Universidade Federal do Maranhão
Cadeira: Qualidade De Software
Professor: Samyr Beliche Vale
Discente:Thiago Augusto Pereira Amaral

Trabalho de Qualidade de Software: Métricas e Refatoração

[Link do projeto no github]

1.0 Objetivo do trabalho

Para a realização deste trabalho, será aplicado os conceitos discutidos em sala de aula para analisar o código desenvolvido na última tarefa(link para o trabalho). Utilizaremos métricas de software para avaliar a qualidade do código da aplicação, identificando áreas que necessitam de melhorias. A partir dessa análise, procederemos com a refatoração do código com o objetivo de aprimorar essas métricas e, consequentemente, a qualidade geral do software.

2.0 Métricas de software

Métricas de software são indicadores numéricos que representam diversas características do código-fonte, processos de desenvolvimento e produtos de software. Elas podem ser classificadas em métricas de produto (linhas de código, complexidade ciclomática), métricas de processo (tempo de desenvolvimento, taxas de defeitos) e métricas de projeto (esforço estimado, custo).

Essas métricas ajudam a avaliar a qualidade do código, identificando áreas que precisam de melhorias e facilitando a detecção de partes difíceis de manter ou com alta complexidade, orientando o processo de refatoração. Elas fornecem dados concretos para estimar esforços, custos e prazos, auxiliando na gestão e planejamento de projetos. Além disso, permitem monitorar o progresso e a eficácia das práticas de desenvolvimento, promovendo a melhoria contínua dos processos e produtos. As métricas também oferecem uma base objetiva para decisões técnicas e gerenciais, minimizando a subjetividade e aumentando a eficiência.

Neste trabalho, analisaremos partes do código utilizando três métricas: **Complexidade Ciclomática de McCabe, Linhas de Código e Profundidade de Aninhamento Condicional.**

2.1 Complexidade Ciclomática de McCabe (CC)

É uma medida que avalia a complexidade da estrutura de controle de um software. Ela indica o número de caminhos linearmente independentes dentro do código, o que corresponde ao número mínimo de caminhos de execução necessários para testar todas as possibilidades lógicas do programa. Em outras palavras, a CC quantifica a complexidade do fluxo de controle,

ajudando a identificar partes do código que podem ser difíceis de entender, testar e manter.

2.2 Linhas de Código

A métrica de Linhas de Código (LOC) contabiliza o número total de linhas de um programa, abrangendo todas as linhas de código, declarações e instruções. Dependendo do contexto, essa contagem pode ou não incluir comentários e linhas em branco. Apesar de sua simplicidade, LOC é uma métrica poderosa para avaliar a complexidade de entidades de software. Ela oferece uma visão clara do tamanho do código, o que pode ajudar a identificar a complexidade e o esforço necessário para desenvolvimento e manutenção.

2.3 Profundidade de Aninhamento Condicional

Essa métrica mede o nível de aninhamento das estruturas condicionais dentro de um bloco de código. Ela conta quantos níveis de profundidade as instruções condicionais, como if, else if, e else, possuem dentro de um método ou função. Cada vez que uma nova estrutura condicional é aninhada dentro de outra, a profundidade aumenta. Por exemplo, se uma função possui um if dentro de outro if, a profundidade de aninhamento é 2.
Ela ajuda a avaliar a complexidade do código; um alto nível de aninhamento pode indicar um código complexo e difícil de entender, o que aumenta a chance de erros e dificulta a manutenção

3.0 Apresentação dos métodos e a análise segundo as métricas

Nesta etapa do trabalho, serão selecionadas duas funções da aplicação e analisaremos elas de acordo com as métricas apresentadas.Para fazer isso, utilizaremos a ferramenta MetricsTree no Intellij para analisar o código.

3.1 adicionarNotaAoAluno()

O método adicionarNotaAoAluno() em Java tem como objetivo adicionar uma nova nota a um aluno específico, seguindo as regras de média e quantidade máxima de notas.

```java
public void adicionarNotaAoAluno(int idAlund, float nota) {  no usages
    List<AlunoModel> alunos = jsonManager.carregarDadosAlunos();
    if (alunos != null) {
        for (AlunoModel aluno : alunos) {
            if (aluno.getIdAluno() == idAlund) {
                if (aluno.getListaNotas() != null && aluno.getListaNotas().size() >= aluno.getNUMAXNOTAS()) {
                    float mediaAtual = aluno.calcularMedia();
                    if (mediaAtual < 7) {
                        float menorNota = Collections.min(aluno.getListaNotas());
                        if (nota > menorNota) {
                            aluno.getListaNotas().remove(menorNota);
                            aluno.getListaNotas().add(nota);
                            aluno.setMedia(aluno.calcularMedia());
                            jsonManager.salvarDadosAlunos(alunos);
                        } else {
                            System.out.println("Nota não adicionada.");
                        }
                    } else {
                        System.out.println("O aluno já possui três notas e sua média é maior que 7.0. Não é possível adicionar mais notas.");
                        return;
                    }
                } else {
                    aluno.adicionarNota(nota);
                    if (aluno.getListaNotas().size() == aluno.getNUMAXNOTAS()) {
                        float novaMedia = aluno.calcularMedia();
                        System.out.println("Nova média calculada: " + novaMedia);
                    } else {
                        System.out.println("Nota adicionada com sucesso.");
                    }
                    jsonManager.salvarDadosAlunos(alunos);
                    return;
                }
            }
        }
    }
}
```

Figura 1 - Método adicionarNotaAoAluno

3.1.1 Complexidade Ciclomática

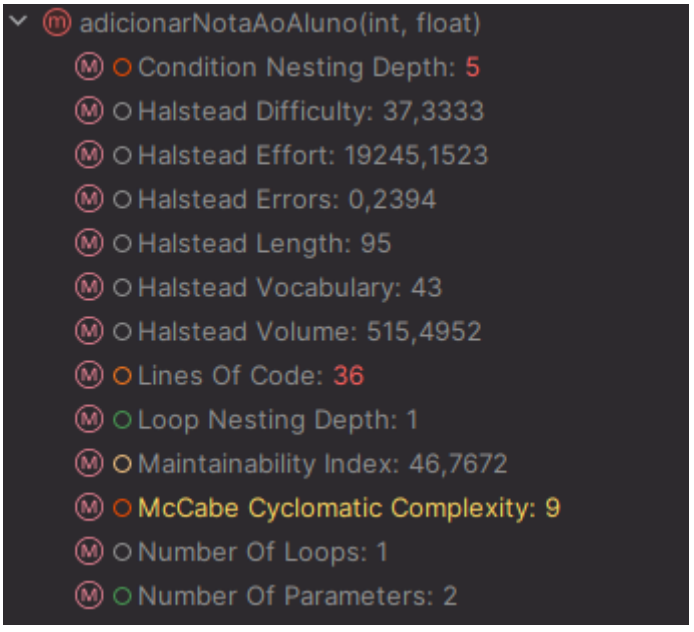

Figura 2 - Métricas para adicionarNotaAoAluno

A análise ciclomática resultou em um valor de 9. Isso significa que existem 9 caminhos de execução distintos através do método.Isso implica que:

- **Manutenção**: Um valor de 9 indica que o método é relativamente complexo e pode ser mais difícil de entender, testar e manter. Métodos com alta complexidade ciclomática são mais propensos a conter erros e podem necessitar de refatoração.

- **Testes**: Para garantir que todas as possíveis execuções do método sejam testadas, seriam necessários pelo menos 9 casos de teste diferentes, o que seria trabalhoso.

- **Legibilidade:** A alta complexidade pode afetar a legibilidade do código, tornando-o mais desafiador para novos desenvolvedores ou revisores entenderem rapidamente o fluxo lógico.

3.1.2 Linhas de código

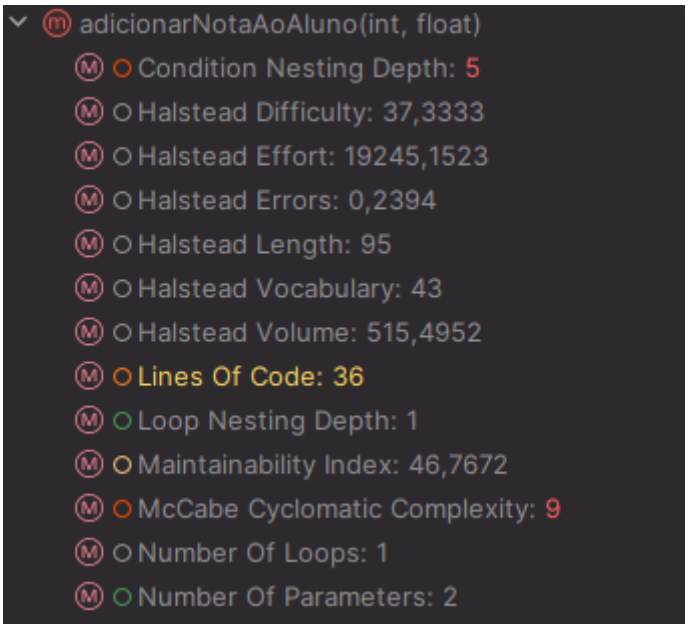

Figura 3 - Métricas para adicionarNotaAoAluno

Foi informado que a métrica Lines of Code(Linhas de código) é igual a 36, o que significa que o método contém 36 linhas de código.Isso implica que:

- **Tamanho e Complexidade**:É relativamente longo.Um método com tantas linhas pode ser considerado grande e, possivelmente, mais complexo.
- **Manutenibilidade**: Qualquer mudança ou correção pode exigir uma compreensão detalhada de todas as partes do método.

- **Legibilidade**: Métodos mais longos podem ser menos legíveis.Desenvolvedores que revisarem o código podem achar difícil seguir o fluxo lógico.

3.1.3 Profundidade de Aninhamento Condicional

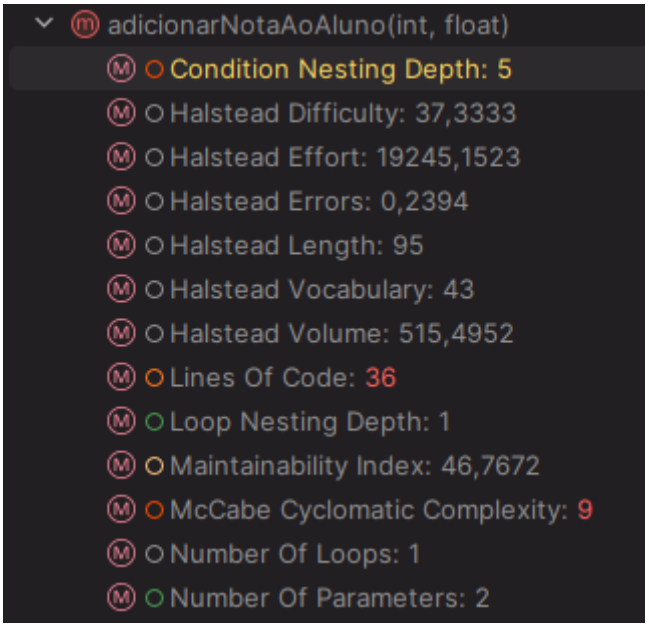

Figura 4 - Métricas para adicionarNotaAoAluno

No caso do código fornecido, a profundidade de aninhamento é 5, o que significa que há cinco níveis de estruturas condicionais aninhadas.Isso implica que:

- **Compreensão**: Quanto maior a profundidade de aninhamento, mais difícil é para um desenvolvedor entender rapidamente o fluxo lógico do código
- **Manutenção**: Alterações ou correções podem se tornar mais complicadas devido à necessidade de garantir que todas as ramificações condicionais e caminhos lógicos ainda funcionem corretamente após a modificação.
- **Bugs**: Códigos com muitos níveis de aninhamento são mais propensos a conter erros, especialmente erros relacionados a condições mal definidas ou esquecidas.

3.2 ExcluirAluno()

Esse método recebe um identificador de aluno (idAluno) como parâmetro e retorna um valor inteiro indicando se o aluno foi removido com sucesso.

```java
public int excluirAluno(int idAluno) {  no usages
    int removed = 0;
    List<AlunoModel> alunos = jsonManager.carregarDadosAlunos();
    if (alunos != null) {
        if(alunos.removeIf(aluno -> aluno.getIdAluno() == idAluno)){
            removed = 1;
        }
        jsonManager.salvarDadosAlunos(alunos);

        List<TurmaModel> turmas = jsonManager.carregarDadosTurmas();
        if (turmas != null) {
            for (TurmaModel turma : turmas) {
                turma.removerAluno(idAluno);
            }
            jsonManager.salvarDadosTurmas(turmas);
        }
    }
    return removed;
}
```

Figura 5 - Código do método ExcluirAluno()

3.2.1 Complexidade Ciclomática

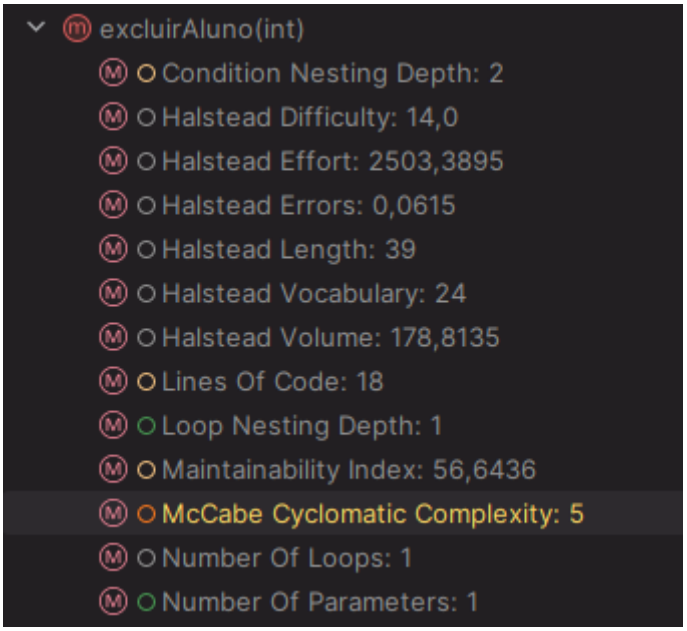

Figura 6 - Métricas para o método ExcluirAluno()

A análise ciclomática resultou em um valor de 9. Isso significa que existem 5 caminhos de execução distintos através do método.Implica na mesma situação abordada anteriormente, apesar de ser menos complexo.

3.2.2 Linhas de código

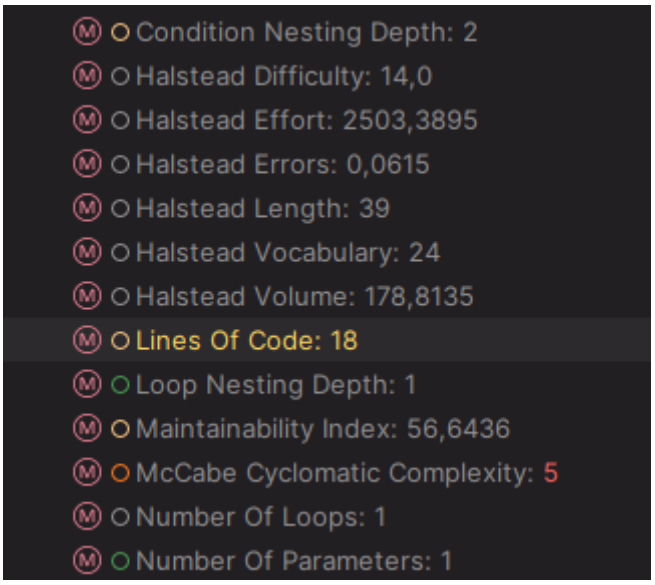

Figura 7 - Métricas para o método ExcluirAluno()

O método tem 18 linhas de código, o que apresenta uma quantidade moderada. Embora não seja excessivamente longo, ele engloba várias responsabilidades que poderiam ser divididas em métodos menores para melhorar a legibilidade, manutenibilidade, reutilização de código e testabilidade.

3.2.3 Profundidade de Aninhamento Condicional

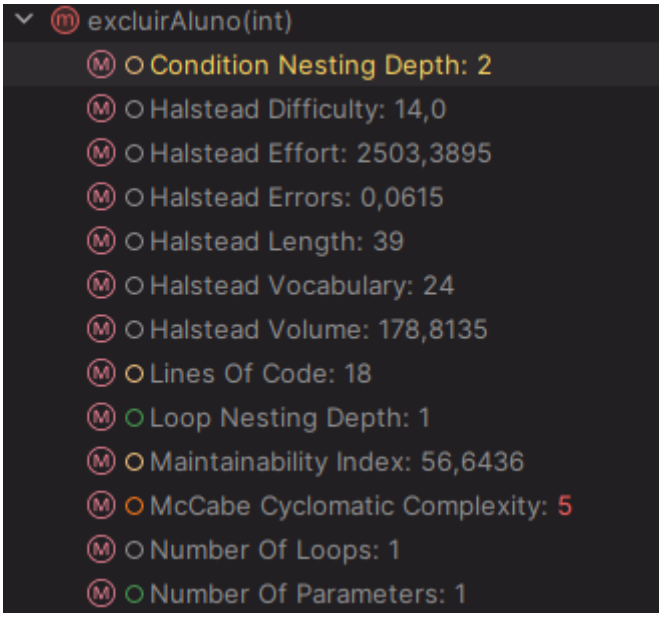

Figura 8 - Métricas para o método ExcluirAluno()

Isso significa que o código contém estruturas condicionais (como if e for) que são aninhadas em até dois níveis. Esse nível de profundidade indica uma complexidade mais simples, onde o código mantém uma estrutura relativamente fácil de entender.

4.0 Refatoração

Agora que apresentamos os métodos-alvo e analisamos o que as métricas indicam sobre eles, vamos refatorar o código com o objetivo de aprimorar essa análise e, consequentemente, melhorar o código como um todo.

4.1 AdicionaNotaAoAluno()(Refatoração)

```
59      public void adicionarNotaAoAluno(int idAluno, float nota) {  11 usages
60          List<AlunoModel> alunos = jsonManager.carregarDadosAlunos();
61          if (alunos == null) {
62              return;
63          }
64
65          for (AlunoModel aluno : alunos) {
66              if (aluno.getIdAluno() == idAluno) {
67                  processarNotaAluno(aluno, nota, alunos);
68                  return;
69              }
70          }
71      }
```

Figura 9 - Código AdicionarNotaAoAluno

Essa refatoração melhora a estrutura do código ao quebrar o método original em métodos menores e mais focados. Isso resulta em uma menor complexidade ciclomática, menos linhas de código e menor profundidade de aninhamento condicional.

4.2 ExcluirAluno()(Refatoração)

```
26      public int excluirAluno(int idAluno) {  no usages
27          int removed = 0;
28          List<AlunoModel> alunos = jsonManager.carregarDadosAlunos();
29
30          if (alunos != null && removerAlunoDaLista(alunos, idAluno)) {
31              removed = 1;
32              jsonManager.salvarDadosAlunos(alunos);
33              atualizarTurmas(idAluno);
34          }
35
36          return removed;
37      }
```

Figura 10 - Código ExcluirAluno()

Nesta refatoração, a função foi simplificada ao extrair a lógica de remoção de aluno e atualização das turmas em métodos separados.Isso reduz a complexidade ciclomática, a profundidade de aninhamento condicional e as linhas de código dentro do método principal, mantendo os nomes das variáveis e métodos consistentes com o restante do código.

5.0 Novas Métricas

Após a refatoração das nossas funções, é hora de revelar como essa mudança impactou os resultados obtidos a partir da análise das métricas.

5.1 AdicionaNotaAoAluno()(Métricas novas)

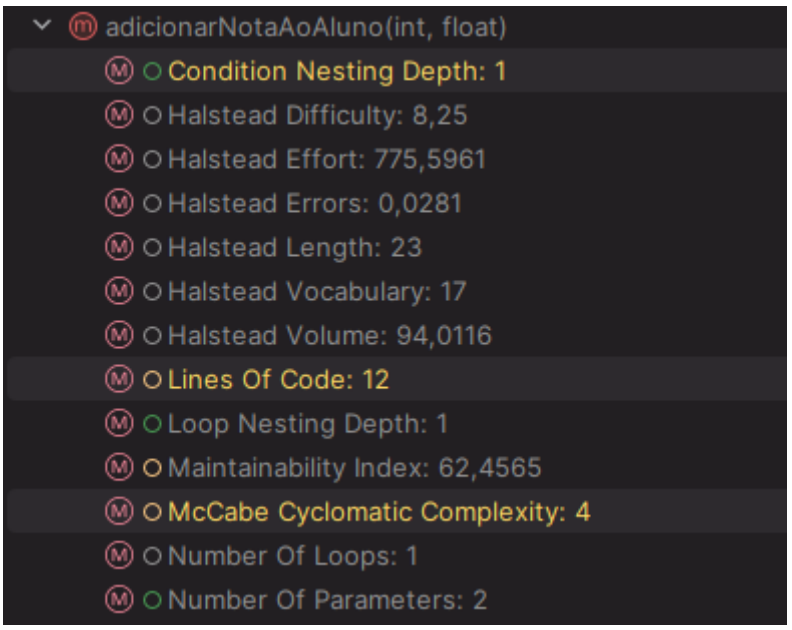

Figura 11 - Métricas adicionarNotaAoAluno()

5.2 ExcluirAluno()(Métricas novas)

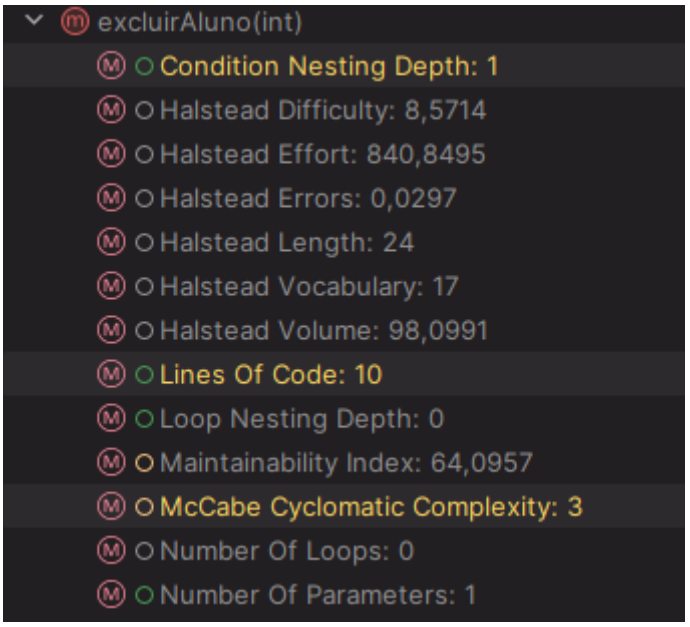

Figura 12 - Métricas ExcluirAluno()

Em ambos os casos, podemos perceber que as métricas diminuíram, consequentemente melhorando diversos aspectos do nosso código. A refatoração proporcionou uma estrutura de código mais limpa e organizada. Funções bem definidas e modularizadas facilitaram a leitura e compreensão

do código, tornando mais fácil para os desenvolvedores realizar manutenções e atualizações futuras.

6.0 Conclusão

Neste trabalho, aplicamos métricas de software para avaliar e refatorar duas funções de uma aplicação, focando em melhorar a qualidade do código. Utilizamos as métricas de Complexidade Ciclomática de McCabe, Linhas de Código e Profundidade de Aninhamento Condicional para identificar áreas críticas e, em seguida, realizamos refatorações visando simplificar e melhorar a manutenibilidade do código.

A melhoria das métricas após a refatoração evidenciou a eficácia das técnicas aplicadas, resultando em um código mais organizado e fácil de entender. Este exercício reforça a importância das métricas de software como ferramentas essenciais para a gestão da qualidade de código, fornecendo dados concretos que orientam o processo de desenvolvimento e refatoração.