Demo: Behavior Tree in Unity with Behavior Bricks

Material in this demo courtesy of

- Aversa, Davide. *Unity Artificial Intelligence Programming*, Fifth Edition. Packt Publishing, March 2022. Chapter 9.
- https://bb.padaonegames.com/doku.php?id=quick:design
- https://bb.padaonegames.com/doku.php?id=quick:program

In AI, finite state machines (FSMs) have a fatal flaw: it is tough to make them scale once there are many states and transitions.

Behavior trees (BTs) are an improvement; they are fast, provide reusability, and are easy to maintain. After their introduction in 2004 with Halo 2, they quickly became the preferred decision-making technique in games.

See the first example later in this document.

When we execute a BT's node, the node can return three states: **success**, **failure**, or **running** (if the node's execution is spread over multiple frames, for instance, if it plays an animation). When the BT executor runs a tree, it starts from the root and **executes every node in order**, according to rules written in the nodes themselves.

A node can be of three types:

- A **task** (a node without children), also called a **leaf**.
- A **decorator** (a node with a single child)
- A **composite** (a node with multiple children)

In general, **leaves** represent the Action that the characters can do or know (that is why they are commonly called an Action or Task); they may be actions such as GoToTarget, OpenDoor, Jump, or TakeCover, but also things like IsObjectNear? or IsHealthLow?. These actions depend on the character, the game, and the general game implementation.

A **decorator** is a node that modifies (decorates) the sub-tree under it (therefore, it can decorate both composite and task nodes). Some standard decorators:

- **Negate** node - inverts the return value of the sub-tree; for instance, if the sub-tree returns Success, the decorator returns Failure and vice versa (of course, if the sub-tree returns Running, the decorator returns Running as well).
- **Repeat** node - repeats its sub-tree a certain number of times.

A composite node represents a node with multiple children, and it is the most interesting case. There are two common composite nodes:

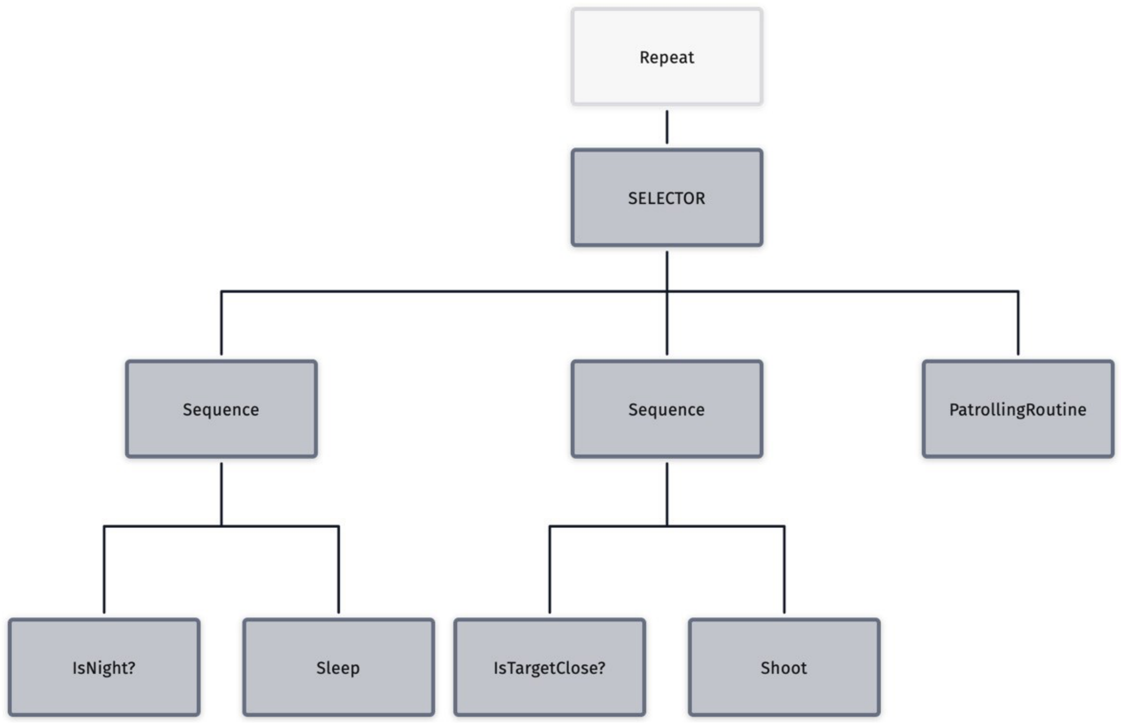
- **Sequence**, which runs all its children in order and returns *Success* if—and only if—all its children return *Success*,
- **Selector**, which tries to execute all its children in order but returns *Success* as soon as one of its children returns *Success*.
- many BT implementations contain many more composite nodes (such as nodes that run their children in parallel or according to some dynamic priority value).

Of course, this tree structure is not enough. Nodes need to exchange information with each other or with the game world. For instance, a GoToTarget node needs to know the target and its location; an IsObjectClose? node needs to know which object we are referring to and what distance we consider close. Naturally, we could write a GoToX node for each object in the game (such as GoToTree01 and GoToDoor23), but you can easily imagine that this becomes messy very quickly.

For this reason, all the BT implementations contain a data structure called **Blackboard**. As in a real-life physical blackboard, **every node can write and read data into it**; we just need to specify where to look for each node.

A simple example – a patrolling robot

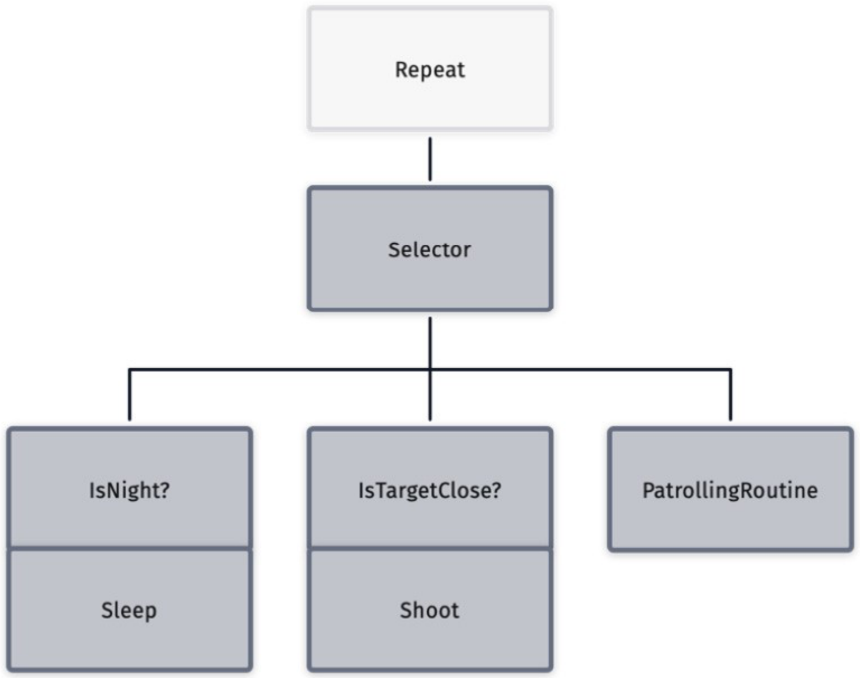
Let's look at this example: a patrolling robot that shoots anything that gets near it but works only during the daytime. We show the possible BT for this kind of agent in the following diagram:



Let's run this BT, assuming that the target is close and it is not night:

1) The first node is a **Repeat** decorator; it does nothing but cycle the BTs.
2) The **SELECTOR** node starts executing its first child; we go down to the "Sequence" on the left.
3) We are now at the first **Sequence** node; again, we execute the first node. **IsNight**? returns Failure (because it is not night!). Whenever one node returns Failure, the whole Sequence node returns Failure.
4) We traverse back up the tree to the **SELECTOR** node; now, we go to the second branch.
5) Again, we execute a **Sequence** (the one in the middle of the diagram).
6) This time, however, **IsTargetClose?** returns *Success*, so we can proceed to the next node, **Shoot**, which runs a game function spawning an in-game projectile.

The pattern of **Sequence | Condition | Action** is equivalent to: "if Condition is Success then Action." This pattern is so common that many BT implementations allow you to stack the Condition and the Action together. Therefore, we can rewrite the tree as follows:



Hands-On

Let's implement a simple BT in Unity.

The **Behavior Bricks** plugin for Unity gives you an easy interface for working with Behavior Trees.

1. Go online to the **Assets Store** and **add** the **Behavior Bricks** package (see Chapter 9) to My Assets.
2. Restart Unity Hub and make a new 3D project in Unity.
3. Import **Schurter_starter_kit.unitypackage.** Once it imports, from Assets > Scenes, open the scene named **Basic-01**. It gives us an Arena populated with:

    a. a **First PersonController** character we can move around using WASD keys;

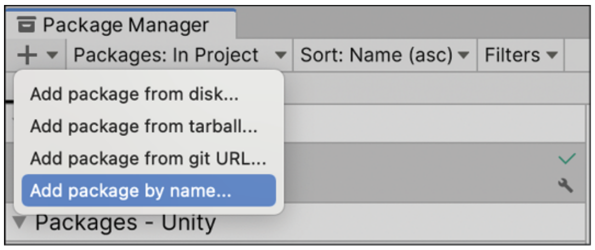
    b. an enemy that can chase us in the form of a red **SphereNPC1** positioned at x4, y0.5, z4 with a Rigidbody, Mass 3, Drag 3;

    c. and a similar blue **SphereNPC2** that just sits there and will distract the red sphere by the time we're done.

4. In Unity, **make the Arena into a NavMesh**:
    a. Go to Window > **Package Manager**

    b. Click the + sign in the upper-left corner and select **Add package by name**:

    c. Enter **com.unity.ai.navigation** and click Add.



5. Once the package has been added, select the Arena, click on Add Component in the Inspector window, and choose Navigation > **NavMeshSurface**.

6. With the Arena still selected, in the Inspector find NavMeshSurface and click the "**Bake**" button. A NavMesh appears in your Project folder in a Scenes > Basic-01 folder.

7. Select **SphereNPC1** and in the Inspector, Add Component > Navigation > **Nav Mesh Agent**, and set its **Speed = 2**.

8. Be sure you have the Behavior Bricks package in your Assets (After adding it to My Assets in the Assets Store, to see it in my Package Manager under "My Assets," I had to completely close out of Unity and Unity Hub, then restart).

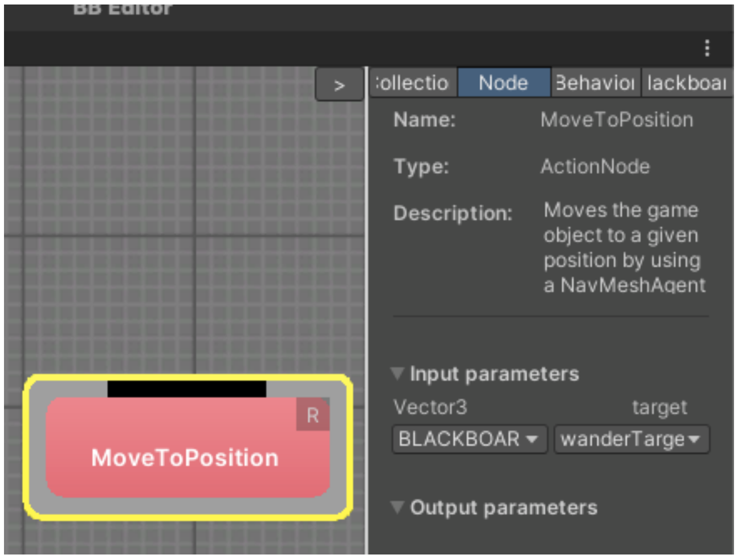
9. In the Project panel it is recommended you make a **Behaviors folder** in Assets.

10. Inside that new folder right-click > **Create > Behavior Tree** (it's at the bottom of the list, probably). I named mine **NPC1_Behavior.asset**.

11. If it doesn't open a BB Editor panel, double-click the new behavior to open it.

12. **Right-click** (I had trouble with this until I used an actual mouse) the big blank grid area of the BB Editor panel and a list appears. In the Actions > Navigation category choose to create a **MoveToPosition** action.

13. With MoveToPosition selected, in the righthand side in the BB Inspector, go to the Node tab and in the **input parameters** use the combobox where CONSTANT is selected by default. Change it to BLACKBOARD. Name it **wanderTarget** and click the **Create** button.



Name: MoveToPosition
Type: ActionNode
Description: Moves the game object to a given position by using a NavMeshAgent

Input parameters
Vector3 target
BLACKBOAR wanderTarge

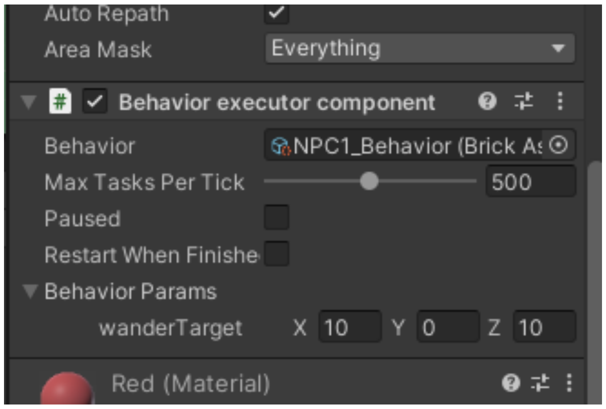
Output parameters

14. Now in the Blackboard tab of the BB Inspector we can find the new behavior input parameter with that name. You can change the value for the parameter to x10, y0, z10, or set it as instructed later...

15. Back in the Unity Hierarchy, select SphereNPC1 and in the Inspector **Add Component > Behavior Bricks > Behavior Executor Component**.

16. Looking at that "Behavior Executor Component" section in the Inspector, drag the behavior from the Project panel to **fill the "Behavior" field** in the Inspector.

17. Also in the Inspector's "Behavior Executor Component" section fill in public parameters you've set up... **set wanderTarget to x10, y0, z10.**



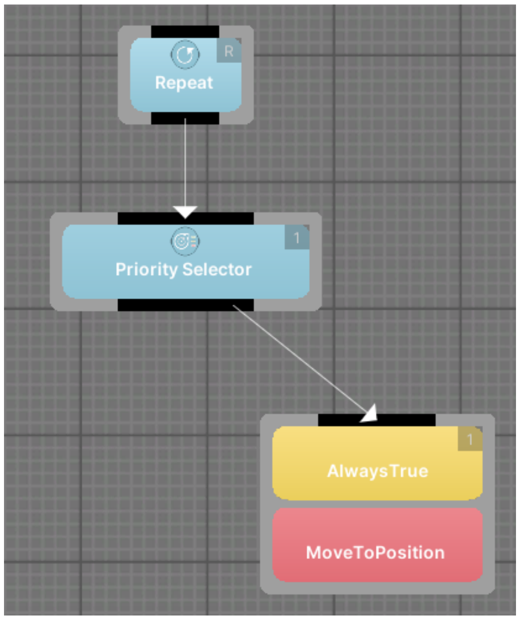
Auto Repath ✓
Area Mask: Everything
Behavior executor component
Behavior: NPC1_Behavior (Brick A:
Max Tasks Per Tick: 500
Paused
Restart When Finishe
Behavior Params
wanderTarget X 10 Y 0 Z 10
Red (Material)

18. Test the project. The red sphere goes to the desired location and stops.

19. Back in the BB Editor panel, right-click the big grid area and from the list choose to add a **Composites > RepeatDecorator**. Move it near the top.

20. Right-click the big grid area and from the list choose to add a **Composites > Priority Selector Node.** Move it between the other two nodes.

21. Click the bottom black edge of the **Repeat** node then move the mouse to pull an arrow to **connect it** to the top black edge of the **Priority Selector**.



22. Drag the bottom black edge of the **Priority Selector** Node to **connect it** to the top black edge of the **MoveToPosition**. Note the yellow AlwaysTrue condition node that appears on top of the MoveToPosition action. Every child of a priority must be a guarded node composed of a condition plus an action (or behavior). Therefore when a Priority node is connected to a MoveToPosition node, a default AlwaysTrue condition is automatically added to guard the MoveToPosition action node.

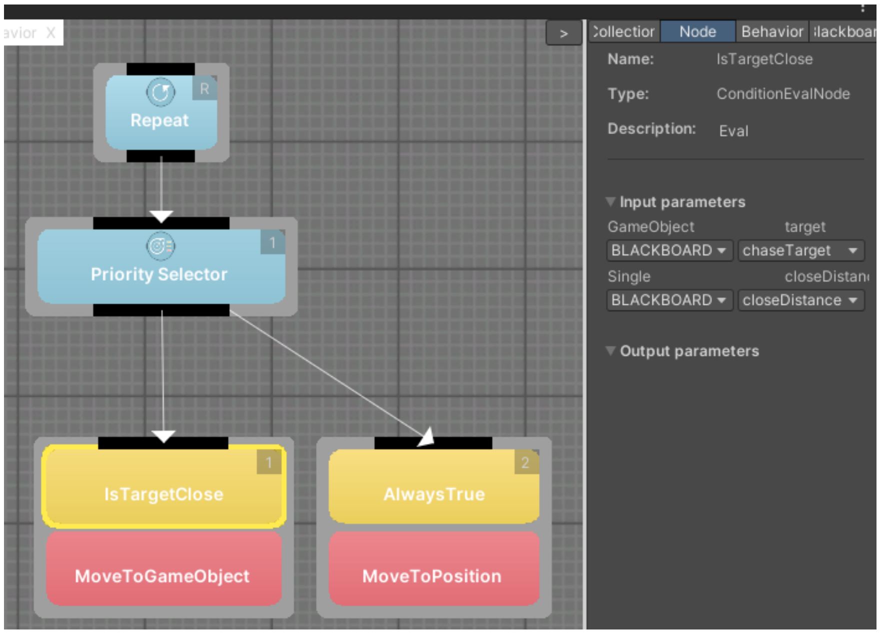
23. We're going to get the red sphere to chase the player if the player is close, but otherwise return to the 10 0 10 position: Right-click the big grid area and from the list choose to add an **Actions > Navigation > MoveToGameObject**.

24. With MoveToGameObject selected, in the righthand side in the BB Inspector, go to the Node tab and in the input parameters note the combobox where CONSTANT is selected by default. Change it to **BLACKBOARD**. Name the target **chaseTarget** and click the **Create** button.

25. Drag the bottom black edge of the **Priority Selector** Node to **connect it** to the top black edge of the **MoveToGameObject**. Note the yellow AlwaysTrue condition node that appears on top of the MoveToGameObject action.

26. Right-click that yellow condition and choose "**Replace with...**". Choose to replace it with **Conditions > Perception > IsTargetClose**.

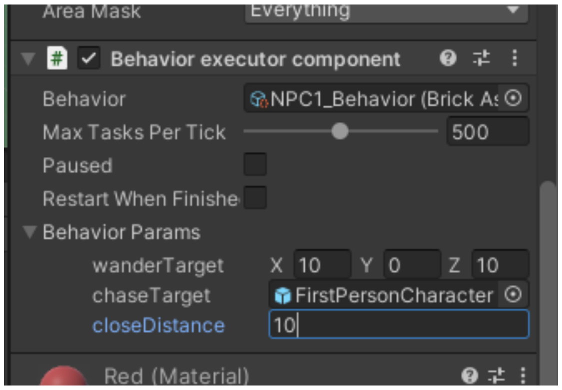
27. With **IsTargetClose** selected, in the righthand side in the BB Inspector, go to the Node tab and in the input parameters note the combobox where CONSTANT is selected by default. Change it to BLACKBOARD. In the target field choose **chaseTarget** (you created it earlier). Also, for the **closeDistance**, change it to **BLACKBOARD**, and name the target **closeDistance** and click the **Create** button.



Name: IsTargetClose
Type: ConditionEvalNode
Description: Eval

GameObject target
BLACKBOARD chaseTarget
Single closeDistance
BLACKBOARD closeDistance

28. Back in the Unity Hierarchy, select SphereNPC1 and in the Inspector's "Behavior Executor Component" section fill in some parameters: drag **FirstPersonCharacter** from the Hierarchy to the **ChaseTarget** field, and set **closeDistance = 10**.



Behavior: NPC1_Behavior (Brick As
Max Tasks Per Tick: 500
Paused
Restart When Finishe
Behavior Params
wanderTarget X 10 Y 0 Z 10
chaseTarget FirstPersonCharacter
closeDistance 10

29. Test the project. The red sphere keeps wanting to go to the location, but if you get near it, it comes after you!

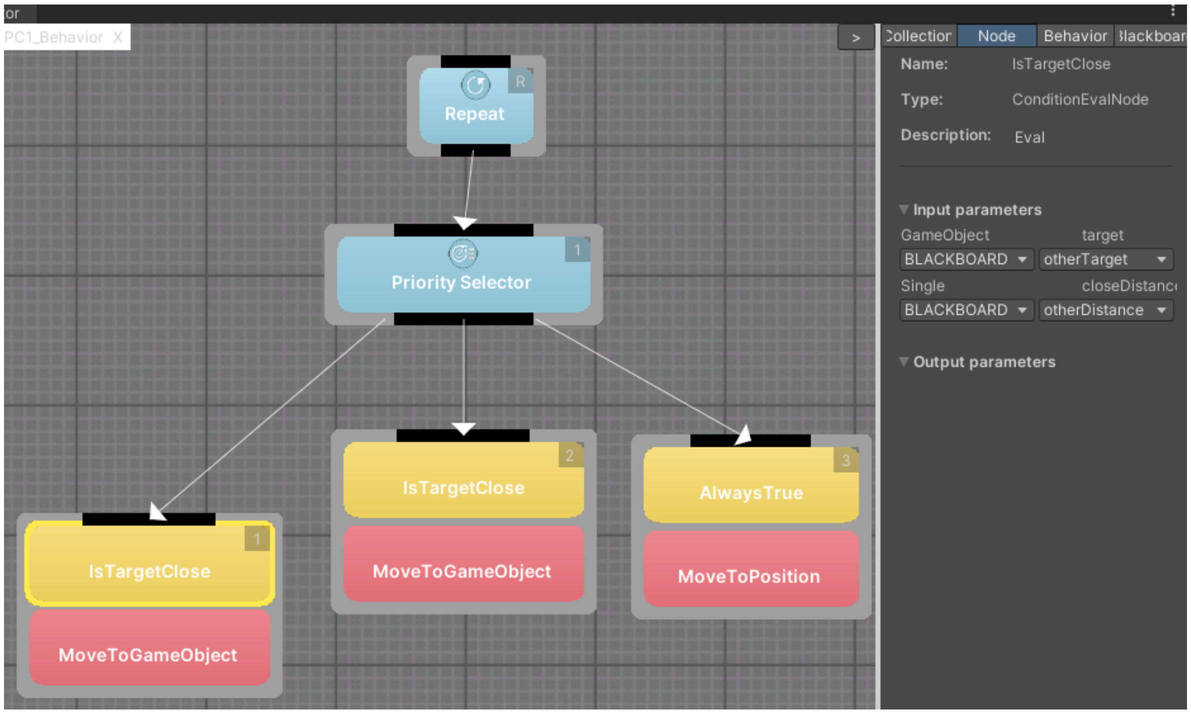
30. Now we'll add chasing the blue sphere to the red sphere's list of priorities, too: Return to the BB Editor panel, right-click the big grid area and choose to create **ANOTHER Actions > Navigation > MoveToGameObject** action.

31. With **MoveToGameObject** selected, in the righthand side in the BB Inspector, go to the Node tab and in the input parameters change CONSTANT to **BLACKBOARD**. Where it lets you name the target, choose "new parameter" and name it **otherTarget** and click the **Create** button.

32. Drag the bottom black edge of the **Priority Selector** Node to **connect it** to the top black edge of the newer **MoveToGameObject**. Note the yellow AlwaysTrue condition node that appears on top of the MoveToGameObject action.

33. Right-click the new node's default yellow AwaysTrue condition and choose "Replace with...". Choose to replace it with **Conditions > Perception > IsTargetClose**.

34. With **IsTargetClose** selected, in the righthand side in the BB Inspector, go to the Node tab and in the input parameters change CONSTANT to **BLACKBOARD**. For its parameter choose **otherTarget** (you created it earlier). Also, for the closeDistance, change it to **BLACKBOARD**, and where it lets you name the target, choose "new parameter" and name it **otherDistance** and click the **Create** button.

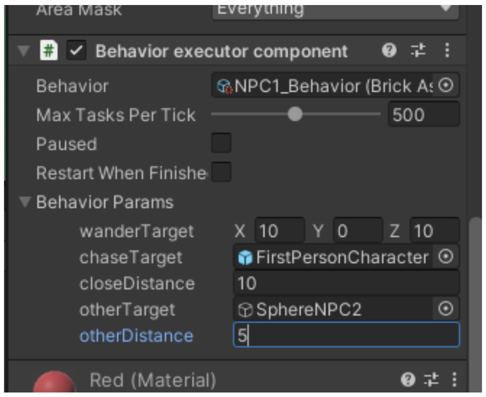


Name: IsTargetClose
Type: ConditionEvalNode
Description: Eval

Input parameters
GameObject target
BLACKBOARD otherTarget
Single closeDistance
BLACKBOARD otherDistance

Output parameters

35. Back in the Unity Hierarchy, select SphereNPC1 and in the Inspector's "Behavior Executor Component" section fill in some parameters: drag **sphereNPC2** (the blue one) from the Hierarchy to the **OtherTarget** field, and set **otherDistance = 5**.



Behavior executor component

Behavior	NPC1_Behavior (Brick As
Max Tasks Per Tick	500
Paused	
Restart When Finishe	
Behavior Params	
wanderTarget	X 10 Y 0 Z 10
chaseTarget	FirstPersonCharacter
closeDistance	10
otherTarget	SphereNPC2
otherDistance	5

36. Test the project. The red sphere keeps wanting to go to the location, but if you get near it, it comes after you, and if you lead it near the blue sphere it goes after the blue sphere... it's got its priorities!

How can you get the red sphere to prefer to come after you? In the BB Editor, drag the grey frames around; if the MoveToGameObject dealing with the Player is farther left, its priority number changes (the little number at its top right).

Creating a Custom Action

You need to know that some actions are categorized as an ***immediate*** action: it finishes in the same game loop it starts. The opposite are ***latent*** actions, that last for multiple game loops. MoveToPosition or MoveToGameObject are latent; they take repeated increments to complete. When programming a new action, note that difference.

Please note the name difference between the Behavior Bricks **OnUpdate()** method and the common **MonoBehavior::Update():** this difference is deliberate. We will need to use the class **BasePrimitiveAction,** which does not inherit from MonoBehavior at all.

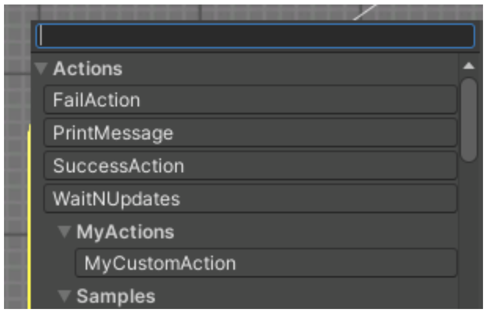
OnUpdate() must return a **TaskStatus** value. This will inform the execution engine if the action has finished its execution or should still be invoked in the next game loop.

1. In a MyActions folder, create a new C# script and name it **myCustomAction.cs**.
2. Open it into your preferred editor and substitute the code:

```
using UnityEngine;

using Pada1.BBCore; // Code attributes
using Pada1.BBCore.Tasks; // TaskStatus
using Pada1.BBCore.Framework; // BasePrimitiveAction

[Action("MyActions/MyCustomAction")]
[Help("Runs several steps of an imaginary procedure")]
public class MyCustomAction : BasePrimitiveAction
 {
        // Define the input parameter myCustomInputParameter,
        // and provide a default value of 23.0
        // when used as CONSTANT in the editor.
        [InParam("myCustomInputParameter", DefaultValue = 23f)]
        public float myCustomInputParameter;

        // Main class method, invoked by the execution engine.
        public override TaskStatus OnUpdate()
        {
                // Put some typical code procedure here, whatever you want.
                Debug.Log("Step one of some imaginary procedure");
                Debug.Log("Step two of some imaginary procedure");
                Debug.Log("Step three of some imaginary procedure, this time showing " +
        myCustomInputParameter);
                // The action is completed. We must inform the execution engine.
                return TaskStatus.COMPLETED;
        } // OnUpdate
} // class MyCustomAction
```

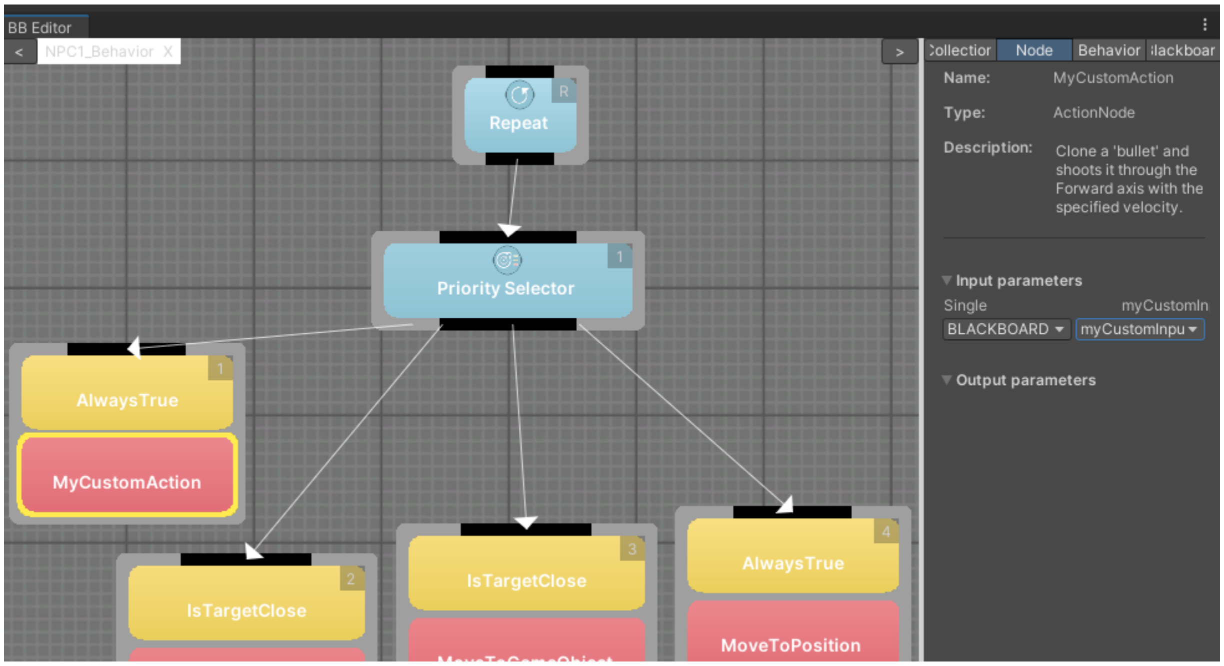
Class fields like myCustomInputParameter each get the **InParam** attribute to ensure that they will be accessible from the Behavior Bricks editor. If we put it into the Blackboard, the default value of 23.0 will vanish; it is only used when the parameter is set using a constant in the editor. But we can add it back in Blackboard, or in the Inspector.

3. Now we'll use our custom action in the Behavior Bricks Editor: Right-click the big grid area and in the Actions > MyActions category choose to create a `myCustomAction` action.



4. With the `myCustomAction` node selected, in the righthand side in the BB Inspector, go to the Node tab and in the input parameters change CONSTANT to BLACKBOARD. From the dropdown on the right choose "New Parameter," and name it `myCustomInputParameter`. Click "Create."

5. Drag the bottom black edge of the **Priority Selector** Node to **connect** to the `myCustomAction`. Position the `myCustomAction` node just to the left of the other nodes.



Name: MyCustomAction

Type: ActionNode

Description: Clone a 'bullet' and shoots it through the Forward axis with the specified velocity.

Input parameters

Single myCustomIn

Output parameters

6. Back in the Unity Hierarchy, select SphereNPC1 and in the Inspector's "Behavior Executor Component" section you can fill in a value for the `myCustomInputParameter` parameter you set up.

7. Test the project. The red sphere just sits and keeps logging messages to the console. But that's because our custom action tells it to!

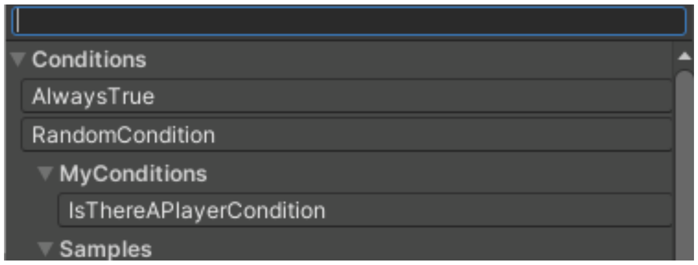
Experiment in the BB Editor Make your custom action less of a priority and play the game again. Change its yellow AlwayTrue condition to a different condition and play the game again.

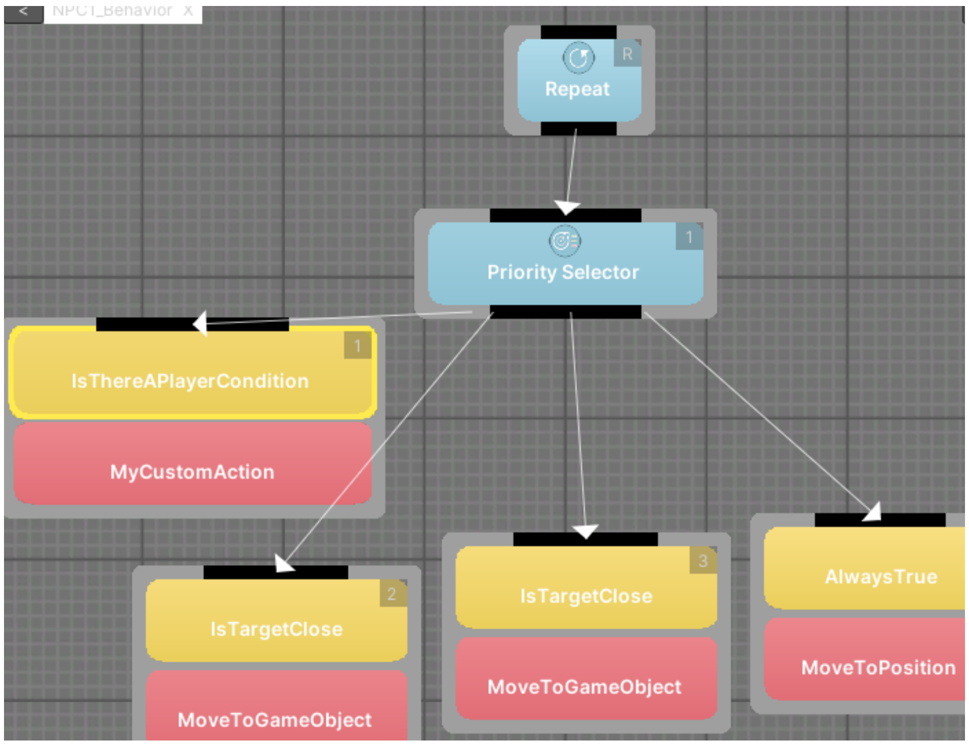
Creating a Custom Condition

1. In Assets, create a MyConditions folder, and in it create a new C# script and name it **IsThereAPlayerCondition.cs.**
2. Open it into your preferred editor and substitute the code:

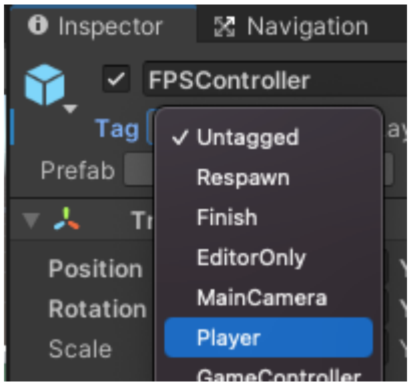
```
using UnityEngine;
using Pada1.BBCore;           // Code attributes
using Pada1.BBCore.Framework; // ConditionBase
using Pada1.BBCore.Tasks;     // TaskStatus
using System.Collections.Generic;

/// <summary>
/// IsThereAPlayerCondition is a custom condition. It inherits from ConditionBase and Checks whether there is a
GameObject tagged "Player". If so, it does the action, naturally preventing the Object this is attached to from
doing other actions.
/// </summary>
[Condition("MyConditions/IsThereAPlayerCondition")]
[Help("Checks whether there is a GameObject tagged 'Player'.")]
public class IsThereAPlayerCondition : ConditionBase
{

    /// <summary>
    /// This method overrides the one built into the ConditionBase class... in fact, you HAVE to override it. It
runs a custom function defined farther down, whch returns true or false.
    /// </summary>
    /// <returns>True if there is a GameObject tagged 'Player'.</returns>
    public override bool Check()
    {
        Debug.Log("We're running Check()");
        return searchForPlayer();
    }

    /// <summary>
    /// This method overrides the one built into the ConditionBase class. It is invoked by the execution engine 
when the condition is used in a priority selector and its last value was false.
    /// </summary>
    /// <returns>It must return COMPLETED when the value becomes true. In other cases, it can return RUNNING if 
the method should be invoked again in the next game cycle, or SUSPEND if we will be notified of the change 
through any other mechanism.</returns>
    public override TaskStatus MonitorCompleteWhenTrue()
    {
        Debug.Log("MonitorCompleteWhenTrue is running");
        // If there is a GameObject tagged 'Player'...
        if (Check())
        {
            Debug.Log("MonitorCompleteWhenTrue says TaskStatus.COMPLETED");
            //Stop running function
            return TaskStatus.COMPLETED;
        }
        // Otherwise allow this condition to be called again next Update()
        Debug.Log("MonitorCompleteWhenTrue says TaskStatus.RUNNING");
        return TaskStatus.RUNNING;
    }

    /// <summary>
    ///Similar to MonitorCompleteWhenTrue, but used when the last condition value was true and the execution 
engine is checking that it has not become false.
    /// </summary>
    /// <returns>It must return FAILED when the value becomes false. In other cases, it can return RUNNING if the 
method should be invoked again in the next game cycle, or SUSPEND if we will be notified of the change through 
any other mechanism.</returns>
    public override TaskStatus MonitorFailWhenFalse()
    {
        Debug.Log("MonitorFailWhenFalse is running");
        // If there is NO GameObject tagged 'Player'...
        if (!Check())
        {
            // Return the news that the condition is false.
            Debug.Log("MonitorFailWhenFalse says TaskStatus.FAILED");
            return TaskStatus.FAILED;
        }
        // Otherwise allow this condition to be called again next Update()
        Debug.Log("MonitorFailWhenFalse says TaskStatus.RUNNING");
        return TaskStatus.RUNNING;
    }

    // This custom method looks through the game for something with the tag 'Player.' It returns true if Player 
was found, false if not.
    private bool searchForPlayer()
    {
        GameObject playerGO = GameObject.FindGameObjectWithTag("Player");

        if (playerGO == null)
        {
            return false;
        }
        return playerGO != null;
    }
}
```

37. Right-click the `myCustomAction` node's default yellow AwaysTrue condition and choose "Replace with...". Choose to replace it with **Conditions > MyConditions > IsThereAPlayerCondition.cs.**





38. Back in the Unity Hierarchy, select **FPSController** and in the Inspector, in the top left corner change the "**Tag**" dropdown to say "**Player**" instead of "Untagged."



39. Test the project. The red sphere just sits and keeps logging messages to the console while it thinks there's a "Player" around. in the Inspector, in the top left corner change the "**Tag**" dropdown for **FPSController** and to say "**Untagged**" instead of "Player." The red sphere comes after you now that there's nobody with a Player tag. Your custom condition worked!