

TypeScript e Design Patterns

Por João Siles

Olá meus lindos aluninhos! Na aula de hoje vamos entender um pouco sobre **Design Patterns**.

Design Patterns (Padrões de Projeto) são soluções reutilizáveis para problemas comuns no desenvolvimento de software. Eles surgiram como uma forma de documentar boas práticas, facilitando a comunicação entre desenvolvedores e tornando o código mais organizado, de fácil manutenção e escalável.

✓ Para que servem?

- Resolver problemas recorrentes.
- Melhorar a legibilidade do código.
- Facilitar manutenção e evolução.
- Promover boas práticas de desenvolvimento.

! Quando usar?

- Quando você percebe que está resolvendo um problema recorrente.
- Quando precisa tornar seu código mais flexível e escalável.
- Quando quer melhorar a colaboração entre equipes com uma linguagem de padrões comum.

Mas antes de conhecer os 4 padrões mais comuns com os quais já trabalhei precisamos entender o conceito de **instância** que provavelmente foi trabalho na disciplina de Lógica de Programação. Uma **instância** é um **objeto criado a partir de uma classe**, pense na classe como um molde e a instância como o **objeto físico** feito a partir desse molde. Observe isso nesse código:

```
1 class Carro {  
2     marca: string;  
3  
4     constructor(marca: string) {  
5         this.marca = marca;  
6     }  
7 }  
8  
9 const carro1 = new Carro('Toyota'); // ♦ Instância do Carro  
10 const carro2 = new Carro('Honda'); // ♦ Outra instância  
11  
12 console.log(carro1.marca); // Toyota  
13 console.log(carro2.marca); // Honda  
14
```

Cada vez que usamos `new Carro()`, estamos criando uma **nova instância**, ou seja, um novo objeto independente com seus próprios dados. Nem sempre faz sentido criar várias instâncias. Dependendo do problema, **queremos uma única instância compartilhada**, ou então, precisamos de uma **forma padronizada de criar diferentes objetos**, ou ainda, precisamos que objetos **se comuniquem automaticamente**. Com base nisso vamos ver nossos padrões?

Singleton

Imagine que você precisa de uma **Central de Controle de Tráfego**, que gerencia informações como semáforos e velocidade nas estradas. Não faz sentido ter várias centrais. **Solução:** Usar o padrão **Singleton** para garantir uma única instância da central.

```
1 class CentralDeTrafego {
2     private static instance: CentralDeTrafego;
3
4     private constructor() {
5         console.log('Central de Tráfego iniciada');
6     }
7
8     static getInstance(): CentralDeTrafego {
9         if (!CentralDeTrafego.instance) {
10             CentralDeTrafego.instance = new CentralDeTrafego();
11         }
12         return CentralDeTrafego.instance;
13     }
14
15     emitirAlerta(mensagem: string) {
16         console.log(`Alerta: ${mensagem}`);
17     }
18 }
19
20 // Uso
21 const central1 = CentralDeTrafego.getInstance();
22 const central2 = CentralDeTrafego.getInstance();
23
24 central1.emitirAlerta('Acidente na rodovia!');
25
26 console.log(central1 === central2); // true (mesma instância)
27
```

Factory Method

Uma montadora fabrica diferentes tipos de carros: **SUV**, **Esportivo**, **Sedan**. Não faz sentido criar esses carros manualmente no código toda vez. **Solução:** Usar o padrão **Factory**, que cria carros conforme a necessidade.



```
1  interface Carro {
2      dirigir(): void;
3  }
4
5  class SUV implements Carro {
6      dirigir() {
7          console.log('Dirigindo um SUV');
8      }
9  }
10
11 class Esportivo implements Carro {
12     dirigir() {
13         console.log('Dirigindo um Esportivo');
14     }
15 }
16
17 class Sedan implements Carro {
18     dirigir() {
19         console.log('Dirigindo um Sedan');
20     }
21 }
22
23 class Montadora {
24     static fabricarCarro(tipo: string): Carro {
25         if (tipo === 'SUV') {
26             return new SUV();
27         } else if (tipo === 'Esportivo') {
28             return new Esportivo();
29         } else if (tipo === 'Sedan') {
30             return new Sedan();
31         } else {
32             throw new Error('Tipo de carro desconhecido');
33         }
34     }
35 }
36
37 // Uso
38 const meuCarro = Montadora.fabricarCarro('Esportivo');
39 meuCarro.dirigir();
```

Observer

Quando o motorista aciona o freio, você quer que:

- O painel mostre uma luz de freio.
- As lanternas acendam.
- Um alerta sonoro toque, se for uma frenagem brusca.

Sem o padrão **Observer**, cada sistema teria que consultar o estado do freio o tempo todo. Ineficiente. **Solução:** Aplicar esse pattern permite que todos os sistemas sejam notificados automaticamente quando o freio é acionado.



```
1  interface Observer {
2      update(evento: string): void;
3  }
4
5  class Painel implements Observer {
6      update(evento: string) {
7          console.log(`Painel: ${evento}`);
8      }
9  }
10
11 class LuzDeFreio implements Observer {
12     update(evento: string) {
13         console.log(`Luz de freio ativada: ${evento}`);
14     }
15 }
16
17 class AlertaSonoro implements Observer {
18     update(evento: string) {
19         console.log(`Alerta sonoro: ${evento}`);
20     }
21 }
22
23 class Freio {
24     private observers: Observer[] = [];
25
26     registrar(observer: Observer) {
27         this.observers.push(observer);
28     }
29
30     remover(observer: Observer) {
31         this.observers = this.observers.filter(o => o !== observer);
32     }
33
34     pressionar() {
35         console.log('Freio pressionado!');
36         this.notificar('Freio acionado');
37     }
38
39     private notificar(evento: string) {
40         this.observers.forEach(o => o.update(evento));
41     }
42 }
43
44 // Uso
45 const freio = new Freio();
46 const painel = new Painel();
47 const luz = new LuzDeFreio();
48 const alerta = new AlertaSonoro();
49
50 freio.registrar(painel);
51 freio.registrar(luz);
52 freio.registrar(alerta);
53
54 freio.pressionar();
```

Strategy

Um carro pode ter modos de direção:

- Esportivo (mais potência).
- Econômico (mais eficiente).
- Off-road (para terrenos difíceis).

Sem o **Strategy**, cada modo seria um if gigante no código. **Solução:** Permitir trocar o comportamento dinamicamente.



```
1  interface ModoDeDirecao {
2      dirigir(): void;
3  }
4
5  class Esportivo implements ModoDeDirecao {
6      dirigir() {
7          console.log('Modo Esportivo: Potência máxima!');
8      }
9  }
10
11 class Economico implements ModoDeDirecao {
12     dirigir() {
13         console.log('Modo Econômico: Economia de combustível.');
14     }
15 }
16
17 class OffRoad implements ModoDeDirecao {
18     dirigir() {
19         console.log('Modo Off-road: Preparado para terrenos difíceis.');
20     }
21 }
22
23 class Carro {
24     constructor(private modo: ModoDeDirecao) {}
25
26     setModo(modo: ModoDeDirecao) {
27         this.modo = modo;
28     }
29
30     dirigir() {
31         this.modo.dirigir();
32     }
33 }
34
35 // Uso
36 const meuCarro = new Carro(new Esportivo());
37 meuCarro.dirigir();
38
39 meuCarro.setModo(new Economico());
40 meuCarro.dirigir();
41
42 meuCarro.setModo(new OffRoad());
43 meuCarro.dirigir();
```

Design Patterns não são regras obrigatórias, são ferramentas. Use quando eles ajudam a tornar seu código mais **organizado, flexível e de fácil manutenção**. Para saber mais sobre eles você pode utilizar o site <https://refactoring.guru/design-patterns>.

The screenshot shows the homepage of the Refactoring Guru website. At the top, there's a red sidebar on the left featuring a cartoon raccoon holding a wrench, with the text "REFACTORING · GURU ·". Below it are links for "Premium Content", "Refactoring", "Design Patterns", "What is a Pattern", "Catalog", "Code Examples", and "Sign in / Contact us". The main content area has a blue header with "SPRING SALE" and a "Shop Now!" button. The main title "DESIGN PATTERNS" is in large, bold, black letters. To the right of the title is a cartoon raccoon standing next to a bicycle. Below the title, there's a section titled "Benefits of patterns" with a sub-section "Classification". Both sections have small descriptions and "More about" buttons. The background features a stylized illustration of people working in an office environment.

★ Premium Content

Refactoring

Design Patterns

What is a Pattern

Catalog

Code Examples

Sign in Contact us

SPRING SALE

Shop Now!

English

Contact us

Sign in

DESIGN PATTERNS

Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

What's a design pattern?

Benefits of patterns

Patterns are a toolkit of solutions to common problems in software design. They define a common language that helps your team communicate more efficiently.

More about the benefits »

Classification

Design patterns differ by their complexity, level of detail and scale of applicability. In addition, they can be categorized by their intent and divided into three groups.

More about the categories »