Introduction to ECS

OOP Quirks & Limitations

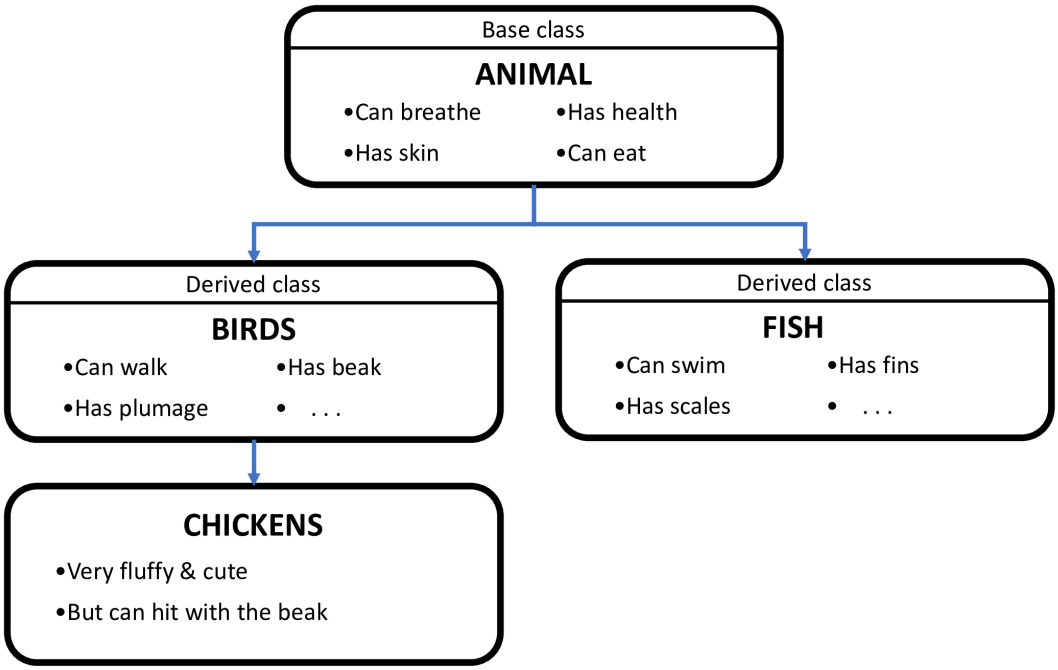
Prior to exploring the ECS main concepts, we should address the object-oriented programming (OOP) pattern and some of the issues related to it. That“s because the current state of Unreal Engine“s user-level API is basically OOP-based. Even the actor“s component model is somewhat limited and is mostly static in nature. It really shouldn“t be confused with a data-oriented approach we“re presenting you with.

A large portion of C++ language is also OOP and it was one of the main reasons behind its original design. It seems like the general top-level UE“s architecture was heavily based on C++ some features, or at least was heavily inspired by them. That“s quite logical if you think of it. In the end C++ is the main language behind the tech. The Unreal build toolset (UBT and UHT) is implemented in C# which is also an OOP language by the way.

So, when we develop games (and software in general) the OOP-way, we usually decompose all of the available entities and concepts into the hierarchies of their corresponding types. In terms of Unreal Engine those are mainly called Objects (or `UObjects` if we are talking about C++ coding part). We create the hierarchies of those classes where each class introduces some new properties and methods while also inherting the properties and methods of its parent (base) class. All of Unrealean classes are derived from a single root Object (`UObject`) ancestor class in the end. Every time we create a new Blueprint (or a new C++ class) it basically derives from this root class.

As the time developing our game goes we usually start to notice that some of our classes tend to have more and more in common. So we begin to refactor the codebase (or the Blueprint assets), extracting those common properties and methods into some separate base classes and try to derive the other classes from them. This task can already be quite sophisticated since Unreal“s object model does not support multiple inheritance (while C++ actually does), so you basically can“t derive your class from the two common ones. Anways, those new common classes in turn may become too common so the new common common class is introduced and the process repeats.

A quite popular example for that approach is a tree of animal inheritance, where a root class represents basically any animal existing (something like `UObject`), while others derived represent some separate animal sorts and kinds on their own:



The descendant types inherit the properties of their respective base types, so by altering the base class properties we also change the current state of the objects, if they are of some derived descendant type. We can also introduce some "virtual" overridable methods to be able to modify their behavior in the derived classes.

All of that functionality surely look quite useful and nice, isn“t it? Well yes it is, until you have like 100 classes. The problem here is that your game logic becomes too scattered across the class-layers in the hierarchy. You tend to forget what is defined on what level, what is overridable and what“s not. The amount of coupling between the entities becomes scary and error-prone. Your coding partners who wrote some aspect of the base classes, become the keepers of some «secret knowledge». The time of refactoring sky-rockets for this every-growing hierarchy. This «ceremonial» job becomes your job itself, but not the development of the final end-user features. Not a good thing at all, considering how limited the time and resources can be.

ECS to The Rescues

A solution to this hassle could be an approach called Entity–component–system (or ECS for short).

There are no hierarchies in terms of ECS, as every game object (i.e. *entity*) is basically flat in its nature. It consists of data-only parts called *components*. You can combine those arbitrary on a per-entity basis.

Want your RPG character become a poison damage dealer? Add a `poisonous` component to it and consider it done, as long as you also implement a *system* that actually solves the poison mechanic.

Systems are remote to the details they operate upon. That“s also a feature of ECS. They are like distant code blocks that take certain details as their operands and evaluate a certain result. Systems are run after other systems and together they form a complete pipeline of your game.

That“s basically it. But how come such a useful concept is not implemented in Unreal Engine yet? Well, it actually is, take Niagara as an example. The rendering pipeline can also be considered a pipeline. In other words, that may be all sorts of specific data-driven approaches taken in Unreal Engine, but none of them are as general-purpose as its OOP nature. We provide such approach for any of the user-available side of the Engine!

Make sure your read our Apparatus to ECS Glossary to get to know our terminology.

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++.

From:
http://turbanov.ru/wiki/ - **Turbopedia**

Permanent link:
http://turbanov.ru/wiki/en/toolworks/docs/apparatus/ecs?rev=1638554719

Last update: **2021/12/03 18:05**