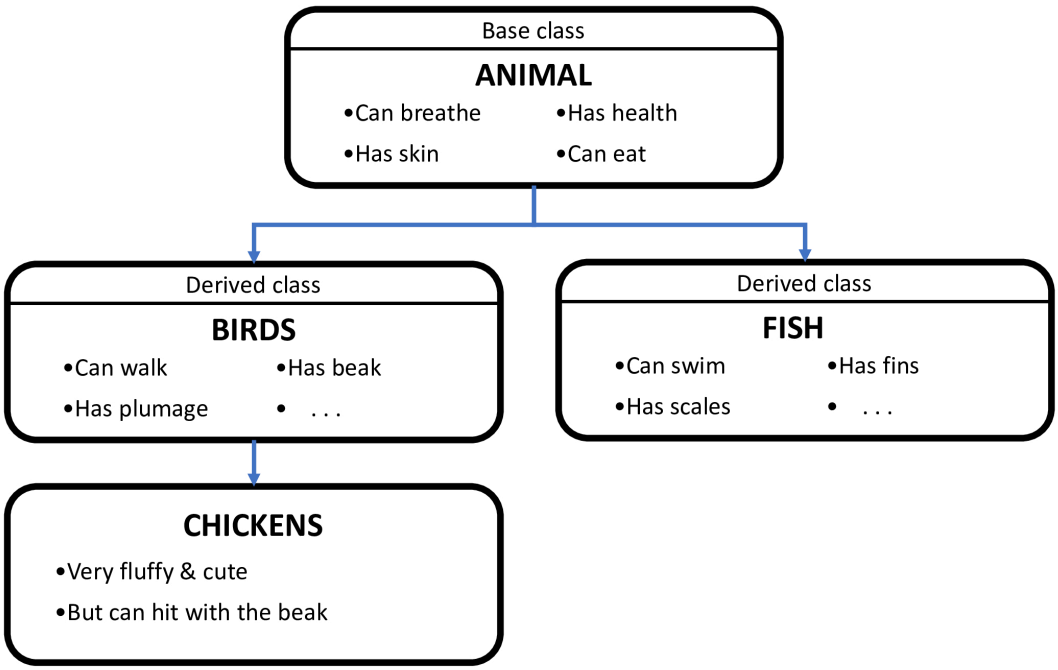
Introduction to ECS

Talking about object-oriented programming (OOP), we consider our practical task as multiplicity of special abstract things. In terms of Unreal Engine these abstractions are mainly called Objects, or `UObjects` if we are talking about C++ coding. Furthermore we apply a common principle like an inheritance model upon them. We create some main abstraction layer called ‘base class’, define some properties (i.e. variables and functions) in it and then by deriving a new type from that class we create other abstractions which in turn get all the parental properties and also define their own additional, distinctive ones. A quite popular example for that approach is a tree of animal inheritance, where a base class represents any animal existing, while the others (derived from the base one) represent some separate animal sorts and kinds:



By altering the base class properties we also change the current state of the whole object of a derived type. We can also use some “virtual” overridable methods to modify the behavior for the derived classes. The problem here is that a game logic can become too scattered across those classes and layers, essentially making those layers horizontal as opposed to a vertical inheritance.

A solution to this game development problem could be an approach called ECS.

Apparatus provides all of the basic ECS idioms and even more. To be unambiguous (and self-sustained, actually) our framework uses a different naming scheme as compared to the classic ECS. Here is the list of analogous terms:

ECS Term	Apparatus Term
Entity	Subject
Component	Detail
System	Mechanic
A group of Systems	Mechanism

ECS Term	Apparatus Term
Archetype	Fingerprint
Chunk	Belt

About ECS+

Apparatus takes a broader approach on ECS that we essentially call ECS+ over here. It introduces a greater amount of flexibility with the help of detail (component) inheritance support and sparse expandable, dynamic belts (chunks and archetypes), subject (entity) booting, etc. The term ECS+ in itself should be treated as more of a synonym for the “Apparatus” wording itself, since the product actually implements it and uses a different terminology from the ground up.

Generally speaking, in our own development philosophy, engineering should never be “pure” or “mathematically correct” but mainly applicable and useful to the end user and/or the developer. Unreal Engine in itself is actually a good demonstration of this approach with so many special cases and design patterns. By having both OOP and ECS patterns combined Apparatus takes the best of three worlds, with UE‘s blueprint world counted.

Built-in Subjects

At a lower level, `Subjective` is an interface and any class that implements it may be called a `Subject`. There are several subjects already implemented in the framework:

- `SubjectiveActorComponent` (based on `ActorComponent`),
- `SubjectiveUserWidget` (based on `UserWidget`),
- `SubjectiveActor` (based on `Actor`).

Those should be sufficient for the most cases, but you can easily implement your own additional subject classes in C++.

Built-in Mechanisms

The `Mechanical` class is also an interface, with the most useful mechanisms already implemented:

- `MechanicalActor` (inherited from `Actor`),
- `MechanicalGameModeBase` (`GameModeBase`),
- `MechanicalGameMode` (`GameMode`).

You would hardly ever need to implement your own mechanisms. Should you wish to, you can also do it in C++.