Введение в ECS

Неудобства ООП: причуды и ограничения

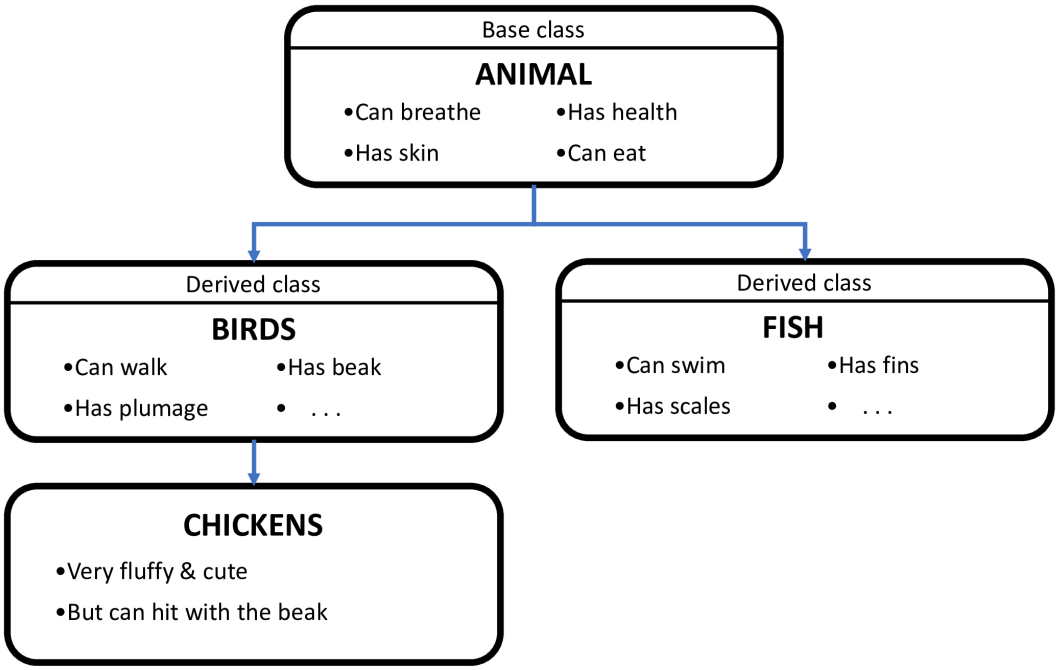
Прежде чем переходить к обсуждению ECS, стоит обратиться к Объектно-ориентированному программированию (ООП) и его проблемам, так как пользовательский API Unreal Engine'a основан именно на парадигмах ООП. Actor-component модель тоже, в некотором смысле, ограничена, и по своей натуре в большей степени является статичной. Не следует путать эти подходы с data-ориентированным, который мы вам предлагаем.

Большая часть C++ - это тоже ООП, что и стало основной причиной такого дизайна языка. Похоже, основная архитектуры UE верхнего уровня была в большой степени базирована на особенностях C++, или, по крайней мере, была вдохновлена ими. Если задуматься, это довольно логично. В конце концов, C++ - это основной язык, на котором строятся технологии. Unreal build toolset (а конкретней UBT - UnrealBuildTool - и UHT - UnrealHeaderTool) реализован на C#, который, к слову, тоже объектно-ориентированный.

Итак, программируя игры (или программное обеспечение вообще) в стиле ООП, мы разбиваем всевозможные сущности и концепции в иерархию их соответствующих типов. В терминах Unreal Engine все эти абстракции называются Объектами (или `UObjects`, если мы говорим про C++ составляющую). Мы создаём иерархию классов, где каждый класс определяет новые свойства и методы, одновременно наследуя свойства и методы родительского класса. В конечном счёте, все классы Unreal-а наследуются от единого корня - `UObject` предка. Каждый созданный новый блупринт (или C++ класс) является наследником этого базового класса.

По истечении некоторого времени разработки мы замечаем, что наши классы начинают всё более и более походить друг на друга. Тогда мы проводим так называемый рефакторинг кода (или Blueprint-ассетов), выделяя схожие свойства в отдельный базовый класс и устанавливая связи наследства в дочерних классах. Уже эта задача может быть довольно сложной, так как объектная модель Unreal-а не позволяет множественное наследование (в то время как C++ позволяет), поэтому вы не можете наследоваться от двух или более классов. В любом случае, новые родительские классы могут иметь много общего, и тогда вводится новый ещё более общий класс, и процесс повторяется.

Довольно известный пример этом подходу - дерево наследования животных, где классом-корнем выступает абстракция, представляющая собой любое живое существо (аналог `UObject`), а остальные абстракции являются отдельными животными видами:



Потомственные типы наследуют свойства их соответствующих базовых типов, и, таким образом, изменяя свойства базового класса, мы также меняем поведение всех потомков. Мы можем так же ввести такую технологию, как “перегружаемые методы” ("virtual overridable methods"), чтобы иметь возможность переопределять поведение функций в классах-потомках.

Весь этот кипишь выглядит довольно неплохо, разве нет? Да, всем этим можно пользоваться, но лишь до тех пор, пока у вас нету 100 классов. Проблема в том, что игровая логика становится очень размазанной по всем уровням иерархии. Вы всё чаще забываете, что было определено в конкретном классе, какие методы можно перегружать, какие нет. Количество взаимосвязей катастрофически увеличивает возможность ошибки, а неразбериха только усугубляется. Ваши коллеги, писавшие код используемых классов, становятся хранителями каких-то “тайных знаний”. С ростом дерева классов время, требуемое на рефакторинг, увеличивается с экспоненциальной скоростью. Этот “ритуал” становится неотъемлемой частью работы, но к самой разработке утилит для целевого пользователя всё это мало относится. Совершенно неприемлемый подход, учитывая то, как ограничено бывает время и ресурсы.

Спасительный ECS

Решением этих хлопот может быть такой подход, как Entity–component–system (или кратко - ECS).

В ECS нету никаких иерархий, каждый объект игры (т.е. *сущность* - англ. *entity*) изначально нейтральный. Он состоит только из частей данных, называемых *компонентами*. Вы можете комбинировать их на конкретной сущности произвольным образом.

Вы хотите дать своему RPG-герою возможность наносить урон ядом? Просто добавьте компонент отравляющий и, считай, дело сделано; конечно, если была реализована соответствующая *система*, которая эту логику определяет.

Система - это удалённый агент, оперирующий над деталями. В этом ещё одна особенность ECS. Системы представляют собой что-то наподобие внешнего кода, который берёт определённые компоненты, как свои входные данные, и, оперируя над ними, возвращает результат. Системы выполняются одна за другой и вместе формируют то, что называется игровым механизмом.

На этом всё. Но как такой простой, удобный и гибкий принцип не был реализован в Unreal Engine до сих пор? Ну, на самом деле был: возьмите Niagara, например. Устройство рендера так же может быть рассмотрено как разработанный софт. Другими словами, какие сорта специальных data-ориентированных подходов в UE не возьми, ни один из них не является настолько выделенным, настолько универсальным, ни один из них не удовлетворял бы критериям общего назначения, как это делает Unreal-овская природа ООП. Мы предлагаем подход, обладающий всеми этими качествами и пригодный для *любой* доступной разработчику стороны Unreal Engine!

Убедитесь в том, что вы прочитали Словарь Apparatus, чтобы различать введённые нами новые термины.

Про ECS+

Apparatus берёт более широкое направление, чем обычный ECS, поэтому мы и называем его ECS+. Вводится бóльшая гибкость при помощи поддержки наследования деталей (компонентов), динамических расширяемых belt-ов (наравне с более статичными *chunk*-ми и *archetype*-ми), subject-booting и др. Само понятие ECS+ должно рассматриваться как синоним слову “Apparatus”, так как плагин реализует и полноценно использует именно эту терминологию с самого своего начала.

Вообще говоря, в нашей философии разработки, проектирование никогда не должно быть “математически корректным”, но, напротив, - более применимым и удобным в использовании как для конечного пользователя, так и для программиста. Unreal Engine, конечно, является хорошим примером такого подхода со всеми своими концепциями и паттернами. Соединяя в себе оба паттерна ООП и ECS, дополняя функционал в UE-блупринтах, Apparatus включает всё самое лучшее из 3-х миров, тем самым становясь незаменимым инструментом разработки.

Встроенные сущности

На низком уровне, `Subjective` - это интерфейс, и любой класс, который его имплементирует, может быть назван сущностью (`Subject`). Есть несколько встроенных в фреймворк subject-ов:

- `SubjectiveActorComponent` (на базе `ActorComponent`),
- `SubjectiveUserWidget` (на базе `UserWidget`),
- `SubjectiveActor` (на базе `Actor`).

Этого должно хватить в подавляющем большинстве случаев использования, но вы, естественно, можете реализовать свои subject-классы в C++.

Встроенные механизмы

Класс `Mechanical` - это также интерфейс, с уже реализованными основными, самыми полезными механизмами:

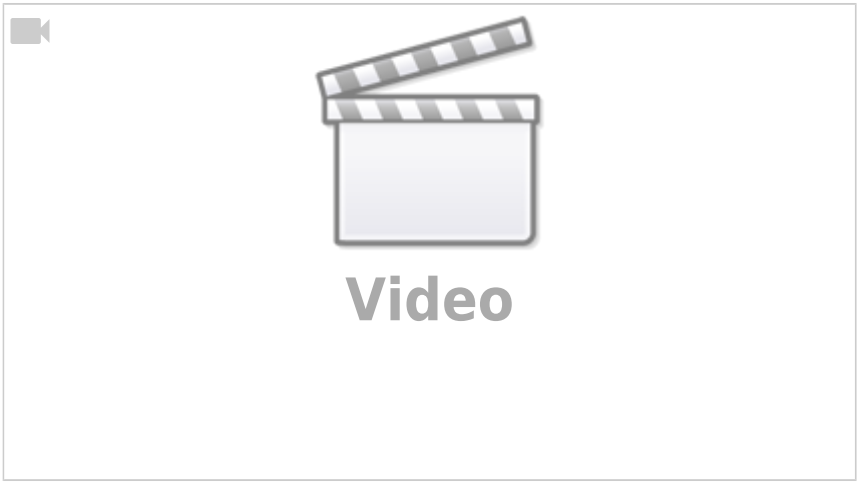
- `MechanicalActor` (наследованный от `Actor`),
- `MechanicalGameModeBase` (`GameModeBase`),
- `MechanicalGameMode` (`GameMode`).

Очень вряд ли вам потребуется дополнять функционал своими механизмами. Если такая потребность возникнет, вы также можете достичь этого также при помощи C++.

Сторонние ресурсы

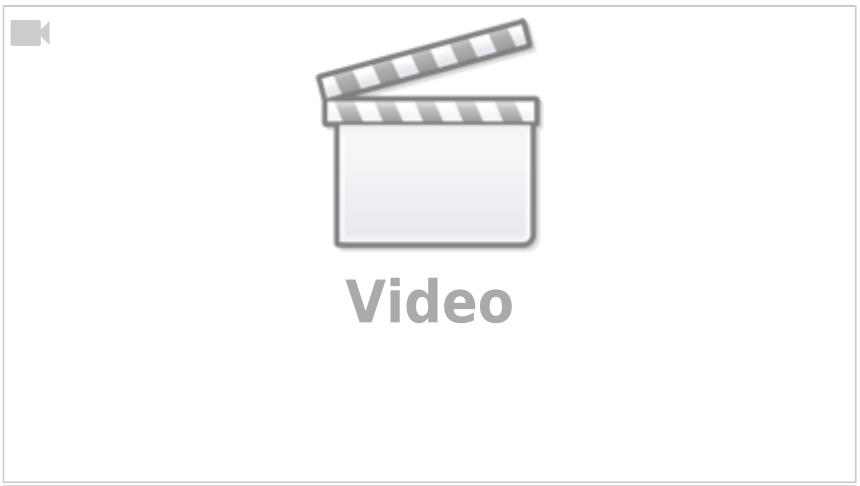
Вместе с нашим сообществом мы собрали коллекцию полезной информации, касающейся ECS и вообще data-ориентированного подхода в целом. Вы можете изучить её для углубления своих знаний.

Ресурсы на русском

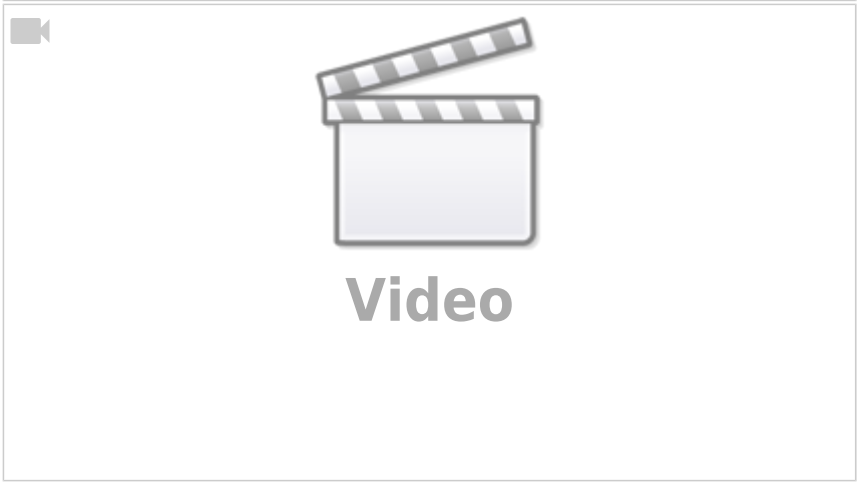


Англоязычные ресурсы

Здесь на английском, но вы всё равно можете посмотреть их, периодически пользуясь переводчиком.







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

Permanent link:
http://turbanov.ru/wiki/ru/toolworks/docs/apparatus/ecs

Last update: **2021/12/18 12:19**