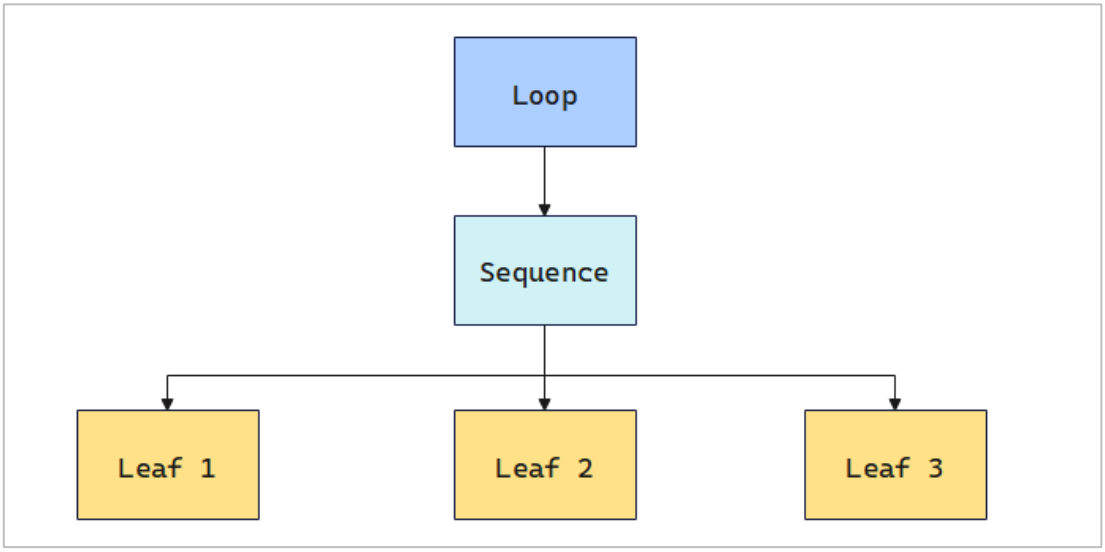
GUÍA BÁSICA DE CREACIÓN DE ÁRBOLES DE COMPORTAMIENTO

Esta guía muestra los pasos a seguir para crear el siguiente árbol de comportamiento.



Antes de crear los árboles de comportamiento hay que importar los espacios de nombres `BehaviourAPI.Core` y `BehaviourAPI.BehaviourTrees`.

Para crear un árbol de comportamiento se crea un objeto de la clase `BehaviourTree`.

```
BehaviourTree bt = new BehaviourTree();
```

Los nodos de los árboles de comportamiento se crean de abajo a arriba, empezando por los nodos hoja y terminando en el nodo raíz.

Después de crear todos los nodos se especifica cual es el nodo raíz del árbol. Si no, será el primero en haber sido creado.

```
bt.SetRootNode(root_node);
```

Por último, para ejecutar el árbol se debe lanzar el método `Start` al comienzo, y después ejecutar el método `Update` en cada iteración.

```
BehaviourTree bt = new BehaviourTree();// Al principio de la ejecución se crea el grafo.
bt.Start();  // En el primer frame se inicia la ejecución del grafo (p.e. Start en Unity).
bt.Update(); // En cada bucle de ejecución se actualiza el grafo (p.e. Update en Unity).
```

***NOTA**: Estos métodos sólo deben ejecutarse de esta forma en el grafo principal, no en los subgrafos.*

Nodos hoja

Los nodos hoja ejecutan una acción y actualizan su estado interno (Status) en funcion del resultado de la acción. Para crear un nodo hoja primero se debe crear la acción que ejecuta:

Como crear acciones:
Para crear la acción hay que añadir el espacio de nombres `BehaviourAPI.Core.Actions` y crear el objeto:

```
FunctionalAction action = new FunctionalAction(StartMethod, UpdateMethod, StopMethod);
```

Los parámetros del constructor de `FunctionAction` son métodos que se ejecutarán cuando el estado comience su ejecución, en cada frame y cuando termine respectivamente. Los métodos `Start` y `Stop` deben devolver `void` y el método `Update` debe devolver `Status`.

Se pueden crear objetos de la clase FunctionalAction especificando solo algunos de los 3 métodos. Si no se especifica el método Update, la acción devolverá siempre Running y no terminará nunca.

Después de crear la acción se crea el nodo hoja pasando como parámetro la acción creada anteriormente. El valor Status del nodo hoja se actualiza con el valor devuelto por la acción.

En el ejemplo de la guía se crean tres nodos hoja, cada uno con su propia acción.

```
LeafNode leaf1 = bt.CreateLeafNode(action1);
LeafNode leaf2 = bt.CreateLeafNode(action2);
LeafNode leaf3 = bt.CreateLeafNode(action3);
```

Nodos compuestos

Los nodos compuestos ejecutan cada uno de sus hijos uno a uno hasta que devuelven un determinado valor. Para crear un nodo compuesto se usa el método `CreateComposite`, especificando el tipo en el parámetro genérico. Como argumentos se pasan la lista de nodos hijos y un booleano que indica si el orden de ejecución será aleatorio o no.

En el ejemplo de la guía, se crea un nodo secuencia cuyos hijos son los tres nodos hoja creados antes.

```
SequencerNode sequence = bt.CreateComposite<SequencerNode>(false, leaf1, leaf2, leaf3);
```

A continuación, se explica cómo crear cada tipo de nodo compuesto y cómo funciona.

Nodo secuencia

Nodo compuesto que ejecuta sus hijos hasta que uno de ellos devuelva `Failure` o hasta que haya ejecutado todos, en cuyo caso devolverá `Success`.

```
SequencerNode composite = bt.CreateComposite<SequencerNode>(false, nodo1, nodo2, ...);
```

Nodo selector

Al contrario que el nodo secuencia, este ejecuta sus hijos hasta que uno devuelva `Success` o hasta que haya ejecutado todos, en cuyo caso devolverá `Failure`.

```
SelectorNode composite = bt.CreateComposite<SelectorNode>(false, nodo1, nodo2, ...);
```

Se incluyen más tipos de nodos compuestos. Para ver todos, consulta la documentación completa.

Nodos decoradores

Los nodos decoradores sirven para modificar el valor Status del nodo hijo. Para crear un nodo decorador se usa el método CreateDecorator especificando el tipo en el parámetro genérico y pasando como argumento el nodo hijo.

En el ejemplo de la guía, se crea un nodo decorador bucle como padre del nodo secuencia creado en el paso anterior.

```
LoopNode loop = bt.CreateDecorator<LoopNode>(sequence);
```

A continuación, se muestra cómo se crean los distintos tipos de nodos decoradores y para qué sirve cada uno.

Nodo inversor

Devuelve el resultado del nodo hijo invertido, devolviendo Success si el estado del hijo era `Failure` y viceversa.

```
InverterNode decorator = bt.CreateDecorator<InverterNode>(childNode);
```

Nodo de éxito

Modifica el estado de su nodo hijo para devolver `Success` independientemente de si ha recibido `Success` o `Failure`.

```
SucceederNode decorator = bt.CreateDecorator<SucceederNode>(childNode);
```

Nodo bucle

Ejecuta el nodo hijo un número determinado de veces, de forma que siempre devuelve `Running`, hasta que han terminado todas las iteraciones. Se puede especificar el número de iteraciones en la variable `Iterations` o usando el método `SetIterations`. Por defecto el número de iteraciones es -1, lo que equivale a infinitas.

```
LoopNode loop = bt.CreateDecorator<LoopNode>(childNode);
loop.Iterations = 5;
```

Nodo bucle hasta éxito o fallo

Ejecuta el nodo hijo hasta que devuelve el valor especificado (`Success` o `Failure`). Se puede especificar el valor que debe devolver el hijo para que termine el bucle (por defecto `Success`), y un número máximo de iteraciones (por defecto -1 o infinitas).

```
LoopUntilNode loopUntil = bt.CreateDecorator<LoopUntilNode>(childNode);
loopUntil.TargetStatus = Status.Failure;
loopUntil.MaxIterations = 5;
```

Nodo temporizador

Espera un tiempo determinado para ejecutar su nodo hijo. El tiempo se especifica en la variable “Time” (en segundos).

```
TimerDecoratorNode timer = bt.CreateDecorator<TimerDecoratorNode>(childNode);
timer.Time = 10f;
```

Nodo condicional

Cuando comienza a ejecutarse comprueba una condición. Si la condición no se cumple, devolverá `Failure` y no ejecutará su nodo hijo. En cambio, si se cumple, lo ejecutará y devolverá su valor `Status` directamente. La condición se crea en forma de percepción y se especifica en la variable `Perception`.

Como crear percepciones:

Para crear la percepción hay que añadir el espacio de nombres `BehaviourAPI.Core.Perceptions` y crear el objeto:

```
ConditionPerception perception = new ConditionalPerception(InitMethod, CheckMethod, ResetMethod);
```

Las percepciones, al igual que las acciones, se crean con tres métodos que sirven para inicializar, comprobar y resetear la percepción respectivamente. Los métodos `Init` y `Reset` deben devolver `void` y el método `Check` debe devolver un valor `bool`.

También es posible especificar solo algunos de los métodos y si el método `Check` no se especifica devolverá `false`.

Existen otros tipos de percepciones:

- Percepciones compuestas: Realizan una operación lógica con el resultado de otras percepciones.

```
AndPerception and = new AndPerception(perception1, perception2, …);
OrPerception or = new OrPerception(perception1, perception2, …);
```

- Percepciones temporales: Devuelven false hasta que pasa una cantidad determinada de tiempo (en segundos).

```
TimerPerception timer = new TimerPerception(5);
```

- Percepciones de ejecución: Comprueba si el valor status de un nodo o grafo es igual a un valor. Puede servir para comunicar sistemas de comportamiento de distintos agentes.

```
Node node = …;
BehaviourGraph graph = …;
ExecutionStatusPerception stPerception = new ExecutionStatusPerception(node, StatusFlags.Running);
ExecutionStatusPerception stPerception = new ExecutionStatusPerception(graph, StatusFlags.Running);
```

```
ConditionNode condition = bt.CreateDecorator<ConditionNode>(childNode);
Condition.Perception = new ConditionalPerception(InitMethod, CheckMethod, Reset);
```

Se incluyen más tipos de nodos decoradores. Para ver todos, consulta la documentación completa.

Crear subgrafos con árboles de comportamiento.

Crear un subgrafo dentro de un árbol de comportamiento

Los árboles de comportamiento permiten ejecutar subgrafos dentro de los nodos hoja. Para ello se debe crear una acción de tipo `SubsystemAction` pasando como argumento el subgrafo deseado. Después se pasará esta acción al nodo hoja correspondiente.

```
BehaviourTree mainTree = new BehaviourTree();
BehaviourGraph subgraph = ...;
SubsystemAction action = new SubsystemAction(subgraph);
LeafNode leaf = mainTree.CreateLeafNode(action);
```

El nodo hoja actualizará su valor `Status` con el valor del subgrafo, por lo que cuando el subgrafo termine su ejecución y su estado interno cambie a `Success` o `Failure`, el árbol padre continuará su ejecución al siguiente nodo.

Salir de un subárbol de comportamiento

Para salir de un subgrafo de tipo `BehaviourTree` basta con hacer que termine su ejecución, cuando el nodo raíz devuelva `Success` o `Failure`. Este valor será el que devuelva la acción que contiene al árbol.