Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Análise, detecção e correção de code smells na refatoração de aplicações web

Christian Luis Marcondes Costa, Gabriel Tomaz Lima

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Marcelo Antonio Marotta

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Análise, detecção e correção de code smells na refatoração de aplicações web

Christian Luis Marcondes Costa, Gabriel Tomaz Lima

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Marcelo Antonio Marotta (Orientador)
CIC/UnB

Prof. Dr. Rafael Rabelo Nunes
FACE/UnB

Prof. Dr. Marcos Fagundes Caetano
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 9 de agosto de 2021

Dedicatória

0.1 Dedicatória de Christian

Dedico a minha dupla de TCC, Gabriel, que me motivou nos tempos difíceis e esteve nas comemorações dos bons momentos. Ao grupo de amigos que esteve ao meu lado durante o curso, Alison, Bruno e mais uma vez Gabriel, pois se não fosse vocês, talvez eu não tivesse sido aprovado nas disciplinas complicadas de final de curso e o estudo com certeza não renderia tanto.

À minha irmã, Bianca que é uma grande amiga e compartilha dos momentos bons e ruins vividos em casa. Foi muito divertido ouvir o seu dia a dia na escola e também compartilhar os meus na faculdade.

Ao meu primeiro chefe, Luis Cláudio, que me ensinou como trabalhar e ser mais persistente ao tentar resolver um problema. Os conhecimentos desse estágio levo para a vida.

Ao meu amigo e ex-companheiro de código, Ricardo Augusto, que além de me apresentar o caminho das pedras da carreira de desenvolvimento e ser como um professor da área profissional, também me ajudou bastante com seus conselhos, conversas e dicas sobre todo tipo de assunto. Eu me tornei um profissional melhor graças a você.

Dedico também aos meus pais, Jacinto e Deodete, que sempre me deram educação, suporte e carinho para eu poder conquistar tudo que eu quisesse. Vocês tornaram minha caminhada até aqui bem mais tranquila. Se não fosse por vocês, esse trabalho não seria possível.

Eu poderia dedicar esse trabalho a muitas outras pessoas que me ajudaram durante a graduação, tanto na vida pessoal quanto na acadêmica e profissional, mas no final toda essa jornada de dedicação foi para uma pessoa em especial.

Essa é por mim.

0.2 Dedicatória de Gabriel

Dedico primeiramente a Deus por me conceder o privilégio de estudar nesta Universidade, tenho certeza que os caminhos que trilhei nestes anos de graduação foram guiados por Ti e apesar das dificuldades nunca me abandonaste. Aos meus pais, Geralda e Raimundo por serem a fortaleza em minha vida, grande parte do meu esforço foi para honrá-los. Em memória do meu Pai que não poderá estar presente neste momento feliz de conclusão de curso, mas que ficou extremamente orgulhoso ao me ver ingressando nesta Universidade, seus ensinamentos estão gravados em meu coração. A minha mãe por toda dedicação, apoio, amor e cuidado, todo esforço que eu realizar para te retribuir nesta vida não será o suficiente por tudo que fez e ainda faz por mim.

À minha noiva e futura esposa Stéfany pelo carinho, atenção e apoio durante todos esses anos, por me suportar nos momentos mais difíceis. Você é um anjo que Deus colocou em minha vida, sem você esta jornada seria ainda mais árdua, não tenho palavras para agradecer o tanto que fizeste por mim.

Como mencione acima, Deus me guiou durante minha graduação e tenho certeza que Ele colocou vocês em meu caminho. Renato Rossi, amigo e gestor, agradeço pelos conselhos que me destes nestes anos de trabalho, pela compreensão e apoio, por todas as concessões que fizestes para que eu pudesse me dedicar aos estudos. Professor e amigo Rafael Rabelo, por todo o suporte durante o curso, pela confiança em meu trabalho. Serei eternamente grato aos senhores.

À minha irmã, Anna Cláudia por acreditar em mim e torcer pelo meu progresso. Ao Pedro, meu outro Pai que Deus me deu, pelo apoio e dedicação para comigo. A todos aos meus amigos e familiares que não conseguirei mencionar nesta breve mensagem, mas que torcerem por mim e acreditaram no meu potencial.

E por fim, ao meu amigo Christian que esteve comigo durante grandes desafios enfrentados ao decorrer da graduação, sua ajuda foi fundamental para vence-los. A confiança que depositou em mim e neste projeto, por toda ajuda e apoio que me deste, você é um dos grandes responsáveis por esta conquista.

Agradecimentos

Agradecemos primeiramente ao nosso professor e orientador Marcelo Marotta, que mesmo com muitos afazeres e compromissos sempre esteve disponível para sanar as nossas dúvidas com relação ao TCC e da mesma forma ao ministrar a disciplina de Computação Experimental, você tornou a difícil experiência de estudar na UnB mais fácil e agradável, por mais professores como você.

Aos vários autores da computação que publicaram seus conhecimentos online, possibilitando o aprendizado e a evolução do conhecimento.

À Universidade de Brasília e toda sua equipe técnica, servidores e docentes, que apesar de sofrer com vários problemas causados por fatores internos e externos, possibilitaram o nosso aprendizado e a abertura várias oportunidades.

Resumo

Softwares de apoio a decisão são de grande importância em diversos ramos da sociedade. Entretanto esses softwares são usualmente privados e possuem alto custo no mercado, surgindo assim a necessidade de se desenvolver um software gratuito e de código aberto. O software MyMCDA-C em forma de uma aplicação *web* foi desenvolvido com intuito de suprir estas necessidades e, apesar de conter várias falhas de implementação em sua primeira versão, o software tornou-se de grande valia para comunidade acadêmica. Essas falhas motivam o uso do processo de refatoração, o qual visa mitigar as falhas de implementação, modificando e aprimorando o software de modo a manter o comportamento do mesmo e minimizar as oportunidades de introduzir novos erros. Um ponto importante deste processo é a detecção e mitigação de *code smells* que são problemas comuns em softwares que podem dificultar a manutenção. Na análise da literatura são verificados os impactos de implementações que contém *code smells* e como diferentes autores chegam a diferentes conclusões sobre como mitiga-los. Neste estudo, buscou-se refatorar a primeira versão do MyMCDA-C, detectando, analisando e corrigindo os *code smells* encontrados e, assim, dando mais qualidade ao sistema tornando-o mais confiável e modular, além de adicionar novas funcionalidades ao sistema de modo a melhorar a experiência do usuário.

Palavras-chave: Code Smell, MCDA-C, Aplicação Web, Arquitetura de Software

Abstract

Decision Aid support software are important to many branches of society, however the price of these is very difficult to afford, rising the need to develop a free and open source software. In order to meet this need, the students developed the MyMCDA-C web application and despite having many implementation flaws in its first version, the software has become of great value for the academic community. The refactoring process aims to mitigate implementation flaws, modifying and improving software quality while maintaining its behavior and minimizing the chances of introducing new bugs. Another part of this process is the detection of code smells, which are common problems found in software that can make it difficult to maintain. The literature review verified the impacts of code smells and that different authors draw different conclusions, which is due to the fact the programmer's experience is a relevant aspect in the detection and mitigation of code smells. An efficient way to extend the lifetime of a software is to make the code easier to maintain. In this study, we refactored the first version of MyMCDA-C through detecting, analyzing and correcting the flaws found and thus, giving more quality to the system, making it more reliable, free of code smells and with decoupled modules. The software also got new features added to it in order to improve user experience.

Keywords: Code Smell, MCDA-C, Web Aplication, Software Architecture

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

HTML HyperText Markup Language.

JWT Json Web Token.

M-MACBETH Measuring Attractiveness by a Categorical Based Evaluation Technique.

MVC Model View Controller.

MyMCDA-C Modelo de análise Multicritério de Apoio à Decisão construtivista..

SOLID Single-responsibility principle, Open–closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle.

UML Unified Modeling Language.

UnB Universidade de Brasília.

Capítulo 1

Introdução

Sistemas de apoio a decisão são fundamentais para a execução e administração de projetos e execução de atividades em quaisquer ramos da sociedade, como ramos administrativos, elétricos e hídricos. Nesse contexto, ganham destaque os softwares privados disponíveis no mercado e que aplicam a metodologia de apoio à decisão como base, por exemplo, o *Measuring Attractiveness by a Categorical Based Evaluation Technique* (M-MACBETH). Porém, estes software são licenciados e possuem um valor de venda e licenciamento extremante caros, tornando-os impedidos de serem utilizados por comunidades pequenas e para uso pessoal. Dessa forma, surgiu-se a necessidade de criar um software, o MyMCDA-C, de código aberto, para habilitar o uso de sistemas de apoio a decisão sem a necessidade de compra ou venda de software.

O software MyMCDA-C foi desenvolvido em sua primeira versão no ano de 2018 com intuito de suprir as necessidades do Departamento de Administração da Universidade de Brasília (UnB). A primeira versão do sistema foi desenvolvida através de um projeto de iniciação cientifica, em forma de uma aplicação *web*. Na época, em face da necessidade imediata de um software de apoio a decisão e por conta da pouca experiência dos desenvolvedores envolvidos, ainda que em seu melhor esforço, os alunos que desenvolveram o software MyMCDA-C, não se atentaram as boas práticas de programação e a padronização do código, gerando o que é chamado na literatura de *code smells*.

Após o lançamento de sua primeira versão, o MyMCDA-C tornou-se de grande utilidade para comunidade acadêmica, em especial para Universidade de Brasília (UnB). O software serviu como base para diversas pesquisas, artigos e trabalhos de conclusão de curso. Como o estudo realizado por Moreira et al [1] sobre da análise de risco na segurança cibernética, que aplicou a metologia multicritério com auxílio do MyMCDA-C. Além disso, outros estudos foram realizados em diversas áreas, sendo elas gestão de processos com o trabalho de Barbalho et al [2], *Marketing Digital* por Lourenço et al [3], *E-commerce* e Vendas por Wielewski et al [4] e Lopes Wielewski et al [5].

Considerando os problemas de desenvolvimento, a refatoração é o processo de modificar e melhorar a parte interna de um software sem alterar o comportamento do mesmo, é uma maneira disciplinada de aprimorar o código minimizando a probabilidade de introduzir erros [6]. O termo *Code smells* é por definição o indicativo de problemas em pontos que impactam nas características de um software, podendo assim serem descritos como confiáveis, compreensíveis, legíveis ou com tendências a erros [7]. Walter e Alkhaeir [7], em seu artigo intitulado *A relação entre code smells e padrões de projeto: Um estudo exploratório*, definem *code smells* como uma metáfora em alto nível para sinais de um design inadequado que podem afetar a manutenção futura de um código. Ainda segundo o artigo, *code smells* são sintomas que podem apontar para um problema, apesar de esse não ser sempre o caso.

A partir deste contexto, o objetivo do presente trabalho foi analisar, detectar e corrigir *code smells* da primeira versão do software MyMCDA-C. Além de desenvolver uma versão do software com novas funcionalidades. Este estudo se deu com base em uma robusta análise de código, a fim de encontrar possíveis falhas de implementação da versão inicial. Esta análise foi realizada com base na literatura, como por exemplo o livro de Fowler [6] que descreve os principais *code smell* e a forma de mitiga-los, além do estudo de Aniche *et al.* [8] que descreve os principais *code smell* encontrados em arquiteturas MVC. Após o levantamento das principais falhas do software, foi desenvolvida a segunda versão com correção das falhas encontrados em conjunto com a nova arquitetura e organização do software. Em seguida, as melhorias propostas na seção 5.3 foram desenvolvidas resultando na terceira versão do sistema.

Na literatura, diversos estudos analisam o impacto de implementações de baixa qualidade em inúmeros aspectos relacionados a manutenção, porém os resultados dos trabalhos divergem e levam a diferentes conclusões [7]. Por exemplo, Walter e Alkhaeir [7] expõe alguns estudos que apresentam resultados em que *code smells* estão relacionados com uma maior densidade de defeitos enquanto outros artigos colocam a prova a crença de que *code smells* estão fortemente relacionados com problemas de manutibilidade. Outros fatores também relevantes para detecção de *code smells* são a experiência do programador, o tamanho do código e o domínio sobre o software.

O artigo de Khalid et al [9], denominado *Software Design Smell Detection: a Systematic Mapping Study*, relata que problemas na implementação de código costumam aparecer tardiamente, e as opções para solução desses problemas são bastante complexas. Isso aumenta o débito técnico, sendo a alternativa mais viável refazer o software. Outro ponto importante abordado por Khalid é que, caso hajam trechos do software que sofram com programação de baixa qualidade ou com um desenho ruim, estes devem ser identificados e melhorados. Um *code smell* pode afetar a compreensão, manutibilidade, entendimento

e reusabilidade de um código e devem ser considerados como um alerta de um possível débito técnico que podem levar ao fracasso do projeto.

O software MyMCDA-C é baseado no Modelo Multicritério de Apoio à Decisão - Construtivista (MCDA-C). A metodologia consiste basicamente em três etapas distintas: sistematização do modelo; avaliação dos grupos; e conclusões finais [10]. Na metodologia, existem aspectos subjetivos e aspectos objetivos ao empregar a metodologia dentro de um processo decisório, em cada uma das etapas. As características subjetivas relacionam o juízo de valor que cada um dos atores faz aos critérios que devem ser considerados dentro de um processo decisório qualquer. Já as características objetivas estão relacionadas com o processo de coleta de dados realizada através do contato com os usuários, as ações na tomada de decisão, e as considerações finais baseadas nos resultados gerados pela utilização da metodologia. O MyMCDA-C recebe os dados coletados pelo usuário e através de fórmulas e cálculos transforma dados qualitativos em quantitativos, auxiliando os gestores na tomada de decisão. A figura 1.1 representa as etapas do modelo decisório até a obtenção dos resultados.

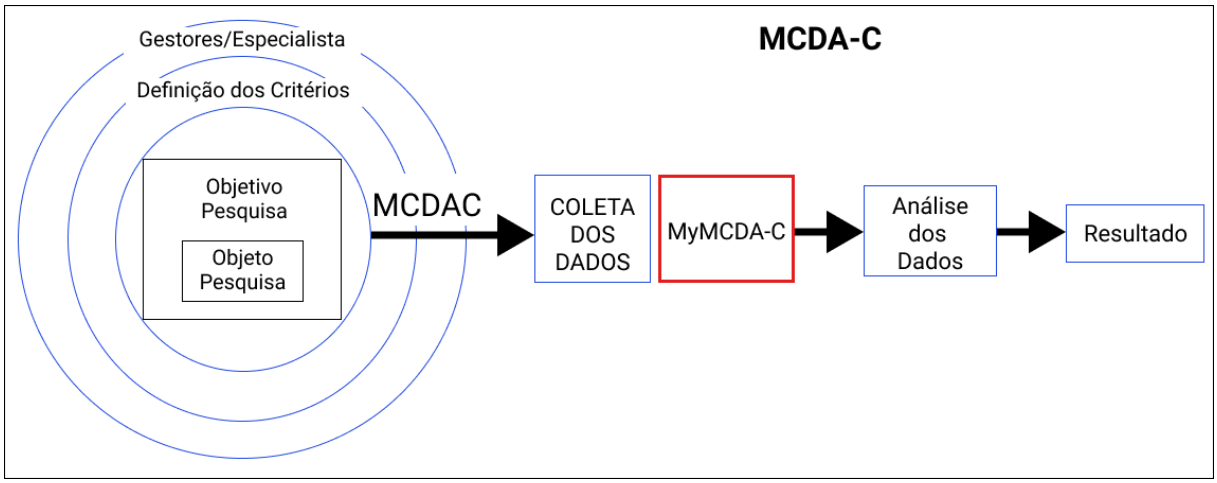

Figura 1.1: Modelo Multicritério.

Na concepção inicial do software MyMCDA-C, não houve uso de padrões de projeto, com exceção do MVC (Model View Controller). Técnicas para evitar o surgimento de *code smells* não foram consideradas durante o desenvolvimento do sistema. Neste trabalho, pretende-se mitigar o problema de manutenções futuras do software MyMCDA-C, tornando-o mais simples de compreender. O software MyMCDA-C foi refatorado para tornar o sistema confiável, livre de *code smells* e com módulos desacoplados. Isso possibilitou a inserção de um novo módulo de forma simples e escalável.

Durante a criação desse novo módulo do *software*, foi considerada também a usabilidade do sistema. Os usuários do MyMCDA-C tinham muitos problemas com a usabili-

dade, principalmente com a navegação no sistema. Para simplificar e facilitar a navegação do MyMCDA-C, foi realizado um redesenho do sistema, principalmente na parte de Projetos. Nos resultados do presente trabalho, foi realizado um estudo comparativo da navegação no MyMCDA-C original e no MyMCDA-C com o novo módulo.

O presente trabalho é organizado da seguinte maneira: no Capítulo 3 são apresentados artigos e livros que motivam e baseiam o presente trabalho correlacionados entre si. No Capítulo 4 é realizado o levantamento dos *code smells* e feita a proposta de solução ao nível arquitetural. A correção dos *code smells* ao nível de código e as novas funcionalidades são descritas no Capítulo 5. No Capítulo 6 os impactos das correções são exibidas e por fim, no Capítulo 7 são realizadas as considerações finais sobre o trabalho.

Capítulo 2

Fundamentação Teórica

O *code smell* é uma indicação presente no código que normalmente corresponde a um problema mais profundo no *software*. Essa definição tem suas nuances, já que na tradução literal do termo para 'cheiro de código' temos a definição da palavra 'cheiro' como algo fácil de identificar. Um dos exemplos mais claros disso é o Método Longo, que pode ser facilmente farejado só de se observar o código. Outra nuance do termo *code smell* é a de que nem todo cheiro indica um problema e com isso, nem todo *code smell* precisa de correção [11]. No presente capítulo serão detalhados os principais *code smells* que fizeram parte desse trabalho.

Um dos *code smells* mais comuns, o Código Duplicado, pode ser facilmente identificado quando se observa a mesma estrutura de código sendo utilizada em mais do que um local. Quando o Código Duplicado é encontrado na mesma classe, basta utilizar a técnica de Extração de Método e chamar esse trecho de código encapsulado nos locais aonde existia o Código Duplicado. Quando o Código Duplicado está presente em duas classes sem relação direta, a recomendação é extrair o trecho duplicado com a técnica de Extrair Classe e então realizar essa chamada nos locais onde havia a duplicação de código [6].

Os programas que tem o maior tempo de vida são aqueles com métodos curtos. Desde o início da programação, as pessoas envolvidas perceberam que a dificuldade de entendimento de um método aumenta conforme o seu tamanho. Em linguagens mais antigas, as chamadas de sub rotinas precisavam de atenção especial na troca de contexto, o que fazia que desenvolvedores evitassem a prática da escrita de métodos curtos. Linguagens de programação modernas orientadas à objetos praticamente eliminaram essa preocupação de troca de contexto, facilitando a decomposição de métodos longos. Em 99% dos casos, utilizar a técnica de Extração de Método pode resolver esse *code smell* de Método Longo. A técnica de Extrair Método consiste em encapsular fragmentos de código em um novo método [6].

O *code smell* Classe Extensa ocorre quando uma classe está sobrecarregada com múltiplas responsabilidades. A Classe Extensa pode ser identificada por muitas variáveis de instância ou simplesmente por possuir bastante código. Para corrigir a Classe Extensa, pode se utilizar a técnica de Extrair Classe ou Extrair Subclasse. Extrair Classe consiste em extrair alguns métodos da classe original e criar uma classe independente da primeira, enquanto o método de Extrair Subclasse cria uma classe que herda da original. Classe Extensa também pode ser causada pelo *code smell* de Código Duplicado [6].

Quando um trecho de código é desenvolvido, às vezes são considerados vários casos especiais que não são necessários, mas são implementados para caso sejam necessários no futuro. A Generalidade Especulativa ocorre quando um método trata vários casos simultaneamente, podendo ser nomeado de forma ambígua e abstrata. Esse *code smell* pode ser resolvido utilizando Renomear Método, em alguns casos unido à técnica Extrair Método extração[6].

Feature Envy é um *code smell* que descreve um método com mais interesse em outra classe do que na classe em que está contido. Traduzindo o termo, *Feature Envy* significa "inveja de funcionalidade", e ocorre quando um método realiza a chamada de métodos de outro objeto para, por exemplo, calcular um valor. Para resolver o *Feature Envy* é simples, basta mover o método para a classe que ele está mais interessado. Em alguns casos, apenas parte do método sofre de 'inveja' e, nesse caso o recomendado é Extrair Método e mover esse método para a classe de interesse extração[6].

Capítulo 3

Trabalhos Relacionados

O capítulo apresenta artigos e livros relacionados ao presente trabalho, incluindo questionários de pesquisa e artigos de identificação e classificação de *code smells* e anti padrões. Também são expostos trabalhos que auxiliam o entendimento da correlação de padrões de projeto e *code smells*.

3.1 Refatoração: Aperfeiçoando o Design de Códigos Existentes

O termo *code smells* foi utilizado pela primeira vez por Kent Beck enquanto ajudava Martin Fowler na escrita de seu livro *Refatoração: Aperfeiçoando o Design de Códigos Existentes* [11]. Os *code smells* são abordados na literatura por diversos autores, que propõem formas de classificar e identificar este conceito. Fowler foi o primeiro autor a propor e definir em seu livro o termo *code smells*.

O termo *code smells* é definido como um sintoma de um problema mais profundo presente no sistema [11]. Um *code smell* nem sempre é um problema, mas sim um possível indicador de uma implementação que exige refatoração [11]. Em seu livro [6], Fowler descreve mais de 20 categorias de *code smells* e a forma de refatorá-los. A maioria das categorias de *code smells* levantadas por Fowler e outros, serão investigadas e mitigadas durante a refatoração da aplicação MyMCDA-C.

Apesar de o livro tratar de *code smells*, o ponto principal do livro é a refatoração de software. Fowler define refatoração e também ajuda a entender o porque é uma boa ideia refatorar um software [6]. O livro também mostra diversos métodos de refatoração e de como melhorar a legibilidade e o entendimento do código [6]. Por fim, o livro cita estratégias gerais de como conduzir a refatoração do software, como ter um objetivo e quando parar.

O livro *Refatoração: Aperfeiçoando o Design de Códigos Existentes* escrito por Fowler é de extrema importância para este trabalho. Após a leitura, foi possível entender, definir e identificar o que é um *code smell*. Além disso, também foi possível aprender bastante e entender como fazer uma refatoração para cada *code smell* encontrado.

3.2 Um Estudo Empírico de *code smells* em Projetos JavaScript

Segundo Saboury *et al.* [12], JavaScript é uma das mais poderosas linguagens de programação já desenvolvidas e ganhou notoriedade na última década. Assim como aplicações construídas em outras linguagens de programação, aplicações javascript não estão livres de *code smells*.

No trabalho de Saboury *et al.* [12], 537 versões dos 5 frameworks mais populares em Javascript foram analisadas em busca de *code smells* sendo elas o *express*, *grunt*, *less.js* e *request*. Além de realizar analises de sobrevivência, comparando o tempo até a ocorrência de uma falha, em arquivos com e sem *code smells*. Os autores buscaram responder duas perguntas chaves em seu trabalho, *Os arquivos JavaScript com code smells são igualmente propensos a falhas?* e *Os arquivos JavaScript com code smells são igualmente propensos a falhas?*.

Saboury *et al.* detectaram 12 categorias de *code smells* e em resposta às duas perguntas realizadas no início do artigo chegaram estas conclusões. Classes contendo *code smells* são 65% mais chances de causar problemas no funcionamento do sistema do que classes que não contém *code smells*. Arquivos contendo diferentes categorias de *code smells* não tem a mesma probabilidade de conter erros e os desenvolvedores que detectarem dois tipos específicos de *code smells* como: *reatribuição de variáveis* e *atribuições em instruções condicionais* devem refatorar o código de modo a remover estas imperfeições, visto que aumentam o risco de falhas no sistema.

O artigo de Saboury *et al.* [12] é pertinente para o presente trabalho, pois apresenta alternativas para contornar os problemas causados por *code smells*. A partir de uma análise mais profunda do código pelos desenvolvedores, torna-se possível detectar e mitigar as principais categorias de *code smells* para a linguagem JavaScript. Além de apresentar uma lista de *code smells* com a descrição de cada um deles, auxiliando na detecção e correção dos mesmos. Tal lista será utilizada como parte da análise do trabalho proposto nessa monografia.

3.3 Avaliando o impacto de padrões de projeto e dependências anti padrão em mudanças e falhas

De acordo com Jaafar *et al.* [13], padrões de projeto são soluções para problemas recorrentes, com o objetivo de aumentar a usabilidade, flexibilidade e facilidade de manutenção de um software. Entretanto, trabalhos anteriores verificaram que alguns padrões de projeto estão diretamente relacionados aos *code smells* [13]. Por outro lado, foi verificado que seguir um anti padrão causa problemas que realça as fraquezas de um software, dificultando a manutenção do software [13].

Apesar de ser necessária uma maior atenção com relação aos *code smells* quando utilizando um padrão de projeto, foi verificado que seguir um anti padrão, soluções inadequadas para um problema, normalmente tem maior probabilidade de causar problemas [13]. Co-dependência entre classes, sejam implementadas utilizando padrões de projeto ou anti padrões, são prováveis de causar defeitos [13]. Apesar de co-dependência entre classes ser problemática, quando se usa um padrão, a probabilidade da classe causar defeitos é menor [13].

Esse artigo foi de grande contribuição para o atual trabalho, pois mostra que os padrões de projeto podem conter *code smells*, mas que mesmo assim é melhor aplica-los do que implementar um anti padrão. Sempre que possível, padrões de projetos que possuem *code smells* inerentes as suas definições devem ser evitados. O artigo [13] também reforça o que foi dito por Fowler, de que *code smells* não necessariamente indicam um problema, mas sim um sintoma de um possível problema [11].

3.4 Análise de *code smells* em Arquiteturas Model-View-Controller

No trabalho desenvolvido por Aniche *et al.* [8], procura-se analisar os *code smells* gerados pela arquitetura Model View Controller (MVC). Geralmente, os trabalhos que visam analisar *code smells* estão mais focados no aspecto amplo do termo, os quais afetam softwares. No artigo, os autores visaram uma arquitetura específica a MVC, pois está arquitetura pode ser afetada por algumas práticas inadequadas específicas deste contexto.

O artigo analisa *code smells* em sistemas que adotam a arquitetura MVC, 53 desenvolvedores foram entrevistados ou responderam questionários de modo a validar um conjunto de *code smells* em diferentes perspectivas. Dentre os feitos obtidos no estudo, tem-se o catálogo com 6 *code smells* específicos da arquitetura MVC e que *code smells* nessa arqui-

tetura estão sujeitos a mudanças e possuem maior probabilidade de conter algum defeito. Os autores validaram a pesquisa de diferentes perspectivas em quatro etapas.

Primeiro, avalia-se a relação entre *code smells*, alterações no código e tendências a defeitos. Em segundo, investiga-se quando um code smell é inserido no código e por quanto tempo ele persiste até ser corrigido. Terceiro, 21 desenvolvedores foram entrevistados de modo a verificar sua percepção da definição de *code smells*. Quarta e última etapa, com o catálogo de *code smells* definidos, quatro desenvolvedores especialistas que trabalhavam com diferentes tecnologias foram entrevistados para implementação de um aplicativo MVC de modo a checar a generalização do catálogo definido. Um dos resultados obtidos pelo estudo demonstra que dentre os 6 categorias de *code smells* identificados, a maioria das classes são afetadas pelo problema de *Fat Repository*(20.5%), ou seja, um repositório que gerência muitas entidades. Seguido por *Promiscuous Controller*(12.2%) uma classe que oferece muitas ações e *Brain Controllller*(7.4%) classe com muito controle de fluxo.

O sistema desenvolvido no presente trabalho foi projetado na arquitetura MVC em sua primeira versão, segundo Aniche et al. [8] essa arquitetura não é livre de *code smells*. Os resultados obtidos pelos autores do artigo servem de base para serem usados para MVCs em qualquer contexto, além de servir como base para a busca de *code smells* no software analisado no presente estudo por isso foi de grande importância para o estudo.

3.5 A relação entre *code smells* e padrões de projeto: Um estudo exploratório

Para aplicar padrões de projeto em nosso código precisamos primeiro entender a relação entre padrão de projeto e *code smells*. Padrões de projeto representam soluções genéricas recomendadas para diversos problemas de design [7]. Já *code smells* são sintomas que podem dificultar a manutenção de um sistema de software [7]. Intuitivamente, é possível esperar que os dois conceitos são mutuamente exclusivos [7].

O entendimento da correlação entre *code smells* e padrão de projeto é de extrema importância para o presente trabalho, pois é de notório saber que *code smells* podem causar problemas futuros em software [12]. No estudo de Walter et al. [7], foi concluído que utilizam padrões de projeto exibem menos *code smells* do que outras classes. O estudo também respondeu se o número de classes com *code smells*, mas sem padrão, e o número de classes com *code smells*, mas com padrão de projeto, muda durante a evolução de um sistema, porém os resultados foram inconclusivos.

Com leitura do artigo de Walter et al. [7], foi possível descobrir quais *code smells* estão relacionados a quais padrões de projeto, o que foi relevante na escolha dos padrões a se utilizar para desenvolvimento do presente trabalho. Utilizando o trabalho de Walter et al.

[7] como base, foi verificado que a presença de um padrão de projeto está correlacionado com ausência de *code smells* e que State-Strategy, Adapter-Command, a Factory são padrões com baixa correlação com *code smells*. Uma observação a se fazer é que foi verificado que o Singleton é o padrão de projeto com menor correlação com *code smells*, o que contradiz outros estudos.

3.6 Compreendendo mal-entendidos no código-fonte

Para identificar o que pode levar a ocorrência de problemas em um software, é preciso primeiro entender que a compreensão incorreta de um trecho de código por parte dos desenvolvedores podem levar a erros futuros. De acordo com Gopstein *et al.* [14], átomos de confusão são padrões que levam ao desenvolvedor a interpretar o código erroneamente. Entretanto, há uma certa dificuldade em identificar esses átomos de confusão, pois há muita subjetividade no que pode ou não ser um átomo.

Os autores selecionaram um conjunto de átomos de confusão conhecidos e podem confundir programadores. Um experimento foi realizado com 73 participantes com intuito de apresentar esses trechos de código e mostrar que eles podem aumentar significativamente a taxa de código mal interpretados, quando comparados com códigos que não apresentavam o mesmo padrão. Em seguida, foram selecionados os programas mais confusos, ou seja, os que tem mais probabilidade de confundir um programador, e um estudo foi realizado com 43 participantes de modo a mensurar em termos de confusão do programador, como esses átomos de confusão eram removidos.

Os autores chegaram a conclusão de que, considerando os códigos menos claros em comparação aos demais em termos de entendimento, os participantes têm estatisticamente maior probabilidade de cometer erros em código pouco compreensíveis. Os resultados são refletidos nos programas analisados. Em cada par de perguntas, sobre o entendimento do código, os programas com menos átomos de confusão foram identificados com maior precisão do que na versão mais ofuscada.

Para guiar a escolha de *code smells* foi considerado o trabalho de Gopstein *et al.* [14] que traz uma lista abrangente de átomos de confusão e suas possíveis transformações para evitar esses problemas e, apesar do artigo se tratar de átomos de confusão na linguagem C foi possível aproveitar os conceitos abordados no estudo, de modo a identificar e corrigir "átomos de confusão"no software desenvolvido.

3.7 Detecção de *code smells* em software: um estudo de mapeamento sistemático

De acordo com Khalid *et al.* [9], *code smells* são definidos como indicadores de situações que afetam negativamente a qualidade do software com facilidade de entendimento, testabilidade, extensibilidade, reusabilidade e facilidade de manutenção em geral [9]. Facilitar a manutenção é um dos pontos principais para tornar a evolução de software mais simples [9]. Detectar *code smells* é importante para ajudar desenvolvedores a tomar a decisão quando melhorando o processo de evolução do software [9]. O estudo realizado por Khalid et al [9] durante 18 anos analisou 395 artigos que tratam de modelos de detecção de códigos defeituosos.

Os autores analisaram as técnicas empregadas na detecção de *code smells*, as ferramentas que foram utilizadas, os tipos identificados e os recursos aplicados na avaliação de código defeituoso [9]. Foi confirmado que diferentes conceitos são usados nos artigos selecionados para descrever *code smells*, variedade essa que faz *code smells* serem categorizados divergentemente [9]. Também foram encontrados na literatura diferentes modelos de qualidade na literatura, e cada modelo define um conjunto de critérios de qualidade de software [9]. Cada critério inclui um grupo de sub-critérios que afetam a qualidade [9]. Esses sub-critérios eram comuns entre os modelos de qualidade [9]. A maioria dos critérios de qualidade relacionados com *code smells* tinham prioridade máxima de remoção do software [9].

Um modelo para análise e detecção de códigos defeituosos é importante para dar suporte a desenvolvedores em relação ao processo de evolução de um software [9]. Diversas literaturas propõe modelos para detecção e análise de *code smells*, porém ainda não há um modelo padrão para solucionar o problema. Ao avaliar se um código possui ou não *code smells*, é importante considerar diversos métodos para obter uma maior confiança e conseguir identificar e remediar problemas [9]. O software analisado no presente artigo utilizou os conhecimentos da literatura para apoiar a detecção de códigos defeituosos.

3.8 Confusão nas revisões do código: razões, impactos e estratégias de enfrentamento

Em projetos de software, é natural que se trabalhe em equipe no desenvolvimento unindo as habilidades e forças de cada integrante. Nesses projetos, é importante utilizar a revisão de código para garantir a qualidade do software [15]. Entretanto, em alguns casos os códigos escritos podem estar confusos e isso pode ter impacto negativo no software,

fazendo que códigos sejam aprovados cegamente [15]. Para resolver esse problema de confusão no entendimento de código em revisões de código é necessário estabelecer uma boa comunicação com a equipe e pedir por mais informações e ter discussões *offline* sobre o código[15], o que além de ser beneficial para o software possibilita a transferência de conhecimento e o pensamento crítico.

O trabalho realizado por [15] investigou as razões e impactos de equívocos em revisões de código, bem como as estratégias adotadas por desenvolvedores para lidar com esses problemas. Uma estratégia de triangularização foi adotada visando combinar as análises das respostas da pesquisa e dos comentários das revisões de código. Construindo uma estrutura para compreensão de equívocos em códigos abrangendo os processos de revisão de código, o artefato que está sendo revisado, os próprios desenvolvedores e a relação deles com o artefato.

Os resultados obtidos pelo estudo foram encontrados os 30 motivos mais comuns para equívocos em código, dentre eles os mais comuns são a falta de raciocínio nas soluções, discussão de requisitos não funcionais da solução e falta de familiaridade com o código já desenvolvido. Dentre os impactados dos equívocos em códigos, os mais relevantes são os atrasos, diminuição na qualidade da revisão e discussões adicionais. Enfim, para lidar com esses problemas, 13 estratégias foram listadas por desenvolvedores, dentre elas estão solicitação de mais informações, aumentar a familiaridade com o código e discussões *offline*.

Tendo como base a literatura acima, foi possível obter ferramentas para realizar a refatoração do software que é estudo desse trabalho. A literatura foi usada como guia para identificar os equívocos no código e os anti padrões do software. Os trabalhos acima também tiveram relevância ao auxiliar na escolha de padrões de projeto e para buscar alternativas aos códigos que apresentavam *code smells*.

Capítulo 4

Proposta

Neste Capítulo, apresenta-se primeiro, as estratégias utilizadas para detectar e corrigir *code smells* e em seguida as métricas utilizadas para avaliar os resultados do trabalho e a forma utilizada para gera-las. Então, o diagrama de classes da primeira versão do software MyMCDA-C é apresentado, expondo os *code smells* identificados em nível arquitetural. Em seguida, é apresentada a proposta de refatoração da arquitetura do software, onde são descritas as tomadas de decisão de projeto para a mitigação dos *code smells*.

4.1 Análise e Detecção de *Code Smells*

Com o intuito de identificar os *code smells* da primeira versão do software MyMCDA-C, foram realizadas reuniões com objetivo de mapear os possíveis erros de implementação da ferramenta. Nestas reuniões, todas as funções, métodos e classes do sistema passaram por uma análise minuciosa. Para fundamentar esta análise um estudo foi realizado com base na literatura de modo identificar as principais falhas de código presentes em softwares, além dos indicativos de problemas mais comuns em arquiteturas MVC e aplicações *web*.

Um dos estudos que basearam a análise e detecção de *code smells* foi livro de Martin Fowler [6], que descreve mais de 20 categorias de *code smells* e a forma de corrigi-los. Um trecho de código suspeito nem sempre é um problema de verdade, sendo necessária uma análise mais profunda para ver se existe um problema implícito, esses códigos com possíveis defeitos são geralmente um indicador de um problema e não o problema em si. Os melhores *code smells* são geralmente aqueles mais fáceis de detectar, e geralmente levam a problemas realmente interessantes [11]. Segundo Fowler, um bom ponto sobre *code smells* e que até programadores com pouco experiência conseguem detectá-los mesmo que não saibam corrigi-los.

Outro estudo importante que serviu como base para análise e detecção de problemas no código foi o de Aniche *et al.* [8]. Neste trabalho, são listados os 6 principais *code*

smells encontrados na arquitetura MVC, em que a primeira versão do MyMCDA-C foi desenvolvida. Dentre esses *code smells* os que se destacaram para este estudo foram os problemas de *Promiscuous Controller* e *Brain Controller* que são problemas relatados por Fowler como Método Longo e Classe extensa já descritos no capítulo 2. Porém, esses problemas são específicos da arquitetura MVC.

Por fim, após as consultas realizadas a literatura, o código fonte da primeira versão do software foi analisado detalhadamente. Todos os arquivos das camadas de visualização, controle e modelo foram avaliados. Os autores abriram os arquivos e mapearam todos os métodos presentes e então analisaram se naquela determinada função ou classe poderia conter um *code smell*. Essa análise era baseada na experiência com desenvolvimento de software dos autores e também pelo conhecimento adquirido através do estudo da literatura. Ao detectar um possível *code smell* o problema encontrado era descrito, bem como a solução para o problema era relatada. Todos os *code smells* encontrados foram descritos e corrigidos no capítulo 5.

4.2 Descrição das Métricas

Com objetivo principal de remover os *code smells* do MyMCDA-C, foi feito o levantamento de todos os *code smells* do sistema. Tendo as quantidades de *code smells*, foi possível demonstrar o que já foi encontrado na literatura anteriormente. A métrica de quantidade de *code smells* visa medir, em partes, a qualidade do software, onde menos *code smells* implicam em uma maior qualidade. Na versão inicial do MyMCDA-C, a métrica também ajuda a entender quais *code smells* mais afetam o software.

Nesse trabalho, foi utilizada a métrica de quantidade de funções por controladora. Para isso, foi feito o levantamento dos números de função por controladora na versão antiga e na versão atualizada do MyMCDA-C. O objetivo de utilizar essa métrica é para tentar agrupar melhor os métodos e evitar um problema observado na versão inicial do software: o de classes com apenas um método. Nessa métrica de quantidade de funções por controladora, uma concentração maior (mas não excessiva) representa um resultado positivo para o presente trabalho.

O objetivo da melhoria da experiência do usuário é tornar tarefas num software fáceis de ser realizadas. Para melhorar a usabilidade do software, foi utilizada uma heurística que traduzida é chamada "Regra de 1 Clique". Essa regra diz que todo clique ou interação deve aproximar o usuário de seu objetivo, enquanto eliminando o máximo de alternativas que o não levam ao seu destino [16]. De modo a melhorar a usabilidade, foi realizado um redesenho do software para tentar reduzir o número de cliques entre quaisquer duas telas. Por ser um sistema relativamente pequeno, foi efetuada a análise manual do menor número

de cliques entre as telas do MyMCDA-C. O levantamento foi efetuado na versão inicial do sistema e em seguida comparado com o levantamento realizado na versão atualizada com o novo módulo. Nessa métrica, um número menor de interações representa um resultado positivo.

4.3 Diagrama de Classes do Projeto Antigo do Sistema MyMCDA-C

Nessa seção é apresentado o diagrama de classes do sistema 1.0 na figura 4.1. Algumas classes, métodos e atributos foram marcadas com um círculo vermelho, indicando a presença de *code smells*. As ligações apontadas pelas setas indicam que uma classe usa ou tem relação com a outra classe. No caso das *views*, as pastas representam as páginas do sistema e a seta indica que a classe é usada na página indicada.

A linguagem de modelagem unificada (UML) contribui na modelagem de sistemas de inúmeras formas. O diagrama de classes é um dos tipos mais comuns de diagramas *UML*, pois mapeiam claramente as classes, módulos, atributos, relações e operações de um sistema, auxiliando na visualização da estrutura de aplicações. Empregado por engenheiros de software com intuito de documentar arquiteturas de sistemas, o diagrama de classes pode ser descrito como um diagrama de estrutura, pois representa as configurações de um software. Além disso, pode ser padronizado para uso em abordagens de programação orientada ao objeto, como a utilizada nesta pesquisa.

A figura 4.1 representa a estrutura do software MyMCDA-C dividido em 3 camadas que configuram arquitetura MVC. As controladoras são responsáveis por coordenar o fluxo entre as camadas de modelos (*models*) e visualizações (*views*). Normalmente, as classes controladoras não contêm estado, pois representam o ponto de ligação entre outras classes e gerenciam o fluxo de dados. Controladoras não devem conter lógicas complexas de negócio e devem oferecer um número limitado de serviços para outras classes, mantendo o foco em uma determinada tarefa [8]. A camada de modelos representa as entidades relacionais do modelo de negócio. Nesta camada, outras categorias de arquiteturas podem ser empregadas como, objetos, repositório e serviços. A primeira versão do software foi desenvolvida utilizando a arquitetura MVC, fornecendo toda a estrutura necessária para criação de uma aplicação web. Contendo 8 controladoras, 9 *models* e 5 controladoras específicas para gerenciar as *views* correspondentes ao módulo de cliente e a camada de visualização, como explicadas nas subseções a seguir.

4.3.1 Controladoras das Visualizações

Para o controle das páginas de visualização (*views*) foram utilizadas cinco classes. Essas classes podem também ser chamadas controladoras ou classes controladoras. As controladoras desse sistema são a *BasicExampleCtrl*, *ProjectController*, *CriterionController*, *ResultController* e *ScaleController* e terão seu funcionamento detalhado nessa seção.

Começando pelo *BasicExampleCtrl*, sua função é controlar a página da árvore de critérios. O funcionamento dessa classe começa pelo método *find()*, que busca todos os critérios de um determinado projeto. Com os critérios carregados, o método *list_to_tree()* é utilizado para organizar a lista de critérios em um formato para ser exibido em tela. A criação, edição e remoção de um critério também é feita pelo *BasicExampleCtrl*. A controladora *BasicExampleCtrl* também gerencia a visualização dos subcritérios de um critério, dando também possibilidade de expandir ou retrair todos os critérios.

A classe *ProjectController* é utilizada em grande parte das páginas do sistema, mas a maioria dessas páginas utiliza apenas a parte de buscar os dados do projeto para exibir o título do projeto. *ProjectController* é utilizada de forma mais ampla na página inicial pós acesso ao sistema para exibir os projetos e escalas do usuário. Para controlar a criação e edição de projetos, *ProjectController* também é utilizada, bem como a criação, edição e exclusão de escalas.

A controladora *CriterionController* é utilizada na página de taxas de contribuição. Nessa página é possível escolher a porcentagem de contribuição de cada critério para o projeto. A classe *CriterionController* busca os critérios e organiza os dados para exibir em tela. Ao fim do preenchimento das taxas de contribuição, a classe *CriterionController* verifica se a soma das porcentagens equivalem a 100 e, se estiver tudo certo, envia os dados para processamento.

Apesar de ser uma classe grande, a *ResultController* se responsabiliza apenas pela página de resultados do projeto. A classe *ResultController* tem diversas funções nessa página, começando pela busca dos critérios e para cada critério é feita outra busca pelos seus resultados. Com os critérios e seus resultados, os dados são organizados de forma que página seja exibida com as informações organizadas para o usuário. Além da conter a responsabilidade pelas tabelas da página, a classe também gerencia a exibição dos gráficos em cada uma das três abas de resultados, quando necessário.

Por fim, a classe *ScaleController* é responsável pela página de ordenação de critérios e pela página de seleção da moda de um critério. Para cada uma das páginas os critérios são buscados em um método específico e cada busca é tratada de uma maneira diferente pela controladora. Na página de ordenação, os critérios ordenados são buscados e exibidos em tela para poderem ser reordenados pelo método de clicar e arrastar. Na página de escolha de modas, a classe *ScaleController* realiza a busca dos critérios e escalas e com esses dados

faz o cálculo dos valores de cada um das opções de escala. Após o preenchimento das modas do critério, é feita uma validação pela *ScaleController* para verificar se todos os critérios foram preenchidos corretamente.

4.3.2 Controladoras

A controladora *RegisterController* é responsável por fornecer o serviço de registro de novos usuários, bem como a validação, intermediando o fluxo de informações enviado pelo módulo de visualização até a persistência no banco de dados. Por padrão, este controlador visa fornecer essa funcionalidade sem exigir nenhum código adicional.

Os serviços de conexão, cadastro e *login* de usuários com redes sociais são fornecidos pela controladora *SocialAuthController*. O gerenciamento do acesso aos provedores de redes sociais são administrados por esta controladora, com objetivo de concentrar os serviços em apenas uma classe. O módulo de cliente envia uma requisição com as especificações do serviço que deseja utilizar e as informações exigidas, a controladora envia os dados aos provedores da rede social em específico que retornam as informações do usuário caso as credências sejam válidas. Em posse desses dados, os modelos utilizados pela *SocialAuthController* persistem os dados do usuário. Os provedores das redes sociais *Google* e *Facebook* foram acoplados a esta classe.

A atribuição de carregar o conteúdo da visualização e o serviço de atualização dos dados de usuário é realizada pela controladora *AccountController*. O serviço *update()* presente na classe, quando requisitado, recebe as informações do usuário e utiliza o modelo *User* de modo a persistir as informações no banco de dados. A função *index()* retorna o documento HTML com formulário de atualização dos dados do usuário. A classe *UsersController* é responsável apenas por carregar duas visualizações, a *index()* e a *privacy()* onde os termos de uso e política de privacidade da aplicação são disponibilizados, de acordo as exigências de uso e proteção dos dados captados.

A controladora *ProjectController* fornece diversos serviços, divididos em dois tipos: os que proveem conteúdos das páginas da aplicação relacionadas a projetos, e os que persistem, consultam, atualizam e removem uma determinada informação ou conjunto de informações. Os serviços para prover páginas são os métodos, *index()*, pois carrega a página que lista os projetos de um usuário, *create()* apresenta o formulário para cadastro de projeto e a *edit()* para o formulário de atualização de projeto. Os outros métodos visam a manipulação dos registros de projetos, encarregadas de fornecer os seguintes serviços: *find()* consulta os dados pelo identificador do usuário, *findProject()* retorna as informações pelo identificador do projeto, *getAnswersByProject()* retorna as opções de resposta de um projeto, *store()* cria ou edita informações e por fim *remove()*, que exclui o registro do banco de dados.

Os serviços responsáveis pela manutenção e gerenciamento das opções de resposta é realizado pela controladora *OptionAnswerController*. Nesta classe estão contidos três serviços, conectados com a classe de modelo *OptionAnswer*, que realizam as funções de consulta pelo identificador do projeto e remoção dos registros por identificador do projeto ou da opção de resposta. Sendo eles respectivamente, *getOptions()*, *remove()* e *removeByIdProject()*.

Uma das principais controladoras do sistema é a *CriterionController*, onde são realizados todos os serviços relacionados aos critérios da aplicação. A classe fornece as funções que retornam as páginas *web*, onde são construídas a árvore, ordenação, resultado da análise dos critérios entre outras, totalizando seis visualizações. Além disso, a classe também possui serviços para consulta de informações como, *findOrder(), findOrderWithProject(), findScaleResultByCriterion(), find() e findTree()*. Já para manipulação dos dados, há serviços que executam a persistência e atualização de registros, sendo *store(), saveSort(), saveScaleResult()* e outro para exclusão *remove()*.

A controladora *ScaleController* gerencia os serviços relacionados a classe modelo Escala e Opções de resposta, conforme representado na figura 4.1. A consulta de informações é realizada por duas funções *getAll()* e *getScaleUser()*, a persistência das informações é de responsabilidade das funções *store()* e *saveOptionsAnswer()*, e por fim, a exclusão de uma escala é efetuada pela função *remove()*.

4.3.3 Modelos

As classes de modelo são responsáveis por controlar e gerenciar a maneira como os dados se comportam na aplicação através dos métodos e da lógica de negócio. A representação da base de dados na aplicação é também uma das atribuições dessas classes, onde cada uma recebe um parâmetro que identifica a qual tabela do banco de dados está representando. A seguir as classes modelos serão listadas.

A camada de modelo conta com um total de nove classes, iniciando pela classe *User*, responsável pelo armazenamento dos dados do usuário da aplicação, bem como a classe *SocialAccount* que armazena o identificador de redes sociais dos usuários que utilizam este meio para cadastro e *login*. Em conjunto com a *SocialAccountService* que fornece métodos para que essas ações sejam concluídas. A classe *Project* persiste os dados de projeto. Os projetos possuem uma escala que é representada pela classe *Scale*. As opções de respostas são caracterizadas pela classe *OptionAnswer* que guarda os dados utilizados para avaliar os critérios, esste representado por *Criterion*. Outras duas classes guardam as informações ligadas ao mesmo, sendo a *SortLastCriteria* para ordem de importância e a *ScaleResult* para o resultado dos cálculos das escalas de cada critério.

4.3.4 Análise Arquitetural

Uma análise em nível arquitetural foi realizada de modo a avaliar e identificar possíveis falhas na arquitetura que pudessem gerar *code smells*. Analisando o sistema a nível arquitetural com auxílio da figura 4.1, levantaram-se algumas questões em relação às boas práticas de programação e manutenibilidade do código.

Na primeira versão, muitas das regras de negócio da aplicação eram mantidas na camada de visualização. Na construção das visualizações (*views*), o sistema realizava uma série de cálculos, o que tornava o processamento de dados oneroso, violando o próprio padrão de desenho MVC, além de ser um grave erro arquitetural, pois a camada de visualização não pode conter as regras de negócio. Para exemplificar, o método *calculeteFinalResult* da controladora *ResultController* da camada de visualização faz o cálculo de valores para exibir na página de resultados, algo que deveria ser calculado pelo servidor. A controladora *CriterionController* presente na região *controllers* da figura 4.1 é um exemplo do *code smells* chamado de Classe Extensa [6]. Essa categoria de equívoco é bem comum em arquiteturas MVC, visto que a classe assume diversas responsabilidades que deveriam ou poderiam estar em outras entidades [8].

Um *code smell* conhecido como *Speculative Generality* [6] ou generalidade especulativa, é identificado quando um método é descrito com nome abstrato e/ou estranho. Este comportamento foi identificado em três controladoras: *ProjectController*, *CriterionController* e *ScaleController*. A partir da avaliação da figura 4.1 notou-se a inexistência de métodos específicos para atualização de registros, somente para armazenamento de dados. No entanto, em uma análise mais detalhada verificou-se que estes métodos de armazenamento realizavam ambas as tarefas. Assim, de modo a corrigir este problema, duas técnicas podem ser empregadas: renomear o método tornando-o mais claro e descritivo ou extrair uma das tarefas para um novo método.

Outro problema detectado, foi o uso de controladoras para realizar determinadas funções e/ou manipulação de dados que deveriam ser realizados por outras classes. Esse é um clássico caso de *Feature envy* [6], termo utilizado para se referir ao comportamento de métodos ou funções que parecem mais interessados nos dados provenientes de outras entidades do que nos dados de sua própria classe [7]. Seguindo na análise desse problema, a controladora *ScaleController* persiste as informações do modelo *OptionAnswer* que é de responsabilidade da controladora *OptionAnswerController*. Observa-se este mesmo comportamento na *CriterionController* que deveria manipular os dados do modelo *Criterian*, mas realiza operações em dois outros modelos: *ScaleResult* e *SortLastCriteria*. Uma Informação interessante levantada por Aniche et al. [8], mostra que os desenvolvedores gastam mais tempo na manutenção de classes afetadas pelos *code smells* de Classe Extensa e *Feature envy* do que em outros problemas, reforçando a importância da correção destes

equívocos.

Outros problemas foram identificados nas controladoras responsáveis por gerenciar a camada de visualização, apresentada na figura 4.1 na região das *Views*. Por exemplo, a controladora *BasicExampleCtrl*, é possível notar que a classe contém vários métodos, caracterizando uma Classe Extensa, que é quando uma classe está realizando muitas tarefas. Quando isso ocorre, pode-se intercorrer um Código Duplicado [6]. Para uma classe com muito código, a ação sugerida é Extrair Classe ou Extrair Subclasse. A suspeita por Código Duplicado em *BasicExampleCtrl* aumenta quando se observa os métodos *newItem* e *newSubItem*. Os métodos *save* e *saveNodeSelecion* também podem ter Código Duplicado por terem nome e possivelmente funcionalidades similares.

Observando o diagrama na figura 4.1, a controladora *ProjectController* do módulo de visualização (*view*) é bem extensa com relação ao número de métodos, o que caracteriza uma Classe Extensa. Para a resolução de Classe Extensa, Fowler [6] sugere utilizar Extrair Classe ou Extrair Subclasse quando há muito código. Outra característica da *ProjectController* da visualização (*view*) é a sua utilização em várias páginas, ou seja, mais um indício de que a controladora pode estar fazendo tarefas em excesso. Os métodos *changeGood()* e *changeNeutral()*, *find()* e *findProject()*, *getScales()* e *findScales()*, *remove()* e *removeScale()* são possíveis candidatos ao *code smell* Código Duplicado.

A controladora *CriterionController* do módulo *view* aparenta ter uma quantidade razoável de métodos, mas não o suficiente para caracterizar uma Classe Extensa. Uma avaliação a nível de código será realizada para determinar se é o caso ou não. Pelos nomes, os métodos *saveEffort()*, *saveContribuitionRateOrEffortGerneral()* e *saveContribuitionRate()* são possíveis candidatos ao *code smell* Mudança Divergente. A Mudança Divergente ocorre quando toda mudança na lógica acarreta atualização de múltiplos métodos na mesma, muito similar ao Código Duplicado. Para resolver o *code smell* Mudança Divergente, é necessário criar uma classe com a técnica Extrair Classe.

Em *ResultController* do módulo de visualização, há grandes problemas com *code smells*, o mais claro sendo o tamanho da classe já caracterizando uma Classe Extensa, observado pelo número de variáveis de instância e pelo número de métodos. Pela lista de atributos, é possível observar que todas as variáveis com o início *main* tem grandes probabilidades de ser o *code smell* Agrupamento de Dados, que é quando os três ou mais dados sempre estão sendo usados em conjunto em múltiplos métodos. A solução para o *code smell* Agrupamento de Dados é extrair a classe com os campos para transformar esses dados em um objeto [6].

Existem muitos exemplos de Código Duplicado na classe *ResultController* da *view*. Primeiramente, os métodos *buildLevels()*, *checkLevelCriterion()* estão presente tanto em *ResultController* quanto em *CriterionController*, sugerindo uma Extração de Classe. O

método *findProject()* esta presente em *ResultController*, *CriterionController* e em *ProjectController*. Além disso, é possível notar que existem dois métodos com o mesmo nome: *fillDataWithResult()* e uma possível solução é a remoção de um dos métodos.

Por fim, a classe *ScaleController* do modulo de visualização também sofre com o problema de Classe Extensa. É possível notar que o método *list_to_tree()* é um Código Duplicado da classe *BasicExampleCtrl* e que *fillDataWithResult()* é um Código Duplicado de *ResultController*. Conforme explicado anteriormente, a forma de se mitigar um Código Duplicado em classes diferentes é utilizando Extrair Classe.

Em todas as controladoras, os métodos e as variáveis de instância da figura 4.1 marcadas com o círculo vermelho possuem alguma falha de implementação ou *code smells*. No Capítulo 5 estes problemas foram detalhados e corrigidos. Além dos *code smells* encontrados a nível arquitetural, o presente trabalho trata também de *code smells* a nível de código no Capítulo 5.

Na figura 4.1, as conexões de seta entre as controladoras da camada de visualização (*view*) e as páginas descritas por texto simples representam que a página usa informações fornecidas pela controladora. As conexões de seta entre as classes de *controller* e os *models* representam que a classe controladora utiliza o modelo indicado em alguns de seus métodos.

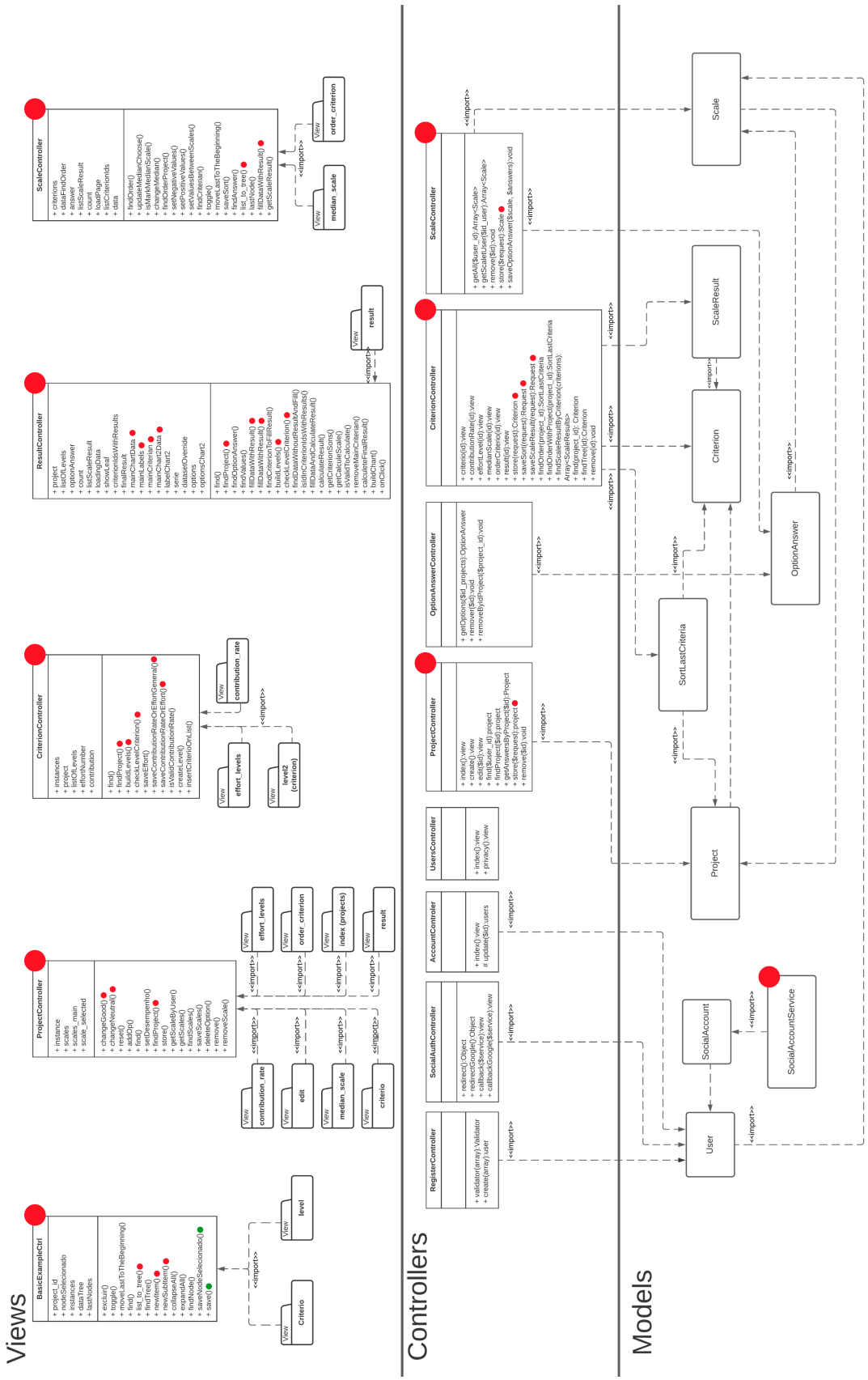

Figura 4.1: Diagrama de Classes (as controladoras, métodos e atributos marcados com um círculo vermelho contém *code smells*) .

4.4 Diagrama de Classes do Projeto Novo

Nesta seção é apresentado o diagrama de arquitetura da aplicação e a justificativa do uso da arquitetura proposta. São apresentados também os diagramas de classe do módulo de cliente e do módulo de servidor. Para cada módulo é explicado funcionamento de cada classe, suas conexões e o motivo de sua implementação para mitigar os *code smells*.

4.4.1 Arquitetura da Aplicação

Nesta nova versão, a arquitetura da aplicação funciona diferentemente. A primeira versão era baseada na arquitetura MVC onde todos os módulos estavam concentrados em uma única aplicação. A camada de visualização foi reprojetada em uma nova arquitetura, separada das camadas de controle e modelo. Uma nova aplicação foi construída para administrar a camada de visualização onde o usuário interage com o sistema, fornece e recebe os dados. Desta forma, uma das falhas das arquiteturas da primeira versão foi removida, tirando as regras de negócio que foram erroneamente desenvolvidas na camada de visualização e transferidas para uma arquitetura exclusiva, para atender as requisições onde se encontram os módulos de controle e modelo. Na versão nova do sistema, o módulo de servidor e o módulo de cliente foram desenvolvidos de forma independente. Com essa arquitetura, foi possível desenvolver módulo de cliente e servidor paralelamente. Na figura 4.2, observa-se uma representação do fluxo de dados do sistema.

Partindo pelo usuário, o sistema pode ser acessado através de um navegador (*browser*). Ao realizar o acesso, é feita uma requisição da página ao servidor. Essa requisição chega ao servidor de aplicação *NGINX*, que busca os arquivos do projeto Vue.js que é um *framework* para desenvolvimento de interfaces web. Com os arquivos já compilados para a utilização *web*, o *NGINX* retorna esses arquivos para o usuário. O *NGINX* é um software de código aberto que pode, entre outras funcionalidades, ser utilizado como um servidor *web* [17]. Por ser um *SPA*(Single-Page Application), o Vue.js retorna todos os arquivos e páginas em apenas uma requisição, possibilitando que futuramente a aplicação possa funcionar (até um certo ponto) sem conexão com a rede. *SPA* é uma implementação de aplicação que carrega apenas um único documento web, e então atualiza o corpo desse documento utilizando *javascript* [18].

Com a página já carregada no navegador, o usuário pode interagir com o servidor por meio da interface. Ao fazer uma ação no sistema que manipula dados que necessitam persistir entre sessões, uma requisição em formato *JSON* é enviada ao módulo de servidor, representado pela área do *Node.js* na figura 4.2. O *Node.js* é um *framework* que permite a execução de códigos *javascript* em um servidor e o JSON (JavaScript Object Notation)

é um modelo para armazenamento e transmissão de informações no formato texto, muito utilizado em aplicações Web [19].

Ao receber a requisição, o módulo de servidor faz a correspondência com a rota requisitada e envia os dados recebidos para a controladora, chamando método correspondente. Na controladora, os dados recebidos são utilizados para fazer a manipulação dos modelos. O modelo efetua então a consulta no banco de dados *MySQL* e formata os dados para devolver a controladora. A controladora utiliza esses dados para realizar as operações necessárias e por meio das rotas devolve uma resposta em formato *JSON* para o navegador. Ao chegar no navegador, o *Vue.js* se encarrega de atualizar o conteúdo da página com a resposta recebida.

O sistema de *login* e sessão dos usuários foi reconstruído para controlar o acesso à aplicação. Com a nova arquitetura desenvolvida em que os módulos de cliente e servidor estão desacoplados, os módulos foram projetados para utilizar um método de autenticação entre duas partes através de um *Token* criptografado e assinado, que certifica uma requisição *web*. O *JSON Web Token*(JWT) gerado pelo módulo de servidor no *login*, contém as informações que identificam o usuário que está acessando o sistema, além de incluir os dados para verificar a validade do mesmo. JWT é um padrão aberto que define uma forma compacta e auto-contida de transmitir informação seguramente entre duas partes como um objeto *JSON* [20]. O JWT então é enviado pelo módulo de cliente no cabeçalho de cada requisição. Com isso, o módulo de servidor então valida esse cabeçalho e procede ou não com a requisição.

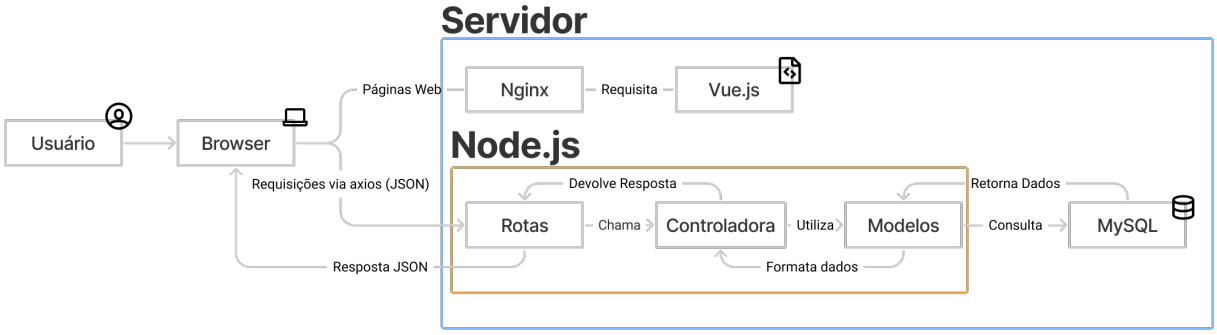

Figura 4.2: Diagrama da arquitetura web.

4.4.2 Proposta Arquitetural do Módulo de Servidor

A nova arquitetura do servidor é divida em três camadas, como apresentado na figura 4.3. Nesta nova etapa o modulo do servidor foi construído separadamente do módulo de cliente. As camadas estão separadas entre, controladoras(*Controllers*), modelo(*Models*) e serviço(*Service*), onde cada uma tem uma função específica. As controladoras gerenciam o

fluxo e controle dos dados, as classes modelos representam a abstração do banco de dados e realizam a gerência do comportamento das informações por regras de negócio, lógica e funções. Já a camada de serviço fornece métodos compartilhados entre as controladoras com intuito de evitar a duplicação de código, aliado as boas práticas de programação. A seguir as camadas do servidor serão descritas.

Na figura 4.3, o diagrama de classes representa o sistema em sua versão 2.0. As ligações entre as classes foram coloridas para tornar melhor a visualização da figura. A estrutura do diagrama permanece bem semelhante à figura da primeira versão no que se refere a organização, porém grandes mudanças arquiteturais foram realizadas de modo a remover as falhas encontradas na primeira versão. É possível notar que houve uma diminuição no número de classes, pois uma controladora foi removida e das nove classes de modelos, restaram-se sete classes.

Controller

Com intuito de melhorar a qualidade do código, manutenção das boas práticas de programação e adequar-se a nova linguagem, modificações foram realizadas na camada de controle. Em relação à primeira versão as classes controladoras *RegisterController*, *SocialAuthController* e *AccountController* foram removidas, com as funções da *RegisterController* e *AccountController* sendo transferidas para controladora *UserController* de modo a concentrar todos os métodos relacionados a usuário. Uma nova controladora chamada *SessionController* foi criada com objetivo de reunir todas as funções relativas a controle de acesso ao sistema e redes sociais, assim absorvendo as atribuições que antes eram realizadas pela *SocialAuthController*. Em todas as controladoras, métodos para validação e tratamento de exceções foram criados, com intuito de retornar mensagens claras e objetivas de sucesso ou erro nas ações executadas, o que não ocorria na versão anterior.

A controladora *ProjectController* tem o mesmo comportamento da primeira versão, porém agora conta com uma validação de dados mais robusta, um método específico para atualizar as informações e uma função para duplicar os dados de um projeto, esta última sendo parte das melhorias adicionadas ao novo *software*. Como mencionado anteriormente a controladora *UserController* reúne todas as funções relacionadas a usuários, como inclusão, atualização e busca de usuário, além da validação de dados. A *ScaleController*, realiza as mesmas ações da primeira versão como descrito na Seção 4.3.2. Alguns métodos tornaram-se obsoletos na classe *OptionsAnswersController* sendo implementado apenas a função para exclusão. A *CriteriaController* foi otimizada em relação à classe anteriormente criada, passou de quinze métodos para apenas sete, limitando-se a gerenciar o fluxo de dados relacionados a classe modelo *Criterian*.

Na atual versão duas novas controladoras foram adicionadas *SessionController* e *ScaleResultController*. A primeira como mencionado no início da seção, foi desenvolvida para controle de acesso ao sistema e gerenciamento conexões via redes sociais. Antes essas funções estavam dispersas por diversas classes, agora estão concentradas em apenas um local com maior controle e robustez. A classe *ScaleResultController* é responsável por fornecer os dados das análises realizadas pelo sistema, antes essas funções estavam todas reservadas para controladora *CriteriaController*, porém para melhor organização e modelagem do código está nova classe ficou com esta tarefa em específico.

Service

A camada intermediária *Service* foi criada com intuito de facilitar o compartilhamento de serviços entre as controladoras. Como mencionado em seções anteriores, muitos dos cálculos eram realizados pela camada de cliente, causando um problema arquitetural. Nesta nova versão, os cálculos da modelagem multicritério foram transferidos para a classe *Util*, podendo ser utilizada por outras controladoras. Esta classe possui dez funções, empregada pelas classes *ScaleResultController* e *CriteriaController*, como apresentado na figura 4.3. Outro arquivo pertencente a esta camada é o *SocialUser* responsável por auxiliar a controladora *SessionController* na persistência dos dados de usuários oriundos de redes sociais, ela realiza conexão com duas classes modelos correlatas, sendo elas *User* e *SocialAccount*.

Modelos

A camada de modelos da versão atual é bem semelhante à primeira implementação, a única diferença é a linguagem *JavaScript* que os modelos foram desenvolvidos. As classes gerenciam a maneira como os dados se comportam na aplicação da mesma forma como descrito na Seção 4.3.3. A classe *SortLastCriteria* foi removida no novo sistema, pois as suas atribuições foram absorvidas pelo modelo *Criteria*, sem prejuízo a qualidade do código.

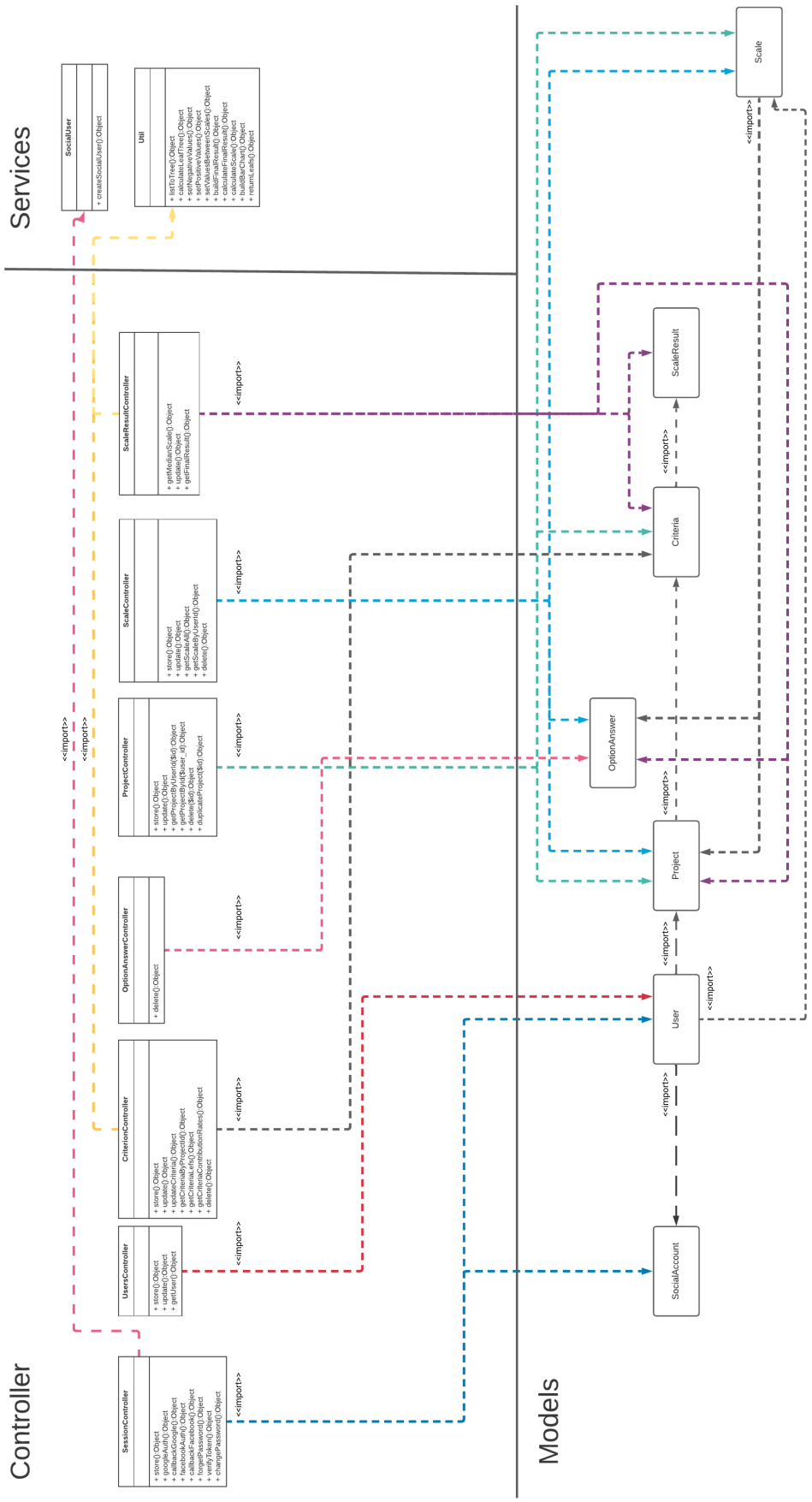

Figura 4.3: Novo Diagrama de Classes (Módulo de Servidor).

4.4.3 Análise Arquitetural do Módulo de Servidor

Um dos *code smells* presentes no sistema e detectados por meio da Figura 4.1 é o da Classe Extensa. A controladora *CriterionController* possuía quinze serviços dos mais diversos tipos relacionados ou não com objetivo da controladora, o que caracterizava um exemplo deste problema. De modo a remover este erro de projeto, uma estratégia conhecida como extração de classe [6] foi aplicada, consistindo em transferir os serviços que não fazem parte especificamente de uma classe para outra com as mesmas finalidades. Um exemplo dessa estratégia foi a transferência do serviço *saveScaleResult()* da *CriterionController* para *ScaleResultController*, presente na Figura 4.1. Na nova versão a controladora *CriterionController* passou de quinze para sete serviços, eliminando o problema da Classe Extensa.

Uma função com comportamento muito generalista, onde realiza mais de uma tarefa aliado a um nome dúbio que não descreve o seu real propósito, era um equívoco cometido em algumas classes controladoras da versão anterior. Com intuito de solucionar este problema, foram aplicadas duas estratégias, uma para extração do método que consiste em retirar o trecho de código que está presente em uma função, mas realiza uma tarefa diferente do estava proposto. Outra estratégia é a de *Mover o Método* que consiste em transferir o método para classe que melhor define sua utilidade [6].

A controladora *CriterionController* possuía três exemplos desse problema, a função *store()* realizava criação atualização de critérios, no novo sistema ela foi mantida como demonstrado na Figura 4.3, porém uma nova função chamada *update()* foi adicionada especificamente para atualizar e validar os dados, ou seja, evitando que a função realizasse duas tarefas, aplicando a técnica de extração de método. O comportamento da função *saveScaleResult()* foi transferido para o método *update()* da controladora *ScaleResultController* criada especificamente para controlar as informações dos resultados das escalas. O último problema classe estava na função *saveSort()*, porém com o novo modelo este método tornou-se obsoleto e foi removido.

Outra classe que compartilha do mesmo problema é a *ProjectController* que na versão anterior possuía apenas um método *store()* que gravava e atualizava os dados do projeto, já na nova versão conforme a Figura 4.4 a classe controladora de projeto possui duas funções sendo uma para salvar e outra para atualizar os dados, *store()* e *update()* respectivamente. Assim o problema de generalidade especulativa foi resolvido. Ainda nessa categoria de erro a classe *ScaleController* continha um exemplo de erro de generalidade, o primeiro método *store()* realizava duas tarefas, assim sendo extraído e duas funções novas foram criadas para cada responsabilidade.

Um problema recorrente encontrado na primeira implementação conhecido como *feature envy*, onde a classe implementa alguns métodos mais interessados em dados de outras

classes do que da própria, como já descrito em seções anteriores. De modo a corrigir esse problema, a estratégia de *mover o método* foi aplicada para que assim as funções estivessem estabelecidas nas controladoras corretas. A classe *CriterionController* presente na Figura 4.1 realizava a busca de informações pela função *findScaleResultByCriterion()*, persistência e atualização de informações por *saveScaleResult()*, sendo todas relacionadas a classe modelo *ScaleResult*. Com intuito de separar as responsabilidades e mitigar o *code smell* encontrado, uma nova classe controladora foi criada especificamente para gerenciar essas informações, chamada *ScaleResultController* apresentada na Figura 4.3. As funções foram migradas, refatoradas e outras adicionadas, a *update()* foi criada substituindo o trecho da função *saveScaleResult()* referente a atualização das informações. Além da função *findScaleResultByCriterion()* ter sido refatorada e substituída pela *getFinalResult()* que possui maior robustez e retorna os dados de forma mais precisa.

Algumas implementações realizadas no módulo de cliente da primeira versão continham *code smells*. Dentre esses problemas, um é denominado *código duplicado* [6], ocorre quando uma função é implementada uma ou mais vezes no mesmo sistema. Além disso, muitos desses métodos foram implementados com lógicas de negócio que não deveriam ser providas pelo módulo de cliente e sim pelo servidor. A função *list_to_tree()* estava presente em duas controladoras *BasicExampleCtrl* e *ScaleController* da *view* apresentadas na Figura 4.1. De modo a corrigir esse problema, a função foi transferida para a classe *Util* no módulo *Services* presente na Figura 4.3, deste modo qualquer controladora pode utilizar esta função sem a necessidade de duplicar o código.

4.4.4 Proposta Arquitetural do Módulo de Cliente

Ao utilizar o Vue.js todo componente, seja uma página, uma tabela ou um botão, é controlado por uma classe. Ao escrever o HTML do componente, é possível renderizar outros componentes no original. Para corrigir os problemas estruturais no módulo de cliente antigo, foram criados componentes reutilizáveis para evitar o Código Duplicado que é um dos maiores problemas da versão antiga. Componentes visuais que não possuem nenhuma lógica de funcionamento foram excluídos da Figura 4.4, pois não são relevantes para o trabalho. Métodos *created()* e *mounted()* são executados sempre que o componente é criado e renderizado, respectivamente.

A classe *Login* da Figura 4.4 é responsável por realizar o acesso do usuário ao sistema e guardar as informações relevantes sobre a identidade do usuário para realizar ações futuras no sistema. O formulário de acesso é representado pela variável *form*. O método *submit* é responsável por enviar o *form* preenchido para o módulo de servidor. Em caso de erro (senha incorreta, por exemplo), o usuário é notificado e durante a espera pela resposta da requisição a variável *done* impede que a ação seja enviada novamente. Caso o

usuário deseje realizar o acesso pelas redes sociais, o método *redirectToSocialLogin* envia uma requisição com o link de acesso gerado no módulo de servidor e o redireciona para tela de login da rede social específica.

A classe *Register* é responsável pelo cadastro do usuário no sistema. Ao realizar o cadastro, o usuário é automaticamente conectado ao sistema. Como a tela de acesso e a de registro resultam no acesso do usuário, um módulo externo se responsabiliza por fazer a conexão do usuário, evitando o *code smell* Código Duplicado. *ProfileEdit* na Figura 4.4 é responsável pela edição dos dados pessoais do usuário conectado ao sistema. Em *ProfileEdit*, o usuário pode também atualizar a sua senha de acesso. Com relação ao sistema anterior, essa tela não foi alterada.

No sistema atual, a lógica de exibir o título foi encapsulada na classe *ProjectHeading*, que busca os dados do projeto e exibe nas páginas que o utilizam. O componente é aplicado no interior da página da árvore de critérios, na página de resultados do projeto, na tela da escolha de modas e na tela de ordenação. O componente *ProjectHeading* busca as informações utilizando o método *created* que faz a requisição para o módulo de servidor. *ProjectsTable* e *ScaleTable* são componentes com a função de lidar com a tabela de projetos e de escalas, respectivamente. Na classe da tabela de projetos é possível realizar a ação de excluir um projeto. Já na classe da tabela de escalas, além de ser possível excluir, também é possível criar e editar uma escala com o componente de *ScaleFormModal* que será explicado com mais detalhes. Quando é enviada uma requisição para o módulo de servidor, ao obter uma resposta, ambas as classes *ProjectsTable* e *ScalesTable* atualizam o conteúdo da página.

Por ser mais complexa, a criação e edição de um projeto ficam em uma página separada, controlada pela classe *ProjectsForm*. Nessa tela existe uma lógica que utiliza a *performance* captada no formulário *form* e a formata para exibir ao usuário a *performanceMax* e a *performanceMin*. Essa classe *ProjectsForm* também utiliza um componente interno que busca e lista as escalas do usuário, podendo ser utilizado em outro local caso necessário. O componente da classe *ScaleFormModal* é utilizado internamente em *ScalesTable* para lidar com a criação e edição de uma escala. Ao criar ou editar uma escala, os dados sobre essa escala são repassados para o componente pai, no caso *ScalesTable*. A atualização da tabela é feita quando *ScalesTable* recebe os dados de *ScaleFormModal*.

A tela seguinte à criação e edição de projetos é a tela da árvore de critérios. Os dois principais componentes utilizados na tela de árvore de critério são *CriteriaTree* e *CriterionFormModal*. *CriteriaTree* é responsável pela exibição dos critérios e seus subcritérios. O componente da classe *CriteriaTree* foi realizado recursivamente, ou seja, tem como componente interno o próprio *CriteriaTree*. A classe *CriterionFormModal* também é um componente interno de *CriteriaTree* é responsável pela criação e edição de um critério.

CriterionFormModal funciona de maneira similar à *ScaleFormModal* e em caso de sucesso na criação e edição de um critério, repassa os dados para o componente pai atualizar a tela.

Os componentes *CriteriaTree* e *CriterionFormModal* são internos ao componente *ProjectCriteria*. A classe *ProjectCriteria* é responsável por buscar os critérios do projeto e formatar esses dados para *CriteriaTree*. Após a criação de um critério em *CriterionFormModal*, *ProjectCriteria* envia o novo critério para a *CriteriaTree*. De maneira mais simples, *ProjectCriteria* é responsável por fazer a ligação entre o formulário de critérios e a árvore de critérios, mantendo os dados sempre atualizados.

Após a criação/edição da árvore de critérios, a próxima tela é a de taxas de contribuição. Nessa tela existem múltiplas tabelas para que o usuário possa especificar qual a relevância de cada critério, com um valor de 1 a 100. Cada uma dessas tabelas é controlada pela classe *ContributionRate*. A classe *ContributionRate* é responsável por enviar as taxas de contribuição para o módulo de servidor, além de verificar se a soma das taxas de contribuições da tabela somam 100% e alertar o usuário caso não seja. A tela seguinte é a de ordenação dos critérios, onde o usuário pode clicar e arrastar para trocar a ordem dos critérios. A classe responsável por essa página é a *CriteriaOrderForm*. Quando o usuário clica em "salvar"nessa página, a ordem que está sendo exibida em tela é enviada para o módulo de servidor e então é gravada.

Ao realizar a ordenação de critérios, o próximo passo é escolher a resposta moda para cada critério. O componente principal dessa tela é o *CriteriaModesForm* que é um formulário faz a listagem das opções de cada critério para o usuário marcar a resposta moda. O método *isValid* determina se o formulário de *CriteriaModesForm* foi preenchido corretamente. Caso o formulário seja preenchido corretamente o usuário pode salvar e passar para a próxima página, caso contrário a navegação para a próxima página fica bloqueada.

Ao escolher as modas e salvar, o próximo e último passo do projeto é a visualização dos resultados. *ProjectResults*, *ProjectResultsAreaGraph* e *ProjectResultsBarGraph* são os principais componentes dessa tela. O componente *ProjectResults* tem como responsabilidade buscar os resultados do projeto, e para cada resultado mostrar o gráfico de área e o gráfico de barras. *ProjectResultsAreaGraph* e *ProjectResultsBarGraph* são os componentes do gráfico de área e do gráfico de barras respectivamente. Nos dois gráficos, a variável *chartOptions* é utilizada para configurar o gráfico e a variável *series* serve para guardar que dados serão exibidos.

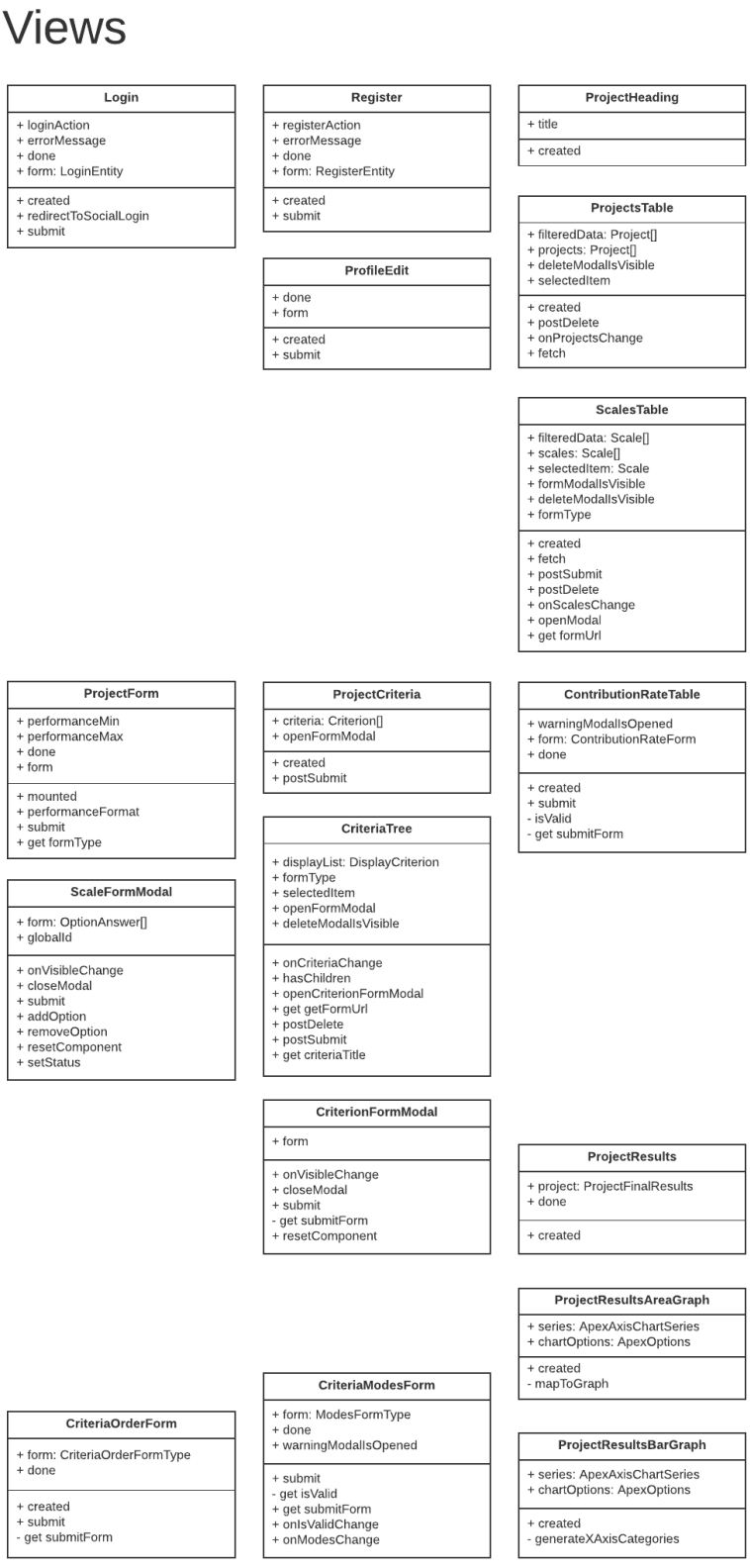

Figura 4.4: Novo Diagrama de Classes (Módulo de Cliente).

4.4.5 Análise Arquitetural do Módulo de Cliente

A classe *ProjectHeading* foi criada no sistema atual para resolver um *code smell* do antigo *ProjectController* presente na camada de visualização (*view*) da Figura 4.1. A classe *ProjectController* era instanciada em várias páginas do sistema antigo apenas para exibir o título do projeto, trazendo código desnecessário para as páginas. O código de exibição de título no projeto antigo era Código Duplicado em todas as páginas que o utilizavam. O componente *ProjectHeading* do sistema novo compõe a estrutura das páginas de projeto e com isso só é instanciado uma vez, evitando o Código Duplicado da função *findProject()* espalhada pelo projeto anterior.

A controladora *ProjectController* da camada de visualização no sistema antigo sofria com o *code smell* de Classe Extensa por vários motivos, entre eles ter responsabilidade sobre as escalas do projeto. Para mitigar Classe Extensa do *ProjectController* foi criada a classe *ScaleFormModal* no novo sistema, extraindo parte do código para uma nova classe. *ScaleFormModal* implementa as funções *changeGood()* e *changeNeutral()* de *ProjectController* com o método *setStatus()*. A lógica de adicionar e remover uma opção de resposta de uma escala e a lógica de salvar a escala também foi transferida para *ScaleFormModal*, o que reduziu bastante o tamanho da classe *ProjectController*. Para resolver o Código Duplicado em *CriterionController*, causados por *saveContributionRateOrEffortGeneral()* e *saveContributionRateOrEffort()* foi extraída uma nova classe. A *ContributionRateTable* foi responsável por corrigir parte do Código Duplicado de *CriterionController*. Por ser um método que valida, formata e envia os dados para o servidor foi realizada a segmentação desse código em três métodos menores. Apesar de não possuir o *code smell* de Classe Extensa, utilizar a estratégia de Extrair Classe em *CriterionController* contribuiu para tornar a classe mais enxuta.

A classe *ResultController* no sistema tinha o *code smell* de Agrupamento de Dados, além de ser uma Classe Extensa. Para resolver o Agrupamento de Dados das variáveis de montagem de gráfico, o conjunto de dados se tornou dois objetos próprios, o *series* e o *chartOptions* nas classes *ProjectResultsAreaGraph* e *ProjectResultsBarGraph*. Extrair os dados de *ResultController* para duas classes específicas também contribuiu para uma redução no tamanho da classe. A classe *ProjectResults* que busca os dados de resultados de projeto e repassa os dados para os gráficos e tabelas também contribuiu para mitigar a Classe Extensa. Grande parte dos cálculos de *ResultController* realizados na camada de visualização foram transferidos para o módulo de servidor, de modo a reduzir o tamanho da classe e remover lógica de negócio do módulo de cliente.

Além das correções dos *code smells* no módulo do cliente, foram implementadas novas funcionalidades no sistema. A Figura 4.5 representa as alterações realizadas no projeto livre de *code smells* em relação ao projeto com as novas funcionalidades. Classes repre-

sentadas com a borda grossa vermelha foram excluídas. Classes com a borda grossa verde foram adicionadas ao projeto. Por fim, as classes que possuem setas, indicam como a classe era na versão do projeto livre de *code smells* e como ficou após a inserção das novas funcionalidades. A implementação das novas funcionalidades são explicadas com mais detalhes no Capítulo 5.

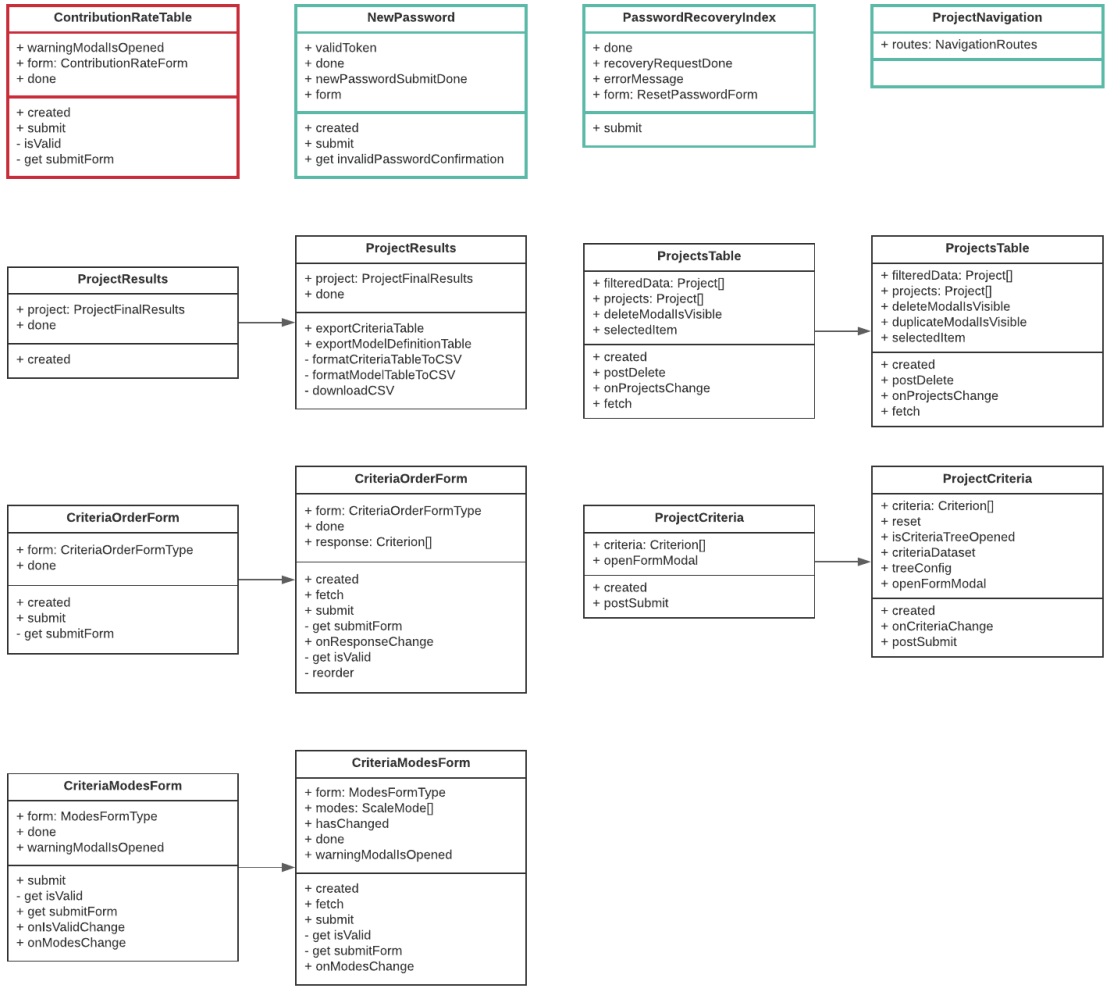

Figura 4.5: Mudanças causadas pela implementação de novas funcionalidades (Módulo de Cliente).

Capítulo 5

Protótipo

Em uma ampla análise realizada na versão 1.0 do software MyMCDA-C, diversos *code smells* foram encontrados. A metodologia utilizada para detectar e corrigir os problemas no código foram baseadas na obra de Martin Fowler, *Refatoração: Aperfeiçoando o Design de Códigos Existentes* [6]. Um total de 30 problemas em implementações foram detectadas nos modúlos do servidor do sistema e cliente, e em ambas as partes os problemas detectados foram corrigidos na versão 2.0 do software.

A nova versão do software MyMCDA-C no módulo de servidor foi desenvolvida na linguagem de programação JavaScript utilizando o framework *NodeJs(v15.5.1)*. A nova linguagem permite um maior controle e gerenciamento de estrutura de dados, além de permitir a modularização do código, o que viabiliza a aplicação de padrões de projeto e o fácil acoplamento de novos módulos e reuso de código. O módulo do cliente foi construído em *Vue.js(v2.0)*, um framework progressivo que possibilita a sua adoção incremental em projetos já existentes [21]. Além disso, o *Vue.js* possibilita a criação de componentes, o que deu estrutura para modularizar e reutilizar código que se repetia pelo projeto anterior, fornecendo a camada de visualização para os usuários.

Na versão anterior do sistema, a parte da lógica de negócio estava presente no módulo de cliente. Para evitar essa prática, a seção do sistema com a lógica de negócio foi transferida para o módulo do servidor. Com isso, o sistema possui uma divisão clara de responsabilidades. Lembrando-se que o presente trabalho tem por objetivo analisar o software MyMCDA-C com relação aos seus *code smells*, feito o levantamento destes em nível arquitetural, o próximo passo compreende a análise de *code smells* em nível de usabilidade.

Para a usabilidade do software, foi realizada uma reunião com usuários do sistema para entender os problemas enfrentados e o que seria interessante adicionar ao sistema. Incrementar o software para melhorar a experiência do usuário foi um dos objetivos do presente trabalho. Adicionar novas funcionalidades foi colocado como meta após a mi-

tigação dos *code smells* como uma forma de provar que o sistema com os *code smells* mitigados é de mais fácil manutenção.

O software MyMCDA-C sofria com uma navegação engessada. Para ir aos resultados de um projeto já concluído, era necessário passar por várias páginas que já foram preenchidas anteriormente, e não havia nenhuma forma de voltar para as páginas anteriores aos resultados sem ter que voltar à página inicial do projeto. Para resolver esse problema, a interface do novo sistema foi projetada com uma atenção especial para a usabilidade e navegação. O termo usabilidade é definido como o alcance onde um sistema, produto ou serviço pode ser utilizado por um usuário específico para alcançar um objetivo específico, com eficiência e satisfação num contexto específico de uso [22].

Por fim, também como objetivo secundário, foi realizado o re-desenho do sistema. Foi implementada uma interface mais limpa e fácil de entender e utilizar. Esse objetivo está diretamente ligado com usabilidade do software e experiência do usuário, de modo a tornar a utilização do sistema mais agradável. O termo experiência do usuário é definido como a percepção e resposta de um usuário pelo uso de um sistema, produto ou serviço [22]. Refatorar o software MyMCDA-C com tecnologias atuais foi colocado como um objetivo e será melhor explorado no desenvolvimento deste capítulo.

5.1 Mitigação de *code smells* no módulo de cliente

A classe *ProjectController* é responsável por mostrar um cabeçalho no contexto de projeto em múltiplas páginas. As páginas que usam extensivamente a classe *ProjectController* são a de listagem de projetos e critérios e a de edição e criação de projeto. Na página de listagem, o *ProjectController* é responsável por listar os projetos e os critérios, bem como lidar com criar, atualizar e excluir um critério. Nas páginas de edição e criação de projeto o *ProjectController* é responsável por editar e criar um projeto, além de buscar as escalas no módulo de servidor.

O *code smell* da Figura 5.1 é descrito por Fowler como uma Classe Extensa. A Classe Extensa ocorre quando uma classe está realizando mais tarefas do que deveria [6]. Nesse caso, a classe *ProjectController* faz operações com *scale_selected*, algo que deveria estar em *ScaleController*.

É possível notar também que o trecho da Figura 5.1 pode ser considerado Código Duplicado. O Código Duplicado ocorre quando uma estrutura de código está presente em múltiplos locais. Para resolver esse problema foi utilizada a Extração de Método, uma técnica que agrupa códigos similares no mesmo método para evitar *code smells* do tipo Código Duplicado e também Classe Extensa [6].

```
$scope.changeGood = function (option) {
    for(var i  in $scope.scale_selected.option_answer){
        if($scope.scale_selected.option_answer[i].answer != option.answer && option.good == 1 ){
            $scope.scale_selected.option_answer[i].good =0;
        }
    }
};

$scope.changeNeutral = function (option) {
    for(var i  in $scope.scale_selected.option_answer){
        if($scope.scale_selected.option_answer[i].answer != option.answer && option.neutral == 1  ){
            $scope.scale_selected.option_answer[i].neutral =0;
        }
    }
};
```

Figura 5.1: *Code smell* de Classe Extensa e Código Duplicado em ProjectController.

```
setStatus (option: OptionAnswer, status: 'neutral' | 'good') {
  this.form = this.form.map(scaleOption => {
    scaleOption[status] = false
    return scaleOption
  })
  option[status] = true
}
```

Figura 5.2: Correção do Code smell de ProjectController.

Os códigos das Figuras 5.3 e 5.4 também fazem parte do *ProjectController*. O método *reset* define o projeto selecionado (chamado *instance*) e a escala selecionada com um valor que representa que nada foi selecionado. O método *addOp* adiciona uma nova opção com valores padrão a uma escala selecionada.

Esses trechos contribuíam para o *code smell* Classe Extensa, já que estavam um pouco fora de contexto. Para resolver o problema, os trechos foram movidos para uma classe dedicada: *ScaleFormModal*. Essa técnica é definida por Fowler [6] como Extrair Classe sendo uma das recomendações para corrigir Classes Extensa.

```
$scope.reset = function(){
    $scope.instance = {
        'id' : null,
    };
    $scope.scale_selected ={
        'description': null,
        'option_answer': [{
            'id': null,
            'answer': null,
            'neutral': 0,
            'good': 0,
            'delete':1,
        }],
    };
};

$scope.addOp = function(){
    $scope.scale_selected.option_answer.push({
        'id': null,
        'answer': null,
        'neutral': 0,
        'good': 0,
        'delete': $scope.scale_selected.option_answer.length +1,
    });
};
```

Figura 5.3: Code smell de ProjectController.

```
$scope.saveScales = function(id){

    var description="";
    if($scope.scale_selected.user_id == null)
        $scope.scale_selected.user_id = id;

    for(var i in $scope.scale_selected.option_answer)
    {
        if( i == 0){
            description = $scope.scale_selected.option_answer[i].answer;
        }else{
            description =  description + " - " + $scope.scale_selected.option_answer[i].answer;
        }
    }
    $scope.scale_selected.description = description;

    $http.post("/scale/store",$scope.scale_selected).then(function (response) {

        appInfo("Successfully save!");

        getScaleByUser(response.data.user_id);

    }, function (response) {
    }).finally(function(){
        loadingCenter("pageContent",false);
    });

};
```

Figura 5.4: Code smell de ProjectController.

A classe *BasicExampleCtrl* é responsável pela manipulação de critérios de um projeto e pela ordenação da prioridade desses critérios. Os métodos *newItem* e *newSubItem* tem o objetivo muito similar de adicionar um novo critério a árvore de critérios. A diferença entre estes dois métodos é que um adiciona um critério principal ao projeto e o outro adiciona sub critérios para compor um critério.

Nos dois métodos da Figura 5.5, da classe *BasicExampleCtrl*, é possível perceber que algumas partes são similares. Código similar em métodos diferentes podem caraterizar um Código Duplicado e contribuir para uma Classe Extensa. A solução para esse *code smell* foi isolar a parte de manipulação de critério na classe *CriterionFormModal* no novo sistema, utilizando a técnica de Extrair Classe. Ao concluir a inserção ou edição de um critério, o *CriterionFormModa*l emite um evento com os dados do critério e então a atualização é efetuada no componente pai.

```
$scope.newItem = function (project_id) {
  var nodeData = $scope.data;
  var node = nodeData.push({
    id:null,
    sequence: nodeData.length + 1,
    title: (nodeData.length + 1),
    nodes: [],
    project_id: project_id,
    criterion_id:null,
  });
  $scope.nodeSelecionado = nodeData[node-1];
};

$scope.newSubItem = function (nodeData) {
  if(nodeData.nodes == null){
    nodeData.nodes = [];
  }
  var node = nodeData.nodes.push({
    id:null,
    sequence: nodeData.sequence * 10 + (nodeData.nodes.length+1),
    title: nodeData.title + '.' + (nodeData.nodes.length + 1),
    name: null,
    nodrop: true,
    nodes: [],
    project_id: nodeData.project_id,
    criterion_id: nodeData.id ,
  });
  $scope.nodeSelecionado = nodeData.nodes[node-1];
};
```

Figura 5.5: Code smell de BasicExampleCtrl.

O método *buildLevels* da Figura 5.6 é responsável por construir as tabelas de critérios, utilizado para exibir todos os critérios e subcritérios na página das taxas de contribuição. Esse método é de certa forma extenso e de difícil entendimento. O objetivo desse método é transformar a lista de critérios do módulo de servidor em uma listagem de critérios, agrupados por critério "pai".

O método pode ser categorizado como Método Longo, um dos *code smells* listados por Fowler, o que pode dificultar a manutenção do software [6]. Para resolver este *code smell*, o método foi reescrito de uma forma compacta no novo sistema, sem prejuízos para o funcionamento do software, conforme a Figura 5.8.

```
var buildLevels = function(criterions){
    var list_sequences = [];
    for(var i in criterions){
      if(criterions[i].criterion_id == null ){
        list_sequences.push({
          sequence :criterions[i].sequence,
          sequences: [],
        });
        $scope.listOfLevels.push({
          level :criterions[i],
          criterion: [],
        });
      }
    }
    for(var u in list_sequences){
      for(var v in criterions ){
        var one = String(criterions[v].sequence).charAt(0);
        if(list_sequences[u].sequence == one){
          list_sequences[u].sequences.push(criterions[v].sequence);
        }
      }
    }
    for(var u in list_sequences){
      for(var v in criterions ){
        var one = String(criterions[v].sequence).charAt(0);
        if(list_sequences[u].sequence == one & criterions[v].criterion_id != null ){
          checkLevelCriterion($scope.listOfLevels[u], criterions[v]);
          //$scope.listOfLevels[u].criterion.push(criterions[v]);
        }
      }
    }
};
```

Figura 5.6: Code smell de CriterionController.

```
var checkLevelCriterion = function(listCriterions,criterian){
    var number = criterian.sequence.toString().length;
    var titleLength = criterian.title.length;
    if(listCriterions.criterion.length == 0 ){
      listCriterions.criterion.push({
          step :number > 1 ? criterian.sequence.toString().substring(0,number-1) : number,
          titleGroup : criterian.title.substring(0,titleLength-1),
          criteria: [criterian],
        });
    }else{
      for(var i in listCriterions.criterion){
        if(listCriterions.criterion[i].step == criterian.sequence.toString().substring(0,number-1)
          listCriterions.criterion[i].criteria.push(criterian);
          return;
        }
      }
      listCriterions.criterion.push({
            step :number > 1 ? criterian.sequence.toString().substring(0,number-1) : number,
            titleGroup : criterian.title.substring(0,titleLength-1),
            criteria: [criterian],
          });

    }
};
```

Figura 5.7: Trecho chamado por buildLevels de CriterionController.

```
listToTree(list) {
    const map = [];
    const roots = [];
    let i;
    let node = {};
    for (i = 0; i < list.length; i += 1) {
        map[list[i].id] = i; // initialize the map
        list[i].children = []; // initialize the children
    }
    for (let x = 0; x < list.length; x += 1) {
        node = list[x];
        if (node.criterion_id !== null) {
            // if you have dangling branches check that map[node.parentId] exists
            list[map[node.criterion_id]].children.push(node);
        } else {
            roots.push(node);
        }
    }

    return roots;
}
```

Figura 5.8: Correção do Code smell de CriterionController.

Os métodos *saveContributionRateOrEffort* e *saveContributionRateOrEffortGeneral* da Figura 5.9 são pertencentes a classe *CriterionController*. A função desses dois métodos

é validar e salvar as porcentagens das taxas de contribuição de um critério ou subcritério. Ao examinar os métodos é possível verificar que a implementação é praticamente idêntica, caracterizando o *code smell* de Código Duplicado. O Código Duplicado pode dificultar bastante a manutenção do software, pois quando o código precisa ser modificado, é necessário lembrar de todos os lugares aonde o trecho está presente. No sistema novo, os dois Códigos Duplicados foram centralizados em um só lugar numa nova classe — *ContributionRateTable* — e o módulo de servidor aceita uma lista de critérios como argumento.

```
$scope.saveContributionRateOrEffort = function(contribution,type){

  if(type == 'rate'){
    var totalContribution = 0 ;

    for(var i in contribution.criteria){
      totalContribution = totalContribution + contribution.criteria[i].percent;
    }
    if(totalContribution > 100 || totalContribution < 100 ){
      confirm("Contribution rate should be 100% ")
      return;
    }
  }
  for(var i in contribution.criteria){
    loadingCenter("pageContent",true);
    //$scope.instance.project_id =1;

    $http.post("/criterions/store",contribution.criteria[i]).then(function (response) {

    }, function (response) {
    }).finally(function(){
      loadingCenter("pageContent",false);
    });
  }
  appInfo("Successfully save!");
};
```

```
$scope.saveContributionRateOrEffortGeneral = function(contribution,type){

  if(type == 'rate'){
    var totalContribution = 0 ;
    for(var i in contribution){
      totalContribution = totalContribution + contribution[i].level.percent;
    }
    if(totalContribution > 100 || totalContribution < 100 ){
      confirm("Contribution rate should be 100% ")
      return;
    }
  }

  for(var i in contribution){
    loadingCenter("pageContent",true);
    //$scope.instance.project_id =1;

    $http.post("/criterions/store",contribution[i].level).then(function (response) {


    }, function (response) {
    }).finally(function(){
      loadingCenter("pageContent",false);
    });
  }
  appInfo("Successfully save!");
};
```

Figura 5.9: Code smell de saveContributionRateOrEffort e saveContributionRateOrEffortGeneral no CriterionController.

Na Figura 5.10, temos a refatoração dos códigos da Figura 5.9. É possível notar que os métodos *saveContributionRateOrEffort* e *saveContributionRateOrEffortGeneral* foram subdivididos em três outros métodos menores no novo sistema. Os programas que tem um tempo de vida mais longos são aqueles que tem métodos curtos [6]. Os métodos foram assim divididos para favorecer a separação de contextos: *isValid* valida o formulário, *submitForm* prepara o formulário para envio e *submit* envia o resultado de *submitForm* para o módulo de servidor.

```
submit () {
  if (this.isValid()) {
    this.done = false
    axios.patch(`/projects/${this.projectId}/criteria`, this.submitForm)
      .then(() => {
        this.setToast({ message: 'Taxas de Contribuição Atualizadas' })
      })
      .finally(() => { this.done = true })
  } else {
    this.warningModalIsOpened = true
  }
}

private isValid () {
  const total = this.form.reduce((acc, criterion) => {
    return acc + criterion.percent
  }, 0)

  return total === 100
}

private get submitForm () {
  return this.form.map(part => {
    return { id: part.id, percent: part.percent }
  })
}
```

Figura 5.10: Correção do Code smell de saveContributionRateOrEffort e saveContributionRateOrEffortGeneral no CriterionController.

O método *findOrderProject* da Figura 5.11 é utilizado para buscar as escalas e critérios. O método também é responsável por calcular os valores de cada opção da escala de um critério e esse valor calculado é utilizado para preencher as tabelas na tela durante a escolha das modas. Por ser um método que realiza múltiplas tarefas, pode ser considerado um Método Longo. Para resolver esse problema, parte da lógica foi transferida para o módulo do servidor e o método foi dividido em métodos menores. No módulo de cliente, a classe *ProjectCriteriaModes* é responsável por buscar os dados e a classe *ContributionRateTable* faz a manipulação dos dados recebidos.

```
$scope.findOrderProject = function(id){
  $scope.findAnswer(id);

  loadingCenter("pageContent",true);

  $http.get('/criterions/find_order_project/'+id).then(function (response) {

      $scope.data = response.data;

      for(var i in $scope.data)
      {
        $scope.data[i].scales = [];
        for(var a in $scope.answer){
          $scope.data[i].scales.push(
            {
              id: null,
              n_id: $scope.count,
              answer : $scope.answer[a].answer,
              value:null,
              neutral: $scope.answer[a].neutral,
              good : $scope.answer[a].good,
              median : 0,
              criterion_id: $scope.data[i].id,
              option_answer_id: $scope.answer[a].id,
```

Figura 5.11: Code smell de ScaleController.

Na classe *ResultController*, o método *buildLevels* é idêntico ao método *buildLevels* da classe *CriterionController*, mostrado na Figura 5.6 e tem a mesma funcionalidade. O objetivo desse método em *ResultController* é o mesmo de *CriterionController*: transformar a lista de critérios do módulo de servidor em uma listagem de critérios, agrupados por critério "pai". Esse code smell de Código Duplicado foi resolvido com a mesma solução apresentada na Figura 5.8. Vale ressaltar que o novo método *listToTree* foi extraído para uma nova classe, conforme a técnica de Extrair Classe, possibilitando o uso do método em *ResultController* e *CriterionController* sem duplicação de código.

```
var buildLevels = function(criterions){
    var list_sequences = [];
    for(var i in criterions){
        if(criterions[i].criterion_id == null ){
          list_sequences.push({
            sequence :criterions[i].sequence,
            sequences: [],
          });

          $scope.listOfLevels.push({
            level :criterions[i],
            criterion: [],
          });
        }
    }
    for(var u in list_sequences){
        for(var v in criterions ){
          var one = String(criterions[v].sequence).charAt(0);
          if(list_sequences[u].sequence == one){
            list_sequences[u].sequences.push(criterions[v].sequence);

          }
        }
    }
    for(var u in list_sequences){
        for(var v in criterions ){
          var one = String(criterions[v].sequence).charAt(0);
          if(list_sequences[u].sequence == one & criterions[v].criterion_id != null ){
```

Figura 5.12: Code smell de ResultController.

O método *calculeteFinalResult* de *ResultController* na Figura 5.13 é um método que faz o cálculo dos resultados. O método utiliza também o cálculo de escalas para calcular o resultado. Além disso, o método também realiza o cálculo dos valores que populam o gráfico.

Esse método pode ser categorizado como um Método Longo por ter múltiplas responsabilidades. Para resolver esse problema, o cálculo foi dividido em vários métodos menores e extraído para uma nova classe, aumentando a legibilidade do código. Foram utilizadas as técnicas Extrair Método e Extrair Classe para resolver esse *code smell*.

```
var calculeteFinalResult = function()
{
    var mainCriterian = [];
    $scope.finalResult.scales = [];
    for(var a in $scope.optionAnswer){
        $scope.finalResult.scales.push(
        {
            id: null,
            n_id: $scope.count,
            answer : $scope.optionAnswer[a].answer,
            value:null,
            neutral: $scope.optionAnswer[a].neutral,
            good : $scope.optionAnswer[a].good,
            median : 0,
            option_answer_id: $scope.optionAnswer[a].id,
        });
    }

    for(var i in $scope.dataTree)
    {
        if($scope.dataTree[i].criterion_id == null)
        {
            mainCriterian.push($scope.dataTree[i]);
        }
    }
```

Figura 5.13: Code smell de ResultController.

```
async getFinalResult(req, res) {
    const project_id = req.params.id;

    // verify if id is valid
    if (Number.isNaN(project_id)) {
        return res.status(400).json({ error: { mensagem: 'Project id Inválido!' } });
    }

    const listCriteria = await Criteria.findAll({
        where: { project_id },
        raw: true,
    });

    // find project
    const project = await Project.findByPk(project_id);

    const options = await OptionAnswer.findAll({
        where: { scale_id: project.scale_id },
        attributes: ['id', 'answer', 'neutral', 'good'],
        order: ['id'],
        raw: true,
    });

    const result = await util.buildFinalResult(listCriteria, options);

    return res.status(200).json(result);
}
```

Figura 5.14: Correção do Code smell de ResultController.

5.2 Mitigação de *code smells* no módulo do servidor

O modulo do servidor é responsável por desenvolver a camada interna do sistema, realizar a persistência dos dados, conexão com o banco de dados, implementar as regras de negócio e processar os dados recebidos. A primeira versão do sistema MyMCDA-C foi projetada e desenvolvida em PHP. Em uma análise realizada, foram detectados e corrigidos os *code smells* encontrados em controladoras da primeira versão. Nos parágrafos a seguir, os problemas e a solução são listados e descritos.

Na Figura 5.15 o método *create()* da controladora *UserController* é utilizado para realizar a persistência do cadastro de usuário na tabela do banco de dados de usuários. Porém, a função *User::Create* recebe os parâmetros de usuário diretamente na controladora, podendo causar a repetição de código caso seja necessário criar um usuário em outra parte do sistema diferente de registro. Além disso, outro problema pode ser gerado se um novo campo for adicionado a tabela de Usuários, assim sendo necessário buscar todas as referências de *User::create* pelo código para incluir o novo campo.

```php
/**
 * Create a new user instance after a valid registration.
 *
 * @param  array  $data
 * @return \App\User
 */
protected function create(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'university' => $data['university'],
        'country' => $data['country'],
        'password' => bcrypt($data['password']),
    ]);
}
```

Figura 5.15: Code smell no arquivo UserController função create() - versão 1.0 MyMCDA-C.

O problema mencionado acima é um exemplo de código duplicado [6]. De modo a solucionar o problema, na nova versão 2.0 do software MyMCDA-C foi inserida uma biblioteca para o mapeamento do banco de dados relacional chamada *Sequelize* na versão 6.4. Esta biblioteca é responsável pelo Mapeamento objeto-relacional ou *Object-relational mapping (ORM)* que visa o mapeamento e gerenciamento do banco de dados.

Utilizando a função *create()* do *Sequelize* que se assemelha a função utilizada na versão anterior do software, foi possível mitigar o problema relatado acima. A função recebe em um só parâmetro os dados do usuário, e esta é uma estratégia conhecida como extração do método [6], utilizada para contornar este problema. A Figura 5.16 mostra o trecho de código com a resolução do problema.

```
const { id, name, email } = await User.create(userData);
```

Figura 5.16: Correção na falha da função para criar usuário - versão 2.0 MyMCDA-C.

Na Figura 5.17 são implementados duas funções *createOrGetUser()* e *createOrGetUserGoogle()* que registram os dados de usuários que utilizam as redes sociais *Facebook* e

Google respectivamente para realizar o cadastro e *login*. Porém, a forma como as funções foram implementadas faz com que elas estejam sujeitas repetição de código. Percebe-se que a cada nova rede social inserida na controladora, é necessário adicionar uma nova função para a persistência dos dados.

```php
public function createOrGetUserGoogle(ProviderUser $providerUser)
{
    $account = SocialAccount::whereProvider('google')
        ->whereProviderUserId($providerUser->getId())
        ->first();

    if ($account) {
        return $account->user;
    } else {

        $account = new SocialAccount([
            'provider_user_id' => $providerUser->getId(),
            'provider' => 'google'
        ]);

        $user = User::whereEmail($providerUser->getEmail())->first();

        if (!$user) {

            $user = User::create([
                'email' => $providerUser->getEmail(),
                'name' => $providerUser->getName(),
                'password' => '',
                'country' => ' ',
                'university' => ' ',
            ]);
        }

        $account->user()->associate($user);
        $account->save();

        return $user;

    }

}
```

```php
public function createOrGetUser(ProviderUser $providerUser)
{
    $account = SocialAccount::whereProvider('facebook')
        ->whereProviderUserId($providerUser->getId())
        ->first();

    if ($account) {
        return $account->user;
    } else {

        $account = new SocialAccount([
            'provider_user_id' => $providerUser->getId(),
            'provider' => 'facebook'
        ]);

        $user = User::whereEmail($providerUser->getEmail())->first();

        if (!$user) {

            $user = User::create([
                'email' => $providerUser->getEmail(),
                'name' => $providerUser->getName(),
                'password' => '',
                'country' => ' ',
                'university' => ' ',
            ]);
        }

        $account->user()->associate($user);
        $account->save();

        return $user;

    }
}
```

Figura 5.17: Code smell na classe SocialAccountService na função para registrar usuários de redes sociais - versão 1.0 MyMCDA-C.

O problema relatado pode ser descrito como um código duplicado [6]. Com intuito de contornar o problema identificado foi implementada uma nova função chamada *createSocialUser()* apresentada na Figura 5.18, que agrupa o comportamento das duas funções em apenas uma e cujo nome explica sua finalidade. Aplica-se assim a estratégia de extração de método [6], tornando o código limpo e reutilizável.

```
async createSocialUser(provider_id, userEmail, userName, provider) {
    const userSocial = await SocialAccount.findOne({
        where: { provider_id },
        include: {
            model: User,
            attributes: ['id', 'name', 'email'],
        },
    });

    let id = null;
    let name = null;
    let email = null;
    let userData = null;

    if (!userSocial) {
        userData = await User.create({ name: userName, email: userEmail });

        if (!userData) {
            return {
                status: 400,
                response: { error: { mensagem: 'Error ao criar Usuário!' } },
            };
        }

        await SocialAccount.create({ user_id: userData.id, provider, provider_id });
    }

    id = userSocial ? userSocial.User.id : userData.id;
    name = userSocial ? userSocial.User.name : userData.name;
    email = userSocial ? userSocial.User.email : userData.email;

    return {
        status: 200,
        response: {
            user: {
                id,
                name,
                email,
            },
            token: jwt.sign({ id }, AuthConfig.secret, {
                expiresIn: AuthConfig.expiresIn,
            }),
        },
    };
}
```

Figura 5.18: Correção da função para cadastro de usuários de redes sociais - versão 2.0 MyMCDA-C.

Um dos problemas detectados por meio da análise arquitetural foi o *code smell speculative generality* quando um método realiza mais de uma função a que é destinado. Esse problema foi detectado em três controladoras, em sua maioria os métodos foram desenvolvidos com intuito de criar e atualizar registros em mesmo lugar, porém o nome das funções não deixava este intuito claro. A controladora *ProjectController* da primeira versão continha uma função *store()* com este problema, como apresentado na Figura 5.19. Além disso, é possível notar que a função não possui nenhum tratamento de erro ou exceções. Visando remover estes problemas dois novos métodos foram adicionados à nova controladora de projetos, o *store()* e *update()* que contam agora com validação dos dados e tratamento de exceções, apresentado na Figura 5.20.

```php
public function store(ProjectsFormRequest $request){

    $project = Project::FindOrNew($request->id);
    $project->fill($request->all());
    $project->user_id = $request->user_id;
    $project->scale_id = $request->scale_id;
    $project->save();

    return $project;
}
```

Figura 5.19: Code smell na classe ProjectController na função para cadastro de projeto - versão 1.0 MyMCDA-C.

```javascript
async store(req, res) {
    // Fields Validation
    const schema = Yup.object().shape({
        project_goal: Yup.string().required(),
        performance: Yup.string().required(),
        steps: Yup.number().required(),
        scale_id: Yup.number().required(),
    });

    try {
        await schema.validate(req.body, { abortEarly: false });
    } catch (error) {
        return res.status(400).json(error);
    }

    const project = req.body;
    project.user_id = req.userId;

    // create project
    try {
        const { id, project_goal } = await Project.create(project);

        return res.status(200).json({
            id,
            project_goal,
        });
    } catch (error) {
        return res.status(400).json({ error: { mensagem: 'Erro! Falha ao salvar projeto.' } });
    }
}
```

```javascript
async update(req, res) {
    // Fields Validation
    const schema = Yup.object().shape({
        id: Yup.number().required(),
        project_goal: Yup.string(),
        performance: Yup.string(),
        steps: Yup.number(),
        scale_id: Yup.number(),
    });

    try {
        await schema.validate(req.body, { abortEarly: false });
    } catch (error) {
        return res.status(400).json(error);
    }

    // update project
    try {
        const project = req.body;

        // find project
        const projects = await Project.findByPk(project.id);

        // verify if projects is empty
        if (!projects) {
            return res.status(401).json({ error: { mensagem: 'Projeto Inválido!' } });
        }

        // update project
        const { id, project_goal } = await projects.update(project);

        return res.status(200).json({
            id,
            project_goal,
        });
    } catch (error) {
        return res.status(400).json({ error: { mensagem: 'Erro! Falha ao atualizar projeto.' } });
    }
}
```

Figura 5.20: Correção da função para cadastro e atualização de projeto - versão 2.0 MyMCDA-C.

Outra controladora que continha o mesmo problema relatado acima de *speculative generality* é a *ScaleController*, dois métodos chamados *store()* e *saveOptionsAnswer()* realizavam a criação e a atualização dos dados, conforme apresentado na Figura 5.21. De modo a corrigir este problema e adicionar a estrutura de validação e tratamento de exceções, duas funções foram desenvolvidas na nova controladora, uma para realizar a tarefa de persistência e outra para atualizar as informações. As funções *store()* e *update()* presentes nas Figuras 5.22 e 5.23 também absorveram a responsabilidade da função da primeira versão *saveOptionsAnswer()*, pois tratam-se de áreas correlatas no software.

```php
public function store(Request $request){

    $scale = Scale::FindOrNew($request->id);
    $scale->fill($request->all());
    $scale->user_id = $request->user_id;
    $scale->save();

    $this->saveOptionsAnswer($scale,$request->option_answer);


    return $scale;
}
```

Figura 5.21: Code smell na classe ScaleController nas funções de cadastro de escala e opções de resposta - versão 1.0 MyMCDA-C.

```javascript
async store(req, res) {
    // Fields Validation
    const schema = Yup.array().of(
        Yup.object().shape({
            answer: Yup.string().required(),
            good: Yup.boolean().required(),
            neutral: Yup.boolean().required(),
        }),
    );

    if (!(await schema.isValid(req.body))) {
        return res.status(400).json({ error: { mensagem: 'Dados Inválidos!' } });
    }

    const optionsAnswers = req.body;
    // create description
    const reducer = (description, arr, index) => (index === 0 ? arr.answer : `${description} - ${arr.answer}`);
    const description = optionsAnswers.reduce(reducer, '');

    try {
        // persist scale
        const { id } = await Scale.create({
            user_id: req.userId,
            description,
        });

        // insert id_scale on options
        optionsAnswers.forEach((element, index) => {
            optionsAnswers[index].scale_id = id;
        });

        // Persist options
        return await OptionAnswer.bulkCreate(optionsAnswers).then(() => Scale.findOne({
            where: { id },
            attributes: ['id', 'description'],
            include: {
                model: OptionAnswer,
                as: 'optionAnswers',
                attributes: ['id', 'answer', 'neutral', 'good'],
            },
        })).then((options) => res.status(200).json(options));
    } catch (error) {
        return res.status(400).json({ error: { mensagem: 'Erro ao criar Scale!' } });
    }
}
```

Figura 5.22: Correção da função para cadastro de escala e opções de resposta - versão 2.0 MyMCDA-C.

Figura 5.23: Correção da função para atualização dos dados de escala e opções de resposta - versão 2.0 MyMCDA-C.

Observou-se o mesmo comportamento na classe *CriteriaController* da primeira versão, três funções desempenhavam o mesmo papel de criar e atualizar informações. Os métodos *store()*, *saveSort()* e *saveScaleResult()* apresentados na Figura 5.24 estavam com este problema, a função *saveSort()* não foi desenvolvida na segunda versão do software, pois se tornou obsoleta. O método *saveScaleResult()* teve sua responsabilidade transferida para classe *ScaleResultController*. Por fim, apenas a função *store()* foi desenvolvida na controladora, sendo removida sua atribuição para atualizar em uma nova função *update()*, conforme apresentado na Figura 5.25.

```php
public function store(Request $request){
    $criterion = Criterion::FindOrNew($request->id);
    $criterion->fill($request->all());
    $criterion->criterion_id = $request->criterion_id;
    $criterion->project_id = $request->project_id;

    $criterion->save();


    return $criterion;
}

public function saveSort(Request $request){
    foreach($request->all() as $criteriaArr)
    {
        $criteria = SortLastCriteria::FirstOrNew(['criterion_id' =>$criteriaArr['id']]);
        $criteria->criterion_id = $criteriaArr['id'];
        $criteria->order = $criteriaArr['order'];
        $criteria->project_id = $criteriaArr['project_id'];

        $criteria->save();

    }
    return $request;

}

public function saveScaleResult(Request $request){
    foreach($request->all() as $requestArr)
    {
        foreach($requestArr['scales'] as $scaleArr)
        {
            $scale = ScaleResult::FirstOrNew(['id' =>$scaleArr['id']]);
            $scale->value = $scaleArr['value'];
            $scale->median = $scaleArr['median'];
            $scale->criterion_id = $scaleArr['criterion_id'];
            $scale->option_answer_id = $scaleArr['option_answer_id'];

            $scale->save();
        }

    }
    return $request;
}
```

Figura 5.24: Code smell na classe CriteriaController - versão 1.0 MyMCDA-C.

```javascript
async store(req, res) {
    const schema = Yup.object().shape({
        name: Yup.string().required(),
        title: Yup.string(),
        percent: Yup.number(),
        criterion_id: Yup.number(),
    });

    try {
        await schema.validate(req.body, { abortEarly: false });
    } catch (error) {
        return res.status(400).json(error);
    }

    const criterion = req.body;
    criterion.project_id = req.params.id;

    // create criterion
    try {
        const {
            id, name, title, percent, criterion_id, project_id,
        } = await Criteria.create(criterion);

        return res.status(200).json({
            id,
            name,
            title,
            percent,
            criterion_id,
            project_id,
            children: [],
        });
    } catch (error) {
        return res.status(400).json({ error: { mensagem: 'Erro! Falha ao salvar critério.' } });
    }
}
```

```javascript
async update(req, res) {
    // Fields Validation
    const schema = Yup.object().shape({
        id: Yup.number().required(),
        name: Yup.string().required(),
        title: Yup.string(),
        percent: Yup.string(),
        order: Yup.number(),
    });

    try {
        await schema.validate(req.body, { abortEarly: false });
    } catch (error) {
        return res.status(400).json(error);
    }

    const project_id = req.params.id;
    const id_criterion = req.params.criterion_id;
    const criterion = req.body;

    // verify if id is valid
    if (Number.isNaN(project_id) || Number.isNaN(id_criterion)) {
        return res.status(400).json({ error: { mensagem: 'Ids Inválidos!' } });
    }

    try {
        const criterionResult = await Criteria.findByPk(id_criterion);
        const {
            id, name, title, percent,
        } = await criterionResult.update(criterion);

        return res.status(200).json({
            id, name, title, percent,
        });
    } catch (error) {
        return res.status(400).json({ error: { mensagem: error } });
    }
}
```

Figura 5.25: Correção das funções para criar e atualizar os dados de critério - versão 2.0 MyMCDA-C.

5.3 Melhorias adicionadas no MyMCDA-C versão 3.0

Para incrementar o sistema, novas funcionalidades foram desenvolvidas com intuito de melhorar a experiência geral de uso do sistema. A primeira versão do sistema possuía diversos problemas de implementação que dificultavam a navegação pela área de projeto. As melhorias para alcançar esse objetivos são explicadas com mais detalhes ao longo desta seção.

5.3.1 Duplicar Projeto

Na tela principal do sistema, na lista de projetos do usuário, foi adicionada uma nova funcionalidade para duplicar projeto. Com essa funcionalidade é possível duplicar o projeto com todas as variáveis, tais como: critérios, porcentagens, ordenação, modas e os resultados obtidos. Essa funcionalidade foi requisitada por alguns usuários com objetivo de realizar análises em um projeto em cenários diferentes.

5.3.2 Visualização da Árvore de Critérios

Como melhoria para o sistema, foi requisitado a implementação de uma nova forma de visualizar a Árvore de Critérios. Além de poder visualizar, o usuário também pode ampliar, reduzir e mover a imagem. Se o usuário desejar esconder os critérios filhos de um critério, basta clicar no critério e a imagem será redesenhada conforme solicitado. Abaixo, na Figura 5.26, é possível ver um exemplo de como essa funcionalidade foi implementada no sistema.

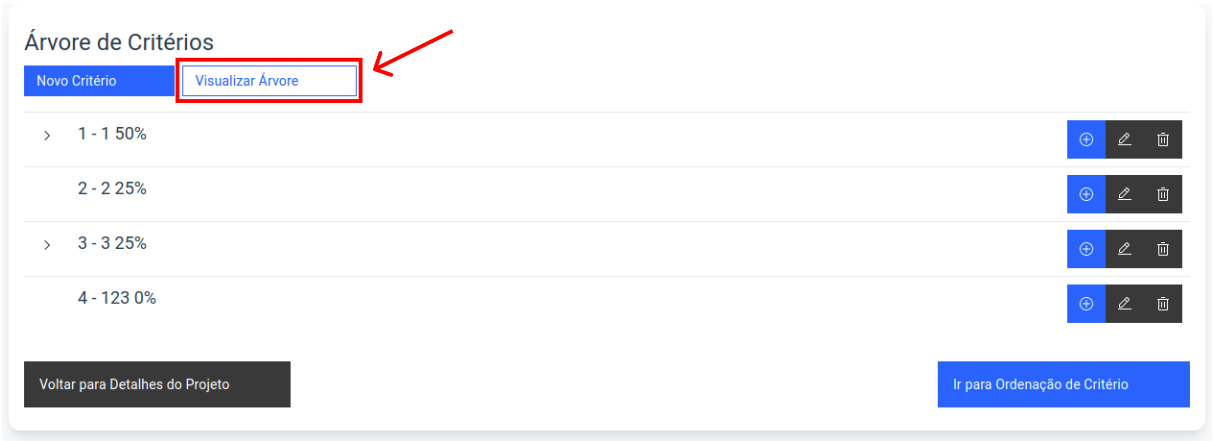

Figura 5.26: Botão para Visualização da Árvore de Critérios.

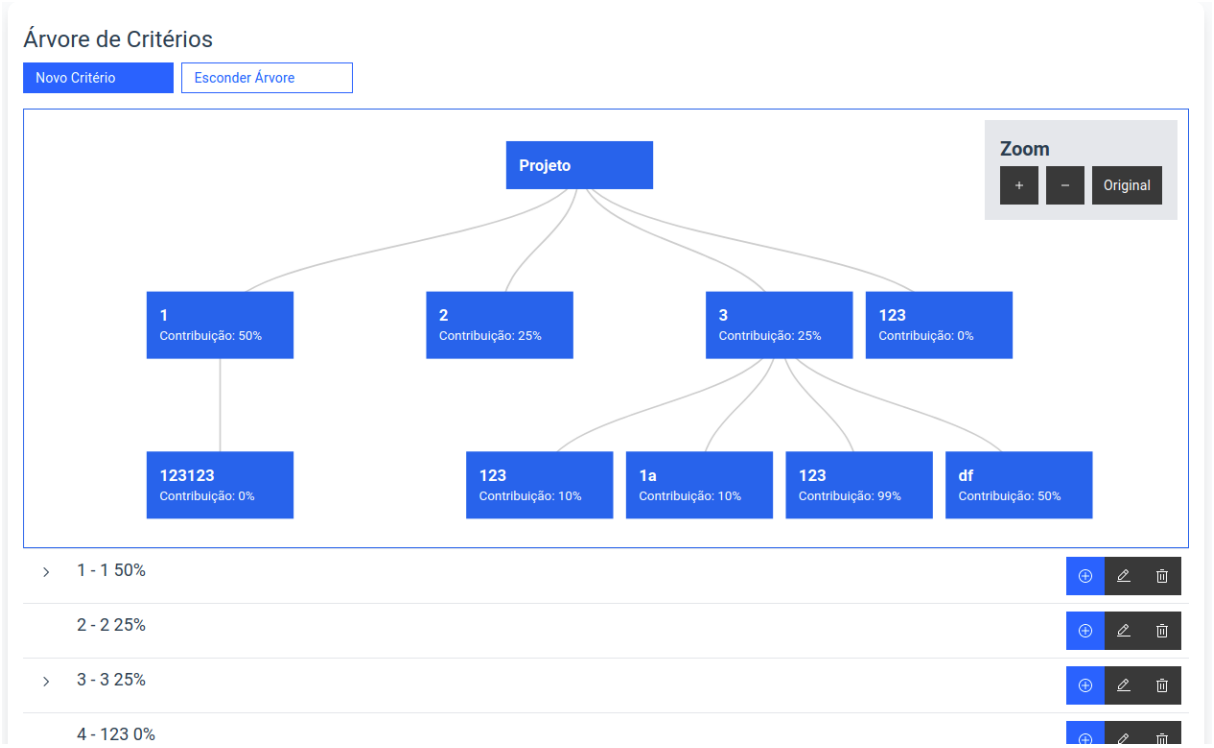

Figura 5.27: Visualização da Árvore de Critérios.

5.3.3 Barra de Navegação do Projeto

A navegação dentro de um projeto era um dos problemas que o sistema possuía. Para observar os resultados de um projeto já finalizado, eram necessários ao menos 5 cliques (com rolagem de página em alguns casos) a partir da página inicial. Com a nova versão, é possível chegar na página de resultado com apenas 2 cliques. Em todas as etapas do projeto é possível navegar para outra etapa pela barra de navegação da Figura abaixo.

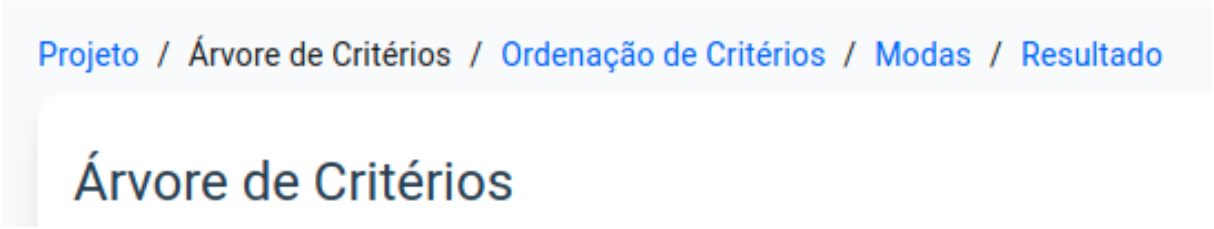

Figura 5.28: Barra de Navegação do Projeto.

5.3.4 Redução do Número de Telas

Outro problema relatado era que a tela de Taxas de Contribuição era desnecessária e que essas taxas poderiam ser preenchidas no momento da criação dos critérios. Isso facilitaria o processo de preenchimento do projeto. Atendendo ao pedido, foi adicionado o campo para a taxa de contribuição na criação e edição de critério.

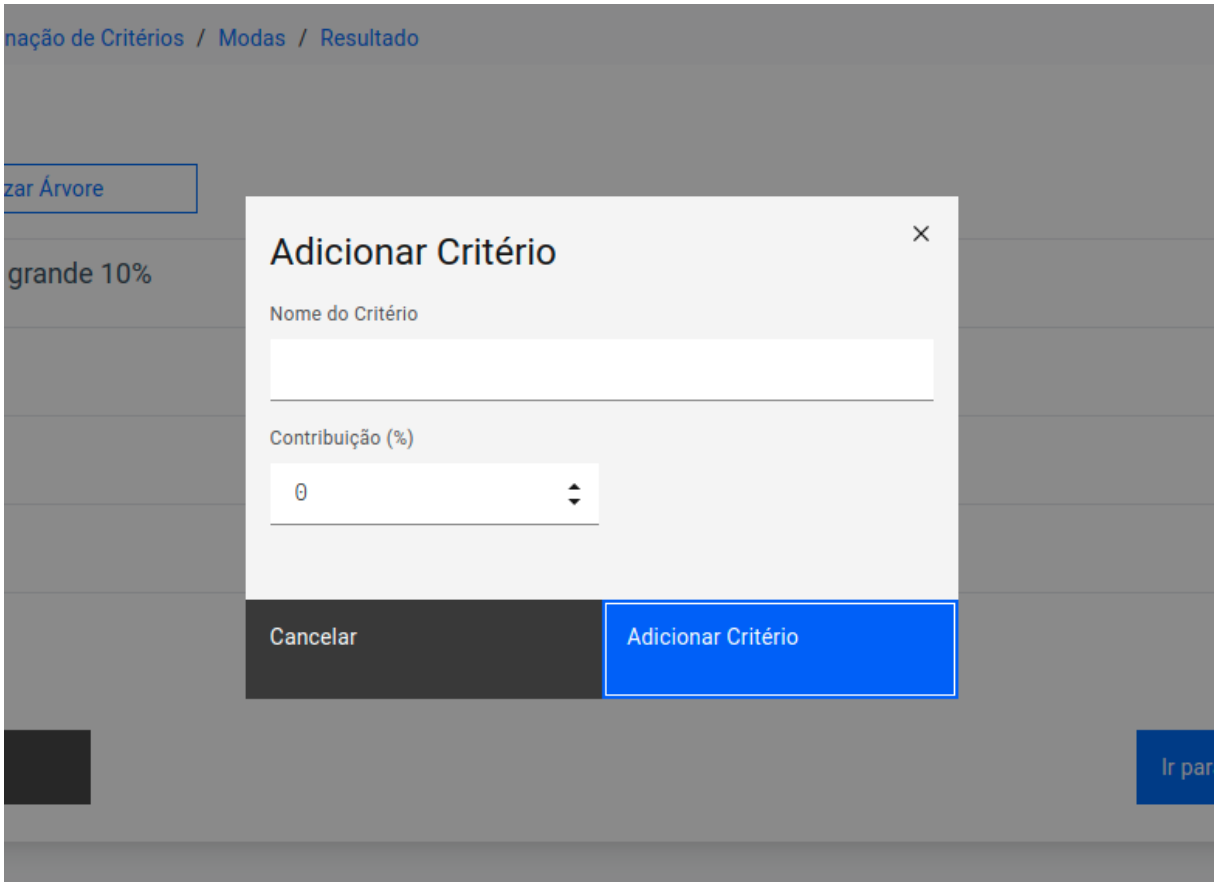

Figura 5.29: Tela de criação de critério no sistema novo, com o campo de taxa de contribuição.

5.3.5 Ordenação de Critérios

No sistema original, a ordenação de critérios era feita utilizando a função “clica e arrasta”, porém muitos usuários se queixavam de terem dificuldades, já que a lista de critérios pode ser extensa. O problema era mais evidente no momento de clicar, segurar, rolar a página e então soltar. Na nova versão, o usuário pode enumerar as posições dos critérios e visualizar os critérios em ordem ao salvar ou ao clicar em “Exibir Ordenado”.

Figura 5.30: Tela de Ordenação de Critérios.

5.3.6 Exportar Gráficos e Tabelas de Resultado

Observando os trabalhos que usaram o sistema original como ferramenta, foi verificado que grande parte desses trabalhos utilizaram fotos dos gráficos e das tabelas do resultado. Com intuito de facilitar esse processo, foram criados botões para exportar os dados da tabela e os gráficos. Além disso, a aba de "folhas" na página de resultados não possuía gráficos, e isso foi corrigido na nova versão do sistema.

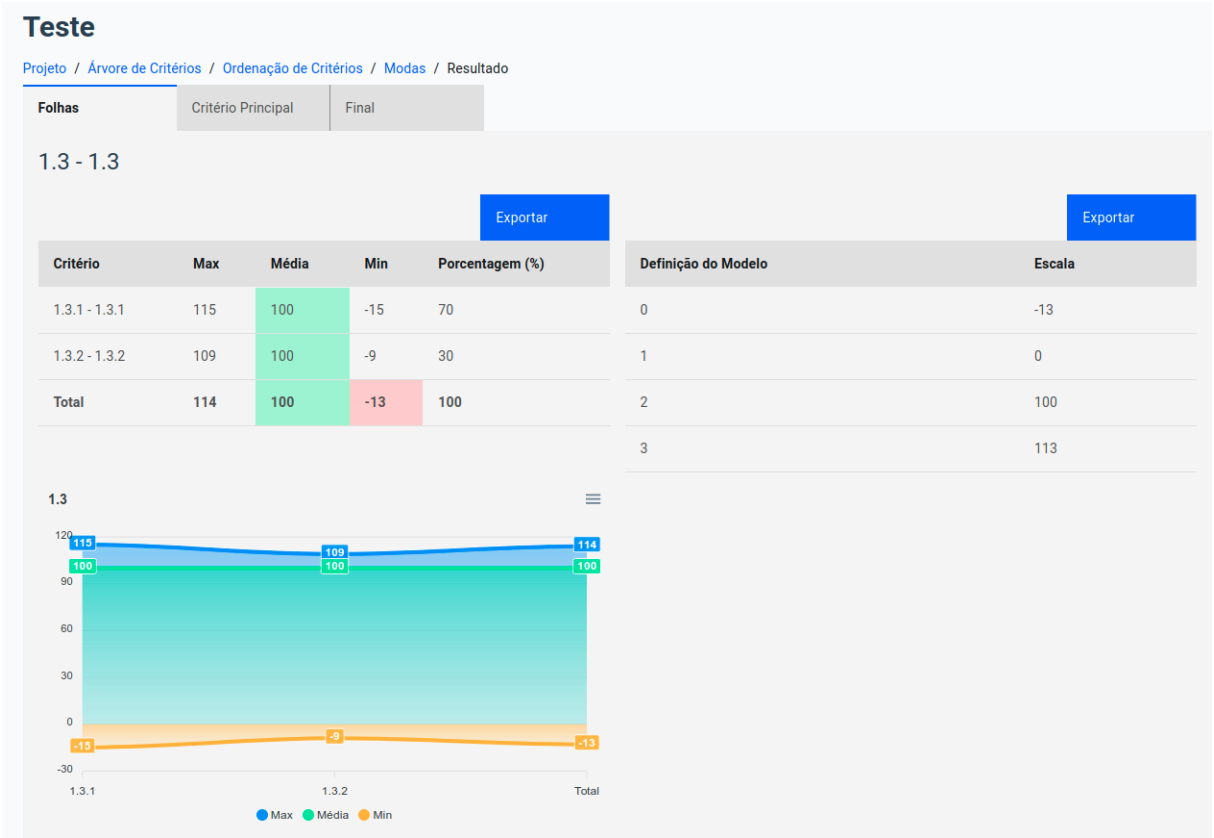

Figura 5.31: Tabela e gráficos de resultados com funcionalidade de Exportar.

Capítulo 6

Resultados

Na primeira versão do sistema MyMCDA-C (versão antiga), muitos usuários reclamavam sobre a dificuldade de navegar pelo projeto. Para ir da tela de *login* até a tela de resultados de um projeto eram necessários, no melhor caso, 6 cliques. O novo sistema foi desenhado com a intenção de fornecer uma melhor experiência de usabilidade e facilidade na navegação. Na Figura 6.1, é possível observar o número de cliques para navegar da página da coluna até a página da linha no novo sistema. Algumas páginas não estão listadas nas linhas ou colunas da tabela, isso ocorre porque ou essas páginas redirecionam para links externos (*google-login*, por exemplo) ou só podem ser acessadas de maneira externa (*recover-password/new-password/:token*, por exemplo).

Para realizar a coleta da quantidade mínima de cliques necessários para realizar a navegação, foi considerado o caminho mínimo de uma tela até a outra via interface, independente de ser o fluxo esperado ou não. Para exemplificar, se uma tela pode ser alcançada por meio do menu superior ao invés da navegação interna, e é o menor caminho, esse caminho foi realizado. Foram desconsideradas quaisquer interações com campos de formulários durante os testes, como, por exemplo, os cliques para digitar o email e senha durante o acesso ao sistema.

	/login	/register	/recover-password	/recover-password/new-password/:token	/projects	/projects/create	/projects/:id/edit	/projects/:id/results	/projects/:id/criteria	/projects/:id/criteria/order	/projects/:id/modes	/account
/login	-	1	1	2	1	1	1	1	1	1	1	1
/register	1	-	2	3	2	2	2	2	2	2	2	2
/recover-password	1	2	-	3	2	2	2	2	2	2	2	2
/google-login	1	2	2	3	2	2	2	2	2	2	2	2
/facebook-login	1	2	2	3	2	2	2	2	2	2	2	2
/projects	1	1	2	3	-	1	1	1	1	1	1	1
/projects/create	2	2	3	4	1	-	2	2	2	2	2	2
/projects/:id/edit	2	2	3	4	1	2	-	1	1	1	1	2
/projects/:id/results	3	3	4	5	2	3	2	-	1	1	1	3
/projects/:id/criteria	2	2	3	4	1	2	1	1	-	1	1	2
/projects/:id/criteria/order	3	3	4	5	2	3	2	1	1	-	1	3
/projects/:id/modes	3	3	4	5	2	3	2	1	1	1	-	3
/account	2	2	3	4	1	1	1	1	1	1	1	-

Figura 6.1: Quantidade mínima de cliques necessários para ir da página do cabeçalho para a página da lateral da tabela no novo sistema.

Na Figura 6.2, foi feita a coleta dos dados de cliques mínimos para realizar a navegação, da mesma forma conduzida no novo sistema. É possível notar que em alguns links, existe um link na cor verde-escuro. Esse link em verde-escuro representa o link equivalente no sistema novo. Alguns links também estão marcados com um ou dois asteriscos, a motivação para esta notação é explicada nas observações, localizadas na parte inferior da figura.

	/login	/register	/recover-password	/recover-password/new-password/:token **	/projects	/projects/create	/projects/:id/edit	/projects/:id/criterio (/projects/:id/criteria)	/projects/:id/criterio/contribution_rate *	/projects/:id/order_criterio (/projects/:id/criteria/order)	/projects/:id/criterio/median_scale (/projects/:id/modes)	/projects/:id/criterio/result (/projects/:id/results)	/account
/login	-	1	1	-	3	3	3	3	3	3	3	3	3
/register	1	-	1	-	4	4	4	4	4	4	4	4	4
/password/reset (/recover-password)	1	2	-	-	4	4	4	4	4	4	4	4	4
/redirectGoogle (/google-login)	1	2	2	-	4	4	4	4	4	4	4	4	4
/redirect (/facebook-login)	1	2	2	-	4	4	4	4	4	4	4	4	4
/projects	1	1	2	-	-	1	1	1	1	1	1	1	1
/projects/create	2	2	3	-	1	-	2	2	2	2	2	2	2
/projects/edit/:id (/projects/:id/edit)	2	2	3	-	1	2	-	2	2	2	2	2	2
/projects/:id/criterio (/projects/:id/criteria)	2	2	3	-	1	2	2	-	1	2	2	2	2
/projects/:id/criterio/contribution_rate *	3	3	4	-	2	3	3	1	-	1	2	2	3
/projects/:id/order_criterio (/projects/:id/criteria/order)	4	4	5	-	3	4	4	2	1	-	1	3	4
/projects/:id/criterio/median_scale (/projects/:id/modes)	5	5	6	-	4	5	5	3	2	1	-	4	5
/projects/:id/criterio/result (/projects/:id/results)	6	6	7	-	5	6	6	4	3	2	1	-	6
/account	3	3	3	-	2	2	2	2	2	2	2	2	-

Obs:
* Tela integrada ao **/projects/:id/criteria** no novo sistema
** Tela impossível de se alcançar devido a falha do sistema

Figura 6.2: Quantidade mínima de cliques necessários para ir da página do cabeçalho para a página da lateral da tabela no sistema antigo.

Ao avaliar o código do sistema 1.0, verificou-se que o código era de difícil entendimento e manutenção. Para tentar entender a causa do problema e melhorar a qualidade do código, foi feito levantamento dos principais *code smells* do sistema. Na tabela 6.1 é possível notar e comprovar que o *code smell* mais comum que é o Código Duplicado[6].

Tabela 6.1: Número dos principais *code smells* encontrados em cada classe agrupados por tipo.

Classe	Classe Extensa	Código Duplicado	Método Longo
ProjectController.js	1	3	
BasicExampleCtrl.js	1	3	1
CriterionController.js		5	
ScaleController.js	1		1
ResultController.js	1	5	1
CriterionController.php	1		
SocialAccountService.php		1	
Total	**5**	**17**	**3**

A tabela 6.1, o cabeçalho representa o *code smell* encontrado. Na esquerda da tabela é apresentada a classe do sistema antigo, a extensão *.js* ou *.php*, neste caso, significa se a classe está no módulo de cliente ou no módulo de servidor, respectivamente. O cruzamento entre linha e coluna apresenta o número de *code smells* daquele tipo encontrado na classe indicada. Se o cruzamento de linha e coluna está vazio, quer dizer que nenhum *code smell* daquele tipo foi encontrado naquela classe. No caso de Código Duplicado dentro da mesma classe, os métodos envolvidos foram contados apenas uma vez.

6.1 Número de funções por Controladora

Com intuito de comparar o avanço na construção do software no módulo do servidor, foi realizado um estudo nas controladoras responsáveis por fornecer os dados para o módulo do cliente. Visando a melhoria e o aperfeiçoamento do software, algumas controladoras da versão 1.0 foram removidas, tornando assim o software mais limpo e fácil para manutenção. Os dados foram coletados de modo a apurar a quantidade de funções adicionadas e/ou removidas da versão 1.0, implementada na linguagem *PHP* para versão 2.0 desenvolvida em NodeJs.

Tabela 6.2: Comparativo da quantidade de funções por controladora.

Controladoras	**Versão 1.0**	**Versão 2.0**
CriterionController.php / .js	15	7
ProjectController.php / .js	8	5
ScaleController.php / .js	5	5
OptionAnswerController.php / .js	3	1
SessionController.js	X	8
UserController.php / .js	2	3
ForgotPasswordController.php	1	X
ResetPasswordController.php	1	X
LoginController.php	1	X
SocialAuthController.php	8	X
RegisterController.php	1	X
AccountController.php	2	X
HomeController.php	2	X
ScaleResultController.js	X	3
Total	**49**	**32**

A primeira versão continha 15 controladoras, já a nova versão contém apenas sete. O número total de funções passou de 49 para 23, ressaltando que algumas das funções para o fornecimento das telas foram realizadas pelo módulo de cliente desenvolvido em Vue.js, contribuindo para a redução de métodos e de controladoras.

A tabela 6.2 apresenta os dados coletados em relação a versões do software e o comparativo entre as funções desenvolvidas em cada controladora. A primeira coluna apresenta os nomes das controladoras bem como as extensões de cada versão, sendo *PHP* e *JavaScript(JS)* respectivamente. A segunda e terceira colunas apresentam os valores das quantidades de funções por versão. As linhas em que a controladora estiver marcada com um X representam a remoção da mesma ou a não correspondência de classes entre versões. A última linha apresenta os totais somados de cada coluna. Esta tabela é importante para ter a conhecimento de quanto o código evoluiu na organização e aperfeiçoamento do sistema. Mantendo as funcionalidades da versão anterior, demostrando assim a evolução do sistema em comparação a versão anterior.

6.2 Número de linhas por Controladora

Uma métrica de software é definida como um número ou símbolo para indicar o atributo de um projeto de software, com a intenção de prover uma descrição correta do projeto.

Resumidamente, existem duas categorias para métricas: diretas e indiretas. Métricas diretas são aquelas que não dependem de nenhum outro atributo do projeto, e com isso, linhas de código são uma métrica direta. No entanto, é de interesse do estudo os fatores de qualidade, tais como complexidade, esforço e densidade de defeitos, métricas indiretas, visto que não podem ser medidas diretamente e devem ser derivadas e comparadas com outras medidas. As linhas de código podem ser definidas como uma métrica indireta, quando são utilizadas como uma estimativa dos fatores de qualidade dos projetos [23].

Tabela 6.3: Comparativo da quantidade de linhas de código por controladora.

Controladoras	**Versão 1.0**	**Versão 2.0**
CriterionController.php / .js	111	178
ProjectController.php / .js	53	113
ScaleController.php / .js	47	140
RegisterController.php	34	X
SocialAuthController.php	30	X
AccountController.php	18	X
OptionAnswerController.php / .js	16	20
UserController.php / .js	15	76
HomeController.php	14	X
ResetPasswordController.php	13	X
ForgotPasswordController.php	12	X
LoginController.php	12	X
SessionController.js	X	121
ScaleResultController.js	X	96
Total	**375**	**744**

Apesar de o número de linhas no código não ser um parâmetro preciso em relação à qualidade do sistema e ser de difícil interpretação, foram computados os números de linhas de código em cada classe controladora. Foram desconsiderados comentários e linhas em branco. A tabela 6.3 possui 3 colunas onde a primeira apresenta o nome das controladoras analisadas bem como as extensões de cada versão, sendo elas *PHP* e *JavaScript (JS)*. A segunda e terceira colunas apresentam as quantidades de linhas de código por controladora e na última linha o total geral de cada versão é exibido.

Na tabela 6.3 é possível notar que houve um aumento nas linhas de código do programa. A explicação para esse aumento se dá pelo fato de que, na nova versão do sistema MyMCDA-C 2.0, todas as requisições recebidas pelo módulo do servidor são tratadas com relação à forma e ao tipo dos dados recebidos. Também há o tratamento de erros em cada um dos métodos das classes, tornando o software mais seguro e estável. Nas

linhas marcadas com um X, a controladora foi removida da primeira versão ou adicionada apenas na segunda.

Para realizar esta etapa de contagem de linhas de código, foi utilizada uma ferramenta para análise de código desenvolvida por Al Danial [24]. Em sua versão 2.8.0 o pacote *Cloc*, distribuído pelo gerenciador de pacotes *NPM*, foi instalado em ambas as versões do MyMCDA-C, de modo a mensurar a quantidade de linhas de código em cada uma das classes controladoras, fornecendo os dados para construção da Tabela 6.3.

Capítulo 7

Conclusão

O presente trabalho apresentou a análise, detecção e correção de *code smells* no software MyMCDA-C, bem como a implementação de novas funcionalidades de modo a aprimorar a experiência do usuário e a robustez do sistema. Manutenção e refatoração de um software é um trabalho que exige multidisciplinaridade dos envolvidos no desenvolvimento. É necessário considerar a experiência do usuário, qualidade e facilidade de manutenção do código. Após uma análise de código da versão 1.0 do software MyMCDA-C, foi possível levantar um total de 30 *code smells*. Em uma reunião com os responsáveis e usuários do MyMCDA-C, também foi possível levantar problemas de usabilidade que dificultavam o dia a dia do usuário.

Com a lista de defeitos e *code smells*, os estudos realizados foram utilizados na refatoração do software, resultando em uma nova versão, com um código livre de *code smells* e com mais transparência para codificar as melhorias. O impacto desses aperfeiçoamentos foram positivos, gerando uma diminuição no número mínimo de cliques para realizar diversas ações no software, em especial a navegação do sistema, que era uma das principais reclamações entre os usuários.

Trabalhos futuros derivados do presente estudo podem considerar a aplicação dos conceitos de programação orientada a objetos SOLID, para avaliar os impactos da implementação dos princípios. Uma possível evolução para o software pode ser a implementação de comparação entre projetos criados, de modo a facilitar a análise das decisões tomadas pelos usuários. Outro ponto interessante para evoluir o software, seria a integração com um sistema de formulário digital. Dessa forma, não será necessário o preenchimento manual de algumas informações pelo usuário e o sistema ficará responsável por calcular todos os resultados. Outra opção para trabalhos futuros é a implementação ou integração de um algoritmo de inteligência artificial para ajudar a tomada de decisão.

Referências

[1] Moreira, Fernando Rocha, Demétrio Antônio Da Silva Filho, Georges Daniel Amvame Nze, Rafael Timóteo de Sousa Júnior e Rafael Rabelo Nunes: *Evaluating the performance of nist's framework c controls through a constructivist multicriteria methodology.* IEEE Access, 9:129605–129618, 2021. 1

[2] Barbalho, Leane, Evaldo César Cavalcante Rodrigues e Clarissa Melo Lima: *Análise multicritério das lacunas entre logística reversa e processamento de bens.* 2020. 1

[3] Poliana, Lourenço, Lima Clarissa e Evaldo Rodrigues: *InfluÊncia do instagram no comportamento do consumidor.* FACES Journal, 19(2), 2021. `http://revista.fumec.br/index.php/facesp/article/view/7523`. 1

[4] Wielewski, Gabriel Lopes, Erica Rodrigues do Amaral, Aldery Silveira Júnior, Evaldo Cesar Rodrigues e Clarissa Melo Lima: *AnÁlise da usabilidade de um site de compras e vendas no brasil.* 30 Enangrad - Encontro Nacional dos Cursos de Graduação em Administração, 2019. `https://app.angrad.org.br/anais/artigo/aa6ebd40-dc00-4d44-9c9b-6a5289064789`. 1

[5] Wielewski, Guilherme Lopes, Aldery Silveira Júnior, Evaldo Cesar Rodrigues, Clarissa Melo Lima e Rafael Rabelo Nunes: *AnÁlise da usabilidade de um site de compras e vendas no brasil.* 30 Enangrad - Encontro Nacional dos Cursos de Graduação em Administração, 2019. `https://app.angrad.org.br/anais/artigo/efb20d1e-ff83-4834-8c96-b697e9b1681b`. 1

[6] Fowler, Martin: *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999, ISBN 0-201-48567-2. 2, 5, 6, 7, 14, 20, 21, 29, 30, 36, 37, 38, 41, 44, 50, 51, 64

[7] Walter, Bartosz e Tarek Alkhaeir: *The relationship between design patterns and code smells: An exploratory study.* Information and Software Technology, 74, março 2016. 2, 10, 11, 20

[8] Aniche, Maurício, Gabriele Bavota, Christoph Treude, Marco Aurelio Gerosa e Arie Deursen: *Code smells for model-view-controller architectures.* Empirical Software Engineering, setembro 2017. 2, 9, 10, 14, 16, 20

[9] Alkharabsheh, Khalid, Yania Crespo, M. Manso e José Taboada: *Software design smell detection: a systematic mapping study.* Software Quality Journal, páginas 1–80, outubro 2018. 2, 12

[10] Rodrigues, Evaldo César Cavalcante: *Metodologia para investigação da percepção das inovações na usabilidade do sistema metroviário: uma abordagem antropotecnológica.* página 262, 2014. http://dx.doi.org/10.26512/2014.12.T.17745. 3

[11] Fowler, Martin: *Codesmell.* https://martinfowler.com/bliki/CodeSmell.html, acesso em 2021-08-12. 5, 7, 9, 14

[12] Saboury, Amir, Pooya Musavi, Foutse Khomh e Giuliano Antoniol: *An empirical study of code smells in javascript projects.* páginas 294–305, fevereiro 2017. 8, 10

[13] Jaafar, Fehmi, Yann Gaël Guéhéneuc, Sylvie Hamel, Foutse Khomh e Mohammad Zulkernine: *Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults.* Empirical Software Engineering, março 2015. 9

[14] Gopstein, Dan, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin Yeh e Justin Cappos: *Understanding misunderstandings in source code.* páginas 129–139, agosto 2017. 11

[15] Ebert, Felipe, Fernando Castor, Nicole Novielli e Alexander Serebrenik: *Confusion in code reviews: Reasons, impacts, and coping strategies.* páginas 49–60, fevereiro 2019. 12, 13

[16] , Uxdesignworld : *3-click rule and usability - ux design world*, May 2021. https://uxdworld.com/2020/01/28/3-click-rule-and-usability/, acesso em 2021-10-11. 15

[17] Sysoev, Igor: *What is nginx?* https://www.nginx.com/resources/glossary/nginx/, acesso em 2021-08-19. 24

[18] contributors, MDN: *Spa (single-page application).* https://developer.mozilla.org/en-US/docs/Glossary/SPA, acesso em 2021-08-19. 24

[19] contributors, MDN: *Json.* https://developer.mozilla.org/en-US/docs/Glossary/JSON, acesso em 2021-08-20. 25

[20] Auth0: *Json web token.* https://jwt.io/introduction, acesso em 2021-08-19. 25

[21] You, Evan: *Vue.js.* https://vuejs.org/, acesso em 2021-07-23. 36

[22] ISO: *Iso 9241.* https://www.iso.org/obp/ui/, acesso em 2021-08-20. 37

[23] Barb, Adrian, Colin Neill, Raghvinder Sangwan e Michael Piovoso: *A statistical study of the relevance of lines of code measures in software projects.* Innovations in Systems and Software Engineering, 10, dezembro 2014. 66

[24] Dodds, Kent C.: *cloc - npm.* https://www.npmjs.com/package/cloc, acesso em 2021-08-07. 67

Anexo I

Acesso aos repositórios do MyMCDA-C

Links de acesso para o novo MyMCDA-C:

- github.com/christianlmc/mymcdac-front (versões 2.0 e 3.0 do módulo de cliente)
- github.com/gabrieltomazz/mymcdac-backend (versões 2.0 e 3.0 do módulo de servidor)