

Design Patterns Utilizados no Projeto de Tênis de mesa

1. Singleton

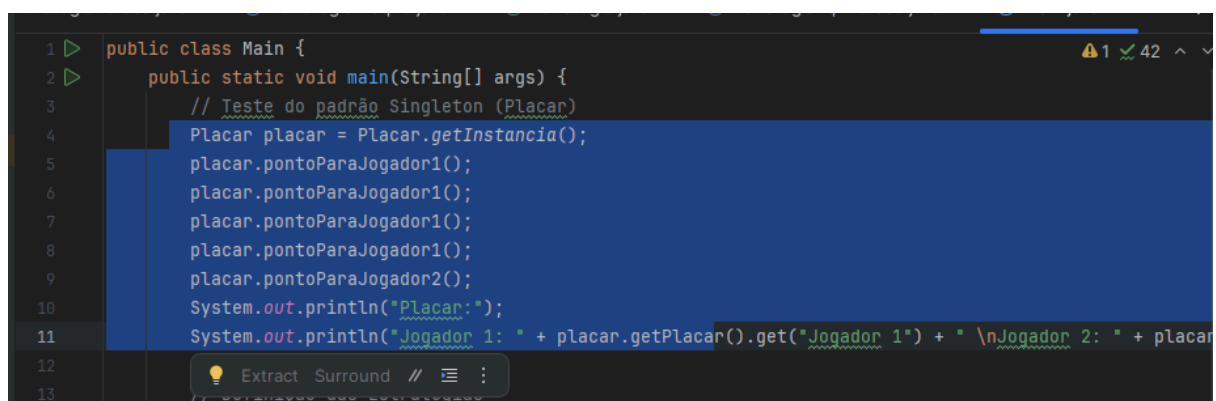
O padrão **Singleton** é um padrão criacional que tem como objetivo garantir que uma classe tenha apenas uma instância e fornecer um ponto global de acesso a essa instância.

No projeto de tênis de mesa o padrão **Singleton** foi aplicado para a classe **Placar**. A ideia é garantir que exista apenas uma instância do placar durante a execução do jogo, o que é fundamental para gerenciar e acompanhar o estado atual do jogo.

Justificativa

Usar o padrão **Singleton** para o **Placar** evita a criação de múltiplas instâncias desnecessárias, o que poderia levar a inconsistências nos dados do jogo. Além disso, o **Singleton** garante que todas as partes do código interajam com o mesmo placar.

Nota-se que existe apenas uma instância



```
1 public class Main {
2     public static void main(String[] args) {
3         // Teste do padrão Singleton (Placar)
4         Placar placar = Placar.getInstance();
5         placar.pontoParaJogador1();
6         placar.pontoParaJogador1();
7         placar.pontoParaJogador1();
8         placar.pontoParaJogador1();
9         placar.pontoParaJogador2();
10        System.out.println("Placar:");
11        System.out.println("Jogador 1: " + placar.getPlacar().get("Jogador 1") + " \nJogador 2: " + placar
12
13
```

2. Builder

O padrão Builder é um padrão **criacional** que permite a construção de objetos complexos passo a passo. Ele desacopla a construção de um objeto da sua representação final.

No projeto, o padrão Builder foi utilizado na construção de objetos **Jogador**. O **JogadorBuilder** permite criar instâncias de **Jogador** de maneira fluida, especificando nome, estilo de jogo, nível de habilidade e estratégia de forma modular e clara.

Justificativa

O uso do padrão **Builder** facilita a criação de objetos **Jogador** com várias características e configurações. Isso torna o código mais limpo e flexível, além de permitir a criação de jogadores com diferentes combinações de atributos sem sobrecarregar o construtor da classe **Jogador**.

Código mais limpo e flexível

```
// Teste do padrão Builder (Jogador)
Jogador jogador1 = new Jogador.JogadorBuilder()
    .nome("Vinicius")
    .estiloDeJogo("Ataque")
    .nivelDeHabilidade(9)
    .estrategia(estrategiaAtaque)
    .build();
```

3. Strategy

O padrão **Strategy** é um padrão comportamental que permite que você defina e altere o comportamento de um objeto em tempo de execução.

No projeto, o padrão Strategy foi aplicado para definir as diferentes estratégias de jogo (**EstrategiaAtaque**, **EstrategiaDefesa**, **EstrategiaEquilibrada**). Cada estratégia representa um comportamento específico que um jogador pode adotar durante o jogo.

Justificativa

O uso do padrão **Strategy** permite que as estratégias de jogo sejam definidas e alteradas de forma independente. Isso permite a flexibilidade, permitindo que o comportamento do jogador seja modificado sem alterar a lógica principal do jogo.

Mudando o comportamento em tempo de execução

```
// Definição das Estratégias
Estrategia estrategiaAtaque = new EstrategiaAtaque();
Estrategia estrategiaDefesa = new EstrategiaDefesa();
Estrategia estrategiaEquilibrada = new EstrategiaEquilibrada();

// Teste do padrão Builder (Jogador)
Jogador jogador1 = new Jogador.JogadorBuilder()
    .nome("Vinicius")
    .estiloDeJogo("Ataque")
    .nivelDeHabilidade(9)
    .estrategia(estrategiaAtaque)
    .build();

Jogador jogador2 = new Jogador.JogadorBuilder()
    .nome("Bruno")
    .estiloDeJogo("Defensivo")
    .nivelDeHabilidade(7)
    .estrategia(estrategiaDefesa)
    .build();
```