Role Play Attributes README

There comes a time when many a gamedev sets out for adventure but first must create their own stat class. That is to say a class which captures a game "stat", or "statistic" for lack of a better word, like health, attack, defense, etc. This one is mine. Oh wait! there is a better word: "attribute." But I don't want to take "attribute"[1] from your game, so I'll call mine `IModifiableValue<T>`. Consider it a sound building block for you to create your own attribute class.

These attributes and their derivatives may affect and be effected by a multitude of transient things, e.g, a sword that bestows an attack advantage; a shield that raises one's defense; a ring that regenerates health. Because of that attributes ought to respect the following requirements.

Requirements

- An attribute's value shall be altered non-destructively.
- Because so many things may alter an attribute, it shall notify us when changed.

Features

- Interface Based

The heart of this library is defined by a few interfaces. So one can substitute their own implementations. Especially helpful if one wants to define modifiers in Unity for instance.

- Supports Generics

Too many libraries decide all attributes must be a `float` or an `int`. With this library one can choose which type suits an attribute, and they may still interact with one another.

- Generic Math

Many a library probably shied away from generics because .NET has not had generic math support, i.e., it was not possible to write `T Plus<T>(T a, T b) => a + b`. The release of .NET 7 will have that support, which this library makes use of. In addition a workaround makes it possible to present the same API with netstandard2.0, which is important if one wants to use this library with Unity3D.

- Flexible Modifiers

Sure, one can add, minus, multiply, and divide their stats, but what about clamping the value? Is it easy to add that feature? With this library one can implement their own modifier or create ad hoc ones easily.

Supports

- Unity 2021.3 and later

Barebones Example

```
using SeawispHunter.RolePlay.Attributes;

var health = new ModifiableValue<float>(100f);
Console.WriteLine($"Health is {health.value}."); // Prints: Health is 100.
health.modifiers.Add(Modifier.Times(1.10f, "+10% health"));
Console.WriteLine($"Health is {health.value}."); // Prints: Health is 110.
health.modifiers.Add(Modifier.Plus(5f, "+5 health"));
Console.WriteLine($"Health is {health.value}."); // Prints: Health is 115.
```

[1] Or "Attribute", "IAttribute", etc.

Attribute

At its root, an attribute has an `initial.value`. With no modifiers present, its `value` equals its `initial.value`; its `value` is altered by modifiers starting from its `initial.value`.

```
public interface IModifiableValue<T> {
  T initial.value { get; set; }
  T value { get; }
  /** The list implementation sets up property change events automatically. */
  ICollection<IModifier<T>> modifiers { get; }
  event PropertyChangedEventHandler PropertyChanged;
}
```

Just to be explicit, an attribute's value v that had an initial value i and three modifiers m_1, m_2, m_3 would be computed like this:

$$v = m_3(m_2(m_1(i)))$$

Modifier

A modifier accepts a value and can change it arbitrarily.

```
public interface IModifier<T> {
  bool enabled { get; set; }
  T Modify(T given);
  event PropertyChangedEventHandler PropertyChanged;
}
```

However, often times the changes one wants to make are simple: add a value, multiple a value, or substitute a value so these are made convenient for `int`, `float`, and `double` types.

```
public static class Modifier {
  public static IModifier<T> Plus(T value);
  public static IModifier<T> Minus(T value);
  public static IModifier<T> Times(T value);
  public static IModifier<T> Divide(T value);
  public static IModifier<T> Substitute(T value);
}
```

Change Propogation

These classes use the `INotifyPropertyChanged` interface to propogate change events, so any modifier that's changed or added will notify its attribute which will notify any of its listeners. No need to poll for changes to an attribute.

Abridged API

The API shown above is abridged to make its most salient points easy to understand. The actual code includes some abstractions like `IValue<T>` and `IReadOnlyValue<T>` which are used to make attributes reuseable as modifiers for instance.

Indeed this README has outright lies and is better for it.

Further Examples

Using Notifications

```
var health = new ModifiableValue<float>(100f);
health.PropertyChanged += (_, _) => Console.WriteLine($"Health is {health.value}.");
health.modifiers.Add(Modifier.Times(1.10f, "+10% health"));
// Prints: Health is 110.
health.modifiers.Add(Modifier.Plus(5f, "+5 health"));
// Prints: Health is 115.
```

Modeling a Consumable Attribute

Let's create a current health value to pair with a max health attribute.

```
var maxHealth = new ModifiableValue<float>(100f);
var health = Value.WithBounds(maxHealth.value, 0f, maxHealth);

health.PropertyChanged += (_, _)
  => Console.WriteLine($"Health is {health.value}/{maxHealth.value}.");
// Prints: Health is 100/100.
health.value -= 10f;
// Prints: Health is 90/100.
maxHealth.modifiers.Add(Modifier.Plus(20f, "+20 level gain"));
// Prints: Health is 90/120.
```

Using an Attribute as a Modifier

In addition to creating modifiers with a static value like `Modifier.Plus(20f)`, one can also create dynamic modifiers based on other values or attributes.

Suppose "max health" is affected by "constitution" like this.

```
var constitution = new ModifiableValue<int>(10);
int level = 10;
// We can project values with some limited LINQ-like extension methods.
var hpAdjustment = constitution
  .Select(con => (float) Math.Round((con - 10f) / 3f) * level);
var maxHealth = new ModifiableValue<float>(100f);

maxHealth.PropertyChanged += (_, _)
  => Console.WriteLine($"Max health is {maxHealth.value}.");
maxHealth.modifiers.Add(Modifier.Plus(hpAdjustment));
// Prints: Max health is 100.
constitution.initial.value = 15;
// Prints: Max health is 120.
```

Note: the attributes are different data types: constitution is an `int`, maxHealth is a `float`.

One might notice that `hpAdjustment` depends on the value of `level`. One would hope that a change to `level` would notify `hpAdjustment` and ultimately `maxHealth`; however, because `level` is an `int` that won't happen. See the Advanced Examples for how to include `level` changes elegantly.

Creating New Modifiers

New modifiers can be created by implementing the `IModifier<T>` interface or by using the convenience methods in `Modifier` like `FromFunc()` shown below. Perhaps one has armor that bestows different affects

depending on the phase of the moon[2].

```
var moonArmor = new ModifiableValue<float>(20f);
var fullMoonModifier
  = Modifier.FromFunc((float x) => DateTime.Now.IsFullMoon() ? 2 * x : x);
moonArmor.modifiers.Add(fullMoonModifier);
```

Ordering Modifiers

The priority of a modifier defines its order. The default priority is `0`. Lower numbers apply earlier; higher numbers apply later. Modifiers of the same priority apply in the order of they were inserted. This also demonstrates how we can clamp a value by creating an ad hoc modifier with a `Func<float,float>`.

```
var maxMana = new ModifiableValue<float>(50f);
var mana = ModifiableValue.FromValue(maxMana);
var manaCost = Modifier.Minus(0f);
mana.modifiers.Add(manaCost);
mana.PropertyChanged += (_, _)
  => Console.WriteLine($"Mana is {mana.value}/{maxMana.value}.");
mana.modifiers.Add(priority: 100,
                   Modifier.FromFunc((float x) => Math.Clamp(x, 0, maxMana.value));
// Prints: Mana is 50/50.
manaCost.value = 1000f;
// Prints: Mana is 0/50.
```

Time Out a Modifier

There are `EnableAfter()` and `DisableAfter()` extension methods for `IModifier<T>`.

```
var armor = new ModifiableValue<int>(10);
var powerUp = Modifier.Plus(5);
armor.modifiers.Add(powerUp);
health.PropertyChanged += (_, _) => Console.WriteLine($"Armor is {armor.value}.");
// Prints: Armor is 15.
powerUp.DisableAfter(TimeSpan.FromSeconds(20f));
// ...
// [Wait 20 seconds.]
// Prints: Armor is 10.
```

Advanced Examples

Synthesizing Multiple Values

In the above constitution example, `level` is an `int` so `hpAdjustment` does not update when it's changed. Instead, we can take another page out of LINQ and use a `Zip()` extension method to synthesize two values. This way a change to either `level` or `constitution` will notify `hpAdjustment` and therefore `maxHealth`.

```
var constitution = new ModifiableValue<int>(10);
var level = new Value<int>(10);
// We can project values, with some limited LINQ-like extension methods.
var hpAdjustment = constitution.Zip(level,
                     (con, lev) => (float) Math.Round((con - 10f) / 3f) * lev);
var maxHealth = new ModifiableValue<float>(100f);

maxHealth.PropertyChanged += (_, _)
```

[2]Unfortunately there is no such extension method `IsFullMoon()` for DateTime by default but one can add it.

```
  => Console.WriteLine($"Max health is {maxHealth.value}.");
maxHealth.modifiers.Add(Modifier.Plus(hpAdjustment));
// Prints: Max health is 100. (unchanged)
constitution.initial.value = 15;
// Prints: Max health is 120.
level.value = 15;
// Prints: Max health is 130.
```

Writing Your Own Attribute Class

One can go far with `IModifiableValue<T>` but one may want to organize modifiers beyond just their order at some point. There are plenty of ways to do this. Seeing as this library's design was informed by a number of sources:

- RPGSystems: Stat System 03: Modifiers by Jacob Penner;
- Character Stats (aka Attributes) System by Kryzarel.

What would their stats classes look like rewritten in this library?

Jacob Penner's Stat Class

Here is an example of what an attribute class might look like if organized like Jacob Penner:

```
public class PennerStat<T> : ModifiableValue<T> {
  public readonly IModifiableValue<T> baseValuePlus = new ModifiableValue<T>();
  public readonly IModifiableValue<T> baseValueTimes = new ModifiableValue<T>(one);
  public readonly IModifiableValue<T> totalValuePlus = new ModifiableValue<T>();
  public readonly IModifiableValue<T> totalValueTimes = new ModifiableValue<T>(one);

  public PennerStat() {
    // value = (baseValue * baseValueTimes + baseValuePlus)
    //         * totalValueTimes + totalValuePlus
    modifiers.Add(100, Modifier.Times(baseValueTimes));
    modifiers.Add(200, Modifier.Plus(baseValuePlus));
    modifiers.Add(300, Modifier.Times(totalValueTimes));
    modifiers.Add(400, Modifier.Plus(totalValuePlus));
  }

  private static T one => Modifier.GetOp<T>().one;
}
```

Kryzarel's Stat Class

Kryzarel's Stat class might look like this:

```
public class KryzarelStat<T> : ModifiableValue<T> {
  public enum Priority {
    Flat = 100,
    PercentAdd = 200,
    PercentTimes = 300
  };

  public readonly IModifiableValue<T> flat = new ModifiableValue<T>();
  public readonly IModifiableValue<T> percentAdd = new ModifiableValue<T>(one);
  public readonly IModifiableValue<T> percentTimes = new ModifiableValue<T>(one);
```

```
  public KryzarelStat() {
    // value = (baseValue + flat) * percentAdd * percentTimes
    modifiers.Add((int) Priority.Flat, Modifier.Plus(flat));
    modifiers.Add((int) Priority.PercentAdd, Modifier.Times(percentAdd));
    modifiers.Add((int) Priority.PercentTimes, Modifier.Times(percentTimes));
  }

  private static T one => Modifier.GetOp<T>().one;
}
```

Some care might need to be taken when adding modifiers to preserve the original's behavior.

```
var stat = new KryzarelStat(30f);
// flat expects plus modifiers (or minus).
stat.flat.modifiers.Add(Modifier.Plus(10f));
// percentAdd expects plus modifiers (or minus).
stat.percentAdd.modifiers.Add(Modifier.Plus(10f));
// percentTimes expects times modifiers.
stat.percentTimes.modifiers.Add(Modifier.Times(1.2f, "+20%"));
```

But that's either a discipline you can adopt or a convenience method you can write. A small price to pay for the flexibility these modifiers provide.

Other Stat Classes

See the Style.cs file for more examples.

And please feel free to share any that you develop with me @shanecelis.

Notes

Dealing with Math in Generics

.NET 7 has generic math operators, which will be a godsend. It will allow us to write methods like this:

```
T Plus<T>(T a, T b) where T : INumber<T> => a + b;
```

which is invalid for prior versions.

This attributes library makes use of this generic math, however, we also want to support `netstandard2.0` because that's what Unity supports. So here's a trick given by this article to allow you to do generic math without .NET 7's `INumber<T>` support.

```
interface IOperator<T> {
  T Plus(T a, T a)
}

struct OpFloat : IOperator<float> {
  public float Plus(float a, float b) => a + b;
}

void SomeProcessing<T, TOperator>(...) where TOperation : struct, IOperator<T> {
  T var1 = ...;
  T var2 = ...;
  T sum = default(TOperator).Plus(var1, var2);  // This is zero cost!
}

void Caller() {
```

```
  SomeProcessing<float, OpFloat>(...);
}
```

Installing

Find the `manifest.json` file in the `Packages` directory in your project and edit it as follows:

```
{
  "dependencies": {
    "com.seawisphunter.roleplay.attributes":
      "https://github.com/shanecelis/SeawispHunter.RolePlay.Attributes.git#unity3d",
    ...
  },
}
```

License

This library is released under the MIT license.

The samples are released under the Unity Asset Store End User License. See sample README for details.

Acknowledgments

This project was inspired and informed by the following sources:

- How to Make an RPG: Stats by Dan Schuller;
- RPGSystems: Stat System 03: Modifiers by Jacob Penner;
- Using the Composite Design Pattern for an RPG Attributes System Daniel Sidhion;
- Character Stats (aka Attributes) System by Kryzarel. Kryzarel has an associated Unity3D Character Stats asset.

I am indebted to each of them for the generosity they showed in writing about the role playing attributes problem, both for their prose and code.

This package was originally generated from unity-package-template.

Author

This project was written by Shane Celis who can often be found on twitter. Other assets are available at seawisphunter.com and the asset store and other repositories on his github.