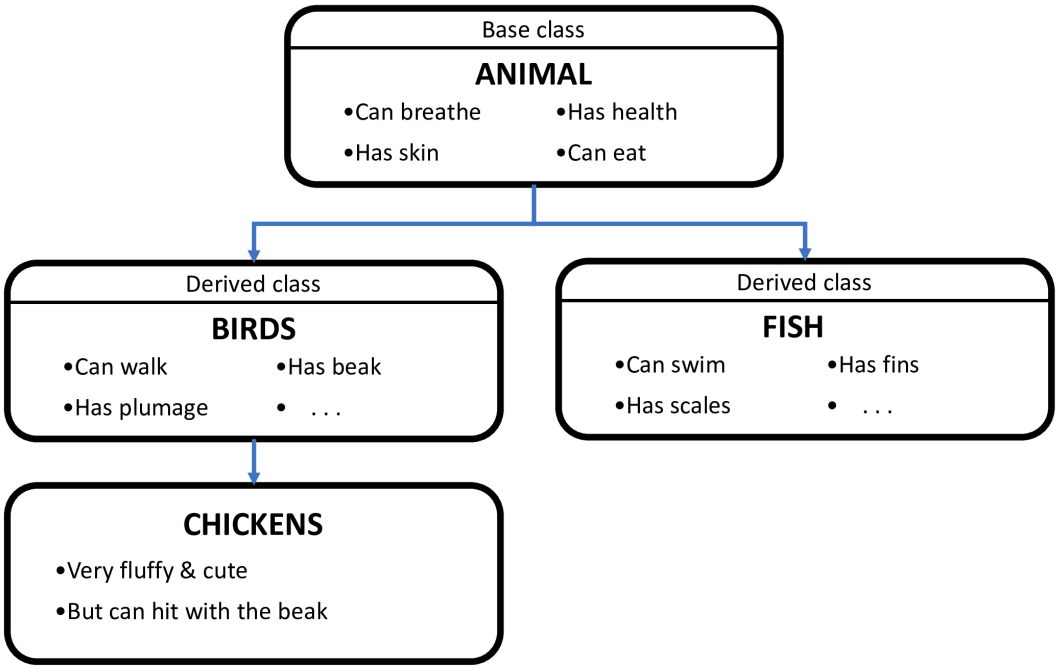
Introduction to ECS+

Talking about OOP (object-oriented programming) we consider our practical task as multiplicity of special abstract things. In terms of UE4 these abstractions are named UObjects and over them we can apply such a principle like an inheritance. What does it mean? We create some main abstraction called ‘Base class’, define properties (i.e. variables and functions) and by deriving from that class we can create other abstractions which will get all the parent properties and define their own additional or distinctive ones. The most popular example in that way is the tree of animals inheritance, where the base class represent any animal exist, but the others (derived from the base one) represent separate animal sorts:



By using base class properties we can change current state of objects of derived class. But the problem is the game logic can become too scattered across the classes. In game development there is such an approach named ECS (that is Entity component system).

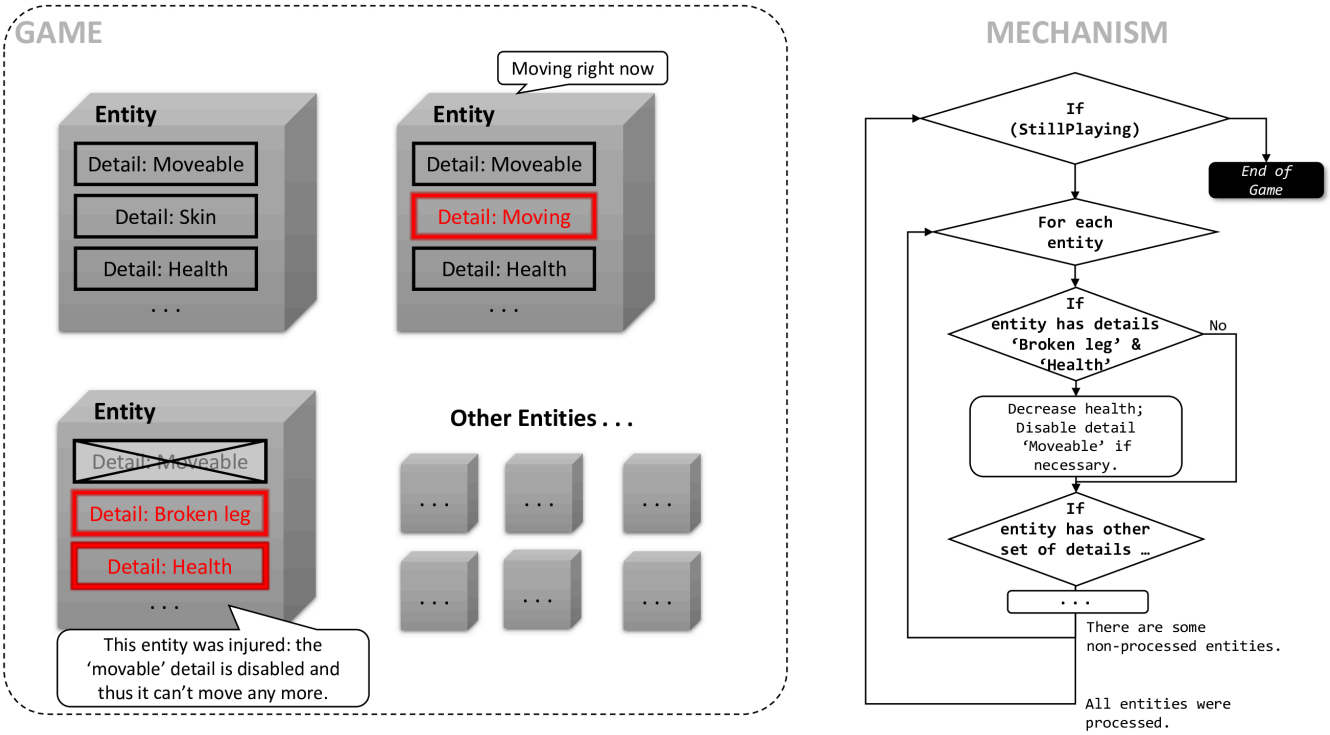
So what are the main concepts? Here we make an attempt to understand each abstraction as individual entity. Globally, there are some states each entity may be in or not. From this point of view, each entity can be represented as vector of 1-es and 0-es, where each vector’s component represents if the entity at the current moment is in the defined state or not. That’s more: some entities may be constructed in such a way that they fundamentally can’t go to certain states. For example, let us say we declare state ‘swimming’ and the entity ‘house’ then (that’s quite logical) we can’t move ‘house’ entity to the ‘swimming’ state. So then each vector now have different components number depending on which entity it represents. Now let’s call vector component representing certain state type as ‘detail’. Each entity has a variety of details and each detail of the entity has the value 0 (‘disabled’) if the entity isn’t in the detail state right now, and the value 1(‘enabled’) if the entity is in the corresponding state.

You can imagine each entity as a box of details. These details have the bool variable ‘bActive’ and if it’s set to 0 - then the detail is disabled and is enabled otherwise. There is ability to remove and add details dynamically if it’s necessary. For example ‘man’ can get hurt from the pistol shot in the chest but only not when he has the bulletproof vest (technically you can declare such a detail like ‘getting a

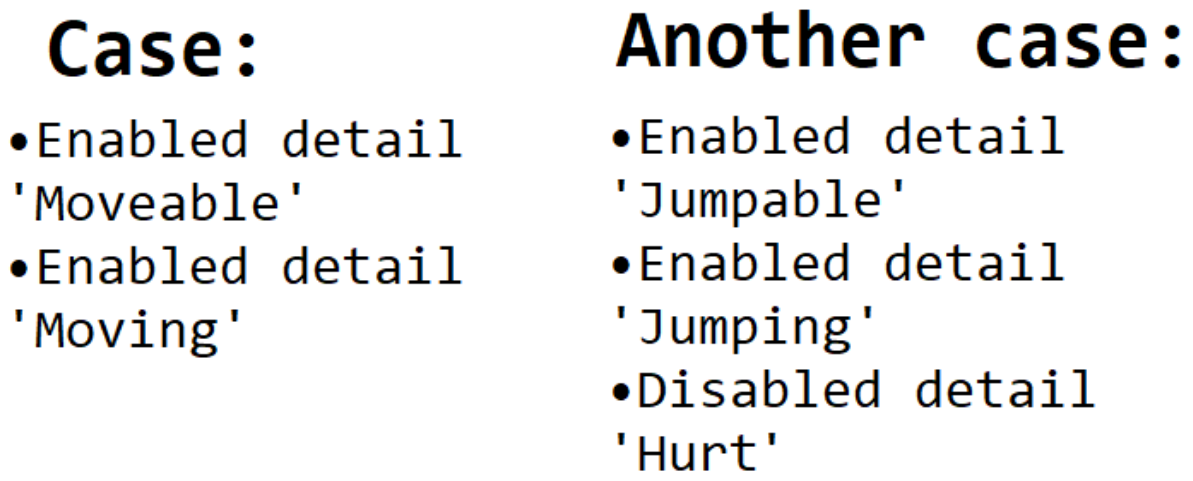
pistol shot' what will be something like 'temporary state' and immediately should be replaced by other detail like 'injured' one).

Ok, we have some entities which contain different details. Furthermore we can apply some changes to all the entities based on which details they have and if these details are enabled or not. For example, for each entity which has the details 'burnable' and 'burning' both enabled, we can decrease their health (or armor) and then disable (or absolutely remove) 'burning' details from them (simply speaking, the fire went out).

As you understand these changes are actually defining the whole game logic and we have to run them continuously during the playing to keep dynamics of states. It's also quite clear we can place all the code which will apply these changes in the one class, so it becomes easier to view and modify them. That class called 'Mechanism' and it's responsible for applying changes to each object depending on which details it has. Now the whole picture is next:



Here disabled details are crossed out, but details which will produce some changes are highlighted. Of course a mechanism can't process entities parallelly, instead it iterate other each entity and check determined cases. Each case declares which details should the entity has and which of them should be enabled or disabled. If an entity complies with the case requirements, some changes will be applied to it, else the next case will be checked out and so so on. Thus, the 'case' is something contains a vector of 0-es and 1-es representing 'being in certain states':



A programmer declares cases and what actions to undertake for each entity what complying with the case. **You should also understand that the ‘another case’ in the example above will determine what to execute over the entities what have the detail ‘hurt’ disabled and the entities what haven’t this detail.** You also may say the iteration across all the entities is **slower** than the use of UE4 dispatchers or simple function calling, but on the low level it’s coded in the most optimized way, so actually there is no any difference.