Multiplayer Ability System Design Document

-Unreal Engine 4

Table of Contents

[Project Brief 2](#_Toc30945049)

[Perquisition Setup 3](#_Toc30945050)

[Main Architecture 4](#_Toc30945051)

[AAttributeComp 4](#_Toc30945052)

[*UAttributePreset:* 4](#_Toc30945053)

[*UStatusPreset:* 4](#_Toc30945054)

[AModifierTargetComp 4](#_Toc30945055)

[AAbilitySetComp 5](#_Toc30945056)

[*UAbilitySet:* 5](#_Toc30945057)

[Logic Flow Example 6](#_Toc30945058)

[Detailed Class Breakdown 7](#_Toc30945059)

[UCustomObject : UObject 7](#_Toc30945060)

[AAttributeComp : AActorComponent 8](#_Toc30945061)

[*UAttributePreset:* 8](#_Toc30945062)

[*UStatusPreset:* 9](#_Toc30945063)

[AModifierTargetComp : AActorComponent 10](#_Toc30945064)

[UModifier : UCustomObject 11](#_Toc30945065)

[AAbilitySetComp : AActorComponent, UAbilitySet : UCustomObject 12](#_Toc30945066)

[UAbility : UCustomObject 13](#_Toc30945067)

[Development Schedule 15](#_Toc30945068)

[Risk Assessment & Mitigation Methods 16](#_Toc30945069)

# Project Brief

An Ability Framework in Unreal Engine 4 that fully supports Networks. The framework is intended for Designers to expand with and create new abilities without worrying about network and basic setups. The system should work on LAN and/or other Online Subsystems (Steam etc.)

# Perquisition Setup

A simple 1v1 pvp GameMode has setup to demonstrate this system, LAN/Steam OSS feature has been implemented and tested. Players can host or join network sessions by either LAN or Steam.



# Main Architecture



Each Actor can choose to implement from 3 core ActorComponents:

* UAttributeComp
* UModifierTargetComp
* UAbilitySetComp

## UAttributeComp

UAttributeComp stores an AAttributePreset : ACustomActor and an AStatusPreset : ACustomActor, which includes the essential variable and status for the parent actor. Actor should implement this component if the actor wants its attributes being affected by other systems, like Abilities or Modifiers.

### *AAttributePreset:*

Stores properties that other system can interact with, they are usually the most common variables in a game, like Health, MaxHealth, Mana, MaxMana, Stamina, etc. But it can also be anything if the parent actor is not the actual player character. The variables are replicated using their own user defined OnRep\_Function(). It also supports broadcast event dispatchers.

### *AStatusPreset:*

Stores status for parent actor, such as sleeping, stun, drunk, silent, etc. Also, respond to any user defined event delegate. E.g, When OnRep\_HealthChanged() is called in paired UAttributePreset, let CheckSleeping() return false.

## UModifierTargetComp

UModifierTargetComp maintains an TArray<AModifier\*>, it handles if the Modifier is being blocked by an existing modifier, or if the Modifier was successfully added, but an existing modifier already exists. (In this case, OnReapply() will be called on existing modifier) Actor should implement this component if the actor needs to be applied with modifiers by other actors

## UAbilitySetComp

UAbilitySetComp is relatively simple, it stores an AAbilitySet : ACustomActor, as well as an InitializeAbilitySet() Funciton. Actor should implement this component if the actor needs to cast ability. The main purpose for this class is:

* Separate logic from ability and actor, perform as an extra abstract layer
* Passing Actor Reference to each ability, so each ability can get the caster

### *AAbilitySet:*

AAbilitySet inherits from ACustomActor, which stores the actual Abilities - TArray<AAbility\*>. It also has an InitializeAbilities(), which passes the Caster to each ability. By doing so, different characters can have different ability sets.

# Logic Flow Example



Following along the red line, Actor A activates an ability, after checked by server, the ability called ApplyModifierTo(AActor Target, AModifier NewModifier, float Duration) function through interface, by doing so, a modifier was applied to Actor B’s UModifierTargetComp, through a bunch of checks, the modifier was successfully applied, then the modifier did corresponding functions and garbage collected.

# Detailed Class Breakdown

## ACustomActor : AActor



As shown above, ACustomActor is the basic object in this framework, which performs as a simple startup point for all actors in this framework. The main reason for using AActor instead of UObject is to use the network supporting feature.

## UAttributeComp : UActorComponent



As mentioned above, UAttributeComp has 2 member properties: AttributeClass and StatusClass, It also performs InitializeAttributePreset() and InitializeStatusPreset() after BeginePlay(), which passes the outer Actor to AAttributePreset and AStatusPreset. This component is replicated.

### *AAttributePreset:*

User define any property in this class, such as Health, Mana, etc. It also contains a list of polymorphic functions:

* AddAttribute ( Template<T> TargetProperty , Template<T> DeltaValue )
* MultiplyAttribute ( Template<T> TargetProperty , Template<T> Multipler )
* SetAttribute ( Template<T> TargetProperty , Template<T> NewValue )

These functions can be called by anything that wants to change these attributes, such as receiving damage, or responses to the Enum\_ModifyMethod in FCostStruct, see below:



### *AStatusPreset:*

AStatusPreset stores a bunch of predefined Booleans, which could be anything similar to Stun, Drunk, Fly, Sleep, Run, Heal, etc. By default, a couple check functions are created, and user can override them freely. If one of them return true (For example:



OnRep\_StunChanged(bool) will be called, allow corresponding logic to be applied to the caster, for example, disable player’s input when stunned. Usually, status is being changed by a AModifier, the reset function should be handled in AModifier’s OnPreExpire() function.

## UModifierTargetComp : UActorComponent



UModifierTargetComp responds to interface message call ApplyModifierTo(Modifier, Duration). It first checks if the modifier type or modifier itself, is explicitly blocked by any existing modifier. Otherwise the modifier will be added to modifier list (call OnReapply() if the modifier is already in the list) Then, the BlockType and BlockList from this modifier will be registered into this component, for future modifier apply check.

After modifier has been added to ModifierList, corresponding LifeCycleHandler(float Duration) will be called to control the life cycle of this modifier, user can manually start tick() by calling StartIntervalThink().

Eventually LifeCycleHandler(float Duration) fires OnPreExpire() –> OnReadyToExpire() call chain when duration ended, given user the last chance to do any logic before the modifier is being removed, then RemoveFromModifierList(Modifier) will be called by AModifierTargetComp, the modifier will be pop out from Modifier List and GCed in next GC cycle (Or manually GCed).

## AModifier : ACustomActor



Once LifeCycleHandler(float Duration) is being called from UModifierTargetComp, OnApplied() will be fired first to do any initialization logic. By default, the modifier will not tick on its own, unless user manually calls StartIntervalThink(float DeltaTime)

AModifier class has a bunch of events to respond to UModifierTargetComp events:

* OnApplied()
* OnReapplied()
* OnBlocked()
* OnStackChanged(UInt8 NewStack)

LifeCycleHandler(float Duration) eventually fires OnPreExpire() -> OnReadyToExpire() call chain.

## UAbilitySetComp : UActorComponent, AAbilitySet : ACustomActor



UAbilitySetComp and AAbilitySet are simple classes, UAbilitySetComp holds an AAbilitySet, and an InitializeAbilitySet() function to pass parent actor into AAbilitySet

AAbilitySet stores a TArray<AAbility\*> as well as an InitializeAbilities() function, which passes both BelongingActor and BelongingAbilitySet to each AAbility class

## AAbility : ACustomActor



When activating an Ability, TryActivate() is fired, results in a function call chain, ActivatePrecheck() is called first, to ensure the Ability is not in cooldown, not in use or disabled by modifiers. Then followed with CostPrecheck() to sync with server and see if caster has enough stats to perform this action. If both passed, CommitAbility() is called, and this is the last chance server can abort this behavior.

Once committed, ApplyCost() is called to modify caster’s stats as ability cost, then the actual lifecycle of performing the ability is started:

* OnActivated() is called immediately when the ability is activated, allowing initialization logic to run here.
* OnTakeEffectNotifyActivated() is called when ability montage reaches TakeEffect Notify, since an AnimMontage can have multiple notifies, so this ability supports combo attack.
* StartIntervalThink(float DeltaTime) is manually called to constantly fire OnIntervalThink() based on DeltaTime.
* OnIntervalThink() is constantly fired by StartIntervalThink every fixing amount of time.
* StartCooldown() is manually called to start cooldown, based on whether user wants the ability to cooldown at the start of this ability or at the end. After cooldown finished, bInCooldown is set to false.
* EndAbility() is manually called to finish this ability, it has to be called otherwise the ability will last forever!
* OnAbilityPreEnd() is called after user manually called EndAbility(), allowing any last logic to perform before the ability stops
* OnAbilityReadyToEnd() is followed by OnAbilityPreEnd(), which sets bCanActivate to true and stops OnIntervalThink().

# Development Schedule

The first 7 weeks will be the actual core features and ground up works development stage, and the second 7 weeks will be mainly working on high level stuffs, playtest, polish and bugfix

|  |  |
| --- | --- |
| Weeks | Schedule |
| 1 | Pitch Doc & Basic Project Setup & Framework Design |
| 2 | Full Design Doc & Framework Design Complete |
| 3 | Create Barebone Classes and Helper Functions |
| 4 | Building Ability System |
| 5 | Building Ability System |
| 6 | Building Modifier System |
| 7 | Building Modifier System |
| 8 | Building Supporting Systems & Works (Heros, DBs, FXs, Levels, GMs, etc.) |
| 9 | Create abilities based on this project |
| 10 | Create abilities based on this project |
| 11 | Playtest |
| 12 | Bugfix & polish |
| 13 | Bugfix & Polish |
| 14 | Polish & RTM |

# Risk Assessment & Mitigation Methods

* Lack of experience on design ability systems, might have cases which I didn’t think of at the beginning
	+ Research on other game’s ability systems and see how they did it, try to abstract the core architecture but with a smaller scope
* Lack of experience on network systems
	+ Scope down to avoid complex variable replication and RPCs
	+ Start with basic variable replication and RPCs
	+ Research on available network examples
* Cannot precisely estimate exactly how much time is needed to make the core system
	+ The rest 7 weeks are less packed, so more timeframe is provided if something unexpected happens