Comparison of Example3_1 with Example3_2

What is happening?

Both examples contain logic for finding a path to the enemy and updating the player's state.

Advantages and Disadvantages

In `Example3_1`, the code looks slightly simpler, but the `UpdatePathToEnemy` method is responsible for all the logic of creating the path and updating the player's state. This combination reduces flexibility if these two aspects might change independently in the future.

In `Example3_2`, I like that the path creation logic is moved to a separate method, `TryBuildPathToCoord`, which improves the readability and maintainability of the code. As a result, we more clearly separate responsibilities between path creation and updating the player's state.

However, in both examples, the pathfinding occurs directly in the `Player` class. This greatly reduces code flexibility, forcing the `Player` class to always depend on a single pathfinding algorithm. To increase the flexibility of the logic, I would recommend using the Strategy pattern.

Additionally, the path calculation logic is called every `Update`, which can negatively affect performance since path calculation can take significant time. In this case, I would recommend throttling the path update and calling it not every frame but, for example, every second. Additionally, if the `currentEnemy` position does not change, the path might not need to be updated at all (though this is very situational, as other objects or characters might be moving).

Another point to pay attention to is the number of `null` checks. While a `null` check itself is not inherently wrong, in situations where reference-type objects can be returned from a method, it's preferable to know explicitly whether they can be `null`. For this purpose, nullable types should be used. Nullable types allow you to explicitly indicate that a value can be `null`, and this will be reflected in methods and properties. This is good for informativeness and helps handle some errors. For example, in the `TryBuildPathToCoord` method, you can explicitly indicate that the returned path might be `null`. If you want to avoid `null` checks altogether, you could consider the option where the method returns an empty list if the path is not found.

My Version

The source code can be found here:https://github.com/Igor1818/ExamplesForRefactoring/blob/main/Example3/Example3_Refactored.cs

Abstract strategy and concrete user implementation:

```
public interface IPathfindingStrategy
{
    List<Vector2>? FindPath(Vector2 start, Vector2 target);
}

public class CustomPathfindingStrategy : IPathfindingStrategy
{
    public List<Vector2>? FindPath(Vector2 start, Vector2 target)
    {
        return < много строк кода по созданию пути из start в target >;
    }
}
```

The Player class now gets the strategy in the constructor:

```
public class Player
{
    private bool isMoving;
    private Vector2 currentPosition;
    private Player? currentEnemy;
    private IPathfindingStrategy pathfindingStrategy;

    public Player(IPathfindingStrategy strategy)
    {
        pathfindingStrategy = strategy;
    }

    public void Update()
    {
        if (currentEnemy == null)
            return;

        var builtPath = pathfindingStrategy.FindPath(currentPosition, currentEnemy.currentPosition);
        UpdateMovementState(builtPath);
    }

    private void UpdateMovementState(List<Vector2>? path)
    {
        if (path == null)
        {
            currentEnemy = null;
            isMoving = false;
        }
        else
        {
            isMoving = true;
        }
    }
}
```

Client:

```
class Program
{

    public static void Main(string[] args)
    {
        IPathfindingStrategy pathfindingStrategy = new CustomPathfindingStrategy();
        var player = new Player(pathfindingStrategy);

        // This method can now be called in the update loop:
        player.Update();
    }
}
```

In this example, I applied the Strategy pattern. As a result, we can easily change or add new pathfinding strategies without modifying the `Player` class code. The code is open for extension but closed for modification, aligning with the Open/Closed Principle (OCP).