Quite simply, all movement modes we create have two primary logical needs:

1. Requirements that must be met to **enter** a movement mode.
2. Requirements that must fail in order to **exit** a movement mode.

For example, let's take a look at a movement mode we have worked with before:

1. Try to enter the wall run movement mode by checking for suitable surfaces. If found, ensure the character meets various status requirements, such as not being stunned, being alive, and that they are currently in the MOVE_Falling movement mode so they can't wall run from the ground.
2. While wall running, keep checking the wall run requirements until one of them fails. Then, exit wall running into a suitable alternative movement mode, such as MOVE_Falling again.

As you can see, the CanWallRun() method you would create for trying to start wall running can be **reused** here, linking the entry and exit conditions intrinsically. Code reuse is great! You only need to update one little function to modify your logic. Imagine having to track down multiple functions to add one requirement. You might even miss one and create annoying bugs!

In fact, this pattern exists for almost everything in game development:

1. Check **Requirements** for applying an effect and **Blockers** that prevent activation.
2. If successful, apply the effect's various **conditions**.
3. Check Requirements for maintaining the effect or the sudden appearance of a Blocker while active.
4. When a Requirement fails or a Blocker appears, deactivate/remove the effect and remove its conditions.

Does this sound familiar? Well, it's basically the concept that most **Ability Systems** are based upon. If you take a look at GAS, what does it ultimately do? Pretty much exactly what we have above, but abstracted out into arbitrary **Abilities (Actions)**, **Cues**, and **Effects**!

Do you want to add a Stunned condition to an opponent after pressing a button? Well, we first need to see if you are allowed to cast that **Ability** by checking for **Requirements**, such as having enough mana. Then, check for **Blockers**, which are usually **Gameplay Tags** detailing stuff like Status. tunned or Status.CastingAbility. In this case, these would block you from casting this ability.
If you pass these checks, you will then cast the ability, which will play a **Cue** (typically, a cosmetic effect like VFX, etc), and try to apply the stun **Effect** to your opponent. The effect might have its own requirements and blockers to check if it actually can be applied to the opponent, like the opponent having a stun prevention buff. If it succeeds, it will then apply the Status.Stunned tag for a certain duration or however you set it up.

Of course, there is far more complexity to every ability system, especially Unreal's GAS, but these are the basics. All it does is provide a standard framework that facilitates the most common needs for game development. That is what makes it so powerful, especially when combined with its advanced networking capabilities.

However, it is also plain to see that you can **build your own ability system** following these basic principles. You just need to create reusable components that facilitate the logic we discussed above. Abstracting your mechanics out into their respective parts, such as the overarching **Action/Ability** (trying to attack), the **Requirements** and **Blockers** that determine its success, the **Cues** (cosmetics) that should play upon success, and the **Effect/Conditions** that are applied to the target (damage, debuffs, etc). Can an effect remove another effect? If so, you need effect IDs like Effect.Debuff.Movement. **You can do SO MUCH with Gameplay Tags**! Then, you just need to consider how you manage these effects, such as whether they are infinite or should have a duration, and clean them up appropriately.

This is the **key to good gameplay architecture**. Your ability as a gameplay programmer to take a design concept and abstract it out into its various pieces will allow you to craft efficient, modular, reusable, and maintainable code. However, you must keep in mind that a system like this does **add a learning curve**. GAS can be scary when you first come across it for this exact reason. It is a massively abstract system that has been built to facilitate just about any game. This has the downside of introducing immense complexity, possibly to the detriment of your team.

But, this then introduces the opportunity to create more intuitive tools built on top of your complex backend to allow everyone to very easily achieve their goals while maintaining your architectural beauty. Software engineering is an endless balancing act between trying to show off your programming skills and actually creating usable tools.