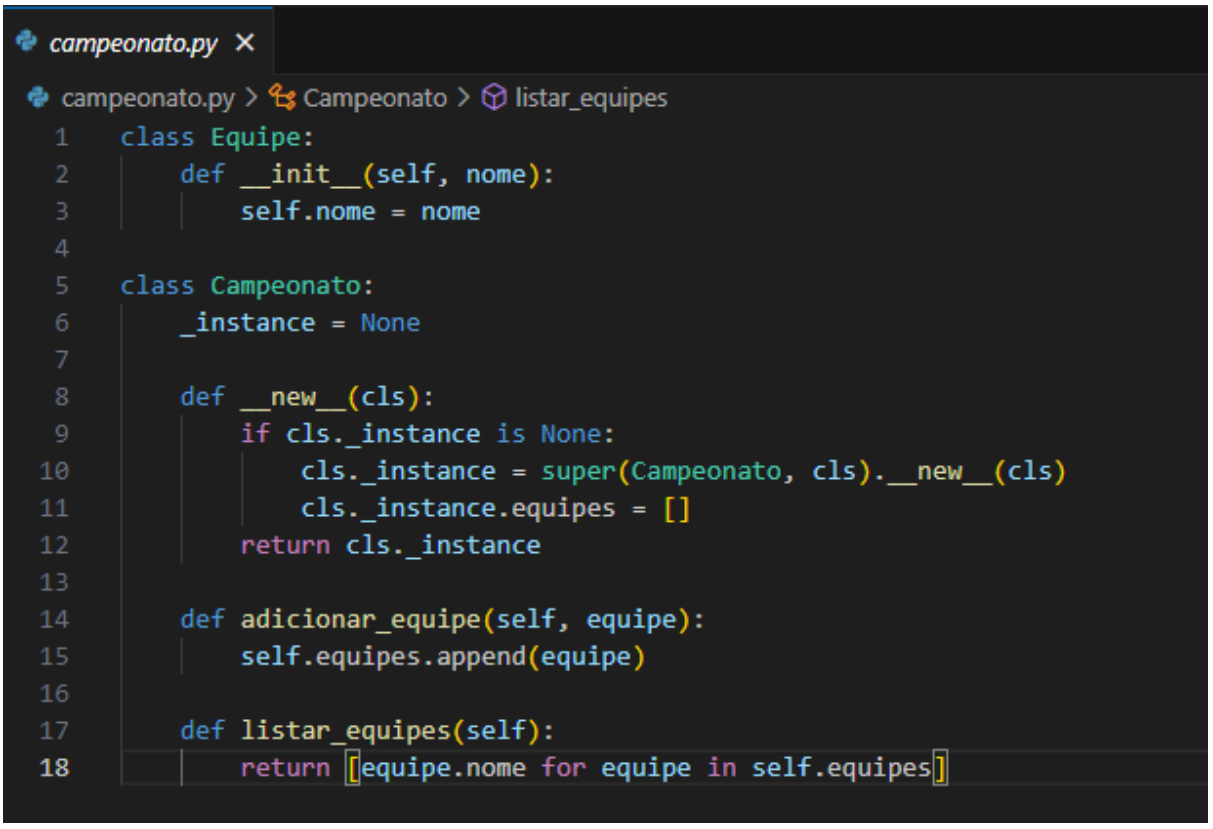
Aplicabilidade e Justificativa dos Design Patterns Utilizados

Análise dos padrões de design (design patterns) empregados no projeto de software desenvolvido. Os padrões utilizados são Singleton, Builder, Adapter e Observer. A seguir, cada um desses padrões é discutido em termos de aplicabilidade e justificativa para o seu uso.

Singleton

Aplicabilidade:

O padrão Singleton é utilizado para garantir que uma classe tenha apenas uma instância e forneça um ponto de acesso global a essa instância. No código fornecido, o padrão Singleton é aplicado na classe campeonato. Ao utilizar este padrão, assegura-se que o campeonato possua uma única instância ao longo de toda a aplicação, evitando inconsistências que poderiam surgir caso múltiplas instâncias fossem criadas inadvertidamente.



Justificativa:
A escolha do padrão Singleton para a classe Campeonato é justificada pela necessidade de centralizar a gestão das equipes participantes. Como o campeonato representa um contexto global dentro da aplicação, a existência de múltiplas instâncias poderia levar a resultados incoerentes, como a duplicação de equipes ou a manipulação de diferentes listas de equipes em diferentes partes da aplicação. Portanto, o Singleton garante que todas as partes do sistema compartilhem o mesmo campeonato, mantendo a integridade dos dados.

Builder

Aplicabilidade:
O padrão Builder é utilizado para a criação de objetos complexos de maneira flexível e estruturada. No código fornecido, o Builder é empregado na construção de um objeto carro. Através deste padrão, a construção de um carro pode ser feita passo a passo, especificando seus componentes (motor, chassis, pneus) de forma modular.

```python
class CarroBuilder:
    def __init__(self):
        self.carro = Carro()

    def construir_motor(self, motor):
        self.carro.motor = motor
        return self

    def construir_chassis(self, chassis):
        self.carro.chassis = chassis
        return self

    def construir_pneus(self, pneus):
        self.carro.pneus = pneus
        return self

    def get_carro(self):
        return self.carro
```

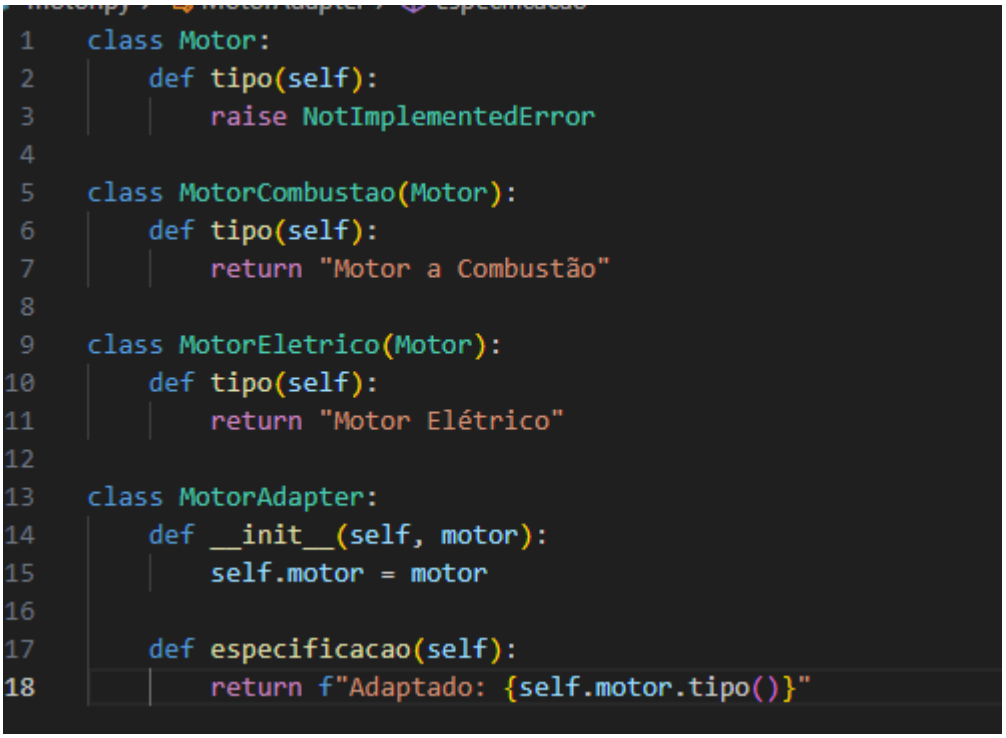
Justificativa:
A utilização do padrão Builder é justificada pela complexidade envolvida na criação de um objeto Carro, que possui múltiplos componentes com diferentes configurações possíveis. O Builder facilita a criação desses objetos, permitindo a personalização de suas partes sem a necessidade de lidar diretamente com um construtor extenso e complicado. Este padrão é especialmente útil quando as

configurações do objeto podem variar amplamente, como é o caso de diferentes modelos de carros que podem ter diferentes motores, chassis e pneus.

Adapter

Aplicabilidade:

O padrão Adapter é usado para permitir que duas interfaces incompatíveis trabalhem juntas. No projeto, o Adapter é aplicado para adaptar motores de combustão interna (MotorCombustao) e motores elétricos (MotorEletrico) a uma interface comum, possibilitando que ambos sejam tratados de forma uniforme no sistema.



```python
class Motor:
    def tipo(self):
        raise NotImplementedError

class MotorCombustao(Motor):
    def tipo(self):
        return "Motor a Combustão"

class MotorEletrico(Motor):
    def tipo(self):
        return "Motor Elétrico"

class MotorAdapter:
    def __init__(self, motor):
        self.motor = motor

    def especificacao(self):
        return f"Adaptado: {self.motor.tipo()}"
```

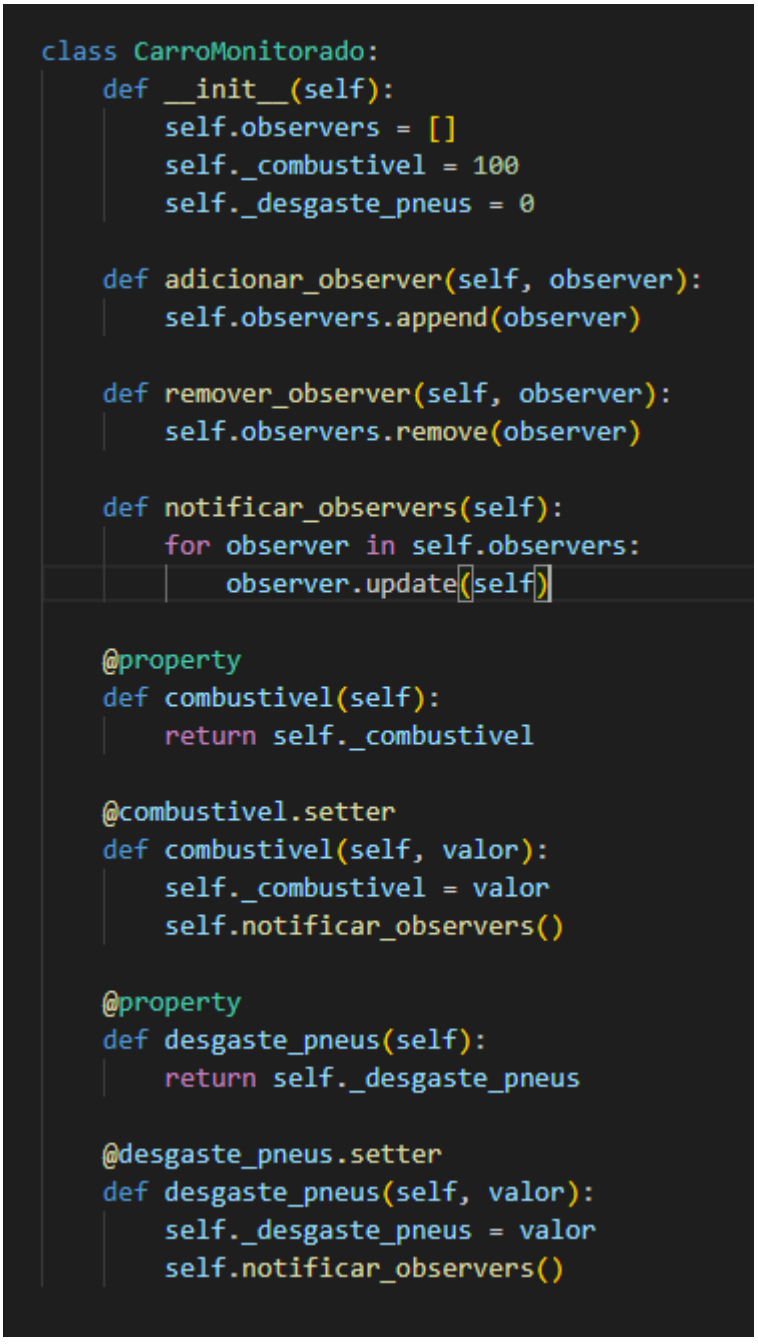
Justificativa:

A escolha do padrão Adapter se justifica pela necessidade de integrar diferentes tipos de motores em um sistema que espera uma interface comum. Como os motores a combustão e elétricos possuem implementações distintas, o Adapter permite que ambos sejam usados de maneira intercambiável sem modificar o código existente que depende dessa interface comum. Isso aumenta a flexibilidade e a extensibilidade do sistema, permitindo que novos tipos de motores sejam facilmente integrados no futuro.

Observer

Aplicabilidade:

O padrão Observer é empregado para estabelecer uma dependência entre objetos de forma que, quando um objeto muda de estado, todos os seus dependentes sejam notificados e atualizados automaticamente. No código fornecido, este padrão é utilizado para monitorar o estado de um carro (CarroMonitorado), notificando observadores (CombustivelObserver, PneusObserver) sobre mudanças no estado de combustível e desgaste dos pneus.



```python
class CarroMonitorado:
    def __init__(self):
        self.observers = []
        self._combustivel = 100
        self._desgaste_pneus = 0

    def adicionar_observer(self, observer):
        self.observers.append(observer)

    def remover_observer(self, observer):
        self.observers.remove(observer)

    def notificar_observers(self):
        for observer in self.observers:
            observer.update(self)

    @property
    def combustivel(self):
        return self._combustivel

    @combustivel.setter
    def combustivel(self, valor):
        self._combustivel = valor
        self.notificar_observers()

    @property
    def desgaste_pneus(self):
        return self._desgaste_pneus

    @desgaste_pneus.setter
    def desgaste_pneus(self, valor):
        self._desgaste_pneus = valor
        self.notificar_observers()
```

```python
# Observer: Monitorando o estado do carro
carro_monitorado = CarroMonitorado()
carro_monitorado.adicionar_observer(CombustivelObserver())
carro_monitorado.adicionar_observer(PneusObserver())

# Alterando estados do carro para acionar os observers
carro_monitorado.combustivel = 5  # Aviso de combustível baixo
carro_monitorado.desgaste_pneus = 80  # Aviso de desgaste de pneus
```

Justificativa:
O uso do padrão Observer é justificado pela necessidade de implementar um mecanismo de monitoramento em tempo real do estado de um carro, onde múltiplos aspectos (como combustível e pneus) precisam ser observados simultaneamente. Com o Observer, cada aspecto pode ser monitorado de forma independente, permitindo que o sistema reaja a diferentes condições sem a necessidade de uma lógica centralizada e complexa. Este padrão promove um design desacoplado e extensível, facilitando a adição de novos observadores ou estados monitorados no futuro.

Conclusão

Os padrões de design Singleton, Builder, Adapter e Observer foram escolhidos para resolver problemas específicos de design e estruturação do código. Cada um deles contribui para um sistema mais organizado, flexível e fácil de manter, garantindo que o software seja robusto e preparado para futuras extensões. O Singleton centraliza o gerenciamento de estado global, o Builder facilita a criação de objetos complexos, o Adapter promove a interoperabilidade entre diferentes interfaces, e o Observer oferece um mecanismo eficiente de notificação e reação a mudanças de estado.