UNIVERSIDADE FEDERAL DO CEARÁ

***CAMPUS* SOBRAL**

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

MESTRADO ACADÊMICO EM ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

RAIMUNDO ALAN FREIRE MOREIRA

ENTRE MÉTRICAS E PERCEPÇÃO: USO DE APRENDIZADO DE MÁQUINA E LLMS NA DETECÇÃO E REFATORAÇÃO DE CODE SMELLS

SOBRAL

2025

RAIMUNDO ALAN FREIRE MOREIRA

ENTRE MÉTRICAS E PERCEPÇÃO: USO DE APRENDIZADO DE MÁQUINA E LLMS NA DETECÇÃO E REFATORAÇÃO DE CODE SMELLS

Dissertação apresentada ao Curso de Mestrado Acadêmico em Engenharia Elétrica e de Computação do Programa de Pós-Graduação em Engenharia Elétrica e de Computação do *Campus* Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Engenharia Elétrica e de Computação. Área de Concentração: Sistemas de Informação

Orientador: Prof. Dr. Fischer Jônatas Ferreira

Coorientador: Prof. Dr. Gustavo Andrade do Vale

SOBRAL

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

M839e Moreira, Raimundo Alan Freire.
Entre Métricas e Percepção: Uso de Aprendizado de Máquina e LLMs na Detecção e Refatoração de Code Smells / Raimundo Alan Freire Moreira. – 2025.
132 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Campus de Sobral, Programa de Pós-Graduação em Engenharia Elétrica e de Computação, Sobral, 2025.
Orientação: Prof. Dr. Fischer Jônatas Ferreira.
Coorientação: Prof. Dr. Gustavo Andrade do Vale.

1. qualidade de código. 2. code smells. 3. aprendizado de máquina. 4. modelos de linguagem de grande porte. 5. ferramentas de pesquisa. I. Título.

CDD 621.3

RAIMUNDO ALAN FREIRE MOREIRA

ENTRE MÉTRICAS E PERCEPÇÃO: USO DE APRENDIZADO DE MÁQUINA E LLMS NA DETECÇÃO E REFATORAÇÃO DE CODE SMELLS

Dissertação apresentada ao Curso de Mestrado Acadêmico em Engenharia Elétrica e de Computação do Programa de Pós-Graduação em Engenharia Elétrica e de Computação do *Campus* Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Engenharia Elétrica e de Computação. Área de Concentração: Sistemas de Informação

Aprovada em: 30 de setembro de 2025

BANCA EXAMINADORA

Prof. Dr. Fischer Jônatas Ferreira (Orientador)
Universidade Federal de Itajubá (UNIFEI)

Prof. Dr. Gustavo Andrade do Vale (Coorientador)
Universidade Federal de Minas Gerais (UFMG)

Prof. Dr. Evilásio Costa Júnior
Universidade Federal do Ceará (UFC)

Prof. Dr. Mauricio Ronny de Almeida Souza
Universidade Federal de Lavras (UFLA)

Dedico este trabalho à minha família, fonte de amor, força e inspiração em todos os momentos. À minha esposa e filha, por serem meu alicerce diário, e à minha mãe, pelo exemplo de perseverança e dedicação. Ao meu pai, em memória, cuja presença permanece viva em minhas lembranças e em cada conquista alcançada.

AGRADECIMENTOS

A Deus, pela força, proteção e sabedoria em cada etapa desta caminhada. Sem a fé e a confiança depositadas em sua presença, os desafios enfrentados ao longo desta jornada teriam sido ainda maiores. Foi ele quem renovou minhas energias nos momentos de incerteza e me deu serenidade para perseverar diante das dificuldades.

À minha mãe, pelo apoio inestimável em todas as etapas da minha vida e por me ensinar, desde cedo, que a educação é o bem mais precioso que alguém pode conquistar. Seu exemplo de dedicação, resiliência e amor sempre foi a base sobre a qual construí meus sonhos.

À minha esposa, Elisângela, pelo amor incondicional, pelo companheirismo em todos os momentos e pela paciência nas inúmeras vezes em que precisei estar ausente. Sua força para sustentar as responsabilidades do cotidiano e sua confiança em mim foram determinantes para que eu tivesse tranquilidade e segurança para avançar. Este trabalho é também fruto da sua dedicação silenciosa e de sua presença constante ao meu lado.

À minha filha, Lys, pelo carinho imenso e pela capacidade de compreender, mesmo tão jovem, as ausências necessárias em função dos estudos. O amor incondicional que recebi dela foi combustível diário para continuar, mesmo diante das maiores dificuldades. A ela, dedico este esforço, na esperança de que um dia veja neste trabalho um exemplo de perseverança e valorização da educação.

Ao Prof. Dr. Fischer Jônatas Ferreira, pela orientação segura e sempre pautada pela paciência, pelo rigor científico e pela generosidade em compartilhar seus conhecimentos. Sua confiança em meu trabalho e sua capacidade de guiar este estudo com clareza foram fundamentais para que eu alcançasse este resultado.

Ao Prof. Dr. Gustavo do Vale, pela coorientação dedicada e pelo olhar atento, que acrescentou contribuições valiosas ao longo desta pesquisa. Sua sensibilidade acadêmica e postura colaborativa enriqueceram não apenas este estudo, mas também minha formação como pesquisador.

Ao Prof. Dr. Márcio Amora, pelo incentivo e pela conhecimento transmitido durante a disciplina de Inteligência Computacional Aplicada.

A todos os professores que fizeram parte de minha formação, por transmitirem mais do que conhecimento técnico. A cada um deles, agradeço pelos valores éticos, pela inspiração, pela postura humana e pelo afeto, aspectos que transcendem os conteúdos e que se tornam exemplos de vida.

Aos colegas de trabalho e ao Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE), pelo apoio institucional e pela compreensão nas adaptações necessárias para conciliar as demandas profissionais com as acadêmicas. A flexibilidade concedida e o incentivo recebido foram indispensáveis para que eu pudesse me dedicar ao mestrado sem abrir mão das minhas responsabilidades.

Aos colegas de mestrado, especialmente Geraldo Martins, Rhuan Nunes, Expedito Magalhães, Marcelo Estevão e Iago Magalhães, pela amizade construída ao longo desses anos. As discussões em sala, as trocas de experiências e até os momentos de descontração nos corredores e no restaurante da UFC foram fundamentais para tornar a jornada mais leve e significativa.

A todos aqueles que, direta ou indiretamente, contribuíram para esta conquista, deixo registrado meu profundo reconhecimento. Cada gesto de apoio, cada palavra de incentivo e cada demonstração de confiança teve importância singular nesta trajetória.

Aos membros da banca examinadora, Prof. Dr. Evilásio Costa Júnior e Prof. Dr. Mauricio Ronny de Almeida Souza, pela generosidade em dedicar parte de seu tempo à leitura, análise e apreciação deste trabalho.

“Qualquer tolo pode escrever código que um computador entende. Bons programadores escrevem código que humanos podem entender.”

(Martin Fowler)

RESUMO

A qualidade de software é determinante para a manutenibilidade, legibilidade, testabilidade e evolução de sistemas, sendo que grande parte do custo de manutenção está associada à compreensão de código existente. Nesse contexto, os *code smells* surgem como indícios de problemas estruturais ou estilísticos que degradam a arquitetura e alimentam o débito técnico. Embora detectores de *code smells* baseados em métricas sejam amplamente utilizados, eles enfrentam limites de interpretabilidade e generalização. Algoritmos de aprendizado de máquina (*machine learning* – ML) e modelos de linguagem de grande porte (*large language models* – LLMs) ampliam as possibilidades de automação na detecção de *code smells*, no entanto, podem introduzir anomalias. Esta dissertação investiga como técnicas de inteligência artificial podem apoiar a melhoria da qualidade de software, conectando sinais objetivos de ML à percepção humana em refatorações realizadas por LLMs. O estudo foi conduzido em três etapas. Na primeira, cinco classificadores supervisionados foram aplicados a quatro *code smells* nos sistemas do *Qualitas Corpus*. Modelos baseados em árvores performaram melhor e superaram 96% de acurácia estabelecendo *baselines* replicáveis. Na segunda etapa, foi desenvolvida a ferramenta **TwinCode**, para experimentos empíricos com comparação lado a lado de trechos de código e questionários estruturados, validada com 12 participantes que apontaram alta usabilidade e consistência. Na terceira etapa, investigou-se a percepção de desenvolvedores sobre refatorações com LLMs, por meio de comparações cegas e entrevistas. Em 97% das escolhas os entrevistados escolheram as versões refatoradas e associaram a escolha a ganhos de legibilidade, modularidade e manutenibilidade, ainda que com alertas para riscos de *over-engineering*. Os resultados demonstram que: (i) algoritmos de ML oferecem sinais robustos para detecção de *code smells*, (ii) TwinCode contribui para padronização metodológica de estudos empíricos e (iii) LLMs possuem potencial de apoiar refatorações quando aplicados com supervisão crítica. Contudo, esta dissertação integra métricas objetivas e julgamentos humanos, oferecendo evidências aplicáveis à pesquisa acadêmica e à prática profissional de desenvolvimento de software.

Palavras-chave: qualidade de código; *code smells*; aprendizado de máquina; modelos de linguagem de grande porte; ferramentas para pesquisa empírica.

ABSTRACT

Software quality is essential for maintainability, readability, testability, and the long-term evolution of systems, with a large portion of maintenance costs related to understanding existing code. In this context, code smells emerge as indicators of structural or stylistic issues that degrade architecture and introduce technical debt. While metric-based detectors are widely used, they face limits of interpretability and generalization, whereas recent advances with machine learning (ML) algorithms and large language models (LLMs) expand automation possibilities but might introduce anomalies. This dissertation investigates how artificial intelligence techniques can support software quality improvement by connecting objective ML signals with human perception in refactorings produced by LLMs. The study was organized in three stages. In the first, five supervised classifiers were applied to four smells identified in systems from Qualitas Corpus. Tree-based models performed better and surpassing accuracy of 96% establishing replicable baselines. In the second stage, we propose **TwinCode** a open source tool developed to support empirical experiments with side by side code comparison and structured questionnaires. The tool was validated with 12 participants who reported high usability and consistency. In the third stage, we examined the developers' perceptions of LLM-based refactorings through blind comparisons and interviews. In 97% of choices, intervieews chose the refactored versions and associated their choice with improvements in readability, modularity, and maintainability, even though they reported over-engineering issues. Our results demonstrate that: (i) ML algorithms provide robust signals for code smell detection, (ii) TwinCode contributes to the methodological standardization of empirical studies, and (iii) LLMs have potential to support refactorings when applied with critical supervision.Taken together, this dissertation integrates objective metrics and human judgments, offering evidence applicable to both academical and professional software development.

Keywords: code quality; code smells; machine learning; large language models; empirical research tools.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AI	*Artificial Intelligence*
CSS	*Cascading Style Sheets*
DP	Desvio Padrão
HTML	*HyperText Markup Language*
IA	Inteligência Artificial
IHC	Interação Humano-Computador
LLMs	*Large Language Models*
ML	*Machine Learning*
MLP	*Multilayer Perceptron*
MVC	*Model*, *View* e *Controller*
PHP	PHP: *Hypertext Preprocessor*
PLN	Processamento de Linguagem Natural
SaaS	*Software as a Service*
SVM	*Support Vector Machines*
TCLE	Termo de Consentimento Livre e Esclarecido
UX	*User Experience*

SUMÁRIO

1 INTRODUÇÃO

A qualidade de software é determinante para a manutenibilidade, legibilidade, testabilidade e evolução de sistemas ao longo do seu ciclo de vida; estima-se que até 80% do custo total de um sistema esteja associado à manutenção, em grande parte dedicada à compreensão de código existente (KRASNER, 2021; RAHMAN *et al.*, 2024). Nesse cenário, *code smells* surgem como indícios de problemas estruturais/estilísticos que, embora não inviabilizem a execução, degradam a arquitetura e alimentam o débito técnico, afetando a sustentabilidade de projetos (FOWLER, 2018; YAMASHITA; MOONEN, 2013; OUNI *et al.*, 2017; PALOMBA *et al.*, 2018). Detectores baseados em métricas e análise estática têm sido amplamente utilizados (FONTANA *et al.*, 2016), enquanto avanços recentes incorporam técnicas de aprendizado de máquina (do inglês *Machine Learning* (ML)). Modelos baseados em árvores (ABDOU; DARWISH, 2024) têm alcançado alta acurácia da detecção de *code smells*. Por outro lado, modelos de linguagem de grande porte (do inglês *Large Language Models* (LLMs)) têm sido eficazes e mantido boas práticas quando refatorando artefatos de código (CHEN *et al.*, 2021; ROZIERE *et al.*, 2023; ACHIAM *et al.*, 2023). Ao mesmo tempo, estudos apontam limites e riscos, como a geração de novos *smells* e dificuldades de consistência semântica em refatorações complexas (VELASCO *et al.*, 2025; BÖRSTLER *et al.*, 2023). Para minimizar essas anomalias, alguns pesquisadores destacam o papel de engenharia de *prompt* para melhorar a qualidade das refatorações e soluções propostas por LLMs (WHITE *et al.*, 2023).

Apesar do aparato técnico disponível, persiste uma tensão entre marcações algorítmicas e julgamentos humanos, pois desenvolvedores frequentemente discordam de detectores automáticos por avaliarem a qualidade de forma mais ampla (clareza semântica, consistência estilística, esforço cognitivo) (BUSE; WEIMER, 2009; POSNETT *et al.*, 2011; BINKLEY *et al.*, 2013). Rahman *et al.* (2024), por meio de uma revisão sistemática da literatura, apontam desafios de interpretabilidade e generalização em contextos heterogêneos. No campo da refatoração com LLMs, a literatura carece de evidências sobre *como* desenvolvedores percebem o código produzido e *quais* critérios humanos orientam tais julgamentos. Adicionalmente, há fragilidades metodológicas, pois estudos frequentemente recorrem a soluções *ad hoc* para coleta/comparação de versões, o que prejudica padronização e replicabilidade entre investigações (SANTOS; GEROSA, 2018).

Diante desse quadro, há oportunidade para uma abordagem integrada que conecte (i) sinais objetivos de ML na detecção de *smells*, (ii) protocolos/infraestruturas que viabilizem

estudos empíricos replicáveis (comparações cegas com coleta estruturada) e (iii) a análise da percepção de desenvolvedores sobre refatorações por LLMs. Tal integração busca, simultaneamente, reduzir o hiato entre métricas e julgamento humano, orientar a adoção responsável de LLMs em fluxos de trabalho (potencializando benefícios e mitigando riscos (VELASCO *et al.*, 2025; BÖRSTLER *et al.*, 2023)) e prover evidências úteis tanto para a comunidade científica quanto para a prática profissional, em linha com a evolução recente da área de engenharia de software e computação como um todo (FONTANA *et al.*, 2016; ABDOU; DARWISH, 2024; CHEN *et al.*, 2021; ROZIERE *et al.*, 2023; ACHIAM *et al.*, 2023; WHITE *et al.*, 2023).

1.1 Objetivo da Dissertação

O objetivo principal é **investigar *como* técnicas de inteligência artificial podem apoiar a melhoria da qualidade de software**, conectando sinais objetivos de detecção de *code smells* a julgamentos humanos sobre qualidade de código. Para alcançar este objetivo, definimos dois objetivos específicos:

- **OE1 — Detecção de *Code Smells* com ML.** Investigar, de forma sistemática, o desempenho de classificadores supervisionados na identificação de quatro *code smells* (*Data Class*, *God Class*, *Feature Envy* e *Long Method*).
- **OE2 — Percepção Humana de Qualidade de Software e Refatorações com LLMs.** Investigar, por meio de *comparações cegas* e entrevistas, como desenvolvedores julgam versões originais versus refatoradas por LLMs e *quais* critérios orientam tais julgamentos (p. ex., legibilidade, modularidade, manutenibilidade).

O **OE1** é motivado pelo papel central de detectores baseados em métricas e ML na identificação de indícios de baixa qualidade, bem como pelos desafios de generalização e interpretabilidade reportados na literatura (FONTANA *et al.*, 2016; RAHMAN *et al.*, 2024; ABDOU; DARWISH, 2024). O **OE2** decorre da necessidade de compreender a *qualidade percebida* do código gerado/refatorado por LLMs, dada a tensão conhecida entre marcações automáticas e julgamentos humanos (BUSE; WEIMER, 2009; POSNETT *et al.*, 2011; BINKLEY *et al.*, 2013), bem como dos riscos e limites na prática de refatoração automática e do papel da engenharia de *prompt* adequado (WHITE *et al.*, 2023; CHEN *et al.*, 2021; ROZIERE *et al.*, 2023; ACHIAM *et al.*, 2023). Para viabilizar estudos empíricos *replicáveis* nesse segundo

objetivo específico, desenvolvemos a ferramenta **TwinCode**, que integra comparação lado a lado e questionários estruturados, reduzindo soluções *ad hoc* e favorecendo padronização (SANTOS; GEROSA, 2018). A Figura 1 mostra a relação entre os objetivos do estudo.

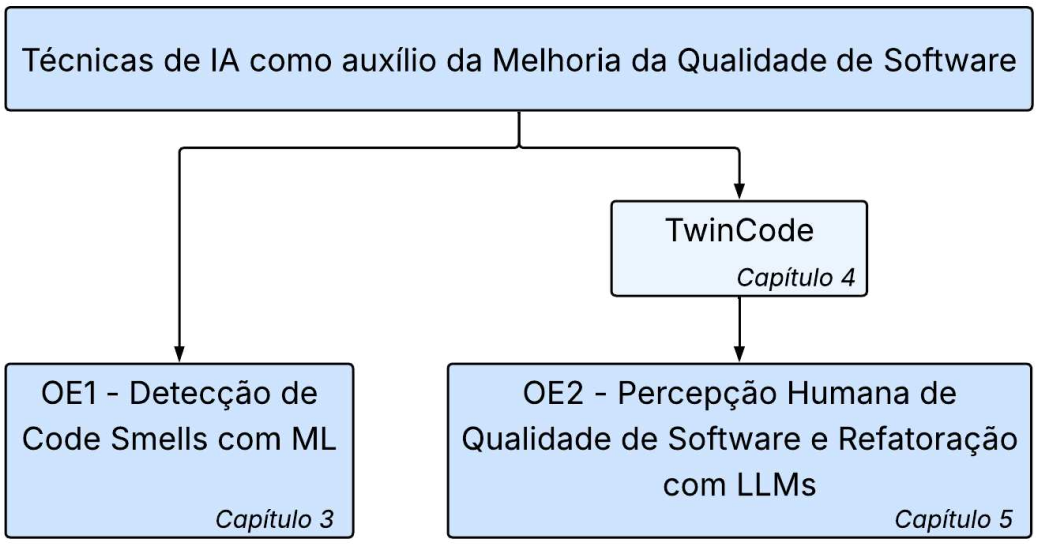


Figura 1 – Visão geral da dissertação
Fonte: Elaborado pelo autor (2025)

1.2 Contribuições

Nesta dissertação são apresentadas diversas contribuições que utilizam técnicas de inteligência artificial como auxílio na melhoria da qualidade de software. A seguir, são mencionadas e explicadas as principais extensões dessas contribuições.

1. **Detecção de *code smells* com ML.** Foi criado um benchmark sistemático de cinco classificadores supervisionados (Árvore de Decisão, Floresta Aleatória, *Gradient Boosting*, *Support Vector Machines* (SVM) e *Multilayer Perceptron* (MLP)) para quatro *smells* (*Data Class*, *God Class*, *Feature Envy*, *Long Method*) com dados derivados do *Qualitas Corpus* de Tempero *et al.* (2010) e comparação com/sem validação cruzada. Com resultado, obtive-se acurácias entre **89,7%** e **99,2%**, com picos de **96,8%** (*Data Class*), **96,3%** (*God Class*), **98,4%** (*Feature Envy*) e **99,2%** (*Long Method*). Modelos baseados em árvores mostraram desempenho consistentemente superior. Diferenças entre cenários com/sem validação cruzada não foram estatisticamente significativas. Dentre as principais contribuições, destaca-se o estabelecimento de *baselines* fortes e protocolo comparativo

replicável para detecção de *code smells*. E a **publicação** do estudo nos anais da *XXVII Ibero-American Conference on Software Engineering (CIbSE 2024) - (MOREIRA et al., 2024)*.

2. **Ferramenta *TwinCode*.** Foi realizado o projeto e implementação de uma ferramenta para comparação *lado a lado* de artefatos de código com realce de sintaxe, numeração de linhas e questionários por comparação. Desenvolvida em PHP: *Hypertext Preprocessor* (PHP)/Laravel para *backend* e a biblioteca Javascript React junto com TailwindCSS para estilização das páginas. Realizou-se uma avaliação exploratória com **12 participantes**. Como resultados, obteve-se boa aceitação do núcleo de inspeção (comparação, *syntax highlighting*, numeração); **consistência interna** elevada e indicadores favoráveis de potencial de adoção (p. ex., média $\approx 4{,}42$ para uso acadêmico em escala de 1–5). O estudo também mostrou oportunidades priorizadas de melhoria no fluxo pares–questionários e em funcionalidades auxiliares (exportação, filtros, versionamento). A ferramenta foi **registrada no INPI** (BR512025003573-0). Dentre as contribuições do estudo destaca-se a disponibilização do código-fonte da ferramenta que busca padronizar a coleta de evidências em estudos de comparação de código e **viabiliza replicabilidade** metodológica.

3. **Percepção de Desenvolvedores sobre Qualidade do Software e Refatorações com LLMs.** Foi elaborado uma investigação as cegas e randomizada com **7** desenvolvedores, **5** pares por entrevistado (total **35** julgamentos), amostrados de um conjunto de **80** artefatos (12 sistemas do *Qualitas Corpus*). Cada par continha o código original e o código com refatorações produzidas pelo modelo *Qwen2.5-Max*. Como resultado, **34 de 35** escolhas ($\approx 97{,}1\%$) favoreceram as versões refatoradas; critérios centrais de decisão incluíram *legibilidade*, *modularidade* e *manutenibilidade*. A revelação posterior da autoria dos códigos refatorados por LLM não alterou as opiniões dos entrevistados, pois a escolha veio por meio de análise técnica. Os entrevistados mencionaram riscos de *over-engineering* e perda de contexto em casos específicos quanto ao uso de LLMs para refatoração de código. Como contribuições, pode-se mencionar evidência empírica exploratória sobre a **aceitação e critérios humanos** aplicados a refatorações por LLMs e **insumos práticos** para adoção responsável dessas ferramentas em fluxos de desenvolvimento de software.

Como pode-se ver, os três estudos se complementam. A detecção baseada em ML provê **sinais objetivos** robustos. O estudo com LLMs revela **preferências e critérios humanos**

de qualidade e a *TwinCode* conecta ambos por meio de um **instrumento replicável**. Em conjunto, o estudo contribui para a área ao (i) consolidar *baselines* e protocolos para detecção de *codes smells*; (ii) oferecer um **artefato/ferramenta** que padroniza e acelera a pesquisa empírica; e (iii) produzir **evidências centradas no desenvolvedor** sobre refatorações assistidas por LLM, orientando pesquisadores, construtores de ferramentas e desenvolvedores rumo a uma adoção mais informada e responsável de IA na melhoria da qualidade de software.

1.3 Estrutura do Trabalho

Os restante deste trabalho está organizado em cinco capítulos. O **Capítulo 2** consolida conceitos e trabalhos relacionados sobre qualidade de código, *code smells*, métricas de software, técnicas de ML aplicadas à detecção e o papel de LLMs na refatoração, além de discutir lacunas que motivam este estudo. O **Capítulo 3** detalha a formulação do problema de detecção de code smells, o conjunto de *smells* investigados e os classificadores supervisionados considerados, apresentando o delineamento comparativo dos resultados obtidos, bem como os resultados alcançados em estudos anteriores. O **Capítulo 4** descreve a TwinCode desenvolvida para comparação *lado a lado* de trechos de código com coleta estruturada via questionários, sua arquitetura e o papel da ferramenta como infraestrutura para estudos empíricos replicáveis, bem como a avaliação da ferramenta. O **Capítulo 5** apresenta o estudo baseado em comparações cegas e entrevistas, conduzido para compreender como desenvolvedores julgam versões originais *versus* refatoradas por LLMs e quais critérios orientam tais julgamentos. Por fim, o **Capítulo 6** sintetiza os resultados, discute implicações para a prática e para a pesquisa e apresenta direções de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os fundamentos que embasam esta pesquisa. A Seção 2.1 apresenta conceitos básicos e históricos sobre *code smells* e métricas de software. A Seção 2.2 apresenta histórico sobre qualidade de software. A Seção 2.3 aprofunda-se em práticas de refatoração de código. A Seção 2.4 apresenta técnicas de aprendizado de máquina. A Seção 2.5 apresenta uma abordagem sobre normalização e validação de dados. A Seção 2.6 discute sobre ferramentas e usabilidade. A Seção 2.7 trata da percepção humana sobre qualidade de software. Por fim, a Seção 2.8 analisa a inserção das LLMs em estudos na área de engenharia de software.

2.1 *Code Smells* e Métricas de Software

O termo *code smell* foi introduzido por Riel (1996) e popularizado por Fowler (1999) para designar indícios de problemas estruturais ou estilísticos em um código-fonte que, embora não representem defeitos funcionais imediatos, podem comprometer sua qualidade a longo prazo. Esses indícios atuam como "sinais de alerta" que sugerem a necessidade de refatoração, uma vez que estão frequentemente associados à degradação da manutenibilidade, à perda de legibilidade e ao aumento da complexidade (BROWN *et al.*, 1998; YAMASHITA; COUNSELL, 2013). Dessa forma, os *code smells* não são erros de compilação ou execução, mas sintomas de design pobre que aumentam os custos de manutenção e elevam o risco de falhas futuras.

Entre os diversos tipos catalogados por (FOWLER, 1999), esta pesquisa concentra-se em quatro dos mais recorrentes e estudados: (i) **God Class**, caracterizada por classes excessivamente grandes, com múltiplas responsabilidades e baixo nível de coesão (FONTANA *et al.*, 2016); (ii) **Data Class**, definida por classes que funcionam apenas como contêineres de dados, apresentando atributos públicos ou *getters/setters* sem encapsulamento adequado (FONTANA *et al.*, 2016; DEWANGAN *et al.*, 2021); (iii) **Feature Envy**, *smell* em nível de método que acessa mais atributos de outras classes do que da sua própria, revelando alto acoplamento e violação de encapsulamento (BROWN *et al.*, 1998; FONTANA *et al.*, 2016); e (iv) **Long Method**, que ocorre quando um método é demasiadamente extenso, combinando múltiplas funcionalidades e prejudicando a legibilidade (MCCONNELL, 2004; DEWANGAN *et al.*, 2021). Esses quatro *smells* foram selecionados pela literatura devido à sua alta incidência em sistemas reais e ao impacto negativo significativo na qualidade do software (FONTANA *et al.*, 2016; ABDOU; DARWISH, 2024).

A detecção de *smells* está intimamente ligada ao uso de **métricas de software**, que oferecem uma base quantitativa para avaliação objetiva de atributos internos do código. Métricas são definidas como medidas que capturam propriedades do software em diferentes níveis de granularidade, tais como método, classe, pacote e projeto (FONTANA *et al.*, 2016). Entre as mais utilizadas destacam-se: (i) *complexidade ciclomática* (MCCABE, 1976), que quantifica a quantidade de caminhos independentes em um método, servindo como indicador de dificuldade de teste e compreensão; (ii) *coesão* e *acoplamento*, métricas fundamentais da qualidade orientada a objetos propostas por Chidamber e Kemerer (1994), que medem respectivamente o grau de inter-relação interna de uma classe e sua dependência de outras classes; (iii) *tamanho de métodos e classes*, associado ao número de linhas de código e de atributos; (iv) índices compostos como o *Maintainability Index*, que agregam múltiplas métricas para inferir a facilidade de manutenção (COLEMAN *et al.*, 1994).

Embora as métricas forneçam evidências objetivas, diversos estudos apontam que sua interpretação nem sempre é trivial e pode divergir da percepção de desenvolvedores sobre qualidade (BUSE; WEIMER, 2009; POSNETT *et al.*, 2011). Essa tensão entre medidas automáticas e avaliação humana justifica a necessidade de abordagens empíricas que combinem análise quantitativa e julgamento subjetivo, como proposto neste trabalho.

Assim, a compreensão de *code smells* e métricas de software constitui a base teórica que sustenta a presente pesquisa, permitindo tanto a avaliação algorítmica por meio de técnicas de aprendizado de máquina quanto a investigação da percepção humana em relação à qualidade de código refatorado.

2.2 Qualidade de Software

A avaliação da qualidade de software é um tema muito importante em Engenharia de Software, buscando oferecer parâmetros objetivos e subjetivos que orientem tanto o desenvolvimento quanto a manutenção de sistemas. Diferentes modelos foram propostos ao longo da história para estruturar a noção de qualidade, traduzindo-a em dimensões mensuráveis e relacionadas a atributos internos e externos do produto (MCCALL *et al.*, 1977; BOEHM, 1976; PRESSMAN *et al.*, 1995; KITCHENHAM; PFLEEGER, 1996). Os primeiros esforços sistematizados surgiram no final da década de 1970. McCall *et al.* (1977) propuseram um modelo que agrupava atributos em três categorias: *fatores de produto*, *critérios de qualidade* e *métricas*, com ênfase em características como confiabilidade, manutenibilidade e eficiência. Boehm *et*

al. (1976) sugeriram um modelo hierárquico que organizava a qualidade em atributos de alto nível (como utilidade e portabilidade) e métricas mais específicas, estabelecendo a relação entre requisitos de usuários e propriedades técnicas do software.

Posteriormente, com a consolidação da normalização internacional, foi publicada a ISO/IEC 9126 (ISO/IEC, 2001), que se tornou referência para a avaliação de qualidade de produto de software. Esse modelo definiu seis características principais: funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade, cada uma delas desdobrada em subcaracterísticas que permitiam operacionalizar a avaliação. Apesar de sua relevância, a norma recebeu críticas pela ausência de diretrizes claras para definição de métricas específicas, o que dificulta sua aplicação prática em cenários reais (KITCHENHAM, 1996; WASHIZAKI *et al.*, 2004). A evolução natural desse esforço resultou no ISO/IEC 25010, publicado em 2011, que ampliou e refinou a estrutura anterior. O novo modelo passou a contemplar oito características: adequação funcional, desempenho e eficiência, compatibilidade, usabilidade, confiabilidade, segurança, manutenibilidade e portabilidade. Além disso, introduziu a distinção entre **qualidade de produto** e **qualidade em uso**, reconhecendo que a experiência do usuário final constitui parte essencial da avaliação (ISO/IEC, 2011). Essa abordagem responde a críticas anteriores ao incorporar explicitamente dimensões de percepção e contexto de uso.

No âmbito da pesquisa em qualidade de código, tais modelos são fundamentais para situar métricas e indicadores empregados em análises automatizadas. Métricas como complexidade ciclomática (MCCABE, 1976), acoplamento e coesão (CHIDAMBER; KEMERER, 1994), bem como índices compostos como o *Maintainability Index* (COLEMAN *et al.*, 1994), podem ser mapeadas a dimensões dos modelos ISO, particularmente manutenibilidade, eficiência e confiabilidade. Por outro lado, aspectos subjetivos como legibilidade e clareza, apontados por estudos empíricos como os de Posnett *et al.* (2011) e Binkley *et al.* (2013), dialogam com as características de usabilidade e qualidade em uso.

Em síntese, os modelos de qualidade de software não se limitam a organizar atributos em categorias conceituais: eles possibilitam estabelecer conexões entre medições quantitativas, obtidas a partir de métricas, e interpretações qualitativas realizadas por desenvolvedores e usuários. Nesse cenário, o ISO/IEC 25010 destaca-se por oferecer um quadro de referência que articula dimensões objetivas, como manutenibilidade e eficiência, com aspectos subjetivos, como clareza e usabilidade, favorecendo uma visão mais integrada da qualidade do código.

2.3 Refatoração de Código

A refatoração de código é definida como o processo sistemático de reestruturar o código-fonte existente de um sistema de software sem alterar seu comportamento externo (FOWLER, 1999). O objetivo é melhorar atributos internos do código, tais como legibilidade, simplicidade, manutenibilidade e extensibilidade, reduzindo a complexidade acidental que tende a se acumular durante o ciclo de vida do software. Assim, enquanto a correção de defeitos visa restaurar a funcionalidade, a refatoração concentra-se em melhorar a qualidade estrutural e não funcional do sistema. Historicamente, a noção de refatoração remonta a trabalhos de Opdyke (1992), que introduziu o conceito em sua tese de doutorado, propondo operações de transformação (*refactorings*) para melhorar a estrutura interna de sistemas orientados a objetos. Posteriormente, Fowler (1999) sistematizou e popularizou um catálogo de 22 tipos de *code smells* acompanhados de refatorações correspondentes, como *Extract Method*, *Move Method*, *Encapsulate Field* e *Replace Conditional with Polymorphism*. Essa sistematização consolidou a refatoração como prática na engenharia de software, integrada a metodologias ágeis e práticas de desenvolvimento como *Continuous Integration* e *Test-Driven Development* (BECK, 2003).

Métodos de refatoração podem ser classificadas em cinco principais categorias: (i) refatorações de extração e decomposição, como a criação de métodos ou classes menores; (ii) refatorações de movimentação, que reposicionam atributos ou métodos em classes mais adequadas; (iii) refatorações de encapsulamento, que melhoram o controle de acesso a dados; (iv) refatorações de simplificação de expressões e estruturas condicionais; (v) refatorações de herança, que reorganizam hierarquias de classes para melhorar coesão e reduzir acoplamento (FOWLER, 1999; MENS; TOURWÉ, 2004).

Diversos estudos têm ressaltado que a refatoração não se limita a um exercício de "embelezamento" do código, mas atua como um mecanismo estratégico de contenção do débito técnico e de preservação da arquitetura ao longo do tempo (YAMASHITA; COUNSELL, 2013). Essa perspectiva ganha força quando se observa que a presença de *smells* está frequentemente associada a falhas futuras e a dificuldades de compreensão. Nesse sentido, a refatoração funciona como uma espécie de "manutenção preventiva", capaz de reduzir riscos e sustentar a evolução do sistema (SAHIN *et al.*, 2014). Evidências empíricas reforçam esse papel: Ouni *et al.* (2016) e Hilmi *et al.* (2023) apontam que a remoção de *smells* contribui não apenas para diminuir a incidência de defeitos, mas também para ampliar a produtividade de desenvolvedores, que passam a interagir com um código mais legível, modular e previsível. No contexto desta pesquisa,

o uso de algoritmos de ML e LLMs buscam apoiar esse processo de refatoração, fornecendo alternativas automáticas que podem ser comparadas à percepção crítica de programadores.

Embora tradicionalmente realizada de forma manual, a refatoração tem sido progressivamente apoiada por ferramentas automatizadas, tais como Eclipse, IntelliJ IDEA e NetBeans, que incorporam catálogos de refatorações seguras e auxiliam no processo. Contudo, essas ferramentas ainda apresentam limitações, especialmente em cenários mais complexos que exigem julgamento humano, como a decisão sobre modularização adequada ou clareza de nomenclaturas (GE *et al.*, 2012). Mais recentemente, pesquisas têm explorado a aplicação de técnicas de aprendizado de máquina e, em particular, de LLMs, como forma de sugerir refatorações de maneira automática e contextualizada (CHEN *et al.*, 2025). Essa abordagem representa uma evolução importante, pois permite aliar a sistematização de boas práticas codificadas em catálogos clássicos ao potencial de generalização e adaptação de modelos de IA, promovendo refatorações que preservam funcionalidade e atendem a padrões de estilo e clareza cognitiva.

A refatoração de código desempenha um papel estratégico ao articular a detecção de *code smells* com a melhoria efetiva da qualidade de software. Mais do que uma técnica corretiva, ela funciona como ponto de convergência entre métricas objetivas, análises automatizadas e percepções subjetivas de desenvolvedores. Esse caráter híbrido explica sua relevância não apenas para a avaliação de algoritmos de detecção, mas também para a análise crítica de refatorações sugeridas por modelos de linguagem.

2.4 Aprendizado de Máquina na Detecção de Smells

A detecção automática de *code smells* é um campo de pesquisa que tem ganhado relevância crescente, pois busca reduzir a subjetividade inerente ao processo manual e aumentar a escalabilidade da análise de qualidade de código. A abordagem tradicional de detecção, baseada em inspeções humanas e em ferramentas estáticas de análise, apresenta limitações relacionadas à consistência dos resultados e à dificuldade de lidar com grandes bases de código (MANTYLA *et al.*, 2003). Nesse contexto, o uso de técnicas de ML representa um avanço significativo, ao permitir que algoritmos aprendam padrões associados a *smells* a partir de métricas de software e bases rotuladas.

O *Aprendizado de Máquina* (AM), ou ML, é um subcampo da Inteligência Artificial voltado ao desenvolvimento de algoritmos capazes de aprender padrões a partir de dados e, com isso, realizar previsões ou tomar decisões sem que tenham sido explicitamente programados para

cada tarefa (MITCHELL, 1997; RUSSELL; NORVIG, 2021). No Aprendizado de Máquina, segundo Ray (2019), um programa de computador é designado para realizar determinadas tarefas e diz-se que a máquina aprendeu com sua experiência se seu desempenho mensurável nessas tarefas melhora à medida que adquire cada vez mais experiência na execução delas. Dessa forma, os modelos de ML tomam decisões e fazem previsões baseadas em dados, explorando regularidades presentes em grandes volumes de informação.

Neste estudo, foram selecionados cinco algoritmos de aprendizado de máquina para comparação quanto ao desempenho na detecção de *code smells*. São eles: ***Multilayer Perceptron*** (MLP), uma rede neural multicamada com capacidade de capturar relações não lineares (RUCK *et al.*, 1990); **Árvore de Decisão**, algoritmo hierárquico baseado em regras de divisão sucessiva dos dados (QUINLAN, 1990); **Floresta Aleatória**, técnica de *ensemble learning* que combina múltiplas árvores para aumentar robustez (BREIMAN, 2001); **Gradiente Boost**, método aditivo que corrige iterativamente os erros de modelos anteriores (FRIEDMAN, 2001); e ***Support Vector Machines*** (SVM), que busca hiperplanos ótimos para separar classes em espaços de alta dimensionalidade (NOBLE, 2006). Esses algoritmos foram escolhidos por representarem abordagens clássicas e consolidadas na literatura, além de oferecerem diferentes formas de lidar com a complexidade do problema de classificação de *smells*.

Fontana *et al.* (2016) realizaram um dos estudos de referência nesse campo, utilizando 74 sistemas do *Qualitas Corpus*, repositório criado por Tempero *et al.* (2010), e 16 classificadores de aprendizado de máquina para detectar quatro *smells* — *God Class*, *Data Class*, *Feature Envy* e *Long Method*. Os resultados mostraram que diferentes algoritmos apresentam desempenhos variados dependendo do *smell* em análise, sugerindo que não há um modelo universalmente ótimo. Trabalhos posteriores, como os Kaur e Kaur (2021) e Abdou e Darwish (2024), reforçam essa constatação, apontando que técnicas baseadas em *ensembles*, como *Random Forest* e *Gradient Boosting*, tendem a alcançar maior acurácia e robustez em comparação a algoritmos individuais.

A aplicação de ML na detecção de *smells* geralmente segue um processo composto por quatro etapas principais: (i) extração de métricas do código-fonte, como complexidade, coesão, acoplamento e tamanho; (ii) pré-processamento dos dados, incluindo normalização e tratamento de desbalanceamento de classes; (iii) treinamento supervisionado de classificadores a partir de bases rotuladas por especialistas ou heurísticas; e (iv) avaliação por meio de métricas de desempenho, tais como acurácia, precisão, sensibilidade e F1-*score* (MHAWISH; GUPTA, 2020; NUCCI *et al.*, 2018).

Além disso, recentes avanços têm explorado o uso de *deep learning* na detecção de *smells*, empregando redes neurais convolucionais (CNNs) e recorrentes (RNNs) para capturar padrões semânticos e estruturais diretamente do código (ALAZBA *et al.*, 2023). Contudo, esses modelos ainda enfrentam desafios relacionados à interpretabilidade e à necessidade de grandes volumes de dados rotulados, o que limita sua aplicabilidade em contextos industriais. A principal contribuição do uso de ML para detecção de *smells* está na possibilidade de transformar um processo subjetivo em um procedimento mais objetivo e automatizado, sem perder de vista a complexidade inerente ao julgamento humano. Nesse sentido, a literatura indica que a combinação entre abordagens algorítmicas e avaliação por desenvolvedores pode gerar resultados mais confiáveis (POSNETT *et al.*, 2011; SANTOS; GEROSA, 2018).

O aprendizado de máquina desempenha papel duplo: de um lado, fornece modelos de predição de elevada acurácia para identificar *code smells* em larga escala; de outro, serve como contraponto objetivo às percepções humanas coletadas via experimentos empíricos, permitindo investigar convergências e divergências entre medidas automáticas e julgamentos subjetivos.

2.5 Normalização e Validação dos Dados

A qualidade dos resultados obtidos em experimentos com aprendizado de máquina depende não apenas da escolha de algoritmos, mas também do tratamento prévio dos dados e da forma como o desempenho dos modelos é avaliado. Nesse sentido, a normalização e a validação dos dados representam etapas fundamentais para garantir consistência, comparabilidade e generalização dos resultados (HAN *et al.*, 2022). Em problemas de classificação de *code smells*, as métricas extraídas do código-fonte apresentam escalas heterogêneas: enquanto a complexidade ciclomática é expressa em valores inteiros e potencialmente altos, o acoplamento pode assumir valores pequenos, e o número de atributos ou métodos tende a variar em escalas intermediárias. Essa heterogeneidade pode induzir viés em algoritmos de ML, sobretudo em métodos baseados em distância, como k-NN, ou em modelos sensíveis à escala de atributos, como Redes Neurais e SVM (AL-SHALABI *et al.*, 2006).

A normalização busca reduzir esse viés ao transformar os dados para uma escala comparável. De modo geral, pode-se representar o processo de transformação de um vetor de atributos x em x' por meio de um escalonamento (E) e um fator de tradução (T): $x' = Ex + T$. Diferentes técnicas podem ser aplicadas, dependendo da distribuição dos dados e das exigências do modelo: escalonamento para intervalos, transformação logarítmica, padronização por Z-score

e normalização Min-Max (HAN *et al.*, 2022). Neste trabalho, optou-se pela normalização Min-Max, que dimensiona os valores para o intervalo $[0,1]$, sendo dada pela expressão: $x' = \frac{x-\min(x)}{\max(x)-\min(x)}$

Outro aspecto essencial é a forma de avaliar a capacidade de generalização dos modelos treinados. Em contextos de pesquisa acadêmica, a simples divisão dos dados em treino e teste pode levar a estimativas instáveis, sobretudo quando os conjuntos de dados são limitados ou desbalanceados. Para mitigar esse problema, a **validação cruzada** (*cross-validation*) tornou-se prática consolidada (ARLOT; CELISSE, 2010).

Na validação cruzada *k-fold*, o conjunto de dados é dividido em k subconjuntos de tamanho aproximadamente igual. Em cada iteração, $k-1$ subconjuntos são usados para treinamento e o subconjunto restante é reservado para teste. Esse procedimento é repetido k vezes, de forma que cada subconjunto seja utilizado exatamente uma vez como conjunto de teste. A média dos resultados obtidos constitui uma estimativa mais robusta do desempenho real do modelo, reduzindo a variabilidade associada a uma única partição dos dados (ARLOT; CELISSE, 2010; KOHAVI *et al.*, 1995).

A avaliação do desempenho dos modelos requer também a aplicação de métodos estatísticos. Medidas como média e desvio padrão sintetizam tendências centrais e dispersão em torno dos resultados. Para um conjunto de n observações $x_1, x_2, \ldots, x_n$, a média é dada por: $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ e o desvio padrão amostral por: $s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2}$

Para comparações entre condições pareadas, como desempenho de algoritmos com e sem validação cruzada, pode-se utilizar o **teste de Wilcoxon para postos sinalizados**, adequado para pequenas amostras e sem pressuposição de normalidade (WILCOXON, 1945). O estatístico de Wilcoxon é dado pela soma dos postos positivos: $W = \sum_{i=1}^{n} R_i^+$ onde R_i^+ representa o posto atribuído aos casos em que a diferença $d_i > 0$. A hipótese nula (H_0) assume que a mediana das diferenças é zero. A análise também deve considerar os riscos de erro: o **erro tipo I**, com probabilidade α, ocorre quando se rejeita H_0 sendo ela verdadeira; e o **erro tipo II**, com probabilidade β, ocorre quando não se rejeita H_0 mesmo sendo falsa. O poder estatístico do teste é expresso como $1-\beta$.

Em experimentos que envolvem percepção humana, como questionários aplicados a desenvolvedores, utiliza-se frequentemente a escala Likert, estruturada em múltiplos níveis de concordância (JOSHI *et al.*, 2015). Para avaliar a consistência interna dos itens, emprega-se o **Alfa de Cronbach** (CRONBACH, 1951), definido como: $\alpha = \frac{k}{k-1}\left(1 - \frac{\sum_{i=1}^{k}\sigma_i^2}{\sigma_T^2}\right)$ onde k é o

número de itens, σ_i^2 é a variância de cada item e σ_T^2 é a variância total do escore composto. Valores de $\alpha \geq 0.7$ são geralmente considerados indicativos de consistência aceitável (GLIEM; GLIEM, 2003). No contexto da detecção de *code smells*, a normalização assegura que métricas como complexidade, coesão e acoplamento sejam analisadas em bases comparáveis, evitando que atributos em maior escala dominem o processo de classificação. A validação cruzada, por sua vez, confere robustez às estimativas de desempenho, reduzindo vieses decorrentes de partições específicas dos dados.

Complementarmente, a aplicação de fundamentos estatísticos, como médias, desvios padrão e testes não paramétricos, como Wilcoxon, possibilita avaliar de forma crítica as diferenças de desempenho entre algoritmos, ao mesmo tempo em que a consideração dos erros tipo I e II garante interpretação mais cautelosa dos resultados. Em paralelo, instrumentos de medição aplicados em pesquisas empíricas, como escalas Likert e o Alfa de Cronbach, permitem avaliar a consistência interna e a confiabilidade das percepções coletadas junto a desenvolvedores, integrando assim dimensões subjetivas ao processo de análise.

2.6 Ferramentas e Usabilidade em Pesquisas Acadêmicas

A evolução de ferramentas de apoio à Engenharia de Software tem desempenhado papel fundamental na análise e detecção de *code smells*, bem como na avaliação de qualidade de código em ambientes acadêmicos e industriais. Soluções como PMD, Checkstyle, iPlasma e JCodeOdor se destacam por automatizar a coleta de métricas e identificação de padrões problemáticos (LIGGESMEYER; TRAPP, 2009; FONTANA *et al.*, 2016). No entanto, tais ferramentas apresentam limitações quanto à interpretabilidade dos resultados e à replicabilidade de experimentos, o que impacta diretamente sua adoção em pesquisas empíricas (MÄNTYLÄ; LASSENIUS, 2006; SAHIN *et al.*, 2014). No contexto acadêmico, a disponibilidade de ferramentas que conciliem análise técnica com coleta estruturada de percepções é especialmente relevante. Estudos empíricos em Engenharia de Software demandam não apenas métricas quantitativas, mas também instrumentos que capturem dimensões subjetivas, como legibilidade e clareza, sob a ótica de desenvolvedores (BINKLEY *et al.*, 2013; POSNETT *et al.*, 2011). Diversos autores ressaltam que a confiabilidade dos resultados depende de procedimentos sistemáticos e do suporte de ferramentas capazes de reduzir vieses e assegurar a rastreabilidade das evidências (WOHLIN *et al.*, 2012; RUNESON *et al.*, 2012; FALESSI *et al.*, 2018).

A usabilidade dessas ferramentas é um aspecto crítico. Segundo Nielsen (1994), a

usabilidade pode ser avaliada por meio de heurísticas que incluem facilidade de aprendizagem, eficiência, memorabilidade, redução de erros e satisfação do usuário. Esses critérios tornaram-se referência no campo de Interação Humano-Computador (IHC) e têm sido aplicados também à avaliação de sistemas de apoio à pesquisa (SAURO; LEWIS, 2016). Em pesquisas empíricas, a adoção de ferramentas com baixa usabilidade pode introduzir vieses significativos, como aumento da carga cognitiva dos participantes ou dificuldades na execução de tarefas propostas. Evidências nesse sentido foram relatadas por Mäntylä (2005), ao demonstrar que mesmo em avaliações de *code smells* há baixa concordância entre avaliadores humanos, indicando que o uso de métricas ou ferramentas pouco intuitivas pode comprometer a confiabilidade dos resultados. De forma complementar, Fakhoury *et al.* (2018) mostraram que léxicos de código pobres e problemas de legibilidade elevam a carga cognitiva dos desenvolvedores, sugerindo que ferramentas com baixa usabilidade podem potencializar esses efeitos, interferindo na validade interna de experimentos empíricos.

Outro elemento importante é a integração de métodos de coleta de dados qualitativos e quantitativos. Questionários baseados em escalas Likert, por exemplo, são amplamente utilizados para medir percepções subjetivas de forma padronizada (JOSHI *et al.*, 2015), enquanto técnicas de análise de conteúdo (BARDIN, 2016) permitem examinar respostas abertas e identificar categorias emergentes. Uma ferramenta bem projetada deve incorporar tais mecanismos, de forma a apoiar não apenas a execução técnica de experimentos, mas também a análise metodológica robusta de seus resultados.

Nesse cenário, iniciativas recentes, como a TwinCode (Capítulo 4), buscam preencher essa lacuna ao aliar funcionalidades de comparação de código com instrumentos integrados de coleta de dados, alinhando-se às demandas metodológicas da Engenharia de Software empírica. Portanto, a discussão sobre ferramentas e usabilidade em pesquisas acadêmicas não se restringe a aspectos técnicos, mas se estende à viabilidade de experimentos replicáveis, à confiabilidade das percepções coletadas e ao fortalecimento do rigor científico.

2.7 Percepção Humana de Qualidade de Código

A avaliação da qualidade de software não pode ser reduzida apenas a indicadores objetivos, visto que a percepção humana desempenha papel central na interpretação e julgamento de atributos de código. Desenvolvedores, revisores e pesquisadores frequentemente avaliam trechos de software não apenas com base em métricas estruturais, mas também a partir de

fatores cognitivos, como clareza, familiaridade e consistência estilística (POSNETT *et al.*, 2011; BINKLEY *et al.*, 2013).

Estudos empíricos têm mostrado que a percepção de qualidade de código é influenciada por múltiplos elementos. Buse e Weimer (2009), por exemplo, evidenciaram que atributos como legibilidade e nomenclatura têm impacto direto na avaliação subjetiva dos desenvolvedores, muitas vezes mais do que métricas tradicionais como acoplamento ou complexidade. Nesse sentido, fatores como o tempo necessário para compreender um trecho, o esforço cognitivo associado e a facilidade de navegação na estrutura do código tornam-se determinantes para a percepção de qualidade. Posnett *et al.* (2011) destacam que a legibilidade é fortemente associada à compreensibilidade do código, mas não é totalmente capturada por métricas objetivas. Binkley *et al.* (2013), em um estudo com 120 participantes, concluíram que mudanças em elementos aparentemente superficiais, como estilo de indentação e nomes de variáveis, impactam significativamente a percepção dos desenvolvedores sobre a clareza e simplicidade do código. Isso demonstra que medidas estritamente quantitativas não são suficientes para explicar a experiência humana de interação com software.

No âmbito da Engenharia de Software empírica, a análise da percepção humana é frequentemente realizada por meio de experimentos controlados, entrevistas e questionários estruturados, que capturam tanto respostas quantitativas quanto qualitativas (via análise de conteúdo). Santos e Gerosa (2018) enfatizam que tais abordagens são fundamentais para compreender como desenvolvedores interpretam atributos como modularidade, manutenibilidade e legibilidade, uma vez que o julgamento humano é inevitável em processos como revisão de código e avaliação de soluções alternativas de design. Outro aspecto importante refere-se às diferenças entre níveis de experiência. Pesquisadores como Begel e Simon (2008) mostraram que programadores juniores tendem a valorizar mais a clareza superficial e a consistência visual, enquanto desenvolvedores sêniores concentram-se em dimensões mais abstratas, como modularidade e custo de manutenção. Essa heterogeneidade sugere que a percepção de qualidade não é homogênea, mas mediada por fatores de experiência, contexto de uso e objetivos de desenvolvimento.

2.8 Modelos de Linguagem de Grande Porte (LLMs)

LLMs representam um avanço recente no campo do Processamento de Linguagem Natural (PLN), sendo baseados em arquiteturas de *deep learning*, em particular nos *transformers*

propostos por Vaswani *et al.* (2017). Esses modelos são treinados em grandes volumes de texto, utilizando mecanismos de atenção para capturar relações contextuais de longo alcance entre tokens, o que lhes permite gerar, completar e reescrever trechos de linguagem natural e de programação com alta fluidez e coerência (BROWN *et al.*, 2020).

Na área da Engenharia de Software, pesquisas recentes têm explorado o uso de LLMs para apoiar diretamente o processo de refatoração de código. Modelos como Codex (CHEN *et al.*, 2021), Code LLaMA (ROZIERE *et al.*, 2023) e GPT-4 (ACHIAM *et al.*, 2023) demonstraram capacidade de propor modificações estruturais em classes e métodos, frequentemente alinhadas a práticas descritas em catálogos clássicos de refatoração (FOWLER, 1999). White *et al.* (2023) mostram que os LLMs conseguem sugerir transformações como *Extract Method* e *Move Method* preservando a semântica do programa, enquanto Chen *et al.* (2025) apontam sua eficácia na remoção de determinados *code smells*.

Entre as principais vantagens do uso de LLMs nesse domínio estão: (i) a capacidade de lidar com múltiplos contextos de programação, inclusive linguagens diferentes; (ii) a possibilidade de gerar explicações textuais que auxiliam na compreensão das refatorações propostas; (iii) a integração natural com fluxos de trabalho de desenvolvimento, como revisões de código em sistemas de controle de versão. Por outro lado, limitações relevantes ainda precisam ser enfrentadas: estudos indicam que os modelos têm propensão a gerar *code smells* (VELASCO *et al.*, 2025) e enfrentam desafios em compreensão semântica dinâmica e riscos de alucinação, o que compromete a interpretabilidade das saídas (MA *et al.*, 2023).

Do ponto de vista metodológico, os LLMs oferecem uma oportunidade única de combinar a objetividade de algoritmos de aprendizado com a subjetividade da avaliação humana. Ao propor refatorações, os modelos tornam-se passíveis de avaliação empírica por desenvolvedores, permitindo investigar não apenas sua eficácia técnica, mas também sua aceitação e adequação no contexto do trabalho real de programação.

Assim, os LLMs configuram-se como elemento promissor na investigação aqui proposta, funcionando como ponte entre a detecção automatizada de *code smells*, tradicionalmente apoiada em métricas e aprendizado de máquina, e a refatoração de código mediada por julgamentos humanos. Essa intersecção reforça o caráter interdisciplinar da pesquisa, ao integrar fundamentos de Engenharia de Software, PLN e IHC.

3 DETECÇÃO DE *CODE SMELL* COM APRENDIZADO DE MÁQUINA

Este capítulo compartilha material com uma publicação: "Estudo Empírico: Detecção de Code Smells com Aprendizado de Máquinas" (MOREIRA et al., 2024)

Code smell é uma característica estrutural do *software* que indica um aspecto no código ou no design que pode causar problemas na manutenção do *software*. *Code smell* não é um erro no sistema, pois não impede o funcionamento do programa, mas pode aumentar o risco de falha do *software* ou desacelerar o desempenho. Notavelmente, a predição precoce do *code smell* durante a fase de desenvolvimento é muito importante, especialmente em projetos grandes (ABDOU; DARWISH, 2024). O processo de refatoração é fundamental para eliminar *code smells* e melhorar a qualidade do *software*. Fowler (2018) apresenta uma definição de 22 tipos de *code smells* no código-fonte e oferecem algumas operações de refatoração para corrigi-los.

O impacto dos *code smells* no *software* foram examinados por vários estudos e revelaram seu efeito indesejável na qualidade do *software* (YAMASHITA; MOONEN, 2012; YAMASHITA; MOONEN, 2013; SAHIN *et al.*, 2014). Outros autores também investigaram os resultados da remoção dos *code smells* na redução da probabilidade de falhas e erros no sistema de *software*. Hilmi *et al.* (2023) analisaram os desafios decorrentes dos *code smells*, os quais têm efeitos adversos no processo de desenvolvimento de *software*. Ouni *et al.* (2016) e Fowler (2018) recomendaram a aplicação de refatoração no *software* para eliminar *code smells*. Dewangan *et al.* (2021) mostraram evidências do papel fundamental das métricas na detecção de *code smells*. Além disso, eles identificaram que as métricas ajudam na compreensão do código-fonte medindo tanto aspectos funcionais quanto não funcionais do *software*.

Travassos *et al.* (1999) e Ciupke (1999) investigaram a detecção de *Code Smell* por meio de detecção manual. Moha *et al.* (2009) e Tsantalis e Chatzigeorgiou (2009) introduziram abordagens baseados em métricas. Yamashita e Moonen (2013) e Tarwani e Chug (2016) avaliam problemas de manutenibilidade causado por *code smells*. Fontana *et al.* (2016) e Dewangan *et al.* (2021) abordaram a detecção por aprendizado de máquina. Ferramentas de apoio ao desenvolvimento, como SonarQube[1] e PMD[2], já incorporam mecanismos de identificação de

[1] Disponível em: <https://www.sonarsource.com/products/sonarqube/> Acesso em: 3 fev. 2024
[2] Disponível em: <https://pmd.github.io/> Acesso em: 4 fev. 2024

code smells durante o processo de construção do *software*, ainda que baseados majoritariamente em regras estáticas e métricas pré-definidas. Entretanto, observa-se que a maioria dos estudos sobre aprendizado de máquina na detecção de code smells ocorre em ambientes experimentais isolados, sem integração direta com o fluxo contínuo de desenvolvimento e manutenção. Essa lacuna evidencia a necessidade de investigar como técnicas de IA podem ser incorporadas de forma prática e sistemática ao ciclo de vida do software. Nesse sentido, a detecção automatizada de *code smells* por meio de ML pode representar uma alternativa viável para ampliar o suporte ao desenvolvedor em todas as etapas do ciclo de vida do *software*.

O objetivo deste capítulo é investigar a eficácia dos algoritmos de aprendizado de máquina na detecção de *code smells*. Além disso, foi analisado se a técnica de validação cruzada contribui para melhorar os resultados e reduzir as ameaças à validade deste estudo. Para isso, adota-se um conjunto de dados com quatro tipos de *code smells*: *God Class*, *Data Class*, *Feature Envy* e *Long Method*. Cinco algoritmos de aprendizado de máquina (**MLP**, **Árvore de Decisão**, **Floresta Aleatória**, **Gradiente Boost** e **SVM**) são aplicados nos conjuntos de dados. Como resultado, o algoritmo de Floresta Aleatória (97,0%) obteve o melhor desempenho médio, em termos de acurácia, comparado com MLP (94,3%), Árvore de Decisão (93,9%), *Gradiente Boost* (96,6%) e SVM (91,8%).

Este trabalho possui 5 contribuições principais, sendo elas:

- **Avaliação Sistemática Multi-*smell* e Multi-algoritmo.** O estudo utiliza um *benchmark* de cinco classificadores supervisionados (MLP, Árvore de Decisão, Floresta Aleatória, Gradient Boosting e SVM) aplicados aos *code smells Data Class*, *God Class*, *Feature Envy* e *Long Method*, com justificativa técnica para a seleção e descrição do processo típico de aplicação de ML à detecção de *code smells*.
- **Base de Dados Curada e Mensuração Ampla.**: Uso do *Qualitas Corpus* de Tempero *et al.* (2010) (74 sistemas, 51.826 classes, 404.316 métodos) e de múltiplos detectores (iPlasma, PMD, Fluid Tool, Anti-Pattern Scanner), cobrindo seis dimensões de métricas (tamanho, complexidade, coesão, acoplamento, encapsulamento e herança).
- **Desempenho Elevado e *Baselines* Fortes.** Acurácia variando de **89,7%** a **99,2%** entre *code smells* e algoritmos, com picos de **96,8%** para *Data Class*, **93,7%** para *God Class*, **98,4%** para *Feature Envy* e **99,2%** para *Long Method*.
- **Robustez Frente à Validação Cruzada.** Teste de Wilcoxon (pares Sem vs. Com validação cruzada, $n = 5$ algoritmos) não indicou diferenças significativas nas acurácias (todos $p >$

0,05; *Feature Envy* com tendência, $p = 0{,}063$), corroborando estabilidade dos resultados.

- **Síntese Comparativa da Literatura.** Confirmação de que não há modelo universalmente ótimo e de que algoritmos de árvore e *ensembles* (como Floresta Aleatória e Gradient Boosting) tendem a maior acurácia/robustez para detecção de *code smells*.

O restante deste capítulo está organizado da seguinte forma. Na Seção 3.1, é descrito a metodologia usada para o desenvolvimento deste trabalho. Na Seção 3.2, apresenta-se os resultados e analise dos dados. Na Seção 3.3, descreve-se trabalhos relacionados. Na Seção 3.4, é discutido as ameaças a validade da pesquisa. Finalmente, na Seção 3.5, é dada as conclusões o estudo.

3.1 Metodologia

O objetivo principal deste estudo é avaliar o desempenho de técnicas de aprendizado de máquina para a detecção de *code smells*. Como objetivo específico, busca-se avaliar o impacto da validação cruzada na precisão dos algoritmos utilizados. Para alcançar tais objetivos, são exploradas as duas questões de pesquisa descritas a seguir.

- **QP1**. Qual a eficácia de cinco técnicas de *machine learning* (MLP, Árvore de Decisão, Floresta Aleatória, Gradiente Boost e SVM) para detecção de *code smells*?
- **QP2**. Qual o impacto da validação cruzada no conjunto de dados utilizado?

A Figura 2 apresenta as etapas seguidas neste trabalho. Inicialmente, o conjunto de dados é coletado, seguido pelas etapas de pré-processamento e normalização. Esse pré-processamento é necessário para minimizar diferenças nas faixas dos conjuntos de dados e possibilitar a obtenção de melhores parâmetros pelos algoritmos. Em seguida, os dados são divididos em três grupos/conjuntos: treinamento, validação cruzada e teste. Após essa divisão, o desempenho dos algoritmos de aprendizado de máquina é calculado no conjunto de treinamento. Na sequência, aplica-se a técnica de validação cruzada em 10 partes, a fim de avaliar e comparar o desempenho de cada experimento durante o processo de treinamento. Por fim, realiza-se a avaliação final com o conjunto de testes. A seguir, são detalhados o processo de coleta de dados e a avaliação de desempenho.

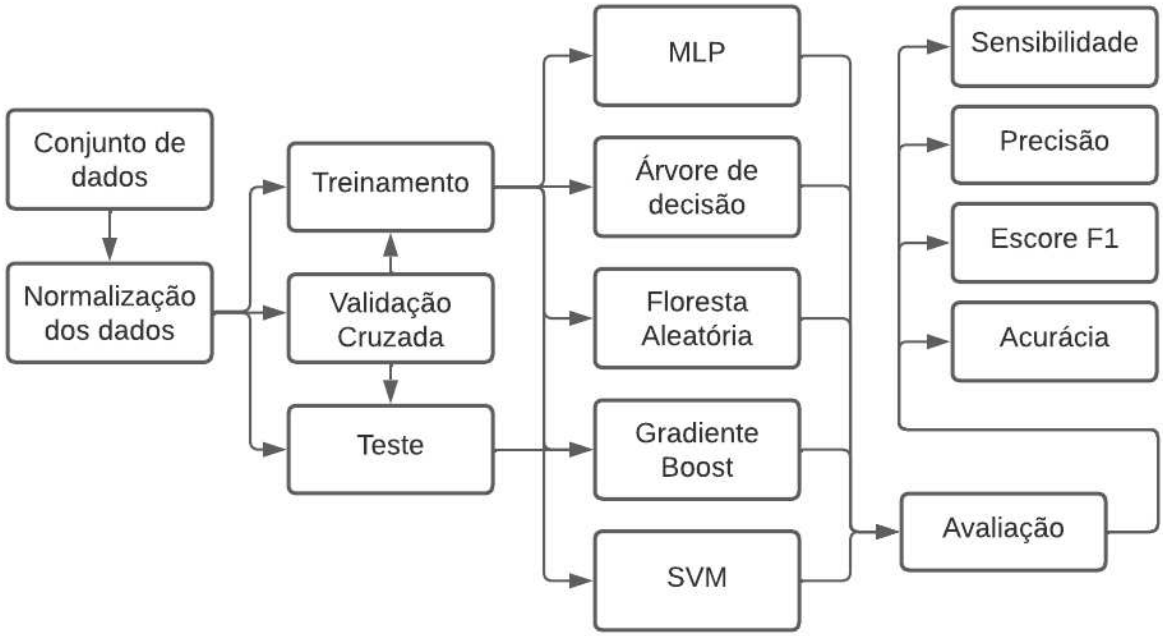


Figura 2 – Fluxo de trabalho proposto.
Fonte: Elaborado pelo autor (2024)

3.1.1 Conjunto de Dados

O conjunto de dados usado foi obtido por Abdou e Darwish (2024) que se basearam no trabalho de Fontana *et al.* (2016). Este conjunto de dados é compostos por 74 sistemas desenvolvidos em Java de diferentes tamanhos e domínios, chamado de *Qualitas Corpus* (QC). Esses sistemas possuem um conjunto de métricas orientadas a objetos coletadas abrangendo os níveis de método, classe, pacote e projeto (TEMPERO *et al.*, 2010). Algumas dessas métricas foram definidas conforme o aspecto da qualidade de *software*, como complexidade, tamanho e acoplamento. Outras métricas dependem da contagem da composição em pacotes ou classes. Os *code smells* foram definidos no nível de método ou classe. No nível de método, foi escolhido para detectar *Feature Envy* e *Long Method*, enquanto no nível de classe, detectamos *Data Class* e *God Class*. Os autores do conjunto de dados escolheram estes quatro *code smells* devido sua alta incidência e pelo impacto negativo na qualidade dos sistemas.

A Tabela 1 apresenta o tamanho geral dos sistemas selecionados por Fontana *et al.* (2016). Na pesquisa de Abdou e Darwish (2024) foram utilizados apenas os dados correspondentes aos *code smells God Class*, *Data Class*, *Feature Envy* e *Long Method*, que são o foco dos experimentos descritos deste estudo. Como destacado por Fontana *et al.* (2016) e Abdou e Darwish (2024), o uso de múltiplos sistemas heterogêneos é fundamental para assegurar que os resultados de aprendizado de máquina não dependam de um conjunto de dados específico e

possam ser generalizados.

Tabela 1 – Visão geral dos sistemas do *Qualitas Corpus*

Característica	Quantidade
Número de Sistemas	74
Número de Pacotes	3.420
Número de Classes	51.826
Número de Métodos	404.316
Número de Linhas de Código	6.785.568

Fonte: Fontana *et al.* (2016)

A Tabela 2 apresenta as ferramentas utilizadas para identificar os *code smells* investigados. A **iPlasma** foi empregada na detecção de todos os *smells*, enquanto a **PMD** contribuiu especificamente para *Long Method* e *God Class*, a **Fluid Tool** para *Data Class* e *Feature Envy*, e o **Anti-Pattern Scanner** exclusivamente para *Data Class*. A seleção dessas ferramentas considerou três critérios: (i) gratuidade, (ii) facilidade de configuração e generalização dos resultados e (iii) diversidade nas regras de detecção. As métricas utilizadas por elas abrangem seis dimensões de qualidade de *software*: tamanho, complexidade, coesão, acoplamento, encapsulamento e herança. As descrições detalhadas podem ser consultadas em material complementar [3].

A ferramenta **iPlasma**[4] é uma plataforma de análise estática que fornece métricas de código orientadas a objetos e permite identificar potenciais *smells* com base em limiares pré-definidos. O **PMD**[5] é uma ferramenta de código aberto que analisa código-fonte em diversas linguagens, identificando problemas recorrentes como variáveis não utilizadas, métodos longos ou complexos e estruturas de controle aninhadas. A **Fluid Tool** foi desenvolvida no contexto da abordagem DECOR e utiliza regras declarativas para especificar e detectar *code smells* (MOHA *et al.*, 2009). Por fim, o **Anti-Pattern Scanner** foi projetado para identificar anti-padrões de projeto que frequentemente se manifestam como *code smells*, combinando métricas de código e heurísticas específicas (TSANTALIS; CHATZIGEORGIOU, 2009).

A Tabela 3 apresenta a divisão do conjunto de dados após a análise das ferramentas e a análise manual descrita em Fontana *et al.* (2016). A classificação foi realizada em dois grupos, de forma semelhante a trabalho anterior (ABDOU; DARWISH, 2024). O primeiro grupo corresponde aos artefatos sem presença de *code smells*, denominados **falsos positivos**. O

[3] Disponível em: <https://essere.disco.unimib.it/machine-learning-for-code-smell-detection/>
[4] Disponível em: <http://loose.upt.ro/research/tools/iplasma>. Acesso em: 15 set. 2025
[5] Disponível em: <https://pmd.github.io/>. Acesso em: 15 set. 2025

Tabela 2 – Detectores de *code smells*

Code Smell	Ferramenta/ Regra de Detecção
Long method	iPlasma, PMD
Data class	iPlasma, Fluid Tool, Anti-Pattern Scanner
God class	iPlasma, PMD
Feature Envy	iPlasma, Fluid Tool

Fonte: Fontana *et al.* (2016)

segundo grupo reúne os artefatos classificados como **verdadeiros positivos**, indicando que a análise dos autores apontou a presença de *code smells*.

Tabela 3 – Composição do conjunto de dados

Code Smell	**Falsos Positivos**	**Verdadeiros Positivos**	**Total**
Long method	280	140	420
Data class	151	269	420
God class	154	266	420
Feature Envy	280	140	420

Fonte: Elaborado pelo autor

3.1.2 Avaliação de Desempenho

Para medir o desempenho dos algoritmos de aprendizado de máquina considera-se quatro métricas: **Precisão**, **Sensibilidade**, **F1-score** e **Acurácia**. O cálculo das quatro métricas investigadas baseia-se nos valores de verdadeiro positivo (TP), verdadeiro negativo (TN), falso positivo (FP) e falso negativo (FN) que são definidos a seguir.

- **Verdadeiro Positivo (TP)**: casos em que o algoritmo identificou corretamente a presença de um *code smell*.
- **Falso Positivo (FP)**: casos em que o algoritmo classificou incorretamente um trecho como contendo *code smell*, quando na realidade não continha.
- **Verdadeiro Negativo (TN)**: casos em que o algoritmo identificou corretamente a ausência de *code smell*.
- **Falso Negativo (FN)**: casos em que o algoritmo deixou de identificar um *code smell* existente, classificando-o como inexistente.

Esses parâmetros são calculados usando uma matriz de confusão que contém as informações reais e previstas reconhecidas pelos classificadores de detecção de padrões de projeto (CATAL, 2012). As equações para o cálculo das avaliações de desempenho são mostradas a seguir.

- **Precisão** é a proporção de casos em que o modelo acertou ao prever uma classe positiva, em relação a todas as vezes que ele previu essa classe — incluindo os acertos e os erros. Essa descrição é representada na Equação 3.1.

$$Precisão = \frac{TP}{TP+FP} \tag{3.1}$$

- **Acurácia** é a razão entre o número total de previsões corretas (verdadeiros positivos e verdadeiros negativos) e o número total de previsões realizadas (incluindo verdadeiros positivos, falsos positivos, verdadeiros negativos e falsos negativos). Essa descrição é representada na Equação 3.2.

$$Accurácia = \frac{TP+TN}{TP+FP+TN+FN} \tag{3.2}$$

- **Sensibilidade ou revocação** é a proporção de casos positivos que o modelo identificou corretamente em relação ao total de casos que realmente são positivos, como exposto na Equação 3.3.

$$Sensibilidade = \frac{TP}{TP+FN} \tag{3.3}$$

- **F1-Score** é a média harmônica entre a Precisão e a Sensibilidade, como visto na Equação 3.4. Hossin e Sulaiman (2015) fala que F1-score busca equilíbrio entre essas precisão e sensibilidade, especialmente útil quando há uma distribuição desigual entre classes. Por exemplo, quando o números de classes sem *code smells* é muito maior que o número de classes sem nenhum *code smell*).

$$F1\text{-}Score = 2 \times \frac{\text{Precisao} \times \text{Sensibilidade}}{\text{Precisao} + \text{Sensibilidade}} \tag{3.4}$$

A **acurácia** foi adotada como métrica principal por sintetizar, em um único valor, a proporção de acertos do classificador, sendo intuitiva e amplamente utilizada em problemas de classificação supervisionada (FOODY, 2023). Embora métricas como Precisão, Sensibilidade e F1-*score* ofereçam informações complementares, a acurácia se destaca por sua simplicidade interpretativa e por refletir de forma direta a capacidade dos algoritmos em distinguir corretamente trechos com e sem *code smells* (GOPALAKRISHNA *et al.*, 2013).

3.2 Resultados

Nesta seção, os resultados obtidos são descritos a partir da aplicação dos algoritmos de aprendizado de máquina ao conjunto de dados. A apresentação está organizada em duas subseções principais: a subseção 3.2.1 analisa a eficácia dos algoritmos considerando a acurácia em diferentes tipos de *code smells*, destacando aqueles que alcançaram melhor desempenho e discutindo suas variações. A subseção 3.2.2 avalia o impacto da técnica de validação cruzada nos experimentos, comparando os resultados com o procedimento tradicional de partição simples e discutindo suas implicações metodológicas.

3.2.1 QP1 - Eficácia dos Algoritmos

Como mencionado anteriormente, utilizou-se cinco algoritmos de aprendizado de máquina para detecção de quatro *code smells* em projetos de *software*. Os quatro tipos de *code smells* possuem dois níveis de granularidade: classes e métodos. A Tabela 4 compara os resultados das métricas com todos os *code smells*, sem validação cruzada, usando a técnica *holdout* dividindo o conjunto de dados em 70% para treino e 30% para teste e também foi usado para validação, pois esses dados não foram usados no treinamento dos algoritmos. Como pode-se observar os algoritmos obtiveram bom desempenho. Com resultados da acurácia variando entre 89,7% para os algoritmo Árvore de Decisão e SVM nos *code smells Data class e God Class*, respectivamente e 99,2% para os algoritmos Árvore de Decisão, Floresta Aleatória e Gradiente *Boost* para o *code smell Long Method*.

Analisando os dados gerado pelo algoritmo MLP, pode ser visto que seu melhor desempenho em termos de acurácia é para o *code smell Long Method* (96,0%), porém o mesmo ainda se mostra muito eficiente na detecção dos demais *code smells* com valor mínimo de acurácia de 92,9% para *God Class*, os demais *smells* alcançaram valores de 93,7% (*Data Class*)

Tabela 4 – Comparação das métricas

Code Smell	Algoritmo	Precisão	Sensibilidade	F1-Score	Acurácia
Data Class	MLP	93,1%	93,7%	93,4%	93,7%
	Árvore de Decisão	88,9%	90,0%	89,3%	89,7%
	Floresta Aleatória	96,7%	**96,7%**	**96,7%**	**96,8%**
	Gradiente Boost	**97,0%**	96,3%	96,6%	**96,8%**
	SVM	91,3%	91,2%	91,3%	92,3%
God Class	MLP	**92,9%**	91,7%	92,3%	92,9%
	Árvore de Decisão	88,9%	89,2%	89,0%	89,7%
	Floresta Aleatória	84,1%	**92,4%**	**93,1%**	**93,7%**
	Gradiente Boost	**92,9%**	91,7%	92,3%	92,9%
	SVM	91,1%	87,0%	88,5%	89,7%
Feature Envy	MLP	93,2%	93,8%	93,5%	94,4%
	Árvore de Decisão	95,7%	96,6%	96,3%	96,8%
	Floresta Aleatória	**98,1%**	**98,1%**	**98,1%**	**98,4%**
	Gradiente Boost	97,5%	96,8%	97,2%	97,6%
	SVM	91,2%	89,8%	89,2%	91,3%
Long Method	MLP	95,2%	95,8%	95,5%	96,0%
	Árvore de Decisão	**99,4%**	98,8%	**99,1%**	**99,2%**
	Floresta Aleatória	98,8%	**99,4%**	**99,1%**	**99,2%**
	Gradiente Boost	**99,4%**	98,8%	**99,1%**	**99,2%**
	SVM	92,8%	92,8%	92,8%	93,7%

Fonte: Elaborado pelo autor (2024)

e 94,4% (*Feature Envy*). Para o algoritmo Árvore de Decisão, o seu melhor resultado em termos de acurácia foi para o *code smell Long Method* (99,2%). No entanto, este algoritmo teve o pior desempenho entre os algoritmos analisados para os *code smells Data Class* (89,7%) e *God Class* (89,7%), *Feature Envy* (96,8%). O SVM se mostrou promissor, com acurácia de 93,7% para *Long Method*, 92,3% para *Data Class*, 89,7% para *God Class* e 91,3% para *Feature Envy*. Os resultados de acurácia para Floresta Aleatória são os mais promissores, pois obteve melhor acurácia para os quatro *code smells*, sendo 96,8% para *Data Class*, 93,7% para *God Class*, 98,4% para *Feature Envy* e 99.2% para Long Method. Já o Gradiente *Boost* ficou logo atrás da Floresta Aleatória com acurácia igual nos *code smells Data Class* (96,8%) e *Long Method* (99,2%), mas um pouco menos eficiente para *God Class* (92,9%) e *Feature Envy* (97,6%).

Resumo da resposta da QP1. A acurácia, variou de 89,7% à 99,2%. O algoritmo *Árvore de Decisão* obteve melhor acurácia para todos os *code smells*, sendo 93,7% para *God Class*, 96,8% para *Data Class*, 98,4% para *Feature Envy* e 99,2% para *Long Method*.

3.2.2 QP2 - Eficácia dos Algoritmos com Validação Cruzada

A Tabela 5 apresenta os dados referentes ao desempenho da acurácia dos algoritmos empregando a técnica de validação cruzada de 10 partes. É mostrado, com exceção do algoritmo SVM para *Feature Envy* (88,5%), que todos os algoritmos alcançaram acurácia superior a 90% para todos os *code smells*, onde é destacado o desempenho do algoritmo Floresta Aleatória obteve 94,2% para o *code smell Data class*, 96,3% para *God Class*, para *Feature Envy* 94,5% e 99,0% para *Long Method*.

Tabela 5 – Desempenho dos algoritmos com validação cruzada em 10 partes (Acurácia)

	Data Class	*God Class*	*Featury Envy*	*Long Method*
MLP	91,9%	91,6%	90,5%	98,3%
Árvore de Decisão	90,8%	91,2%	93,6%	98,6%
Floresta Aleatória	**94,2%**	**96,3%**	**94,5%**	**99,0%**
Gradiente Boost	93,2%	94,6%	93,9%	98,6%
SVM	91,2%	91,2%	88,5%	97,3%

Fonte: Elaborado pelo autor (2024)

A Tabela 6 apresenta uma comparação dos resultados em termos de acurácia sem validação cruzada (resultados apresentados na Tabela 4) e com validação cruzada (resultados apresentados na Tabela 5). Essa comparação é apenas para **efeito didático**, já que a finalidade da validação cruzada é a mitigação de possíveis falhas no processo de treinamento da técnica *holdout*. Analisando a acurácia de *Data Class*, apenas Árvore de Decisão obteve melhores resultados utilizando validação cruzada (89,7% *versus* 90,8%). No caso de *God Class* apenas o algoritmo MLP não obteve melhora da acurácia (92,9% e 91,6%). Os demais algoritmos obtiveram melhora de pelo menos 1,5%. Por exemplo, SVM melhorou de 89,7% para 91,2%. No caso de *Feature Envy* nenhum algoritmo obteve melhora com a validação cruzada. A acurácia caiu pelo menos 2,8% e chegando a 3,9% no caso de MLP (94,4% para 90,5) e Floresta Aleatória (97,6% para 93,9%). No caso de *Long Method*, a acurácia melhorou apenas para dois algoritmos: MLP de 96,0% para 98,3% e SVM de 93,7% para 97,3%. Os demais algoritmos que haviam performado melhor em toda a análise tiveram sua acurácia reduzida de 99,2% para 98,6% nos casos de Árvore de Decisão e Gradiente Boost e de 99,2% para 99,0% no caso de Floresta Aleatória.

Para avaliar se a aplicação de validação cruzada impactou significativamente a acurácia dos algoritmos, foi realizada uma comparação estatística entre os cenários *Sem* e

Tabela 6 – Comparação dos algoritmos **com** validação cruzada e **sem** validação cruzada em termos de acurácia

Algoritmo	*Data Class*		*God Class*		*Feature Envy*		*Long Method*	
	Sem	**Com**	**Sem**	**Com**	**Sem**	**Com**	**Sem**	**Com**
MLP	**93,7**	91,9	**92,9**	91,6	**94,4**	90,5	96,0	**98,3**
Arvore de Decisão	89,7	**90,8**	89,7	**91,2**	**96,8**	93,6	**99,2**	98,6
Floresta Aleatória	**96,8**	94,2	93,7	**96,3**	**98,4**	94,5	**99,2**	99,0
Gradiente Boost	**96,8**	93,2	92,9	**94,6**	**97,6**	93,9	**99,2**	98,6
SVM	**92,3**	91,2	89,7	**91,2**	**91,3**	88,5	93,7	**97,3**

Fonte: Elaborado pelo autor (2024)

Com validação cruzada. Como os dados são **pareados** (o mesmo algoritmo avaliado sob duas condições distintas), foi utilizado o **teste de Wilcoxon para postos sinalizados** (WILCOXON, 1945). Esse teste não paramétrico foi escolhido por três motivos principais: (i) o número de pares é reduzido ($n = 5$ algoritmos), o que compromete a robustez de testes paramétricos como o teste t pareado; (ii) não é possível assumir normalidade para as diferenças entre as condições; (iii) o teste de Wilcoxon é robusto para comparar distribuições dependentes, focando nas diferenças de medianas. A Tabela 7 apresenta os resultados do teste de Wilcoxon para cada *code smell* investigado. Observa-se que, em todos os casos, os valores de p são superiores a $0,05$, indicando ausência de diferenças estatisticamente significativas entre as acurácias *Com* e *Sem* validação cruzada. O resultado mais próximo de significância ocorreu em *Feature Envy* ($p = 0,063$), sugerindo uma tendência, mas ainda sem evidência suficiente ao nível de 5%.

Tabela 7 – Resultados do teste de Wilcoxon comparando acurácia *Sem* vs. *Com* validação cruzada.

Métrica	**Estatística (W)**	**Valor-p**
Data Class	1,5	0,188
God Class	1,0	0,125
Feature Envy	0,0	0,063
Long Method	6,0	0,813

Fonte: Elaborado pelo autor (2024)

Na Figura 3 apresentamos a diferença de acurácia (*Sem* - *Com*) para cada algoritmo, em cada *code smell*. Valores positivos indicam maior desempenho *Sem* validação cruzada, enquanto valores negativos indicam maior desempenho *Com*. Nota-se que as variações são pequenas e inconsistentes entre os algoritmos, reforçando os achados estatísticos de ausência de efeito sistemático.

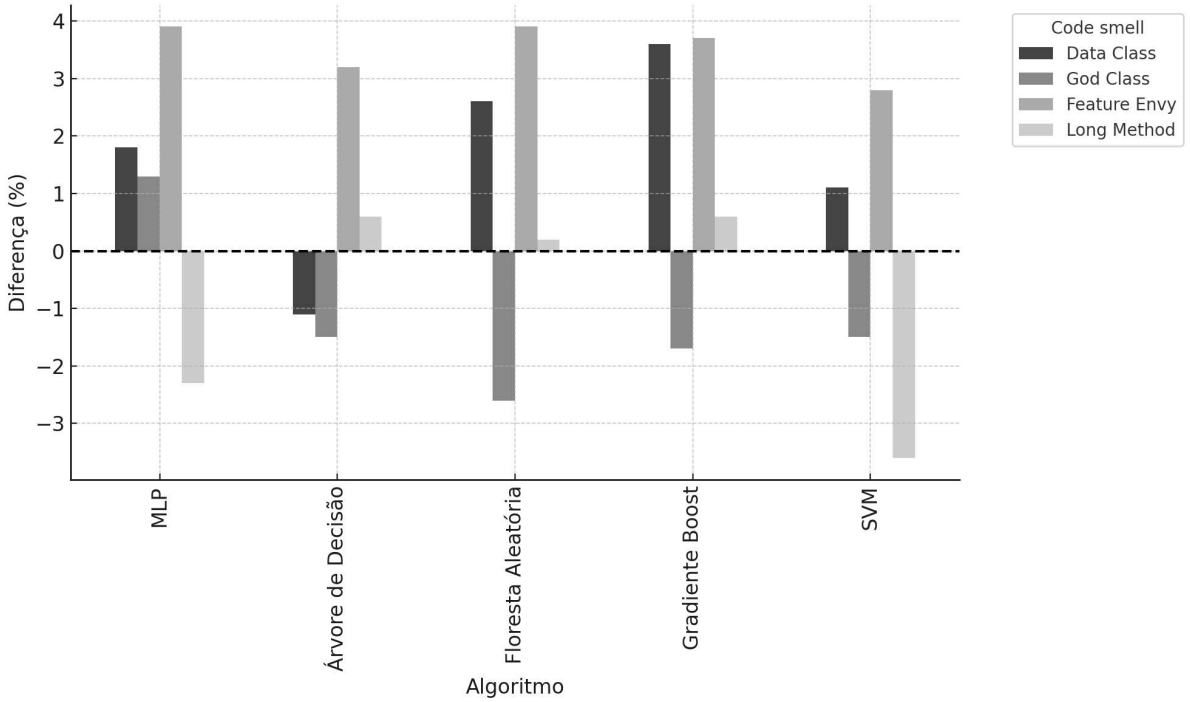


Figura 3 – Diferença de acurácia (*Sem - Com*) por algoritmo em cada métrica. Escala de cinza.
Fonte: Elaborado pelo autor (2024)

Resumo da resposta da QP2. Os resultados indicam que a validação cruzada não alterou significativamente a acurácia média dos algoritmos avaliados. Embora pequenas diferenças tenham sido observadas em alguns cenários, a ausência de significância estatística sugere que tais variações podem ser atribuídas ao acaso, dada a amostra reduzida. A implicação prática é que, para este conjunto de experimentos, a escolha entre utilizar ou não validação cruzada não levou a mudanças consistentes no desempenho observado.

3.3 Trabalhos Relacionados

Foram propostas diversas ferramentas para detecção de *code smells*, abrangendo tanto ferramentas comerciais quanto protótipos de pesquisa. Fontana e Zanoni (2017) falam que essas ferramentas utilizam uma variedade de técnicas para identificar *code smells*: algumas se baseiam em métricas (LANZA; MARINESCU, 2007; FONTANA *et al.*, 2015), outras empregam uma linguagem de especificação própria (MOHA *et al.*, 2009), realizam análise de programas para encontrar oportunidades de refatoração (TSANTALIS; CHATZIGEORGIOU, 2009; TSANTALIS; CHATZIGEORGIOU, 2011), exploram a análise de repositórios de *software* (PALOMBA *et al.*, 2015) ou ainda recorrem a técnicas de aprendizado de máquina. A seguir descreve-se abordagens que empregam técnicas de aprendizado de máquina.

No estudo de Fontana *et al.* (2016), foram analisados quatro *code smells* — *God Class*, *Data Class*, *Feature Envy* e *Long Method* — utilizando 74 sistemas do *Qualitas Corpus* e 16 classificadores de aprendizado de máquina. O referido trabalho reporta valores médios de acurácia distintos para cada *smell*: 74% para *God Class*, 77% para *Data Class*, 93% para *Feature Envy* e 92% para *Long Method*. Em comparação, a abordagem adotada nesta pesquisa obteve resultados superiores, alcançando 98,5% para *God Class*, 93,8% para *Data Class*, 98,4% para *Feature Envy* e 99,2% para *Long Method*.

Nucci *et al.* (2018) replicaram o estudo de Fontana *et al.* (2016) e questionaram a validade dos altos índices de desempenho reportados, argumentando que estes derivavam do uso de *datasets* artificialmente balanceados. Ao realizarem experimentos em cenários mais realistas, observaram uma redução significativa na acurácia média dos classificadores, com valores de aproximadamente 76% para *God Class* e *Data Class*, 92% para *Feature Envy* e 93% para *Long Method*. Esses resultados contrastam com os obtidos nesta pesquisa, que alcançaram 98,5% para *God Class*, 93,8% para *Data Class*, 98,4% para *Feature Envy* e 99,2% para *Long Method*. A comparação evidencia que, embora os achados de Nucci *et al.* (2018) reforcem a necessidade de maior rigor metodológico na construção e validação dos conjuntos de dados, os resultados aqui apresentados mostram que é possível obter desempenhos superiores na detecção de *code smells*.

Mhawish e Gupta (2020) propuseram um *framework* de predição de *code smells* baseado em métricas de *software* e técnicas de aprendizado de máquina, utilizando e reformulando os conjuntos de dados de Fontana *et al.* (2016) para criar versões *binary-label*, *multi-label* e reequilibradas. Foram avaliados seis algoritmos — SVM, MLP, *Deep Learning*, *Decision Tree*, *Random Forest* e *Gradient Boosted Trees* — obtendo-se resultados expressivos, com acurácia de até 99,7% para *Data Class*, 98,4% para *God Class*, 97,9% para *Feature Envy* e 95,9% para *Long Method*. Além disso, os autores aplicaram técnicas de seleção de atributos baseadas em algoritmo genético e otimização de parâmetros via *grid search*, o que contribuiu para a melhoria da performance. Em comparação, os resultados desta pesquisa também apresentaram elevados índices de acurácia — 93,8% para *Data Class*, 98,5% para *God Class*, 98,4% para *Feature Envy* e 99,2% para *Long Method* —, diferenciando-se principalmente pela ênfase na análise metodológica do impacto da validação cruzada sobre a robustez dos modelos.

Pushpalatha e Mrunalini (2021) propuseram uma abordagem de aprendizado de máquina para detectar *code smells* e observaram as métricas que desempenham papéis críticos no processo de detecção. Eles aplicaram algoritmo genético baseado em duas técnicas de seleção

de recursos e técnica de otimização de parâmetros baseada em uma pesquisa em grade. Eles obtiveram valores de acurácia na previsão de *Data Class*, *God Class* e *Long Method* em 98,05%, 97,56% e 94,31%, respectivamente, usando o método GA_CFS.

Abdou e Darwish (2024) apresentam um estudo comparativo de técnicas de aprendizado de máquina para classificar a gravidade de *code smells* em sistemas de *software*. Os pesquisadores propuseram um modelo baseado em métricas de *software* e aprendizado de máquina para detectar esses problemas. Eles utilizaram diferentes abordagens, incluindo classificação multinomial, ordinal e regressão, e avaliaram a precisão na ordenação e classificação da gravidade dos *code smells*. Como resultados eles obtiveram acurácia de: 93,0% para *Data Class*, 92,0% para *God Class*, 97,0% para *Feature Envy* e 97,0% para *Long Method*.

Kaur e Kaur (2021) apresentaram a utilização de técnicas de aprendizado de conjunto (*Ensemble Learning*) em conjunto com técnicas de seleção de características por correlação, aplicadas sobre métricas extraídas pela ferramenta CKJM e *code smells* identificados pelo JCodeOdor, considerando três sistemas Java de código aberto: *DrJava*, *EMMA* e *FindBugs*. Para a avaliação, foram empregados os classificadores *Bagging* e *Random Forest*, analisados a partir de quatro medidas de desempenho: acurácia (P1), G-*mean* 1 (P2), G-*mean* 2 (P3) e F-*measure* (P4). Os experimentos contemplaram os *code smells Message Chains*, *Dispersed Coupling*, *Shotgun Surgery*, *Brain Method*, *Data Class* e *God Class*, cujo os resultados são separados por classificador, considerando presença ou ausência de *smell*. Comparado como o presente trabalho eles foram superiores obtendo 100% de acurácia tanto para *Data Class* quanto *God Class*. Ressalta-se, entretanto, que os *datasets* empregados por eles diferem dos utilizados nesta pesquisa, pois os mesmos utilizaram apenas 3 sistemas, enquanto mo presente estudo foi utilizado um *dataset* com 74 sistemas, como pode ser visto na Tabela 1.

A Tabela 8 apresenta uma síntese comparativa entre a acurácia obtida neste trabalho e aquela reportada em estudos relacionados. Observa-se que os resultados variam de acordo com o tipo de *code smell*, refletindo as particularidades dos conjuntos de dados e dos métodos empregados em cada pesquisa. Para o *Data Class*, o melhor resultado foi obtido por Kaur e Kaur (2021) com 100%, enquanto o presente trabalho alcançou 93,8%, permanecendo em um patamar competitivo em relação a outros estudos, como Abdou e Darwish (2024) (93,0%). No caso do *God Class*, novamente Kaur e Kaur (2021) obteve a maior acurácia (100%) seguido por Mhawish e Gupta (2020) que alcançou (98,5%), sendo que o resultado obtido nesse trabalho (96,3%) também se mostra expressivo e superior à grande parte da literatura, como os 92,0% reportados

por Abdou e Darwish (2024) e 83,0% em Nucci *et al.* (2018). Em relação ao *Feature Envy*, este trabalho apresentou a melhor acurácia entre os estudos comparados, alcançando 98,4%. Esse valor supera o resultado mais próximo (98,0%) reportado por Mhawish e Gupta (2020), com uma diferença de 0,4 pontos percentuais. No *Long Method*, também atingiu o melhor desempenho, com 99,2%, superando em 2,2 pontos percentuais o resultado de Abdou e Darwish (2024) (97,0%) e em 3,2 pontos percentuais o de Mhawish e Gupta (2020) (96,0%).

Tabela 8 – Comparação da acurácia de trabalhos relacionados com o presente trabalho

Autor	**Conjunto de Dados**			
	Data Class	*God Class*	*Feature Envy*	*Long Method*
Fontana e Zanoni (2017)	77,0%	74,0%	93,0%	92,0%
Nucci *et al.* (2018)	83,0%	83,0%	84,0%	82,0%
Mhawish e Gupta (2020)	99,7%	98,5%	97,9%	95,9%
Pushpalatha e Mrunalini (2021)	98,0%	97,6%	-	94,3%
Abdou e Darwish (2024)	93,0%	92,0%	97,0%	97,0%
Kaur e Kaur (2021)	**100%**	**100%**	-	-
Presente trabalho	93,8%	96,3%	**98,4%**	**99,2%**

Fonte: Elaborado pelo autor (2025)

Esses achados indicam que o método proposto obteve desempenho particularmente superior para os *code smells* de granularidade de método (*Feature Envy* e *Long Method*), alcançando valores próximos a 100% e estabelecendo um avanço em relação ao estado da arte. Cabe ressaltar, entretanto, que a comparação entre trabalhos distintos possui caráter descritivo, uma vez que diferentes autores podem ter adotado conjuntos de dados, técnicas de pré-processamento e protocolos experimentais distintos. Ainda assim, os resultados sugerem forte evidência de que a abordagem proposta é competitiva e apresenta ganhos consistentes em cenários relevantes para a detecção automática de *code smells*.

3.4 Ameaças à Validade da Pesquisa

Nesta seção é discutido as principais ameaças à validade do estudo e as medidas adotadas para mitigá-las, organizadas em quatro dimensões clássicas (WOHLIN *et al.*, 2012): validade de construção, interna, externa e de conclusão.

Validade de Construção. São duas ameaças a construção deste estudo. *Métrica de avaliação* - a acurácia pode inflar desempenho em cenários desbalanceados. Como forma de mitigar essa ameaça reporta-se resultados estratificados por *smell* e compara-se com trabalhos relacionados; análises complementares (p. ex., precisão, revocação, F1 e AUC) são recomendadas

para estudos futuros. *Representação e extração de características* - a escolha de atributos (métricas estáticas, métricas, etc.) influencia a capacidade de generalização. Para mitigar essa ameaça foi utilizado *Qualitas Corpus* (TEMPERO *et al.*, 2010) um *dataset* conceituado e utilizado em outros estudos (FONTANA; ZANONI, 2017; ABDOU; DARWISH, 2024; NUCCI *et al.*, 2018) e é mantido o mesmo conjunto de atributos entre condições comparadas para isolar o efeito dos tratamentos.

Validade Interna. A seguir são discutidas três ameaças internas à validade do estudo. *Vazamento de informação (data leakage)* – partições de treino e teste com sobreposição (por exemplo, arquivos/clones quase idênticos) podem inflar resultados. Para evitar esse risco, foram adotados protocolos de separação por projeto/arquivo sempre que aplicável e o pipeline de pré-processamento foi revisado a fim de impedir vazamentos. *Configuração dos modelos e aleatoriedade* – hiperparâmetros, inicializações aleatórias (p. ex., MLP) e variação de *seeds* podem alterar o desempenho. A mitigação foi realizada por meio da fixação de *seeds* reprodutíveis, documentação dos hiperparâmetros e aplicação do mesmo procedimento em todos os algoritmos comparados. *Comparação entre condições* – a comparação *Sem* vs. *Com* validação cruzada foi pareada por algoritmo, mas diferenças residuais de partição podem permanecer. Para reduzir essas diferenças, utilizou-se o teste não paramétrico pareado (Wilcoxon), apropriado para n reduzido e sem suposição de normalidade das diferenças, conforme reportado na Seção 3.2.2.

Validade Externa. Duas ameaças externas à validade do estudo são consideradas. *Generalização para outros projetos e linguagens* – os resultados refletem os 74 projetos analisados, oriundos de diversos domínios e desenvolvidos em Java. Projetos com estilos de codificação, linguagens de programação, arquiteturas ou convenções distintas podem apresentar padrões diferentes de *code smells*. Para mitigar essa limitação, foram selecionados múltiplos projetos e discutidos os limites de generalização; recomenda-se que estudos futuros ampliem a diversidade (p. ex., tamanho, domínio e linguagem de programação). *Comparação com a literatura* – a comparação descritiva com trabalhos relacionados é limitada por diferenças de conjuntos de dados e protocolos. Em razão disso, foi explicitado o caráter descritivo dessas comparações e sugerida a realização de avaliações em *benchmarks* compartilhados, quando disponíveis.

Validade de Conclusão. *Poder estatístico* – o número de pares na comparação *Sem* vs. *Com* validação cruzada é pequeno (n=5 algoritmos), o que reduz o poder para detectar efeitos sutis. Para fortalecer os achados, foi utilizado o teste de Wilcoxon para pares, valores-p foram reportados e tendências próximas ao limiar de significância foram destacadas; replicações

com mais algoritmos/projetos são recomendadas para aumentar o poder estatístico. *Múltiplas comparações* – a avaliação de quatro *smells* pode inflar a taxa de erro Tipo I. Para reduzir esse impacto, os resultados foram interpretados de forma conservadora e, em trabalhos subsequentes, sugerem-se correções (p. ex., Holm–Bonferroni) quando múltiplos testes confirmatórios forem conduzidos.

Em síntese, apesar dessas ameaças, foram adotadas decisões metodológicas conservadoras, aplicados testes estatísticos apropriados ao desenho pareado e realizada a documentação do protocolo experimental, de modo a fortalecer a confiabilidade e a interpretabilidade dos achados deste estudo.

3.5 Conclusão

Neste capítulo, foi proposta uma abordagem baseada em aprendizado de máquina para a detecção de *code smells* em sistemas de *software*. Foram aplicados cinco algoritmos (MLP, Árvore de Decisão, Floresta Aleatória, Gradiente Boost e SVM) combinados com métricas de *software* extraídas do conjunto de dados de Fontana *et al.* (2016). O estudo contemplou quatro *code smells*: *Data Class*, *God Class*, *Feature Envy* e *Long Method*, com o objetivo de avaliar o desempenho relativo dos algoritmos e o impacto da validação cruzada nos resultados de acurácia.

Os experimentos mostraram que os classificadores baseados em árvores (Árvore de Decisão, Floresta Aleatória e Gradiente Boost) apresentaram desempenho superior, destacando-se especialmente na detecção de *Long Method*, com acurácia de até 99,2%. Para *Feature Envy*, obtive-se valores acima de 96%, consolidando a robustez dos classificadores. Em contrapartida, o desempenho foi ligeiramente inferior para *Data Class*, onde a Árvore de Decisão registrou 89,7%, embora os demais algoritmos tenham alcançado cerca de 96,8%. A acurácia para *God Class* variou entre 91,2% e 96,3%, com ganho de desempenho quando aplicada a validação cruzada. Os algoritmos MLP e SVM também demonstraram boa performance, embora com resultados inferiores aos modelos de árvore (Árvore de decisão e Floresta Aleatória).

As contribuições deste estudo residem em três aspectos principais: (i) a demonstração da efetividade de algoritmos de aprendizado de máquina na detecção de diferentes tipos de *code smells*; (ii) a análise comparativa entre múltiplos algoritmos, evidenciando que métodos baseados em árvores oferecem maior acurácia; e (iii) a investigação do impacto da validação cruzada, mostrando que, apesar de não gerar diferenças estatisticamente significativas, influencia o desempenho em alguns cenários. Além disso, os resultados obtidos para *Feature Envy* e *Long*

Method se mostraram competitivos e superiores aos de estudos anteriores, reforçando a relevância da abordagem proposta.

Por fim, este capítulo estabelece a base para os próximos capítulos. No capítulo seguinte, é explorado a aplicação de modelos de linguagem de grande porte (LLMs) para a detecção de *code smells*. Para isso, emprega-se a ferramenta TWINCODE, desenvolvida para analisar e comparar diferentes versões de código-fonte, de modo a investigar o potencial de LLMs como alternativa e complemento às técnicas tradicionais de aprendizado de máquina.

4 TWINCODE

A qualidade de código é central para manutenibilidade, legibilidade, testabilidade e evolução de sistemas (MARTIN, 2009; FOWLER, 2018). Apesar do avanço em métricas automatizadas, a literatura aponta descompasso entre indicadores objetivos e a percepção de desenvolvedores (BUSE; WEIMER, 2009; POSNETT *et al.*, 2011), além da carência de plataformas integradas para estudos empíricos com comparação de código lado a lado e coleta de dados estruturada em formulários. Essa lacuna limita replicabilidade e comparação entre investigações. Como resposta, é proposto a TwinCode, uma ferramenta que integra, em um único ambiente, comparação de trechos lado a lado, cadastro de pares de código, questionários configuráveis por comparação e geração de relatórios. A arquitetura adota PHP/Laravel, React/TailwindCSS e MariaDB, priorizando transparência e adaptabilidade.

A avaliação da TwinCode seguiu abordagem exploratória com 12 participantes, majoritariamente de alto nível de escolaridade e experiência. O instrumento combinou afirmativas avaliadas na escala Likert e questões abertas para examinar facilidade de uso, fluxo funcional para estudos empíricos e potencial de adoção acadêmica. As análises incluíram estatísticas descritivas, percentuais de acordo e síntese temática das respostas qualitativas.

Os resultados indicam bom desempenho do núcleo de inspeção: a visualização *lado a lado* e as pistas visuais foram bem avaliadas. O fluxo funcional atendeu aos requisitos centrais, com destaque para criação de pares e questionários por comparação, e um ponto de atenção na associação pares–questionários. O potencial de adoção acadêmica foi alto e observou-se consistência interna elevada no bloco quantitativo ($\alpha = 0{,}900$). As respostas abertas apontaram melhorias prioritárias em ergonomia visual, saliência de navegação, *feedbacks* de estado e funcionalidades auxiliares como exportação, filtros e versionamento.

Este trabalho possui 6 contribuições principais, sendo elas:

- **TwinCode como artefato científico integrado.** Uma plataforma que reúne, em um único ambiente, (i) comparação de trechos de código lado a lado com realce de sintaxe e numeração de linhas, (ii) questionários configuráveis por comparação para instrumentação dos estudos, e (iii) geração de relatórios estruturados, reduzindo preparo ad hoc e favorecendo padronização e reprodutibilidade. Publicamente disponível em (MOREIRA *et al.*, 2025).
- **Endereçamento de lacuna prática na literatura.** Entrega de uma solução focada em estudos empíricos de qualidade de código (não apenas visualização/diff), que organiza o fluxo de trabalho (cadastro, pareamento, associação com questionários e coleta), promovendo

replicabilidade entre investigações.

- **Validação exploratória com evidências quantitativas e qualitativas.** Estudo com 12 participantes, combinando escala Likert e análise temática; resultados indicam boa aceitação da visualização lado a lado e do fluxo funcional, alto potencial de adoção acadêmica e *consistência interna*.
- **Protocolo de avaliação replicável.** Combinação de estatísticas descritivas, análise de consistência interna e leitura qualitativa por análise de conteúdo, oferecendo um roteiro metodológico reutilizável para avaliar ferramentas científicas similares.
- **Reconhecimento formal do artefato.** Registro da TwinCode no INPI (nº BR512025003573-0), estabelecendo precedência e reforçando originalidade no contexto nacional de ferramentas para pesquisa em engenharia de software.

Este capítulo está organizado em seis seções. A Seção 4.1 descreve a metodologia do estudo, incluindo delineamento, instrumentos e procedimentos de análise. A Seção 4.2 apresenta a arquitetura e as funcionalidades da TwinCode. A Seção 4.3 reporta a validação e os resultados, incluindo as respostas às questões de pesquisa. A Seção 4.4 discute os trabalhos relacionados. A Seção 4.5 apresenta as ameaças à validade. Por fim, a Seção 4.6 traz as considerações finais e diretrizes para trabalhos futuros.

4.1 Metodologia

O desenvolvimento da ferramenta TwinCode emergiu da constatação de uma lacuna significativa no ecossistema de soluções voltadas à realização de estudos empíricos que demandam a comparação sistemática de trechos de código-fonte. Embora existam ferramentas para análise estática, inspeção automatizada ou revisão manual de código, observa-se uma carência de plataformas que integrem, de forma orgânica, a apresentação paralela de fragmentos de código com mecanismos estruturados de coleta de dados, como questionários controlados. Nesse contexto, a TwinCode propõe-se a suprir essa demanda ao articular, em um mesmo ambiente, funcionalidades que favorecem o delineamento de experimentos computacionais, especialmente aqueles voltados à avaliação da qualidade de código, percepção de desenvolvedores e práticas de refatoração.

4.1.1 Objetivos da Pesquisa

O objetivo principal desta pesquisa consiste em avaliar a facilidade de uso, a eficiência funcional e o potencial de adoção da TwinCode enquanto ferramenta de apoio à realização de estudos empíricos voltados à qualidade de código.

De forma complementar, definem-se os seguintes objetivos específicos:

- Mensurar o nível de facilidade de uso percebida da interface da TwinCode por desenvolvedores, com foco em aspectos como clareza das informações, navegabilidade, adequação da visualização de código e possibilidades de customização visual;
- Avaliar a pertinência e a suficiência das funcionalidades oferecidas pela TwinCode para a condução de estudos empíricos sobre qualidade de código, considerando os recursos de comparação de trechos, integração de questionários e apresentação de dados;
- Investigar o potencial de adoção da ferramenta em ambientes acadêmicos, com base em uma análise preditiva fundamentada na percepção integrada de facilidade de uso e adequação funcional por parte dos usuários participantes;
- Mapear as funcionalidades mais valorizadas pelos usuários e categorizar as melhorias prioritárias a serem implementadas, com vistas à otimização da ferramenta enquanto instrumento de suporte à pesquisa científica na área de Engenharia de Software.

4.1.2 Questões Pesquisa

Com base nos objetivos delineados para este estudo, foram formuladas quatro questões de pesquisa (QPs), que orientam a investigação e delimitam o escopo analítico necessário ao desenvolvimento e à avaliação da ferramenta proposta. As QPs estão descritas a seguir:

- **QP1** – Qual é o nível de facilidade de uso percebida da TwinCode, considerando os aspectos de clareza, navegação e funcionalidades de visualização de código?
- **QP2** – A TwinCode atende aos requisitos funcionais necessários para condução de estudos empíricos sobre qualidade de código?
- **QP3** – Qual é o potencial de adoção da TwinCode como ferramenta de pesquisa em ambientes acadêmicos?
- **QP4** - Quais são as funcionalidades mais valorizadas na TwinCode e quais melhorias são prioritárias para otimizar a ferramenta como instrumento de pesquisa?

As questões foram elaboradas de forma a assegurar coerência metodológica e relevân-

cia prática, promovendo uma análise crítica das contribuições potenciais da solução apresentada para o campo da engenharia de software. Essas quatro questões articulam-se de maneira complementar, proporcionando uma base sólida para a avaliação tanto técnica quanto empírica da ferramenta. Elas permitem identificar os diferenciais da ferramenta proposta, ao mesmo tempo em que oferecem subsídios para seu aprimoramento contínuo e sua inserção efetiva em práticas de pesquisa científica.

4.1.3 Validação

Adotou-se uma análise mista (quantitativa e qualitativa) para validar a ferramenta. Os dados quantitativos aferem objetivamente aspectos como facilidade de uso, organização e eficácia no apoio a estudos comparativos. Os dados qualitativos captam percepções e experiências dos participantes. A análise mista oferece evidências complementares para julgar a utilidade da TwinCode e orientar melhorias.

Com o intuito de viabilizar a coleta de dados quantitativos alinhada aos objetivos específicos da pesquisa, elaborou-se um questionário estruturado, concebido como instrumento central de avaliação da ferramenta TwinCode. O questionário foi organizado em cinco seções principais descritas a seguir.

1. Apresentou o Termo de Consentimento Livre e Esclarecido (TCLE), garantindo a adesão ética dos participantes à pesquisa.
2. Concentrou-se na validação de uso da ferramenta, permitindo confirmar se os usuários haviam compreendido suas funcionalidades e interações.
3. Foi dedicada à caracterização do perfil dos participantes, contemplando aspectos como nível de experiência em desenvolvimento de software e familiaridade com conceitos relacionados à qualidade de código.
4. Abordou diretamente a avaliação da ferramenta desenvolvida. Para isso, foram utilizadas questões baseadas em uma escala do tipo *Likert*, que permitiram mensurar, de forma padronizada, a percepção dos usuários quanto a critérios como facilidade de uso, clareza da interface, relevância funcional e aplicabilidade em contextos reais de uso.
5. Reuniu considerações finais dos participantes, com foco em críticas construtivas, sugestões de aprimoramento e comentários abertos sobre a experiência de utilização da TwinCode.

Os participantes foram selecionados por conveniência e convidados a acessar a aplicação TwinCode e, em seguida, responder ao questionário estruturado. Não foram fornecidas

instruções detalhadas sobre o uso da ferramenta, de modo que a interação ocorreu de forma autônoma e espontânea, refletindo a experiência natural de exploração do sistema. Após explorar livremente suas funcionalidades, os participantes preencheram o formulário, que reuniu tanto questões fechadas quanto abertas. As respostas foram coletadas automaticamente e organizadas para análise quantitativa e qualitativa.

Essa estrutura metodológica buscou garantir não apenas a coerência e completude da coleta de dados, mas também a qualidade e profundidade das informações obtidas.

4.2 Estrutura da Ferramenta

Para a construção da TwinCode, optou-se pela utilização de tecnologias de código aberto amplamente consolidadas (PHP/Laravel, Javacript/React CSS/TailwindCSS e MariaDB), uma decisão estratégica que proporcionou diversos benefícios ao longo do desenvolvimento. A adoção dessas tecnologias não apenas conferiu maior flexibilidade ao processo de implementação, permitindo ajustes rápidos e personalizados, como também garantiu a transparência e a acessibilidade do projeto, princípios fundamentais em iniciativas voltadas à pesquisa acadêmica. Além disso, o uso de soluções *open source* facilita a replicabilidade da ferramenta por outros pesquisadores, incentivando sua adaptação e evolução em diferentes contextos.

4.2.1 Arquitetura e Tecnologias

A TwinCode foi desenvolvido com arquitetura monolítica, decisão orientada pela simplicidade de desenvolvimento, implantação unificada e facilidade de depuração. Conforme Blinowski *et al.* (2022), essa abordagem oferece vantagens significativas em projetos de escopo controlado, permitindo construção, testes e *deployment* como unidade coesa. Embora Blinowski *et al.* (2022) reconheçam limitações em sistemas de grande escala, no contexto específico da TwinCode a arquitetura monolítica é adequada e estratégica, facilitando a manutenção de padrões consistentes e simplificando a depuração (BREKALO; SEDLAREVIĆ, 2024).

A ferramenta implementa separada entre *front-end* e *back-end*. Para o *back-end* foi utilizado PHP[1] na versão 8.4, essa linguagem foi escolhida por seu ecossistema consolidado e comunidade ativa. Laravel[2] (versão 12) foi adotado, como *framework*, pela solidez de sua arquitetura *Model*, *View* e *Controller* (MVC), que favorece organização modular do código

[1] Disponível em: <https://www.php.net> Acesso em: 4 mai. 2025
[2] Disponível em: <https://laravel.com/> Acesso em: 4 mai. 2025

e facilita a replicabilidade dos experimentos. Neste contexto, a estrutura do Laravel permite desenvolvimento e validação dos componentes do *back-end*, alinhando-se aos princípios de rigor metodológico acadêmico.

Para o *front-end* (interface do usuário), foram adotados React[3] (versão 18.2.0) e Tailwind CSS[4] (versão 4.1.5). O React possibilitou estruturação em componentes modulares e reutilizáveis, facilitando manutenção e testabilidade. O Tailwind CSS, com sua abordagem *utility-first*, proporcionou controle granular sobre a estilização através de classes utilitárias aplicadas diretamente no *HyperText Markup Language* (HTML), eliminando a necessidade de folhas de estilo extensas. O processo de *build* remove automaticamente *Cascading Style Sheets* (CSS) não utilizado, otimizando o tamanho dos arquivos e melhorando os tempos de carregamento.

O sistema de banco de dados utiliza MariaDB[5] versão 11, solução relacional *open source* escolhida por sua estabilidade, licença permissiva adequada ao contexto acadêmico, e compatibilidade nativa com PHP e Laravel. Essa integração simplifica o desenvolvimento e assegura desempenho consistente em operações de manipulação e consulta de dados.

O ambiente de desenvolvimento utiliza Docker integrado ao Laravel Sail[6], proporcionando configuração padronizada e reproduzível de todos os serviços necessários (servidor web, banco de dados e ambiente PHP). Essa containerização elimina inconsistências ambientais e facilita a replicabilidade dos experimentos, aspectos fundamentais para pesquisas científicas.

4.2.2 Módulos e Interfaces

A TwinCode possui três níveis de acesso que atendem diferentes perfis de usuários: (i) seção pública com informações gerais sobre a ferramenta; (ii) área de acesso por *token* para participantes de pesquisas, permitindo submissão de respostas e visualização de comparações conforme o delineamento experimental; e (iii) área restrita para pesquisadores e administradores, controlada por autenticação com credenciais pré-cadastradas. Esta estrutura garante segurança dos dados experimentais e controle adequado de permissões.

A área restrita concentra-se em dois módulos principais. O módulo de usuários gerencia as contas cadastradas na aplicação, incluindo operações de criação, edição, visualização e remoção de usuários, além de controle de permissões administrativas. O módulo de pesquisas

[3] Disponível em: <https://react.dev/> Acesso em: 4 mai. 2025
[4] Disponível em: <https://tailwindcss.com/> Acesso em: 4 mai. 2025
[5] Disponível em: <https://mariadb.org/> Acesso em: 4 mai. 2025
[6] Disponível em: <https://laravel.com/docs/12.x/sail> Acesso em: 5 mai. 2025

Figura 4 – Interface do módulo de pesquisas: ambiente integrado para gerenciamento de investigações científicas

Fonte: Elaborado pelo autor (2025)

constitui o núcleo da ferramenta, oferecendo ambiente integrado para gerenciamento completo das investigações científicas. Suas funcionalidades incluem: criação de novas pesquisas, cadastro e edição de questionários, inserção de trechos de código para análise comparativa, vinculação de questionários específicos a cada comparação, e geração de relatórios consolidados com os dados coletados.

A Figura 4 apresenta a interface do módulo de pesquisas, que permite cadastrar novos estudos, realizar buscas em pesquisas registradas e acessar a listagem completa dos trabalhos armazenados. O design prioriza simplicidade e facilidade de uso, oferecendo navegação intuitiva que simplifica o fluxo das atividades de pesquisa e a organização sistemática dos dados. A Figura 5 apresenta a interface de visualização de trechos de código e consolida informações essenciais como metadados, descrições e classificações dos códigos cadastrados. Além da visualização, oferece funcionalidades de edição dos dados, remoção controlada de elementos e vinculação de questionários específicos às comparações. Esta abordagem integrada fortalece o controle metodológico e a rastreabilidade dos elementos utilizados na avaliação, contribuindo para a robustez da investigação através da sistematização de processos que tradicionalmente requerem coordenação manual entre múltiplas ferramentas.

Figura 5 – Interface de gerenciamento de trechos de código: visualização sistemática e controle metodológico

Fonte: Elaborado pelo autor (2025)

4.2.3 Exemplo de Uso

Nesta seção é apresentado um exemplo prático da TwinCode utilizando dois trechos de código. O exemplo ilustra como a ferramenta pode ser empregada na comparação de códigos e na coleta de percepções sobre qualidade do código, destacando funcionalidades voltadas à análise de legibilidade e manutenibilidade por desenvolvedores. A Figura 6 apresenta dois trechos de código PHP com implementações de uma função de validação de dados com complexidades ciclomáticas distintas. O Código 1 apresenta complexidade ciclomática 3, utilizando estrutura linear com validações sequenciais. O Código 2 possui complexidade ciclomática 7, implementando estruturas aninhadas com múltiplas condições. Embora ambos executem a mesma funcionalidade, as diferenças estruturais permitem investigar como variações na complexidade afetam a percepção de qualidade dos desenvolvedores. Note que os trechos de código possuem numeração de linhas sincronizada, facilitando a referência precisa durante a análise. Os mecanismos de destaque visual identificam automaticamente as principais diferenças estruturais, como observado na comparação entre as linhas 2-9 do Código 1 e as linhas 2-22 do Código 2. Recursos adicionais incluem ajuste dinâmico do tamanho da fonte, permitindo personalização da visualização conforme as necessidades do participante.

A Figura 7 ilustra o sistema de questionários que combina perguntas: (i) em escala

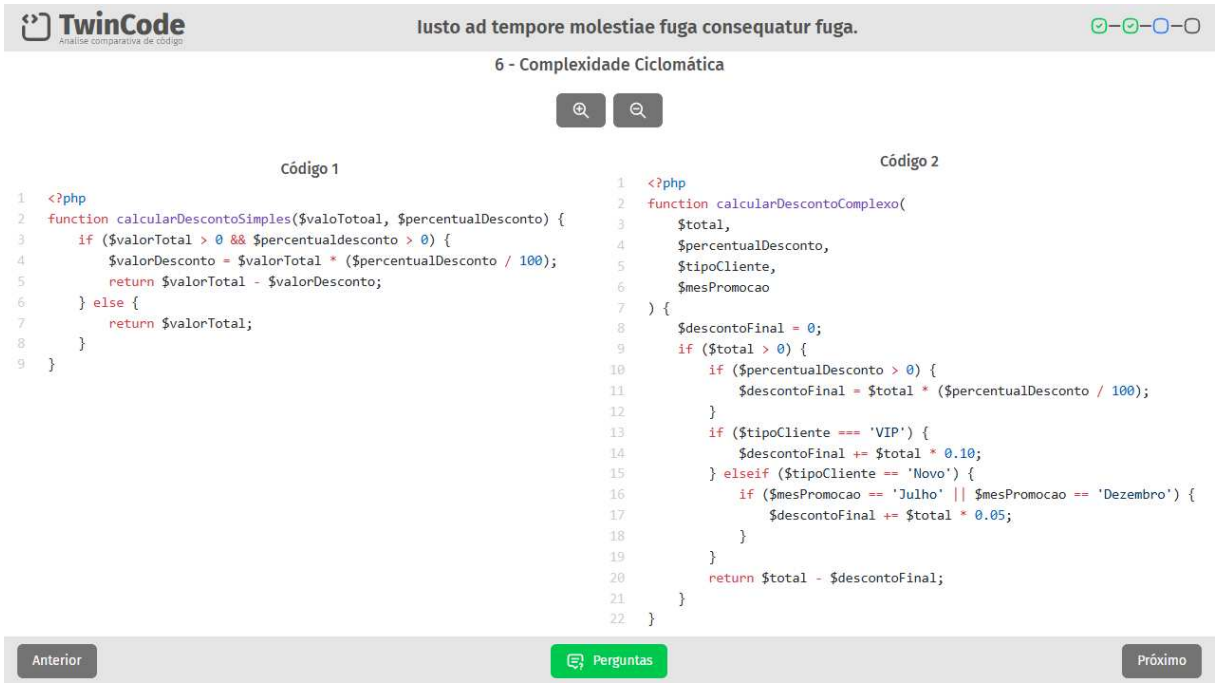


Figura 6 – Interface de comparação: visualização lado a lado de códigos com diferentes complexidades ciclomáticas

Fonte: Elaborado pelo autor (2025)

Likert, (ii) objetivas de única escolha, (iii) objetivas de múltipla escolha e (iv) subjetivas com campo de resposta aberta. As questões quantitativas abordam aspectos específicos como legibilidade (*"Quão fácil é compreender a funcionalidade do Código 1"*) e manutenibilidade (*"O Código 1 é mais manutenível que o Código 2?"*). Para a questão de múltipla escolha aborda características de qualidade de software (*Qual das seguintes características de qualidade de software você acredita que foi mais impactada pelas diferenças entre o Código 1 e Código 2?*). O campo de comentários permite capturar justificativas e observações adicionais dos participantes. Esta abordagem híbrida possibilita tanto análises quantitativas das avaliações quanto análise qualitativa dos comentários, enriquecendo a compreensão sobre os critérios utilizados pelos desenvolvedores na avaliação de qualidade.

A ferramenta permite personalizar os questionários para cada par de códigos, adaptando as perguntas aos objetivos da pesquisa. Pesquisadores podem criar estudos focados em diferentes aspectos da qualidade, como por exemplo para detecção de *code smells*, avaliação de práticas de refatoração, ou análise de padrões de codificação. Os dados coletados são automaticamente organizados em relatórios que facilitam a análise posterior, incluindo estatísticas descritivas das respostas quantitativas e categorização dos comentários qualitativos. Este exemplo demonstra o potencial da TwinCode para apoiar estudos empíricos sobre qualidade de código, oferecendo ambiente controlado para comparações sistemáticas e coleta estruturada de dados. A

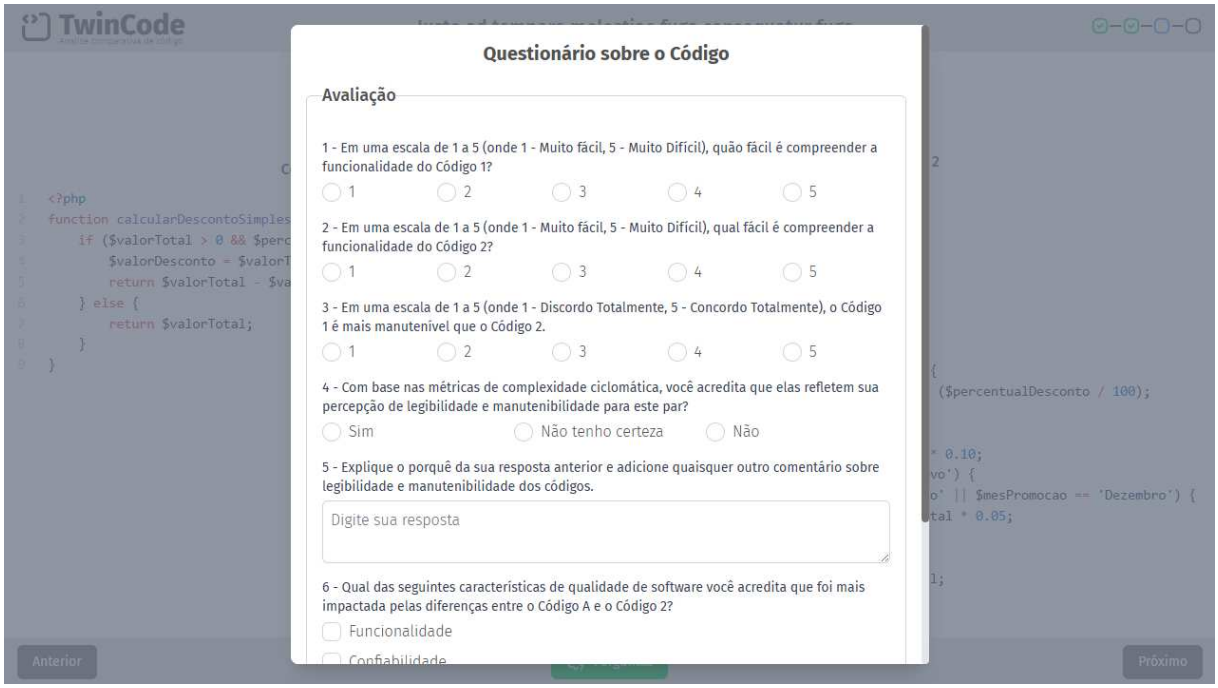


Figura 7 – Sistema de questionários: combinação de escalas Likert e campos abertos para coleta de dados

Fonte: Elaborado pelo autor (2025)

integração entre visualização comparativa e instrumentos de coleta em uma única plataforma simplifica a condução de experimentos e contribui para a sistematização de pesquisas na área de engenharia de software.

4.3 Validação e Resultados

Esta seção apresenta os resultados da avaliação da TwinCode quanto à facilidade de uso, à aderência funcional para estudos empíricos e ao potencial de adoção em ambientes acadêmicos. A ferramenta TwinCode foi avaliada por acadêmicos e profissionais da área de desenvolvimento por meio de um formulário com 21 questões.

A Tabela 9 sintetiza os resultados descritivos das afirmativas avaliadas no questionário, apresentando as medidas de tendência central e dispersão. De forma geral, observa-se que todas as médias situaram-se acima de 3,7 em uma escala de 1 a 5, com destaque para a percepção de potencial de adoção da TwinCode em ambientes acadêmicos (média 4,42; mediana 5). Os itens associados à clareza da interface, à navegação e às funcionalidades de visualização de código também receberam avaliações positivas, embora com variações nos desvios-padrão que refletem diferentes percepções individuais. Esses dados fornecem a base quantitativa para a análise das questões de pesquisa (QP1, QP2, QP3 e QP4), discutidas nas subseções seguintes.

Tabela 9 – Estatísticas descritivas das afirmativas do questionário

Afirmativa	Média	Mediana	DP
A interface da ferramenta é clara e fácil de usar.	3.75	4	1.06
A navegação entre os diferentes módulos da ferramenta é intuitiva.	3.83	4	0.72
A visualização dos pares de código lado a lado é adequada para comparação.	4.08	5	1.56
A numeração de linhas e o realce de sintaxe (syntax highlighting) melhoram a compreensão do código.	4.00	5	1.60
A ferramenta permite ajustar o tamanho da fonte de forma útil para a leitura.	3.75	4	1.29
A criação de pares de código é simples e funcional.	4.08	4	1.31
A ferramenta permite cadastrar questionários específicos para cada comparação de código.	4.17	5	1.03
A associação entre comparações de código e questionários é eficiente.	3.75	4	0.87
O formulário de questionário apresenta as perguntas de maneira clara e objetiva.	4.08	4	1.00
A ferramenta é adequada para estudos que avaliam qualidade de código.	4.17	5	1.03
A TwinCode possui potencial para ser utilizada em pesquisas acadêmicas com desenvolvedores.	4.42	5	0.90

Fonte: Elaborado pelo autor (2025)

4.3.1 Caracterização dos Participantes

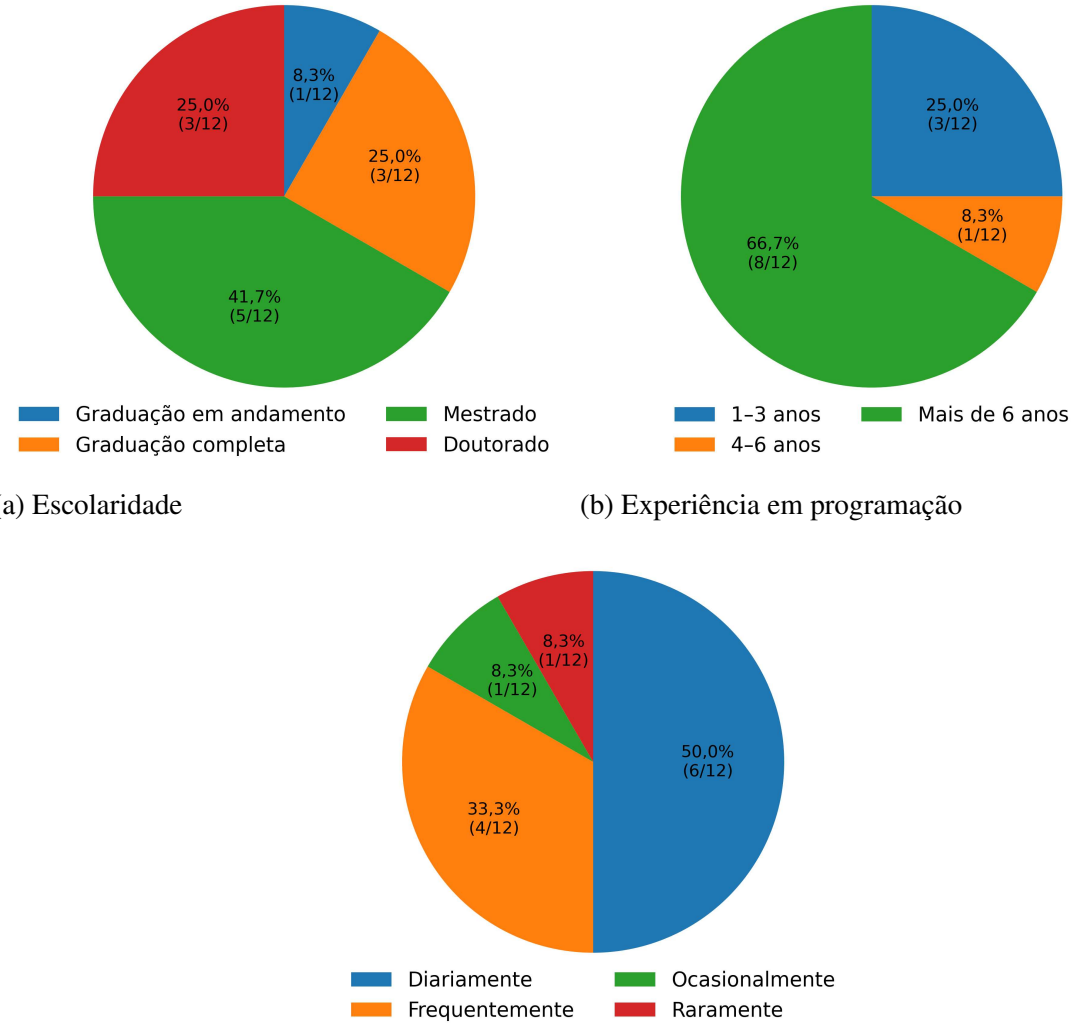
A amostra foi composta por 12 participantes. A caracterização considerou (i) *escolaridade*, (ii) *tempo de experiência em programação* e (iii) *frequência de contato com código* no cotidiano. Adicionalmente, registrou-se a experiência prévia em estudos sobre qualidade de código para contextualizar a interpretação dos resultados.

No que diz respeito à *escolaridade* (Figura 8a), nota-se uma predominância de participantes com pós-graduação *stricto sensu* (66,7%), distribuídos entre **mestrado** (41,7%; 5/12) e **doutorado** (25,0%; 3/12). Outros três participantes possuíam **graduação completa** (25,0%), enquanto apenas um ainda estava em **graduação em andamento** (8,3%). Esse quadro revela um grupo fortemente qualificado, com trajetória acadêmica que tende a favorecer maior familiaridade com metodologias de pesquisa e a oferecer respostas mais consistentes. Por outro lado, a concentração em níveis avançados de escolaridade limita a extrapolação dos resultados para públicos menos titulados.

Quando se observa o *tempo de experiência em programação* (Figura 8b), percebe-se que dois terços da amostra relataram possuir **mais de seis anos** de prática (66,7%; 8/12). Outros três participantes situaram-se na faixa de **1 a 3 anos** (25,0%), e apenas um declarou experiência entre **4 e 6 anos** (8,3%). Esse cenário evidencia que a amostra é majoritariamente composta por desenvolvedores sêniores, com repertório suficiente para avaliar comparações de código com segurança, mas com pouca representação de perfis mais iniciantes.

A *frequência de contato com código* no cotidiano reforça esse perfil. Metade dos participantes afirmou programar **diariamente** (50,0%; 6/12), enquanto um terço declarou contato

frequente (33,3%; 4/12). Apenas um relatou interação **ocasional** (8,3%) e outro **rara** (8,3%). Esse padrão de exposição contínua à prática da programação sugere um grupo acostumado a lidar com código no dia a dia, o que reduz riscos de ruídos de interpretação e fortalece a validade da avaliação da ferramenta.



(a) Escolaridade

(b) Experiência em programação

(c) Frequência de contato com código

Figura 8 – Caracterização dos participantes

Fonte: Elaborado pelo autor (2025)

No que se refere à *área de formação*, após a normalização dos rótulos, percebe-se uma forte concentração em cursos ligados à Computação e às Engenharias. Destacam-se *Ciência(s) da Computação*, que responde por 41,7% da amostra (5 de 12 participantes), seguida por *Análise e Desenvolvimento de Sistemas* e *Sistemas de Informação*, ambos com 16,7% (2/12 cada). Em menor proporção, aparecem formações em *Engenharia da Computação*, *Engenharia de Software* e *Tecnologia em Mecatrônica Industrial*, cada uma representando 8,3% da amostra

(1/12). Esse quadro mostra um grupo majoritariamente oriundo de áreas técnico-científicas, diretamente relacionadas ao desenvolvimento e à avaliação de software, o que reforça a adequação do perfil para analisar comparações de código e avaliar instrumentos voltados à qualidade.

De modo geral, trata-se de uma amostra com *alta titulação*, *experiência profissional acumulada* e *contato frequente* com programação, predominando formações da área de computação e engenharias. Esse perfil contribui para a validade interna da avaliação da ferramenta, por refletir critérios de desenvolvedores experientes, embora exija cautela quanto à validade externa, sobretudo para públicos com menor escolaridade ou menos vivência prática. Como contexto adicional, 58,3% dos participantes (7/12) já haviam participado de estudos ou avaliações relacionados à qualidade de código, o que indica familiaridade prévia com o tema. Tal característica pode influenciar positivamente a profundidade das respostas, sem comprometer a interpretação dos achados quando consideradas essas limitações.

4.3.2 Facilidade de Uso Percebida da Interface (QP1)

A análise da facilidade de uso da TwinCode evidenciou percepções positivas e consistentes em relação à clareza da interface, à navegação e às funcionalidades voltadas à visualização de código. Conforme apresentado na Tabela 9, a afirmativa *"A interface da ferramenta é clara e fácil de usar"* alcançou média de 3,75 (mediana 4, Desvio Padrão (DP) 1,06), sugerindo concordância geral, ainda que com alguma variação entre os respondentes. De forma semelhante, a *"navegação entre os diferentes módulos da ferramenta é intuitiva"* obteve média de 3,83 (mediana 4, DP 0,72), reforçando a percepção de que o fluxo de interação é satisfatório, mesmo sem consenso absoluto. No que se refere às funcionalidades de visualização, os resultados foram ainda mais expressivos. A *"visualização dos pares de código lado a lado"* registrou média de 4,08, com mediana máxima de 5 (DP 1,56), sendo o item mais valorizado da seção. O recurso de *"numeração de linhas e realce de sintaxe"* apresentou média de 4,00 (mediana 5, DP 1,60), confirmando sua importância para a legibilidade e compreensão do código. Já a possibilidade de *"ajustar o tamanho da fonte"* recebeu média de 3,75 (mediana 4, DP 1,29), sinalizando utilidade, mas também evidenciando espaço para ajustes ergonômicos.

As respostas qualitativas dialogam com esses resultados. A comparação lado a lado, acompanhada da numeração e do realce de sintaxe, foi citada repetidamente como o recurso mais útil para apoiar a leitura e a análise. Além disso, os questionários integrados às comparações foram destacados pela capacidade de organizar e sistematizar a coleta de dados. Entre as

sugestões de aprimoramento, sobressaíram a necessidade de simplificar o processo de vinculação entre pares de código e questionários, garantir persistência das configurações visuais (como fonte e tema) e disponibilizar tutoriais curtos para o *onboarding*. Tais apontamentos indicam que, embora a facilidade de uso já seja considerada satisfatória, há espaço para evoluções que tornem a experiência mais fluida e intuitiva.

Resumo da resposta da QP1. Os resultados quantitativos e qualitativos apontam que a TwinCode apresenta um bom nível de facilidade de uso percebida, com destaque para as funcionalidades de comparação e legibilidade de código. Assim, conclui-se que a ferramenta oferece uma experiência clara e funcional, embora dependa de ajustes ergonômicos e de melhorias no fluxo de interação para alcançar maior eficácia.

4.3.3 Eficiência Funcional para Estudos Empíricos (QP2)

Para responder à QP2, a análise da Tabela 9 indica que a TwinCode atende aos requisitos funcionais necessários para a condução de estudos empíricos sobre qualidade de código. A afirmativa *"A criação de pares de código é simples e funcional"* apresentou média de 4,08 (mediana 4,5; DP 1,31), o que revela que a maioria dos participantes percebe essa etapa como prática e intuitiva. Resultado semelhante foi observado no item *"A ferramenta permite cadastrar questionários específicos para cada comparação de código"*, que obteve média de 4,17 (mediana 5; DP 1,11), sinalizando forte concordância quanto à relevância desse recurso para a organização de experimentos. Entretanto, a *"associação entre comparações de código e questionários"* recebeu avaliação menos favorável, com média de 3,75 (mediana 4; DP 1,36). Embora considerada funcional, essa etapa foi descrita como mais trabalhosa e menos fluida em comparação às demais, evidenciando um gargalo que pode comprometer a experiência de uso em cenários mais complexos.

As respostas qualitativas ajudam a compreender melhor a eficiência funcional da ferramenta. Os participantes reconheceram a importância da integração entre pares de código e questionários, mas pediram que o processo de vinculação fosse mais direto e que houvesse maior transparência sobre o estado do vínculo. Outras sugestões incluíram a possibilidade de exportar relatórios e a adoção de mecanismos de versionamento, ambos vistos como incrementos que poderiam ampliar a eficiência operacional da ferramenta.

Resumo da resposta da QP2. A TwinCode contempla os principais requisitos funcionais esperados em um ambiente de pesquisa empírica, com destaque para a criação de pares e o cadastro de questionários. Contudo, a associação entre esses elementos ainda demanda refinamentos, de modo a reduzir barreiras e garantir maior fluidez no uso em estudos de maior escala.

4.3.4 Potencial de Adoção em Ambientes Acadêmicos (QP3)

Os dados da Tabela 9 indicam que a TwinCode apresenta elevado potencial de adoção em contextos acadêmicos. A afirmativa *"A ferramenta é adequada para estudos que avaliam qualidade de código"* alcançou média de 4,17 (mediana 4; DP 0,83), evidenciando concordância consistente quanto à sua aplicabilidade em investigações empíricas. Ainda mais expressiva foi a avaliação da afirmativa *"A TwinCode possui potencial para ser utilizada em pesquisas acadêmicas com desenvolvedores"*, que obteve média de 4,42 (mediana 5; DP 0,79), com predominância de respostas concentradas nos níveis mais altos da escala. Esses resultados mostram que as participantes não apenas reconhecem a adequação funcional da ferramenta, mas também projetam sua utilização em cenários de ensino e pesquisa, considerando-a um recurso com potencial de integração em práticas de avaliação e experimentação. As respostas qualitativas reforçam essa percepção: além de valorizar a comparação de código lado a lado e os questionários integrados, algumas participantes sugeriram a inclusão de funcionalidades adicionais, como exportação de relatórios e mecanismos de versionamento, que poderiam ampliar ainda mais a viabilidade de uso em ambientes institucionais.

Resumo da resposta da QP3. Em síntese, a TwinCode é percebida como uma ferramenta promissora para adoção acadêmica, oferecendo suporte adequado a estudos empíricos e apresentando potencial de integração tanto em atividades de ensino-aprendizagem quanto em pesquisas colaborativas com desenvolvedores.

4.3.5 Funcionalidades Valorizadas e Prioridades de Melhoria (QP4)

A análise qualitativa das respostas abertas do questionário permite identificar tanto as funcionalidades mais valorizadas da TwinCode quanto as melhorias consideradas prioritárias. Entre os recursos destacados, a *comparação de código lado a lado* aparece como o núcleo

da experiência de uso, quase sempre associada à *numeração de linhas* e ao *realce de sintaxe*, elementos reconhecidos como essenciais para garantir clareza e legibilidade na análise. O módulo de *questionários integrados às comparações* também foi frequentemente mencionado, sendo valorizado por oferecer um meio estruturado de coletar dados em experimentos de refatoração e avaliação de qualidade de código.

Paralelamente, surgiram sugestões que apontam para gargalos na experiência de uso. O aspecto mais recorrente foi a *associação entre pares de código e questionários*, considerada útil, mas pouco intuitiva, o que levou à demanda por simplificação do fluxo e maior visibilidade do estado de vínculo. Outras observações incluíram a necessidade de *persistência de configurações visuais* (como fonte e tema), maior fluidez na rolagem e tutoriais breves para facilitar o *onboarding*, de modo a reduzir a curva de aprendizagem. Também foram sugeridas funcionalidades adicionais, como *exportação de relatórios em PDF*, mecanismos de *filtros e busca* e recursos de *versionamento de código*, que poderiam ampliar a robustez da ferramenta.

Resumo da resposta da QP4. Em síntese, a TwinCode já reúne um conjunto de funcionalidades reconhecidas como valiosas, sobretudo a visualização lado a lado com realce de sintaxe e a integração de questionários, mas sua consolidação como ferramenta acadêmica depende de avanços em ergonomia, simplificação de fluxos e inclusão de funcionalidades complementares. Esses pontos configuram um roteiro claro para o direcionamento de futuras versões da ferramenta.

4.4 Trabalhos Relacionados

A análise de ferramentas existentes para comparação de código e estudos empíricos revela diferentes abordagens metodológicas, cada uma com características distintas para um contexto de uso. Esta seção apresenta as principais soluções disponíveis, organizadas por categoria, destacando suas funcionalidades, limitações e adequação para pesquisas acadêmicas.

Santos e Gerosa (2018) desenvolveram uma aplicação web, construída *ad hoc* para o experimento, a fim de investigar o impacto de boas práticas de codificação na percepção de qualidade por desenvolvedores, priorizando o controle da apresentação dos trechos e o registro de variáveis como resolução de tela e tempo de resposta. Não há informação sobre abertura do código, o que limita seu reuso. A ferramenta exibe pares de *snippets* em que, para cada uma das onze práticas analisadas, um trecho seguia a recomendação e o outro a violava, permitindo

mensurar empiricamente efeitos sobre legibilidade e preferências. Em contraste, a TwinCode generaliza esse suporte metodológico ao integrar a visualização lado a lado com questionários configuráveis por comparação e relatórios integrados, viabilizando reuso entre estudos.

Frick *et al.* (2018) introduzem o **DiffViz**, uma ferramenta interativa projetada para aprimorar a visualização de alterações em código a partir de uma arquitetura modular e independente do algoritmo de comparação. O sistema combina um *back-end* em Java, responsável pela análise dos trechos, com um *front-end* em JavaScript, voltado à navegação, permitindo integrar diferentes algoritmos de *diff*, como GumTree, MTDIFF e IJM. Entre as funcionalidades oferecidas estão a visualização lado a lado, o realce de mudanças, o mapeamento de nós alterados e a possibilidade de importar código manualmente, por meio de repositórios GitHub ou arquivos JSON. Embora o DiffViz se destaque por fornecer uma inspeção detalhada e flexível das diferenças entre versões, seu propósito principal permanece centrado na análise precisa das modificações. Em contraste, a TwinCode amplia essa perspectiva ao orientar-se para a condução de estudos empíricos, integrando a comparação lado a lado com questionários configuráveis e relatórios estruturados, de modo a favorecer a coleta e a sistematização de percepções em pesquisas acadêmicas.

Soluções comerciais como Pretty Diff[7], CodeScene[8] e Diffchecker[9] são amplamente utilizadas no contexto profissional. O Pretty Diff oferece visualização lado a lado com suporte a múltiplas linguagens, realce de sintaxe e formatação automatizada, com foco em análise sintática detalhada. O CodeScene integra métricas de complexidade, histórico de alterações e *hotspots* de manutenção, fornecendo uma abordagem visual orientada por dados históricos para decisões sobre qualidade de *software*. O Diffchecker, por sua vez, é uma solução mais simples e de uso imediato, voltada à comparação de trechos de código, textos ou arquivos em geral, destacando diferenças de forma clara, mas sem oferecer recursos avançados de análise ou integração com fluxos de pesquisa. Embora úteis, essas ferramentas não foram concebidas para experimentos controlados ou estudos empíricos acadêmicos e não oferecem, de forma nativa, coleta estruturada de dados, questionários personalizados ou controle de variáveis experimentais, lacuna que a TwinCode busca suprir.

A Tabela 10 apresenta a comparação entre as principais ferramentas identificadas. Essa síntese permite visualizar quais funcionalidades são oferecidas em cada contexto e quais

[7] Disponível em: <https://prettydiff.com/3/> Acesso em: 12 mai. 2025
[8] Disponível em: <https://codescene.com/> Acesso em: 12 mai. 2025
[9] Disponível em: <https://www.diffchecker.com/> Acesso em: 12 mai. 2025

lacunas permanecem abertas para aplicações em pesquisas empíricas, lacunas que a TwinCode busca preencher. Observa-se que a TwinCode se diferencia por reunir, em um único ambiente, comparação visual, questionários configuráveis, foco acadêmico, coleta estruturada e disponibilidade aberta, atendendo de forma integrada a demandas de replicabilidade, instrumentação e gerenciamento de dados para estudos empíricos. Importante destacar que, ao contrário de ferramentas como DiffViz, Pretty Diff ou Diffchecker, a TwinCode não busca evidenciar diferenças entre trechos, mas sim disponibilizar os códigos lado a lado como suporte à avaliação da qualidade, mantendo o foco na percepção dos desenvolvedores em contextos controlados de pesquisa.

Tabela 10 – Comparação entre ferramentas de análise de código

Ferramenta	**Comparação Visual**	**Questionários**	**Foco Acadêmico**	**Open Source**	**Coleta Estruturada**
Santos e Gerosa (2018)	✔	✔	✔	N/I	✔
DiffViz (FRICK *et al.*, 2018)	✔	✖	✖	✔	✖
Pretty Diff	✔	✖	✖	✖	✖
CodeScene	✔	✖	✖	✖	✖
Diffchecker	✔	✖	✖	✖	✖
TwinCode	✔	✔	✔	✔	✔

Fonte: Elaborado pelo autor (2025)

Esta combinação de características posiciona a TwinCode como uma contribuição específica para a comunidade de pesquisa em Engenharia de Software, preenchendo lacuna identificada na disponibilidade de ferramentas especializadas para estudos empíricos sobre percepção humana de qualidade de código.

4.5 Ameaças à Validade da Pesquisa

Esta seção apresenta as ameaças à validade do estudo, organizadas de acordo com a proposta de Wohlin *et al.* (2012): validade interna, externa, de construção e de conclusão.

Validade Interna. Refere-se ao quanto os efeitos observados podem ser atribuídos de fato à TwinCode. Foi identificado cinco ameaças à validade interna. A *aplicação do questionário logo após a demonstração* pode ter sofrido com vieses de autorrelato, como desejabilidade social, e também com viés de memória. No entanto, a coleta imediata reduziu esquecimentos, e o anonimato assegurou liberdade de opinião, atenuando a pressão por respostas socialmente aceitas. Outro ponto é a *ausência de grupo controle* ou de comparação com outra ferramenta, o que impede isolar efeitos específicos da solução. Ainda assim, optou-se por um delineamento

exploratório, adequado para uma primeira avaliação de aceitabilidade, deixando comparações diretas para etapas futuras. Também é possível que o *posicionamento dos blocos do questionário* tenha gerado efeitos de ordem, já que os itens de facilidade de uso foram respondidos antes dos de adoção. A decisão de manter essa sequência buscou refletir o fluxo natural da experiência de uso — primeiro interagir, depois refletir sobre adoção — mas contrabalançamentos poderão ser testados em replicações. Outro fator a considerar é o *efeito novidade*: por se tratar de uma solução inédita no contexto dos participantes, existe a possibilidade de avaliações mais favoráveis pelo simples caráter inovador. Esse risco pode ser reduzido em estudos futuros com sessões repetidas, que tendem a diluir o entusiasmo inicial. Por fim, a *alta participação dos sujeitos do estudo em estudos de qualidade de código* (58,3%), isso poderia calibrar expectativas. Ainda que isso introduza familiaridade, também garante que os julgamentos foram feitos por pessoas com repertório adequado, reduzindo interpretações equivocadas.

Validade Externa. Relaciona-se à possibilidade de generalizar os resultados para outros contextos. Identificou-se quatro ameaças externas à validade do estudo. A *qualidade da amostra*, composta majoritariamente por profissionais de pós-graduação com mais de seis anos de experiência, não representa iniciantes ou perfis menos experientes. Apesar disso, essa característica confere robustez às respostas, já que foram baseadas em critérios técnicos sólidos. Outro ponto é que o *ambiente de demonstração* não reproduz integralmente situações reais de pesquisa, com maior complexidade e pressão de tempo. Em contrapartida, a aplicação em condições controladas assegurou homogeneidade e reduziu fatores externos que poderiam distorcer a avaliação. O *tempo restrito de interação* também pode ter subestimado aspectos ergonômicos do uso prolongado. Ainda assim, a coleta imediata capturou percepções autênticas sobre a primeira experiência com a ferramenta. Além disso, *diferenças de infraestrutura local* e de curva de aprendizagem podem limitar a replicação em contextos distintos, mas esse risco é parcialmente mitigado pelo fato de a TwinCode ser baseada em tecnologias abertas e de fácil acesso. Futuras replicações podem ampliar o alcance por meio de amostragens mais diversas, estudos em campo, avaliações longitudinais e definição de requisitos mínimos de ambiente para reduzir variações técnicas.

Validade à Construção. Diz respeito ao quanto os instrumentos realmente capturam os construtos de “facilidade de uso” e “adequação funcional”. Foram identificadas 4 ameaças à construção do estudo. Como os itens foram elaborados especificamente para este estudo, sem o uso de escalas padronizadas, existe o *risco de lacunas na cobertura*. Esse risco foi parcialmente

mitigado pelo embasamento teórico utilizado na formulação dos itens e pela consistência interna obtida (alfa de Cronbach = 0,900), que assegura confiabilidade nas medidas. Outro ponto é que a *avaliação se concentrou em percepções imediatas*, sem métricas comportamentais como tempo em tarefa ou *logs* de uso. A coleta imediata reduziu a influência da memória, mas reconhece-se que medidas objetivas fortaleceriam as conclusões. Também se deve considerar a *seleção dos trechos de código usados na demonstração*: ainda que representativos, eles não abrangem toda a diversidade de cenários possíveis, o que pode limitar a abrangência da avaliação. Esse risco pode ser reduzido em replicações com conjuntos mais variados de exemplos, cobrindo diferentes domínios e níveis de complexidade. Além disso, a *dependência de um único método*, o questionário, pode ter introduzido variância de método comum. Para reduzir esse efeito, foram incluídas respostas abertas, que trouxeram nuances qualitativas importantes. Em estudos futuros, recomenda-se a combinação de escalas consagradas, validação psicométrica e triangulação com métricas observacionais.

Validade de Conclusão. Refere-se à solidez das inferências estatísticas. Foram identificadas 3 ameaças a validade da conclusão do estudo. O *tamanho da amostra* ($n = 12$) naturalmente limita o poder dos testes e a precisão das estimativas. Esse limite foi reconhecido desde o início, e por isso as análises foram predominantemente descritivas, apoiadas em médias, medianas e desvios padrão, de modo a oferecer transparência e não induzir a interpretações infladas. As *múltiplas comparações* entre itens aumentam o risco de achados espúrios; para reduzir esse efeito, evitou-se enfatizar resultados isolados, dando maior atenção a padrões consistentes e ao cruzamento com os dados qualitativos. Por fim, a opção por *análises descritivas* restringe a força das conclusões, mas foi adequada ao caráter exploratório desta etapa de validação. Em futuras replicações, recomenda-se ampliar o número de participantes, incluir intervalos de confiança e tamanhos de efeito, bem como adotar testes robustos, como *bootstrapping*, com delineamentos pré-registrados. Assim, embora os resultados devam ser interpretados com cautela, eles oferecem indícios confiáveis dentro do escopo desta investigação inicial.

Em suma, embora esta etapa apresente limitações inerentes ao delineamento exploratório, os resultados apontaram médias e medianas favoráveis, consistência interna elevada e alinhamento com as percepções qualitativas. Esses achados indicam que a TwinCode já oferece evidências sólidas de utilidade e aplicabilidade, reforçando sua viabilidade como instrumento de pesquisa. As ameaças aqui discutidas estabelecem não apenas os limites do presente estudo, mas também diretrizes valiosas para ciclos futuros, nos quais será possível ampliar a robustez

inferencial e a generalização dos resultados. Assim, apesar dos desafios, esta investigação constitui um passo importante para consolidar a TwinCode como uma ferramenta acadêmica confiável e em evolução contínua.

4.6 Conclusão

A TwinCode representa uma contribuição concreta para a instrumentalização de estudos empíricos em engenharia de software, ao oferecer uma solução integrada orientada à comparação de trechos de código com suporte à instrumentação por questionários específicos por comparação. A validação de caráter exploratório conduzida nesta etapa, evidenciou pontos fortes no núcleo de *visualização lado a lado*, na numeração de linhas e no *syntax highlighting*, além da boa aceitação de questionários e da criação de pares com baixo atrito. Em paralelo, foram identificadas oportunidades de aprimoramento no fluxo de associação entre comparações e questionários, no design de experiência do usuário e em funcionalidades auxiliares (filtros, exportação e versionamento), estabelecendo base objetiva para evolução incremental da ferramenta.

A convergência entre um desenho metodológico alinhado a protocolos reprodutíveis e uma facilidade de uso satisfatória, ainda em amadurecimento, evidencia que ferramentas acadêmicas podem conciliar robustez científica e aplicabilidade prática. Consideradas as limitações desta etapa como amostra majoritariamente sênior, aplicação autoaplicável e ausência de comparação, os resultados sustentam a viabilidade da TwinCode como instrumento para apoiar investigações controladas sobre qualidade de código, sem prejuízo de aperfeiçoamentos incrementais no design de experiência do usuário e fluxo.

Espera-se que a disponibilização da TwinCode e dos achados desta pesquisa fomente estudos empíricos com maior qualidade metodológica sobre percepção humana de qualidade de código, tema central para compreender fatores associados à produtividade e à sustentabilidade do desenvolvimento de *software*.

Para estudos futuros, recomenda-se: (a) validação em contextos reais por meio de delineamentos longitudinais; (b) avaliação comparativa com ferramentas de referência para quantificar vantagens específicas; (c) ampliação e estratificação da base de usuários (experiência, escolaridade e contexto institucional); (d) incorporação de métricas objetivas de eficiência (tempo em tarefa, taxa de erro, *logs*) em triangulação com percepções; e (e) investigação de fatores que influenciam a adoção de ferramentas acadêmicas especializadas.

Quanto ao desenvolvimento, destacam-se oportunidades como integração com reposi-

tórios amplamente utilizados (p. ex., GitHub, GitLab), implementação de análises automatizadas de métricas de qualidade e ampliação do suporte a diferentes artefatos de *software* além de código-fonte. Tais direções, somadas às melhorias de *User Experience* (UX) e orquestração do fluxo, tendem a robustecer a utilidade científica e a adoção acadêmica da TwinCode.

5 PERCEPÇÃO DE DESENVOLVEDORES SOBRE QUALIDADE DE CÓDIGO REFATORADO POR MODELO DE LINGUAGEM DE GRANDE PORTE

A crescente integração de modelos de linguagem de grande porte (do inglês Large Language Models – LLMs) aos fluxos de desenvolvimento tem ampliado a automação de tarefas como geração e *refatoração* de código, com efeitos diretos sobre práticas de qualidade e produtividade (XUE *et al.*, 2025; LYU *et al.*, 2025). Entretanto, permanece pouco explorado *como* desenvolvedores percebem a qualidade do código alterado por LLMs, especialmente quando a refatoração visa mitigar *code smells*, e *quais critérios* humanos orientam tais julgamentos em cenários reais (CHEN *et al.*, 2021; HE *et al.*, 2025). Com base nessa lacuna, este capítulo investiga a percepção de desenvolvedores brasileiros sobre código refatorado por LLMs, combinando comparações cegas entre versões original/refatorada e entrevistas semi-estruturadas. A motivação é dupla: (i) fornecer evidências empíricas que ajudem equipes a decidir *quando* e *como* incorporar LLMs de forma responsável nos seus fluxos, e (ii) mapear atributos de qualidade valorizados pelos profissionais (p. ex., legibilidade, manutenibilidade e modularidade), gerando insumos para diretrizes de adoção e futuras pesquisas.

Para cobrir esta lacuna, este estudo envolveu sessões de entrevistas individuais de cerca de 45 minutos, nas quais os entrevistados avaliaram cinco pares de códigos (originais e versões refatoradas pelo modelo Qwen2.5-max[1]) por meio da ferramenta TwinCode, que ofereceu visualização lado a lado dos trechos de código comparados. As entrevistas foram gravadas e transcritas e organizadas em cinco blocos: (i) caracterização do perfil do entrevistado, (ii) percepção de qualidade de software, (iii) análise comparativa dos códigos, (iv) discussão sobre ferramentas e reflexões sobre qualidade de código e (v) revelação do uso de IA. A coleta integrou dados quantitativos (preferências declaradas) e qualitativos (justificativas e percepções), analisados pela técnica de *análise de conteúdo* de Bardin (2016). Essa técnica permite identificar critérios de qualidade, expectativas e limitações atribuídas à tecnologia.

Os resultados revelam convergência em torno de características de qualidade de código como legibilidade, manutenibilidade e modularidade, preferência majoritária dos entrevistados (83%) pelos códigos refatorados em sua totalidade e 97% dos artefatos analisados. Essa percepção se manteve quando revelado que o código foi refatorado pela LLM, pois as decisões se fundamentaram em características intrínsecas de qualidade. Ao mesmo tempo, emergiu uma aceitação ponderada das LLMs, no qual os entrevistados informaram que são ferramentas úteis

[1] Disponível em: <https://chat.qwenlm.ai/> Acesso em: 9 abr. 2025

para aumentar produtividade e auxiliar em contextos de pressão temporal, mas que exigem supervisão humana para mitigar riscos como complexidade desnecessária e perda de contexto.

Este trabalho possui quatro contribuições principais, sendo elas:

- **Evidências empíricas inéditas.** Fornecimento de resultados originais no contexto nacional sobre a percepção de desenvolvedores em relação a refatorações realizadas por LLMs;
- **Contribuição metodológica.** Demonstração da utilidade de comparações cegas como abordagem para avaliar qualidade de código sob a ótica humana;
- **Mapeamento de critérios de avaliação.** Identificação de atributos valorizados pelos profissionais, como legibilidade, manutenibilidade e modularidade;
- **Implicações práticas.** Apontamento de oportunidades e limitações para a integração responsável de LLMs em fluxos de desenvolvimento, oferecendo subsídios tanto para a prática profissional quanto para futuras pesquisas na área.

Este capítulo está organizado da seguinte forma. A Seção 5.1 descreve o delineamento metodológico, incluindo seleção de dados, processo de refatoração e protocolos de coleta e análise. A Seção 5.2 apresenta os resultados, estruturados por critérios de qualidade, preferências de código, impacto da revelação sobre uso de IA e expectativas sobre a tecnologia. Seção 5.3 descreve trabalhos relacionados. A Seção 5.4 relata as ameaças à validade do estudo, enquanto a Seção 5.5 sintetiza as conclusões, contribuições, limitações e direções para trabalhos futuros.

5.1 Metodologia

Este estudo fundamenta-se em abordagem qualitativa que combina comparações cegas de código com entrevistas semi-estruturadas para investigar a percepção de desenvolvedores sobre qualidade de código refatorado por LLMs. O delineamento metodológico foi estruturado em quatro componentes: (i) objetivos e questões de pesquisa, (ii) composição do conjunto de dados experimentais do *Qualitas Corpus* (TEMPERO *et al.*, 2010), (iii) detalhamento do processo de refatoração e (iv) seleção dos entrevistados, procedimentos de coleta e análise qualitativa. Esta estruturação visa eliminar vieses relacionados à origem tecnológica e capturar critérios utilizados por desenvolvedores na avaliação de qualidade de *software*.

A Figura 9 apresenta o fluxograma metodológico utilizado no estudo. O processo inicia-se com a definição dos objetivos e questões de pesquisa, etapa fundamental para orientar todas as fases subsequentes. Em seguida, ocorre a definição do *prompt* seguindo as técnicas de White *et al.* (2023) e a seleção do LLM utilizado nas refatorações. Paralelamente, realiza-se

a seleção dos dados a partir do *Qualitas Corpus*, de modo a compor o conjunto de trechos de código a serem analisados. Esta seleção foi feita a partir de 815 classes e métodos classificados com os *code smells Feature Envy*, *Long Method*, *Data Class* e *God Class*. Dessa seleção foram escolhido aleatoriamente 20 casos de cada *code smell*, totalizando 80 trechos de código para uso nas entrevistas. Em seguida os dados são refatorados e armazenados para cruzamento dos pares, código original e refatorado. A partir desses pares, foram elaboradas as comparações cegas, para ocultar dos entrevistados a origem de cada trecho avaliado.

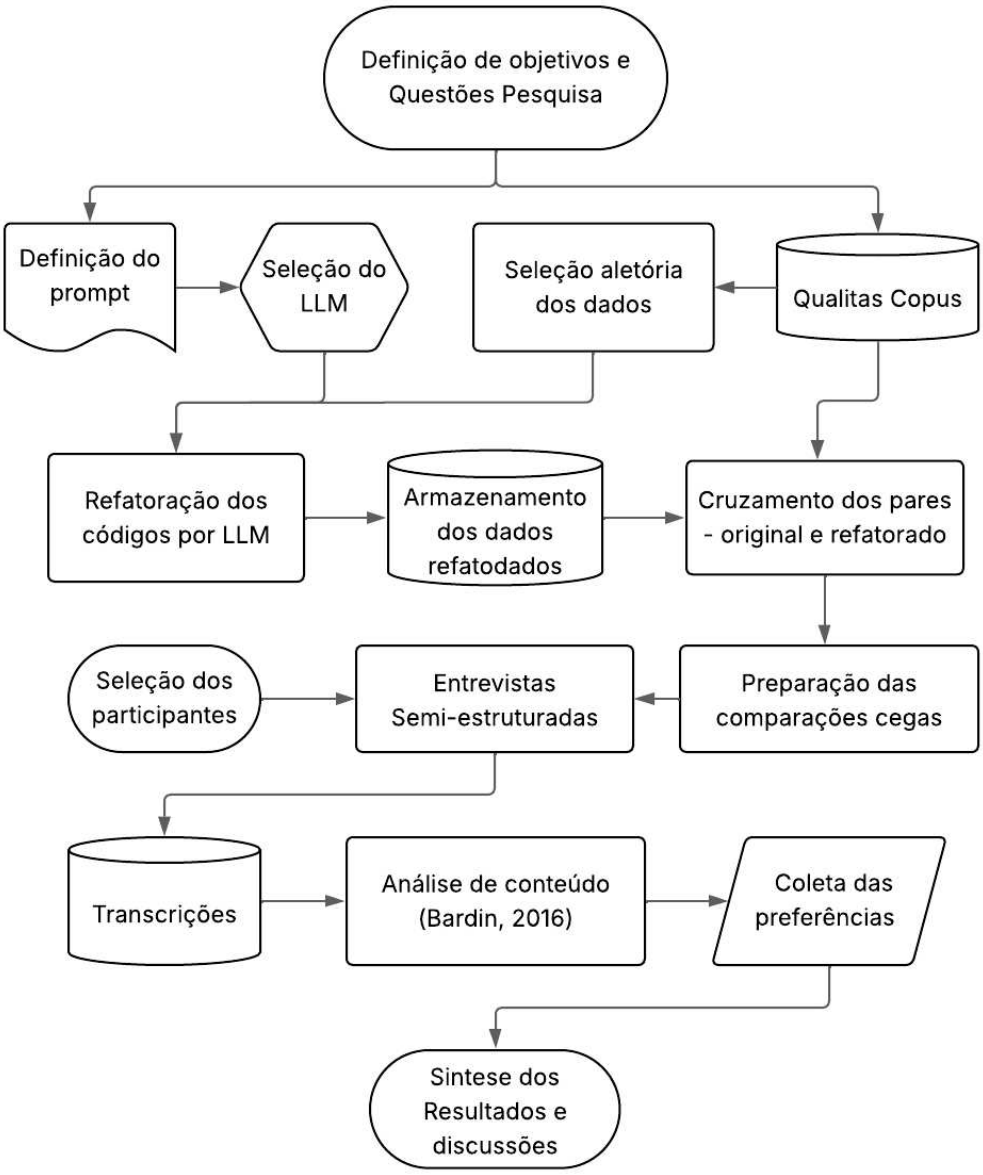


Figura 9 – Fluxo de trabalho proposto
Fonte: Elaborado pelo autor (2025)

Posteriormente, ocorre a seleção dos entrevistados e a aplicação das entrevistas semi-

estruturadas, nas quais os desenvolvedores analisaram os códigos apresentados. Nessa etapa, foram coletadas as preferências e justificativas em relação às versões originais e refatoradas, sendo todas as interações registradas em transcrições. Por fim, os dados coletados foram submetidos à análise de conteúdo proposta por Bardin (2016), possibilitando a categorização e interpretação das percepções manifestadas. A etapa final consistiu na síntese dos resultados e discussões, que integra os achados obtidos e fornece subsídios para a compreensão das contribuições e limitações do uso de LLMs na refatoração de código.

5.1.1 Objetivos da Pesquisa

O objetivo principal desse capítulo é **investigar a percepção de desenvolvedores sobre a qualidade de código refatorado por LLMs**. De forma complementar, definiu-se quatro objetivos específicos.

1. Identificar e categorizar os critérios que desenvolvedores utilizam espontaneamente para avaliar qualidade de código, estabelecendo uma taxonomia conceitual dos fatores que influenciam percepções de qualidade no contexto de desenvolvedores profissionais;
2. Analisar preferências entre versões originais e refatoradas por LLMs por meio de comparações cegas (desenvolvedores não sabem qual é o código refatorado, nem quem refatorou), quantificando e qualificando as escolhas dos desenvolvedores sem viés relacionado à origem do código;
3. Investigar mudanças na percepção de qualidade após revelação sobre uso de inteligência artificial na refatoração, avaliando o impacto de preconceitos tecnológicos nas avaliações dos desenvolvedores;
4. Examinar expectativas, preocupações e perspectivas dos desenvolvedores sobre integração futura de LLMs em processos de melhoria de qualidade de código, identificando fatores facilitadores e barreiras à adoção tecnológica.

A partir dos objetivos delineados para este estudo, foram formuladas quatro questões de pesquisa (QPs) que estruturam a investigação empírica sobre percepção de desenvolvedores em relação ao código refatorado por LLMs. A seguir, são apresentadas cada questão de pesquisa e suas respectivas justificativas.

QP1: Quais critérios desenvolvedores utilizam espontaneamente para avaliar qualidade de código?

Essa questão investiga os parâmetros que os desenvolvedores utilizam de forma natural ao julgar um código. Ao identificar critérios como legibilidade, manutenibilidade, modularidade ou testabilidade, é possível compreender quais dimensões da qualidade são mais valorizadas pelos desenvolvedores na prática.

QP2: Desenvolvedores conseguem identificar diferenças qualitativas entre código original e refatorado por LLMs em comparações cegas?

Nessa questão pesquisa é buscado verificar se, sem influência de preconceitos tecnológicos, os desenvolvedores percebem distinções relevantes entre as duas versões de código. Trata-se de um passo decisivo, pois avalia a efetividade da refatoração automática em termos práticos. Explora-se se as refatorações realizadas pelo LLM geram, de fato, mudanças notáveis e reconhecidas como melhorias.

QP3: Como a revelação sobre uso de IA influencia a percepção de qualidade dos desenvolvedores?

Na questão de pesquisa três, explora-se o papel do viés tecnológico no julgamento de qualidade. Ao comparar as respostas obtidas antes e depois da revelação do uso de IA, verifica-se a existência de resistência ou de confiança no código refatorado pela LLM.

QP4: Quais são as expectativas e preocupações dos desenvolvedores sobre o uso de LLMs para melhoria de qualidade de código?

Para essa questão, busca-se ampliar a análise para além da comparação direta entre códigos, procurando compreender como os profissionais percebem o futuro dessa tecnologia. A investigação de expectativas e preocupações permite construir uma visão equilibrada sobre as possibilidades e os desafios do uso de LLMs.

5.1.2 Conjunto de Dados

O estudo utilizou códigos do *Qualitas Corpus* versão 20130901r[2], proposto por Tempero *et al.* (2010). Neste capítulo são investigados os quatro *code smells* analisados por Fontana *et al.* (2016) e Abdou e Darwish (2024) no Capítulo 3: **God Class**, **Long Method**, **Feature Envy** e **Data Class**. Esses *code smells* foram escolhidos por representarem categorias distintas

[2] Disponível em: <https://qualitascorpus.com/download/> Acesso em: 2 abr. 2025

de problemas: enquanto **God Class** e **Data Class** evidenciam falhas estruturais relacionadas ao desenho das classes, **Long Method** e **Feature Envy** estão associados à implementação e ao uso inadequado de responsabilidades nos métodos. Dessa forma, o conjunto abrange tanto problemas de projeto de alto nível quanto defeitos mais granulares. Para compor a base experimental, foram selecionadas aleatoriamente 20 classes de cada tipo de *code smell*, totalizando uma amostra com 80 artefatos destinados à refatoração e à avaliação dos entrevistados.

A Tabela 11 apresenta em detalhes o processo de seleção e filtragem da amostra utilizada. Inicialmente, foram identificados 815 trechos com ocorrência de *code smells*. Em seguida, por meio de um *script* desenvolvido em Python[3], realizou-se o cruzamento desses trechos com o código-fonte dos sistemas pertencentes ao *Qualitas Corpus*. Esse procedimento permitiu localizar 740 trechos correspondentes, enquanto 75 não puderam ser associados ao respectivo código-fonte.

Tabela 11 – Processo de seleção das amostras para refatoração

Code Smell	**Identificados Inicialmente**	**Após Filtragem**	**Selecionados para Refatoração**
Long Method	140	127	20
Data Class	269	241	20
God Class	266	242	20
Feature Envy	140	130	20
Total	**815**	**740**	**80**

Fonte: Elaborado pelo autor (2025)

Após o processo de refatoração que resultou em 80 pares de código (original e refatorado), foram selecionadas aleatoriamente 5 comparações únicas para cada entrevistado durante as entrevistas. Cada participante analisou um conjunto distinto de código, garantindo diversidade na amostra e evitando viés de familiarização. A Tabela 12 apresenta os 12 sistemas do *Qualitas Corpus* selecionados aleatoriamente para as comparações realizadas nas entrevistas. Esses sistemas foram utilizados por conterem classes e métodos classificados com algum dos *code smells* investigados. Dessa forma, o número de sistemas analisados nesta avaliação foi limitado em função da quantidade de entrevistados considerados neste estudo, descritos na Seção 5.1.4.

[3] Disponível em: <https://github.com/alanfm/dissertation_cap5>

Tabela 12 – Sistemas do *Qualitas Corpus* analisados pelos entrevistados

Sistema	Versão	Domínio	Descrição
Heritrix	1.14.4	Arquivamento Web	Rastreador web desenvolvido pelo Internet Archive para coleta e arquivamento de conteúdo digital
JSPWiki	2.6.3	Wiki/CMS	Mecanismo de Wiki em Java construído com componentes J2EE padrão
mvnForum	1.2.3	Fórum	Ferramenta de fórum desenvolvida em Java para discussões online
JHotDraw	7.0.6	Framework Gráfico	Framework Java para gráficos bidimensionais e construção de editores de desenho
DrawSWF	1.2.9	Aplicação Gráfica	Aplicação de desenho simples em Java para criação de arquivos animados SWF
ProGuard	4.4	Otimização	Sistema para otimizar, ofuscar e reduzir código Java através de análise de bytecode
HSQLDB	1.8.0.10	Banco de Dados	Sistema de banco de dados relacional implementado em Java
Xerces	2.9.1	Processamento XML	Analisador de XML de alto desempenho da família Apache Xerces
MegaMek	0.35.18	Jogo	Jogo de estratégia baseado no universo BattleTech
Art of Illusion	2.7.2	Modelagem 3D	Estúdio para modelagem e renderização 3D
JSML	1.1.1	Processamento	Sistema para descompilação de arquivos .class do Java
EMMA	2.0.5312	Análise Cobertura	Ferramenta de análise de cobertura de código Java usando instrumentação de bytecode

Fonte: Elaborado pelo autor (2025)

5.1.3 Processo de Refatoração

Todos os modelos avaliados, ChatGPT-3.5[4], DeepSeek-R1[5], Google Gemini 2.0 flash[6] e Qwen2.5-max, apresentaram desempenho satisfatório na compreensão de código Java, aplicação de princípios de qualidade de *software* e identificação de *code smells*, entretanto o Qwen2.5-max foi selecionado por disponibilizar acesso gratuito a funcionalidades avançadas, garantindo a replicabilidade do estudo e a viabilidade econômica da pesquisas.

O desenvolvimento do *prompt* estruturado fundamentou-se nas técnicas estabelecidas por White *et al.* (2023), integrando quatro componentes metodológicos essenciais para assegurar qualidade e consistência da refatoração: (i) definição explícita dos objetivos de refatoração, assegurando clareza quanto às metas de melhoria de qualidade; (ii) incorporação de diretrizes de estilo de codificação Java, garantindo aderência a convenções consolidadas da linguagem; (iii) instruções específicas para tratamento dos *code smells* considerados no estudo (*god class*,

[4] Disponível em:<https://chat.openai.com/> Acesso em: 7 abr. 2025
[5] Disponível em: <https://chat.deepseek.com/> Acesso em: 8 abr. 2025
[6] Disponível em: <https://gemini.google.com/> Acesso em: 9 abr. 2025

long method, *feature envy* e *data class*); e (iv) padronização do formato de saída, assegurando consistência na análise e reprodutibilidade dos resultados. No Código-fonte 1 é apresentado o conteúdo integral do *prompt*, seguindo as técnicas mencionadas acima.

Código-fonte 1 – Prompt usado para refatoração do códigos.

```
<prompt>
    <contexto>
        Você é um especialista em engenharia de software, com profundo conhecimento em Java, refatoração de código e detecção de code smells.
        Seu objetivo é analisar um trecho de código e identificar a presença de code smells específicos.
        Além disso, você deve sugerir uma refatoração detalhada, explicando como o código pode ser melhorado para seguir boas práticas de programação.
        <restricoes>
            Limite-se a analisar apenas os seguintes code smells: god class, long method, feature envy e data class.
        </restricoes>
    </contexto>
    <tarefa>
        Analise o seguinte código Java e determine se ele contém algum dos seguintes code smells:
        <lista>
            <item>
                God Class (Classe excessivamente grande e com muitas responsabilidades).
            </item>
            <item>
                Long Method (Métodos muito longos e difíceis de entender).
            </item>
            <item>
                Feature Envy (Um método que acessa mais dados de outra classe do que da sua própria).
            </item>
            <item>
```

```
23              Data Class (Uma classe que apenas armazena dados, sem
    comportamentos relevantes).
24          </item>
25      </lista>
26      Se houver um ou mais code smells, explique quais são, por que eles
    ocorrem e quais os impactos negativos no código.
27      Depois, proponha uma refatoração detalhada, mostrando a versão
    melhorada do código e justificando as mudanças feitas.
28      Sempre que for adicionado um arquivo com código java, faça a análise
    automaticamente.
29  </tarefa>
30  <formato>
31      Sua resposta deve seguir esta estrutura:
32      <lista>
33          <item>
34              1. Code Smells Detectados: Liste os problemas encontrados e
    explique cada um.
35          </item>
36          <item>
37              2. Justificativa: Por que esse code smell é um problema no
    código analisado?
38          </item>
39          <item>
40              3. Código Refatorado: Apresente uma versão melhorada do
    código.
41          </item>
42          <item>
43              4. Explicação da Refatoração: Explique cada modificação
    realizada e como ela melhora a qualidade do código.
44          </item>
45      </lista>
46  </formato>
47  <exemplo>
48      ...
```

```
    </exemplo>
</prompt>
```

Fonte: Elaborado pelo autor (2025)

A implementação desses componentes estruturais foi complementada por controles específicos voltados à preservação da integridade funcional e direcionamento preciso das intervenções. As instruções para manutenção da funcionalidade original buscou asseguraram que o processo de refatoração preservasse o comportamento dos métodos e classes, priorizando melhorias estruturais sem alteração semântica. O foco específico na eliminação dos *code smells* identificados direcionou as intervenções para os problemas estruturais detectados, garantindo que a refatoração abordasse sistematicamente as deficiências qualitativas presentes no código original.

O processo de refatoração seguiu protocolo sistematizado em seis etapas sequenciais para garantir consistência metodológica e controle de variáveis experimentais, sendo elas: (i) aplicação do *prompt* padronizado, elaborado com base nas diretrizes de White *et al.* (2023); (ii) inserção manual de cada classe no modelo Qwen2.5-max para obtenção das versões refatoradas; (iii) armazenamento organizado das 80 versões processadas, assegurando rastreabilidade entre originais e refatoradas; (iv) remoção sistemática de comentários em ambas as versões (original e refatorada) para garantir comparação cega livre de indicadores de origem; (v) seleção aleatória de cinco pares únicos de códigos (original e refatorado) para cada entrevistado para garantir diversidade na amostra e eliminar efeitos de familiarização entre diferentes sessões de entrevista; e (vi) randomização da posição dos códigos dentro de cada par, de modo que o original e o refatorado pudessem aparecer indistintamente como “Código 1” ou “Código 2” para eliminar viés de posição e assegurar que as preferências dos entrevistados refletissem qualidade percebida ao posicionamento visual na ferramenta.

5.1.4 Seleção dos Entrevistados e Coleta e Análise dos Dados

O estudo envolveu desenvolvedores profissionais selecionados por amostragem de conveniência, estratégia que possibilitou perfis heterogêneos em termos de experiência e atuação profissional, o que enriqueceu a análise qualitativa dos dados.

As entrevistas semi-estruturadas foram realizadas remotamente via Google Meet [7], formato que permitiu maior flexibilidade de participação e acesso a desenvolvedores geografica-

[7] Disponível em: <https://meet.google.com> Acesso em: 2 jun. 2025

mente distribuídos. O tempo médio de 45 minutos por sessão e a gravação integral das sessões utilizou o *software* OBS Studio[8], garantindo qualidade técnica adequada para posterior processamento dos dados. Todas as entrevistas foram transcritas através do programa Vibe[9], *software* de código aberto, gratuito, offline e usa LLM para maior fidelidade ao áudio. O protocolo seguiu diretrizes éticas rigorosas, incluindo consentimento informado e garantias de confidencialidade dos dados coletados.

As entrevistas seguiram um roteiro estruturado em quatro blocos principais, como pode ser visto no Apêndice D, esse roteiro inicia-se pela apresentação da pesquisa e caracterização do perfil profissional dos entrevistados durante cerca de dez minutos, estabelecendo contexto experiencial necessário para interpretação posterior das respostas. A fase central concentrou-se na comparação de códigos por aproximadamente 25 minutos, utilizando a ferramenta TwinCode, descrita no Capítulo 4, para apresentar os cinco pares de códigos em análise cega. O terceiro segmento explorou ferramentas e reflexões sobre qualidade levando entorno de dez minutos, contextualizando as avaliações individuais dentro de práticas profissionais mais amplas. A fase final, aproximadamente cinco minutos, revelou o uso de IA na refatoração e coletou ponderações sobre implicações futuras da tecnologia. A ferramenta TwinCode, teve papel essencial para apresentação comparativa dos códigos, oferecendo numeração sincronizada de linhas que facilitou referência precisa durante as discussões, realce de sintaxe para melhor legibilidade do código Java, ajuste dinâmico de fonte conforme necessidades visuais dos entrevistados, e interface responsiva acessível remotamente via Ngrok[10] para garantir funcionalidade adequada durante as sessões virtuais.

A análise dos dados seguiu a técnica de análise de conteúdo de Bardin (2016), aplicando especificamente a modalidade temática às transcrições das entrevistas realizadas. O processo analítico respeitou as três etapas fundamentais propostas pela autora: pré-análise para organização do material transcrito, exploração sistemática mediante codificação para identificar padrões recorrentes e critérios de avaliação, e tratamento interpretativo dos resultados obtidos. A categorização dos dados seguiu abordagem indutiva, resultando na emergência de cinco categorias principais que capturaram dimensões específicas da percepção dos desenvolvedores: (i) critérios de qualidade espontaneamente mencionados, (ii) preferências entre versões de código, (iii) justificativas para escolhas realizadas, (iv) percepções sobre inteligência artificial

[8] Disponível em: <https://obsproject.com> Acesso em: 5 jun. 2025
[9] Disponível em: <https://thewh1teagle.github.io/vibe/> Acesso em: 9 jul. 2025
[10] Disponível em: <https://ngrok.com/> Acesso em: 7 jul. 2025

pós-revelação e (v) expectativas futuras sobre integração de LLMs. Essas categorias encontram-se organizadas no Apêndice B, em forma de tabela que apresenta as unidades de registro, seus respectivos códigos e categorias, bem como exemplos literais de falas e a interpretação analítica associada.

5.2 Resultados

Esta seção organiza os resultados obtidos no estudo empírico em seis subseções. A Seção 5.2.1 apresenta a caracterização dos entrevistados, de modo a contextualizar o perfil da amostra e sua diversidade de experiências. A Seção 5.2.2 apresenta os critérios de qualidade de código identificados espontaneamente pelos desenvolvedores (QP1). A Seção 5.2.3 mostra as comparações entre versões originais e refatoradas, destacando preferências e justificativas (QP2). A Seção 5.2.4 apresenta o impacto da revelação sobre o uso de IA nas percepções dos entrevistados (QP3). A Seção 5.2.5 relata as expectativas e preocupações em relação ao uso de LLMs para melhoria da qualidade de código (QP4). Por fim, a Seção 5.2.6 apresenta uma análise integrativa dos resultados, sintetizando os achados quantitativos e qualitativos e discutindo suas implicações para a compreensão da percepção dos desenvolvedores sobre código refatorado por modelos de linguagem de grande porte LLMs.

5.2.1 Caracterização dos Entrevistados

A amostra é composta por sete entrevistados, contemplando diferentes níveis de senioridade: três autodeclarados juniores, dois plenos e dois seniores, distribuídos em distintas áreas de desenvolvimento de *software*. Como desenvolvimento *web*, móvel, sistemas embarcados e plataformas *Software as a Service* (SaaS). Essa diversidade permitiu identificar percepções distintas sobre qualidade de código, uma vez que cada nível de maturidade tende a valorizar aspectos específicos do processo de construção de *software*. Enquanto os mais experientes ressaltaram a importância da manutenção e da refatoração ao longo do ciclo de vida do código, os juniores enfatizaram a simplicidade e a legibilidade imediata como elementos centrais para definir a qualidade. Essa multiplicidade de cenários contribuiu para que fossem trazidas perspectivas complementares sobre a aplicabilidade dos critérios de qualidade. Os entrevistados que atuam como profissionais no mercado destacaram a relevância de arquiteturas escaláveis e do uso de padrões de projeto, enquanto aqueles com maior vínculo acadêmico priorizaram a clareza, a

organização estrutural e o valor pedagógico de determinadas soluções de código. Essa variação ampliou o escopo interpretativo da pesquisa.

A Tabela 13 detalha as características dos entrevistados, apresentando informações sobre experiência profissional, área de especialização e ferramentas de desenvolvimento habitualmente utilizadas. Como pode ser visto, há uma boa variabilidade de nível de senioridade, nesse quesito os entrevistado se classificaram por autodeclaração. O tempo de experiência mostra que os seniores tem 15 anos, os plenos mais de 4 anos e os juniores variando de 11 anos a um ano e meio. As área de atuação é bem diversifica, assim como as ferramentas. A seguir, são apresentados alguns trechos das entrevistas dos participantes referentes a seus perfis e às ferramentas utilizadas. E5, se autodeclarou como júnior "*eu considero minha experiência como júnior*", relatou ter cerca de "*seis anos*" de experiência em desenvolvimento *web*, destacando que atua com "*PHP, MySQL e alguma ferramenta para modelar banco de dados*", o que confirma sua prática em ferramentas variadas e foco em aplicações *web*. No nível pleno, E4 afirmou: "*Eu sou programador dart, pleno, basicamente*", acrescentando que possui "*quatro anos no mercado*" e que sua atuação se concentra "*principalmente em aplicativos, 90%*", utilizando como principais ferramentas o "*Android Studio com Xcode*". Já entre os sêniores, E7 declarou: "*Sênior [...] trabalho há 15 anos"*, ressaltando sua experiência no desenvolvimento de "*plataformas SaaS*" e seu uso cotidiano de ambientes diversos: "*Tenho utilizado mais o VS Code como IDE, mas já utilizei o IntelliJ também, Eclipse [...] utilizo Google Cloud, já utilizei AWS [...] também Docker e Kubernetes para orquestração*".

Tabela 13: Perfil dos entrevistados

ID	Nível	Experiência	Área Principal	Ferramentas
E1	Sênior	2 anos mercado, 15 total	*Web* (*back/front*)	VS Code, Cursor
E2	Júnior	11 anos	Embarcados, IA	Python, VS Code
E3	Pleno	5 anos	Educação, *Web*	Não especificado
E4	Pleno	4 anos mercado	*Mobile* (Flutter)	Android Studio
E5	Júnior	6 anos	*Web*	Ferramentas variadas
E6	Júnior	1,5 anos mercado	*Web* (Node.js/React)	VS Code
E7	Sênior	15 anos	SaaS, Plataformas	VS Code, IntelliJ

Fonte: Elaborado pelo autor (2025)

5.2.2 Critérios de Qualidade de Código (QP1)

De modo geral, todos os entrevistados demonstraram familiaridade com conceitos básicos de qualidade de código, ainda que nem sempre tenham utilizado terminologia técnica formal para descrevê-los. Termos como legibilidade, modularidade, padronização e manutenibilidade apareceram de forma recorrente nos discursos, mesmo quando expressos de maneira mais empírica e baseada em vivências pessoais. Essa característica reforça a relevância de considerar a experiência prática dos entrevistados na avaliação da qualidade de *software*, uma vez que o entendimento sobre o tema não se restringe a definições acadêmicas, mas é construído a partir da interação cotidiana com diferentes tipos de código e contextos de desenvolvimento.

A análise qualitativa das justificativas revelou que os desenvolvedores utilizam métodos predominantemente empíricos para avaliação, conforme indicado na Tabela 14. Os entrevistados (E1, E2, E3 e E7) mencionaram explicitamente a experiência como critério de avaliação da qualidade de código. Os demais entrevistados (E4, E5 e E6) não citaram diretamente a experiência pessoal como fundamento; entretanto, a análise semântica dos trechos a seguir, referenciando as escolhas pelo código refatorado, permite interpretar o uso da experiência pessoal na classificação da qualidade do código: E4 "*Claramente, a principal coisa que faz um código ser bom é ser legível e padronizado. O código 1 tem muita coisa fora do padrão.*"; E5 "*Não tem nem como comparar, o código 1 tá enxuto, perfeito, bem resumido. O código 2 tá enorme e pouco legível.*"; e E6 "*O código 2 está bem mais modular, responsabilidades divididas, mais legível.*".

Tabela 14: Experiência como métodos para classificação de qualidade de código

Entrevistado	Nível	Exemplo de fala
E1	Sênior	*Eu vejo muito esse código e reconheço como gambiarra porque era o que eu faria há 20 anos atrás.*
E2	Júnior	*Eu não tenho experiência de mercado tão grande... então me considero júnior. Minha avaliação é mais pela experiência acadêmica e pessoal.*
E3	Pleno	*A qualidade de um código se reflete quando, com os anos, ainda consigo dar manutenção naquele código.*
E7	Sênior	*Vejo métodos gigantes, ifs dentro de ifs... isso a experiência mostra que é difícil de manter.*

Fonte: Elaborado pelo autor (2025)

Após o processamento das entrevistas, foram identificados seis critérios de qualidade de código: legibilidade, manutenibilidade, modularidade, padronização de nomenclatura, funcionalidade e simplicidade. A Figura 10 apresenta a distribuição de ocorrências desses critérios entre

os entrevistados, resultante de análise semântica das falas, detalhada no Apêndice A. Observa-se a predominância de legibilidade, manutenibilidade e modularidade, evidenciando a ênfase dos entrevistados em atributos que favorecem a compreensão, a evolução e a organização estrutural do código. A seguir, é descrita a frequência com que cada critério de qualidade foi mencionado pelos entrevistados, permitindo identificar os mais relevantes na avaliação de código.

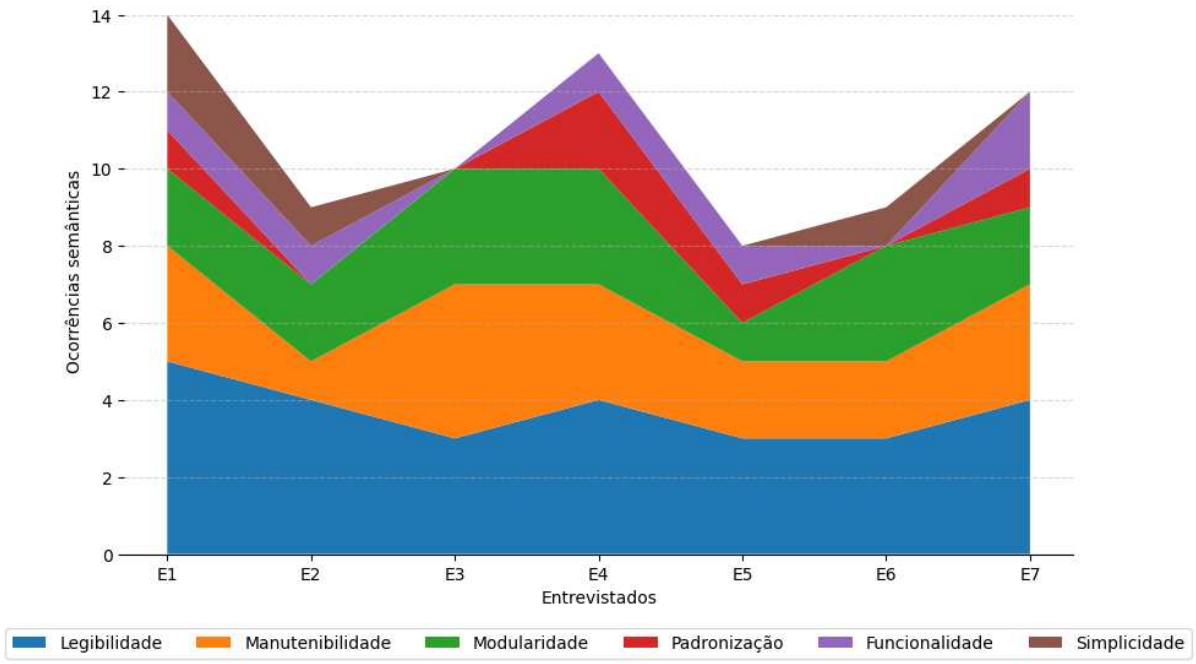


Figura 10: Categorias por entrevistado
Fonte: Elaborado pelo autor (2025)

Legibilidade (7/7 entrevistados): Houve unanimidade na percepção de que o código deve ser fácil de ler e compreender. Os entrevistados destacaram que a clareza visual e a escolha adequada de nomes para variáveis e funções são determinantes para a compreensão do código. Como exemplificado por E4: *"A principal coisa que faz um código ser bom para manter é um código que duas pessoas vão bater o olho e vão compreender a lógica ali."*

Manutenibilidade (6/7 entrevistados): Associada à capacidade de realizar alterações e extensões no código sem comprometer sua integridade. Foi citada como um dos critérios essenciais para garantir a longevidade do *software*, aparecendo entre os "*três principais critérios*" de E7, juntamente com legibilidade e funcionalidade.

Modularidade (6/7 entrevistados): Relacionada à separação clara de responsabilidades e à aplicação de princípios como responsabilidade única. Essa prática foi reconhecida como essencial para manter a organização e a escalabilidade do sistema. E3 exemplificou: *"Separação de responsabilidades"*.

Padronização de Nomenclatura (3/7 entrevistados): Considerada parte integrante da legibilidade, refere-se ao uso consistente de convenções de nomes para variáveis, funções e métodos, facilitando a compreensão por diferentes desenvolvedores. E4 destacou: *"Nomear bem variáveis e funções"*.

Funcionalidade (3/7 entrevistados): Refere-se à garantia de que o código atenda plenamente aos requisitos previstos e execute as tarefas de forma correta. Foi citada por E7 como um dos pilares da qualidade de código.

Simplicidade (3/7 entrevistados): Preferência por soluções enxutas e objetivas, evitando complexidade desnecessária. E1 comentou: *"Eu gosto de códigos pequenos, mas que sejam legíveis"*.

Essa distribuição evidencia não apenas o consenso em torno de determinados aspectos, como a legibilidade, mas também a diversidade de perspectivas sobre o que caracteriza um código de qualidade. Diferentemente da Figura 10, que sintetiza ocorrências a partir de análise semântica das falas, as citações diretas dos entrevistados apresentadas acima consideram apenas menções textuais explícitas aos termos, podendo, portanto, divergir ligeiramente dos totais previamente reportados. As citações foram extraídas e classificadas pelos seis critérios de qualidade reportados, preservando a literalidade e o contexto imediato de enunciação. Essa distinção metodológica reforça a rastreabilidade entre os resultados e o *corpus* original e está detalhada no Apêndice A.

A Figura 11 apresenta a nuvem de palavras construída a partir das sete transcrições. O procedimento de construção da nuvem de palavras consiste em: (i) unificar as transcrições em um único corpus; (ii) converter letras maiúsculas para minúsculas e *tokenizadas* por *regex* com preservação de acentuação; e (iii) promover limpeza linguística, ou seja, remover palavras frequentes, mas sem conteúdo semântico do português e de marcadores conversacionais (p.ex., “né”, “tá”, “uhum”), à exclusão de artigos e pronomes e à filtragem de verbos por abordagem híbrida (lista manual de formas frequentes e heurística morfológica por terminações), incluindo a retirada de itens de baixo conteúdo informativo (“é”, “e”). Para evitar dispersão entre flexões, aplicou-se normalização de número sem afetar os acentos (regras: “ões/ães $\rightarrow$ ão”, “eis $\rightarrow$ el”, “is/es/s $\rightarrow$ singular”), de modo que pares como “variável/variáveis” convergissem para “variável”. A nuvem foi gerada a partir das frequências consolidadas e diagramada no Infogram[11], ao passo que o processamento foi implementado em Python (*pandas*, *regex*), assegurando rastreabilidade

[11] Disponível em: <https://infogram.com/> Acesso em: 11 ago. 2025

e reprodutibilidade das etapas e parâmetros.

Figura 11: Nuvem de palavras
Fonte: Elaborado pelo autor (2025)

Como resultados, observa-se a forte centralidade de "*código*", termo mais proeminente, seguido de "*qualidade*"; em torno desses núcleos gravitam referências práticas como "*sistema*", "*função*", "*ferramenta*", "*exemplo*", "*software*" e "*variável*", bem como atributos internos associados à avaliação técnica: "*legível/legibilidade*", "*manutenção*", "*teste*", "*complexidade*" e "*ciclomática*". O tamanho relativo das palavras indica maior frequência de menção e sugere que os entrevistados concentram o discurso na avaliação da qualidade do código, articulando critérios de legibilidade, testabilidade e controle de complexidade, além de situarem tais julgamentos no contexto de uso (ferramentas, sistemas e exemplos). Em termos quantitativos, "*código*" responde por 10,7% das palavras filtradas (893 de 8.342), ao passo que "*qualidade*" perfaz 1,95%; aparecem ainda "*sistema*" (1,29%), "*questão*" (1,08%), "*função*" (1,07%), "*melhor*" (1,03%), "*ferramenta*" (0,92%), "*exemplo*" (0,91%), "*software*" (0,80%) e "*variável*" (0,62%), corroborando a predominância de vocabulário relacionado à análise de trechos de código.

Resumo da resposta da QP1. Em síntese, desenvolvedores avaliam a qualidade de código por critérios majoritariamente empíricos, ancorados na prática profissional, com consenso forte em torno da legibilidade, manutenibilidade e modularidade. Nomenclatura, funcionalidade e simplicidade surgem como fatores complementares que reforça a clareza, a evolução segura e a organização estrutural do *software*. Esse panorama delimita, portanto, os atributos que efetivamente orientam o julgamento humano e fornecem a base para interpretar as escolha comparativas discutidas nas análises subsequentes.

5.2.3 Comparações de Código (QP2)

A questão de pesquisa (QP2) buscou investigar se desenvolvedores conseguem identificar diferenças qualitativas entre códigos originais e versões refatoradas por LLMs quando submetidos a comparações cegas. Para isso, foram apresentados cinco pares de códigos aos entrevistados, sem qualquer indicação de origem, e cada entrevistado justificou suas escolhas com base em critérios de qualidade de *software*.

Os resultados mostram que os desenvolvedores, em sua maioria, conseguiram perceber diferenças consistentes entre as versões apresentadas. As justificativas apontam que a legibilidade foi o critério mais recorrente: códigos mais concisos, claros e bem indentados foram associados a maior qualidade, como evidenciado nas falas de E1, E2 e E3, que destacaram preferir códigos "*enxutos*" e "*mais fáceis de ler e manter*".

Outro critério amplamente citado foi a modularidade, ressaltada por E3, E6 e E7, que valorizaram versões em que havia separação de responsabilidades e menor acoplamento. Para esses entrevistados, a divisão adequada em classes e funções foi entendida como sinal de maior manutenibilidade e testabilidade. Ainda assim, alguns entrevistados, como E4 e E6, advertiram que modularizações mal planejadas podem aumentar a complexidade, indicando que nem sempre a refatoração é sinônimo de qualidade superior.

A complexidade ciclomática apareceu como elemento de crítica em falas de E6 e E7, que rejeitaram códigos com condicionais aninhadas ou *loops* redundantes, preferindo soluções mais lineares e diretas. Já o E5 trouxe uma visão diferenciada ao considerar que, em determinados contextos de ensino, o código original poderia ser mais didático, por tornar a sequência lógica mais evidente a iniciantes, mesmo que fosse mais extenso.

A Figura 12 apresenta a distribuição das escolhas dos entrevistados entre códigos

originais e refatorados. Observa-se uma tendência praticamente unânime em favor das versões refatoradas: dos 35 julgamentos individuais (sete entrevistados em cinco comparações cada), 34 escolhas (97,1%) recaíram sobre o código refatorado e apenas uma escolha (2,9%) favoreceu o código original. Esse único caso correspondeu ao entrevistado E5, que destacou o potencial didático do código original por tornar a lógica mais explícita a iniciantes. Em outras palavras, para 7 dos 7 desenvolvedores, a maioria dos 5 pares foi avaliada como de maior qualidade na versão refatorada (teste binomial unilateral vs $0,5$: $p = 0,0078$). Em nível de par, 5 dos 5 pares tiveram maioria pró-refatorado ($p = 0,031$). Em nível de voto, 34 dos 35 votos favoreceram o refatorado ($p \approx 1,0 \times 10^{-9}$). Esses resultados convergem para a conclusão de que as versões refatoradas pelas LLMs apresentam maior qualidade do que as originais.

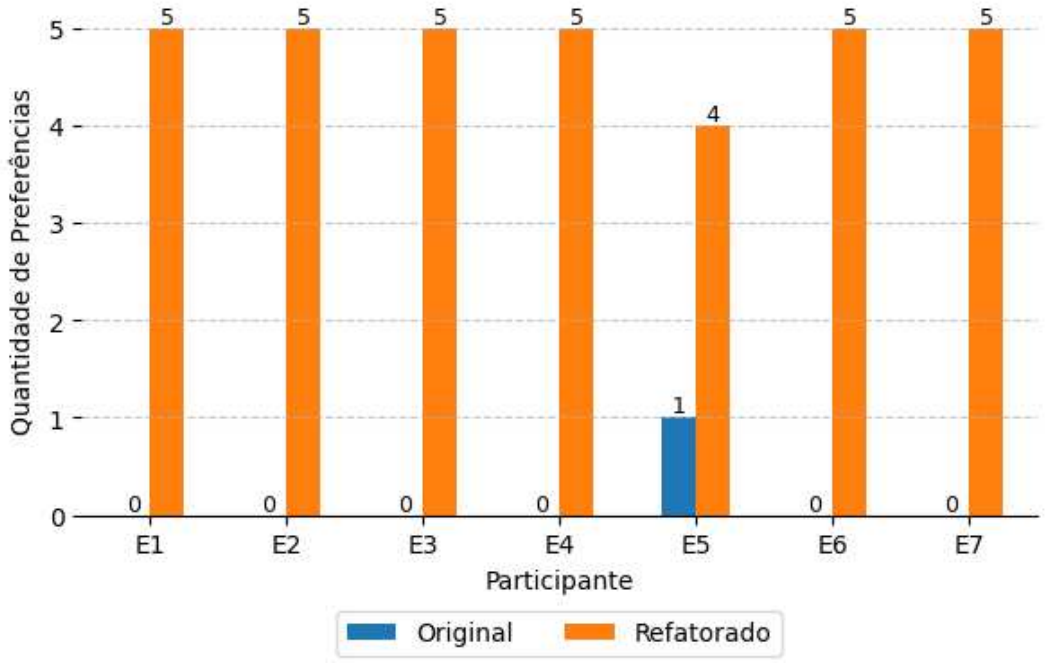


Figura 12: Distribuição geral de preferências
Fonte: Elaborado pelo autor (2025)

Os dados reforçam que os desenvolvedores, quando submetidos a comparações cegas, identificaram diferenças qualitativas consistentes entre as versões, atribuindo valor às refatorações geradas pela LLM. As justificativas recorrentes alinham-se a dimensões de qualidade de *software* amplamente reconhecidas, como legibilidade, modularidade, simplicidade e manutenibilidade. Assim, a análise quantitativa corrobora a interpretação qualitativa: embora nem todas as preferências tenham sido absolutas em termos de contexto, a percepção coletiva aponta que as refatorações automatizadas foram consideradas superiores em quase todos os cenários.

Resumo da resposta da QP2. Os desenvolvedores foram capazes de identificar diferenças qualitativas entre o código original e o refatorado por LLMs em um cenário de comparações cegas. A predominância das escolhas pelas versões refatoradas (34 de 35 avaliações) demonstra que atributos como legibilidade, modularidade e simplicidade foram amplamente reconhecidos como superiores. Ainda que tenham surgido ressalvas pontuais, como a percepção de que o código original poderia ser mais didático para iniciantes, o conjunto das justificativas evidencia que os entrevistados não apenas distinguiram as versões, mas atribuíram valor às melhorias introduzidas pela refatoração automática. Esses achados reforçam a ideia de que LLMs podem produzir efeitos perceptíveis e consistentes na qualidade do código, mesmo quando a autoria é ocultada.

5.2.4 Impacto da Revelação sobre IA (QP3)

A terceira questão de pesquisa (QP3) investigou como a revelação de que os códigos avaliados haviam sido refatorados por modelos de linguagem de grande porte (LLMs) influenciou a percepção de qualidade dos entrevistados. De forma geral, a revelação de que uma das versões havia sido gerada por LLM não afetou negativamente as escolhas dos entrevistados. Ao contrário, a maioria dos entrevistados demonstrou surpresa positiva, reconhecendo que os códigos gerados pela LLM correspondiam a critérios técnicos já utilizados em suas práticas profissionais. Entrevistados como E1, E2 e E3 afirmaram que suas escolhas não se alterariam após saber da intervenção da IA, pois os critérios de escolha haviam se baseado em atributos de qualidade de código como legibilidade, modularidade e simplicidade. Conforme indicado na fala de E1: "*Mesmo sabendo que foi uma IA que gerou, eu ainda escolheria o mesmo código, porque ele está mais claro e fácil de manter.*" Essa postura sugere que a qualidade percebida independe da autoria, desde que o código atenda a padrões técnicos consistentes. O E2 reforçou essa perspectiva ao afirmar: "*O que me fez escolher não foi quem escreveu, mas a forma como o código ficou mais simples e legível.*" Já o E3 destacou: "*Pra mim não muda nada, porque a análise que fiz foi olhando a clareza e a organização. Se foi humano ou IA, o resultado continua válido.*"

Por outro lado, emergiram posicionamentos críticos e cautelosos. Entrevistados como E4 e E5 ponderaram que, embora a refatoração automática apresente resultados satisfatórios, a confiabilidade plena da solução exige validação humana. O entrevistado E4 ressaltou: "*Funciona*

bem, mas eu não colocaria em produção sem revisar, porque a IA não sabe das regras de negócio." Da mesma forma, E5 alertou: "*É impressionante ver que foi feito por IA, mas precisa ter alguém conferindo, porque pode faltar contexto do sistema real.*" Além disso, foi observada uma dimensão de adequação pedagógica. E5 destacou que, em alguns casos, códigos mais extensos e detalhados (mesmo quando considerados menos elegantes tecnicamente) podem ser mais apropriados para fins de ensino: "*Para quem está começando, às vezes é melhor ver o passo a passo no código original, porque ajuda a entender a lógica.*" Esse aspecto evidencia que a percepção de qualidade pode variar de acordo com o contexto de aplicação — técnico ou educacional.

A Tabela 15 reúne as percepções dos entrevistados sobre o papel dos LLMs na qualidade de código, destacando tanto os benefícios identificados quanto as limitações. E1, por exemplo, afirmou que a IA "*ajuda muito, especialmente quando estou cansado*", mas ressaltou que sua função é apenas "*facilitar, não substituir*". Em linha semelhante, E7 reconheceu que a tecnologia "*acelera o desenvolvimento, mas exige validação*", entendendo que ela não elimina a necessidade do desenvolvedor, mas adapta suas funções. Já E2 apontou que a IA "*contribui para atributos de qualidade, mas não para aprendizado*", reforçando a visão de que seu papel é de ferramenta de apoio, não de ensino. Por outro lado, E4 é cauteloso, ao afirmar que a IA pode ser "*boa para padronização, mas pode aumentar a complexidade*", destacando que, embora útil, seu uso requer supervisão crítica. Esses exemplos revelam uma visão pragmática: os entrevistados reconhecem os ganhos da tecnologia como aliada no desenvolvimento, mas reafirmam que ela não substitui o trabalho humano.

Tabela 15: Percepções sobre IA na qualidade de código

ID	Opinião sobre Qualidade	Visão sobre Papel da IA
E1	"*IA ajuda muito, especialmente quando estou cansado*"	"*Facilita, mas não substitui*"
E2	"*Contribui para atributos de qualidade, mas não para aprendizado*"	"*Ferramenta de apoio*"
E3	"*Ajuda até certo ponto, depois pode atrapalhar*"	"*Complementa, mas não substitui*"
E4	''*Boa para padronização, mas pode aumentar complexidade*"	"*Ferramenta de apoio*"
E5	"*Auxilia positivamente no desenvolvimento*"	"*Necessita supervisão humana*"
E6	"*Útil para sugestões e diferentes perspectivas*"	"*Ferramenta para produtividade*"
E7	"*Acelera o desenvolvimento, mas exige validação*"	"*Não substitui, adapta funções*"

Fonte: Elaborado pelo autor (2025)

Ao relacionar as percepções sobre a IA (Tabela 15) com o perfil dos entrevistados

(Tabela 13), observa-se que o nível de senioridade influencia diretamente a forma como os desenvolvedores avaliam o impacto da tecnologia na qualidade de código. Os júniores (E2, E5 e E6) tendem a valorizar a IA como recurso de apoio prático e ganho imediato de produtividade, mas também revelam preocupações com o aprendizado e a necessidade de supervisão. O entrevistado E2, por exemplo, ressaltou que a ferramenta "*não contribui para aprendizado*", enquanto o entrevistado E5 a considera uma "*espécie de refino*" que exige validação humana. Já o E6, com pouca experiência de mercado, destacou o uso de IA como "*uma opinião a mais*" durante o processo de desenvolvimento. Entre os plenos (E3 e E4), o discurso é mais ambivalente: ambos reconhecem que a IA pode trazer melhorias em legibilidade e padronização, mas alertam para o risco de aumento desnecessário da complexidade e para a necessidade de cautela no uso em produção. O E3 sintetizou essa visão ao afirmar que a ferramenta "*ajuda até certo ponto, depois pode atrapalhar*", enquanto o E4 enfatizou que a IA "*pode mexer além do que era preciso*". Nesse nível, há uma percepção clara de que a IA deve ser usada com moderação, especialmente em tarefas críticas de refatoração. Por fim, os sêniores (E1 e E7) apresentam uma perspectiva mais estratégica e madura. Ambos reconhecem limitações práticas, como erros frequentes ou necessidade de revisão, mas veem a IA como um catalisador de mudanças no papel do desenvolvedor. O E1 declarou que utiliza a ferramenta para acelerar tarefas repetitivas, embora precise ajustar saídas incorretas, enquanto o E7 destacou que "q*uem não usa LLM está ficando para trás*", indicando que a adoção da IA já é vista como diferencial competitivo. Nesse grupo, a IA não é apenas suporte técnico, mas também um elemento transformador do futuro da profissão.

Resumo da resposta da QP3. A revelação sobre o uso de IA influenciou a percepção dos desenvolvedores, mas não de forma a reduzir a avaliação positiva da qualidade. A surpresa inicial foi acompanhada pelo reconhecimento de que as refatorações geradas por LLMs incorporaram critérios consistentes de legibilidade, modularidade e simplicidade. Ainda assim, surgiram ressalvas importantes: juniores destacaram o caráter de apoio da tecnologia, plenos apontaram riscos de complexidade desnecessária e perda de contexto, enquanto sêniores enfatizaram a necessidade de revisão e supervisão crítica antes de adoção em produção. Assim, conclui-se que LLMs podem contribuir de forma robusta para a qualidade de código, desde que utilizadas de forma responsável e em complementaridade à expertise humana.

5.2.5 Expectativas sobre LLMs (QP4)

A análise das entrevistas evidenciou forte convergência em quatro aspectos centrais sobre o papel dos LLMs no desenvolvimento. Todos os entrevistados conceituaram a IA como ferramenta de apoio, e não como substituta do desenvolvedor, refletindo uma visão pragmática e compatível com as capacidades tecnológicas atuais. Essa perspectiva conecta-se à percepção recorrente da necessidade de supervisão humana, destacada pela maioria, indicando que a automação é vista como meio para potencializar competências, não para eliminá-las. Também houve concordância quanto ao potencial de ganhos de produtividade, com reconhecimento de que a tecnologia pode acelerar o desenvolvimento e aumentar a eficiência em tarefas específicas. Por fim, a aplicação da IA foi considerada especialmente útil em contextos de pressão temporal, fadiga ou execução de tarefas repetitivas, reforçando sua natureza de suporte às atividades do programador.

Embora exista esse consenso geral, a análise por nível de senioridade revelou diferenças significativas nas expectativas em relação ao futuro uso de LLMs. Os sêniores (E1 e E7) demonstraram uma visão mais estratégica, considerando a tecnologia um diferencial competitivo inevitável, ainda que dependente de revisão crítica. Já júniores e plenos reconheceram ganhos imediatos de produtividade, legibilidade e padronização, mas destacaram preocupações quanto à complexidade desnecessária, perda de contexto em códigos mais extensos e risco de dependência, especialmente no aprendizado de boas práticas. Essa distinção evidencia que, enquanto os mais experientes projetam a integração de LLMs como inevitável e transformadora do mercado de desenvolvimento, os menos experientes tendem a enfatizar sua utilidade prática no presente, condicionada ao uso moderado e supervisionado.

Em relação às limitações, emergiram três categorias principais. A primeira refere-se à tendência ao excesso de complexidade, com observações pelo entrevistado E4: "*a maioria das IAs vai tentar ajeitar uma coisa, mas vai mexer além do que era pra ela ter mexido*", destacando que em processos de refatoração a ferramenta pode inserir condicionais, abstrações ou modificações que não eram requeridas, o que dificulta a manutenção. A segunda limitação aponta para a dificuldade em manter em manter o contexto em códigos mais extensos ou arquiteturas mais complexas. O E3 relatou que a ferramenta "*ajuda até certo ponto*", mas depois "*se perde na implementação*", exigindo constantes pedidos de correção do usuário; para ele, a IA consegue gerar trechos úteis, mas não sustenta consistência quando precisa lidar com dependências múltiplas ou fluxos de negócio completos. Por fim, a terceira limitação envolve a

necessidade de validação humana, aspecto amplamente enfatizado por entrevistados experientes como o E7, que afirmou: "*Claro que eu evitaria de usar LLM pra código direto em produção. Então, acho que uma revisão de códigos, ela é necessária.*" Essa posição ecoa em outros entrevistados, como o E5, que definiu a IA como "*uma espécie de refino, mas que ainda vai precisar do desenvolvedor pra avaliar*". Dessa forma, ainda que reconheçam os benefícios da tecnologia, os desenvolvedores apontam que seu uso seguro e eficaz depende de moderação, supervisão crítica e adaptação ao contexto de cada projeto.

Resumo da resposta da QP4. Os entrevistados projetam expectativas majoritariamente positivas, ainda que acompanhadas de cautela. Houve consenso em considerar a IA uma ferramenta de apoio, útil para potencializar competências, mas não como substituta do desenvolvedor, sendo valorizada sobretudo em contextos de pressão temporal, fadiga ou execução de tarefas repetitivas. Em geral, os entrevistados reconheceram o potencial da tecnologia para aumentar a produtividade, acelerar tarefas rotineiras e apoiar processos como refatoração e revisão de código. Entre os menos experientes (juniores e plenos), destacou-se a percepção de ganhos imediatos em aprendizagem, legibilidade e padronização, acompanhados, contudo, de preocupações quanto à complexidade desnecessária, perda de contexto em códigos extensos e risco de dependência no aprendizado de boas práticas. Já os sêniores (E1 e E7) apresentaram uma visão mais estratégica, considerando a adoção de LLMs um diferencial competitivo inevitável, sintetizada na fala de E7 ao afirmar que "*se você não usa LLM, você tá ficando pra trás*". Apesar das diferenças de ênfase, todos os níveis de senioridade convergiram na ideia de que a adoção futura deve ocorrer com supervisão crítica, integração gradual e adaptação de práticas de engenharia, reforçando a complementaridade entre automação e expertise profissional.

5.2.6 Análise dos Resultados

A convergência em torno de legibilidade, manutenibilidade e modularidade corrobora princípios amplamente estabelecidos na literatura sobre qualidade de código (MARTIN, 2009; FOWLER, 2018). Apesar de variações nas ênfases individuais, os entrevistados demonstraram compartilhar valores fundamentais sobre o que caracteriza um código de qualidade. A predominância de preferência pelos códigos refatorados sugere que as LLMs são capazes de produzir código de qualidade comparável e, em vários casos, percebida como superior, ao desenvolvido

originalmente. Essa percepção positiva esteve associada a critérios como legibilidade, modularidade e simplicidade. Entretanto, também foram registradas ressalvas quanto a possíveis excessos de modificação, aumento desnecessário de complexidade e perda de contexto em trechos mais elaborados, reforçando que a qualidade percebida não é uniforme em todos os casos.

A ausência de mudanças significativas na avaliação após a revelação de que uma das versões havia sido gerada por IA indica que as preferências permaneceram fundamentadas nos mesmos critérios técnicos aplicados inicialmente. As decisões dos entrevistados não foram guiadas pela origem do código, mas por características associadas à qualidade conforme seus próprios referenciais. A unanimidade quanto à IA como ferramenta de apoio, não como substituta do desenvolvedor, aliada à ênfase na necessidade de supervisão humana, reflete uma visão pragmática sobre seu uso no desenvolvimento de *software*. Essa perspectiva reconhece seu potencial para ampliar capacidades e produtividade, ao mesmo tempo em que preserva o papel central do julgamento humano na garantia da qualidade.

5.3 Trabalhos Relacionados

Chen (2024) conduziu estudo de caso no estúdio TiMi investigando o impacto do *Artificial Intelligence* (AI)-*pair programming* na qualidade de código e satisfação dos desenvolvedores através de análise de 10 projetos com IA e 10 sem IA. O estudo demonstrou que projetos com AI-*pair programming* apresentaram melhor qualidade de código (menor complexidade ciclomática: 12,3 vs 15,7; maior cobertura: 82,6% vs 76,4%) e maior satisfação dos desenvolvedores (4,3 vs 3,6). Utilizando ferramentas como GitHub Copilot e ChatGPT, os resultados revelaram benefícios como economia de tempo e melhoria de qualidade, mas também desafios relacionados à confiabilidade e autonomia. Embora focado em AI-*pair programming* colaborativo, seus achados sobre percepção positiva de desenvolvedores e melhoria objetiva na qualidade fornecem base empírica complementar ao presente estudo sobre refatoração automatizada, demonstrando que diferentes aplicações de IA no desenvolvimento podem produzir benefícios consistentes.

Ribeiro *et al.* (2024) investigaram a percepção de desenvolvedores sobre *code smells* e o uso de ferramentas automatizadas para sua detecção. Por meio de entrevistas e síntese temática, os autores identificaram diferentes compreensões sobre o conceito de *code smell*, bem como fatores que influenciam sua introdução e a adoção de ferramentas de suporte, como custos de configuração, cultura organizacional e priorização de entregas. A pesquisa reforça a importância de compreender não apenas os aspectos técnicos da qualidade de código, mas

também as barreiras práticas e contextuais que afetam a incorporação de mecanismos de melhoria na rotina de desenvolvimento, aspecto que dialoga diretamente com os objetivos do presente trabalho.

Santos e Gerosa (2018) conduziram estudo empírico sobre percepção de boas práticas de codificação, avaliando o impacto de 11 práticas de codificação Java na legibilidade percebida por desenvolvedores através de comparações cegas entre pares de *snippets* de código. Com 62 participantes (55 estudantes universitários e 7 programadores profissionais), os autores utilizaram aplicação web customizada para apresentar aleatoriamente 10 pares de código por participante, derivando as práticas avaliadas dos modelos de legibilidade de Buse e Weimer (2009) e Scalabrino *et al.* (2016). Os resultados demonstraram que 8 das 11 práticas apresentaram efeitos estatisticamente significativos: 7 melhoraram a legibilidade (incluindo uso de linhas em branco após chaves, limitação de 80 caracteres por linha, e evitar múltiplas declarações por linha) e 1 a piorou (abertura de chaves na mesma linha da declaração). Crucialmente, seus achados sobre a importância da legibilidade como critério universal de qualidade corroboram os resultados deste trabalho, embora o foco tenha sido em práticas convencionais de codificação em vez de código gerado por IA, complementando esta investigação ao estabelecer *baseline* sobre percepção de qualidade em contextos tradicionais de desenvolvimento.

Kudriavtseva *et al.* (2025) conduziram *survey* com 105 desenvolvedores sobre percepções de segurança de código gerado por IA, identificando que desenvolvedores experientes são mais céticos sobre segurança enquanto desenvolvedores juniores tendem a superestimar suas capacidades - padrão potencialmente explicado pelo efeito Dunning-Kruger que oferece contexto para interpretar variações por experiência observadas em a pesquisa aqui conduzida. Os autores reportaram que 61% dos desenvolvedores gastam mais tempo em revisões de segurança de código IA, convergindo com os achados deste trabalho sobre a necessidade de supervisão humana, embora a abordagem quantitativa sobre percepções de segurança se diferencie metodologicamente da investigação aqui realizada, baseada em comparações cegas sobre qualidade geral. Dessa forma, apresenta-se uma perspectiva complementar sobre a aceitação de código gerado por inteligência artificial.

Sheard *et al.* (2024) abordaram a perspectiva educacional sobre as ferramentas de IA. Foram entrevistados 12 educadores de programação de três países sobre ferramentas de geração de código, revelando preocupações quanto ao excesso de confiança estudantil e à perda de aprendizado de fundamentos, convergindo com os achados deste trabalho sobre a necessidade de

supervisão humana. Particularmente relevante é o consenso educacional de que a introdução de IA deve ocorrer apenas após o domínio dos fundamentos de programação, perspectiva que ressoa com a observação de que desenvolvedores percebem a inteligência artificial como ferramenta de apoio, e não como substituta de competências fundamentais, demonstrando que diferentes *stakeholders* do ecossistema de desenvolvimento compartilham preocupações similares sobre a integração responsável da automação de código.

Börstler *et al.* (2023) conduziram estudo qualitativo com 34 entrevistas semi-estruturadas envolvendo desenvolvedores profissionais, professores e estudantes da Europa e Estados Unidos, analisando 130 exemplos de código em 14 linguagens diferentes para investigar percepções sobre qualidade de código. Utilizando codificação temática, os autores identificaram que legibilidade e estrutura foram as propriedades mais comumente mencionadas para definir qualidade (82% e 65% dos participantes, respectivamente), seguidas por documentação, compreensibilidade e manutenibilidade. O estudo revelou que ao analisar exemplos concretos, desenvolvedores focam em estrutura como propriedade fonte levando à compreensibilidade e manutenibilidade, e que desenvolvedores experientes possuem visão mais ampla de qualidade considerando múltiplas categorias simultaneamente. Seus achados sobre critérios valorizados (legibilidade, estrutura e compreensibilidade) fornecem base empírica que corrobora a investigação aqui conduzida, estabelecendo padrões sobre percepção de qualidade em contextos tradicionais de desenvolvimento.

Em resumo, os trabalhos relacionados reforçam a relevância da investigação sobre como desenvolvedores percebem a qualidade de código em diferentes contextos, sejam eles marcados por práticas tradicionais (SANTOS; GEROSA, 2018; BÖRSTLER *et al.*, 2023), pelo uso de ferramentas automatizadas de apoio (RIBEIRO *et al.*, 2024), pelo *AI-pair programming* (CHEN, 2024) ou pelas preocupações de segurança e aprendizagem associadas ao código gerado por IA (SHEARD *et al.*, 2024). O presente estudo, entretanto, distingue-se por adotar um delineamento experimental baseado em comparações cegas entre código original e código refatorado por LLMs, aliado a entrevistas semiestruturadas para análise qualitativa. Essa combinação metodológica permitiu capturar tanto a preferência objetiva dos desenvolvedores quanto suas justificativas subjetivas, revelando critérios como legibilidade, modularidade e simplicidade como centrais na avaliação da qualidade. Como resultado inédito, a investigação aqui conduzida demonstra que, mesmo sem conhecimento da autoria, os participantes atribuíram sistematicamente maior qualidade às versões refatoradas por IA, mas ressaltaram a necessidade de supervisão crítica e adequação ao contexto. Dessa forma, este estudo contribui ao campo ao oferecer evidências

empíricas sobre a percepção de qualidade de código gerado por LLMs, estabelecendo um contraponto às pesquisas focadas em segurança, ensino ou práticas convencionais e ampliando a compreensão sobre a integração responsável da automação no desenvolvimento de *software*.

5.4 Ameaças à Validade

Durante a condução desta investigação empírica, buscaram-se procedimentos de controle que contribuíssem para a redução de vieses e para a consistência dos resultados obtidos. Ainda assim, algumas ameaças à validade precisam ser reconhecidas, tanto para promover transparência científica quanto para orientar melhorias em estudos futuros. Nesta seção, são discutidas as ameaças à validade dos resultados obtidos, organizadas conforme as quatro categorias propostas por Wohlin *et al.* (2012): validade interna, externa, de construção e de conclusão.

Validade Interna. A validade interna refere-se à possibilidade de vieses que comprometam a interpretação causal dos resultados. Três ameaças à validade interna do estudo foram identificadas. A primeira relaciona-se ao fato de que os *entrevistados já possuíam familiaridade prévia com práticas de refatoração*, o que pode ter influenciado suas escolhas em favor do código refatorado. Além disso, não foi *controlado o efeito da ordem de apresentação das versões*, o que poderia gerar viés de preferência; para minimizar essa ameaça, os códigos originais e refatorados foram apresentados de forma aleatória. Outra limitação consiste no fato de que *o estudo não isolou variáveis externas*, como estilo de programação individual ou experiência prévia, que podem ter impactado as percepções dos entrevistados. Variáveis como estilo de programação estão fora de controle, enquanto, em relação à experiência prévia, foram apresentadas análises separando os participantes segundo esse critério.

Validade Externa. A validade externa refere-se à possibilidade de generalização dos resultados para outros contextos. Duas ameaças à validade externa do estudo foram identificadas. O estudo contou apenas com um *grupo reduzido de desenvolvedores*, o que limita a extrapolação dos resultados. Apesar do grupo ser pequeno, os entrevistados possuem características de formação acadêmica e experiência distintas. Da mesma forma, *foram avaliados apenas cinco pares de classes específicas*, não sendo possível assegurar que os resultados se mantenham em diferentes domínios de *software*, linguagens de programação ou níveis de complexidade de sistemas. A limitação do número de artefatos é inerente à profundidade da análise. O aumento dessa quantidade provavelmente resultaria em respostas mais superficiais. Por esse motivo, o número de artefatos foi limitado a cinco.

Validade de Construção. A validade de construção está relacionada à adequação das medidas utilizadas para capturar o construto de "qualidade de código". Neste trabalho, a qualidade foi inferida a partir da percepção subjetiva dos desenvolvedores, sem o apoio de métricas objetivas complementares Tal decisão pode limitar a precisão do conceito medido, uma vez que diferentes entrevistados podem ter interpretado "qualidade" de formas distintas. No entanto, como as entrevistas foram gravadas e houve diálogo, os participantes foram instigados a definir de forma clara o que consideravam como qualidade de código.

Validade de Conclusão. A validade de conclusão envolve a robustez estatística e a força das inferências obtidas. Duas ameaças à validade da conclusão do estudo são discutidas. O *número reduzido de entrevistados e de pares avaliados pode levar a baixa potência estatística*. Embora os resultados indiquem forte preferência pelas versões refatoradas, a amostra pequena aumenta a chance de erro tipo I (superestimar evidências) ou tipo II (não detectar efeitos em casos específicos). Além disso, a *ausência de análise quantitativa mais robusta*, como testes estatísticos formais para avaliar significância e intervalos de confiança, limita a solidez das conclusões para algumas questões de pesquisa.

5.5 Conclusão

Este capítulo investigou a percepção de desenvolvedores sobre a qualidade de código refatorado por LLMs por meio de um delineamento baseado em *comparações cegas* entre versões original/refatorada, aliado a entrevistas semiestruturadas, combinação que permitiu capturar tanto preferências objetivas quanto justificativas subjetivas sobre os critérios usados em julgamentos de qualidade de software.

Os resultados indicam congruência no conceito de qualidade de software, englobando principalmente legibilidade, manutenibilidade e modularidade. Em relação às comparações de código, observou-se preferência sistemática pelas versões refatoradas: a preferência foi majoritária entre os participantes como um todo (83%), confirmando-se em 97% dos artefatos analisados e em 97% das comparações, com ênfase em atributos como legibilidade, manutenibilidade e modularidade. Importante destacar que a revelação de que as refatorações foram produzidas por IA não alterou substancialmente os julgamentos, sugerindo que as preferências se fundamentaram em critérios técnicos e não em preconceitos quanto à autoria. Por fim, foi reconhecido o benefício da utilização de LLMs, com convergência na ideia de que a adoção futura deve ocorrer com supervisão crítica, integração gradual e adaptação de práticas de engenharia, reforçando a

complementaridade entre automação e expertise profissional.

Apesar do quadro positivo, as análises também evidenciaram limitações e pontos de atenção: riscos de *over-engineering* e perda de contexto em trechos complexos, o que reforça a necessidade de supervisão humana criteriosa na incorporação de LLMs a fluxos de desenvolvimento. Tais achados consolidam três contribuições principais: (i) evidências empíricas inéditas sobre a aceitação de refatorações realizadas por LLMs sob a ótica de desenvolvedores; (ii) demonstração da utilidade de comparações cegas como método para avaliar qualidade de código do ponto de vista humano; e (iii) mapeamento de critérios e limites práticos que orientam uma adoção responsável dessas ferramentas.

Como direções futuras, recomenda-se avançar com estudos longitudinais e avaliações em contextos reais; comparar sistematicamente com ferramentas de referência para quantificar vantagens e limites; ampliar e estratificar a base de participantes; e integrar os resultados a métricas objetivas de eficiência (tempo em tarefa, taxa de erro, *logs*) para fortalecer a robustez inferencial. Em síntese, os achados ampliam a compreensão sobre como integrar, de forma responsável, automação baseada em LLMs ao desenvolvimento de software, preservando critérios técnicos de qualidade e o protagonismo da revisão humana.

6 CONSIDERAÇÕES FINAIS

Este capítulo é dividido em três partes. A Seção 6.1 apresenta uma síntese dos resultados da dissertação. A Seção 6.2 discute implicações e lições aprendidas para pesquisadores, construtores de ferramentas e desenvolvedores. Por fim, a Seção 6.3 aponta direções para trabalhos futuros.

6.1 Conclusões

Esta dissertação investigou qualidade de código a partir de três eixos complementares: (i) detecção de *code smells* com algoritmos de ML; (ii) desenvolvimento e validação da ferramenta TwinCode para apoiar estudos empíricos; e (iii) análise da percepção de desenvolvedores sobre refatorações produzidas por LLMs.

No Capítulo 3, é mostrado que modelos supervisionados alcançam desempenho elevado na detecção dos *code smells Data Class*, *God Class*, *Feature Envy* e *Long Method*. Em termos de acurácia, observou-se variação de **89,7% a 99,2%**, com picos de **96,8%** (*Data Class*), **96,3%** (*God Class*), **98,4%** (*Feature Envy*) e **99,2%** (*Long Method*). É importante destacar a superioridade consistente de métodos baseados em árvores (especialmente *Random Forest* e *Decision Tree*). Adicionalmente, testes de Wilcoxon não indicaram diferenças estatisticamente significativas entre cenários com e sem validação cruzada de 10 *folds* (todos os p-valores $> 0{,}05$), sugerindo robustez dos resultados sob esse procedimento.

No Capítulo 4, é apresentado a **TwinCode**, ferramenta científica, que dentre outras funções, é utilizada para comparação lado a lado de trechos de código com questionários e geração de relatórios. A validação exploratória contou com 12 participantes, indicando *núcleo de inspeção* bem avaliado (visualização lado a lado, numeração de linhas e *syntax highlighting*), **alta consistência interna** do instrumento quantitativo e *potencial de adoção acadêmica* elevado. Ao mesmo tempo, evidenciou oportunidades de melhoria no fluxo de associação pares–questionários, ergonomia/UX e funcionalidades auxiliares (p. ex., exportação e filtros). A ferramenta foi **registrada no INPI** (BR512025003573-0).

No Capítulo 5, é conduzido entrevistas semi-estruturadas com sete desenvolvedores, cada um comparando cinco pares de código em desenho cego (total de 35 julgamentos) selecionados de um conjunto de 80 artefatos refatorados por *Qwen2.5-max* e derivados de 12 sistemas do *Qualitas Corpus*. Os entrevistados **preferiram o código refatorado em 34 de 35 comparações**

($\approx 97{,}1\%$; teste binomial, $p < 0{,}001$). As justificativas convergiram para *legibilidade*, *modularidade* e *manutenibilidade* como critérios centrais, com ressalvas à *complexidade desnecessária* em alguns trechos. A revelação posterior de que as versões escolhidas eram produzidas por LLM não alterou a fundamentação das decisões, ancoradas em atributos intrínsecos de qualidade.

Em conjunto, os três eixos mostram que abordagens baseadas em ML e LLMs podem se complementar. Modelos supervisionados fornecem *sinais objetivos* consistentes para detecção de *code smells*. Por outro lado, refatorações assistidas por LLM tendem a produzir versões preferidas por desenvolvedores sob critérios humanos de qualidade. A TwinCode, por sua vez, viabiliza a investigação empírica controlada que conecta essas duas frentes.

6.2 Implicações e Lições Aprendidas

Implicações e Lições Aprendidas para Pesquisadores. Com este estudo, identificam-se três principais implicações para pesquisadores: (i) o *gap* entre métricas objetivas e julgamento humano permanece e requer *protocolos mistos* (quantitativos e qualitativos) para interpretação rigorosa; (ii) *árvores de decisão* e *ensembles* mostraram-se linhas de base fortes para detecção de *code smells*, úteis como *baselines* em estudos comparativos; (iii) a ausência de diferenças materiais entre cenários com/sem validação cruzada (10-*fold*) sugere robustez dos achados frente a esse controle específico, embora não elimine outras ameaças (p. ex., desbalanceamento e *leakage*). Como lições aprendidas, destacam-se três pontos: (i) a necessidade de reportar resultados por *smell* (e não apenas médias) e utilizar testes não paramétricos para contrastar *setups* experimentais; (ii) a importância de triangular *detecção* (ML) com *percepção* (estudos cegos), empregando instrumentos com consistência interna verificada ($\alpha \approx 0{,}9$); e (iii) a relevância de utilizar amostras derivadas de datasets curados/conhecidos (p. ex., Qualitas Corpus) e rastrear o funil de seleção de artefatos.

Implicações e Lições Aprendidas para Construtores de Ferramentas. A avaliação da TwinCode indica que *comparação lado a lado* com realce e numeração de linhas é rapidamente compreendida, enquanto o *fluxo de associação* pares–questionários demanda simplificação e *feedbacks* de estado mais salientes. Recursos de *exportação*, *filtros* e *versionamento* aumentam a utilidade científica e a adoção. Como lições aprendidas citamos: (i) priorizar *UX research* no fluxo de criação/associação de comparações e questionários; (ii) oferecer telemetria e exportação de relatórios para reuso/reprodutibilidade de estudos; (iii) registrar propriedade intelectual e adotar *stack* aberto para facilitar replicação (PHP/Laravel, React/Tailwind, MariaDB).

Implicações e Lições Aprendidas para Desenvolvedores. Em comparações cegas, desenvolvedores tenderam a preferir versões refatoradas por LLM com base em *legibilidade*, *modularidade* e *manutenibilidade*; ainda assim, salientaram riscos de *complexidade desnecessária* e perda de contexto, reforçando a necessidade de *supervisão humana* e revisão criteriosa. Como lições aprendidas listamos: (i) usar LLMs como *copilotos de refatoração*, priorizando legibilidade (nomes, extração de métodos) e modularidade observáveis; (ii) aplicar revisão técnica para mitigar sobre-engenharia (p. ex., fragmentação excessiva e condicionais aninhadas); e (iii) integrar ferramentas experimentais (como a TwinCode) em *code reviews* internos para coletar feedback estruturado de outros desenvolvedores.

6.3 Trabalhos Futuros

A continuidade desta pesquisa pode avançar em diversas frentes, a começar pelo aprimoramento dos experimentos com aprendizado de máquina para detecção de *code smells*. Embora este estudo tenha considerado acurácia, precisão, sensibilidade e F1-*score*, futuras investigações podem incluir métricas adicionais, como AUC (Área sob a Curva ROC) e MCC (*Matthews Correlation Coefficient*), que fornecem uma avaliação mais robusta em cenários desbalanceados e permitem análises mais aprofundadas do desempenho dos algoritmos. Além disso, recomenda-se a utilização de diferentes conjuntos de dados, contemplando múltiplas linguagens de programação e domínios de aplicação, de modo a ampliar a validade externa dos resultados. Outro caminho promissor consiste em investigar técnicas de aprendizado profundo e arquiteturas híbridas que possam capturar padrões mais complexos em códigos de maior escala.

Em relação à *TwinCode*, trabalhos futuros devem priorizar sua validação em contextos mais amplos e diversificados, com amostras que contemplem diferentes níveis de experiência, formações acadêmicas e inserções profissionais, de modo a ampliar a validade externa dos resultados. Recomenda-se também a realização de estudos longitudinais que acompanhem a evolução do uso da ferramenta ao longo do tempo. Outra direção promissora consiste em expandir o suporte para diferentes artefatos de *software*, para além do código-fonte, o que pode ampliar a utilidade científica e profissional da ferramenta como recurso de apoio a investigações empíricas em engenharia de software. Também se destaca a possibilidade de implementar mecanismos que favoreçam o acompanhamento das fases de pesquisas científicas, bem como a inclusão de sistemas de pontuação nos questionários, de forma a apoiar práticas de avaliação da qualidade de software no ensino.

No que se refere à investigação da percepção de desenvolvedores sobre refatorações geradas por LLMs, trabalhos futuros podem aprofundar a análise em cenários mais complexos e heterogêneos, explorando diferentes linguagens de programação, tipos de sistemas e níveis de complexidade do código. Recomenda-se a realização de experimentos com maior número de participantes e delineamentos comparativos entre grupos de perfis distintos, permitindo identificar fatores culturais, organizacionais e individuais que influenciam a aceitação das refatorações. Além disso, a congruência entre percepções subjetivas e métricas objetivas de qualidade (tempo em tarefa, taxa de erros, métricas de legibilidade e manutenibilidade) pode fornecer uma visão mais completa da eficácia das LLMs. Estudos longitudinais, que acompanhem a evolução da aceitação e eficácia dessas ferramentas ao longo do tempo, também se mostram fundamentais para compreender sua evolução, uso e aderência de pesquisadores.

REFERÊNCIAS

ABDOU, A.; DARWISH, N. Severity classification of software code smells using machine learning techniques: A comparative study. **Journal of Software: Evolution and Process**, Wiley Online Library, v. 36, n. 1, p. e2454, 2024.

ACHIAM, J.; ADLER, S.; AGARWAL, S.; AHMAD, L.; AKKAYA, I.; ALEMAN, F. L.; ALMEIDA, D.; ALTENSCHMIDT, J.; ALTMAN, S.; ANADKAT, S. *et al.* Gpt-4 technical report. **arXiv preprint arXiv:2303.08774**, 2023.

AL-SHALABI, R.; KANAAN, G.; GHARAIBEH, M. Arabic text categorization using knn algorithm. In: **Proceedings of The 4th international multiconference on computer science and information technology**. [S.l.: s.n.], 2006. v. 4, p. 5–7.

ALAZBA, A.; ALJAMAAN, H.; ALSHAYEB, M. Deep learning approaches for bad smell detection: a systematic literature review. **Empirical Software Engineering**, Springer, v. 28, n. 3, p. 77, 2023.

ARLOT, S.; CELISSE, A. A survey of cross-validation procedures for model selection. 2010.

BARDIN, L. **Análise de Conteúdo**. São Paulo: Edições 70, 2016.

BECK, K. **Test-driven Development: By Example**. [S.l.]: Addison-Wesley Professional, 2003.

BEGEL, A.; SIMON, B. Novice software developers, all over again. In: **Proceedings of the fourth international workshop on computing education research**. [S.l.: s.n.], 2008. p. 3–14.

BINKLEY, D.; DAVIS, M.; LAWRIE, D.; MALETIC, J. I.; MORRELL, C.; SHARIF, B. The impact of identifier style on effort and comprehension. **Empirical software engineering**, Springer, v. 18, n. 2, p. 219–276, 2013.

BLINOWSKI, G.; OJDOWSKA, A.; PRZYBYŁEK, A. Monolithic vs. microservice architecture: A performance and scalability evaluation. **IEEE access**, IEEE, v. 10, p. 20357–20374, 2022.

BOEHM, B. W. Software engineering. **IEEE Trans. Comput.**, IEEE Computer Society, USA, v. 25, n. 12, p. 1226–1241, dez. 1976. ISSN 0018-9340. Disponível em: <https://doi.org/10.1109/TC.1976.1674590>.

BOEHM, B. W.; BROWN, J. R.; LIPOW, M. Quantitative evaluation of software quality. In: **Proceedings of the 2nd International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society Press, 1976. (ICSE '76), p. 592–605.

BÖRSTLER, J.; BENNIN, K. E.; HOOSHANGI, S.; JEURING, J.; KEUNING, H.; KLEINER, C.; MACKELLAR, B.; DURAN, R.; STÖRRLE, H.; TOLL, D. *et al.* Developers talking about code quality. **Empirical Software Engineering**, Springer, v. 28, n. 6, p. 128, 2023.

BREIMAN, L. Random forests. **Machine learning**, Springer, v. 45, p. 5–32, 2001.

BREKALO, S.; SEDLAREVIĆ, T. The comparison of monolithic mvc and microservices architectures in laravel applications. **Politehnika i dizajn**, v. 12, n. 03, 2024.

BROWN, T. B.; MANN, B.; RYDER, N.; SUBBIAH, M.; KAPLAN, J. D.; DHARIWAL, P.; NEELAKANTAN, A.; SHYAM, P.; SASTRY, G.; ASKELL, A.; AGARWAL, S.; HERBERT-VOSS, A.; KRUEGER, G.; HENIGHAN, T.; CHILD, R.; RAMESH, A.; ZIEGLER, D. M.; WU, J.; WINTER, C.; HESSE, C.; CHEN, M.; SIGLER, E.; LITWIN, M.; GRAY, S.; CHESS, B.; CLARK, J.; BERNER, C.; MCCANDLISH, S.; RADFORD, A.; SUTSKEVER, I.; AMODEI, D. Language models are few-shot learners. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2020. v. 33, p. 1877–1901.

BROWN, W. H.; MALVEAU, R. C.; MCCORMICK, H. W. S.; MOWBRAY, T. J. **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. 1st. ed. USA: John Wiley & Sons, Inc., 1998. ISBN 978-0-471-19713-3.

BUSE, R. P.; WEIMER, W. R. Learning a metric for code readability. **IEEE Transactions on software engineering**, IEEE, v. 36, n. 4, p. 546–558, 2009.

CATAL, C. Performance evaluation metrics for software fault prediction studies. **Acta Polytechnica Hungarica**, Obuda University, v. 9, n. 4, p. 193–206, 2012.

CHEN, A.; WONG, C.; SHARIF, B.; PERUMA, A. Exploring code comprehension in scientific programming: Preliminary insights from research scientists. **arXiv preprint arXiv:2501.10037**, 2025.

CHEN, M.; TWOREK, J.; JUN, H.; YUAN, Q.; PINTO, H. P. D. O.; KAPLAN, J.; EDWARDS, H.; BURDA, Y.; JOSEPH, N.; BROCKMAN, G. *et al.* Evaluating large language models trained on code. **arXiv preprint arXiv:2107.03374**, 2021.

CHEN, T. The impact of ai-pair programmers on code quality and developer satisfaction: Evidence from timi studio. In: **Proceedings of the 2024 International Conference on Generative Artificial Intelligence and Information Security**. [S.l.: s.n.], 2024. p. 201–205.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Transactions on software engineering**, IEEE, v. 20, n. 6, p. 476–493, 1994.

CIUPKE, O. Automatic detection of design problems in object-oriented reengineering. In: IEEE. **Proceedings of technology of object-oriented languages and systems-TOOLS 30 (Cat. No. PR00278)**. [S.l.], 1999. p. 18–32.

COLEMAN, D.; ASH, D.; LOWTHER, B.; OMAN, P. Using metrics to evaluate software system maintainability. **Computer**, IEEE, v. 27, n. 8, p. 44–49, 1994.

CRONBACH, L. J. Coefficient alpha and the internal structure of tests. **psychometrika**, Springer-Verlag, v. 16, n. 3, p. 297–334, 1951.

DEWANGAN, S.; RAO, R. S.; MISHRA, A.; GUPTA, M. A novel approach for code smell detection: An empirical study. **IEEE Access**, v. 9, p. 162869–162883, 2021.

FAKHOURY, S.; MA, Y.; ARNAOUDOVA, V.; ADESOPE, O. The effect of poor source code lexicon and readability on developers' cognitive load. In: **Proceedings of the 26th conference on program comprehension**. [S.l.: s.n.], 2018. p. 286–296.

FALESSI, D.; JURISTO, N.; WOHLIN, C.; TURHAN, B.; MÜNCH, J.; JEDLITSCHKA, A.; OIVO, M. Empirical software engineering experts on the use of students and professionals in experiments. **Empirical Software Engineering**, Springer, v. 23, n. 1, p. 452–489, 2018.

FONTANA, F. A.; FERME, V.; ZANONI, M.; ROVEDA, R. Towards a prioritization of code debt: A code smell intensity index. In: IEEE. **2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)**. [S.l.], 2015. p. 16–24.

FONTANA, F. A.; MÄNTYLÄ, M.; ZANONI, M.; MARINO, A. Comparing and experimenting machine learning techniques for code smell detection. **Empirical Software Engineering**, v. 21, p. 1143–1191, 2016. Disponível em: <https://api.semanticscholar.org/CorpusID:16222152>.

FONTANA, F. A.; ZANONI, M. Code smell severity classification using machine learning techniques. **Knowledge-Based Systems**, Elsevier, v. 128, p. 43–58, 2017.

FOODY, G. M. Challenges in the real world use of classification accuracy metrics: From recall and precision to the matthews correlation coefficient. **Plos one**, Public Library of Science San Francisco, CA USA, v. 18, n. 10, p. e0291908, 2023.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. [S.l.]: Addison-Wesley, 1999.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. 2nd. ed. [S.l.]: Addison-Wesley, 2018.

FRICK, V.; WEDENIG, C.; PINZGER, M. Diffviz: A diff algorithm independent visualization tool for edit scripts. In: IEEE. **2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.], 2018. p. 705–709.

FRIEDMAN, J. H. Greedy function approximation: A gradient boosting machine. **The Annals of Statistics**, v. 29, n. 5, p. 1189–1232, out. 2001. ISSN 0090-5364, 2168-8966. Publisher: Institute of Mathematical Statistics. Disponível em: <https://projecteuclid.org/journals/annals-of-statistics/volume-29/issue-5/Greedy-function-approximation-A-gradient-boosting-machine/10.1214/aos/1013203451.full>.

GE, X.; DUBOSE, Q. L.; MURPHY-HILL, E. Reconciling manual and automatic refactoring. In: IEEE. **2012 34th International Conference on Software Engineering (ICSE)**. [S.l.], 2012. p. 211–221.

GLIEM, J. A.; GLIEM, R. R. Calculating, interpreting, and reporting cronbach's alpha reliability coefficient for likert-type scales. In: MIDWEST RESEARCH-TO-PRACTICE CONFERENCE IN ADULT, CONTINUING, AND COMMUNITY .... [S.l.], 2003.

GOPALAKRISHNA, A. K.; OZCELEBI, T.; LIOTTA, A.; LUKKIEN, J. J. Relevance as a metric for evaluating machine learning algorithms. In: SPRINGER. **International Workshop on Machine Learning and Data Mining in Pattern Recognition**. [S.l.], 2013. p. 195–208.

HAN, J.; PEI, J.; TONG, H. **Data mining: concepts and techniques**. [S.l.]: Morgan kaufmann, 2022. 64–65 p.

HE, J.; TREUDE, C.; LO, D. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. **ACM Transactions on Software Engineering and Methodology**, ACM New York, NY, v. 34, n. 5, p. 1–30, 2025.

HILMI, M. A. A.; PUSPANINGRUM, A.; SIAHAAN, D. O.; SAMOSIR, H. S.; RAHMA, A. S. *et al.* Research trends, detection methods, practices, and challenges in code smell: Slr. **IEEE Access**, IEEE, v. 11, p. 129536–129551, 2023.

HOSSIN, M.; SULAIMAN, M. N. A review on evaluation metrics for data classification evaluations. **International journal of data mining & knowledge management process**, Academy & Industry Research Collaboration Center (AIRCC), v. 5, n. 2, p. 1, 2015.

ISO/IEC. **ISO/IEC 9126. Software engineering – Product quality**. [S.l.]: ISO/IEC, 2001.

ISO/IEC. **ISO/IEC 25010. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models**. [S.l.]: ISO/IEC, 2011.

JOSHI, A.; KALE, S.; CHANDEL, S.; PAL, D. K. Likert scale: Explored and explained. **British journal of applied science & technology**, Sciencedomain International, v. 7, n. 4, p. 396, 2015.

KAUR, I.; KAUR, A. A novel four-way approach designed with ensemble feature selection for code smell detection. **IEEE Access**, IEEE, v. 9, p. 8695–8707, 2021.

KITCHENHAM, B.; PFLEEGER, S. L. Software quality: the elusive target [special issues section]. **IEEE software**, Ieee, v. 13, n. 1, p. 12–21, 1996.

KITCHENHAM, B. A. Evaluating software engineering methods and tool—part 2: selecting an appropriate evaluation method—technical criteria. **SIGSOFT Softw. Eng. Notes**, Association for Computing Machinery, New York, NY, USA, v. 21, n. 2, p. 11–15, mar. 1996. ISSN 0163-5948. Disponível em: <https://doi.org/10.1145/227531.227533>.

KOHAVI, R. *et al.* A study of cross-validation and bootstrap for accuracy estimation and model selection. In: MONTREAL, CANADA. **Ijcai**. [S.l.], 1995. v. 14, n. 2, p. 1137–1145.

KRASNER, H. The cost of poor software quality in the us: A 2020 report. **Proc. Consortium Inf. Softw. QualityTM (CISQTM)**, v. 2, p. 3, 2021.

KUDRIAVTSEVA, A.; HOTAK, N. A.; GADYATSKAYA, O. My code is less secure with gen ai: Surveying developers' perceptions of the impact of code generation tools on security. In: **Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing**. [S.l.: s.n.], 2025. p. 1637–1646.

LANZA, M.; MARINESCU, R. **Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems**. [S.l.]: Springer Science & Business Media, 2007.

LIGGESMEYER, P.; TRAPP, M. Trends in embedded software engineering. **IEEE software**, IEEE, v. 26, n. 3, p. 19–25, 2009.

LYU, M. R.; RAY, B.; ROYCHOUDHURY, A.; TAN, S. H.; THONGTANUNAM, P. Automatic programming: Large language models and beyond. **ACM Transactions on Software Engineering and Methodology**, ACM New York, NY, v. 34, n. 5, p. 1–33, 2025.

MA, W.; LIU, S.; LIN, Z.; WANG, W.; HU, Q.; LIU, Y.; ZHANG, C.; NIE, L.; LI, L.; LIU, Y. Lms: Understanding code syntax and semantics for code analysis. **arXiv preprint arXiv:2305.12138**, 2023.

MÄNTYLÄ, M. Two experiments on subjective evaluation of code evolvability. 2005.

MANTYLA, M.; VANHANEN, J.; LASSENIUS, C. A taxonomy and an initial empirical study of bad smells in code. In: IEEE. **International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.** [S.l.], 2003. p. 381–384.

MÄNTYLÄ, M. V.; LASSENIUS, C. Subjective evaluation of software evolvability using code smells: An empirical study. **Empirical Software Engineering**, Springer, v. 11, n. 3, p. 395–431, 2006.

MARTIN, R. C. **Clean code: a handbook of agile software craftsmanship**. [S.l.]: Pearson Education, 2009.

MCCABE, T. J. A complexity measure. **IEEE Transactions on software Engineering**, IEEE, n. 4, p. 308–320, 1976.

MCCALL, J. A.; RICHARDS, P. K.; WALTERS, G. F. **Factors in software quality. volume i. concepts and definitions of software quality**. [S.l.], 1977.

MCCONNELL, S. **Code complete : a practical handbook of sofware construction**. Microsoft Press, 2004. ISBN 978-0-7356-1967-8. Disponível em: <https://thuvienso.hoasen.edu.vn/handle/123456789/8847>.

MENS, T.; TOURWÉ, T. A survey of software refactoring. **IEEE Transactions on software engineering**, IEEE, v. 30, n. 2, p. 126–139, 2004.

MHAWISH, M. Y.; GUPTA, M. Predicting code smells and analysis of predictions: Using machine learning techniques and software metrics. **Journal of Computer Science and Technology**, Springer, v. 35, p. 1428–1445, 2020.

MITCHELL, T. M. Does machine learning really work? **AI magazine**, v. 18, n. 3, p. 11–11, 1997.

MOHA, N.; GUÉHÉNEUC, Y.-G.; DUCHIEN, L.; MEUR, A.-F. L. Decor: A method for the specification and detection of code and design smells. **IEEE Transactions on Software Engineering**, IEEE, v. 36, n. 1, p. 20–36, 2009.

MOREIRA, R.; BRAZ, L.; FERREIRA, F.; AMORA, M. Estudo empírico: detecção de code smells com aprendizado de máquinas. In: **Anais do XXVII Congresso Ibero-Americano em Engenharia de Software**. Porto Alegre, RS, Brasil: SBC, 2024. p. 301–312. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/cibse/article/view/28455>.

MOREIRA, R. A. F.; VALE, G. do; FERREIRA, F. J. **TwinCode**. 2025. GitHub. Disponível em: <https://github.com/alanfm/twincode>. Acesso em: 15 ago. 2025.

NIELSEN, J. Heuristic evaluation. In: **Usability inspection methods**. [S.l.: s.n.], 1994. p. 25–62.

NOBLE, W. S. What is a support vector machine? **Nature biotechnology**, Nature Publishing Group UK London, v. 24, n. 12, p. 1565–1567, 2006.

NUCCI, D. D.; PALOMBA, F.; TAMBURRI, D. A.; SEREBRENIK, A.; LUCIA, A. D. Detecting code smells using machine learning techniques: are we there yet? In: IEEE. **2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)**. [S.l.], 2018. p. 612–621.

OPDYKE, W. F. **Refactoring object-oriented frameworks**. [S.l.]: University of Illinois at Urbana-Champaign, 1992.

OUNI, A.; KESSENTINI, M.; CINNÉIDE, M. Ó.; SAHRAOUI, H.; DEB, K.; INOUE, K. More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. **Journal of Software: Evolution and Process**, Wiley Online Library, v. 29, n. 5, p. e1843, 2017.

OUNI, A.; KESSENTINI, M.; SAHRAOUI, H.; INOUE, K.; DEB, K. Multi-criteria code refactoring using search-based software engineering: An industrial case study. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM New York, NY, USA, v. 25, n. 3, p. 1–53, 2016.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; FASANO, F.; OLIVETO, R.; LUCIA, A. D. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In: **Proceedings of the 40th International Conference on Software Engineering**. [S.l.: s.n.], 2018. p. 482–482.

PALOMBA, F.; NUCCI, D. D.; TUFANO, M.; BAVOTA, G.; OLIVETO, R.; POSHYVANYK, D.; LUCIA, A. D. Landfill: An open dataset of code smells with public evaluation. In: IEEE. **2015 IEEE/ACM 12th Working Conference on Mining Software Repositories**. [S.l.], 2015. p. 482–485.

POSNETT, D.; HINDLE, A.; DEVANBU, P. A simpler model of software readability. In: **Proceedings of the 8th working conference on mining software repositories**. [S.l.: s.n.], 2011. p. 73–82.

PRESSMAN, R. S. *et al.* **Engenharia de software**. [S.l.]: Makron books São Paulo, 1995. v. 6.

PUSHPALATHA, M.; MRUNALINI, M. Predicting the severity of open source bug reports using unsupervised and supervised techniques. In: **Research Anthology on Usage and Development of Open Source Software**. [S.l.]: IGI Global, 2021. p. 676–692.

QUINLAN, J. R. Decision trees and decision-making. **IEEE Transactions on Systems, Man, and Cybernetics**, IEEE, v. 20, n. 2, p. 339–346, 1990.

RAHMAN, H. U.; SILVA, A. R. da; ALZAYED, A.; RAZA, M. A systematic literature review on software maintenance offshoring decisions. **Information and Software Technology**, Elsevier, v. 172, p. 107475, 2024.

RAY, S. A quick review of machine learning algorithms. In: IEEE. **2019 International conference on machine learning, big data, cloud and parallel computing (COMITCon)**. [S.l.], 2019. p. 35–39.

RIBEIRO, F.; FERNANDES, E.; FIGUEIREDO, E. A preliminary interview study on developers' perceptions of code smell detection in industry. In: SPRINGER. **International Conference on the Quality of Information and Communications Technology**. [S.l.], 2024. p. 344–352.

RIEL, A. J. **Object-Oriented Design Heuristics**. 1st. ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN 978-0-201-63385-6.

ROZIERE, B.; GEHRING, J.; GLOECKLE, F.; SOOTLA, S.; GAT, I.; TAN, X. E.; ADI, Y.; LIU, J.; SAUVESTRE, R.; REMEZ, T. *et al.* Code llama: Open foundation models for code. **arXiv preprint arXiv:2308.12950**, 2023.

RUCK, D. W.; ROGERS, S. K.; KABRISKY, M. Feature selection using a multilayer perceptron. **Journal of neural network computing**, v. 2, n. 2, p. 40–48, 1990.

RUNESON, P.; HOST, M.; RAINER, A.; REGNELL, B. **Case study research in software engineering: Guidelines and examples**. [S.l.]: John Wiley & Sons, 2012.

RUSSELL, S. J.; NORVIG, P. Artificial intelligence: A modern approach, global edition 4e. Pearson, 2021.

SAHIN, D.; KESSENTINI, M.; BECHIKH, S.; DEB, K. Code-smell detection as a bilevel problem. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM New York, NY, USA, v. 24, n. 1, p. 1–44, 2014.

SANTOS, R. M. a. dos; GEROSA, M. A. Impacts of coding practices on readability. In: **Proceedings of the 26th Conference on Program Comprehension**. New York, NY, USA: Association for Computing Machinery, 2018. (ICPC '18), p. 277–285. ISBN 9781450357142. Disponível em: <https://doi.org/10.1145/3196321.3196342>.

SAURO, J.; LEWIS, J. R. **Quantifying the user experience: Practical statistics for user research**. [S.l.]: Morgan Kaufmann, 2016.

SCALABRINO, S.; LINARES-VASQUEZ, M.; POSHYVANYK, D.; OLIVETO, R. Improving code readability models with textual features. In: IEEE. **2016 IEEE 24th International Conference on Program Comprehension (ICPC)**. [S.l.], 2016. p. 1–10.

SHEARD, J.; DENNY, P.; HELLAS, A.; LEINONEN, J.; MALMI, L.; SIMON. Instructor perceptions of ai code generation tools-a multi-institutional interview study. In: **Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1**. [S.l.: s.n.], 2024. p. 1223–1229.

TARWANI, S.; CHUG, A. Predicting maintainability of open source software using gene expression programming and bad smells. In: IEEE. **2016 5th international conference on reliability, Infocom technologies and optimization (trends and future directions)(ICRITO)**. [S.l.], 2016. p. 452–459.

TEMPERO, E.; ANSLOW, C.; DIETRICH, J.; HAN, T.; LI, J.; LUMPE, M.; MELTON, H.; NOBLE, J. The qualitas corpus: A curated collection of java code for empirical studies. In: **2010 Asia Pacific Software Engineering Conference**. [S.l.: s.n.], 2010. p. 336–345.

TRAVASSOS, G.; SHULL, F.; FREDERICKS, M.; BASILI, V. R. Detecting defects in object-oriented designs: using reading techniques to increase software quality. **ACM sigplan notices**, ACM New York, NY, USA, v. 34, n. 10, p. 47–56, 1999.

TSANTALIS, N.; CHATZIGEORGIOU, A. Identification of move method refactoring opportunities. **IEEE Transactions on Software Engineering**, IEEE, v. 35, n. 3, p. 347–367, 2009.

TSANTALIS, N.; CHATZIGEORGIOU, A. Ranking refactoring suggestions based on historical volatility. In: IEEE. **2011 15th European conference on software maintenance and reengineering**. [S.l.], 2011. p. 25–34.

VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, Ł.; POLOSUKHIN, I. Attention is all you need. **Advances in neural information processing systems**, v. 30, 2017.

VELASCO, A.; RODRIGUEZ-CARDENAS, D.; ALIF, L. R.; PALACIO, D. N.; POSHYVANYK, D. How propense are large language models at producing code smells? a benchmarking study. In: IEEE. **2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)**. [S.l.], 2025. p. 96–100.

WASHIZAKI, H.; YAMAMOTO, H.; FUKAZAWA, Y. A metrics suite for measuring reusability of software components. In: IEEE. **Proceedings. 5th International Workshop on enterprise networking and computing in healthcare industry (IEEE Cat. No. 03EX717)**. [S.l.], 2004. p. 211–223.

WHITE, J.; FU, Q.; HAYS, S.; SANDBORN, M.; OLEA, C.; GILBERT, H.; ELNASHAR, A.; SPENCER-SMITH, J.; SCHMIDT, D. C. A prompt pattern catalog to enhance prompt engineering with chatgpt. **arXiv preprint arXiv:2302.11382**, 2023.

WILCOXON, F. Individual comparisons by ranking methods. **Biometrics Bulletin**, International Biometric Society, v. 1, n. 6, p. 80–83, 1945. Disponível em: <https://doi.org/10.2307/3001968>.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in Software Engineering**. Berlin, Heidelberg: Springer, 2012. ISBN 978-3-642-29044-2.

XUE, Z.; ZHANG, X.; GAO, Z.; HU, X.; GAO, S.; XIA, X.; LI, S. Clean code, better models: Enhancing llm performance with smell-cleaned dataset. **arXiv preprint arXiv:2508.11958**, 2025.

YAMASHITA, A.; COUNSELL, S. Code smells as system-level indicators of maintainability: An empirical study. **Journal of Systems and Software**, Elsevier, v. 86, n. 10, p. 2639–2653, 2013.

YAMASHITA, A.; MOONEN, L. Do code smells reflect important maintainability aspects? In: IEEE. **2012 28th IEEE international conference on software maintenance (ICSM)**. [S.l.], 2012. p. 306–315.

YAMASHITA, A.; MOONEN, L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: IEEE. **2013 35th International Conference on Software Engineering (ICSE)**. [S.l.], 2013. p. 682–691.

APÊNDICE A – CODIFICAÇÃO SEMÂNTICA DETALHADA

Este apêndice apresenta a codificação semântica detalhada das entrevistas realizadas com os sete participantes do estudo. O material foi estruturado individualmente por entrevistado, de modo a destacar as falas que fundamentaram a atribuição de pontuação para cada critério de qualidade de código identificado na análise (legibilidade, manutenibilidade, modularidade, padronização, simplicidade e funcionalidade). A organização em categorias permite visualizar como diferentes níveis de senioridade e áreas de atuação influenciaram as percepções sobre qualidade de código, além de fornecer evidências textuais que sustentam as análises discutidas no Capítulo 5. Dessa forma, este apêndice cumpre o papel de tornar transparente o processo de análise qualitativa, reforçando a validade interpretativa dos resultados apresentados.

Entrevistado 1 - E1 (Sênior)

Legibilidade

- **Falas que geraram pontuação**:
    1. “Para mim, código bem feito, ele tem que ser primeiramente legível. Quanto mais legível, melhor.”
    2. “A da direita tá bem mais tranquila, bem mais fácil de entender.”
    3. “Não dá pra visualizar exatamente, mas a da direita reduziu bem, ficou mais legível.”
    4. “Legibilidade é o critério prioritário pra mim.”
    5. “Prefiro código que qualquer pessoa bata o olho e entenda.”

Manutenibilidade

- **Falas que geraram pontuação**:
    1. “O código 2, sendo pequeno e curto, dá para analisar de melhor forma. Seria melhor para longo prazo.”
    2. “Quando separa em classes diferentes, cada uma com sua responsabilidade, fica mais fácil dar manutenção.”
    3. “A simplicidade ajuda na manutenção futura.”

Modularidade

- **Falas que geraram pontuação**:
  1. “O lado direito separou em duas classes diferentes, cada uma com sua responsabilidade.”
  2. “O uso de interface ajuda na modularidade e manutenção.”

Padronização

- **Fala que gerou pontuação**:
  1. “Um código bem escrito segue padrão de nomenclatura, senão dificulta.”

Funcionalidade

- **Fala que gerou pontuação**:
  1. “O código precisa funcionar corretamente, não adianta ser só bonito.”

Simplicidade

- **Falas que geraram pontuação**:
  1. “Prefiro código conciso e claro.”
  2. “Quanto menos linhas desnecessárias, melhor.”

Entrevistado 2 - E2 (Júnior)

Legibilidade

- **Falas que geraram pontuação**:
  1. “Eu gosto de códigos que sejam pequenos, mas que sejam legíveis.”
  2. “Não muito resumido, nem muito extenso.”
  3. “O segundo é mais fácil de ler.”
  4. “Prefiro código que dê pra entender rápido.”

Simplicidade

- **Fala que gerou pontuação**:

1. “Não muito resumido, nem muito extenso, um meio termo.”

Funcionalidade

- **Fala que gerou pontuação**:
  1. “O código tem que ser funcional.”

Modularidade

- **Falas que geraram pontuação**:
  1. “Gostei do código 2 porque fez em várias classes, orientação a objetos.”
  2. “Organizar em classes ajuda a entender.”

Entrevistado 3 - E3 (Pleno)

Legibilidade

- **Falas que geraram pontuação**:
  1. “A da direita tá bem mais tranquila, bem mais fácil de entender.”
  2. “Prefiro código que facilite leitura.”
  3. “Separo legibilidade de manutenção, mas ambos caminham juntos.”

Manutenibilidade

- **Falas que geraram pontuação**:
  1. “Qualidade de um código é refletida quando, com os anos, ainda consigo dar manutenção nele.”
  2. “Separar em classes ajuda a dar manutenção.”
  3. “Interfaces aumentam manutenção futura.”
  4. “Códigos longos dificultam manutenção.”

Modularidade

- **Falas que geraram pontuação**:
  1. “Uso de interface melhora a manutenabilidade.”
  2. “Separação de responsabilidades é importante.”

3. “Arquitetura impacta modularidade.”

Entrevistado 4 - E4 (Pleno)

Legibilidade

- **Falas que geraram pontuação**:
  1. “Claramente, a principal coisa que faz um código ser bom é ser legível e padronizado.”
  2. “O código 1 tem muita coisa fora do padrão.”
  3. “Prefiro código padronizado e legível.”
  4. “Legibilidade e manutenção caminham juntas.”

Manutenibilidade

- **Falas que geraram pontuação**:
  1. “Um código precisa ser dinâmico e escalável. Se não for, deprecia rápido.”
  2. “Códigos mais limpos duram mais.”
  3. “Separação em camadas facilita manutenção.”

Modularidade

- **Falas que geraram pontuação**:
  1. “Sempre foco em ter controllers, views e models separados.”
  2. “Arquitetura MVC facilita modularidade.”
  3. “Classes bem separadas melhoram modularidade.”

Padronização

- **Falas que geraram pontuação**:
  1. “O código 1 tem muita coisa fora do padrão.”
  2. “Nomenclaturas inconsistentes atrapalham.”

Funcionalidade

- **Fala que gerou pontuação**:
  1. “Precisa também funcionar, não adianta ser só bonito.”

Entrevistado 5 - E5 (Júnior)

Legibilidade

- **Falas que geraram pontuação**:
  1. “Não tem nem como comparar, o código 1 tá enxuto, perfeito, bem resumido.”
  2. “O código 2 tá enorme e pouco legível.”
  3. “Prefiro código organizado em blocos.”

Manutenibilidade

- **Falas que geraram pontuação**:
  1. “Com certeza o código 1. Compactado, fácil de manter.”
  2. “Códigos mais simples são mais fáceis de mexer.”

Modularidade

- **Fala que gerou pontuação**:
  1. “O código 1 está bem separado em blocos.”

Padronização

- **Fala que gerou pontuação**:
  1. “Separação em blocos facilita leitura.”

Funcionalidade

- **Fala que gerou pontuação**:
  1. “Para iniciante, talvez o código 2 fosse mais fácil de ir seguindo.”

Entrevistado 6 - E6 (Júnior)

Legibilidade

- **Falas que geraram pontuação**:
  1. “O código 2 está bem mais modular, responsabilidades divididas, mais legível.”

2. “Prefiro o que facilita leitura.”
3. “Códigos claros ajudam.”

Manutenibilidade

- **Falas que geraram pontuação**:
  1. “Código 2 seria mais fácil ao longo prazo.”
  2. “Modularidade ajuda manutenção.”

Modularidade

- **Falas que geraram pontuação**:
  1. “O código 2 está bem dividido, mais classes, responsabilidades separadas.”
  2. “Classes pequenas melhoram modularidade.”
  3. “Responsabilidades bem definidas.”

Simplicidade

- **Fala que gerou pontuação**:
  1. “O código 1 assume muita responsabilidade, métodos gigantes, difícil de manter.”

Entrevistado 7 - E7 (Sênior)

Legibilidade

- **Falas que geraram pontuação**:
  1. “Um código de qualidade tem que ser fácil de ler, funcional e fácil de manter.”
  2. “Prefiro código claro.”
  3. “Legibilidade facilita colaboração.”
  4. “Código legível reduz erros.”

Manutenibilidade

- **Falas que geraram pontuação**:
  1. “Responsabilidades claras, lógica não complexa. Isso facilita manutenção.”
  2. “Códigos mais simples são melhores de manter.”

3. “Prefiro modularização para manutenção futura.”

Modularidade

- **Falas que geraram pontuação**:
  1. “As responsabilidades estão modularizadas em classes menores.”
  2. “Separar responsabilidades melhora modularidade.”

Padronização

- **Fala que gerou pontuação**:
  1. “Nomear bem variáveis e funções.”

Funcionalidade

- **Falas que geraram pontuação**:
  1. “Um código tem que ser funcional.”
  2. “Funcionalidade é essencial para qualidade.”

APÊNDICE B – ANÁLISE DE CONTEÚDO DAS ENTREVISTAS

Este apêndice apresenta a análise de conteúdo completa das entrevistas, sistematizada segundo a técnica de (BARDIN, 2016). As falas dos participantes foram segmentadas em unidades de registro, cada uma associada a um código e a uma categoria temática que reflete os critérios de qualidade de código identificados (legibilidade, manutenibilidade, modularidade, padronização, funcionalidade e simplicidade). Para cada unidade, são apresentados exemplos literais de falas, seguidos de uma interpretação analítica, permitindo compreender como os participantes justificaram suas escolhas e quais dimensões da qualidade foram mais valorizadas. Esta organização fornece transparência metodológica e reforça a validade dos resultados discutidos no Capítulo 5.

Tabela 16: Análise de Conteúdo das Entrevistas

Unidade de Registro	**Código**	**Categoria**	**Exemplo de Fala**	**Interpretação**
Legibilidade como critério central (E1)	LEGIB	Legibilidade	"*Código deve ser primeiramente legível; quanto mais legível, melhor.*" (E1)	Legibilidade aparece como eixo estruturante da avaliação de qualidade.
Equilíbrio entre tamanho e clareza (E2)	LEGIB	Legibilidade	"*Gosto de códigos pequenos, mas legíveis; nem muito resumidos nem extensos.*" (E2)	Legibilidade associada a tamanho adequado e compreensão imediata.
Facilidade de entendimento visual (E3)	LEGIB	Legibilidade	"*A versão da direita está mais tranquila, mais fácil de entender.*" (E3)	Clareza da estrutura e leitura fluida favorecem o julgamento positivo.
Legibilidade + padronização (E4)	LEGIB	Legibilidade	"*Código bom é legível e padronizado; o código 1 tem muita coisa fora do padrão.*" (E4)	A legibilidade é reforçada por convenções estáveis de escrita.
Texto enxuto e claro (E5)	LEGIB	Legibilidade	"*O código 1 está enxuto, bem resumido; o 2 é enorme e pouco legível.*" (E5)	Códigos mais curtos e diretos são percebidos como mais legíveis.
Legibilidade via modularidade (E6)	LEGIB	Legibilidade	"*O código 2 está mais modular, responsabilidades divididas, mais legível.*" (E6)	Separação de responsabilidades favorece a leitura.
Leitura e manutenção (E7)	LEGIB	Legibilidade	"*Código de qualidade tem que ser fácil de ler e manter.*" (E7)	Legibilidade articulada com manutenibilidade como critério de qualidade.

Unidade de Registro	Código	Categoria	Exemplo de Fala	Interpretação
Pequeno e melhor no longo prazo (E1)	MANUT	Manutenibilidade	"*O código 2, por ser curto, é melhor para analisar e manter no longo prazo.*" (E1)	Tamanho moderado reduz esforço de manutenção futura.
Teste e manutenção facilitados (E2)	MANUT	Manutenibilidade	"*No segundo é mais fácil de modularizar e testar.*" (E2)	Estrutura mais clara facilita evolução e correção.
Manutenção ao longo dos anos (E3)	MANUT	Manutenibilidade	"*Qualidade se reflete quando consigo dar manutenção com o passar dos anos.*" (E3)	Critério de qualidade vinculado ao ciclo de vida prolongado.
Dinamicidade e escalabilidade (E4)	MANUT	Manutenibilidade	"*Se não for dinâmico e escalável, o código deprecia rápido.*" (E4)	Manutenção depende de arquitetura pensada para evoluir.
Compacto e fácil de mexer (E5)	MANUT	Manutenibilidade	"*O código 1 é compactado e fácil de manter.*" (E5)	Estruturas concisas simplificam a manutenção cotidiana.
Modularidade ajuda manutenção (E6)	MANUT	Manutenibilidade	"*Código 2 seria mais fácil ao longo prazo; divisão em classes ajuda.*" (E6)	Separação de responsabilidades reduz acoplamento e custo de mudança.
Clareza de lógica e responsabilidades (E7)	MANUT	Manutenibilidade	"*Responsabilidades claras e lógica não complexa facilitam manutenção.*" (E7)	Boa organização interna favorece intervenções futuras.
Separação em classes (E1)	MODUL	Modularidade	"*Separou em classes diferentes, cada uma com sua responsabilidade.*" (E1)	Modularidade como mecanismo de controle de complexidade.
OO e organização em classes (E2)	MODUL	Modularidade	"*Gostei do código 2 por usar várias classes (orientação a objetos).*" (E2)	Distribuição de responsabilidades melhora entendimento.
Interfaces e manutenção (E3)	MODUL	Modularidade	"*Uso de interface melhora a manutenibilidade.*" (E3)	Contratos explícitos estabilizam integrações e evolução.
Arquitetura MVC (E4)	MODUL	Modularidade	"*Foco em manter controllers, views e models separados.*" (E4)	Padrões arquiteturais estruturam módulos e camadas.
Blocos organizados (E5)	MODUL	Modularidade	"*O código 1 está bem separado em blocos, legível.*" (E5)	Segmentação por blocos contribui para o raciocínio local.
Mais classes, menos acoplamento (E6)	MODUL	Modularidade	"*O código 2 está bem dividido, mais classes, responsabilidades separadas.*" (E6)	Modularidade reduz acoplamento e facilita reuso.
Modularização e reuso (E7)	MODUL	Modularidade	"*Responsabilidades modularizadas em classes menores facilitam reuso.*" (E7)	Componentização incentiva reutilização e evolução incremental.

Unidade de Registro	Código	Categoria	Exemplo de Fala	Interpretação
Padrões de escrita (E4)	PADR	Nomenclatura	"*Código bom é legível e padronizado; há coisas fora do padrão.*" (E4)	Conformidade com convenções melhora coesão do time.
Nomes descritivos (E7)	PADR	Nomenclatura	"*Nomear bem variáveis e funções.*" (E7)	Nomeação clara reduz ambiguidade e custos de leitura.
Observância de convenções (E1)	PADR	Nomenclatura	"*Código bem escrito segue padrão; sem isso, dificulta.*" (E1)	Consistência de estilo como sinal de maturidade e qualidade.
Cumprimento de requisitos (E2)	FUNC	Funcionalidade	"*O código tem que ser funcional.*" (E2)	Funcionamento correto é requisito mínimo de qualidade.
Funcionar antes de estética (E4)	FUNC	Funcionalidade	"*Não adianta ser só bonito; precisa funcionar.*" (E4)	Prioridade para comportamento correto sobre aparência.
Funcional e sustentável (E7)	FUNC	Funcionalidade	"*Código de qualidade é funcional, fácil de ler e manter.*" (E7)	Funcionalidade integrada a legibilidade e manutenção.
Preferência por soluções enxutas (E1)	SIMP	Simplicidade	"*Prefiro código conciso e claro.*" (E1)	Remoção de complexidade acidental e foco no essencial.
Evitar excesso de extensão (E2)	SIMP	Simplicidade	"*Pequeno e legível; nem muito resumido nem muito extenso.*" (E2)	Simplicidade como equilíbrio entre brevidade e clareza.
Crítica a métodos gigantes (E6)	SIMP	Simplicidade	"*Métodos gigantes e muita responsabilidade; difícil de manter.*" (E6)	Indício de necessidade de decomposição/refatoração.
Critério espontâneo de clareza (E1)	ESPONT	Critérios espontâneos	"*Prefiro código que qualquer pessoa bata o olho e entenda.*" (E1)	Legibilidade mencionada de forma espontânea como prioridade universal.
Critério espontâneo de eficiência (E6)	ESPONT	Critérios espontâneos	"*Códigos claros ajudam, mas também tem que ser eficientes.*" (E6)	Eficiência aparece associada ao julgamento de clareza.
Escolha pelo código enxuto (E5)	PREF	Preferência	"*Não tem nem como comparar, o código 1 tá enxuto, perfeito, bem resumido.*" (E5)	Preferência clara pelo código mais compacto.
Preferência por modularidade (E6)	PREF	Preferência	"*O código 2 está bem mais modular, responsabilidades divididas.*" (E6)	Opção motivada pela separação de responsabilidades.
Justificativa por legibilidade (E3)	JUST	Justificativa	"*A da direita tá bem mais tranquila, bem mais fácil de entender.*" (E3)	Escolha fundamentada na facilidade de leitura.

Unidade de Registro	Código	Categoria	Exemplo de Fala	Interpretação
Justificativa por padronização (E4)	JUST	Justificativa	"*O código 1 tem muita coisa fora do padrão.*" (E4)	Critério de padronização norteia a decisão.
IA como ferramenta de apoio (E5)	IAPOS	Pós-revelação	"*Eu acredito que ela vai ser uma ferramenta, né? Pra auxiliar. Acho que é isso.*" (E5)	Percepção positiva, mas reforçando caráter complementar da IA.
Risco no uso em produção (E7)	IAPOS	Pós-revelação	"*Claro que eu evitaria de usar LLM pra código direto em produção. Então, acho que uma revisão de códigos, ela é necessária.*" (E7)	Ênfase na necessidade de supervisão humana.
IA como catalisador de produtividade (E3)	FUT	Expectativas futuras	"*Vai deixar a coisa muito mais fluida [...] mas acabar [com o trabalho do dev], acho que não.*" (E3)	Expectativa de ganhos de produtividade sem substituição completa.
Integração inevitável (E7)	FUT	Expectativas futuras	"*Se você não usa LLM, você tá ficando pra trás. Você tá perdendo uma oportunidade.*" (E7)	Expectativa de adoção crescente como diferencial competitivo.

Fonte: Elaborado pelo autor (2025)

APÊNDICE C – QUADROS DE ESCOLHAS E JUSTIFICATIVAS DOS ENTREVISTADOS

Este apêndice apresenta os quadros completos com as escolhas de código (original ou refatorado) e as respectivas justificativas fornecidas pelos entrevistados (E1–E7), referentes à **Questão de Pesquisa 2 (QP2)**.

Tabela 17: Escolhas e justificativas do entrevistado E1 nas cinco comparações

Comparação	Código escolhido	Justificativa
1ª	Refatorado	"*Cara, esse lado esquerdo está muito complexo... o outro está bem mais enxuto, simples. Cada função no seu lugar... Eu acho que seria o dois. Sem dúvida.*"
2ª	Refatorado	"*Sem dúvida o código 1... Para manutenção, sem dúvida o código 1. Para clareza, para adicionar funcionalidade, eu escolheria ele.*"
3ª	Refatorado	"*Legibilidade do lado esquerdo... Do lado direito está muito verboso, muito complexo... acoplando demais... O esquerdo está mais enxuto, responsabilidades isoladas.*"
4ª	Refatorado	"*Os dois estão bem... O um está mais verboso, mas ainda assim legível. O dois está menos verboso e a leitura está mais linear... Para manutenção, o segundo.*"
5ª	Refatorado	"*Legibilidade o um está... Se faz a mesma coisa, cara, está muito bom. Para manutenabilidade eu escolheria o um. Complexidade, o dois está mais complexo.*"

Fonte: Elaborado pelo autor (2025)

Tabela 18: Escolhas e justificativas do entrevistado E2 nas cinco comparações

Comparação	Código escolhido	Justificativa
1ª	Refatorado	"*Eu vejo todas essas características no código 2. [...] é menor, mais fácil de ler, melhor indentado. No primeiro eu me perco em vários comandos, acho mais difícil de ler e manter.*"
2ª	Refatorado	"*Dessa vez eu vou no código 1 [...] ele resumiu o código, evitou criar funções por completo, fez tudo em uma linha. Códigos menores eu acho mais fácil de ler.*"
3ª	Refatorado	"*O primeiro usa muita estrutura condicional, fica complexo. O segundo está mais legível e fácil de manter. [...] A legibilidade é melhor no 2.*"
4ª	Refatorado	"*Dessa vez o primeiro é um pouco mais legível que o segundo. [...] Para teste também saiu melhor. Para manter, seria um pouco mais fácil do que o segundo.*"
5ª	Refatorado	"*O primeiro eu acho mais legível, menos complexo para manter. Então seria o número 1, considero melhor.*"

Fonte: Elaborado pelo autor (2025)

Tabela 19: Escolhas e justificativas do entrevistado E3 nas cinco comparações

Comparação	Código escolhido	Justificativa
1ª	Refatorado	"*De cara, pra mim, o da direita é mais fácil de dar manutenção; fica mais legível do que deixar só string.*"
2ª	Refatorado	"*O código 2 divide os atributos por classe; não preciso ficar vendo um monte de variável solta no início.*"
3ª	Refatorado	"*Pra mim o código 2 é muito melhor; soube dividir em classes e agrupar responsabilidades, ficando mais legível.*"
4ª	Refatorado	"*Aqui o código 1 está melhor: em vez de várias linhas no* `try`*, ele separa em função, o que facilita a manutenção. O 2 tem uma função gigante que faz mais de uma coisa.*"
5ª	Refatorado	"*No 2, as variáveis foram separadas em objetos; fica mais manutenível, menos acoplado, legível e até mais performático.*"

Fonte: Elaborado pelo autor (2025)

Tabela 20: Escolhas e justificativas do entrevistado E4 nas cinco comparações

Comparação	Código escolhido	Justificativa
1ª	Refatorado	"*O código 2 está melhor por ser mais simples. No código 1 tem muita coisa não padronizada, textos dentro do código e variáveis soltas. O código 2 já usa manager para cuidar disso.*"
2ª	Refatorado	"*Claramente o primeiro; o segundo é bem inicial, um esboço. O primeiro tem boas normas aplicadas.*"
3ª	Refatorado	"*Para manutenção, o código 1 é mais funcional; consigo mexer de forma mais específica. No 2, a validação não está tão direta e é fácil quebrar algo.*"
4ª	Refatorado	"*O código 2 está mais dividido, bem modularizado. No código 1 encontrei apenas uma classe. Modularidade é essencial para manutenção no longo prazo.*"
5ª	Refatorado	"*Com certeza o código 1; é mais fácil de manter, melhora o ciclo de vida do software. O código 2 tem muitas variáveis locais e pouco aproveitamento externo.*"

Fonte: Elaborado pelo autor (2025)

Tabela 21: Escolhas e justificativas do entrevistado E5 nas cinco comparações

Comparação	Código escolhido	Justificativa
1ª	Refatorado	"*Não tem nem como comparar, o código 1 está enxuto, perfeito, bem resumido. O código 2 está enorme e pouco legível.*"
2ª	Refatorado	"*O código 1 parece mais organizado, o acoplamento das classes está bem alinhado. Ele abrange todas as qualidades: legibilidade, testabilidade, modularidade, manutenabilidade.*"
3ª	Original	"*Eu diria que para o iniciante, talvez o código 2 fosse mais fácil dele ir seguindo, né? Assim, para aprendizado. Porque você vai acompanhando cada passo, está tudo ali, não precisa ficar procurando em outro lugar.*"
4ª	Refatorado	"*O código 1 está bem legível, modularizado, compacto, perfeito. O 2 tem muitos ifs e elses, prejudicando a legibilidade.*"
5ª	Refatorado	"*Com certeza o código 1. O 2 é muito extenso, demanda bastante tempo para compreender e manter.*"

Fonte: Elaborado pelo autor (2025)

Tabela 22: Escolhas e justificativas do entrevistado E6 nas cinco comparações

Comparação	Código escolhido	Justificativa
1ª	Refatorado	"*O código 2 está bem mais modular, dividido e legível. O código 1 assume muita responsabilidade, métodos gigantes e mais difícil de dar manutenção.*"
2ª	Refatorado	"*O código 2 está bem mais modularizado. O código 1 tem excesso de responsabilidade. O 2 facilita testes e manutenção.*"
3ª	Refatorado	"*O código 2 é mais legível, simples e fácil de entender. O 1 é mais complexo e difícil de entender o que a função faz.*"
4ª	Refatorado	"*O código 2 tem complexidade ciclomática elevada, muitos ifs aninhados. O código 1 usa switch case, mais legível e facilita manutenção.*"
5ª	Refatorado	"*O código 1, com certeza. Melhor legibilidade, modularização e manutenção. O código 2 é basicamente uma classe única fazendo várias funções, atrapalha a qualidade.*"

Fonte: Elaborado pelo autor (2025)

Tabela 23: Escolhas e justificativas do entrevistado E7 nas cinco comparações

Comparação	Código escolhido	Justificativa
1ª	Refatorado	"*Com certeza, da direita. Está bem mais fácil de entender. O código 1 está extenso, concentrando responsabilidades. O 2 está mais modularizado, conciso e sem lógica muito complexa.*"
2ª	Refatorado	"*O código 1 está mais conciso, métodos claros, nomes e variáveis induzem ao entendimento. O código 2 tem uma classe gigante e difícil de entender, com várias variáveis repetitivas.*"
3ª	Refatorado	"*O código 2 está mais conciso e modularizado, mais fácil de entender. O código 1 tem mais de 700 linhas, diferentes funcionalidades, deveria ser refatorado.*"
4ª	Refatorado	"*O código 2 está mais conciso, agrupado por responsabilidades. Mais fácil de entender, classes bem divididas. O código 1 tem muitas condicionais desnecessárias e repetidas.*"
5ª	Refatorado	"*O código 1 é mais conciso, segue padrão de agrupamento de responsabilidades, consigo entender rápido. O código 2 tem loops aninhados, condicionais complexas, alta complexidade ciclomática e duplicação.*"

Fonte: Elaborado pelo autor (2025)

APÊNDICE D – *ROTEIRO DE ENTREVISTA*

Neste apêndice é apresentado o roteiro seguido nas entrevistas, divido em quatro blocos principais: coletado perfil do entrevistado, comparativa de códigos, ferramentas e reflexões sobre qualidade de código e encerramento.

1. **Introdução - Abertura**
    a) Agradecimento ao entrevistado pela participação.
    b) **Explicação sobre o objetivo da entrevista:**
        i. Analisar percepções sobre qualidade de código em diferentes contextos.
        ii. Comparar 5 códigos retirados de projetos reais (*QualitasCorpus*).
        iii. Coletar *insights* sobre boas práticas e manutenção de código.
        iv. Informação sobre a gravação da entrevista e confidencialidade dos dados.
    c) Perguntar se há alguma dúvida antes de começarmos.
2. **Perfil do Entrevistado**
    a) Qual é o seu nível de experiência em desenvolvimento de software? (Júnior, Pleno, Sênior)
    b) Há quanto tempo você trabalha com Programação?
    c) Que tipo de software você normalmente desenvolve?
    d) Ferramentas que usa ou usou no trabalho?
    e) Quais critérios você geralmente usa para avaliar a qualidade de um código?
3. **Análise Comparativa dos Códigos - Para cada par de códigos:**
    a) Quais são os atributos de qualidade de código você encontrou nos pares de código (ex.: legibilidade, testabilidade, modularidade, manutenibilidade, extensibilidade)?
    b) Qual código você considera que usou boas práticas para manter a qualidade do código? E como influencia durante o ciclo de vida do software?
    c) Quais sinais ou indicadores você observa para identificar um código com baixa qualidade?
    d) Qual dos dois códigos você considera de melhor qualidade? Por quê?
    e) Qual código você consideraria mais fácil de manter a longo prazo? Justifique sua resposta.
    f) A legibilidade do código foi impactada positivamente ou negativamente versão 1 ou versão 2? Como?
4. **Ferramentas e Reflexão sobre Qualidade de Código**

a) Você já utilizou ferramentas automatizadas de análise de qualidade de código? Se sim, quais e o que achou delas?
b) Você considera útil a aplicação de métricas (ex.: complexidade ciclomática, acoplamento, cobertura de testes) para avaliar a qualidade? Por quê?
c) No seu fluxo de trabalho, há revisões de código (*code review*)? O que você observa durante essas revisões?
d) Já trabalhou em projetos onde a refatoração foi evitada por receio de introduzir *bugs* ou complicar a manutenção? Poderia compartilhar sua experiência?
e) Como a qualidade do código influencia a produtividade e moral do time?
f) Você já trabalhou em projetos em que a baixa qualidade de código gerou dívidas técnicas difíceis de resolver? Pode contar como foi?

5. **Encerramento**
   a) O que você achou do exercício de análise comparativa? Foi útil para sua percepção sobre qualidade de código?
   b) Para sua surpresa, o código refatorado foi gerado por um modelo de linguagem treinado para otimizar código. Você mudaria suas escolhas após essa revelação? Qual é sua opinião sobre o uso de inteligência artificial no quesito qualidade de código?
   c) Você considera que as sugestões geradas por uma IA podem substituir parcialmente o trabalho de um desenvolvedor na melhoria do código?
   d) **Finalização:**
      i. Agradecimento pela participação e tempo dedicado.
      ii. Reafirmação sobre a confidencialidade das respostas.
      iii. Pergunta se há alguma consideração final que gostaria de adicionar.