UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

MARJORY MEL FERREIRA FERRO LEMOS

LISTA 1: DESIGN PATTERNS

ARQUITETURA DE SOFTWARE

CORNÉLIO PROCÓPIO

2024

1 PADRÕES DE PROJETOS UTILIZADOS

Este projeto implementa dois principais padrões de projeto: Composite e Strategy. A seguir, são apresentados os padrões, as justificativas e os detalhes de sua implementação.

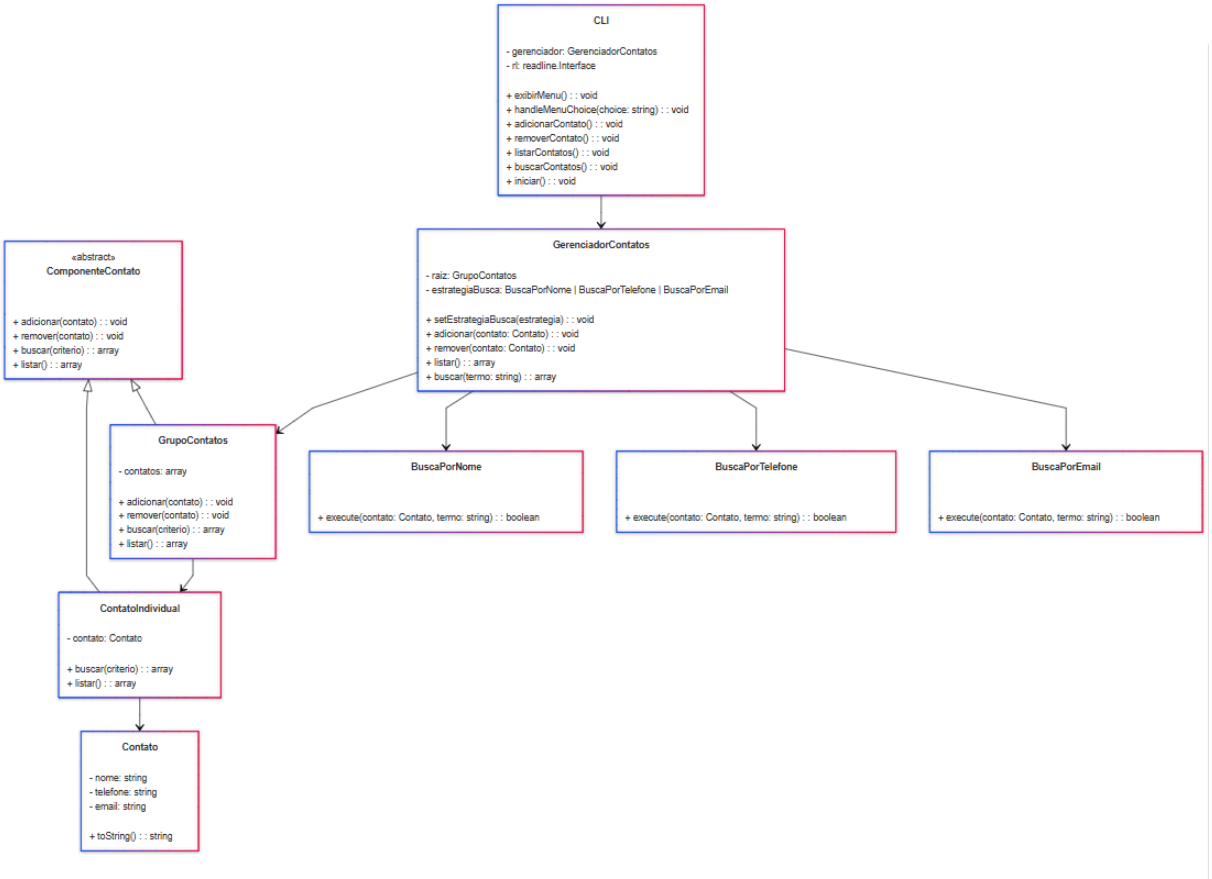

Figura 1. Diagrama do Projeto

1.1. Padrão Composite

O padrão Composite é utilizado para tratar hierarquias de objetos de forma uniforme. Ele permite que objetos individuais e grupos de objetos sejam tratados de maneira semelhante, facilitando operações recursivas como listagem e busca.

Por que usar Composite?

- Estruturas hierárquicas, como árvores, são comuns em problemas que lidam com coleções aninhadas.
- Simplifica a adição e remoção de elementos de forma consistente.

- Permite que operações como busca ou listagem sejam aplicadas uniformemente a contatos individuais e grupos.

Como foi implementado?

- A classe Contato representa a unidade básica de dados, armazenando informações como nome, telefone e email.

```
export class Contato {
  constructor(nome, telefone, email) {
    this.nome = nome;
    this.telefone = telefone;
    this.email = email;
  }

  toString() {
    return `Nome: ${this.nome}, Telefone: ${this.telefone}, Email: ${this.email}`;
  }
}
```

Figura 2. Classe Contato

- A classe base ComponenteContato define métodos genéricos (adicionar, remover, buscar e listar) que serão implementados ou sobrescritos pelas subclasses.

```
export class ComponenteContato {
    adicionar(contato) {}
    remover(contato) {}
    buscar(criterio) {}
    listar() {}
}
```

Figura 3. Classe Componente Contato

- As subclasses ContatoIndividual e GrupoContatos implementam a interface definida pela classe base:
    - ContatoIndividual encapsula um único objeto Contato.

```
export class ContatoIndividual extends ComponenteContato {
    constructor(contato) {
      super();
      this.contato = contato;
    }

    buscar(criterio) {
      return criterio(this.contato) ? [this.contato] : [];
    }

    listar() {
      return [this.contato];
    }
  }

```

Figura 4.Subclasse Contato Individual

- GrupoContatos gerencia uma coleção de contatos ou subgrupos de contatos, permitindo operações recursivas (por exemplo, busca em todos os contatos do grupo e seus subgrupos).

```
export class GrupoContatos extends ComponenteContato {
  constructor() {
    super();
    this.contatos = [];
  }

  adicionar(contato) {
    this.contatos.push(contato);
  }

  buscar(criterio) {
    return this.contatos.flatMap((contato) => contato.buscar(criterio));
  }
}
```

Figura 5.Subclasse Grupo de Contatos

1.2. Padrão Strategy

O padrão Strategy foi utilizado para permitir diferentes critérios de busca (por nome, telefone ou email). Ele facilita a substituição do algoritmo de busca sem modificar o sistema principal.

Por que usar Strategy?

- Segue o princípio aberto/fechado, permitindo que novas estratégias de busca sejam adicionadas sem alterar o código existente.
- Desacopla a lógica de busca do gerenciamento de contatos.
- Facilita a reutilização de estratégias em outros contextos, caso necessário.

Como foi implementado?

- Foram criadas classes específicas para cada tipo de busca (BuscaPorNome, BuscaPorTelefone, BuscaPorEmail), todas implementando o mesmo método (execute) para seguir uma interface consistente.
- A classe GerenciadorContatos permite configurar a estratégia de busca dinamicamente, delegando a execução à estratégia selecionada.

```
export class BuscaPorNome {
    execute(contato, termo) {
      return contato.nome.toLowerCase().includes(termo.toLowerCase());
    }
  }
```

Figura 6: Classe Busca Por Nome (exemplo de uma das Classes específicas criadas)

```
export class GerenciadorContatos {
  constructor() {
    this.raiz = new GrupoContatos();
    this.estrategiaBusca = new BuscaPorNome();
  }

  setEstrategiaBusca(estrategia) {
    this.estrategiaBusca = estrategia;
  }

  buscar(termo) {
    return this.raiz.buscar((contato) =>
      this.estrategiaBusca.execute(contato, termo)
    );
  }
}
```

Figura 7: Classe Gerenciador Contatos

2 MELHORES PRÁTICAS E PADRÕES DE CODIFICAÇÃO

2.1. Boas Práticas no Desenvolvimento

2.1.1. Modularização

- Cada classe ou funcionalidade foi dividida em módulos separados, permitindo o reuso de código, fácil manutenção e testes unitários específicos.

2.1.2. Uso de ES6+

- Implementação de classes, template literals e arrow functions.
- Exemplo: this.contatos.flatMap((contato) => contato.buscar(criterio));

2.1.3. Nomes Semânticos

- Nomes claros e descritivos para classes, métodos e variáveis.
- Exemplo: GrupoContatos, setEstrategiaBusca.

2.1.4. Tratamento de Erros

- Operações como busca e remoção consideram casos em que contatos podem não ser encontrados, evitando comportamentos inesperados.

2.2. Padrões de Codificação

2.2.1. Linters

- O projeto utiliza **ESLint** para garantir consistência no estilo de código.

2.2.2. Convenções de Estilo

- Identação padronizada (2 espaços).
- Uso de camelCase para variáveis e métodos.
- Classes iniciando com letras maiúsculas (PascalCase).

2.2.3. Separação de Responsabilidades

- Cada classe tem uma responsabilidade única, seguindo o Princípio da Responsabilidade Única (Single Responsibility Principle).
- Exemplo: GerenciadorContatos gerencia contatos; CLI lida com a interface.

REFERÊNCIAS

Código-Fonte no Github. Disponível em <https://github.com/marjorymell/gerenciador-contatos>.

Structural Design Patterns. Disponível em: <https://refactoring.guru/design-patterns/composite>. Acesso em: 12 nov .2024.

Structural Design Patterns. Disponível em: <https://refactoring.guru/design-patterns/strategy>. Acesso em: 12 nov .2024.

TEIXEIRA, M. **Como Configurar ESLint e Prettier 2021 | Medium.** Disponível em: <https://matheusteixeirajs.medium.com/como-configurar-eslint-e-prettier-para-seus-projetos-em-react-nodejs-e-typescript-53a2c0b9f5d4>. Acesso em: 12 nov .2024.