UNIVERSIDADE ESTADUAL DA PARAÍBA - UEPB
CAMPUS VII - GOVERNADOR ANTÔNIO MARIZ
CENTRO DE CIÊNCIAS EXATAS E SOCIAIS APLICADAS
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FRANCISCO GABRIEL DE OLIVEIRA NEVES

FATORES RELACIONADOS À COMPREENSÃO DE CÓDIGO COM A PRESENÇA DE REFATORAÇÃO: UMA ABORDAGEM INSPIRADA NA *GROUNDED THEORY*

PATOS - PB
2025

FRANCISCO GABRIEL DE OLIVEIRA NEVES

FATORES RELACIONADOS À COMPREENSÃO DE CÓDIGO COM A PRESENÇA DE REFATORAÇÃO: UMA ABORDAGEM INSPIRADA NA *GROUNDED THEORY*

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Ciência da Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de bacharel em Ciência da Computação.

Área de concentração: Computação - Engenharia de Software.

Orientador: Prof. Dr. José Aldo Silva da Costa.

PATOS - PB
2025

N518f Neves, Francisco Gabriel de Oliveira.

Fatores relacionados à compreensão de código com a presença de refatoração [manuscrito] : uma abordagem inspirada na Grounded Theory / Francisco Gabriel de Oliveira Neves. - 2025.

60 f. : il. color.

Digitado.

Trabalho de Conclusão de Curso (Graduação em Ciência da computação) - Universidade Estadual da Paraíba, Centro de Ciências Exatas e Sociais Aplicadas, 2025.

"Orientação : Prof. Dr. José Aldo Silva da Costa, Coordenação do Curso de Computação - CCEA".

1. Refatoração. 2. Compreensão de código. 3. Grounded Theory. 4. Ensino de programação. I. Título

21. ed. CDD 005.1

Elaborada por Kaliane Eveny Martins de Oliveira - CRB - 15/986 BSC7

FRANCISCO GABRIEL DE OLIVEIRA NEVES

FATORES RELACIONADOS À COMPREENSÃO DE CÓDIGO COM A PRESENÇA DE REFATORAMENTO: UMA ABORDAGEM INSPIRADA NA GROUNDED THEORY

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Ciência da Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação

Aprovada em: 06/06/2025.

BANCA EXAMINADORA

Documento assinado eletronicamente por:

- **Rosangela de Araujo Medeiros** (***.723.558-**), em **17/06/2025 10:25:28** com chave **88ffa6744b7e11f098901a7cc27eb1f9**.
- **José Aldo Silva da Costa** (***.862.334-**), em **17/06/2025 00:58:06** com chave **46a54eb84b2f11f0abeb1a7cc27eb1f9**.
- **Vinícius Augustus Alves Gomes** (***.754.334-**), em **17/06/2025 01:53:18** com chave **fc59b6344b3611f0901b2618257239a1**.

Documento emitido pelo SUAP. Para comprovar sua autenticidade, faça a leitura do QrCode ao lado ou acesse https://suap.uepb.edu.br/comum/autenticar_documento/ e informe os dados a seguir.
Tipo de Documento: Folha de Aprovação do Projeto Final
Data da Emissão: 17/06/2025
Código de Autenticação: cc1e9e

RESUMO

O trabalho tem como objetivo investigar os fatores que influenciam a compreensão de códigos refatorados no contexto acadêmico, utilizando entrevistas e como método de análise qualitativa uma inspiração na *Grounded Theory*. A problemática é fundamentada nas dificuldades enfrentadas por estudantes do curso de Ciência da Computação, especialmente nas disciplinas de programação, em que a compreensão de trechos de código pode ser desafiadora, sobretudo quando há pouco domínio das linguagens utilizadas. Embora a literatura aponte os benefícios da refatoração, como melhorias na estrutura e clareza do código, ainda existem lacunas no entendimento do impacto direto dessa prática na compreensão por parte dos estudantes. Na pesquisa adotou-se entrevistas semi estruturadas com 10 estudantes e 10 profissionais da área, buscando identificar os fatores que facilitam ou dificultam o entendimento de códigos refatorados, considerando aspectos como clareza, simplicidade e redução de erros. A análise qualitativa dos dados visou derivar lições aprendidas e contribuir para o avanço em estudos futuros sobre o tema, além de oferecer subsídios para aprimorar práticas pedagógicas e profissionais relacionadas ao ensino e desenvolvimento de software. Conclui-se que a refatoração possui potencial tanto para melhorar a aprendizagem da programação, quanto para funcionar como um eixo orientador da compreensão de código. Esta pesquisa revelou que práticas como nomes descritivos, extração de métodos e modularização favorecem significativamente a clareza semântica e a manutenção do código. No entanto, os benefícios dessas estratégias são mais bem aproveitados por determinado grupo, enquanto outros indivíduos ainda enfrentam barreiras relacionadas a formações técnicas frágeis, o que limita sua capacidade de aplicar e compreender plenamente os ganhos proporcionados pela refatoração.

Palavras-chave: Refatoração; Compreensão de código; Grounded Theory; Ensino de programação.

ABSTRACT

This study aims to investigate the factors that influence the comprehension of refactored code in the academic context, using interviews and a qualitative analysis approach inspired by Grounded Theory. The research problem is based on the difficulties faced by Computer Science students, especially in programming courses, where understanding code snippets can be challenging, particularly when there is limited mastery of the programming languages involved. Although the literature highlights the benefits of refactoring, such as improvements in code structure and clarity, there are still gaps in understanding its direct impact on students' code comprehension. The study employed semi-structured interviews with 10 students and 10 industry professionals to identify the factors that facilitate or hinder the understanding of refactored code, considering aspects such as clarity, simplicity, and error reduction. The qualitative data analysis aimed to extract lessons learned and contribute to future research on the subject, while also offering insights to improve pedagogical and professional practices related to software development and programming education. The findings indicate that refactoring has the potential to enhance programming learning and serve as a key element in code comprehension. This research revealed that practices such as descriptive naming, method extraction, and modularization significantly improve semantic clarity and code maintainability. However, the benefits of these strategies are more effectively leveraged by certain groups, while others still face challenges due to weak technical backgrounds, which limits their ability to fully apply and understand the advantages offered by refactoring.

Keywords: Refactoring; Code comprehension; Grounded Theory; Programming education.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

fRMI Imagem por Ressonância Magnética Funcional

GT Grounded Theory

TFD Teoria Fundamentada

SUMÁRIO

1 INTRODUÇÃO

O aumento de alunos na graduação de Ciência da Computação vem se tornando um fator relevante para entender a influência da tecnologia na sociedade, de forma que o curso prepara os estudantes para diversas linguagens de programação e áreas de atuação profissional. Dados da *Revista Pesquisa FAPESP* mostram que, entre 2014 e 2023, o número de ingressantes em cursos da área de computação no Brasil mais que triplicou, passando de 146 mil para 473 mil, o que evidencia o crescimento da demanda por profissionais capacitados na área de tecnologia e inovação. Durante a graduação, é comum sentir certa dificuldade em aprender ou compreender trechos de código nas disciplinas de programação, principalmente quando não se tem familiaridade com a linguagem em questão.

Nesse contexto acadêmico, surge a Engenharia de Software, que é uma disciplina que abrange um conjunto de práticas, métodos e ferramentas utilizadas para a concepção, desenvolvimento e manutenção de sistemas de software de alta qualidade. Segundo Sommerville (2011), a Engenharia de Software não se limita apenas à codificação, mas também envolve a gestão de requisitos, design arquitetural, verificação e validação, com o objetivo de garantir que o software atenda às expectativas dos usuários.

Um dos pilares dessa disciplina é a área de Qualidade de Software, que abrange atributos como usabilidade, eficiência, confiabilidade e manutenção. Pressman (2019) destaca que a Qualidade de Software não é um fator isolado, mas sim o resultado de um processo bem estruturado, que inclui boas práticas de engenharia desde as fases iniciais do desenvolvimento até a entrega do produto final. Para assegurar essa qualidade, a Engenharia de Software faz uso de métodos rigorosos de testes, revisões de código e ferramentas de automação, como destacado por Bass et al. (2013), que também enfatizam a importância da arquitetura de software no sucesso de projetos complexos.

Na literatura, para lidar com problemas de código mal estruturado, destaca-se a refatoração, um método que tem como objetivo melhorar a qualidade do código, promovendo alterações em sua estrutura interna sem modificar seu comportamento funcional.

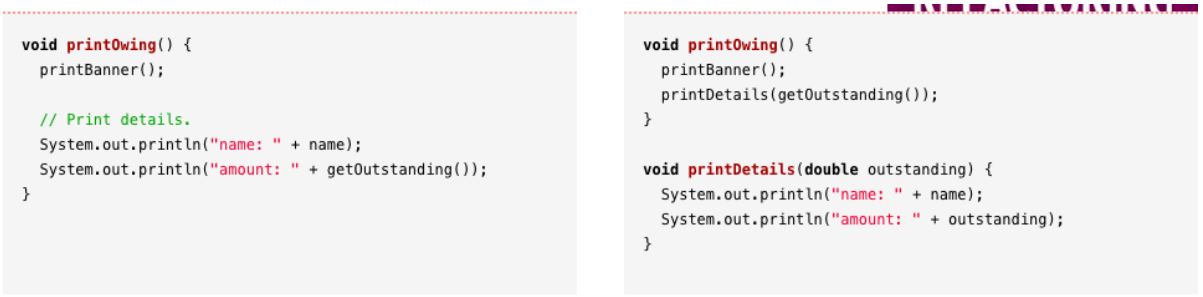
A refatoração é uma atividade que impacta positivamente a manutenção e evolução do código, e reduz a possibilidade de erros. Freire e Cheque (2007) avaliam que a refatoração torna o código mais fácil de ser entendido e menos custoso de ser alterado, garantindo simplicidade, flexibilidade e clareza.

1.1 Problemática

É fundamental compreender de que forma a refatoração impacta o entendimento dos estudantes em relação ao código. Busca-se analisar se, de fato, esse recurso contribui para tornar a leitura mais fácil, além de identificar os fatores que dificultam ou facilitam a compreensão e a aprendizagem de códigos que passaram por processos de refatoração.

A Figura 1 ilustra um exemplo prático de aplicação da refatoração, com o objetivo de melhorar a organização do código. No lado esquerdo da figura, observa-se um método responsável por exibir informações de detalhes. No lado direito da figura, foi aplicada a técnica de refatoração conhecida como Extrair Método, cujo princípio consiste em isolar um trecho específico em um novo método, atribuindo-lhe um nome descritivo.

Figura 1- Exemplo de Extrair Método



```
void printOwing() {
  printBanner();

  // Print details.
  System.out.println("name: " + name);
  System.out.println("amount: " + getOutstanding());
}
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails(double outstanding) {
  System.out.println("name: " + name);
  System.out.println("amount: " + outstanding);
}
```

Fonte: Fowler (2018).

Ao aplicar essa refatoração, autores como Fowler (2018) acreditam e defendem que melhora a estrutura e por consequência a compreensão de código. Por exemplo, ao ler o código do lado direito, o desenvolvedor pode ter uma ideia mais clara do comportamento por causa do nome do método. No entanto, esta afirmação é feita sem dados empíricos que demonstrem esse real impacto na compreensão do código.

1.2 Justificativa

Há uma necessidade por estudos que investiguem como a compreensão dos desenvolvedores é impactada por essa melhoria. Apesar da existência de estudos controlados que buscam medir esse impacto e relacionar os resultados com teorias consolidadas, há uma falta de estudos qualitativos que investiguem os fatores relacionados à compreensão a partir das próprias percepções do desenvolvedor.

Neste cenário, surge a teoria fundamentada com um método para investigar os fatores que geram esse problema do ponto de vista dos estudantes. A teoria fundamentada nesse cenário torna-se útil porque cria uma teoria com base nos dados coletados em entrevistas para identificação dos principais fatores relacionados à compreensão e aprendizagem de código.

Apesar da literatura trazer indícios sobre os benefícios da refatoração, como melhorar a estrutura, adicionar nomes mais claros, diminuir a duplicação de código, entre outros, pode haver divergência quanto ao real impacto na compreensão de código dos estudantes. Nem sempre o uso da refatoração torna-se claro o suficiente para identificar se realmente traz melhorias na performance do desenvolvedor, como rapidez de entendimento e facilidade de leitura, portanto, é necessário investigar os fatores que influenciam a compreensão de código a partir dos desenvolvedores.

1.3 Objetivos

Objetivo geral

- Investigar os fatores relacionados à compreensão e entendimento do código com a presença de refatoração.

Objetivos específicos

- Realizar um estudo por meio de entrevistas, inspirado na *Grounded Theory* para identificar os fatores relacionados ao entendimento na presença de refatoração;

- Reforçar o valor da pesquisa qualitativa primária no campo da Ciência da Computação.

2 REFERENCIAL TEÓRICO

2.1 Fundamentos de refatoração

A refatoração de código é um processo essencial na Engenharia de Software, focado em melhorar a estrutura interna de um software sem alterar seu comportamento externo. A prática de refatoração visa tornar o código mais legível, compreensível e fácil de manter, além de possibilitar a implementação de futuras modificações ou expansões com baixo risco de introduzir falhas.

Segundo Fowler (2018), a refatoração é uma técnica disciplinada que permite ao desenvolvedor reorganizar a base de código de forma contínua, aumentando a clareza e reduzindo a complexidade técnica. O autor argumenta que, ao longo do ciclo de vida do software, é comum que o código torne-se fragmentado e menos eficiente, o que faz da refatoração uma prática indispensável para garantir a qualidade do software a longo prazo.

Martin (2009) reforça a importância da refatoração ao apontar que um código bem refatorado é mais adaptável a mudanças. A mudança é uma constante no desenvolvimento de software, e o código que não é devidamente refatorado tende a acumular uma "dívida técnica", ou seja, problemas que tornam o software mais difícil e caro de modificar e manter no futuro. A dívida técnica, conforme explica Cunningham (1992), refere-se às escolhas de design e implementação que facilitam soluções rápidas a curto prazo, mas que criam desafios adicionais a longo prazo, sendo a refatoração uma das principais ferramentas para lidar com essa questão.

Além disso, Beck (2000) destaca que a refatoração é um elemento chave no desenvolvimento ágil, já que esse modelo de trabalho pressupõe ciclos curtos de entrega e mudanças frequentes. Nesse contexto, a refatoração contínua permite que as equipes mantenham a qualidade do código enquanto respondem rapidamente às necessidades dos clientes. Para Beck, a refatoração é uma técnica corretiva, e também uma prática preventiva, que visa evitar que o código se degrade com o tempo, preservando sua flexibilidade e robustez.

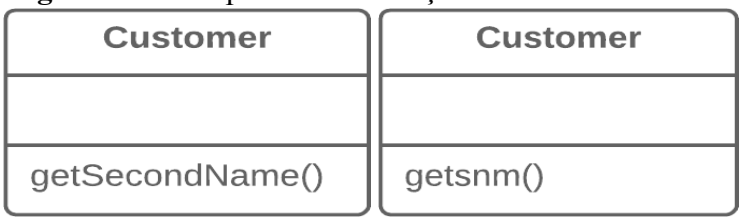
A refatoração de código é portanto um processo fundamental para garantir a manutenção e a evolução contínua de sistemas de software. Neste sentido a prática de refatoração vai além da simples melhoria estética do código, desempenha um papel central na sustentabilidade do projeto de software a longo prazo, diminuindo a dívida técnica e tornando o código mais legível e adaptável, contribuindo diretamente para a qualidade final do produto.

2.1.2 Técnicas de refatorações

No contexto da refatoração de código, diversas técnicas são aplicadas para melhorar a qualidade do código sem alterar seu comportamento externo. Essas técnicas são amplamente discutidas por autores como Fowler (2018), Mens (2021) e Beck (2000), e cada uma aborda aspectos específicos da estrutura e organização do código.

Existem diversas técnicas de refatoração, cada uma voltada para resolver problemas específicos de design ou implementação. A renomeação de variáveis é uma técnica que, conforme descrito por Fowler (2018), consiste em alterar o nome de variáveis, métodos ou classes para algo mais descritivo, facilitando a compreensão do código. Por exemplo, na Figura 2 a seguir, do lado esquerdo temos uma variável genérica “getsnm( )” , no contexto do código é uma abreviação do inglês “second name”, para refatorar essa variável em algo mais compreensível ela foi renomeada do lado direito como “getSecondName( )”, o que auxilia a melhorar a legibilidade e facilita o entendimento do seu propósito no código.

Figura 2 - Exemplo de renomeação de variáveis e refatoração de variáveis



Fonte: Fowler (2018).

Um dos princípios básicos da refatoração é evitar a duplicação de código. Fowler (2018) ressalta que quando se encontram trechos de código duplicados em diferentes partes de um programa, eles devem ser consolidados em uma única função. Por exemplo, se um código para calcular impostos aparece em vários locais, é preferível criar uma função CalcularImposto() e chamá-la onde for necessário, reduzindo a chance de erros e facilitando a manutenção.

Na Figura 3, a duplicação de código ocorre porque os dois métodos precoComImpostoProdutoA e precoComImpostoProdutoB possuem a mesma lógica de calcular o preço total de um produto, incluindo um imposto de 20%. Ambos contêm as mesmas instruções para calcular o imposto e retornar o valor final (preco + imposto), o que resulta em redundância.

Figura 3 - Exemplo de duplicação de código

```java
public class Produto {

    // Método para calcular o preço total de um Produto A com imposto
    public double precoComImpostoProdutoA(double preco) {
        double imposto = preco * 0.2; // 20% de imposto
        return preco + imposto;
    }

    // Método para calcular o preço total de um Produto B com imposto
    public double precoComImpostoProdutoB(double preco) {
        double imposto = preco * 0.2; // 20% de imposto
        return preco + imposto;
    }
}
```

Fonte: Elaborado pelo autor (2024).

Na Figura 4, pode-se observar como a refatoração atua na resolução do problema de duplicação de código da Figura 3 ao introduzir o método genérico calcularImposto, que centraliza a lógica de cálculo do imposto. Esse método recebe como parâmetros o preço do produto e a taxa de imposto, realizando o cálculo multiplicando o preço pela taxa e retornando o resultado. Dessa forma, o método precoComImposto foi simplificado para utilizar o método genérico, passando o preço do produto e a taxa fixa de 20% como argumentos. Com essa abordagem, o cálculo do imposto foi delegado ao método centralizado, eliminando redundâncias e tornando o código mais organizado e eficiente.

Figura 4 - Exemplo de refatoração em código duplicado

```java
public class Produto {

    // Método para calcular o imposto
    public double calcularImposto(double preco, double taxa) {
        return preco * taxa;
    }

    // Método para calcular o preço total de um produto com imposto
    public double precoComImposto(double preco) {
        double imposto = calcularImposto(preco, 0.2); // 20% de imposto
        return preco + imposto;
    }
}
```

Fonte: Elaborado pelo autor (2024).

Por fim, uma técnica muito utilizada é o método inline, uma técnica de refatoração que substitui uma chamada de método pelo seu conteúdo, eliminando o método original quando ele é pequeno, redundante ou desnecessário. Na Figura 5, à esquerda temos o código sem refatoração, pode-se ver claramente método simplesmente delega para outro método. Em si, essa delegação não é problema, mas quando há muitos métodos assim, eles se tornam um emaranhado confuso e difícil de classificar.

À direita temos o código refatorado que substituiu as chamadas ao método pelo conteúdo do método. Fowler (2018) ressalta que essa abordagem simplifica o código ao remover abstrações desnecessárias, especialmente quando o método é usado apenas uma vez, tornando o código mais direto e fácil de compreender. No entanto, Mens (2021) alerta que o uso excessivo do Inline Method pode resultar em métodos longos e complexos, comprometendo a clareza. Assim, a aplicação dessa técnica deve ser equilibrada, visando manter a simplicidade sem sacrificar a modularidade do código.

Figura 5 - Exemplo de aplicação do método inline

```
class PizzaDelivery {
  // ...
  int getRating() {
    return moreThanFiveLateDeliveries() ? 2 : 1;
  }
  boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }
}
```

```
class PizzaDelivery {
  // ...
  int getRating() {
    return numberOfLateDeliveries > 5 ? 2 : 1;
  }
}
```

Fonte: Fowler (2018).

2.2 Compreensão de código

2.2.1 Conceitos e definições

A compreensão do código é um aspecto fundamental do processo de refatoração. De acordo com Martin (2009), para que um pesquisador possa refatorar o código de maneira eficaz, ele precisa ser capaz de entender a lógica e a estrutura do software existente. A compreensão de código envolve a análise da forma como as diferentes partes do programa interagem e como elas contribuem para o funcionamento geral do sistema.

Beck (2000) complementa essa ideia ao afirmar que, quando o código é complexo ou mal estruturado, torna-se mais difícil de ser compreendido, o que leva a um aumento na

probabilidade de erros durante o processo de manutenção. Assim, técnicas de refatoração são aplicadas para melhorar essa compreensão e facilitar a manutenção contínua.

Assim, a compreensão de código é a capacidade de entender a estrutura e a lógica de um programa. Segundo Storey (2006), a clareza e a simplicidade do código facilitam a leitura e a compreensão, o que é essencial para a manutenção e evolução do software. Técnicas como uma documentação clara e comentários eficientes são essenciais para auxiliar no processo de compreensão. Quanto mais compreensível for o código, mais eficiente será o processo de refatoração, uma vez que o desenvolvedor poderá identificar mais facilmente os pontos que necessitam de melhorias.

2.2.2 Teorias de compreensão de código

A abordagem de compreensão de código Top-Down refere-se à maneira como os desenvolvedores abordam a interpretação do código a partir de uma visão geral para uma análise mais detalhada. Essa metodologia é caracterizada pelo foco inicial em conceitos gerais e na estrutura global do sistema antes de explorar os detalhes específicos da implementação. O objetivo é formar uma imagem mental ampla do código, usando conhecimento prévio, padrões e abstrações conhecidas para prever a funcionalidade do programa.

Segundo Soloway e Ehrlich (1984), os desenvolvedores frequentemente se baseiam em esquemas e estruturas mentais para guiar a interpretação de sistemas complexos, uma prática típica da abordagem Top-Down. Essa estratégia é particularmente eficaz quando os desenvolvedores já possuem familiaridade com o domínio da aplicação, pois permite uma navegação guiada pelas funcionalidades de alto nível antes de se imergir em aspectos técnicos e específicos.

Ao contrário da abordagem Top-Down, a compreensão Bottom-Up é um processo que começa com a análise minuciosa de partes individuais do código para, gradativamente, construir uma compreensão global do sistema. Os desenvolvedores que adotam essa estratégia iniciam a leitura de funções, blocos de código ou módulos específicos para entender a lógica subjacente, sem inicialmente considerar o contexto geral.

Brooks (1983) argumenta que, na abordagem Bottom-Up, o desenvolvedor cria uma "estrutura mental" a partir da observação de pequenos elementos do código, construindo gradualmente uma compreensão mais ampla do sistema. Esse método é ideal em situações em

que o desenvolvedor não possui familiaridade prévia com o sistema, permitindo que os detalhes guiem a construção de uma imagem mais completa.

A abordagem de compreensão Mista, ou Mixed, representa uma integração dos métodos Top-Down e Bottom-Up, reconhecendo que ambas as estratégias possuem vantagens específicas que, quando combinadas, podem fornecer uma compreensão mais completa e precisa do código. A compreensão mista envolve alternar entre uma análise geral e uma análise detalhada, dependendo da necessidade e do contexto do problema.

Von Mayrhauser e Vans (1995) destacam que o uso combinado das estratégias Top-Down e Bottom-Up permite que o desenvolvedor ajuste sua estratégia de leitura conforme surgem novos desafios, maximizando a eficiência na interpretação de código complexo. Essa abordagem é frequentemente vista como uma maneira mais natural e flexível de abordar a compreensão de sistemas grandes e desconhecidos, especialmente quando o código possui várias camadas de abstração.

2.2.3 Método e estratégias de investigação de compreensão de código

A compreensão de código é um tema central na Engenharia de Software, e diversas estratégias têm sido desenvolvidas para investigar como os desenvolvedores entendem e manipulam sistemas complexos. As abordagens utilizadas para estudar essa compreensão podem ser divididas em qualitativas e quantitativas. A seguir serão apresentadas e discutidas cada uma dessas abordagens.

As abordagens qualitativas concentram-se na coleta de dados sobre as experiências subjetivas dos desenvolvedores ao interagir com o código. Métodos como entrevistas e questionários estruturados têm sido utilizados para identificar as dificuldades enfrentadas e as estratégias aplicadas durante a leitura e compreensão do código. Segundo Nardi, "a análise qualitativa permite compreender não apenas os obstáculos enfrentados pelos programadores, mas também as estratégias cognitivas que eles adotam ao navegar por sistemas complexos" (Nardi, 1993).

Por meio dessas técnicas, é possível obter insights profundos sobre as percepções pessoais dos programadores, revelando os principais desafios e os padrões cognitivos que emergem durante a atividade de interpretação. Essa abordagem é particularmente valiosa para entender aspectos não mensuráveis diretamente, como frustrações, intuições e métodos de resolução de problemas.

As abordagens quantitativas, todavia, são focadas na mensuração objetiva do desempenho dos desenvolvedores em tarefas específicas, como a modificação ou correção de código. Estudos nesta linha têm como objetivo quantificar a eficiência e a precisão dos programadores durante a execução de tarefas práticas, fornecendo dados mensuráveis sobre o processo de compreensão. Storey (2006) enfatiza que o uso de métricas quantitativas, como tempo de execução e acurácia, é fundamental para avaliar a eficácia da compreensão de código em diferentes cenários, permitindo uma análise comparativa do desempenho dos desenvolvedores.

Essas métricas incluem o tempo de execução necessário para concluir uma tarefa e a precisão das soluções fornecidas, permitindo uma análise comparativa da eficácia das diferentes estratégias de leitura de código. Essa abordagem é útil para avaliar de forma padronizada o desempenho em cenários reais de desenvolvimento.

2.2.4 Técnicas qualitativas específicas

Dentro das investigações qualitativas, diversas técnicas têm sido exploradas para aprofundar o entendimento sobre como os desenvolvedores compreendem o código. Uma dessas técnicas é a coleta de opiniões, que possibilita compreender os obstáculos enfrentados por programadores ao interpretar sistemas complexos. Por meio das percepções coletadas, torna-se viável identificar áreas críticas e propor melhorias nas práticas de desenvolvimento, contribuindo para a construção de um conhecimento mais profundo sobre o tema.

Outra técnica aplicada são os questionários estruturados, amplamente utilizados para mapear estratégias cognitivas adotadas pelos desenvolvedores, fornecendo insights detalhados sobre os métodos empregados para identificar erros e compreender a lógica do código, permitindo assim correlacionar padrões de pensamento com práticas de codificação. Dessa forma, desempenham um papel fundamental na análise sistemática das abordagens adotadas no cotidiano dos estudantes de programação.

Além disso, entrevistas aprofundadas oferecem uma perspectiva única sobre as experiências individuais dos desenvolvedores. Elas permitem explorar de forma rica e detalhada as técnicas e estratégias aplicadas ao lidar com diferentes tipos de código. Essas interações qualitativas proporcionam um entendimento mais abrangente sobre como programadores abordam problemas, compreendem lógicas e enfrentam desafios de codificação.

2.2.5 Métricas experimentais na investigação da compreensão

Na investigação experimental da compreensão de código, diversas métricas têm sido empregadas para avaliar a eficiência e a eficácia das práticas de compreensão de código. O tempo de execução é uma métrica comum, pois mede a rapidez com que os programadores conseguem completar tarefas específicas. Essa medida reflete a eficiência cognitiva e prática na navegação e manipulação de trechos de código. Por outro lado, a acurácia avalia a precisão das respostas fornecidas pelos desenvolvedores, indicando a capacidade de interpretar corretamente a lógica durante tarefas práticas, como observado por Oliveira et al. (2021).

Técnicas avançadas, como rastreamento ocular (*eye tracking*) e ressonância magnética funcional (fMRI), têm ganhado destaque por sua capacidade de analisar o esforço visual e a atividade cerebral durante a leitura de código. Essas tecnologias revelam os pontos de maior atenção dos programadores e como o cérebro processa informações relacionadas ao código. Essa abordagem contribui para uma compreensão mais abrangente dos processos cognitivos envolvidos, oferecendo uma visão detalhada e científica sobre como o código é interpretado.

3 METODOLOGIA

3.1 *Grounded Theory*

Proposta por Glaser e Strauss (2008), *Grounded Theory* ou "teoria fundamentada" é uma metodologia qualitativa cujo modelo de investigação objetiva criar uma relação entre a teoria e a realidade estudada, com especial destaque ao papel ativo do investigador. Essa metodologia fundamenta-se no método da descoberta, através das condições contextuais em que os fenômenos ocorrem. As teorias devem ser determinadas com base em uma linha de investigação, na qual utiliza-se uma abordagem interativa na coleta e análise dos dados que lhes dão origem, organizado-os em uma sequência que tende para uma maior complexidade e integração. Logo, trata-se de um processo indutivo do conhecimento.

O processo começa com a identificação do problema de investigação. Nesta fase, o pesquisador registra todas as questões pertinentes, estabelecendo os limites do fenômeno em estudo. O objetivo principal é formular perguntas abertas que possibilitem uma análise flexível das respostas, permitindo também um estudo aprofundado e comprometido com as limitações do problema. Contudo, as próprias questões são dinâmicas e podem ser reavaliadas ao longo do tempo, não se esgotando no início da pesquisa. Nesse contexto, destaca-se o princípio fundamental da análise na *Grounded Theory*: o método de comparação constante. Este método envolve um movimento contínuo entre a análise do pesquisador e um retorno constante aos dados, até que o processo alcance um ponto de saturação.

À medida que o método avança, novas questões surgem e são validadas, passando de perguntas abertas para outras mais específicas e direcionadas ao contexto do estudo. Segundo Strauss e Corbin (2008), a construção da amostra se dá no decorrer da própria análise, questões e ideias vão se completando e tomando feição representativa das nuances do fenômeno, sendo direcionada aos dados. Consequentemente, é consistente analisar os dados coletados, por meio de entrevistas, por exemplo, conforme são obtidos, até que as categorias se estabilizem e novos eventos não acrescentem informações significativas ao estudo. No que tange à codificação, podemos identificar três tipos de abordagens: aberta, axial e seletiva.

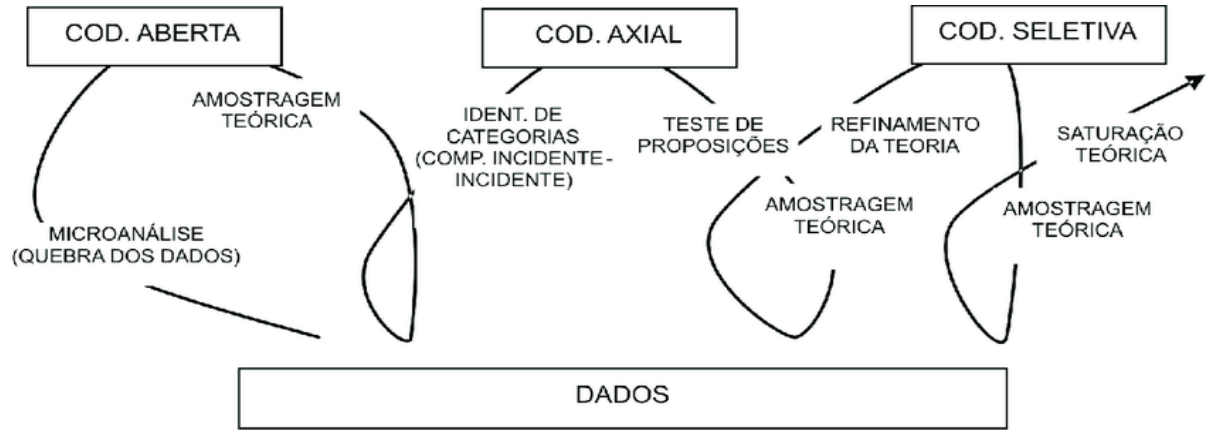
A codificação aberta envolve decompor os dados em unidades de análise, examiná-los, compará-los, conceituá-los e categorizá-los. Esse processo focaliza a atenção no fenômeno e facilita a construção do conhecimento indutivo. Inicialmente, os dados são fragmentados em elementos que representem o fenômeno – uma observação, uma pergunta, um evento – e são analisados, nomeados e conceituados conforme as respostas surgem. Em seguida, os conceitos desenvolvidos são agrupados em categorias com base em suas

similaridades. Portanto, a codificação aberta consiste na identificação de categorias e na definição flexível de suas propriedades e dimensões.

A codificação axial ocorre após a codificação aberta e envolve a reorganização dos dados já conceituados, indo além das propriedades e dimensões previamente definidas. Nesse estágio, uma categoria é escolhida como central, e as demais são subordinadas a ela, estabelecendo-se relações causais, contextuais ou intervenientes.

Por fim, a codificação seletiva é o processo de selecionar a categoria central em torno da qual as demais serão integradas. Embora semelhante à codificação axial, essa abordagem refina e representa os dados em um nível mais alto de abstração. Nesta fase, o pesquisador constrói a narrativa descritiva do fenômeno em estudo, integrando as categorias em uma teoria consolidada. Na Figura 6, apresenta-se uma síntese dos passos da Grounded Theory para um melhor entendimento.

Figura 6 - Processo de codificação do método da Grounded Theory



Fonte: Bandeira-de-Mello e Cunha (2006, com adaptações).

3.2 Método de pesquisa

Com o intuito de atender os objetivos e o foco de interesse desta pesquisa, a abordagem qualitativa foi escolhida para o seu desenvolvimento. Tal escolha levou também em consideração que essa abordagem possibilita a exploração da realidade de forma mais completa e profunda possível, destacando as relações e as estruturas nas quais estão inseridos os seres humanos investigados (Minayo, 2011).

A *Grounded Theory* é uma metodologia apropriada para investigar as percepções dos estudantes sobre a compreensão de código, pois permite desenvolver teorias baseadas em dados empíricos coletados diretamente com os participantes do estudo. Ao explorar as

experiências dos estudantes com códigos refatorados, é possível captar nuances e construir uma teoria fundamentada nas respostas e nos relatos dos participantes. Essa abordagem é especialmente adequada para estudar um fenômeno ainda pouco explorado, como a compreensão de código em um contexto educacional e profissional. Ao desenvolver uma teoria a partir dos dados, evitamos suposições prévias, permitindo que as percepções dos próprios desenvolvedores guiem as conclusões do estudo.

Essa postura investigativa encontra respaldo na perspectiva de Wazlawick (2021), que destaca a importância de observar a realidade tal como ela é vivida e significada pelos indivíduos. Para o autor, compreender um fenômeno demanda acesso direto às experiências e interpretações dos sujeitos envolvidos, uma vez que “a realidade de segunda ordem é constituída pelas construções humanas sobre o mundo” (WAZLAWICK, 2021).

Nesse contexto, este estudo se caracteriza como uma pesquisa primária de caráter exploratório, centrada na escuta ativa e na análise das percepções dos entrevistados. Tal abordagem permite acessar essas construções subjetivas e produzir conhecimento ancorado nas vivências concretas dos participantes, oferecendo uma compreensão mais sensível e situada do fenômeno investigado.

Para este estudo, adotamos uma abordagem inspirada na *Grounded Theory* de Strauss e Corbin (2008), que enfatiza a coleta e análise sistemática dos dados para construir teorias a partir dos mesmos. Segundo tais pesquisadores, essa abordagem permite identificar padrões e relacionamentos, organizando dados qualitativos em categorias através de processos de codificação aberta, axial e seletiva. Essa metodologia é caracterizada pela análise constante e interativa dos dados, um processo que permite a descoberta e o refinamento das categorias conforme novas informações surgem ao longo do estudo. A coleta de dados e a análise acontecem simultaneamente, de forma a aprimorar o desenvolvimento de uma teoria robusta e confiável, alinhada com as percepções dos participantes.

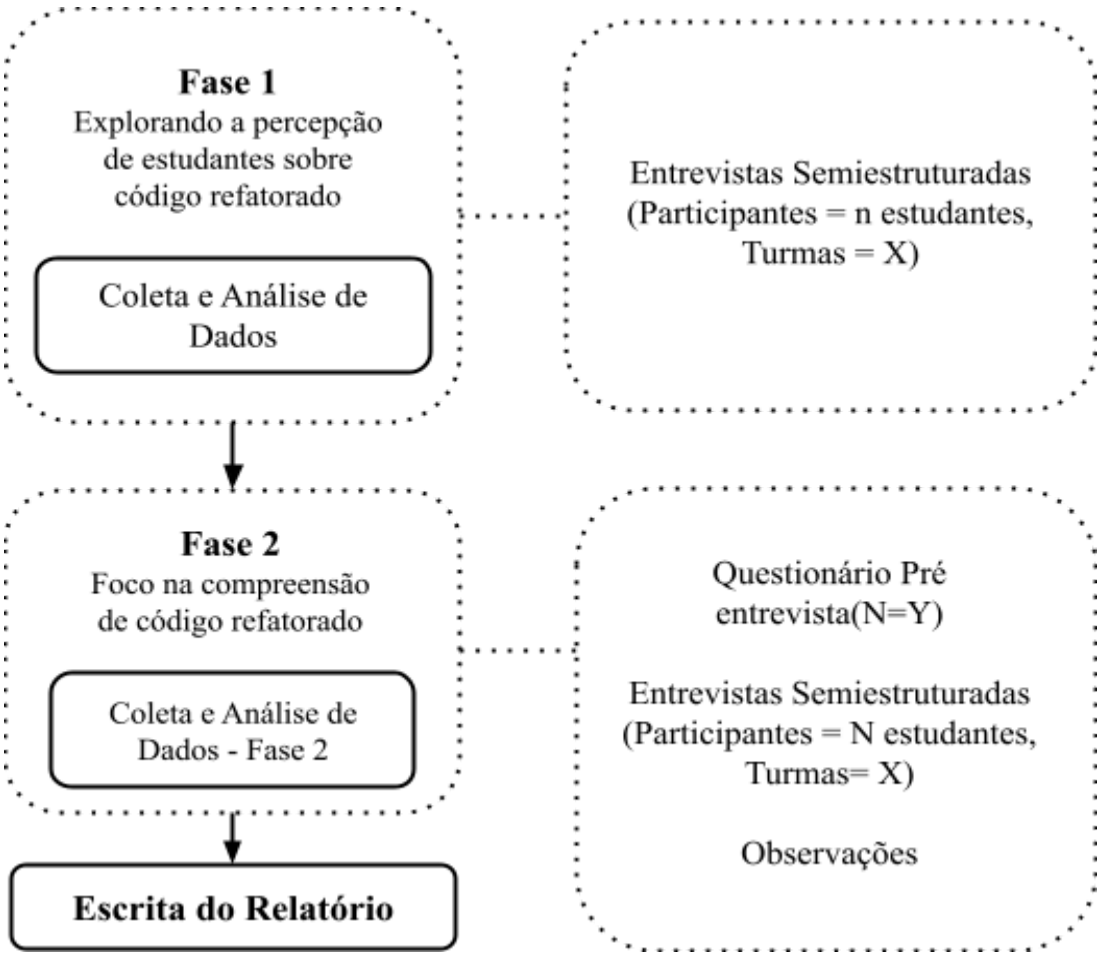
O processo de pesquisa seguirá o fluxo representado na Figura 7, com base na proposta de Masood (2020). Cada etapa de coleta e análise dos dados é representada no fluxo, com ênfase nas fases de codificação e validação dos dados com os participantes. Na Fase 1, o objetivo foi explorar a percepção dos estudantes sobre código refatorado. Nesse momento inicial, o foco estava em captar as opiniões, impressões e experiências gerais dos participantes em relação ao tema. Para isso, serão realizadas entrevistas semiestruturadas com um grupo de estudantes (representado como "n estudantes") de turmas específicas ("Turmas = X"). A coleta e análise dos dados dessa fase foram fundamentais para identificar os conceitos

iniciais e padrões emergentes relacionados às percepções dos estudantes, fornecendo embasamento para investigações mais aprofundadas.

A Fase 2 teve como objetivo aprofundar-se na compreensão dos estudantes sobre o código refatorado, expandindo e refinando as descobertas da Fase 1. Nessa etapa, houve a aplicação de questionários pré-entrevista (representado como "N = Y") para capturar informações iniciais que pudessem direcionar a coleta de dados subsequentes. Em seguida, as entrevistas semiestruturadas foram realizadas novamente com os mesmos respondentes ou com outros estudantes dependendo da necessidade, mantendo o foco nas mesmas turmas ("Turmas = X"). Além disso, observações foram incorporadas para compreender de forma prática como os estudantes interagem com o código refatorado em um contexto real. A análise dos dados dessa fase permitiu o desenvolvimento de categorias mais detalhadas e refinadas, possibilitando um entendimento mais aprofundado sobre o tema.

Por fim, todo o processo culminou na escrita do relatório, no qual os dados coletados e analisados nas fases 1 e 2 foram consolidados. Este relatório busca integrar as descobertas emergentes da análise qualitativa, apresentando os conceitos, categorias e teorias desenvolvidas a partir dos dados. Seguindo os princípios da *Grounded Theory*, a abordagem iterativa e indutiva garante que as informações coletadas direcionem a construção teórica de maneira fundamentada e consistente.

Figura 7 - Fluxograma do método de pesquisa inspirado na Grounded Theory



Fonte: Masood (2020, com adaptações).

3.3 Condução das entrevistas

Para a coleta de dados, foram realizadas entrevistas semiestruturadas com os estudantes, permitindo-lhes descrever suas experiências pessoais ao tentar compreender e manipular códigos refatorados. Esse formato de entrevista oferece flexibilidade para explorar diferentes aspectos da experiência dos participantes, enquanto fornece uma estrutura que facilita a análise dos dados coletados. As entrevistas serão gravadas e transcritas para análise.

As perguntas das entrevistas foram abertas para incentivar os participantes a discutir suas motivações, dificuldades e facilidades na compreensão de códigos. As questões exploratórias incluem:

1. Como você se sentiu ao ler esse código?
2. O que facilitou ou dificultou seu entendimento?
3. Você percebeu alguma diferença entre as versões?

4. Se tivesse que modificar esse código, qual versão preferiria? Por quê?
5. Que estratégias você usou para compreender o código?
6. Você já encontrou códigos semelhantes em sua experiência? Como lidou com eles?

Foram essas seis questões que nortearam o início das entrevistas, porém, em busca de resultados mais expressivos sobre os reais motivos por trás de algumas dificuldades enfrentadas pelos participantes houveram algumas mudanças no roteiro das entrevistas, de forma que foi acrescentado mais uma questão para os estudantes a partir do respondente número 5, e para os profissionais uma questão personalizada para cada um deles a partir do participante número 15. Para os estudantes de número 5 ao de número 10 foi a seguinte questão: Como você avalia sua formação em programação orientada a objetos?

Já para os profissionais, respondentes de número 15 ao número 20, as questões a serem respondidas foram:

Respondente número 15 - Como você vê a importância da legibilidade em códigos que outros profissionais vão reutilizar?

Respondente número 16 - Como o uso de testes automatizados influencia sua confiança para alterar um código?

Respondente número 17 - Em sua prática profissional, quais critérios você considera indispensáveis antes de aprovar um código em revisão?

Respondente número 18 - Que papel você acredita que a documentação de código desempenha em refatorações como essa?

Respondente número 19 - Como você equilibra a pressão por entregas com a necessidade de escrever código limpo?

Respondente número 20 - Como você mede o impacto de uma refatoração na qualidade do projeto?

Essas novas perguntas foram fruto do processo contínuo de comparação e análise dos dados coletados, inspirado na *Grounded Theory*. À medida que as entrevistas eram realizadas e os dados analisados, surgiam padrões e categorias que indicavam lacunas de compreensão ou aspectos ainda não suficientemente explorados. Assim, as novas questões foram elaboradas com base nos *insights* emergentes, com o objetivo de aprofundar a investigação e refinar a construção teórica de forma progressiva, respeitando a natureza iterativa e emergente do método.

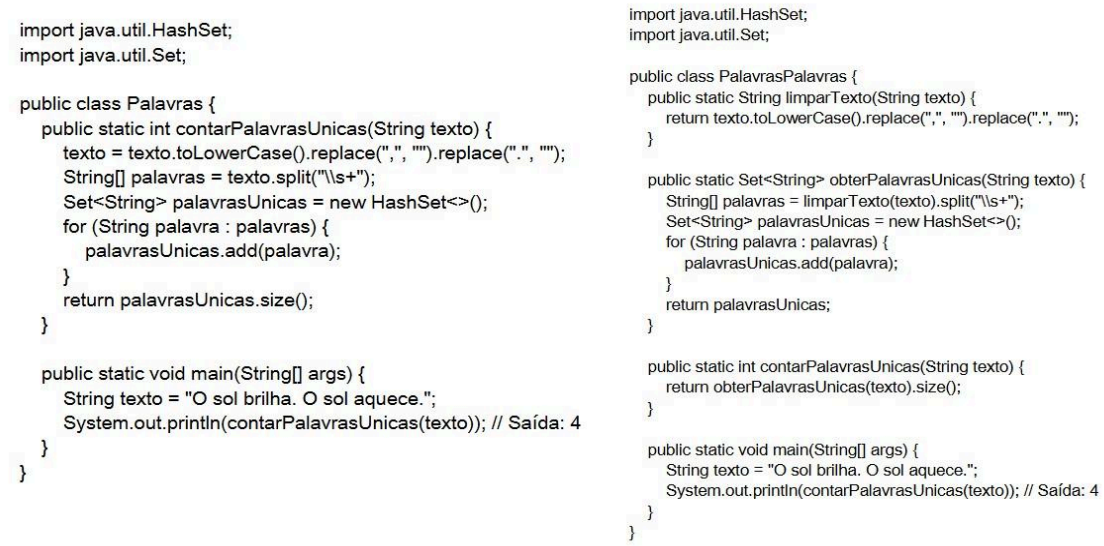
3.3.1 Elaboração das tarefas

Para garantir que os participantes pudessem compartilhar experiências ricas e detalhadas, foram elaborados trechos de código para os estudantes analisarem duas versões de cada vez: uma antes e outra após a refatoração. Cada participante teve a oportunidade de revisar trechos de código que passaram por melhorias estruturais e estilísticas. Essas questões permitiram avaliar como a refatoração influencia a compreensão e a capacidade de adaptação dos estudantes e profissionais em relação ao código apresentado.

Além disso, as respostas ajudaram a identificar aspectos positivos e negativos do código refatorado sob a perspectiva dos participantes. A seguir estão expostas as questões elaboradas que os respondentes analisaram durante a entrevista.

A Figura 8 a seguir foi a primeira comparação de códigos mostrada aos respondentes, exemplificando a comparação entre o código não refatorado à esquerda e o código após a refatoração utilizando o método extrair à direita.

Figura 8 - Primeira comparação de códigos



```
import java.util.HashSet;
import java.util.Set;

public class Palavras {
    public static int contarPalavrasUnicas(String texto) {
        texto = texto.toLowerCase().replace(",", "").replace(".", "");
        String[] palavras = texto.split("\\s+");
        Set<String> palavrasUnicas = new HashSet<>();
        for (String palavra : palavras) {
            palavrasUnicas.add(palavra);
        }
        return palavrasUnicas.size();
    }

    public static void main(String[] args) {
        String texto = "O sol brilha. O sol aquece.";
        System.out.println(contarPalavrasUnicas(texto)); // Saída: 4
    }
}
```

```
import java.util.HashSet;
import java.util.Set;

public class PalavrasPalavras {
    public static String limparTexto(String texto) {
        return texto.toLowerCase().replace(",", "").replace(".", "");
    }

    public static Set<String> obterPalavrasUnicas(String texto) {
        String[] palavras = limparTexto(texto).split("\\s+");
        Set<String> palavrasUnicas = new HashSet<>();
        for (String palavra : palavras) {
            palavrasUnicas.add(palavra);
        }
        return palavrasUnicas;
    }

    public static int contarPalavrasUnicas(String texto) {
        return obterPalavrasUnicas(texto).size();
    }

    public static void main(String[] args) {
        String texto = "O sol brilha. O sol aquece.";
        System.out.println(contarPalavrasUnicas(texto)); // Saída: 4
    }
}
```

Fonte: Elaborado pelo autor (2025).

Dando continuidade à entrevista, a segunda questão demonstrou a utilização do método inline, e desta vez, houve a alteração na ordem dos código: o código refatorado pelo método inline ficou à esquerda e o não refatorado à direita. A Figura 9 a seguir representa a nova questão problema.

Figura 9 - Segunda comparação de códigos

```
public class Primo {
    public static void main(String[] args) {
        int numero = 29;
        boolean primo = true;
        if (numero < 2) primo = false;
        for (int i = 2; i <= Math.sqrt(numero); i++) {
            if (numero % i == 0) {
                primo = false;
                break;
            }
        }
        System.out.println(primo); // Saída: true
    }
}
```

```
public class Primo {
    public static boolean ehPrimo(int n) {
        if (n < 2) return false;
        for (int i = 2; i <= Math.sqrt(n); i++) {
            if (n % i == 0) return false;
        }
        return true;
    }

    public static void main(String[] args) {
        int numero = 29;
        System.out.println(ehPrimo(numero)); // Saída: true
    }
}
```

Fonte: Elaborado pelo autor (2025).

Por fim, na terceira questão houve outra alteração na posição do código, sendo o da esquerda não refatorado e o da direita refatorado utilizando o método renomear, conforme a Figura 10 a seguir que exemplifica a situação problema apresentada aos participantes.

Figura 10 - Terceira comparação de códigos

```
public class ContaBancaria {
    private double saldo;

    public ContaBancaria(double saldoInicial) {
        saldo = saldoInicial;
    }

    public String isStatus() {
        if (saldo > 0) {
            return "Conta Ativa";
        } else if (saldo == 0) {
            return "Conta Zerada";
        } else {
            return "Conta Negativa";
        }
    }

    public static void main(String[] args) {
        ContaBancaria conta = new ContaBancaria(-50);
        System.out.println(conta.isStatus());
    }
}
```

```
public class ContaBancaria {
    private double saldo;

    public ContaBancaria(double saldoInicial) {
        saldo = saldoInicial;
    }

    public String obterStatus() {
        if (saldo > 0) {
            return "Conta Ativa";
        } else if (saldo == 0) {
            return "Conta Zerada";
        } else {
            return "Conta Negativa";
        }
    }

    public static void main(String[] args) {
        ContaBancaria conta = new ContaBancaria(-50);
        System.out.println(conta.obterStatus());
    }
}
```

Fonte: Elaborado pelo autor (2025).

3.3.2 Seleção de respondentes

Os respondentes foram contatados por meio de convites feitos em ambientes acadêmicos oficiais e plataformas digitais utilizadas pelos estudantes, como grupos de

discussão online e redes sociais específicas do curso de Ciência da Computação, locus de realização do presente estudo.

A seleção buscou incluir alunos matriculados em disciplinas de programação ou relacionados à Engenharia de Software, garantindo que eles tivessem experiência prática com a linguagem Java e a lógica mínima para um possível entendimento ou resolução das questões propostas. A diversidade de habilidades e níveis de conhecimento entre os participantes será um fator decisivo durante a investigação do código gerado, de forma que a inclusão de alunos de diferentes semestres fornecerá uma perspectiva abrangente sobre a compreensão de código refatorado.

O processo de seleção, portanto visou recrutar um grupo que representasse diversas abordagens e experiências com a compreensão de código, possibilitando a construção de uma teoria rica e fundamentada sobre as percepções dos estudantes sobre a prática de refatoração e metodologia de programação aplicada nos diferentes períodos do curso.

3.4 Análise de dados

Conforme descrito anteriormente na Figura 7, foi aplicada a codificação aberta nas transcrições das entrevistas, identificando temas principais e dividindo o conteúdo em unidades de significado. Essa fase permitiu fragmentar os dados em categorias iniciais sem qualquer estrutura pré definida, com o objetivo de capturar a variedade de percepções dos estudantes sobre a compreensão de códigos refatorados.

Em seguida, foi utilizada a codificação axial para reorganizar as categorias identificadas, formando subcategorias e relacionando-as para estruturar os temas principais que emergem dos dados. Esse processo permitiu analisar como as categorias estão interligadas, facilitando a compreensão das conexões entre as experiências relatadas pelos estudantes.

Na fase de codificação seletiva, todas as categorias principais foram integradas em um modelo teórico, enfatizando os fatores mais relevantes para o estudo. Essa etapa teve como objetivo construir uma teoria final sobre as percepções dos participantes, destacando como eles interpretaram e compreenderam o código refatorado e os aspectos que mais influenciaram na sua experiência.

4 RESULTADOS E DISCUSSÕES

A partir da análise qualitativa das respostas das entrevistas com 20 respondentes, por meio do método de análise inspirado na *Grounded Theory*, emergiram sete categorias centrais de *selective coding* (código seletivo) que orientaram a compreensão dos dados. O código seletivo é a etapa em que o pesquisador integra e refina as categorias principais desenvolvidas nas fases anteriores (open coding e axial coding). Nesse momento, busca-se identificar uma categoria central, que representa o fenômeno principal da pesquisa, e relacioná-la às demais categorias, organizando os dados em uma explicação teórica coerente e abrangente.

A convergência entre os dois grupos de participantes (estudantes e profissionais) revelou que há aspectos compartilhados na experiência com códigos refatorados, enquanto outras dimensões evidenciam diferenças importantes, sobretudo ligadas à maturidade técnica e à bagagem prática.

4.1 Compreensão mediada por estrutura

Tanto estudantes quanto profissionais apontaram que estruturas claras e segmentadas do código facilitam a leitura e compreensão. Métodos coesos, nomes descritivos e responsabilidades bem definidas contribuem para a fluidez da leitura e reduzem o esforço cognitivo. Esse fator demonstra o impacto direto da refatoração sobre a legibilidade e a navegabilidade do código, fortalecendo o entendimento de que a clareza estrutural é um facilitador universal, independente do nível de experiência. O Quadro 1 a seguir foi extraído da tabela de análise de dados e codificações.

Para chegar à categoria “Compreensão mediada por estrutura”, as transcrições das respostas foram inicialmente organizadas em uma planilha com colunas específicas para as falas dos participantes, codificações abertas e observações preliminares. As falas que indicavam sensações de clareza, conforto, facilidade de leitura ou que destacavam aspectos como a divisão em métodos, nomes descritivos e coesão funcional foram marcadas com códigos abertos como “clareza com divisão em métodos”, “leitura mais fluida” e “benefício da nomeação clara”. Essa etapa foi essencial para mapear os elementos que os participantes associavam diretamente à forma como o código estava estruturado.

Na fase de codificação axial, esses códigos abertos foram agrupados em torno de eixos temáticos, como “vantagens da nomeação e estruturação” e “alívio cognitivo proporcionado por código limpo”. Esses eixos demonstravam que, para os participantes, a estrutura do código funcionava como uma mediação para a compreensão. A construção do

selective coding foi então orientada por essa constatação recorrente: o impacto direto da refatoração sobre a compreensão do código era percebido por ambos os grupos, revelando-se como um fenômeno transversal e central.

Quadro 1 - Compreensão mediada por estrutura

Resposta	Open Coding	Axial Coding	Selective Coding
Me senti bem mais confortável com a versão refatorada. A divisão em métodos deixou tudo mais claro.	Conforto ao ler código; Clareza com divisão em métodos	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura
Fiquei meio confuso com o nome isStatus. Parecia outra coisa.	Nomeação confusa; Interpretação incorreta	Impacto da má nomeação	Compreensão mediada por estrutura
Me senti menos cansado com a versão refatorada. Parecia que o código respirava.	Cansaço reduzido; Leitura mais fluida	Alívio cognitivo proporcionado por código limpo	Compreensão mediada por estrutura
A terminologia técnica dos métodos ajuda quando bem nomeada.	Benefício da terminologia clara; Importância da nomeação	Vantagens da nomeação	Compreensão mediada por estrutura
Dificultou o acoplamento de responsabilidades num único método. Quando está tudo misturado, exige mais energia cognitiva.	Acoplamento de responsabilidades; Sobrecarga cognitiva	Dificuldade com métodos sobrecarregados	Compreensão mediada por estrutura
Faltou clareza na delimitação de responsabilidades. A versão refatorada alinha melhor com padrões de projeto.	Falta de delimitação; Aderência a padrões	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura

Fonte: Elaborado pelo autor (2025).

4.2 Limitações de formação técnica

Em ambos os grupos, limitações técnicas influenciaram negativamente a compreensão do código, mas se manifestaram de formas distintas. Estudantes relataram uma formação precária em Programação Orientada a Objetos, marcada por instabilidade docente e ausência de prática consistente. Já entre os profissionais, as limitações aparecem mais ligadas a lacunas específicas ou ao acúmulo de más práticas ao longo da carreira. O Quadro 2 exemplifica a presença dessa categoria em ambos os perfis e reforça a necessidade de uma formação mais sólida e contínua, tanto no contexto educacional quanto no profissional.

A categoria “Limitações de formação técnica” emergiu com clareza já nas primeiras rodadas de codificação aberta, especialmente a partir das respostas dos estudantes. Frases como “ficamos sem professor”, “não conseguimos fixar bem Java”, ou “não aprendemos orientação a objetos direito” foram sistematicamente agrupadas com códigos como

“formação prejudicada”, “ausência de orientação” e “falta de domínio conceitual”. Esses códigos foram associados a evidências de fragilidade pedagógica e lacunas no ensino formal.

Na codificação axial, identificou-se que essas falas giravam em torno de eixos como “instabilidade acadêmica” e “aprendizagem prejudicada”. As categorias axiais foram fundamentais para perceber que a dificuldade na leitura de código não estava ligada apenas ao conteúdo do código, mas às limitações prévias de conhecimento técnico dos participantes. A construção do selective coding “Limitações de formação técnica” foi assim sustentada por esse conjunto coerente de códigos que apontavam para os problemas enfrentados na formação acadêmica.

Quadro 2 - Limitações de formação técnica

Resposta	Open Coding	Axial Coding	Selective Coding
A verdade é que na disciplina de orientação a objetos a gente ficou sem professor metade do semestre.	Formação prejudicada; Ausência de orientação	Instabilidade acadêmica	Limitações de formação técnica
Dificultou bastante o fato de não termos tido um professor fixo em Programação Orientada a Objetos.	Falta de professor fixo; Dificuldade com base teórica	Instabilidade acadêmica	Limitações de formação técnica
Não saber usar bem os conceitos de classe e método.	Dificuldade com POO; Falta de domínio conceitual	Aprendizagem prejudicada	Limitações de formação técnica
Vi sim, mas até pra perceber isso eu demorei porque não sabia como os métodos funcionavam.	Dificuldade em perceber mudança; desconhecimento dos métodos	Má formação técnica	Limitações de formação técnica
Sim, mas não entendi o porquê da mudança.	Reconhecimento de diferença; ausência de justificativa clara	Instabilidade acadêmica	Limitações de formação técnica
Sim, a segunda pareceu melhor, mas ainda difícil pra mim.	Preferência pela refatorada; persistência da dificuldade	Instabilidade acadêmica	Limitações de formação técnica

Fonte: Elaborado pelo autor (2025).

4.3 Facilidade de manutenção

A refatoração foi associada à maior facilidade de manutenção e evolução do sistema, especialmente entre os profissionais, no Quadro 3 a seguir, destacam-se aspectos como testabilidade, reaproveitamento de funções e colaboração em equipe. Estudantes também reconheceram essas vantagens, mas de forma mais teórica ou intuitiva, muitas vezes acompanhada por insegurança técnica para aplicar tais conceitos. Essa discrepância indica que o ensino de refatoração precisa ir além da teoria e promover experiências práticas de manutenção.

A categoria “Facilidade de manutenção” surgiu da recorrência de comentários sobre modularidade, segurança ao modificar o código, reutilização de métodos e testabilidade. No processo de codificação aberta, essas ideias foram traduzidas em termos como “segurança para manutenção”, “testabilidade” e “risco reduzido”. A planilha permitiu visualizar como essas respostas se agrupavam em torno de práticas que facilitam a evolução de sistemas, revelando uma preocupação mais prática, especialmente entre os profissionais.

Na fase de codificação axial, essas expressões foram consolidadas em categorias intermediárias, como “segurança para manutenção” e “reutilização e testes”, que formavam um padrão consistente: a refatoração não era apenas uma melhoria estética, mas instrumental para manutenção e escalabilidade. A partir dessas relações, a codificação seletiva foi determinada com base na frequência e centralidade dessas vantagens percebidas, culminando na categoria “Facilidade de manutenção”.

Quadro 3 - Facilidade de manutenção

Resposta	Open Coding	Axial Coding	Selective Coding
Com certeza a refatorada. A modularização me dá mais segurança pra mexer.	Modularidade, segurança	Segurança para manutenção	Facilidade de manutenção
A com extração de métodos. Reduz a chance de quebrar algo por engano.	Risco reduzido	Segurança para manutenção	Facilidade de manutenção
A refatorada. Daria pra reaproveitar funções menores e facilitar mocks em testes.	Reutilização, testabilidade	Testabilidade	Facilidade de manutenção
A refatorada, sem dúvida. É mais modular e dá menos medo de mexer.	Modularidade, confiança	Segurança para manutenção	Facilidade de manutenção

A com nome obterStatus, me economizou tempo.	Nomeação clara, economia de tempo	Clareza semântica	Facilidade de manutenção
A segunda. Dá pra manter com mais facilidade.	Manutenibilidade de código refatorado	Manutenção mais simples	Facilidade de manutenção

Fonte: Elaborado pelo autor (2025).

4.4 Estratégias cognitivas

Os participantes recorreram a diferentes estratégias cognitivas para compreender o código como ilustrado, como leitura por nomes de métodos, analogias com estruturas conhecidas, construção de diagramas mentais e decomposição em responsabilidades. Como ilustrado no Quadro 4, tais estratégias revelam como a leitura de código não é apenas um processo técnico, mas também cognitivo, exigindo habilidades de abstração e raciocínio lógico. As estratégias utilizadas refletem diferentes graus de maturidade, sendo mais articuladas e metódicas entre profissionais.

Durante a análise aberta, várias falas mencionaram formas particulares de leitura de código, como leitura linha por linha com anotações, análise *top-down* ou leitura por blocos funcionais. Esses comportamentos foram registrados com códigos como “apoio externo”, “abordagem *top-down*” e “mapeamento funcional”. O uso de diferentes estratégias revelou não apenas a diversidade de formas de leitura, mas também pistas sobre o nível de maturidade e abstração de cada participante.

A codificação axial organizou essas estratégias em torno de categorias como “representação externa” e “análise hierárquica”. Essa fase evidenciou que compreender e analisar código exige muito mais do que conhecer a sintaxe, exige artifícios cognitivos que são muitas vezes invisíveis, como uma espécie de *feeling* (do inglês, sentimento) que norteia o conhecimento empírico nesses momentos de resolução de problemas. O código seletivo “Estratégias cognitivas” sintetiza essa articulação entre os métodos de leitura utilizados e o esforço cognitivo individual, revelando que compreender código é um processo ativo, que envolve abstração e táticas pessoais.

Quadro 4 - Estratégias cognitivas

Resposta	Open Coding	Axial Coding	Selective Coding
Leitura linha por linha com anotações no papelzinho.	Apoio externo	Representação externa	Estratégias cognitivas
Divido a leitura por métodos e tento mapear a função de cada um.	Mapeamento funcional	Leitura orientada a estrutura	Estratégias cognitivas
Análise top-down. Começo pela visão geral, depois desço nos métodos.	Abordagem top-down	Análise hierárquica	Estratégias cognitivas
Fui tentando imaginar o que o método fazia antes de ler o conteúdo.	Inferência semântica prévia	Antecipação por nomeação	Estratégias cognitivas
Eu fui destacando blocos na minha cabeça.	Agrupamento mental de blocos	Organização cognitiva	Estratégias cognitivas
Leio os nomes primeiro. Eles me guiam.	Nomeação como guia	Organização cognitiva	Estratégias cognitivas

Fonte: Elaborado pelo autor (2025).

4.5 Experiência prévia com complexidade

A familiaridade com códigos complexos influenciou diretamente a capacidade de interpretar estruturas refatoradas. Estudantes, em geral, demonstraram menor exposição a projetos reais ou códigos em produção, o que os torna mais vulneráveis a dificuldades de abstração. Por outro lado, profissionais relataram experiências passadas com manutenção de legados ou projetos colaborativos, o que os tornou mais aptos a reconhecer padrões e boas práticas. O Quadro 5 a seguir exemplifica bem como a experiência impacta na resposta dos participantes.

A presença de termos como “códigos legados”, “refatorar aos poucos”, “uso de TDD” e “refatoração contínua” conduziu à codificação aberta inicial com categorias como “refatoração incremental”, “boas práticas” e “abordagem gradual”. Ao revisar essas codificações, percebeu-se que elas compartilhavam uma base comum: todas apontavam para experiências anteriores que treinavam o olhar para estruturas complexas.

Na codificação axial, esses elementos foram organizados em torno de “estratégia de manutenção” e “estratégias ágeis”, revelando que a complexidade do código é mais bem manejada por quem já vivenciou sistemas robustos através de estágios ou da própria rotina de

trabalho. A codificação seletiva “Experiência prévia com complexidade” emergiu a partir da repetição dessa competência narrativa, onde os participantes mais experientes demonstravam segurança e familiaridade com abordagens refinadas de refatoração e manutenção.

Quadro 5 - Experiência prévia com complexidade

Resposta	Open Coding	Axial Coding	Selective Coding
Sim, especialmente em sistemas legados. Refatorar aos poucos ajuda muito.	Refatoração incremental	Estratégia de manutenção	Experiência prévia com complexidade
Sim, uso TDD e refatoração contínua como estratégia.	Boas práticas	Estratégias ágeis	Experiência prévia com complexidade
Sempre. O ideal é refatorar em etapas com cobertura de testes.	Abordagem gradual	Estratégia de manutenção	Experiência prévia com complexidade
Muitas vezes. Geralmente, proponho refatorações incrementais e documentadas.	Estratégia organizada	Estratégia de manutenção	Experiência prévia com complexidade
Sim, uso sempre pair programming quando o código é muito fechado.	Colaboração em equipes de trabalho	Apoio colaborativo	Experiência prévia com complexidade
Sim, em refatorações grandes usamos ferramentas como SonarQube e revisões em camadas.	Ferramentas de suporte	Engenharia de qualidade	Experiência prévia com complexidade

Fonte: Elaborado pelo autor (2025).

4.6 Formação fraca e desestruturada vs. Manutenção e escalabilidade

Essa categoria revelou uma forte distinção entre os dois grupos. Os estudantes expressaram de forma recorrente uma insatisfação com a formação recebida em POO, marcada pela descontinuidade e superficialidade dos conteúdos. Já os profissionais demonstraram maior preocupação com aspectos como escalabilidade e sustentabilidade do código a longo prazo. Essa diferença visualizada na Tabela 6, sugere que há uma lacuna crítica entre a formação acadêmica e as demandas reais do mercado de trabalho, reforçando a urgência de um alinhamento curricular mais efetivo com as práticas da indústria.

Essa categoria foi inicialmente percebida como duas trilhas distintas: de um lado, estudantes relatando dificuldades educacionais; do outro, profissionais destacando preocupações com manutenibilidade e colaboração em larga escala. Na codificação aberta, foram gerados códigos como “instabilidade docente”, “ensino superficial”, “manutenção futura” e “colaboração em times grandes”.

A codificação axial organizou esses códigos em duas dimensões complementares: “instabilidade na docência” e “segurança para manutenção”. O cruzamento desses eixos revelou uma tensão formativa: enquanto estudantes lidam com lacunas de base, profissionais já operam em contextos que exigem visão sistêmica e práticas escaláveis. Essa oposição fundamentou a criação da codificação seletiva “Formação fraca e desestruturada vs. Manutenção e escalabilidade”, pois ela sintetiza a dissociação entre o que é ensinado e o que é demandado na vida profissional.

Quadro 6 - Formação fraca e desestruturada vs. Manutenção e escalabilidade

Resposta	Open Coding	Axial Coding	Selective Coding
A refatorada é mais compatível com manutenção em times grandes.	Compatibilidade com equipes maiores; manutenção	Segurança para manutenção	Manutenção e escalabilidade
Sim, a refatorada favorece a manutenção futura e a colaboração.	Manutenção futura; colaboração	Manutenção e escalabilidade	Manutenção e escalabilidade
Fraca. A gente teve muita troca de professor, e quando tinha aula era bem rasa. Não deu tempo de fixar Java direito.	Instabilidade docente, ensino superficial, falta de consolidação dos conteúdos	Instabilidade na docência	Formação fraca e desestruturada
Falhou muito. Foram vários professores substitutos e conteúdos que não avançaram.	Troca de professores, ausência de continuidade no conteúdo	Instabilidade na docência	Formação fraca e desestruturada
Muito fraca. Os professores não explicavam, e a maioria das aulas era leitura de slide.	Ensino passivo, ausência de explicações claras	Falta de planejamento pedagógico	Formação fraca e desestruturada

Fonte: Elaborado pelo autor (2025).

4.7 Teoria

A partir da integração dos selective coding, emergiu a categoria central: "Compreensão de código como fenômeno mediado por fatores como estrutura e experiência, condicionada pela qualidade da formação". Essa categoria, representa uma síntese da teoria desenvolvida: a clareza estrutural do código facilita a compreensão, mas seu impacto é condicionado pela bagagem técnica e pelas experiências prévias dos leitores. A formação frágil compromete esse processo, enquanto experiências práticas ampliam a capacidade de interpretar, manter e evoluir sistemas de forma mais eficiente.

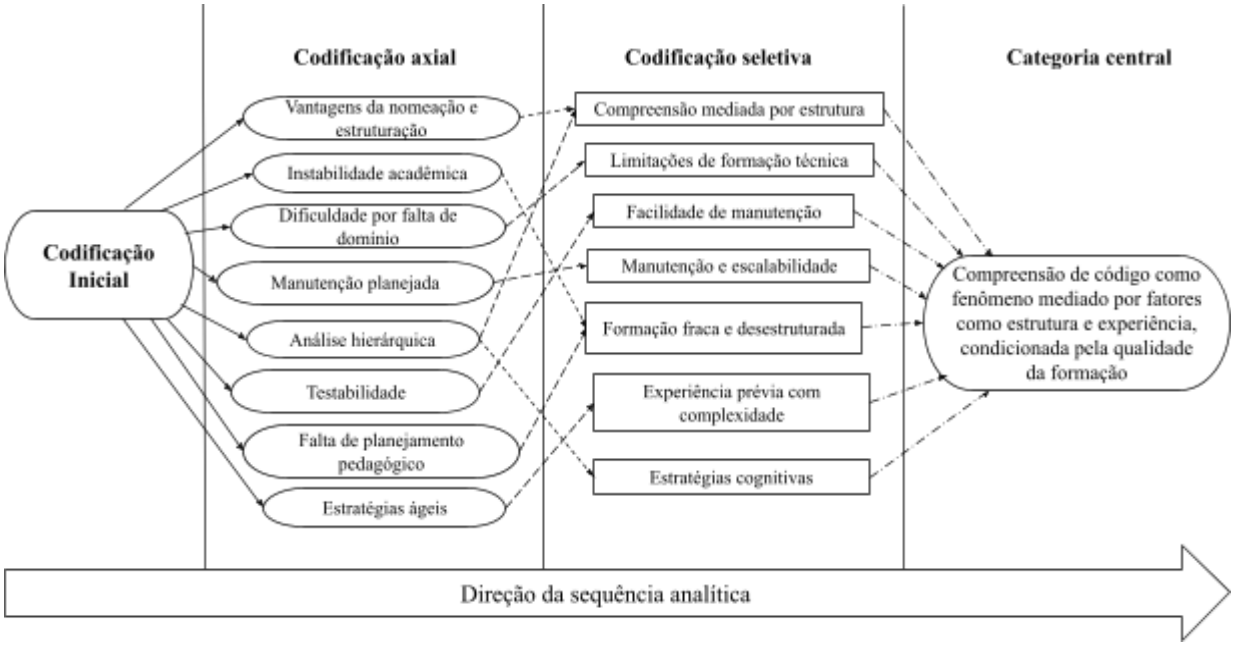
A pesquisa mostra que a estrutura do código, especialmente quando bem definida por meio de técnicas de refatoração, extração de métodos e nomes claros, beneficia diretamente a leitura e compreensão de código. Códigos organizados, com funções pequenas e bem nomeadas, são percebidos como mais fáceis de entender, modificar e testar. Essa clareza estrutural não só reduz a carga cognitiva, mas também aumenta a confiança dos leitores em relação à modificação e manutenção do sistema, como indicado por diversos participantes da pesquisa.

Essa percepção positiva não ocorre de forma universal ou automática. Ela é condicionada pela formação técnica e pelas experiências anteriores dos indivíduos. Aqueles com formação fraca ou desestruturada, especialmente estudantes que relataram dificuldades relacionadas à forma como as disciplinas de programação orientada a objetos foram ensinadas, demonstraram mais insegurança e resistência mesmo diante de códigos mais bem organizados. Esse achado indica que a capacidade de aproveitar os benefícios da refatoração do código depende de pré-requisitos formativos que nem sempre estão presentes no contexto acadêmico avaliado.

Por outro lado, os participantes com maior experiência prática, sejam profissionais ou estudantes com vivências extracurriculares, relataram mais facilidade em interpretar e interagir com código refatorado. Isso demonstra que a experiência prévia funciona como catalisador da compreensão, reforçando a base de conhecimento prático e conceitual para reconhecer padrões, antecipar comportamentos e abstrair lógicas que não estão explicitamente visíveis. Em outras palavras, a estrutura do código ajuda, mas o indivíduo que compreende precisa saber como ler essa estrutura, e isso é ensinado (ou não) pela formação e aprendido (ou não) pela prática.

A compreensão de código refatorado não é uma habilidade isolada, mas uma prática situada que se desenvolve na interseção entre clareza estrutural, qualidade da formação e vivência prática. Essa relação dinâmica entre os elementos ajuda a explicar por que o mesmo código pode ser considerado fácil para alguns e desafiador para outros. Na figura 11 a seguir, pode-se ver como as codificações se interconectam entre si e trazem à tona a categoria central do estudo levantado.

Figura 11 - O processo de codificação na TFD



Fonte: Elaborada pelo autor (2025).

5 CONSIDERAÇÕES FINAIS

Este trabalho buscou compreender os fatores que influenciam a leitura e a compreensão de código refatorado por estudantes de Ciência da Computação e por programadores profissionais, utilizando uma inspiração na Grounded Theory como referencial metodológico. A partir da coleta e análise qualitativa dos dados, foi possível identificar elementos centrais que impactam diretamente na forma como diferentes perfis de desenvolvedores interpretam e interagem com trechos de código.

A principal descoberta desta pesquisa está na relevância da refatoração do código como eixo orientador da compreensão. Tanto estudantes quanto profissionais destacaram que a presença de nomes descritivos, extração de métodos e modularização favorece significativamente a clareza semântica e a manutenção do código. Contudo, enquanto os profissionais conseguem se beneficiar dessas boas práticas com maior autonomia, os estudantes demonstram limitações decorrentes de formações técnicas frágeis, o que prejudica sua capacidade de explorar essas vantagens de maneira plena.

Além disso, os dados revelaram um conjunto de estratégias cognitivas distintas adotadas pelos participantes para entender o código, como leitura top-down, analogias com práticas conhecidas e divisão lógica por responsabilidades. Tais estratégias evidenciam que, embora a experiência prévia com complexidade seja um diferencial importante, o suporte estrutural do código atua como um facilitador essencial da compreensão para ambos os grupos.

Outra contribuição importante está na constatação de que, para os estudantes, a dificuldade de entendimento está muitas vezes relacionada à experiência educacional desestruturada, marcada por lacunas no ensino de programação orientada a objetos, como identificado nas respostas à pergunta sobre como avaliavam sua formação em programação orientada a objetos. Já entre os profissionais, os relatos indicam que a compreensão e manutenção do código estão fortemente vinculadas a critérios como legibilidade, escalabilidade e colaboração em equipe.

Dessa forma, esta pesquisa evidencia a necessidade de uma maior integração entre práticas de refatoração e estratégias pedagógicas no ensino de programação. É fundamental que o ambiente acadêmico não apenas ensine os conceitos técnicos, mas também promova a reflexão crítica sobre a legibilidade, reutilização e manutenção do código como parte essencial da formação de futuros desenvolvedores.

Como trabalhos futuros, sugere-se a ampliação da pesquisa para outros contextos, incluindo profissionais de áreas específicas de atuação e estudantes de diferentes instituições, a fim de verificar se os padrões identificados se mantêm em realidades diversas. Por fim, esta investigação reforça o valor da pesquisa qualitativa primária no campo da Ciência da Computação, pois permitiu acessar diretamente as percepções e experiências dos participantes, construindo uma teoria fundamentada em suas vivências concretas.

Os resultados obtidos contribuem para o debate sobre o ensino de programação, e também oferecem subsídios práticos para melhorias curriculares e metodológicas nas instituições de ensino superior, além de influenciar práticas de desenvolvimento em contextos profissionais.

REFERÊNCIAS

BANDEIRA-DE-MELLO, R.; CUNHA, C. J. C. de A. Grounded theory. In: GODOI, C. K.; BANDEIRA-DE-MELLO, R.; SILVA, A. B. da (org.). **Pesquisa qualitativa em estudos organizacionais: paradigmas, estratégias e métodos**. São Paulo: Saraiva, 2006. p. 253.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software architecture in practice**. 3. ed. Boston: Addison-Wesley, 2013.

BECK, K. **Extreme programming explained: embrace change**. 2. ed. Boston: Addison-Wesley, 2000.

BROOKS, R. E. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, v. 18, n. 6, p. 543-554, 1983.

CUNNINGHAM, W. The WyCash portfolio management system. *OOPSLA'92 Experience Report*, 1992.

FEITELSON, D. G. **Workload modeling for computer systems performance evaluation**. Cambridge: Cambridge University Press, 2021.

FOWLER, M. **Refactoring: improving the design of existing code**. 2. ed. Boston: Addison-Wesley, 2018.

FREIRE, A.; CHEQUE, P. Refatoração: melhorando a qualidade de código pré-existente. In: CURSO DE VERÃO, 2007, São Paulo. Anais [...]. São Paulo: IME/USP, 2007. Disponível em: http://ccsl.ime.usp.br/agilcoop/files/2-Refatoracao.pdf. Acesso em: 20 ago. 2024.

FUNDAÇÃO DE AMPARO À PESQUISA DO ESTADO DE SÃO PAULO (FAPESP). Número de matriculados em Ciência da Computação cresce 50% em cinco anos no Brasil. *Agência FAPESP*, 10 jun. 2022. Disponível em: https://agencia.fapesp.br/numero-de-matriculados-em-ciencia-da-computacao-cresce-50-em-cinco-anos-no-brasil/39215/. Acesso em: 10 jan. 2025.

MARTIN, R. C. **Clean code: a handbook of agile software craftsmanship**. Boston: Prentice Hall, 2009.

MASOOD, Z.; HODA, R.; BLINCOE, K. How agile teams make self-assignment work: a grounded theory study. *Empirical Software Engineering*, v. 25, n. 6, p. 4962-5005, 2020.

MENS, T. **Software evolution**. 2. ed. Cham: Springer, 2021.

MINAYO, M. C. S. Trabalho de campo: contexto de observação, interação e descoberta. In: MINAYO, M. C. S.; DESLANDES, S. F.; GOMES, R. (org.). **Pesquisa social: teoria, método e criatividade**. 34. ed. Petrópolis: Vozes, 2011.

NARDI, B. A. **A small matter of programming: perspectives on end user computing**. Cambridge: MIT Press, 1993.

OLIVEIRA, A. B.; FERREIRA, M. J.; COSTA, R. L. Cognitive processes in software comprehension. *Journal of Software Engineering and Applications*, v. 14, n. 12, p. 558-575, 2021.

PRESSMAN, R. S. **Engenharia de software: uma abordagem profissional**. 8. ed. Porto Alegre: AMGH, 2019.

SOMMERVILLE, I. **Engenharia de software**. 9. ed. São Paulo: Pearson, 2011.

SOLOWAY, E.; EHRLICH, K. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, v. SE-10, n. 5, p. 595-609, 1984.

STOREY, M. A. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, v. 14, n. 3, p. 187-208, 2006.

STRAUSS, A.; CORBIN, J. **Pesquisa qualitativa: técnicas e procedimentos para o desenvolvimento de teoria fundamentada**. 2. ed. Porto Alegre: Artmed, 2008.

VON MAYRHAUSER, A.; VANS, A. M. Program understanding behavior during debugging of large scale software. In: EMPIRICAL STUDIES OF PROGRAMMERS, 1995. Proceedings [...]. p. 124-144, 1995.

WAZLAWICK, P.; BEAVIN, J. H.; JACKSON, D. D. **A realidade da realidade: confusão, desinformação, comunicação**. 4. ed. São Paulo: Cultrix, 2021.

APÊNDICE - Tabela de codificação inspirada na GT

Participante	Nº Pergunta	Resposta	Open Coding	Axial Coding	Selective Coding
P1	1	Me senti bem mais confortável com a versão refatorada. A divisão em métodos deixou tudo mais claro.	Clareza com divisão em métodos	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura
P2	1	Fiquei meio confuso com o nome isStatus. Parecia outra coisa.	Nomeação confusa; Interpretação incorreta	Impacto da má nomeação	Compreensão mediada por estrutura
P3	1	Me senti menos cansado com a versão refatorada. Parecia que o código respirava.	Cansaço reduzido; Leitura mais fluida	Alívio cognitivo proporcionado por código limpo	Compreensão mediada por estrutura
P4	1	Tranquilo até. Achei os nomes bons.	Nomeação adequada; Tranquilidade na leitura	Clareza por boa nomeação	Compreensão mediada por estrutura
P5	1	Fiquei perdido. Não é só o código em si, mas acho que nunca tive uma base boa em Java.	Desorientação; Falta de base em linguagem	Déficit de formação técnica	Limitações de formação técnica
P6	1	Me senti inseguro. A linguagem ainda me é difícil.	Insegurança; Dificuldade com linguagem	Déficit de formação técnica	Limitações de formação técnica

P7	1	Com muita dificuldade. Senti que me faltava base.	Dificuldade de compreensão; Falta de formação	Falta de conhecimento	Limitações de formação técnica
P8	1	Frustrado. Eu não sabia por onde começar.	Frustração; Falta de direcionamento	Falta de estrutura clara	Compreensão mediada por estrutura
P9	1	Perdido. Não entendi o objetivo logo de cara.	Confusão com o propósito do código	Ambiguidade na intenção do código	Compreensão mediada por estrutura
P10	1	Tive dificuldade, especialmente com os nomes dos métodos.	Nomes dos métodos geram confusão	Nomeação inadequada	Compreensão mediada por estrutura
P11	1	O código sem refatoração me deixou um pouco apreensivo. Estava muito denso. A versão refatorada trouxe mais clareza.	Apreensão; Densidade; Clareza pós-refatoração	Complexidade do código sem refatoração	Compreensão mediada por estrutura
P12	1	Tranquilo com a versão refatorada. A outra é mais difícil de dar manutenção.	Tranquilidade; Dificuldade de manutenção em código original	Manutenibilidade do código	Compreensão mediada por estrutura
P13	1	A versão sem refatorar me incomodou. A refatorada estava mais alinhada com boas práticas	Incômodo; Alinhamento com boas práticas	Aderência a padrões de projeto	Compreensão mediada por estrutura

P14	1	Versão não refatorada me dá sensação de dívida técnica. A outra flui melhor.	Dívida técnica; Leitura fluida	Qualidade técnica percebida	Compreensão mediada por estrutura
P15	1	Fora do meu domínio direto, mas achei a versão com nomes melhores mais compreensível.	Nomeação ajuda na compreensão mesmo fora da área	Nomes como facilitadores de entendimento	Compreensão mediada por estrutura
P16	1	A versão não refatorada me passou a sensação de urgência, de que foi feita às pressas. Já a refatorada parecia mais pensada.	Sensação de improviso; Código mais planejado	Percepção de cuidado no design	Compreensão mediada por estrutura
P17	1	O código sem refatoração viola princípios básicos de projeto. Fica difícil visualizar a arquitetura.	Violação de boas práticas; Arquitetura difícil de ver	Quebra de princípios de design	Compreensão mediada por estrutura
P18	1	A leitura do código sem refatoração foi cansativa. A refatorada me deu mais segurança.	Segurança com refatoração	Confiabilidade em organização estrutural	Compreensão mediada por estrutura
P19	1	A versão sem refatoração parece amadora. A outra passa profissionalismo.	Percepção de amadorismo; Profissionalismo na refatorada	Qualidade percebida	Compreensão mediada por estrutura

P20	1	A versão sem refatorar me causou incômodo imediato. A outra me deixou confortável.	Incômodo com código original; Conforto com refatorada	Confortabilidade em código refatorado	Compreensão mediada por estrutura
P1	2	A separação ajudou demais. No outro, eu ficava meio perdido, parecia um bloco só.	Separação de responsabilidades; Confusão com código monolítico	Clareza estrutural; Problemas com blocos únicos	Compreensão mediada por estrutura
P2	2	O nome do método atrapalhou. Achei que era booleano.	Nomeação confusa; Interpretação errada de tipo	Impacto da má nomeação	Compreensão mediada por estrutura
P3	2	A organização. A primeira versão me deu um nó.	Falta de organização; Confusão mental	Desorganização de código	Compreensão mediada por estrutura
P4	2	Nomeação ajudou muito. Sem nomes bons, não rola.	Nomeação clara; Facilitação pela nomenclatura	Benefício de nomes descritivos	Compreensão mediada por estrutura
P5	2	A verdade é que na disciplina de orientação a objetos a gente ficou sem professor metade do semestre.	Formação prejudicada; Ausência de orientação	Instabilidade acadêmica	Limitações de formação técnica
P6	2	Dificultou bastante o fato de não termos tido um professor fixo em Programação Orientada a Objetos.	Falta de professor fixo; Dificuldade com base teórica	Instabilidade acadêmica	Limitações de formação técnica

P7	2	Não saber usar bem os conceitos de classe e método.	Dificuldade com POO; Falta de domínio conceitual	Aprendizagem prejudicada	Limitações de formação técnica
P8	2	A estrutura dos métodos até ajudou, mas senti que não sabia a base.	Estrutura útil; Falta de base teórica	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura
P9	2	Perdido. Não entendi o objetivo logo de cara.	Falta de propósito claro; Desorientação	Ambiguidade de código	Compreensão mediada por estrutura
P10	2	A falta de clareza dos nomes e minha falta de conhecimento.	Nomeação ruim; Falta de conhecimento técnico	Déficit de formação técnica	Limitações de formação técnica
P11	2	Facilitou muito o fato dos métodos extraírem responsabilidades específicas. Dificulta quando tudo está num só bloco.	Extração de responsabilidades; Dificuldade com código monolítico	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura
P12	2	Facilitou a semântica dos nomes e a coesão dos métodos.	Clareza semântica; Coesão funcional	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura
P13	2	Facilitou a clareza nos nomes e a separação das lógicas.	Nomeação clara; Lógicas bem separadas	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura

P14	2	Dificultou a falta de coesão. A outra versão é mais orientada a responsabilidades.	Falta de coesão; Ausência de orientação a responsabilidades	Impacto negativo da má estruturação	Compreensão mediada por estrutura
P15	2	A terminologia técnica dos métodos ajuda quando bem nomeada.	Benefício da terminologia clara; Importância da nomeação	Vantagens da nomeação	Compreensão mediada por estrutura
P16	2	Dificultou o acoplamento de responsabilidades num único método. Quando está tudo misturado, exige mais energia cognitiva.	Acoplamento de responsabilidades; Sobrecarga cognitiva	Dificuldade com métodos sobrecarregados	Compreensão mediada por estrutura
P17	2	Faltou clareza na delimitação de responsabilidades. A versão refatorada alinha melhor com padrões de projeto.	Falta de delimitação; Aderência a padrões	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura
P18	2	O tamanho dos métodos na primeira versão foi um empecilho.	Métodos longos; Dificuldade de leitura	Impacto negativo da má estruturação	Compreensão mediada por estrutura
P19	2	Facilitou a coesão e o uso de nomes descritivos. Dificultou quando tudo estava aninhado.	Coesão; Nomeação descritiva; Problema com aninhamento	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura

P20	2	Facilitou o encapsulamento. Dificultou a falta de separação de responsabilidades.	Encapsulamento positivo; Dificuldade com código misto	Benefício da separação de responsabilidades	Compreensão mediada por estrutura
P1	3	Total. A com métodos separados parecia quase um passo a passo.	Métodos separados; estilo “passo a passo”	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura
P2	3	Sim, a renomeada fez muito mais sentido.	Renomeação significativa; sentido maior	Importância de nomeação	Compreensão mediada por estrutura
P3	3	Sim, na leitura de código. Uma flui, a outra trava.	Fluxo de leitura de código estruturado e não estruturado	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura
P4	3	Sim. Com nome certo, quase nem precisa ler o corpo do método.	Nomes autoexplicativos; leituras superficiais	Importância de nomeação	Compreensão mediada por estrutura
P5	3	Vi sim, mas até pra perceber isso eu demorei porque não sabia como os métodos funcionavam.	Dificuldade em perceber mudança; desconhecimento dos métodos	Má formação técnica	Limitações de formação técnica
P6	3	Sim, na clareza e divisão. Mas demorei pra perceber.	Clareza; divisão de responsabilidades	Clareza e divisão de responsabilidades	Compreensão mediada por estrutura

P7	3	Sim, mas nem consegui explorar bem por falta de conhecimento	Falta de exploração; desconhecimento conceitual	Instabilidade acadêmica	Limitações de formação técnica
P8	3	Sim, mas não entendi o porquê da mudança.	Reconhecimento de diferença; ausência de justificativa clara	Instabilidade acadêmica	Limitações de formação técnica
P9	3	Sim, a segunda pareceu melhor, mas ainda difícil pra mim.	Preferência pela refatorada; persistência da dificuldade	Instabilidade acadêmica	Limitações de formação técnica
P10	3	Sim, nomes mais descritivos ajudam demais.	Nomes descritivos; auxílio direto	Importância de nomeação	Compreensão mediada por estrutura
P11	3	Sim, na leitura e na manutenção futura. Refatorado é mais legível.	Legibilidade; manutenção futura	Manutenibilidade e legibilidade	Manutenção e escalabilidade
P12	3	Total. Uma é de fácil leitura e a outra demanda esforço cognitivo maior.	Clareza semântica	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura
P13	3	Sim, a estrutura refatorada melhora inclusive a testabilidade.	Testabilidade; benefício da estrutura	Testabilidade	Manutenção e escalabilidade
P14	3	Sim. Refatorar separa contextos e permite reuso.	Separação de contextos; reuso	Impacto positivo da refatoração	Manutenção e escalabilidade

P15	3	Sim, principalmente na clareza semântica dos nomes.	Clareza semântica	Importância de nomeação	Compreensão mediada por estrutura
P16	3	A diferença foi gritante. A refatorada te guia pela leitura.	Clareza semântica	Vantagens da nomeação e estruturação	Compreensão mediada por estrutura
P17	3	A refatorada é mais compatível com manutenção em times grandes.	Compatibilidade com equipes maiores; manutenção	Planejamento de manutenção	Manutenção e escalabilidade
P18	3	Sim, a refatorada favorece a manutenção futura e a colaboração.	Manutenção futura; colaboração	Planejamento de manutenção	Manutenção e escalabilidade
P19	3	Muito. A refatorada parece preparada para crescer.	Preparação para crescimento; escalabilidade	Planejamento de manutenção	Manutenção e escalabilidade
P20	3	Claro. Uma está preparada para evoluir, a outra está pronta pra quebrar.	Preparo para evolução	Planejamento de manutenção	Manutenção e escalabilidade
P1	4	A refatorada, sem dúvida. É mais modular e dá menos medo de mexer.	Modularidade, confiança	Segurança para manutenção	Facilidade de manutenção
P2	4	A com nome obterStatus, me economizou tempo.	Nomeação clara, economia de tempo	Clareza semântica	Facilidade de manutenção

P3	4	A segunda. Dá pra manter com mais facilidade.	Facilidade de manutenção	Manutenção mais simples	Facilidade de manutenção
P4	4	A com nome explicativo. Economia de tempo.	Nomes significativos, agilidade	Clareza semântica	Compreensão mediada por estrutura
P5	4	Provavelmente a com mais divisão, mas mesmo assim me senti inseguro.	Preferência pela modular, insegurança	Dificuldade por falta de domínio	Limitações de formação técnica
P6	4	A refatorada, mas precisaria de ajuda pra mexer.	Preferência, mas dependência	Dificuldade por falta de domínio	Limitações de formação técnica
P7	4	Talvez a refatorada, mas não conseguiria sozinho.	Dependência, dificuldade pessoal	Falta de autonomia técnica	Limitações de formação técnica
P8	4	A primeira, porque eu tentei seguir pelo que já vi, mesmo com dificuldade.	Apoio em experiências anteriores	Apego a padrões conhecidos	Limitações de formação técnica
P9	4	Nenhuma, na verdade. Eu teria que estudar antes.	Necessidade de estudo prévio	Dificuldade por falta de domínio	Limitações de formação técnica
P10	4	A segunda, mais compreensível.	Clareza, legibilidade	Clareza semântica	Compreensão mediada por estrutura
P11	4	Com certeza a refatorada. A modularização me	Modularidade, segurança	Segurança para manutenção	Facilidade de manutenção

		dá mais segurança pra mexer.			
P12	4	A com extração de métodos. Reduz a chance de quebrar algo por engano.	Risco reduzido	Segurança para manutenção	Facilidade de manutenção
P13	4	A refatorada. Daria pra reaproveitar funções menores e facilitar mocks em testes.	Reutilização, testabilidade	Testabilidade	Facilidade de manutenção
P14	4	A com extração. Dá pra aplicar SOLID com mais facilidade.	Aderência a boas práticas	Boas práticas	Facilidade de manutenção
P15	4	A que tem nomes mais precisos. Evita ambiguidade.	Nomeação clara	Clareza semântica	Facilidade de manutenção
P16	4	Prefiro a refatorada porque dá mais controle para testar partes específicas.	Testabilidade local	Testabilidade	Facilidade de manutenção
P17	4	A refatorada, sem dúvida. Reduz o risco de erro e aumenta a previsibilidade.	Menor risco, previsibilidade	Segurança para manutenção	Facilidade de manutenção
P18	4	Refatorada. Permite isolar comportamentos e reaproveitar partes.	Isolamento, reutilização	Reutilização	Facilidade de manutenção

P19	4	A refatorada. Ela já está pronta para testes e alterações.	Pronta para testes e alterações	Testabilidade	Facilidade de manutenção
P20	4	A refatorada. Equipe consegue trabalhar melhor em paralelo com esse tipo de estrutura.	Colaboração em equipes de trabalho	Trabalho colaborativo	Facilidade de manutenção
P1	5	Li primeiro por cima, depois fui seguindo os métodos.	Leitura superficial seguida de leitura modular	Estratégia exploratória	Estratégias cognitivas
P2	5	Fui tentando imaginar o que o método fazia antes de ler o conteúdo.	Inferência semântica prévia	Antecipação por nomeação	Estratégias cognitivas
P3	5	Eu fui destacando blocos na minha cabeça.	Agrupamento mental de blocos	Organização cognitiva	Estratégias cognitivas
P4	5	Leio os nomes primeiro. Eles me guiam.	Nomeação como guia	Organização cognitiva	Estratégias cognitivas
P5	5	Tentei seguir a lógica, mas me senti inseguro o tempo todo.	Raciocínio lógico, insegurança	Barreiras de compreensão	Limitações de formação técnica
P6	5	Li várias vezes e comparei trecho a trecho.	Leitura comparativa	Comparação iterativa	Estratégias cognitivas
P7	5	Tentei resolver de baixo pra cima, mas	Estratégia reversa	Leitura inversa	Estratégias cognitivas

		tava complicado de entender.			
P8	5	Fui escrevendo o código no papel pra ver o que fazia.	Escrita manual para apoio	Representação externa	Estratégias cognitivas
P9	5	Tentei ver o que cada função tava fazendo, mas tava complicado de entender o que tava pedindo.	Tentativa frustrada	Barreiras de compreensão	Limitações de formação técnica
P10	5	Leitura linha por linha com anotações no papelzinho.	Apoio externo	Representação externa	Estratégias cognitivas
P11	5	Divido a leitura por métodos e tento mapear a função de cada um.	Mapeamento funcional	Leitura orientada a estrutura	Estratégias cognitivas
P12	5	Análise top-down. Começo pela visão geral, depois desço nos métodos.	Abordagem top-down	Análise hierárquica	Estratégias cognitivas
P13	5	Costumo aplicar leitura com foco em responsabilidades por método.	Abordagem por responsabilidades	Responsabilidade de métodos	Estratégias cognitivas
P14	5	Verifico primeiro se respeita SRP e depois entendo os fluxos.	Validação de princípio SOLID	Princípios de engenharia	Estratégias cognitivas

P15	5	Comparei os nomes com a funcionalidade. Fiz analogia com scripts de automação.	Associação nome-função	Nomeação	Estratégias cognitivas
P16	5	Fui quebrando mentalmente as etapas e comparando com o que conheço de MVC.	Associação com padrões	Conhecimento prévio	Estratégias cognitivas
P17	5	Li como se fosse revisar um PR. Analiso a coerência da estrutura antes de ir para detalhes.	Visão de revisor	Leitura estruturada	Estratégias cognitivas
P18	5	Crio uma espécie de diagrama mental das funções para visualizar dependências.	Diagrama mental	Representação visual	Estratégias cognitivas
P19	5	Leio os nomes dos métodos como se fossem comentários. Isso guia minha leitura.	Nomeação como documentação	Nomeação	Estratégias cognitivas
P20	5	Faço uma análise de fluxo lógico. Imagino o que cada parte entrega como output.	Análise de entradas e saídas	Raciocínio lógico	Estratégias cognitivas

P1	6	Já, em projeto de disciplina. Sofri bastante quando estava tudo em um método só.	Experiência negativa com métodos grandes	Código difícil de manter	Experiência prévia com complexidade
P2	6	Já vi nomes ruins antes, mas costumo renomear pra não me atrapalhar depois.	Prática de renomeação	Nomeação	Experiência prévia com complexidade
P3	6	Sim, mas demoro o triplo pra entender quando tá bagunçado.	Código bagunçado atrasa compreensão	Impacto da desorganização	Experiência prévia com complexidade
P4	6	Já, e costumo refatorar pra não passar sufoco depois.	Proatividade na refatoração	Adaptação ativa	Experiência prévia com complexidade
P5	6	Já, e geralmente dependo muito da ajuda de algum colega ou Stack Overflow.	Dependência de colegas ou fontes	Falta de autonomia técnica	Limitações de formação técnica
P6	6	Já, e fiquei muito dependente do chatgpt ou dos colegas.	Uso de ferramentas de apoio	Dependência técnica	Limitações de formação técnica
P7	6	Sim, mas evito mexer quando não sei.	Evita modificar sem conhecimento	Dependência técnica	Limitações de formação técnica
P8	6	Já, em estágio, mas sempre com supervisão.	Acompanhamento contínuo	Falta de autonomia técnica	Limitações de formação técnica

P9	6	Pouco, mas quando sim, deixava alguém mais experiente modificar.	Terceiriza a manutenção	Falta de autonomia técnica	Limitações de formação técnica
P10	6	Já, em um trabalho final. Pedi ajuda do grupo.	Dependência de auxílio externo	Colaboração por necessidade	Limitações de formação técnica
P11	6	Sim, especialmente em sistemas legados. Refatorar aos poucos ajuda muito.	Refatoração incremental	Estratégia de manutenção	Experiência prévia com complexidade
P12	6	Sim, uso TDD e refatoração contínua como estratégia.	Boas práticas	Estratégias ágeis	Experiência prévia com complexidade
P13	6	Sempre. O ideal é refatorar em etapas com cobertura de testes.	Abordagem gradual	Estratégia de manutenção	Experiência prévia com complexidade
P14	6	Sim. Em revisão de PR sempre aponto pra extração se vejo método grande.	Feedback em código de equipe	Cultura de refatoração	Experiência prévia com complexidade
P15	6	Sim, com scripts de provisionamento. Nome claro é essencial.	Necessidade de nomeação clara	Importância da semântica	Experiência prévia com complexidade
P16	6	Demais. Refatoração e criação de testes são os primeiros	Prioriza refatoração e testes	Estratégia de manutenção	Experiência prévia com complexidade

		passos em código legado.			
P17	6	Sim, criamos RFCs e definimos critérios mínimos antes de aceitar código novo em produção.	Governança de código	Cultura de engenharia	Experiência prévia com complexidade
P18	6	Muitas vezes. Geralmente, proponho refatorações incrementais e documentadas.	Estratégia organizada	Estratégia de manutenção	Experiência prévia com complexidade
P19	6	Sim, uso sempre pair programming quando o código é muito fechado.	Colaboração em equipes de trabalho	Apoio colaborativo	Experiência prévia com complexidade
P20	6	Sim, em refatorações grandes usamos ferramentas como SonarQube e revisões em camadas.	Ferramentas de suporte	Engenharia de qualidade	Experiência prévia com complexidade
P5	7	Fraca. A gente teve muita troca de professor, e quando tinha aula era bem rasa. Não deu tempo de fixar Java direito.	Instabilidade docente, ensino superficial, falta de consolidação dos conteúdos	Instabilidade na docência	Formação fraca e desestruturada
P6	7	Falhou muito. Foram vários professores substitutos e	Troca de professores, ausência de	Instabilidade na docência	Formação fraca e desestruturada

		conteúdos que não avançaram.	continuidade no conteúdo		
P7	7	Muito fraca. Os professores não explicavam, e a maioria das aulas era leitura de slide.	Ensino passivo, ausência de explicações claras	Falta de planejamento pedagógico	Formação fraca e desestruturada
P8	7	Muito ruim. O professor nem aparecia para dar aula, e quando vinha era só passando slide e ficar mexendo no celular dele	Falta de comprometimento docente, aulas improvisadas	Falta de planejamento pedagógico	Formação fraca e desestruturada
P9	7	Deixou muito a desejar. A disciplina foi improvisada o semestre inteiro.	Disciplina desorganizada, falta de planejamento pedagógico	Falta de planejamento pedagógico	Formação fraca e desestruturada
P10	7	Muito abaixo do necessário. Quase ninguém saiu sabendo usar bem Java.	Baixo domínio de Java, falha na formação técnica	Má formação acadêmica	Formação fraca e desestruturada
P15	7	É essencial. O código precisa ser autodescritivo, especialmente em equipes multidisciplinares.	Clareza de código; Colaboração em equipes de trabalho	Clareza e compreensão de código	Compreensão mediada por estrutura
P16	7	Totalmente. Só mudo código com teste cobrindo o que estou refatorando.	Segurança em testes para refatoração	Segurança para manutenção	Facilidade de manutenção

		Sem isso, vira loteria.			
P17	7	Sigo a base do refatoramento, se o código não permite fácil compreensão por parte da equipe como um todo, volta pro dev.	Refatoramento como base de trabalho	Clareza e compreensão de código	Compreensão mediada por estrutura
P18	7	Documentação clara reduz o tempo de onboarding e evita más interpretações de intenções do código.	Clareza na documentação evita desperdício de tempo e má interpretação de código	Clareza e compreensão de código	Compreensão mediada por estrutura
P19	7	Negociação com o time e PO. Código sujo hoje vira dívida amanhã.	Negociação com o time; Problemas com falta de boas práticas de programação	Clareza e compreensão de código	Compreensão mediada por estrutura
P20	7	Pelo tempo de manutenção, quantidade de bugs e facilidade de entendimento por novos membros.	tempo de manutenção; Quantidade de bugs; Facilidade de entendimento	Clareza e compreensão de código	Compreensão mediada por estrutura