Universidade Estadual de Londrina

BRUNO HENRIQUE STRIK

UMA ABORDAGEM BASEADA EM INTELIGÊNCIA ARTIFICIAL PARA IDENTIFICAÇÃO E CLASSIFICAÇÃO AUTOMATIZADA DE PROBLEMAS NA APRENDIZAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETOS POR MEIO DA ANÁLISE DE CÓDIGO-FONTE

LONDRINA
2025

BRUNO HENRIQUE STRIK

UMA ABORDAGEM BASEADA EM INTELIGÊNCIA ARTIFICIAL PARA IDENTIFICAÇÃO E CLASSIFICAÇÃO AUTOMATIZADA DE PROBLEMAS NA APRENDIZAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETOS POR MEIO DA ANÁLISE DE CÓDIGO-FONTE

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. André Luís Andrade Menolli

LONDRINA

2025

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema de Bibliotecas da UEL

S917u Strik, Bruno Henrique.
UMA ABORDAGEM BASEADA EM INTELIGÊNCIA ARTIFICIAL PARA IDENTIFICAÇÃO E CLASSIFICAÇÃO AUTOMATIZADA DE PROBLEMAS NA APRENDIZAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETOS POR MEIO DA ANÁLISE DE CÓDIGO-FONTE / Bruno Henrique Strik. - Londrina, 2025.
102 f.

Orientador: André Luís Andrade Menolli.
Dissertação (Mestrado em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2025.
Inclui bibliografia.

1. Qualidade de código - Tese. 2. Informática na educação - Tese. 3. Inteligência artificial - Tese. I. Menolli, André Luís Andrade. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 519

BRUNO HENRIQUE STRIK

UMA ABORDAGEM BASEADA EM INTELIGÊNCIA ARTIFICIAL PARA IDENTIFICAÇÃO E CLASSIFICAÇÃO AUTOMATIZADA DE PROBLEMAS NA APRENDIZAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETOS POR MEIO DA ANÁLISE DE CÓDIGO-FONTE

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

Orientador: Prof. Dr. André Luís Andrade Menolli
Universidade Estadual de Londrina

Prof. Dr. Bruno Bogaz Zarpelão
Universidade Estadual de Londrina

Prof. Dr. João Coelho Neto
Universidade Estadual do Norte do Paraná

Londrina, 26 de março de 2025.

AGRADECIMENTOS

Aos meus pais, Sérgio e Cristina, pelos ensinamentos para a vida e os estudos.

À minha esposa Bruna, pela compreensão e apoio constantes.

Ao professor André Menolli, pela orientação e parceria nesta caminhada.

Aos colegas e amigos do IFPR, pelas leituras, discussões e conselhos valiosos.

À todas as pessoas que fizeram parte da minha trajetória profissional e acadêmica.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

STRIK, B. H.. **Uma abordagem baseada em Inteligência Artificial para Identificação e Classificação Automatizada de Problemas na Aprendizagem de Programação Orientada a Objetos por Meio da Análise de Código-Fonte**. 2025. 101f. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2025.

RESUMO

O acesso público a ferramentas de inteligência artificial generativa tem revolucionado a *práxis* de diversas atividades humanas, incluindo o campo educacional. Docentes e estudantes manifestaram percepções variadas, embora haja um consenso quanto ao potencial dessas ferramentas em apoiar tanto os processos produtivos da computação quanto os de ensino e aprendizagem. Apesar de sua inovação, a tecnologia ainda carece de abordagens eficazes para aplicação no contexto educacional, a fim de transcender seu uso como mera geradora de soluções imediatas e garantir suporte adequado ao desenvolvimento intelectual e técnico dos estudantes. O objetivo desta dissertação é propor uma abordagem baseada em inteligência artificial para detectar indícios de problemas na aprendizagem de programação orientada a objetos observáveis diretamente no código-fonte produzido por estudantes e gerar *feedbacks* educacionais para os problemas identificados. Para tanto, foram desenvolvidas heurísticas específicas para a caracterização, identificação, classificação e tratamento desses problemas, além de uma ferramenta capaz de aplicá-las. Com base em um protótipo desenvolvido e testado por professores de Computação de diversas Instituições de Ensino Superior, averiguou-se que a abordagem é viável e apresenta potencial para fornecer retornos personalizados, contribuindo de forma eficaz para o processo de aprendizagem de programação orientada a objetos.

Palavras-chave: Inteligência artificial generativa. Ensino de programação orientada a objetos. Inteligência artificial na educação.

STRIK, B. H.. **An Artificial Intelligence-Based Approach for Automated Identification and Classification of Problems in Object-Oriented Programming Learning Through Source Code Analysis**. 2025. 101p. Master's Thesis (Master in Science in Computer Science) – State University of Londrina, Londrina, 2025.

ABSTRACT

Public access to generative artificial intelligence tools has revolutionized the praxis of various human activities, including the educational field. Teachers and students have expressed varied perceptions, although there is a consensus on the potential of these tools to support both productive computing processes and teaching and learning. Despite their innovation, the technology still lacks effective approaches for application in the educational context to go beyond merely generating immediate solutions and to provide adequate support for students' intellectual and technical development.

The objective of this dissertation is to propose an artificial intelligence-based approach to detect signs of problems in learning object-oriented programming, which are observable directly in the source code produced by students, and to generate educational feedback for the identified issues. To achieve this, specific heuristics were developed for the characterization, identification, classification, and treatment of these problems, as well as a tool capable of applying them.

Based on a prototype developed and tested by Computer Science professors from various Higher Education Institutions, it was found that the approach is feasible and has the potential to provide personalized feedback, effectively contributing to the learning process of object-oriented programming.

Keywords: GPT. Object-oriented programming education. Artificial intelligence in education.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AIEd	Artificial Intelligence in Education (Inteligência Artificial na Educação)
API	Application Programming Interface (Interface de Programação de Aplicação)
CC	Ciência da Computação
CS	*Code Smells*
GenAI	Inteligência Artificial Generativa
GPT	Generative Pre-trained Transformer (Transformador Pré-treinado Generativo)
IA	Inteligência Artificial
LLM	Large Language Model (Modelo de linguagem de grande escala)
OO	Orientação a Objetos
POO	Programação Orientada a Objetos
REST	Representational State Transfer (Transferência de Estado Representacional)
RNN	Recurrent Neural Networks (Redes Neurais Recursivas)
VSCode	Visual Studio Code

SUMÁRIO

1 INTRODUÇÃO

A aprendizagem de programação de computadores é um processo desafiador [3, 4]. Apesar de as tecnologias recentes terem possibilitado novas abordagens de ensino com linguagens simplificadas e até mesmo lúdicas, a compreensão de conceitos complexos e abstratos ainda apresenta uma curva de aprendizado íngreme [5]. Diversos estudos têm se dedicado a entender o processo de aprendizagem de programação, identificar suas dificuldades intrínsecas e propor abordagens alternativas para mitigá-las [6]. Quando a temática aborda a Programação Orientada a Objetos (POO), o desafio é ampliado, pois, além da aprendizagem das peculiaridades da linguagem, é necessária atenção a todo o seu paradigma teórico [7].

Pesquisadores que analisam as dificuldades no aprendizado de programação orientada a objetos apontam que a estrutura curricular dos cursos muitas vezes não favorece a qualidade do código produzido [5, 8]. As disciplinas iniciais tendem a valorizar principalmente a realização de objetivos de aprendizagem relacionados à sintaxe e à funcionalidade, frequentemente em detrimento de requisitos de estrutura e estilo. Quando os estudantes avançam para etapas superiores e precisam atingir objetivos de qualidade [9], frequentemente carregam consigo concepções ineficazes, necessitando de um processo de "demolição e reconstrução"[10].

Como observado por Keuning; Jeuring & Heeren [9], qualidade do código é um termo sem um significado claro e com várias interpretações. Neste trabalho, caracterizamos a qualidade do código como um aspecto que surge após a escrita do código, lidando com a análise, reflexão e melhoria das características estáticas do programa. Problemas nesses aspectos são frequentemente denominados como *code smells*. Segundo Fowler [11], *code smells* podem indicar um problema no design de código funcionalmente correto, afetando atributos de qualidade do software. Exemplos incluem duplicação, código morto, código excessivamente complexo e código com baixa coesão e alto acoplamento.

A produção de código orientado a objetos de baixa qualidade não está relacionada somente à não aplicação de boas práticas de programação. Dificuldades de compreensão, típicas de qualquer etapa de aprendizagem, também estão presentes. *Antipatterns* — soluções recorrentes e precárias que violam princípios de qualidade e boas práticas— *code smells* e dívidas técnicas — soluções subótimas que acumulam custos futuros em manutenção — são alguns sintomas comuns de código de baixa qualidade, mas também indicam uma aprendizagem pouco efetiva [12]. No contexto da educação em programação orientada a objetos, a correta identificação desses problemas é um processo diagnóstico necessário para a mediação eficaz da aprendizagem [13].

Ao consultar a literatura, identificamos poucos trabalhos que classificam os problemas na aprendizagem de programação orientada a objetos. Gutiérrez; Guerrero & López-Ospina [12] organizam esses problemas em 14 categorias, baseando-se na revisão de 56 estudos. Outros trabalhos que investigam os problemas na aprendizagem da programação orientada a objetos, como os de Thomasson; Ratcliffe & Thomas [14], Biju [15], Holland; Griffiths & Woodman [16] e Or-Bach & Lavy [17], classificam as dificuldades dentro dos conceitos fundamentais da orientação a objetos: abstração, encapsulamento, herança e polimorfismo [18].

Identifica-se, portanto, que a maior parte dos estudos opta por refletir os indicadores sintomáticos em violações de princípios do paradigma. Entretanto, essa forma modularizada e simplificada de classificação pode, ao mesmo tempo, ocultar problemas estruturais de aprendizagem. A classificação sintomática, sem uma reflexão sobre suas causas, limita o entendimento das dificuldades de aprendizagem dos estudantes. Como este trabalho busca mapear as dificuldades de compreensão dos conceitos da programação orientada a objetos, é fundamental ir além e refletir sobre os equívocos ou insuficiências de compreensão que determinam esses indicadores sintomáticos.

Como ponto de partida, é preciso definir que a análise criteriosa do código-fonte vai além da simples mimetização do processo de compilação. Um código que atinge seus objetivos, ainda que de forma pobre e repleto de *code smells*, é considerado eficaz pelo compilador. É comum que, em disciplinas introdutórias, esse tipo de problema seja ignorado para evitar que a complexidade excessiva prejudique a aprendizagem de estudantes ainda pouco experientes. Quando se trata de programação orientada a objetos, é necessário não apenas analisar se o código executa corretamente, mas também verificar se ele segue os princípios do paradigma.

Neste trabalho, é aplicada a IA generativa para automatização do processo de identificação de dificuldades de aprendizagem a partir do código-fonte. Dada a complexidade da análise do código, do mapeamento de problemas e da reflexão sobre indicadores de dificuldades, a IA generativa poderia contribuir significativamente [19]. Suas capacidades de compreensão e geração de texto, compreensão de código em diversas linguagens, consideração do contexto, semântica e intenção são adequadas para a implementação da proposta.

Inspirado nas experiências positivas do potencial educativo da IA relatados por Baidoo-Anu & Ansah [20], Prather et al. [21] e também por Silva et al. [22], este trabalho propõe uma abordagem baseada em inteligência artificial para detectar indícios de problemas na aprendizagem de programação orientada a objetos observáveis diretamente no código-fonte produzido por estudantes e gerar feedbacks educacionais para os problemas identificados . De forma semelhante, os trabalhos de Rajala et al. [23], Hellas et al. [24] e Sheese et al. [25] trazem impressões e exploram a aplicação de IA generativa no processo

de ensino-aprendizagem de programação. Os trabalhos mapeados avaliam principalmente a capacidade da IA generativa como ferramenta consultiva e geradora de código, explorando as implicações e o impacto de sua adoção no processo educacional. Indo além, este trabalho explora as capacidades da IA como ferramenta auxiliar no processo educativo, com atuação delimitada de um Modelo de Linguagem de Grande Escala (Large Language Model, LLM) especialmente ajustado.

Destaca-se ainda a novidade deste trabalho pois, conforme o mapeamento de Strik; Menolli & Brancher [26], ainda não há na literatura trabalhos que focam em identificar problemas de aprendizagem de programação a partir da identificação e mapeamento de problemas de qualidade de código.

1.1 Objetivos

O objetivo geral deste trabalho é propor uma abordagem baseada em inteligência artificial para detectar indícios de problemas na aprendizagem de programação orientada a objetos observáveis diretamente no código-fonte produzido por estudantes e gerar feedbacks educacionais para os problemas identificados. Os objetivos específicos estabelecidos são:

- Caracterizar, com base no trabalho de Gutiérrez; Guerrero & López-Ospina [12] os indicadores de dificuldades de aprendizagem de programação orientada a objetos a partir de problemas de qualidade de código;
- Mapear e classificar, com base na literatura e em evidências empíricas, indicadores e categorias de dificuldades de aprendizagem, culminando em um mapeamento pioneiro que classifica e caracteriza dificuldades de aprendizagem e seus indícios observáveis em código;
- Configurar um modelo de inteligência artificial generativa para analisar código-fonte e identificar problemas de qualidade;
- Configurar um segundo modelo de inteligência artificial generativa para produzir mensagens de *feedback* com orientações educacionais personalizadas de acordo com as dificuldades de aprendizagem mapeadas;
- Desenvolver um protótipo de ferramenta que permita ao estudante submeter seu código-fonte para análise, e receber *feedback* educacional personalizado que contribua em seu processo de aprendizagem de programação orientada a objetos;
- Avaliar a heurística e a ferramenta via análise exploratória de desempenho, avaliação com especialistas por meio de questionário e avaliação da usabilidade.

1.2 Estrutura de pesquisa

Esta seção descreve a estrutura da pesquisa, seguida pelo detalhamento metodológico aplicado em cada fase. A Figura 1 ilustra as etapas sequenciais e as fases que compõem este trabalho. Este estudo é caracterizado como uma pesquisa aplicada, objetivando tanto a criação de novos conhecimentos quanto sua aplicação prática.

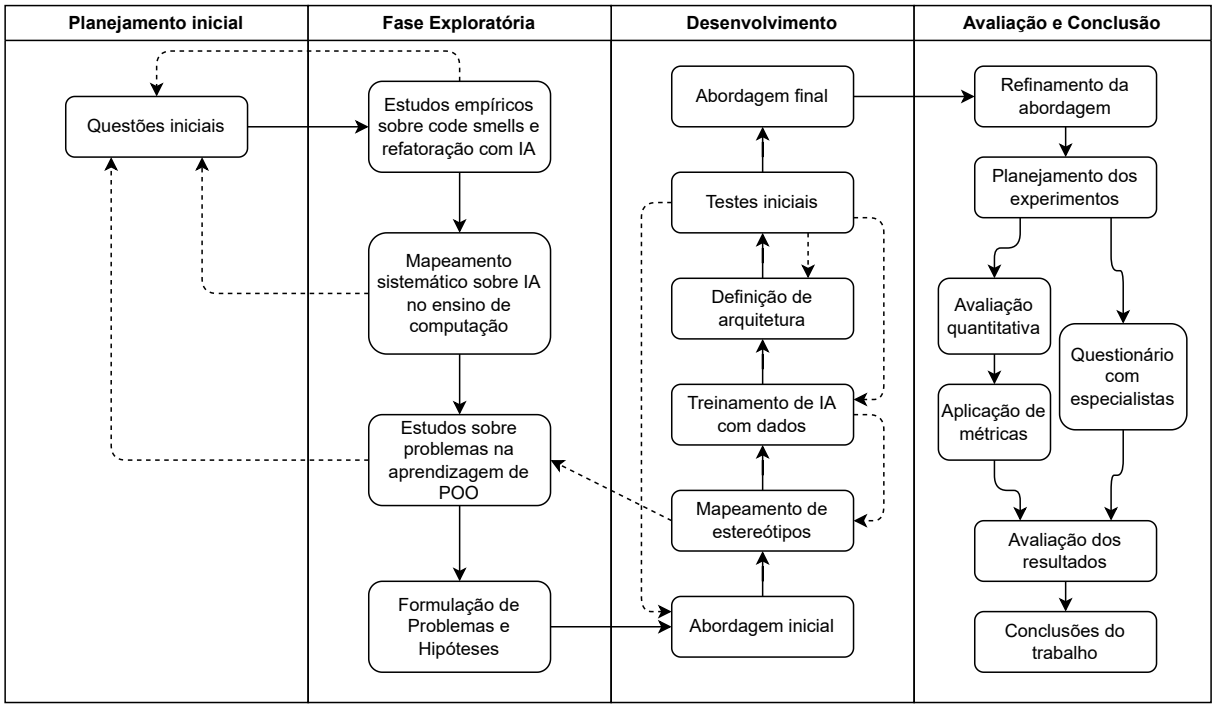


Figura 1 – Estrutura de pesquisa. Fonte: autor.

1.2.1 Planejamento inicial

Esta fase, composta exclusivamente pela etapa de Questões Iniciais, indica os processos de aquisição inicial de dados e conhecimentos necessários para a definição do escopo e do caminho da pesquisa. O trabalho de pesquisa foi orientado pela formulação de questões iniciais, utilizadas para delimitação do escopo: "Como as ferramentas de IA generativa são utilizadas no contexto do ensino de computação?", "Quais são as dificuldades presentes no processo de ensino e aprendizagem de programação orientada a objetos?"e "Como a IA generativa pode ser utilizada para auxiliar no diagnóstico de problemas de aprendizagem de programação orientada a objetos?".

1.2.2 Fase exploratória

A fase exploratória engloba grande parte da pesquisa externa e coleta de informações. Nela, acontecem as etapas de mapeamento, estudo e coleta de informações para a definição dos objetivos.

Nesta etapa, foi realizado um mapeamento sistemático [26] que identificou e classificou diversos trabalhos que investigaram a aplicação de inteligência artificial generativa no ensino de computação. Foram identificadas preocupações com plágio, percepção de aprendizagem e capacidade da IA, mas também que a tecnologia apresenta um grande potencial latente a ser explorado, indicando oportunidades para estudos de sua aplicação.

Além do mapeamento, foram realizados estudos empíricos que avaliaram a capacidade da IA generativa de realizar a identificação de *code smells* e refatoração, nos quais a IA obteve um desempenho positivo na identificação de problemas. Contudo, sua capacidade de realizar refatoração automaticamente é limitada, principalmente devido aos limites da interface.

Estudos sobre problemas na aprendizagem de programação também foram realizados, permitindo a caracterização das dificuldades encontradas no processo educativo, apresentadas na seção 2.4.

1.2.3 Desenvolvimento

A fase de desenvolvimento agrega os procedimentos realizados referentes ao desenvolvimento da proposta de abordagem, ao mapeamento dos indicativos de problemas de aprendizagem no contexto do ensino de programação orientada a objetos, ao treinamento da ferramenta de inteligência artificial com os indicativos mapeados e à definição da arquitetura. Em seguida, alguns testes iniciais foram realizados, acompanhados dos eventuais ajustes necessários, culminando no desenvolvimento da abordagem final.

1.2.4 Avaliação e conclusão

A fase de avaliação e conclusão consiste na execução de uma série de experimentos para validação da abordagem definida e posterior discussão dos resultados. Experimentos empíricos avaliaram diversos aspectos da ferramenta de inteligência artificial, contribuindo com valiosas informações sobre sua aplicabilidade como ferramenta de apoio ao ensino.

1.3 Organização do trabalho

Esta seção tem como objetivo apresentar a estrutura e a disposição deste trabalho. Desta forma, busca-se oferecer uma compreensão clara do fluxo de ideias e do encadeamento lógico das fases que o constituem.

- Aporte Teórico
  - IA generativa no ensino de computação: Nesta seção é apresentado de forma breve o mapeamento sistemático realizado sobre a aplicação de inteligência

artificial generativa no ensino de ciência da computação. Também são contextualizados os conceitos, técnicas e principais ferramentas atualmente aplicadas e disponíveis para o emprego da inteligência artificial nos processos de ensino-aprendizagem de Ciência da Computação.

  - Qualidade de código no ensino de programação: Nesta seção, são apresentadas as principais formas de avaliação da qualidade de código no ensino da programação, estabelecendo o conceito de qualidade de código considerado neste trabalho.
  - Indicadores de problemas no código: Nesta seção, apresentamos diversos conceitos sobre características observáveis no código-fonte que são consideradas indicadores de problemas de qualidade.
  - Aprendizagem de programação: Nesta seção, é contextualizado o processo de aprendizagem de programação comumente observado em cursos da área de computação [27], com observações especiais para as dificuldades identificadas na aprendizagem, métodos e abordagens pedagógicas baseadas na literatura e em evidências empíricas oriundas da experiência pedagógica do autor.

- Abordagem Proposta

  - Mapeamento de dificuldades: Nesta seção é apresentado o mapeamento das dificuldades de aprendizagem e dos problemas de qualidade de código que os indicam.
  - Heurística de análise de código: Nesta seção está detalhada uma proposta de heurística que estabelece uma série de procedimentos para análise de código, identificação de problemas de qualidade, reflexão dos problemas de aprendizagem relacionados e produção de *feedback* educacional.

- Ferramenta

  - Arquitetura da ferramenta: Nesta seção é apresentada a arquitetura de uma ferramenta que implementa a heurística de análise de código apresentada na seção anterior.
  - Modelo de inteligência artificial: Nesta seção é apresentado o LLM empregado na ferramenta e os procedimentos de sua utilização, com uma rápida introdução sobre o funcionamento de sua arquitetura interna.

- Avaliação

  - Capacidade de detecção da IA: Nesta seção, são discutidos os resultados de experimentos realizados para avaliar a capacidade do LLM empregado na ferramenta de detectar diversos problemas de qualidade de código em amostras variadas.

– Aspectos subjetivos: Aqui são apresentados resultados de uma pesquisa qualitativa realizada por especialistas no ensino de computação que utilizaram a ferramenta proposta, discutindo as impressões e observações dos participantes.
– Ameaças à validade da proposta: Nesta seção, são discutidas possíveis ameaças que podem impactar os resultados subótimos da proposta e da ferramenta apresentadas.

2 FUNDAMENTAÇÃO TEÓRICA

O ensino de programação orientada a objetos tem sido um desafio constante para docentes e estudantes. A complexidade dos conceitos envolvidos, como encapsulamento, herança, polimorfismo e abstração, muitas vezes resulta em dificuldades de aprendizagem que se refletem na qualidade do código produzido pelos alunos.

Este capítulo tem como objetivo apresentar uma revisão abrangente dos trabalhos relacionados mais relevantes relacionados à aplicação da inteligência artificial generativa no ensino de ciência da computação, discutir os conceitos de qualidade de código no contexto educacional, os indicadores de problemas no código-fonte, e as dificuldades comuns enfrentadas pelos estudantes no aprendizado de POO.

2.1 IA generativa no ensino de computação

O campo da inteligência artificial aplicada à educação não é novo. Já em 1970, um estudo de Jaime R. Carbonell introduziu um protótipo que era executado em um PDP-10 e interagia dialogicamente com os estudantes, auxiliando-os em seus estudos [28]. Embora o protótipo tivesse limitações devido à tecnologia de sua época, ele já destacava o potencial das ferramentas de aprendizado assistidas por IA. Cinquenta anos depois, o uso extensivo da IA na educação ainda não havia se tornado realidade até o lançamento do ChatGPT [29], que foi disponibilizado ao público em novembro de 2022 [30]. Esse evento marcou a primeira vez que uma ferramenta de IA baseada na arquitetura GPT (Transformador Pré-Treinado Generativo) foi acessível gratuitamente ao público em geral. As características impressionantes dessa ferramenta, junto com sua capacidade de interações em linguagem natural semelhantes às humanas, levaram à sua rápida adoção pelo público [31].

LLMs são um tipo de modelo de IA desenvolvido para entender e gerar linguagem natural [32]. Eles são treinados com enormes quantidades de dados textuais e baseados em arquiteturas de Redes Neurais Profundas. O GPT [33] é uma arquitetura específica de LLMs desenvolvida pela OpenAI, que serve de base para ferramentas como o Microsoft Copilot[1], o Codex (hoje Github Copilot[2]) e o mais amplamente utilizado e conhecido ChatGPT[3].

Embora o impacto total do ChatGPT ainda seja desconhecido, alguns estudos [34, 35, 36, 37] indicam que sua introdução pode ser tão transformadora quanto a Internet. Assim como a Internet mudou fundamentalmente a maneira como acessamos e

[1] https://www.microsoft.com/pt-br/microsoft-copilot
[2] https://github.com/features/copilot
[3] https://chatgpt.com

compartilhamos informações, a IA generativa como o ChatGPT pode mudar a forma como trabalhamos, aprendemos e interagimos com a tecnologia, algo que já está sendo observado na educação [38].

O uso educacional do ChatGPT e de outras ferramentas de IA generativa não foi inicialmente planejado; seu uso emergiu de forma exploratória por estudantes e educadores [39]. Inicialmente despercebida, essa utilização logo chamou a atenção e levantou preocupações entre muitos educadores e pesquisadores sobre sua influência nos processos de aprendizagem [40]. O surgimento e a popularização da IA generativa foram recebidos com reações diversas. Inicialmente, preocupações sobre seu uso levaram a proibições ou restrições [41], não apenas dentro da educação [42]. Independentemente da percepção de diferentes partes interessadas, uma revolução está acontecendo, e seu impacto não pode ser ignorado. Reformas educacionais significativas podem ser necessárias, ou ao menos mudanças profundas devem ser consideradas, pois combater ou ignorar essa tecnologia não parece ser uma abordagem produtiva. Diante dos novos desafios trazidos pelas ferramentas de IA generativa, é necessário compreender essas tecnologias e considerar os papéis que elas podem desempenhar no ensino e aprendizado da ciência da Computação.

Para sintetizar os trabalhos que exploram empiricamente o uso de ferramentas de IA generativa baseadas em redes neurais com arquitetura transformer no contexto educacional, foi conduzido um mapeamento sistemático, que focou nos estudos publicados nos primeiros 16 meses após o lançamento do ChatGPT.

A pesquisa inicial resultou na coleta de 5825 estudos provenientes das principais bases de dados acadêmicas: ACM, IEEE Xplore, Scopus, Springer Link e Web of Science. Após a aplicação de procedimentos criteriosos, incluindo a remoção de duplicatas, a utilização de critérios de inclusão e exclusão, além do método de *backward snowballing*, foram selecionados 31 artigos. Buscou-se então investigar em quais contextos e de que maneira as ferramentas de IA são utilizadas, os temas que as empregam e as percepções e preocupações gerais de estudantes e instrutores em relação à sua aplicabilidade e impacto.

Os mapeamento sistemático *GPT AI in Computer Science Education: A Systematic Mapping Study* [26] foi apresentado no XXXV Simpósio Brasileiro de Informática na Educação, e seu artigo detalha em profundidade os critérios e método empregados.

O mapeamento apontou que a IA generativa tem sido utilizada principalmente para geração de código, depuração, explicação de código e clarificação conceitual em tarefas de programação. As ferramentas de IA generativa foram adotadas em várias disciplinas de Ciência da Computação, sendo as relacionadas à programação as de maior prevalência nos estudos coletados.

Diversas impressões e preocupações foram identificadas nos estudos do mapeamento. As percepções positivas observadas nos 31 trabalhos mapeados são apresentadas

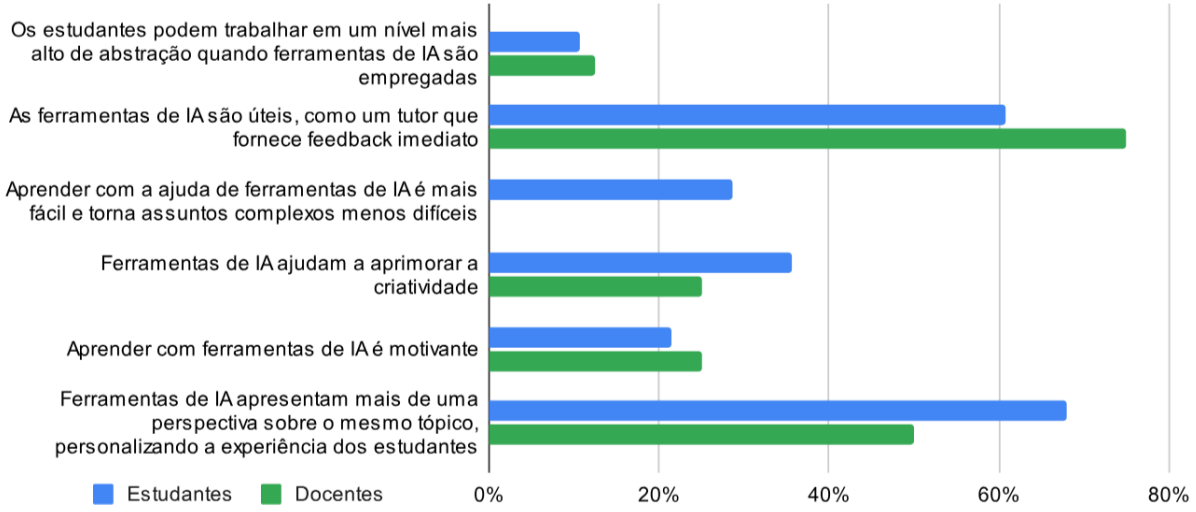


Figura 2 – Percepções positivas sobre o uso de IA generativa no ensino de computação. Fonte: autor.

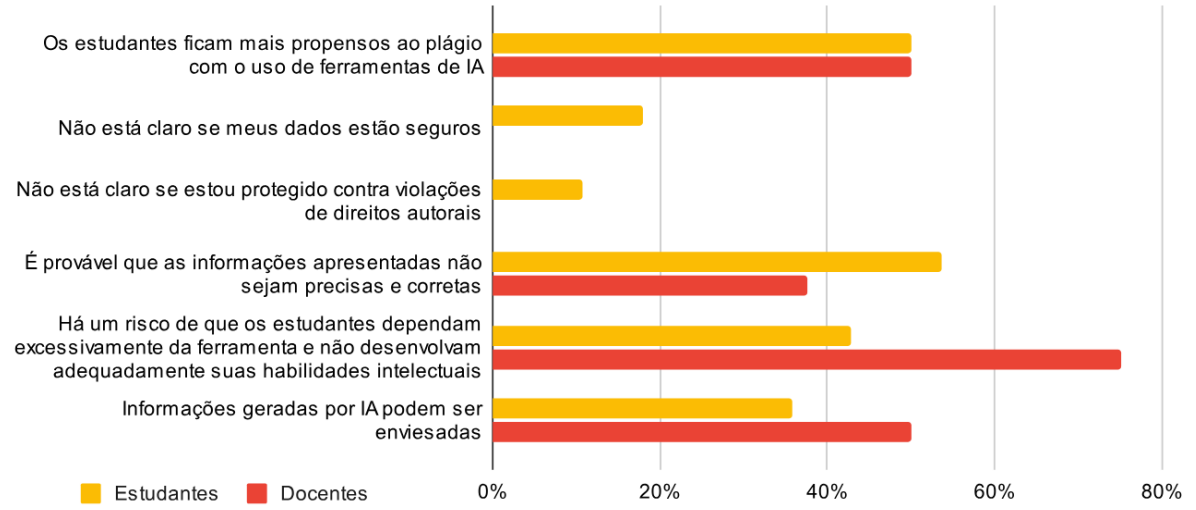


Figura 3 – Percepções negativas sobre o uso de IA generativa no ensino de computação. Fonte: autor.

na Figura 2, que quantifica seus principais apontamentos. Entre as observações sobre as contribuições positivas das ferramentas de IA generativa, 61% dos estudos que envolvem estudantes e 75% dos que envolvem docentes destacam que as ferramentas de IA auxiliam os estudantes em suas atividades acadêmicas de maneira semelhante a um tutor, fornecendo um retorno imediato. A capacidade das ferramentas de IA generativa de fornecer diferentes perspectivas e explicações alternativas sobre os temas também é destacada como um aspecto positivo em 68% dos estudos que envolvem estudantes e 50% dos que envolvem docentes, o que destaca o potencial da IA como ferramenta de personalização do processo educativo.

No entanto, a incerteza sobre a corretude das informações geradas é uma percepção

negativa comum observada nos estudos, juntamente com os possíveis vieses que podem afetar a precisão e exatidão das informações [26].

Alguns estudos expressaram preocupações sobre a influência pedagógica das ferramentas de IA no ensino de Ciência da Computação. A Figura 3 mostra as principais percepções negativas da aplicação empírica das ferramentas de IA. Além das questões de viés — apontadas em 36% dos estudos que envolvem estudantes e 50% dos que envolvem docentes — e precisão — apontados em 54% e 38% dos estudos, respectivamente. Os estudos apontam que os estudantes podem se tornar excessivamente dependentes das ferramentas de IA, negligenciando o desenvolvimento de suas próprias habilidades e intelecto, como apontado em 43% dos estudos que envolvem estudantes e 75% dos que envolvem docentes.

Além disso, a necessidade de uma ferramenta de IA generativa que esteja ciente de seu papel como auxiliar didático foi enfatizada em vários artigos [26]. A tendência natural dessas ferramentas de produzir diretamente as respostas pode não ser tão benéfica para o aprendizado, pois permite que os estudantes contornem o processo e se concentrem excessivamente na resposta. Embora essa preocupação se alinhe com questões de integridade acadêmica, também foi observado por alguns estudos que a ferramenta nem exacerba nem alivia essa dificuldade preexistente, pois os estudantes que simplesmente plagiam continuarão a fazê-lo, independentemente da incorporação das ferramentas de IA.

Vários estudos apontaram que a adoção da IA generativa deve ser cuidadosamente planejada para evitar a substituição das fundações conceituais e habilidades técnicas pela capacidade de usar a ferramenta de maneira eficaz, prejudicando assim o desenvolvimento intelectual dos estudantes. Os riscos da adoção inadequada da IA generativa em atividades de ensino precisam ser mitigados através de um planejamento metodológico profundo e cuidadoso.

A precisão das afirmações produzidas pela IA também é analisada, com um alto nível de respostas incompletas ou incorretas relatado [43, 44, 45, 24, 46]. Alguns estudos [47, 48] indicam que sugestões bem elaboradas podem mitigar esse problema, mas, mesmo assim, a IA generativa não deve ser considerada a única fonte de informação. Todas as informações fornecidas pela IA devem ser verificadas, pois a natureza estocástica da IA a torna vulnerável a vieses e informações incorretas.

O mapeamento também identificou estudos que abordaram a perspectiva dos instrutores nos primeiros meses após o lançamento público do ChatGPT e de outras ferramentas de IA, cujos apontamentos indicam preocupações mais intensas e até o desejo de que a tecnologia fosse banida dos processos educacionais. Estudos subsequentes não apresentaram esse desejo com tanta frequência, indicando que a familiarização de estudantes e instrutores com as ferramentas permitiu uma melhor compreensão das inseguranças

iniciais e uma visão mais positiva de seu potencial.

Apesar das várias preocupações apresentadas, a conclusão geral do mapeamento é que o uso da IA generativa é benéfico e positivo. Seus problemas podem ser mitigados refinando sua incorporação nas atividades de ensino e amadurecendo a relação entre estudantes e ferramentas. Os estudos identificaram sentimentos positivos em relação à confiança dos estudantes ao usar ferramentas de IA generativa, comparando sua contribuição à de um tutor sempre disponível, abordando o mesmo tema de várias maneiras conforme a conveniência de aprendizado do estudante.

2.2 Qualidade de código no ensino de programação

Qualidade de software e qualidade de código são conceitos entrelaçados e frequentemente confundidos. Consideramos que a qualidade de código é um conceito mais específico dentro da norma ISO/IEC 25000 [49], que trata da qualidade de software.

O conceito de qualidade de código [9] considera as características estáticas do programa observáveis diretamente a partir da análise de seu código-fonte. Desta forma, seu foco não contempla a análise dinâmica do programa, como, por exemplo, a performance de sua execução.

No contexto educacional, Stegeman; Barendsen & Smetsers [50] desenvolveram um modelo com seis critérios para avaliar a qualidade de código produzido pelos estudantes. Desenvolvido a partir da revisão literária de manuais de boas práticas e da experiência de professores de programação, os critérios abordam comentários (conteúdo e sintetização), formatação (consistência e expressividade), layout (afinidade, organização e presença de *dead code*), nomenclatura (consistência e significação), estrutura (abstração, duplicação, foco, modularização, uso de tipos, adequação dos métodos e fragmentação) e expressividade (fraseamento, clarificação e controle de fluxo).

Outros estudos também se dedicaram a estabelecer critérios para avaliar a qualidade do código no contexto educacional. Hamm et al. [51] não delimitam critérios específicos, mas estabelecem a abrangência de seus critérios para contemplar, além de aspectos funcionais, estrutura e documentação. Howatt [52] define que a avaliação dos programas desenvolvidos por estudantes deve contemplar a executabilidade, o atendimento às especificações do instrutor, a adesão ao estilo da linguagem utilizada, a presença de comentários efetivos, a legibilidade e o planejamento.

A partir da investigação dos trabalhos citados, adicionados aos de Becker [53] e Smith & Cordova [54], é possível verificar que a qualidade de código não só está presente e sistematizada no contexto educacional, mas também não é um assunto novo. Entretanto, não foram identificados trabalhos que parametrizam a avaliação da qualidade de código no ensino da programação orientada a objetos. Diversos parâmetros de qualidade

são também aplicáveis nesse contexto, porém os critérios identificados não contemplam aspectos importantes. A programação orientada a objetos vai além da programação introdutória e adiciona um conjunto de conceitos novos que, refletidos em código, requerem sua validação no processo educacional.

Ainda sobre o processo avaliativo, a classificação de qualidade e verificação de conformidade, como visto nos estudos apontados, são importantes no processo educacional. Entretanto, sua abordagem dentro de um processo apenas somativo limita a contribuição no processo educativo. Como o objetivo deste estudo é a proposta de uma abordagem não somativa, mas formativa, é necessário buscar as fragilidades que caracterizam a ocorrência de problemas na qualidade de código.

2.3 Indicadores de problemas no código

A análise do código-fonte de estudantes pode revelar uma série de indicadores que apontam para dificuldades na assimilação dos conceitos fundamentais da Programação Orientada a Objetos (POO). Alguns desses indicadores são refletidos na presença de *code smells* [11], bem como em violações de princípios de design reconhecidos, como a Lei de Demeter [55], o princípio *Tell, Don't Ask* [56], e os princípios SOLID [57]. A ocorrência dessas violações, além de representar um problema de qualidade de código, indica dificuldades na aprendizagem dos conceitos centrais da POO. Tais violações sugerem uma compreensão insuficiente do encapsulamento, abstração, responsabilidade e outros pilares da orientação a objetos.

A seguir, são detalhados os princípios cuja violação pode indicar dificuldades na aprendizagem de programação orientada a objetos:

SOLID

Os Princípios SOLID [57] são um conjunto de diretrizes usadas no desenvolvimento de software orientado a objetos. Introduzidos por Martin [57], esses princípios visam melhorar a qualidade, flexibilidade e sustentabilidade do código. SOLID é um acrônimo para cinco princípios fundamentais:

1. *Single Responsibility Principle* (SRP) - Princípio da Responsabilidade Única: Cada classe deve ter uma única responsabilidade, lidando apenas com uma funcionalidade ou tarefa específica do sistema. O SRP incentiva a coesão e facilita a manutenção, pois a mudança em um aspecto do sistema impacta apenas uma classe específica.

2. *Open/Closed Principle* (OCP) - Princípio do Aberto/Fechado: Entidades de software (classes, módulos, funções) devem estar abertas para extensão, mas fechadas para modificação. Isso significa que o comportamento de uma classe deve poder ser

estendido sem alterar seu código original. Esse princípio favorece o uso de interfaces e abstrações, permitindo a evolução do sistema sem comprometer sua estabilidade.

3. *Liskov Substitution Principle* (LSP) - Princípio da Substituição de Liskov: Objetos de uma classe derivada devem permitir sua substituição por objetos de sua classe base sem alterar a corretude do sistema. Esse princípio, introduzido por Liskov & Wing [58], assegura que uma subclasse preserve o comportamento esperado de sua superclasse, garantindo polimorfismo seguro e previsível.

4. *Interface Segregation Principle* (ISP) - Princípio da Segregação de Interfaces: Módulos de software não devem ser forçados a depender de interfaces que eles não utilizam. Este princípio incentiva a criação de interfaces menores e mais específicas, em vez de interfaces grandes e generalizadas. Ao dividir interfaces em elementos menores e focados, assegura-se que os clientes dependam apenas do que realmente precisam.

5. *Dependency Inversion Principle* (DIP) - Princípio da Inversão de Dependência: Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. Além disso, abstrações não devem depender de detalhes, mas os detalhes devem depender de abstrações. Esse princípio incentiva o uso de injeção de dependência e programação orientada a interfaces, promovendo o desacoplamento entre os componentes do sistema.

Tell, Don't Ask

O princípio *Tell, Don't Ask* [56] é uma diretriz de design de software que incentiva a abordagem de comunicação entre objetos baseada em delegação, ao invés de consulta de dados. Em vez de um objeto "perguntar"a outro por informações para depois tomar uma decisão com base nessas informações, o primeiro objeto deve simplesmente "dizer"ao segundo o que fazer, delegando a responsabilidade da ação ao próprio objeto. Essa abordagem reforça a ideia de encapsulamento, no qual cada objeto é responsável por suas próprias operações e estados, resultando em código mais legível, modular e menos propenso a erros.

Lei de Demeter

Também conhecida como Princípio do Menor Conhecimento [55], é uma diretriz de design de software que sugere que um objeto deve ter o menor conhecimento possível sobre os detalhes internos de outros objetos. O objetivo é minimizar o acoplamento entre classes, promovendo um sistema mais modular, flexível e fácil de manter. Ela afirma que um objeto só deve interagir com ele mesmo, com seus atributos diretos, com objetos que foram passados como parâmetros, com objetos que ele cria diretamente e com componentes de seus objetos diretamente associados. Desta forma, o princípio visa evitar o acoplamento

entrelaçado, no qual mudanças em uma classe acabam afetando várias outras devido à dependência de seus detalhes internos.

Code smells

Indicadores de fragilidades em código-fonte são, de acordo com Fowler [11] e Shvets [59], denominados *Code Smells*. Em código-fonte funcional, porém mal desenvolvido, *code smells* são indicativos da necessidade de refatoração ocasionada por escolhas pouco otimizadas de design.

O termo *Code Smells* foi cunhado, conforme relatado por Fowler [11] em seu livro "*Refactoring: Improving the Design of Existing Code*", por Kent Beck. Segundo Beck e Fowler, código desenvolvido de forma precária, rotinas mal planejadas e gambiarras podem ser identificados a partir da observação do código. Em sua obra, foram definidas classificações para os diversos tipos de *code smells*, expandidas por Shvets [59] e aqui apresentadas de acordo com a categorização do último:

- *Bloaters*: São códigos, métodos e classes que se tornaram excessivamente longos e complexos, dificultando sua manutenção. Geralmente, são resultados de processos incrementais desacompanhados de refatoração. Dentro deste grupo, pode-se encontrar *code smells* dos tipos:
  - *Long Method*: Acontece quando um método é muito extenso, contendo funções que poderiam ser distribuídas em outros métodos menores. Geralmente, sua ocorrência está relacionada ao desenvolvimento incremental do software, quando múltiplas pequenas adições acabam desenvolvendo uma classe longa e sobrecarregada, quando deveriam ser divididas em métodos menores com funções mais específicas.
  - *Large Class*: Acontece quando uma classe é sobrecarregada com excessivos atributos e métodos.
  - *Primitive Obsession*: Acontece na utilização de tipos primitivos quando o uso de objetos é mais adequado. O uso de *float* ao invés de objetos para representar valores monetários, por exemplo, é uma incidência deste *smell*.
  - *Long Parameter List*: Ocorre em métodos com muitos parâmetros. Geralmente, é caracterizada por métodos com mais de quatro parâmetros, muitas vezes de tipos primitivos, quando a parametrização por objetos ou polimorfismo estático é mais adequada.
  - *Data Clumps*: Ocorre quando trechos idênticos de código são repetidos em classes diversas. Um estereótipo deste problema é a repetição das configurações para conexão com banco de dados em várias classes, quando deveriam ser centralizadas e gerenciadas em sua própria classe.

- *Object-orientation abusers*: Os *smells* desta categoria são resultantes da aplicação incorreta ou incompleta dos princípios da programação orientada a objetos.
  - *Switch statements*: É relativamente incomum o uso de estruturas `switch/case` ou estruturas `if/else` muito longas na programação orientada a objetos. De modo geral, os *switches* são, com raras exceções, indicativos de problemas, e devem ser substituídos pela correta aplicação de polimorfismo.
  - *Temporary fields*: São atributos presentes em classes, porém usados apenas em situações muito específicas, de modo que sua integração na classe não se justifica.
  - *Refused bequest*: É um tipo de aplicação incorreta do princípio da herança. Ocorre quando a classe pai compartilha poucos métodos ou atributos com a classe filha, indicando que a relação é muito fraca. Nestes casos, a estrutura das classes deve ser repensada ou a herança substituída por delegação.
  - *Alternative classes with different interfaces*: Ocorre quando duas classes diferentes apresentam funções idênticas, porém com nomes diferentes. Isso pode ocorrer quando um programador não tem conhecimento de todo o projeto e adiciona uma nova classe sem saber que já estava implementada. Quando ocorre, pode ser solucionada pela unificação ou, quando houver pequenas diferenças, extração em superclasse.
- *Change preventers*: Esta categoria caracteriza os *smells* que ocorrem quando, ao realizar uma alteração em algum lugar de seu código, várias outras partes precisam ser ajustadas também.
  - *Divergent change*: Ocorre quando, ao alterar um item em uma classe, muitas alterações nessa classe são necessárias. Um exemplo típico é na adição de uma nova categoria de produto, que requer o ajuste dos métodos de busca, exibição, ordenação, etc. Isso ocorre quando a estrutura da classe não foi bem planejada ou há código duplicado, e pode ser corrigido ao extrair partes comuns para uma superclasse.
  - *Shotgun surgery*: Parecido com o *Divergent Change*, porém ocorre quando uma alteração requer pequenos ajustes em diversas classes diferentes. Pode ocorrer quando uma única responsabilidade é dividida em muitas classes, e pode ser corrigido ao mover atributos e métodos, em alguns casos para novas classes, unificando-os.
  - *Parallel inheritance hierarchies*: Ocorre quando há duas ou mais hierarquias de classes que são interdependentes, ou seja, para cada classe em uma hierarquia, existe uma classe correspondente em outra hierarquia. A solução requer

a unificação das hierarquias, reduzindo o código duplicado e melhorando a organização do código.

- *Dispensables*: Esta categoria engloba elementos desnecessários, que, quando retirados, tornam o código mais limpo, eficiente e compreensível.
  - *Comments*: Comentários são importantes para auxiliar na manutenção do código e no trabalho em equipe, porém, quando são utilizados em excesso, podem indicar gambiarras, código removido e esquecido ou código de baixa legibilidade.
  - *Duplicate code*: Ocorre quando dois fragmentos de código são idênticos ou executam a mesma tarefa. Geralmente, ocorre quando múltiplos desenvolvedores trabalhando no mesmo projeto não têm conhecimento do trabalho de seus colegas, levando à implantação duplicada de soluções. É fácil identificar código idêntico, porém, códigos diferentes que realizam a mesma função são de identificação mais complexa.
  - *Lazy class*: É caracterizado pela existência de classes minúsculas, de pouca utilidade. Geralmente, ocorrem quando classes refatoradas perdem importância. A delegação de sua funcionalidade a outras e subsequente deleção é uma solução aplicável.
  - *Data class*: Refere-se a classes que contêm apenas atributos e métodos de acesso, sem outras funções no projeto. Algumas arquiteturas requerem sua existência, então sua refatoração precisa ser cautelosa.
  - *Dead code*: Ocorre quando variáveis, atributos, métodos ou classes não são mais usados. Sua incidência é observada após refatorações que mudam a dinâmica do software, tornando elementos do código obsoletos, requerendo sua remoção.
  - *Speculative generality*: É caracterizado por classes, métodos, atributos ou parâmetros criados em vista de uma futura expansão que não se realiza ou para tratamento de situações hipotéticas que não ocorrem. Sua remoção limpa o código, reduz a complexidade do projeto e aumenta a eficiência.
- *Couplers*: Os *smells* desta seção ocorrem quando há delegação excessiva entre as classes, o que torna o projeto muito complexo e as classes excessivamente acopladas.
  - *Feature envy*: Ocorre quando um método acessa excessivamente os dados de outro objeto, mais frequentemente do que os de sua própria classe. Sua solução requer a movimentação dos métodos para a classe à qual é mais íntima, porém, é necessário avaliar a arquitetura implementada, pois alguns modelos requerem sua existência.

- *Inappropriate intimacy*: Ocorre quando uma classe utiliza atributos ou métodos internos de outra classe. Além do excessivo acoplamento, pode indicar problemas de encapsulamento e herança.
- *Message chains*: Sua incidência é caracterizada em situações em que um objeto requisita um segundo objeto, que por sua vez requisita um terceiro e assim por diante. É um indicativo forte de acoplamento excessivo e inadequado, requerendo uma análise da possibilidade de realocação do método final dentro da arquitetura.
- *Middle man*: É tipificado em classes cuja única função é delegar sua funcionalidade a uma terceira, e costuma ser acompanhada do *code smell Message chain*. Sua remoção e realocação dos métodos é indicada nestas situações.

2.4 Aprendizagem de programação

Moser [5] aponta que a aprendizagem de programação é um processo intimidante que exige o desenvolvimento e equilíbrio de habilidades em múltiplas camadas. O autor observa que o processo de compreensão e desenvolvimento das habilidades necessárias é um processo *bottom-up* que começa pela sintaxe, seguida pela estrutura e estilo. Tan; Ting & Ling [10] observam que, durante este processo progressivo de aprendizagem, é comum o desenvolvimento de conceitos equivocados ou imprecisos, dado que a atenção é direcionada principalmente para conceitos fundamentais de sintaxe, em detrimento de estrutura e estilo. Isso produz estudantes excessivamente focados em linguagens de programação específicas, em detrimento do domínio da habilidade geral de programação, o que se manifesta de diversas formas: desde a dificuldade em aprender outras linguagens de programação até a persistência de maus hábitos que incidem na baixa qualidade do software produzido.

2.4.1 Dificuldades no aprendizado de programação

Alguns estudos dedicaram-se a analisar as dificuldades de aprendizagem no contexto da programação, como os já citados trabalhos de Moser [5], Tan; Ting & Ling [10], Watson & Li [3], Martins; Almeida Souza Concilio & Paiva Guimarães [4], Konecki [6], Mazaitis [8], Keuning; Jeuring & Heeren [9], e Tan; Ting & Ling [10].

Lahtinen; Ala-Mutka & Järvinen [60] analisaram relatos de mais de 500 estudantes de programação introdutória ao redor do mundo, coletando suas impressões de aprendizagem e relatos de dificuldades. Suas conclusões reafirmam que o processo de aprendizagem de programação é difícil [3, 4], destacando as dificuldades de abstração e construção dos programas.

Em 2020, Cheah [61] publicou uma revisão da literatura identificando os fatores que contribuem para as dificuldades no ensino e na aprendizagem da programação de computadores. Dentre os fatores apontados, destacam-se a falta de uma base sólida de lógica de programação, o uso de ferramentas e linguagens mais adequadas para a indústria do que para propósitos educacionais, e a ansiedade causada pela percepção de que programar é muito difícil. Além destes, o autor destacou que o erro mais crítico que pode ocorrer é a projeção de um modelo mental falso, pois leva a um design incorreto, resultados errados e bugs.

Ao estreitar o enfoque para estudos que analisam dificuldades na aprendizagem de programação orientada a objetos, destacam-se alguns poucos trabalhos. Ismail; Ngah & Umar [62] afirmam que o processo de ensino e de aprendizagem na programação orientada a objetos é diferente da programação introdutória ou estruturada. Os autores afirmam que o uso de técnicas convencionais, como pseudocódigo e fluxogramas, é adequado apenas para o ensino de programação estruturada. Quando se trata de ensinar programação orientada a objetos, essas técnicas convencionais falham em fornecer explicações e compreensão adequadas para os alunos.

O trabalho de Kölling [63] aborda as dificuldades no ensino de programação orientada a objetos. O autor afirma que, infelizmente, a programação orientada a objetos costuma ser precedida pelo ensino de programação procedural, contribuindo para uma visão equivocada de que a orientação a objetos é apenas mais um recurso da linguagem que pode ser ensinado depois das estruturas de controle, ponteiros e recursão. O autor defende que a orientação a objetos é um paradigma fundamental que molda toda a nossa maneira de pensar sobre como mapear um problema para um modelo algorítmico. Ela determina, de maneira fundamental, a estrutura de até mesmo programas simples. Não pode ser "adicionada"a outras construções da linguagem; em vez disso, substitui a estrutura fundamental da programação procedural e, por esse motivo, deve ser ensinada desde o início.

Ainda que os diversos trabalhos citados tenham se debruçado sobre o estudo das dificuldades da aprendizagem de programação, apenas o trabalho de Gutiérrez; Guerrero & López-Ospina [12] caracteriza os tipos de dificuldades identificadas. Seu trabalho "*Ranking of problems and solutions in the teaching and learning of object-oriented programming*"estabelece 14 categorias para classificar as dificuldades de aprendizagem dos estudantes.

Na seção 3.1, são discutidos em detalhe a reflexão de algumas destas categorias junto aos indicadores de problemas, que são tratados como indicadores de dificuldades na aprendizagem.

3 ABORDAGEM PROPOSTA

Nesta seção, é apresentada a abordagem proposta para identificar e correlacionar problemas de qualidade de código com dificuldades de aprendizagem na programação orientada a objetos. São discutidos o mapeamento entre indicadores de problemas no código e dificuldades no entendimento dos conceitos fundamentais da POO; a heurística da ferramenta de IA que realiza a análise automatizada do código-fonte dos estudantes; e a interação com o modelo de IA da ferramenta.

A Figura 4 apresenta graficamente a proposta, destacando seus principais agrupamentos: o Mapeamento de Dificuldades, detalhado na seção 3.1; e a Heurística de Análise de Código, detalhada na seção 3.2.

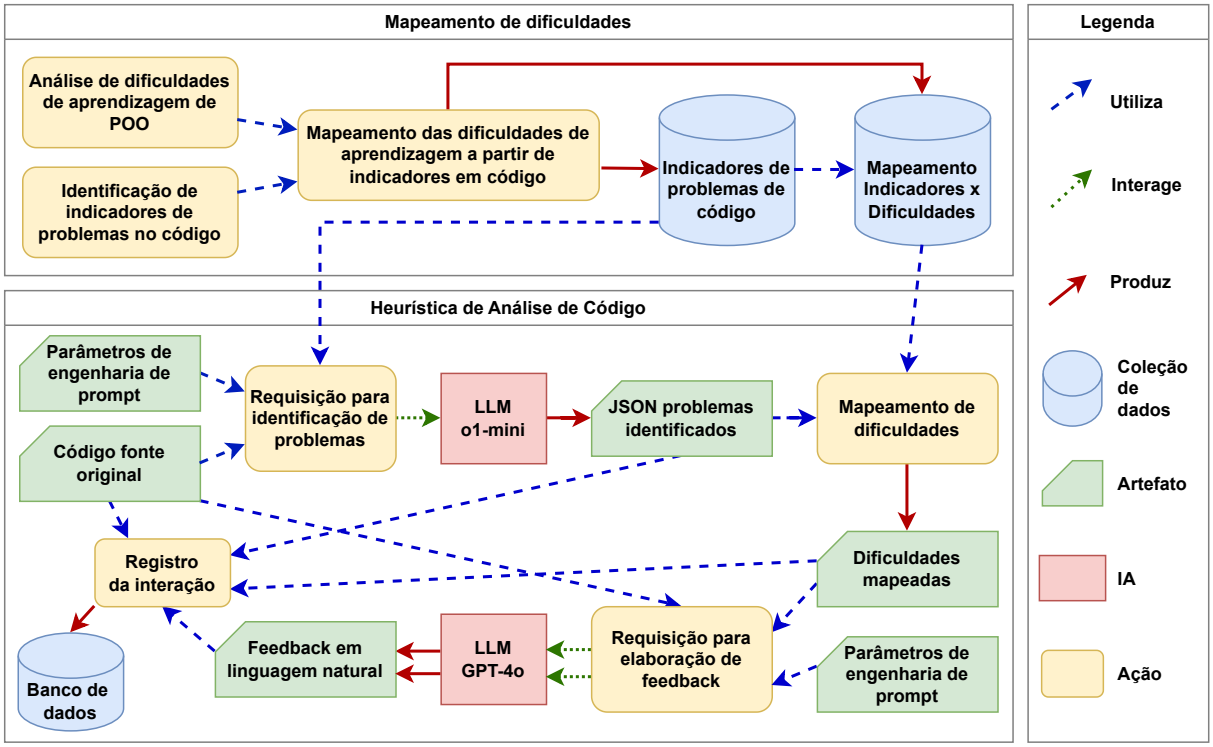


Figura 4 – Esquema da abordagem proposta. Fonte: autor.

3.1 Mapeamento de dificuldades

A análise dos parâmetros adota alguns preceitos e orientações da Teoria Fundamentada em Dados, que, segundo Corbin & Strauss [64], têm o objetivo de identificar, desenvolver e relacionar conceitos. Tem como base, de um lado, as categorias de dificuldades de aprendizagem na POO definidas por Gutiérrez; Guerrero & López-Ospina [12], e de outro, os indicadores de problemas no código-fonte. A codificação axial, apresen-

tada na Figura 5 é realizada buscando-se estabelecer relações e conexões entre ambos os lados, estabelecendo uma nova classificação. Esta classificação da relação de problemas no código e dificuldades de aprendizagem correspondentes é aplicável como parâmetro principal da heurística apresentada neste trabalho. A figura 5 apresenta a implementação desta metodologia.

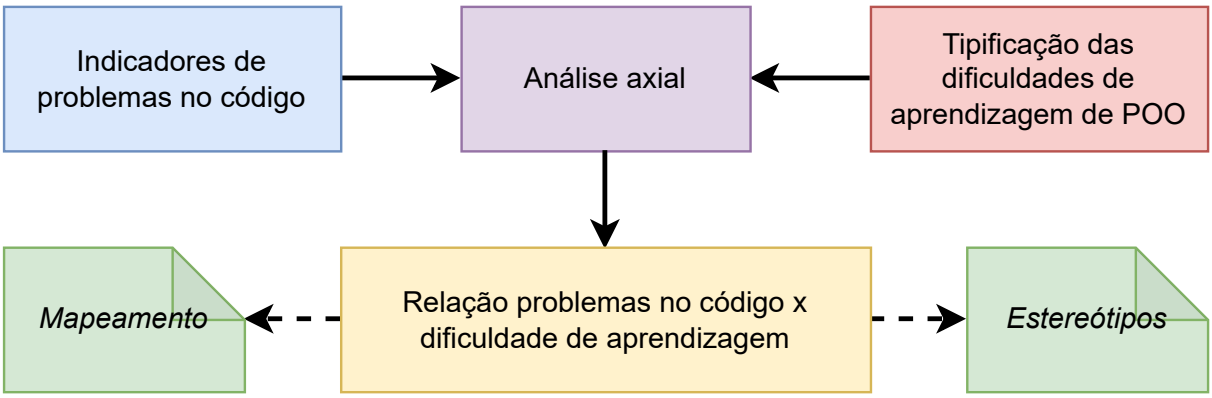


Figura 5 – Metodologia de mapeamento. Fonte: autor.

A análise axial resulta em dois artefatos: o mapeamento relacional da incidência de problemas no código e sua reflexão em indicadores das dificuldades de aprendizagem de programação orientada a objetos, e a caracterização estereotipada dos problemas no código. A Figura 6 apresenta o mapeamento, e na sequência é discutida a estereotipação dos problemas de código e aprendizagem.

Para o mapeamento, foram estabelecidas seis categorias de dificuldades, inspiradas na divisão apresentada por Gutiérrez; Guerrero & Lopez-Ospina [12] e discutida na seção 2.4.1. Esta divisão, apresentada a seguir, busca privilegiar a classificação das dificuldades relacionadas à programação orientada a objetos que sejam observáveis apenas a partir do código-fonte do estudante, a fim de que sua identificação possa ser realizada a partir dos estereótipos e através da análise do código por inteligência artificial, discutida na seção 4.2. Assim, foram definidas seis categorias de dificuldades de aprendizagem:

- C1: Dificuldades relacionadas ao entendimento de classes
- C2: Dificuldades em entender o conceito de método
- C3: Dificuldades em entender e implementar a orientação a objetos
- C4: Dificuldades em entender relacionamento entre objetos
- C5: Dificuldades em entender polimorfismo e sobrecarga
- C6: Dificuldades em entender encapsulamento

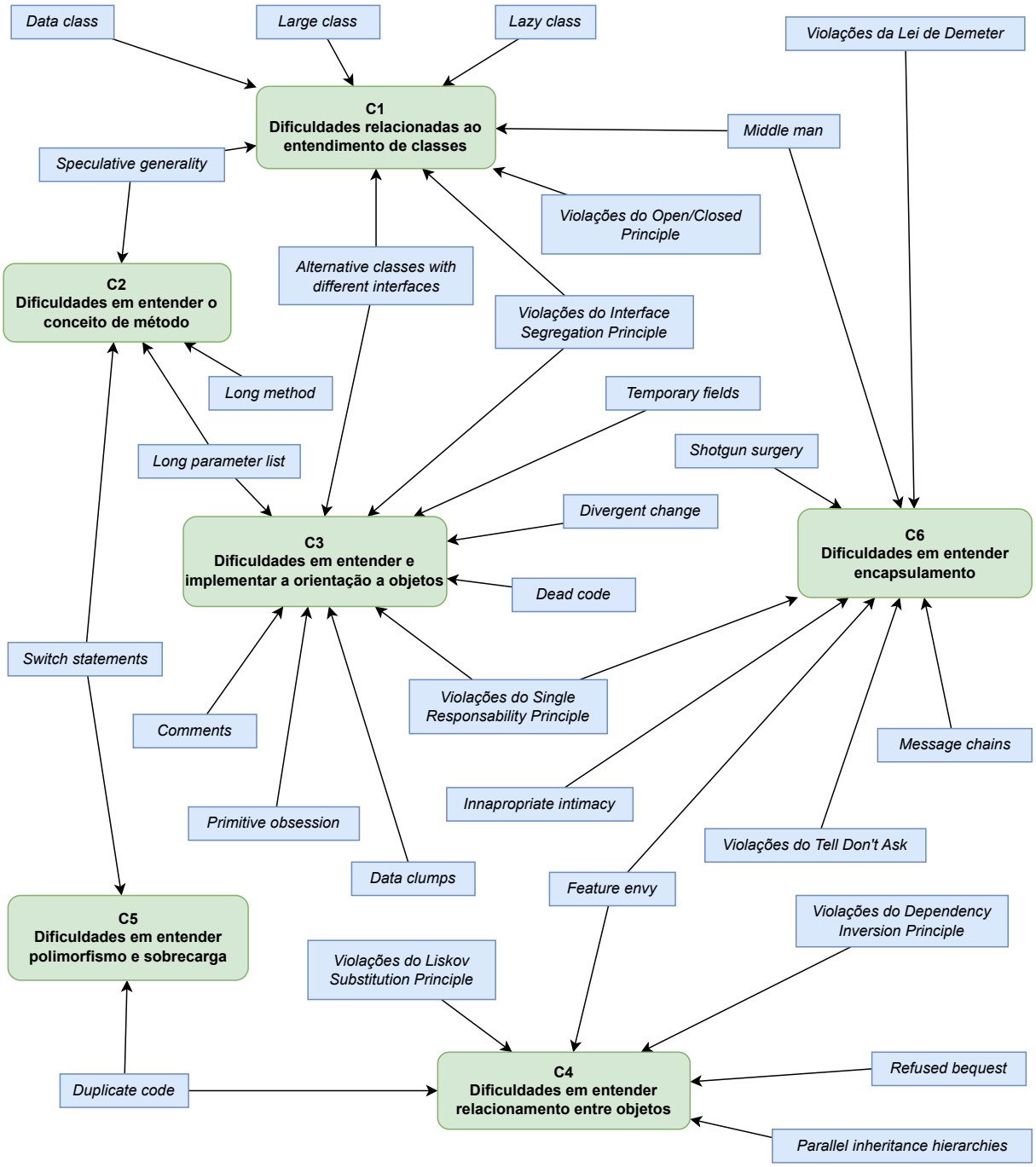


Figura 6 – Mapa de associação de problemas no código-fonte e dificuldades de aprendizagem. Fonte: autor.

O trabalho original de Gutiérrez; Guerrero & Lopez-Ospina [12] apresentava 14 categorias de dificuldades de aprendizagem aplicadas a todo o processo de ensino-aprendizagem de programação orientada a objetos:

- D01: Dificuldade em entender objetos e sua natureza dinâmica
- D02: Dificuldades relacionadas ao entendimento de classes

- D03: Dificuldade em entender o conceito de método
- D04: Dificuldade em entender, ensinar e implementar a orientação a objetos
- D05: Dificuldade em entender relacionamento entre objetos
- D06: Dificuldade em entender polimorfismo e sobrecarga
- D07: Dificuldade em entender encapsulamento
- D08: Complexidade com as linguagens de programação e ferramentas utilizadas no ensino e aprendizado de orientação a objetos
- D09: Dificuldade em ensinar e entender tópicos gerais de programação
- D10: Dificuldade em desenvolver o pensamento abstrato
- D11: Dificuldade em entender análise e design de software
- D12: Dificuldade em entender reutilização
- D13: Dificuldade com metodologias e técnicas de administração e gerenciamento de projetos
- D14: Dificuldade na implementação e manutenção de software

Como a proposta deste trabalho é a identificação de problemas de aprendizagem a partir do código-fonte, algumas categorias de dificuldades originais do trabalho de Gutiérrez; Guerrero & Lopez-Ospina [12] não se aplicam ao escopo ou se aplicam apenas parcialmente.

A categoria C1 corresponde integralmente à dificuldade D02, que abrange os desafios enfrentados pelos estudantes na assimilação da natureza estática e da hierarquia das classes. São incluídos nessa categoria problemas de qualidade de código que indicam uma implementação inadequada da estrutura de classes.

De modo semelhante, a categoria C2 corresponde integralmente à dificuldade D03, que se refere a desafios relacionados à implementação de métodos. Nessa categoria, são incluídas implementações inadequadas ou incorretas de métodos.

A categoria C3 apresenta um escopo mais amplo, pois abrange dificuldades relacionadas à compreensão e implementação da programação orientada a objetos. Alguns conceitos discutidos por Gutiérrez, Guerrero e Lopez-Ospina [12] são contemplados nessa categoria, especialmente as dificuldades D01 e D04. A dificuldade D01 é integralmente compatível, uma vez que está diretamente relacionada ao conceito de objeto. Já a dificuldade D04 é apenas parcialmente considerada, pois sua dimensão associada a desafios no

processo de ensino é excluída, uma vez que tais dificuldades não são observáveis diretamente no código-fonte e exigiriam uma análise mais ampla do contexto pedagógico, o que ultrapassa o escopo deste estudo.

A categoria C4 corresponde às dificuldades na compreensão do relacionamento entre objetos, alinhando-se ao que é descrito na dificuldade D05.

A categoria C5 inclui integralmente a dificuldade D06, que define dificuldades no entendimento do polimorfismo e sobrecarga, e também a dificuldade D12, sobre reuso, segundo o entendimento de que uma estrutura polimórfica adequada reduz a redundância e a ocorrência de código duplicado [65].

A categoria C6 é associada a dificuldades com encapsulamento, alinhado ao definido na dificuldade D07.

Algumas dificuldades definidas por Gutiérrez; Guerrero & Lopez-Ospina [12] não são aplicáveis ao escopo, totalmente ou em parte, e não foram incluídas na organização de seis categorias. A dificuldade D04 trata do processo de ensino, que extrapola a possibilidade de identificação de problemas a partir do código-fonte. A D08 também vai além por especificar dificuldades com as linguagens e ferramentas de desenvolvimento. A dificuldade D09 inclui conceitos básicos e gerais de programação, como variáveis, funções e estruturas de controle, assuntos que são identificados a partir do código-fonte porém anteriores à POO.

As dificuldades D10, relacionada ao pensamento abstrato, a D11, relacionada a aplicação de conceitos de design em *Unified Modeling Language* (UML), a D13, relacionada ao gerenciamento de projetos e a D14, relacionada a implementação e manutenção do software, extrapolam o escopo de problemas observáveis em código e não foram incluídas na organização de seis categorias.

3.1.1 Categoria C1: Dificuldades relacionadas ao entendimento de classes

Essa dificuldade é descrita como a complexidade apresentada pelos estudantes ao assimilar a natureza estática e a profundidade das classes. É desafiador para eles entender a hierarquia e a identificação das classes corretas. Os indicadores de problemas em código-fonte correspondentes a estas dificuldades são:

- *Code smell alternative classes with different interfaces*: classes que deveriam ter comportamentos relacionados ou similares devem compartilhar uma abstração comum, como uma interface ou classe base.
- *Code smell data class*: caracteriza a falha em capturar a abstração completa de um conceito, pois se limita a armazenar dados sem incluir comportamentos associados a esses dados.

- *Code smell large class*: quando uma classe possui uma grande quantidade de métodos e atributos, torna-se difícil entender seu propósito ou como deve ser usada, o que indica que as responsabilidades podem estar misturadas, prejudicando a manutenção.
- *Code smell lazy class*: quando a classe não representa um comportamento ou conceito significativo.
- *Code smell middle man*: quando a classe intermediária não está gerenciando ou ocultando detalhes de implementação de maneira eficaz.
- *Code smell speculative generality*: indica uma abstração prematura ou desnecessária, na qual a classe é mais complexa do que o necessário para o problema atual.
- Violações do *Interface Segregation Principle*: quando o princípio é violado, interfaces grandes e generalizadas acabam determinando responsabilidades desnecessárias, o que quebra a abstração adequada.
- Violações do *Open/Closed Principle*: quando o princípio é violado, classes ou módulos existentes precisam ser modificados diretamente para implementar novas funcionalidades, em vez de serem estendidos.

3.1.2 Categoria C2: Dificuldades em entender o conceito de método

Essa categoria engloba a complexidade apresentada pelos estudantes ao assimilar o conceito de método. Os estudantes apresentam dificuldades para elaborar e implementar estruturas de métodos, além de não entenderem como reutilizá-los ou onde colocá-los corretamente. Os indicadores de problemas em código-fonte correspondentes a estas dificuldades são:

- *Code smell long method*: indica que a abstração não foi bem aplicada, pois o método está lidando com muitos detalhes em vez de encapsular uma ação mais concisa e clara.
- *Code smell long parameter list*: indica que os dados necessários não estão sendo parametrizados adequadamente através de objetos ou há excesso de responsabilidades no método.
- *Code smell speculative generality*: indica uma abstração prematura ou desnecessária, em que o método é mais complexo do que o necessário para o problema atual.
- *Code smell switch statements*: o uso excessivo de estruturas de controle como `switch` ou `if-else` para determinar o comportamento de diferentes tipos é um sinal de que o polimorfismo não está sendo observado de maneira adequada nos métodos, sendo necessária a avaliação da possibilidade de aplicação de sobrecarga.

3.1.3 Categoria C3: Dificuldades em entender e implementar a orientação a objetos

Essa dificuldade está relacionada ao desafio de adotar o paradigma orientado a objetos, pois seu processo formativo inicial é geralmente baseado em programação puramente estrutural. A natureza modular do paradigma orientado a objetos é concebida como um desafio para os educadores, já que, nesse processo, é comum que os estudantes assimilem concepções errôneas e apresentem dificuldades em entender e implementar os padrões orientados a objetos. Os indicadores de problemas em código-fonte correspondentes a estas dificuldades são:

- *Code smell alternative classes with different interfaces*: classes que deveriam ter comportamentos relacionados ou similares devem compartilhar uma abstração comum, como uma interface ou classe base.
- *Code smell comments*: indica que o próprio código não é suficientemente claro ou autoexplicativo, mascarando uma implementação pobre da orientação a objetos. Código bem abstraído deve ser suficientemente legível e expressivo por si só, minimizando a necessidade de comentários.
- *Code smell data clumps*: indica a necessidade de que as variáveis ou parâmetros que frequentemente se repetem devem ser encapsulados em uma classe e aplicada a herança.
- *Code smell dead code*: viola o princípio de abstração da orientação a objetos, pois esses elementos não representam comportamentos ou dados relevantes para o sistema, sendo desnecessários.
- *Code smell divergent change*: indica que a classe está assumindo múltiplas responsabilidades não relacionadas.
- *Code smell long parameter list*: indica que os dados necessários não estão sendo parametrizados adequadamente através de objetos, indicando problemas na adequação do código aos princípios da programação orientada a objetos.
- *Code smell primitive obsession*: os tipos primitivos devem ser abstraídos para classes apropriadas, garantindo o cumprimento da programação orientada a objetos no projeto.
- *Code smell temporary fields*: atributos devem pertencer apenas a contextos em que são realmente necessários, indicando que há falhas na abstração do problema em código orientado a objetos.

- Violações do *Interface Segregation Principle*: quando o princípio é violado, interfaces grandes e generalizadas acabam impondo métodos desnecessários aos clientes, o que quebra a abstração adequada.
- Violações do *Single Responsibility Principle*: quando o princípio é violado, uma classe assume múltiplas responsabilidades não relacionadas, resultando em baixa coesão.

3.1.4 Categoria C4: Dificuldades em entender relacionamento entre objetos

Refere-se à dificuldade que os estudantes têm ao entender e implementar relações orientadas a objetos, como associação, dependência, generalização/especialização-herança, composição e agregação. Esses problemas são comuns devido à falta de experiência dos alunos em relação ao paradigma de programação orientada a objetos. Os estudantes geralmente apresentam dificuldades no processo de modelagem dessas relações e, consequentemente, na implementação e aplicação de conceitos que frequentemente são concebidos como complexos. Os indicadores de problemas em código-fonte correspondentes a estas dificuldades são:

- *Code smell duplicate code*: indica uma falha na criação de abstrações adequadas, pois trechos de código repetidos poderiam ser abstraídos em métodos ou classes reutilizáveis relacionadas.
- *Code smell feature envy*: sugere que a lógica de um método deveria ser movida para a classe excessivamente acessada, de modo a manter o encapsulamento adequado.
- *Code smell parallel inheritance hierarchies*: indica uma má aplicação de herança, na qual a relação entre as classes deveria ser refatorada para evitar a duplicação de hierarquias e melhorar o design.
- *Code smell refused bequest*: indica que a herança foi mal aplicada, pois a subclasse não deveria herdar comportamentos que não lhe são úteis.
- Violações do *Dependency Inversion Principle*: quando o princípio é violado, classes de alto nível dependem diretamente de classes concretas de baixo nível, em vez de dependerem de interfaces ou abstrações.
- Violações do *Liskov Substitution Principle*: indica uma má aplicação da herança, na qual a subclasse não preserva a lógica ou as garantias estabelecidas pela superclasse.

3.1.5 Categoria C5: Dificuldades em entender polimorfismo e sobrecarga

Essa dificuldade está relacionada ao alto nível de complexidade que os conceitos de polimorfismo e sobrecarga apresentam no momento de iniciar um estudante na área

de programação. Os indicadores de problemas em código-fonte correspondentes a estas dificuldades são:

- *Code smell duplicate code*: indica uma falha na criação de abstrações adequadas, pois trechos de código repetidos poderiam ser abstraídos em métodos polimórficos e aplicada a sobrecarga.
- *Code smell switch statements*: o uso excessivo de estruturas de controle como switch ou if-else para determinar o comportamento de diferentes tipos é um sinal de que o polimorfismo não está sendo observado de maneira adequada nos métodos, sendo necessária a avaliação da possibilidade de aplicação de sobrecarga.

3.1.6 Categoria C6: Dificuldades em entender encapsulamento

Essa categoria está relacionada à dificuldade de assimilação de várias concepções errôneas sobre o entendimento de encapsulamento, modularidade e ocultação de informações. Os indicadores de problemas em código-fonte correspondentes a estas dificuldades são:

- *Code smell feature envy*: sugere que a lógica de um método deveria ser movida para a classe excessivamente acessada, de modo a manter o encapsulamento adequado.
- *Code smell inappropriate intimacy*: viola o princípio de encapsulamento, que preconiza que os detalhes internos de uma classe devem ser ocultos e protegidos.
- *Code smell message chains*: indica que o encapsulamento está sendo violado, pois o código depende fortemente de uma cadeia de objetos internos, expondo detalhes de implementação de várias classes e aumentando o acoplamento.
- *Code smell middle man*: ocorre quando a classe intermediária não está gerenciando ou ocultando detalhes de implementação de maneira eficaz.
- *Code smell shotgun surgery*: indica que o comportamento não está devidamente encapsulado em uma única classe ou módulo, forçando mudanças em vários lugares.
- Violações da Lei de Demeter: demonstra que o encapsulamento está sendo comprometido pela revelação de aspectos da implementação interna das classes.
- Violações do *Single Responsibility Principle*: quando o princípio é violado, uma classe assume múltiplas responsabilidades não relacionadas, resultando em baixa coesão.
- Violações do princípio *Tell don't ask*: quebra o encapsulamento, já que o objeto expõe seu estado interno a outro objeto, em vez de encapsular o comportamento e gerenciar seus próprios dados.

Ao estabelecer uma relação estruturada entre problemas recorrentes no código-fonte e suas correspondentes dificuldades conceituais, esse mapeamento contribui para o avanço do estado da arte no ensino de POO, fornecendo um *framework* analítico que permite a identificação de desafios enfrentados pelos estudantes.

3.2 Heurística de Análise de Código

A análise de código-fonte com o objetivo de identificar problemas e gerar devolutivas educacionais, conforme proposto neste trabalho, é realizada por meio de uma série de procedimentos descritos detalhadamente nesta seção. A Figura 7 apresenta a abordagem heurística, ilustrando os procedimentos aqui delineados e estabelecidos.

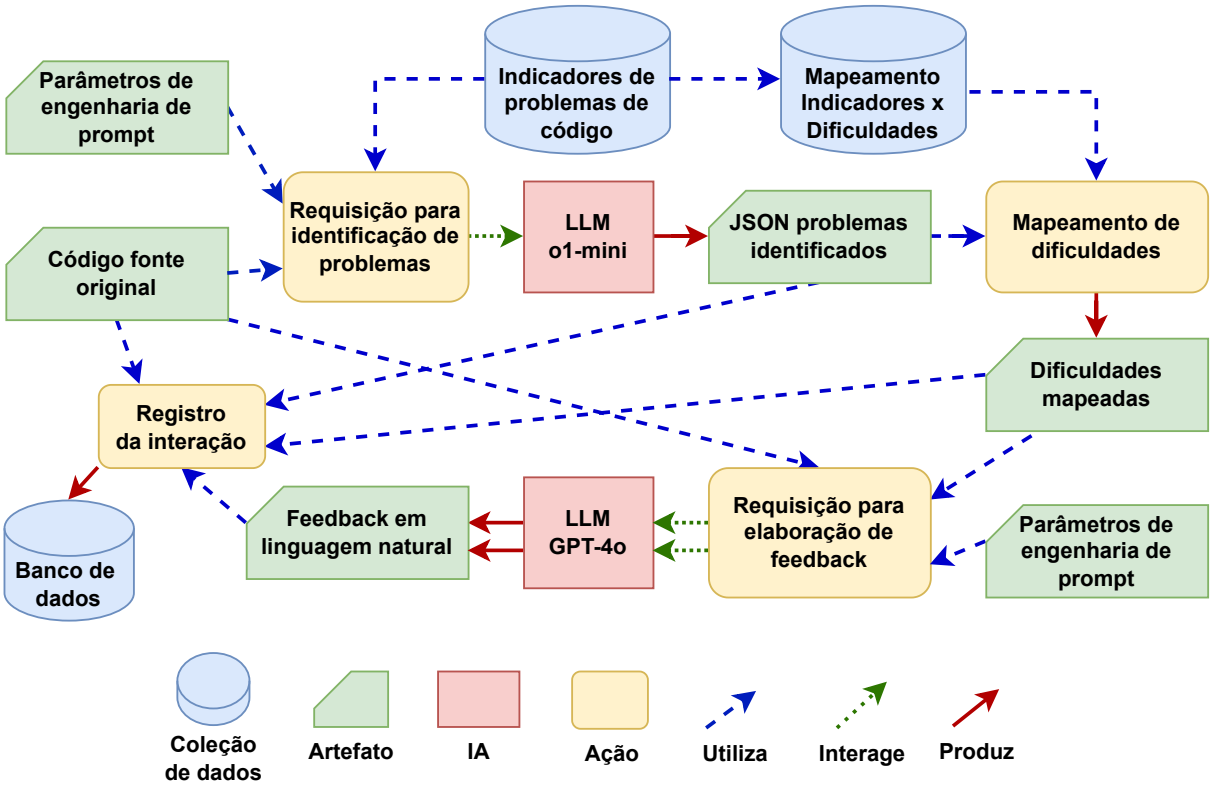


Figura 7 – Heurística de análise de código. Fonte: autor.

3.2.1 Requisição para identificação de problemas

A heurística inicia sua primeira interação com dois artefatos principais: o código-fonte a ser analisado e os parâmetros de engenharia de *prompt*. Segundo Liu et al. [47], *prompt* é um conjunto de instruções fornecido a um LLM que o programa, personalizando e/ou aprimorando suas capacidades. A engenharia dos *prompts* submetidos aos modelos define as características de suas respostas [48, 47], contendo instruções que delimitam a atuação do modelo de IA a partir da especificação de um perfil — o de um professor de ciência da computação — e de sua função: analisar o código-fonte fornecido e identificar

a ocorrência de problemas. A requisição também inclui uma estrutura *JavaScript Object Notation* (JSON) de referência que lista e qualifica os problemas de código a serem verificados, especificando que o retorno deve sinalizar se os problemas foram encontrados. O código a seguir ilustra uma parte dessa estrutura:

```
{
    "data_class_smell": {
        "type": "boolean",
        "required": true,
        "description": "Indicates whether the code has a data
            class smell"
    },
    "large_class_smell": {
        "type": "boolean",
        "required": true,
        "description": "Indicates whether the code
            has a large class smell"
    },
    "lazy_class_smell": {
        "type": "boolean",
        "required": true,
        "description": "Indicates whether the code has a
            lazy class smell"
    },
    "open_close_principle_violation": {
        "type": "boolean",
        "required": true,
        "description": "Indicates whether there is a
            violation of the Open/Closed Principle"
    },
    ...
}
```

A requisição é então submetida ao primeiro LLM, que utiliza os recursos de sua base de conhecimento para executar os procedimentos especificados. O LLM escolhido não requer ajuste prévio, porém deve ser habilitado para realizar análise de código-fonte e possuir uma base de conhecimento adequada para tarefas relacionadas à programação. Modelos como Llama 2[1] e Gemini Nano[2], por exemplo, são inadequados para esta tarefa

[1] https://huggingface.co/meta-llama/Llama-2-7b
[2] https://deepmind.google/technologies/gemini/nano/

por não terem base de conhecimento adequada, enquanto modelos como o o1-mini[3], Code Llama[4] e StarCoder[5] o são pois sua base inclui código-fonte.

O retorno do LLM consiste em um documento estruturado, no qual são listados os problemas de código e os valores booleanos que indicam sua incidência no código-fonte, conforme os metadados da requisição respondida. O código a seguir exemplifica como devem ser os retornos:

```
"checks": {
    "data_class_smell": false,
    "large_class_smell": false,
    "lazy_class_smell": false,
    "open_close_principle_violation": false,
    "speculative_generality_smell": false,
    "alternative_classes_with_different_interfaces_smell": false,
    "interface_segregation_principle_violation": false,
    "middle_man_smell": false,
    "long_method_smell": false,
    "long_parameter_list_smell": false,
    "switch_statements_smell": true,
    "comments_smell": false,
    "data_clumps_smell": false,
    "dead_code_smell": false,
    "divergent_change_smell": false,
    "primitive_obsession_smell": false,
    "temporary_fields_smell": false,
    "single_responsability_principle_violation": false,
    "parallel_inheritance_hierarchies_smell": false,
    "refused_bequest_smell": false,
    "dependency_inversion_principle_violation": false,
    "liskov_substitution_principle_violation": false,
    "duplicate_code_smell": false,
    "feature_envy_smell": false,
    "inappropriate_intimacy_smell": false,
    "message_chains_smell": false,
    "shotgun_surgery_smell": false,
    "demeter_law_violation": false,
    "tell_dont_ask_principle_violation": false
```

[3] https://openai.com/index/openai-o1-mini-advancing-cost-efficient-reasoning/
[4] https://ollama.com/library/codellama
[5] https://huggingface.co/blog/starcoder

```
}
```

3.2.2 Mapeamento de dificuldades

Os indicadores de problemas de código retornados pela primeira interação com a IA passam, então, por um processo de mapeamento. Os valores booleanos desses indicadores são analisados em conjunto com o mapa de dificuldades e indicadores apresentado na Figura 6. Uma nova lista de dificuldades mapeadas é gerada, contendo aquelas que apresentaram ao menos um indicador positivo, as quais são marcadas como presentes, e aquelas que não apresentaram indicadores, marcadas como ausentes.

Como exemplo, se um trecho de código apresentar o *Code Smell Duplicate Code*, o mapeamento indicará dificuldades em entender polimorfismo, sobrecarga e também dificuldades em compreender o relacionamento entre objetos. O código a seguir exemplifica as dificuldades mapeadas neste caso:

```
{
    "classes_difficulties": false,
    "method_difficulties": false,
    "object_orientation_difficulties": false,
    "object_relations_difficulties": true,
    "polymorphism_overloading_difficulties": true,
    "encapsulation_difficulties": false
}
```

3.2.3 Requisição para elaboração de *feedback*

Aproveitando a reconhecida capacidade dos modelos de IA generativa na construção de textos em linguagem natural, esta etapa tem como objetivo elaborar um *feedback* educativo personalizado, contribuindo para a superação das dificuldades de aprendizagem identificadas nas etapas anteriores.

É importante destacar que, segundo as observações de Lau & Guo [41], Rahman & Watanobe [66], e também de Silva et al. [22], quando o objetivo da interação com a IA é pedagógico a geração automática de código das atividades propostas é indesejada, pois desincentiva o estudante à reflexão e favorece o plágio. A requisição de *feedback* deve ser adequadamente parametrizada buscando garantir que a ferramenta contribua de modo seguro e positivo para o estudante.

É necessária a adição de parâmtros em linguagem natural para elaboração adequada do texto de *feedback*: uma clara especificação dos tópicos a serem tratados — delimitando ao modelo sua tarefa de aconselhar sobre problemas de qualidade limitados ao escopo e com finalidade educacional, e controles que impeçam a IA de refatorar o

código do estudante. Além disso, a requisição deve incluir o código original, o contexto dos problemas de código identificados e as dificuldades de aprendizagem associadas, assegurando que o modelo disponha de informações completas para elaborar um *feedback* preciso e relevante.

A requisição ao LLM é realizada em duas etapas: na primeira, busca-se gerar um *feedback* breve e direto para exibição imediata ao usuário; na segunda, busca-se produzir um *feedback* completo, detalhando cada uma das dificuldades identificadas e fornecendo orientações educativas específicas para abordá-las.

3.2.4 Registro da interação

A heurística determina que todos os artefatos, exceto os parâmetros de engenharia de *prompt*, sejam documentados em uma interação e armazenados em banco de dados. Embora não haja uma aplicação específica para esses documentos na heurística, seu armazenamento é destinado ao uso futuro, com a finalidade de manter dados para a geração de relatórios, indicadores e auditorias. Cada documento deve possuir um identificador único que possibilite sua consulta a qualquer momento.

4 FERRAMENTA CODEBUDDY

Para a primeira implementação prática da heurística descrita no capítulo anterior, foi desenvolvido um protótipo de aplicação web simples, oferecendo uma interface temporária para testes e experimentação. A escolha por uma aplicação web deveu-se à sua flexibilidade e facilidade de modificação, características ideais para um protótipo temporário. Essa aplicação foi utilizada para validar a implementação da heurística, que posteriormente foi transformada em uma API (*Application Programming Interface*, ou Interface de Programação de Aplicação) e acessada por meio de uma extensão para a IDE Microsoft Visual Studio Code[1], frequentemente abreviada como VSCode.

A ferramenta foi denominada **CodeBuddy**, destacando o conceito de parceria e suporte contínuo ao usuário. Sua API pode ser acessada por meio da extensão para VSCode "*CodeBuddy for CS Education*"ou através de uma interface web que incorpora um cliente da API com funcionalidades iguais às da extensão.

Até o final de 2024, não foram identificadas outras ferramentas disponíveis no Marketplace de extensões do VSCode que utilizem IA com enfoque educacional, tornando esta, a primeira vista, uma solução inédita.

4.1 Arquitetura da ferramenta

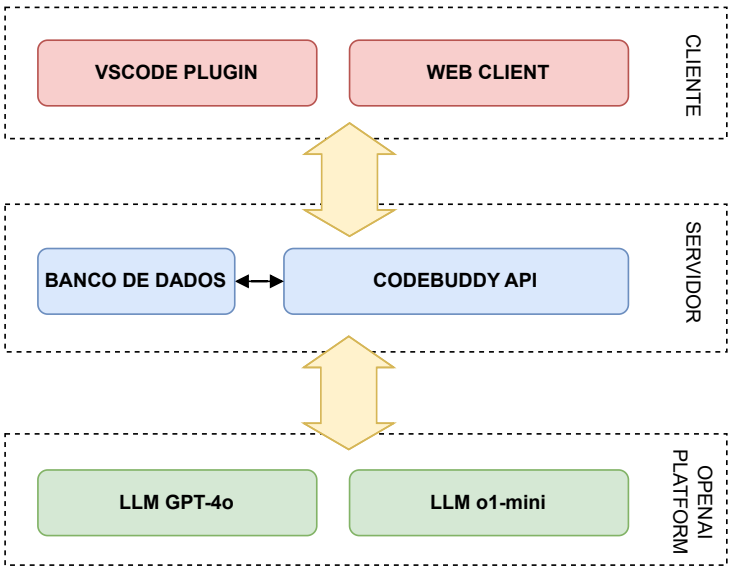


Figura 8 – Arquitetura da ferramenta. Fonte: autor.

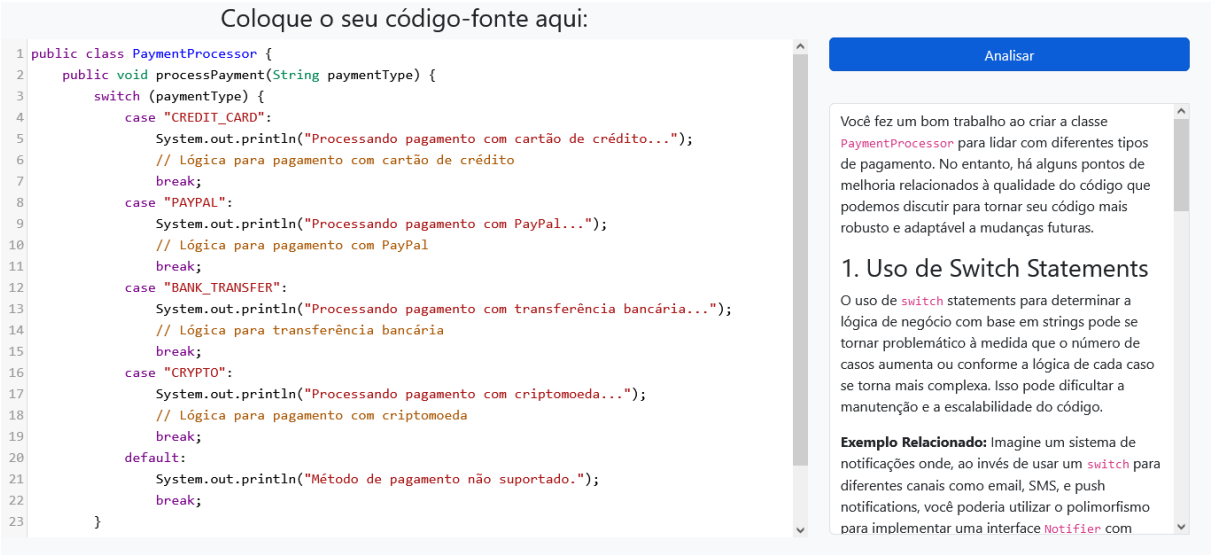
O *software* que implementa a heurística apresentada neste trabalho adota uma estrutura cliente-servidor, sendo a estrutura cliente composta pela extensão do VSCode

[1] https://code.visualstudio.com/

e alternativamente pelo cliente web, e a estrutura servidor composta pela API e pelo banco de dados, conforme detalhamento da Figura 8. Além dessas duas camadas, há uma camada extra composta pelos modelos de IA, fornecidos pela OpenAI como um serviço de sua plataforma. Os detalhes sobre o funcionamento dos LLM de arquitetura transformer, que enquadram os modelos de IA desta ferramenta, são discutidos em detalhe na seção 4.2.1.

4.1.1 Cliente web

Nesta abordagem, a interação do usuário com a ferramenta é realizada através de uma página web que encapsula a interface de utilização. Esta interface, desenvolvida com HTML, CSS e JavaScript e apresentada visualmente na Figura 9, permite que o usuário adicione seu código-fonte e o submeta ao servidor, que executa as tarefas necessárias e apresenta a resposta de *feedback* na área lateral.




```
1  public class PaymentProcessor {
2      public void processPayment(String paymentType) {
3          switch (paymentType) {
4              case "CREDIT_CARD":
5                  System.out.println("Processando pagamento com cartão de crédito...");
6                  // Lógica para pagamento com cartão de crédito
7                  break;
8              case "PAYPAL":
9                  System.out.println("Processando pagamento com PayPal...");
10                 // Lógica para pagamento com PayPal
11                 break;
12             case "BANK_TRANSFER":
13                 System.out.println("Processando pagamento com transferência bancária...");
14                 // Lógica para transferência bancária
15                 break;
16             case "CRYPTO":
17                 System.out.println("Processando pagamento com criptomoeda...");
18                 // Lógica para pagamento com criptomoeda
19                 break;
20             default:
21                 System.out.println("Método de pagamento não suportado.");
22                 break;
23         }
```

Figura 9 – Interface Web. Fonte: autor.

As respostas retornadas pela API são formatadas em Markdown, que passa por conversão para HTML/CSS através da biblioteca Marked[2], mais adequado para exibição no navegador. A área para adição do código-fonte emprega a biblioteca JavaScript CodeMirror[3], que, de forma automática, formata a exibição do código inserido, simulando uma IDE. A coleta do código adicionado nesta área, a construção da requisição HTTP, a submissão, o recebimento e a exibição das respostas são realizados via JavaScript, tornando a interface do usuário uma aplicação de página única. Sendo estas tarefas simples, apenas a biblioteca JavaScript Axios[4] foi utilizada, sendo responsável pelas requisições HTTP à API.

[2] https://marked.js.org/
[3] https://codemirror.net/
[4] https://axios-http.com

4.1.2 Extensão do Visual Studio Code

De forma similar ao cliente web, a extensão do Visual Studio Code também encapsula a comunicação com a API CodeBuddy, porém apresenta características próprias sobre a forma de interação do usuário e apresentação do retorno. Enquanto o cliente web exige que o cliente insira o código manualmente, os recursos de extensão da IDE Visual Studio Code permitem uma abordagem mais direta: o usuário pode submeter o código de uma classe diretamente através do menu de contexto. Ao clicar com o botão direito do mouse sobre um arquivo Java no Explorer, o menu apresentado na Figura 10 é exibido com a opção "Analisar com CodeBuddy"entre as opções padrões do menu. O mesmo item de menu é acessível através de clique com o botão direito no Editor.

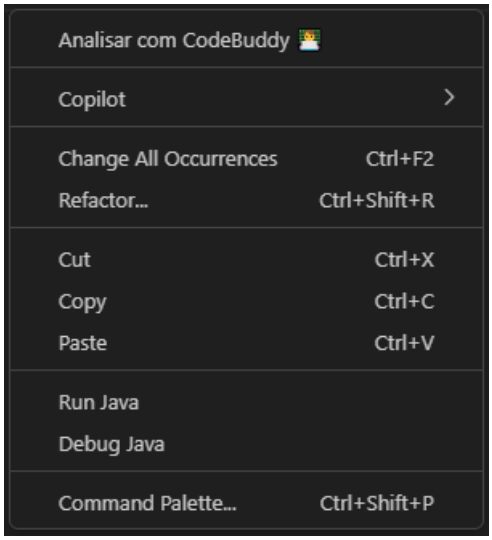


Figura 10 – Menu de contexto do Visual Studio Code exibindo a opção de uso da ferramenta. Fonte: autor.

A exibição do *feedback* retornado pela IA na extensão difere em sua forma daquela apresentada na interface web. Após o usuário iniciar o comando pelo menu de contexto, uma nova aba é aberta no Editor. Durante o processamento da requisição, uma barra de progresso infinita é exibida, em conformidade com as diretrizes de design de interação e experiência do usuário descritas por Nielsen [67]. Assim que a requisição é concluída, o conteúdo retornado é exibido na mesma aba que mostrava o progresso, conforme apresentado na Figura 11.

A extensão pode ser adicionada a qualquer instalação do Visual Studio Code versão 1.94 (setembro/2024) ou superior, em ambientes Linux, Windows ou Mac. Ela pode ser obtida através do próprio menu de extensões da IDE ou através do Visual Studio Marketplace[5].

[5] https://marketplace.visualstudio.com/items?itemName=brunostrikti.codebuddycsedu

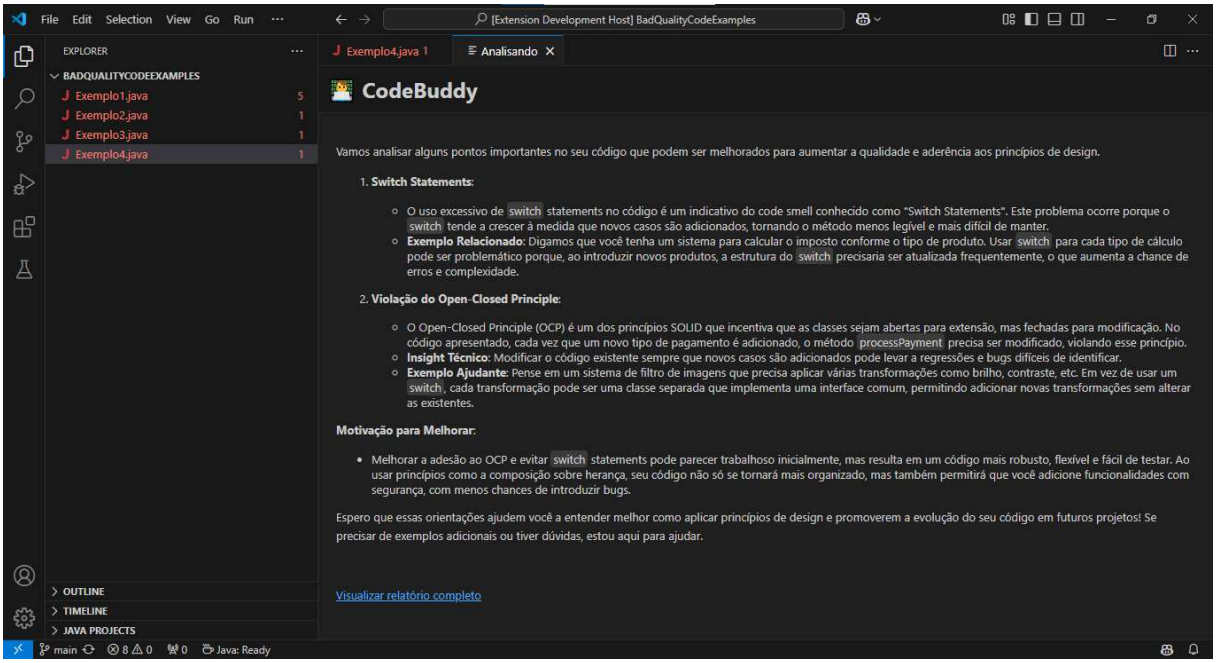


Figura 11 – Extensão do Visual Studio Code exibindo *feedback*. Fonte: autor.

A plataforma Visual Studio Code Extension API possibilita o desenvolvimento de extensões utilizando TypeScript ou JavaScript, sendo esta última a linguagem escolhida para o projeto. A decisão foi motivada pela ampla variedade de bibliotecas disponíveis no ecossistema JavaScript, destacando-se, neste caso, o uso das bibliotecas Axios e Marked, empregadas de maneira semelhante à utilizada no cliente web.

4.1.3 Camada servidor

A camada do servidor centraliza a lógica de negócio da ferramenta. Sua responsabilidade inclui receber o código-fonte enviado pelo cliente, seja por meio da aplicação web ou da extensão VSCode, construir e parametrizar as diversas requisições aos LLMs, processar e analisar as respostas, mapear problemas de qualidade do código em dificuldades de aprendizagem e registrar as interações no banco de dados, conforme a heurística detalhada na Figura 7.

A atuação da camada no servidor é iniciada pelo recebimento da requisição HTTP do cliente, que carrega o código-fonte a ser analisado. Segue-se então uma série de procedimentos que preparam a primeira requisição ao LLM, que será responsável por identificar os problemas de qualidade de código.

A primeira requisição é construída contendo o código-fonte do estudante, instruções de atuação, uma função de referência para a saída estruturada e instruções que caracterizam cada um dos problemas de código que devem ser analisados de forma que o LLM tenha referências. A requisição é construída com técnicas de engenharia de *prompt* [48, 47] e possui a seguinte estrutura:

- Diretrizes gerais de comportamento para o modelo (*role: system*): "Você é um professor que analisa o código-fonte dos seus alunos e gera um ‘code_problems_report‘, marcando ‘true‘ para os problemas identificados no trecho fornecido e ‘false‘ para aqueles não presentes no trecho fornecido.".
- Função de referência (*functions*): A estrutura JSON que determina a estrutura do ‘code_problems_report‘ que deve ser construído pelo modelo. Esta função foi elaborada segundo a especificação da OpenAI API [68].
- Dados do usuário (**role: user**): O código-fonte original e a solicitação de não refatorar o código diretamente.

A requisição é então enviada e o retorno coletado. A Seção 4.2 detalha o funcionamento do modelo de IA. Espera-se que o retorno do LLM contenha um documento JSON que obedeça a estrutura enviada como referência.

A partir do documento JSON que contém os problemas de código identificados, a ferramenta então inicia o mapeamento das dificuldades de aprendizagem relacionadas. A referência para este mapeamento encontra-se na Seção 3.1. Este procedimento produz um novo documento JSON, agora com valores booleanos para as dificuldades de aprendizagem.

Obtida a lista de dificuldades inicia-se a etapa de geração de *feedback*, em que o objetivo é obter duas respostas de *feedback*: a primeira, mais direta e destinada a exibição imediata ao usuário; e a segunda, mais completa e com reflexões destinadas a apoiar o processo educativo, apresentada ao usuário como um documento anexo. Ambas as mensagens de *feedback* são geradas pelo mesmo LLM GPT-4o, sendo diferente a parametrização de cada um dos inputs.

Para a requisição do *feedback* rápido são enviados ao modelo os seguintes parâmetros:

- Diretrizes gerais de comportamento para o modelo (*role: system*): "Você é um professor que analisa o código-fonte dos seus alunos, identifica problemas de qualidade de código e gera um *feedback* educacional. Certifique-se de não refatorar diretamente o código-fonte fornecido, mas você pode apresentar exemplos relacionados. Seu *feedback* deve conter detalhes técnicos de cada problema presente, insights técnicos e exemplos relacionados, e ajudar na motivação do estudante".
- Dados de interações anteriores (*role: assistant*): O documento com os problemas de qualidade e o mapeamento de dificuldades.
- Dados do usuário (**role: user**): O código-fonte original e a solicitação de não refatorar o código diretamente.

Para a requisição do *feedback* completo são enviados ao modelo os seguintes parâmetros:

- Diretrizes gerais de comportamento para o modelo (*role: system*): "Elabore um detalhado texto de orientação educacional para estudantes de programação orientada a objetos que oriente e explique, com exemplos java, como superar determinadas dificuldades de aprendizagem, sem refatorar o código original".
- Dados de interações anteriores (*role: assistant*): O documento com os problemas de qualidade e o código-fonte original.
- Dados do usuário (**role: user**): O documento que consta as dificuldades de aprendizagem identificadas.

O retorno ao usuário é construído com o *feedback* rápido. O *feedback* completo é convertido para HTML/CSS no servidor pela biblioteca Marked e armazenado, sendo retornado ao usuário apenas seu identificador para visualização posterior.

Após a obtenção dos dois *feedbacks* em linguagem natural produzidos pelo LLM, é construído um documento contendo todos os dados da interação. Este documento é armazenado em um banco de dados MongoDB para uso futuro e segue a estrutura apresentada no trecho a seguir:

```
{
    "code": "public class Calculadora { public double
        calcular(double a, double...",
    "timestamp": "2024-12-12T16:53:11.653Z",
    "provider": "web",
    "client_ip": "::1",
    "checks":
    {
        "data_class_smell": false,
        "large_class_smell": false,
        "lazy_class_smell": false,
        "open_close_principle_violation": true,
        "speculative_generality_smell": false,
        ...
    },
    "difficulties":
    {
        "classes_difficulties": true,
```

```
        "method_difficulties": true,
        "object_orientation_difficulties": false,
        "object_relations_difficulties": false,
        "polymorphism_overloading_difficulties": true,
        "encapsulation_difficulties": false
    },
    "feedback_checks": "Vamos analisar o codigo da sua classe
        Calculadora e discutir alguns...",
    "feedback_difficulties": "A programacao orientada a objetos
        (POO) e um paradigma fun...",
    "report": " <!DOCTYPE html>
                <html lang="pt-br">
                <head>...",
    "guid": "dab75f59-02b5-4cd2-9232-38a74731d747"
}
```


A estrutura do documento que registra a interação apresentado acima é composto dos seguintes dados:

- ***Code:*** O código enviado para análise na ferramenta.
- ***Timestamp:*** Data e hora da interação, em formato ISO 8601.
- ***Provider:*** O tipo do cliente que realizou a chamada ao assistente (página web ou *plugin* do VSCode).
- ***Client* IP:** O endereço IP do cliente.
- ***Checks:*** Um objeto que contém valores booleanos para os tipos de problemas de código identificados.
- ***Difficulties:*** Um objeto que contém valores booleanos para os problemas de aprendizagem associados aos indicadores de problemas presentes no código.
- ***Feedback checks:*** *Feedback* resumido
- ***Feedback difficulties:*** *Feedback* completo, com exemplos e sugestões de melhorias.
- ***Report:*** Texto do *feedback* completo estilizado em HTML, ideal para impressão.
- **GUID:** Identificador único de 128 bits.

4.2 Modelo de Inteligência Artificial

Os modelos de Inteligência Artificial empregados neste trabalho são versões personalizadas derivadas dos modelos fundacionais o1-mini e GPT-4o. Ambos são modelos de IA disponíveis na plataforma OpenAI [6] e baseados na arquitetura Transformer, projetada para processar e gerar texto ao longo de múltiplas camadas de atenção e *feed-forward*.

O primeiro modelo, baseado no modelo fundacional o1-mini, é responsável pela identificação automática de indicadores de problemas no código e o segundo modelo, baseado no modelo fundacional GPT-4o, pela construção de *feedback* personalizado e contextualizado. Para execução destas tarefas cada modelo foi parametrizado para os procedimentos específicos de detecção e classificação, no caso do primeiro; e geração de *feedback* em linguagem natural, no caso do segundo. A arquitetura transformer dos modelos fundacionais empregados é detalhada a seguir.

4.2.1 A arquitetura Transformer

Apresentado pela primeira vez em 2017 [1], Transformer é uma arquitetura de rede neural que está por trás dos *chatbots* ChatGPT, Gemini [7] e Claude [8]. Ao contrário das antigas Redes Neurais Recorrentes (RNN) que processavam o texto sequencialmente, a arquitetura Transformer utiliza mecanismo de atenção, que processa sequências de texto, seja de código-fonte ou linguagem natural, analisando a relação entre todas as palavras simultaneamente e paralelamente. A característica central da arquitetura é o modelo codificador-decodificador, cujo funcionamento é representado na Figura 12 e discutido nas próximas seções.

4.2.1.1 Codificador

A principal função do codificador é transformar os *tokens* de entrada em representações contextualizadas. Diferentemente dos modelos anteriores que processavam *tokens* independentemente, o codificador Transformer captura o contexto de cada *token* em relação à sequência inteira [69].

Antes de adentrar nas camadas do codificador, o Transformer começa convertendo *tokens* de entrada - que geralmente representam palavras - em vetores (*input embedding*) que carregam o seu significado semântico. Diferente dos RNNs que processam seus *tokens* sequencialmente, os Transformers o fazem de forma paralela, requerendo a adição de codificadores posicionais que fornecem informações sobre a posição de cada *token* da sequência de entrada.

[6] https://platform.openai.com
[7] https://deepmind.google/technologies/gemini/
[8] https://www.anthropic.com/claude

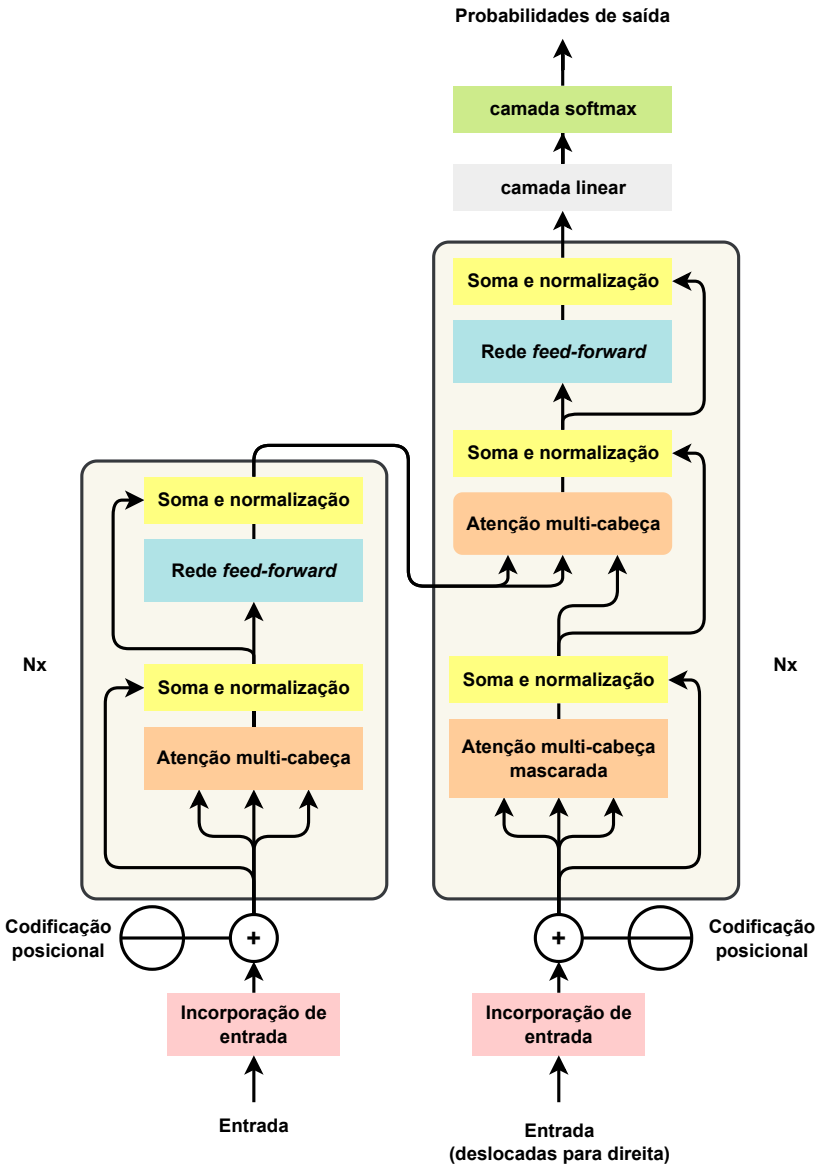


Figura 12 – Arquitetura transformer. Traduzido pelo autor de Vaswani et al.[1]

Cada camada do codificador transforma as sequências de entrada em uma representação contínua e abstrata que encapsula as informações aprendidas de toda a sequência. Cada camada é composta de dois mecanismos principais: o mecanismo de *multi-head attention* e a rede *feed-forward*, cada um deles acompanhado por processos de adição e normalização que visam seu aprofundamento.

O mecanismo de *multi-head attention* relaciona os *tokens* de entrada através de valores de autoatenção, comparando matematicamente o nível de relação de cada um deles com todos os outros *tokens* de entrada. Isso permite que o modelo capture informações contextuais de toda a sequência.

Cada *token* passa então por uma pequena rede *feed-forward*. Essa rede é composta de duas camadas lineares totalmente conectadas, com uma função de ativação ReLU

(*Rectified Linear Unit*) entre elas responsável por ajustar e extrair características mais profundas de cada *token* [70].

A saída da camada final do codificador é um conjunto de vetores, cada um representando a sequência de entrada com valores contextuais. Essa saída é então usada como entrada para o decodificador, orientando-o a prestar atenção às palavras certas no processo de decodificação.

4.2.1.2 Decodificador

A função do decodificador está centrada na criação de sequências de texto. Equipado com um conjunto de camadas semelhante ao codificador, o decodificador possui duas camadas de *multi-head attention*, uma camada de *feed-forward* e incorpora conexões residuais e normalização após cada camada. A parte final do processo do decodificador envolve uma camada linear, que funciona como um classificador, complementada por uma função softmax [71] para calcular as probabilidades de palavras diferentes.

O mecanismo de autoatenção do decodificador difere em relação ao do codificador por não permitir a influência de *tokens* subsequentes na produção da saída. Uma máscara *look-ahead* é aplicada aos *tokens* para garantir que as previsões para uma determinada posição dependam apenas de resultados conhecidos em posições anteriores a ela.

O decodificador opera de forma autorregressiva, iniciando seu processo com um token de início. Em seguida, ele opera com uma lista das saídas anteriores junto das saídas do codificador, retendo assim informações do contexto. Esse processo acontece até que o decodificador gere um token que sinaliza o final da saída, concluindo assim o processo.

4.2.2 Escolha do modelo-base

Para a execução de tarefas específicas, modelos-base de IA generativa precisam de adequações que os alinhem aos objetivos desejados em contextos particulares. Para os objetivos propostos neste trabalho é necessário o envio de determinados parâmetros ao modelo, que são aplicadas através de metadados nas requisições:

1. Ajustar as requisições ao LLM para que tenham parâmetros de referência dos estereótipos de problemas de código cuja identificação é esperada, refinando sua capacidade de detecção.

2. Definir um padrão estruturado para a saída dos indicadores identificados, facilitando sua posterior classificação.

3. Ajustar a saída de seu *feedback* para que seja adequada ao contexto e personalizada para as dificuldades manifestadas.

Tendo em vista os objetivos desta proposta, é necessário definir a estratégia de interação com o modelo de IA: em uma primeira interação, o modelo de IA deve ser capaz de identificar os problemas de código e produzir uma saída estruturada específica (adequações 1 e 2). Em interações subsequentes, um texto de *feedback* em linguagem natural deve ser produzido (adequação 3). Cada interação exige um comportamento específico do modelo, portanto dois modelos com ajustes diferentes são requeridos. Dentre os modelos base pré-treinados disponíveis na plataforma da OpenAI, o modelo o1-mini é o que apresenta melhor desempenho em codificação [2], alcançando 92,4% de precisão na escala HumanEval, contra 90,2% do GPT-4o [72], sendo portanto adotado como modelo-base nos procedimentos de detecção de problemas de código.

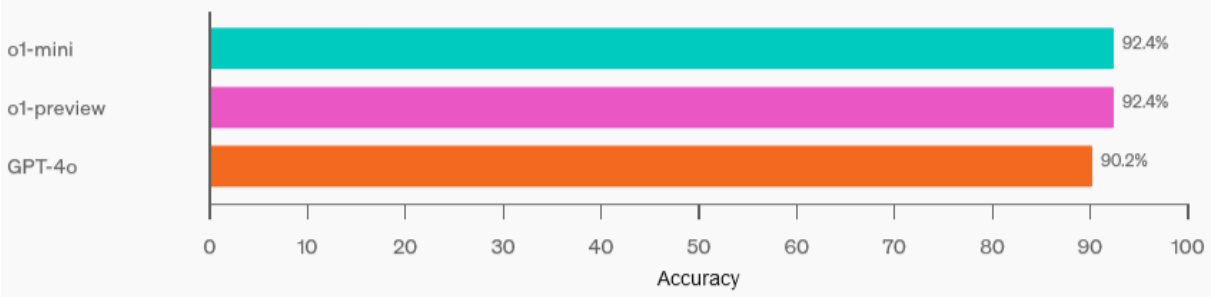


Figura 13 – Desempenho comparativo dos três melhores modelos da plataforma OpenAI na escala HumanEval. Fonte: OpenAI [2].

A tarefa de geração de *feedback* baseia-se na reconhecida capacidade do modelo GPT de gerar texto em linguagem natural [73]. Utilizando as informações sobre os problemas de código identificados pelo primeiro modelo, o código original do estudante e metadados relacionados ao retorno esperado, a IA será acionada para elaborar mensagens de *feedback* com finalidade educacional, seguindo a heurística detalhada na Seção 3.2. Entre os modelos-base disponíveis na plataforma OpenAI, o GPT-4o destaca-se como a opção mais vantajosa para tarefas de geração de texto em linguagem natural, devido à sua menor latência, alta capacidade e boa relação de custo x benefício. O custo por milhão de tokens é de US$ 3,00 para o modelo o1-mini e US$ 2,50 para o modelo GPT-4o.

4.2.2.1 Outras plataformas de IA

A escolha da plataforma da OpenAI para a implementação da ferramenta proposta deve-se a diversos fatores, entre eles a familiaridade do autor com os recursos da API, a maior disponibilidade de recursos no momento do início do projeto e a documentação robusta. No entanto, o uso de outras plataformas de LLM, como o Gemini, o Vortex AI e, principalmente, o recém-lançado DeepSeek, ainda precisa ser explorado no futuro, considerando suas possíveis vantagens em termos de desempenho, custo e flexibilidade para atender as necessidades do projeto.

4.2.3 Parametrização do modelo de IA

Nos modelos de IA da plataforma OpenAI o treinamento pode ser executado em diferentes abordagens: Através do ajuste fino (*fine-tuning*), que permite o ajuste do modelo base utilizando um conjunto personalizado de dados; e pela engenharia de *prompt*, sendo instruções detalhadas, parâmetros de formato, comportamento e contexto incluídos diretamente nos **prompts**.

Em um primeiro momento, foi experimentado o treinamento do modelo GPT-4o-mini através de (*fine-tuning*). Para este treinamento, foi fornecido um conjunto de dados de referência composto por diversos pares de dados contendo um trecho de código com incidências estereotipadas de problemas de código, acompanhado de sua respectiva identificação feita por humanos. Esse conjunto de dados é adicionado à base própria do modelo, com o objetivo de ampliar sua capacidade e precisão na detecção de problemas no código.

Após o processo de treinamento com uma base de 302.436 *tokens*, sua capacidade de detecção de problemas de código foi experimentada, obtendo uma média de acerto de 58,9% na detecção dos problemas presentes nos testes.

A partir de 20 de novembro de 2024, os novos modelos o1-preview e o1-mini foram disponibilizados na plataforma OpenAI. Embora esses modelos não permitam *fine-tuning*, foram treinados com aprendizado por reforço [74] para realizar raciocínios complexos e são acessíveis pelo ponto de acesso (*endpoint*) *Chat Completions*. Experimentos realizados com o modelo o1-mini, parametrizado por meio de engenharia de *prompt* e utilizando o mesmo conjunto de dados previamente testado, apresentaram uma taxa média de detecção de 97,6%, evidenciando o enorme potencial desse modelo na identificação de problemas em código. A Seção 5.1 apresenta em detalhes os testes de precisão realizados nos modelos.

4.2.4 OpenAI API

A API desenvolvida pela OpenAI permite a criação, integração e personalização de modelos de IA baseados nos vários modelos fundacionais da plataforma [68]. Ela permite a integração de aplicações através de chamadas de transferência de estado representacional (REST, *Representational State Transfer*). Diversos *endpoints* da API podem ser integrados:

- *Audio*: para transposição de texto em voz e voz em texto.
- *Batch*: para processamentos diversos em larga escala.
- *Chat completions*: para interações textuais.
- *Embeddings*: para análise de relação entre vetores.

- *Images*: para processamento e geração de imagens.
- *Moderations*: para análise do potencial prejudicial de textos e imagens.

Dada a natureza textual do código-fonte e do *feedback* que se deseja produzir, o *endpoint* Chat Completions é o mais adequado dentre os disponíveis na plataforma.

O *endpoint Chat Completions* permite também a adaptação do comportamento dos modelos de IA conforme os requisitos do projeto, customizando suas instruções através de dados adicionais relevantes ao contexto desejado, como detalhes de comportamento, especificação de formato de saída estruturada e exemplos de interações, conforme detalhano na Seção 4.2.3. Um aspecto especialmente útil é a especificação do formato de saída, pois a identificação de padrões específicos no código-fonte fornecido é fundamental para o diagnóstico e *feedback* ao usuário.

As interações são realizadas através de requisições para o *endpoint Chat Completions*, que são chamadas em que o papel de cada ator é delimitado, assim como no exemplo do trecho a seguir. Na primeira chamada, que faz a identificação dos problemas do código, a especificação do papel da IA delimita-o como revisor de código e acompanha um objeto JSON descritivo do formato da saída estruturada esperada. Já nas chamadas seguintes, responsáveis pela produção do *feedback*, especificam os elementos que deverão compor o *feedback* a ser gerado, acompanhado pela lista de problemas identificados anteriormente, do código-fonte do estudante, e do mapa de problemas no código-fonte e dificuldades de aprendizagem apresentado na Seção 3.1.

```
messages: [
    {
        "role": "system",
        "content": [
            {
                "type": "text",
                "text": "Seja o mestre Yoda e responda a conselhos
                    de seu padawan."
            }
        ]
    },
    {
        "role": "user",
        "content": [
            {
                "type": "text",
                "text": "Mestre, como fazer boas escolhas?"
```

```
            }
        ]
    },
    {
        "role": "assistant",
        "content": [
            {
                "type": "text",
                "text": "Distinguir o certo do errado voce vai
                    quando em paz você estiver."
            }
        ]
    }
]
```

4.2.5 Saída estruturada

A definição do padrão estruturado de saída é realizada com o objetivo de garantir que as informações produzidas pelo modelo sejam facilmente processadas. Este ajuste normatiza a saída para que o modelo devolva listas de objetos e não texto em linguagem natural, facilitando a transformação dos valores JSON e simplificando a manipulação dos dados. No caso do primeiro modelo, a especificação determina que a saída deve conter informações booleanas sobre a incidência ou não de cada um dos problemas classificados no código fonte fornecido e analisado. No retorno, cada código de problema é acompanhado de um valor booleano apontando a incidência do problema relacionado. O trecho de código apresentado na Seção 3.2.2 apresenta um exemplo de saída retornado pelo modelo de IA para um trecho de código com problemas de *Code Smell Switch Statements*.

4.2.6 Elaboração do *feedback* pelo modelo

A produção de *feedback* explora a capacidade de produção de texto em linguagem natural do modelo de IA com a finalidade de produzir um texto formativo ao usuário. Tendo como base informações dos indicadores de problemas mapeados em dificuldades de aprendizagem e metadados de contexto, a IA analisa e fornece informações valiosas para auxiliar o estudante em sua aprendizagem.

Para a elaboração do *feedback*, as informações são fornecidas diretamente ao modelo em uma segunda e terceira chamadas de API, utilizando técnicas de engenharia de *prompt*. O *prompt* submetido ao modelo de IA para a finalidade de *feedback* carrega diversas informações registradas, com o objetivo de instruir o modelo a personalizar a resposta. Além das orientações de personalização, o *prompt* contém o código-fonte original, metada-

dos, os indicadores identificados e as dificuldades mapeadas. Após a submissão do *prompt* para a API do modelo, a resposta é processada e retornada, sendo então registrada e exibida ao usuário.

5 AVALIAÇÃO

Este capítulo dedica-se à avaliação multidimensional da heurística e da ferramenta que a aplica. A avaliação busca compreender não apenas a performance técnica da solução proposta, mas também sua potencial contribuição no processo educativo e sua adequação às necessidades e expectativas dos usuários.

Nesta avaliação multidimensional foram avaliadas:

- A capacidade do LLM detectar problemas de qualidade de código;
- A percepção de especialistas em ciência da computação sobre a contribuição educacional da ferramenta;
- A percepção de especialistas em ciência da computação sobre a aplicabilidade da ferramenta em atividades educacionais;
- A aceitação da ferramenta pelos usuários.

5.1 Avaliação dos problemas de código

Com o objetivo de avaliar a capacidade do modelo de IA em identificar problemas de qualidade de código, foram realizados diversos experimentos de análise de código utilizando o LLM o1-mini.

Os experimentos consistiram na submissão, via API, de diversos trechos de código cuja incidência de problemas de qualidade era previamente conhecida. A resposta do LLM foi analisada para verificar se os problemas esperados eram corretamente identificados. Em todos os casos, as submissões foram acompanhadas de parâmetros de *prompt*, conforme a descrição da primeira requisição apresentada na Seção 4.1.3.

Para avaliação do desempenho do modelo foram utilizadas três populares métricas utilizadas na avaliação de modelos de linguagem: Precisão, *Recall*, *F1 Score* e Acurácia.

A precisão é obtida pela divisão da quantidade de problemas que o modelo apontou e estavam corretos, divididos pela quantidade de problemas que o modelo apontou, corretos ou não. Esta métrica aponta se as respostas do modelo atendem ao esperado.

$$\text{Precisão} = \frac{PositivosVerdadeiros}{PositivosVerdadeiros + FalsosPositivos}$$

O *Recall* mede a proporção de respostas corretas fornecidas pelo modelo em relação ao total de respostas que deveriam ter sido dadas. Essa métrica também é conhecida

como taxa de detecção: de todos os exemplos que o modelo poderia detectar, quantos ele realmente conseguiu apontar.

$$\text{Recall} = \frac{PositivosVerdadeiros}{PositivosVerdadeiros + NegativosFalsos}$$

O *F1 score* é a média harmônica entre a precisão e o recall. Ele é usado para encontrar um equilíbrio entre a Precisão e o *Recall*, sendo particularmente útil quando é importante considerar tanto os erros de omissão (baixa cobertura ou baixo recall) quanto os erros de comissão (respostas incorretas ou baixa precisão).

$$F_1 = 2 \times \frac{\text{Precisão} \times \text{Recall}}{\text{Precisão} + \text{Recall}}$$

A acurácia mede a proporção de previsões corretas em relação ao total de exemplos avaliados, indicando quantas vezes o modelo acertou, considerando tanto os positivos quanto os negativos.

$$\text{Acuracia} = \frac{PositivosVerd + NegativosVerd}{PositivosVerd + NegativosVerd + PositivosFalsos + NegativosFalsos}$$

Dois conjuntos de amostras foram utilizados nestes experimentos. O primeiro conjunto consiste de 87 amostras de trechos de código com problemas de qualidade inseridos propositalmente, desenvolvidos pelo autor com base nas observações de Lanza & Marinescu[75], Fowler [11] e Pereira et al. [76]. O segundo conjunto de dados contém três projetos funcionais desenvolvidos em Java e obtidos em repositórios públicos do GitHub. Todas as amostras de código-fonte utilizadas nos experimentos, estão disponíveis publicamente no GitHub[1].

No primeiro conjunto de amostras, cada um dos indicadores de problemas listados na Seção 2.3 estava presente em pelo menos três classes distintas. O conjunto de dados foi então analisado pelo modelo o1-mini, e os resultados para Precisão, *Recall*, *F1 Score* e Acurácia são apresentados na Tabela 1. A quantidade e tipos de problemas em cada amostra são detalhados no Apêndice B.

Problema de código	**Precisão**	**Recall**	**F1 Score**	**Acurácia**
Alternative classes with different interfaces	1,00	0,50	0,66	0,98
Comments	0,75	1,00	0,85	0,99
Data class	0,88	1,00	0,93	0,97
Data clumps	0,71	1,00	0,83	0,98

[1] https://github.com/brunostrik/TreinamentoModeloEducacional

Dead code	0,50	0,33	0,40	0,97
Divergent change	1,00	0,33	0,50	0,98
Duplicate code	0,80	1,00	0,88	0,99
Feature envy	1,00	0,33	0,50	0,98
Inappropriate intimacy	0,50	0,33	0,40	0,97
Large class	1,00	1,00	1,00	1,00
Lazy class	0,75	0,60	0,66	0,97
Long method	1,00	0,60	0,75	0,98
Long parameter list	0,77	1,00	0,87	0,98
Message chain	1,00	0,33	0,50	0,98
Middle man	0,80	1,00	0,88	0,97
Parallel inheritance hierarchies	1,00	0,33	0,50	0,98
Primitive obsession	1,00	0,66	0,80	0,99
Refused bequest	1,00	0,66	0,80	0,99
Shotgun surgery	1,00	1,00	1,00	1,00
Speculative generality	1,00	1,00	1,00	1,00
Switch statements	1,00	1,00	1,00	1,00
Temporary fields	1,00	1,00	1,00	1,00
Violações da Lei de Demeter	1,00	1,00	1,00	1,00
Violações do Tell Don't Ask	1,00	0,33	0,50	0,98
Violações do Single Responsability Principle	0,92	0,92	0,92	0,98
Violações do Open/Closed Principle	1,00	1,00	1,00	1,00
Violações do Liskov Substitution Principle	1,00	1,00	1,00	1,00
Violações do Interface Segregation Principle	1,00	1,00	1,00	1,00
Violações Dependency Inversion Principle	1,00	0,33	0,50	0,98
Total	**0,91**	**0,75**	**0,81**	**0,98**

Tabela 1 – Resultado dos testes de Precisão, *Recall*, *F1 Score* e Acurácia com o primeiro conjunto de dados. Fonte: autor.

O segundo conjunto de dados foi coletado a partir de projetos públicos do GitHub. Para a busca e seleção dos projetos, utilizou-se a palavra-chave "projeto"e aplicou-se um filtro para a linguagem Java. Dentre os 50 primeiros projetos listados, foram selecionados aleatoriamente três, desde que contivessem até 10 classes, excluídas da contagem classes de teste e interface de usuário. De cada projeto, cinco classes foram selecionadas para o

experimento.

As classes selecionadas foram analisadas pela ferramenta JDeodorant[2], reconhecida por sua capacidade de identificar problemas de qualidade de código [77] e pelo autor. Em seguida foram analisadas pelo LLM o1-mini e os problemas de qualidade identificados foram comparados. Os resultados de Precisão, *Recall*, *F1 Score* e Acurácia são apresentados na Tabela 2. Os problemas de código que não estavam presentes e não foram identificados foram suprimidos. A quantidade e tipos de problemas em cada classe são detalhados no Apêndice C.

Problema de código	**Precisão**	**Recall**	**F1 Score**	**Acurácia**
Comments	0,91	0,91	0,91	0,86
Data class	1,00	1,00	1,00	1,00
Data clumps	1,00	1,00	1,00	1,00
Dead code	1,00	1,00	1,00	1,00
Duplicate code	1,00	0,75	0,86	0,93
Feature envy	0,00	1,00	0,00	0,93
Inappropriate intimacy	0,00	0,00	1,00	0,86
Large class	1,00	1,00	1,00	1,00
Lazy class	1,00	1,00	1,00	1,00
Long method	1,00	1,00	1,00	1,00
Long parameter list	1,00	1,00	1,00	1,00
Middle man	0,00	1,00	0,00	0,93
Parallel inheritance hierarchies	1,00	0,00	0,00	0,86
Primitive obsession	0,50	1,00	0,67	0,93
Shotgun surgery	1,00	1,00	1,00	1,00
Violações do Tell Don't Ask	0,00	1,00	0,00	0,93
Violações do Single Responsability Principle	0,80	0,80	0,80	0,86
Violações Dependency Inversion Principle	1,00	0,00	0,00	0,93
Total	**0,73**	**0,80**	**0,68**	**0,94**

Tabela 2 – Resultado dos testes de Precisão, *Recall*, *F1 Score* e Acurácia com o segundo conjunto de dados. Fonte: autor.

A média obtida entre os dois projetos foi de 0,82 para Precisão, 0,77 para *Recall*, 0,75 para *F1 Score* e 0,96 para Acurácia. Não há na literatura delimitações fixas para interpretação destes resultados, sendo orientada a análise caso a caso [78]. Geral-

[2] https://users.encs.concordia.ca/ñikolaos/jdeodorant/

mente melhores indicadores de Precisão são requeridos em projetos em que o custo de um falso positivo é alto, como por exemplo em casos jurídicos e neste projeto. Já indicadores mais altos de *Recall* são importantes quando a identificação de todos os casos positivos é crucial mesmo que ocorram eventuais falsos positivos, como por exemplo na detecção de doenças graves. Um **F1 Score** elevado é requerido quando é necessário um equilíbrio entre os dois indicadores. Um valor elevado para Acurácia é desejável em todas as situações. Considerado o contexto, o desempenho do modelo o1-mini, avaliado tanto em experimentos direcionados à detecção de problemas específicos quanto em experimentos com código-fonte de projetos reais, pode ser considerado bom.

A segunda amostra, composta pelos projetos obtidos no GitHub, obteve um *score* de desempenho inferior em relação à primeira amostra. Destaca-se que a segunda amostra é composta por apenas 15 classes e alguns dos problemas de código incidiram em poucas ocasiões, indicando resultados mais extremos do que seriam obtidos em um conjunto de dados mais extenso e variado.

Alguns problemas de código são de difícil detecção pelo LLM somente a partir da observação de classes isoladas, como apontado por Pomian et al. [79], e obtiveram desempenho abaixo da média. Um exemplo desta dificuldade é o *Code Smell Parallel Inheritance Hierarchies*, que exige uma compreensão de múltiplas classes para caracterização do problema, o que ocorreu com a segunda amostra.

Liu et al. [80] aponta que prompts que incluem classes de contexto além da própria classe analisada apresentam melhores resultados na indicação de problemas de código, consoante ao melhor desempenho das amostras submetidas que continham mútiplas classes em um único arquivo.

Além das ressalvas acima destacadas, foi observada certa inconsistência no modelo na detecção de *Code Smells Comments* e *Data Class*, especialmente na primeira amostra. O *Code Smell Comments* apresentou um número elevado de falsos positivos, sendo por vezes trechos curtos apontados como problemáticos dependendo do conteúdo dos comentários: Comentários descritivos eram corretamente apontados como problemáticos, já comentários desconexos e sem sentido não eram identificados.

Comportamento semelhante também foi observado no caso do *Code Smell Data Class*, quando o apontamento de classes não era claro, tendo classes parecidas ora apontadas como problemáticas, ora não. Classes cujos nomes dos atributos indicassem endereço, nome e documento de identificação geralmente eram sinalizadas, porém se os atributos fossem menos óbvios o problema não era percebido.

Esta identificação pela clareza semântica dos nomes dos atributos também era preponderante para o apontamento do *Code Smell Data Clumps*. Classes que repetidamente encapsulavam dados triviais como cadastros de pessoas, vendas e produtos eram facil-

mente identificados, porém faltava precisão em situações mais peculiares como dimensões de sólidos complexos.

De modo geral e avaliando as necessidades do projeto, o desempenho na identificação dos problemas no código pode ser considerado promissor. Avanços futuros nos modelos de IA e a esperada possibilidade de treinamento de modelos mais avançados tendem a permitir um desempenho melhorado. A primeira amostra também obteve desempenho melhor em comparação à segunda, em que o formato de classes isoladas e tamanho reduzido impactou na produção de indicadores inferiores aos da primeira amostra.

5.2 Avaliação da ferramenta por especialistas

Com o objetivo de analisar o uso da ferramenta sob a perspetiva do docente, foi conduzida uma pesquisa do tipo *survey*, conduzida durante três semanas e que contou com a participação de 10 professores do ensino superior na área de computação. Os participantes foram convidados a explorar livremente a ferramenta e opinar sobre sua experiência.

A pesquisa foi estruturada de forma a avaliar diferentes aspectos, agrupados em tópicos com diferentes objetivos:

1. **Perfil dos participantes:** este tópico agrupa questões sobre a experiência profissional dos participantes;
2. **Identificação de problemas no código:** neste tópico é coletada a percepção dos participantes sobre a capacidade da ferramenta identificar problemas de qualidade de código;
3. ***Feedback:*** aqui são registradas as observações dos participantes acerca do texto de *feedback* produzido pela ferramenta;
4. **Ensino e aprendizagem:** neste tópico os participantes contribuem com sua percepção sobre a contribuição educacional da ferramenta, tendo como ponto de partida as percepções individuais sobre a incidência de problemas de qualidade de código em sala de aula;
5. **Uso da ferramenta:** aqui os professores contribuem relatando de que forma exploraram a ferramenta;
6. **Usabilidade:** esta seção aplica um questionário SUS (*System Usability Scale*) que coleta dados sobre a usabilidade da ferramenta;
7. **Oportunidades de aplicação:** aqui os professores detalham como imaginam a aplicação da ferramenta em atividades educacionais;

Os participantes responderam a 37 questões, sendo a maioria do tipo Likert complementada por questões abertas de redação livre. A Tabela 3 apresenta a organização do questionário aplicado. O formulário de pesquisa completo está disponível no Apêndice D.

Seção	**Likert/Seleção**	**Livre**	**Mult. Escolha**
1. Perfil dos participantes	3	3	2
2. Identificação de problemas no código	1	1	0
3. *Feedback*	3	1	0
4. Ensino e aprendizagem	4	1	0
5. Uso da ferramenta	0	1	22
6. Usabilidade	10	0	0
7. Oportunidades de aplicação	1	2	2

Tabela 3 – Organização das seções do questionário aplicado. Fonte: autor.

5.2.1 Perfil dos participantes

Todos os participantes convidados são docentes em cursos de nível superior classificados pela CAPES dentro da área 10300007 (Ciência da Computação), distribuídos entre sete instituições de ensino superior federais, estaduais, privadas e do terceiro setor. 30% dos participantes possuem título de doutor, 60% são mestres e 10% especialistas. O tempo médio de experiência na docência é de 12,7 anos. A maior parte dos participantes é formado em Ciência da Computação 70%, seguido por Sistemas de Informação com 20%, e Engenharia da Computação com 10%. Além da experiência acadêmica, a média de experiência profissional na área de computação dos participantes é de 7,7 anos. A Figura 14 detalha a área e nível de formação dos participantes.

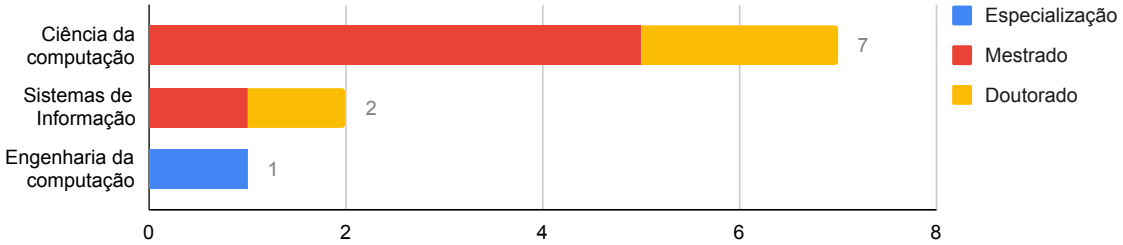


Figura 14 – Perfil de formação dos participantes. Fonte: autor.

Quando perguntados sobre sua experiência com o tema da programação orientada a objetos, 40% dos participantes se consideram especialistas, 30% afirmam ter conhecimento avançado, 20% intermediário e 10% pouco conhecimento. Sobre príncipios de qualidade de código 30% dos participantes afirmam ter conhecimento avançado, outros 30% conhecimento intermediário, 30% afirmam conhecer pouco do assunto e 10% não conhecem. A Figura 15 detalha a relação entre os níveis de conhecimento de programação orientada a objetos e princípios de qualidade de código, sendo que os que consideram ter maiores níveis em qualidade de código também o consideram em POO.

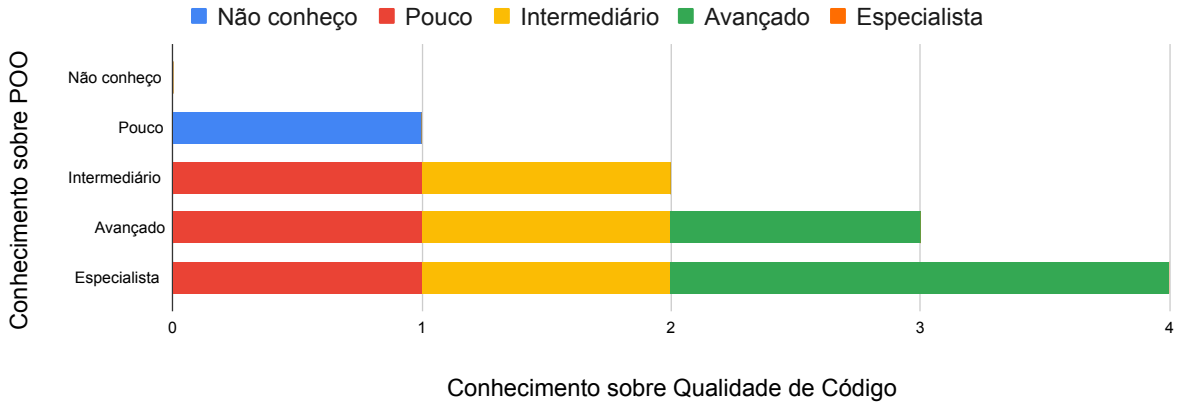


Figura 15 – Conhecimento dos participantes em POO e qualidade de código. Fonte: autor.

A atuação docente em atividades de ensino também foi registrada, com predomínio de disciplinas de programação. A Figura 16 detalha os componentes curriculares ministrados pelos participantes.

Figura 16 – Resultados das questões sobre a atuação dos participantes como docente em disciplinas ou componentes curriculares de computação. Fonte: autor.

Os resultados sobre o perfil dos participantes indicam uma amostra profissional, experiente e com sólida formação, adequado ao escopo da pesquisa.

5.2.2 Identificação de problemas no código

Esta seção da pesquisa busca obter dos participantes percepções sobre a capacidade da IA identificar problemas de qualidade no código. Duas questões buscam validar se os participantes consideram a ferramenta capaz de identificar problemas de qualidade no código. A primeira, respondida em escala Likert [81], questiona diretamente a impressão dos participantes, conforme a Figura 17

Na segunda questão da seção os participantes tiveram a oportunidade de registrar

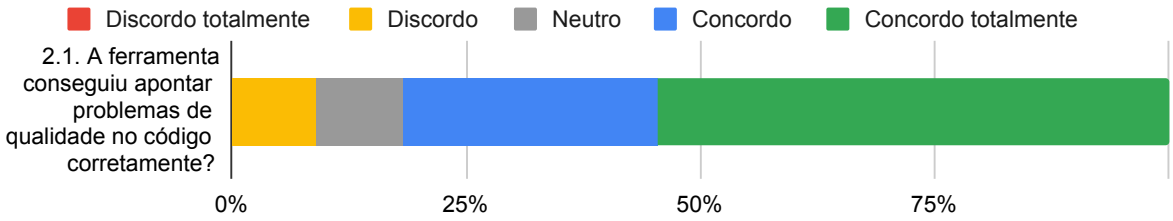


Figura 17 – Fonte: autor.

comentários sobre a capacidade da ferramenta em identificar problemas de qualidade no código. Dois comentários foram inseridos: um deles apontou que a ferramenta foi capaz de identificar problemas que o próprio especialista não havia notado, enquanto o outro apontou que a ferramenta demonstrou dificuldades em lidar com soluções que implementam soluções computacionais complexas, como padrões de projeto.

As observações dos especialistas ressaltam as limitações dos modelos de linguagem de grande porte (LLMs). Embora avançados e, por vezes, surpreendentes, tais modelos permanecem restritos à sua base de treinamento e ao escopo da classe apresentada, demonstrando dificuldades na compreensão profunda de características de projetos que extrapolem o código fornecido. No caso de padrões de projeto, em especial, a limitação do escopo de compreensão do LLM à classe submetida pode ocasionar o apontamento de falsos positivos. Um exemplo são as classes POJO (*Plain Old Java Object*, ou Objeto Java Comum) na qual pode ser apontado o *code smell data class*.

5.2.3 Feedback

Os participantes responderam quatro perguntas sobre o *feedback* produzido pela ferramenta, sendo três respondidas com base em uma escala Likert e uma aberta. As perguntas foram elaboradas com o objetivo de avaliar a opinião dos especialistas sobre o *feedback* produzido pela ferramenta. A Figura 18 apresenta os resultados das três perguntas obejtivas.

As questões 3.1 e 3.2 apresentaram resultados amplamente positivos. Já a questão 3.3 apresentou um número elevado de discordâncias, algumas delas comentadas na questão aberta 3.4, que busca obter comentários dos especialistas sobre os *feedbacks* produzidos na experimentação da ferramenta.

Diversas respostas indicaram que o texto é bem estruturado e segue uma ordem lógica que favorece a compreensão do estudante. No entanto, a inconsistência entre as respostas foi apontada como um aspecto que requer atenção, uma vez que foram observados diferentes tipos de respostas em interações semelhantes.

Entre os especialistas que discordaram na questão 3.3, os comentários fornecem indícios sobre a motivação dessa discordância. Um participante afirmou que o nível de

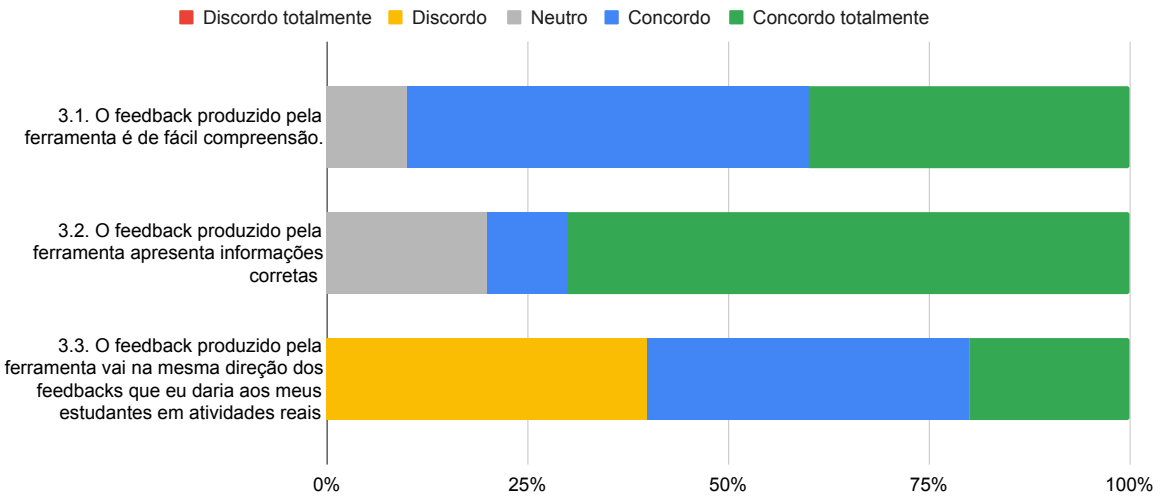


Figura 18 – Resultados das questões sobre o *feedback*. Fonte: autor.

aprofundamento das análises pode gerar confusão em estudantes iniciantes, pois o texto de *feedback* pode conter conceitos avançados ainda não introduzidos e que "quando se está estudando orientação a objetos, principalmente no início, é necessário sacrificar certos conceitos". Outro argumentou que o texto não considera a trajetória de aprendizagem do estudante: "O *feedback* não leva em conta o histórico do estudante, é momentâneo. É preciso conhecer mais a trajetória do estudante para um bom *feedback*". Essa crítica foi complementada por um terceiro comentário: "O ideal seria considerar a trajetória do estudante, mas isso nenhuma IA é capaz de fazer, precisa ser o professor".

Essas últimas observações, entretanto, envolvem um julgamento que exige capacidade humana para sua consideração, o que sugere que a questão pode não ter sido totalmente eficaz em comunicar seu objetivo. Além disso, não foi possível estabelecer uma relação entre as discordâncias e o nível de experiência profissional ou acadêmica dos participantes, tampouco com seu conhecimento sobre POO e conceitos de qualidade de código.

Considerando todas as questões da seção é possível observar que o *feedback* produzido pela ferramenta apresenta informações corretas e é de fácil compreensão. A redação do texto, entretanto, ainda que avaliada positivamente em mais da metade dos casos ainda carece de melhorias, cujo avanço no desenvolvimento dos modelos de linguagem poderão sanar. Alguns apontamentos indicaram necessidades além das capacidades de qualquer modelo de inteligência artificial, como a necessidade de ponderar sobre o histórico, contexto, e características individuais de cada estudante. Estas observações são importantes para destacar que por mais avançado que uma inteligência artificial possa ser, ele não será capaz de substituir a mediação docente.

5.2.4 Ensino e aprendizagem

Neste tópico os participantes foram perguntados sobre a percepção de utilidade da ferramenta em mitigar a produção de código de baixa qualidade pelos estudantes e contribuir no processo de aprendizagem.

A primeira pergunta busca conhecer a impressão dos docentes sobre a frequência observada de código de baixa qualidade produzido por estudantes, cujo resultado indica sua onipresença. As três perguntas seguintes buscam identificar a percepção sobre a utilidade da ferramenta no processo educativo, principalmente como instrumento de apoio para melhorar a qualidade do código dos estudantes. Estes resultados são apresentados na Figura 19.

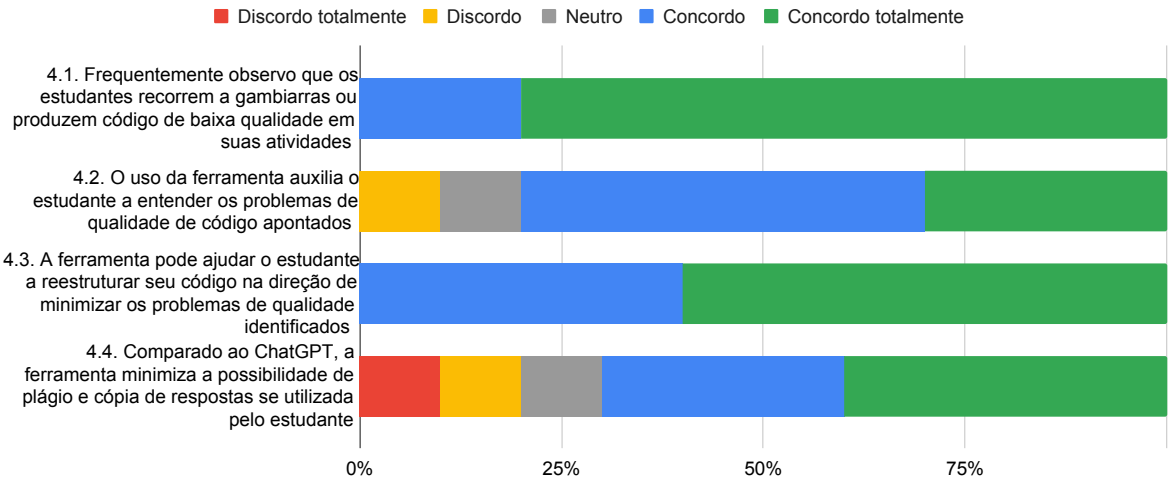


Figura 19 – Resultados das questões sobre ensino e aprendizagem. Fonte: autor.

A última questão da seção é de redação aberta e investiga a percepção dos participantes sobre a contribuição da ferramenta no processo de aprendizagem dos estudantes. Diversos comentários destacam que a ferramenta eleva o nível de abstração, em concordância com as observações de Lau & Guo [41] e Zastudil et al. [46]. Além disso, enfatizam a personalização das orientações fornecidas aos estudantes, alinhando-se às análises de Budhiraja et al. [82], Dai et al. [83] e Prather et al. [84].

Ao analisar todas as questões e contribuições da seção conclui-se por unanimidade que a ferramenta contribui para que o estudante produza código de melhor qualidade. As preocupações com plágio ainda estão presentes, mesmo aplicadas técnicas que evitem a refatoração direta do código pela IA. Há também uma elevada concordância em relação à contribuição para melhoria da compreensão de conceitos relacionados aos problemas qualidade de código.

5.2.5 Uso da ferramenta

Para avaliar os diferentes aspectos relacionados ao uso da ferramenta, foram aplicadas três questões voltadas a investigar a forma que a ferramenta foi utilizada pelos especialistas.

A primeira questão busca identificar quantas interações distintas foram realizadas com a ferramenta, sendo 5,2 a quantidade média com valores que variam entre 2 e 15 interações.

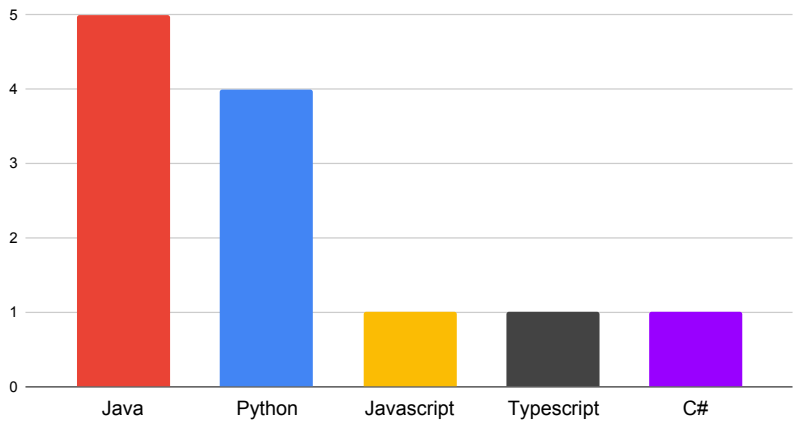


Figura 20 – Linguagens de programação experimentadas. Fonte: autor.

A segunda questão busca conhecer a origem dos código-fonte submetidos para análise da ferramenta. Os resultados indicam que a maior parte (6) dos participantes experimentou a ferramenta somente com códigos próprios, apenas um participante limitou-se aos trechos do repositório de exemplo fornecido e três utilizaram código próprio e também do repositório. Embora a ferramenta tenha sido projetada para trechos de código-fonte em linguagem Java, a experimentação de trechos em outras linguagens de programação foi permitida e observada. Conforme a Figura 20, cinco linguagens de programação diferentes foram experimentadas, sendo Java a linguagem mais utilizada.

5.2.6 Usabilidade

A usabilidade da ferramenta foi avaliada por meio de um questionário baseado na *System Usability Scale* (SUS), um método amplamente utilizado para medir a experiência do usuário em termos de facilidade de uso, eficiência e satisfação [85]. A *System Usability Scale* (SUS) foi desenvolvida por John Brooke em 1986 como uma ferramenta simples e confiável para avaliar a usabilidade de sistemas interativos. Trata-se de um questionário padronizado composto por dez afirmações, alternando entre aspectos positivos e negativos, que os participantes avaliam em uma escala Likert de cinco pontos. O escore final varia de 0 a 100 e fornece uma medida subjetiva da usabilidade percebida, sendo que valores

acima de 68 são geralmente considerados indicativos de uma boa experiência do usuário. Os resultados obtidos foram consolidados e são apresentados na Figura 21.

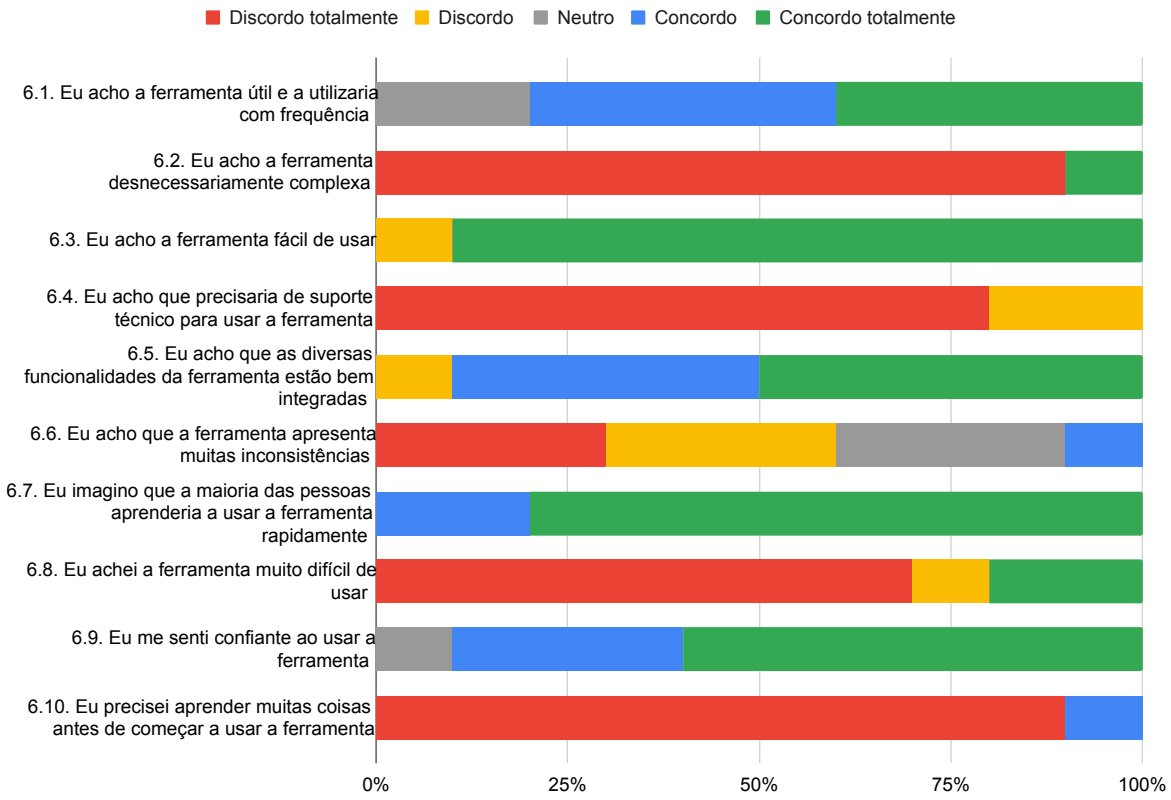


Figura 21 – Resultados das questões sobre usabilidade. Fonte: autor.

O escore final na escala SUS é calculado de acordo com as respostas selecionadas, em que cada uma delas possui um valor de um a cinco:

- Discordo totalmente: 1 ponto
- Discordo: 2 pontos
- Neutro: 3 pontos
- Concordo: 4 pontos
- Concordo totalmente: 5 pontos

As questões de número ímpar são questões positivas, e seu valor é calculado subtraindo 1 ponto do valor da resposta, sendo a_1 os coeficientes fornecidos em sequência e $(i-1)$ o fator multuplicador, baseado na posição:

$$Q = \sum_{i=1}^{n} a_i \times (i-1)$$

Por sua vez, as questões pares são negativas e seu valor é calculado subtraindo o valor da resposta de 5:

$$Q = \sum_{i=1}^{n} a_i \times (5 - i)$$

Isso garante que todas as respostas sejam convertidas para uma escala em que 0 representa a pior experiência e 4 a melhor. Os valores são então somados e multiplicados por 2,5 para obter um escore final entre 0 e 100. Como aqui foram aplicados 10 questionários, cada valor é multiplicado pela quantidade de respostas assinaladas, e o resultado final dividido por 10 para cálculo da média.

$$SUS = \frac{\left(\sum_{i=1}^{10} Q_i\right) \times 2.5}{N}$$

O resultado então é interpretado em uma escala que classifica valores até 50 como inaceitáveis, de 50 a 70 como marginais e acima de 70 como aceitáveis.

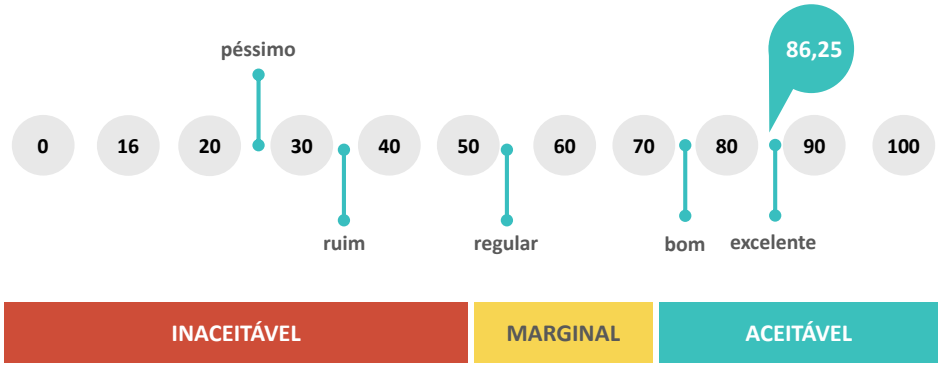


Figura 22 – Níveis na escala SUS

A avaliação da ferramenta resultou em um score de **86,25 pontos**, que de acordo com a escala apresentada na Figura 22 indica que a usabilidade pode ser avaliada como **excelente** na escala SUS.

5.2.7 Oportunidades de aplicação

Nesta seção o objetivo foi avaliar a aplicabilidade da ferramenta em sala de aula, identificando também de que forma sua inserção pode ser realizada de acordo com a impressão dos participantes.

A maior parte dos participantes (80%) é otimista sobre a viabilidade de uso da ferramenta em sala de aula. Os 20% restantes não afirmaram sua inviabilidade e optaram pela neutralidade. A questão seguinte complementa os resultados ao explorar por quem e em que situações seria interessante a aplicação da ferramenta. Nenhum participante

assinalou não vislumbrar aplicações potenciais da ferramenta. As potenciais aplicações assinaladas são:

- Pelos estudantes, como uma ferramenta revisora de suas atividades, assinalada por 9 participantes;
- Pelo docente, para revisão das atividades dos estudantes, assinalada por 7 participantes;
- Pelos estudantes, como ferramenta de apoio durante as atividades, assinalada por 5 participantes;
- Pelo docente, para preparação das atividades de ensino, assinalada por 2 participantes.

As disciplinas que os participantes apontaram viável a aplicação da ferramenta são apresentadas na Figura 23. Nenhum dos participantes assinalou que não considerava útil a aplicação da ferramenta.

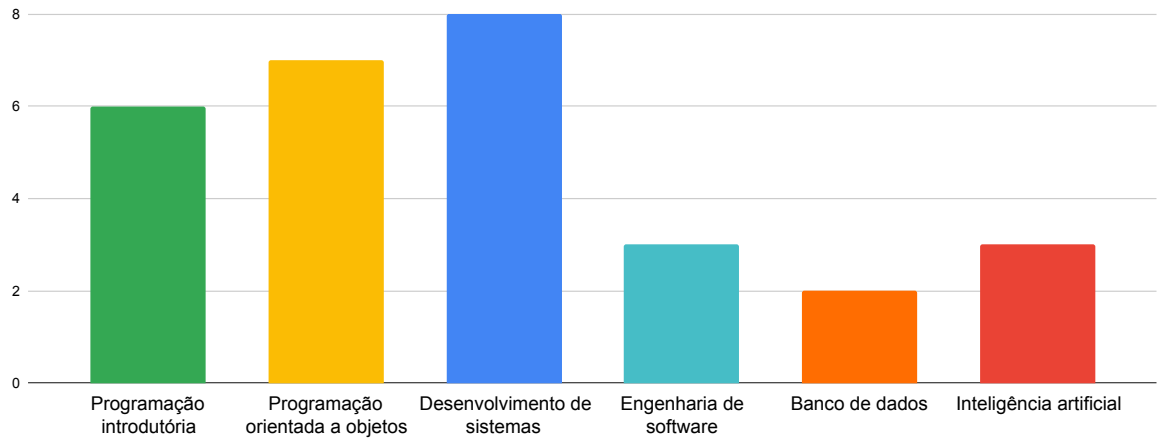


Figura 23 – Respostas da questão 7.3: Em quais disciplinas você considera útil a aplicação da ferramenta? Fonte: autor.

As duas questões seguintes eram de preenchimento livre. A primeira buscava contribuições sobre como a ferramenta poderia ser aprimorada. As sugestões indicam que a ferramenta poderia apresentar em seu *feedback* exemplos melhorados e adequados ao nível de aprendizado do aluno, garantindo o foco do estudante no conceito desejado. A última questão, também de preenchimento livre, solicitava a contribuição dos participantes sobre impressões gerais do uso da ferramenta. As respostas inseridas destacam a capacidade da ferramenta detectar violações de boas práticas e problemas estruturais, porém apontam que a o procedimento de análise é demorado, podendo causar frustração e abandono.

A seção de oportunidades de aplicação apresenta uma concordância da aplicabilidade da ferramenta sobretudo em disciplinas relacionadas à programação de nível avançado. A opinião dos participantes sobre a forma de uso é variada, com equilíbrio na compreensão de uso potencial por estudantes e docentes, com predominância do papel de ferramenta revisora em ambos os casos.

5.3 Ameaças à validade da proposta

A proposta apresentada neste trabalho, embora promissora, enfrenta algumas ameaças à sua validade que devem ser consideradas para uma avaliação mais completa e crítica. Essas ameaças podem ser categorizadas em diferentes dimensões, incluindo a representatividade dos dados, a precisão dos modelos de IA, e a aplicabilidade prática da ferramenta em contextos educacionais reais.

5.3.1 Representatividade dos Dados de Treinamento

Uma das principais ameaças à validade da proposta é a diversidade dos exemplos utilizados para treinar o modelo de IA. Embora o modelo tenha sido treinado com um conjunto de dados que inclui várias classes de código com problemas de qualidade previamente identificados, é possível que esses exemplos não cubram todas as nuances e variações de problemas que podem surgir em diferentes contextos de programação. Isso pode limitar a capacidade do modelo de identificar problemas em códigos que apresentem características não presentes no conjunto de treinamento.

Outra questão relevante é a precisão dos exemplos utilizados no treinamento. Se os exemplos não forem suficientemente precisos em representar os problemas de código que se deseja identificar, o modelo pode aprender padrões incorretos ou incompletos, resultando em uma detecção imprecisa de problemas.

5.3.2 Precisão do Modelo de IA

Embora o modelo o1-mini tenha apresentado uma taxa de acerto de 81,5% na detecção de problemas de código em testes controlados, é importante considerar que essa precisão pode variar em cenários reais, em que o código pode ser mais complexo ou apresentar problemas menos evidentes. O segundo conjunto experimentado busca simular este cenário e apresentou um bom desempenho, porém as infinitas possibilidades de *input* tornam a capacidade de acerto imponderável.

Os modelos de linguagem de grande escala (LLMs) têm limitações inerentes, como a incapacidade de compreender completamente o contexto de um projeto maior ou a aplicação de padrões de projeto complexos. Isso pode resultar em feedbacks simplistas ou

inadequados para problemas mais complexos, como observado por alguns participantes da pesquisa.

5.3.3 Aplicabilidade em Contextos Educacionais Reais

A ferramenta proposta pode enfrentar desafios de integração com os processos de ensino já estabelecidos. Professores podem ter resistência ou dificuldades em planejar e executar suas aulas incorporando a ferramenta em suas práticas pedagógicas, especialmente se ela exigir mudanças significativas na forma como as atividades são planejadas e avaliadas.

Outra ameaça é a possibilidade de os estudantes se tornarem excessivamente dependentes da ferramenta, utilizando-a como uma solução rápida para corrigir problemas de código sem realmente entender os conceitos subjacentes. Isso pode prejudicar o desenvolvimento de habilidades de pensamento crítico e resolução de problemas.

Embora a ferramenta tenha sido projetada para fornecer feedbacks educacionais, há também o risco de que o modelo de IA gere informações incorretas ou enviesadas, especialmente em casos em que o código submetido é ambíguo ou complexo. Isso pode levar a feedbacks que não apenas podem ser inúteis, mas também potencialmente prejudiciais para o aprendizado do estudante.

5.3.4 Avaliação Subjetiva dos Participantes

A avaliação subjetiva dos professores, embora valiosa, pode ser influenciada por fatores como a familiaridade com a ferramenta, o tempo de uso e as expectativas iniciais. Isso pode resultar em uma avaliação que não reflete completamente a eficácia da ferramenta em um contexto mais amplo.

A pesquisa realizada com professores de ciência da computação pode não ser representativa de todos os contextos educacionais. A amostra, embora diversificada em termos de instituições, pode não capturar a variedade de desafios enfrentados por professores em diferentes regiões ou com diferentes níveis de recursos.

5.3.5 Sustentabilidade e Manutenção da Ferramenta

Atualização dos Modelos: A ferramenta depende de modelos de IA que estão em constante evolução. Manter a ferramenta atualizada com os modelos mais recentes e garantir que ela continue a funcionar de forma eficaz pode ser um desafio.

6 CONSIDERAÇÕES E OPORTUNIDADES FUTURAS

Este trabalho propôs uma abordagem inovadora para a identificação e classificação automatizada de problemas na aprendizagem de programação orientada a objetos por meio da análise de código-fonte, utilizando inteligência artificial generativa. A ferramenta desenvolvida demonstrou ser viável e eficaz na detecção de problemas de qualidade de código e na geração de feedbacks educacionais personalizados, contribuindo para o processo de aprendizagem dos estudantes.

A partir da análise de código-fonte, a ferramenta foi capaz de identificar uma variedade de problemas, como *code smells* e violações de princípios de *design*, e correlacioná-los com dificuldades específicas de aprendizagem, como a compreensão de classes, métodos, encapsulamento e polimorfismo. A utilização dos modelos de linguagem de grande escala o1-mini e GPT-4o permitiu a criação de uma solução que não apenas detecta problemas, mas também fornece orientações educacionais contextualizadas, ajudando os estudantes a superar suas dificuldades.

Os resultados dos experimentos indicaram que o modelo o1-mini apresentou um bom desempenho na detecção de problemas de código. Além disso, a avaliação subjetiva realizada com professores de ciência da computação destacou o potencial da ferramenta como um recurso educacional valioso. Os participantes reconheceram a capacidade da ferramenta de fornecer feedbacks claros e úteis, embora tenham apontado limitações em relação à análise de soluções complexas e à necessidade de maior consistência no *feedback* gerado. A usabilidade da ferramenta foi avaliada como excelente, com um obtendo 86,25 pontos na escala SUS, indicando que a interface é intuitiva e de fácil utilização.

No entanto, é importante reconhecer as limitações e ameaças à validade da proposta. A diversidade, tamanho e precisão dos dados de treinamento, a capacidade do modelo de lidar com códigos complexos e a integração da ferramenta em contextos educacionais reais são desafios que precisam ser abordados em futuras expansões do projeto. Além disso, a dependência excessiva dos estudantes em relação à ferramenta e a possibilidade de *feedbacks* incorretos ou enviesados são questões que exigem atenção contínua.

A ferramenta foi desenvolvida e testada principalmente com código em Java, porém os testes indicaram também potencial com outras linguagens. O alcance para outras linguagens de programação, como as da plataforma .NET, C++, JavaScript, poderia ser ampliado e avaliado, tornando a ferramenta mais versátil.

Além de suas interfaces Web e *Plugin* do Visual Studio Code, a ferramenta pode ser expandida e integrada a ambientes virtuais de aprendizagem (AVAs) como o Moodle[1].

[1] https://moodle.com/

Isso permitiria que os estudantes recebessem feedbacks automatizados em tempo real, mesmo em contextos em que o acesso a professores ou tutores é limitado.

O contínuo lançamento de modelos de maior desempenho permitem a exploração das capacidades dos LLMs e o desenvolvimento da ferramenta para funcionar com outros modelos além dos o1-mini e GPT-4o. Modelos especializados como o GitHub Copilot[2], xAI Grok[3], Vertex AI[4], DeepSeek R1[5] dentre inúmeros outros modelos atuais e futuros podem ser experimentados, avaliados e incorporados.

Este trabalho é um início de uma longa caminhada de investigações sobre o uso de IA generativa na educação, destacando a importância de uma abordagem equilibrada que combine tecnologia avançada com práticas pedagógicas eficazes e buscando a superação do frequente uso marginal e não planejado dos LLMs. Com ajustes e expansões, a CodeBuddy pode se tornar um recurso bastante útil para professores e estudantes, promovendo a qualidade do código e a compreensão dos conceitos fundamentais da programação orientada a objetos.

[2] https://copilot.github.com/
[3] https://x.ai/grok
[4] https://cloud.google.com/vertex-ai
[5] https://www.deepseek.com/

REFERÊNCIAS

[1] VASWANI, A. et al. Attention is all you need. *CoRR*, abs/1706.03762, 2017. Disponível em: <http://arxiv.org/abs/1706.03762>.

[2] OPENAI. *OpenAI o1-mini: Advancing Cost-Efficient Reasoning*. 2024. Accessed: 2024-11-22. Disponível em: <https://openai.com/index/openai-o1-mini-advancing-cost-efficient-reasoning/>.

[3] WATSON, C.; LI, F. W. B. Failure rates in introductory programming revisited. In: *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*. New York, New York, USA: ACM Press, 2014. p. 39–44. Disponível em: <https://doi.org/10.1145/2591708.2591749>.

[4] MARTINS, V. F.; CONCILIO, I. de A. S.; GUIMARÃES, M. de P. Problem based learning associated to the development of games for programming teaching. *Computer Applications in Engineering Education*, Wiley Online Library, v. 26, n. 5, p. 1577–1589, 2018.

[5] MOSER, R. A fantasy adventure game as a learning environment: why learning to program is so difficult and what can be done about it. In: *Proceedings of the 2nd conference on Integrating technology into computer science education*. New York, NY, USA: Association for Computing Machinery, 1997. (ITiCSE '97), p. 114–116. ISBN 9780897919234. Disponível em: <https://doi.org/10.1145/268819.268853>.

[6] KONECKI, M. Problems in programming education and means of their improvement. *DAAAM international scientific book*, DAAAM International Vianna, Austria, v. 2014, p. 459–470, 2014.

[7] ATALAYA, O. F. C. Analysis of learning difficulties in object oriented programming in systems engineering students at untels. Universidad Nacional Tecnológica de Lima Sur, 2020.

[8] MAZAITIS, D. The object-oriented paradigm in the undergraduate curriculum: a survey of implementations and issues. *ACM SIGCSE Bulletin*, v. 25, n. 3, p. 58–64, 1993. Disponível em: <https://doi.org/10.1145/165408.165432>.

[9] KEUNING, H.; JEURING, J.; HEEREN, B. A systematic mapping study of code quality in education. In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. New York, NY, USA: Association for Computing Machinery, 2023. (ITiCSE 2023), p. 5–11. ISBN 9798400701382. Disponível em: <https://doi.org/10.1145/3587102.3588777>.

[10] TAN, P.-H.; TING, C.-Y.; LING, S.-W. Learning difficulties in programming courses: undergraduates' perspective and perception. In: IEEE. *2009 International Conference on Computer Technology and Development*. [S.l.], 2009. v. 1, p. 42–46.

[11] FOWLER, M. *Refactoring improving the design of existing code*. [S.l.]: Addison-Wesley, 2009.

[12] GUTIéRREZ, L. E.; GUERRERO, C. A.; LóPEZ-OSPINA, H. A. Ranking of problems and solutions in the teaching and learning of object-oriented programming. *Education and Information Technologies*, v. 27, n. 5, p. 7205–7239, jun. 2022. ISSN 1360-2357, 1573-7608.

[13] ALA-MUTKA, K. M. A survey of automated assessment approaches for programming assignments. *Computer science education*, Taylor & Francis, v. 15, n. 2, p. 83–102, 2005.

[14] THOMASSON, B.; RATCLIFFE, M.; THOMAS, L. Identifying novice difficulties in object oriented design. *ACM SIGCSE Bulletin*, ACM New York, NY, USA, v. 38, n. 3, p. 28–32, 2006.

[15] BIJU, S. M. Difficulties in understanding object oriented programming concepts. In: SPRINGER. *Innovations and Advances in Computer, Information, Systems Sciences, and Engineering*. [S.l.], 2013. p. 319–326.

[16] HOLLAND, S.; GRIFFITHS, R.; WOODMAN, M. Avoiding object misconceptions. In: . [S.l.: s.n.], 1997. v. 29, p. 131–134. ISBN 0897918894.

[17] OR-BACH, R.; LAVY, I. Cognitive activities of abstraction in object orientation: an empirical study. *SIGCSE Bull.*, Association for Computing Machinery, New York, NY, USA, v. 36, n. 2, p. 82–86, jun 2004. ISSN 0097-8418. Disponível em: <https://doi.org/10.1145/1024338.1024378>.

[18] SOMMERVILLE, I. *Software Engineering*. 10th. ed. [S.l.]: Pearson, 2015. ISBN 0133943038.

[19] MENOLLI, A.; STRIK, B.; RODRIGUES, L. Teaching refactoring to improve code quality with chatgpt: An experience report in undergraduate lessons. In: *Proceedings of the XXIII Brazilian Symposium on Software Quality*. New York, NY, USA: Association for Computing Machinery, 2024. (SBQS '24), p. 563–574. ISBN 9798400717772. Disponível em: <https://doi.org/10.1145/3701625.3701681>.

[20] BAIDOO-ANU, D.; ANSAH, L. O. Education in the era of generative artificial intelligence (ai): Understanding the potential benefits of chatgpt in promoting teaching and learning. *Journal of AI*, İzmir Academy Association, v. 7, n. 1, p. 52–62, 2023.

[21] PRATHER, J. et al. The robots are here: Navigating the generative ai revolution in computing education. In: *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education*. New York, NY, USA: Association for Computing Machinery, 2023. (ITiCSE-WGR '23), p. 108–159. ISBN 9798400704055. Disponível em: <https://doi.org/10.1145/3623762.3633499>.

[22] SILVA, C. A. G. d. et al. Chatgpt: Challenges and benefits in software programming for higher education. *Sustainability*, MDPI, v. 16, n. 3, p. 1245, 2024.

[23] RAJALA, J. et al. \"call me kiran\--chatgpt as a tutoring chatbot in a computer science course. In: *Proceedings of the 26th International Academic Mindtrek Conference*. [S.l.: s.n.], 2023. p. 83–94.

[24] HELLAS, A. et al. Exploring the responses of large language models to beginner programmers' help requests. In: *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*. [S.l.: s.n.], 2023. p. 93–105.

[25] SHEESE, B. et al. Patterns of student help-seeking when using a large language model-powered programming assistant. In: *Proceedings of the 26th Australasian Computing Education Conference*. [S.l.: s.n.], 2024. p. 49–57.

[26] STRIK, B. H.; MENOLLI, A.; BRANCHER, J. D. Gpt ai in computer science education: A systematic mapping study. In: *Proceedings of the XXXV Simpósio Brasileiro de Informática na Educação*. Rio de Janeiro, Brazil: [s.n.], 2024. Accepted, to appear.

[27] ZORZO, A. F. et al. *Referenciais de Formação para os Cursos de Graduação em Computação*. [S.l.]: Sociedade Brasileira de Computação (SBC), 2017. 153 p.

[28] CARBONELL, J. R. Ai in cai: An artificial-intelligence approach to computer-assisted instruction. *IEEE transactions on man-machine systems*, IEEE, v. 11, n. 4, p. 190–202, 1970.

[29] WILLIAMSON, B. The social life of ai in education. *International Journal of Artificial Intelligence in Education*, Springer, v. 34, n. 1, p. 97–104, 2024.

[30] OPENAI. *Introducing ChatGPT*. 2022. <https://openai.com/index/chatgpt/>. Accessed: 2024-06-01.

[31] CALDARINI, G.; JAF, S.; MCGARRY, K. A literature survey of recent advances in chatbots. *Information*, MDPI, v. 13, n. 1, p. 41, 2022.

[32] MINAEE, S. et al. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.

[33] ACHIAM, J. et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[34] HAMMAD, R.; BAHJA, M. Opportunities and challenges in educational chatbots. *Trends, Applications, and Challenges of Chatbot Technology*, IGI Global, p. 119–136, 2023.

[35] GHASSEMI, M. et al. Chatgpt one year on: who is using it, how and why? *Nature*, Nature Publishing Group UK London, v. 624, n. 7990, p. 39–41, 2023.

[36] HWANG, G.-J.; CHANG, C.-Y. A review of opportunities and challenges of chatbots in education. *Interactive Learning Environments*, Taylor & Francis, v. 31, n. 7, p. 4099–4112, 2021.

[37] ROTMAN, D. How chatgpt will revolutionize the economy. we need to decide what that looks like. *MIT Technology Review*, March 2023. Disponível em: <https://www.technologyreview.com/2023/03/25/1070275/chatgpt-revolutionize-economy-decide-what-looks-like/>.

[38] MOSAIYEBZADEH, F. et al. Exploring the role of chatgpt in education: Applications and challenges. In: *Proceedings of the 24th Annual Conference on Information Technology Education*. New York, NY, USA: Association for Computing Machinery, 2023. (SIGITE '23), p. 84–89. ISBN 9798400701306. Disponível em: <https://doi.org/10.1145/3585059.3611445>.

[39] DEMPERE, J. et al. The impact of chatgpt on higher education. *Frontiers in Education*, v. 8, 2023. ISSN 2504-284X. Disponível em: <https://www.frontiersin.org/articles/10.3389/feduc.2023.1206936>.

[40] FARHI, F. et al. Analyzing the students' views, concerns, and perceived ethics about chat gpt usage. *Computers and Education: Artificial Intelligence*, v. 5, p. 100180, 2023. ISSN 2666-920X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2666920X23000590>.

[41] LAU, S.; GUO, P. From"ban it till we understand it"to"resistance is futile": How university programming instructors plan to adapt as more students use ai code generation and explanation tools such as chatgpt and github copilot. In: *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*. [S.l.: s.n.], 2023. p. 106–121.

[42] BBC News. *Italy Temporarily Bans ChatGPT Over Privacy Concerns*. 2023. <https://www.bbc.com/news/technology-65139406>. Accessed: 2024-06-01.

[43] ALBONICO, M.; VARELA, P. J. A report on the use of chatgpt in software engineering and systems analysis courses. In: *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2023. p. 303–311.

[44] MAHER, M. L.; TADIMALLA, S. Y.; DHAMANI, D. An exploratory study on the impact of ai tools on the student experience in programming courses: an intersectional analysis approach. In: IEEE. *2023 IEEE Frontiers in Education Conference (FIE)*. [S.l.], 2023. p. 1–5.

[45] KIESLER, N.; LOHR, D.; KEUNING, H. Exploring the potential of large language models to generate formative programming feedback. In: IEEE. *2023 IEEE Frontiers in Education Conference (FIE)*. [S.l.], 2023. p. 1–5.

[46] ZASTUDIL, C. et al. Generative ai in computing education: Perspectives of students and instructors. In: IEEE. *2023 IEEE Frontiers in Education Conference (FIE)*. [S.l.], 2023. p. 1–9.

[47] LIU, P. et al. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, ACM New York, NY, v. 55, n. 9, p. 1–35, 2023.

[48] WHITE, J. et al. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[49] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 25000:2014 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. Geneva, Switzerland, 2014. Disponível em: <https://www.iso.org/standard/64764.html>, Acesso em: 12 ago. 2024.

[50] STEGEMAN, M.; BARENDSEN, E.; SMETSERS, S. Towards an empirically validated model for assessment of code quality. In: *Proceedings of the 14th Koli Calling international conference on computing education research*. [S.l.: s.n.], 2014. p. 99–108.

[51] HAMM, R. W. et al. A tool for program grading: The jacksonville university scale. In: *Proceedings of the fourteenth SIGCSE technical symposium on Computer science education*. [S.l.: s.n.], 1983. p. 248–252.

[52] HOWATT, J. W. On criteria for grading student programs. *ACM SIGCSE Bulletin*, ACM New York, NY, USA, v. 26, n. 3, p. 3–7, 1994.

[53] BECKER, K. Grading programming assignments using rubrics. In: *Proceedings of the 8th annual conference on Innovation and technology in computer science education*. [S.l.: s.n.], 2003. p. 253–253.

[54] SMITH, L.; CORDOVA, J. Weighted primary trait analysis for computer program evaluation. *Journal of Computing Sciences in Colleges*, Consortium for Computing Sciences in Colleges, v. 20, n. 6, p. 14–19, 2005.

[55] LIEBERHERR, K. J.; HOLLAND, I. M. Assuring good style for object-oriented programs. *IEEE software*, IEEE, v. 6, n. 5, p. 38–48, 1989.

[56] THOMAS, D.; HUNT, A. *The Pragmatic Programmer: your journey to mastery*. [S.l.]: Addison-Wesley Professional, 2019.

[57] MARTIN, R. C. Design principles and design patterns. *Object Mentor*, v. 1, n. 34, p. 597, 2000.

[58] LISKOV, B. H.; WING, J. M. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM New York, NY, USA, v. 16, n. 6, p. 1811–1841, 1994.

[59] SHVETS, A. *Dive into refactoring*. [S.l.]: Refactoring Guru, 2019.

[60] LAHTINEN, E.; ALA-MUTKA, K.; JäRVINEN, H.-M. A study of the difficulties of novice programmers. In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. New York, NY, USA: Association for Computing Machinery, 2005. (ITiCSE '05), p. 14–18. ISBN 1595930248. Disponível em: <https://doi.org/10.1145/1067445.1067453>.

[61] CHEAH, C. S. Factors contributing to the difficulties in teaching and learning of computer programming: A literature review. *Contemporary Educational Technology*, Bastas, v. 12, n. 2, p. ep272, 2020.

[62] ISMAIL, M. N.; NGAH, N. A.; UMAR, I. N. Instructional strategy in the teaching of computer programming: a need assessment analyses. *The Turkish Online Journal of Educational Technology*, v. 9, n. 2, p. 125–131, 2010.

[63] KÖLLING, M. The problem of teaching object-oriented programming, part 1: Languages. *Journal of Object-oriented programming*, SIGS Publications, v. 11, n. 8, p. 8–15, 1999.

[64] CORBIN, J.; STRAUSS, A. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. [S.l.]: Sage publications, 2014.

[65] KARAHASANOVIC, A.; LEVINE, A.; THOMAS, R. Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study. *Journal of Systems and Software*, v. 80, p. 1541–1559, 09 2007.

[66] RAHMAN, M. M.; WATANOBE, Y. Chatgpt for education and research: Opportunities, threats, and strategies. *Applied Sciences*, MDPI, v. 13, n. 9, p. 5783, 2023.

[67] NIELSEN, J. *Usability Engineering*. Boston: Academic Press, 1993.

[68] OPENAI. *Introdução à Referência da API*. 2024. Acessado em: 05 de dezembro de 2024. Disponível em: <https://platform.openai.com/docs/api-reference>.

[69] DataCamp Team. *How Transformers Work: An Introduction*. 2023. Accessed: 2024-11-12. Disponível em: <https://www.datacamp.com/tutorial/how-transformers-work>.

[70] ARORA, R. et al. Understanding deep neural networks with rectified linear units. *arXiv preprint arXiv:1611.01491*, 2016.

[71] FRANKE, M.; DEGEN, J. The softmax function: Properties, motivation, and interpretation. PsyArXiv, 2023.

[72] CHEN, M. et al. *Evaluating Large Language Models Trained on Code*. 2021. Disponível em: <https://arxiv.org/abs/2107.03374>.

[73] YENDURI, G. et al. Gpt (generative pre-trained transformer)–a comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions. *IEEE Access*, IEEE, 2024.

[74] LI, S. E. Deep reinforcement learning. In: *Reinforcement learning for sequential decision and optimal control*. [S.l.]: Springer, 2023. p. 365–402.

[75] 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.

[76] REIS, J. Pereira dos et al. Code smells detection and visualization: a systematic literature review. *Archives of Computational Methods in Engineering*, Springer, v. 29, n. 1, p. 47–94, 2022.

[77] TSANTALIS, N. et al. *Tsantalis/JDeodorant: Jdeodorant*. 2010. Disponível em: <https://github.com/tsantalis/JDeodorant>.

[78] LENARDUZZI, V. et al. A critical comparison on six static analysis tools: Detection, agreement, and precision. *CoRR*, abs/2101.08832, 2021. Disponível em: <https://arxiv.org/abs/2101.08832>.

[79] POMIAN, D. et al. Next-generation refactoring: Combining llm insights and ide capabilities for extract method. In: IEEE. *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2024. p. 275–287.

[80] LIU, B. et al. *An Empirical Study on the Potential of LLMs in Automated Software Refactoring*. 2024. Disponível em: <https://arxiv.org/abs/2411.04444>.

[81] LUNA, S. M. M. Manual práctico para el diseño de la escala likert. *Xihmai*, v. 2, n. 4, 2012. Disponível em: <https://revistas.lasallep.edu.mx/index.php/xihmai/article/view/101>.

[82] BUDHIRAJA, R. et al. “it’s not like jarvis, but it’s pretty close!”-examining chatgpt’s usage among undergraduate students in computer science. In: *Proceedings of the 26th Australasian Computing Education Conference*. [S.l.: s.n.], 2024. p. 124–133.

[83] DAI, W. et al. Can large language models provide feedback to students? a case study on chatgpt. In: IEEE. *2023 IEEE International Conference on Advanced Learning Technologies (ICALT)*. [S.l.], 2023. p. 323–325.

[84] PRATHER, J. et al. “it’s weird that it knows what i want”: Usability and interactions with copilot for novice programmers. *ACM Transactions on Computer-Human Interaction*, ACM New York, NY, v. 31, n. 1, p. 1–31, 2023.

[85] BROOKE, J. Sus: A quick and dirty usability scale. *Usability Eval. Ind.*, v. 189, 11 1995.

Apêndices

APÊNDICE A – LISTA DE PROBLEMAS DE CÓDIGO E SIGLAS PARA INTERPRETAÇÃO DOS APÊNDICES

Problema de código	**Sigla**
Alternative classes with different interfaces	ACDI
Comments	COMM
Data class	DATA
Data clumps	CLUM
Dead code	DEAD
Divergent change	DIVC
Duplicate code	DUPL
Feature envy	ENVY
Inappropriate intimacy	INNA
Large class	LARG
Lazy class	LAZY
Long method	LOME
Long parameter list	LOPL
Message chain	MSGC
Middle man	MIDM
Parallel inheritance hierarchies	PIHI
Primitive obsession	POBS
Refused bequest	REFB
Shotgun surgery	SHOT
Speculative generality	SGEN
Switch statements	SWIT
Temporary fields	TEMP
Violações da Lei de Demeter	DEME
Violações do Tell Don't Ask	TELL
Violações do Single Responsability Principle	SRPV
Violações do Open/Closed Principle	OCPV
Violações do Liskov Substitution Principle	LSPV
Violações do Interface Segregation Principle	ISPV
Violações Dependency Inversion Principle	DIPV

APÊNDICE B – RESULTADOS COMPLETOS DA ANÁLISE DO PRIMEIRO GRUPO DE AMOSTRAS PELO LLM O1-MINI

Amostra	Problemas esperados	Problemas identificados
1	ACDI	
2	ACDI, DATA	ACDI, DATA
3	ACDI, DATA	DATA
4	COMM	COMM, DUPL
5	COMM	COMM
6	COMM, DATA	COMM, DATA
7	DATA, SRPV	DATA, SRPV
8	DATA	DATA
9	DATA	DATA
10	CLUM, DUPL, LOPL, SRPV	CLUM, DUPL, LOPL, MIDM, SRPV
11	DATA, CLUM, LOPL, SRPV	DATA, CLUM, LOPL, SRPV
12	DATA, CLUM	DATA, CLUM
13	DEAD	
14	DEAD, LARG, LOME, SRPV	CLUM, DEAD, LARG, LOME, SRPV
15	DEAD, LOME	INNA, LOME, SRPV
16	DIVC, SRPV	SRPV
17	DIVC, SRPV	SRPV
18	DIVC, SRPV	DIVC
19	DUPL	DUPL
20	DUPL	DEAD, DUPL
21	DUPL	COMM, DUPL
22	ENVY, MIDM, SRPV	MIDM, SRPV
23	ENVY, MIDM	ENVY, MIDM
24	ENVY	
25	INNA	MIDM
26	INNA	DATA, INNA, MIDM
27	INNA	
28	DATA, LARG	DATA, LARG
29	DATA, CLUM, LARG, SRPV	DATA, CLUM, LARG, SRPV
30	DATA, LARG, LOPL	DATA, LARG, LOPL

31	DATA, LAZY	DATA
32	DATA, LAZY	DATA, LAZY
33	DATA, LAZY	DATA, LAZY
34	LOME	LOME
35	LOME	
36	LOME	
37	DATA, LOPL	DATA, LOPL
38	DATA, CLUM, LOPL, SRPV	DATA, CLUM, LAZY, LOPL, SRPV
39	DATA, LOPL	DATA, LOPL
40	DATA, MSGC, MIDM	DATA, MSGC, MIDM
41	LAZY, MSGC, MIDM	DATA, MIDM
42	MSGC, MIDM	MIDM
43	MIDM	MIDM
44	MIDM	MIDM
45	MIDM	MIDM
46	PIHI	PIHI
47	PIHI	
48	PIHI	
49	POBS	LOPL, POBS
50	DATA, LOPL, POBS	DATA, LOPL
51	POBS	CLUM, LOPL, POBS
52	REFB, ISPV	ISPV
53	ACDI, MIDM, REFB, ISPV	ACDI, MIDM, REFB, ISPV
54	MIDM, REFB, ISPV	MIDM, REFB, ISPV
55	SHOT, SRPV	SHOT, SRPV
56	SHOT	SHOT
57	SHOT	SHOT
58	SGEN	SGEN
59	SGEN	SGEN
60	SGEN	SGEN
61	SWIT, OCPV	SWIT, OCPV
62	SWIT, OCPV	SWIT, OCPV
63	SWIT, OCPV	SWIT, OCPV
64	TEMP	TEMP
65	TEMP	TEMP
66	TEMP	TEMP
67	DEME	DEME
68	LAZY, DEME	LAZY, DEME

69	DEME	DEME
70	TELL	
71	TELL	TELL
72	TELL	DATA
73	DATA, SRPV	DATA, SRPV
74	DATA, SRPV	DATA, SRPV
75	DATA, SRPV	DATA, SRPV
76	OCPV	OCPV
77	OCPV	OCPV
78	OCPV	OCPV
79	LSPV	LSPV
80	LSPV	LSPV
81	LSPV	LSPV
82	ISPV	ISPV
83	ISPV	ISPV
84	ISPV	ISPV
85	MIDM, DIPV	MIDM
86	DIPV	DIPV
87	MIDM, DIPV	MIDM

APÊNDICE C – RESULTADOS COMPLETOS DA ANÁLISE DO SEGUNDO GRUPO DE AMOSTRAS PELO LLM O1-MINI

Projeto	**Classe**	**Problemas esperados**	**Problemas identificados**
Rent-a-car	Booking	COMM, DATA, CLUM, DUPL, LARG, LAZY, LOME, SHOT, SRPV	COMM, DATA, CLUM, DUPL, LARG, LAZY, LOME, SHOT, SRPV
Rent-a-car	Car	COMM, DATA, CLUM, DUPL, LARG, LOME, LOPL, POBS, SRPV	COMM, DATA, CLUM, LARG, LOME, LOPL, POBS, SRPV
Rent-a-car	CarOwner	DATA, DUPL, LARG, LOME	DATA, DUPL, LARG, LOME, SRPV
Rent-a-car	Person	COMM, DATA, CLUM, DEAD, DUPL	COMM, DATA, CLUM, DEAD, DUPL
Biblioteca	Cliente	COMM, DATA, LOPL	COMM, DATA, LOPL
Biblioteca	CriaConexao	COMM, SRPV, DIPV	Nenhum
Biblioteca	Emprestimo	COMM, DATA	COMM, DATA, TELL
Biblioteca	Livro	COMM, DATA, CLUM, LOPL	COMM, DATA, CLUM, LOPL
Biblioteca	Multa	COMM, DATA, LOPL, SRPV	COMM, DATA, LOPL, POBS, SRPV
Cadastro	ConexaoUtil	COMM, INNA	COMM
Cadastro	LoginDAO	COMM, DEAD, LOME, PIHI, SRPV	COMM, DEAD, ENVY, INNA, LOME, SRPV
Cadastro	LoginDTO	DATA	DATA
Cadastro	PessoaDAO	COMM, DEAD, LARG, LOME, PIHI	COMM, DEAD, LARG, LOME
Cadastro	PessoaDTO	DATA, DEAD	COMM, DATA, DEAD, MIDM

APÊNDICE D – FORMULÁRIO DA PESQUISA: ANÁLISE DO AGENTE CODEBUDDY

Prezado(a) professor(a),

Gostaríamos de convidá-lo(a) a participar da pesquisa, intitulada "UMA ABORDAGEM BASEADA EM INTELIGÊNCIA ARTIFICIAL PARA IDENTIFICAÇÃO E CLASSIFICAÇÃO AUTOMATIZADA DE PROBLEMAS NA APRENDIZAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETOS POR MEIO DA ANÁLISE DE CÓDIGO-FONTE". Este estudo está sendo conduzido pelo pesquisador Bruno Henrique Strik, docente do IFPR e mestrando do Programa de Pós Graduação em Ciência da Computação da UEL, sob orientação do prof. Dr. André Menolli.

Objetivo do Estudo: Objetivo desde estudo é compreender as percepções de professores de ciência da computação e informática, com foco em avaliar a contribuição da ferramenta de IA CodeBuddy no aprendizado dos estudantes.

Confidencialidade e Privacidade: Todos os dados coletados serão tratados de forma anônima e sigilosa. Nenhuma informação pessoal ou identificável será divulgada. Você poderá desistir de participar a qualquer momento, sem necessidade de justificativa, e os dados coletados até então serão excluídos. A participação é voluntária, sem qualquer custo ou benefício financeiro. Todos os dados serão armazenados em sistemas protegidos por senha e acessíveis apenas pelo pesquisador principal, garantindo o anonimato. Os resultados serão apresentados de forma agregada, sem identificação individual dos participantes.

Metodologia: A coleta de dados será realizada por meio de um questionário online. As perguntas abordarão o perfil demográfico e profissional do docente e suas percepções sobre a contribuição da ferramenta no processo de aprendizagem de programação.

Participação: Sua participação nesta pesquisa é totalmente voluntária. Você pode optar por não responder ou interromper sua participação a qualquer momento, sem prejuízo algum.

Dados Coletados: Informações demográficas; Endereço de e-mail; Experiência profissional e atuação como docente; Impressões e percepções sobre o uso da ferramenta; Comentários, sugestões e respostas às perguntas do questionário.

Direitos do Participante: De acordo com a LGPD, você tem os seguintes direitos: Solicitar informações sobre o tratamento de seus dados; Retirar seu consentimento a qualquer momento, sem prejuízo à sua participação; Solicitar a exclusão de seus dados pessoais coletados, caso deseje.

Pesquisador: Bruno Henrique Strik (UEL e IFPR) bruno.strik@ifpr.edu.br

Orientador: André Menolli (UENP e UEL) menolli@uenp.br

Ao avançar, você declara que leu e compreendeu os objetivos, procedimentos e condições desta pesquisa, assim como os direitos garantidos pela LGPD. Concorda em participar voluntariamente e autoriza o uso dos seus dados para os fins descritos acima.

Observação: Para respostas na escala Likert, considere os seguintes valores:

1. Discodo totalmente
2. Discordo
3. Neutro
4. Concordo
5. Concordo totalmente

Agradecemos sua colaboração e contribuição!

Experimentação da Ferramenta

Antes de iniciar a pesquisa, experimente a ferramenta Codebuddy, que pode ser acessada clicando neste link[1]. Esta é uma ferramenta que utiliza inteligência artificial para, a partir de código fonte, identificar problemas de qualidade (code smells, violações do SOLID, etc) e problemas de aprendizagem associados a eles, produzindo *feedback* com finalidade educacional. O LLM por trás da ferramenta está validado para analisar código Java, porém é capaz de analisar outras linguagens de programação.

A ferramenta é simples e intuitiva, mas se considerar necessário há instruções simples de uso neste quadro[2].

Se quiser experimentar também a versão plugin do Visual Studio Code você pode encontrá-la no Marketplace[3].

Você pode testar a ferramenta com código-fonte próprio ou, se quiser, pode utilizar nossa lista de exemplos, que pode ser acessada aqui[4].

Experimentada a ferramenta, prossiga com a pesquisa.

1. Perfil do Participante

Nesta seção, responda com os dados de sua trajetória acadêmica e profissional.

[1] https://srvbruno.vps.webdock.cloud/
[2] https://srvbruno.vps.webdock.cloud/manual.pdf
[3] https://marketplace.visualstudio.com/items?itemName=brunostrikti.codebuddycsedu
[4] https://github.com/brunostrik/BadQualityCodeExamples

1.1. Qual seu e-mail?

(Texto livre, opcional)

1.2. Qual sua formação acadêmica? (caixas de seleção)

( ) Ciência da computação

( ) Sistemas de Informação

( ) Análise e Desenvolvimento de Sistemas

( ) Engenharia de Software

( ) Outros (texto livre)

1.3. Qual sua maior titulação? (múltipla escolha)

( ) Graduação (licenciatura, bacharelado ou tecnólogo) ( ) Especialização (Pós-graduação Lato Sensu) ( ) Mestrado ( ) Doutorado

1.4. Quantos anos de experiência você possui no exercício da docência?

(Número)

1.5. Além da docência, você possui experiência profissional na área da computação? Quantos anos?

(Número, opcional)

1.6. Assinale quais assuntos abaixo você ministra ou já ministrou como docente, em qualquer disciplina: (caixas de seleção)

( ) Programação introdutória (Lógica, Algoritmos)

( ) Programação orientada a objetos

( ) Desenvolvimento de sistemas (desktop, web, móvel ou embarcado)

( ) Engenharia de software

( ) Banco de dados

( ) Inteligência artificial

( ) Infraestrutura de TI

( ) Outros (texto livre)

1.7. Sobre programação orientada a objetos, você considera ter qual nível de experiência? (múltipla escolha)

( ) Nenhum

( ) Pouco

( ) Intermediário

( ) Avançado

( ) Especialista

1.8. Você é familiarizado com princípios de qualidade de código (Clean Code, SOLID, etc)? (múltipla escolha)

( ) Não conheço

( ) Pouco

( ) Intermediário

( ) Avançado

( ) Especialista

2. Identificação de Problemas no Código

Nesta seção, responda com suas impressões sobre a capacidade da ferramenta identificar problemas de qualidade de código (code smells, violações do SOLID, etc).

2.1. A ferramenta conseguiu apontar problemas de qualidade no código corretamente?

(Escala Likert)

2.2. Você tem alguma observação sobre a capacidade da ferramenta identificar problemas de qualidade de código?

(Texto livre, opcional)

3. Feedback

Nesta seção, responda com suas impressões sobre o texto de *feedback* produzido pela ferramenta, incluindo o relatório completo.

3.1. O *feedback* produzido pela ferramenta é de fácil compreensão.

(Escala Likert)

3.2. O *feedback* produzido pela ferramenta apresenta informações corretas.

(Escala Likert)

3.3. O *feedback* produzido pela ferramenta vai na mesma direção dos feedbacks que eu daria aos meus estudantes em atividades reais.

(Escala Likert)

3.4. Que aspectos você considera positivos e negativos no *feedback* produzido pela ferramenta?

(Texto livre, opcional)

4. Ensino e Aprendizagem

Nesta seção, responda com suas impressões sobre a contribuição da ferramenta no processo educacional.

4.1. Com que frequência você observa que os estudantes recorrem a gambiarras ou produzem código de baixa qualidade em suas atividades?

( ) Nunca

( ) Raramente

( ) Às vezes

( ) Frequentemente

( ) Sempre

4.2. O uso da ferramenta auxilia o estudante a entender os problemas de qualidade de código apontados.

(Escala Likert)

4.3. A ferramenta pode ajudar o estudante a reestruturar seu código na direção de minimizar os problemas de qualidade identificados.

(Escala Likert)

4.4. Comparado ao ChatGPT, a ferramenta minimiza a possibilidade de plágio e cópia de respostas se utilizada pelo estudante.

(Escala Likert)

4.5. De que formas você percebe que a ferramenta pode contribuir com a aprendizagem do estudante?

(Texto livre, opcional)

5. Uso da Ferramenta

Nesta seção, responda sobre o seu uso experimental da ferramenta.

5.1. Quantos códigos-fonte distintos você utilizou na ferramenta?

(Número)

5.2. Assinale de acordo com a fonte do código-fonte que você utilizou na ferramenta

( ) Utilizei os códigos-fonte do repositório de exemplo

( ) Utilizei códigos-fonte próprios

5.43. Qual a linguagem de programação dos códigos-fonte experimentados?

( ) Java

( ) Python

( ) Javascript

( ) C#

( ) C++

( ) Outro (texto livre)

6. Usabilidade

Nesta seção, responda sobre sua experiência de uso da ferramenta. O questionário desta seção é baseado na escala SUS - System Usability Scale.

6.1. Eu acho a ferramenta útil e a utilizaria com frequência

(Escala Likert)

6.2. Eu acho a ferramenta desnecessariamente complexa

(Escala Likert)

6.3. Eu acho a ferramenta fácil de usar

(Escala Likert)

6.4. Eu acho que precisaria de suporte técnico para usar a ferramenta

(Escala Likert)

6.5. Eu acho que as diversas funcionalidades da ferramenta estão bem integradas

(Escala Likert)

6.6. Eu acho que a ferramenta apresenta muitas inconsistências

(Escala Likert)

6.7. Eu imagino que a maioria das pessoas aprenderia a usar a ferramenta rapidamente

(Escala Likert)

6.8. Eu achei a ferramenta muito difícil de usar

(Escala Likert)

6.9. Eu me senti confiante ao usar a ferramenta

(Escala Likert)

6.10. Eu precisei aprender muitas coisas antes de começar a usar a ferramenta

(Escala Likert)

7. Oportunidades de Aplicação

Nesta seção, responda de acordo com as suas impressões sobre as possibilidades de aplicação prática da ferramenta

7.1. O uso da ferramenta é viável em sala de aula

(Escala Likert)

7.2. De que forma você consegue enxergar potencial de aplicação educacional da ferramenta? (caixas de seleção)

( ) Não consigo enxergar potencial de aplicação educacional da ferramenta

( ) Pelo docente, para revisão das atividades dos estudantes

( ) Pelo docente, para preparação das atividades de ensino

( ) Pelos estudantes, como ferramenta de apoio durante as atividades

( ) Pelos estudantes, como uma ferramenta revisora de suas atividades

( ) Outros (texto livre)

7.3. Em quais disciplinas você considera útil a aplicação da ferramenta? (caixas de seleção)

( ) Não considero útil a aplicação da ferramenta

( ) Programação introdutória (Lógica, Algoritmos)

( ) Programação orientada a objetos

( ) Desenvolvimento de sistemas (desktop, web, móvel ou embarcado)

( ) Engenharia de software

( ) Banco de dados

( ) Inteligência artificial

( ) Infraestrutura de TI

( ) Outros (texto livre)

7.4. Como a ferramenta poderia ser aprimorada?

(Texto livre, opcional)

7.5. Utilize este espaço para adicionar comentários e sugestões

(Texto livre, opcional)

TRABALHOS PUBLICADOS PELO AUTOR

Trabalhos publicados pelo autor durante o programa.

Bruno H. Strik, André Menolli, Jacques D. Brancher. **GPT AI in Computer Science Education: A Systematic Mapping Study**, XXXV Simpósio Brasileiro de Informática na Educação (SBIE), 2024. (Qualis CC 2020, B1)

André Menolli, Bruno H. Strik, Luiz Rodrigues. **Teaching Refactoring to Improve Code Quality with ChatGPT: An Experience Report in Undergraduate Lessons**, XXIII Simpósio Brasileiro de Qualidade de Software (SBQS), 2024. (Qualis CC 2020, B3)

Bruno H. Strik. **Inteligência Artificial Generativa.** Primeiros Passos. São Paulo: Ed. Uiclap, 2025.