# GUÍA BÁSICA DE API DE SISTEMAS DE COMPORTAMIENTO EN UNITY

## GUÍA DE UNITY TOOLKIT

*Behaviour API Unity Toolkit* son un conjunto de herramientas que facilitan la creación y el uso de sistemas de comportamiento en *Unity*, aunque su uso no es obligatorio. A continuación, se explican los principales elementos que contiene:

### *BehaviourRunner*

Se trata de un *Script* preconfigurado para ejecutar un sistema de comportamiento. En lugar de crear una clase desde 0, es posible crear una que herede de *BehaviourRunner* y así solo es necesario sobrescribir el método *CreateGraph* sin tener que implementar cuando se lanzan los eventos del sistema de comportamiento, ya que el *script* se encarga de asociarlos a los eventos de Unity (*Start, Update, OnEnable*, etc).

El script tiene dos parámetros configurables:

- *ExecuteOnLoop:* Si se activa, el sistema se reiniciará cuando termine su ejecución.
- InterruptOptions: Decide que eventos se lanzan cuando se habilite y deshabilite el script. Por defecto no se lanzará ningún evento.

***Importante**:* Los métodos correspondientes a los eventos de Unity, es decir *Awake, Start, Update, OnEnable* y *OnDisable* no deben implementarse en los *Scripts* que hereden de *BehaviourRunner*. En su lugar se deben sobrescribir los métodos de dicha clase (*Init, OnStarted, OnUpdated, OnUnpaused* y *OnPaused* respectivamente)

### *UnityExecutionContext*

El contexto de ejecución es objeto compartido entre todos los nodos, acciones y percepciones de un sistema de comportamiento. Este *toolkit* incluye la clase *UnityExecutionContext* que permite usar esta funcionalidad para que las acciones y percepciones accedan al *GameObject* que contiene el sistema y a sus componentes.

La clase *BehaviourRunner* utiliza por defecto esta funcionalidad, pero si se usa otra clase, se debe crear una instancia de *UnityExecutionContext* pasando al constructor el propio script y después de crear el sistema usar el método *SetExecutionContext* sobre el grafo principal. Si el sistema usa subgrafos no es necesario repetir el proceso en estos.

```
BehaviourGraph graph = …
UnityExecutionContext context = new UnityExecutionContext(this);
graph.SetExecutionContext(context);
```

### *UnityActions* y *UnityPerceptions*

Se han creado tipos específicos de acciones y percepciones para usar el contexto de ejecución de Unity. Para usarlas se crea una clase que herede de *UnityAction* o *UnityPerception*. Estas clases añaden dos características.

- Contienen una propiedad *context* para poder acceder al objeto de la escena y a su componente. Por ejemplo, es posible modificar la posición del agente usando *context.Transform*.
- Contiene el método *OnSetContext* que se ejecuta una sola vez cuando se propaga el contexto de ejecución por los nodos. Este método puede sobrescribirse en las subclases para guardar referencias a componentes del objeto y así evitar tener que usar el método *GetComponent* cada vez que se necesite ese componente.

```
public abstract class ExampleUnityAction : UnityAction
{
    private MyComponent _myComponent;

    protected override void OnSetContext()
    {
          _myComponent = context.GetComponent<MyComponent>();
    }

    public Status Update()
    {
          if (_myComponent.boolProperty)
          {
              _myComponent.ActionMethod();
              return Success;
          }
          return Running;
    }
}
```

# GUÍA DE USO DE SMART OBJECTS EN UNITY

Los SmartObjects son elementos que pueden ser usados por personajes o agentes, siendo el propio objeto el que establece como se usa. Además, los *SmartObject* sirven para cubrir necesidades definidas en los agentes.

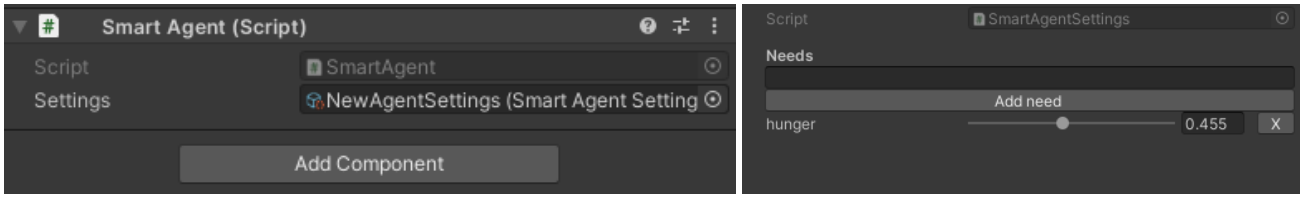
El uso de los *SmartObject* se basa en peticiones y respuestas. El sistema de comportamiento solicita al objeto su uso y este le responde con una interacción, que contiene la acción que el agente debe completar para usar correctamente el objeto.

Los elementos principales del modelo de *SmartObjects* son el agente, los objetos y las acciones de petición.

## Smart Agent

El agente es el elemento principal que permite interactuar con los *SmartObjects.* Para crear uno hay que añadir a un objeto el componente *SmartAgent*. Después se crea una configuración para el agente haciendo clic derecho sobre la vista del proyecto y seleccionado *Create > BehaviourAPI > SmartObjects > Smart Agent Settings*. Al hacer clic sobre el archivo generado se abrirá su inspector y podremos añadir y borrar necesidades, así como modificar su valor inicial.

Una vez creado el archivo de configuración se arrastra hasta el campo *Settings* del agente.




## Smart Objects

Para añadir un nuevo tipo de objeto inteligente hay que crear un script que herede de *SmartObject* e implementar los siguientes métodos.

- *ValidateAgent*: decide si el agente que quiere interactuar con el objeto puede hacerlo. Puede usarse para comprobar si el agente tiene algún componente en concreto o cumple alguna condición.

```
public override bool ValidateAgent (SmartAgent agent)
{
    return true;
}
```

- *GetCapabilityValue:* Este método sirve para comprobar las capacidades del objeto para cubrir una necesidad concreta, recibiendo por parámetro el nombre de la necesidad y devolviendo la capacidad asociada en forma de valor numérico.

```
public override float GetCapabilityValue (string needName)
{
    return 0f;
}
```

- *RequestInteraction:* recibe como parámetros el agente y los datos de la petición y devuelve un objeto de la clase *SmartInteraction*. Para crear la interacción hay que crear antes la acción que se va a ejecutar y las capacidades que se aplican cuando se complete para pasarlas al constructor junto con el agente.

```
public override SmartInteraction RequestInteraction (SmartAgent agent, RequestData requestData)
{
    Action action = ...;
    Dictionary<string, float> capabilityMap = ...;
    SmartInteraction interaction = new SmartInteraction (action, agent, capabilityMap);
    return interaction;
}
```

## RequestActions

Para que un sistema de comportamiento comunique agentes con *SmartObjects* es necesario usar *RequestActions*. Para crear una acción de este tipo, se crea una clase que hereda de *UnityRequestAction* e implementa los siguientes métodos:

- *GetRequestedSmartObject*: Devuelve el objeto al que se realiza la petición.
- *GetRequestData:* Devuelve los datos que se incluirán con la petición. En la mayoría de los casos no será necesario enviar ningún dato, y en otros, ese dato será solo el nombre de la necesidad que se quiere cubrir. Dependerá de como use los datos de la petición el *SmartObject* concreto.

```
protected override SmartObject GetRequestedSmartObject ()
{
      // return ...
}

protected override RequestData GetRequestData ()
{
      // return ...
}
```

Todas las *RequestActions* necesitar una referencia a un *SmartAgent* para funcionar. En la herramienta se incluyen dos tipos de *RequestActions*:

- *TargetRequestAction*: Se crear usando el agente, una referencia a un *SmartObject* y una instancia de *RequestData*. Cuando se ejecute la acción, se interactuará con el objeto establecido.
- *RandomRequestAction*: Se crea usando el agente y una instancia de *RequestData*. Cuando se ejecute la acción, se interactuará con un objeto aleatorio.

Para buscar los *SmartObjects* de la escena, se proporciona la clase *SmartObjectManager* que guarda una lista con todos los objetos, aunque es posible crear nuevas formas de gestionarlos.

## Funcionamiento de una interacción con un *SmartObject*

1. El grafo de comportamiento entra en un nodo con una acción de tipo *RequestAction*.
2. Cuando se lanza el evento *Start* de la acción, esta busca un *SmartObject*. Si lo encuentra, le envía una referencia al agente y el resto de los datos de la petición (*RequestData*).
3. El *SmartObject* usa los datos recibidos para crear una instancia de *SmartInteraction* que contiene la acción que se va a ejecutar y se la envía a la *RequestAction*.
4. Si se ha recibido una interacción correctamente, se lanza el método *Start* de la acción que contiene.

5. A partir de este momento, los métodos lanzados en la *RequestAction* se propagan a la interacción. Hasta que termine o se pare. Si no se ha recibido una interacción se devolverá *Failure* directamente.

## Ejemplo de uso de *SmartObject*

A continuación, se muestra los pasos a seguir para construir un *SmartObject* simple que mueva al personaje hasta una posición concreta y después imprima un mensaje por consola. Cuando la interacción termine el objeto cubrirá la necesidad de *"ejercicio"* del agente.

El primer paso es crear el agente. Se crea un objeto en la escena, añadiendo el componente *SmartAgent* y se crea un archivo de configuración de agentes añadiendo la necesidad ejercicio con valor inicial 0.

Después se crea una clase *ExampleSmartObject* que hereda de *SmartObject* y se implementan sus métodos. En este caso la acción de la interacción es una secuencia de dos acciones: *movementAction* y *logAction*.

```
public class ExampleSmartObject: SmartObject
{
    public Vector3 pos;

    public override SmartInteraction RequestInteraction (SmartAgent agent, RequestData requestData)
    {
        BehaviourTree bt = new BehaviourTree ();

        Action movementAction = new WalkAction(_pos);
        LeafNode movementNode = bt.CreateLeafNode(movementAction);

        Action logAction = new DebugLogAction("El agente ha llegado al destino");
        LeafNode logNode = bt.CreateLeafNode(logAction);

        SequencerNode seq = bt.CreateComposite<SequencerNode>(false, movementNode, logNode);
        bt.SetRootNode(seq);

        Action action = new SubsystemAction(bt);

        Dictionary<string, float> capabilities = new Dictionary<string, float>();
        capabilities["ejercicio"] = 0.5f;
        return new SmartInteraction(action, agent, capabilities);
    }

    public override bool ValidateAgent(SmartAgent agent)
    {
        return true;
    }

    public override float GetCapabilityValue (string capabilityName)
    {
        if (capabilityName == "ejercicio") return 0.5f;
        else return 0f;
    }
}
```

El último paso será crear el sistema de comportamiento. Para simplificar el ejemplo, este sistema será un árbol de comportamiento con un único nodo hoja. La acción de este nodo será de tipo *TargetRequestAction*.

```
Public class ExampleSOBehaviourRunner: BehaviourRunner
{
    public SmartAgent agent; // Referencia al componente SmartAgent del objeto
    public ExampleSmartObject SmartObject; // Referencia al SmartObject creado anteriormente.

    protected override BehaviourGraph CreateGraph()
    {
          BehaviourTree bt = new BehaviourTree ();
          Action action = new TargetRequestAction (agent, SmartObject, new RequestData ());
          bt.CreateLeafNode(action);
          return bt;
    }
}
```