Strategy Design Pattern

The **Strategy Pattern** is a behavioral design pattern that lets you define a family of algorithms, encapsulate each one as a separate class, and make them interchangeable. In our Unity game, we use this pattern to create different attack behaviors for enemies.

Problem

In our game, we want to include multiple types of enemies, each with its unique attack style:

- **Melee Enemies** that attack up close.
- **Ranged Enemies** that shoot projectiles from a distance.
- **Magic Enemies** that cast powerful spells.

Initially, you might be tempted to implement all attack behaviors within a single enemy class. However, this approach quickly leads to bloated, hard-to-maintain code as you add more attack styles.

Each time you introduce a new attack type or modify an existing one, you risk breaking the existing functionality, making the code fragile and difficult to extend.

Solution

The **Strategy Pattern** suggests that we extract each attack behavior into its own class. In our game:

- I created an `IAttackStrategy` interface that defines a common method, `Attack()`.
- Different attack behaviors were implemented as separate classes:
  - `MeleeAttack`
  - `RangedAttack`
  - `MagicAttack`

Each enemy has a reference to an `IAttackStrategy` object and delegates the attack action to it. This makes the enemy class independent of any specific attack implementation.

```
public interface IAttackStrategy
{
```

```
void Attack();
}
```

How It Works in Our Game

- The `Enemy` class is the **Context**. It maintains a reference to an `IAttackStrategy` object.
- Concrete strategies like `MeleeAttack`, `RangedAttack`, and `MagicAttack` implement the `IAttackStrategy` interface.

```
// Melee Attack Strategy
public class MeleeAttack : IAttackStrategy
{
    public void Attack()
    {
        Debug.Log("Melee Attack: Swinging a sword!");
    }
}

// Ranged Attack Strategy
public class RangedAttack : IAttackStrategy
{
    public void Attack()
    {
        Debug.Log("Ranged Attack: Shooting an arrow!");
    }
}

// Magic Attack Strategy
public class MagicAttack : IAttackStrategy
{
    public void Attack()
    {
        Debug.Log("Magic Attack: Casting a fireball!");
    }
}
```

Each class implements `IAttackStrategy` and provides a unique implementation of the `Attack()` method.

- We can switch an enemy's attack behavior dynamically by assigning a different strategy to the enemy.

This design allows us to easily introduce new attack types in the future without modifying the existing enemy logic. We simply create a new class implementing `IAttackStrategy` and plug it into any enemy.

Benefits in Our Game Design

- **Open/Closed Principle**: We can add new attack types without modifying existing enemy code.
- **Easier Maintenance**: Each attack behavior is encapsulated in its own class, reducing complexity.
- **Flexibility**: We can dynamically change enemy behaviors during gameplay.