# Identificação de Melhorias com Padrões de Projeto no Código TabelaHash

Este documento detalha as melhorias aplicáveis ao código da classe TabelaHash, utilizando diferentes padrões de projeto para melhorar a modularidade, organização e eficiência. As melhorias incluem o uso de Factory Method, Strategy, Singleton, Template Method e Observer.

## 1. Criação de uma Fábrica de Tabelas Hash para Diferentes Estratégias de Colisão - Factory Method

Motivo:  
Atualmente, o código mistura o tratamento de colisão por encadeamento e endereçamento aberto na mesma classe. Utilizar o Factory Method permite criar tabelas específicas para cada tipo de tratamento de colisão, facilitando a manutenção e expansão futura.  
  
Aplicação:

// Classe abstrata TabelaHashBase  
public abstract class TabelaHashBase {  
 protected int capacidade;  
 protected int[] chaves;  
 protected Object[] valores;  
  
 public TabelaHashBase(int capacidade) {  
 this.capacidade = capacidade;  
 this.chaves = new int[capacidade];  
 this.valores = new Object[capacidade];  
 }  
  
 public abstract void inserir(int chave, Object valor);  
 public abstract Object buscar(int chave);  
}  
  
// Implementações específicas para cada tipo de colisão  
public class TabelaEncadeamento extends TabelaHashBase {   
 // Implementação do encadeamento  
}  
  
public class TabelaEnderecamentoAberto extends TabelaHashBase {   
 // Implementação do endereçamento aberto  
}  
  
// Fábrica de Tabelas Hash  
public class TabelaHashFactory {  
 public static TabelaHashBase criarTabela(String tipo, int capacidade) {  
 if (tipo.equals("Encadeamento")) {  
 return new TabelaEncadeamento(capacidade);  
 } else if (tipo.equals("EnderecamentoAberto")) {  
 return new TabelaEnderecamentoAberto(capacidade);  
 }  
 return null;  
 }  
}  
  
// Uso da Fábrica  
TabelaHashBase tabela = TabelaHashFactory.criarTabela("Encadeamento", 10);

## 2. Aplicação do Padrão Strategy para Diferentes Estratégias de Hashing

Motivo:  
Alterar a estratégia de hashing, como trocar o método de endereçamento ou função de hash, é facilitado pelo padrão Strategy, que permite trocar algoritmos de forma intercambiável.  
  
Aplicação:

// Interface para estratégia de hash  
public interface HashingStrategy {  
 int hash(int chave, int capacidade);  
}  
  
// Implementações de estratégias de hash  
public class LinearProbingStrategy implements HashingStrategy {   
 // Implementação de Linear Probing  
}  
  
public class QuadraticProbingStrategy implements HashingStrategy {   
 // Implementação de Quadratic Probing  
}  
  
// Uso da estratégia de hash na TabelaHash  
public class TabelaHash {  
 private HashingStrategy hashingStrategy;  
  
 public TabelaHash(HashingStrategy strategy) {  
 this.hashingStrategy = strategy;  
 }  
}

## 3. Implementação do Padrão Singleton para Gerenciamento de Configurações da Tabela Hash

Motivo:  
Para centralizar as configurações da tabela hash, como capacidade e fator de carga, o padrão Singleton garante que apenas uma instância de configuração esteja em uso.  
  
Aplicação:

public class ConfiguracaoTabelaHash {  
 private static ConfiguracaoTabelaHash instancia;  
 private int capacidade;  
 private int fatorCarga;  
  
 private ConfiguracaoTabelaHash() {  
 this.capacidade = 10;  
 this.fatorCarga = 2;  
 }  
  
 public static ConfiguracaoTabelaHash getInstancia() {  
 if (instancia == null) {  
 instancia = new ConfiguracaoTabelaHash();  
 }  
 return instancia;  
 }  
}  
  
// Uso  
ConfiguracaoTabelaHash config = ConfiguracaoTabelaHash.getInstancia();

## 4. Modularizar Funções de Redimensionamento Usando o Padrão Template Method

Motivo:  
Se o redimensionamento tiver uma estrutura comum com variações por implementação, o Template Method permite um esqueleto compartilhado, facilitando a manutenção.  
  
Aplicação:

public abstract class TabelaHashTemplate {  
 protected int[] chaves;  
 protected Object[] valores;  
  
 public void redimensionar() {  
 int novaCapacidade = obterNovaCapacidade();  
 int[] novasChaves = new int[novaCapacidade];  
 Object[] novosValores = new Object[novaCapacidade];  
  
 for (int i = 0; i < chaves.length; i++) {  
 if (chaves[i] != 0) {  
 int indice = hash(chaves[i], novaCapacidade);  
 // Inserção no novo array  
 }  
 }  
  
 chaves = novasChaves;  
 valores = novosValores;  
 }  
  
 protected abstract int hash(int chave, int capacidade);  
 protected abstract int obterNovaCapacidade();  
}

## 5. Aplicação do Padrão Observer para Notificar Alterações de Capacidade

Motivo:  
Em um sistema onde o redimensionamento pode afetar outras partes, o padrão Observer permite manter componentes atualizados sobre alterações de capacidade sem acoplamento direto.  
  
Aplicação:

public interface Observer {  
 void atualizar(int novaCapacidade);  
}  
  
public class TabelaHash implements Observable {  
 private List<Observer> observers = new ArrayList<>();  
  
 public void adicionarObserver(Observer observer) {  
 observers.add(observer);  
 }  
  
 private void notificarObservers() {  
 for (Observer observer : observers) {  
 observer.atualizar(capacidade);  
 }  
 }  
  
 // No redimensionamento  
 private void redimensionar() {  
 // Redimensionamento...  
 notificarObservers();  
 }  
}