antes e depois da nova proposta, em conjunto com as camadas citadas anteriormente.

O código da Figura 5.6 representa o processo original da criação de membros (usuários) para um mesmo projeto (classe `Project`) na WebSTAMP. Conforme pode ser visto na Linha 01 da figura, esta classe possui uma função chamada `add (Request $request)`. Nesta função, ocorre a criação de um objeto do tipo `Project` com suas propriedades configuradas com os valores oriundos da requisição HTTP (parâmetro `$request` - Linha 01) entre as Linhas 02 e 05. Na Linha 06 realiza-se o armazenamento do projeto no banco de dados. Por fim, entre as Linhas 07 e 09 é aplicado uma lógica para criação do endereço (URI) de localização do projeto e nas Linhas 10 a 20 são feitas as associações de usuários (*Users*) ao novo projeto que está sendo criado. Este método, apesar de poucas linhas de código, apresenta baixa legibilidade, alto acoplamento (podendo ser observado entre as Linhas 11 e 20 devido a dependência com as classes `User` e `Team`), o que dificulta a atividade de testes.

```
01  public function add(Request $request) {
02    $project = new Project();
03    $project->name = $request->input('name');
04    $project->type = $request->input(type);
05    $project->description = $request->input(description);
06    $project->save();
07    $url = $project->name . " " . $project->id;
08    $project->URL = str_slug(url, '-');
09    $project->save();
10    $users = explode(";", $request->input('shared'));
11    foreach($users as $user)      {
12      $team = new Team();
13      $getUserId = User::where('email', $user)->get();
14      $team->user_id = 0;
15      foreach ($getUserId as $userid) {
16        $team->user_id = $userid->id;
17      }
18      $team->project_id = $project->id;
19         $team->save();
20      }
21     return redirect(route('projects'));
22  }
```

Figura 5.6 - Código atual para criação de um `Project` no `Controller`
Fonte: autor da pesquisa, 2021.

Dessa forma, alteramos o *design* do método `add` distribuindo suas responsabilidades para as camadas de *Service* e *Repository*, no intuito de termos melhor legibilidade, facilidade de manutenção e melhor testabilidade. O trecho explicitado na Figura 5.7 pertence ao controlador de criação de um projeto (`ProjectController`) que recebe dados e realiza a delegação destes para a camada de *Serviço*. Nas Linhas 02 a 06 ocorre a criação do objeto `Projeto`, com os dados oriundos da requisição (parâmetro `$request` - Linha 01) e este é repassado às demais camadas pela Linha 07.

```
01  public function add(Request $request) {
02    $project = new Project();
03    $project->name = $request->input("name");
04    $project->description = $request->input("description");
05    $project->type = $request->input("type");
06    $project->users = array_filter(explode(";", $request->shared));
07    return $this->projectService->add($project);
08  }
```

Figura 5.7- Código novo para criação de um `Project` no `Controller`
Fonte: dados da pesquisa, 2021

Por sua vez, a camada *Service* (Figura 5.8) realiza a lógica de negócios com auxílio da camada *Repository*.

```
01  public function add(Project $project) {
02    $usersEmail = $project->users;
03    unset($project->users);
04    $projectSaved = $this->projectRepository->add($project);
05    $project->URL = str_slug($project->name . " " .
06                    strval($projectSaved->id), '-');
07    $this->projectRepository->update($project);
08    return $this->addUsersOnProject($projectSaved->id, $usersEmail);
09  }
```

Figura 5.8 - Método da classe `Project` na camada *Service* para criação de um projeto
Fonte: autor da pesquisa, 2021.

Por fim, na camada *Repository* (Figura 5.9) ocorre a persistência dos dados no Banco de dados, retornando o resultado da operação para as camadas *Service* e *Controller* por meio da instrução `return`. Dessa forma, a Linha 02 realiza a inserção dos dados no banco de dados e as Linhas 03 e 04 realizam o retorno do projeto salvo no banco de dados em um objeto juntamente com seus usuários associados.

```
01  public function add($project)  {
02     parent::save($project);
03     return Project::where('id',
04     Project::max('id'))->with('users')->get()->first();
05  }
```

Figura 5.9 - Código atual para criação de um `Project` na camada *Repository*
Fonte: autor da pesquisa, 2021.

Por último, realizamos soluções semelhantes para todas as demais classes presentes no modelo de domínio da WebSTAMP, tais como `Hazards`, `Losses`, `SystemGoals`, `ControlActions` etc.

6 RECOMENDAÇÕES DE UMA SUÍTE DE TESTES AUTOMATIZADOS PARA O *BACK-END* DA WEBSTAMP

Este capítulo apresenta uma proposta de um conjunto (*suíte*) de testes automatizados para o módulo servidor da WebSTAMP (*back-end*). A *suíte* de testes tem como objetivo verificar a corretude das implementações no que tange à aderência aos requisitos da WebSTAMP.

A presença de testes, sejam de *unidade*, *integração* ou *fim-a-fim*, traz confiabilidade e qualidade ao código, facilitando a realização de manutenções, pois pode se pôr a prova o que foi produzido e alterado. Dessa forma, além das melhorias descritas no Capítulo 5, desenvolvemos uma coleção (suíte) de testes para a WebSTAMP.

Em nossa proposta, os *testes de unidade* estão presentes nas camadas de serviço. Um *teste de unidade*, segundo Kaczanowski (2013), não pode interagir com o banco de dados; por isso, usamos *mocks* para simular o mecanismo de persistência de dados na WebSTAMP.

A Figura 6.1 apresenta um exemplo de teste de unidade na criação de um `Project` na WebSTAMP. Conforme pode ser visto na figura, nas Linhas 02 até a 07 ocorre a criação do objeto `Project`. O objeto do tipo `Project` trafega pelos demais componentes da WebSTAMP (controlador, serviço e de repositório). Das Linhas 09 a 14, são criados dois objetos *Mocks*, que possuem como responsabilidade "*imitar*" o comportamento de um `ProjectRepository` e um `UserRepository`, e na Linha 16 ocorre a criação do *Service* com a injeção dos objetos *Mocks* passados como parâmetro no construtor. Dessa forma, quando um `ProjectRepository` receber a chamada do método `add`, este retornará o objeto do tipo `Project`. Comportamento análogo acontece na chamada do método `addUsersOnProject`, `getIdByEmail` (Linhas 12 a 14). Após os passos anteriormente mencionados, ocorre a chamada ao *Service* (Linha 17) e verificação dos dados de saída nas Linhas 18 a 19.

Os *testes de integração* estão presentes na verificação da persistência no banco de dados. Como pode ser visto na Figura 6.2, entre as Linhas 02 e 07, ocorre a criação do objeto `Project` oriundo da camada *Controller* e delegado para a camada *Service* para persistência no banco de dados. As Linhas 09 e 10 representam a criação de um objeto *Service* e a chamada ao método `add` para inserção do objeto `Project` no banco de dados. Após os passos anteriormente mencionados, ocorre a chamada no banco de dados da Linhas 11 a 18 para verificar se os dados foram inseridos corretamente e se os dados que estão presentes no banco de dados são os

mesmos solicitados na inserção. Por fim, vale a pena reforçar que os testes foram executados no banco de dados da máquina local, pois no banco de dados usado em produção estão os dados utilizados por usuários/clientes da aplicação, e se realizarmos algum teste, por exemplo de alterar/remover dados, perderemos dados reais de clientes.

```
01 public function test_create_project(){
02   $projectInsert = new Project();
03   $projectInsert->name = "Insulin Pump Analysis";
04   $projectInsert->description =  "Project to test the doses insulin pump";
05   $projectInsert->type = "Security";
06   $projectInsert->URL = "insulin-pump-analysis-30";
07   $projectInsert->users = ["teste@teste.com.br"];
08   //Create a mock ProjectRepositoryClass.
09   $mockProjectRepository = $this->createMock(ProjectRepository::class);
10   $mockUserRepository    = $this->createMock(UserRepository::class);
11   //configure the mock.
12   $mockProjectRepository->method('add')->willReturn($this->project);
13   $mockProjectRepository->method('addUsersOnProject')->
14   willReturn($this->projectWithUsers);
15   $mockUserRepository->method('getIdByEmail')->willReturn($this->user->id);
16   $projectService = new ProjectService($mockProjectRepository,
17 $mockUserRepository);
18   $project = $projectService->add($projectInsert);
19   $this->assertEquals($projectInsert->name, $project->getAttribute('name'));
20   $this->assertEquals($projectInsert->type, $project->getAttribute('type'));
   }
```

Figura 6.1 - Exemplo de teste de unidade para criação de um projeto na camada *Service* juntamente com *mock* da camada *Repository*
Fonte: autor da pesquisa, 2021.

```
01 public function test_create_project(){
02   $projectInsert = new Project();
03   $projectInsert->name = "Insulin Pump Analysis";
04   $projectInsert->description =  "Insulin Pump Analysis";
05   $projectInsert->type = "Security";
06   $projectInsert->URL = "insulin-pump-analysis-30";
07   $projectInsert->users = ["teste@teste.com.br"];
08
09   $projectService = new ProjectService(new ProjectRepository());
10   $projectSaveDataBase = $projectService->add($projectInsert);
11   $this->assertDataBaseHas("projects",  [
12             'name' => $projectInsert->name
13   ]);
14   $this->assertEquals($projectInsert->name, $projectSaveDataBase->name);
15   $this->assertEquals($projectInsert->description,
16                       $projectSaveDataBase->description);
17   $this->assertEquals($projectInsert->users[0]->email,
18   $projectSaveDataBase->users[0]->email);
19 }
```

Figura 6.2 - Exemplo de teste de integração para criação de um projeto (objeto do tipo `Project`) verificando a existência da criação no banco de dados
Fonte: autor da pesquisa, 2021.

Testes *fim-a-fim* estão presentes na verificação do comportamento do *front-end* e também da saída da WebSTAMP após processamento das requisições, desde o *controlador* até a *camada de persistência* e devolução dos resultados. No presente trabalho vale evidenciar, que para execução dos testes *fim-a-fim,* foi utilizado parte do *front-end* do projeto original.

A Figura 6.3 apresenta um exemplo de teste *end-to-end* simulando um usuário sem permissão de escrita na WebSTAMP. O comportamento final esperado é o término da execução no navegador *web* em uma tela com a mensagem de acesso não autorizado.

Como pode ser visto na figura, nas Linhas 03 a 09, ocorre uma preparação no banco de dados, para que toda vez que o teste for executado, no momento de registrar um novo usuário e um novo projeto, não haja erro de duplicidade, sendo possível executar o teste N vezes. Nas Linhas 11 a 13, ocorrem algumas configurações para o teste que envolvem o Selenium e o *driver* do Google Chrome. A partir da Linha 14, onde é solicitado a maximização da janela do navegador, até a Linha 124, onde termina-se a simulação de usuário navegando na WebSTAMP, ocorre todo o processo do teste *end-to-end.*

Neste teste, podemos destacar a grande utilização de expressões *XPath* para localização de elementos HTML nas páginas navegadas (exemplo nas Linhas 77, 85 e 93) e também a associação do novo usuário criado (Linhas 26 a 63), com um perfil que não possui permissão para remover o novo projeto criado no banco de dados (Linhas 112 e 117). Além disso, podemos destacar também a grande utilização da função *sleep* (exemplo nas linhas 20, 41 e 49), na qual tem o papel de fazer com que o Selenium espere alguns segundos para continuar sua execução, fazendo com que as páginas possam ser carregadas completamente e, assim não causando nenhum erro de propriedade não encontrada durante a execução.

Por fim, entre as Linhas 128 e 142, ocorrem as verificações do teste para validar a resposta produzida contrastando com a resposta esperada (Linha 128) e para validação da presença dos dados no banco de dados (Linhas 134 a 142).

```
01  public function teste_create_user_without_permission_in_project(){
02
03  $user = DB::table('users')->where('email', 'pedro_1998@gmail.com')->get();
04
05  if(count($user) > 0){
06        DB::table('users')->where('email', 'pedro_1998@gmail.com')->delete();
07        DB::table('projects')->where('name', "Nome Projeto com Selenium
08  Usuario sem Acesso de Escrita")->delete();
09  }
10
11  $host = 'http://localhost:4444/wd/hub';
12  $driver = RemoteWebDriver::create($host, DesiredCapabilities::chrome());
13  $mouse = $driver->getMouse();
14  $driver->manage()->window()->maximize();
```

```
15
16  sleep(2);
17
18  $driver->get("http://localhost:8000/");
19
20  sleep(2);
21
22  $botaoRegister = $driver->findElement(
23      WebDriverBy::xpath("/html/body/nav/div[6]/div/a")
24  );
25
26  $mouse->click($botaoRegister->getCoordinates());
27
28  $inputName = $driver->findElement(
29      WebDriverBy::id("name")
30  );
31  $inputEmail = $driver->findElement(
32      WebDriverBy::id("email")
33  );
34  $inputPassword = $driver->findElement(
35      WebDriverBy::id("password")
36  );
37  $inputPasswordConfirm = $driver->findElement(
38      WebDriverBy::id("password-confirm")
39  );
40
41  sleep(2);
42
43  $inputName->sendKeys("Pedro Lopes");
44
45  sleep(2);
46
47  $inputEmail->sendKeys("pedro_1998@gmail.com");
48
49  sleep(2);
50
51  $inputPassword->sendKeys("selenium123");
52
53  sleep(2);
54
55  $inputPasswordConfirm->sendKeys("selenium123");
56
57
58  $inputPasswordConfirm = $driver->findElement(
59      WebDriverBy::xpath("/html/body/div[1]/div/form/div[5]/button")
60  );
61
62
63  $mouse->click($inputPasswordConfirm->getCoordinates());
64
65  sleep(5);
66
67  $botaoNomeUsuario = $driver->findElement(
68    WebDriverBy::xpath("/html/body/nav/div[2]/div[1]/a")
69  );
70
71  $action = new WebDriverActions($driver);
72  $action->moveToElement($botaoNomeUsuario)->perform();
73
74  sleep(2);
75
76  $botaoMyProjects = $driver->findElement(
77      WebDriverBy::xpath("/html/body/nav/div[2]/div[2]/div[1]/a")
78  );
79
80  $mouse->click($botaoMyProjects->getCoordinates());
81
```

```
82  sleep(3);
83
84  $botaoCriarProjeto = $driver->findElement(
85      WebDriverBy::xpath("/html/body/div[1]/div[1]/div[2]")
86  );
87
88  $mouse->click($botaoCriarProjeto->getCoordinates());
89
90  sleep(2);
91
92  $driver->findElement(
93      WebDriverBy::xpath("/html/body/div[3]/div[2]/form/div[1]/div[2]/input")
94  )->sendKeys("Nome Projeto com Selenium Usuario sem Acesso de Escrita");
95
96  sleep(2);
97
98  $driver->findElement(
99    WebDriverBy::xpath("/html/body/div[3]/div[2]/form/div[1]/div[3]/textarea")
100 )->sendKeys("Descrição Projeto com Selenium Usuario sem Acesso de Escrita");
101
102 sleep(2);
103
104 $botaoAdicionarProject = $driver->findElement(
105     WebDriverBy::xpath("/html/body/div[3]/div[2]/form/div[2]/button")
106 );
107
108 $mouse->click($botaoAdicionarProject->getCoordinates());
109
110 sleep(2);
111
112 $idUser  = DB::table('users')->where('email', 'pedro_1998@gmail.com')-
113 >first()->id;
114
115 DB::insert('insert into user_roles (user_id, role_id) values (?, ?)',
116 [$idUser, 2]);//Adicionando o usuário como Safety Specialist, pois o perfil
117 Safety Specialist possui acesso de escrita
118
119 $botaoDeleteProject = $driver->findElement(
120 WebDriverBy::xpath("/html/body/div[1]/div[1]/div[3]/ul/li/div[2]/form[3]/div
121 /input[4]")
122 );
123
124 $mouse->click($botaoDeleteProject->getCoordinates());
125
126 sleep(4);
127
128 $this->assertEquals($driver->getTitle(), "Forbidden");
129
130 sleep(5);
131
132 $driver->close();
133
134 $this->assertDatabaseHas("users", [
135         'name' => "Pedro Lopes",
136         'email' => "pedro_1998@gmail.com"
137     ]);
138
139 $this->assertDatabaseHas("projects", [
140         'name' => "Nome Projeto com Selenium Usuario sem Acesso de Escrita"
141     ]);
142 }
```

Figura 6.3 - Exemplo de teste *end-to-end* escrito com Selenium para verificação se um usuário sem permissão não consegue remover um projeto
Fonte: autor da pesquisa, 2021.

Além desse teste de *end-to-end* com Selenium, um teste de integração simples foi realizado na ferramenta Postman para verificar a corretude dos dados devolvidos da aplicação após a refatoração. Como pode ser observado na Figura 6.4, a Linha 01 é responsável por converter o retorno da aplicação da WebSTAMP em um JSON. Nas Linhas 02 a 08, ocorrem asserções a fim de verificar se o retorno da aplicação está correto ou não de acordo com os dados enviados na requisição.

```
01  var jsonData = JSON.parse(responseBody);
02  tests["Status code correto."] = responseCode.code === 200;
03  postman.setEnvironmentVariable("id_project_criado", jsonData.id);
04  tests["Nome projeto criado correto."] = "Nome Projeto com Postman" ===
05  jsonData.name;
06  tests["Description projeto criado correto."] = "Descrição Projeto com Postman"
07  === jsonData.description;
08  tests["Type projeto criado correto."] = "Security" === jsonData.type;
```

Figura 6.4 - Exemplo de teste de integração com a ferramenta Postman
Fonte: autor da pesquisa, 2021.

Foram realizados processos análogos para criação de testes para outras classes presentes no sistema. O Apêndice A apresenta uma lista completa de todos os testes automatizados criados para a WebSTAMP.

Por fim, a Figura 6.5 exibe o resultado da execução dos testes de unidade e de integração, sendo executado 51 testes e 71 asserções.

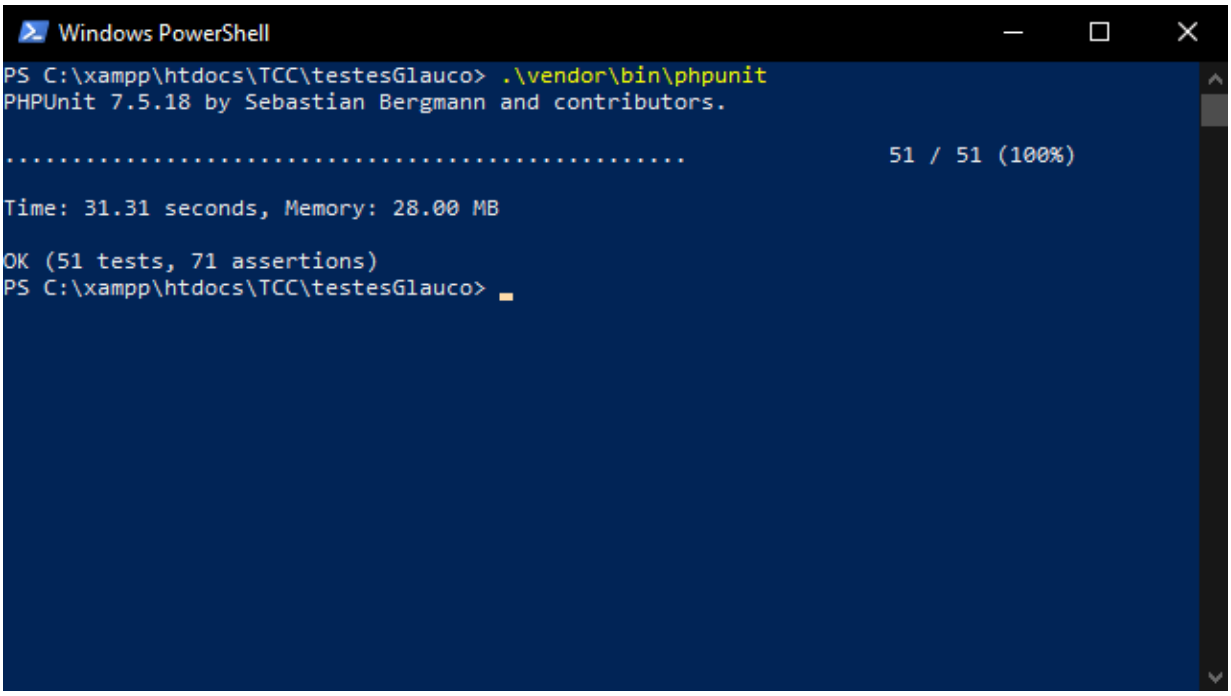

Figura 6.5– Resultado execução testes de unidade e integração
Fonte: autor da pesquisa, 2021.

7 DISCUSSÕES E TRABALHOS FUTUROS

Este capítulo apresenta algumas discussões relacionadas ao presente trabalho realizado, juntamente com propostas para trabalhos futuros.

Acreditamos que a proposta de melhorias no *design* em conjunto com uma suíte de testes automatizados tem o potencial de facilitar a evolução da WebSTAMP. Nossa crença é fundamentada na literatura discutida neste trabalho e na prática durante o desenvolvimento, em especial com a possibilidade de execução de testes de regressão de forma automatizada.

As camadas *Service* e *Repository (patterns)* auxiliam na atividade de testes e também nas manutenções, pois consegue-se realizar o processo de *Mock* do banco de dados. O aumento de coesão e baixo acoplamento entre os componentes presentes nas camadas ajudam na organização da ferramenta, além de ajudar também na identificação de erros.

A WebSTAMP é uma aplicação em constante evolução, desenvolvida paralelamente a este trabalho (embora o autor deste trabalho tenha participado de algumas reuniões com o time de desenvolvimento da WebSTAMP). Dessa forma, este trabalho foi desenvolvido de forma independente. Portanto, existe a possibilidade de que a versão atual da WebSTAMP já tenha recebido alguma atualização, em paralelo com o desenvolvimento do presente trabalho, possibilitando que alguma atividade deste já tenha sido implementada.

Outro ponto que pode ser destacado é a reutilização, sem nenhuma alteração, de parte do *front-end* da WebSTAMP para o teste *end-to-end*, pois nossa proposta foi relacionada somente a camada *back-end*, e com o *front-end* pré-existente do projeto original foi possível realizar os testes *end-to-end* propostos. Também no teste *end-to-end* houve muita utilização das expressões XPath para encontrar os elementos nas páginas, pois diversos elementos com com mesmos ID`s nas páginas dificultavam a seleção do elemento específico que estávamos querendo possuir controle no teste.

Ainda, sobre a criação da camada de autorização, que possui influência no *front-end*, deixamos para os mantenedores da WebSTAMP decidirem a criação dos papéis, juntamente com a recomendação da criação de um painel para *Administrador* para gerenciar os usuário e permissões, na qual temos muitos disponíveis no mercado como adminLTE (AdminLTE, 2014), Ample Admin (Ample Admin, 2018) e Star Admin (Star Admin, 2017).

Ademais, vale citar que o SGBD utilizado no presente trabalho é o MariaDB, um projeto *fork* que nasceu baseado no MySQL e é sempre atualizado com o mesmo. A escolha deste para utilização no trabalho não foi baseada em nenhum requisito não funcional como desempenho

ou segurança por exemplo, mas por apenas ser o banco de dados que havia sido instalado na máquina de desenvolvimento. E com o recurso *migrations* do framework Laravel, o banco de dados é indiferente, podendo ser Mysql, MariaDB, PostgreSQL etc. O mesmo ponto sobre a escolha do MariaDB vale para as versões do PHP e Laravel, do PHP sendo escolhido por apenas ser a versão que havia disponível na máquina de desenvolvimento e do Laravel a versão disponível no site.

Como trabalhos futuros para a WebSTAMP, sugerimos a possibilidade da criação de microsserviços para o *back-end* e a criação do *front-end* em micro *front-end*, além de uma inspeção do código refatorado com a ferramenta SonarQube(SonarQube, 2006) e uma avaliação com a equipe de WebSTAMP a fim de verificar o tempo e esforços das manutenções futuras.

As arquiteturas em microsserviços e micro *front-end* consistem em pequenos serviços independentes que se comunicam entre si utilizando APIs. Elas tornam o desenvolvimento mais fácil, permitindo os desenvolvedores trabalharem em pontos isolados sem a necessidade de qualquer orquestração complicada entre os serviços. Além disso, oferecem maior escalabilidade, maior disponibilidade - cada serviço é independente, ou seja, uma determinada parte do serviço pode parar, mas outras podem continuar -, além de possuir diversas tecnologias altamente utilizadas no mercado, como Angular.js (Angularjs, 2010) e React.js (Reactjs, 2013) e estarem em constantes atualizações e melhorias.

Por fim, o SonarQube é uma ferramenta utilizada para inspecionar a qualidade do código com a análise estática para detecção de bugs, *bad smells*, vulnerabilidades de segurança e etc. A ferramenta também possui métricas que podem indicar níveis de acoplamento entre classes com base na análise do código, e com isso, mensuramos o trabalho desenvolvido com base nos resultados das métricas.

8 CONCLUSÃO

Desenvolver *software* com baixas probabilidades de ocorrências de erros é uma tarefa árdua que exige muita dedicação da equipe de desenvolvedores. Porém, a definição de sucesso ou fracasso consiste no controle da qualidade do sistema que se está desenvolvendo. Dessa forma, quando se trata de grandes sistemas, princípios de *design* como alta coesão e baixo acoplamento, testes e legibilidade do código devem ser levados em consideração, pois são fatores que possuem grande influência nas manutenções do *software*. Portanto, quando temos grandes sistemas, mas com ausência de qualidade de *design* e testes automatizados, a verificação da corretude dos requisitos por um trabalho manual acaba sendo custosos devido aos esforços.

Por fim, o desenvolvimento do presente trabalho possibilita concluir que a divisão da aplicação em camadas utilizando *design pattern* como o *Repository* e *Service* por exemplo, possibilita e facilita a criação de testes, aumenta a coesão e desacoplamento entre as classes e também a legibilidade do código.

Acreditamos que a proposta apresentada neste trabalho de melhorias no *design* do *back-end* da WebSTAMP em conjunto com uma suíte de testes automatizados permite que manutenções futuras da WebSTAMP sejam feitas em menor tempo e esforço. A camada de autorização baseada em papéis também proposta neste trabalho tem o potencial de aumentar a segurança (confidencialidade, integridade e disponibilidade) dos dados envolvidos nas análises (projetos) de situações de perigo criadas com a WebSTAMP.

REFERÊNCIAS

ADMINLTE, 2014. Disponível em: https://adminlte.io. Acesso em: 02 fev. 2021.

ANGULARJS, 2010. Disponível em: https://angularjs.org/. Acesso em: 02 fev. 2021.

AMPLE ADMIN, 2018. Disponível em: https://themeforest.net/item/ample-admin-ultimate-dashboard-template/19578653. Acesso em: 02 fev. 2021.

AMPATZOGLOU, A; CHATZIGEORGIOU, A; CHARALAMPIDOU, S; AVGERIOU, P. **The Effect of GoF Design Patterns on Stability: a case study**. IEEE Transactions On Software Engineering, [S.L.], v. 41, n. 8, p. 781-802, 1 ago. 2015.

ARQUILLIAN, 2010. Disponível em: https://arquillian.org/. Acesso em: 02 Nov. 2019.

BERNARDO, P C; KON, F. **A Importância dos Testes Automatizados**. Engenharia de Software Magazine, v.1, n. 3, p. 54-57. 2008.

BERTOLINO, A. **Software Testing Research: Achievements, Challenges, Dreams. Future of Software Engineering**, IEEE Computer Society. Pisa, Itália. 2007. Disponível em: http://selab.netlab.uky.edu/homepage/sw-test-roadmap-bertolino.pdf. Acesso em: 02 nov, 2019.

CARDOSO, A; ANICHE, M. **Test-Driven Development: Teste e Design no Mundo Real com PHP**. [S. l.]: A Casa do Código, 2015.

CHROMEDRIVER, 2012. Disponível em: https://chromedriver.chromium.org/home. Acesso em: 02 nov, 2019.

COLLINS, E. F.; DE LUCENA JR, V. F. **Software Test Automation Practices in Agile Development Environment: An Industry Experience Report**. In: INTERNATIONAL WORKSHOP ON AUTOMATION OF SOFTWARE TEST (AST), 7, 2012, Zurich: S, **Anais** [...] Zurich: IEE, 2012. P. 57-63. Disponível em: https://ieeexplore.ieee.org/document/6228991. Acesso em: 20 jun. 2021.

DA SILVA, D. M.; SIQUEIRA, R. D. **Framework Para Automação de Testes**. 2013. 71f. Trabalho de Conclusão de Curso (Bacharel em Ciência da Computação) Universidade Federal de Alfenas, UNIFAL, Alfenas, 2013. Disponível em: https://www.bcc.unifal-mg.edu.br/biblioteca/producao-discente/monografias/monografias-2012-2/monografia-douglas-miranda-da-silva-e-renan-domingues-siqueira/. Acesso em: 20 jun. 2021.

DBUNIT, 2020. Disponível em: http://dbunit.sourceforge.net/. Acesso em: 02 nov. 2019.

GAMMA, E.; HELM, R.; JOHNSON, R; VLISSIDES, J. **Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos**. 2. ed. [S. l]: Bookman, 2000.

ELISH, K. O.; ALSHAYEB, M. **Investigating the Effect of Refactoring on Software Testing Effort**. In: SOFTWARE ENGINEERING CONFERENCE, 16, **Anais [...],** Asia pacific, p. 29-34, 2009. I

ELMASRI, R; SHAMKANT, B; NAVATHE. **Fundamentals of Database Systems**. [S. l]: Benjamin Cummings, 1994.

EVANS, E. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. [S. l.]: Addison Wesley, 2003.

FERREIRA FILHO, J. I.; DA SILVA, O. A. **Desenvolvimento Orientado a Testes de Aceitação**. Goiânia: Pontifícia Universidade Católica (PUC-GO), 2009. Disponível em: https://www.yumpu.com/pt/document/read/14852812/desenvolvimento-orientado-a-testes-de-aceitacao. Acesso em: 20 set.2019

FOWLER, M. "**Inversion of Control Containers and the Dependency Injection Pattern**." [S. l.]: Fowler, 2004. Disponível em: https://martinfowler.com/articles/injection.html. Acesso em: 02 Jan. 2021.

FOWLER, M**. Patterns of Enterprise Application Architecture**. Canadá: Pearson Education, 2002.

FOWLER, M; BECK, K; BRANT, J; OPDYKE, W; ROBERTS, D. **Refactoring: Improving the Design of Existing Code.** [S. l.]: Addison-Wesley Professional, 1999.

JMOCK, 2007. Disponível em: http://jmock.org/. Acesso em: 02 nov. 2019

JUNIT, 1998, Disponível em: http://junit.org/ . Acesso em: 02 Nov. 2019..

KACZANOWSKI, T. **Practical Unit Testing with JUnit and Mockito**. kaczanowscy.pl Tomas Kaczanowskii, First printing, 2013.

KUMAR, Divya; MISHRA, K. K. **The Impacts of Test Automation on Software's Cost, Quality and Time to Market.** Procedia Computer Science, [S.L.], v. 79, p. 8-15, 2016.

LARAVEL, 2011. Disponível em: https://laravel.com/. Acesso em: 02 nov. 2019.

LARMAN, C. **Applying UML and patterns: an introduction to object oriented analysis and design and iterative development**. Pearson Education, India, 2012.

LEVESON, N. G; THOMAS, J. P. **STPA handbook.** Cambridge: MA, 2018.

LEVESON, N. G. **Engineering a Safer World: Systems Thinking Applied to Safety**. Cambridge: MIT Press Ltd, 2011.

MALHOTRA, R.; CHUG, A. **An Empirical Study to Assess the Effects of Refactoring on Software Maintainability**. In: INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTING, COMMUNICATIONS AND INFORMATICS (ICACCI). **Anais [...],** IEEE, 2016. p. 110-117.

MARIADB, 2009. Disponível em: https://mariadb.org. Acessado em: 02 nov. 2019.

MESZAROS, G. **xUnit Test Patterns, Refactoring Test Code**, Amsterdan: Addison Wesley, 2007.

MOCKERY, 2015. Disponível em: http://docs.mockery.io/en/latest/. Acesso em: 02 nov. 2019.

MOCKITO, 2008. Disponível em: http://mockito.org. Acessado em: 02 nov. 2019

MYERS, G. J; SANDLER, C.; BADGETT, **The Art of Software Testing**, 2. Ed. Hoboken, New Jersey, John Wiley & Sons, 2004.

MYSQL, 1995. Disponível em: https://www.mysql.com/. Acessado em: 02 nov. 2019.

PHOCKITO, 2013. Disponível em: https://github.com/hafriedlander/phockito. Acesso em: 02 nov. 2019.

PHPUNIT, 2001. Disponível em: https://phpunit.de/. Acesso em: 02 nov. 2019.

POSTMAN, 2014. Disponível em: https://www.postman.com/. Acessado em: 02 nov. 2019.

RATZINGER, Jacek; FISCHER, Michael; GALL, Harald C. **Improving evolvability through refactoring**. In : ASSOCIATION FOR COMPUTING MACHINERY. **MSR '05** : Proceedings of the 2005 international workshop on Mining software repositories. New York, NY : ACM Press, 2005. p. 1-5. DOI: https://doi.org/10.1145/1083142.1083155.

REACTJS, 2013. Disponível em: https://reactjs.org/. Acesso em: 02 fev. 2021.

SELENIUM WEBDRIVER, 2004. Disponível em: https://www.seleniumhq.org/projects/webdriver/ Acesso em: 02 nov. 2019.

STAUFFER, M. **LARAVEL: Up & Running**, 2. Ed. O'Reilly Media, 2019.

SONARQUBE, 2006. Disponível em: https://www.sonarqube.org/ Acesso em: 18 set. 2021.

SOUZA, F. G., Pereira, D. P., Pagliares, R. M., Nadjm-Tehrani, S., & Hirata, C. M. **WebSTAMP: a web application for stpa & stpa-sec**. Matec Web Of Conferences, Roubaix, FR, v. 273, p. 02010, 2019. ISSN 2261-236X. DOI: http://dx.doi.org/10.1051/matecconf/201927302010. Disponível em: https://www.matec-conferences.org/articles/matecconf/pdf/2019/22/matecconf_icsc_eswc2018_02010.pdf. Acesso em: 16 jul. 2021.

STAR ADMIN, 2017. Disponível em: https://www.bootstrapdash.com/product/star-admin-free/. Acesso em: 02 fev. 2021.

UNIT, 1998. Disponível em: http://junit.org/. Acesso em: 02 nov. 2019.

APÊNDICE A - TIPOS DE TESTES

O Quadro A.1 apresenta as classes de testes construídas no presente trabalho juntamente com uma breve descrição do objetivo da classe.

Tipo Teste	**Classe Teste**	**Descrição**
Unidade	`ProjectServiceTest`	A classe de teste tem como objetivo realizar a testagem dos métodos CRUD disponíveis na camada *Service* da classe `Project`. Além disso, como se trata de um teste de unidade, *Mocks* foram utilizados para isolar a classe em teste (SUT) de dependências, como a classe `ProjectRepository`.
Unidade	`LossServiceTest`	A classe de teste tem como objetivo realizar a testagem dos métodos CRUD disponíveis na camada *Service* da classe `Loss`. Além disso, como se trata de um teste de unidade, *Mocks* foram utilizados para isolar a classe em teste (SUT) de dependências, como a classe `LossRepository`.
Unidade	`HazardServiceTest`	A classe de teste tem como objetivo realizar a testagem dos métodos CRUD disponíveis na camada *Service* da classe `Hazard`. Além disso, como se trata de um teste de unidade, *Mocks* foram utilizados para isolar a classe em teste(SUT) de dependências, como a classe `HazardRepository`.
Unidade	`SystemGoalService`	A classe de teste tem como objetivo realizar a testagem dos métodos CRUD disponíveis na camada *Service* da classe `SystemGoal`. Além disso, como se trata de um teste de unidade, *Mocks* foram utilizados para isolar a classe em teste (SUT) de dependências, como a classe `SystemGoalRepository`.
Unidade	`ComponentServiceTest`	A classe de teste tem como objetivo realizar a testagem dos métodos *create* e *delete* disponíveis na camada *Service* da classe Component. Além disso, como se trata de um teste de unidade, *Mocks* foram utilizados para isolar a classe em teste (SUT) de dependências, como a classe `ComponentRepository`.

Unidade	`ControllerServiceTest`	A classe de teste tem como objetivo realizar a testagem dos métodos CRUD disponíveis na camada *Service* da classe `Controller`. Além disso, como se trata de um teste de unidade, *Mocks* foram utilizados para isolar a classe em teste (SUT) de dependências.
Integração	`IProjectServiceTest`	A classe de teste tem como objetivo realizar a testagem dos métodos CRUD disponíveis na camada *Service* da classe `Project`. Como se trata de um teste de integração, as verificações dos dados foram realizadas diretamente no banco de dados para comprovar a efetividade dos métodos.
Integração	`ILossServiceTest`	A classe de teste tem como objetivo realizar a testagem dos métodos CRUD disponíveis na camada *Service* da classe `Loss`. Como se trata de um teste de integração, as verificações dos dados foram realizadas diretamente no banco de dados para comprovar a efetividade dos métodos.
Integração	`IHazardServiceTest`	A classe de teste tem como objetivo realizar a testagem dos métodos CRUD disponíveis na camada *Service* da classe `Hazard`. Como se trata de um teste de integração, as verificações dos dados foram realizadas diretamente no banco de dados para comprovar a efetividade dos métodos.
Integração	`ISystemGoalService`	A classe de teste tem como objetivo realizar a testagem dos métodos CRUD disponíveis na camada *Service* da classe `SystemGoal`. Como se trata de um teste de integração, as verificações dos dados foram realizadas diretamente no banco de dados para comprovar a efetividade dos métodos
Integração	`IComponentServiceTest`	A classe de teste tem como objetivo realizar a testagem dos métodos *create* e *delete* disponíveis na camada *Service* da classe `Component`. Como se trata de um teste de integração, as verificações dos dados foram realizadas diretamente para comprovar a efetividade dos métodos.
Integração	`IControllerServiceTest`	A classe de teste tem como objetivo realizar a testagem dos métodos CRUD disponíveis na camada *Service* da classe `Controller`. Como se trata de um teste de integração, as verificações dos dados foram realizadas diretamente no banco de dados para comprovar a efetividade dos métodos.
end-to-end	`EndToEndStepOneTest`	A classe de teste tem como objetivo realizar o teste *end-to-end* de parte da aplicação. Na classe em questão temos três métodos, no qual dois estão associados às permissões de um usuário para poder remover ou não um projeto, e o terceiro consiste no

		preenchimento de todos os dados para o Step One da ferramenta, sendo que a ferramenta é dividida em quatro *steps*.

Quadro A.1 - Testes e seus tipos que foram realizados no presente trabalho. (Testes disponíveis em: https://bit.ly/2TAr5j2).
Fonte: autor da pesquisa, 2021.