Alguns Padrões de Projeto com TypeScript

Uma abordagem prática para desenvolvimento moderno

Agenda

- Introdução aos Padrões de Projeto
- Padrões Criacionais (Factory, Singleton, Builder)
- Padrões Estruturais (Adapter, Decorator, Facade)
- Padrões Comportamentais (Observer, Strategy, State)
- Aplicações Práticas e Boas Práticas

O que são Padrões de Projeto?

- Soluções reutilizáveis para problemas comuns de design de software
- Criados pela "Gang of Four" (GoF) em 1994
- Promovem código mais limpo, manutenível e escalável
- Facilitam a comunicação entre desenvolvedores

Princípios fundamentais

Abstração, Encapsulamento, Polimorfismo, Composição sobre Herança

Por que usar Padrões de Projeto com TypeScript?

- TypeScript oferece tipagem estática que complementa os padrões
- Interfaces e tipos tornam os padrões mais robustos
- Autocompletion e detecção de erros em tempo de compilação
- Melhor documentação e manutenção do código

Categorias de Padrões de Projeto

Criacionais

Abstraem o processo de instanciação de objetos

Ex: Factory, Singleton, Builder, Prototype, Abstract Factory

Estruturais

Definem como classes e objetos são compostos

Ex: Adapter, Decorator, Facade, Composite, Proxy, Bridge

Comportamentais

Tratam da comunicação entre objetos

Ex: Observer, Strategy, State, Command, Chain of Responsibility

Padrões Criacionais

- **Factory:** Cria objetos sem especificar a classe exata
- **Singleton:** Garante uma única instância de uma classe
- **Builder:** Constrói objetos complexos passo a passo

Factory Pattern

Problema e Solução

- **Problema:** Criação de objetos sem acoplar o código à classes concretas
- **Solução:** Interface para criar objetos, deixando subclasses decidirem qual classe instanciar
- **Aplicação:** APIs, frameworks, sistemas com múltiplas implementações

Estrutura

```
Creator
   ├── FactoryMethod()
   └── SomeOperation()

Product  (interface)

ConcreteCreatorA
   └── FactoryMethod()  →  ConcreteProductA

ConcreteCreatorB
   └── FactoryMethod()  →  ConcreteProductB
```

Factory Pattern em TypeScript

```typescript
// Interface do produto
interface Payment {
  process(amount: number): boolean;
}

// Implementações concretas
class CreditCardPayment implements Payment {
  process(amount: number): boolean {
    console.log(`Processando ${amount} via Cartão de Crédito`);
    return true;
  }
}

class PayPalPayment implements Payment {
  process(amount: number): boolean {
    console.log(`Processando ${amount} via PayPal`);
    return true;
  }
}

// Factory
class PaymentFactory {
```

Singleton Pattern

Problema e Solução

- **Problema:** Necessidade de uma única instância global
- **Solução:** Classe que garante apenas uma instância e acesso global
- **Aplicação:** Conexões de banco de dados, caches, configurações

Estrutura

```
Singleton
  ├── -instance: Singleton (static)
  ├── -constructor() (private)
  └── +getInstance(): Singleton (static)
      └── if instance == null
          └── instance = new Singleton()
      └── return instance
```

Singleton Pattern em TypeScript

```typescript
class DatabaseConnection {
  private static instance: DatabaseConnection;
  private constructor() {
    // Inicialização da conexão
    console.log("Conexão com o banco de dados inicializada");
  }

  public static getInstance(): DatabaseConnection {
    if (!DatabaseConnection.instance) {
      DatabaseConnection.instance = new DatabaseConnection();
    }
    return DatabaseConnection.instance;
  }

  public query(sql: string): void {
    console.log(`Executando query: ${sql}`);
  }
}

// Uso
const db1 = DatabaseConnection.getInstance();
const db2 = DatabaseConnection.getInstance();
```

Builder Pattern

Problema e Solução

- **Problema:** Construção de objetos complexos com muitos parâmetros
- **Solução:** Separar a construção de um objeto complexo da sua representação
- **Aplicação:** Objetos com muitos parâmetros opcionais, configurações complexas

Estrutura

```
Builder  (interface)
   ├── buildPartA()
   └── buildPartB()

ConcreteBuilder
   ├── product
   ├── buildPartA()
   ├── buildPartB()
   └── getResult()

Director
   └── construct(builder)

Product
```

Builder Pattern em TypeScript

```typescript
class RequestBuilder {
  private url: string = '';
  private method: 'GET' | 'POST' | 'PUT' | 'DELETE' = 'GET';
  private headers: Record<string, string=""> = {};
  private body: any = null;

  setUrl(url: string): RequestBuilder {
    this.url = url;
    return this;
  }

  setMethod(method: 'GET' | 'POST' | 'PUT' | 'DELETE'): RequestBuilder {
    this.method = method;
    return this;
  }

  setHeader(key: string, value: string): RequestBuilder {
    this.headers[key] = value;
    return this;
  }

  setBody(body: any): RequestBuilder {
```

Padrões Estruturais

- **Adapter:** Compatibiliza interfaces incompatíveis
- **Decorator:** Adiciona responsabilidades a objetos dinamicamente
- **Facade:** Fornece interface simplificada para um subsistema complexo

Adapter Pattern

Problema e Solução

- **Problema:** Interfaces incompatíveis entre classes
- **Solução:** Converter a interface de uma classe em outra esperada pelo cliente
- **Aplicação:** Integração com bibliotecas de terceiros, sistemas legados

Estrutura

```
Client
   └── Uses    Target

Target  (interface)
   └── request()

Adapter
   ├── adaptee
   └── request()  →  adaptee.specificRequest()

Adaptee
   └── specificRequest()
```

Adapter Pattern em TypeScript

```typescript
// Interface esperada pelo cliente
interface ModernPaymentAPI {
  processPayment(amount: number): void;
}

// Serviço legado com interface incompatível
class LegacyPaymentSystem {
  makePayment(value: number, currency: string): boolean {
    console.log(`Pagamento de ${value} ${currency} processado pelo sistema legado`);
    return true;
  }
}

// Adapter para compatibilizar as interfaces
class PaymentAdapter implements ModernPaymentAPI {
  private legacySystem: LegacyPaymentSystem;

  constructor(legacySystem: LegacyPaymentSystem) {
    this.legacySystem = legacySystem;
  }

  processPayment(amount: number): void {
```

Decorator Pattern

Problema e Solução

- **Problema:** Adicionar funcionalidades a objetos sem alterar sua estrutura
- **Solução:** Envolver objetos em classes decoradoras com funcionalidades adicionais
- **Aplicação:** Logging, caching, autorização, validação

Estrutura

```
Component  (interface)
  └── operation()

ConcreteComponent
  └── operation()

Decorator
  ├── component
  └── operation()  →  component.operation()

ConcreteDecoratorA
  └── operation()  →  super.operation()  +  newBehavior()

ConcreteDecoratorB
  └── operation()  →  super.operation()  +  newBehavior()
```

Decorator Pattern em TypeScript

```typescript
// Interface componente
interface DataSource {
  readData(): string;
  writeData(data: string): void;
}

// Componente concreto
class FileDataSource implements DataSource {
  constructor(private filename: string) {}

  readData(): string {
    console.log(`Lendo dados do arquivo ${this.filename}`);
    return `dados de ${this.filename}`;
  }

  writeData(data: string): void {
    console.log(`Escrevendo "${data}" no arquivo ${this.filename}`);
  }
}

// Decorator base
abstract class DataSourceDecorator implements DataSource {
```

Facade Pattern

Problema e Solução

- **Problema:** Complexidade de um subsistema com muitas classes
- **Solução:** Interface simplificada para um conjunto de interfaces complexas
- **Aplicação:** APIs, bibliotecas, frameworks, integrações

Estrutura

```
Facade
   └── Métodos simplificados

Subsistema
   ├── ComponenteA
   ├── ComponenteB
   └── ComponenteC

Cliente
   └── Usa a Facade em vez de interagir diretamente com
```

Facade Pattern em TypeScript

```
// Subsistemas complexos
class AudioSystem {
  setVolume(level: number): void {
    console.log(`Volume ajustado para ${level}`);
  }

  tune(frequency: number): void {
    console.log(`Sintonizado em ${frequency}MHz`);
  }
}

class DisplaySystem {
  setBrightness(level: number): void {
    console.log(`Brilho ajustado para ${level}`);
  }

  setResolution(width: number, height: number): void {
    console.log(`Resolução ajustada para ${width}x${height}`);
  }
}

class InputSystem {
```

Padrões Comportamentais

- **Observer:** Notifica objetos sobre mudanças de estado
- **Strategy:** Encapsula algoritmos intercambiáveis
- **State:** Altera o comportamento quando o estado interno muda

Observer Pattern

Problema e Solução

- **Problema:** Notificar múltiplos objetos sobre mudanças
- **Solução:** Mecanismo de assinatura para notificar objetos sobre eventos
- **Aplicação:** Eventos de UI, notificações em tempo real, reatividade

Estrutura

```
Subject
   ├── observers: Observer[]
   ├── attach(observer)
   ├── detach(observer)
   └── notify()

Observer  (interface)
   └── update()

ConcreteObserverA
   └── update()

ConcreteObserverB
   └── update()
```

Observer Pattern em TypeScript

```typescript
// Interface do Observer
interface Observer {
  update(data: any): void;
}

// Subject (objeto observado)
class NewsPublisher {
  private observers: Observer[] = [];
  private latestNews: string = '';

  addObserver(observer: Observer): void {
    this.observers.push(observer);
  }

  removeObserver(observer: Observer): void {
    const index = this.observers.indexOf(observer);
    if (index !== -1) {
      this.observers.splice(index, 1);
    }
  }

  notifyObservers(): void {
```

Strategy Pattern

Problema e Solução

- **Problema:** Diferentes algoritmos para diferentes situações
- **Solução:** Encapsular algoritmos em classes separadas e torná-los intercambiáveis
- **Aplicação:** Validação, ordenação, cálculos, regras de negócio variáveis

Estrutura

```
Context
   ├── strategy
   ├── setStrategy(strategy)
   └── doSomething() → strategy.execute()

Strategy  (interface)
   └── execute()

 ConcreteStrategyA
   └── execute()

 ConcreteStrategyB
   └── execute()
```

Strategy Pattern em TypeScript

```typescript
// Interface da estratégia
interface SortStrategy {
  sort(data: number[]): number[];
}

// Estratégias concretas
class BubbleSortStrategy implements SortStrategy {
  sort(data: number[]): number[] {
    console.log("Ordenando com Bubble Sort");
    // Implementação do bubble sort
    return [...data].sort((a, b) => a - b);
  }
}

class QuickSortStrategy implements SortStrategy {
  sort(data: number[]): number[] {
    console.log("Ordenando com Quick Sort");
    // Implementação do quick sort
    return [...data].sort((a, b) => a - b);
  }
}
```

State Pattern

Problema e Solução

- **Problema:** Comportamento que muda baseado no estado interno
- **Solução:** Objeto que altera seu comportamento quando seu estado muda
- **Aplicação:** Workflows, máquinas de estado, controle de processos

Estrutura

```
Context
   ├── state
   ├── setState(state)
   ├── request1()
   └── request2()

State  (interface)
   ├── handle1(context)
   └── handle2(context)

ConcreteStateA
   ├── handle1(context)
   └── handle2(context)

ConcreteStateB
   ├── handle1(context)
   └── handle2(context)
```

State Pattern em TypeScript

```
// Interface de estado
interface OrderState {
  processOrder(order: Order): void;
  cancelOrder(order: Order): void;
}

// Estados concretos
class PendingState implements OrderState {
  processOrder(order: Order): void {
    console.log("Processando pedido pendente...");
    order.setState(new ProcessingState());
  }

  cancelOrder(order: Order): void {
    console.log("Pedido pendente cancelado");
    order.setState(new CanceledState());
  }
}

class ProcessingState implements OrderState {
  processOrder(order: Order): void {
    console.log("Pedido já está em processamento");
```

Aplicações Práticas no Desenvolvimento Web

Frontend

- Gerenciamento de estado (Redux/MobX)
- Componentização (React/Vue/Angular)
- Validação de formulários
- Renderização condicional

Backend

- Middleware (Express/NestJS)
- Injeção de dependências
- Acesso a dados (Repositories)
- APIs e rotas

Fullstack

- Arquitetura de aplicações
- Comunicação entre camadas
- DTOs e mapeamento de objetos
- Testes automatizados

Boas Práticas

- Não force o uso de padrões onde não são necessários
- Combine padrões para resolver problemas complexos
- Mantenha a simplicidade quando possível
- Documente o uso de padrões no seu código
- Use interfaces para desacoplar implementações

Recursos para Aprofundamento

Livros e Websites

- "Design Patterns" (GoF)
- "Refactoring to Patterns" (Joshua Kerievsky)
- refactoring.guru
- sourcemaking.com

Cursos e Repositórios

- github.com/torokmark/design_patterns_in_typescript
- Udemy: Design Patterns em TypeScript
- Pluralsight: Design Patterns Library
- Frontend Masters: TypeScript Fundamentos

Perguntas e Discussão

- Quais padrões você já utilizou em seus projetos?
- Quais desafios você encontrou ao implementar padrões?
- Como os padrões podem melhorar a arquitetura dos seus projetos?

Obrigado!