UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS VII – PATOS
CENTRO DE CIÊNCIAS EXATAS E SOCIAIS APLICADAS – CCEA
CIÊNCIAS DA COMPUTAÇÃO

JOSÉ DIEGO FERREIRA DA SILVA

O IMPACTO DA COMPLEXIDADE CICLOMÁTICA NA REFATORAÇÃO DO CÓDIGO

PATOS - PB

2025

JOSÉ DIEGO FERREIRA DA SILVA

O IMPACTO DA COMPLEXIDADE CICLOMÁTICA NA REFATORAÇÃO DO CÓDIGO

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

Área de concentração: Complexidade de código e refatoração.

Orientador: Esp. José Jandilson de Sousa Arruda

PATOS - PB

2025

S586i Silva, José Diego Ferreira da.
O impacto da complexidade ciclomática na refatoração do código [manuscrito] / José Diego Ferreira da Silva. - 2025.
66 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. Grad. Jose Jandilson de Sousa Arruda, Coordenação do Curso de Computação - CCEA".

1. Complexidade ciclomática. 2. Qualidade de código. 3. Refatoração. 4. Métricas de software. 5. Testabilidade. I. Título

21. ed. CDD 005.14

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

JOSÉ DIEGO FERREIRA DA SILVA

O IMPACTO DA COMPLEXIDADE CICLOMÁTICA NA REFATORAÇÃO DO CÓDIGO

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:

- **Harllem Alves do Nascimento** (***.796.924-**), em **16/06/2025 14:19:54** com chave **1e8136764ad611f0a89c1a1c3150b54b**.
- **Jannayna Domingues Barros Filgueira** (***.837.144-**), em **16/06/2025 07:56:39** com chave **948e25f84aa011f0b0c806adb0a3afce**.
- **Jose Jandilson de Sousa Arruda** (***.131.684-**), em **16/06/2025 04:12:19** com chave **3d92214c4a8111f0891d1a7cc27eb1f9**.

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: 16/06/2025
Código de Autenticação: 15e9be

“Aprender a criar códigos limpos é uma tarefa árdua e requer mais do que o simples conhecimento dos princípios e padrões.” (Martin, 2011)

AGRADECIMENTOS

Em primeiro lugar, a Deus que foi um guia excepcional em toda essa jornada até o final do curso.

À minha mãe Joana, e a todos os meus familiares que me apoiaram durante todo esse tempo.

À Jandilson de Sousa, professor e orientador por ter se empenhado nas sugestões ao longo de toda a orientação, todo tempo dedicado não será em vão.

À professora Dra. Rosângela Medeiros, pelo entusiasmo de ensinar diversas metodologias, entre outras grandes lições que foram excepcionais para estruturação deste trabalho de conclusão.

À professora Suelen, assim como Rosângela foi essencial na produção deste trabalho.

Aos demais professores que ministraram disciplinas excepcionais durante todos esses anos de curso de Computação na UEPB.

Aos colegas de turma que sempre me apoiaram e foram essenciais em momentos difíceis que porventura se desdobraram durante o curso.

RESUMO

Na Ciência da Computação, garantir a qualidade do código-fonte é um dos principais desafios enfrentados durante o desenvolvimento de software. Com o aumento da complexidade dos sistemas, cresce também a necessidade de adotar métricas que auxiliem na identificação de trechos críticos e na orientação de melhorias. Este trabalho investiga o impacto da complexidade ciclomática na qualidade do código, destacando sua utilidade como métrica para análise, teste e manutenção. A complexidade ciclomática quantifica o número de caminhos independentes em um programa, sendo uma ferramenta fundamental para identificar pontos de atenção que podem comprometer a legibilidade, testabilidade e manutenibilidade do sistema. Para embasar a análise, foi realizada uma revisão bibliográfica em bases reconhecidas, como *IEEE Xplore*, *ResearchGate, NIST*, *SBC Open Lib*, entre outras, além da aplicação prática de técnicas de refatoração orientadas por essa métrica em exemplos de código desenvolvidos em Java. A metodologia aplicada nos testes vai desde o estudo do código-fonte utilizando a IDE *Intellij* até a aplicação da métrica tanto pelo plugin *MetricsReloaded* quanto pela fórmula de MacCabe e posteriormente a refatoração. Os resultados evidenciam como a refatoração baseada na complexidade ciclomática contribui significativamente para a simplificação estrutural e melhoria da qualidade do software.

Palavras-chave: Complexidade ciclomática. Qualidade de código. Refatoração. Métricas de software. Testabilidade.

ABCTRACT

In Computer Science, ensuring source code quality is one of the main challenges faced during software development. As systems become more complex, the need to adopt metrics that help identify critical sections and guide improvements also grows. This work investigates the impact of cyclomatic complexity on code quality, highlighting its usefulness as a metric for analysis, testing, and maintenance. Cyclomatic complexity quantifies the number of independent paths in a program and is a fundamental tool for identifying points of concern that may compromise the readability, testability, and maintainability of the system. To support the analysis, a literature review was carried out in recognized databases, such as IEEE Xplore, ResearchGate, NIST, SBC Open Lib, among others, in addition to the practical application of refactoring techniques guided by this metric in code examples developed in Java. The methodology applied in the tests varies from studying the source code using the Intellij IDE to applying the metric using the MetricsReloaded plugin and the MacCabe formula and, subsequently, refactoring. The results show how refactoring based on cyclomatic complexity contributes significantly to structural simplification and improvement of software quality.

Keywords: Cyclomatic complexity. Code quality. Refactoring. Software metrics. Testability.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE GRÁFICOS

SUMÁRIO

1 INTRODUÇÃO

Com a evolução da Engenharia de *Software*, cresce junto a busca constante por códigos mais eficientes. Dentro desse contexto, a complexidade ciclomática tem se destacado como uma métrica essencial para avaliar a qualidade estrutural do código, facilitando sua manutenção, refatoração e testabilidade. Dessa forma, códigos com alta complexidade são mais difíceis de testar, manter e até mesmo entender.

Assim, o presente estudo tem o objetivo de analisar a influência da complexidade ciclomática em trechos de códigos, identificando possíveis problemas com alta complexidade e a possível refatoração utilizando ferramentas ou análise minuciosa do código. Para tanto, foi realizada uma pesquisa bibliográfica do tipo explicativa utilizando artigos, monografias e outros documentos relacionados a métrica da complexidade ciclomática.

Ademais, os resultados obtidos demonstram que a utilização da complexidade ciclomática como métrica de apoio na refatoração, permite identificar trechos críticos de código. Ao aplicar as principais técnicas de refatoração, observou-se uma redução significativa em tempo de execução e linhas de códigos, apesar do aumento do valor da complexidade em alguns exemplos, os exemplos ficaram mais legíveis e modulares.

A complexidade ciclomática, proposta por Thomas McCabe em 1976, é uma métrica que permite quantificar a complexidade de um trecho de código ao mensurar a quantidade de caminhos independentes no fluxo de execução. Essa prática é essencial para avaliar a cobertura de testes e identificar possíveis necessidades de refatoração. Para tanto, utiliza-se o grafo de fluxo como representação visual e matemática do programa.

Nesse contexto, destacando a relevância dos grafos, Gomes (2022, p. 5) defende que os grafos são abstrações matemáticas capazes de resolver uma ampla variedade de problemas em diversas áreas do conhecimento. Além disso, conforme Wijendra e Hewagamage (2021), o grafo de fluxo de controle descreve a estrutura lógica de um módulo de programa, no qual os nós representam expressões computacionais e as arestas indicam o controle entre essas expressões.

A métrica da complexidade ciclomática calcula todos os caminhos lineares independentes possíveis para a execução de um programa. Com isso, Pressman e Maxim (2016, p. 502), afirmam que um caminho é considerado linearmente

independente quando introduz novos comandos ou condições no grafo de fluxo. Nesse sentido, Barichello (2022, p. 18), defende que para um programa ser de qualidade não basta apenas ser somente funcional, é necessário que ele esteja bem estruturado, claro, coerente e de fácil manutenibilidade.

Assim, a complexidade ciclomática desempenha um papel crucial, pois permite mensurar a quantidade máxima de caminhos independentes que o código pode conter, influenciando diretamente a testabilidade e a manutenibilidade. Dessa forma, a métrica auxilia na identificação de trechos de código que podem demandar refatoração, uma vez que o valor obtido no cálculo corresponde exatamente à quantidade de testes necessários para garantir a cobertura completa.

1.1 Problemática

Um dos grandes fatos na área estudada é a relação de problemas na construção do código-fonte. Os erros podem estar associados a estruturas elementares do algoritmo como componentes, classes, funções ou métodos (Júnior, 2017). Diante disso, com o aumento da demanda por *software* de qualidade, torna-se essencial garantir que o código seja claro, modular e de fácil manutenção. No entanto, muitos projetos apresentam trechos complexos, com estruturas difíceis de entender, testar e modificar.

Com o cenário proposto, a refatoração surge como uma estratégia para melhorar a qualidade interna do código sem alterar seu comportamento externo. Contudo, identificar estruturas problemáticas nem sempre é uma tarefa simples. Nesse contexto, surge o problema: como utilizar métricas objetivas, como a complexidade ciclomática, para guiar e justificar o processo de refatoração de código, tornando-o mais eficiente e mensurável?

1.2 Objetivos

Assim, o presente trabalho tem como objetivo geral investigar como a métrica de complexidade ciclomática pode ser utilizada como instrumento de apoio na identificação problemas e refatoração do código. Quanto aos objetivos específicos, apresentam-se os seguintes:

1. Estudar os fundamentos teóricos da complexidade ciclomática e sua aplicação no contexto de Engenharia de *Software*;
2. Aplicar a métrica de complexidade ciclomática em trechos de código-fonte, identificando pontos críticos que possam indicar necessidade de refatoração;
3. Realizar a refatoração dos trechos de códigos que tiverem sua complexidade alta, adotando boas práticas de programação;
4. Avaliar os resultados da refatoração por meio da comparação da complexidade ciclomática e do grafo de fluxo do antes e do depois da intervenção;
5. Analisar o impacto da complexidade ciclomática na qualidade do código-fonte, com foco em aspectos como legibilidade, manutenibilidade e testabilidade;
6. Verificar a relação entre complexidade ciclomática e refatoração de código.

1.3 Justificativa

Na Engenharia de *Software*, a qualidade do código-fonte é um fator determinante para garantir a evolução contínua, a manutenção eficiente e a redução de custos futuros. Contudo, com o tempo e a necessidade por entregas rápidas, é comum que o código se torne complexo. Segundo McCabe (1976), quanto maior o valor da complexidade, maior será a dificuldade de se entender, modificar e testar o código fonte. Assim, o aumento da complexidade de um código pode comprometer sua qualidade.

Códigos complexos exigem maior esforço de manutenção e apresentam maior propensão a erros. Dessa forma, torna-se fundamental investigar o impacto dessa métrica na qualidade do código, possibilitando a identificação de possíveis trechos que necessitem de modificação. Nesse cenário, a refatoração surge como uma prática essencial para reestruturar trechos problemáticos sem alterar seu comportamento.

O presente trabalho se justifica pela necessidade de aplicar métricas como ferramenta de apoio à tomada de decisão. Assim, ao utilizar a complexidade ciclomática como guia para a refatoração, espera-se contribuir para a melhoria da qualidade em sistemas, promovendo práticas mais sustentáveis durante o desenvolvimento.

1.4 Resumo da metodologia

Perante as justificativas propostas, a execução deste trabalho foi realizada por meio de uma revisão dos fundamentos teóricos da complexidade ciclomática, sua relevância prática e a análise de trabalhos relacionados que complementam essa discussão. Dessa forma, a pesquisa tem característica bibliográfica e apresenta aspectos explicativos.

Para Wazlawick (2021, p. 43), a pesquisa bibliográfica envolve o estudo de artigos, teses, livros e outras publicações científicas que forneçam uma base teórica sólida e um panorama atual sobre o tema proposto. Além disso, este trabalho analisa exemplos de aplicação, o que permite ilustrar a aplicabilidade, limitações e contribuições dessa métrica em cenários práticos.

O estudo proposto utiliza diversas bases de dados, tanto nacionais como internacionais, dando foco nas nacionais e a partir do ano de 2021 para estudos relacionados, quanto ao estudo da métrica foi utilizado repositórios desde 1976, ano do artigo proposto por McCabe. Os materiais utilizados vão desde o artigo de McCabe, quanto artigos e livros relacionados a complexidade ciclomática, testes de software, grafos, refatoração e Engenharia de Software.

1.5 Estrutura do trabalho

Este trabalho está organizado em seis capítulos, além desta introdução. O Capítulo 1 apresenta uma visão geral do tema, contextualiza o problema de pesquisa, define os objetivos geral e específicos e justifica a importância do estudo, além de antecipar a estrutura do trabalho. O Capítulo 2 aborda a fundamentação teórica, descrevendo os principais conceitos relacionados à complexidade ciclomática, desde sua origem proposta por McCabe até seus fundamentos matemáticos, como grafos de fluxo de controle, destacando sua relevância para a qualidade e refatoração de código.

No Capítulo 3, são detalhados os procedimentos metodológicos utilizados, incluindo os critérios de seleção e exclusão de material, as ferramentas adotadas, o processo de refatoração e os critérios de avaliação. O Capítulo 4 apresenta os resultados obtidos com a aplicação da métrica da complexidade ciclomática em exemplos, discutindo os efeitos observados após a refatoração. Por fim, o Capítulo 5

traz as considerações finais, destacando as principais conclusões, as contribuições e limitações da pesquisa, bem como sugestões para investigações futuras.

2 FUNDAMENTAÇÃO TEÓRICA

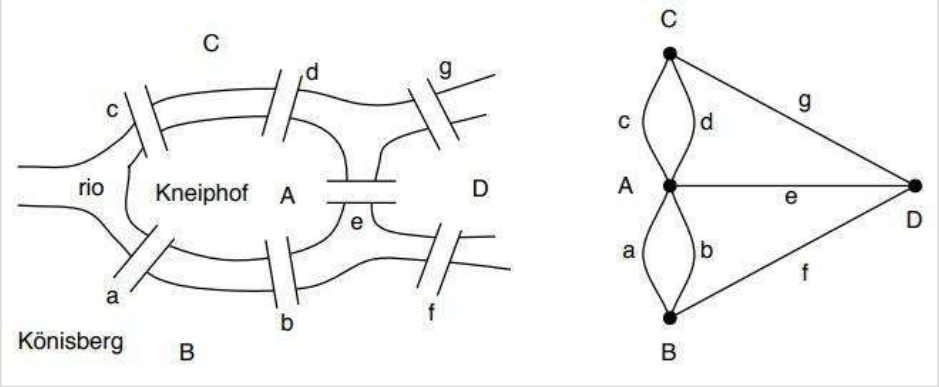
Neste capítulo, foi abordado os principais conceitos relacionados à complexidade ciclomática, sua origem proposta por McCabe, seus fundamentos matemáticos e sua relevância para qualidade e refatoração de código.

2.1 Teoria dos Grafos

O estudo de grafos tem uma grande relevância na Ciência da Computação, tendo uma disciplina própria no curso. A teoria dos grafos tem diversas aplicações no contexto de algoritmos, complexidade do código, entre outras. Diante disso, é necessário um breve resumo histórico sobre o estudo.

Os grafos têm origem em um problema antigo denominado o problema das sete pontes de *Konisberg* onde a cidade possui quatro partes conectadas por sete pontes como ilustra a Figura 1. Gomes (2022, p. 7) explica que essa foi a primeira vez que grafos foram usados, ele ainda fala que o problema está descrito em um artigo publicado por Leonard Euler. O problema consistia em determinar se é possível sair de qualquer ponto da cidade, atravessar cada uma das sete pontes somente uma vez e retornar ao ponto de partida.

Figura 1 - Diagrama representando o problema de Konisberg



Fonte: Gomes (2022, p. 8)

A Teoria dos Grafos tem diversas aplicações, não somente na área de computação, mas também em diversas outras áreas. Dessa forma, Gomes (2022, p. 5) explica que grafos são poderosas abstrações matemáticas utilizadas para modelar e resolver diversos problemas na Ciência da Computação e outras áreas. Seguindo

essa premissa, Melo (2014, p. 1) define que um grafo consiste em um conjunto finito e não vazio de objetos chamados vértices, juntamente com um conjunto de pares não ordenados de vértices chamados de arestas.

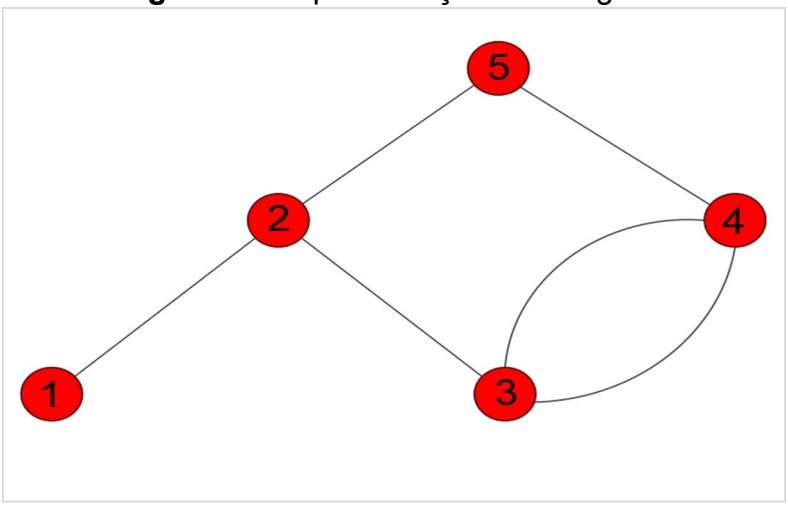
Com base nas informações apresentadas, podemos definir que um grafo é um conjunto de pontos chamados de vértices e um conjunto de pares chamados de arestas. Os vértices são chamados de nós ou *nodes* (N) e as arestas de *edges* (E).

Seguindo esses conceitos, Teoria dos Grafos é um ramo da Matemática que estuda as propriedades e as aplicações de grafos, ela tem um papel fundamental em diversas áreas do conhecimento, incluindo Ciência da Computação, Engenharia de Software, Redes de Comunicação, entre diversas outras.

A

Figura **2** mostra um exemplo de um grafo, onde os (vértices ou nós) são os círculos pintados em vermelhos e as arestas são as linhas que conectam os círculos:

Figura 2 - Representação de um grafo



Fonte: Elaboração própria (2024)

2.2 Aplicações de grafos em complexidade ciclomática

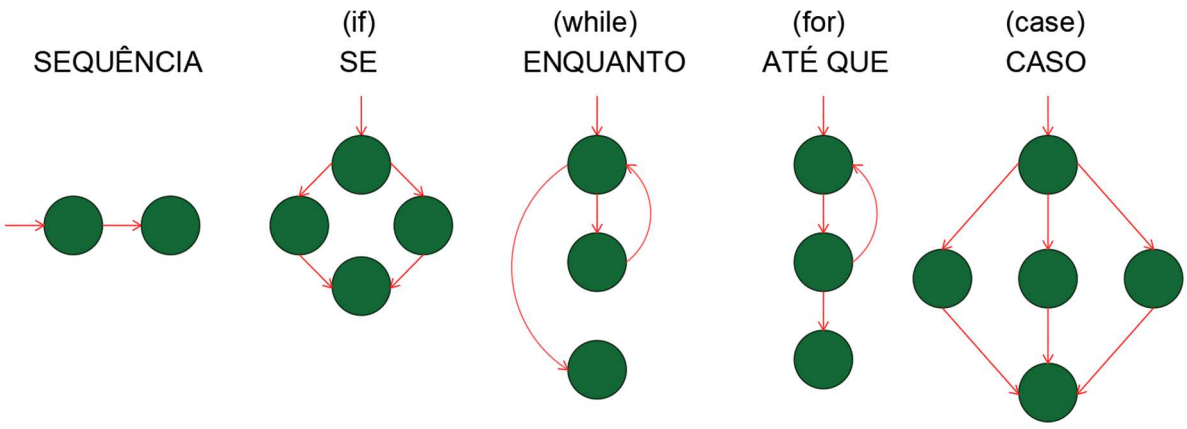
De acordo com Ajala *et al.* (2016, p. 589), o grafo da complexidade ciclomática tem suas bases na teoria dos grafos e oferece uma métrica de software muito útil. Ainda mais, existem inúmeras possibilidade de como os grafos podem ser usados na computação e na complexidade ciclomática é utilizado o grafo de fluxo.

Diante disso, Wijendra e Hewagamage (2021, p. 15) explicam que o grafo de fluxo de controle descreve a estrutura lógica dos módulos de programa em que seus blocos ou nós, representam as declarações ou expressões computacionais e as conexões ou arestas representam o controle entre os blocos.

> Os grafos de fluxo de controle descrevem a estrutura lógica dos módulos de software. Um módulo corresponde a uma única função ou sub-rotina. Em linguagens típicas, tem um único ponto de entrada e saída e pode ser usado como um componente de design por meio de um mecanismo de chamada/retorno (Watson; McCabe, 1996, p. 7).

Diante desses conceitos, o fluxo de execução de um programa pode ser modelado como um grafo direcionado, onde cada vértice representa um ponto de decisão e as arestas representam as transições possíveis de um estado para outro. Para Pressman e Maxim (2016, p. 501), o grafo de fluxo representa o fluxo de controle lógico usando a notação ilustrada na Figura 3. Nela também são ilustradas algumas formas de grafos para diferentes tipos de blocos de códigos, como uma sequência, uma condição, um laço e um caso.

Figura 3 - Notações de grafo de fluxo



Fonte: Elaboração própria (2025), baseado em Pressman e Maxim (2016, p. 501)

2.3 Métrica de complexidade ciclomática

A métrica de complexidade ciclomática é uma técnica usada em Engenharia de Software, proposta por Thomas McCabe em 1976 para medir a complexidade de um programa baseado no grafo de fluxo de controle. Ela indica a quantidade de caminhos

independentes no código, ou seja, quantos caminhos diferentes podem ser seguidos durante a execução do programa. Isso ajuda a avaliar a quantidade de teste, manutenção e compreensão do código.

> Um caminho independente é qualquer caminho através do programa que introduz pelo menos um novo conjunto de comandos de processamento ou uma nova condição. Quando definido em termos de um grafo de fluxo, um caminho independente deve incluir pelo menos uma aresta que não tenha sido atravessada antes de o caminho ser definido (Pressman; Maxim, 2016, p. 502).

Segundo McCabe (1976), a complexidade ciclomática é derivada da análise do grafo de fluxo do programa, onde cada nó representa um bloco de código e cada aresta indica a transição ou ramificações entre esses blocos. No Quadro 1 é descrito um breve resumo sobre arestas e nós em contexto de programa ou código-fonte.

Quadro 1 - Explicação sobre arestas e nós

Arestas (E - *edges*)
Representam as transições de um nó para outro no grafo de controle de fluxo. Em um trecho do programa, uma aresta é uma conexão ou ramificação entre dois blocos de código
Nós (N - *nodes*)
Os nós representam blocos de código ou pontos de decisão. São os pontos no grafo de fluxo de controle onde o código realiza uma ação ou toma uma decisão. Cada instrução básica, sem ramificações e cada decisão como um *if* ou *while* é representada por um nó

Fonte: Elaboração própria (2024).

Assim, a métrica proposta oferece uma visão crítica sobre o nível de dificuldade envolvido na compreensão, modificação e teste do código. Nesse sentido, testes de softwares estão ligados diretamente com complexidade, em especial o teste do caminho básico.

> O teste de caminho básico permite ao projetista de casos de teste derivar uma medida da complexidade lógica de um projeto procedimental e usar essa medida como guia para definir um conjunto base de caminhos de execução. Casos de teste criados para exercitar o conjunto base executam com certeza todas as instruções de um programa pelo menos uma vez durante o teste. (Pressman; Maxim, 2016, p. 500)

Ademais, limitar a complexidade durante o desenvolvimento de um software é uma boa prática, pois módulos com alta complexidade tendem a erros. Estabelecer

um limite na complexidade pode ser uma estratégia eficaz para garantir a qualidade e previsibilidade do software. Baseado nas informações propostas, é visível a importância do estudo da complexidade ciclomática na Engenharia de *Software*, em especial a área de testes e refatoração de código-fonte.

Desse modo, no estudo de McCabe (1976) ele desenvolveu diversas fórmulas para cálculo da complexidade ciclomática, onde cada uma tem sua particularidade sobre o tipo de algoritmo. A fórmula geral para calcular a complexidade ciclomática desenvolvida por Thomas McCabe é representada a seguir.

$$v(G) = E - N + 2 * P \quad (1)$$

No Quadro 2, cada componente da fórmula geral da complexidade ciclomática é descrito. O valor de P será igual a 1 quando o grafo de fluxo representar um único componente conectado, ou seja, quando não houver subgrafos independentes no programa analisado. Nesse contexto, sendo P = 1, a fórmula pode ser simplificada para:

$$v(G) = E - N + 2 \quad (2)$$

Quadro 2 - Representação dos itens da fórmula

Item	Descrição
v(G)	É a complexidade ciclomática;
E	Representa o número de arestas (*edges*);
N	Representa o número de nós (nodes);
P	Representa o número de componentes conexos.

Fonte: Elaboração própria (2024).

Em seu artigo, McCabe (1976) ainda explica que o valor calculado da métrica depende unicamente das estruturas de decisões. Nesse contexto, na linguagem de programação estruturada é utilizada uma fórmula mais simplificada onde apenas os pontos de decisão serão contabilizados. A fórmula simplificada é:

$$v(G) = D + 1 \quad (3)$$

De modo que D representa o número de pontos de decisão ou blocos (como estruturas *if*, *while*, *for*). Na fórmula geral, são acrescidos dois blocos a mais, o de início ou entrada do programa e outro de finalização ou saída do programa.

A complexidade de uma coleção de grafos de controle com vários componentes conectados P é igual à soma de suas complexidades (McCabe, 1976). Assim, uma maneira eficiente e confiável de calcular a complexidade é utilizando ferramentas automatizadas, pois a ferramenta pode abranger centenas de módulos e milhares de linhas de códigos (Watson; McCabe, 1996, p. 29).

Dessa maneira, como os programas tendem a ser mais complexos e possuem milhares ou milhões de linhas de códigos, a utilização de alguma ferramenta automatizada é crucial tanto na leitura quanto na geração do grafo de fluxo de controle. Os programas mencionados são partes do código em si, métodos ou funções. Para o presente estudo, foram utilizados apenas trechos de códigos e para calcular sua complexidade, utilizou-se a fórmula:

$$v(G) = E - N + 2$$

2.4 Impacto da complexidade na qualidade do código

A simplicidade do código é um dos pilares fundamentais para garantir a qualidade e a sustentabilidade de um software ao longo do tempo. Códigos simples, bem estruturados e com baixa complexidade ciclomática tendem a ser mais compreensíveis para diferentes desenvolvedores, facilitando o trabalho em equipe, a detecção de falhas e a realização de melhorias contínuas.

Dessa forma, analisar os principais aspectos que compõem a simplicidade como legibilidade, manutenibilidade e testabilidade é essencial para compreender os impactos positivos que essa característica pode trazer ao desenvolvimento de software. Quando esses três fatores estão bem equilibrados, o desenvolvimento e a evolução do software se tornam mais eficientes e confiáveis, diminuindo a possibilidade de manutenção ou refatoração.

2.4.1 Legibilidade

A legibilidade diz respeito à facilidade com que um código pode ser lido e compreendido por outros desenvolvedores, o que reduz o tempo necessário para entender sua lógica e identificar possíveis problemas. Gomes (2021, p. 30), aponta que o código mal escrito pode ser mal compreendido, colaborando para o aumento na complexidade e defeitos no código.

2.4.2 Manutenibilidade

A manutenibilidade refere-se à capacidade de modificar o código com facilidade, seja para corrigir erros, adaptar a novas necessidades ou melhorar sua estrutura, sem comprometer o funcionamento geral do sistema. Ribeiro e Travassos (2015) afirmam que atividades de retrabalho, frequentemente causadas por códigos difíceis de entender, representam uma parte significativa dos custos em projetos de desenvolvimento de software.

2.4.3 Testabilidade

Já a testabilidade está relacionada à facilidade de aplicar testes no código, permitindo verificar seu comportamento e assegurar que ele funcione corretamente em diferentes situações. Pressman e Maxim (2021, p. 772) diz que teste é um conjunto de atividades que podem ser planejadas com antecedência e executadas sistematicamente.

Diante disso, complementando o contexto de código limpo, Martin (2011, p. 14) sugere que não basta escrever um bom código ele precisa ser mantido sempre limpo. Entretanto, códigos complexos dificultam a identificação de problemas e tornam o processo de testes mais desafiador.

2.5 Relação entre complexidade ciclomática e refatoração de código

A complexidade ciclomática, por quantificar os caminhos independentes de execução de um código, serve como um indicativo direto da necessidade de refatoração. Quando essa métrica atinge valores elevados, sugere que o trecho analisado possui muitas ramificações lógicas, o que pode dificultar sua compreensão, manutenção e testabilidade.

Com isso, no estudo realizado, o grafo de fluxo foi essencial para identificar caminhos longos ou complexos. Dessa forma, Aho *et al.* (2008, p. 336) explica que

quando um programa em código é particionado em blocos, o fluxo de controle entre eles é representado por meio de um grafo de fluxo.

Ainda mais, nos exemplos representados no item 4, todos os grafos gerados por ferramenta ou manuais tem um ponto de partida e de finalização do trecho de código estudado. Diante disso, o grafo de fluxo inclui uma aresta que parte do nó de início até o primeiro bloco básico executável e outra que leva ao nó de saída a partir de blocos que podem encerrar a execução do programa (Aho *et al.*, 2008).

Nesse contexto, a refatoração torna-se uma estratégia essencial para reduzir a complexidade, promovendo um código mais simples, legível e eficiente. Ao identificar pontos críticos por meio da análise da complexidade, é possível aplicar técnicas de refatoração específicas que reorganizem a estrutura interna do código sem alterar seu comportamento externo, contribuindo para um software mais sustentável e com menor propensão a falhas. Para Fowler (2019, p. 90), uma das principais características da refatoração, é o fato dela não alterar o comportamento externo do programa.

> A refatoração é o processo de modificar um sistema de software de modo que não altere comportamento externo do código, embora melhore a sua estrutura interna. E uma maneira disciplinada de reorganizar o código, minimizando as chances de introduzir bugs. Em sua essência, ao refatorar, você estará aperfeiçoando o design do código depois que ele foi escrito (Fowler, 2019, p. 13).

2.6 Principais métodos de refatoração

A refatoração é o processo de reorganizar e melhorar o código-fonte sem alterar seu comportamento externo. Essa prática tem papel fundamental na redução da complexidade ciclomática entre outros tipos de complexidade, tornando o código mais legível, reutilizável e testável. Assim, a métrica identifica possíveis pontos de refatoração em estruturas condicionais ou repetição. Segundo Fowler (2019, p. 311) uma das principais fontes de complexidade em um programa é uma lógica condicional complexa.

> Durante a fase de refatoração, podemos aplicar qualquer conceito sobre um bom projeto de software. Podemos aumentar a coesão, diminuir o acoplamento, separar preocupações, modularizar as preocupações do sistema, reduzir nossas classes e funções, escolher nomes melhores, e por aí vai. (Martin, 2011, p. 172).

Assim, com base nas ideias propostas por Fowler (2019), destacam-se a seguir cinco dos métodos de refatoração mais apropriados para este estudo. Essas técnicas visam aprimorar a legibilidade, a manutenibilidade e a qualidade estrutural do código.

A técnica 1 – Extrair Método (*Extract Method*) – Consiste em mover trechos de código complexo ou que tem um propósito bem definido para um método ou função com o nome significativo. Essa prática facilita a compreensão do que o código faz e reduz a complexidade da função principal e ao utilizar nomes descritivos, o código se torna mais legível e expõe melhor seu propósito.

A técnica 2 – Substituir Condicional Aninhada por Cláusulas de Guarda (*Replace Nested Conditional with Guard Clauses*) – Essa técnica evita estruturas condicionais *if-else* profundamente aninhadas ao lidar com condições de erro ou casos especiais. As exceções são tratadas no início, mantendo o fluxo mais discreto, aumentando a legibilidade e o fluxo principal mais evidente.

A técnica 3 – Substituir Condicional por Polimorfismo (*Replace Conditional with Polymorphism)* – Subistitui estruturas condicionais como *if* ou *switch* que dependem de um tipo com a utilização de classes e métodos polimórficos. Esta técnica evita duplicação e melhora a escalabilidade e manutenção do código. Em vez de depender de estruturas condicionais espalhadas, cada comportamento fica encapsulado em sua respectiva classe, seguindo os princípios da orientação a objetos.

A técnica 4 – Decompor Condicional (*Decompose Conditional*) – Divide uma estrutura condicional complexa em partes menores, extraindo tanto a condição quanto os blocos de código-fonte. Ao decompor condicional, o código fica mais compreensivo e testável e ao isolar a lógica da condição, o mesmo se torna mais claro.

A técnica 5 – Consolidar Expressão Condicional (*Consolidate Conditional Expression*) – Agrupa diversas condições que levam à mesma ação, transformando em uma única expressão condicional. Esta reduz a repetição desnecessária e simplifica a lógica. Consolidar essas condições em uma única verificação diminui a chance de erro.

O Quadro 3 apresenta uma síntese dessas técnicas, suas descrições, seus benefícios e suas aplicações.

Quadro 3 – Principais Técnicas de refatoração

Técnica	Descrição	Benefício	Aplicação
1	Isola partes de um código complexo em funções menores e nomeadas de forma clara.	Diminui o número de instruções por função e melhora a legibilidade.	Ideal para blocos *if, switch*, ou laços que deixam a função longa.
2	Usa retornos antecipados (*return*, *continue*, etc.) para evitar profundos níveis de indentação.	Reduz a aninhamento de blocos e melhora a legibilidade.	Quando há muitos *if/else* internos.
3	Substitui blocos *switch* ou *if/else if* com várias verificações por classes com métodos especializados	Reduz a quantidade de ramificações e separa o comportamento.	Quando diferentes comportamentos dependem de um tipo ou estado.
4	Extrai expressões de condição e blocos *then/else* em funções separadas.	Reduz a complexidade e melhora a clareza de cada parte da lógica condicional	-
5	Une várias expressões *if* que produzem o mesmo resultado	Evita duplicações de fluxo e diminui a contagem de caminhos	-

Fonte: Elaboração própria (2025), baseado em Fowler (2019).

As cinco técnicas propostas por Fowler (2019) foram utilizadas no presente trabalho e para cada técnica foi criado um código com o objetivo de aplicar o método e verificar o resultado.

2.7 Trabalhos correlatos

A complexidade ciclomática foi proposta por McCabe (1976) como uma métrica para mensurar a complexidade de trechos de código em programas computacionais. Essa métrica avalia a quantidade de caminhos linearmente independentes que um programa pode seguir durante sua execução, sendo amplamente utilizada na definição de estratégias de teste, especialmente no planejamento de testes estruturados.

Watson e McCabe (1996) complementaram esse conceito ao apresentarem a metodologia de teste de caminho base (*basis path testing*), que utiliza a complexidade ciclomática como fundamento para garantir a cobertura adequada dos caminhos lógicos do programa. Essa abordagem permite verificar se os testes exercitam todas as possibilidades de fluxo do código, contribuindo para a identificação de falhas.

Barichello (2022) destaca que a manutenção de softwares com estruturas internas altamente complexas representa um desafio significativo para os desenvolvedores. Nesse contexto, o autor defende o uso de técnicas de refatoração

como forma de melhorar a arquitetura interna do código, reduzir sua complexidade e, consequentemente, aumentar a qualidade e a manutenibilidade do sistema. O presente trabalho assemelha-se mais com os estudos propostos Barichello (2022) no quesito do estudo sobre refatoração, quanto a complexidade ciclomática, todo o trabalho se baseia em McCabe (1976).

3 METODOLOGIA

No presente estudo, foi produzido um trabalho sobre complexidade ciclomática que foi proposta por Thomas McCabe em 1976 e a importância de sua utilização na qualidade e refatoração do código. Assim, a base teórica principal será o artigo do próprio autor, que foi publicado em 1976 e posteriores trabalhos do mesmo autor em 1996, ademais serão utilizados diversos materiais relacionados com a métrica como livros, artigos entre outros documentos filtrados entre os anos 2000 e 2025.

A pesquisa é classificada quanto à sua natureza como bibliográfica, onde foi realizada uma revisão de trabalhos anteriores sobre o tema proposto. Dessa forma, podemos também, classificar o trabalho estudado como pesquisa secundária, segundo Wazlawick (2021, p. 41), a pesquisa secundária ou bibliográfica busca obter informações a partir de trabalhos já publicados.

Ainda segundo Wazlawick (2021, p. 43), segundo os objetivos da pesquisa, pode-se classificar o estudo como explicativo, onde o objetivo é analisar os dados observados e compreender as causas e implicações da complexidade ciclomática na qualidade do código.

3.1 Definição dos filtros de busca

Dentre os materiais utilizados, o foco das buscas foram artigos, periódicos e outros materiais que contenham exatamente o tema de complexidade ciclomática, refatoração de código, e outros temas que são similares. Quanto ao intervalo de tempo do material, para os temas chaves como a métrica utilizada, foi utilizado artigos desde 1976. Por outro lado, para o material referente a refatoração outros similares como qualidade de código, testes, código limpo foram utilizados apenas matérias dentre o ano de 2000 até o ano de 2025.

Além disso, quanto a língua utilizada dos materiais, para o tema central considerou-se utilizar língua inglesa e para os conseguintes foi utilizado apenas material na língua portuguesa. Quanto ao filtro dos itens citados, no Quadro 4, é demonstrado os critérios de inclusão e exclusão.

Quadro 4 - Critérios de inclusão e exclusão

Critérios de inclusão e exclusão	
Critérios de inclusão	• Para o tema central, artigos que contenham exatamente o tema explicito no corpo; • Estudos que apresentem métricas de complexidade ou qualidade do código; • Para os temas correlatos, artigos sobre os temas e que sejam do ano 2000 até 2025; • Estudos com resumo bem definido, delimitando o foco do trabalho.
Critérios de exclusão	• Estudos que não abordem métricas de complexidade de código; • Estudos não relacionados na área da computação; • Material que não tenham o tema principal ou secundários deste estudo; • Estudos com resumo fora de contexto.

Fonte: Elaboração própria (2024).

3.2 Escolha das fontes de dados

A seleção das fontes de dados para o presente trabalho foi norteada por critérios de relevância, atualidade para os temas secundários e confiabilidade. Para embasar a análise da complexidade do código e o uso de métricas apropriadas, foram priorizadas fontes primárias e secundárias, incluindo artigos científicos que tenham publicações especializadas.

Diante disso, para a presente pesquisa foram utilizados base de dados de reconhecimento e qualidade, inclusive nas principais bases foram aplicados filtros para identificar publicações com impacto significado na Engenharia de *Software*. Ainda mais no Quadro *5* estão descritas as bases de forma detalhada.

Quadro 5 - Bases utilizadas na pesquisa

Bases utilizadas na pesquisa		
Base	**Descrição**	**Link**
IEEEXPLORE	Instituto de Engenheiros Elétricos e Eletrônicos	https://ieeexplore.ieee.org
NIST	Instituto Nacional de Padrões e Tecnologia	https://www.nist.gov
UFA	Repositório institucional da Universidade Federal de Alagoas	https://www.repositorio.ufal.br
Research Gate	Rede social voltada para pesquisadores	https://www.researchgate.net
UFES	Portal de Periódicos Eletrônicos da Universidade Estadual de Feira de Santana	https://periodicos.uefs.br

UNESP	Repositório institucional da Universidade Estadual Paulista	https://repositorio.unesp.br
RIUT	Repositório institucional da Universidade Tecnológica do Paraná	https://repositorio.utfpr.edu.br
SBC	Sociedade Brasileira de Computação	https://sol.sbc.org.br
UFPB	Repositório institucional da Universidade Federal da Paraíba	https://repositorio.ufpb.br/
IFC	Editora do Instituto Federal Catarinense	https://editora.ifc.edu.br/

Fonte: Elaboração própria (2024).

3.3 Exploração das *strings* e classificações de busca

Para assegurar uma pesquisa abrangente e obter fontes relevantes sobre o estudo, foram utilizadas diversas *strings* de busca, tanto em português quanto em inglês, para explorar as principais bases de dados as buscas foram realizadas tanto no google acadêmico quanto no *scielo* que são as principais bases de referências e ambos realizam buscas em diversos repositórios online. e ainda foram aplicados alguns filtros específicos nas buscas.

Seguindo esse contexto, parte do material encontrado foi pesquisado diretamente no site do google, onde só foram considerados trabalhos de relevância e que estivessem presente em algum dos repositórios apresentados no Quadro 5. Diante disso, respeitando o filtro aplicado entre os anos 2000 até 2025, as palavras-chave e expressões empregadas estão apresentadas no Quadro 6.

Quadro 6 - *Strings* de buscas

Strings utilizadas nas buscas		
	Google Acadêmico	**Scielo**
"complexidade ciclomática"	137	0
"*cyclomatic complexity*"	5.480	0
"refatoração de código"	191	0
"*code refactoring*"	4.710	3
"métricas de *software*"	358	4
"*software metrics*"	27.300	10
"complexidade de McCabe"	6	0
"McCabe *complexity*"	9.180	0
"teste de *software*"	1.150	0
"*software tests*"	274	7
"refatoração de código" + "complexidade ciclomática"	15	0
"refatoração" + "complexidade ciclomática"	216	0

Fonte: Elaboração própria (2024).

Diante das buscas, o google acadêmico foi mais eficaz para busca de material na língua brasileira. Entretanto, grande parte do material listado nas pesquisas tinha pouco ou quase nenhuma ligação com os objetivos do estudo presente, quanto ao *scielo*, este é mais específico quanto ao tipo de material de pesquisa, retorna apenas resultados de estudos publicados na língua inglesa.

3.4 Seleção dos trabalhos

Nesta etapa, para a seleção do material referenciado, foram adotados alguns critérios específicos para garantir a relevância e qualidade das fontes utilizadas, como data, excetuando-se os artigos desenvolvidos por McCabe, foi utilizado material datado a partir do ano 2000, com o objetivo de explorar informações atualizadas. Para o tema, foram considerados apenas materiais com ligação direta ou indireta ao tema deste trabalho, garantindo alinhamento com os objetivos da pesquisa.

No mais**,** quanto a organização do material e documentos editados, todo o material encontrado, incluindo links e arquivos, foram armazenados em uma pasta específica. Nesta pasta, foi criado um arquivo de *excel* com a estrutura total do estudo, incluindo material com informações dos repositórios encontrados e *links* de acesso.

3.5 Ferramentas utilizadas nos códigos

Atualmente existem diversas ferramentas para medição da complexidade ciclomática, como por exemplo o *SonarQube*, *PMD*, *Checkstyle*, *CodeMR, MetricsReloaded,* entre outros. Entretanto, a maioria das ferramentas disponíveis eram pagas, o que dificultou na execução de testes com a implementação do desenho do grafo de fluxo. Com isso, o *MetricsReloaded* foi o único com instalação simples, porém o *plugin* mencionado mesmo não tem ferramenta para a impressão do grafo de fluxo.

A IDE utilizada para os testes foi o *IntelliJ IDEA Community Edition*, da *JetBrains*. A ferramenta empregada para medir a complexidade foi um *plugin MetricsReloaded* que pode ser instalado na própria IDE e é capaz de calcular diversas métricas, como complexidade ciclomática, *fan-in/fan-out*, linhas de código, entre outras. Para a geração dos grafos de fluxo, utilizou-se a ferramenta *AutoCAD*, baseando-se no grafo de fluxo gerado pelo *Graphviz online*.

4 RESULTADOS E DISCUSSÃO

Neste capítulo serão expostos os resultados e discussão obtidos por meio da análise da complexidade ciclomática de diferentes trechos de código-fonte escritos em Java. Todos os exemplos foram testados com bibliotecas especificas da linguagem para quantificar o tempo de execução em microssegundos. Quantos aos gráficos, todos foram utilizados com escala logarítmica com base 2, aplicadas no eixo y para melhor visualização.

4.1 Apresentação dos dados obtidos

Para se obter um bom resultado e poder aplicar todas as cinco técnicas analisadas, os códigos foram elaborados com diferentes estruturas de controle de fluxo e posteriormente refatorados com o objetivo de reduzir sua complexidade.

Para o desenvolvimento, foi utilizada a IDE *IntelliJ*, em conjunto com o plugin *MetricReload*, responsável por calcular automaticamente esse tipo de métrica. Dessa forma, os valores obtidos foram analisados tanto diretamente na ferramenta quanto descritos e explicados ao longo deste trabalho. Para o cálculo de eficiência de tempo, utilizou-se a seguinte formula: ((tempo inicial – tempo final)/tempo inicial) * 100.

Antes de apresentar os códigos analisados, é importante destacar as principais estruturas de controle utilizadas na linguagem Java, que afetam diretamente a complexidade ciclomática. Essas estruturas estão resumidas no Quadro 7, e incluem comandos de decisão, laços de repetição, tratamento de exceções e operadores que introduzem desvios no fluxo de execução.

Quadro 7 - Principais estruturas de decisão Java

Estruturas de controle de fluxo	
Condições lógicas	*if, else if, switch*
Repetições	*for, while, do-while*
Exceções	*try-catch-finally*
Operadores ternários	*cond ? a : b*
Operadores lógicos	&& (E), || (OU)

Fonte: Elaboração própria (2025).

4.2 Exemplo 1 – técnica: extrair método

A técnica Extrair Método consiste em mover blocos de código de uma função longa para funções menores e bem nomeadas. O objetivo é dividir responsabilidades, melhorar a legibilidade e reduzir a complexidade ciclomática da função principal. Com isso, cada método passa a representar uma parte clara e isolada da lógica, facilitando testes, manutenção e futuras alterações no sistema.

4.2.1 Exemplo 1

O código analisado pertence a um método de cálculo de desconto para clientes, considerando o tipo de cliente, valor da compra e se é o aniversário do cliente. A função centralizava toda a lógica, incluindo decisões de negócio, verificações de condições aninhadas e cálculos misturados. No mais, os itens A e B descrevem sobre a estrutura e a aplicação da métrica no exemplo. Já o grafo do exemplo foi representado na Figura 4 em formato de fluxograma.

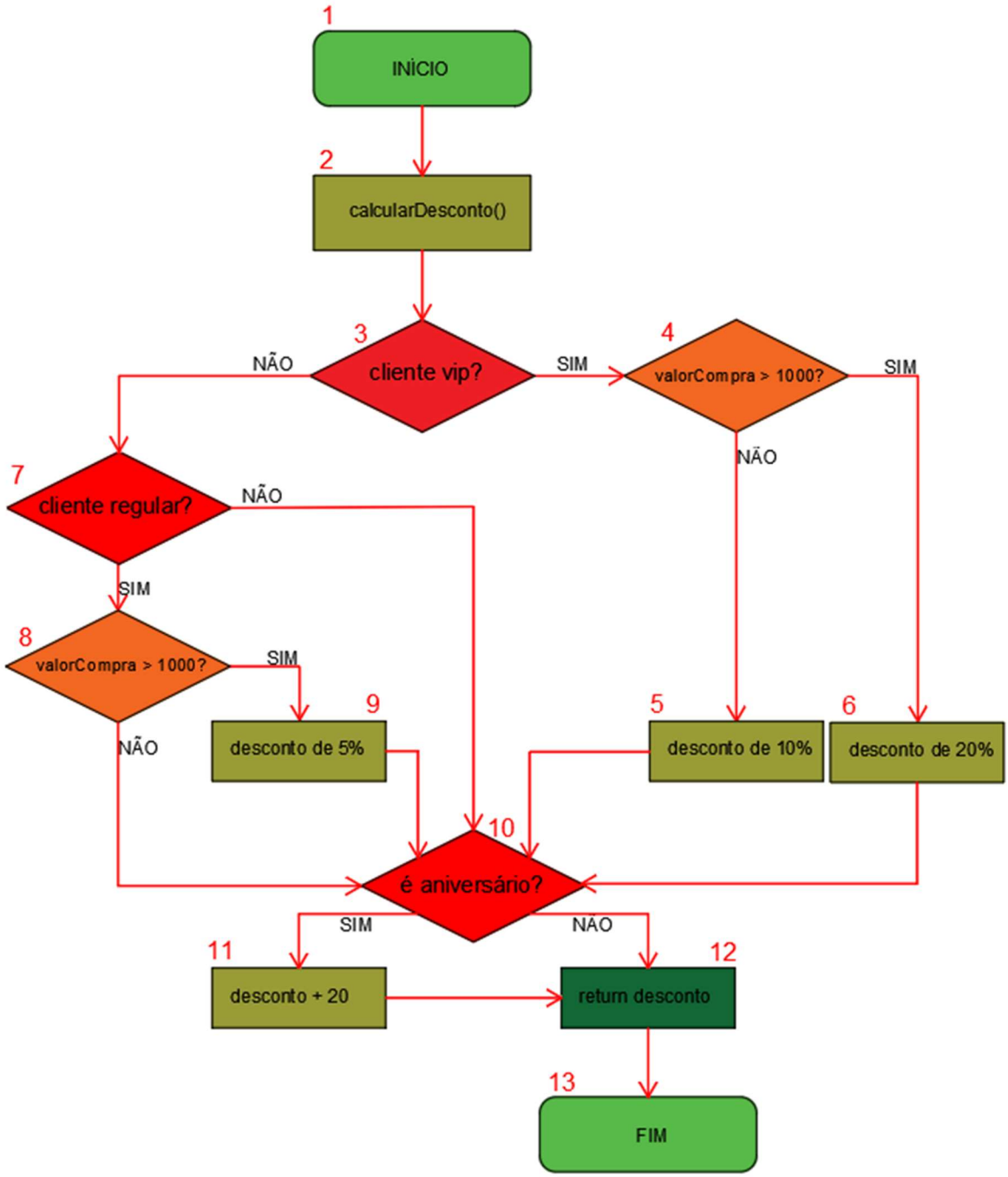
A) Principais problemas estruturais do primeiro exemplo:

- Função única com múltiplas responsabilidades.
- Condicionais aninhadas (*if/else*).
- Acúmulo de regras em um único ponto.
- Baixa legibilidade e alta chance de erro ao alterar regras.

B) Aplicando a complexidade ciclomática, tem-se os seguintes valores:

- Medida com *MetricsReloaded* = 6.
- Quantidade de nós: 13;
- Quantidade de arestas: 17;
- Aplicando a fórmula de McCabe:
  V(G) = E - N + 2
  V(G) = 17 - 13 + 2 = 6.

Figura 4 – Grafo do exemplo 1



Fonte: Elaboração própria (2025), baseado em (McCabe, 1976)

No grafo de fluxo numerado ou fluxograma, cada bloco foi numerado e os caminhos são representados pelas diversas sequências formadas até a finalização do programa. Assim, todos os caminhos independentes e dependentes estão descritos no Quadro 8.

Quadro 8 - Caminhos do exemplo 1

Lista de caminhos possível (independentes e dependentes)	
Caminhos independentes	
Caminho 1	1 → 2 → 3 → 4 → 5 → 10 → 11 → 12 → 13
Caminho 2	1 → 2 → 3 → 4 → 6 → 10 → 11 → 12 → 13
Caminho 3	1 → 2 → 3 → 7 → 8 → 9 → 10 → 11 → 12 → 13
Caminho 4	1 → 2 → 3 → 7 → 10 → 11 → 12 → 13
Caminhos derivados dos independentes	
Caminho 5 (deriva do 1)	1 → 2 → 3 → 4 → 5 → 10 → 12 → 13
Caminho 6 (deriva do 2)	1 → 2 → 3 → 4 → 6 → 10 → 12 → 13
Caminho 7 (deriva do 3)	1 → 2 → 3 → 7 → 8 → 9 → 10 → 12 → 13
Caminho 8 (deriva do 5)	1 → 2 → 3 → 7 → 8 → 10 → 12 → 13
Caminho 9 (deriva do 4)	1 → 2 → 3 → 7 → 10 → 12 → 13
Caminho 10 (deriva do 2 e 3)	1 → 2 → 3 → 7 → 8 → 10 → 11 → 12 → 13

Fonte: Elaboração própria (2025), baseado em (McCabe, 1976).

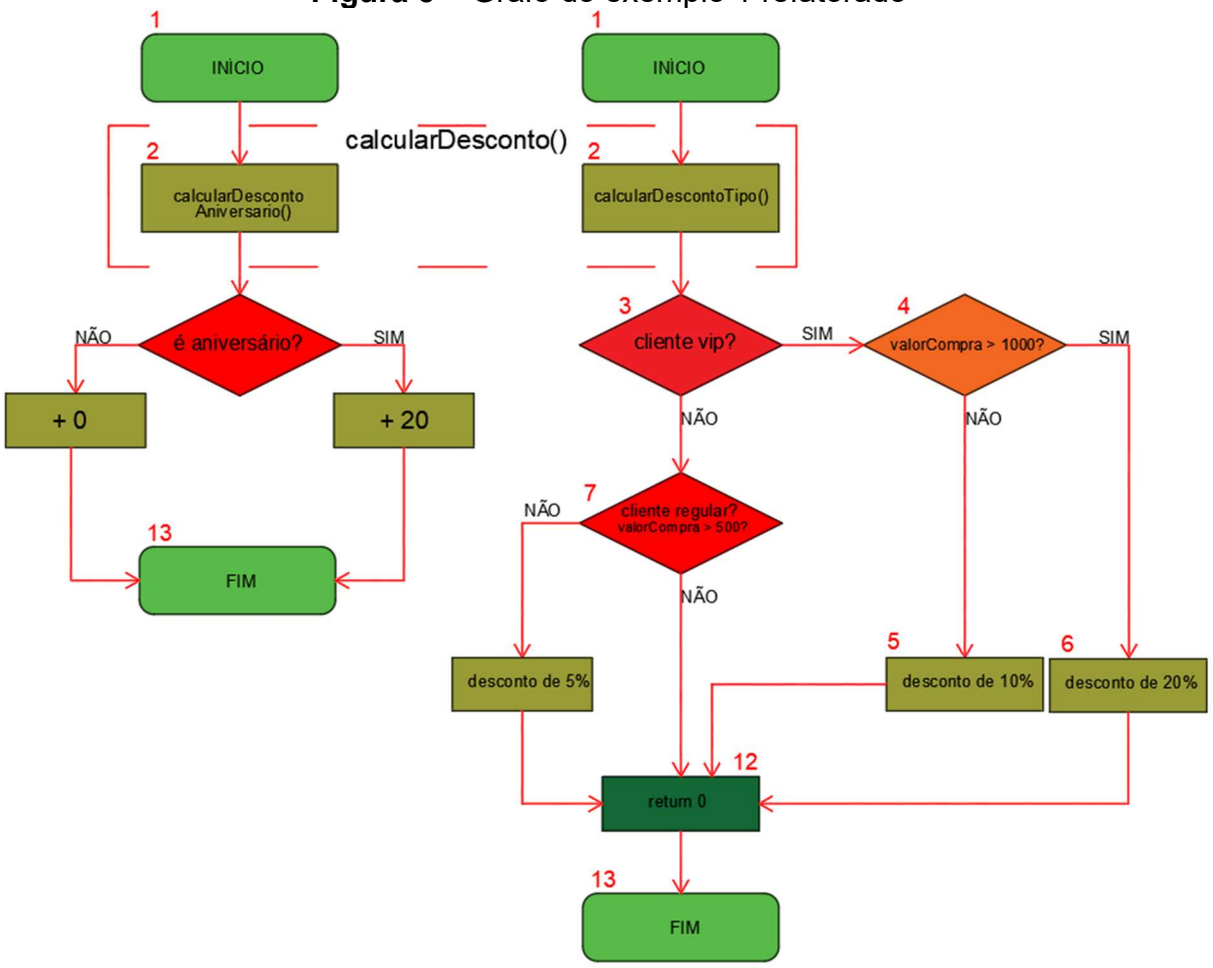
4.2.2 Aplicação da técnica de refatoração

Com a refatoração aplicada utilizando a técnica para Extrair Função ou Método, dois novos métodos foram criados, calcularDescontoPorTipo() e calcularDescontoAniversario(), deixando a função principal mais legível e focada. Além da técnica descrita, o fato de separar a lógica em funções especificas é parte de outra técnica chamada Decompor Condicional. Ademais, os principais objetivos da técnica específica são:

- Separar cada bloco condicional e cálculo em métodos com nomes descritivos.
- Tornar o método principal uma composição legível das regras de negócio.
- Preparar o código para futuras extensões.

Na Figura 5 pode-se observar o código-fonte refatorado, com os novos métodos adicionados e, ao lado, o grafo de sua execução.

Figura 5 – Grafo do exemplo 1 refatorado



Fonte: Elaboração própria (2025), baseado em (McCabe, 1976)

4.2.3 Análise comparativa antes e depois da refatoração

No exemplo 1, o código apresentava alta verbosidade concentrada em um único método. Embora a refatoração tenha aumentado o número de métodos e distribuído a complexidade entre eles, a legibilidade e testabilidade foram aprimoradas. Dessa forma, no Quadro 9 é demonstrado um breve resultado pós-refatoração.

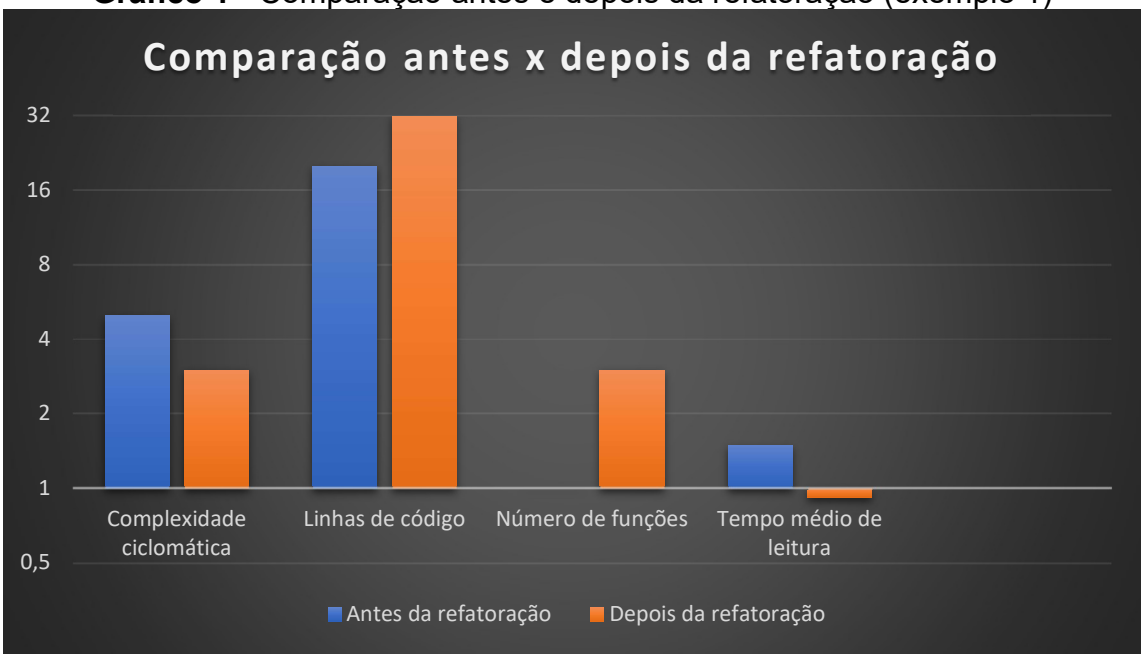
Quadro 9 - Resultados da refatoração

Métrica	Antes da refatoração	Depois da refatoração
Complexidade ciclomática	5	3
Linhas de código	20	32
Números de métodos	1	3
Tempo de execução (µs)	0.149	0.092

Fonte: Elaboração própria (2025).

No Gráfico 1 pode-se notar a diferença antes e depois da refatoração. Percebe que houve uma melhora na complexidade ciclomática, porém um aumento no número de linhas de código e no número de funções. Diante disso, o aumento de linhas não afeta na melhoria da estrutura do exemplo, pois tal prática pode ser utilizada na organização visual e prática do código, como a separação de métodos por exemplo.

Gráfico 1 - Comparação antes e depois da refatoração (exemplo 1)



Fonte: Elaboração própria (2025).

4.2.4 Discussão crítica dos resultados

A aplicação da técnica resultou em uma estrutura mais clara, modular e legível. Apesar do aumento no número de funções e linhas de código, a complexidade ciclomática foi significativamente reduzida e distribuída, favorecendo a manutenção e evolução do código. Quanto ao cálculo da eficiência do tempo de execução:

((0,149 – 0,092) / 0.149) * 100 = 38,25%

A refatoração permitiu:

- Tornar o código mais fácil de testar individualmente, pois cada função passou a tratar de um único aspecto;
- Reduzir o acoplamento e a dependência de múltiplas condições em um único bloco;
- Criar um ponto de entrada (calcularDesconto) que descreve claramente o fluxo das regras de negócio;
- Aumentar a eficiência de tempo de execução em torno de 38%.

No exemplo 1, antes da refatoração, houve dificuldade na contagem dos caminhos independentes. A ferramenta *MetricReloaded* apontou uma complexidade ciclomática igual a 6 ou seis caminhos independentes. Contudo, ao aplicar a fórmula clássica de McCabe e analisar os caminhos no grafo de controle de fluxo, foi possível confirmar o mesmo valor 6, porém a contagem direta dos caminhos levou a 5 independentes. Esse cenário evidencia o quão complexo e difícil de interpretar o código estava antes da refatoração, reforçando a importância da técnica aplicada.

4.3 Exemplo 2 – técnica: substituir condicional aninhada por cláusulas de guarda

A técnica Substituir Condicional Aninhada por Cláusulas de Guarda consiste em eliminar *if* aninhados substituindo-os por retornos antecipados (*return*) ou interrupções claras no fluxo do método. Essa abordagem reduz o nível de indentação, simplifica o controle de fluxo e melhora a legibilidade e a manutenção do código.

4.3.1 Exemplo 2

O código do exemplo 2 pertence a uma função de validação de *status* de usuário, cujo objetivo é verificar se o usuário é válido, ativo e não banido, retornando uma resposta apropriada para cada situação apontada. Para isso, nos itens A e B são demonstrados a estrutura e a aplicação da métrica no exemplo. Quanto ao grafo, ele foi representado na Figura 6 em formato de fluxograma.

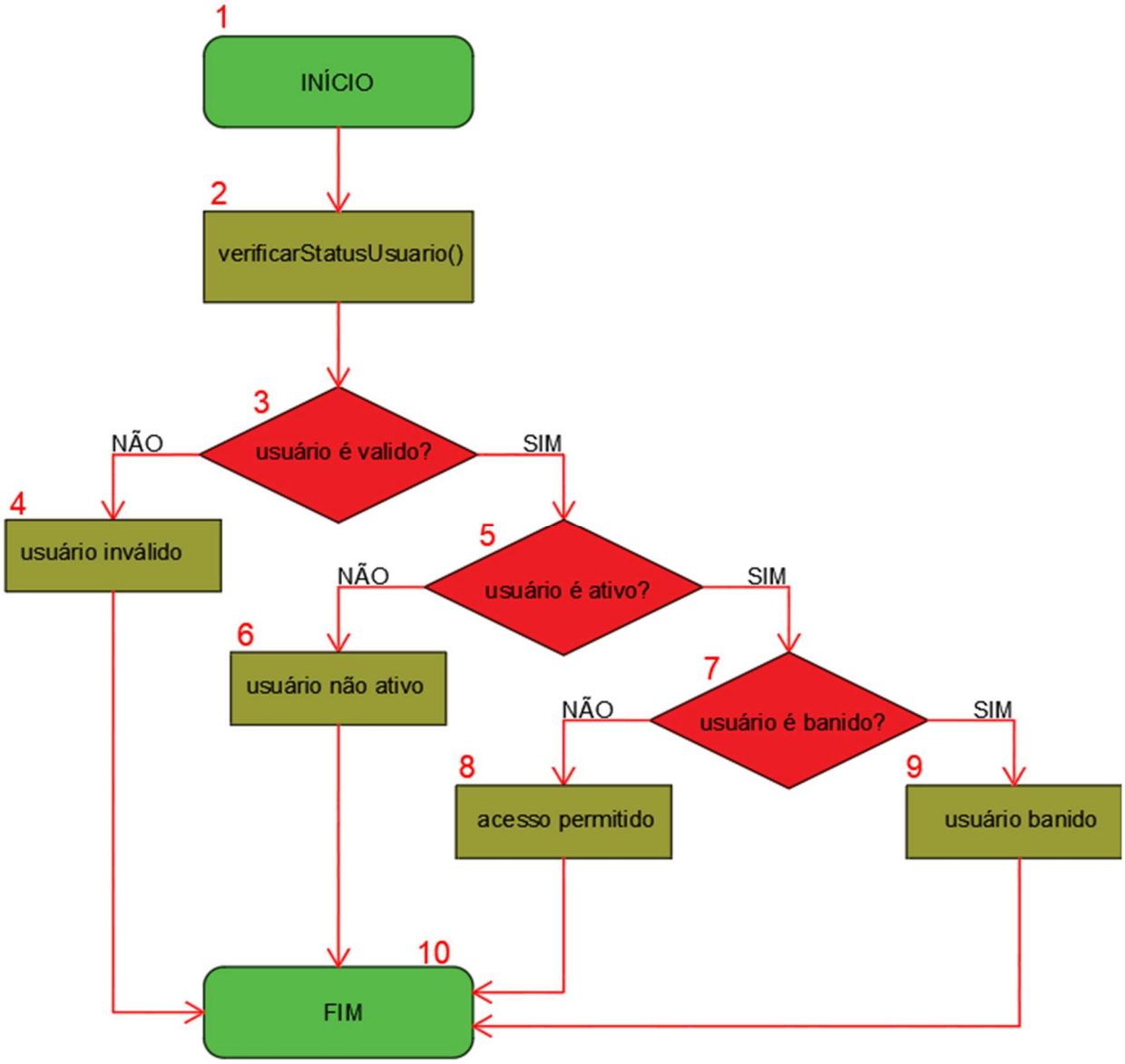
A) Principais problemas estruturais do segundo exemplo:

- Uso de condicionais aninhadas (*if* dentro de *if*), dificultando a leitura e aumentando a profundidade de indentação. Cada nova verificação era feita dentro da anterior, exigindo leitura encadeada para entender a lógica.
- Lógica pouco clara para os casos de erro ou bloqueio, uma vez que, o retorno "usuário inválido" só é alcançado após verificar todas as demais condições.
- Necessidade de interpretar todas as condições até chegar à resposta final, isso obriga o leitor a analisar todo o encadeamento antes de entender o resultado.
- Baixa legibilidade e maior chance de erro ao modificar a lógica, qualquer mudança nas regras exige entender toda a cadeia de condicionais.

B) Aplicando a complexidade ciclomática, tem-se os seguintes valores:

- Complexidade medida com o *MetricsReload* = 4;
- Aplicando a fórmula de McCabe:
  V(G) = E - N + 2P = V(G) = 12 - 10 + 2 * 1 = 4.

Figura 6 – Grafo do exemplo 2



Fonte: Elaboração própria (2025), baseado em (McCabe, 1976)

No grafo de fluxo, cada bloco ou nó foi numerado e foram gerados quatro caminhos possíveis até a finalização do programa, os quais são mostrados no Quadro 10.

Quadro 10 - Caminhos possíveis do exemplo 2

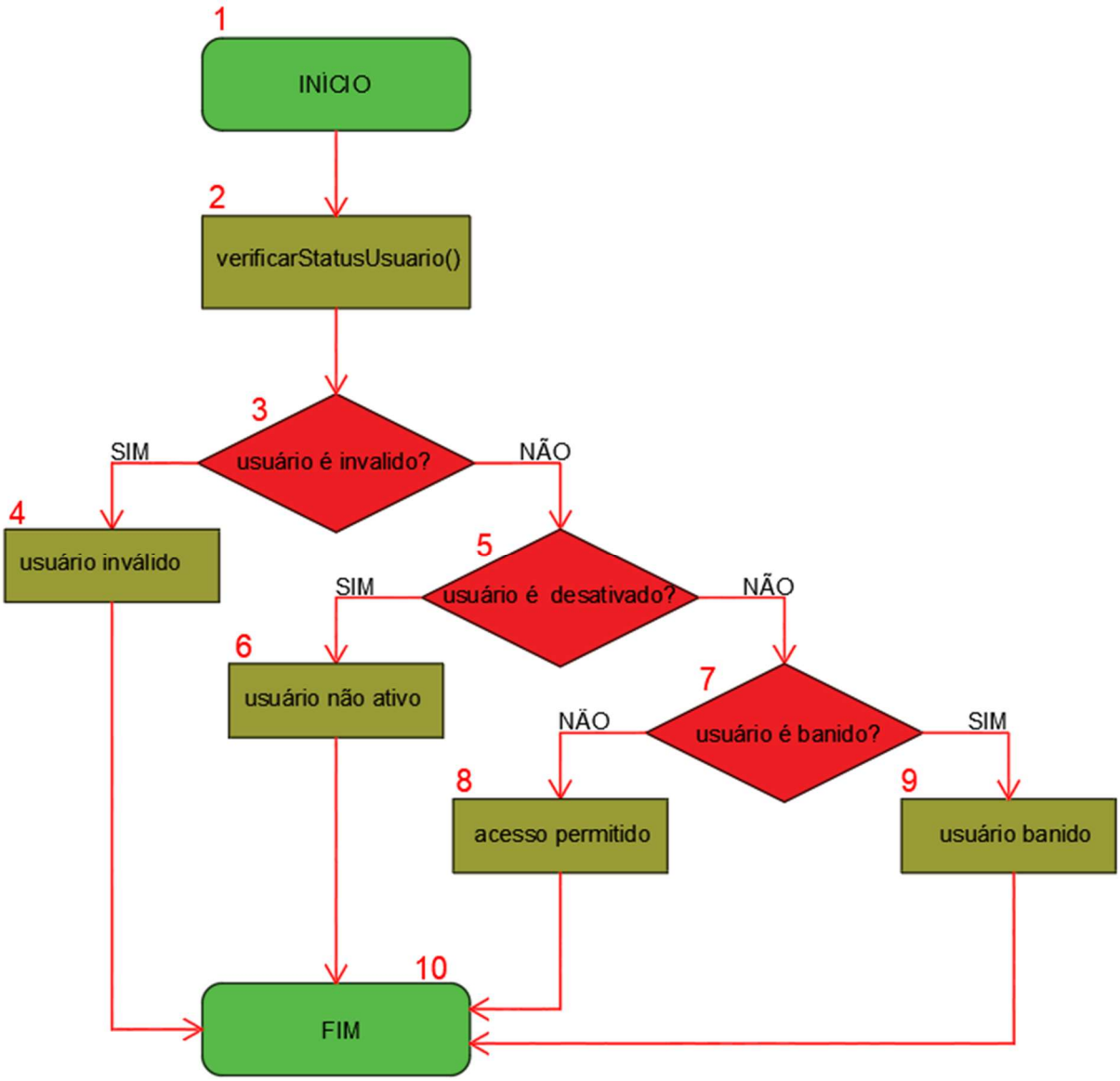
Lista de caminhos possível (independentes e dependentes)	
Caminhos independentes	
Caminho 1	1 → 2 → 3 → 4 → 10
Caminho 2	1 → 2 → 3 → 5 → 6 → 10
Caminho 3	1 → 2 → 3 → 5 → 7 → 8 → 10
Caminho 4	1 → 2 → 3 → 5 → 7 → 9 → 10

Fonte: Elaboração própria (2025), baseado em (McCabe, 1976).

4.3.2 Aplicação da técnica de refatoração

A técnica aplicada no exemplo proposto tem o objetivo de eliminar os níveis de aninhamento e simplificar a leitura da função, o grafo para a refatoração difere poucas coisas do grafo anterior, conforme mostrado na Figura 7. As verificações foram reorganizadas para que cada uma seja feita de forma isolada, com retorno imediato ao detectar qualquer condição que impeça o acesso.

Figura 7 – Grafo do exemplo 2 refatorado



Fonte: Elaboração própria (2025), baseado em (McCabe, 1976).

4.3.3 Análise comparativa antes e depois da refatoração

No exemplo 2, apesar de ser intuitivo antes da refatoração, a função apresentava aninhamentos profundos e exigia que o desenvolvedor lesse toda a estrutura para entender os possíveis retornos. Após a refatoração, o método passou a ter um fluxo mais linear e previsível, com cláusulas de guarda para cada condição crítica, tais mudanças estão representadas no Quadro 11.

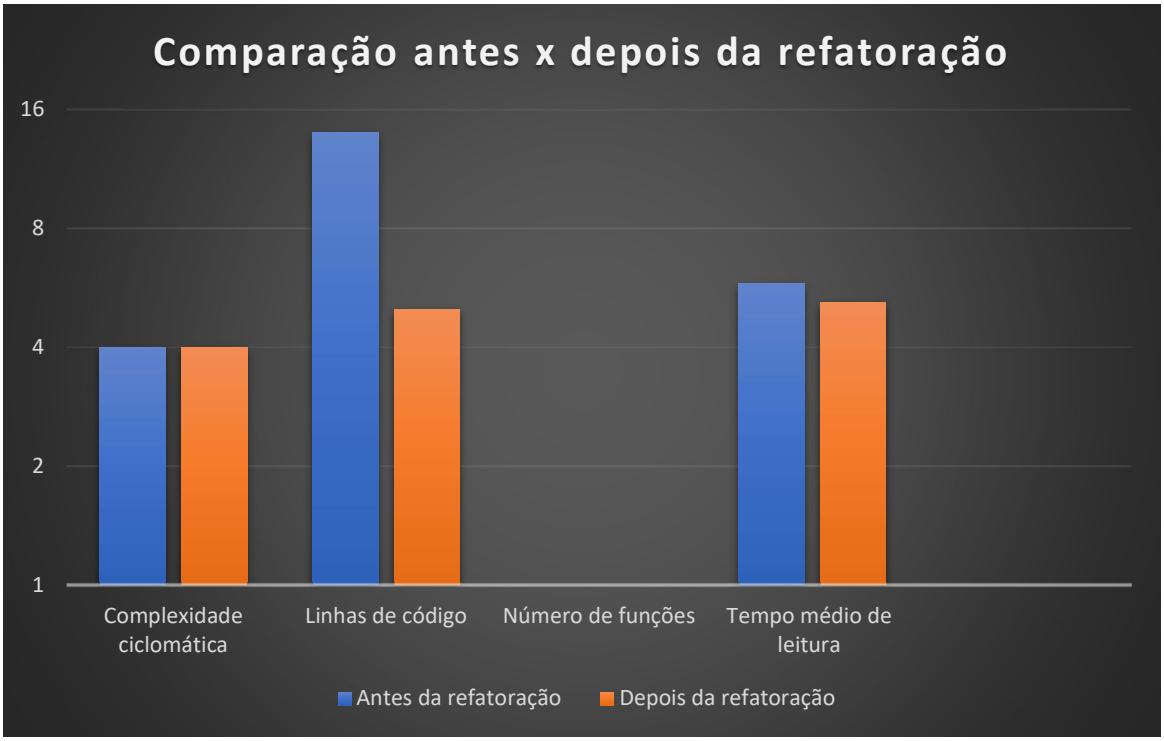
Quadro 11 - Resultados da refatoração

Métrica	Antes da refatoração	Depois da refatoração
Complexidade ciclomática	4	4
Linhas de código	14	5
Números de métodos	1	1
Tempo de execução (µs)	5,8	5,2

Fonte: Elaboração própria (2025).

No Gráfico 2 pode-se verificar a comparação com relação a complexidade, linhas de códigos, números de funções e tempo médio de leitura.

Gráfico 2 - Comparação antes e depois da refatoração (exemplo 2)



Fonte: Elaboração própria (2025).

4.3.4 Discussão crítica dos resultados

A aplicação da técnica de cláusulas de guarda resultou em uma função mais legível e menos propensa a erros. Embora a complexidade ciclomática tenha se mantido em 4, a redução do nível de aninhamento e a linearidade do fluxo de controle melhoraram significativamente a clareza e a manutenção do código. Quanto ao cálculo da eficiência do tempo de execução:

((5,8 – 5,2) / 5,8) * 100 = 10,34%

Pode-se notar que a refatoração:

- Melhorou a compreensão do código;
- Reduziu os níveis de aninhamento;
- Tornou o fluxo mais previsível e fácil de alterar;
- Favoreceu uma abordagem mais limpa e objetiva na lógica de validação;
- Aumentar a eficiência de tempo de execução em torno de 10,34%.

O desafio do exemplo 2 foi na construção da refatoração e a dificuldade em visualizar condicional *if* sem o *else.* Entretando, no exemplo estudado o tempo e quantidade de linhas de código deixam claro os resultados positivos e refletem a importância da aplicação de métricas e técnicas de refatoração.

4.4 Exemplo 3 – técnica: substituir condicional por polimorfismo

A técnica Substituir Condicional por Polimorfismo é usada quando diferentes comportamentos são implementados através de condicionais com base em tipos ou valores. Ao invés de usar instruções switch ou múltiplos if, a lógica é transferida para subclasses ou objetos especializados, aproveitando o polimorfismo para encapsular as variações de comportamento.

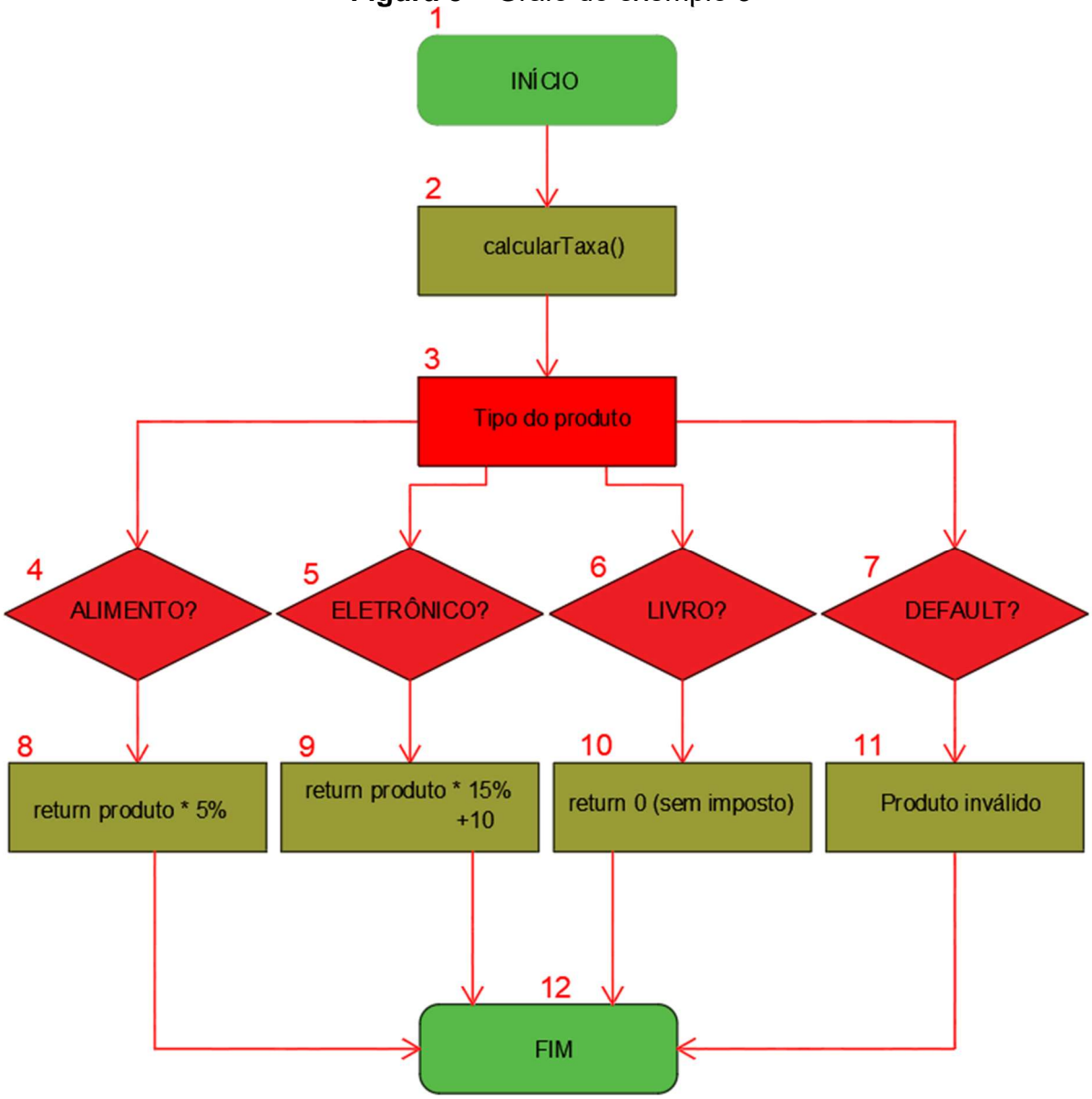
4.4.1 Exemplo 3

O código do terceiro exemplo calcula diferentes valores de impostos com base no tipo de produto utilizando uma estrutura condicional chamado *switch.* O código aplica diversas alíquotas e fórmulas específicas conforme categoria do item. Para o exemplo proposto, o grafo de fluxo construído está apresentado na Figura 8. Ademais, nos itens A e B estão informações sobre a estrutura e complexidade do exemplo.

A) Os principais problemas estruturais verificados são:
   - Uso de switch para delegar lógica de negócio;
   - Difícil de manter ou estender para novos tipos;
   - Baixa coesão: o cálculo do imposto não pertence diretamente à classe do tipo.

B) Aplicando a complexidade ciclomática, tem-se os seguintes valores:
   - Complexidade medida com o *MetricsReload* = 4;
   - Aplicando a fórmula de McCabe:
     V(G) = E - N + 2P = V(G) = 14 - 12 + 2 * 1 = 4.

Figura 8 – Grafo do exemplo 3



Fonte: Elaboração própria (2025), baseado em (McCabe, 1976)

No grafo do exemplo 3 é possível verificar os caminhos na utilização do *switch,* sendo 1 para cada caso. O Quadro 12 mostra quais são os quatro caminhos possíveis obtidos a partir do exemplo da Figura 8.

Quadro 12 - Caminhos possíveis do exemplo 3

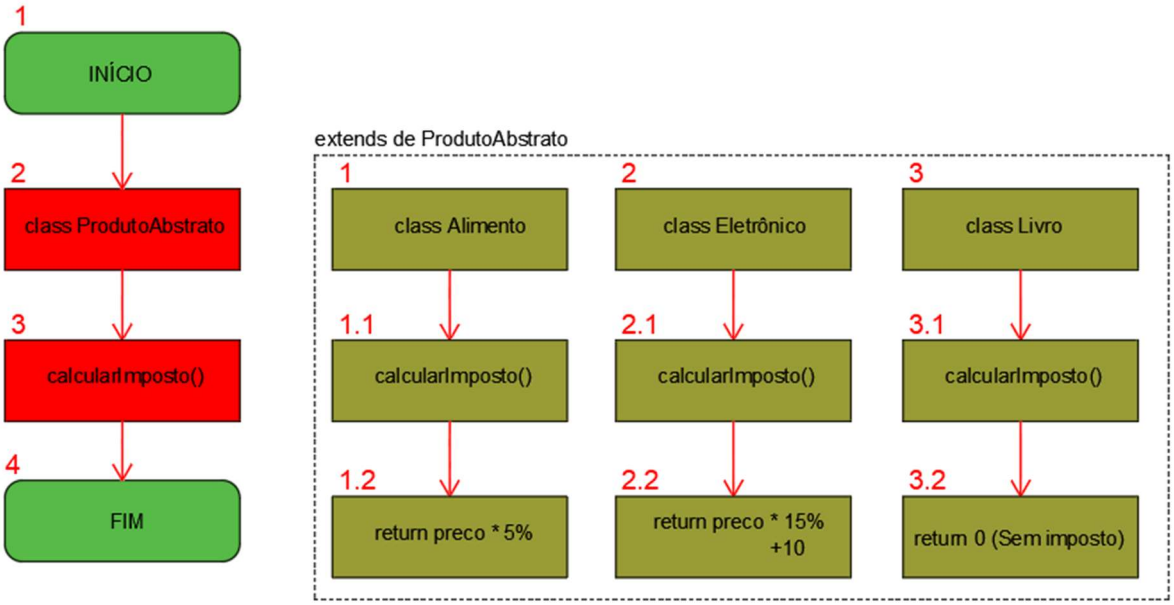
Lista de caminhos possível (independentes e dependentes)	
Caminhos independentes	
Caminho 1	1 → 2 → 3 → 4 → 8 → 12
Caminho 2	1 → 2 → 3 → 5 → 9 → 12
Caminho 3	1 → 2 → 3 → 6 → 10 → 12
Caminho 4	1 → 2 → 3 → 7 → 11 → 12

Fonte: Elaboração própria (2025), baseado em (McCabe, 1976).

4.4.2 Aplicação da técnica de refatoração

Para o exemplo descrito, foi criada uma classe abstrata chamada Produto, que define o método calcularImposto() como abstrato. Em seguida, foram implementadas subclasses para produto, alimento, eletrônico e livro. Cada uma das classes sobrescreve o método de acordo com a lógica de cálculo de imposto correspondente. Essa abordagem torna o código mais modular, flexível e aderente aos princípios da programação orientada a objetos, facilitando futuras extensões e manutenções e quanto ao grafo pode ser visto na Figura 9.

Figura 9 – Grafo do exemplo 3 refatorado



Fonte: Elaboração própria (2025), baseado em (McCabe, 1976).

4.4.3 Análise comparativa antes e depois da refatoração

No exemplo apresentado, antes da refatoração, o cálculo de imposto era realizado por meio de uma estrutura condicional *switch*, que centralizava toda a lógica em um único método e tornava o código menos flexível, apesar de intuitivo. Após a refatoração com o uso de polimorfismo, o fluxo de execução tornou-se mais linear e previsível, com cada tipo de produto encapsulando sua própria lógica de cálculo de imposto, as mudanças são visíveis no Quadro 13.

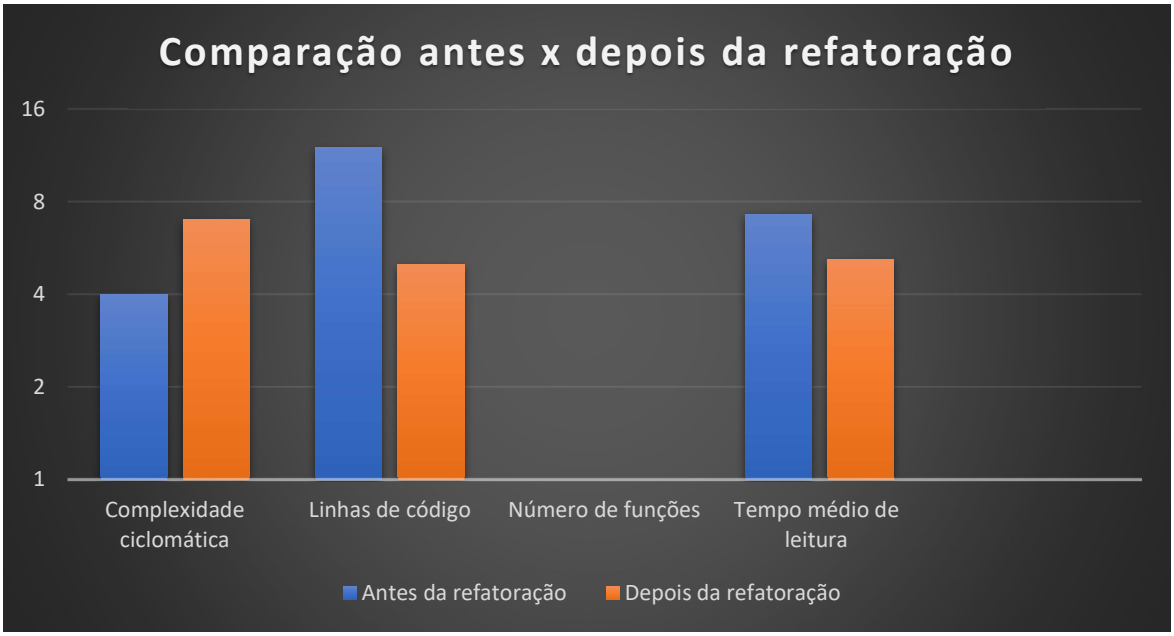
Quadro 13 - Resultados da refatoração

Métrica	Antes da refatoração	Depois da refatoração
Complexidade ciclomática	4	7
Linhas de código	12	5
Números de métodos	1	1
Tempo de execução (µs)	7,3	3,6

Fonte: Elaboração própria (2025).

O Gráfico 3 mostra a comparação entre a complexidade ciclomática, as linhas de código, números de funções e tempo médio de leitura, antes de depois da refatoração.

Gráfico 3 - Comparação antes e depois da refatoração (exemplo 3)



Fonte: Elaboração própria (2025).

4.4.4 Discussão crítica dos resultados

A substituição de condicionais por polimorfismo trouxe maior coesão, extensibilidade e respeito aos princípios de orientação a objetos. Embora o número de classes aumente, o comportamento se torna mais previsível e isolado. Além disso, a complexidade ciclomática pode parecer ter aumentado, mas está subdividida entre os métodos das novas classes. Quanto ao cálculo da eficiência do tempo de execução:

((7,3 – 3,6) / 7,3) * 100 = 50,68%

A refatoração permitiu:

- Eliminar estruturas switch ou múltiplos *if* espalhados;
- Facilitar a adição de novos tipos sem tocar no código existente;
- Melhorar o encapsulamento das regras de negócio;
- Testes específicos por classe/tipo com clareza;
- Aumentar a eficiência de tempo de execução em torno de 50,68%.

O principal desafio para o exemplo presente foi a reorganização das classes e aplicação dos conceitos de programação orientada a objetos, mas o ganho em modularidade e principalmente em tempo mostrou a importância e relevância do estudo.

4.5 Exemplo 4 – técnica: decompor condicional

A técnica Decompor Condicional tem como objetivo tornar expressões condicionais complexas mais legíveis e compreensíveis ao extrair suas partes para métodos nomeados. Isso reduz a carga cognitiva, melhora a clareza e facilita a manutenção.

4.5.1 Exemplo 4

O código do exemplo 4 é responsável por verificar se um cliente pode ser aprovado para um possível empréstimo, considerando idade, renda e histórico de crédito. A lógica de aprovação está concentrada em uma única condicional composta. O grafo do exemplo proposto define os caminhos para a finalização do programa conforme está apresentado na Figura 10. Quanto aos problemas estruturais e complexidade, estão descritas no item A e B.

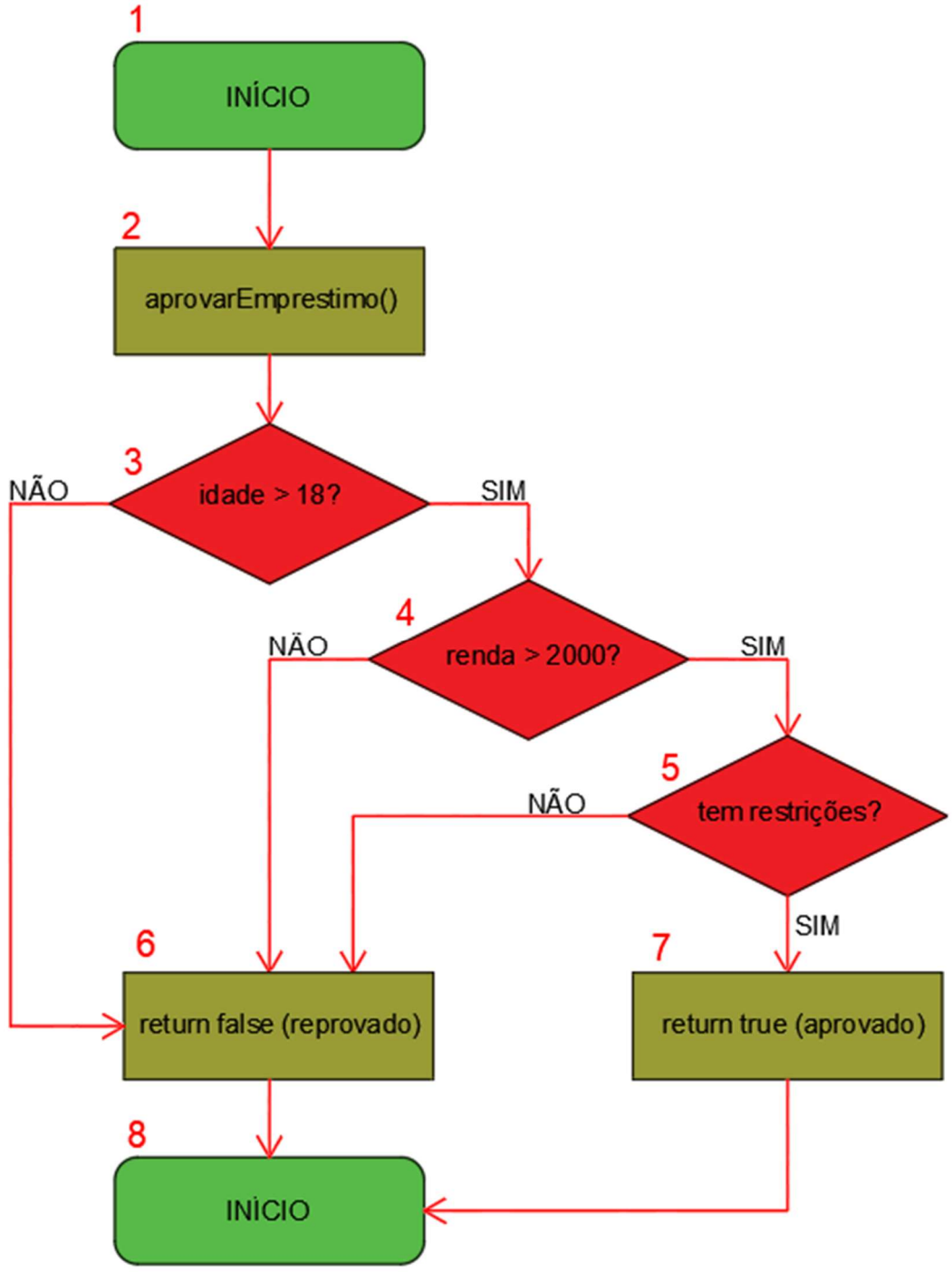
A) Principais problemas estruturais:

- Lógica de decisão condensada em uma única expressão booleana;
- Dificuldade de entender os critérios individuais;
- Baixa legibilidade e reutilização de lógica restrita.

B) Aplicando a complexidade ciclomática, tem-se os seguintes valores:

- Complexidade medida com o *MetricsReload* = 4;
- Aplicando a fórmula de McCabe:
  V(G) = E - N + 2P = V(G) = 7 - 5 + 2 * 1 = 4.

Figura 10 – Grafo do exemplo 4



Fonte: Elaboração própria (2025), baseado em (McCabe, 1976).

No grafo criado, cada bloco ou nó foi numerado como os anteriores e os caminhos possíveis estão apresentados a seguir no Quadro 14.

Quadro 14 - Caminhos possíveis do exemplo 4

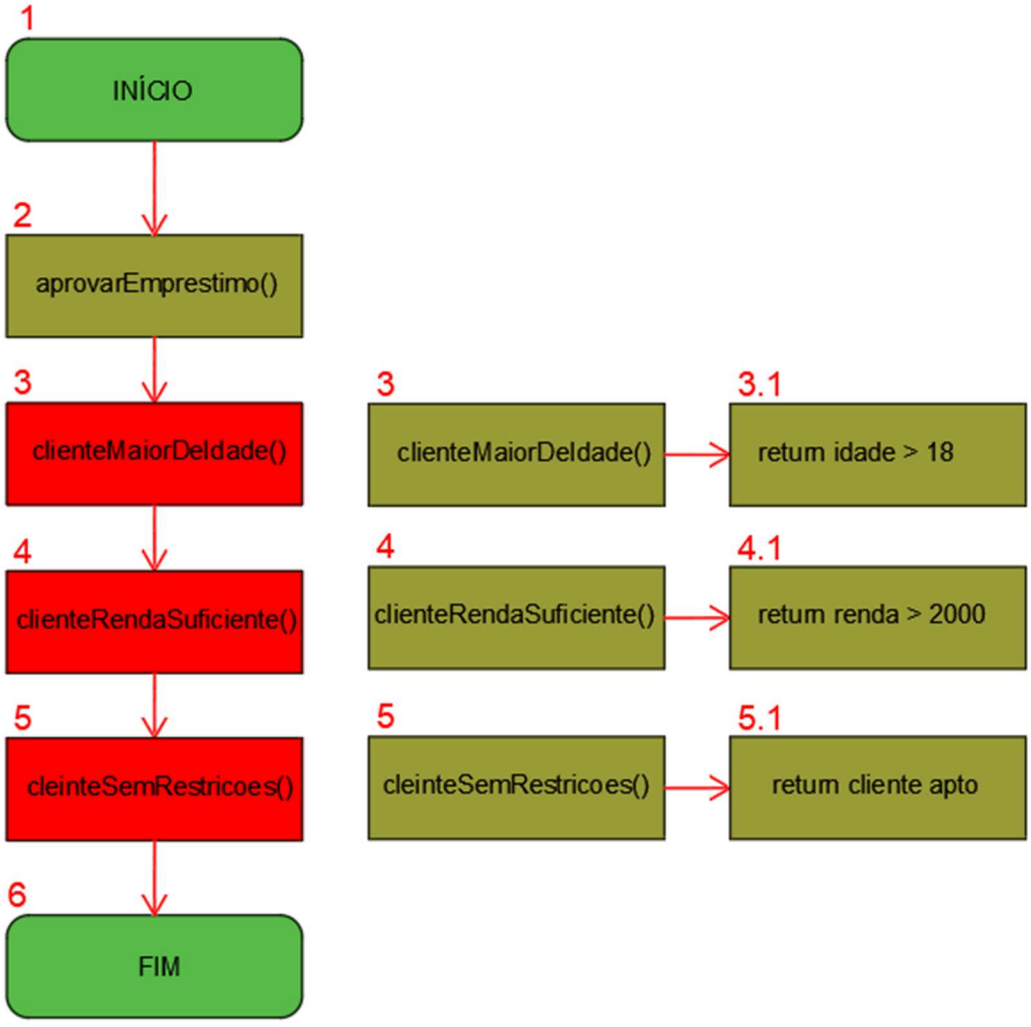
Lista de caminhos possível (independentes e dependentes)	
Caminhos independentes	
Caminho 1	1 → 2 → 3 → 4 → 5 (*true*)
Caminho 2	1 → 2 → 3 → 5
Caminho 3	1 → 2 → 5
Caminho já percorrido	
Caminho 4	1 → 2 → 3 → 4 → 5 (*false*)

Fonte: Elaboração própria (2025), baseado em (McCabe, 1976).

4.5.2 Aplicação da técnica de refatoração

No Exemplo 4, a técnica de refatoração aplicada consiste na extração de cada critério de aprovação para métodos nomeados individualmente. Essa abordagem torna a lógica do código mais legível e expressiva, como demonstrado a seguir, na Figura 11.

Figura 11 – Grafo do exemplo 4 refatorado



Fonte: Elaboração própria (2025), baseado em (McCabe, 1976).

4.5.3 Análise comparativa antes e depois da refatoração

O método responsável pela aprovação do empréstimo concentrava toda a lógica em uma única expressão condicional composta, o que tornava o código menos legível e dificultava a compreensão dos critérios avaliados. Após a refatoração, os critérios foram extraídos para métodos nomeados individualmente, tornando a lógica mais clara, modular e expressiva. Entretanto, mesmo com a modularização do código, essa mudança não se mostrou eficaz suficiente. A comparação entre as duas versões evidencia os resultados significativos da refatoração, conforme apresenta o Quadro 15.

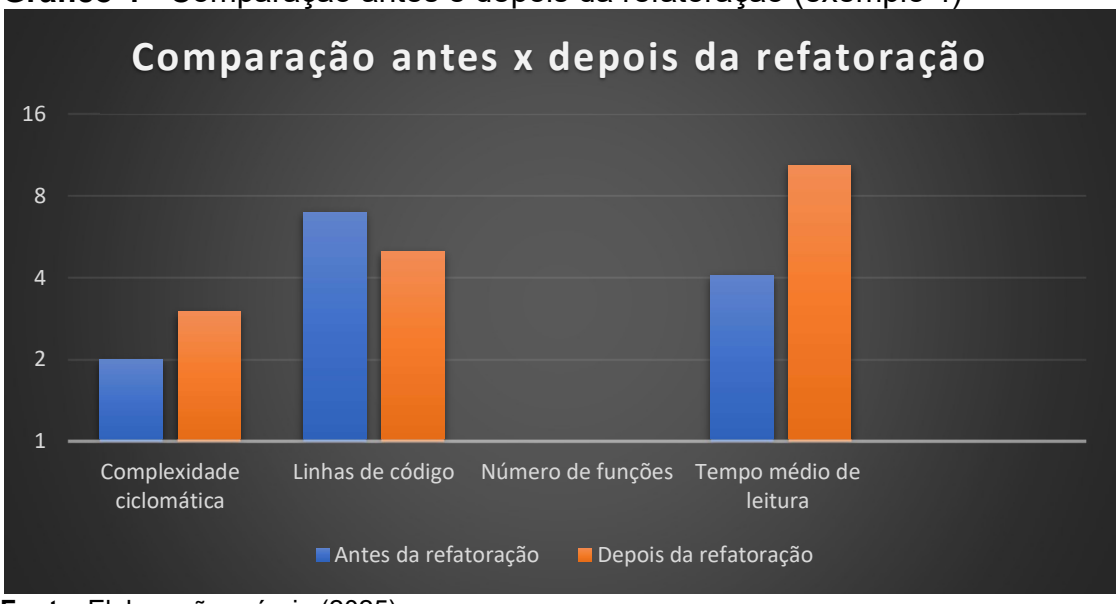
Quadro 15 - Resultados da refatoração

Métrica	Antes da refatoração	Depois da refatoração
Complexidade ciclomática	4	3
Linhas de código	7	5
Números de métodos	1	1
Tempo de execução (µs)	4,1	10,4

Fonte: Elaboração própria (2025).

A comparação do antes e depois da refatoração é mostrada no Gráfico 4.

Gráfico 4 - Comparação antes e depois da refatoração (exemplo 4)



Fonte: Elaboração própria (2025).

4.5.4 Discussão crítica dos resultados

Apesar da queda em tempo de execução, a decomposição da condicional aumentou significativamente a expressividade e clareza do código. A leitura se tornou quase autoexplicativa, e os critérios foram isolados em métodos reutilizáveis, possibilitando a possível reutilização do código. Quanto ao cálculo da eficiência do tempo de execução:

((4,1 – 10,4) / 4,1) * 100 = 153,65%

A refatoração permitiu:

- Melhorar a organização das regras de negócio;
- Facilitar a manutenção e testes unitários por critério;
- Reduziu o risco de erros em futuras alterações;
- Aumentar o tempo de execução em mais de 100%.

No exemplo estudado, a principal dificuldade residiu na demora para compreender as possíveis causas da redução no desempenho, ou seja, o aumento do tempo de execução. A possível causa no aumento de tempo pode estar relacionada na estrutura refatorada com novos métodos separados ou até mesmo no hardware utilizado no teste. Contudo, a técnica de refatoração demonstrou-se útil para promover uma melhor organização e divisão do código-fonte, facilitando seu entendimento e manutenção.

4.6 Exemplo 5 – técnica: consolidar expressão condicional

A técnica Consolidar Expressão Condicional consiste em unir múltiplas condições booleanas que resultam no mesmo desfecho em uma única expressão lógica composta. Essa abordagem reduz a duplicação de código e aprimora a legibilidade ao eliminar ramos redundantes no fluxo de controle, tornando a lógica mais clara e concisa.

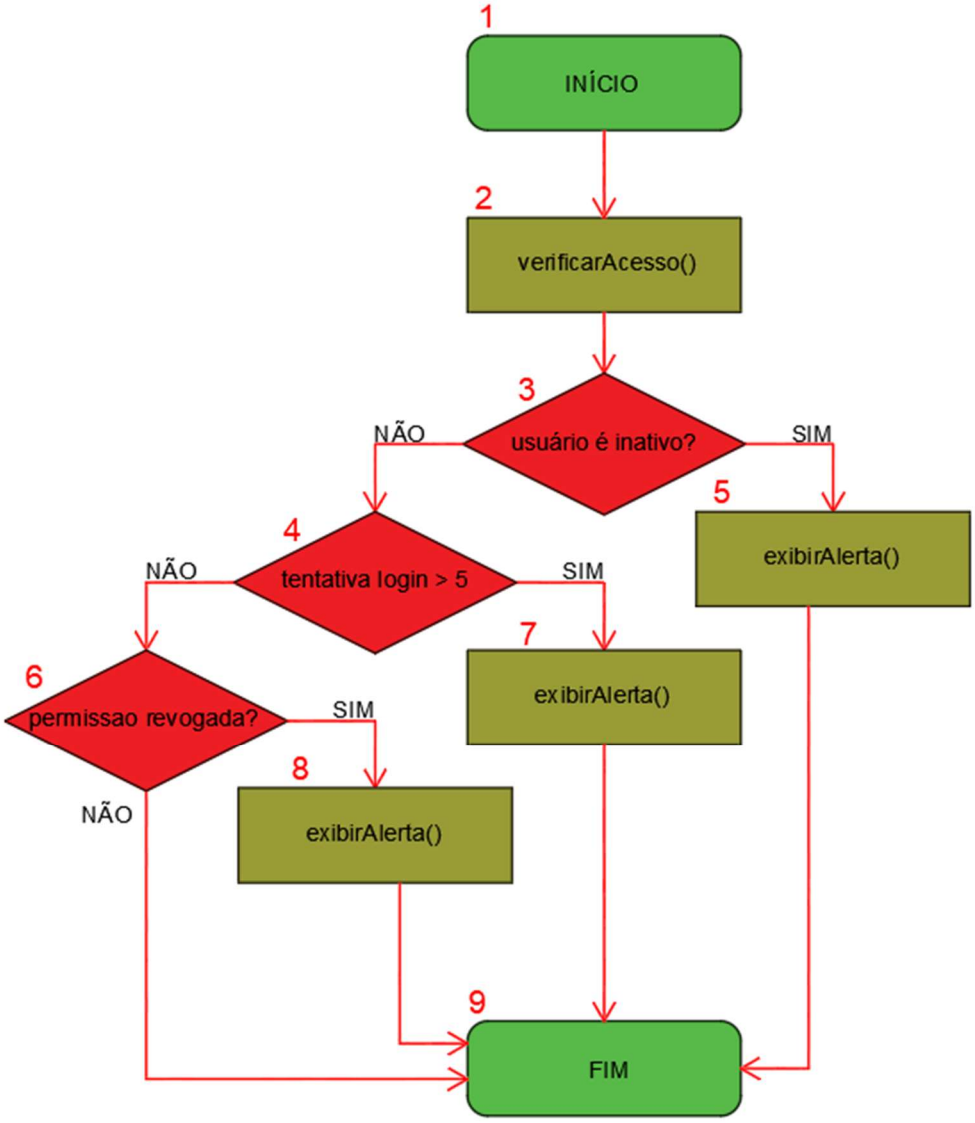
4.6.1 Exemplo 5

O método mostrado no exemplo 5 realiza uma série de verificações relacionadas à segurança do usuário, avaliando diferentes condições de forma individual. Cada condição representa um critério que, se satisfeito, indica um possível problema de acesso ou status do usuário. Embora as verificações sejam feitas separadamente, todas levam à mesma ação: a exibição de uma mensagem de alerta para notificar sobre a situação identificada. Quanto ao grafo de fluxo, está representado na Figura 12 e nos itens A e B estão descritos os problemas estruturais e métrica aplicada.

A) Problemas estruturais definidos a seguir:
- Repetição da mesma ação (exibirAlerta());
- Crescimento desnecessário do código com cada nova condição;
- Dificuldade de manutenção e testes;
- Violação do princípio *DRY* (*Don't Repeat Yourself*).

B) Aplicando a complexidade ciclomática, tem-se os seguintes valores:
- Complexidade medida com o *MetricsReload* → 4;
- Aplicando a fórmula de McCabe:
  V(G) = E - N + 2P = V(G) = 10 - 8 + 2 * 1 = 4.

Figura 12 – Grafo do exemplo 5



Fonte: Elaboração própria (2025), baseado em (McCabe, 1976).

No grado de fluxo cada nó representado segue o caminho até a finalização do programa. Assim todos os caminhos possíveis para o exemplo proposto estão apresentados no Quadro 16.

Quadro 16 - Caminhos possíveis do exemplo 4

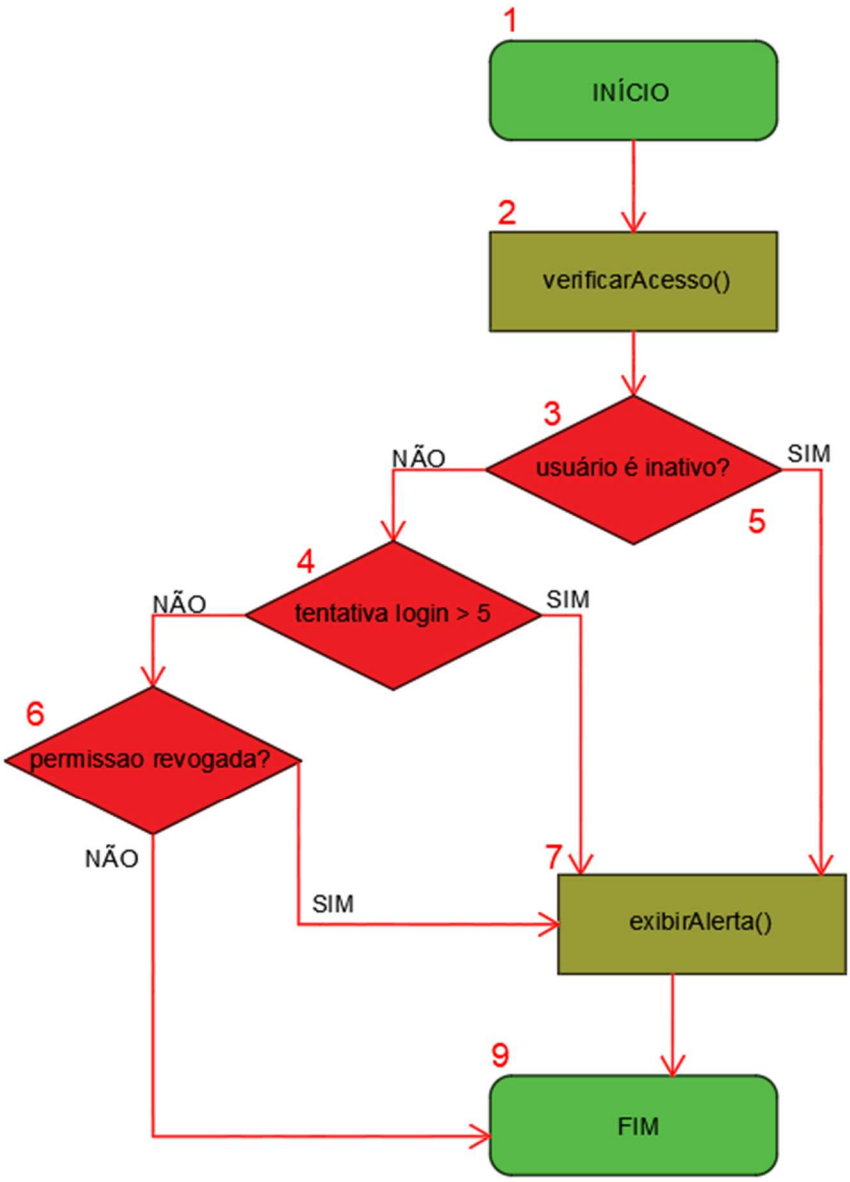
Lista de caminhos possível (independentes e dependentes)	
Caminhos independentes	
Caminho 1	1 → 2 → 3 → 4 → 6 → 9
Caminho 2	1 → 2 → 3 → 4 → 6 → 8 → 9
Caminho 3	1 → 2 → 3 → 4 → 7 → 9
Caminho 4	1 → 2 → 3 → 5 → 9

Fonte: Elaboração própria (2025), baseado em (McCabe, 1976).

4.6.2 Aplicação da técnica de refatoração

No exemplo final, todas as expressões condicionais foram unificadas em uma única condição lógica combinada, simplificando a estrutura do código. Dessa forma, o método verificarAcesso() verifica se o usuário está inativo, se excedeu o número permitido de tentativas de login ou se possui permissão revogada, e executa a exibição do alerta caso qualquer uma dessas condições seja verdadeira, conforme grafo na Figura 13.

Figura 13 – Grafo do exemplo 5 refatorado



Fonte: Elaboração própria (2025), baseado em (McCabe, 1976).

4.6.3 Análise comparativa antes e depois da refatoração

No exemplo 5, o método avaliava as condições de segurança de forma sequencial e separada, com múltiplos blocos condicionais. Cada condição verificava um aspecto diferente do usuário e todas levavam à mesma ação de exibir um alerta. Depois da refatoração, o método foi simplificado ao consolidar todas as condições em uma única expressão lógica usando o operador *OR* (||).

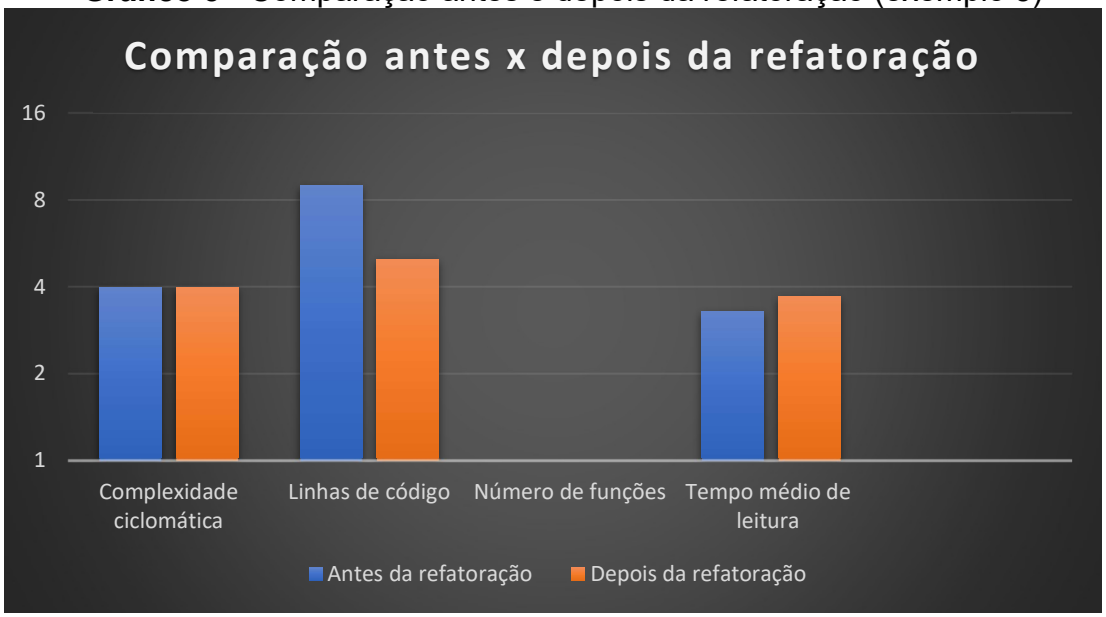
Reduzindo a redundância e tornou o código mais compacto conforme apresentado no Quadro 17. Quanto a comparação do antes de depois da refatoração pode ser vista no Gráfico 5

Quadro 17 - Resultados da refatoração

Métrica	Antes da refatoração	Depois da refatoração
Complexidade ciclomática	4	4
Linhas de código	9	5
Números de métodos	1	1
Tempo de execução (µs)	3,7	3,3

Fonte: Elaboração própria (2025).

Gráfico 5 - Comparação antes e depois da refatoração (exemplo 5)



Fonte: Elaboração própria (2025).

4.6.4 Discussão crítica dos resultados

A consolidação das expressões condicionais tornou o código mais limpo, direto e fácil de entender. Agrupar em uma única expressão lógica as condições que resultam na mesma ação facilita a leitura, elimina repetições desnecessárias e contribui para um código mais coeso. Quanto ao cálculo da eficiência do tempo de execução:

((3,7 – 3,3) / 3,7) * 100 = 10,81%

A refatoração permitiu:

- Simplificar o fluxo de controle;
- Tornar o código mais enxuto;
- Facilitar a adição de novas condições no futuro;
- Reduzir o número de ramos de decisão no código;
- Subir levemente o tempo de execução em 10,81%.

Apesar da leve perda de desempenho com um aumento de aproximadamente 10,81% no tempo de execução, o código tornou-se significativamente mais legível, coeso e fácil de manter. A clareza obtida com a refatoração compensa o pequeno impacto na performance, especialmente em contextos que a legibilidade e a manutenibilidade são prioridades como na análise de complexidade ciclomática.

5 CONSIDERAÇÕES FINAIS

Diante da análise realizada, é possível compreender a relevância da complexidade ciclomática para a refatoração de códigos. Este trabalho permitiu evidenciar como a redução da complexidade pode contribuir para um código mais legível, manutenível e de melhor qualidade.

5.1 Síntese dos resultados

A crescente demanda por qualidade na Engenharia de *Software* reforça a importância de métricas que auxiliem na análise e melhoria do código. Neste contexto, a complexidade ciclomática mostrou-se uma ferramenta eficaz para identificar trechos de código com estrutura excessivamente complexa, que dificultam a manutenção, os testes e a compreensão.

Com base nos exemplos analisados, observou-se que métodos com alta complexidade ciclomática tendem a apresentar estruturas condicionais profundas, o que compromete a legibilidade e a testabilidade do código. A aplicação das técnicas estudadas contribuiu significativamente para a redução da complexidade, dividindo entre métodos secundários e para a melhoria da organização interna do código.

A utilização de ferramentas específicas, como plugins de análise de complexidade e geradores de grafos de fluxo, foi essencial para visualizar e quantificar a complexidade, facilitando a tomada de decisões durante a refatoração. Conclui-se, portanto, que a métrica de complexidade ciclomática, aliada a boas práticas de refatoração, pode contribuir para a produção de códigos mais limpos, sustentáveis e fáceis de manter.

5.2 Limitações do estudo

Este trabalho foi conduzido com exemplos selecionados e controlados, o que, embora tenha permitido uma análise detalhada, não necessariamente representa todos os contextos encontrados em projetos reais. Além disso, o foco exclusivo na complexidade ciclomática limitou a abrangência da avaliação da qualidade do código.

Nesse sentido, quanto as ferramentas utilizadas, grande parte das que foram mencionadas anteriormente na metodologia são pagas ou privadas, dificultando a utilização de ferramenta mais específica. No mais, surgiram outros problemas na elaboração da estrutura do trabalho, começando na ferramenta *Overleaf*, depois no *Google Docs* e por fim no *Microsoft Word*.

5.3 Sugestões de trabalhos futuros

Como possibilidades de continuidade, sugere-se:

- Ampliar o estudo com a análise de projetos reais e de maior escala;
- Integrar outras métricas de qualidade de software para uma avaliação mais completa, como acoplamento, coesão e cobertura de testes;
- Investigar o uso do grafo de fluxo de controle como ferramenta didática para o ensino de algoritmos, associando visualização de lógica com análise de complexidade.

5.4 Contribuições da pesquisa

O presente trabalho contribui para a área de Engenharia de *Software* ao demonstrar, de forma prática, como a métrica de complexidade ciclomática pode ser utilizada para diagnosticar e reduzir problemas em códigos-fonte. A análise de código, combinada com o uso de ferramentas de visualização para os grafos de fluxo de controle, evidenciou a importância do estudo.

Além disso, o estudo sugere a possibilidade de aplicação pedagógica da métrica e do grafo de fluxo, integrando o ensino de algoritmos, demonstrando de forma visual os caminhos que uma estrutura pode ter, o que pode beneficiar estudantes e iniciantes na programação.

REFERÊNCIAS

AHO, Alfred V. *et al.* **Compiladores - Princípios, técnicas e ferramentas**. 2º. ed. São Paulo: Pearson Education do Brasil, v. I, 2008.

AJALA, Vinícius *et al.* Análise da complexidade ciclomática como apoio ao processo de desenvolvimento do pensamento algorítmico. **Análise da Complexidade Ciclomática como Apoio ao Processo de Desenvolvimento do Pensamento Algorítmico**, Santo Ângelo, 4 Julho 2016. 587-596.

BARICHELLO, Luis Gustavo Egger. Identificação de oportunidades de refatoração e análise de qualidade por meio de métricas de código-fone. **Identificação de oportunidades de refatoração e análise de qualidade por meio de métricas de código-fone**, Dois vizinhos, 23 Junho 2022. 94.

FOWLER, Martin. **Refatoração**. 2ª. ed. São Paulo: Novatec, v. I, 2019.

GOMES, Paulo César Rodacki. **Grafos - Conceitos fundamentais, algoritmos e aplicações**. 1ª. ed. Blumenau: IFC, v. I, 2022. 5-8 p.

GOMES, Pedro Henrique de Andrade. Inspeção de código-fonte como subsídio para o processo de ensino e aprendizagem de qualidade de software. **Inspeção de código-fonte como subsídio para o processo de ensino e aprendizagem de qualidade de software**, São José do Rio Preto, 22 Junho 2021. 147.

JÚNIOR, Marcus Aurélio Cordeiro Piancó. Analisando as relações entre mudanças no código fonte e bugs no software. **Analisando as relações entre mudanças no código fonte e bugs no software**, Maceió, 19 Outubro 2017. 67.

MARTIN, Robert C. **Código lipo (Clean code)**. 1ª. ed. Rio de Janeiro: Alta Books, v. I, 2011.

MCCABE, Thomas J. Uma Medida de Complexidade. **Complexidade Ciclomática**, Storrs, 4 Dezembro 1976. 308-320.

MELO, Gildson Soares de. Introdução à Teoria dos Grafos. **Introdução à Teoria dos Grafos**, João Pessoa, 22 Agosto 2014. 35.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de Spftware - Uma abordagem profissional**. 8ª. ed. Porto Alegre: AMGH Editora Ltda, v. I, 2016.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de Software - Uma abordagem profissional**. 9ª. ed. Porto Alegre: AMGH Editora Ltda., v. II, 2021.

RIBEIRO, Talita V.; TRAVASSOS, Guilherme H. Alinhando Perspectivas de Qualidade em Código Fonte a Partir de Estudos Experimentais - Um Caso na Indústria. **Alinhando Perspectivas de Qualidade em Código Fonte a Partir de Estudos Experimentais - Um Caso na Indústria**, Rio de Janeiro, 17 Agosto 2015. 15.

WATSON, Arthur H.; MCCABE, Thomas J. Teste Estruturado: Uma Metodologia de Teste - Usando a Métrica de Complexidade Ciclomática. **Teste Estruturado: Uma Metodologia de Teste - Usando a Métrica de Complexidade Ciclomática**, Ellicott City, 20 Setembro 1996. 124.

WAZLAWICK, Raul Sidnei. **Metodologia de pesquisa para Ciência da Computação**. 3ª. ed. Rio de Janeiro: GEN, v. I, 2021.

WIJENDRA, Dinuka ; HEWAGAMAGE, KP. Análise da Complexidade Cognitiva com Ciclomática. **Análise da Complexidade Cognitiva com Métrica de Complexidade Ciclomática de Software**, Colombo, Sri Lanka, 19 Fevereiro 2021. 14-19.

APÊNDICES

APÊNDICE A – CÓDIGOS DO EXEMPLO 1

Apêndice A.1 – Código original

```
public class DescontoService {
    public double calcularDesconto(Cliente cliente, double
valorCompra) {
        double desconto = 0;
        if (cliente.getTipo().equals("vip")) {
            if (valorCompra > 1000) {
                desconto = valorCompra * 0.2;
            } else {
                desconto = valorCompra * 0.1;
            }
        } else if (cliente.getTipo().equals("regular")) {
            if (valorCompra > 500) {
                desconto = valorCompra * 0.05;
            }
        }
        if (cliente.isAniversario()) {
            desconto += 20;
        }
        return desconto;
    }
}
```

Apêndice A.2 – Código refatorado

```
public class DescontoServiceRefatorado {
    public double calcularDesconto(Cliente cliente, double
valorCompra) {
        double desconto = calcularDescontoPorTipo(cliente,
valorCompra);
        desconto += calcularDescontoAniversario(cliente);
        return desconto;
    }
    private double calcularDescontoPorTipo(Cliente cliente, double
valorCompra) {
        if (cliente.getTipo().equals("vip")) {
            return valorCompra > 1000 ? valorCompra * 0.2 :
valorCompra * 0.1;
        } else if (cliente.getTipo().equals("regular") &&
valorCompra > 500) {
            return valorCompra * 0.05;
        }
        return 0;
    }
    private double calcularDescontoAniversario(Cliente cliente) {
        return cliente.isAniversario() ? 20 : 0;
    }
}
```

APÊNDICE B – CÓDIGOS DO EXEMPLO 2

Apêndice B.1 – Código original

```
public String verificarStatusUsuario(Usuario usuario) {
    if (usuario != null) {
        if (usuario.isAtivo()) {
            if (!usuario.isBanido()) {
                return "Acesso permitido";
            } else {
                return "Usuário banido";
            }
        } else {
            return "Usuário inativo";
        }
    }
    return "Usuário inválido";
}
```

Apêndice B.2 – Código refatorado

```
public String verificarStatusUsuario(Usuario usuario) {
    if (usuario == null) return "Usuário inválido";
    if (!usuario.isAtivo()) return "Usuário inativo";
    if (usuario.isBanido()) return "Usuário banido";
    return "Acesso permitido";

}
```

APÊNDICE C – CÓDIGOS DO EXEMPLO 3

Apêndice C.1 – Código original

```
public static double calcularTaxa(Produto produto) {
    switch (produto.getTipo()) {
        case "ALIMENTO":
            return produto.getPreco() * 0.05;
        case "ELETRONICO":
            return produto.getPreco() * 0.15 + 10;
        case "LIVRO":
            return 0;
        default:
            throw new IllegalArgumentException("Tipo de produto
inválido");
    }
}
```

Apêndice C.2 – Código refatorado

```
abstract class ProdutoAbstrato {
    protected double preco;
    public ProdutoAbstrato(double preco) { this.preco = preco; }

    public abstract double calcularImposto();
}
class Alimento extends ProdutoAbstrato {
    public Alimento(double preco) { super(preco); }
    public double calcularImposto() { return preco * 0.05; }

}
class Eletronico extends ProdutoAbstrato {
    public Eletronico(double preco) { super(preco); }
    public double calcularImposto() { return preco * 0.15 + 10; }

}
class Livro extends ProdutoAbstrato {

    public Livro(double preco) { super(preco); }
    public double calcularImposto() { return 0; }

}
```

APÊNDICE D – CÓDIGOS DO EXEMPLO 4

Apêndice D.1 – Código original

```
public static boolean aprovarEmprestimo(Cliente cliente) {
    if (cliente.getIdade() > 18 && cliente.getRenda() > 2000 &&
!cliente.temRestricoes()) {
        return true;
    }
    return false;
}
```

Apêndice D.2 – Código original

```
public static boolean aprovarEmprestimo(Cliente cliente) {
    return clienteEhMaiorDeIdade(cliente)
        && clienteTemRendaSuficiente(cliente)
        && clienteNaoTemRestricoes(cliente);
}
private static boolean clienteEhMaiorDeIdade(Cliente cliente) {
    return cliente.getIdade() > 18;
}
private static boolean clienteTemRendaSuficiente(Cliente cliente) {
    return cliente.getRenda() > 2000;
}
private static boolean clienteNaoTemRestricoes(Cliente cliente) {
    return !cliente.temRestricoes();
}
```

APÊNDICE E – CÓDIGOS DO EXEMPLO 5

Apêndice E.1 – Código original

```
public static void verificarAcesso(Usuario2 usuario) {
    if (usuario.isInativo()) {
        exibirAlerta();
    } else if (usuario.getTentativasLogin() > 5) {
        exibirAlerta();
    } else if (usuario.temPermissaoRevogada()) {
        exibirAlerta();
    }
}
```

Apêndice E.1 – Código original

```
public static void verificarAcesso(Usuario2 usuario) {
    if (usuario.isInativo() || usuario.getTentativasLogin() > 5 ||
usuario.temPermissaoRevogada()) {
        exibirAlerta();
   }
}
```