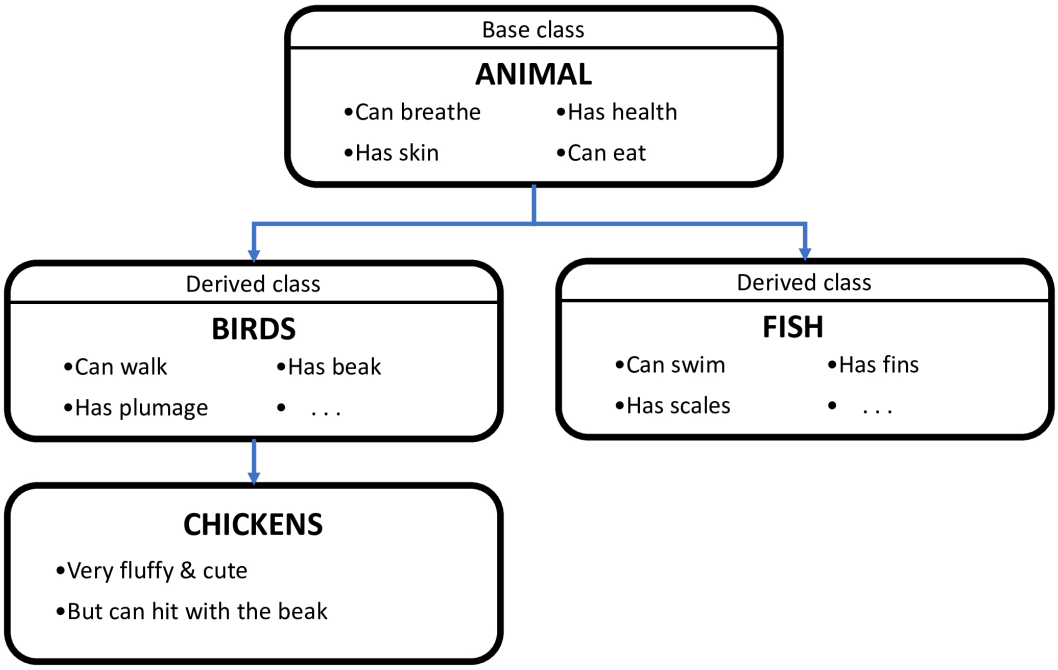
Introduction to ECS

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` code-wise). Furthermore we apply a principle like an inheritance upon them. We create some main abstraction layer called 'Base class', define some properties (i.e. variables and functions) and by deriving from that class we can create other abstractions which in turn get all the parent properties and define their own additional, distinctive ones. The most popular example for that approach is a tree of animal inheritance, where the base class represent any animal existing, but the others (derived from the base one) represent some separate animal sorts and kinds:



By using the base class properties we can change the current state of the objects in the derived class. But the problem is the game logic can become too scattered across those classes and layers.

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

Apparatus provides all of the basic ECS idioms and even more. To be unambiguous the framework it uses a different naming scheme as compared to classic ECS. Here is the list of analogous terms:

ECS Term	Apparatus Term
Entity	Subject
Component	Detail
System	Mechanic
A group of Systems	Mechanism
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). The term ECS+ in itself should be treated as more of a synonym for the “Apparatus” wording itself, since it actually implements it and uses a different terminology from a classic ECS.

Generally speaking, in our own 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.

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++.