

## 0.1 Structure

- **EffectableComponents** are **ActorComponents** that allow for delegation (effects). They have pre-defined places called “**Outlets**” that allow for code modification. Think of **Outlets** like electrical outlets waiting to be plugged into.
  - Let’s use **StatsComponent** as an example. Say we want a Pokémon-style “Adamant” nature (+10% PhA/−10% SpA). One such place for modification is in the function **RecalculateStats**.

```
void UCombatStatsComponent::RecalculateStats(const bool bResetCurrent, const bool bResetHP)
{
    for(const EStatEnum Stat : StatsArray)
    {
        // Cache for delegates
        FCombatStat& TargetStat = GetStatMutable(Stat);
        const float OriginalCurrent = TargetStat.GetCurrentValue();
        const float OriginalPermanent = TargetStat.GetPermanentValue();

        // Resetting this stat?
        bool bReset = bResetCurrent;
        if (Stat == EStatEnum::Health)
        {
            bReset &= bResetHP;
        }

        // Call + execute + call
        RecalculateStatsOutlet.ExecuteBefore(Stat, bReset, OriginalCurrent, OriginalPermanent);
        TargetStat.Update(LevelComponent->GetLevel(), bReset);
        RecalculateStatsOutlet.ExecuteAfter(Stat, bReset, OriginalCurrent, OriginalPermanent);
    }
}
```

- **Outlet arrays** are variables inside of **EffectableComponents**. They hold **Outlets** whose delegates execute when needed.
  - **TODO: Update this!** Let’s use **StatsComponent**’s **AfterRecalculateStatsArray** in our example. In this case, after stats are recalculated (say, on level-up), the base PhA would increase by 10% and the base SpA would decrease by 10% (additively):

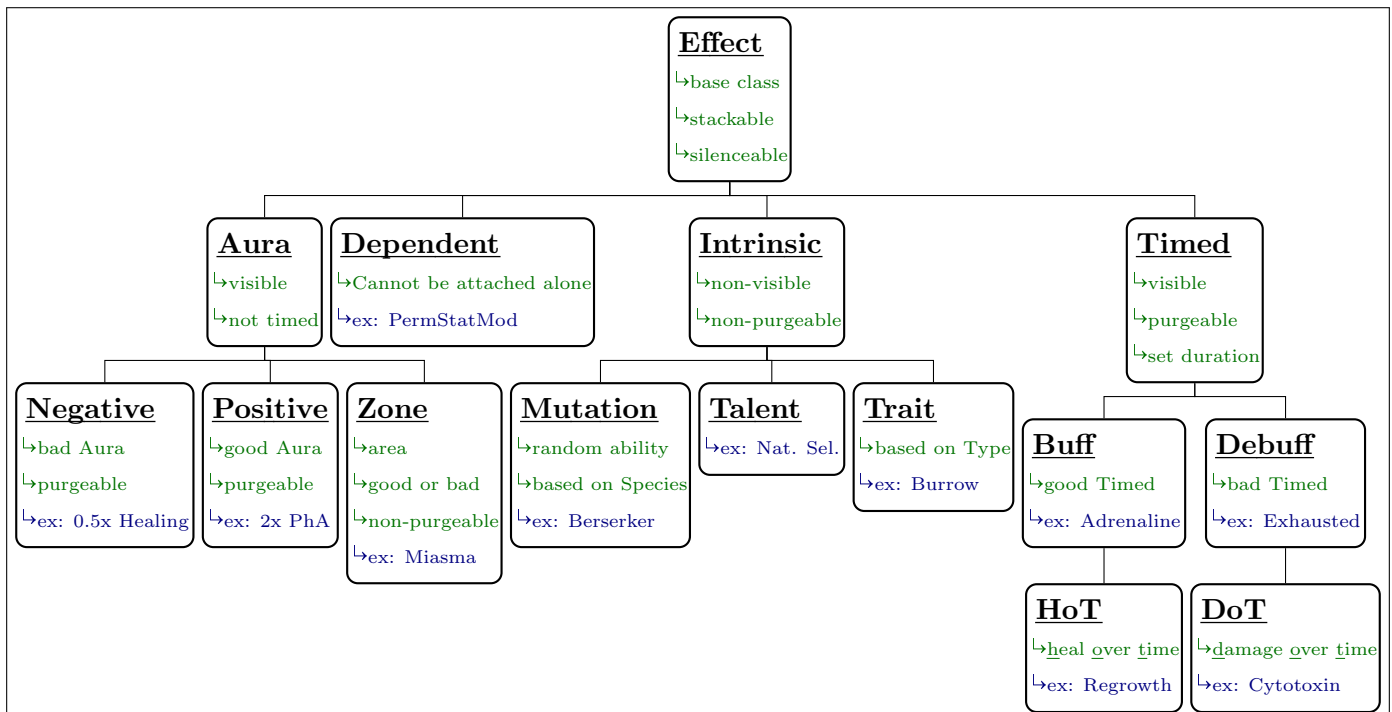
```
// Define "adamant" delegate (+10% PhA/-10% SpA)
UStatsComponent::FRecalculateStatsDelegate AdamantRecalculateDelegate;
AdamantRecalculateDelegate.BindLambda(InFunctor [StatsComponent](FStat* Stat, bool bResetCurrent) -> void
{
    // +10% PhA
    if ( Stat->Name() == StatsComponent->PhysicalAttack.Name())
    {
        Stat->ModifyValue( Modifier: 10, EStatValueType::Permanent, EModificationMode::AddPercentage);
        if (bResetCurrent)
            Stat->ModifyValue( Modifier: 10, EStatValueType::Current, EModificationMode::AddPercentage);
    }

    // -10% SpA
    if ( Stat->Name() == StatsComponent->SpecialAttack.Name())
    {
        Stat->ModifyValue( Modifier: -10, EStatValueType::Permanent, EModificationMode::AddPercentage);
        if (bResetCurrent)
            Stat->ModifyValue( Modifier: -10, EStatValueType::Current, EModificationMode::AddPercentage);
    }
});
StatsComponent->AfterRecalculateStatsArray.Add(AdamantRecalculateDelegate);
```

- **EffectComponents** are **ActorComponents** that plug into **Outlets**. These come in many forms, but an easy example is a **Buff**. **TODO: Describe how this happens with pictures!**

## 0.2 EffectComponent Inheritance

The base classes inherit as:



Some notes:

- Only the base names have been used. That is, the actual names may be `UTimedEffectComponent` instead of simply “Timed”.
- “Purgeable” means it is possible to reduce the stacks of the `UEffectComponent` down to zero (detachment of `UEffectComponent`).
- All `UEffectComponents` are “silenceable”, meaning their effects can be nullified (but not detached or reduced in stacks).
- “Persistent” (meaning that the `UEffectComponent` is not removed upon switching out) should be set on an effect-by-effect basis and not set by the inherited class. For example, some `UNegativeAuraComponents` (such as Pokémon’s Paralysis) may persist upon switching out and others (such as Pokémon’s Confusion) may not.

### 0.3 List of EffectableComponents and Outlets

The following tables show all implemented `EffectableComponents` and their delegate arrays. Note the “base name” indicates existence of “Before” and “After” versions of:

1. the delegate signatures, `FBeforeBaseNameSignature`;
2. the delegate wrappers, `FBeforeBaseNameDelegate`, which are necessary since `TArrays` cannot contain delegates;
3. the private arrays of delegate wrappers,  
`TArray<FBeforeBaseNameOutlet> BeforeDelegates`;
4. a function to execute the arrays, `ExecuteBeforeBaseName`; and
5. `AddBeforeBaseName`, a function to add an `Outlet` to the private array `BeforeDelegates` (which also puts it in the right order based on priority).

Note that the philosophy applies to what is *probable* rather than what is *possible*. Hence the list meant to be practical rather than exhaustive.

Table 1: `Outlets` for `UAffinitiesComponent`

GetUnspentPoints	
► Before	<code>const uint8 OriginalPoints, uint8&amp;ReturnedPoints</code>
► After	<code>const uint8 OriginalPoints, const uint8 ReturnedPoints</code>

Continued on next page

Table 1: Outlets for `UAffinitiesComponent` (Continued)

SetUnspentPoints	
► Before	<code>const uint8 OriginalPoints,</code> <code>const uint8 InputPoints,</code> <code>uint8&amp;SetPoints</code>
► After	<code>const uint8 OriginalPoints,</code> <code>const uint8 InputPoints,</code> <code>const uint8 SetPoints</code>

• • •

Table 2: Outlets for `ULevelComponent`

GetBaseExpYield	
► Before	<code>const float OriginalYield,</code> <code>float&amp;ReturnedYield</code>
► After	<code>const float OriginalYield,</code> <code>const float ReturnedYield</code>

GetCXP	
► Before	<code>const uint32 OriginalCXP,</code> <code>int32&amp;ReturnedCXP</code>
<i>Note:</i>	<code>ReturnedCXP</code> is <code>int32&amp;</code> instead of <code>uint32&amp;</code> for Blueprint compatability.
► After	<code>const uint32 OriginalCXP</code> <code>const int32 ReturnedCXP</code>
<i>Note:</i>	<code>ReturnedCXP</code> is <code>const int32</code> instead of <code>const uint32</code> for Blueprint compatability.

Continued on next page

Table 2: Outlets for ULevelComponent (Continued)

GetExpYield	
► Before	<pre>const float OriginalYield, float&amp;ReturnedYield, const uint16 DefeatedLevel, const uint16 VictoriousLevel</pre>
<i>Note:</i>	“Defeated” and “Victorious” levels are provided for flexibility (e.g., in case you want to yield exp differently based on level difference, although technically you could always back-calculate the level difference based on the equation and <code>OriginalYield</code> ).
► After	<pre>const float OriginalYield, const float ReturnedYield, const uint16 DefeatedLevel, const uint16 VictoriousLevel</pre>
<i>Note:</i>	“Defeated” and “Victorious” levels are provided for symmetry with respect to the <code>Before</code> delegate (since <code>ReturnedValue</code> is already calculated, I can’t think of why you would need them, but you never know!).
GetMaxLevel	
► Before	<pre>const uint16 DefaultMax, int32&amp;AttemptedMax</pre>
<i>Note:</i>	<code>DefaultMax</code> is defined in the code. It should normally be 100, but may change for certain subclasses (e.g., a <code>UBossLevelComponent</code> may have a max of 200 instead). <code>AttemptedMax</code> is <code>int32&amp;</code> instead of <code>uint16&amp;</code> for Blueprint compatability.
► After	<pre>const uint16 DefaultMax const int32 ReturnedMax</pre>
GetMinLevel	
► Before	<pre>const uint16 DefaultMin, int32&amp;AttemptedMin</pre>
<i>Note:</i>	<code>DefaultMin</code> is defined in the code. It should normally be 1, but may change for certain subclasses (e.g., a <code>UEggLevelComponent</code> may have a min of 0 instead for whatever reason). <code>AttemptedMin</code> is <code>int32&amp;</code> instead of <code>uint16&amp;</code> for Blueprint compatability.
► After	<pre>const uint16 DefaultMin const int32 ReturnedMin</pre>
<i>Note:</i>	<code>ReturnedCXP</code> is <code>const int32</code> instead of <code>const uint32</code> for Blueprint compatability.

Continued on next page

Table 2: Outlets for ULevelComponent (Continued)

GetBaseExpYield	
► Before	<code>const float OriginalYield,</code> <code>float&amp;ReturnedYield</code>
► After	<code>const float OriginalYield,</code> <code>const float ReturnedYield</code>
SetBaseExpYield	
► Before	<code>const float OldYield,</code> <code>const float InputYield,</code> <code>float&amp;AttemptedYield</code>
► After	<code>const float OldYield</code> <code>const float InputYield,</code> <code>const float NewYield</code>
<i>Note:</i>	► <code>OldYield</code> is the yield prior to calling <code>SetBaseExpYield</code> , ► <code>InputYield</code> is the original, unmodified input to <code>SetBaseExpYield</code> , ► <code>AttemptedYield</code> is the modified value that will be used to set the base exp yield.
SetCXP	
► Before	<code>const uint32 OldCXP,</code> <code>const int32 InputCXP,</code> <code>int32&amp;AttemptedCXP</code>
<i>Note:</i>	<code>AttemptedCXP</code> is <code>int32&amp;</code> instead of <code>uint32&amp;</code> for Blueprint compatability.
► After	<code>const uint32 OldCXP</code> <code>const int32 InputCXP,</code> <code>const uint32 NewCXP</code>
<i>Note:</i>	<code>UStatsComponent</code> subscribes to this in order to change stats on level change. ► <code>OldCXP</code> is the cumulative experience points prior to calling <code>SetCXP</code> , ► <code>InputCXP</code> is the original, unmodified input to <code>SetCXP</code> , ► <code>AttemptedCXP</code> is the modified value that will be used to set the cumulative experience points.

---

 • • •
 

---

Table 3: Outlets for UStatsComponent

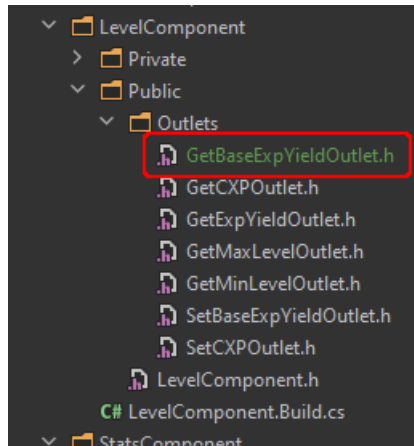
ModifyStat	
► Before	<pre>const EStatEnum TargetStat, const EStatValueType ValueType, const EModificationMode Mode, const float OriginalValue, float&amp;AttemptedValue</pre>
► After	<pre>const EStatEnum TargetStat, const EStatValueType ValueType, const EModificationMode Mode, const float OriginalValue, const float NewValue</pre>
<i>Note:</i>	All “ModifyStat” functions from <code>UStatsComponent</code> (such as <code>ModifyStatsUniformly</code> or <code>RandomizeStats</code> ) go through <code>ModifyStatInternal</code> , which calls this <code>Outlet</code> .
RandomizeStats	
► Before	<pre>const EStatEnum TargetStat, const FStatRandParams OriginalParams, FStatRandParams&amp;ParamsToBeUsed</pre>
► After	<pre>const EStatEnum TargetStat, const FStatRandParams OriginalParams, const FStatRandParams UsedParams</pre>
<i>Note:</i>	The <code>EStatEnum</code> is not the acutal <code>FStat</code> . To get the <code>FStat</code> (such as <code>FHealth</code> ), use <code>UStatsComponent::GetStat(EStatEnum)</code>
RecalculateStats	
► Before	<pre>const EStatEnum TargetStat, const bool bResetCurrent, const float OriginalCurrent, const float OriginalPermanent</pre>
► After	<pre>const EStatEnum TargetStat, const bool bResetCurrent, const float OriginalCurrent, const float OriginalPermanent</pre>

## 0.4 Making Your Own Outlet

As an example, let’s use `GetBaseExpYield`. (You can imagine that this is an important `Outlet` for

tweaking levelling curves.) Here's what to do:

1. **Plan ahead.** I would sincerely recommend you writing down what parameters your **Outlet Before** and **After** delegates take on paper. We go to a few files and it's easy to be inconsistent.
2. **Go to the right directory.** We want to place the **Outlet** inside of **ULevelComponent**, so we'll start with that directory. If yours doesn't contain an "Outlets" directory, create one and place your **Outlet(s)** there.
3. **Copy + paste file.** The easiest way is to copy + paste pre-existing **Outlets**. In this example, we'll copy + paste **SetCXPOutlet.h** and name the new file **GetBaseExpYield.h**.



*Note: this includes both **BeforeGetBaseExpYield** and **AfterGetBaseExpYield** functionality. You don't have to make two different files!*

4. **Replace old name.** Open the new file and you'll still see the base name "SetCXP" everywhere. The easiest way is to do a find+replace "SetCXP" → "GetBaseExpYield". This replaces everything from the **.generated** include to the delegate signatures. If you're curious, you can look more in-depth and replace instances one-by-one.
5. **Declare delegate signatures.** In this case, we want the **Before** delegate signature to take two arguments: the original, unmodified yield and the one that will be returned from the **GetBaseExpYield** function.

```
DECLARE_DYNAMIC_DELEGATE_TwoParams(FBeforeGetBaseExpYieldSignature,  
    const float, OriginalYield, float&, Yield);
```

You should also set the **After** signature in the same manner. *Note: yours might use more than two parameters or different parameter types. Modify accordingly.*

6. **Module API.** Make sure your module API is correct. If not, you'll get mysterious errors about your dll.



```

/**
 * Since delegates can't fit in TArrays, we need to wrap th
 */
USTRUCT(Blueprintable)
struct LEVELCOMPONENT_API FBeforeGetBaseExpYieldDelegate :
{
    GENERATED_BODY()

public:

```

7. **Declare Outlet functions.** In order to be able to call `ExecuteBefore` on your `Outlet`, you need to tell it a few things. The figure below displays a few things in red you should look at:

```

DECLARE_OUTLET_FUNCTIONS_TwoParams(Before, FBeforeGetBaseExpYieldDelegate,
BeforeDelegates, Delegate, const float, float&);

```

- Whether it's a `Before` or `After` type `Outlet`. This affects execution based on priority:

## Priorities

The lower the priority, the farther away it is from execution. If two priorities are tied, the older effect is executed first. Order is set externally by `UEffectsComponent` **TODO: fact check this**. Order:

- Intrinsic **Before** delegates (no **UEffect** affiliated)
- **Before** delegates:
  - \* Priority 1
  - \* Priority 2.a (older)
  - \* Priority 2.b (newer)
  - \* ...
- [Function executes]
- **After** delegates:
  - \* ...
  - \* Priority 2.b (newer)
  - \* Priority 2.a (older)
  - \* Priority 1
- Intrinsic **After** delegates (have the final say)

As an example, consider two delegates: one that says you can't take damage no matter what (call the **UBuff** "Invincible") and another that says damage against you can't be avoided no matter what (call the **UDebuff** "Weakened"). What happens when the target takes damage? Well, it depends on priority:

- They're probably subscribed to the **Before** delegate in **UStatsComponent** called **ModifyStatOutlet** with the target **FStat** being **Health**.
- Note that they're both **Before** delegates.
- Let's say Invincible has Priority 100 and Weakened has Priority 150. The result is the target takes damage because:
  - 1) Invincible first sets the damage to zero.
  - 2) Weakened then sets the damage to no less than its original value.
- If Weakened has lower Priority, the result is flipped and the target takes no damage.

- The parameters you defined in the delegate's signature. I know, I know—anytime you repeat code, you're probably doing something wrong. The biggest issue here is the UHT. The main (but not only) issue is that you can't have **UPROPERTY**s inside macros or the property won't register. If you have a better way of automating this, *tell me!*
- Don't forget the **After** variant's delegates, which should probably be **const**.

8. **Check number of parameters.** I make a point of this because I find it's my most common error. Make sure your declared signature *and* declared `Outlet` function macros have the correct number of params (two in our case). Explicitly, you might need to use `DECLARE_DYNAMIC_DELEGATE_FourParams(...)`.
9. **Declare UPROPERTY.** Inside the `UEffectableComponent` (in this example, `ULevelComponent`), declare the `Outlet` as a variable. Note that it's custom to have this `UPROPERTY` as public and in the "Outlets" category. It's also a good idea to comment the `UPROPERTY` with the parameters.

```
UPROPERTY(BlueprintCallable, BlueprintPure, Category="Level")
float GetBaseExpYield(); // 0 blueprint usages

/**
 * Before Parameters:
 * - [const float] original yield prior to modification
 * - [float&] yield that is being set and then returned
 *
 * After Parameters:
 * - [const float] original yield prior to modification
 * - [const float] yield that is being returned
 */
UPROPERTY(VisibleAnywhere, Category="Level Outlets")
FGetBaseExpYieldOutlet GetBaseExpYieldOutlet;
```

*Note:* I use Rider, so it imports `#includes` automatically. Make sure yours does, too.

10. **Implement.** Now it's time to place your `Outlet` in the appropriate place(s). For our example, it's pretty simple: place it inside of `GetBaseExpYield` in `ULevelComponent`'s `.cpp` file.

```
float ULevelComponent::GetBaseExpYield()
{
    // Get original for delegates
    const float OriginalBaseExpYield = BaseExpYield;

    // Set up the modifiable return value
    float ReturnedBaseExpYield = BaseExpYield;

    // Call before/after delegates
    GetBaseExpYieldOutlet.ExecuteBefore(OriginalBaseExpYield, [&] ReturnedBaseExpYield);
    GetBaseExpYieldOutlet.ExecuteAfter(OriginalBaseExpYield, ReturnedBaseExpYield);

    // Return for use in other functions
    return ReturnedBaseExpYield;
}
```

Note that you might have to do things like cache original values.

11. **A note on complementary delegates.** If you create a `Before Outlet`, you should also create an `After Outlet`. The biggest difference might be the delegate signature (e.g., reference "&" to `const`). An example where this would be necessary is an animation delegate. You only want to fire a "bonus exp" animation *after* the amount of exp has been determined, checked, and is now constant. In some cases, it may not be necessary to have both `Before` and `After` delegates in a function. If you want only one delegate type, or three, or ten, the system is flexible enough to handle it. However, it's recommended to K.I.S.S.

## 0.5 All UEffects

Table 4: All **UBuffComponents** currently implemented and tested.

Buff	Short Description	Implemented via	Priority	Note
Dimensional Shift	See Invulnerable.	See Invulnerable.	See	Inherits directly from <b>UIInvulnerable</b> . It's only around as a Sprit-flavored invulnerability.
Invulnerable	No damage for 1 second	<b>BeforeModifyStat</b>	100	A lot of things can inherit from this, such as <b>UDimensionalShift</b> .

Table 5: All **UDoTComponents** currently implemented and tested.

DoT	Short Description	Implemented via	Priority	Note
Cytotoxin	−1% HP every 1 sec for 3 seconds. 3 stacks max.	<b>TickComponent</b>	50	It's a medium amount, but scales well.

Table 6: All **UHoTComponents** currently implemented and tested.

HoT	Short Description	Implemented via	Priority	Note
Regrowth	+1% HP every 1 sec for 5 seconds. 3 stacks max.	<b>TickComponent</b>	50	It's a medium amount, but scales well.

Table 7: All **UMutationComponents** currently implemented and tested.

Mutation	Short Description	Implemented via	Priority	Note
Berserker Gene	+15% PhA −10% PhD −10% SpD	<b>AfterRecalculateStats</b>	50	It's a little unimaginative, but that's okay.

Table 8: All **UNegativeAuraComponents** currently implemented and tested.

NegativeAura	Short Description	Implemented via	Priority	Note
Wounded Soul	−25% healing.	<b>BeforeModifyStat</b>	100	—

Table 9: All `UPositiveAuraComponents` currently implemented and tested.

PositiveAura	Short Description	Implemented via	Priority	Note
Full Bloom	+20% max HP.	<code>PermStatMod</code>	100	--

## 0.6 Making Your Own Effects

Suppose you want to make your own effect from scratch. `TODO: todo`