DESENVOLVIMENTO WEB III

Design Patterns - Padrões de Criação

Prof. Esp. Érick Henrique Pereira Nicolau

O que são Design Patterns?

- Soluções reutilizáveis para problemas comuns em design de software.
- Catálogo de boas práticas testadas e aprovadas pela comunidade.
- Facilitam a comunicação entre desenvolvedores e promovem a criação de código mais flexível, manutenível e escalável.

Padrões de Criação

- Lidam com a criação de objetos, fornecendo mecanismos para controlar o processo de instanciação.
- Permitem criar objetos de forma flexível e desacoplada do código que os utiliza.
- Principais padrões:
  - Fábrica Abstrata (Abstract Factory)
  - Construtor (Builder)
  - Singleton
  - Protótipo (Prototype)

Fábrica Abstrata (Abstract Factory)

- Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.
- Permite que o cliente crie objetos de uma família sem conhecer os detalhes de suas implementações.
- Promove o baixo acoplamento e a flexibilidade na criação de objetos.

```
class FabricaDeBotoes {
  criarBotao() {
    throw new Error('Método abstrato');
  }
}

class FabricaDeBotoesWindows extends FabricaDeBotoes {
  criarBotao() {
    return new BotaoWindows();
  }
}

class FabricaDeBotoesMac extends FabricaDeBotoes {
  criarBotao() {
    return new BotaoMac();
  }
}

class BotaoWindows
class BotaoMac

// Uso
const fabrica = new FabricaDeBotoesWindows();
const botao = fabrica.criarBotao();
botao.renderizar();
```

Construtor (Builder)

- Separa a construção de um objeto complexo de sua representação, permitindo que o mesmo processo de construção crie diferentes representações.
- Útil quando a criação de um objeto envolve muitos passos ou configurações opcionais.
- Permite criar objetos de forma incremental e controlada.

```
class PizzaBuilder {
  constructor() {
    this.pizza = {};
  }

  comMassa(tipoMassa) {
    this.pizza.massa = tipoMassa;
    return this; // Permite encadeamento de métodos
  }

  comMolho(tipoMolho) {
    this.pizza.molho = tipoMolho;
    return this;
  }

  comRecheio(tipoRecheio) {
    this.pizza.recheio = tipoRecheio;
    return this;
  }

  construir() {
    return this.pizza;
  }
}

// Uso
const pizza = new PizzaBuilder()
  .comMassa('fina')
  .comMolho('tomate')
  .comRecheio('calabresa')
  .construir();

console.log(pizza);
```

Singleton

- Garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso a ela.
- Útil para gerenciar recursos compartilhados, configurações globais ou objetos que devem existir em um único ponto do sistema.
- Deve ser usado com cautela, pois pode dificultar os testes e aumentar o acoplamento.

```
class Configuracoes {
  constructor() {
    if (Configuracoes.instancia) {
      return Configuracoes.instancia;
    }
    Configuracoes.instancia = this;
    // ... (inicialização das configurações)
  }
}

// Uso
const config1 = new Configuracoes();
const config2 = new Configuracoes();
console.log(config1 === config2); // true
```

Vantagens

- Acesso global: A instância única pode ser acessada de qualquer parte do código.
- Controle de acesso: Garante que apenas uma instância exista, evitando problemas de concorrência e inconsistência de dados.
- Economia de recursos: Evita a criação desnecessária de múltiplas instâncias, economizando memória e outros recursos.

Desvantagens

- Acoplamento global: O Singleton pode aumentar o acoplamento entre diferentes partes do sistema, tornando o código mais difícil de testar e modificar.
- Dificuldade de teste: Pode ser mais difícil escrever testes unitários para classes que dependem do Singleton, pois a instância única pode afetar o comportamento dos testes.
- Anti-padrão em alguns casos: Em algumas situações, o uso do Singleton pode ser considerado um anti-padrão, pois pode levar a um design menos flexível e modular.

Protótipo (Prototype)

- Permite criar novos objetos clonando um objeto existente (o protótipo).
- Útil quando a criação de objetos é custosa ou quando você precisa de cópias de um objeto com pequenas variações.
- Promove a flexibilidade na criação de objetos e reduz o número de subclasses

```
const pessoaPrototipo = {
  nome: 'João',
  idade: 30,
  falar() {
    console.log(`Olá, meu nome é ${this.nome}`);
  }
};

const novaPessoa = Object.create(pessoaPrototipo);
novaPessoa.nome = 'Maria';
novaPessoa.falar(); // Saída: Olá, meu nome é Maria
```