Padrões de Projeto

Singleton, Adapter e Iterator

Grupo 1 :

- Alisson Dias
- Fabio Sivellan
- Eric
- Vinicius
- João Vitor Freitas

Singleton - Padrão Criacional

Resolve 2 problemas:

- Garantir que uma classe tenha somente uma instância
- Fornecer um ponto global de acesso para a instância

Solução:

- Fazer o construtor padrão privado, para prevenir que outros objetos usem o operador new com a classe singleton.
- Criar um método estático de criação que age como um construtor, que chama o construtor privado para criar um objeto e o salva em um campo estático. Todas as chamadas seguintes para esse método retornam o objeto em cache.

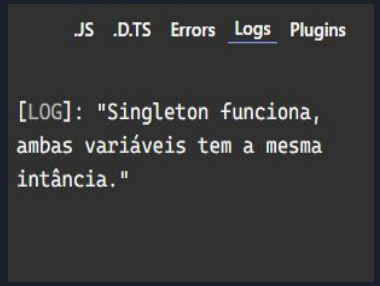
Singleton - Código

```
1  // A classe Singleton define o método "getInstance" que dá acesso ao cliente
2  class Singleton {
3      private static instance: Singleton;
4      private constructor() { } // O construtor singleton sempre deve ser privado para previnir novas
5      // O método estatico controla o acesso da instancia
6      public static getInstance(): Singleton {
7          if (!Singleton.instance) {
8              Singleton.instance = new Singleton();
9          }
10         return Singleton.instance;
11     }
12
```

Singleton - Código

```
18  // Código do cliente
19  function clientCode() {
20      const s1 = Singleton.getInstance();
21      const s2 = Singleton.getInstance();
22
23      if (s1 === s2) {
24          console.log("Singleton funciona, ambas variáveis tem a mesma intância.");
25      } else {
26          console.log("Erro!, variáveis tem instâncias diferentes.");
27      }
28  }
29  clientCode();
```

SAÍDA:



Singleton

Entretanto, o Singleton não é tão utilizado por possuir algumas desvantagens.

Desvantagens:

- Pode mascarar um design ruim, por exemplo, quando os componentes do programa sabem muito sobre cada um.
- Requer tratamento especial em um ambiente multithreaded para que múltiplas threads não possam criar um objeto singleton várias vezes.
- Viola o princípio de responsabilidade única, que diz que uma classe deve fazer apenas uma coisa, deve fazer bem e somente ela.

Adapter - Padrão Estrutural

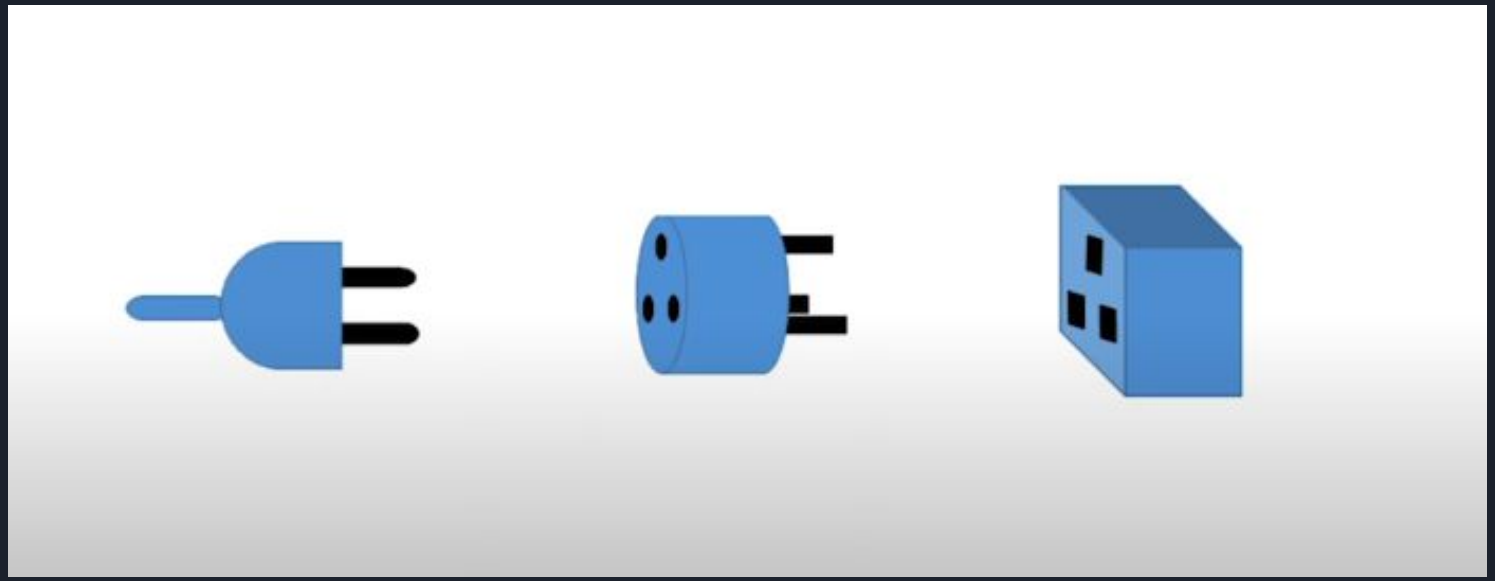
O Adapter (Adaptador) é um padrão de projeto estrutural que permite objetos com interfaces incompatíveis colaborarem entre si.

Adaptadores podem não só converter dados em vários formatos, mas também podem ajudar objetos com interfaces diferentes a colaborar.

- O adaptador obtém uma interface compatível com um dos objetos existentes.
- Usando essa interface, o objeto existente pode chamar os métodos do adaptador com segurança.
- Ao receber a chamada, o adaptador passa o pedido para o segundo objeto, mas em um formato e ordem que o segundo objeto espera.

Adapter

Analogia ao mundo real



Adapter - Vantagens e Desvantagens

Vantagens

- Desacopla o código da aplicação de terceiros.
- Você pode introduzir novos tipos de adaptadores no programa sem quebrar o código existente.

Desvantagens

- A complexidade geral do código aumenta porque você precisa introduzir um conjunto de novas interfaces e classes.
- Sobrecarga de desempenho.
- Potencial para introduzir erros.

Adapter - Código

```
// Classe com a interface incompatível
class SistemaAntigo{
    request(): void{
        console.log("Sistema Antigo: Requisição");
    }
}

// Interface que queremos usar
interface NovoSistema{
    fazerRequisicao(): void;
}

// Adaptador para a classe SistemaAntigo
class Adaptador implements NovoSistema{
    private sistemaAntigo: SistemaAntigo;

    constructor(sistemaAntigo: SistemaAntigo){
        this.sistemaAntigo = sistemaAntigo;
    }

    fazerRequisicao(): void{
        console.log("Adaptador está fazendo uma requisição de dados para o Sistema Antigo");
        this.sistemaAntigo.request();
    }
}

```

Adapter - Código

```
27  // Função principal
28  function principal(){
29    const sistemaAntigo = new SistemaAntigo();
30    const adaptador = new Adaptador(sistemaAntigo);
31
32    // Usando o adaptador para fazer a chamada
33    adaptador.fazerRequisicao();
34  }
35
36  // Executando a função principal
37  principal();
```

```
PROBLEMAS 3   SAÍDA   CONSOLE DE DEPURAÇÃO   TERMINAL   PORTAS

PS C:\Users\Alisson-Pc\Desktop\Curso\Trabalho Padrão> node index.js
Adaptador está fazendo uma requisição para o Sistema Antigo
Sistema Antigo: Requisição
```

Iterator - Padrão Comportamental

- Desacopla a intenção principal do objeto de modo como a sua interação é realizada.
- Permite vários tipos de iterators, facilitando a implementação de novos modos de travessia na mesma coleção.
- Encapsula os detalhes e monitora toda a travessia.
- Permite que a coleção troque de iterador em tempo de execução.
- Geralmente a linguagem de programação disponibiliza maneiras de trabalhar com iteradores.

Use um Iterator quando:

- Você precisa remover a complexidade de travessia de dentro da coleção principal. Isso permite que sua coleção foque apenas em armazenar de maneira eficiente.
- Sua coleção pode ter vários modos de travessia, como crescente, decrescente, pelo menor número de saltos, pulando de dois em dois, ou como preferir.
- Você quer disponibilizar protocolos de travessia para diferentes tipos de coleções.

Iterator - Código

codigo.js > ...

```js
class Iterator {
  constructor(array) {
    this.array = array;
    this.index = 0;
  }

  hasNext() {
    return this.index < this.array.length;
  }

  next() {
    if (this.hasNext()) {
      return this.array[this.index++];
    }
    return null; // ou pode lançar uma exceção para indicar o final da iteração
  }
}

// Exemplo de uso
const myArray = [1, 2, 3, 4, 5];
const iterator = new Iterator(myArray);

while (iterator.hasNext()) {
  console.log(iterator.next());
}
```