Padrões GOF

Nomes:
Bruno Mossmann Almeida
Eduardo Santos da Silva
FIlipe Lipert Evaldt de Senna
Luiz Fernando Silveira
Lucas Maia

PADRÃO ITERATOR

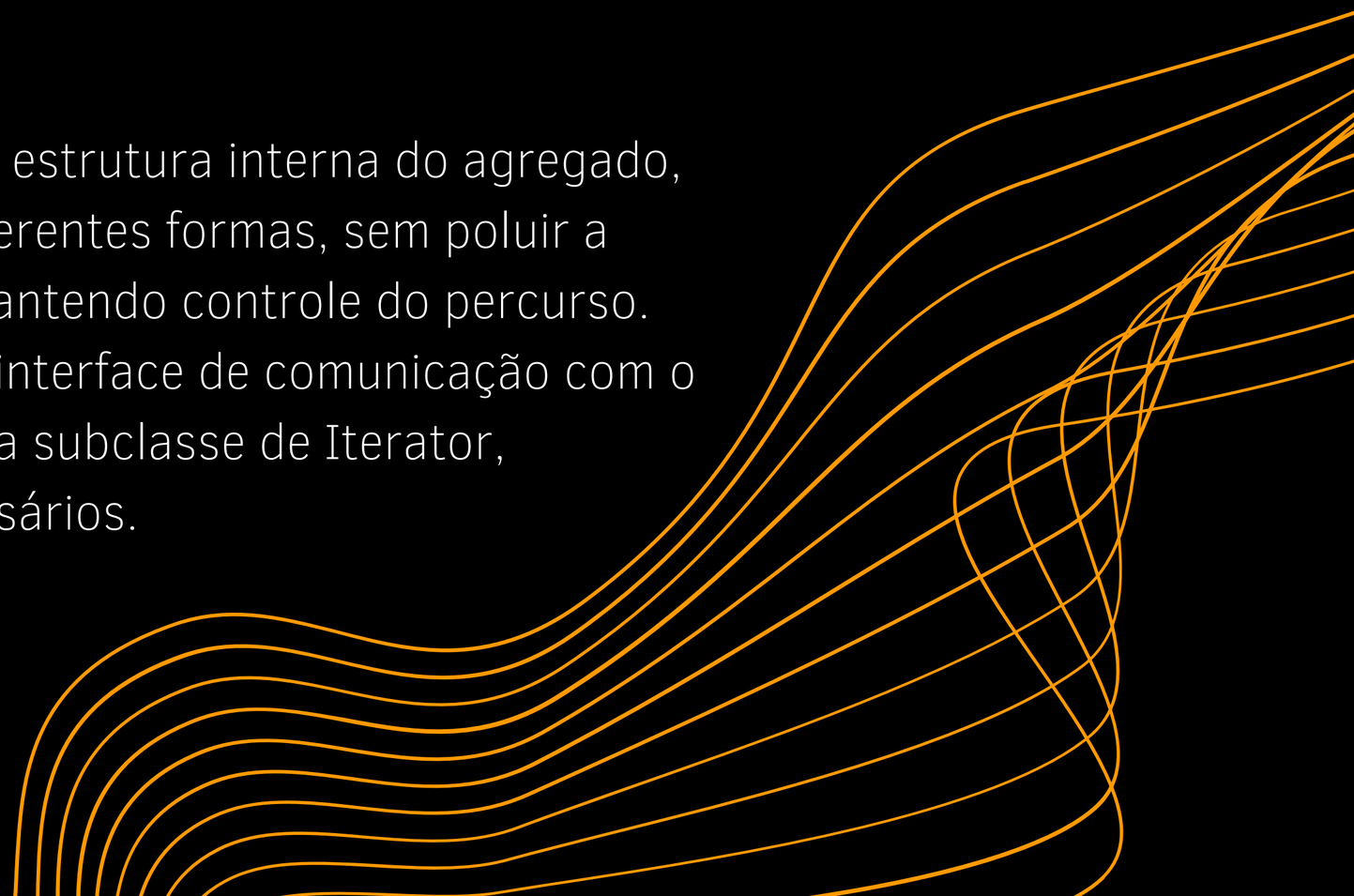
Intenção:

•Existe a necessidade de percorrer agregados quaisquer, em diversas ordens, sem conhecer sua representação subjacente.

Como usar:

•além de acessar os elementos sem conhecer a estrutura interna do agregado, pode ser desejável percorrer o agregado de diferentes formas, sem poluir a interface com inúmeros métodos para isso e mantendo controle do percurso.

•Criar uma classe abstrata Iterator, que terá a interface de comunicação com o cliente. Para cada agregado concreto, criar uma subclasse de Iterator, contendo a implementação dos métodos necessários.



Exemplos:

```ts
import { MyIteratorProtocol } from './my-iterator-protocol';
import { MyDefaultIterator } from './my-default-iterator';

export class MyDataStructure {
  private _items: string[] = [];
  private iterator: MyIteratorProtocol<string> = new MyDefaultIterator(this);

  addItem(...items: string[]): void {
    items.forEach((item) => this.items.push(item));
  }

  get items(): string[] {
    return this._items;
  }

  size(): number {
    return this.items.length;
  }

  changeIterator(iterator: MyIteratorProtocol<string>): void {
    this.iterator = iterator;
  }

  [Symbol.iterator](): MyIteratorProtocol<string> {
    return this.iterator;
  }

  resetIterator(): void {
    this.iterator.reset();
  }
}
```

```ts
import { MyDataStructure } from './my-data-structure';
import { MyReverseIterator } from './my-reverse-iterator';

const dataStructure = new MyDataStructure();
dataStructure.addItem('A', 'B', 'C', 'D', 'E', 'F');

const [a, b] = dataStructure;
console.log('ROUBADOS:', a, b);
console.log('EXECUTEI VÁRIAS COISAS E DEPOIS USEI O ITERATOR');
const [c, d, ...rest] = dataStructure;
console.log(c, d, rest);

dataStructure.resetIterator();
for (const data of dataStructure) {
  console.log(data);
}
console.log();
dataStructure.changeIterator(new MyReverseIterator(dataStructure));

for (const data of dataStructure) {
  console.log(data);
}
console.log();

console.log('AQUI PRECISO ZERAR O ITERATOR');
dataStructure.resetIterator();
for (const data of dataStructure) {
  console.log(data);
}
console.log();
```

PADRÃO STATE

Intenção:

•Permite que um objeto altere seu comportamento de acordo com mudança interna de estado.

•Dará a impressão que o objeto mudou de classe.

Como usar:

• fazer com que um objeto comporte-se de uma forma determinada por seu estado.

• Agregar um objeto Estado e delegar comportamento a ele.

Exemplos:

```typescript
import { ShoppingOrderState } from './shopping-order-state';
import { ShoppingOrder } from './shopping-order';
import { OrderApproved } from './order-approved';
import { OrderRejected } from './order-rejected';

export class OrderPending implements ShoppingOrderState {
  private name = 'OrderPending';

  constructor(private order: ShoppingOrder) {}

  getName(): string {
    return this.name;
  }

  approvePayment(): void {
    this.order.setState(new OrderApproved(this.order));
  }

  rejectPayment(): void {
    this.order.setState(new OrderRejected(this.order));
  }

  waitPayment(): void {
    console.log('O pedido já está no estado pagamento pendente.');
  }

  shipOrder(): void {
    console.log('Não posso enviar o pedido com pagamento pendente.');
  }
}
```

```typescript
import { ShoppingOrderState } from './shopping-order-state';
import { ShoppingOrder } from './shopping-order';
import { OrderPending } from './order-pending';
import { OrderRejected } from './order-rejected';

export class OrderApproved implements ShoppingOrderState {
  private name = 'OrderApproved';

  constructor(private order: ShoppingOrder) {}

  getName(): string {
    return this.name;
  }

  approvePayment(): void {
    console.log('O pedido já está no estado pagamento aprovado.');
  }

  rejectPayment(): void {
    this.order.setState(new OrderRejected(this.order));
  }

  waitPayment(): void {
    this.order.setState(new OrderPending(this.order));
  }

  shipOrder(): void {
    console.log('Enviando pedido para o cliente...');
  }
}
```

PADRÃO COMPOSITE

Intenção:

•compõe objetos em estruturas de árvore para representar hierarquias part-whole.

•Composite deixa o cliente tratar objetos individuais e composição de objetos uniformemente.

Como usar:

•Representar hierarquias de objetos part-whole.

•Permitir aos usuários ignorar a diferença entre composições de objetos e objetos individuais. Todos os objetos na estrutura são tratados uniformemente.

Exemplos:

```typescript
// Component
export abstract class ProductComponent {
  abstract getPrice(): number;

  // eslint-disable-next-line @typescript-eslint/no-unused-vars
  add(product: ProductComponent): void {}
  // eslint-disable-next-line @typescript-eslint/no-unused-vars
  remove(product: ProductComponent): void {}
}

// Leaf
export class ProductLeaf extends ProductComponent {
  constructor(public name: string, public price: number) {
    super();
  }

  getPrice(): number {
    return this.price;
  }
}

// Composite
export class ProductComposite extends ProductComponent {
  private children: ProductComponent[] = [];

  add(...products: ProductComponent[]): void {
    products.forEach((product) => this.children.push(product));
  }

  remove(product: ProductComponent): void {
    const productIndex = this.children.indexOf(product);
    if (productIndex !== -1) this.children.splice(productIndex, 1);
  }

  getPrice(): number {
    return this.children.reduce((sum, child) => sum + child.getPrice(), 0);
  }
}

```

```typescript
// Component
export abstract class ValidationComponent {
  abstract validate(value: unknown): boolean;
}

// Leaf
export class ValidateEmail extends ValidationComponent {
  validate(value: unknown): boolean {
    if (typeof value !== 'string') return false;
    return /@/.test(value);
  }
}

export class ValidateString extends ValidationComponent {
  validate(value: unknown): boolean {
    return typeof value === 'string';
  }
}

export class ValidateNumber extends ValidationComponent {
  validate(value: unknown): boolean {
    if (typeof value !== 'string') return false;
    return /\d+/.test(value);
  }
}

// Composite
export class ValidationComposite extends ValidationComponent {
  private readonly children: ValidationComponent[] = [];

  validate(value: unknown): boolean {
    for (const child of this.children) {
      const validation = child.validate(value);
      if (!validation) return false;
    }
    return true;
  }

  add(...validations: ValidationComponent[]): void {
    validations.forEach((validation) => this.children.push(validation));
  }
}

const validateEmail = new ValidateEmail();
const validateNumber = new ValidateNumber();
const validateString = new ValidateString();
const validationComposite = new ValidationComposite();
validationComposite.add(validateString, validateEmail, validateNumber);
console.log(validationComposite.validate('luiz123@email.com'));
```

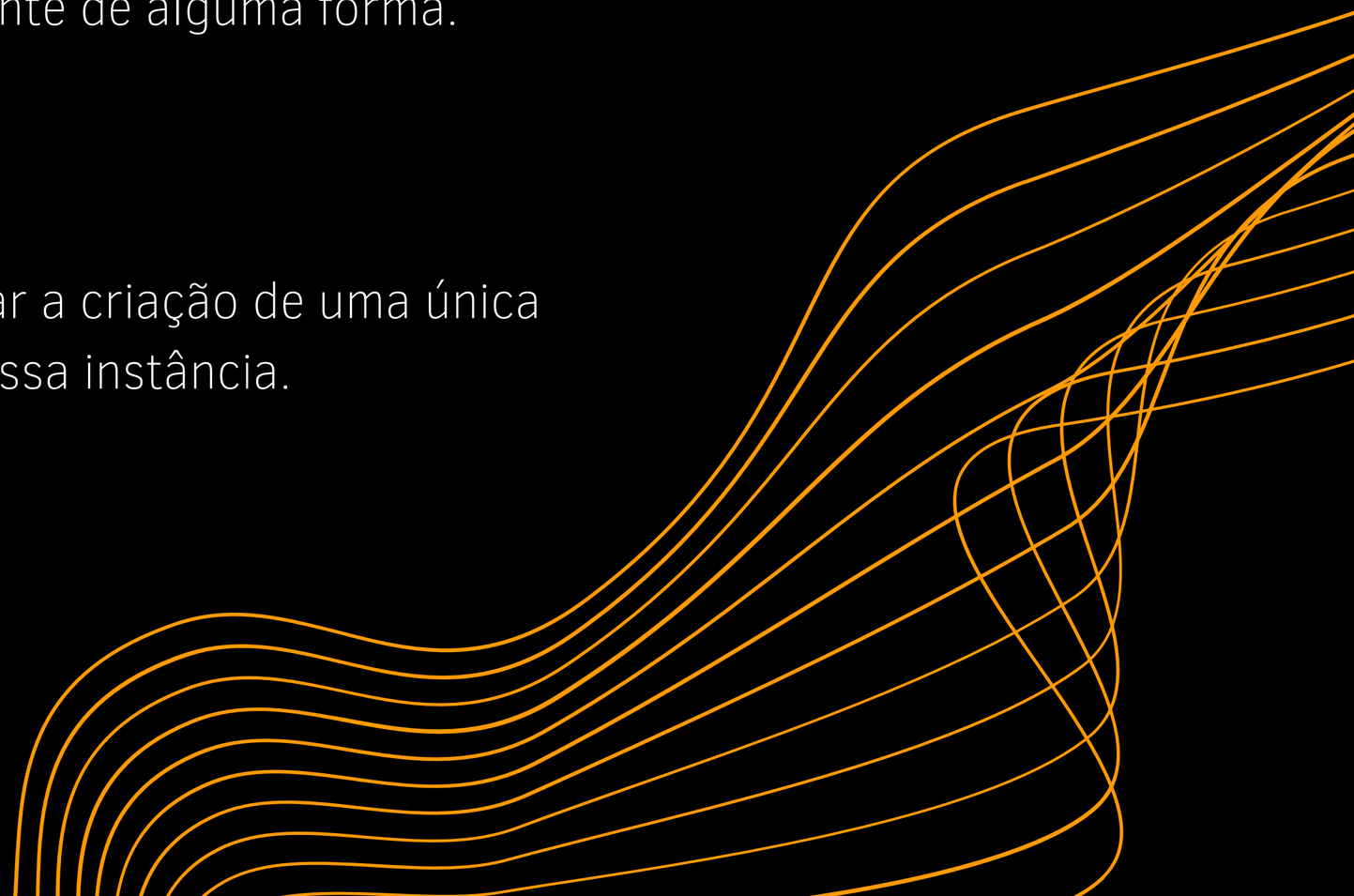
PADRÃO SINGLETON

Intenção:

•Utilizado quando é necessário garantir que uma classe possui apenas uma instância, que fica disponível às aplicações-cliente de alguma forma.

Como usar:

•Fazer a própria classe responsável de controlar a criação de uma única instância e de fornecer um meio para acessar essa instância.



Exemplos:

```ts
import { User } from '../interfaces/user';

export class MyDatabaseClassic {
  private static _instance: MyDatabaseClassic | null = null;
  private users: User[] = [];

  private constructor() {}

  static get instance(): MyDatabaseClassic {
    if (MyDatabaseClassic._instance === null) {
      MyDatabaseClassic._instance = new MyDatabaseClassic();
    }

    return MyDatabaseClassic._instance;
  }

  add(user: User): void {
    this.users.push(user);
  }

  remove(index: number): void {
    this.users.splice(index, 1);
  }

  show(): void {
    for (const user of this.users) {
      console.log(user);
    }
  }
}
```

```ts
import { User } from '../interfaces/user';

const users: User[] = [];

export const MyDatabaseModule = {
  add(user: User): void {
    users.push(user);
  },

  remove(index: number): void {
    users.splice(index, 1);
  },

  show(): void {
    for (const user of users) {
      console.log(user);
    }
  },
};
```

PRÁTICAS DE ENGENHARIA DE SOFTWARE

Professor: Jean Lopes