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);
    }
}
```

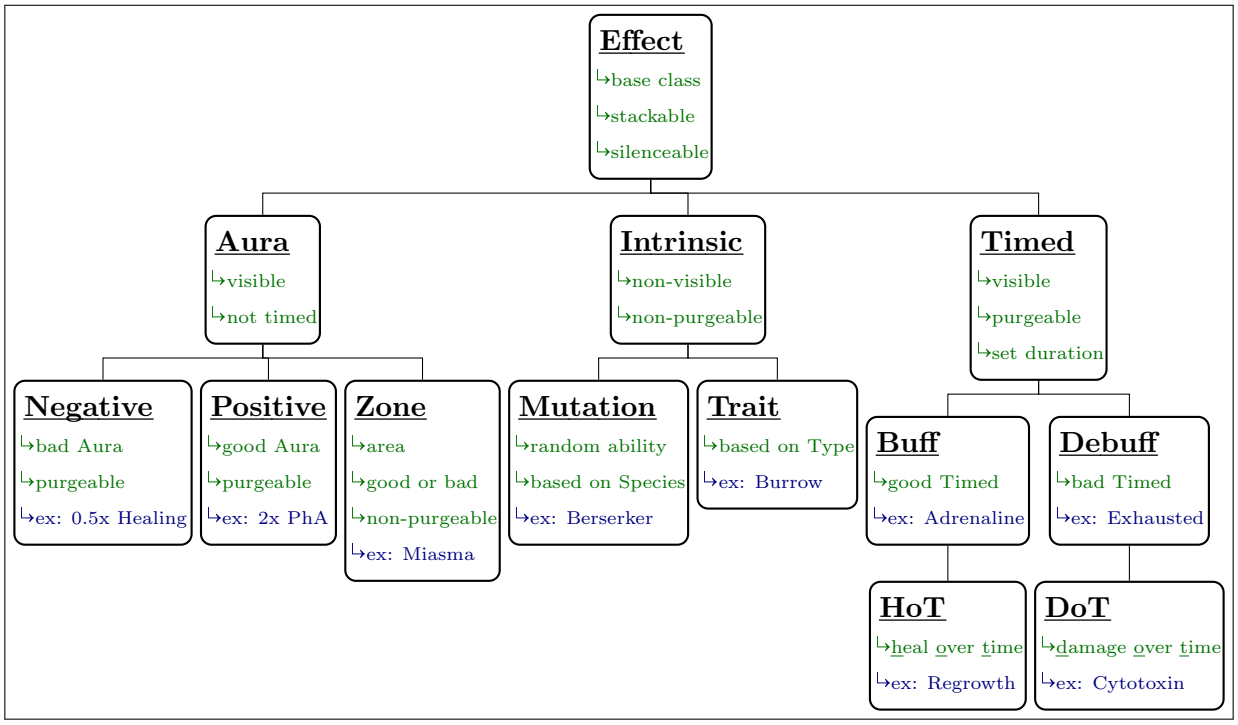
- `Outlet` **arrays** are variables inside of `EffectableComponent`s. They hold `Outlet`s whose delegates execute when needed.
  - `TODO: Update this!` Let's use `StatsComponent`'s `AfterRecalculateStatsArray` in our example. In this case, after stats are recalculated (say, on level-up), the base PhA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

```cpp
// 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 `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 `UEffectComponent`s 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 `UNegativeAuraComponent`s (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 `EffectableComponent`s 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	`const uint8 OriginalPoints,` `uint8& ReturnedPoints`
▸ After	`const uint8 OriginalPoints,` `const uint8 ReturnedPoints`

SetUnspentPoints	

▸ Before	`const uint8 OriginalPoints,`
	`const uint8 InputPoints,`
	`uint8& SetPoints`

▸ After	`const uint8 OriginalPoints,`
	`const uint8 InputPoints,`
	`const uint8 SetPoints`

Table 2: `Outlets` for `ULevelComponent`

GetBaseExpYield	
▸ Before	`const float OriginalYield,` `float& ReturnedYield`
▸ After	`const float OriginalYield,` `const float ReturnedYield`

Continued on next page

Table 2: Outlets for ULevelComponent (Continued)

GetCXP	
▸ Before	const uint32 OriginalCXP, int32& ReturnedCXP
Note:	ReturnedCXP is int32& instead of uint32& for Blueprint compatability.
▸ After	const uint32 OriginalCXP const int32 ReturnedCXP
Note:	ReturnedCXP is const int32 instead of const uint32 for Blueprint compatability.

GetExpYield	
▸ Before	const float OriginalYield, float& ReturnedYield, const uint16 DefeatedLevel, const uint16 VictoriousLevel
Note:	"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 OriginalYield).
▸ After	const float OriginalYied, const float ReturnedYield, const uint16 DefeatedLevel, const uint16 VictoriousLevel
Note:	"Defeated" and "Victorious" levels are provided for symmetry with respect to the Before delegate (since ReturnedValue is already calculated, I can't think of why you would need them, but you never know!).

Continued on next page

Table 2: `Outlets` for `ULevelComponent` (Continued)

GetMaxLevel	
▸ Before	`const uint16 DefaultMax`, `int32& AttemptedMax`
Note:	`DefaultMax` is defined in the code. It should normally be 100, but may change for certain subclasses (e.g., a `UBossLevelComponent` may have a max of 200 instead). `AttemptedMax` is `int32&` instead of `uint16&` for Blueprint compatability.
▸ After	`const uint16 DefaultMax` `const int32 ReturnedMax`

GetMinLevel	
▸ Before	`const uint16 DefaultMin`, `int32& AttemptedMin`
Note:	`DefaultMin` is defined in the code. It should normally be 1, but may change for certain subclasses (e.g., a `UEggLevelComponent` may have a min of 0 instead for whatever reason). `AttemptedMin` is `int32&` instead of `uint16&` for Blueprint compatability.
▸ After	`const uint16 DefaultMin` `const int32 ReturnedMin`
Note:	`ReturnedCXP` is `const int32` instead of `const uint32` for Blueprint compatability.

GetBaseExpYield	
▸ Before	`const float OriginalYield`, `float& ReturnedYield`
▸ After	`const float OriginalYield`, `const float ReturnedYield`

Continued on next page

Table 2: `Outlets` for `ULevelComponent` (Continued)

SetBaseExpYield	
▸ Before	`const float OldYield`, `const float InputYield`, `float& AttemptedYield`

▸ After	`const float OldYield`
	`const float InputYield`,
	`const float NewYield`

Note:	▻ `OldYield` is the yield prior to calling `SetBaseExpYield`, ▻ `InputYield` is the original, unmodified input to `SetBaseExpYield`, ▻ `AttemptedYield` is the modified value that will be used to set the base exp yield.

SetCXP	
▸ Before	`const uint32 OldCXP`, `const int32 InputCXP`, `int32& AttemptedCXP`
Note:	`AttemptedCXP` is `int32&` instead of `uint32&` for Blueprint compatability.

▸ After	`const uint32 OldCXP`
	`const int32 InputCXP`,
	`const uint32 NewCXP`

Note:	`UStatsComponent` subscribes to this in order to change stats on level change. ▻ `OldCXP` is the cumulatie experience points prior to calling `SetCXP`, ▻ `InputCXP` is the original, unmodified input to `SetCXP`, ▻ `AttemptedCXP` is the modified value that will be used to set the cumulative experience points.

Table 3: Outlets for UStatsComponent

ModifyStat	
▸ Before	const EStatEnum TargetStat, const EStatValueType ValueType, const EModificationMode Mode, const float OriginalValue, float& AttemptedValue
▸ After	const EStatEnum TargetStat, const EStatValueType ValueType, const EModificationMode Mode, const float OriginalValue, const float NewValue
Note:	All "ModifyStat" functions from UStatsComponent (such as ModifyStatsUniformly or RandomizeStats) go through ModifyStatInternal, which calls this Outlet.

RandomizeStats	
▸ Before	const EStatEnum TargetStat, const FStatRandParams OriginalParams, FStatRandParams&ParamsToBeUsed
▸ After	const EStatEnum TargetStat, const FStatRandParams OriginalParams, const FStatRandParams UsedParams
Note:	The EStatEnum is not the acutal FStat. To get the FStat (such as FHealth), use UStatsComponent::GetStat(EStatEnum)

RecalculateStats	
▸ Before	const EStatEnum TargetStat, const bool bResetCurrent, const float OriginalCurrent, const float OriginalPermanent
▸ After	const EStatEnum TargetStat, const bool bResetCurrent, const float OriginalCurrent, const float OriginalPermanent

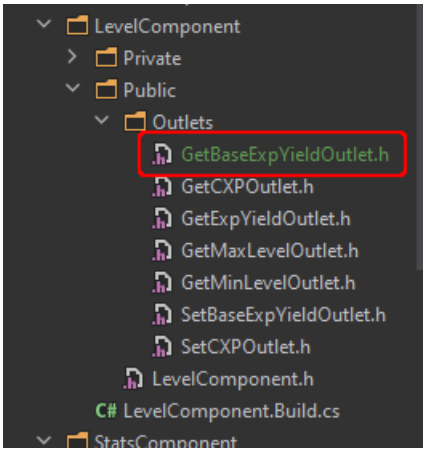
0.4 Making Your Own `Outlet`

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 `Outlet`s. 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!

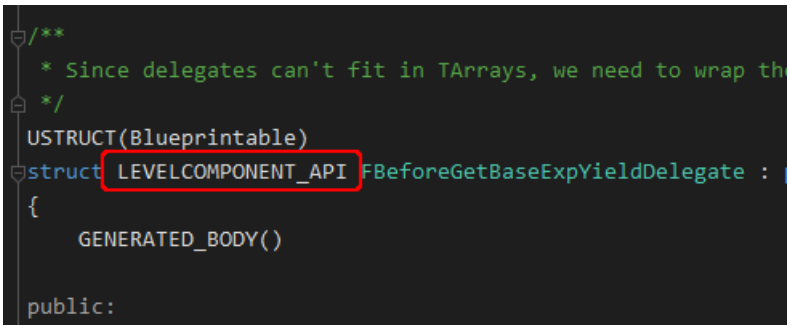
4. **Replace old name.** Open the new file and you'll still see the base name "SetCXP" everywhere. The easiest way is to do a find+replace "SetCXP" → "GetBaseExpYield". This replaces everything from the `.generated` include to the delegate signatures. If you're curious, you can look more in-depth and 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.

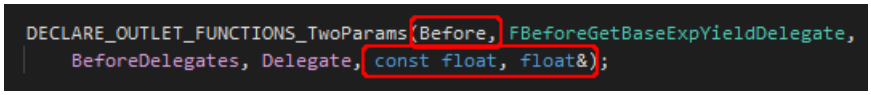
```
DECLARE_DYNAMIC_DELEGATE_TwoParams(FBeforeGetBaseExpYieldSignature,
    const float, OriginalYield, float&, Yield);
```

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.



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:



   - 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.

```
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 `#include`s 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 Making Your Own Effects

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