

TL; DR

TODO:

0.1 Structure

- **EffectableComponents** are **ActorComponents** that allow for delegation (effects). They have predefined 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**. **TODO: Update picture!**

```
void UStatsComponent::RecalculateStats(const bool bResetCurrent)
{
    for(FStat* Stat : StatsArray)
    {
        ExecuteBeforeRecalculateStats(Stat, bResetCurrent);
        Stat->Update(GetLevel(), bResetCurrent);
        ExecuteAfterRecalculateStats(Stat, bResetCurrent);
    }
}
```

- **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 List of EffectableComponents and Outlet Arrays

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

1. the delegate signature **FBaseNameSignature**;
2. the private before/after arrays of **Outlets**:
TArray<FBaseNameOutlet> BeforeBaseName; and
3. a function for each before/after to execute the arrays: **ExecuteBeforeBaseName (...)**.
4. **AddBeforeBaseName**, a function to add an **Outlet** to the private array **BeforeBaseName** (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: Delegate
Ar-
rays
for
`LevelComponent`

TODO:
Todo

Table 2: Delegate Arrays for `LevelComponent`

<code>GetBaseExpYield</code>		
		<i>Note:</i>
► Before	<code>const float OriginalYield, float& Yield</code>	
► After	<code>const float OriginalYield, const float ReturnedYield</code>	
<code>GetCXP</code>		
		<i>Note:</i>
► Before	<code>const uint32 OriginalCXP, int32& ReturnedCXP</code>	<code>ReturnedCXP</code> is <code>int32&</code> instead of <code>uint32&</code> for Blueprint compatability.
► After	<code>const uint32 OriginalCXP const int32 ReturnedCXP</code>	<code>ReturnedCXP</code> is <code>const int32</code> instead of <code>const uint32</code> for Blueprint compatability.
<code>SetBaseExpYield</code>		
		<i>Note:</i>
► Before	<code>const float OldYield, float& AttemptedYield</code>	
► After	<code>const float OldYield const float NewYield</code>	
<code>SetCXP</code>		
		<i>Note:</i>
► Before	<code>const uint32 OldCXP, int32& AttemptedCXP</code>	<code>AttemptedCXP</code> is <code>int32&</code> instead of <code>uint32&</code> for Blueprint compatability.

Continued on next page

Table 2: Delegate Arrays for `LevelComponent` (Continued)

► After	<code>const uint32 OldCXP</code> <code>const uint32 NewCXP</code>	<code>UStatsComponent</code> subscribes to this in order to change stats on level change.
---------	--	---

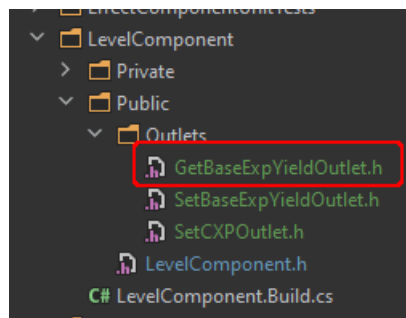
Table 3: Delegate Arrays for `LevelComponent`

TODO:
Todo

0.3 Making Your Own Outlet

As an example, let’s use `BeforeGetBaseExpYield`. (You can imagine that this is an important Outlet for tweaking levelling curves.) Here’s what to do:

1. **Go to the right directory.** We want to place the Outlet inside of `ULevelComponent`, so we’ll start with that directory. If it doesn’t contain an “Outlets” directory, create one and place your Outlet(s) there.
2. **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, but we’ll only talk about the “Before” variant.

3. **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 signature. If you're curious, you can look more in-depth and replace instances one-by-one.

4. **Declare delegate signature.** 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 can also set the “After” signature in the same manner. *Note: yours might use more than two parameters or different parameter types. Modify accordingly.*

5. **Declare Outlet functions.** In order to be able to call `Execute` 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(EDelegateTriggerTiming::Before, FBeforeGetBaseExpYieldDelegate,  
    Delegates, 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 `EffectsComponent`. Order:

- Intrinsic “before” delegates (no Effect attached)
- “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 Outlet in `UStatsComponent` called `BeforeModifyStat` with the target `FStat` being `Health`.
- Note that they’re both “Before” delegates.
- Let’s say Invincible has higher Priority. The result is the target takes damage because:
 - 1) Invincible first sets the damage to zero.
 - 2) Weakened then sets the damage to its original value.
- If Weakened has higher 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!*
 - Repeat for the “After” variant.
6. **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.

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

7. **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
    BeforeGetBaseExpYieldOutlet.Execute(OriginalBaseExpYield, [&]ReturnedBaseExpYield);

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

Note that you might have to do things like cache original values. Also, it may not appear necessary to have both “Before” and “After” Outlets in a function like this. However, it is still recommended to do so.

8. **A note on complementary Outlets.** 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.

0.4 Making Your Own Effects

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