Introduction

Soap RPG Framework is a lightweight, modular framework that takes a different approach compared to other RPG solutions. While many existing frameworks offer comprehensive, pre-built systems, they often come with steep learning curves and rigid structures that can limit your creative freedom. Soap RPG Framework instead focuses on providing a clean, flexible foundation that you can shape to your exact needs.

Key advantages:

- **Modular Architecture**: Pay just for what you need. Soap RPG Framework serves as the foundation, with additional packages building upon it to extend functionality in a modular way.
- **True Flexibility**: Unlike more rigid frameworks, Soap RPG Framework defines only essential concepts, letting you model any game system without being constrained by pre-made assumptions. The limitations are minimal.
- **Gentle Learning Curve**: Start creating immediately with intuitive, inspector-driven workflows, avoiding the complexity of larger frameworks.
- **100% Inspector-Driven**: Make changes and balance your game without touching code. Game designers can tweak values through ScriptableObjects in real-time, even during play mode, without needing recompilation. This also enables rapid testing and debugging by allowing instant value adjustments.
- **Minimal Lock-in**: The framework's lightweight nature means you're never locked into specific game design patterns.

Whether you're creating a traditional RPG, a roguelike/roguelite, an MMO, or even a game with unique mechanics, Soap RPG Framework adapts to your needs without forcing you into predetermined patterns. It provides essential building blocks for managing attributes, statistics, levels, and classes, along with powerful systems for controlling stat growth through customizable formulas, handling game events, and implementing scaling calculations. This lets you focus on the creative aspects of game development while having precise control over how your game elements evolve and interact.

Vocabulary of Soap RPG Framework

The package is developed around the concept of *entity*, so let's clarify what we mean by this term in the context of Soap RPG Framework. In its most minimal version, an entity is a `GameObject` that has a set of statistics. Optionally, an entity can have attributes, can level up, and can have a class. Let's clarify what we mean by each mentioned term.

Statistics (Stat)

A statistic is a value that quantifies an aspect of the entity. The meaning of this aspect is solely due to the concept it refers to.

Examples

In an RPG, a statistic can be `physical damage`. The concept of physical damage refers the player to the amount of damage inflicted by physical attacks, whether with weapons or without. Other statistics can be `ability power`, `defense`, `speed`, `armor penetration`, `range`, etc.

Attributes

An attribute is a value that can influence the value of one or more statistics. The weight of its influence on the statistics can be variable.

Examples

In an RPG, attributes can be: `strength`, `dexterity`, `intelligence`, `constitution`, etc. Considering the previous example of statistics, `strength` could influence `physical damage`, `dexterity` would increase `speed`, `intelligence` would increase `ability power`, and `constitution` would increase `defense`.

Experience and Level

The entity can gain experience and level up. This functionality is used by the class to express how attributes and statistics grow with levels, for that particular class.

Class

The class is associated with a set of statistics and optionally a set of attributes. The class describes how statistics and attributes vary with levels.

Examples

In RPGs most common classes are: `warrior`, `rogue`, `mage`, `paladin`, and so on. These classes have different attribute values. For example, a warrior will have more `strength` and `constitution` than a mage. The `rogue` might have the highest `dexterity`, etc.

How is Soap RPG Framework organized and how does it work?

Entity

A `GameObject` becomes an entity once the `EntityCore` and `EntityStats` `MonoBehaviour`s (Mono) are added to it. `EntityCore` comes with a built-in `EntityLevel` (plain C# `class`) that manages the experience and the level of the entity.

Stat

A `Stat` is a class that derives from `ScriptableObject` (SO) and represents a statistic in the game. Each statistic has a name (the name given to the SO instance of the created `Stat`), and we can choose whether

to provide it with a maximum and/or minimum value. Additionally, we can define how that statistic grows or is reduced, depending on certain `Attributes`.

 StatSet

A `StatSet` is a class derived from ScriptableObject that defines a collection of `Stat`s. Stat sets can be composed by combining other sub-stat sets, enabling hierarchical organization and easy reuse of statistics among different entities or classes.

 EntityStats

`EntityStats` allows us to configure:

- the base statistics
- the flat modifiers
- the *StatToStat* modifiers
- the percentage modifiers We will see what these modifiers are in the section Understanding Stat Modifier Types.

The base statistics can be *fixed*, or instead derive from a class if the entity has one assigned. If we use the fixed ones, we must also provide a `StatSet`, while if we use those of a class, the class's `StatSet` will be used. If the entity levels up and we want its statistics to grow with levels, we are forced to use a class, as the *fixed* statistics are immutable.

 Class

`Class` derives from SO and represents a game class. Each class has a name, a `GrowthFormula` that defines how the base Max HP grows with levels, a `StatSet`, optionally an `AttributeSet`, and associates each `Stat` of the provided StatSet with a `GrowthFormula` that describes how the statistic varies with levels. Similarly, if an `AttributeSet` is provided, it will be possible to associate a `GrowthFormula` for each `Attribute` present in the set, to describe how the attributes vary with levels.

 EntityClass

`EntityClass` derives from Mono and allows us to assign a `Class` to our entity.

 Attribute

An `Attribute` is a class that derives from SO and represents an attribute in the game. Each attribute has a name and, like statistics, can have a maximum and minimum value.

 AttributeSet

An `AttributeSet` is a class that derives from SO and defines a set of `Attribute`s.

EntityAttributes

Optionally, we can add the Mono `EntityAttributes` to our entity if we want to give it attributes. `EntityAttributes` allows us to specify how many attribute points to provide at each new level. These points can be spent on various attributes to increase their value. For `EntityAttributes` we can configure:

- the base attributes
- the flat modifiers
- the percentage modifiers Similarly to `EntityStats`, we can decide whether the base attributes are *fixed* or if they instead derive from the class associated with `EntityClass`.

Growth Formula

To express how `Stats`, `Attributes`, Max HP, and the experience required to level up vary at each level, we can use instances of `GrowthFormula`. This is a class that derives from SO and allows us to define a mathematical function, or a system of functions, that describe how a value changes as levels increase. We will see in more detail how to define a `GrowthFormula` in the Growth Formulas section.

Scaling Formulas

Scaling formulas provide a flexible way to define how values such as damage, healing, or other effects are calculated based on one or more attributes or stats. They allow you to combine base values (which can be constant or level-dependent) with contributions from various stats and attributes, each weighted by customizable scaling components.

Scaling components

Specify how much a particular stat or attribute influences the final value of the scaling formula, enabling complex and dynamic calculations for abilities, equipment, or other game mechanics. This modular approach lets you easily adjust and extend scaling logic to fit your game's needs.

Game Events

Game events are ScriptableObjects that allow you to implement the Observer pattern in your game. They provide a way to decouple systems by broadcasting notifications when something happens (such as a player jumping, leveling up, or taking damage). Listeners can subscribe to these events and react accordingly, all through inspector-driven workflows. Game events can carry context parameters, making them flexible for a wide range of use cases.

Game Event Generators

Game Event Generators are ScriptableObjects that let you define custom game events with up to four context parameters. They automate the creation of event and listener classes, making it easy to extend

your event system for complex gameplay scenarios. You can specify parameter types and documentation, and generate code and assets directly from the inspector.

How is Soap RPG Framework implemented?

The package is developed following the principles of SOAP (Scriptable Object Architecture Pattern), and has been inspired by the GDC talk of Ryan Hipple. In a nutshell, the main benefits provided by this architecture are:

- **encapsulation**: separation of game logic from data. Game logic code shouldn't mix with data. All data is nicely wrapped withing SO instances
- **game designers friendly**: game designers can make changes and balancements from the inspector without touching the code
- **greater reusability**: most features are `ScriptableObject`s that can be reused by many components
- **greater testability**: being data separated from code, is easier to isolate and fix bugs. Moreover, SO events can be raised with ease at the press of a button from the inspector interface, easing and speeding up debugging even further.

Flexibility of Soap RPG Framework

Although the package is specifically designed for RPG games or games with progression systems, its flexibility allows it to be used in almost any game. As it allows creating attributes like `strength`, `dexterity`, `agility`, etc., and statistics such as `physical attack`, `magic power`, `physical defense`, etc., in RPG, Roguelike, MMO games, etc., nothing prevents it from being used, for example, to implement a firearm. The attributes could be `weight`, `size`, `ergonomics`, etc., and the statistics `recoil`, `handling`, `stability`, `intimidation`, etc. Attributes can influence statistics. A heavier weapon could reduce `handling` but increase `stability`. A larger weapon could reduce `handling` but increase `intimidation`. A more ergonomic weapon could reduce `recoil` and increase `handling`. And so on... The weapon's levels, if present, influence the attributes and statistics, progressively improving them. Classes could represent weapon types (assault rifles, snipers, shotguns, etc.), and each class could have its own set of dedicated attributes and statistics. For example, shotguns could have, in addition to the aforementioned ones, the `barrel length` attribute that influences the `pellet spread` statistic.

Namespace ElectricDrill.SoapRpgFramework

Classes

BoundedValue

Abstract base class for values that can be bounded by minimum and maximum limits. Provides functionality to define optional min/max constraints and clamp values within those bounds.

Class

Represents a character class in the RPG system. It can be used to define how attributes and stats grow over levels.

ClassMenuItems

Provides a menu item in the Unity Editor to create a Class asset.

EntityClass

Component that represents the class of an entity in the game. Implements IClassSource to provide access to the entity's class definition.

EntityCore

The core component for any entity in the game. It is the mandatory base class for all entities, providing essential functionality. Provides easy access to the entity's level, stats, and attributes.

EntityLevel

Represents the level and experience system of an entity in the RPG framework. Handles experience gain, level progression, and experience-based calculations. Implements ILevelable to provide standard leveling functionality.

ExcludeFromDerivedTypePickerAttribute

ExpSource

A MonoBehaviour component that provides a source of experience points. Implements IExpSource and can only be harvested once. Once harvested, subsequent calls to Exp will return 0.

GrowthFormula

Represents a formula to calculate growth values for different levels, up to a maximum level.

GrowthFormulaMenuItems

Provides a menu item in the Unity Editor to create a GrowthFormula asset.

Interfaces

IAttributes

Interface for entities that have an attributes system. Provides access to the entity's attributes component.

IClassSource

Interface for objects that provide a source of a class. In RPG games, a class typically defines the role or profession of an entity, such as a warrior, mage, or rogue.

IExpSource

Interface for objects that can provide experience points. Defines the contract for experience sources that can be harvested.

ILevel

Interface for entities that have a level system. Provides access to the entity's level management component.

ILevelable

Interface for entities that have a leveling system based on experience points. Provides the core functionality for level progression and experience management.

IStatSetSource

Interface for objects that provide a source of stat sets. Defines the contract for accessing a StatSet that contains stat definitions.