# DESENVOLVIMENTO WEB III

## Design Patterns – Padrões Comportamentais

Prof. Esp. Érick Henrique Pereira Nicolau

# O que são Padrões Comportamentais?

- Categoria de Design Patterns que lida com a comunicação e interação entre objetos.
- Define algoritmos e responsabilidades de forma flexível e desacoplada.
- Promove a reutilização de código, extensibilidade e manutenção.

# Padrões Comportamentais Abordados

- Observador (Observer)
- Estratégia (Strategy)
- Comando (Command)
- Iterador (Iterator)

# Observador (Observer)

- Objetivo: Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.
- Quando usar: Quando uma mudança em um objeto requer que outros objetos sejam atualizados, e você quer manter esses objetos fracamente acoplados.

```
class Assunto {
    // ... (métodos para adicionar e notificar observadores)
}

class Observador {
    atualizar() {
        // Lógica para reagir à notificação
    }
}
```

```
const botao = {
    estado: 'desligado',
    observadores: [],

    adicionarObservador(observador) {
        // Verificar se o objeto tem o método 'atualizar' antes de adicioná-lo
        if (typeof observador.atualizar === 'function') {
            this.observadores.push(observador);
        } else {
            console.error("Erro: O observador não possui o método 'atualizar'.");
        }
    },

    notificarObservadores() {
        this.observadores.forEach(observador => {
            try {
                observador.atualizar(this.estado);
            } catch (error) {
                console.error("Erro ao notificar observador:", error);
            }
        });
    },

    clicar() {
        this.estado = (this.estado === 'desligado') ? 'ligado' : 'desligado';
        console.log(`Botão: ${this.estado}`);
        this.notificarObservadores();
    }
};
```

```
// Observadores
const lampada = {
    atualizar: function(estadoBotao) {
        console.log(`Lâmpada: ${estadoBotao === 'ligado' ? 'acesa' : 'apagada'}`);
    }
}

const alarme = {
    atualizar: function(estadoBotao) {
        if (estadoBotao === 'ligado') {
            console.log("Alarme: ativado!");
        }
    }
}
```

```
// Adicionando observadores ao botão
botao.adicionarObservador(lampada);
botao.adicionarObservador(alarme);

// Simulando cliques no botão
botao.clicar();
botao.clicar();
```

# Resultado

Botão: ligado

Lâmpada: acesa

Alarme: ativado!

Botão: desligado

Lâmpada: apagada

# Estratégia (Strategy)

- **Objetivo:** Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis.
- **Quando usar:** Quando você tem diferentes algoritmos para realizar a mesma tarefa e quer que o cliente possa escolher qual usar em tempo de execução.

```
class Pagamento {
    constructor(estrategiaPagamento) { /* ... */ }
    processarPagamento(valor) { /* ... */ }
}

class PagamentoCartaoCredito {
    pagar(valor) { /* ... */ }
}

class PagamentoBoleto {
    pagar(valor) { /* ... */ }
}
```

```
class Pagamento {
    constructor(estrategiaPagamento) {
        this.estrategiaPagamento = estrategiaPagamento;
    }

    processarPagamento(valor) {
        this.estrategiaPagamento.pagar(valor);
    }
}

class PagamentoCartaoCredito {
    pagar(valor) {
        console.log(`Processando pagamento com cartão de crédito no valor
de R$ ${valor.toFixed(2)}`);
        // Lógica para processar pagamento com cartão de crédito
    }
}

class PagamentoBoleto {
    pagar(valor) {
        console.log(`Gerando boleto bancário no valor de R$
${valor.toFixed(2)}`);
        // Lógica para gerar o boleto
    }
}
```

```
// Simulando o processo de checkout
const metodoPagamento = 'boleto'; // ou 'cartaoCredito'
const valorTotal = 125.50;

let estrategiaPagamento;

if (metodoPagamento === 'cartaoCredito') {
    estrategiaPagamento = new PagamentoCartaoCredito();
} else if (metodoPagamento === 'boleto') {
    estrategiaPagamento = new PagamentoBoleto();
} else {
    console.log("Método de pagamento inválido!");
}

if (estrategiaPagamento) {
    const pagamento = new Pagamento(estrategiaPagamento);
    pagamento.processarPagamento(valorTotal);
}
```

# Comando (Command)

- Objetivo: Encapsula uma solicitação como um objeto, permitindo parametrizar clientes com diferentes solicitações, enfileirar ou fazer o registro (log) de solicitações e suportar operações que podem ser desfeitas.
- Quando usar: Quando você quer desacoplar o objeto que invoca uma operação do objeto que a executa, ou quando precisa implementar recursos como desfazer/refazer.

```
class ComandoLigarLuz {
    constructor(lampada) { /* ... */ }
    executar() { /* ... */ }
    desfazer() { /* ... */ }
}
```

```
class Lampada {
    constructor() {
        this.ligada = false;
    }

    ligar() {
        this.ligada = true;
        console.log("Lâmpada ligada!");
    }

    desligar() {
        this.ligada = false;
        console.log("Lâmpada desligada!");
    }
}

class ComandoLigarLuz {
    constructor(lampada) {
        this.lampada = lampada;
    }

    executar() {
        this.lampada.ligar();
    }

    desfazer() {
        this.lampada.desligar();
    }
}
```

```
// Criando uma lâmpada
const lampadaSala = new Lampada();

// Criando um comando para ligar a lâmpada
const comandoLigar = new ComandoLigarLuz(lampadaSala);

// Executando o comando
comandoLigar.executar(); // Saída: Lâmpada ligada!

// Desfazendo o comando
comandoLigar.desfazer(); // Saída: Lâmpada desligada!
```

# Iterador (Iterator)

- Objetivo: Fornece uma maneira de acessar os elementos de um objeto agregado sequencialmente sem expor sua representação interna.
- Quando usar: Quando você quer percorrer uma coleção de objetos de forma padronizada, sem se preocupar com a estrutura interna da coleção.

```
class IteratorColecao {
    constructor(colecao) { /* ... */ }
    proximo() { /* ... */ }
    hasNext() { /* ... */ }
}
```

```
class IteratorColecao {
    constructor(colecao) {
        this.colecao = colecao; //Varios livros
        this.index = 0; // índice 0 para controle
    }

    proximo() {
        if (this.hasNext()) {
            return this.colecao[this.index++]; // Retorna o
próximo livro e avança
        } else {
            return null; // Retorna null
        }
    }

    hasNext() {
        return this.index < this.colecao.length; // Verifica
se ainda há livros para percorrer
    }
}
```

```
const livros = [
    { titulo: “Memórias Póstumas de Brás Cubas", autor: "Machado de Assis" },
    { titulo: “O Senhor dos Anéis", autor: “J.R.R Tolkien" },
    { titulo: "1984", autor: "George Orwell" }
];

const iteradorLivros = new IteratorColecao(livros);

while (iteradorLivros.hasNext()) {
    const livro = iteradorLivros.proximo();
    console.log(`Título: ${livro.titulo}, Autor: ${livro.autor}`);
}
```