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`

Delegate Array Base Name	Parameters	Note
GetUnspentPoints	`int&` Unspent points	
SetUnpentPoints	`int&` Current unspent points	
	`int&` Attempted value being set	

Table 2: Delegate Arrays for `LevelComponent`

Delegate Array Base Name	Parameters	Note
GetBaseExpYield	`int` Unaltered base exp yield	
SetBaseExpYield	`int` Unaltered base exp yield	
	`int&` Attempted value being set	
GetExpYield	`UStatsComponent*` Victorious Monster	
	`float&` Awarded exp	
GetCumulativeExp	`int&` Current CXP	
SetCumulativeExp	`int` Current CXP	All other level/exp setters go through here!
	`int&` Attempted CXP	
AddExp	`int` Current exp	No GetExp (=GetLevel)
	`int&` Added exp	
SetLevel	`int` Current level	No GetLevel
	`int&` Attempted level	
MaxLevel	`int&` The maximum level	This is a getter function only
MinLevel	`int&` The minimum level	This is a getter function only

Table 3: Delegate Arrays for StatsComponent

Delegate Array Base Name	Parameters	Note
RandomizeStats	int& Min base stat	This is the one with four parameters, but is called by all others
	int& Max base stat	
	int& Min base pairs	
	int& Max base pairs	
RecalculateStats	FStat* Each stat in the loop	Rather than make each individual stat an EffectableComponent, you can go stat-by-stat here
	bool If true, reset the current stats to match the newly-calculated permanent stats	
ModifyStat	FStat* The stat being modified	
	float& The value of modification	
	EStatValueType& The value type (e.g., current or permanent)	
	EModificationMode& E.g., additive or multiplicative	

0.3 Making Your Own Effects

Suppose you want to make your own effect from scratch. TODO: Lay this out!