Comparison of Example1_1 with Example1_2

What is happening?

In both examples, a Player class is created, and then it takes damage. The two implementations differ slightly from each other.

In the first variant, the Player class is created with an initial amount of health, the value of which is passed through the constructor. This health value is hardcoded. In the second case, the Player class is serializable. This means that we can save and restore the player's state (at least, this is what is implied by the program).

The method of inflicting damage is different. In the first case, it's done through the SetHealth method, which sets an absolute value for the player's health. In the second case, it's done through the Hit method, which changes the player's health by a specified value. Additionally, in the second example, there is a Settings class that defines the amount of damage to the player. It is also serializable.

Advantages and Disadvantages

`Example1_1` looks simpler and clearer. I like that the `Player` class is a pure C# class. This means that we can use a constructor, add mechanics for the `Dispose` method, and not be dependent on any framework. At the same time, this example lacks a mechanism for saving the player, and it has hardcoded data (`NewPlayerHealth`, `Damage`).

`Example1_2` uses a serialization mechanism that allows saving and restoring the player's state and parameters. Therefore, this approach appears more flexible, as values are loaded from files rather than being hardcoded in the program. However, the second example has several drawbacks:

- There's no indication of the initial value for the player's health.
- The serialization mechanism seems strange: According to the program, it is assumed that `Health` (or `Damage`) will be saved. However, this is not the case. The value of the `Health` property will not be automatically restored from the file when using properties. To fix this, it's necessary to replace the private setter with a public one. However, this might be undesirable. Therefore, it's better to use the `[JsonProperty]` attribute. Another option is to use a helper variable.
- Combining the data-saving logic and the `Player` class can lead to problems as the program becomes more complex. I would recommend separating the data to be saved into a different class. This aligns with the SOLID principles, specifically the Single

Responsibility Principle (SRP), which states that a class should have only one reason to change.

- The `Hit` method does not perform any validation of the `damage` value. I would recommend handling the case for a negative value.
- There is no check for when there is no saved data. In this case, default values should be used.

My Version

The source code can be found here: https://github.com/Igor1818/ExamplesForRefactoring/blob/main/Example1/Example1_Refactored.cs

Let's split the stored data into separate lightweight structures:

```
[Serializable]
public struct PlayerData
{
    public int Health { get; set; }
}

[Serializable]
public struct SettingsDat
{
    public int Damage { get; set; }
}
```

For each entity we use pure C# (POCO) class. In the Hit method add a check for the negative value of the damage parameter.

```
public class Settings
{
    public int Damage { get; }

    public Settings(int damage)
    {
        Damage = damage;
    }
}

public class Player
{
    public int Health { get; private set; }

    public Player(int health)
    {
        Health = health;
    }

    public void Hit(int damage)
    {
        if (damage <= 0)
        {
            throw new ArgumentException("Damage must be a positive value.", nameof(damage));
        }

        Health -= damage;
    }
}
```

Client, where all entities are created and checked for the existence of data:

```
class Program
{
    private const string NewPlayerPath = "NewPlayer.json";
    private const string SettingsPath = "Settings.json";
    private const int DefaultHealth = 100;
    private const int DefaultDamage = 10;

    public static void Main(string[] args)
    {
        var health = TryLoadFromFile(NewPlayerPath, out PlayerData playerData) ? playerData.Health : DefaultHealth;
        var player = new Player(health);

        var damage = TryLoadFromFile(SettingsPath, out SettingsData settingsData) ? settingsData.Damage : DefaultDamage;
        var settings = new Settings(damage);

        player.Hit(settings.Damage);
    }

    public static bool TryLoadFromFile<T>(string filePath, out T result)
    {
        // This is where the data from the file is deserialized
        throw new NotImplementedException();
    }
}
```