Unreal Engine 5 - Lesson 3 - Gameplay Framework Introduction

—

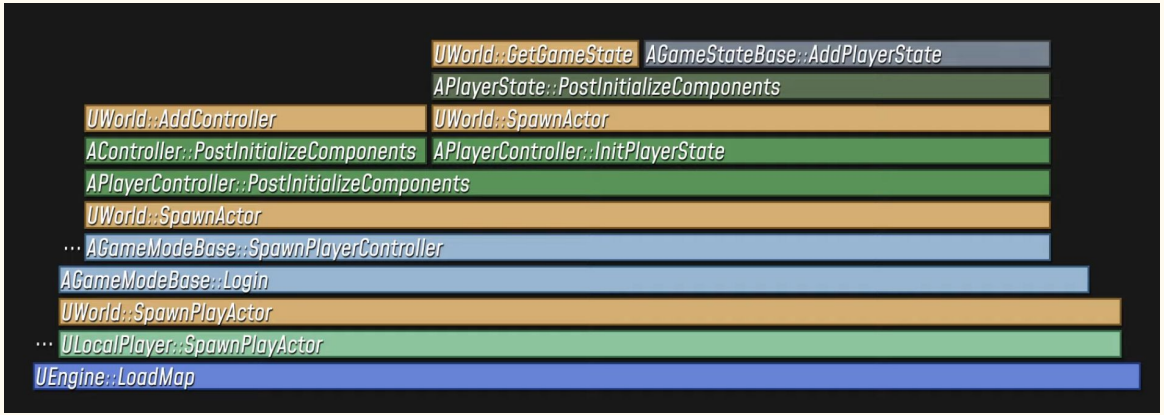
Nicolas Serf - Gameplay Programmer - Wolcen Studio
serf.nicolas@gmail.com

Summary

- What is the Gameplay Framework ?
- Is it mandatory to use it ?
- Lifecycle of the Engine
- Unreal Engine game loop
  - PreInit phase
  - Init phase
  - Start phase
- Final thoughts
- Gameplay Framework main classes
  - UObject
  - Actor
  - Pawn
  - Character
  - Controller
  - Component
  - Game Instance
  - Game Mode
  - Game State
  - Player State

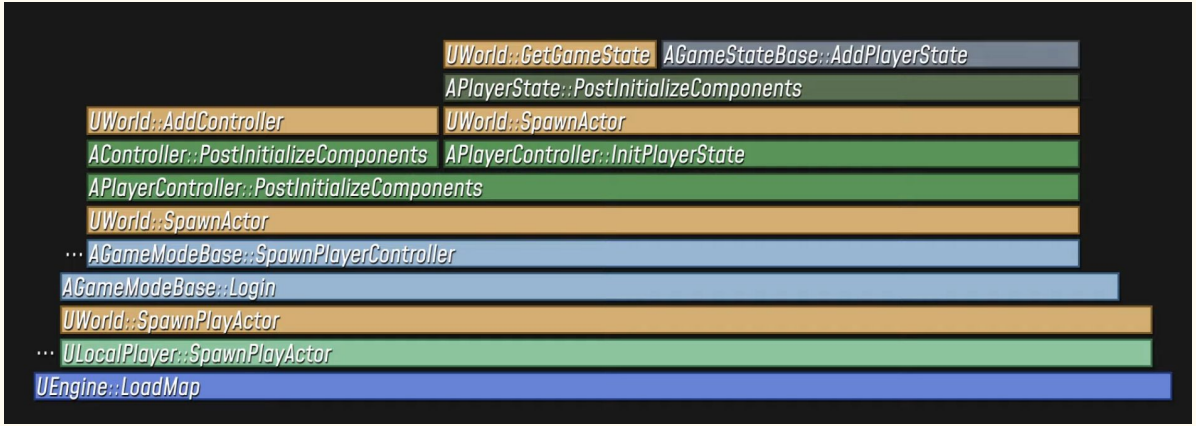
What is the Gameplay Framework ?

- The core framework that is **running** the **whole environment** of the engine
- It **offers** a **standardize hierarchy of class** that are **ready out of the box** and **fully integrating with the initialization** and **runtime** of the engine by default
- It is an **open box** and it is possible to **override nearly anything from it**
- A complexity which is **hidden by default** but still need understanding on how things are related all together
- It basically offers all the core system of any game
    - 3C
    - Networking
    - Animation
    - User interfaces
    - Inputs



Is it mandatory to use it ?

- As we'll be able to see it through the course of that lesson, Gameplay Framework is **deeply responsible** for **initializing** the whole engine **environment**.
- It is not **strictly mandatory** to use the Gameplay Framework in itself, it would be possible to **rewrite the entire loading process** that Unreal is doing for us and rewriting the class hierarchy from scratch.
- Keep in mind that even if you **don't want to use some part** of the Gameplay Framework because it is too complex... just **leave it as it is**. The purpose of that framework is to be **ready out of the box** but **completely open for any modification**.
- To conclude, **in my opinion**, if you don't **want to use** the Gameplay Framework at all, you should consider **moving to another engine**.
    - The gameplay framework is one of the reason that Unreal is a top engine for game development.
    - It is so **deeply rooted** in the development process of the engine
    - Have been develop for years by an army of developer and proof test by **Unreal production** (Fortnite, Paragon, etc...)



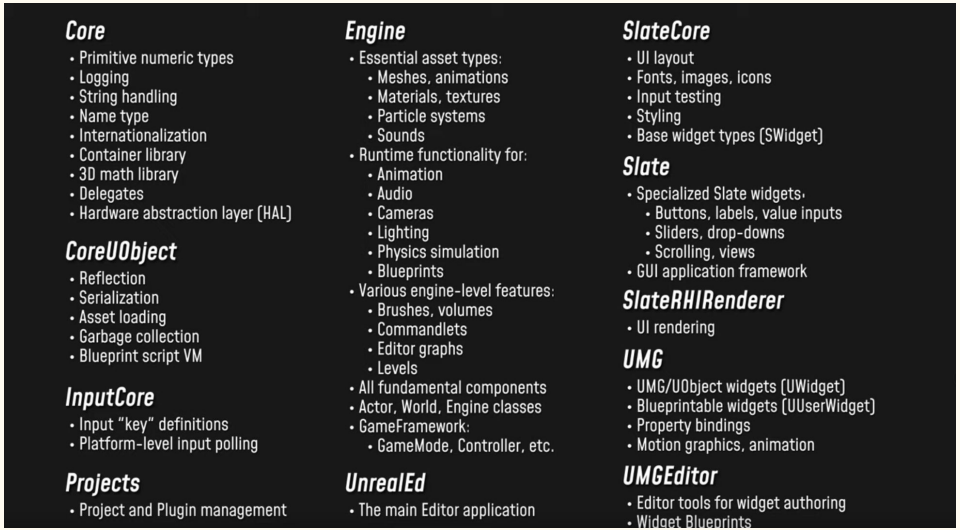
Engine Loop

- If we skip details about the main code of the game loop, we can highlight **3 distinct loading phase**
  - **PreInit** and **module loading**
  - **Init** and **Engine Initialization**
  - **Start** and **World creation**

```cpp
#include "LaunchEngineLoop.h"

FEngineLoop GEngineLoop;
bool GIsRequestingExit = false;

int32 GuardedMain(const TCHAR* CmdLine)
{
    // Run early initialization, load engine modules
    int32 ErrorLevel = GEngineLoop.PreInit(CmdLine);
    if (ErrorLevel != 0 || GIsRequestingExit)
    {
        return ErrorLevel;
    }

    // Create and initialize a UEngine, run late initialization, start the game
    ErrorLevel = GEngineLoop.Init();

    // Every frame: kick off rendering, tick the engine, update RHI
    while (!GIsRequestingExit)
    {
        GEngineLoop.Tick();
    }

    // Run cleanup and exit when requested
    GEngineLoop.Exit();
    return ErrorLevel;
}
```

PreInit - Module concept

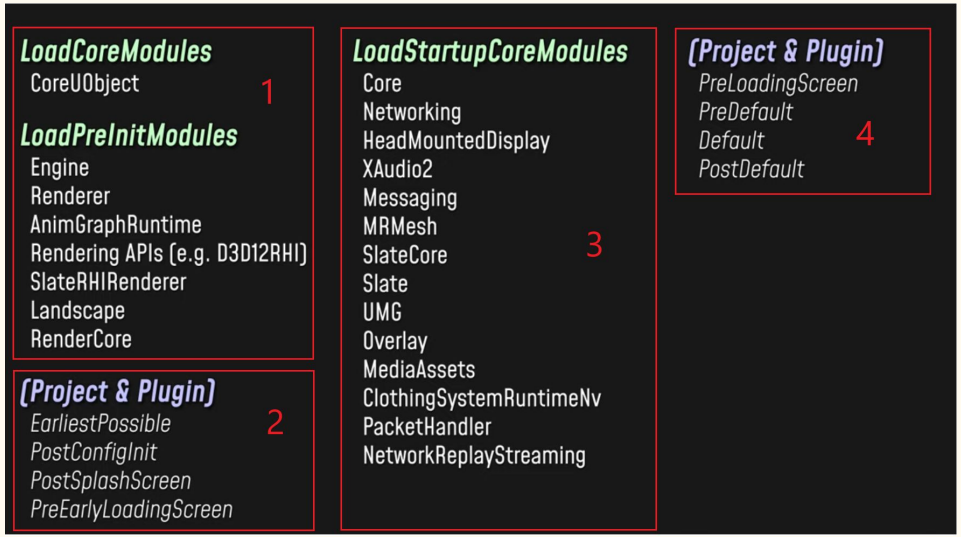
- The engine is built on **module concept**
  - When you create a Cpp project or create a plugin containing one or several source code, you define **source modules.**
  - Inside source module, we can define **ELoadingPhase** to specify when the module should be loaded
  - The engine follow that rule and is divided into **several module**
    - Some are **mandatory** and loaded first
    - Some are **optional** and only loaded on certain platforms



PreInit - Loading phase

- 1 : First of all, when the engine loop starts init phase, it loads some **low level engine modules**
  - **Essentials systems** are initialized
  - **Essentials types** are defines
- 2 : At this point, it is possible to inject in this early stage some module loading from our project or plugins thanks to **ELoadingPhase.**
- 3 : After that, the bulk of **higher level modules of the engine** are loaded.
- 4 : This is the default point where are **project & plugins modules are loaded**. This is where our cpp code is injected into the engine

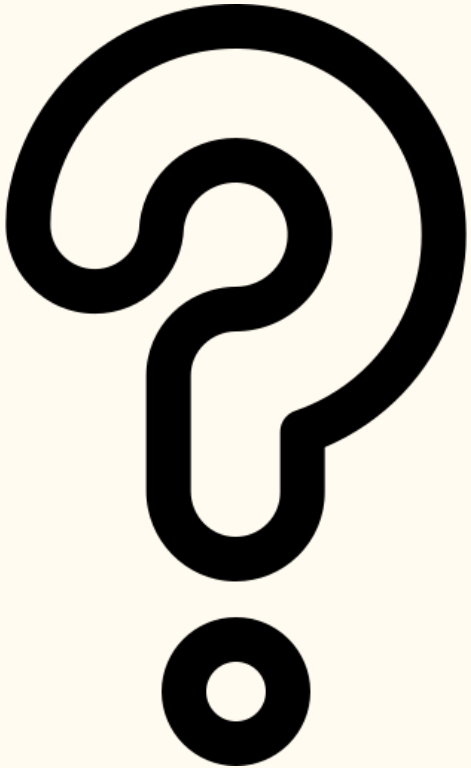
It is important to understand that before step 4, we are just having a **generic engine instance** without any stuff coming from us.



PreInit - Loading phase - What is happening ?

- When our modules are loaded, the engine will **register any UClass** that are defined into that module.
  - That allow to **make reflection aware** of thus classes
  - It constructs the **CDO** (Class Default Object). The CDO is a record of the **default state of our class**. It serves as a **prototype** for further inheritance.
  - It then **calls the constructor** of that class and passes the CDO of the parent class as a template
    - This is the reason why **you must not write any gameplay-related code** in constructors
    - Constructor is used to e**stablishing universal details of the class**, components that are created by it, etc... but for sure nothing related to **particular instance** of that class

- When all classes are registered, the engine calls your module's **StartupModule** function where you can write some custom calls



Init - What is UEngine

- Engine is a core module which is why we'll refer to it as a capital letter at start.
- Engine contains source code and a **UEngine** class which is inherited from **UEditorEngine** and **UGameEngine**
- Engine will check the **engine config file** in order to know **which game engine class** should be used
  - It then **create** the object of that UEngine and set it as the **global variable**
  - This one is then use in various places, and for example displaying a message on screen
- It then inits the **GEngine**, and **broadcast** and **informs** other modules that may need to **initialize** after **GEngine** init
- **GEngine** has a lot of **responsibility**, but its main purpose **Map Loading**

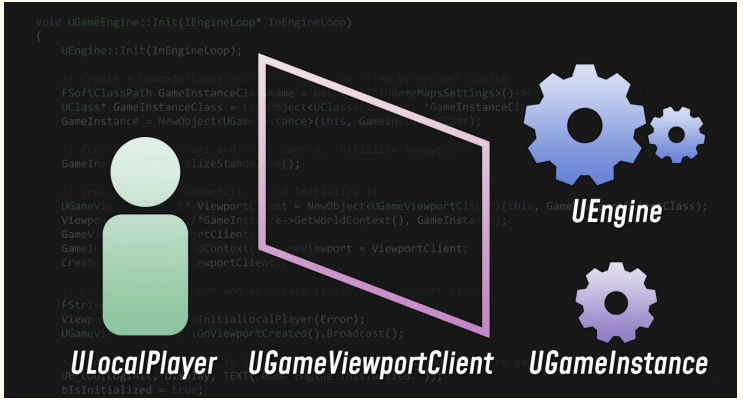
```
int32 FEngineLoop::Init()
{
    // Load the UGameEngine class that's specified in the Engine config file
    FString GameEngineClassName;
    GConfig->GetString(TEXT("/Script/Engine.Engine"), TEXT("GameEngine"), GameEngineClassName, GEngineIni);
    EngineClass = StaticLoadClass(UGameEngine::StaticClass(), nullptr, *GameEngineClassName);

    // Create a new UGameEngine and enshrine it as the global UEngine object
    GEngine = NewObject<UEngine>(GetTransientPackage(), EngineClass);
    check(GEngine);

    // Initialize the engine: this creates UGameInstance and UGameViewportClient
    GEngine->ParseCommandline();
    GEngine->Init(this);
    FCoreDelegates::OnPostEngineInit.Broadcast();

    // Initialize any late-loaded modules
    IProjectManager::Get().LoadModulesForProject(ELoadingPhase::PostEngineInit);
    IPluginManager::Get().LoadModulesForEnabledPlugins(ELoadingPhase::PostEngineInit);

    // Start the game: typically this loads the default map
    GEngine->Start();
    GIsRunning = true;
    FCoreDelegates::OnFEngineLoopInitComplete.Broadcast();
    return 0;
}
```

```
if(GEngine)
    GEngine->AddOnScreenDebugMessage(-1, 15.0f, FColor::Yellow, TEXT("Some debug message!"));
```

Init - Engine Initialization

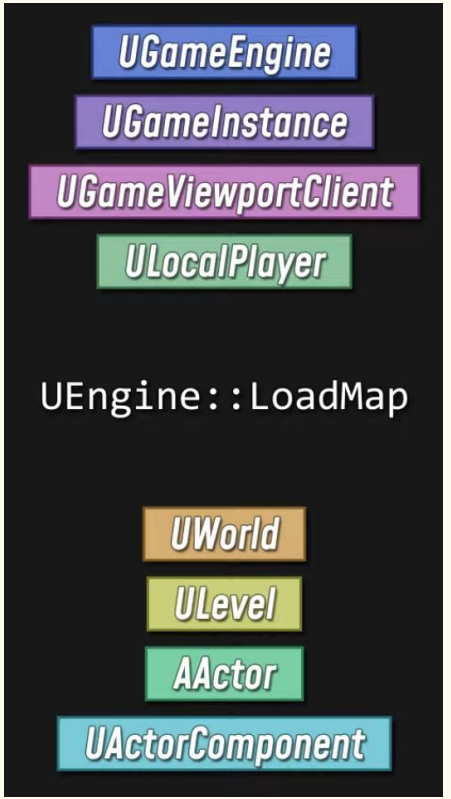
- **Before loading a map**, the Engine **initialize** itself by creating a few important objects
  - **UGameInstance**
    - Was added a couple of years ago
    - It is **spawn** off the UEngine class
    - Allow to **handle more project specific loading** and stuff that was manage by the engine itself before.
  - **UGameViewportClient**
    - It can be related to the **screen itself** of the client.
    - High level interface responsible for **rendering**, **audio** and **input system**.
    - It represent the **interface** between the **user** and the **engine**
  - **ULocalPlayer**
    - It can be related to the **player seating in front of the screen**, meaning that it is its **local** representation in the game (Multiplayer related)

```
void UGameEngine::Init(IEngineLoop* InEngineLoop)
{
    UEngine::Init(InEngineLoop);

    // Create a UGameInstance using the class specified by project config
    FSoftClassPath GameInstanceClassName = GetDefault<UGameMapsSettings>()->GameInstanceClass;
    UClass* GameInstanceClass = LoadObject<UClass>(nullptr, *GameInstanceClassName.ToString());
    GameInstance = NewObject<UGameInstance>(this, GameInstanceClass);

    // Create FWorldContext and dummy UWorld; initialize subsystems
    GameInstance->InitializeStandalone();

    // Create a UGameViewportClient and initialize it
    UGameViewportClient* ViewportClient = NewObject<UGameViewportClient>(this, GameViewportClientClass);
    ViewportClient->Init(*GameInstance->GetWorldContext(), GameInstance);
    GameViewport = ViewportClient;
    GameInstance->GetWorldContext()->GameViewport = ViewportClient;
    CreateGameViewport(ViewportClient);

    // Create a ULocalPlayer and associate it with the viewport client
    FString Error;
    ViewportClient->SetupInitialLocalPlayer(Error);
    UGameViewportClient::OnViewportCreated().Broadcast();

    // Done; now we're ready to load PostEngineInit modules and start the game
    UE_LOG(LogInit, Display, TEXT("Game Engine Initialized."));
    bIsInitialized = true;
```



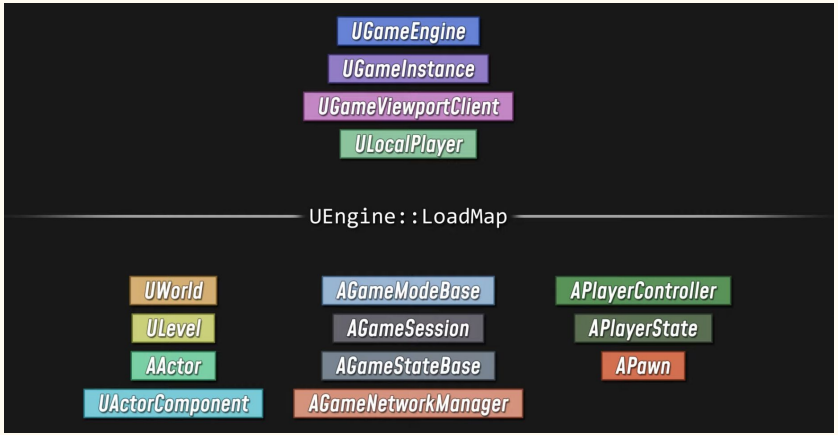
Start - Starting point for World Creation

- When Start is called, it is the moment when our world will be created
  - After LoadMap, we have our **UWorld**, containing our **ULevel** which is containing all our world stuff
- This is where our **Gameplay Frawework** comes to live, with all actor composing our world
  - **AGameModeBase**
  - **AGameSession**
  - **AGameStateBase**
  - **AGameNetworkManager**
  - **APlayerController**
  - **APlayerState**
  - **APAwn**
  - **AActor**
  - **UActorComponent**



Start - Lifetime

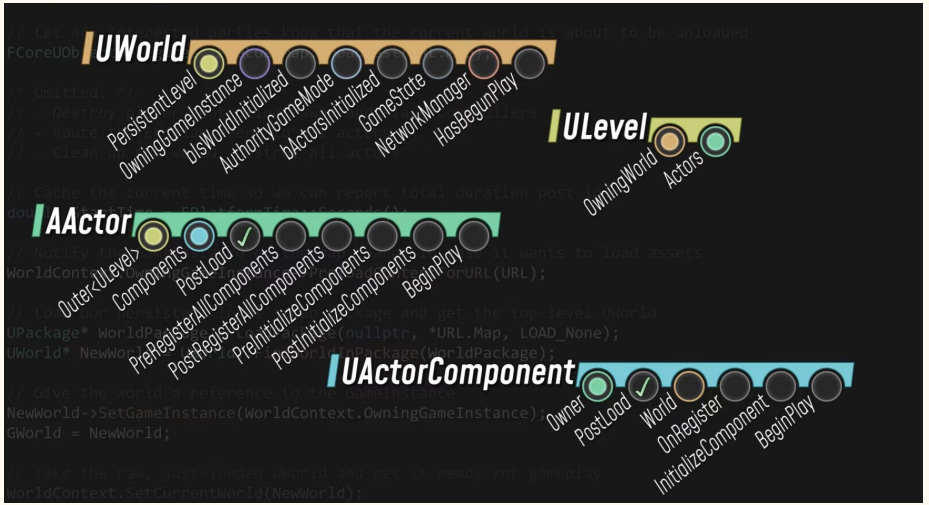
- There is various **UObject** and **Actor** that are **spawned** when a level is loaded, but there is a main difference : the **lifetime**
- In high level there is 2 different lifetime to think about
    - **Before** map is loaded : **Engine objects**
        - This objects are tied to lifetime of the **process**
    - **After** map is loaded : **Game Objects**
        - This objects are tied to lifetime of the **map**
        - **Seamless travel** is a thing that engine support but setup needs to be done
            - Allows to **transition** to a **map** and **keep** certain actors **intact**



Start - Loading the map - Setting up WorldContext

- Loading process of a map is **quite complex** but let's pin down some **details**
  - As a parameter of the LoadMap, there is the **FWorldContext**
    - It is created by the game instance at **Engine initialization** which is persistent and **keep track of which world is loaded up**
  - Main purpose of the **LoadMap** is to have a **UWorld**
    - When working in the editor, there is already a **UWorld** with 1 or more **ULevel** loaded into memory
    - When you **save** the world, **UWorld**, **ULevel** and **all actors** placed are serialized into a **UPackage** and create an asset **.umap**
- During the LoadMap call, the Engine find that **UPackage** and loads it back into memory
- We ensure to reference our newly created **UWorld** in the **FWorldContext,** add it to the root, to prevent it from being garbage collected and **initialize** it for example **physics**, **navigation**, etc...
- Finally, it create and set the **UGameMode**

```
bool UEngine::LoadMap(FWorldContext& WorldContext, FURL URL, class UPendingNetGame* Pending, FString& Error)
```

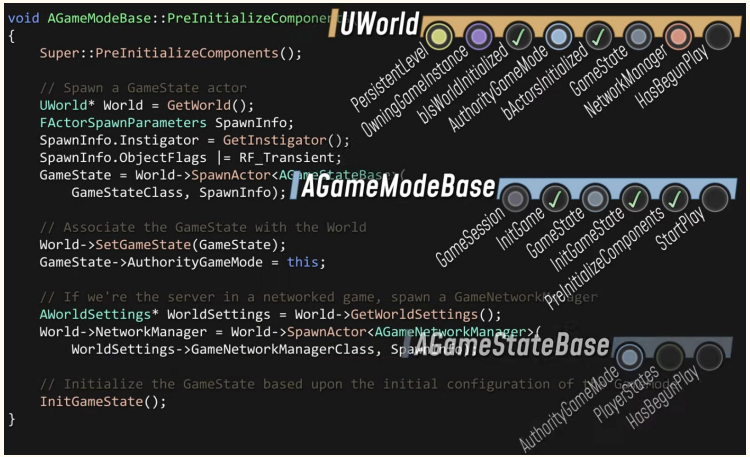


Start - Loading the map - Initialize actors

- This phase allows to actually **bring the world**
- The UWorld iterates over multiple loops
- 1st : It registers all actor component within the world and has 3 important steps
    - Gives it a reference to the **UWorld**
    - Calls **OnRegister**, it can be used for early initialization
    - If it is a **primitive component** (ie : **Renderable**) it is initialize with a **FScene** which is a render thread's version of the **UWorld**
- 2nd : It calls the **GameMode's InitGame** function.
    - It internally spawns a **AGameSession** actor
- 3rd : it iterate over all **ULevel**
    - It initialize all actors in it through 2 passes
        - **PreInitializeComponents** : It happens **after components has been registered**, but **before their initialization**
        - **PostInitializeComponents** : It happens **after initialization.** Here it is fully initialize and ready
    - **AGameMode** being an actor as the other, it follows this PreInitialize and spawns in it the **AGameStateBase** and the **AGameNetworkManager**.

```cpp
void UWorld::InitializeActorsForPlay(const FURL& InURL, bool bResetTime, FRegisterComponentContext* Context)
{
    // Register all actor components in the persistent level only. Note that in actual fact, actors loaded
    // from sublevels have their components registered during FlushLevelStreaming, just before this point.
    PersistentLevel->UpdateLevelComponents(false, Context)

    // Set bActorsInitialized so that future actors will be initialized on spawn
    bActorsInitialized = true;

    // Initialize the game mode: this spawns an AGameSession
    AuthorityGameMode->InitGame(FPaths::GetBaseFilename(InURL.Map), ParseOptions(InURL), Error);

    // Route InitializeComponents (and PreInit/PostInit) to all actors, level-by-level
    for (ULevel* Level : Levels)
    {
        Level->RouteActorInitialize();
    }

    // Fire both a member delegate and a global/static delegate
    FActorsInitializedParams OnActorInitParams(this, bResetTime);
    OnActorsInitialized.Broadcast(OnActorInitParams);
    FWorldDelegates::OnWorldInitializedActors.Broadcast(OnActorInitParams);
}
```

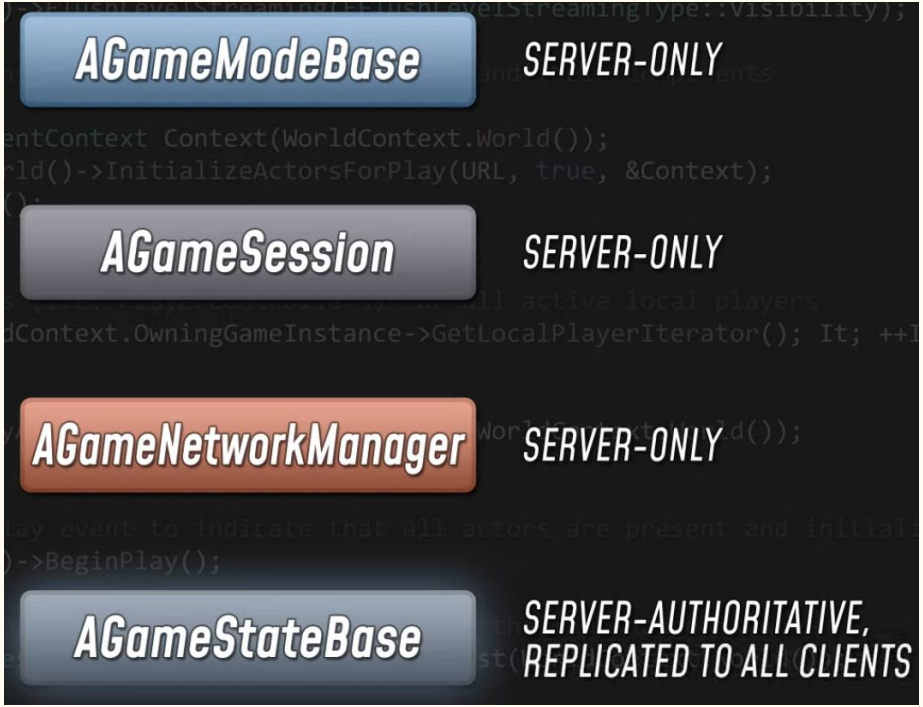
```cpp
void AGameModeBase::PreInitializeComponents()
{
    Super::PreInitializeComponents();

    // Spawn a GameState actor
    UWorld* World = GetWorld();
    FActorSpawnParameters SpawnInfo;
    SpawnInfo.Instigator = GetInstigator();
    SpawnInfo.ObjectFlags |= RF_Transient;
    GameState = World->SpawnActor<AGameStateBase>(
        GameStateClass, SpawnInfo);

    // Associate the GameState with the World
    World->SetGameState(GameState);
    GameState->AuthorityGameMode = this;

    // If we're the server in a networked game, spawn a GameNetworkManager
    AWorldSettings* WorldSettings = World->GetWorldSettings();
    World->NetworkManager = World->SpawnActor<AGameNetworkManager>(
        WorldSettings->GameNetworkManagerClass, SpawnInfo);

    // Initialize the GameState based upon the initial configuration of the level
    InitGameState();
}
```



Recap of Game framework actors representing the game

- We'll go through it in more **details later** on but let's recap what has been **created** by the Engine at loading stage which is **managing overall the state of the game**

- **AGameModeBase**
  - Defines the **rules** of the game, and spawns most of the core gameplay actors
  - **Ultimate authority** of what happens during gameplay
  - Only exists on the **server**
- **AGameSession** :
  - **Approve** login requests
  - Serve as an **interface** for online service (Steam, PSN, etc...)
  - **Only** exists on the **server**
- **AGameNetworkManager** :
  - Configure things like **cheat detection**, **movement prediction**, etc...
  - **Only** exists on the **server**
- **AGameStateBase** :
  - Created on the server and only the server has the authority to change it
  - Store **data** related to **state of the game**, that all players needs to know about
  - **Replicated** on **all clients**

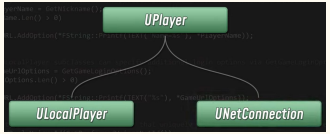


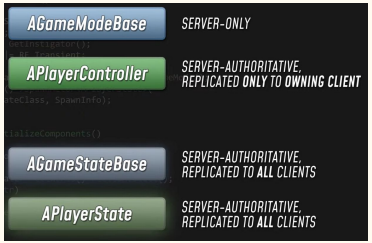
Start - Loading the map - LocalPlayer - PlayerController

- LoadMap then iterates over all **LocalPlayers** present in **GameInstance**.
  - It calls **SpawnPlayActor - PlayActor = PlayerController**
- **ULocalPlayer** is the Engine Object representation of the player
- **APlayerController** is the Game Object representation of the player in the game world
- **UPlayer** is the base class of **ULocalPlayer**, which is also inherited from **UNetConnection**
  - **UNetConnection** is a player connected through a remote process
- Regardless of Local or Remote player, it has to go through a login process
  - The process is handled by the **AGameModeBase**
  - **GameMode's PreLogin** function is only used for remote connection attempts
  - **GameMode's Login** function is then called
    - It will spawns a **PlayerController** Actor and returns it to the world. It will internally spawns a **PlayerState** in the PostInitializeComponents
- **PlayerController** & **PlayerState** are similar to **GameMode** & **GameState**
- At the end, **PlayerController** is associated with **LocalPlayer** and **PostLogin** is called on the **GameMode**

```
// Spawn play actors (i.e. PlayerControllers) for all active local players
for (auto It = WorldContext.OwningGameInstance->GetLocalPlayerIterator(); It; ++It)
{
    FString Error2;
    (*It)->SpawnPlayActor(URL.ToString(1), Error2, WorldContext.World());
}
```

```
bool ULocalPlayer::SpawnPlayActor(const FString& URL, FString& OutError, UWorld* InWorld)
{
    // We're local to the server, so we need to make our own URL to accompany our login attempt
    FURL PlayerURL(nullptr, *URL, TRAVEL_Absolute);

    // Get a player name: this will be the public name from the online service, if available
    FString PlayerName = GetNickname();
    if (PlayerName.Len() > 0)
    {
        PlayerURL.AddOption(*FString::Printf(TEXT("Name=%s"), *PlayerName));
    }

    // Custom ULocalPlayer subclasses can specify additional login options via GetGameLoginOptions
    FString GameUrlOptions = GetGameLoginOptions();
    if (GameUrlOptions.Len() > 0)
    {
        PlayerURL.AddOption(*FString::Printf(TEXT("%s"), *GameUrlOptions));
    }

    // Net ID: typically a big arbitrary number that uniquely identifies the player across the network
    FUniqueNetIdRepl UniqueId(GetPreferredUniqueNetId());

    // Defer to the UWorld to Spawn a PlayerController actor
    const int32 NetPlayerIndex = GEngine->GetGamePlayers(InWorld).Find(this);
    PlayerController = InWorld->SpawnPlayActor(
        this, ROLE_SimulatedProxy, PlayerURL, UniqueId, OutError, NetPlayerIndex);
}
```

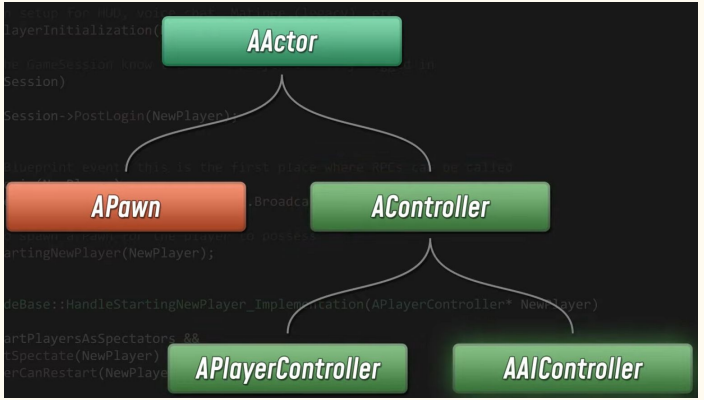




Start - Loading the map - LocalPlayer - Pawn

- By default, on the **PostLogin** of **GameMode**, it will attempt to spawn a **Pawn** for the **PlayerController**
- **APawn** is a specialization of an **AActor** that can be possessed by a **AController**.
- After the **APawn** has been spawned, it'll call **RestartPlayer**

- **Restart** is basically the process of re-attributing a **Pawn** to a **Controller**. It will ensure to
  - Spawn a **APawn** by using correct **Pawn class** specified in the **AGameMode**.
  - Set its transform to a **APlayerStart**
  - **Associate** that **APawn** with the **AController**

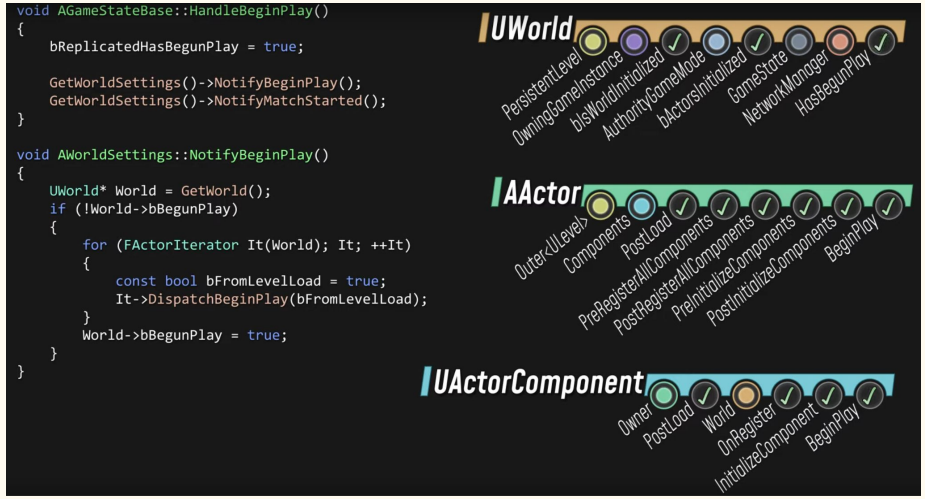



```
void AGameModeBase::RestartPlayer(AController* NewPlayer)
{
    // Find a PlayerStart actor, and use its transform (with no pitch/roll) for the pawn
    AActor* StartSpot = FindPlayerStart(NewPlayer);
    const FRotator StartRotation(0.0f, StartSpot->GetActorRotation().Yaw, 0.0f);
    const FVector StartLocation = StartSpot->GetActorLocation();
    const FTransform SpawnTransform(StartRotation, StartLocation);

    // Get the Pawn class to use for this player; typically DefaultPawnClass
    UClass* PawnClass = GetDefaultPawnClassForController(NewPlayer);

    // Spawn a new Pawn using that class
    FActorSpawnParameters SpawnInfo;
    SpawnInfo.Instigator = GetInstigator();
    SpawnInfo.ObjectFlags |= RF_Transient;
    APawn* NewPawn = GetWorld()->SpawnActor<APawn>(
        PawnClass, SpawnTransform, SpawnInfo);

    // Associate the Pawn with the Controller (before Possess)
    NewPlayer->SetPawn(NewPawn);

    // Let the PlayerStart actor know that it was used
    InitStartSpot(StartSpot, NewPlayer);

    // Possess the Pawn, init control rotation, call SetPlayerDefaults
    FinishRestartPlayer(NewPlayer, StartRotation);
}
```

Start - Loading the map - BeginPlay

- Finally, after all the loading has been made, **LoadMap** function will call **UWorld's BeginPlay function.**
- **BeginPlay** will cascade through a logic of
  - **UEngine** tells the **UWorld** to BeginPlay
  - **UWorld** tells the **AGameMode**
  - **AGameMode** tells the **AWorldSettings**
  - **AWorldSettings** loops over all actors
  - **BeginPlay** is called on all **Actors** that are calling **BeginPlay** on their **components**

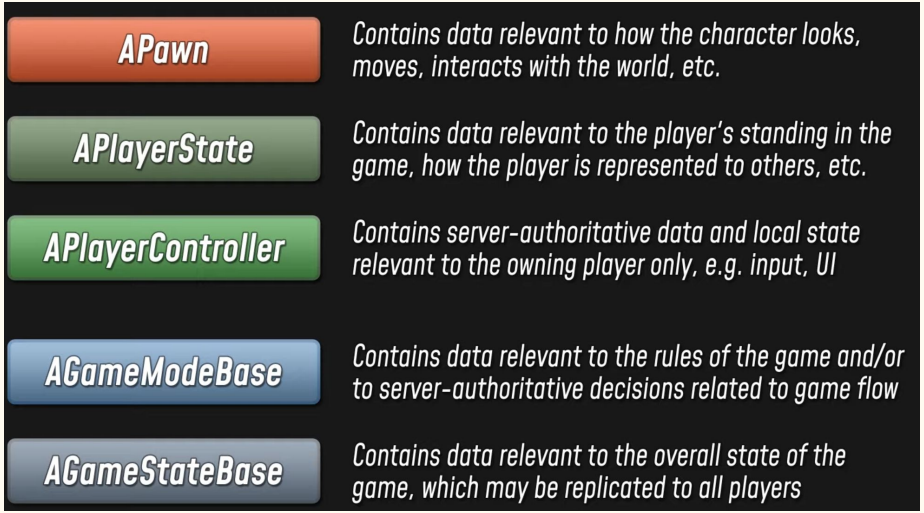
```
void AGameStateBase::HandleBeginPlay()
{
    bReplicatedHasBegunPlay = true;

    GetWorldSettings()->NotifyBeginPlay();
    GetWorldSettings()->NotifyMatchStarted();
}

void AWorldSettings::NotifyBeginPlay()
{
    UWorld* World = GetWorld();
    if (!World->bBegunPlay)
    {
        for (FActorIterator It(World); It; ++It)
        {
            const bool bFromLevelLoad = true;
            It->DispatchBeginPlay(bFromLevelLoad);
        }
        World->bBegunPlay = true;
    }
}
```



Final thoughts

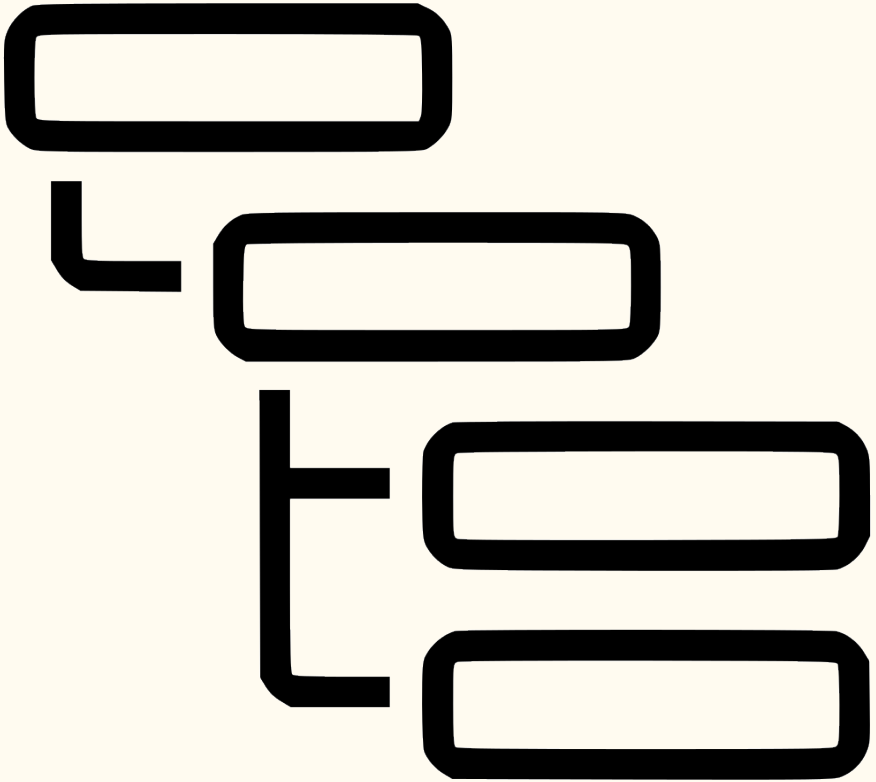
- **Game Framework initialization** is a **really big chunk** which was only **barely** cover for **knowledge purpose**
- Some of you may need to **override** some part of it, some not.
- It is still **important** to know which part is **responsible** of which system.
- Designing data information storage is important has you may fall into a situation where your **datas** are **not located** in the **right place**

- **Overriding** some classes are not always necessary if you just need to **run some code** in **response** to engine initialization events
  - **Callback function** and **static delegate** are used for that and you can connect to them
    - **CoreDelegates.h - FCoreDelegates**
    - **UObjectGlobals.h - FCoreUObjectDelegates**
    - **GameViewportDelegates.h**
    - **GameDelegates.h**
    - **World.h - FWorldDelegates**
  - You can also use **SubSystems**



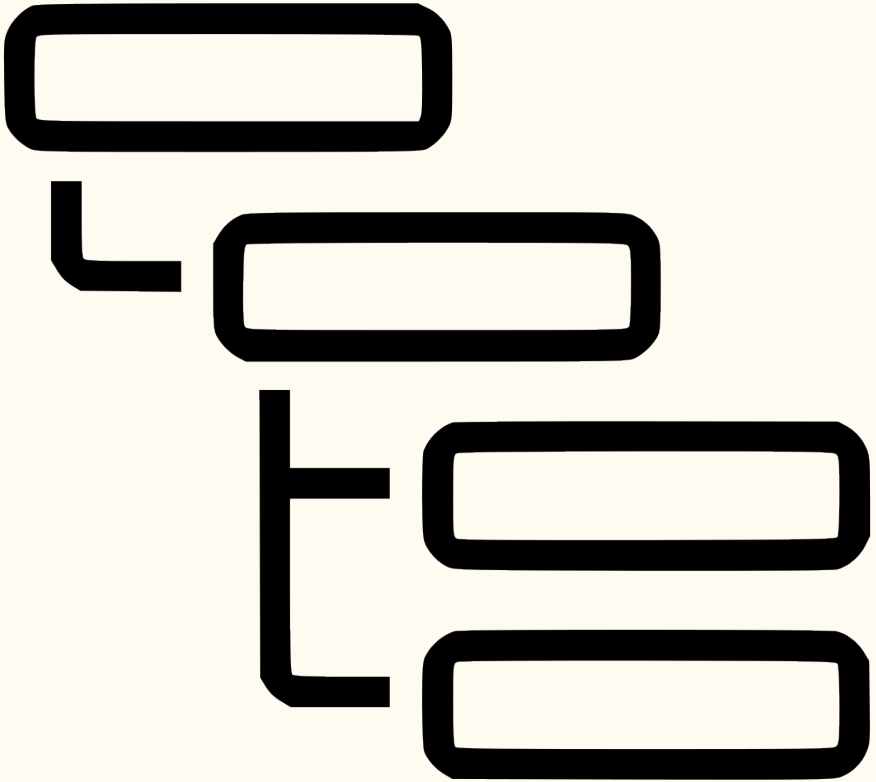
UObject

- It is the **base** class of **all UE objects**
- It makes the **object** interact with Unreal system allowing **reflection** to **exposition**.
- It provides **supports** function for **creating** and **using** objects, and virtual functions that needs to be **override** in child classes.
- **NewObject()** is the simplest UObject **factory** method. It takes several parameters and allow to **create** a UObject.
- UObject are **manage** automatically by the **garbage collector**
    - It means that you need to be **careful** about how you are managing the classes **referencing** UObject.
    - For example, an **Actor** will be less problematic, because it is contains in a **level**, so **garbage collector** will not happen unless you **destroy** it
    - A simple **UObject** is not that simple as it may be contains in **several places**



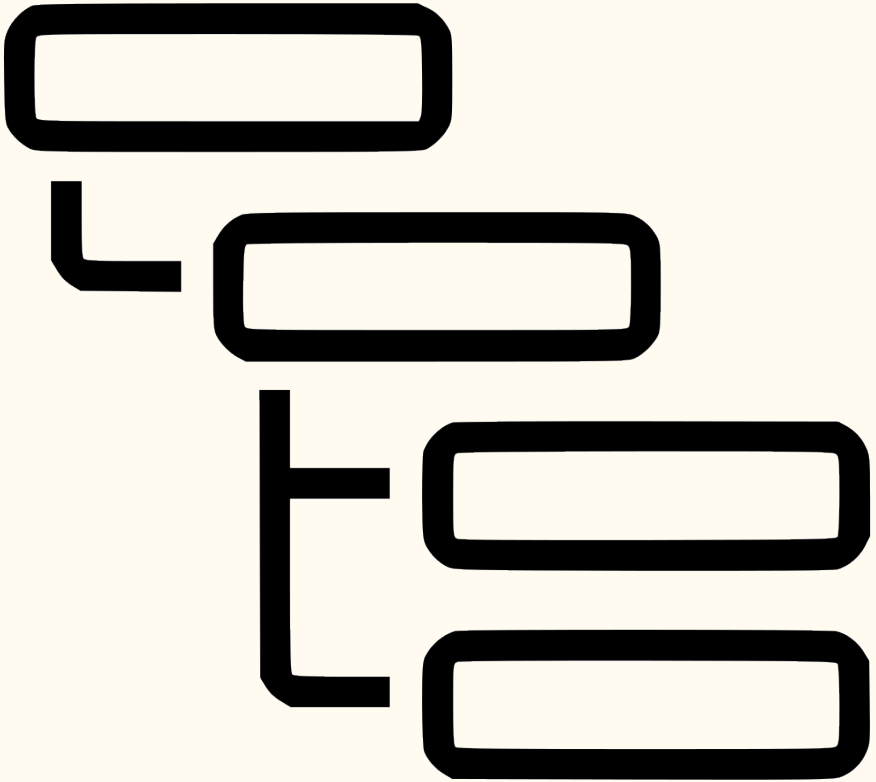
Actor

- Actor is the **base class** for an Object that can be **placed** or **spawned** in a **level**
- In a way, we can see an **Actor** as a **container** that hold special types of objects called **Components**
- In addition to that **Component system**, **replication** is one of the main function of the Actor, allowing to **replicate properties** and **function called** across **network**
- It is important to understand that an **Actor does not** directly **store** a **Transform**. An Actor is forced to have a main component called the **Root**.
    - **Root component** is obviously a **SceneComponent**
    - Being a **SceneComponent**, it'll contains a transform
    - All **other scene component** will be **child** of this one, and have **relative transform** based on it



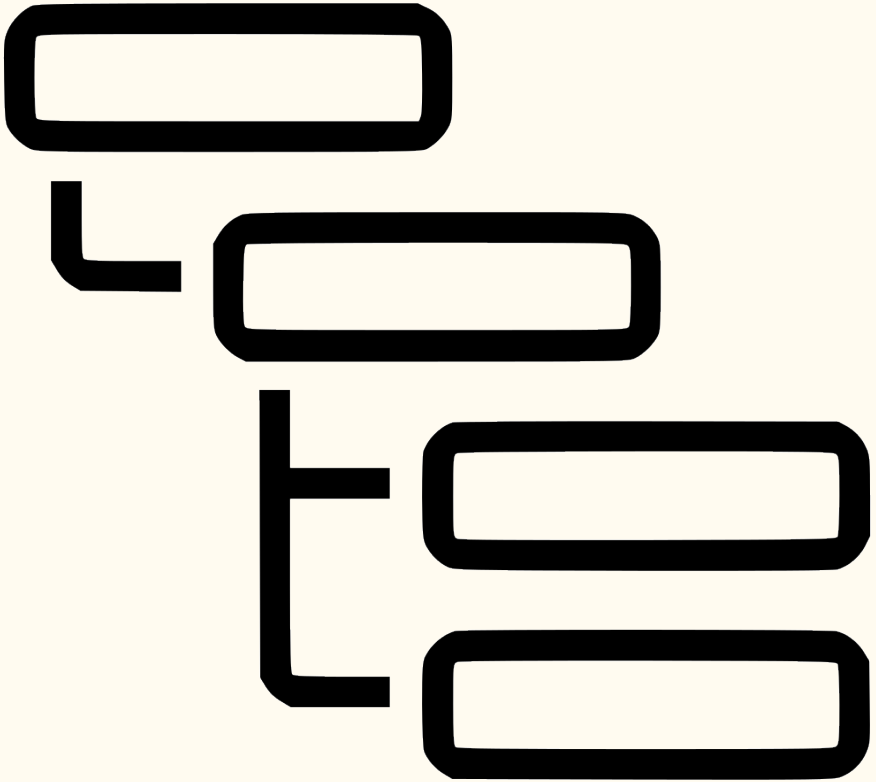
Pawn

- **Pawn** is the **base** class for all actor that needs to be **controlled** in a way, either by **player interaction** or **AI**
- A Pawn is meant to be a **physical representation** of the possession. It can goes from a **spider** to **mecha**
    - **Physical representation** doesn't only mean how it is **visually looking** in the world
    - It also represent **collision** and other **physics interactions**
- By default, there is a **one-to-one** relation between **Controllers** and **Pawns**
- A Pawn **spawn** during **gameplay** is **not automatically possessed** by a **Controller**
- It is most likely that if you need **walk** on your pawn, you'll need **Character** instead



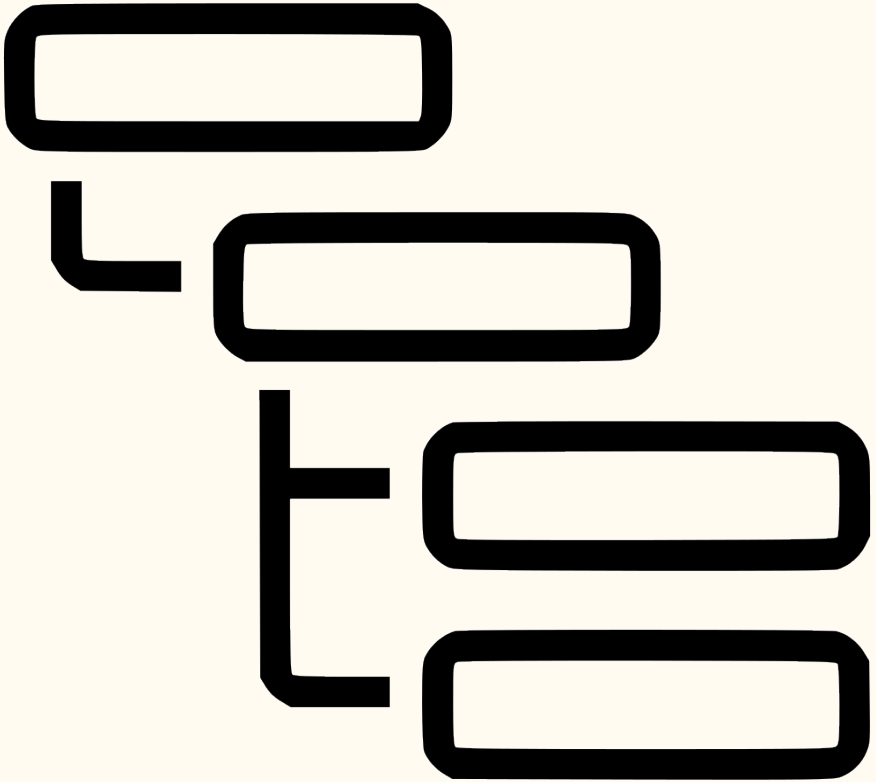
Character

- **Character** is inheriting from Pawn and represent a huge feature extension of it
- We are talking here about **thousand** of line of code that is designed to represent **vertically-oriented** player representation that could **walk**, **run**, **jump**, **fly** or **swing**
- It also contains **implementation** of **basics networking** and **input models.**
- Equally or even more important, it comes with the **CharacterMovementComponent**
    - Again, we are speaking about thousand of line that allows networking through prediction and correction out of the box
- A **Character**, unlike Pawn, comes with a **SkeletalMeshComponent**, to enable **advanced animations** that uses skeleton
- Because of **CharacterMovemenetComponent**, it is needed that the **collider** represent a **vertically-oriented** capsule



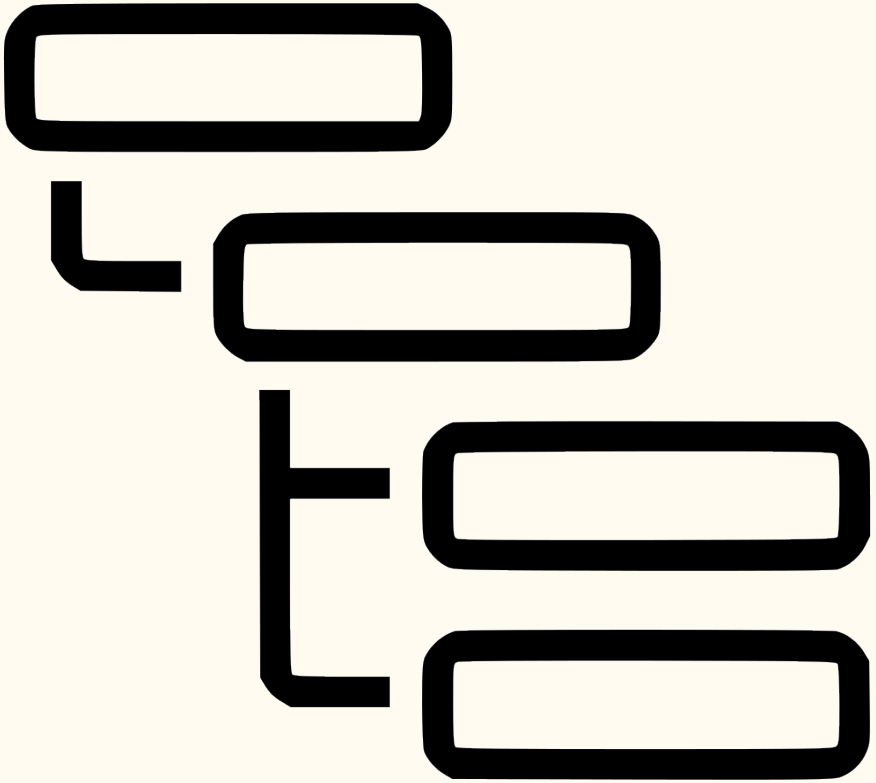
Controller

- A **Controller** is **non-physical actor**
- It **possess** a **Pawn** in order to **control** its actions
- It is important to **understand** that most often, a **Controller** will be living **even** if you **character dies**
    - It is common to **destroy** a Character when you **health** goes **below 0**
    - In that case, we'll not destroy the control, just make sure to **unpossess** the **old Pawn** and **control** the **new spawned** one
- There is 2 different controller type in Unreal
    - **PlayerController** : It is used by human to control pawn, and offer various functionalities which are only available with it like
        - Input handling by default
        - Tracing from mouse
        - Etc...
    - **AIController** : It is used by **computer**. In a **network** environment, this **controller** will only be **created** on the **server**
        - There are packing a bunch of **component** which allow to control an AI with
            - **AIPerception**
            - **Blackboard**
            - Etc...



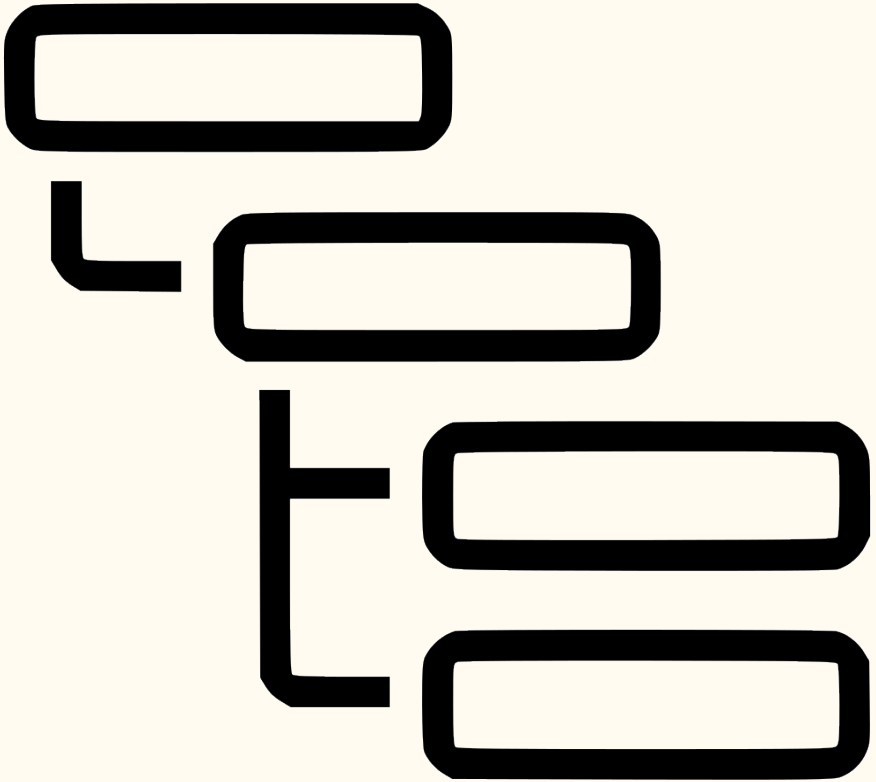
Component

- **ActorComponent** is the **base class** for all component
- **Component** can be defines as **reusable** behavior which needs to goes on various actor
- We can think **components** as **system** which can go from
    - Inventory system
    - Health system
    - Damage system
- It is most likely that this kind of **system** will be only **Actor component** but there is actually **3 different type** of component which you'll **choose based** on what you **need**
    - **ActorComponent :** The most **basic** one which hold all **core** functionalities of component. This one has **no transform** and no **physical behavior**
    - **SceneComponent** : Inheriting from **ActorComponent**, it offers a **transform** and s**upport attachment**. But it has **no rendering** of **collision capabilities**
    - **PrimitiveComponent** : Inheriting from **SceneComponent**, they **contains** or **generate** some sort of **geometry**, generally to be **rendered** or **used as collision data.** Some well known component inherit from it like
        - UShapeComponent
        - StaticMeshComponent
        - SkeletalMeshComponent



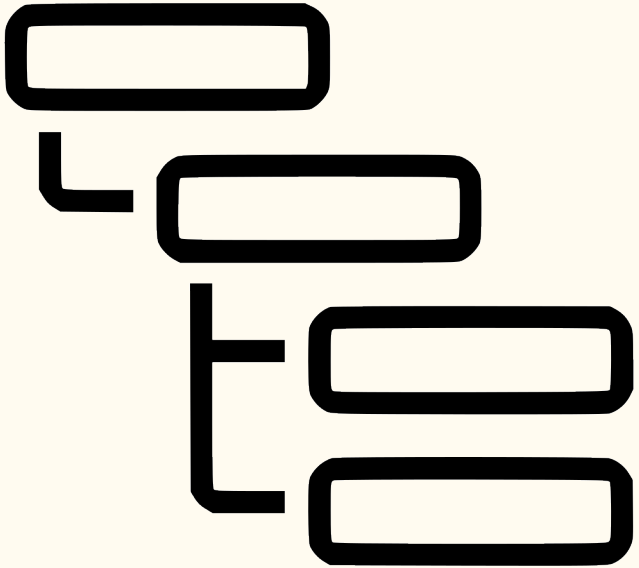
Game Instance

- Sometimes **forgot** in project, **UGameInstance** is actually one of the **most** important class **feature** wise
- It can be seen as a **high-level manager object** for an instance of a **running** game
- It mean that there is **only 1 instance** of this one, and it can be **accessible** from **anywhere**
- From **Project settings,** you can change the **GameInstance** class in order to use a custom one.
- A Game Instance is **created** at **start** of game **executable**
    - It is not designed to be **replicated**, as server and client will **both** have a **distinct game instance** which will run **independently**
    - Therefore, you must **carefully think** what you are putting in the **GameInstance**
    - Obviously, you still can use **RPC** in conjunction to **GameInstance** in order to send message between clients when things happen in the **GameInstance**



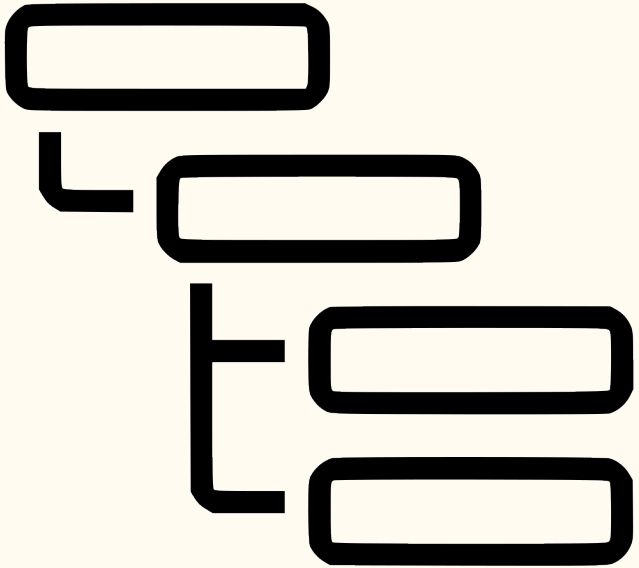
Game Mode

- We'll **highlight** in following slides some **key** actor which may not have that **much importance** in a **singleplayer** environment but a big one for **multiplayer**
- **Game Mode** can be seen as an overall **game manager**
- It handles the **general flow** of your game. It means that it **handles Game States** and how they rotate. An example could be a "**Capture The Flag**"
- It is the **rules giver**, the actor **responsible** for holding and ensuring this informations.
- Some **examples** rules could be
    - Number of player allowed
    - Number of spectators allowed
    - How to enter the game
    - Etc...
- You may see that there is 2 different class from which you can inherit:
    - **AGameMode** : It inehrit from **AGameModeBase** and was present **before** it. It was created for FPS and Unreal Tournament for match issues.
    - **AGameModeBase** : It is a **lightweight** and **streamlined** version of AGameMode. It did become the base class for **GameMode**
- It is common to **create** multiples GameMode because you have **different** type of match with varied rules
    - The **important** thing to understand is that only 1 **GameMode** is **active** at a time
- GameMode is **only present** on the **server** and only the server can **access** or **modify** its informations



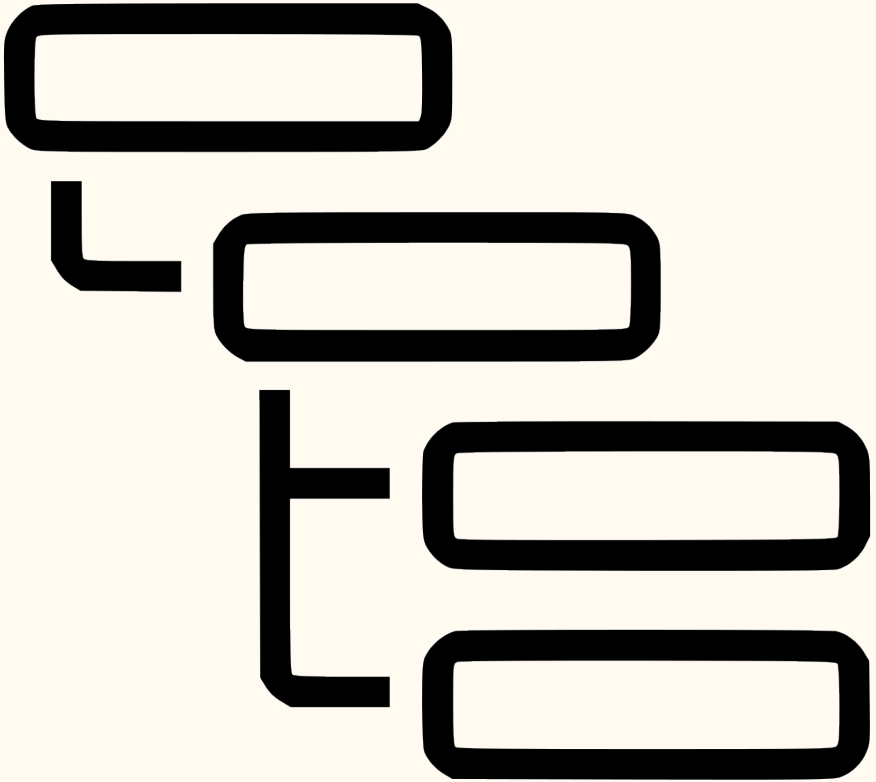
Game State

- **Game State** kind of **works** in **conjunction** with Game Mode
- It is **responsible** to **keeps track** of every **data** relative to the current **state** of the **game** which needs to be **communicate** to **all clients**
    - Timers
    - Scores
    - Winning team
    - Etc...
- It also handles **scripted events** related to the **state.** Let's take this example
    - **PregameState**: Prevents player from performing any action, starts a timer and display it to everyone. When timer expires, ask GameMode to rotate to IngameState
    - **IngameState**: Enable player input, spawn a big loud noise and open players' spawn gates. Open the Capture Point and store the amount of capture time both teams have. When one of the team reaches max score, asks Game Mode to switch to EndgameState.
    - **EndgameState**: Destroy every player's characters and starts a cinematic showing the PlayOfTheGame then asks Game Mode to rotate to ScoreGameState etc.
- Basically Game State needs to be seen as a **data holder** which doesn't **belongs** to a **specific player** but related to the **game mode** and that needs to be **shared**
- The **Game State exists** on the **server** and is **replicated** to all **clients**



Player State

- Finally, let's see the **PlayerState** which can be **seen** like a **Game State specific to player**
- A PlayerState is **created** for **each clients** connected to the server. **PlayerState** are then **replicated** to all clients and contains network related informations like **name**, **score**, etc...
- There is not much to say about it for a basis at it has the same idea as Game State but specific to player



Time to.... highlight a concept

TRC & Console development

Practice

- **General**
  - Inherit from various classes we seen in the lesson like GameMode, Game Instance, LocalPlayer
    - Play with them, override some method and logs some element in order to understand what is going on
    - Look through the hierarchy of calls at the initialization, maybe starting from GameMode
    - Try to add a log when your Controller & Pawn are created (Optional)
  - Create a custom c++ ACharacter called ACustomCharacter and add two functionalities in it
    - A print which will tell the position of the actor every frame
    - A method called sprint which will modify the walk speed of your player
      - Ensure to have the possibility to tweak the sprint speed in the editor
  - Create a custom c++ AController called ACustomController and add two functionalities in it
    - Bind an input in order to activate sprint when the button is held down and cancel sprint when released
    - Every 2 seconds, print a message which tell if your player is currently running or sprinting
  - Create a custom Game Mode and change the pawn spawned by default
    - Register the ACustomCharacter as the default one
    - Register the ACustomController as the default one
- **Follow-through project**
  - Create a specific project Game Mode and make the changes necessary to use it and ensure that it uses the correct classes
  - Create a specific AController for your Guard and ensure to make it possess the AI pawn
    - Create functions even if they are empty for now that'll be use later on, like
      - DetectPlayer : AI part
      - FollowPlayer : AI & Navigation
      - Patrol : AI & Navigation
      - Basically, you can already create those method and implement them, when we'll do AI lesson, we'll be able to rewrite some part of them
  - Create your hierarchy and actor for Ingot. Does it needs a Controller ? Does it needs some important Component ?
  - Create your first Component that'll represent an Inventory, which will be reuse for Player and AI for example containing Ingot count