UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Tecnologia

Vinícius Augusto Prates Lourenço

O Controller Gordo sob o Microscópio: Um Estudo Experimental sobre Qualidade de Código em Arquiteturas MVC

Limeira
2025

Vinícius Augusto Prates Lourenço

O Controller Gordo sob o Microscópio: Um Estudo Experimental sobre Qualidade de Código em Arquiteturas MVC

Monografia apresentada à Faculdade de Tecnologia da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de, na área de Sistemas de Informação.

Orientador: Prof. Dr. Plínio Roberto Souza Vilela

Este trabalho corresponde à versão final da defendida por Vinícius Augusto Prates Lourenço e orientada pelo Prof. Dr. Plínio Roberto Souza Vilela.

Limeira
2025

Ficha catalográfica
Universidade Estadual de Campinas (UNICAMP)
Biblioteca da Faculdade de Tecnologia
Felipe de Souza Bueno - CRB 8/8577

Lourenço, Vinícius Augusto Prates, 1999-
L934c O Controller Gordo sob o microscópio : um estudo experimental sobre qualidade de código em arquiteturas MVC / Vinícius Augusto Prates Lourenço. – Limeira, SP : [s.n.], 2025.

Orientador: Plínio Roberto Souza Vilela.
Trabalho de Conclusão de Curso (Graduação) – Universidade Estadual de Campinas (UNICAMP), Faculdade de Tecnologia.

1. Software - Controle de qualidade. 2. Arquitetura de software. 3. Análise qualitativa. 4. Análise quantitativa. I. Vilela, Plínio Roberto Souza, 1970-. II. Universidade Estadual de Campinas (UNICAMP). Faculdade de Tecnologia. III. Título.

Informações complementares

Palavras-chave em inglês:
Software - Quality control
Software architecture
Qualitative analysis
Quantitative analysis
Titulação: Bacharel
Banca examinadora:
Plínio Roberto Souza Vilela [Orientador]
Lívia Couto Ruback Rodrigues
Gisele Busichia Baioco
Data de entrega do trabalho definitivo: 15-07-2025

Objetivos de Desenvolvimento Sustentável (ODS)
Não se aplica

FOLHA DE APROVAÇÃO

Abaixo se apresentam os membros da comissão julgadora da monografia correspondente ao trabalho de conclusão de curso para obtenção do Título de Bacharel em Sistemas de Informação, a que se submeteu o aluno Vinícius Augusto Prates Lourenço, em 16 de julho de 2025 na Faculdade de Tecnologia – FT/UNICAMP, em Limeira/SP

Prof. Dr. Plínio Roberto Souza Vilela

Presidente da Comissão Julgadora
FT/UNICAMP

Profa. Dra. Lívia Couto Ruback Rodrigues

FT/UNICAMP

Profa. Dra. Gisele Busichia Baioco

FT/UNICAMP

Ata da defesa, assinada pelos membros da Comissão Examinadora, encontra-se no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria de da Faculdade de Tecnologia.

Agradecimentos

Chegar ao final desta jornada é um marco que não se alcança sozinho. O caminho foi longo, por vezes árduo, mas sempre enriquecido pelo apoio, paciência e sabedoria de muitas pessoas a quem desejo expressar minha mais profunda gratidão.

Agradeço, em primeiro lugar, ao meu orientador, Prof. Dr. Plínio Roberto Souza Vilela, pela confiança depositada em mim e neste projeto. Sua orientação precisa, seu rigor metodológico e seus questionamentos perspicazes foram fundamentais para transformar uma ideia inicial em um trabalho de pesquisa com o qual me orgulho. Sua paciência e dedicação foram o alicerce que me permitiu construir esta monografia.

À minha família, em especial aos meus pais, Damião e Márcia, e à minha namorada, Mariana, dedico esta conquista. Vocês foram a base de tudo. Agradeço pelo amor incondicional, pelo incentivo constante e, acima de tudo, pela imensa paciência durante minhas longas horas de ausência e dedicação ao código e à escrita. Sem o apoio de vocês, nada disso seria possível.

Aos meus amigos e colegas de curso, que compartilharam as ansiedades, os cafés e as longas noites de estudo, meu sincero obrigado pela camaradagem e pelo suporte mútuo que tornaram a jornada mais leve e divertida.

Por fim, um agradecimento especial às ferramentas e tecnologias que me auxiliaram neste processo. Em um mundo em constante evolução, as longas noites de diálogo com modelos de inteligência artificial se mostraram, de forma inesperada, uma valiosa parceria na organização de ideias, na superação de bloqueios e no refino do argumento aqui presente.

A todos que, direta ou indiretamente, fizeram parte desta etapa, o meu mais profundo e sincero obrigado.

Resumo

A integridade da arquitetura Model-View-Controller (MVC) é um pilar para a manutenibilidade de sistemas, mas a sua eficácia é frequentemente comprometida pela alocação inadequada de responsabilidades entre as camadas. Este trabalho investiga o impacto de um anti-padrão comum: o deslocamento da lógica de apresentação da View para a camada de Controller. Através de um estudo experimental controlado, duas versões de um sistema Java Swing foram comparadas: uma com um Controller sobrecarregado (Versão A) e outra com uma separação de responsabilidades estrita (Versão B). A qualidade foi avaliada por um painel de arquitetos de software experientes (análise qualitativa) e por meio de métricas de código extraídas via SonarQube (análise quantitativa). Os resultados indicam que a versão arquiteturalmente mais correta (B) foi percebida como marginalmente mais manutenível, mas não apresentou melhora na testabilidade percebida. Notavelmente, não foi encontrada correlação estatística significativa entre a percepção de qualidade dos especialistas e a Dívida Técnica medida pela ferramenta. Conclui-se que as métricas automatizadas, embora eficazes para detectar falhas de implementação, podem ser um proxy inadequado para a saúde arquitetural, reforçando a necessidade de uma sinergia crítica entre a avaliação humana e a automatizada.

Palavras-chave: Qualidade de Software, Arquitetura MVC, Análise Estática de Código, SonarQube, Débito Técnico, Engenharia de Software Empírica.

Abstract

The integrity of the Model-View-Controller (MVC) architecture is a cornerstone for system maintainability, yet its effectiveness is often compromised by the improper allocation of responsibilities between layers. This work investigates the impact of a common anti-pattern: the misallocation of presentation logic from the View to the Controller layer. Through a controlled experimental study, two versions of a Java Swing system were compared: one with an overloaded "Fat Controller" (Version A) and another with a strict separation of concerns (Version B). Quality was assessed by a panel of experienced software architects (qualitative analysis) and through code metrics extracted via SonarQube (quantitative analysis). The results indicate that the architecturally correct version (B) was perceived as marginally more maintainable but showed no improvement in perceived testability. Notably, no statistically significant correlation was found between the experts' perception of quality and the Technical Debt measured by the tool. We conclude that automated metrics, while effective for detecting implementation flaws, can be an inadequate proxy for architectural health, reinforcing the critical need for a synergy between human and automated evaluation.

Keywords: Software Quality, MVC Architecture, Static Code Analysis, SonarQube, Technical Debt, Empirical Software Engineering.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contextualização e Problema de Pesquisa

Com o aumento da complexidade dos sistemas de software, garantir a qualidade do código tornou-se uma tarefa essencial em qualquer etapa do desenvolvimento. Um código mal estruturado com baixa legibilidade ou propenso a falhas pode se transformar rapidamente em dívida técnica, o que compromete a evolução do sistema, encarece a manutenção e até mesmo coloca em risco a segurança da aplicação (Pressman, 2016).

Nesse contexto, surgem diferentes caminhos para avaliar e manter a qualidade do código. Enquanto algumas equipes optam por revisões manuais criteriosas, outras apostam no uso de ferramentas automatizadas e muitas combinam ambas as estratégias em busca de um equilíbrio entre agilidade e profundidade.

A arquitetura Model-View-Controller (MVC), bastante popular por facilitar a separação de responsabilidades, não garante por si só um código limpo ou sustentável. A qualidade final depende de como os desenvolvedores estruturam a lógica, respeitam boas práticas (como os princípios SOLID) (Martin, 2002) e organizam os componentes.

Dentro da arquitetura MVC, uma dúvida recorrente é a exata delimitação das responsabilidades da View e do Controller. Onde a lógica de apresentação deve residir? Deslocar essa lógica para o Controller (uma prática comum) pode gerar acoplamento e ferir os princípios da separação de responsabilidades, mas qual o impacto disso na qualidade do código?

1.2 Justificativa

A decisão de onde alocar a lógica de negócio em uma arquitetura MVC tem implicações diretas na saúde e longevidade de um projeto. No entanto, a literatura acadêmica frequentemente se concentra na comparação entre padrões ideais, deixando uma lacuna na análise de implementações do "mundo real", que muitas vezes evoluem para soluções não ideais sob as pressões de prazos e mudanças de requisitos.

Este trabalho se justifica precisamente por investigar esta "zona cinzenta". Em vez de comparar um anti-padrão com uma arquitetura perfeita, o objetivo aqui é analisar e contrastar dois anti-padrões extremamente comuns na indústria: o "Controller Gordo", que absorve lógica de negócio, e o "Controller Marionetista", que se acopla diretamente à tecnologia da View. A pesquisa busca entender se existe uma diferença mensurável na qualidade percebida e métrica entre esses dois tipos de débito técnico arquitetural. A

questão não é “qual é o certo?”, mas sim: “dentre dois desvios comuns, um é mensuravelmente ’menos pior’ que o outro?”.

A contribuição prática desta abordagem é fornecer evidências que ajudem equipes a tomar decisões mais informadas durante a refatoração de sistemas legados ou em evolução, oferecendo um poderoso artefato de argumentação para justificar o combate a um tipo de anti-padrão em detrimento de outro, com base em seus impactos específicos na manutenibilidade, testabilidade e segurança.

1.3 Objetivos

O presente trabalho tem como objetivo principal avaliar o impacto do deslocamento da lógica de negócio da camada View para a camada Controller na qualidade de código em sistemas que utilizam a arquitetura MVC.

1.4 Objetivos Específicos

Para alcançar o objetivo geral, foram definidos os seguintes objetivos específicos:

1. Realizar uma revisão bibliográfica sobre qualidade de software, arquitetura MVC, anti-padrões e ferramentas de análise estática (Lenarduzzi et al., 2021);
2. Desenvolver um estudo de caso composto por duas versões de um sistema, isolando a estratégia de alocação da lógica de negócio (ora na View, ora no Controller) como variável independente;
3. Coletar dados qualitativos sobre a percepção de qualidade dos arquitetos a partir da comparação entre os dois cenários implementados (Versão A e Versão B);
4. Extrair métricas quantitativas de qualidade de código de ambas as versões do sistema (A e B) utilizando a ferramenta SonarQube (SonarSource, 2025);
5. Analisar e comparar os dados obtidos para validar as hipóteses da pesquisa.

1.5 Estrutura do Trabalho

A presente monografia encontra-se estruturada em seis capítulos, de modo a guiar o leitor de forma progressiva desde a contextualização do problema até as conclusões da pesquisa.

O Capítulo 2 aprofunda a fundamentação teórica, estabelecendo os conceitos de qualidade de software, arquitetura MVC e seus anti-padrões, além de discutir as abordagens de avaliação qualitativa e quantitativa que alicerçam este estudo. Em seguida, o Capítulo 3 é dedicado à descrição do estudo de caso, detalhando a aplicação “Clínica Vet” e as particularidades arquiteturais das duas versões (A e B) que foram desenvolvidas e submetidas à análise. A metodologia empregada na pesquisa é o foco do Capítulo 4, onde são delineados o desenho do estudo experimental, o perfil dos participantes, os procedimentos de coleta de dados e o plano de análise estatística utilizado para a validação das hipóteses. O Capítulo 5 apresenta e discute os resultados obtidos, cruzando os dados da avaliação dos arquitetos com as métricas extraídas pela ferramenta SonarQube, e interpretando os achados à luz da teoria e dos objetivos

propostos. Por fim, o Capítulo 6 sintetiza o trabalho, retomando os objetivos, destacando as principais contribuições da pesquisa, reconhecendo suas limitações e sugerindo caminhos para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

A avaliação da qualidade de um software é uma disciplina multifacetada, que se apoia em conceitos de arquitetura, métricas objetivas e percepção humana. Esta seção estabelece as bases teóricas que fundamentam nosso estudo, partindo dos desafios inerentes à arquitetura MVC, passando pelas abordagens de avaliação de código, até a instrumentalização dessa avaliação por meio de ferramenta de análise estática.

2.1 Padrões Arquiteturais, Anti-Padrões e Qualidade

A construção de software complexo raramente parte do zero. A disciplina de engenharia de software se apoia em um catálogo de soluções consolidadas para problemas recorrentes, conhecidas como Padrões Arquiteturais. Um padrão, como o MVC, é uma solução geral e reutilizável que define uma estrutura organizacional para os componentes de um sistema, promovendo atributos de qualidade desejáveis (Gamma et al., 1994). Em contrapartida, um Anti-Padrão representa uma solução que, embora pareça eficaz a curto prazo, gera consequências negativas, introduzindo débitos técnicos que comprometem a manutenibilidade e a evolução do software a longo prazo (Fowler, 2002). O "Controller Gordo", foco deste estudo, é um exemplo clássico de anti-padrão que emerge de uma aplicação inadequada do padrão MVC.

2.2 A Arquitetura MVC e o Anti-Padrão "Controller Gordo"

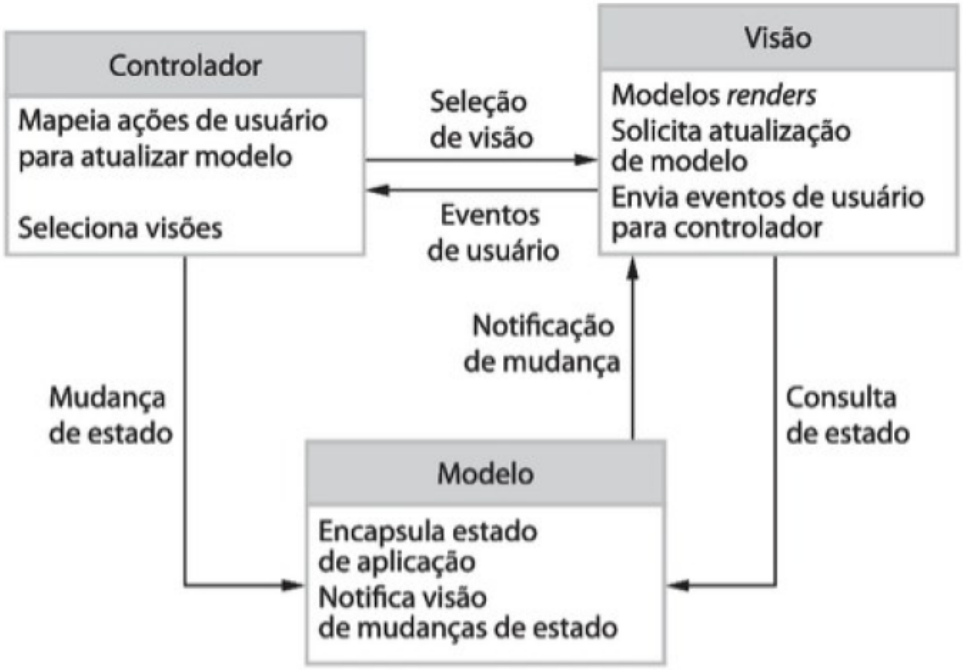

Figura 2.1: Fluxo de interação do padrão arquitetural Model-View-Controller (MVC), ilustrando a separação de responsabilidades.
Fonte: (Gran Cursos Online, 2023)

O padrão de arquitetura Model-View-Controller (MVC) consolidou-se na indústria de software como uma resposta elegante ao desafio de organizar sistemas interativos. Sua premissa fundamental é a separação de responsabilidades (Separation of Concerns), dividindo o sistema em três camadas interconectadas: o Model, que encapsula os dados e as regras de negócio; a View, responsável exclusivamente pela apresentação desses dados ao usuário; e o Controller, que atua como um maestro, orquestrando a interação entre o usuário, a View e o Model.

Para avaliar a qualidade dessa separação, a engenharia de software recorre a dois conceitos fundamentais: coesão e acoplamento. A coesão refere-se ao grau em que os elementos internos de um módulo (ou classe) pertencem uns aos outros, medindo o quão focada e singular é a sua responsabilidade. Uma alta coesão é, portanto, um atributo desejável. Em contrapartida, o acoplamento mede o nível de interdependência entre módulos distintos. O objetivo é alcançar um baixo acoplamento, pois isso indica que uma mudança em um componente tem menor probabilidade de propagar efeitos colaterais para outros, tornando o sistema mais resiliente e fácil de manter.

A correta aplicação do padrão MVC, portanto, busca exatamente maximizar a coesão interna de cada camada e minimizar o acoplamento entre elas, o que, por sua vez, resulta em um sistema mais testável, flexível e de fácil manutenção.

Contudo, a mera adoção do padrão não é uma panaceia. A saúde arquitetural de uma aplicação MVC depende da disciplina da equipe em manter as fronteiras entre as camadas. Um dos desvios mais comuns e danosos é o que a literatura convencionou chamar de "Controller Gordo" (Fat Controller) (Fowler, 2002). Este anti-padrão emerge quando o

Controller, que deveria ser um intermediário enxuto, começa a absorver responsabilidades que não lhe pertencem. As formas mais comuns deste desvio incluem o acúmulo de lógica de negócio (que deveria residir no Model) e de lógica de manipulação da apresentação (que pertence à View). O presente estudo investiga precisamente os impactos de diferentes facetas deste problema, analisando uma versão onde o Controller assume regras de negócio (a Versão A, nosso "Controller Gordo") e outra onde ele se acopla diretamente à tecnologia da interface (a Versão B, nosso "Controller Marionetista"). Ao fazer isso, o Controller viola o Princípio da Responsabilidade Única (SRP) (Martin, 2002), tornando-se uma classe monolítica, difícil de compreender, testar e evoluir. Identificar e medir o impacto deste e de outros "vazamentos de responsabilidade" é o principal desafio na garantia de qualidade de projetos MVC.

Em contraste com o "Controller Gordo", que absorve lógica de negócio, existe uma outra variação de anti-padrão que emerge quando o Controller desenvolve um conhecimento profundo e uma dependência direta da tecnologia de implementação da View. A este desvio podemos denominar "Controller Marionetista" (Puppeteer Controller). Neste cenário, o Controller recebe e armazena referências a componentes visuais concretos (como JTable ou JTextField), manipulando diretamente suas propriedades e estado. Ao fazer isso, a barreira de abstração entre as camadas é quebrada, acoplando a lógica de controle à tecnologia de apresentação e tornando a substituição da View ou a realização de testes unitários no Controller tarefas extremamente difíceis.

2.3 O Princípio da Responsabilidade Única (SOLID)

Um dos pilares do bom design de software orientado a objetos é o Princípio da Responsabilidade Única (SRP), o primeiro dos cinco princípios SOLID de Robert C. Martin (Martin, 2002). Ele afirma que uma classe deve ter uma, e somente uma, razão para mudar. No contexto da arquitetura MVC, o Controller ideal adere a este princípio ao focar exclusivamente na mediação: receber requisições da View, delegar a lógica de negócio ao Model e selecionar a próxima View a ser exibida. Quando o Controller acumula responsabilidades de negócio ou de apresentação, como nos casos estudados, este princípio é diretamente violado.

2.4 Débito Técnico

O conceito de Débito Técnico, cunhado por Ward Cunningham, utiliza uma poderosa metáfora financeira para descrever as consequências de se optar por uma solução de código mais fácil agora, em detrimento de uma abordagem melhor que levaria mais tempo (Cunningham, 1992). Assim como uma dívida financeira, o débito técnico acumula "juros", que se manifestam como um aumento no custo da manutenção e na dificuldade de implementar novas funcionalidades no futuro. Os anti-padrões arquiteturais, como os analisados neste trabalho, são uma forma de débito de alto impacto, pois comprometem a estrutura fundamental do sistema.

2.5 A Avaliação da Qualidade de Código: Abordagens e Métricas

Para diagnosticar a saúde de uma base de código e identificar desvios arquiteturais, a engenharia de software dispõe de um arsenal de técnicas de avaliação, que podem ser categorizadas em duas grandes dimensões: a qualitativa, baseada no julgamento humano, e a quantitativa, baseada em medições objetivas. Uma comparação detalhada entre as características de cada abordagem é apresentada no Apêndice A.

2.5.1 A Dimensão Humana: Análise Qualitativa e Percepção de Especialistas

A análise humana, formalizada em práticas como a revisão por pares (peer review), representa uma avaliação essencialmente cognitiva e contextual. Ela transcende a sintaxe e mergulha na semântica do código, onde um analista experiente avalia atributos subjetivos como clareza e legibilidade. Um código é "claro" quando sua intenção é evidente; é "legível" quando sua estrutura e formatação facilitam a compreensão (Lefever et al., 2021). Mais importante, no contexto deste trabalho, é a capacidade humana de avaliar a adequação arquitetural. Um especialista pode identificar, por meio da inspeção, se um Controller está indevidamente acoplado a componentes da View ou se a coesão de uma classe foi comprometida. Essa percepção, embora subjetiva, é fundamental para capturar nuances de design que ferramentas automatizadas frequentemente ignoram (Borg et al., 2024). É essa expertise que buscamos mensurar através da avaliação dos arquitetos neste estudo.

2.5.2 A Dimensão Objetiva: Métricas Quantitativas e Análise Estática

Em contrapartida à subjetividade humana, a análise estática oferece um método automatizado e preventivo para inspecionar o código-fonte sem executá-lo, gerando métricas quantitativas. Essas métricas traduzem características do código em valores numéricos, permitindo comparações objetivas e o monitoramento do débito técnico (Cunningham, 1992; Lefever et al., 2021).

Para o escopo do nosso estudo, foram selecionadas métricas consolidadas na literatura. As três primeiras, sendo WMC, CBO e RFC, fazem parte do influente conjunto de métricas para design orientado a objetos proposto por Chidamber e Kemerer (Chidamber; Kemerer, 1994):

- **Weighted Methods per Class (WMC):** Mede a soma da complexidade de todos os métodos dentro de uma única classe. Um valor de WMC elevado é um forte indicador de que a classe pode estar violando o Princípio da Responsabilidade Única (SRP), pois sugere que ela acumulou responsabilidades demais para si. No contexto deste estudo, um WMC alto no Controller é o principal sintoma numérico do anti-padrão "Fat Controller".

- **Coupling Between Objects (CBO):** Mede o número de outras classes das quais uma classe depende. Um alto acoplamento em qualquer camada do MVC é indesejável, mas é especialmente problemático quando um Controller se acopla diretamente a detalhes de implementação da View.

- **Response for a Class (RFC):** O número de métodos que podem ser executados em resposta a uma mensagem recebida por um objeto da classe. Assim como o WMC, um RFC alto pode indicar que a classe tem responsabilidades demais.

Adicionalmente, para obter uma visão agregada da qualidade, utilizamos o Maintainability Index, originalmente proposto por Oman e Hagemeister (Oman; Hagemeister, 1992):

- **Maintainability Index (MI):** Um índice composto, calculado a partir de métricas como a Complexidade Ciclomática e Linhas de Código (LOC), que busca fornecer um escore único para a manutenibilidade geral do código.

Essa dualidade entre a percepção qualitativa e a medição quantitativa é central para este trabalho. Buscamos investigar se a avaliação de "bom design" feita por um especialista (análise qualitativa) se correlaciona com bons indicadores nas métricas de código (análise quantitativa).

2.6 Instrumentalização da Análise: O Papel do SonarQube

A coleta manual de métricas quantitativas, embora possível, é um processo lento e propenso a erros. É nesse contexto que plataformas de análise estática como o SonarQube se tornam instrumentais para a pesquisa e para a indústria. O SonarQube é uma plataforma de código aberto que automatiza a inspeção contínua da qualidade do código, integrando-se aos fluxos de desenvolvimento para fornecer feedback rápido sobre a "saúde" do projeto (SonarSource, 2025).

Operando através de Scanners que analisam o código-fonte e enviam relatórios para um Server central, o SonarQube processa os dados e os apresenta em um dashboard interativo. Sua função, no âmbito desta pesquisa, é atuar como a principal ferramenta para a coleta de dados quantitativos. Utilizando perfis de qualidade pré-definidos (como o "Sonar way"), ele calcula automaticamente as métricas de WMC, CBO, complexidade e outras, além de identificar Bugs, Vulnerabilities e *Code Smells* com base em um vasto conjunto de regras. Ao fornecer uma medição objetiva e reprodutível, o SonarQube nos permite operacionalizar a dimensão quantitativa da nossa avaliação comparativa entre as versões A e B do software analisado.

Capítulo 3

Desenvolvimento do Estudo de Caso

Este capítulo detalha o artefato de software que serve como objeto de estudo para o nosso experimento. Apresentamos aqui a descrição funcional do sistema desenvolvido, as tecnologias empregadas e, de forma crucial, as características arquiteturais das duas versões (A e B) que foram submetidas à análise, destacando as diferenças de design que constituem a variável independente desta pesquisa.

Descrição Geral do Sistema "Clínica Vet"

O sistema objeto deste estudo é uma aplicação de desktop para a gestão de uma clínica veterinária. Desenvolvido como um protótipo funcional, ele encapsula as operações essenciais do dia a dia de um estabelecimento do gênero. Suas principais funcionalidades incluem:

- **Gestão de Clientes e Pacientes:** Cadastro, edição, busca e exclusão de clientes (tutores) e de seus respectivos animais (pacientes).
- **Gestão de Veterinários:** Cadastro e consulta de profissionais que atuam na clínica.
- **Controle de Consultas:** Agendamento de novas consultas, associando um animal a um veterinário em uma data e horário específicos.
- **Acompanhamento de Tratamentos e Exames:** Registro de tratamentos contínuos e dos exames solicitados durante as consultas.

A aplicação foi escolhida por sua representatividade de um sistema de informação padrão, com entidades e relacionamentos de complexidade moderada, tornando-a um campo fértil para a análise de decisões arquiteturais no contexto do padrão MVC.

3.1 Tecnologias Utilizadas

O projeto foi desenvolvido utilizando um conjunto de tecnologias consolidadas no ecossistema Java, buscando simular um ambiente de desenvolvimento comum na indústria:

- Linguagem de Programação: Java, utilizando o Java Development Kit (JDK) na versão 17.

- Interface Gráfica (GUI): A camada de apresentação foi construída com a biblioteca nativa Java Swing.
- Persistência de Dados: Foram utilizadas duas abordagens de banco de dados SQL embutido: H2 Database e SQLite, com acesso via JDBC (Java Database Connectivity).
- Análise Estática: A ferramenta empregada para a coleta de métricas quantitativas foi o SonarQube (versão 9.9 LTS), executado através do SonarScanner.

3.2 Caracterização das Versões Analisadas

Antes de nos aprofundarmos na análise do anti-padrão arquitetural que caracteriza cada versão do software, é pertinente apresentar uma visão geral e quantitativa dos artefatos analisados. Esta seção caracteriza as Versões A e B em termos de sua interface gráfica, estrutura de pacotes e volume de código.

Conforme ilustrado nas Figuras 3.1 e 3.2, as duas implementações do sistema, embora funcionalmente equivalentes, apresentam diferenças visuais em sua interface com o usuário. A Versão A utiliza uma disposição de abas para separar as funcionalidades, enquanto a Versão B opta por uma janela única com uma organização distinta de painéis.

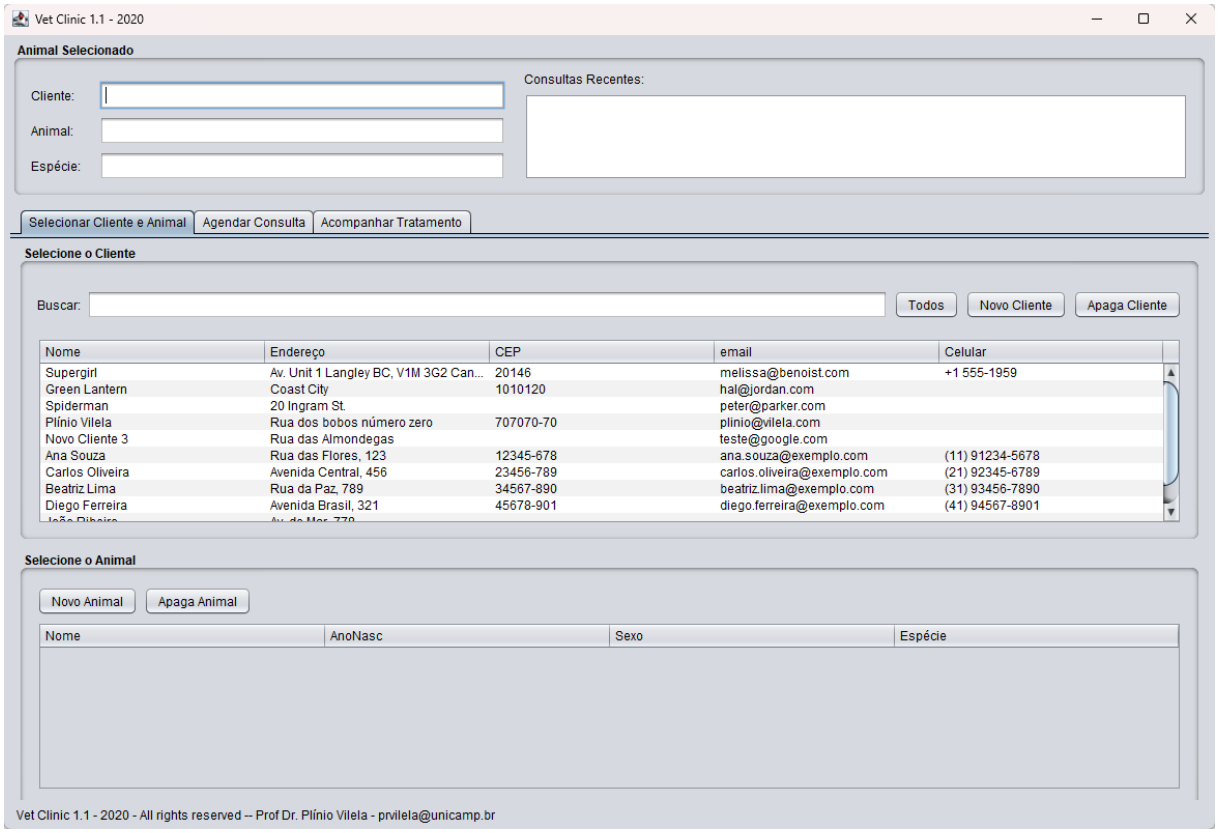

Figura 3.1: Interface gráfica principal da Versão A do sistema 'Clínica Vet'.

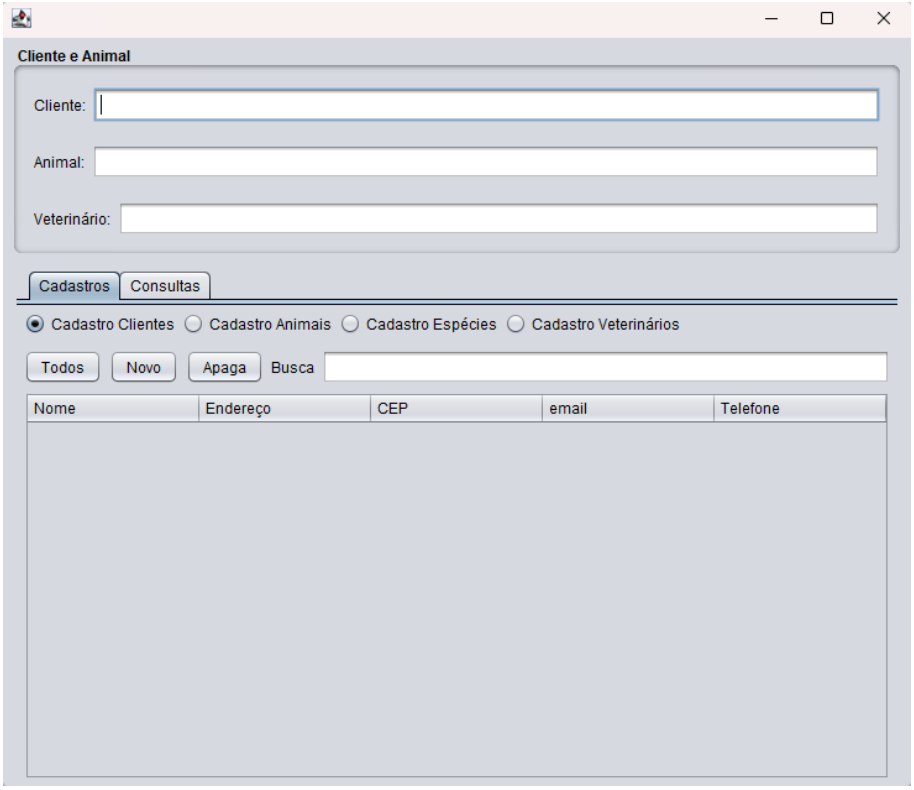

Figura 3.2: Interface gráfica principal da Versão B do sistema 'Clínica Vet'.

É importante esclarecer que a diferença visual entre as interfaces (Figura 3.1 e 3.2) não foi uma decisão de design intencional para o estudo, mas sim uma consequência do processo de refatoração. A transição da arquitetura "Controller Gordo" (Versão A) para a "Controller Marionetista" (Versão B) envolveu não apenas a realocação de lógica no Controller, mas também alterações na própria estrutura das classes da camada View. Ao invés de tentar forçar uma paridade visual que mascararia as diferenças reais do código-fonte, optou-se por apresentar as duas versões em seu estado autêntico, tratando a diferença visual como uma variável de confusão, conforme declarado na Metodologia.

A estrutura de pacotes do projeto foi mantida idêntica em ambas as versões, organizada nas três camadas canônicas do padrão MVC. A refatoração, contudo, resultou em uma redistribuição do código entre estas camadas, como detalhado na Tabela 3.1, que apresenta um comparativo do número de Linhas de Código (LOC).

Pacote	**LOC Versão A**	**LOC Versão B**	**Variação**
`controller`	188	239	+27,13%
`view`	528	569	-5,76%
`model`	1648	1226	-25,61%
Total	**2374**	**2034**	**-14,32%**

Tabela 3.1: Comparativo de Linhas de Código (LOC) por pacote entre as versões A e B do sistema.

É importante notar, ao analisar os dados da Tabela 3.1, uma aparente contradição: a Versão A, que será descrita como "Controller Gordo", possui menos linhas de código

em seu pacote controller do que a Versão B. Isso ocorre porque a "gordura" arquitetural da Versão A reside na quantidade excessiva de responsabilidades e métodos distintos, e não necessariamente no volume de código de cada um. Em contrapartida, o controller da Versão B, embora com menos responsabilidades de negócio, contém uma lógica de fluxo de interface mais verbosa e complexa, o que infla sua contagem de linhas. As seções a seguir detalharão essa diferença qualitativa na arquitetura de cada versão.

A análise da Tabela 3.1 revela uma interessante reorganização do código entre as versões. Conforme esperado, o pacote controller da Versão B cresceu em volume (+27,13%), um reflexo de ter absorvido responsabilidades de fluxo da interface. Contudo, a mudança mais expressiva ocorreu no pacote model, que teve uma redução significativa de mais de 25% em seu tamanho. O pacote view, por sua vez, apresentou um leve crescimento, contrariando a expectativa inicial de que seria reduzido.

3.3 Arquitetura da Versão A ("Controller Gordo")

A Versão A do sistema foi intencionalmente estruturada para refletir um anti-padrão prevalente: o "Controller Gordo" (Fat Controller) ou, em sua forma mais extrema, a "Classe Deus" (God Class). Nesta abordagem, a classe controller.Controller transcende sua função de mediadora e acumula um número excessivo de responsabilidades, centralizando tanto a lógica de negócio quanto a orquestração do fluxo da aplicação.

As principais características arquiteturais desta versão são:

1. Baixa Coesão e Violação do SRP: A classe Controller possui 38 métodos públicos que gerenciam 7 entidades distintas (Cliente, Animal, Veterinário, etc.). Isso viola frontalmente o Princípio da Responsabilidade Única (SRP), tornando a classe um monólito de difícil compreensão e manutenção.

2. Lógica de Negócio na Camada de Controle: Regras de negócio críticas, como a exclusão em cascata, estão implementadas diretamente no Controller. Conforme ilustra o Código 3.1, ao deletar um cliente, é o Controller quem detém a responsabilidade de primeiro apagar os animais e tratamentos associados, uma lógica que deveria pertencer à camada de Model ou a uma camada de serviço.

3. Acoplamento com a Camada de Dados: Todos os métodos acessam a camada de persistência através de chamadas estáticas a Singletons (DAO.getInstance()), o que torna a classe fortemente acoplada à implementação do acesso a dados e virtualmente intestável de forma isolada.

```
public static void deleteClient(Client client){
    // O Controller sabe que precisa apagar os animais primeiro.
    List animals = getAllAnimalsByClientId(client.getId());
    for (Object animal : animals) {
        deleteAnimal((Animal)animal);
    }
    // Somente depois ele apaga o cliente.
    ClientDAO.getInstance().delete(client);
}
```

Código 3.1: Trecho do Controller da Versão A, evidenciando a lógica de negócio de exclusão em cascata.

Além disso, a camada de View na Versão A, especialmente nas classes TableModel, também apresenta violações, chamando diretamente a camada DAO para realizar atualizações, quebrando o fluxo canônico do MVC.

3.4 Arquitetura da Versão B ("Controller Marionetista")

A Versão B representa uma tentativa de refatoração da Versão A. Embora o objetivo fosse buscar uma separação de responsabilidades mais estrita, a implementação resultante introduziu um novo e igualmente grave anti-padrão: o "Controller Marionetista" (Puppeteer Controller).

Nesta arquitetura, o Controller desenvolve um conhecimento profundo e uma dependência direta da tecnologia de implementação da View (Java Swing).

1. Acoplamento Direto com a UI: A classe Controller agora importa componentes Swing (JTable, JTextField, etc.). Métodos como setTextFieldsToControl recebem instâncias diretas desses componentes, permitindo que o Controller os manipule como um marionetista manipula seus bonecos (Código 3.2). Isso quebra a abstração e torna impossível reutilizar o Controller com qualquer outra tecnologia de View.

2. Gestão de Estado Global e Estático: A aplicação passa a gerenciar o estado da seleção do usuário (e.g., clienteSelecionado) por meio de variáveis static no Controller. Esta é uma prática perigosa que cria um estado global, impede a testabilidade e é inerentemente insegura em ambientes com mais de uma thread.

3. Lógica de Apresentação no Controller: O Controller contém lógica que deveria ser exclusiva da View. No método setSelected, por exemplo, é o Controller quem decide qual texto deve ser inserido em qual JTextField da tela.

```
1  public class Controller {
2      private static Cliente clienteSelecionado = null;
3      private static JTextField clienteSelecionadoTextField = null;
4
5      // Controller recebe e armazena refer ncias diretas a componentes
       da UI.
6      public static void setTextFieldsToControl(JTextField cliente, ...) {
7          clienteSelecionadoTextField = cliente;
8          //...
9      }
10
11     public static void setSelected(Object selected) {
12         if (selected instanceof Cliente) {
13             clienteSelecionado = (Cliente) selected;
14             // Controller manipula diretamente o texto de um componente
       Swing.
15
       clienteSelecionadoTextField.setText(clienteSelecionado.getNome());
16         }
17         //...
18     }
19 }
```

Código 3.2: Trecho do Controller da Versão B, mostrando o acoplamento direto com componentes Swing e o gerenciamento de estado estático.

Em suma, enquanto a Versão A peca por misturar as responsabilidades do Model e do Controller, a Versão B comete o erro de fundir as responsabilidades do Controller e da View. Ambas as implementações, embora funcionais, representam desvios significativos do padrão MVC e servem como objetos de estudo ideais para analisar o impacto de diferentes tipos de débito técnico arquitetural.

Capítulo 4

Metodologia

Para investigar o impacto da distribuição de lógica na qualidade de código em sistemas MVC, este trabalho adotou um desenho de estudo experimental controlado. A escolha por um experimento se justifica pela necessidade de isolar variáveis e estabelecer uma base comparativa robusta, permitindo uma análise mais aprofundada das relações de causa e efeito entre as práticas de implementação e seus resultados mensuráveis. Esta seção detalha o desenho deste estudo, os artefatos de software analisados, os procedimentos para coleta de dados e o plano de análise estatística empregado para testar nossas hipóteses.

4.1 Objeto de Estudo e Variáveis da Pesquisa

O objeto central da nossa investigação foi um sistema de desktop para gestão de uma clínica veterinária, desenvolvido em Java com a biblioteca de interface gráfica Swing. A escolha recaiu sobre este projeto por sua representatividade de uma aplicação de negócio padrão, com complexidade suficiente para que os desafios arquiteturais do MVC se manifestassem de forma clara.

A partir desta base de código, foram preparadas duas versões distintas para a comparação, que representam a nossa variável independente — a estratégia de alocação da lógica de apresentação:

- Versão A (Implementação com "Controller Gordo"): Nesta versão, o código foi deliberadamente estruturado para representar um anti-padrão comum. O Controller transcende sua responsabilidade de mediador e passa a conter lógica de negócio explícita. Na prática, isso significa que a classe do Controller recebe instâncias de componentes visuais (como JLabel ou JTable), sendo diretamente responsável por manipular seu estado (e.g., invocar label.setText() ou popular um TableModel).

- Versão B (Implementação com Separação Estrita): Esta versão foi refatorada para aderir estritamente aos princípios do MVC. A View torna-se plenamente responsável por gerenciar seu próprio estado e componentes. O Controller, por sua vez, opera sem qualquer conhecimento sobre os detalhes de implementação da interface; ele apenas recebe eventos da View (como o clique de um botão), aciona as regras de negócio no Model e notifica a View de que uma atualização é necessária, deixando que a própria View decida como se redesenhar, conforme ilustra o diagrama da Figura 4.1.

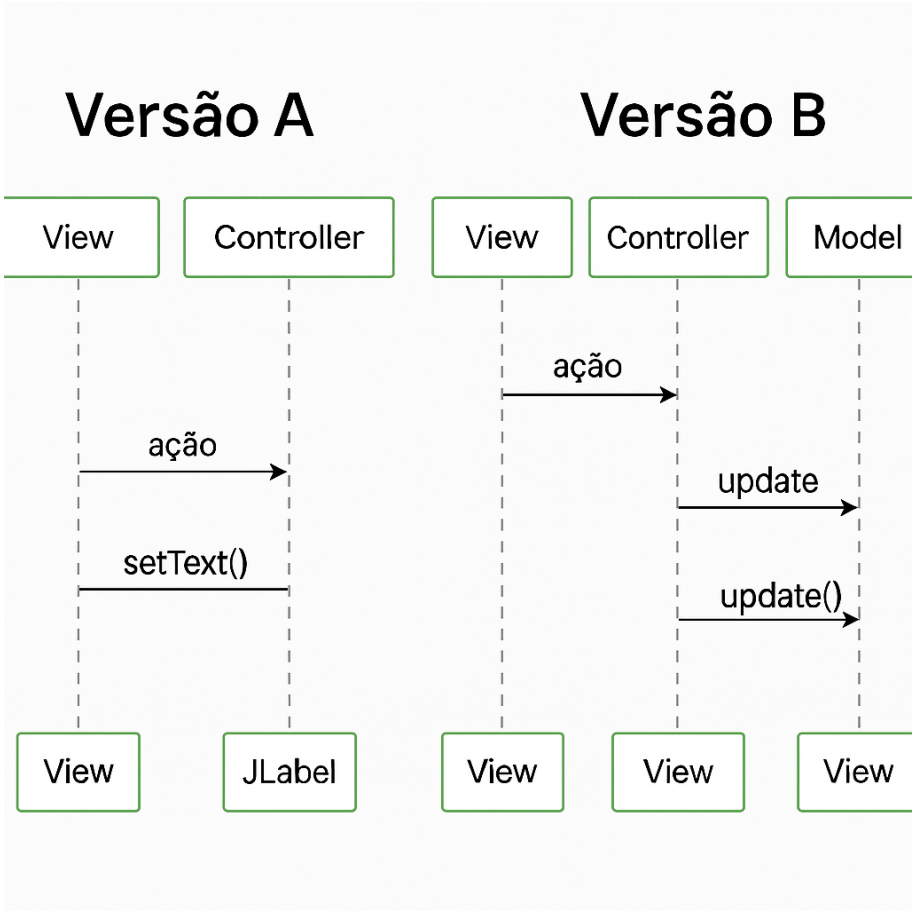

Figura 4.1: Diagrama de interação ilustrando a alocação de lógica na Versão A (esquerda) e na Versão B (direita)

Para medir o impacto dessa variável independente, definimos dois conjuntos de variáveis dependentes:

1. Dependentes Qualitativas: A percepção de qualidade aferida por especialistas. Para isso, foram coletados escores em uma escala Likert de 1 (muito ruim) a 5 (muito bom) para três atributos centrais: Simplicidade, Testabilidade e Manutenibilidade.

2. Dependentes Quantitativas: Métricas objetivas de código, extraídas automaticamente por ferramentas. As principais métricas selecionadas foram WMC (Weighted Methods per Class), CBO (Coupling Between Objects), RFC (Response for a Class) e o Índice de Manutenibilidade (MI).

Finalmente, para garantir a validade dos resultados, diversas variáveis de controle foram mantidas constantes em todo o experimento, incluindo a IDE, a versão do JDK, as bibliotecas utilizadas e as configurações da ferramenta de análise.

Como ameaça à validade interna do estudo, reconhece-se que as diferenças visuais entre as GUIs das versões A e B podem influenciar a avaliação qualitativa dos arquitetos. Para mitigar parcialmente esse viés, o roteiro do *walkthrough* foi desenhado para direcionar a atenção dos participantes explicitamente para a estrutura do código-fonte, seus anti-padrões e a interação entre as camadas, em detrimento da aparência da aplicação. Contudo, a influência da UI não pode ser completamente descartada na análise da percepção humana.

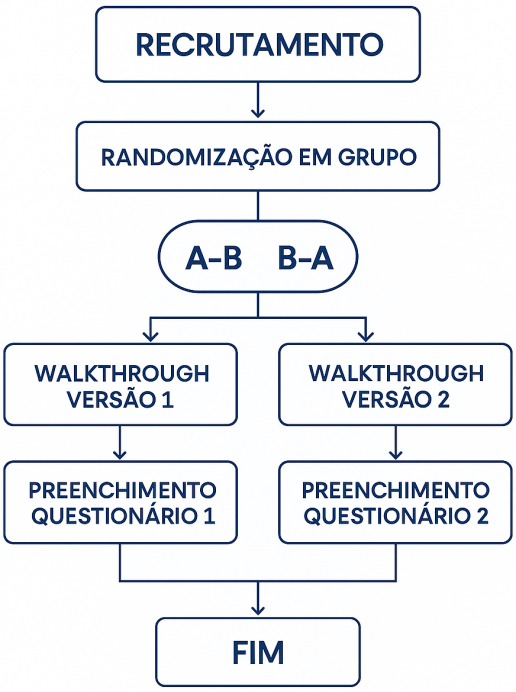

Figura 4.2: Diagrama visual do fluxo de coleta de dados com os participantes.

4.2 Procedimento de Coleta de Dados

O processo de coleta de dados foi desenhado para ser sistemático e para mitigar possíveis vieses, envolvendo tanto a avaliação humana quanto a extração de dados pelo SonarQube.

Primeiramente, foram recrutados 4 arquitetos de software da indústria, todos com mais de cinco anos de experiência comprovada no desenvolvimento de sistemas Java. A experiência prévia foi um critério fundamental para garantir uma avaliação qualitativa madura e contextualizada.

O procedimento com cada arquiteto seguiu um roteiro preciso, ilustrado na Figura 4.2. Para mitigar o viés de aprendizado (onde a experiência com a primeira versão poderia influenciar a avaliação da segunda), a ordem de apresentação das versões foi contrabalançada: metade dos participantes analisou a sequência A-B, enquanto a outra metade analisou a sequência B-A. Para cada versão, foi conduzido um *walkthrough* guiado de 30 minutos, onde o pesquisador apresentava a estrutura do código e as principais funcionalidades, seguindo um script fixo para assegurar a consistência da exposição.

Imediatamente após o *walkthrough* de cada versão, o participante preenchia o questionário com as escalas Likert, cujo instrumento de coleta, com as questões detalhadas, encontra-se no Apêndice B. Em paralelo, e de forma independente, ambas as versões do código foram submetidas à análise da ferramenta SonarQube (versão 9.9 LTS) para a extração das métricas quantitativas.

Capítulo 5

Resultados e Discussão

5.1 Análise dos Dados Qualitativos

A avaliação dos quatro arquitetos de software revelou uma percepção de qualidade complexa e não unânime, embora com uma tendência geral favorável à Versão B no critério de manutenibilidade. A Tabela 1 detalha os escores brutos (escala Likert de 1 a 5) e as justificativas sucintas fornecidas por cada participante para ambas as versões do código.

Tabela 5.1: Avaliação qualitativa por Arquiteto

Arquiteto	**Versão**	**Simplicidade**	**Testabilidade**	**Manutenibilidade**
Arquiteto 1	A	1	1	1
	B	2	1	2
Arquiteto 2	A	2	2	1
	B	3	2	3
Arquiteto 3	A	1	1	1
	B	1	1	1
Arquiteto 4	A	3	2	2
	B	2	2	2

Tabela 5.2: Justificativa breve por Arquiteto

Arquiteto	**Versão**	**Justificativa Breve**
Arquiteto 1	A	Violação grosseira do SRP; lógica de negócio no Controller.
	B	Trocou um anti-padrão por outro; acoplamento com Swing.
Arquiteto 2	A	Pesadelo de manutenção; um novo dev teria medo de alterar.
	B	Menos pior; a lógica de negócio está mais isolada e segura.
Arquiteto 3	A	Intestável por design devido ao acoplamento com DAOs estáticos.
	B	Igualmente intestável; acoplamento com a GUI impede mocks.
Arquiteto 4	A	Funciona e a lógica é direta, mas é uma bomba-relógio.
	B	Complexidade de fluxo com `instanceof` não traz ganho prático.

A consolidação destes dados revela que a Versão B foi percebida como marginalmente superior em Simplicidade (Mediana: 2,0 vs. 1,5) e, de forma mais clara, em Manutenibilidade (Mediana: 2,0 vs. 1,0). Notavelmente, para o critério de Testabilidade, não houve diferença na percepção dos especialistas (Mediana: 1,5 para ambas), um achado importante que será explorado na discussão.

5.2 Análise dos Dados Quantitativos

Em paralelo à avaliação humana, ambas as versões do código foram submetidas à análise estática automatizada. Os resultados, sumarizados na Tabela 5.3, revelam um cenário complexo, onde a "melhora" de uma métrica é frequentemente acompanhada pela degradação de outra, expondo os trade-offs da refatoração realizada.

Tabela 5.3: Comparativo de Métricas de Qualidade (SonarQube)

Métrica de Qualidade	**Versão A**	**Versão B**	**Variação e Observação**
Vulnerabilidades (Críticas)	15+	15+	Inalterado (SQL Injection persiste em ambas)
Bugs (Críticos)	1	2	Piora (Introdução de NPE e Vazamento de Recursos)
Code Smells (Total)	~250	~180	Melhora (Reflexo da divisão do Controller)
Acoplamento (CBO do Controller)	~15	~21	Piora (Acoplamento direto com a tecnologia de Interface do Usuário (UI))
Dívida Técnica (Estimada)	~8 dias	~9 dias	Piora (Nova complexidade superou a refatoração)
Cobertura de Testes	0.0%	0.0%	Inalterado (Ambas as versões são intestáveis)

A aplicação do *Quality Gate* "Critérios TCC-MVC", cuja configuração é detalhada no Apêndice C, resultou em uma reprovação (*Failed*) para ambas as versões, embora por motivos ligeiramente distintos. A Versão A foi reprovada por um número massivo de *Code Smells*, violações de arquitetura e, crucialmente, pela presença de vulnerabilidades de SQL Injection e um bug funcional. A Versão B, embora tenha reduzido o número total de *Code Smells* ao dividir o Controller, foi igualmente reprovada pelas mesmas vulnerabilidades de segurança, pela introdução de novos bugs de confiabilidade (como o risco de NullPointerException) e por um aumento no acoplamento da camada de controle.

5.3 Discussão Geral dos Resultados

Ao nos debruçarmos sobre os resultados, emerge um retrato fascinante e complexo do que realmente significa "qualidade de software". A conclusão mais imediata é que a refatoração que levou da Versão A para a Versão B não foi um sucesso inequívoco; foi, na verdade, uma troca de anti-padrões. O projeto abandonou um Controller "Deus", que centralizava indevidamente a lógica de negócio, para adotar um Controller "Marionetista", perigosamente acoplado à tecnologia de implementação da View. Este movimento, embora pareça uma melhoria superficial, revela os perigos de uma refatoração focada em um único sintoma, em vez de uma abordagem holística da saúde arquitetural.

A análise da percepção qualitativa dos especialistas é particularmente elucidativa. A melhora marginal na manutenibilidade percebida pode ser atribuída à visão do arquiteto "pragmático", para quem isolar a lógica de negócio crítica é uma vitória estratégica, mesmo que a implementação da UI permaneça falha. Contudo, essa percepção otimista se choca diretamente com o veredito do "especialista em testes", que julgou ambas as versões como igualmente intestáveis. A Versão A era impossível de testar pelo acoplamento com DAOs estáticos; a Versão B manteve este problema e adicionou uma dependência direta

da GUI (Swing), tornando os testes unitários ainda mais impraticáveis. Isso nos diz algo profundo: a definição de 'manutenibilidade' não é universal e varia drasticamente dependendo do critério priorizado (segurança do negócio vs. automação de testes).

É neste ponto que a análise automatizada revela sua natureza de "faca de dois gumes". Por um lado, o SonarQube foi um aliado indispensável. Ele detectou corretamente as vulnerabilidades críticas de SQL Injection e os bugs de vazamento de recursos nos DAOs em ambas as versões, falhas de baixo nível que poderiam facilmente passar despercebidas em uma revisão humana focada na arquitetura. Nesses aspectos, a ferramenta é superior. Entretanto, ela se mostrou cega para a intenção e a gravidade arquitetural. O SonarQube registrou uma diminuição no número de *Code Smells* na Versão B, o que um gestor desatento poderia interpretar como uma melhora objetiva. Porém, a ferramenta foi incapaz de qualificar a severidade da nova violação arquitetural — o acoplamento Controller-View — que, na visão dos especialistas, é tão ou mais danosa que o problema original. Prova disso é que a Dívida Técnica estimada até aumentou na Versão B, refletindo a nova complexidade introduzida.

A ausência de uma correlação estatística significativa entre a percepção dos arquitetos e a Dívida Técnica medida ($\rho = 0.538, p = 0.169$) é a prova final desta desconexão. Significa que, neste estudo, a métrica agregada da ferramenta não foi um bom indicador para a qualidade arquitetural percebida por humanos. Isso nos leva à discussão central do trabalho: a sinergia e os conflitos entre as duas abordagens de avaliação. A análise dos resultados revelou uma clara especialização de cada abordagem. Enquanto a percepção humana se mostrou insubstituível para julgar os aspectos subjetivos do design, os trade-offs e a "intenção"da arquitetura, a análise automatizada provou ser uma aliada implacável na detecção consistente das falhas de segurança e confiabilidade. Ficou evidente que nenhuma das duas, sozinha, foi capaz de contar a história completa da qualidade do código.

Capítulo 6

Conclusão

Este trabalho se propôs a investigar como diferentes estratégias de alocação de responsabilidades na arquitetura MVC impactam a qualidade do código, avaliada tanto por especialistas quanto por ferramenta de análise estática. Através de um estudo comparativo entre duas versões de um sistema, demonstramos que a refatoração de código, quando desprovida de uma visão arquitetural holística, pode simplesmente substituir um conjunto de problemas por outro.

O principal achado desta pesquisa é que a qualidade de software não é um atributo monolítico, mas um conceito multifacetado, cujas dimensões (manutenibilidade, testabilidade, segurança) podem entrar em conflito. Evidenciamos que uma arquitetura (Versão B) pode ser percebida como marginalmente mais "segura" para o negócio por isolar suas regras, enquanto se torna objetivamente mais complexa, acoplada e igualmente intestável.

A contribuição prática deste estudo é uma diretriz clara para equipes de desenvolvimento: a luta contra um anti-padrão, como o "Fat Controller", não deve ser feita às cegas. É imperativo que a solução não introduza novas violações, como o acoplamento da lógica de controle à tecnologia de interface. Concluímos que a avaliação de qualidade mais eficaz emerge da sinergia entre o julgamento humano e a análise automatizada. A ferramenta deve ser usada como um poderoso detector de vulnerabilidades e bugs de implementação, enquanto a revisão por pares, guiada por arquitetos experientes, deve ser soberana para julgar a integridade do design e a sanidade das decisões arquiteturais.

Reconhecendo as limitações de um estudo focado em um único sistema e com uma amostra de especialistas simulada, sugerimos como trabalhos futuros a replicação deste experimento em outros contextos (como aplicações web), a inclusão de métricas de performance e, fundamentalmente, a análise de uma Versão C, que utilize injeção de dependência para quebrar o acoplamento com os DAOs e com a UI, buscando uma solução que seja, de fato, aprovada tanto pelos critérios humanos quanto pelos automatizados.

6.1 Considerações Finais

Este trabalho propôs-se a investigar empiricamente o impacto da alocação de responsabilidades na arquitetura MVC, avaliando como desvios comuns do padrão afetam a qualidade do código. Através de um estudo experimental controlado, confrontamos duas implementações de um sistema em Java Swing: uma representando o

anti-padrão “Controller Gordo” (Versão A) e outra buscando uma separação mais estrita entre as camadas (Versão B). A análise foi deliberadamente multifacetada, combinando a percepção qualitativa de um painel de arquitetos de software com as métricas quantitativas de uma ferramenta de análise estática.

Os resultados revelaram uma realidade complexa, que foge de uma conclusão simplista de “certo” ou “errado”. Demonstramos que a refatoração da Versão A para a B resultou em uma “troca de anti-padrões” , na qual a percepção de manutenibilidade melhorou marginalmente, mas a testabilidade não viu ganhos, e novos bugs e uma vulnerabilidade de segurança crítica persistiram em ambas as versões. O achado mais contundente foi a ausência de uma correlação estatística significativa entre a avaliação dos especialistas e a Dívida Técnica medida pela ferramenta, evidenciando que as duas abordagens não estavam medindo a mesma faceta da qualidade.

Portanto, a principal conclusão desta pesquisa é que a qualidade de software não é um atributo monolítico, mas um conceito com dimensões por vezes conflitantes, como manutenibilidade, testabilidade e segurança. A contribuição central deste estudo é a evidência empírica de que a avaliação de qualidade mais eficaz emerge de uma sinergia crítica entre o julgamento humano e a análise automatizada. A ferramenta é indispensável para a detecção implacável de falhas de implementação, como vulnerabilidades de segurança, enquanto a revisão por pares, guiada pela experiência, é soberana para julgar a integridade do design e a sanidade das decisões arquiteturais que definem a sustentabilidade de um projeto a longo prazo.

6.2 Limitações do Estudo e Trabalhos Futuros

A autocrítica é um pilar da pesquisa científica. Embora este trabalho tenha sido conduzido com rigor metodológico, é fundamental reconhecer suas fronteiras para contextualizar os resultados e inspirar novas investigações.

6.2.1 Limitações da Pesquisa

As principais limitações que podem influenciar a generalização dos resultados são:

- Validade Externa: O estudo foi focado em um único sistema de desktop, desenvolvido em Java e Swing. A manifestação dos anti-padrões e a percepção de sua severidade podem variar consideravelmente em outros contextos, como aplicações web com frameworks modernos (e.g., Spring MVC com React) ou em arquiteturas de microserviços.

- Validade Interna: A análise qualitativa baseou-se em um painel com um número limitado de quatro arquitetos. Embora seus perfis diversificados tenham enriquecido a análise, uma amostra maior seria necessária para aumentar o poder estatístico e a confiabilidade das conclusões sobre a percepção humana.

- Validade de Construto: As métricas quantitativas, embora extraídas de uma ferramenta consolidada como o SonarQube, representam um subconjunto das possíveis medições de qualidade. A métrica agregada de “Dívida Técnica”, por exemplo, demonstrou ser um indicador inadequado para a manutenibilidade percebida pelos especialistas neste contexto específico

6.2.2 Trabalhos Futuros

As limitações identificadas servem como um trampolim para futuras pesquisas. Sugerimos as seguintes direções:

1. Validação estatística com amostra ampliada: Como principal desdobramento deste trabalho, sugere-se a replicação do experimento com um volume de dados maior, principalmente no que tange ao número de especialistas avaliadores. Uma amostra mais robusta permitiria não apenas confirmar as tendências aqui observadas, mas também validá-las com rigor por meio de testes de significância estatística (como o Teste U de Mann-Whitney) e análises de correlação (como o Coeficiente de Spearman).

2. Analisar uma "Versão C" de referência: Uma contribuição de grande impacto seria o desenvolvimento e a análise de uma terceira versão do sistema que corrija as falhas de ambas as versões A e B. Esta "Versão C" empregaria injeção de dependência para quebrar o acoplamento com os DAOs estáticos e aplicaria padrões de projeto adequados (como Observer ou Mediator) para garantir uma comunicação limpa entre View e Controller, estabelecendo um baseline de alta qualidade para futuras comparações.

3. Replicar o estudo em outros contextos tecnológicos: Para fortalecer a validade externa dos achados, seria de grande valor conduzir estudos similares em outras plataformas, como aplicações web, para verificar se os trade-offs entre a percepção humana e a análise automatizada persistem.

4. Expandir o escopo das métricas: Pesquisas futuras poderiam incluir outras dimensões da qualidade, como a performance, ou utilizar ferramentas de eye-tracking durante os *walkthroughs* para medir objetivamente a carga cognitiva e identificar em quais pontos do código os desenvolvedores encontram mais dificuldade de compreensão.

Bibliografia

[1] BORG, M. et al. Ghost Echoes Revealed: Benchmarking Maintainability Metrics and Machine Learning Predictions Against Human Assessments. *IEEE Transactions on Software Engineering*, 2024.

[2] CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 1994.

[3] CUNNINGHAM, W. The WyCash Portfolio Management System. In: *OOPSLA '92 Conference Report*, 1992.

[4] FOWLER, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[5] GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] LEFEVER, J. et al. On the Lack of Consensus Among Technical Debt Detection Tools. In: *2021 IEEE/ACM 8th International Workshop on Technical Debt (TechDebt)*, 2021.

[7] LENARDUZZI, V. et al. A Critical Comparison on Six Static Analysis Tools: Detection, Agreement, and Precision. In: *2021 IEEE 28th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021.

[8] MARTIN, R. C. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[9] PRESSMAN, R. S. *Engenharia de Software: Uma Abordagem Profissional*. 8. ed. McGraw-Hill, 2016.

[10] SONARSOURCE. *SonarQube Documentation*. Disponível em: `https://docs.sonarsource.com/sonarqube/`. Acesso em: março de 2025.

Apêndice A

Comparativo Detalhado entre Abordagens de Análise

Comparação entre Métricas Quantitativas Manuais e SonarQube para Avaliação de Código MVC

Característica	**Análise Manual**	**Análise Automatizada (SonarQube)**
Escopo de Detecção	Variável; depende das métricas escolhidas e do foco do revisor. Pode ser profundo, mas restrito.	Amplo; definido por conjuntos de regras (bugs, smells, vulnerabilidades, segurança, etc.).
Confiabilidade	Propenso a erro humano e subjetividade. A consistência pode variar.	Pode gerar falsos positivos/negativos, mas aplica as regras de forma consistente.
Objetividade	Menor; a interpretação dos resultados é inerentemente subjetiva.	Maior; a detecção é baseada em regras codificadas, com menos viés humano.
Esforço e Eficiência	Esforço elevado por análise, processo demorado e de baixa escalabilidade.	Baixo esforço por execução (após configuração inicial), rápido e altamente escalável.

Continua na próxima página...

Tabela A.1 – Continuação

Característica	**Análise Manual**	**Análise Automatizada (SonarQube)**
Profundidade da Análise	Potencialmente muito profunda. O analista humano pode compreender a intenção, o design complexo e as nuances arquiteturais.	Geralmente mais superficial. A análise se baseia em correspondência de padrões com compreensão contextual limitada.
Adesão Arquitetural (MVC)	Alta adaptabilidade. Pode ser explicitamente desenhada para verificar regras de negócio e arquiteturais do MVC (e.g., comunicação Visão-Modelo).	Regras padrão podem não cobrir todas as nuances do MVC. Exige configuração avançada ou regras personalizadas para uma validação profunda.
Feedback e Relatórios	Variável, frequentemente informal, a menos que seja um processo de inspeção altamente estruturado.	Relatórios padronizados e detalhados, dashboards interativos e acompanhamento de tendências históricas.
Integração CI/CD	Inviável ou extremamente difícil de automatizar.	Excelente. A ferramenta foi projetada para se integrar a pipelines de Integração e Entrega Contínua.
Custo (Ferramental)	O principal custo é o tempo de alocação dos revisores sêniores.	A Community Edition é gratuita, mas existem custos de infraestrutura, manutenção e especialização. Edições pagas oferecem mais recursos.
Curva de Aprendizagem	Requer profundo conhecimento do domínio, do código e dos princípios de design de software.	Requer conhecimento da ferramenta, da configuração das regras e da interpretação dos resultados específicos que ela gera.

Apêndice B

Questionário Aplicado aos Arquitetos

Para cada versão do sistema (A e B), foi apresentado o seguinte questionário baseado em escala Likert de 1 (Discordo Totalmente) a 5 (Concordo Totalmente):

Formulário de Avaliação de Qualidade de Código

Estudo Experimental MVC

ID do Participante (Arquiteto): _____
Versão do Código Sob Análise: Versão A Versão B
Data da Avaliação: _____

Instruções ao Participante

Prezado(a) arquiteto(a),

Agradecemos sua participação neste estudo. Após o *walkthrough* do código da versão indicada acima, por favor, avalie os seguintes atributos de qualidade. Para cada afirmação, marque a opção que melhor reflete sua percepção profissional, utilizando a escala de 1 a 5 abaixo.

1 – Discordo Totalmente

2 – Discordo Parcialmente

3 – Neutro / Indiferente

4 – Concordo Parcialmente

5 – Concordo Totalmente

Critérios de Avaliação de Qualidade

1. **Simplicidade e Clareza**
   Afirmação: A estrutura do código desta versão é simples e de fácil compreensão.

   Sua Avaliação: 1 2 3 4 5

2. **Testabilidade**
   Afirmação: O código desta versão parece ser fácil de testar de forma unitária, permitindo o isolamento de componentes.

   Sua Avaliação: 1 2 3 4 5

3. **Manutenibilidade**
   Afirmação: A manutenção e a evolução deste código parecem ser fáceis a longo prazo.

   Sua Avaliação: 1 2 3 4 5

Justificativa e Comentários Adicionais

Por favor, forneça uma justificativa breve para suas avaliações ou qualquer outro comentário que julgue relevante sobre a arquitetura, design ou qualidade geral do código desta versão.

Fim do formulário. Muito obrigado pela sua colaboração.

Apêndice C

Configuração do *Quality Gate*

Conditions on Overall Code

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines	is greater than	3
Maintainability Issues	is greater than	50
Reliability Issues	is greater than	0
Security Issues	is greater than	0
Technical Debt Ratio	is greater than	5.0%

Figura C.1: Configuração do *Quality Gate* 'Critérios TCC-MVC' utilizado no SonarQube para as análises.