Comparison of Example2_1 with Example2_2

What is happening?

In both examples, the functionality of the `Player` class is extended. A mechanic for displaying the player's health is added. If the player's health decreases by more than 10, the widget is colored red.

In the first example, a delegate `HealthChangedDelegate` and an event `HealthChanged` are added for this purpose, which notifies about the change in the player's health. A delegate with parameters is used, which explicitly specifies the parameters passed in the event.

In the second case, `Changed` is declared using the standard `Action` delegate (without parameters) and a custom implementation of `add`/`remove`. The current and previous health values are compared, and the new value is stored in `previousHealth` for subsequent use.

Advantages and Disadvantages

In `Example2_1`, I like the explicit use of a delegate with parameters, which makes the event more informative. However, if additional parameters need to be added in the future, the delegate and related methods will have to be modified.

In `Example2_2`, I like the use of the standard `Action` delegate, which makes the code more concise. However, the lack of parameter passing in the event makes handling less flexible.

Another debatable point is the immediate invocation of the delegate `value` upon subscription to `Action`. This is done to immediately call the event handler to update the UI and display the current data. However, calling the delegate immediately upon subscription can be unexpected and may lead to unforeseen results.

Additionally, a drawback is that storing the previous value occurs in the `ExtProgram` class, which again violates the Single Responsibility Principle (SRP).

My Version

The source code can be found here:
https://github.com/Igor1818/ExamplesForRefactoring/blob/main/Example2/Example2_Refactored.cs

In my variant, I propose to use the Decorator pattern to add new functionality to the existing Player class:

```
public interface IHealth
{
    int Health { get; }

    event Action<int> HealthChanged;
}

public class ExtendedPlayer : IHealth
{
    public int Health => player.Health;
    public event Action<int> HealthChanged;
    private Player player;

    public ExtendedPlayer(Player player)
    {
        this.player = player;
    }

    public void Hit(int damage)
    {
        player.Hit(damage);
        HealthChanged?.Invoke(player.Health);
    }
}
```

I suggest to put all logic with displaying information separately:

```
public class HealthView : IDisposable
{
    private TextView textView = new TextView();
    private int? currentHealth;
    private IHealth healthObject;

    public HealthView(IHealth healthObject)
    {
        this.healthObject = healthObject;
        this.healthObject.HealthChanged += OnHealthChanged;
        OnHealthChanged(this.healthObject.Health);
    }

    public void Dispose()
    {
        healthObject.HealthChanged -= OnHealthChanged;
    }

    private void OnHealthChanged(int newHealth)
    {
        textView.Text = newHealth.ToString();

        if (currentHealth != null && newHealth - currentHealth < -10)
        {
            textView.Color = Color.red;
        }
        else
        {
            textView.Color = Color.white;
        }
        currentHealth = newHealth;
    }
}
```

The HeathView class has no tight coupling with the ExtendedPlayer class and only uses the IHealth interface to get information and subscribe to events. Now HealthView is only responsible for displaying and ExtendedPlayer is responsible for managing the health state.

With this approach, the client side of the code is significantly reduced and each class performs only the functionality assigned to it without violating the Single Responsibility Principle (SRP).

```
public static void Main(string[] args)
{
    ...
    var playerDecorator = new ExtendedPlayer(player);
    var healthView = new HealthView(playerDecorator);

    playerDecorator.Hit(settings.Damage);
}
```