PRÁTICA LABORATORIAL STRATEGY

Objetivos:

- Implementar um exemplo de Strategy

EXERCÍCIOS

Parte 1

1. Vamos considerar um jogo de estratégia/RPG :-) em que diferentes tipos de unidades de combate têm comportamentos de ataque distintos. Vamos implementar o padrão de projeto Strategy para permitir que cada tipo de unidade execute o seu próprio comportamento de ataque.

   Neste exemplo, a interface AttackStrategy define o contrato para os diferentes comportamentos de ataque, e as classes MeleeAttackStrategy, RangedAttackStrategy e MagicAttackStrategy implementam essa interface com as suas próprias lógicas de ataque específicas.

   A classe Unit representa uma unidade de combate do jogo e possui uma estratégia de ataque. O construtor da classe recebe uma implementação de AttackStrategy e o método performAttack executa o comportamento de ataque correspondente.

   No método main, criamos instâncias de Unit para cada tipo de unidade desejado, passando a estratégia de ataque apropriada, e chamamos o método performAttack para realizar o ataque. A mensagem indicando o tipo de ataque é impressa no console.

   Com essa implementação, podemos adicionar novas estratégias de ataque facilmente, basta criar uma classe que implemente AttackStrategy e fornecer a implementação apropriada para o comportamento de ataque.

```
// Interface Strategy
public interface AttackStrategy {
    void attack();
}

// Implementações concretas de Strategy
public class MeleeAttackStrategy implements AttackStrategy {
    @Override
    public void attack() {
        System.out.println("Realizar ataque corpo a corpo!");
        // Lógica específica para ataque corpo a corpo
    }
}

public class RangedAttackStrategy implements AttackStrategy {
    @Override
    public void attack() {
        System.out.println("Realizar ataque à distância!");
        // Lógica específica para ataque à distância
    }
}

public class MagicAttackStrategy implements AttackStrategy {
    @Override
    public void attack() {
        System.out.println("Realizar ataque mágico!");
        // Lógica específica para ataque mágico
    }
}

// Classe que utiliza o padrão Strategy
public class Unit {
    private AttackStrategy attackStrategy;

    public Unit(AttackStrategy attackStrategy) {
        this.attackStrategy = attackStrategy;
    }

    public void performAttack() {
        attackStrategy.attack();
    }
}

// Exemplo de uso
public class Main {
    public static void main(String[] args) {
        Unit meleeUnit = new Unit(new MeleeAttackStrategy());
        meleeUnit.performAttack();

        Unit rangedUnit = new Unit(new RangedAttackStrategy());
        rangedUnit.performAttack();

        Unit magicUnit = new Unit(new MagicAttackStrategy());
        magicUnit.performAttack();
    }
}
```

2. Crie um sistema de processamento de pagamentos que precisa de calcular o valor total a ser pago por um cliente com base em diferentes estratégias de desconto. Vamos implementar o padrão de projeto Strategy para permitir a flexibilidade na escolha da estratégia de desconto.

   Crie a interface DiscountStrategy que define o contrato para as estratégias de desconto, e as classes NoDiscountStrategy, FlatDiscountStrategy e PercentageDiscountStrategy que implementam essa interface com suas próprias lógicas de cálculo de desconto.

   A classe PaymentProcessor utiliza uma estratégia de desconto por vez. O construtor da classe recebe uma implementação de DiscountStrategy e o método calculateTotal calcula o valor total aplicando o desconto adequado.

   No método main, crie uma instância de PaymentProcessor para cada estratégia de desconto desejada e chame o método calculateTotal passando o valor total a ser processado. O resultado é impresso na consola.

   Com essa implementação, podemos adicionar novas estratégias de desconto facilmente, basta criar uma nova classe que implemente DiscountStrategy e fornecer a implementação apropriada para o cálculo do desconto.

```
public class Main {
    public static void main(String[] args) {
        PaymentProcessor paymentProcessor = new PaymentProcessor(new NoDiscountStrategy());
        double total = paymentProcessor.calculateTotal(100.0);
        System.out.println("Total sem desconto: " + total);

        paymentProcessor = new PaymentProcessor(new FlatDiscountStrategy(10.0));
        total = paymentProcessor.calculateTotal(100.0);
        System.out.println("Total com desconto fixo: " + total);

        paymentProcessor = new PaymentProcessor(new PercentageDiscountStrategy(0.2));
        total = paymentProcessor.calculateTotal(100.0);
        System.out.println("Total com desconto percentual: " + total);
    }
}
```

3. Crie um sistema de transporte e implemente diferentes estratégias de cálculo de custos de portes. Vamos utilizar o padrão de projeto Strategy para permitir a flexibilidade na escolha da estratégia de cálculo de custo. Crie a interface ShippingStrategy que define o contrato para as estratégias de cálculo de custo de frete, e as classes StandardShippingStrategy, ExpressShippingStrategy e OvernightShippingStrategy que implementam essa interface com as suas próprias lógicas de cálculo de custo.
   Padrão: Peso*5.0
   Expresso: Peso*10.0
   Noturno: Peso*15.0
   A classe ShippingCalculator deve utilizar uma estratégia de cálculo de custo por vez. O construtor da classe recebe uma implementação de ShippingStrategy e o método calculateShippingCost calcula o custo de portes chamando o método calculateCost da estratégia correspondente.
   No método main, crie uma instância de ShippingCalculator para cada estratégia de cálculo de custo desejada e chamamos o método calculateShippingCost passando o peso do pacote. O custo de porte é impresso na consola.

```
public class Main {
    public static void main(String[] args) {
        ShippingCalculator calculator = new ShippingCalculator(new StandardShippingStrategy());
        double cost = calculator.calculateShippingCost(10.0);
        System.out.println("Custo de portes padrão: €" + cost);

        calculator = new ShippingCalculator(new ExpressShippingStrategy());
        cost = calculator.calculateShippingCost(10.0);
        System.out.println("Custo de portes expresso: €" + cost);

        calculator = new ShippingCalculator(new OvernightShippingStrategy());
        cost = calculator.calculateShippingCost(10.0);
        System.out.println("Custo de portes noturno: €" + cost);
    }
}
```

4. Crie um sistema de processamento de imagens que precisa oferecer diferentes estratégias de filtros para os utilizadores. Vamos implementar o padrão de projeto Strategy para permitir a flexibilidade na escolha da estratégia de filtro.

   Crie a interface ImageFilterStrategy que define o contrato para as diferentes estratégias de filtro, e as classes BlackAndWhiteFilterStrategy, SepiaFilterStrategy e VintageFilterStrategy que implementam essa interface com as suas próprias lógicas de aplicação de filtro:

   BlackAndWhite: System.out.println("Aplicar filtro preto e branco na imagem " + image);

   Sepia: System.out.println("Aplicar filtro sépia na imagem " + image);

   Vintage: System.out.println("Aplicar filtro vintage na imagem " + image);

   A classe ImageProcessor utiliza uma estratégia de filtro por vez. O construtor da classe recebe uma implementação de ImageFilterStrategy e o método applyFilter aplica o filtro correspondente à estratégia.

   No método main, criamos instâncias de ImageProcessor para cada estratégia de filtro desejada, passando a estratégia de filtro apropriada, e chamamos o método applyFilter para aplicar o filtro à imagem. A mensagem indicando o filtro aplicado é impressa na consola

   Com essa implementação, podemos adicionar novas estratégias de filtro facilmente, basta criar uma nova classe que implemente ImageFilterStrategy e fornecer a implementação apropriada para a aplicação do filtro.

```
public class Main {
    public static void main(String[] args) {
        ImageProcessor processor = new ImageProcessor(new BlackAndWhiteFilterStrategy());
        processor.applyFilter("imagem1.jpg");

        processor = new ImageProcessor(new SepiaFilterStrategy());
        processor.applyFilter("imagem2.jpg");

        processor = new ImageProcessor(new VintageFilterStrategy());
        processor.applyFilter("imagem3.jpg");
    }
}
```

Bom trabalho! ☺