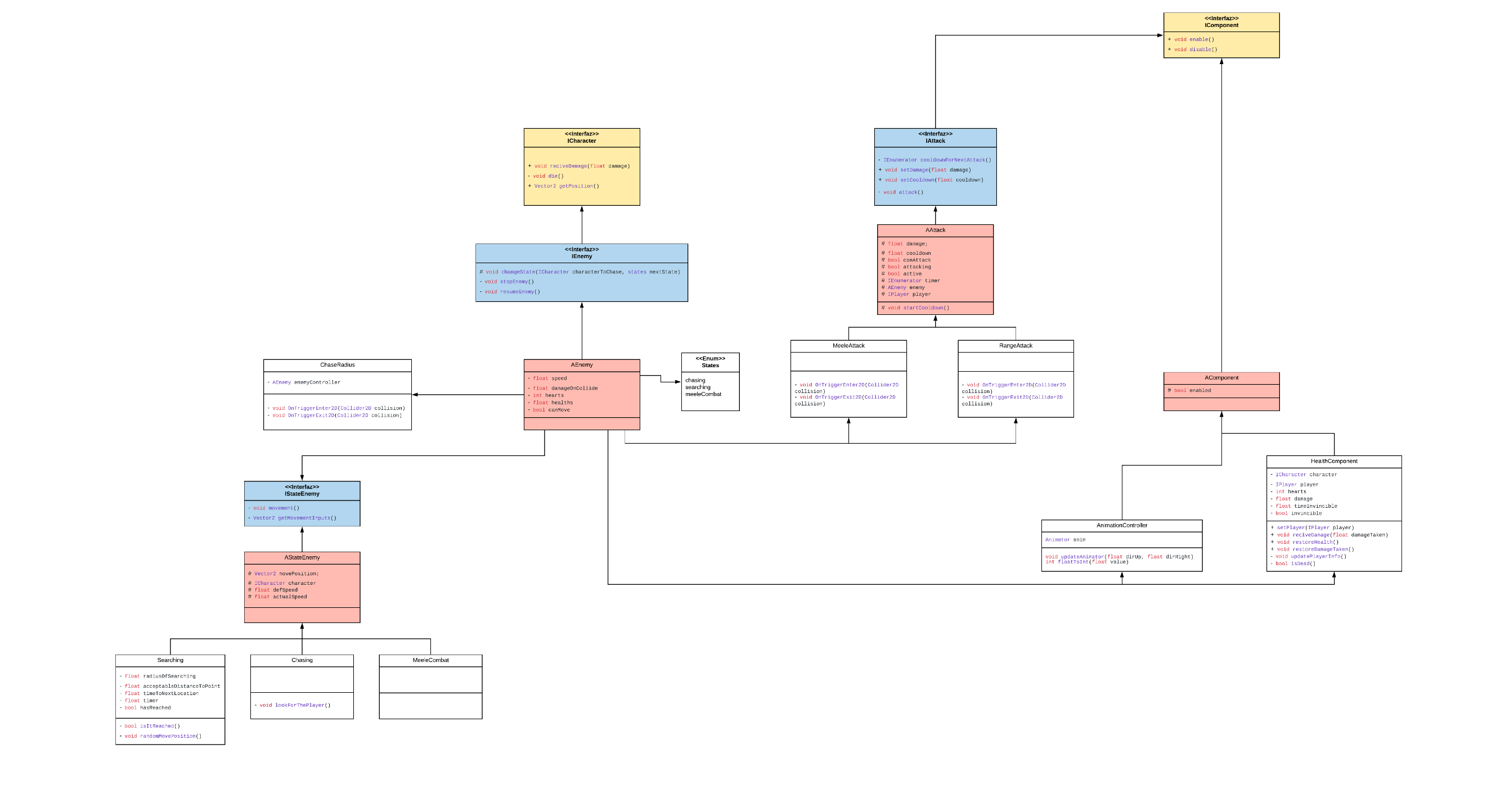
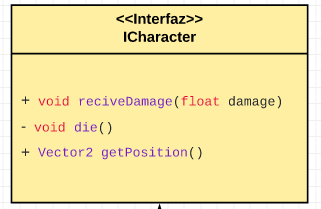
**Enemigo**

Para crear enemigos hemos implementado una arquitectura que facilita la creación del nuevo tipo de enemigos.

****

*ICharacter*

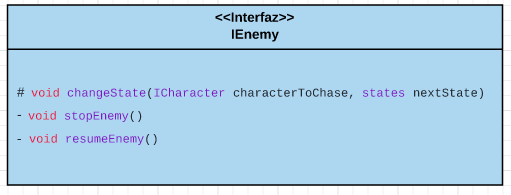
La interfaz ICharacter sirve para los personajes vivos en el juego. En nuestro caso la implementan las clases de enemigos y del jugador.



* **reciveDamage –** sirve para recibir el daño. Utiliza HealthComponent para manejar la vida del personaje
* **die –** sirve para matar al personaje
* **getPosition() –** sirve para obtener la dirección del personaje. Está utilizado en el estado del enemigo para perseguir

*IEnemy*

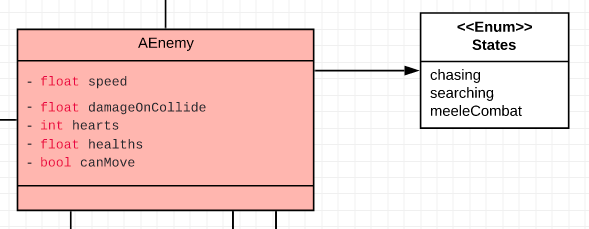
Para los enemigos tenemos la interfaz *IEnemy* que implementa los métodos necesarios para los enemigos. También hereda la interfaz ICharacter.



* **changeState** – sirve para cambiar el estado del enemigo (de esto hablamos más adelante).
  + **characterToChase** – sirve para saber que personaje perseguir
  + **nextState** – para saber en qué estado vamos a cambiar
* **stopEnemy** – deshabilita el enemigo. Sirve para pausar el juego
* **resumeEnemy** – habilita el enemigo. Sirve para continuar el juego

*AEnemy*

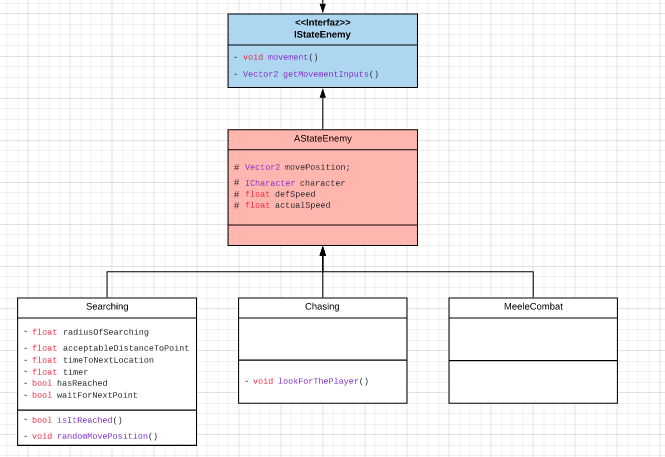
Tenemos una clase abstracta que es principal para los enemigos. Implementa la interfaz IEnemy y tiene la funcionalidad básica para cualquier enemigo. Por ser abstracta podemos modificar la funcionalidad en unas áreas en distintos enemigos si hiciera falta. Los métodos son aquellos que están en la interfaz.



* **speed** – la velocidad del movimiento del enemigo
* **damageOnCollide** – el daño que hace el enemigo cuando se toca con otro personaje. Esto es opcional, se pone al 0 cuando no queremos hacer el daño
* **hearts** – las vidas del personaje. Esta manejado en el HealthComponent
* **healths** – cuando daño puede recibir antes de pedir una vida. Esta manejado en el HealthComponent
* **canMove** – sirve para deshabilitar/habilitar el movimiento
* **states** – sirve para cambiar el estado del enemigo de forma mas sensible y leída

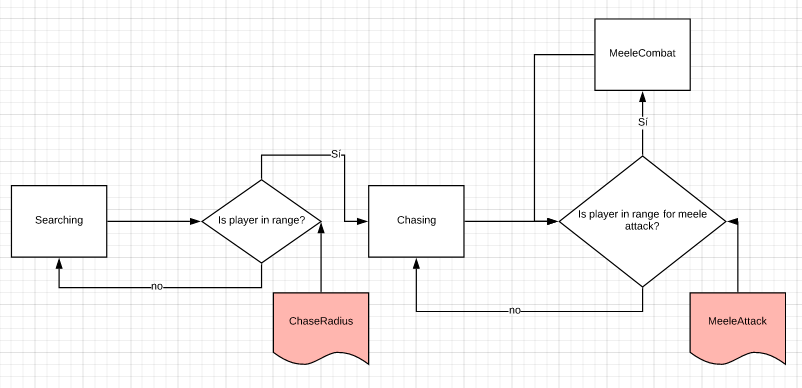
*IStateEnemy*

Es una interfaz para cambiar el estado del enemigo, aplicando el patrón del estado, de forma que no tenemos que preocuparnos cuando hacemos el movimiento. La lógica de movimiento esta implementado aquí ya que tenemos tres estados: searching, chasing, meeleCombat.



* **movement** – mueve el enemigo
* **getMovementInputs** – sirve para obtener los datos para AnimatorController

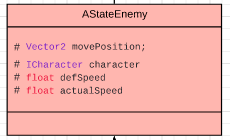
El cambio del estado se hace en la clase AEnemy, en las clases Searching, Chasing, MeeleCombat solo hacemos el comportamiento del movimiento. El comportamiento inicial es Searching.



Para comprobar si el jugador está en el radio de perseguir o en el radio del combate de poca distancia ejecutamos la lógica en las clases ChaseRadius y MeeleAttack correspondiente. Pero para pasar al estado Chasing desde Searching la lógica de la comprobación está en la clase Searching.

*AStateEnemy*

La clase AStateEnemy sirve para tener una funcionalidad básica de del movimiento.

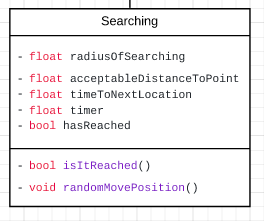


* **movePosition** – el vector que contiene la dirección adonde mueve el enemigo. Esto depende del estado en cual está el enemigo, puede ser que sea la dirección aleatoria para buscar, la dirección donde está el jugador etc…
* **character –** el personaje para perseguir. Puede ser el jugador u otro enemigo dependiente del comportamiento del enemigo especifico.
* **defSpeed –** la velocidad normal. Nunca cambia
* **actualSpeed –** la velocidad actual. Cambia dependiente del estado

Tiene el método movement de IStateEnemy que es Virtual (puede ser sobrescrito) en cual mueve el enemigo a MovePosition. En las clases de estados implementamos la funcionalidad para cambiar MovePosition y también añadimos unas cosas en el método del movimiento (por ejemplo, esperar unos segundos cuando esta buscando el jugador).

*Searching*

Estado para buscar el jugador. Mueve enemigo a la dirección aleatoria y espera allí para cambiar la dirección del segundo movimiento.



Variables:

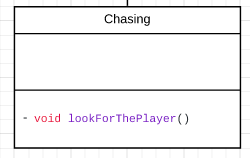
* **radiusOfSearching –** el radio en cual obtenemos el punto para mover
* **acceptableDistanceToPoint –** la distancia mínima para cambiar el punto cuando ha llegado al punto anterior
* **timeToNextLocation –** el tiempo que el enemigo espera antes de cambiar el punto
* **timer –** contador del tiempo en espera
* **hasReached –** boolean que comprueba si el enemigo esta en el punto donde ha movido

Métodos:

* **isItReached –** comprueba si ha llegado a la dirección donde estaba moviendo
* **randomMovePosition –** obtiene la posición aleatoria y la guarda en MovePosition

*Chasing*

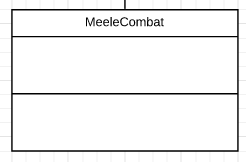
Estado para perseguir el personaje (enemigo o el jugador depende del comportamiento que queremos obtener)



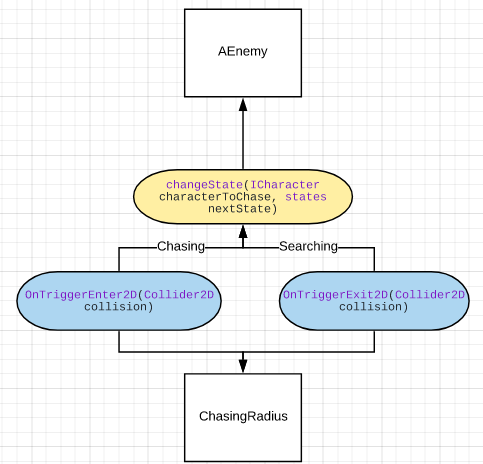
* **lookForThePlayer** – cambia MovePosition a la posición del jugador

*MeeleCombat*

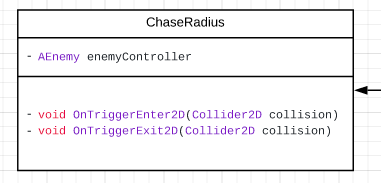
Estado para atacar el jugador. En realidad, lo que hace el estado es pare el enemigo para que pueda pegar si está en el radio del combate.



*SearcingRadius*



La clase SearcingRadius no está relacionada con la clase AEnemy fuertemente. Es decir, en Asset del enemigo tenemos un GameObject que solo contiene script SearcingRadius y el Trigger Collider. Cuando el jugador entra en dicha collider cambiamos el estado de enemy desde Searching a Chasing(Con el método changeState de AEnemy), y cuando sale cambiamos al revés.



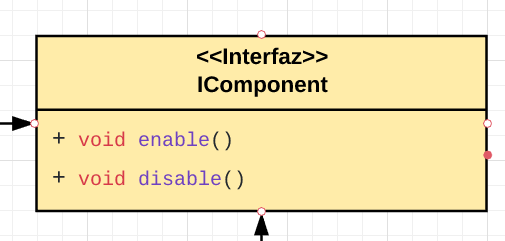
Variables:

* **enemyController –** sirve para lanzar el método de cambio del estado

Métodos:

* **OnTriggerEnter2D –** sirve para cambiar el estado a Chasing
* **OnTriggerExit2D –** sirve para cambiar el estado a Searching

*IComponent*

La interfaz IComponent sirve para los componentes que utilizan los personajes (por ejemplo, HealthComponent, AnimatiorController, Attacks de enemigos y más). 

* **Enable -** habilita el componente
* **Disable** – deshabilita el componente

*IAttack*

La interfaz IAttack nos da funcionalidad necesaria para posibles estilos de ataques. En nuestro caso tenemos solo dos ataques: en la corta distancia y disparar. El enemigo puede tener los dos. Los ataques son componentes por eso pueden ser activos o desactives.

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

* **coldownForNextAttack -** lanza un timer que retrasa siguiente ataque
* **setDamage** – cambia el daño que hace
* **setCooldown** – cambia el tiempo entre ataques
* **attack** – hace daño

Изображение выглядит как текст, карта

Автоматически созданное описание

*AAttack*

La clase abstracta tiene una funcionalidad básica para hacer ataques. Los métodos para las clases hijos tienen que ser: como se hace ataque (Attack), y la lógica como empezamos y acabamos atacar (para poner Attacking a true o false. Normalmente se hace con OnTriggerEnter o OnTriggerExit).

Изображение выглядит как снимок экрана, текст

Автоматически созданное описание

Variables:

* **damage** – el daño que hace
* **cooldown** – el tiempo entre ataques
* **canAttack** – boolean para indicar si estamos en el tiempo de retraso el ataque
* **attacking** – comprueba en cada momento (En el método Update) si ataca o no
* **active** – sirve para deshabilitar/habilitar el componente
* **timer** – Ienumerator para hacer timer de cooldown
* **enemy** – la referencia a enemigo. Sirve para cambiar el estado a MeeleCombat(cuando estamos cerca del jugador para pegar y no movemos)
* **player** – la referencia al jugador para que el recibe el daño

Métodos:

* **startCooldown** –empieza el timer entre ataques

*MeeleAttack and RangeAttack*

Las clases MeeleAttack y RangeAttack hacen dos cosas iguales, pero de la diferente manera. Ambos tienen los métodos OnTriggerEnter y OnTriggerEnter para comprobar si el jugador está en el rango de ataque. Pero MeeleAttack también cambia el estado del enemigo a MeeleCombat y RangeAttack no cambia el estado.

El método del ataque también es diferente en ambas clases. En MeeleAttack solo hace daño al jugador directamente (con el método de ICharacter – player.reciveDamage(float damage)) y en RangeAttack lanza projectile.

Изображение выглядит как снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, карта

Автоматически созданное описание

La lógica de MeeleCombat.

**Imp**

Vamos a ver un ejemplo del enemigo y como funciona todas estas clases. Por ahora solo hemos hablado de las clases en el código, pero en Unity también hay objetos que se llaman GameObject cuales hacen todos los objetos del juego (El mapa, la camera, el jugador, las frutas etc…). Y con cada GameObject podemos añadir los scripts que son nuestras clases.

Изображение выглядит как снимок экрана

Автоматически созданное описание

Aquí podemos ver 4 GameObject de cuales está compuesto nuestro imp. Son: Imp, Sprite, ChaseRadius y Meele Attack. Los nombres no tienen que coincidir de GameObject no tienen que coincidir con los scripts.

Изображение выглядит как текст, карта

Автоматически созданное описание

*IMP*

* **RIgidbody2D** – permita al objeto ser objeto físico en el juego
* **CircleCollider2D** – permita al objeto tocar a otros
* **Imp** – script principal. Hijo de AEnemy

Imp es el GameObject principal, solo el mueve, otros están fuertemente conectados a él (si el mueve otros mueven con el).

*Sprite*

* **SpriteRender** – dibuja el Sprite a la escena
* **Animator** – el componente para manejar las animaciones

*Chase Radius*

* **CircleCollider** – tiene el rango mas grande de todos los colliders de Imp. Es trigger, es decir, se puede entrar en este objeto. Hace events cuando el jugador entra en el rango o sale.
* **ChaseRadius** – el script con la clase ChaseRadius.

*Range Attack*

* **CircleCollider** – el rango de este collider está entre Meele Attack y Chase Radius. Genera eventos para lanzar los projectiles
* **ChaseRadius** – el script con la clase RangeAttack que lanza los projectiles cuando el jugador está en el CircleCollider.

*Meele Attack*

* **CircleCollider** – este collider es un poco más grande que collider principal (de Imp).
* **ChaseRadius** – el script con la clase MeeleAttack que cambia el estado del enemigo de Chasing a MeeleCombat y al revés.

Por tener un gran parte del código en el sistema de AEnemy el código de la clase imp no supera 30 lineas: Изображение выглядит как снимок экрана

Автоматически созданное описание

Y esto no significa que no podemos jugar con su comportamiento. Añadiendo y quitando los componentes podemos hacer el comportamiento necesario sin afectar al sistema en general. Por ejemplo el conejo tiene solo 25 líneas de código y su comportamiento es casi igual pero puede atacar el jugador en la distancia cerca.