TL; DR

TODO:

0.1 Structure

- `EffectableComponent`**s** are `ActorComponent`s that allow for delegation (effects). They have predefined places called "`Outlet`s" that allow for code modification. Think of `Outlet`s 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 `EffectableComponent`s. They hold `Outlet`s 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);
```

- `EffectComponent`**s** are `ActorComponent`s that plug into `Outlet`s. 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 `EffectableComponent`s and their delegate arrays. Note the "base name" indicates existence of:

1. the delegate signature `FBaseNameSignature`;
2. the private before/after arrays of `Outlet`s:
   `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 Arrays for `AffinitiesComponent`

Base Name	Parameters	Note

Table 2: Delegate Arrays for `LevelComponent`

Base Name	Parameters	Note
BeforeSetCXP	`const uint32 OldCXP`, `int32& AttemptedCXP`	`AttemptedCXP` is `int32&` instead of `uint32&` for Blueprint compatability.
AfterSetCXP	`const uint32 OldCXP` `const uint32 NewCXP`	`UStatsComponent` subscribes to this in order to change stats on level change.

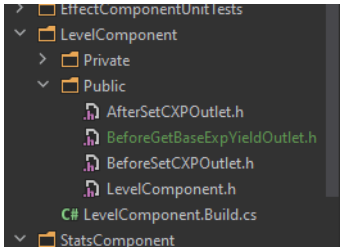
Table 3: Delegate Arrays for `StatsComponent`

Base Name	Parameters	Note

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 go to that directory.
2. **Copy + paste file.** The easiest way is to copy + paste pre-existing Outlets. In this example, we'll copy + paste `AfterSetCXPOutlet.h` and name the new file `BeforeGetBaseExpYield.h`.



3. **Replace old name.** Open the new file and you'll still see the base name "AfterSetCXP" everywhere. The easiest way is to do a find+replace "AfterSetCXP" →

"BeforeGetBaseExpYield". 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 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);
```

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: `TODO: Priority explanation`
- 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. If you have a better way of automating this, *tell me!*

0.4 Making Your Own Effects

Suppose you want to make your own effect from scratch.