Pontifícia Universidade Católica do Paraná

Curso de Sistemas de Informação

Emilly E .Q Da Fonseca

MELHORIAS NO SISTEMA DE TABELA HASH

Trabalho apresentado à disciplina de Padrões de Projeto e Multicamadas, como requisito parcial para a obtenção de nota no 6º período.

Orientador(a): Prof. Vinicius Godoy De Mendonça

Neste documento, apresentarei 5 melhorias que podem ser implementadas no projeto do Sistema de Tabela Hash em um trabalho que realizei no quarto período. Cada uma dessas melhorias será detalhada, explicando sua importância e como aplicá-las utilizando diferentes padrões de projeto.

1. Função de Hash Centralizada

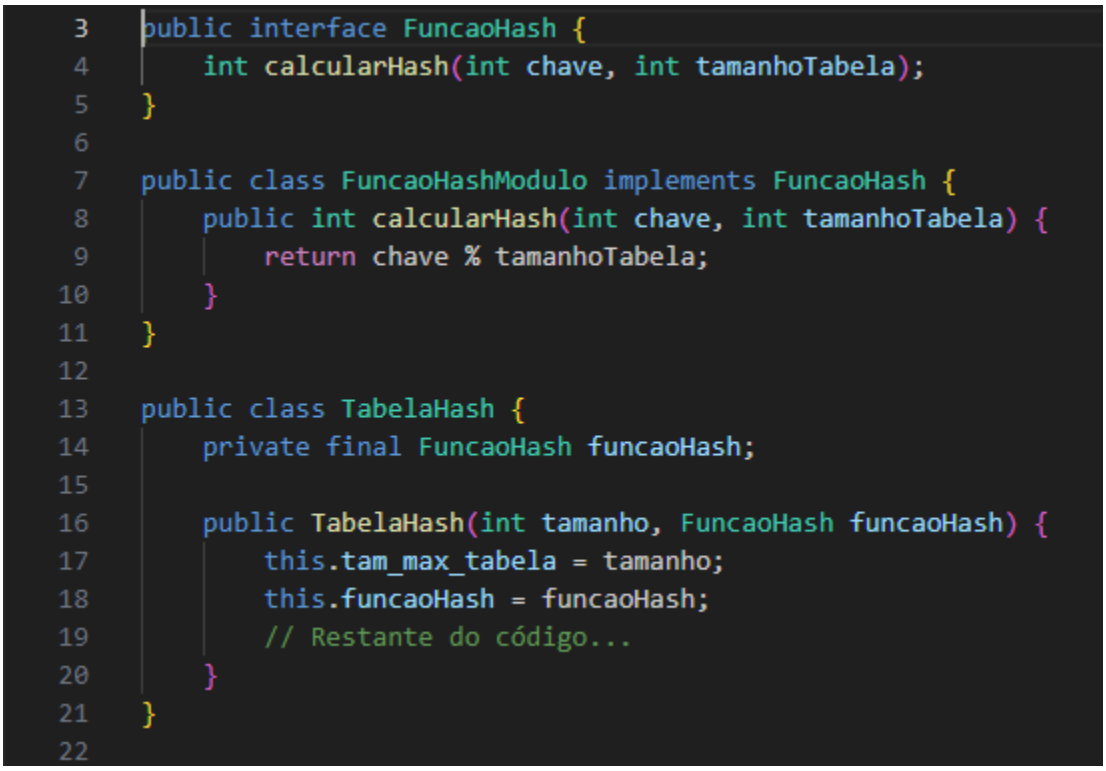
Problema:

Atualmente, a função de hash (FuncaoHash) está sendo repetida em várias partes do código, o que pode causar erros e dificulta a manutenção.

Solução:

Utilizar o padrão de projeto Strategy para criar uma interface que centralize a função de hash. Isso facilita a troca ou modificação da lógica de hash sem impactar o restante do sistema.

O código ficaria dessa forma:



```java
public interface FuncaoHash {
    int calcularHash(int chave, int tamanhoTabela);
}

public class FuncaoHashModulo implements FuncaoHash {
    public int calcularHash(int chave, int tamanhoTabela) {
        return chave % tamanhoTabela;
    }
}

public class TabelaHash {
    private final FuncaoHash funcaoHash;

    public TabelaHash(int tamanho, FuncaoHash funcaoHash) {
        this.tam_max_tabela = tamanho;
        this.funcaoHash = funcaoHash;
        // Restante do código...
    }
}
```

2. Redimensionamento Eficiente

Problema:

O redimensionamento manual está duplicando código e complicando a manutenção.

Solução:

Aplicar o padrão de projeto Template Method para estruturar o processo de redimensionamento. Assim, partes comuns do processo podem ser reutilizadas, simplificando a lógica.

O código ficaria dessa forma:

```java
public abstract class TabelaHash {
    protected int tam_max_tabela;
    protected Aluno[] estrutura;

    public void redimensionar() {
        int novoTamanho = calcularNovoTamanho();
        Aluno[] novaEstrutura = new Aluno[novoTamanho];
        // Copiar os dados para a nova estrutura...
        estrutura = novaEstrutura;
        tam_max_tabela = novoTamanho;
    }

    protected abstract int calcularNovoTamanho();
}
```

3. Tratamento de Colisões

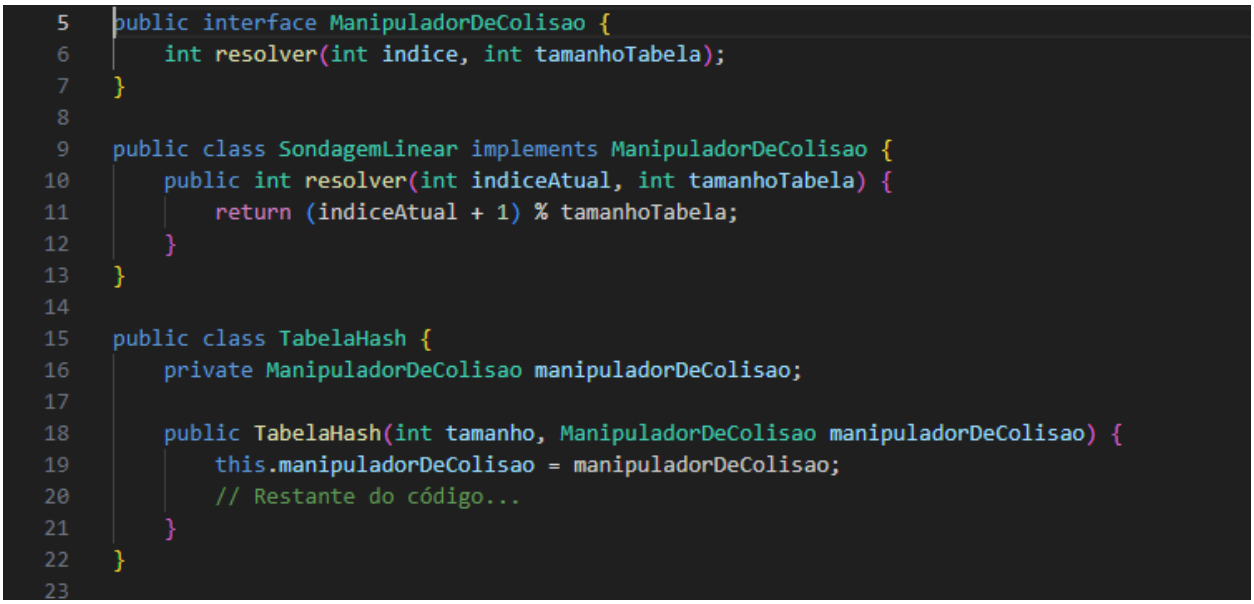
Problema:

A lógica de tratamento de colisões está espalhada por vários métodos, dificultando a leitura e manutenção.

Solução:

Utilizar o padrão Chain of Responsibility para gerenciar diferentes estratégias de resolução de colisões de forma modular e flexível.

O código ficaria dessa forma:



```
5   public interface ManipuladorDeColisao {
6       int resolver(int indice, int tamanhoTabela);
7   }
8
9   public class SondagemLinear implements ManipuladorDeColisao {
10      public int resolver(int indiceAtual, int tamanhoTabela) {
11          return (indiceAtual + 1) % tamanhoTabela;
12      }
13  }
14
15  public class TabelaHash {
16      private ManipuladorDeColisao manipuladorDeColisao;
17
18      public TabelaHash(int tamanho, ManipuladorDeColisao manipuladorDeColisao) {
19          this.manipuladorDeColisao = manipuladorDeColisao;
20          // Restante do código...
21      }
22  }
23
```

4. Impressão Mais Organizada

Problema:

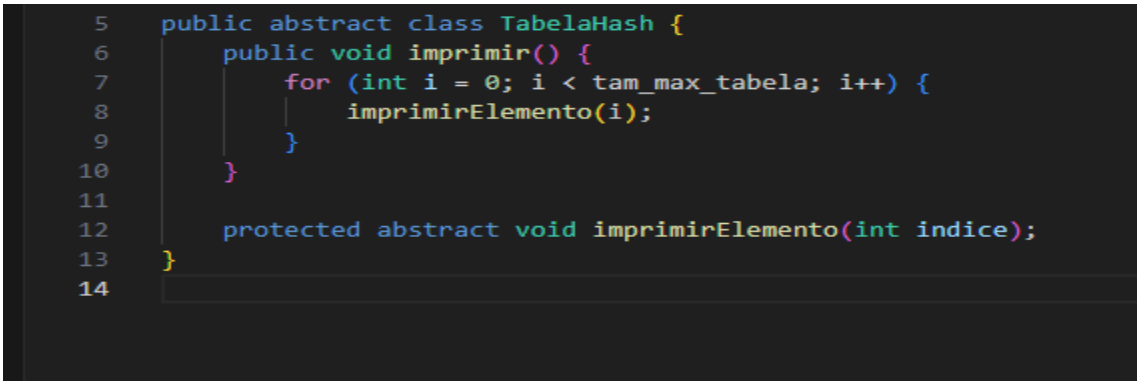
A lógica de impressão de dados está sendo duplicada em várias partes do projeto.

Solução:

Aplicar o padrão Template Method para definir um método de impressão reutilizável e especializável conforme necessário.

O código ficaria dessa forma:

Para a classe abstrata:



```
5   public abstract class TabelaHash {
6       public void imprimir() {
7           for (int i = 0; i < tam_max_tabela; i++) {
8               imprimirElemento(i);
9           }
10      }
11
12      protected abstract void imprimirElemento(int indice);
13  }
14
```

5. Menos Dependências Diretas

Problema:

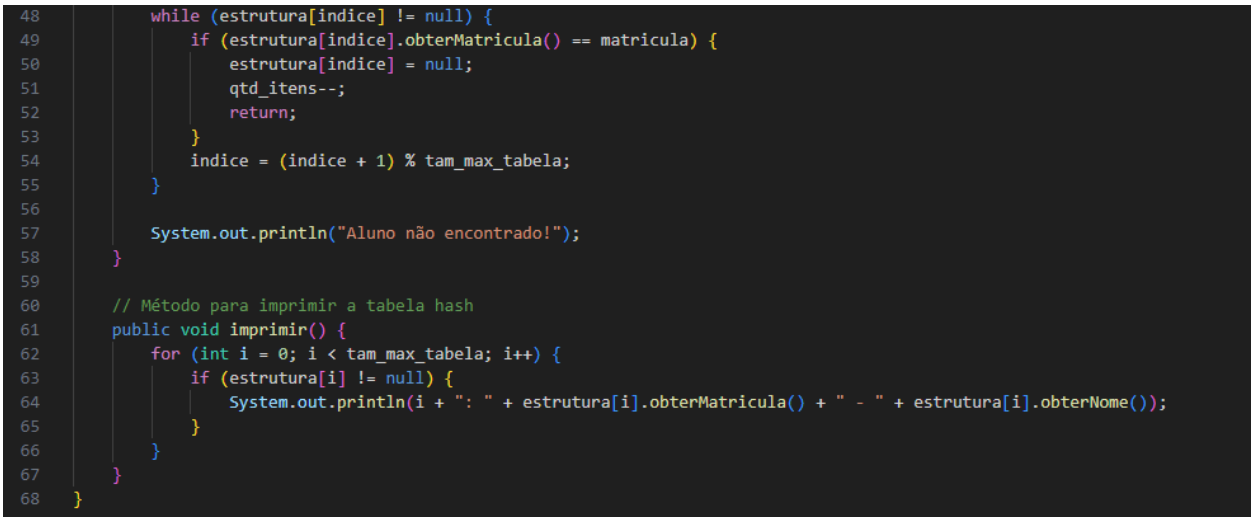
O código está fortemente acoplado, o que significa que alterações em uma parte podem impactar outras partes inesperadamente.

Solução:

Aplicar o padrão Dependency Injection para tornar o código menos acoplado e mais testável.

Como ficaria o código:

```java
public class TabelaHash {
    private FuncaoHash funcaoHash;
    private Aluno[] estrutura;
    private int tam_max_tabela;
    private int qtd_itens;

    public TabelaHash(int tamanho, FuncaoHash funcaoHash) {
        this.funcaoHash = funcaoHash;
        this.tam_max_tabela = tamanho;
        this.qtd_itens = 0;
        this.estrutura = new Aluno[tamanho];
    }

    // Método para inserir um aluno
    public void inserir(Aluno aluno) {
        int indice = funcaoHash.calcularHash(aluno.obterMatricula(), tam_max_tabela);

        while (estrutura[indice] != null) {
            indice = (indice + 1) % tam_max_tabela; // Tratamento de colisão por sondagem
        }

        estrutura[indice] = aluno;
        qtd_itens++;
    }

    // Método para buscar um aluno
    public Aluno buscar(int matricula) {
        int indice = funcaoHash.calcularHash(matricula, tam_max_tabela);

        while (estrutura[indice] != null) {
            if (estrutura[indice].obterMatricula() == matricula) {
                return estrutura[indice];
            }
            indice = (indice + 1) % tam_max_tabela;
        }

        return null; // Aluno não encontrado
    }

    // Método para remover um aluno
    public void remover(int matricula) {
        int indice = funcaoHash.calcularHash(matricula, tam_max_tabela);

        while (estrutura[indice] != null) {
            if (estrutura[indice].obterMatricula() == matricula) {
                estrutura[indice] = null;
                qtd_itens--;
```



```
48          while (estrutura[indice] != null) {
49              if (estrutura[indice].obterMatricula() == matricula) {
50                  estrutura[indice] = null;
51                  qtd_itens--;
52                  return;
53              }
54              indice = (indice + 1) % tam_max_tabela;
55          }
56
57          System.out.println("Aluno não encontrado!");
58      }
59
60      // Método para imprimir a tabela hash
61      public void imprimir() {
62          for (int i = 0; i < tam_max_tabela; i++) {
63              if (estrutura[i] != null) {
64                  System.out.println(i + ": " + estrutura[i].obterMatricula() + " - " + estrutura[i].obterNome());
65              }
66          }
67      }
68  }
```

Conclusão

As sugestões de aperfeiçoamento para o sistema de tabela hash têm como objetivo elevar a flexibilidade, modularidade e manutenibilidadedo código, empregando padrões de projeto adequados para cada modificação. A seguir, ressaltamos as vantagens de cada uma das melhorias realizadas:

1. **Separação da Função de Hash (Padrão de Estratégia)**:

- Vantagem: Facilita a troca da lógica de hash, permitindo o uso de várias funções de hash sem alterar a classe `TabelaHash`. Isso torna mais simples futuras adaptações e aumenta a flexibilidade do código.

2. **Módulo para Tratamento de Colisões**

- Vantagem: Ao centralizar a lógica de colisão, o código se torna mais estruturado e compreensível, facilitando também futuras modificações. Essa configuração possibilita a combinação de variados métodos de resolução de colisão, como sondagem linear, quadrática, entre outros, proporcionando mais alternativas para otimizar o desempenho.

3. **Técnica de Impressão para Exibir o Estado da Tabela**:

- Vantagem: Torna mais fácil a depuração e a compreensão do conteúdo da tabela durante os testes, especialmente em cenários de desenvolvimento. A habilidade de observar o estado da tabela em tempo real auxilia na identificação de erros e colisões de forma ágil.

4. **Capacidade de Ajuste Automático**:

- Vantagem: Ao adicionar uma lógica de redimensionamento dinâmico (que será incorporada), o código conseguirá ajustar automaticamente o tamanho da tabela conforme a necessidade, otimizando o uso da memória e melhorando a eficiência. Essa otimização auxilia na manutenção da tabela eficiente, mesmo com grandes volumes de dados.

5. **Injeção de Dependência para Adaptabilidade:**

Vantagem: Ao aplicar injeção de dependência na função de hash e no método de resolução de colisões, o código torna-se mais modular e menos interligado, facilitando testes e manutenção. Essa configuração possibilita que as dependências sejam modificadas ou atualizadas sem a necessidade de alterar a classe principal, favorecendo uma arquitetura mais sólida.

Vantagens Gerais

Essas atualizações fazem o código mais flexível e expansível, além de prepará-lo para alterações futuras com menor esforço. Com uma estrutura mais modular e fundamentada em padrões de projeto, o sistema de tabela hash passa a ser mais simples de manter e aprimorar, proporcionando uma solução que respeita os princípios de boas práticas de desenvolvimento de software. Isso não só melhora a qualidade do código, mas também diminui o tempo de desenvolvimento necessário para futuras modificações.