



**UNICESUMAR – UNIVERSIDADE CESUMAR**  
CENTRO DE CIÊNCIAS EXATAS TECNOLÓGICAS E AGRÁRIAS  
CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

**THIAGO LUIZ SOARES ARCOVERDE**

**PRINCÍPIOS DE UM CÓDIGO EFICIENTE: BOAS PRÁTICAS PARA ELEVAR A  
QUALIDADE DO SEU CÓDIGO**

Maringá/2024

**THIAGO LUIZ SOARES ARCOVERDE**

**PRINCÍPIOS DE UM CÓDIGO EFICIENTE: BOAS PRÁTICAS PARA ELEVAR A  
QUALIDADE DO SEU CÓDIGO**

Trabalho de Conclusão de Curso apresentado para  
obtenção do grau de bacharel em Engenharia de  
Software, do Centro Universitário Cesumar —  
Unicesumar

Orientador:

MARINGÁ/2024

**THIAGO LUIZ SOARES ARCOVERDE**

**PRINCÍPIOS DE UM CÓDIGO EFICIENTE: BOAS PRÁTICAS PARA ELEVAR A  
QUALIDADE DO SEU CÓDIGO**

Trabalho de Conclusão de Curso apresentado para  
obtenção do grau de bacharel em Engenharia de  
Software, do Centro Universitário Cesumar —  
Unicesumar

Orientador:.

**BANCA EXAMINADORA**

---

**Prof. (Nome do orientador)**

**Afiliações**

---

**Prof. (Nome do professor avaliador)**

**Afiliações**

---

**Prof. (Nome do professor avaliador)**

**Afiliações**

## **AGRADECIMENTOS**

Agradeço a Deus, primeiramente, por toda compaixão, amor, força e resiliência que me forneceu durante toda a minha vida, sem Ele nada seria possível.

A meu pai Eulampio Brito Arcoverde e minha mãe Rosangela Soares Teixeira, por todo apoio, carinho, ensinamentos e sacrifícios que fizeram por mim durante toda a minha vida.

A meu irmão Rodrigo, por ter me instruído e ajudado a começar minha carreira dentro do mundo da engenharia de software.

A minha irmã Renata, por ter me escutado e ajudado dentro de momentos de muita dificuldade em minha vida.

A minha irmã Laryssa, por me ensinar a ver o mundo de outra forma e ter me ajudado a entender as responsabilidades de um irmão mais velho.

A Luana, por todo o seu companheirismo, amizade, apoio e compreensão durante todo o curso.

Ao meu primo Luiz Henrique, que fez parte da minha jornada dentro do mundo de software e me ajudou em muitas dificuldades antes e durante o curso.

A meus amigos, que fizeram parte do dia a dia do curso e conseguiram alegrar momentos de dificuldade e me permitiram compartilhar minhas felicidades.

Ao professor João Choma Neto, por me orientar e ajudar durante todo o processo do meu trabalho de conclusão de curso e me ajudar a fazer isso tudo ser possível.

Aos professores Alexandre Moreno, Aparecido Vilela Junior, Erinaldo Sanches e Thiago Bussola, por me ajudarem a entender como aprender o que é software e o que está muitas vezes além dele.

## RESUMO

Este trabalho tem como objetivo explorar a relação entre a adoção de boas práticas de desenvolvimento de software e a redução da complexidade ciclomática do código. No cenário atual de crescente demanda por software de alta qualidade, torna-se essencial compreender como práticas como DRY (Don't Repeat Yourself), Early Return e Pattern Matching podem influenciar tanto a legibilidade quanto a eficiência do código fonte. Essas abordagens, que fazem parte das metodologias de Clean Code e Qualidade de Software, têm o potencial de melhorar a clareza, a testabilidade e a manutenibilidade do código. A complexidade ciclomática é uma métrica importante que quantifica a complexidade estrutural de um programa, contabilizando o número de caminhos independentes no código. Reduzir essa complexidade facilita a compreensão do código, além de contribuir para a redução de erros e aumento da produtividade durante a manutenção. O objetivo geral deste trabalho é aplicar técnicas como DRY e Pattern Matching para analisar o impacto de sua adoção na complexidade ciclomática de trechos de código. Para isso, será realizada uma análise detalhada através de um estudo de caso, complementada por revisão teórica sobre as boas práticas de desenvolvimento e suas implicações na qualidade do software. A metodologia envolve a aplicação dessas técnicas em trechos de código do livro “Codigo Limpo” de Robert C. Martin, seguido da medição da complexidade ciclomática e outras métricas antes e após as mudanças, com o intuito de verificar a redução da complexidade e os efeitos resultantes. Os resultados esperados incluem a demonstração de que a adoção dessas boas práticas leva à simplificação do código, o que facilita a legibilidade e a manutenção do código. A conclusão do trabalho reforça a importância da aplicação consciente de tais técnicas como um caminho para o desenvolvimento de software mais eficiente, claro e de fácil manutenção, beneficiando tanto os desenvolvedores quanto os usuários finais.

**Palavras-chave:** Complexidade ciclomática. Qualidade. Software. Desenvolvimento.

## ABSTRACT

This work aims to explore the relationship between the adoption of good software development practices and the reduction of cyclomatic code complexity. In the current scenario of growing demand for high-quality software, it is essential to understand how practices such as DRY (Don't Repeat Yourself), Early Return and Pattern Matching can influence both the readability and efficiency of the source code. These approaches, which are part of Clean Code and Software Quality methodologies, have the potential to improve code clarity, testability and maintainability. Cyclomatic complexity is an important metric that quantifies the structural complexity of a program, counting the number of independent paths in the code. Reducing this complexity makes the code easier to understand, in addition to contributing to reducing errors and increasing productivity during maintenance. The general objective of this work is to apply techniques such as DRY and Pattern Matching to analyze the impact of their adoption on the cyclomatic complexity of code snippets. To this end, a detailed analysis will be carried out through a case study, complemented by a theoretical review of good development practices and their implications for software quality. The methodology involves applying these techniques to code snippets from the book "Codigo Limpo" by Robert C. Martin, followed by measuring cyclomatic complexity and other metrics before and after the changes, with the aim of verifying the reduction in complexity and the effects resulting. Expected results include demonstrating that adopting these best practices leads to code simplification, which facilitates code readability and maintainability. The conclusion of the work reinforces the importance of the conscious application of such techniques as a path to the development of more efficient, clear and easy to maintain software, benefiting both developers and end users.

**Keywords:** Cyclomatic complexity, Quality, Software, Development.

## LISTA DE FIGURAS

Figura 1 - Método createPluralDependentMessageParts.....	16
Figura 2 - Método createPluralDependentMessageParts atualizado.....	17
Figura 3 - Método getNextOrNull.....	18
Figura 4 - Método getNextOrNull atualizado.....	18
Figura 5 - Método isInRange.....	19
Figura 6 - Método isInRange atualizado.....	20

## SUMÁRIO

<b>RESUMO</b>	<b>4</b>
<b>ABSTRACT</b>	<b>5</b>
<b>LISTA DE FIGURAS</b>	<b>6</b>
<b>SUMÁRIO</b>	<b>7</b>
<b>1. INTRODUÇÃO</b>	<b>8</b>
<b>2. OBJETIVO GERAL</b>	<b>11</b>
<b>2.1 OBJETIVOS ESPECÍFICOS</b>	<b>11</b>
<b>3. FUNDAMENTAÇÃO TEÓRICA</b>	<b>11</b>
3.1 COMPLEXIDADE CICLOMÁTICA (CC)	11
3.2 CÁLCULO DE CC	12
3.3 FERRAMENTAS PARA CÁLCULO DE CC	12
3.4 O CÓDIGO LIMPO E SEU IMPACTO	13
<b>4. TRABALHOS RELACIONADOS</b>	<b>14</b>
<b>5. METODOLOGIA</b>	<b>14</b>
5.1 ANÁLISE TEÓRICA	14
5.2. ESTUDO DE CASO	15
5.3. ANÁLISE DAS MÉTRICAS	15
5.4. ESTUDO COMPARATIVO	16
5.5. CONSIDERAÇÕES FINAIS	16
<b>6. RESULTADOS</b>	<b>17</b>
6.1 MÉTODO createPluralDependentMessageParts	17
6.2 MÉTODO getNextOrNull	18
6.3 MÉTODO isInRange	20
<b>7. Conclusão</b>	<b>22</b>
<b>REFERÊNCIAS BIBLIOGRÁFICAS</b>	<b>24</b>



## 1. INTRODUÇÃO

Atualmente, vivemos em um contexto em que a demanda por sistemas complexos e multifuncionais aumenta constantemente. Nesse ambiente dinâmico e desafiador, a engenharia de software desempenha um papel fundamental, não apenas no desenvolvimento de soluções que atendam às necessidades dos usuários, mas também na construção de sistemas que sejam sustentáveis e adaptáveis às mudanças. Ian Sommerville, em sua obra "Engenharia de Software", enfatiza a importância de um desenvolvimento que considere não apenas a funcionalidade, mas também a qualidade e a manutenibilidade dos sistemas.

Para atingir esses objetivos, a adoção de padrões e boas práticas se revela essencial. Esses princípios orientadores são fundamentais para a criação de um código que não apenas funcione corretamente, mas que também seja claro, compreensível e fácil de manter ao longo do tempo. Robert C. Martin, conhecido como "Uncle Bob" e autor do influente livro "Código Limpo", argumenta que um código de alta qualidade deve ser caracterizado por aspectos como a escolha adequada de nomes de variáveis, a definição de funções coesas e a organização de classes de forma lógica. Tais práticas não apenas melhoram a legibilidade do código, mas também promovem a colaboração entre equipes de desenvolvimento, reduzindo a incidência de erros e retrabalhos.

Neste trabalho, exploraremos os princípios que fundamentam a construção de um código eficiente, destacando como a aplicação consistente dessas boas práticas pode elevar a qualidade do software produzido. A análise incluirá tanto os conceitos teóricos que sustentam um código limpo quanto exemplos práticos e estratégias para a implementação desses princípios no dia a dia dos desenvolvedores. A busca por um código de qualidade é uma jornada contínua que exige reflexão, aprendizado e aprimoramento constante, e este trabalho se propõe a contribuir para essa discussão essencial no campo da engenharia de software.

Uma das práticas que iremos abordar neste trabalho é como o DRY(don't repeat yourself), que significa "não se repetir" em tradução livre, pode nos auxiliar a criar um código com menor complexidade ciclomática, que de acordo com Arthur H. Watson e Thomas J. McCabe é a medição da quantidade de lógica de decisão em

uma função de um código-fonte. Também usaremos a ideia de pattern matching, uma estrutura de decisão comum em linguagens como C#, F#, Python, Ruby e outras.

A complexidade ciclomática é uma métrica importante na análise de código, pois fornece uma medida quantitativa da complexidade estrutural de um programa. Ela reflete o número de caminhos independentes em um código-fonte, indicando a quantidade de decisões lógicas que precisam ser testadas durante a execução. Quanto maior a complexidade ciclomática, maior o número de caminhos possíveis e, portanto, maior o esforço necessário para realizar testes de cobertura e identificar falhas. Utilizar essa métrica no desenvolvimento de software oferece diversas vantagens, como a melhoria da qualidade do código, uma vez que permite identificar áreas mais complexas que podem precisar de refatoração para reduzir o risco de erros. Além disso, a complexidade ciclomática auxilia na estimativa do tempo e custo de testes, permitindo que os desenvolvedores priorizem testes em regiões de código com alta complexidade, o que pode aumentar a eficiência do processo de desenvolvimento e garantir uma maior confiabilidade no software final (MCCABE, 1976, BASILI et al., 1996).

Neste trabalho, serão explorados os princípios que fundamentam a construção de um código eficiente, destacando como a aplicação consistente dessas boas práticas pode elevar a qualidade do software produzido. A análise incluirá tanto os conceitos teóricos que sustentam um código limpo quanto exemplos práticos e estratégias para a implementação desses princípios no dia a dia dos desenvolvedores. A busca por um código de qualidade é uma jornada contínua que exige reflexão, aprendizado e aprimoramento constante, e este trabalho se propõe a contribuir para essa discussão essencial no campo da engenharia de software.

A metodologia adotada neste trabalho consistiu em um estudo de caso combinado com a análise de fundamentos teóricos relacionados à complexidade ciclomática. Inicialmente, foram selecionados trechos de código do livro “Codigo Limpo” de Robert C. Martin, os quais passaram por uma análise quantitativa para calcular suas métricas de complexidade ciclomática e índice de facilidade de manutenção antes de qualquer modificação. Em seguida, procedeu-se à aplicação de técnicas e boas práticas visando à redução da complexidade ciclomática ou

aumento do índice de facilidade de manutenção, seguida de uma nova medição das métricas para avaliar o impacto das alterações realizadas. Essa abordagem permitiu observar de forma prática a relação entre a refatoração e a melhoria nas métricas, além de fornecer uma base teórica sólida para interpretar os resultados. A fundamentação teórica serviu como alicerce para a escolha das métricas e estratégias de refatoração, enquanto o estudo de caso permitiu validar essas práticas em um contexto real de desenvolvimento de software. Com isso, o trabalho busca demonstrar de maneira empírica os benefícios da aplicação da complexidade ciclomática na melhoria da qualidade do código.

A estrutura do trabalho é organizada da seguinte maneira: na Seção 3 a fundamentação teórica, onde serão apresentados os conceitos básicos e as principais boas práticas que utilizaremos. Já na Seção 04, é apresentado os trabalhos relacionados, na qual fornecem uma visão aproximada do tema na qual é abordado neste trabalho. Na Seção 5 é apresentada a metodologia, na qual descreve o método de pesquisa adotado, incluindo os critérios de comparação e as ferramentas utilizadas. Seção 06, Resultados, onde é exemplificado e demonstrado a forma na qual o uso de boas práticas impacta as métricas que foram escolhidas previamente. A Seção 07 apresenta a conclusão do estudo e as principais descobertas e considerações finais.

## **2. OBJETIVO GERAL**

O objetivo geral deste trabalho é aplicar técnicas como o DRY e o Pattern Matching para analisar o impacto do uso delas na complexidade ciclomática de um trecho de código.

### **2.1 OBJETIVOS ESPECÍFICOS**

Para ser possível alcançar o objetivo geral deste trabalho, é necessário que os seguintes objetivos específicos sejam atingidos:

1. Investigar trechos de códigos onde técnicas e boas práticas de otimização, legibilidade e qualidade de software possam ser aplicadas;
2. Realizar o cálculo de CC antes de aplicar definidas técnicas

3. Aplicar técnicas;
4. Realizar o cálculo de CC após aplicação de técnicas definidas.

### **3. FUNDAMENTAÇÃO TEÓRICA**

#### **3.1 COMPLEXIDADE CICLOMÁTICA (CC)**

A complexidade ciclomática, introduzida por Thomas McCabe, é uma métrica fundamental na análise de software, que quantifica a quantidade de lógica de decisão presente em um módulo de software. Essa métrica é essencial em metodologias de teste estruturado, pois fornece uma estimativa do número mínimo de testes necessários para validar um módulo (MCCABE, 1976). Além disso, a complexidade ciclomática é uma ferramenta valiosa em todas as fases do ciclo de vida do software, começando pelo design, desempenhando um papel crucial na garantia de que o software permaneça confiável, testável e gerenciável (BASILI et al., 1996).

A complexidade ciclomática é baseada na estrutura do gráfico de controle de fluxo do software, que ilustra como as instruções são executadas. Esses gráficos descrevem a lógica estrutural dos módulos de software, onde cada módulo, normalmente, corresponde a uma função ou sub-rotina em linguagens de programação típicas. No contexto da linguagem C, um módulo é representado por uma função, a qual possui um único ponto de entrada e um único ponto de saída (BOOCH, 1994).

#### **3.2 CÁLCULO DE CC**

Definido por McCabe, a complexidade ciclomática (CC) é definida para cada módulo como  $CC = e - n + 2$ , onde  $e$  representa o número de arestas e  $n$  representa o número de nós no gráfico de controle de fluxo. Esta métrica não apenas quantifica a complexidade do módulo, mas também indica a quantidade mínima de testes necessários para cobrir todos os caminhos de execução possíveis.

A complexidade ciclomática é comumente expressa como  $v(G)$ , onde  $v$  representa o número ciclomático segundo a teoria dos grafos, e  $G$  refere-se ao

gráfico específico que modela o fluxo de controle de um programa ou módulo de software. Essa notação evidencia que a complexidade ciclomática é uma função da estrutura do gráfico, capturando as relações de controle entre os diferentes nós e arestas do fluxo de execução. O termo "ciclomático" está relacionado ao número de ciclos fundamentais em gráficos que são conectados e não direcionados (Berge, 1973).

### **3.3 FERRAMENTAS PARA CÁLCULO DE CC**

O cálculo da complexidade ciclomática (CC) pode ser facilitado por meio de ferramentas especializadas que automatizam a análise do código-fonte. Uma dessas ferramentas é o SonarLint, uma extensão open source que oferece suporte a diversas linguagens de programação e se integra a ambientes de desenvolvimento como IntelliJ IDEA, Eclipse e Visual Studio (SONAR, 2024).

O SonarLint é projetado para ajudar os desenvolvedores a identificar e corrigir problemas de qualidade e segurança no código em tempo real, à medida que escrevem. Entre suas funcionalidades, destaca-se a capacidade de calcular a complexidade ciclomática, permitindo que os programadores visualizem essa métrica diretamente em seu ambiente de desenvolvimento (SONAR, 2024).

Com o SonarLint, é possível obter feedback instantâneo sobre a complexidade ciclomática dos métodos e funções, o que auxilia na identificação de áreas do código que podem ser excessivamente complexas e, portanto, mais suscetíveis a erros e dificuldades de manutenção. A ferramenta fornece uma análise visual que destaca os pontos críticos, permitindo que os desenvolvedores façam ajustes e refatorações conforme necessário (SONAR, 2024).

Além disso, o SonarLint é parte do ecossistema SonarQube, que fornece análises mais abrangentes e relatórios detalhados sobre a qualidade do código. Com essa integração, os desenvolvedores podem realizar uma avaliação mais profunda da complexidade ciclomática e de outras métricas importantes, promovendo melhores práticas de codificação e contribuindo para a produção de software de maior qualidade (SONAR, 2024).

Além do SonarLint, também é possível utilizar a ferramenta de análise de métricas do código do Visual Studio, que além da complexidade ciclomática do seu projeto, te oferece outras ferramentas de qualidade de código como o índice de facilidade de manutenção. Utilizaremos essa ferramenta e essas duas métricas para analisar as alterações de código que realizaremos.

### 3.4 O CÓDIGO LIMPO E SEU IMPACTO

O conceito de código limpo (clean code) ganhou destaque na área de desenvolvimento de software principalmente a partir da obra de Robert C. Martin, no livro *Código Limpo: Habilidades Práticas do Agile Software* (MARTIN, 2009). Martin defende que escrever código limpo vai além de seguir boas práticas de sintaxe e estilo de programação: trata-se de escrever código que seja fácil de entender, fácil de manter e que favoreça a continuidade do desenvolvimento de um sistema sem que haja complicações ao longo do tempo.

De acordo com Martin (2009), alguns das qualidades de um código que é considerado limpo são:

- Facilidade em ler e entender ele: o código deve ser compreendido por outros desenvolvedores quando for preciso modificá-lo ou realizar algum tipo de manutenção. A clareza na nomenclatura de variáveis, funções e classes é algo essencial para garantir que o seu código seja intuitivo
- Simplicidade: um código limpo evita o uso de complexidade desnecessária. A simplicidade para Martin (2009), é um valor fundamental no desenvolvimento de software.
- Organização: todo código limpo deve possuir uma boa organização, sendo estruturado e modular, assim facilitando a manutenção e evolução do software. Seus métodos e funções devem ser coesos e com responsabilidades bem definidas.

## 4. TRABALHOS RELACIONADOS

Almeida e Miranda(2010), em *Código Limpo e seu Mapeamento para Métricas de Código Fonte*, apresentam uma visão sobre o que torna um código limpo e faz um levantamento de conceitos relacionados ao tema, sendo possível expor diversas técnicas e mapear algumas métricas sobre o conceito.

## 5. METODOLOGIA

O objetivo principal deste trabalho é avaliar o impacto da aplicação de boas práticas de desenvolvimento, conforme discutido no livro Código Limpo de Robert C. Martin, na melhoria da qualidade do código. A pesquisa adota uma abordagem mista, composta por uma análise teórica associada à aplicação prática em um estudo de caso, visando compreender como a refatoração de código, baseada em princípios como a simplicidade, clareza e modularidade, afeta as métricas de qualidade do código. A metodologia utilizada se divide em duas fases principais: a análise teórica e a aplicação do estudo de caso.

### 5.1 ANÁLISE TEÓRICA

Na primeira etapa, foi realizada uma revisão da literatura, focada nas boas práticas de refatoração de código propostas por Robert C. Martin em Código Limpo, particularmente nos conceitos relacionados à redução da complexidade ciclomática, à melhora da legibilidade e à manutenção do código. A partir dessa base teórica, foram selecionados trechos de código do livro que exemplificam funções com alta complexidade, onde a refatoração poderia trazer melhorias significativas. A análise teórica busca estabelecer os parâmetros para as refatorações, bem como definir as métricas de avaliação da qualidade do código.

### 5.2. ESTUDO DE CASO

A segunda fase consiste na aplicação prática dos conceitos de refatoração abordados na teoria. A partir de trechos de código extraídos do livro de Martin, foi realizada uma análise detalhada da complexidade ciclomática e do índice de facilidade de manutenção, antes e após as refatorações. A escolha das funções para refatoração foi orientada pela análise das métricas de complexidade, procurando identificar métodos com alta complexidade e que pudessem se beneficiar das técnicas de simplificação e reestruturação discutidas no livro. Para realizar as refatorações, foram aplicados dois principais conceitos:

- **Pattern Matching:** Uma técnica moderna disponível em várias linguagens de programação, como C# e F#, que substitui estruturas de decisão como *if-else* ou *switch case* com uma sintaxe mais concisa e legível.

- **Early Return:** Um padrão que visa simplificar a lógica do código ao evitar aninhamentos profundos, favorecendo retornos imediatos quando as condições do fluxo de execução não são atendidas.

Após cada refatoração, foram aplicadas as ferramentas de análise de código do Visual Studio para obter as métricas de complexidade ciclomática e o índice de facilidade de manutenção. O índice de facilidade de manutenção foi escolhido como uma métrica adicional, pois reflete a capacidade de manter e evoluir o código ao longo do tempo. Esta ferramenta gera valores quantitativos que indicam a facilidade com que o código pode ser alterado, corrigido ou estendido sem introduzir novos problemas.

### **5.3. ANÁLISE DAS MÉTRICAS**

A avaliação quantitativa foi centrada na comparação das métricas antes e depois das refatorações. A complexidade ciclomática, que mede o número de caminhos independentes em um código, foi utilizada como principal indicador da complexidade estrutural. Já o índice de facilidade de manutenção, fornecido pela ferramenta de métricas, ajudou a medir a qualidade do código em termos de sua legibilidade e facilidade para realizar modificações.

A comparação entre os valores obtidos antes e depois das refatorações permite verificar o impacto das mudanças na clareza e na facilidade de manutenção do código. Além disso, uma análise qualitativa foi realizada para discutir as implicações das mudanças no design do código, considerando o contexto das funções reescritas e a aplicabilidade das boas práticas para diferentes cenários de desenvolvimento.

### **5.4. ESTUDO COMPARATIVO**

Além das métricas quantitativas, um estudo comparativo foi conduzido entre os métodos refatorados e as versões originais, considerando não apenas as mudanças numéricas nas métricas, mas também a legibilidade, clareza e modularidade do código. Esse estudo qualitativo buscou observar se as refatorações melhoraram de fato a qualidade do código em termos de sua estrutura e organização, ou se algum trade-off em termos de legibilidade ou manutenibilidade foi identificado. Este passo é essencial, pois nem sempre uma redução na



complexidade ciclomática resulta em um código mais fácil de manter, e uma melhoria no índice de manutenção pode depender do contexto de uso da função.

## 5.5. CONSIDERAÇÕES FINAIS

Ao final, o objetivo é analisar se as boas práticas de desenvolvimento, especialmente as propostas por Robert C. Martin, têm impacto positivo não apenas nas métricas quantitativas, mas também na qualidade geral do código em um contexto real de desenvolvimento. A metodologia aplicada combina tanto a teoria consolidada sobre práticas de refatoração de código com um estudo de caso prático, gerando resultados que podem ser aplicados diretamente em cenários reais de desenvolvimento de software. Dessa forma, a pesquisa oferece uma contribuição significativa para a compreensão do impacto das boas práticas na manutenção e evolução de sistemas de software complexos.

## 6. RESULTADOS

### 6.1 MÉTODO *createPluralDependentMessageParts*

Dentro do livro Código Limpo, temos a função *createPluralDependentMessageParts* que após refatorada por Robert Martin, fica da seguinte forma:

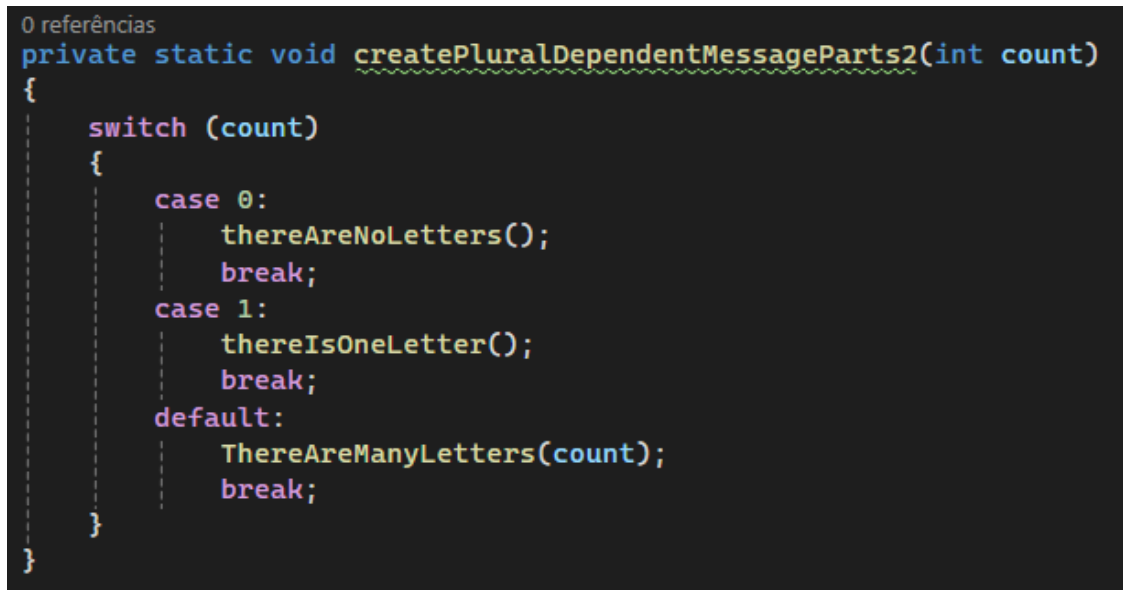
```
private static void createPluralDependentMessageParts(int count)
{
    if (count == 0)
    {
        thereAreNoLetters();
    }
    else if (count == 1)
    {
        thereIsOneLetter();
    }
    else
    {
        ThereAreManyLetters(count);
    }
}
```

Figura 01: Método *createPluralDependentMessageParts*. Fonte: Elaborado pelo autor.

Ao utilizarmos a análise do Visual Studio, temos o seguinte resultado:

- Índice de facilidade de manutenção: 75;
- Complexidade Ciclomática: 3.

Vamos aplicar a ideia de pattern matching através da estrutura de código “switch case”, presente em diversas linguagens como C#, Java e C. Obteremos o seguinte resultado:



```
0 referências
private static void createPluralDependentMessageParts2(int count)
{
    switch (count)
    {
        case 0:
            thereAreNoLetters();
            break;
        case 1:
            thereIsOneLetter();
            break;
        default:
            ThereAreManyLetters(count);
            break;
    }
}
```

Figura 02: Método createPluralDependentMessageParts atualizado. Fonte: Elaborado pelo autor

Após rodarmos a análise, obtivemos as seguintes métricas:

- Índice de facilidade de manutenção: 90;
- Complexidade ciclomática: 4.

Podemos perceber que ao realizarmos essa alteração, temos um aumento de 1 ponto na complexidade ciclomática da nossa função, porém em contrapartida aumentamos o nosso índice de facilidade de manutenção em cerca de 35%. Isso significa que ao utilizarmos estruturas de *pattern matching* para substituir o aninhamento de *else ifs*, podemos aumentar a manutenção do nosso código, gerando um retorno positivo para nossa solução.

## 6.2 MÉTODO *getNextOrNull*

Ainda no livro Clean Code de Robert Martin, apêndice A na página 332, temos a função *getNextOrNull*, que obtém o próximo valor da sequência ou retorna *null* caso ele exceda 100000. A função é apresentada da seguinte forma:

```
0 referências
public static int? getNextOrNull()
{
    if (nextValue < 100000)
        return nextValue++;
    else
        return null;
}
```

Figura 03: Método *getNextOrNull*. Fonte: Elaborado pelo autor.

Ao executarmos a análise de métricas de código, obtemos os seguintes valores:

- Índice de facilidade de manutenção: 91;
- Complexidade ciclomática: 2

Agora, iremos remover a estrutura *e/se* do código, utilizando a ideia de *early return*, onde no nosso contexto, condiz com a ideia de que caso ele não cumpra a condição do próximo valor ser menor que 100000, ele não irá entrar no escopo da estrutura *if* e, portanto, podemos retornar *null*, descartando a necessidade do *e/se* em nosso código. Aplicando as alterações necessárias em nosso código, temos o seguinte resultado:

```
0 referências
public static int? getNextOrNull2()
{
    if (nextValue < 100000)
        return nextValue++;
    return null;
}
```

Figura 04: Método *getNextOrNull* atualizado. Fonte: Elaborado pelo autor.

Ao usarmos a análise do Visual Studio, vamos obter as seguintes métricas dele:

- Índice de Facilidade de Manutenção: 84;
- Complexidade ciclomática: 2.

Podemos perceber que apesar de removermos uma estrutura de código e reduzirmos a quantidade de linhas, seu índice de facilidade de manutenção está menor do que quando comparado a função que utiliza o *else*. Um dos motivos que corroboram para esta conclusão é de que a ideia de *early return* cria diferentes pontos de saída de uma função ou método, o que dificulta o entendimento da função e por consequência irá diminuir a sua manutenibilidade.

### 6.3 MÉTODO *isInRange*

Ainda no livro de Martin, no apêndice B página 388, temos o método *isInRange* que irá validar se as duas datas passadas por parâmetro estão dentro do escopo permitido. em uma conversão livre de Java para C#, ele fica da seguinte forma:

```
0 referências
public static bool isInRange(DateTime d1, DateTime d2, int include)
{
    DateTime start = new DateTime(Math.Min(d1.Ticks, d2.Ticks));
    DateTime end = new DateTime(Math.Max(d1.Ticks, d2.Ticks));
    DateTime s = DateTime.Now;

    if(include == (int)SerialDate.INCLUDE_BOTH)
    {
        return (s >= start && s <= end);
    }
    else if(include == (int)SerialDate.INCLUDE_FIRST)
    {
        return (s >= start && s < end);
    }
    else if(include == (int)SerialDate.INCLUDE_SECOND)
    {
        return (s > start && s <= end);
    }
    else
    {
        return (s > start && s < end);
    }
}
```

Figura 05: Método `isInRange`. Fonte: Elaborado pelo autor.

Quando usamos a ferramenta de métricas e análises do Visual Studio, podemos obter os seguintes valores:

- Índice de Facilidade de Manutenção: 59;
- Complexidade Ciclomática: 8

Nesse método, possuímos uma estrutura grande de *if else* encadeados, temos uma complexidade ciclomática e um índice de facilidade de manutenção um pouco mais alto que os métodos anteriores. Porém, esse método ainda segue as ideias propostas por Martin.

Agora, vamos aplicar a ideia de pattern matching novamente, em um cenário onde temos mais pontos de saída. Aplicando a estrutura de *pattern matching* disponível na linguagem C#, podemos obter o seguinte resultado:

```
0 referências
public static bool isInRange2(DateTime d1, DateTime d2, int include)
{
    DateTime start = new DateTime(Math.Min(d1.Ticks, d2.Ticks));
    DateTime end = new DateTime(Math.Max(d1.Ticks, d2.Ticks));
    DateTime s = DateTime.Now;

    return include switch
    {
        (int)SerialDate.INCLUDE_BOTH => (s >= start && s <= end),
        (int)SerialDate.INCLUDE_FIRST => (s >= start && s < end),
        (int)SerialDate.INCLUDE_SECOND => (s > start && s <= end),
        _ => (s > start && s < end)
    };
}
```

Figura 06: Método `isInRange` atualizado. Fonte: Elaborado pelo autor.

Além de uma grande redução no número de linhas dentro do nosso método, podemos perceber que ele ficou mais conciso, isso é provado pelas seguintes métricas disponibilizadas:

- Índice de Facilidade de Manutenção: 69;
- Complexidade ciclomática: 5

A partir dessas informações, é possível extrair que o uso da estrutura de pattern matching, além de tornar seu código menor, pode melhorar a qualidade do

mesmo, pois vai diminuir a complexidade dele e aumentar o seu índice de facilidade de manutenção.

## 7. Conclusão

A partir da análise e aplicação das técnicas de refatoração discutidas no presente trabalho, foi possível observar como diferentes abordagens de reestruturação de código podem impactar as métricas de qualidade, como a complexidade ciclomática e o índice de facilidade de manutenção. No caso dos métodos analisados, tanto a adoção do pattern matching quanto a aplicação do conceito de early return proporcionaram variações significativas nos resultados das métricas, refletindo diretamente na qualidade e legibilidade do código.

No primeiro caso, ao reescrever o método `createPluralDependentMessageParts` utilizando o pattern matching, notamos um aumento de 35% no índice de facilidade de manutenção, mesmo com um pequeno acréscimo na complexidade ciclomática. Esse aumento no índice de manutenção reflete a clareza proporcionada pela nova estrutura de código, que eliminou a necessidade de aninhamentos de `else ifs`, resultando em um código mais legível e mais fácil de ser modificado no futuro. A análise confirma que, ao optar por abordagens que utilizam pattern matching, podemos melhorar a manutenção do código sem comprometer substancialmente a complexidade.

Por outro lado, ao analisar o método `getNextOrNull`, a aplicação do conceito de early return revelou que, apesar da remoção do `else` e a simplificação do fluxo de execução, houve uma leve queda no índice de facilidade de manutenção. Essa diminuição pode ser atribuída ao aumento no número de pontos de saída do método, o que, em alguns contextos, pode gerar maior dificuldade de entendimento, especialmente em funções mais complexas. Contudo, é importante destacar que essa mudança ainda representa uma melhoria em termos de redução de linhas e simplificação do código, refletindo uma trade-off entre clareza e simplicidade.

Por fim, no método `isInRange`, a utilização do pattern matching também se mostrou eficaz ao reduzir o número de linhas de código e melhorar sua concisão. Apesar de uma leve elevação na complexidade ciclomática, o método se tornou mais fácil de entender e manter, o que é evidenciado pela melhora no índice de

facilidade de manutenção. A refatoração com pattern matching demonstrou que, em cenários com múltiplos pontos de decisão, essa abordagem não apenas diminui o tamanho do código, mas também melhora a legibilidade e a clareza geral, contribuindo positivamente para a manutenção futura. É possível perceber que as técnicas aplicadas vão de encontro com a literatura e conseguem ser impactantes quando referente as métricas definidas dentro dos trechos de código selecionados.

Para estudos futuros, pode-se considerar a realização de análises adicionais focadas em distintas métricas de avaliação de trechos de código. Entre essas métricas, destacam-se o número de linhas de código, o grau de acoplamento entre classes, a profundidade das hierarquias de herança e o tempo de execução, dentre outras possibilidades.

Como contribuição, este trabalho evidenciou a importância de técnicas de refatoração como o uso de pattern matching e early return na melhoria da qualidade do código, principalmente no que tange à facilidade de manutenção e legibilidade. Embora cada abordagem tenha suas particularidades e implicações na complexidade, o estudo de caso apresentado demonstra que, ao adotar boas práticas de refatoração, é possível alcançar um equilíbrio entre clareza e desempenho, resultando em soluções mais robustas e sustentáveis no longo prazo. Assim, a análise e aplicação cuidadosa dessas técnicas são fundamentais para a construção de sistemas mais fáceis de manter e evoluir.

## REFERÊNCIAS BIBLIOGRÁFICAS

SOMMERVILLE, IAN. Engenharia de Software. 9. ed. São Paulo: Pearson, 2011

MARTIN, ROBERT C. Código Limpo

MCCABE, Thomas J. "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric". 1996

MCCABE, T. J. A Complexity Measure. *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, 1976, pp. 308-320.

BASIL, V. R., et al. "The Experience Factory." *Software Engineering, IEEE Transactions on*, vol. 22, no. 6, 1996, pp. 430-442.

BOOCH, G. *Object-Oriented Analysis and Design with Applications*. 2. ed. Addison-Wesley, 1994.

BERGE, C. *Graphs and Hypergraphs*. 2. ed. North-Holland, 1973.

ALMEIDA, Luciana Thomaz; MIRANDA, João Machini. "Código Limpo e seu Mapeamento para Métricas de Código Fonte". 2010.