Comparison of Example4_1 with Example4_2

What is happening?

Both examples describe a system for using cheats in a game. The first example creates a more flexible but more complex system. The second example creates a simpler system.

Advantages and Disadvantages

In `Example4_1`, a complex system for using cheats is created. I like that, thanks to the `ICheatProvider` interface, we get a flexible and extensible system that allows us to add new cheats without changing the existing code. Another advantage of this example is that different classes are responsible for their own functions, thereby demonstrating the principle of separation of responsibilities.

The disadvantages of the first example include the use of Singleton, which can cause difficulties as the system complexity increases. Instead, I recommend using Dependency Injection to improve flexibility. For this purpose, you can use VContainer or Zenject.

Despite some flexibility in Example 1, `CheatManager` not only operates the high-level logic of the panel but also creates and configures individual elements. This approach violates the Single Responsibility Principle (SRP) and can make the system difficult to maintain as it expands. To address this drawback, I recommend using a simple factory for creating UI elements, making the code more flexible.

Another drawback of the first example is the use of nested classes (and interfaces). While using nested classes is not inherently a bad approach, their appropriateness depends on the specific context. If nested classes or interfaces are closely related to the parent class and are not intended to be used outside this context, then using nested classes is justified. However, in this example, `SomeManagerWithCheats` implements `ICheatProvider`, which is nested in `CheatManager`. The fact that the interface is hidden inside the class can be non-obvious and confusing. I recommend moving nested classes and interfaces to separate files, making the code more explicit and easily reusable. Moreover, reducing nesting will improve the code structure and make its structure easier to understand.

In `Example4_2`, a simpler implementation of the cheat mechanism can be observed.

However, this implementation reduces the system's flexibility and extensibility because all cheats are closely tied to the `SomeManagerWithCheats` class. Adding new cheats is difficult, as it requires modifying existing code, which violates the Open/Closed Principle (OCP).

My Version

The source code can be found here: https://github.com/Igor1818/ExamplesForRefactoring/blob/main/Example4/Example4_Refactored.cs

Let's use a simple factory to create individual UI elements. The CheatElementFactory class is responsible for creating UI elements of cheats:

```
public interface ICheatElementFactory
{
    CheatElementBehaviour CreateCheatElement(CheatActionDescription description, Transform parent);
}

public class CheatElementFactory : ICheatElementFactory
{
    private readonly CheatElementBehaviour _cheatElementPrefab;

    public CheatElementFactory(CheatElementBehaviour cheatElementPrefab)
    {
        _cheatElementPrefab = cheatElementPrefab;
    }

    public CheatElementBehaviour CreateCheatElement(CheatActionDescription description, Transform parent)
    {
        var element = UnityEngine.Object.Instantiate(_cheatElementPrefab, parent);
        element.Setup(description);
        return element;
    }
}
```

Take the CheatElementBehaviour class and the ICheatProvider interface out of the CheatManager class.

```
public interface ICheatProvider
{
    IEnumerable<CheatActionDescription> GetCheatActions();
}

public class CheatActionDescription
{
    public readonly string name;
    public readonly Action cheatAction;

    public CheatActionDescription(string name, Action cheatAction)
    {
        this.name = name;
        this.cheatAction = cheatAction;
    }
}
```

The CheatManager class has gotten a bit cleaner and clearer. Now it is only responsible for managing the panel and cheat elements:

```
public class CheatManager
{
    // Using the Singleton pattern can create problems when using multiple instances in different contexts.
    // Instead, I recommend using Dependency Injection to improve flexibility.
    // You can use VContainer or Zenject for this purpose.
    public static readonly CheatManager Instance = new CheatManager();
    private readonly List<ICheatProvider> _providers = new List<ICheatProvider>();
    private GameObject _panelPrefab;
    private GameObject _panel;
    private ICheatElementFactory _elementFactory;

    public void Setup(GameObject panelPrefab, ICheatElementFactory elementFactory)
    {
        _panelPrefab = panelPrefab;
        _elementFactory = elementFactory;
    }

    public void RegProvider(ICheatProvider provider)
    {
        _providers.Add(provider);
    }

    public void ShowCheatPanel()
    {
        if (_panel != null)
        {
            return;
        }

        _panel = UnityEngine.Object.Instantiate(_panelPrefab);
        CreateCheatElements();
    }

    public void HideCheatPanel()
    {
        if (_panel == null)
        {
            return;
        }

        UnityEngine.Object.Destroy(_panel);
        _panel = null;
    }

    private void CreateCheatElements()
    {
        foreach (var provider in _providers)
        {
            foreach (var cheatAction in provider.GetCheatActions())
            {
                _elementFactory.CreateCheatElement(cheatAction, _panel.transform);
            }
        }
    }
}
```

The SomeManagerWithCheats class has also become more self-explanatory. Now it is independent and not related to CheatManager:

```
public class SomeManagerWithCheats : ICheatProvider
{
    private int _health;

    public void Setup()
    {
        CheatManager.Instance.RegProvider(this);
    }

    public IEnumerable<CheatActionDescription> GetCheatActions()
    {
        yield return new CheatActionDescription("Cheat health", () => _health++);
        yield return new CheatActionDescription("Reset health", () => _health = 0);
    }
}
```

As a result of this approach, the system has become more flexible and extensible due to a more proper separation of responsibilities and the use of a simple factory for creating cheat elements. You can easily add new implementations of `ICheatProvider` and `ICheatElementFactory` to support different types of cheats and ways of displaying them. The use of interfaces increases abstraction and allows for adding new implementations, making the system more flexible and maintainable.