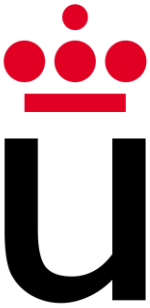


Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Grado en Ingeniería de Computadores

Curso académico 2022/2023

Trabajo de fin de grado

CONTRIBUCIONES Y REFACTORIZACIÓN DE UNA LIBRERÍA DE COMPORTAMIENTO DE AGENTES INTELIGENTES INCORPORANDO SMART OBJECTS

Autor

Alejandro Orejudo Fraile

Tutores

Carlos Garre Del olmo
Aarón Sújar Garrido

Resumen

Los sistemas inteligentes son un elemento básico dentro del diseño de videojuegos, ya que permiten crear personajes muy diversos que pueden adaptarse a cada jugador y así ofrecer una experiencia acorde a sus capacidades. Existen multitud de técnicas para crear estos sistemas inteligentes, cada una con sus propias características que la hacen más adecuada dependiendo de los requisitos y del resultado deseado.

Con esto en mente, este proyecto plantea el desarrollo de una librería para crear y ejecutar distintos sistemas de comportamiento para agentes inteligentes destinada a un entorno educativo, como árboles de comportamiento, máquinas de estados y sistemas de utilidad, diseñada para poder ser ampliada con nuevos tipos de sistemas en el futuro. Esta librería se creará mediante una refactorización completa de una versión previa, de forma que se resolverán posibles problemas de diseño e implementación para hacerla más sencilla de utilizar y más transparente al usuario.

Dentro de la librería, los sistemas de comportamiento se construyen en forma de grafos y funcionan en base a eventos que alteran el estado interno de cada elemento. Para cada uno de los tipos de sistemas incluidos, se han creado una serie de extensiones para ampliar su funcionalidad o facilitar su uso, como máquinas de estados de pila, estados probabilísticos, nuevos tipos de decoradores y nodos compuestos en árboles de comportamiento, o grupos de nodo y herramientas de optimización en sistemas de utilidad.

Finalmente se ha incluido un sistema de *smart objects* que permite a los distintos agentes interactuar con objetos delegando la gestión de su comportamiento a los mismos. Este sistema funcionará como una plantilla adaptable a distintos entornos.

Palabras clave: Sistema de comportamiento, agente inteligente, *NPC*, inteligencia artificial, grafo, nodo, árbol de comportamiento, máquina de estados, sistema de utilidad, *Smart Object*.

Índice de contenido

Índice de figuras

1

Introducción

El comportamiento de personajes y los sistemas de decisión son herramientas fundamentales en muchos campos como la robótica, creación de simulaciones o especialmente en el diseño de videojuegos, donde los personajes son un elemento imprescindible para crear una experiencia adaptada a cada jugador. Además, con la salida de herramientas como *ChatGPT* o *Stable Difusión*, la inteligencia artificial ha ganado mucha popularidad en los últimos tiempos, tanto en el ámbito profesional como para el público general. Se trata de un campo en constante evolución en el que la mayoría de las herramientas quedan obsoletas en poco tiempo y necesitan ser renovadas constantemente para mantenerse en el mercado y no ser sustituidas.

A pesar de esto, hay muchas técnicas para desarrollar sistemas inteligentes que llevan usándose desde los orígenes de la disciplina, y conviven hoy en día con técnicas más modernas como el *Deep-learning* [1]. Otras técnicas como las redes neuronales existen conceptualmente desde hace muchos años, pero han ido puliéndose y mejorándose hasta lograr las herramientas que existen hoy en día.

A pesar de toda esta variedad, no todas las técnicas tienen el mismo objetivo. Mientras que el propósito de las redes neuronales [2] o los modelos de aprendizaje automático suele ser resolver tareas en mucho menos tiempo de lo que lo hacen los humanos, otras técnicas se enfocan en crear agentes con un comportamiento que aparente una inteligencia similar al de una persona, no necesariamente superior.

Este trabajo se centra especialmente en las técnicas de este segundo grupo para crear agentes inteligentes que sirvan para mejorar la experiencia de los usuarios y que den a los desarrolladores un mayor control sobre el resultado. Por ello, se creará una librería en el lenguaje C# que permitirá utilizar estas técnicas para diseñar sistemas de comportamiento en agentes inteligentes. Es importante destacar que el proyecto se va a realizar en paralelo con el desarrollo de una herramienta que integre dicha librería en el motor de videojuegos *Unity* [3], añadiendo también una herramienta para modelar el comportamiento de forma visual.

1.1 Objetivos

El objetivo del proyecto es desarrollar una librería de código para crear sistemas de decisión para agentes inteligentes. La librería se usará principalmente en un entorno educativo, concretamente en la asignatura de comportamiento de personajes en el grado de diseño y desarrollo de videojuegos, con el propósito de ayudar a los estudiantes a entender algunas de las técnicas que existen para crear estos sistemas de decisión, como funciona cada una de ellas y sus ventajas e inconvenientes, mediante un enfoque práctico.

A diferencia de la muchas de las *APIs* comerciales, no se busca un funcionamiento de "caja negra", en el que el programador solo tiene que saber que elementos utilizar según el resultado que desee obtener. Uno de los objetivos es que el usuario pueda entender en todo momento como está funcionando internamente los distintos elementos, por ello una prioridad en el diseño es que el funcionamiento de los sistemas sea transparente al usuario, para que pueda conocer su estado interno en cualquier momento de la ejecución. Además, se ofrecerá el código fuente y una documentación detallada de cada elemento.

Otra característica importante es la modularidad. La herramienta se compondrá de distintos módulos independientes entre sí, facilitando a los usuarios la posibilidad de crear y añadir sus propios módulos, así como extender los existentes, pudiendo adaptar la herramienta a nuevas técnicas en el futuro o a las necesidades de un proyecto en concreto.

En entorno principal en el que se usará la herramienta será el motor de videojuegos *Unity* [3] ya que es usado como hilo conductor de la carrera de Diseño y Desarrollo de videojuegos, además de contar con una documentación y comunidad de usuarios muy amplia. Aun así, la librería debe poder ser usada en otros entornos como la robótica o la creación de simulaciones, por lo que el código debe mantenerse independiente de las *APIs* propias de *Unity*.

En lugar de crear la librería desde cero, se partirá de una versión previa a la que se aplicará una refactorización completa para resolver problemas de diseño y unificar el diseño y comportamiento base de todos los tipos de sistemas. Una vez realizada esta refactorización, se añadirán una serie de extensiones y cambios a cada tipo de sistema concreto para aumentar las capacidades de la herramienta, mejorar la usabilidad y facilitar su uso.

- Para máquinas de estados, se implementarán máquinas de estados de pila, estados probabilísticos y se modificarán las transiciones entre estados para simular un comportamiento de "FSM Mealy", de forma que las transiciones puedan ejecutar acciones.

- En árboles de comportamiento, el desarrollo se centrará en aumentar la colección de nodos disponibles añadiendo nodos que ejecuten comportamiento en paralelo o que evalúen condiciones de forma dinámica, además de simplificar la creación de nuevos tipos de nodos por parte del usuario.
- Por último, los cambios y añadidos en sistemas de utilidad también se centrarán en aumentar la cantidad de nodos disponibles, en este caso de factores de decisión, y en simplificar la creación de nuevos tipos. Además, se añadirá la posibilidad de agrupar las acciones en conjuntos según su prioridad y se proporcionarán herramientas para optimizar el rendimiento y así equipararlo con el resto de los sistemas.

Además de las extensiones dentro de los propios sistemas de comportamiento, se creará un modelo de objetos inteligentes compatible con todos los sistemas de la API. Estos objetos inteligentes permitirán crear elementos que indiquen a los agentes como deben ser usados y de esta forma abstraerlos de su funcionamiento.

1.2 Metodología y herramientas

1.2.1 Planteamiento del trabajo a desarrollar

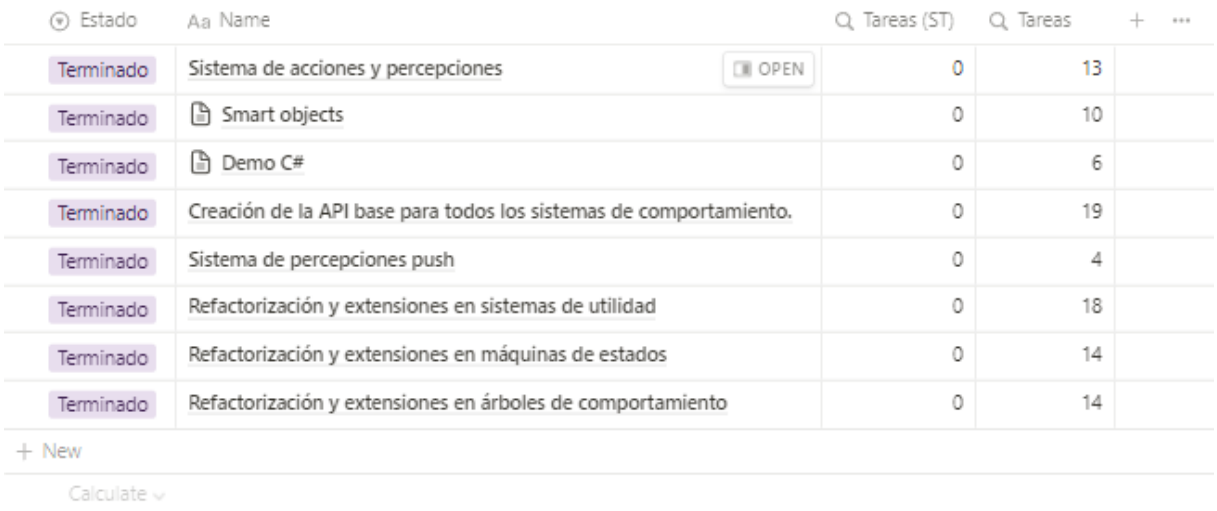
La primera parte del proyecto consistió en un análisis en profundidad de las necesidades que se van a cubrir con la librería: que sistemas de comportamiento se podrían crear y de que formas. Para ello se estudiaron las características de cada uno de los sistemas de la librería, destacando las capacidades específicas de cada uno y en qué casos es aconsejable usarlos, además de realizar un estudio de las distintas librerías existentes.

Tras esto, se analizó la *API* previa para establecer que aspectos era interesante conservar y cuales requerían cambios completos. Dado que la versión anterior había sido usada previamente, se valoró que problemas de uso tenía y que dificultades podía plantear a la hora de cumplir los objetivos del proyecto.

1.2.2 Gestión del proyecto

Una vez planteado el trabajo a realizar, se dividió el trabajo completo en subproyectos en base a las distintas funcionalidades de la *API*. A su vez, estos proyectos se dividen en tareas, intentando que el coste estimado en tiempo de cada tarea sea aproximadamente el mismo.

Para la gestión de estas tareas se ha utilizado la herramienta "*Notion*" [4]. *Notion* es una herramienta gratuita destinada a la organización y a la productividad, que combina creación de bases de datos, *wikis*, notas, gestión de proyectos y documentos. Es multiplataforma, funciona en la nube y gratuita, aunque cuenta con un plan de pago.



Estado	Name	Tareas (ST)	Tareas
Terminado	Sistema de acciones y percepciones OPEN	0	13
Terminado	Smart objects	0	10
Terminado	Demo C#	0	6
Terminado	Creación de la API base para todos los sistemas de comportamiento.	0	19
Terminado	Sistema de percepciones push	0	4
Terminado	Refactorización y extensiones en sistemas de utilidad	0	18
Terminado	Refactorización y extensiones en máquinas de estados	0	14
Terminado	Refactorización y extensiones en árboles de comportamiento	0	14

+ New

Calculate

Fig. 1.1 División en subproyectos del desarrollo de la API

El desarrollo del proyecto ha consistido en periodos o "*sprints*" de entre dos y tres semanas. Al comienzo de cada *sprint* se organizaba una reunión con los tutores para discutir el trabajo realizado en el anterior y decidir el trabajo del siguiente, que se dividía en tres partes:

- Cambios y correcciones a elementos ya desarrollados: El trabajo prioritario de cada sprint es arreglar posibles fallos o bugs de los elementos desarrollados en sprints anteriores.
- Documentación y guías: Una vez que una funcionalidad de la *API* queda cerrada, es importante crear la documentación y las guías correspondientes.
- Nuevas funcionalidades: Siguiendo la planificación inicial, decidir que funcionalidades implementar en el siguiente sprint en base a una estimación del tiempo que va a ser necesario.

Después de establecer el trabajo de un sprint, se actualizan los tableros de *Notion*, borrando y añadiendo las tareas necesarias, y se dividían las tareas del *sprint* a lo largo del tiempo.

1.2.3 Desarrollo del código y documentación

Para desarrollar el código de la librería se ha utilizado *Visual Studio* [5], un entorno de desarrollo que permite editar y compilar código en distintos lenguajes y plataformas como C++ o .NET. También permite crear y ejecutar tests para el código creado.

Por otro lado, se ha creado un repositorio de git para gestionar el control de versiones y los cambios en el desarrollo de la librería. El repositorio se ha alojado en *GitHub* [6], una plataforma web que permite subir repositorios y añade herramientas de colaboración e integración continua.

En cuanto a la documentación de la API se ha usado *DocFx*, una herramienta que permite generar documentación en formato web o *PDF* en base a los comentarios *XML* del código fuente. También permite añadir elementos en formato *markdown*, funcionalidad que se ha usado para crear las guías e información extra sobre la herramienta.

2

Estudio del estado del arte

Antes de comenzar con el desarrollo del proyecto se ha realizado una investigación acerca de las principales técnicas que se usan para crear sistemas de toma de decisiones y que librerías existen para cada una de ellas.

2.1 Técnicas principales

Las primeras técnicas utilizadas provienen del diseño de autómatas y se basan en evaluar condiciones simples. Aunque estas técnicas siguen usándose en la actualidad, hoy en día existen técnicas mucho más complejas enfocadas a entornos más específicos como el diseño de personajes de videojuegos.

A continuación, se analizarán algunas de las más populares, destacando las particularidades de cada una. Es importante destacar que existen muchas más técnicas de las nombradas a continuación, y además muchos estudios de videojuegos, especialmente los más grandes, utilizan sus propias técnicas combinando o adaptando estas.

2.1.1 Arquitecturas de subsunción

La arquitectura de subsunción [8] es una técnica de inteligencia artificial especialmente popular en el campo de la robótica. Se basa en descomponer el comportamiento en elementos más simples distribuidos en capas, de forma que los comportamientos superiores puedan utilizar los inferiores para definir su propio comportamiento.

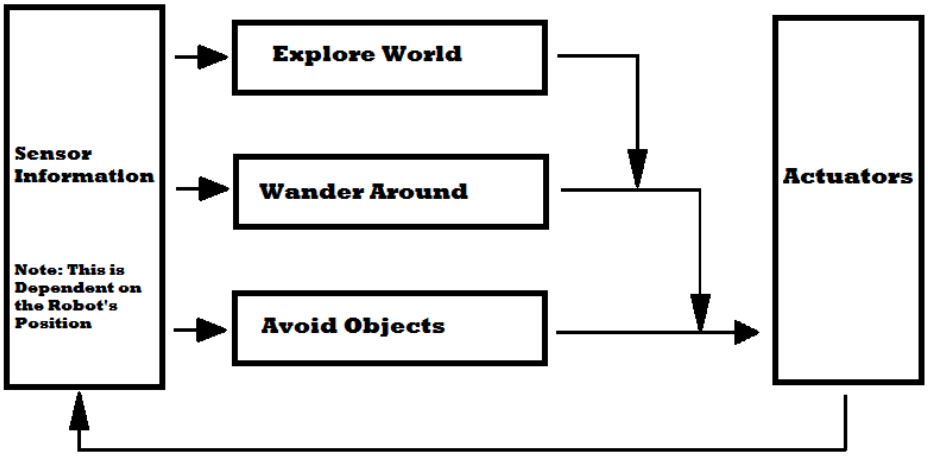


Fig. 2.1 Arquitectura de subsunción [9]

2.1.2 Máquinas de estados finitos

Las máquinas de estados [10] son una técnica utilizada en múltiples disciplinas de la programación para crear sistemas con memoria. Una máquina de estados finitos se compone de un número de estados limitados, y una función de transición, que, dado un estado y una entrada, devuelve una salida que suele consistir en una transición a otro estado.

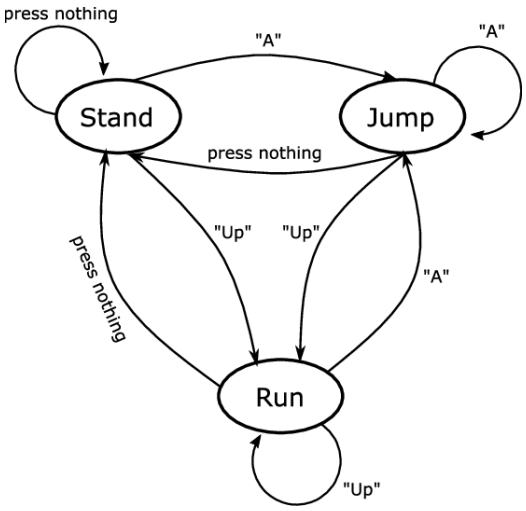


Fig. 2.2 Máquina de estados [11].

2.1.3 Árboles de comportamiento

Los árboles de comportamiento [12] son la técnica más popular y utilizada para crear sistemas de comportamiento en videojuegos. Surgieron como alternativa a las máquinas de estados que presentaban bastantes limitaciones especialmente si el número de estados era muy grande. La diferencia principal entre árboles de comportamiento y máquinas de estados es que se construyen como

un conjunto de nodos conectados en forma de árbol, en el que flujo de ejecución sigue siempre un recorrido trasversal, de forma que los nodos hijos afectan al resultado de los padres.

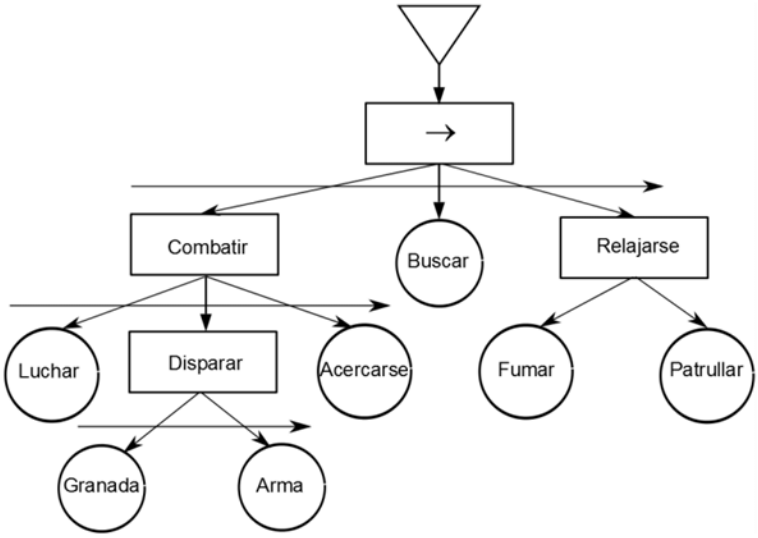


Fig. 2.3 Árbol de comportamiento [13]

2.1.4 Goal oriented action planning (GOAP)

La planificación de acciones orientada a objetivos o *GOAP* [14] es un modelo de inteligencia artificial que se basa en definir una serie de acciones posibles asignando a cada una el resultado de realizarlas, de forma que el agente establezca una secuencia de acciones que le permita cubrir sus objetivos de la manera más eficiente posible. Los objetivos se definen como un estado concreto del entorno que el agente quiere conseguir.

En generar este modelo se usa en contraposición a las máquinas de estados finitos para eliminar el acoplamiento y la dependencia entre estados.

2.1.5 Sistemas de utilidad

Los sistemas de utilidad son un tipo de sistema comportamiento que consiste en seleccionar una acción de un conjunto de acciones en base a un valor numérico o utilidad [15]. Este valor es calculado en base a valores del entorno y se modifica utilizando factores de decisión, que permiten aplicar funciones a estos valores o combinar varios en uno solo. Aunque esta técnica es menos utilizada que otras, como los árboles de comportamiento, gano popularidad con la publicación de *Los Sims*, videojuego en el que la inteligencia de los personajes se basa en este tipo de sistema.

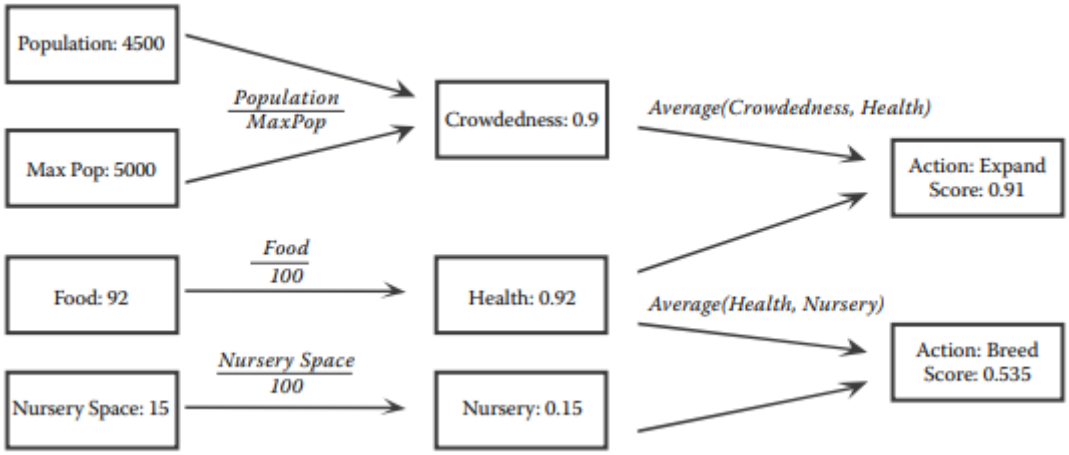


Fig. 2.4 Sistema de utilidad [15]

2.1.6 Smart objects

Los smart objects [16] son objetos que definen por sí mismos como deben ser usados por los distintos agentes. En lugar de ser una técnica para crear un sistema de toma de decisiones para un personaje en concreto, se trata de un mecanismo para abstraer a dicho personaje de la implementación del comportamiento en el objeto.

En general se suelen utilizar implementaciones de smart objects adaptadas a cada proyecto en concreto, ya que las características requeridas pueden variar bastante y no es una solución tan general como pueden ser las máquinas de estados o los árboles de comportamiento.

2.2 Librerías de sistemas de comportamiento

A continuación, se muestra una revisión de múltiples tecnologías de comportamiento de agentes e inteligencia artificial aplicados a diversos campos como la robótica o la creación de personajes en videojuegos.

2.2.1 Java Behaviour Trees

JavaBehaviourTrees o *JBT* [17] es una librería para la creación de árboles de comportamiento en java. Su arquitectura se basa en ejecutar un mismo árbol de comportamiento desde varios puntos usando "BT Executors", de forma que los árboles en sí funcionan como plantillas, manteniéndose independiente del estado de ejecución real. Los "Executors" usan el árbol de comportamiento original para generar una versión en tiempo de ejecución, a la que llaman "ExecutionBT".

La librería usa el concepto de “contexto de ejecución”, que es una estructura de datos o pizarra que permite compartir información entre los nodos del árbol. Desde el nodo raíz, cada nodo pasa el contexto de ejecución a sus nodos hijos. Los nodos pueden pasar su propio contexto directamente u otro contexto modificado.

La última actualización de la librería se publicó en 2013, por lo que algunos de sus elementos están desactualizados.

2.2.2 BehaviourTree.CPP

BehaviourTree.cpp [18] es una librería de C++ enfocada a crear árboles de comportamiento para robótica. Incluye un editor visual llamado “groot” [19] que permite crear los árboles de comportamiento con un sistema drag-and-drop y además incluye un sistema de depuración en tiempo real. También permite ejecutar los árboles directamente desde XML.

Actualmente es un proyecto open source.

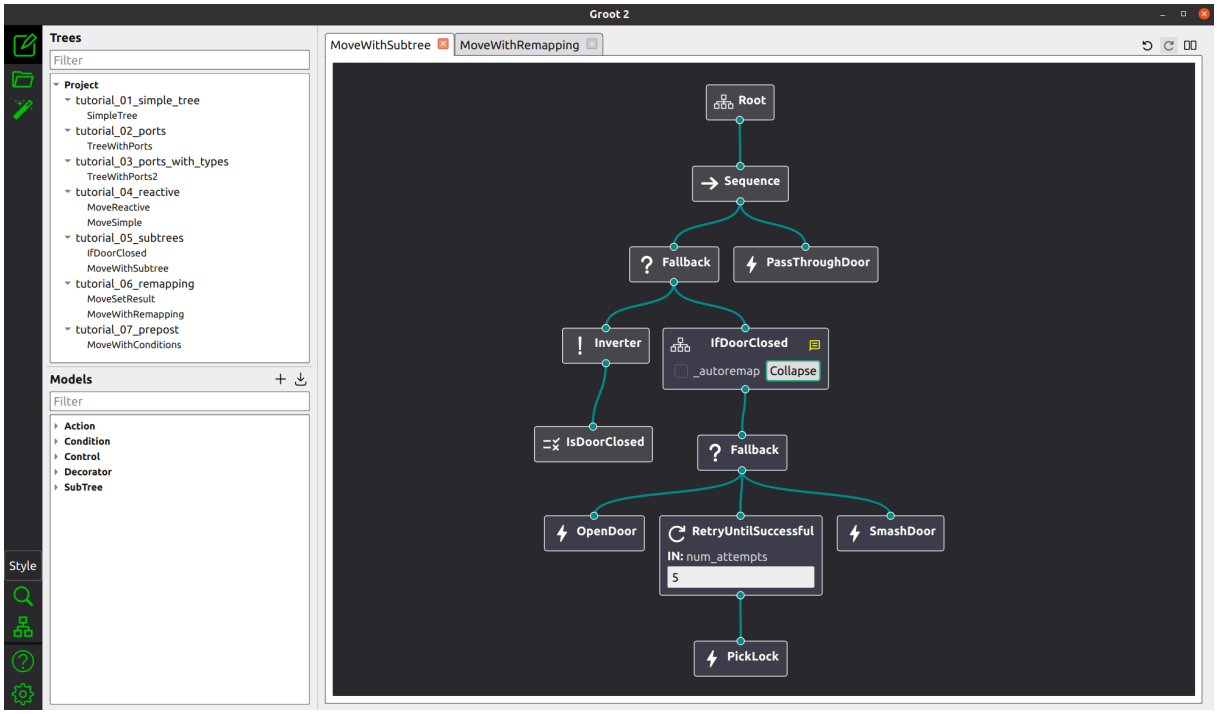


Fig. 2.5 Interfaz de "Groot", el editor visual de árboles de comportamiento de la librería BehaviourTrees.cpp

2.2.3 PyTrees

Al igual que el caso anterior, PyTrees [20] es una librería destinada a robótica, en este caso implementada en *Python* y cuenta con una gran cantidad de tipos de nodos.

Añade también una librería de *JavaScript* que permite visualizar la ejecución de los árboles de comportamiento en tiempo real. Además, incluye varias demos para probar las distintas funcionalidades.

2.2.4 Fluent Behaviour Trees

Fluent Behaviour Trees [21] es una API de árboles de comportamiento desarrollada en C#, específicamente para el motor de videojuegos *Unity*. A diferencia de otras, está enfocada íntegramente en el desarrollo de sistemas por código, en lugar de por datos. Esto significa que no usa herramientas para crear árboles de comportamiento como un editor visual o lenguajes como XML o json. En su lugar, se crean en tiempo de ejecución directamente con código C#.

Otro punto importante es que es una librería enfocada en ofrecer una "interfaz fluida". Una interfaz fluida [22] es un patrón de diseño de APIs que consiste en ofrecer al usuario una forma de crear y configurar objetos basada en métodos encadenados. También se conoce como patrón de encadenamiento de métodos y se usa en múltiples frameworks y herramientas, especialmente en consultas y manipulaciones de datos como JQuery en JavaScript o Linq en C#.

2.2.5 Aivo

Desarrollada por Markus Hjort en 2019, Aivo [23] es una librería de C# que trata de combinar las ventajas de otras APIS como Fluent Behaviour Trees o JBT. El propósito principal es simplificar la forma de crear árboles de comportamiento de forma que los componentes sean lo más reusables posibles.

La API ofrece un conjunto muy básico de nodos, como secuencia, selectores o decoradores inversores, pero proporciona una interfaz para crear nuevos tipos de nodos.

Al igual que la librería de árboles de comportamiento en Java (JBT), *Aivo* usa un contexto de ejecución que se propaga por los nodos y que pueden usar para leer y escribir en variables. En este caso los nodos usan un constructor genérico para especificar qué tipo de contexto se va a utilizar.

2.2.6 BehaviourTrees (Elixir)

Behaviour Trees [24] es una librería desarrollada en Elixir [25], un lenguaje basado en el paradigma de la programación funcional, dentro de la programación declarativa. La programación funcional se basa en equiparar las funciones a cualquier otro tipo de objetos, lo que permite usarlas como entrada

y salida de otras funciones, y de esta forma centrar la lógica en el resultado de una ejecución en lugar de en el proceso que ha seguido. También tiene otras propiedades como la inmutabilidad de los objetos y el determinismo de las funciones. Hoy en día, muchos lenguajes como Java, C# o Python soportan este tipo de programación.

Este enfoque permite crear los nodos de los árboles de comportamiento directamente como funciones, lo que permite extender la funcionalidad de una forma muy sencilla.

2.2.7 Stateless

Stateless [26] es una librería para crear máquinas de estados en .NET.

Usa una interfaz fluida, de forma que el programador define cada estado junto al "flujo de ejecución" de ese estado, lo que implica acciones al salir y entrar al estado y transiciones a otros estados. Además, soporta máquinas de estados jerárquicas.

2.3 Versión previa de la API

Ya que este proyecto consiste en parte en refactorizar una librería, es interesante desarrollar sus características y capacidades antes de realizar ningún cambio.

Se trata de una librería desarrollada en C# para crear sistemas de comportamiento, máquinas de estados y sistemas de utilidad, con la particularidad de que pueden combinarse entre sí para crear sistemas jerárquicos. El desarrollo de la librería ha sido gradual, a partir de tres proyectos distintos. El primero de ellos desarrollo las bases de la herramienta e implementó las máquinas de estados y los árboles de comportamiento [27], el segundo incluyó los sistemas de utilidad [28] y el tercero creo una interfaz gráfica para poder editar los sistemas de comportamiento de forma visual [29].

Aunque los usuarios no pueden extender la funcionalidad de los sistemas de comportamiento, si pueden definir percepciones y establecer métodos que se lanzan en momentos concretos de la ejecución, como cuando se entra en un estado de una máquina de estados. Las percepciones por otro lado son comprobaciones que hacen los sistemas para lanzar transiciones entre estados.

Uno de los problemas de diseño parte de que todos los sistemas funcionan internamente como máquinas de estados, lo que obliga a que todos los cambios en el flujo de ejecución pasen por crear y lanzar una transición. Aunque esto no supone un problema de rendimiento, puede dificultar depurar y

comprobar el estado de un sistema durante su ejecución. Además, no se proporciona ninguna herramienta de depuración, más allá de imprimir por consola cuando un sistema de utilidad cambia su acción seleccionada.

2.4 Conclusiones

Tras el análisis de las distintas librerías disponibles para crear sistemas de comportamiento, uno de los aspectos más importantes a tener en cuenta es que la gran mayoría de ellas están completamente enfocadas a árboles de comportamiento y apenas existen soluciones que permitan crear sistemas jerárquicos combinando varios tipos. Por ello, es interesante desarrollar una *API* centrada en la combinación de distintas técnicas y con un diseño modular y abierto, que permita incluir nuevas técnicas en el futuro, intentando aprovechar los puntos fuertes de estas librerías.

En cuanto al análisis de la versión previa de la librería, se ha concluido que la refactorización de la API se debe centrar en el funcionamiento interno para resolver los problemas de diseño causados por implementar internamente los sistemas como máquinas de estados, y también en hacerla más extensible y adaptable, pero intentando cambiar la interfaz con el usuario lo menos posible.

3

Diseño y desarrollo

En esta sección se describe el proceso de diseño de la herramienta desde las decisiones iniciales a la implementación final. El primer paso en el diseño es decidir que sistemas de comportamiento se van a usar, para estudiar en profundidad cada uno de ellos y así poder crear un modelo común. Después se desarrollará los elementos y cambios aplicados a cada uno de los tipos, así como las funcionalidades añadidas. Por último, se hablará de las funcionalidades que quedan fuera de los propios sistemas de comportamiento.

3.1 Selección de sistemas de comportamiento

En lugar de incluir nuevos tipos de sistemas de comportamiento, la librería incluirá los mismos tipos de la versión previa, centrando el proyecto en realizar una refactorización completa y ampliar cada tipo con nuevas funcionalidades y elementos. Antes de comenzar con el desarrollo es necesario hacer un análisis de cada uno de ellos para entender sus similitudes y diferencias.

3.1.1 Árboles de comportamiento

Los árboles de comportamiento son un tipo de sistema de decisión construido como un conjunto de nodos conectados en forma de árbol, de forma que cada nodo devuelve un resultado a su nodo padre, hasta llegar al nodo raíz. Comúnmente este resultado puede tomar tres valores distintos: *Running* (En ejecución), *Success* (Éxito) y *Failure* (Fallo).

Los nodos de un árbol de comportamiento se clasifican en hojas, decoradores y compuestos. Mientras que los nodos hoja, es decir, aquellos que no tienen hijos, obtienen su resultado de ejecutar una tarea o comprobación, el resto de los nodos lo obtienen de sus nodos hijos.

Los nodos decoradores solo tienen un nodo hijo y su resultado se calcula modificando el resultado de dicho hijo dependiendo del tipo concreto de decorador (bucle, inversor, condicional, etc). Los nodos

compuestos pueden tener varios hijos y los ejecutan de uno en uno. Los tipos más comunes de compuestos son el nodo secuencia y el nodo selector, y se diferencian en que el primero ejecuta sus hijos hasta que alguno devuelva *Failure* y el segundo lo hace hasta que devuelve *Success*.

3.1.2 Máquina de estados finitos

Una máquina de estados finitos en un sistema formado por un número limitado de estados conectados entre sí. En cada momento, la máquina se encuentra en un estado concreto, y puede recibir unos datos de entrada que provoquen una transición de ese estado a otro. Dependiendo de la implementación, la máquina de estados puede ejecutar acciones cuando entra o sale de un estado, cuando se lanza una transición concreta, etc.

En una máquina de estados, existe un estado inicial, en el que comienza la ejecución. Además, puede tener varios estados finales, que se caracterizan por que, una vez alcanzados, no es posible transitar a otro estado.

3.1.3 Sistemas de utilidad

Los sistemas de utilidad permiten seleccionar una acción de un conjunto en base a un valor numérico llamado utilidad, que representa su prioridad. Este valor viene ligado a las necesidades del agente que lo ejecuta o a parámetros del entorno, de forma que seleccione la acción que más le convenga en cada momento. Para calcular la utilidad de cada acción, se usa un tipo de nodos llamado “Factor de decisión”, estos factores pueden ser de tres tipos: factor hoja, curva o fusión.

Los factores hoja obtienen su utilidad de una variable del entorno. El rango de valores puede ser distinto según la variable concreta, por ejemplo, la salud de un personaje en un *RPG* puede medirse de 0 a 100 mientras que el hambre se mide de 0 a 10. Esto hace que los factores tengan que normalizar el valor de utilidad que obtienen en un rango común (0-1).

Las curvas de utilidad sirven para modificar el valor de otro factor aplicándole una función. Se usa en aquellos casos en los que es necesario que la utilidad de un factor no crezca de forma directamente proporcional con una variable concreta. Por ejemplo, si una acción consiste en curar a un personaje, la utilidad de dicha acción debería ser mayor cuanto menor sea la salud del personaje. Existen varios tipos según la función aplicada (lineal, exponencial, umbral, etc).

Los factores fusión permiten combinar la utilidad de varios factores en uno solo. Es útil en los casos en los que querer realizar una acción dependa de más de un parámetro. Por ejemplo, curar a un

personaje puede depender tanto de su salud como de si dispone de algún objeto para curarse. Pueden ser de varios tipos (mínimo, máximo, media ponderada, etc.)

3.2 Modelo común de sistemas de comportamiento

Una vez decidido que sistemas de comportamiento se van a incluir, es necesario crear un modelo común entre todos ellos, que permita combinarlos entre sí y ejecutarlos sin importar el tipo concreto. Este modelo se dividirá en dos partes:

- Modelo de creación: Definir la estructura de los sistemas de comportamiento y que herramientas tendrá el usuario de la API para crearlos.
- Modelo de ejecución: Definir como se modifica el estado interno de los sistemas de comportamiento y sus propiedades en tiempo de ejecución, así como los mecanismos que tiene para interactuar con el entorno.

3.2.1 Modelo de creación

Se ha decidido que los sistemas de comportamiento se diseñen en base a un modelo de grafos dirigidos. Por un lado, las máquinas de estados se pueden entender como un grafo dirigido que permite ciclos, en el que cada estado es un nodo y cada arista es una transición. Los árboles de comportamiento, por otro lado, se basan en árboles, un subtipo de grafo dirigido con unas restricciones concretas: cada nodo solo puede tener un nodo padre (excepto el nodo raíz) y no se permiten bucles.

El caso de los sistemas de utilidad es algo más complejo. La jerarquía de factores de una acción concreta forma un grafo dirigido acíclico, una estructura similar a la de los árboles que elimina la restricción de que un nodo solo puede tener un padre (un mismo factor puede usarse desde varias acciones). Por otro lado, las acciones no necesitan estar conectadas, ya que es el propio sistema de utilidad el que se encarga de cambiar de acción.

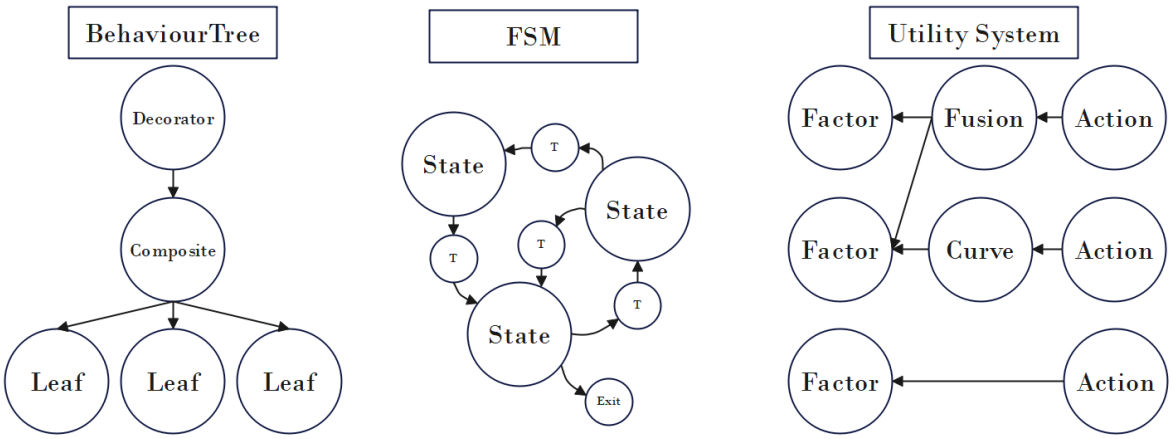


Fig. 3.1 Sistemas de comportamiento modelados como grafos dirigidos.

Para mantener el modelo de diseño independiente del modelo de ejecución, se creará un subtipo de sistema de comportamiento llamado grafo de comportamiento o *BehaviourGraph*. Esto permite añadir nuevos sistemas sin obligar a diseñarlos en forma de grafo, y aun así poder combinarlos con el resto de los tipos. Cada instancia de grafo de comportamiento manejará internamente una lista de nodos y, al ser dirigido, cada nodo tendrá a su vez una lista de nodos padres y una lista de nodos hijos para manejar sus conexiones.

Este modelo presenta un problema con las máquinas de estados, ya que si cada estado almacena directamente a que nodos puede transitar no es posible almacenar más información en la transición, como las condiciones para que se active o posibles acciones que deba realizar el agente al lanzarla. Tampoco es posible crear transiciones de salida, que no tienen ningún estado destino.

Una primera solución planteada fue modelar las conexiones entre nodos como una clase propia. Aunque esto resuelve el problema, añade una carga adicional de recursos al tener que instanciar un objeto extra por cada par de nodos conectados, que en la mayoría de los casos no aporta ninguna información, además de hacer más compleja la construcción de los grafos de comportamiento. La alternativa escogida ha sido implementar las transiciones entre estados como un tipo de nodo independiente (ver fig. 3.1). Este tipo está limitado a un único nodo padre y nodo hijo, que serán el estado inicial y final de la transición. De esta forma podemos guardar toda la información de las transiciones en esta clase sin cambiar el modelo del resto de sistemas.

El esquema de diseño planteado permite crear los grafos de comportamiento únicamente con dos métodos.

- *CreateNode*: Crea una instancia de un tipo de nodo concreto y lo añade al grafo.

- *Connect*: Recibe como parámetro dos nodos y crea una conexión entre ellos.

El usuario de la API no tiene que usar estos métodos directamente, sino que cada tipo de grafo de comportamiento los usa internamente para implementar sus propias funciones para crear y conectar sus nodos (ver fig. 3.2).

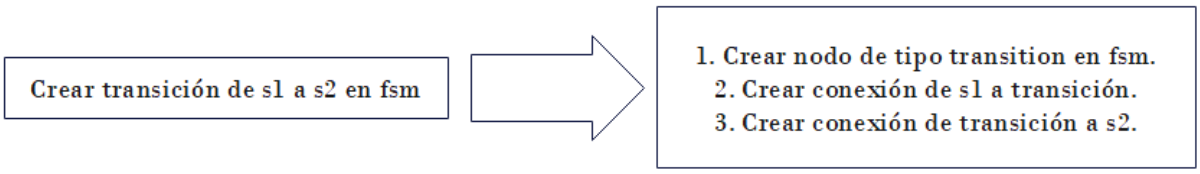


Fig. 3.2 Ejemplo de cómo las máquinas de estados usan los métodos de la API para crear transiciones.

A este esquema se han añadido una serie de restricciones para evitar errores a la hora de añadir un nodo a un grafo o al conectar dos nodos:

- Cada tipo de grafo determina que tipos de nodos puede contener. Por ejemplo, un árbol de comportamiento no puede contener ni estados ni factores, únicamente nodos de árbol.
- Cada nodo determina a que tipo de grafo puede pertenecer. Aunque pueda parecer redundante con la restricción anterior, esto permite crear subtipos de un sistema de comportamiento con nodos propios que el tipo base no puede usar.
- Cada nodo limita la cantidad de nodos padres e hijos que puede tener. Un nodo compuesto de un árbol de comportamiento puede tener varios hijos, mientras que un decorador solamente tiene uno, y un nodo hoja ninguno.
- Cada nodo limita que tipo de nodos puede tener como hijos. Un estado de una máquina de estados solo puede tener como hijos transiciones, mientras que un nodo acción de un sistema de utilidad solo puede tener factores.
- Cada tipo de grafo determina si sus nodos pueden tener conexiones repetidas, es decir, un elemento repetido en sus nodos padres o hijos. Ninguno de los sistemas incluidos permite conexiones repetidas.
- Cada tipo de grafo determina si sus nodos pueden formar ciclos. De los sistemas incluidos, solo las máquinas de estados lo permiten.
- Dos nodos solo pueden conectarse si pertenecen al mismo grafo. Para poder comprobarlo, cada nodo tendrá una referencia al grafo al que pertenece.

Una vez creados todos los elementos del grafo de comportamiento, puede ser necesario acceder a un nodo en concreto. Aunque existe la posibilidad de guardar una referencia a la variable cuando es instanciada, se ha añadido un mecanismo para poder asignar un nombre al nodo cuando se crea y luego

buscarlo dentro del grafo usando dicho nombre. Cuando se asigne un identificador a un nodo, se comprobará internamente si el nombre es válido o si ya existe un nodo con el mismo nombre en el grafo.

3.2.2 Modelo de ejecución

Uno de los problemas principales surge a la hora de querer crear sistemas jerárquicos que combinen distintos tipos de sistemas de comportamiento. Si un árbol de comportamiento ejecuta una máquina de estados finitos desde uno de sus nodos, cuando la ejecución de la máquina de estados termine, ¿El nodo del árbol debería devolver éxito o fallo? En el caso inverso, siendo el árbol el que se ejecuta desde un estado de la máquina de estados, ¿esta última debería tener en cuenta con que valor ha terminado el árbol?

Este problema se ha resuelto añadiendo el concepto de estado de ejecución o *Status* (para distinguirlo de los estados de una máquina de estados) como una propiedad de cada sistema de comportamiento, que puede tomar los siguientes valores:

- *None*: El sistema no se está activo, porque no se ha comenzado la ejecución o porque se ha interrumpido de algún modo.
- *Running*: El sistema se está ejecutando actualmente pero no ha terminado.
- *Success*: La ejecución del sistema ha terminado con éxito.
- *Failure*: La ejecución del sistema ha terminado con fallo.
- *Paused*: La ejecución esta pausada.

Además de los propios sistemas de comportamiento, algunos nodos también tienen su propio estado de ejecución. Aunque una primera opción fue que todos los nodos lo tuvieran, algunos tipos como las transiciones o los factores no lo necesitan, ya que su función es ofrecer información a otros nodos y no ser ejecutados directamente.

Para cambiar el estado de ejecución, los sistemas de comportamiento incluyen varios eventos que el programador puede usar. Estos eventos se implementan de forma distinta según el tipo de sistema, pero una parte del funcionamiento es común en todos ellos.

- *Start*: Se lanza al comienzo de la ejecución y pone el valor *Status* en Running.
- *Update*: Se lanza en cada iteración para actualizar la ejecución del sistema. En algunos casos, como en los árboles de comportamiento, puede poner el valor de *Status* a *Success* o *Failure*.

- *Stop*: Detiene la ejecución, sin importar si ha terminado o si está pausado. Pone el valor Status a *None*.
- *Pause*: Indica que la ejecución se ha pausado. Si el valor Status era *Running* lo cambia a *Paused*.
- *Unpause*: Se lanza para indicar que la ejecución se ha reanudado. Si el valor Status era Running
- *Finish:* Termina la ejecución del sistema con *Success* o *Failure* dependiendo del parámetro especificado. Aunque el programador puede usarlo directamente, está pensado para que lo haga el propio sistema de comportamiento, como en el caso de las transiciones de salida en máquinas de estados.

Este sistema de eventos permite extraer un comportamiento común de todos los sistemas. Este comportamiento se puede modelar como una máquina de estados finitos (ver fig. 3.3), en el que cada valor *Status* en el que puede estar constituye un estado, y cada evento provoca una transición entre estados.

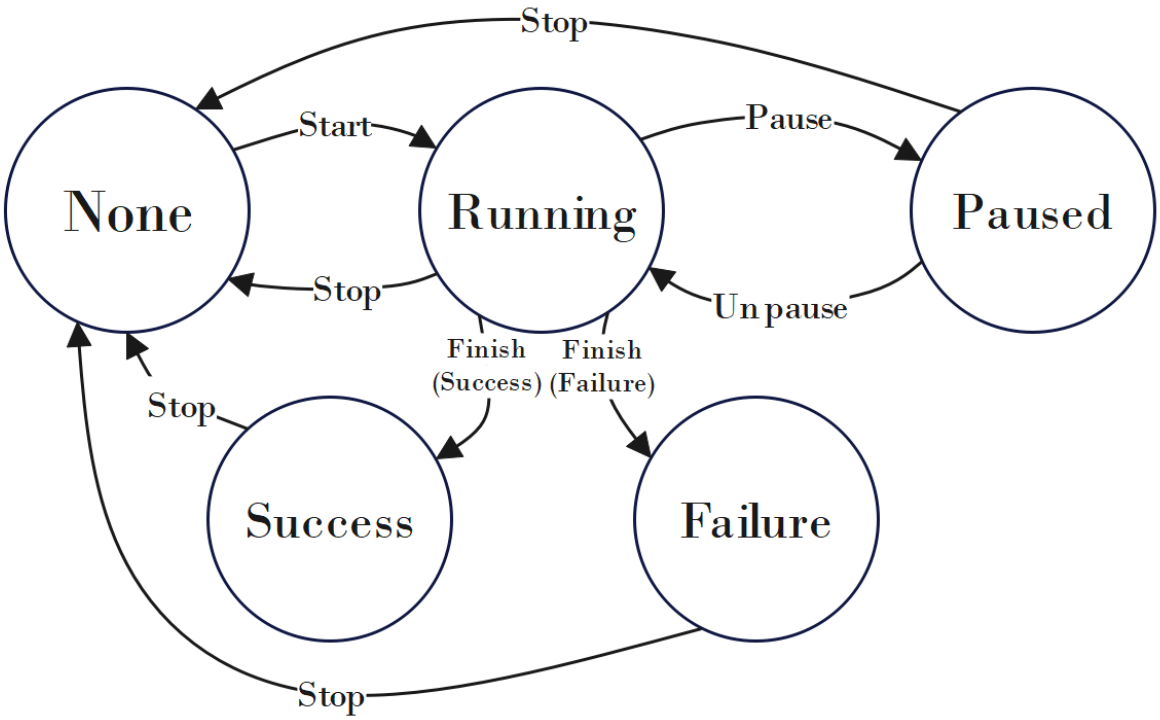


Fig. 3.3 Diagrama de estados de ejecución de un sistema de comportamiento

Este modelo permite al programador conocer el estado de los sistemas de comportamiento en cualquier momento de la ejecución, además de tener un control de que cambios en el estado de ejecución se deben permitir y cuáles no.

3.2.3 Acciones y percepciones

Las acciones y percepciones son mecanismos que usan los sistemas de comportamiento para interactuar con el entorno. Por un lado, las acciones son subrutinas o tareas que el agente ejecuta para modificar el entorno o a sí mismo, por ejemplo, para moverse de un punto a otro, coger un objeto, emitir un sonido, etc. Por otro, las percepciones se usan para que el agente realice alguna comprobación, como la distancia a otro objeto, el valor de alguna propiedad como el hambre o la salud, y evalué una condición. En los grafos de comportamiento, algunos nodos pueden tener asignadas acciones y/o percepciones.

Tanto las acciones como las percepciones implementan una serie de eventos, que son lanzados por el nodo que las contiene, habitualmente cuando el estado de ejecución de dicho nodo cambia, ya sea porque se ha iniciado, pausado, etc. Estos eventos son los siguientes:

- *Start* (*Init* en percepciones): Inicializa la acción o percepción.
- *Stop* (*Reset* en percepciones): Resetea la acción o percepción.
- *Pause* / *Unpause*: Pausa / despausa y la acción o percepción.

Además de estos eventos, las acciones y percepciones se actualizan cada frame mientras están activas. En el caso de las acciones, el evento de actualización (*Update*) devuelve un valor *Status* que los nodos usan para actualizar su propio estado de ejecución. En percepciones, este evento (*Check*) devuelve un valor booleano para indicar si la condición de la percepción se ha cumplido.

Se han incluido varios tipos de acciones y percepciones en la *API*. Además, los usuarios pueden crear sus propios tipos.

- *FunctionalAction*: permite definir un delegado para cada evento de la acción.
- *SimpleAction*: simplificación del tipo anterior que solo define un delegado para el evento *Start*. Se usa en situaciones en los que no se requiere que la acción se ejecute en varios frames, ya que devuelve *Success* directamente.
- *ConditionPerception*: define un delegado para cada evento de la percepción.
- *ExecutionStatusPerception*: Se usa para comprobar el estado de ejecución de otro elemento, ya sea dentro o fuera del grafo de comportamiento. Un uso común de esta percepción es comprobar si una máquina de estados está en un estado concreto.
- *TimerPerception*: Devuelve false hasta que pase una cantidad de tiempo especificada.
- *CompoundPerception*: Sirve para combinar varias percepciones en una, en este caso aplicando una operación lógica a su resultado (AND/OR).

3.2.4 Jerarquías de sistemas de comportamiento

Para crear jerarquías de sistemas de comportamiento que combinen cualquier tipo, se ha creado un tipo especial de Acción llamado *SubsystemAction*. Esta acción contiene un sistema de comportamiento, y llama a sus métodos de ejecución cuando se lanza sus eventos correspondientes.

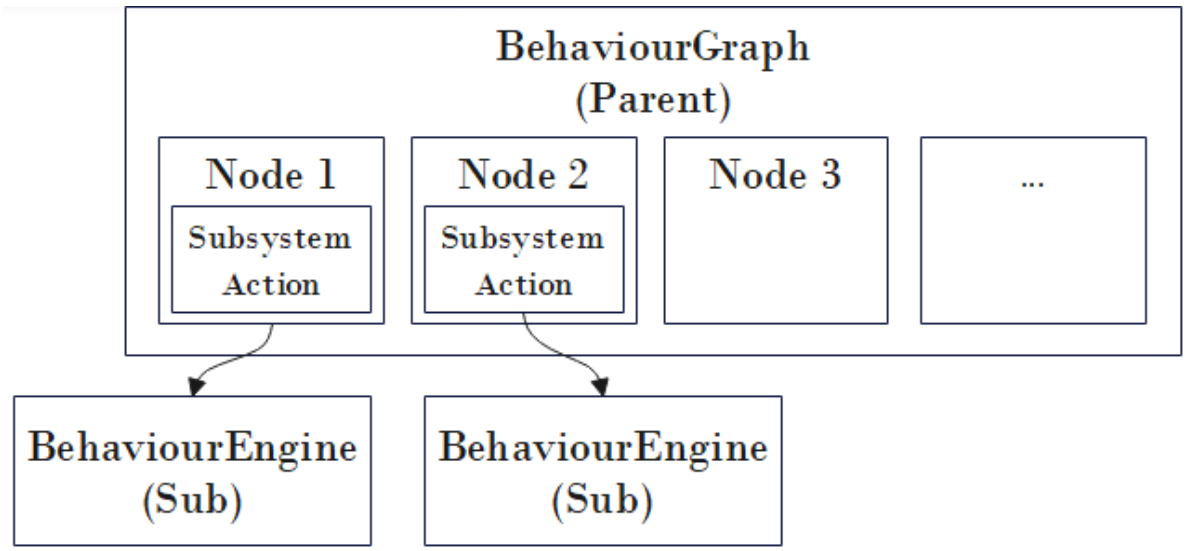


Fig. 3.4 Ejemplo de jerarquía de sistemas de comportamiento con dos subsistemas.

Esta estructura permite crear jerarquías sin límite de profundidad (ver fig. 5), pero el programador es responsable de evitar que se formen bucles al hacer que un sistema de comportamiento sea un subsistema de sí mismo.

Las *SubsystemActions* permiten además especificar otros parámetros para controlar la ejecución:

- *ExecuteOnLoop*: Si se activa, cuando la ejecución del subsistema termine, se reiniciará, de forma que solo se pueda terminar de forma externa, desde el sistema padre.
- *InterruptOptions*: Sirve para definir que evento se debe lanzar en el subsistema cuando es interrumpido desde fuera y cuando es reanudado. Por defecto, se lanzarán los métodos *Stop* y *Start* respectivamente, de forma que la ejecución se detenga completamente, pero puede cambiarse para lanzar los eventos *Pause* y *Unpause* o no lanzar ninguno.

3.2.5 Percepciones push

Si en una máquina de estados se requiere que se lance una transición de un estado a otro cuando se pulsa una tecla, habría que crear una percepción que compruebe en cada ciclo si la tecla está siendo pulsada, lo que no es muy eficiente. Es necesario crear un mecanismo para lanzar directamente la transición desde fuera cuando la tecla sea pulsada, eliminado la comprobación anterior.

Las percepciones push permiten activar ciertos nodos desde fuera del grafo de comportamiento. Para que un nodo pueda ser activable por una percepción push debe implementar el método *Fire* de la interfaz *IPushActivable*. Este método recibe como parámetro un valor *Status* y su comportamiento depende de cómo se implemente en el nodo.

- En transiciones, si el estado origen es el estado actual de la máquina de estados, se lanzará sin realizar ninguna comprobación. El valor pasado como parámetro se ignora.
- En nodos de un árbol de comportamiento, cambia el Status del nodo al valor especificado como parámetro (*Success* o *Failure*). De esta forma es posible forzar que la ejecución de un nodo termine de forma externa.

El uso de la interfaz *IPushActivable* permite a los usuarios de la *API* añadir usos a las percepciones push de una forma sencilla.

3.2.6 Contexto de ejecución

El contexto de ejecución es una herramienta incluida en la API para compartir datos entre los distintos elementos del sistema. Cuando se usa el método *SetContext* en un sistema de comportamiento, este se propaga por todos los elementos (nodos, acciones, percepciones, subsistemas, etc).

Para usar el contexto de ejecución, el usuario de la *API* debe crear una clase que herede de *ExecutionContext* con las variables y métodos que necesite. Una vez creado, puede ser usado desde las acciones y percepciones que cree el usuario, sobrescribiendo el método *SetExecutionContext*.

3.2.7 Diagramas de clases

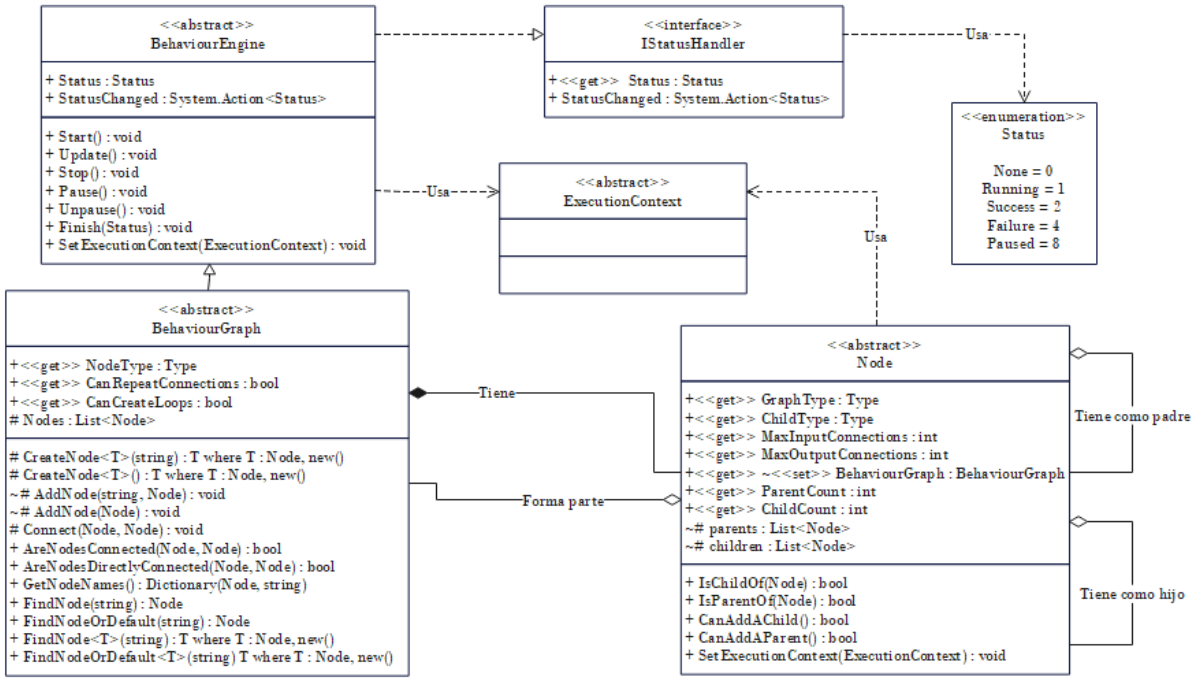


Fig. 3.5 Diagrama de clases de grafos de comportamiento

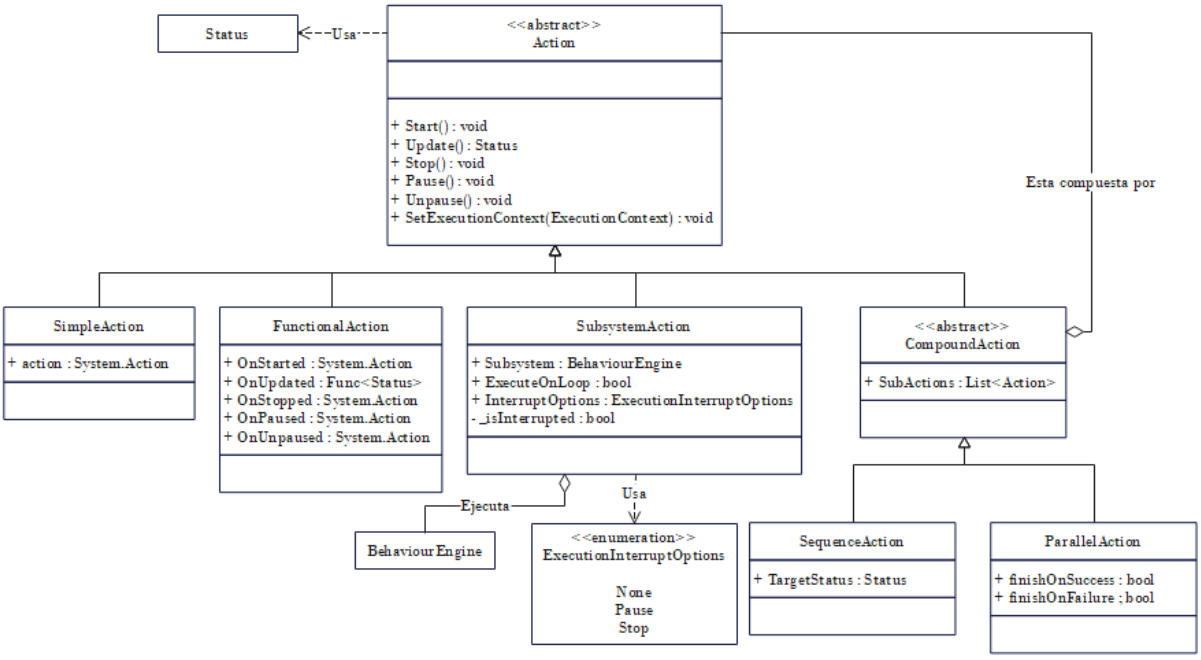


Fig. 3.6 Diagrama de clases de acciones

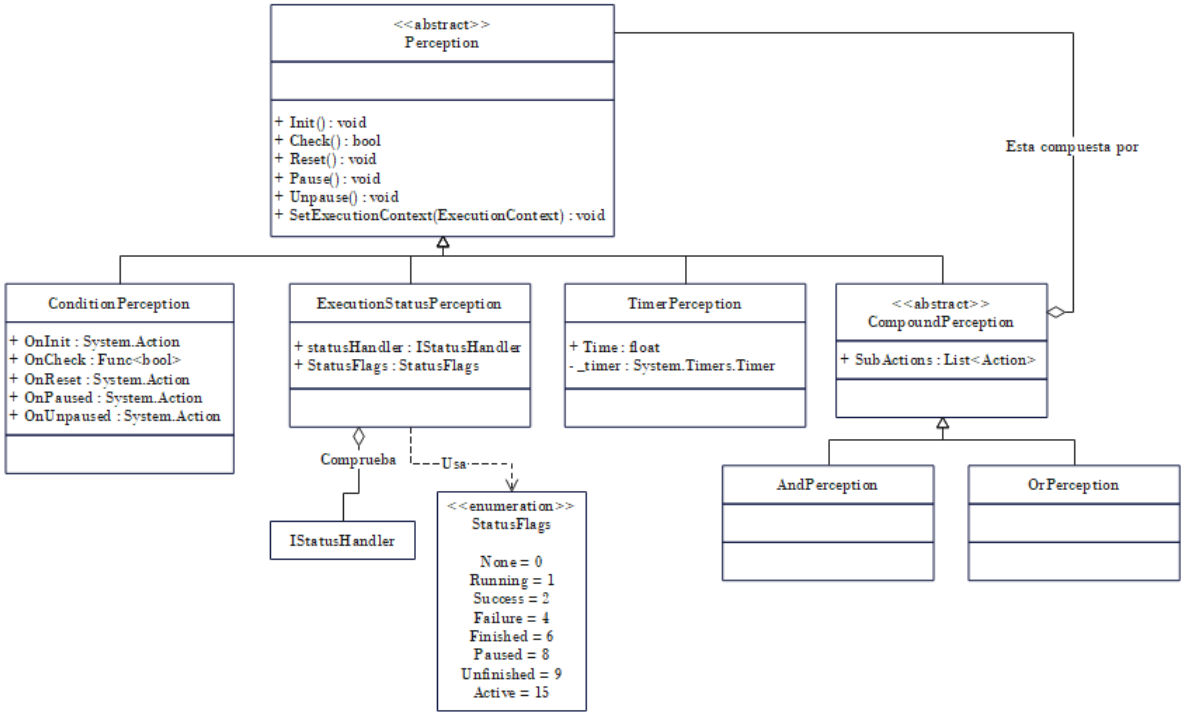


Fig. 3.7 Diagrama de clases de percepciones

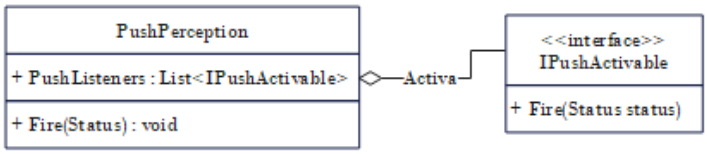


Fig. 3.8 Diagrama de clases de percepciones push

3.3 Árboles de comportamiento

Los árboles de comportamiento son uno de los tres sistemas de comportamiento que se incluyen en la *API*. La ejecución de este tipo de sistema se basa en aprovechar la estructura en forma de árbol para propagar los eventos de ejecución (*Start*, *Update*, etc.) desde el nodo raíz hasta un nodo hoja. Teniendo en cuenta que cada tipo de nodo propaga estos eventos de una forma distinta, es necesario definir una jerarquía de nodos en base a ello.

El tipo de nodo base en los árboles de comportamiento se define en la clase *BTNode*. Esta clase especifica dos restricciones: la primera es que cada nodo solo puede tener un nodo padre, lo que evita la creación de bucles, y la segunda es que solo puede conectarse a otros nodos de la clase *BTNode*. Esta clase se divide en tres tipos, nodos hoja, decoradores y compuestos, que se distinguen por la cantidad de nodos hijo que tienen (ninguno, uno y varios respectivamente). Mientras que los nodos

hoja solo tienen una implementación posible, que consiste en ejecutar una acción y actualizar su valor *Status* con el resultado de dicha acción, los nodos decoradores y compuestos tienen múltiples implementaciones posibles.

Dentro de los nodos decoradores, se ha creado un subtipo llamado *DirectDecorator* para simplificar la implementación de todos aquellos que no definen ninguna condición para que su nodo hijo se ejecute. Esto incluye a todos los decoradores excepto al nodo condicional y el nodo "*Timer*". La clase *DirectDecorador* utiliza el método *ModifyStatus*, que recibe el valor Status del nodo hijo y lo devuelve modificado. En la siguiente tabla se puede ver como se ha implementado este método para crear cada decorador (ver fig. 3.9). Esta clase permite al usuario crear nuevos tipos de decoradores de forma sencilla.

Decorador	**Implementación del método *ModifyStatus***
Succeder	Si recibe *Failure*, devuelve *Success*.
Inverter	Si recibe *Success*, devuelve *Failure* y viceversa.
Loop	Si recibe success o *Failure*, incrementa el contador de iteraciones. Si se ha llegado al numero de iteracciones específico, se devuelve el valor recibido. Si no se ha llegado, se devuelve *Running* y se reinicia el nodo hijo.
LoopUntil	Identico al anterior, pero si el valor recibido coincide con la propiedad *TargetStatus*, devuelve el valor directamente.

Fig. 3.9 Implementación del método ModifyStatus en cada tipo de decorador.

Para los nodos compuestos se ha creado otra clase llamada *SerialCompositeNode* que engloba aquellos que recorren sus nodos hijos uno a uno, es decir, el nodo secuencia y el nodo selector. Esta clase utiliza el método *KeepExecutingNextChild* para controlar el flujo de ejecución. Este método recibe el valor *Status* del nodo hijo actual y devuelve *true* si la ejecución debe saltar al siguiente hijo. En el nodo secuencia devolverá *true* si el valor recibido es *Success* y en el nodo selector si es *Failure*.

A pesar de estos cambios, es posible crear nuevos tipos de nodos heredando directamente de las clases base (*DecoratorNode* y *CompositeNode*). El objetivo de incluir estos subtipos es facilitar la extensión de la *API* y la clasificación de los distintos nodos en base a su función.

Una vez definida la estructura de clases base de los árboles de comportamiento, se han desarrollado una serie de extensiones que añaden distintas funcionalidades, en forma de distintos tipos de nodos, que se detallan a continuación.

3.3.1 Nodo condicional reactivo

Los nodos condicionales básicos son decoradores que evalúan una condición cuando comienza su ejecución. Si la condición se cumple, ejecutan su nodo hijo directamente, sin modificar su resultado devuelto. Aunque esto es útil en muchos casos, en otros puede ser necesario reevaluar la condición varias veces a lo largo de la ejecución.

El nodo condicional reactivo es una variante del nodo condicional basado en la programación reactiva [30], un paradigma de la programación basado en crear sistemas que reaccionen a cambios en los datos. Un ejemplo recurrente para explicar cómo funciona este paradigma es la suma de dos variables. En la programación tradicional, después de sumar dos variables *a* y *b* y guardar el resultado en c, aunque después se asigne a la variable *a* otro valor, el valor de c no cambia. En cambio, en la programación reactiva, cambiar el valor de *a* modificaría también el valor de *c* ya que *c* es reactiva a los cambios en *a*.

Siguiendo este funcionamiento, los nodos condicionales ejecutan su nodo hijo mientras la percepción devuelva *true*, pero la ejecución del nodo hijo no comenzará mientras la percepción devuelva *false*. La percepción se comprobará en cada frame hasta que la ejecución del nodo hijo termine, y si la percepción devuelve *false* una vez comenzada la ejecución del nodo hijo esta se interrumpirá hasta que la percepción vuelva a devolver *true* (ver fig. 3.10).

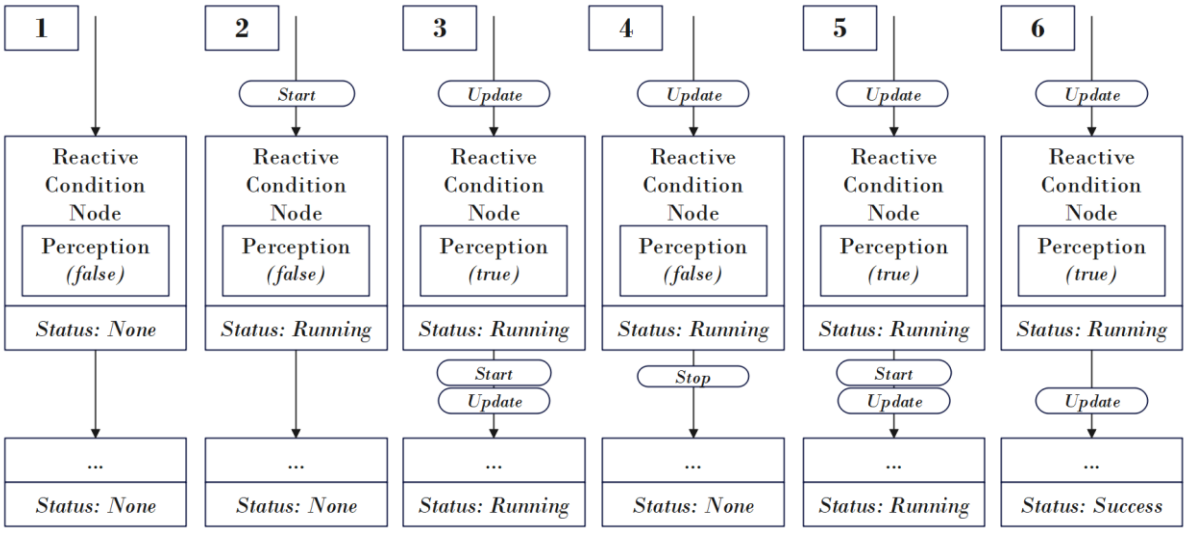


Fig. 3.10 Diagrama de cambios en un nodo condicional reactivo.

Al igual que ocurre con las acciones que contienen subsistemas de comportamiento, es posible elegir que eventos se lancen en el nodo hijo cuando la ejecución se interrumpa y reanude utilizando el tipo *ExecutionInterruptOptions*. Por defecto se lanzarán los eventos *Start* y *Stop*, parando completamente

la ejecución, pero puede cambiarse para que la ejecución solo se pause o para que no se lance ningún evento.

3.3.2 Nodos paralelos

En la implementación previa, los nodos compuestos siempre ejecutan cada nodo hijo hasta que esté devolvía un valor Status concreto. Los nodos paralelos, en cambio, ejecutan todos sus hijos en cada interacción mientras no hayan terminado. Es importante recalcar que los nodos no se ejecutan en un paralelismo real, ya que la versión actual de la *API* no utiliza ningún modelo de concurrencia o paralelismo. Lo que ocurre es que en cada evento *Update*, el nodo paralelo recorre sus hijos, comprueba que su estado actual sea igual a Running y lanza su evento *Update* si es el caso. Los eventos *Start* y *Stop* se propagan siempre a los nodos hijos mientras que los eventos *Update*, *Pause* y *Unpause* solo se propagan si la ejecución no ha terminado (ver fig. 3.11). De esta forma se garantiza que un nodo que ya ha terminado su ejecución no siga actualizándose y no pueda pausarse.

Evento en nodo paralelo	Condición para propagar el evento a un nodo hijo
OnStarted	Siempre
OnUpdated	Solo si Status == Running
OnStopped	Siempre
OnPaused	Solo si Status == Running
OnUnpaused	Solo si Status == Running

Fig. 3.11 propagación de eventos en nodos compuestos paralelos.

Es posible parametrizar cuando termina la ejecución de los nodos paralelos mediante dos valores booleanos, llamados *FinishOnFailure* y *FinishOnSuccess*, que especifican si detener la ejecución cuando un nodo hijo ha terminado con valor *Success* y *Failure* respectivamente. Estos dos valores permiten un total de cuatro configuraciones (ver fig. 3.12).

FinishOnSuccess	FinishOnFailure	Resultado de la ejecución
False	False	La ejecución termina cuando todos los hijos terminen, devolviendo el resultado del último hijo que haya terminado.
True	False	La ejecución termina con Success cuando un hijo termine con Success. En caso contrario termina con Failure.
False	True	La ejecución termina con Failure cuando un hijo termine con Failure. En caso contrario termina con Success.
True	True	La ejecución termina cuando termine uno de los hijos, con el resultado de ese hijo.

Fig. 3.12 Posibles resultados de un nodo compuesto paralelo según su configuración

3.3.3 Acciones secuenciales y paralelas

Si se quiere crear un sistema distinto a un árbol de comportamiento en el que se ejecuten una serie de acciones en una secuencia o en paralelo, con los elementos presentados hasta ahora sería necesario crear un árbol de comportamiento como un subsistema dentro del sistema actual.

Para simplificar esto, se han añadido las acciones compuestas, que permiten ejecutar varias acciones como una sola, sin tener que crear un árbol de comportamiento para ello, aplicando la lógica de los nodos compuestos directamente en las acciones. Se han creado dos tipos de acciones compuestas basadas en los nodos compuestos en serie y en paralelo respectivamente (ver fig. 3.13):

- Acción en serie (*SerialAction*): Acción que contiene un conjunto de subacciones y las ejecuta de una en una. Contiene un parámetro de tipo *Status* llamado *targetStatus* que determina que valor deben devolver las subacciones para que la ejecución pase a la siguiente. Si el valor de este parámetro es *Success*, la acción tendrá el comportamiento de un nodo secuencia, y si el valor es *Failure* el comportamiento será el de un nodo Selector.
- Acción paralela (*ParallelAction*): Ejecuta todas las subacciones en cada frame. A diferencia del nodo paralelo, las acciones paralelas deben almacenar directamente los resultados de las subacciones, pero el funcionamiento es esencialmente el mismo. Al igual que los nodos paralelos, puede parametrizarse cuando termina la ejecución en base al resultado de las subacciones (ver fig. 3.12).

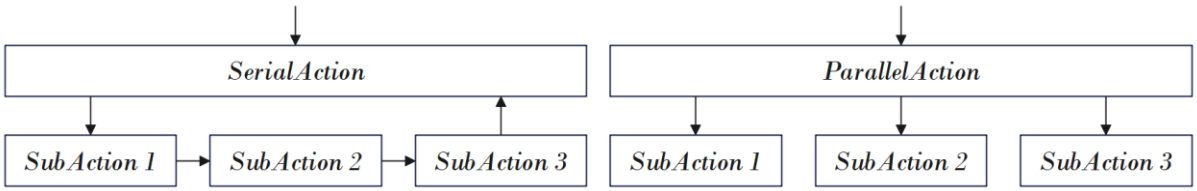


Fig. 3.13 Acciones compuestas en serie y en paralelo

Aunque la *API* solo incluya estos dos tipos de acciones compuestas, se da la posibilidad al usuario de crear más tipos de acciones compuestas creando una clase que herede de *CompoundAction*.

3.3.4 Nodos de selección de rama

Los nodos de selección de rama o *BranchNode* son un tipo de nodo compuesto que se basa en ejecutar solo uno de los nodos hijos en lugar de ejecutar todos, ya sea en serie o en paralelo. Cuando comienza la ejecución de un nodo de selección de rama, este escoge uno de los hijos y propaga los eventos únicamente a ese hijo hasta que termina su ejecución (ver fig. 3.14).

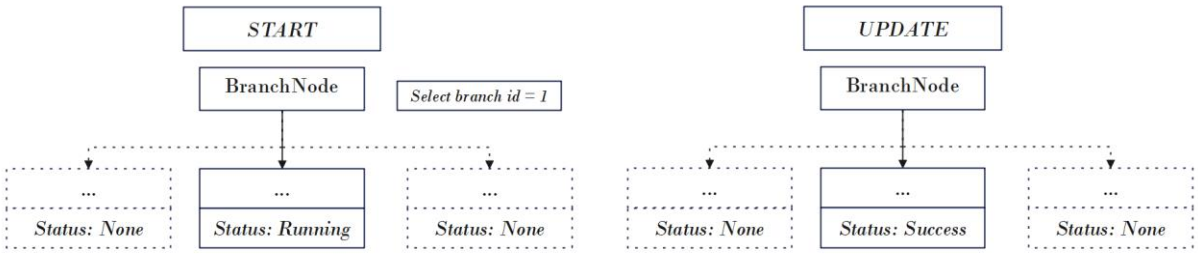


Fig. 3.14 Funcionamiento de un nodo de selección de rama

Se han creado tres tipos de nodos de selección de rama:

- *RandomBranchNode*: Escoge una de las ramas aleatoriamente.
- *ProbabilityBranchNode*: Escoge una de las ramas en base a una probabilidad. A cada nodo se le asigna un valor numérico, por defecto 0, y al comenzar la ejecución se calcula un valor entre o y la suma de todos los valores para decidir cuál es el nodo hijo escogido.
- *FunctionBranchNode*: Usa una función especificada por el usuario para calcular el índice del hijo seleccionado. Si la función devuelve un valor menor que 0 escogerá al primer hijo, y si el valor es mayor o igual a la cantidad de hijos, escogerá al último.

Además, se da al usuario la capacidad de crear más tipos de nodos de selección de rama heredando de la clase *BranchNode* e implementando el método *SelectBranchIndex* que devuelve el índice de la rama seleccionada.

3.3.5 Diagrama de clases de árboles de comportamiento

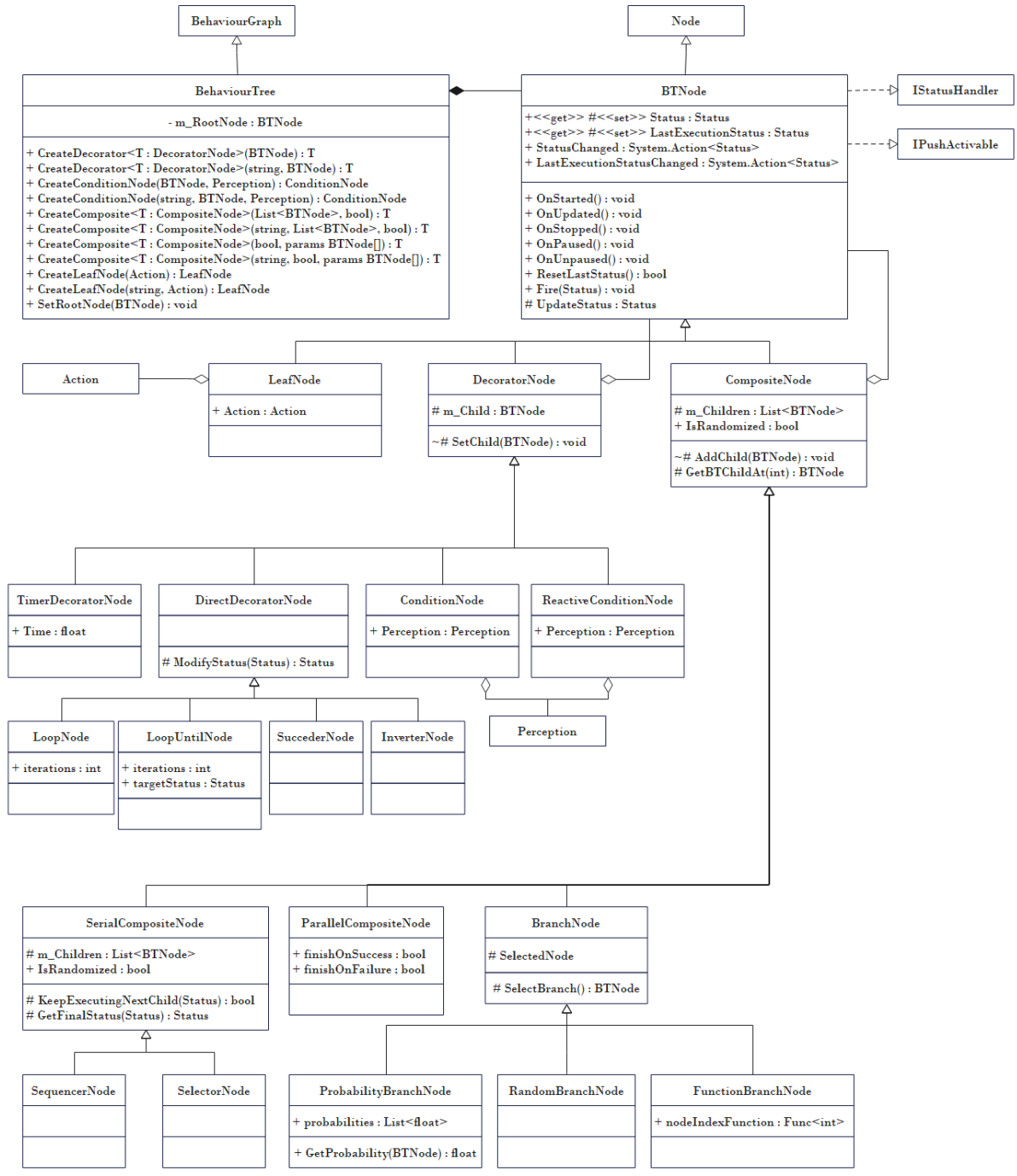


Fig. .3.15 Diagrama de clases de árboles de comportamiento

3.4 Máquinas de estados

Las máquinas de estados funcionan en base a dos tipos de nodos: estados y transiciones. Las transiciones pueden ser entre dos estados o transiciones de salida.

Para este segundo tipo se ha creado la clase *ExitTransition* que, en lugar de cambiar el estado actual de la máquina de estados, termina su ejecución cambiando su valor *Status* a *Success* o *Failure*.

También se ha modificado parte del funcionamiento de las transiciones. En primer lugar, aquellas que no tengan ninguna percepción asignada funcionarán como transiciones automáticas, también llamadas transiciones *epsilon* o *lambda* [31]. Esto quiere decir que se activarán siempre que sean comprobadas. Por defecto, las transiciones se comprueban cada vez que se lance el evento *Update* si su estado inicial es el estado actual de la máquina de estados, pero en algunos casos puede requerirse que no se pueda cambiar de estado mientras la acción que contiene no haya terminado de ejecutarse. Para ello, se ha añadido un parámetro de tipo *StatusFlags* en las transiciones, que permite especificar el valor o valores que puede tener el *Status* de un estado para que sus transiciones sean comprobadas. Además, combinando esta característica con las transiciones automáticas, es posible hacer que una transición se active cuando la ejecución de un estado ha terminado con un valor concreto. También puede usarse para que una transición nunca sea comprobada de forma interna, haciendo que solo sea activable mediante percepciones push.

A continuación, se detallan las extensiones implementadas para las máquinas de estados.

3.4.1 Transiciones Mealy

Las máquinas de estados pueden clasificarse en dos categorías dependiendo de que factores afectan a la salida [32]. Las máquinas de tipo *Moore* son aquellas en las que la salida depende únicamente del estado actual, sin importar como se ha transitado a ese estado, mientras que en las máquinas de tipo *Mealy* esto sí se tiene en cuenta. Si llevamos esta clasificación a las máquinas de estados enfocadas a sistemas de comportamiento, podemos suponer que los datos de entrada se corresponden con las percepciones y los datos de salida con las acciones, por lo que una maquina *Mealy* seria aquella que, para un mismo estado destino, ejecuta acciones distintas dependiendo de la transición que se ha usado para llegar hasta él.

Para implementar este comportamiento, se ha añadido la posibilidad de asignar acciones directamente a las transiciones, con la limitación de que estas se ejecutan únicamente en el *frame* en el que se lanza la transición. Para evitar posibles errores, se lanzarán los métodos *Start*, *Update* y *Stop* de la acción

justo antes de que la transición se lance y el estado cambie. La acción se lanzará tanto si la transición se ha activado de forma interna o externa, mediante percepciones push.

3.4.2 Estados probabilísticos

Los estados probabilísticos permiten asignarle un valor numérico a cada transición que parta del estado. Cada vez que las transiciones sean comprobadas, se calculará un valor entre 0 y la suma total de probabilidades de todas las transiciones, pero si la suma es menor a uno, el valor se calculará entre 0 y 1. Al recorrer las transiciones usará el valor generado para seleccionar una de ellas, de forma que solo esa transición será comprobada.

Si no se ha asignado probabilidad a una transición o su probabilidad es cero, esta tendrá prioridad a las transiciones que si tengan valor asignado. Esto significa que serán comprobadas siempre y se activarán con preferencia al resto de transiciones (ver fig. 3.16).

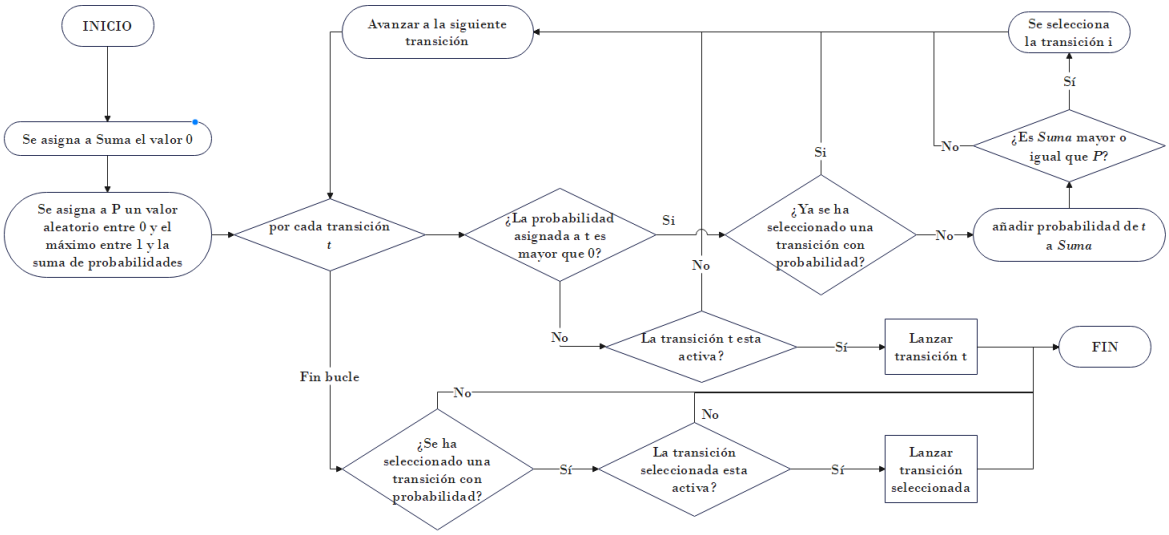


Fig. 3.16 Diagrama de flujo del método CheckTransitions en estados probabilísticos

3.4.3 Máquinas de estados de pila

La máquina de estados de pila o *StackFSM* es una variante de la máquina de estados que permite guardar los estados que se recorren en una estructura de datos de tipo *LIFO* (los elementos que se añaden últimos son los que se sacan primero). De esta forma es posible definir transiciones que en lugar de tener un estado destino, este varíe en función del último valor introducido en la pila. Este tipo de máquina de estados permite reducir el número de transiciones y estados necesarios.

Por ejemplo, en el diagrama siguiente (ver fig. 3.17) se muestran dos ejecuciones para una máquina de estados de pila con tres estados. En ambas ejecuciones se parte de un estado hacia el estado 3 y se guarda dicho estado en la pila. Después, se vuelve al estado inicial sacándolo de la pila.

Es posible implementar el mismo sistema sin utilizar una máquina de estados de pila, pero requeriría fragmentar el estado 3 en dos estados dependiendo de cual fuese el estado previo. Este problema aumenta cuanto mayor sea el número de estados que transitan al estado 3, mientras que usando una máquina de estados de pila no es necesario crear estados adicionales.

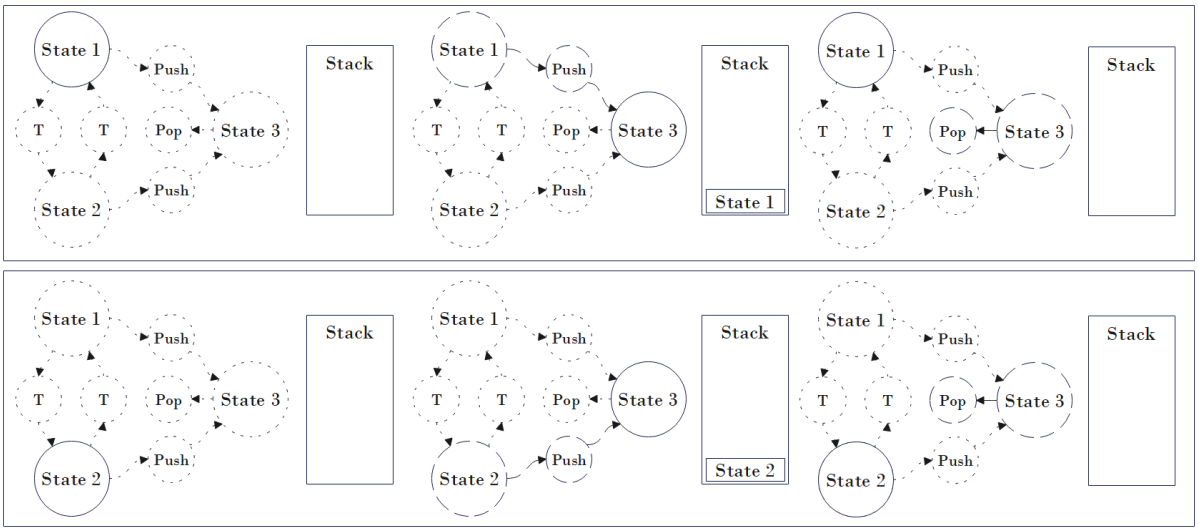


Fig. 3.17 Ejemplo de ejecución de una máquina de estados de pila.

Para implementar la máquina de estados de pila, se ha creado una clase *StackFSM* que hereda de la clase *FSM*. Esta clase añade una pila de estados y añade métodos específicos para crear y usar transiciones que manipulen esa pila. Estas transiciones son de dos tipos:

- *PushTransition*: Esta transición tiene un estado inicial y otro estado final. Cuando la transición se activa, el estado actual de la máquina de estados pasa a ser el estado final y el estado actual (es decir, el estado inicial de la transición) es guardado en la pila.
- *PopTransition*: Esta transición solo tiene un estado inicial. Cuando se activa, la máquina de estados saca un estado de la pila y transita a ese estado. Si la pila está vacía, la transición no hace nada.

Gracias a restringir a que tipo de grafo puede pertenecer un nodo además de que tipos de nodo puede tener un grafo, se puede garantizar que estas transiciones no se pueden usar desde una máquina de estados convencional, lo que evita errores en su ejecución.

3.4.4 Diagrama de clases de máquinas de estados

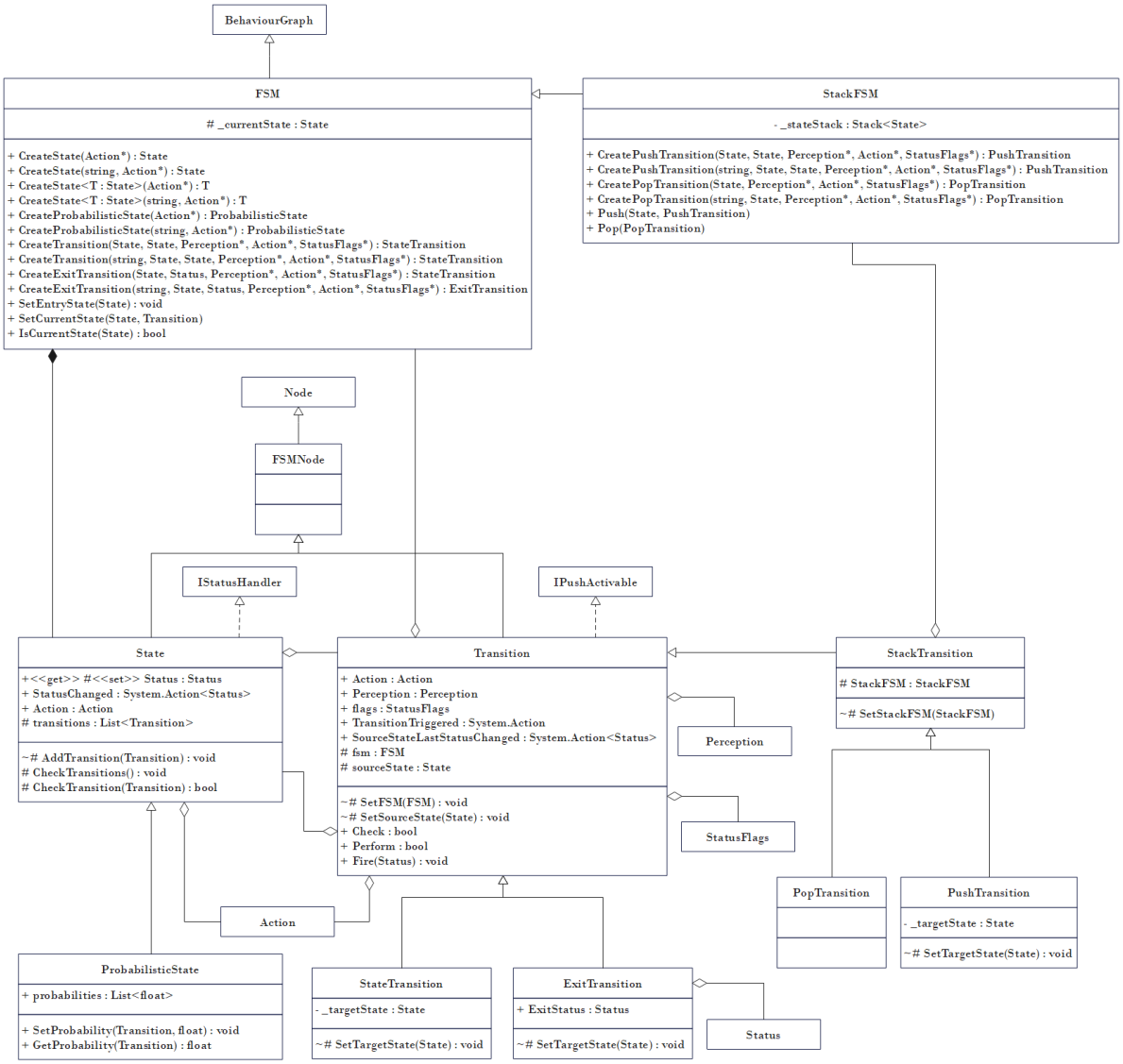


Fig. 3.18 Diagrama de clases de máquinas de estados

3.5 Sistemas de utilidad

Los sistemas de utilidad se han construido como grafos con dos tipos de nodos, los nodos seleccionables y los factores. Los nodos seleccionables son aquellos que pueden ser candidatos dentro del sistema de utilidad y los factores son elementos auxiliares que se usan para definir el valor de utilidad de los primeros.

También se ha añadido la posibilidad de terminar la ejecución de un sistema de utilidad, indicando mediante un *flag* que cuando un nodo acción termine de ejecutarse, el sistema de utilidad termine con

el mismo valor con el que ha terminado la acción. Esto es útil especialmente si el sistema de utilidad está dentro de una jerarquía y su sistema padre es un árbol de comportamiento. Además, se ha creado otro tipo de nodo seleccionable llamado *UtilityExitNode* que no contiene ninguna acción, si no que termina la ejecución del sistema de utilidad con un valor específico.

Se han añadido varios nodos y características a este esquema.

3.5.1 Factores constantes

Los factores constantes son una simplificación de los factores variables. En lugar de definir una función que devuelve un valor numérico que puede cambiar a lo largo del tiempo, los factores constantes se crean usando un valor concreto. A pesar del nombre, si es posible modificar el valor asignado al factor una vez se ha creado. La única diferencia es que este valor no está ligado a ningún parámetro del entorno.

El uso principal de estos factores es definir la utilidad de un nodo seleccionable de forma que actúe como "acción por defecto".

3.5.2 Curvas parametrizables

Las curvas parametrizables o *FunctionCurveFactor* son un tipo de factor curva que permite definir la función que se usará para modificar la utilidad del factor hijo, evitando así tener que crear un nuevo tipo de factor curva para cada función que se quiera usar. Esta curva parametrizable usa un delegado para almacenar la función que se va a usar. De esta forma, puede asignarse cualquier método a la función del factor siempre que sus parámetros y valor de retorno coincidan. En este caso el método debe recibir un valor *float*, que será la utilidad del factor hijo y devolver otro valor *float*.

3.5.3 Buckets

Los *buckets* o grupos de utilidad permiten definir grupos de acciones que tienen prioridad con respecto a otras acciones. Cada uno de estos grupos tiene un umbral de utilidad, cuyo valor define la utilidad mínima que debe tener alguna de las acciones para que se considere válida.

Cuando un sistema de utilidad tiene que calcular su mejor acción, en lugar de recorrer todos los nodos candidatos que contenga el sistema, solo recorrerá aquellos que no estén dentro de ningún grupo, es decir, aquellos que no tengan nodo padre. Los nodos que pertenezcan algún grupo se consideran candidatos dentro de dicho grupo, y se delega a éste la tarea de calcular el mejor candidato.

Para implementar los buckets se ha usado un patrón de diseño de *Composición* [33], utilizado en otras partes de la *API* como los factores fusión o los nodos compuestos en los árboles de comportamiento. Este patrón consiste en crear jerarquías mediante elementos, de forma que los propios elementos y sus composiciones se traten de la misma forma. En este caso, las acciones de utilidad representan los elementos básicos y los buckets representan las composiciones. Además, usar este patrón permite crear grupos dentro de otros y aplicar el cálculo de utilidad de forma recursiva.

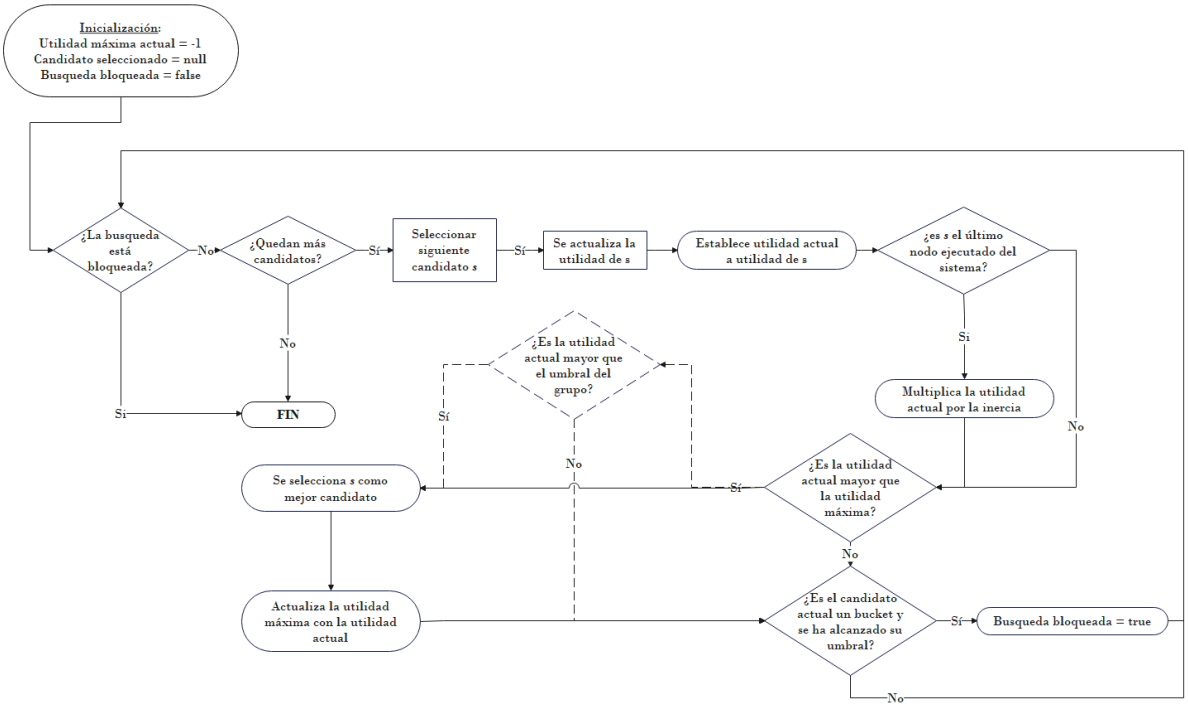


Fig. 3.19 Diagrama del algoritmo de selección de candidato en sistemas de utilidad y buckets.

Para calcular el elemento con mayor utilidad dentro de un *bucket*, se aplica el mismo algoritmo que se usa en los sistemas de utilidad, añadiendo la condición de que una acción solo es seleccionada si su utilidad es mayor o igual que el umbral. La utilidad de un bucket será la utilidad de su mejor candidato, o -1 si no hay ningún candidato válido. En el diagrama anterior (ver fig. 3.19) se muestra el algoritmo de cálculo del mejor candidato, marcando con líneas discontinuas la condición que solo se evalúa en los buckets.

3.5.4 Optimización de la evaluación de utilidad

La implementación de *buckets* ya reduce considerablemente la cantidad de veces que se evalúa la utilidad de las acciones, debido a que cuando se selecciona una acción de un grupo, los elementos posteriores no se tienen en cuenta en esa iteración. Aun así, si el sistema es muy grande, evaluar la utilidad de las acciones en cada frame puede crear un cuello de botella en la ejecución del sistema de comportamiento. Para reducir este problema, se ha añadido varias funcionalidades.

La primera de estas funcionalidades es incluir un "*dirty flag*". Esta técnica consiste en añadir un *flag* o valor booleano a los distintos elementos para indicar si su valor actual es válido o debe ser recalculado, permitiendo ahorrar los recursos que implicarían recalcular el valor cuando no es necesario.

En la implementación previa, si un factor es usado para calcular la utilidad desde dos puntos de la ejecución distintos, su utilidad era calculada dos veces, cuando solo la primera es necesaria.

1. Cuando se lanza el evento Update del sistema de utilidad, todos los nodos ponen su *dirty flag* a true, indicando que su utilidad debe ser recalculada.
2. Según se van recorriendo los nodos del sistema y recalculando su utilidad, su dirty flag se pone de vuelta en false. La actualización de la utilidad se propaga desde los nodos acciones (o grupos) hasta los factores, de padres a hijos.
3. Si se requiere la utilidad de un elemento que ya ha sido actualizado en esa iteración, en lugar de recalcularlo se devuelve directamente el valor de utilidad. En el diagrama (ver fig. 3.20), cuando se calcula la utilidad de la acción 2, no es necesario recalcular la utilidad del factor 3, puesto que ya se actualizó su valor al calcular la utilidad de la acción 1.

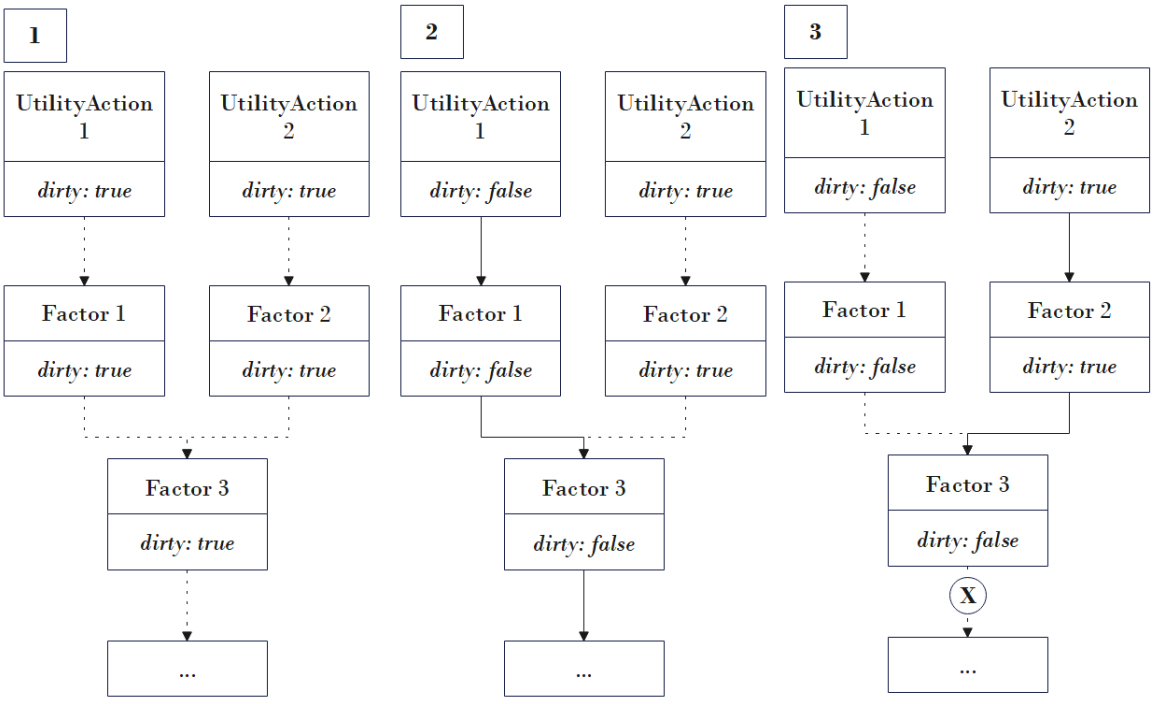


Fig. 3.20 Actualización de utilidad usando dirty flags.

Aunque de esta forma se reduce bastante el coste de actualizar la utilidad, hay que tener en cuenta que en muchos casos no es necesario recalcular la utilidad en cada frame, ya que los parámetros que se usan solo cambian en respuesta a eventos concretos (un personaje pierde vida, gasta o consigue munición, etc.), o incluso puede requerirse cambiar el intervalo en el que se recalcula.

Para permitir a los usuarios decidir cuándo se va a recalcular la utilidad de un nodo, se ha creado la propiedad "*PullingEnabled"*. Esta propiedad es de tipo booleano y tiene valor *true* por defecto. Si su valor es *true*, permite que el evento de recalcular la utilidad se propague a sus nodos hijos, mientras que si es false, la única forma de actualizar la utilidad de ese nodo será lanzando el método *UpdateUtility* manualmente. Aplicar esa propiedad a todos los nodos del sistema de comportamiento permite al usuario tener un control total de cuando actualizar los valores de utilidad de cada uno de ellos.

3.5.5 Diagrama de clases

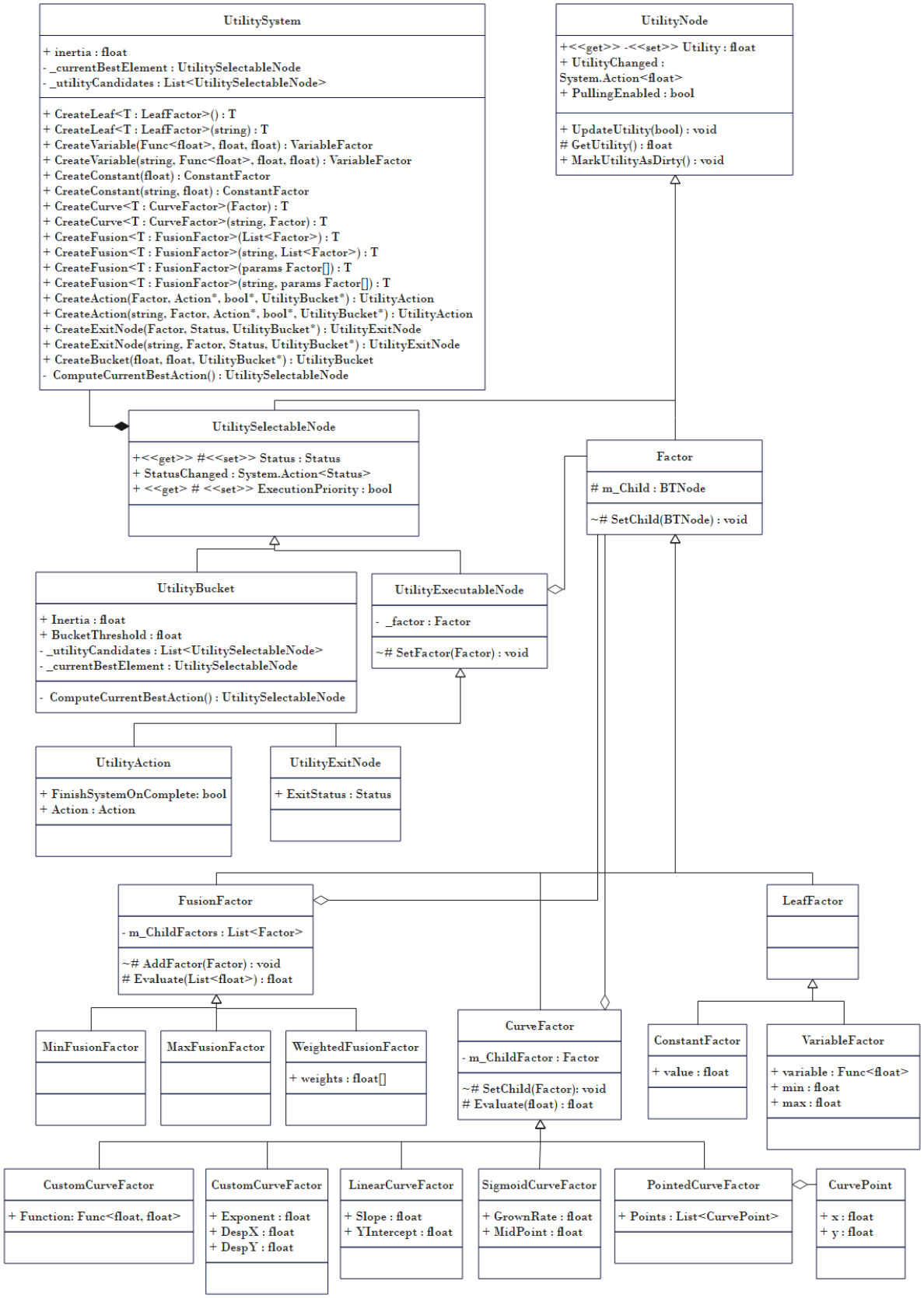


Fig. 3.21 Diagrama de clases de sistemas de utilidad

3.6 Smart Objects

En inteligencia artificial y modelos de sistemas inteligentes, el término *Smart Object* puede hacer referencia a varias cosas. Dentro de este proyecto, se definen por las siguientes características:

- Proporcionan un sistema de comportamiento al agente que quiere usarlo. De esta forma, el agente no tiene que saber cómo usar el objeto, sino que es el objeto el que tiene que conocer que acciones puede realizar el agente y sus características, e indicarle como debe usarlo a través de un sistema de comportamiento.
- Pueden cubrir las distintas necesidades del agente. Los objetos pueden tener la capacidad de cubrir ciertas necesidades definidas en el agente y éste debe poder saber cuáles son las necesidades que puede cubrir el objeto. Esto permite al agente decidir que objeto usar en cada momento.

A la hora de decidir como implementar un sistema de *SmartObject* es necesario decidir antes qué elementos del agente pueden conocer y modificar los objetos, lo cual depende de en qué ámbito se usen. Por ejemplo, si en un videojuego se quiere implementar un SmartObject que haga que el personaje se mueva a un punto concreto, el agente dispondrá de un método *Move* al que se pasa la posición específica. En otros entornos, puede que mover el agente no tenga sentido, o puede que el sistema de coordenadas sea distinto, por lo que no es posible definir un conjunto de propiedades y métodos comunes para todos los agentes. Por todo esto, se ha implementado un sistema abierto y adaptable a distintos entornos, en el que el elemento central es el propio agente, y el resto de los elementos dependen de él.

Para implementar un sistema de objetos inteligentes, primero hay que definir las características del agente. Estos agentes se definen usando la interfaz *ISmartAgent*, que establece los métodos básicos que deben tener:

- *GetNeed:* Devuelve el valor de una necesidad del agente cuyo nombre se pasa como parámetro al método. Esto puede usarse para crear factores que usen la necesidad en un sistema de utilidad.
- *CoverNeed:* Recibe como parámetros el nombre de la necesidad y el valor que se debe aplicar. La forma de cubrir la necesidad depende de cómo se hayan implementado las necesidades en el agente.

Una vez creado el agente, se crean los propios objetos usando la interfaz *ISmartObject*. Esta interfaz es genérica en función al tipo de agente que maneja, lo que permite usar las características del agente

en concreto a la hora de proporcionarle un sistema de comportamiento. Obliga a implementar los siguientes métodos:

- *GetCapabilityValue*: Devuelve la capacidad del objeto de cubrir una necesidad concreta.
- *ValidateAgent*: Devuelve un booleano para indicar si un agente puede usar el objeto o no.
- *RequestInteraction*: Este método proporciona el sistema de comportamiento específico al agente.

Para integrar esta funcionalidad con el resto de la API se ha creado la clase *RequestAction*, que consiste en un tipo especifico de *Action* que permite acceder a los distintos *SmartObjects* y usarlos. Esta clase obtiene un comportamiento generado por un *SmartObject* y le propaga sus eventos de ejecución, de forma que su método *Update* devuelva el resultado del comportamiento proporcionado (ver fig. 3.22).

Este comportamiento consiste en un objeto de la clase *Action* en lugar de ser un sistema de comportamiento (*BehaviourEngine*), lo que da más flexibilidad a los objetos a la hora de generarlos. Si se quiere proporcionar un sistema de comportamiento completo, se devuelve una acción de tipo *SubsystemAction* con el sistema referenciado en la variable *subsystem*.

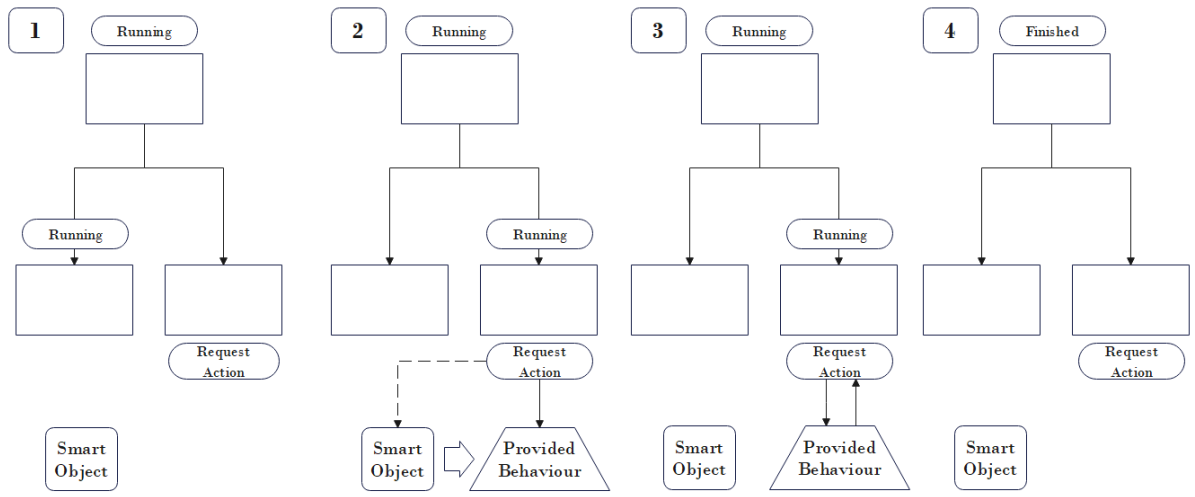


Fig. 3.22 Esquema de uso de Smart Objects

Como muestra el siguiente diagrama (ver fig. 3.23), el uso de los objetos inteligentes se basa en un sistema de petición respuesta. Cuando la ejecución llega a una *RequestAction*, esta manda una solicitud a un *SmartObject*, y este construye una respuesta en forma de sistema de comportamiento en base al agente y a otros datos incluidos en la petición. Una vez recibida la respuesta, la *RequestAction* actua como intermediario entre el nodo y el comportamiento inyectado hasta que termina la ejecución o es interrumpida.

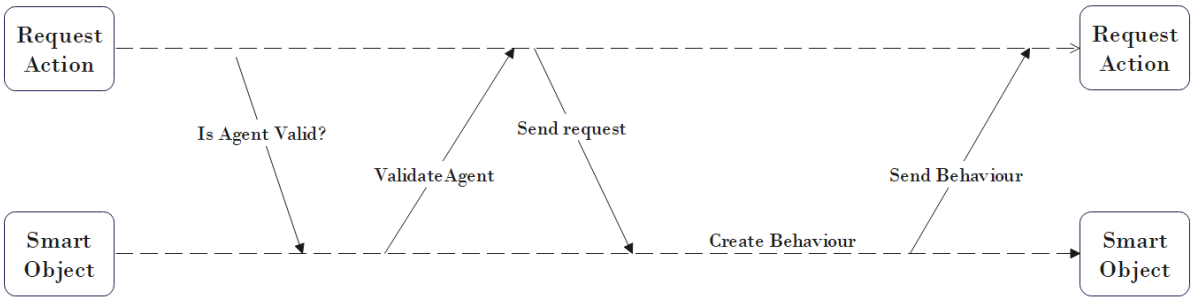


Fig. 3.23 Esquema petición respuesta entre RequestActions y SmartObjects

El flujo de ejecución de un agente que usa un *SmartObject* se divide en 5 fases: búsqueda, selección, validación, petición e interacción.

3.6.1 Búsqueda

La búsqueda es la parte en la que se definen cuáles son los *Smart objects* que el agente puede utilizar. Esta fase depende completamente del entorno en el que se estén utilizando los *Smart objects* y existen múltiples variantes. Puede que un personaje concreto solo tenga acceso a los objetos en un rango determinado, o que sean los propios objetos los que envíen una señal al agente indicando que están disponibles usando un sistema de paso de mensajes.

Debido a esto, se ha decidido que la búsqueda no se implemente en la API base, sino que se delegará a la implementación específica de la API en cada entorno concreto.

3.6.2 Selección

Una vez establecidos los objetos que un agente inteligente tiene disponibles, hay que definir como se escoge el objeto que se va a usar. Esta tarea recae sobre la propia acción, concretamente en el método *GetSmartObject*, por lo que cada tipo de *RequestAction* implementará la selección del *Smart Object* de una forma distinta.

Aun así, se ha incluido en la librería una implementación simple de *RequestAction* llamada *DirectRequestAction*: Permite definir directamente que *Smart Object* se va a utilizar, por lo que las fases de búsqueda y selección quedan resueltas.

En implementaciones más complejas, es posible acceder a cierta información de los Smart Object para hacer la selección:

- Tags: Son una colección de *strings* que define cada *Smart Object*. Es el propio agente, a través de las *RequestActions* el que debe dar significado a esas *strings*. Los *tags* pueden usarse para limitar

que objetos puede usar un agente. Por ejemplo, si se establece que un NPC debe seguir una dieta vegana, solo podrá usar objetos con la etiqueta “vegano” para satisfacer su necesidad “hambre”. Por otro lado, también pueden usarse para restringir el uso de ciertos objetos. Si se define que un agente es alérgico a un alimento en específico, algunos objetos pueden marcarse con una etiqueta para que el agente sepa que no puede usarlos.

- Necesidades: Los objetos permiten a los agentes saber que necesidades pueden cubrir y con que valores, mediante el método *GetCapability*. De esta forma se puede seleccionar el objeto que más convenga en cada momento.

3.6.3 Validación

La validación es el proceso por el cual el *SmartObject* comprueba que el agente puede hacer uso de él, a través del método *ValidateAgent*. Dependiendo de la implementación de la *RequestAction* concreta, si la validación devuelve *false* puede buscarse otro objeto o no. Si se sale del evento *Start* sin un comportamiento proporcionado, se devolverá *Failure* directamente en el evento *Update*.

Algunos usos posibles de la validación son los siguientes:

- El objeto en concreto solo puede ser usado por un agente a la vez, y está ocupado en el momento en el que recibe la petición.
- El objeto puede usarse por varios objetos, pero se ha alcanzado el número máximo.
- El objeto establece una serie de requisitos que debe tener el agente que lo usa.

3.6.4 Petición

Tras seleccionar el *Smart Object* que se quiere usar y validar el uso, el siguiente paso es definir qué datos se van a enviar a dicho objeto para que genere la acción que el agente va a ejecutar.

Se ha decidido establecer la necesidad que se quiere cubrir como parámetro para las peticiones a los *Smart Objects*. De esta forma un mismo objeto puede usarse de formas distintas para cubrir varias necesidades. Como ejemplo de esto se propone un escenario en el que los agentes tienen varias necesidades, entre ellas aprendizaje y entretenimiento. Uno de los Smart Objects disponibles es una estantería con libros que puede usarse de varias formas distintas. Si se usa para cubrir la necesidad de aprendizaje, el agente podría consultar una enciclopedia o un diccionario. Sin embargo, si se usa para entretenimiento, el agente podrá coger una novela y sentarse en un lugar cercano para leer. El objeto proporcionaría comportamientos distintos al agente dependiendo de qué necesidad desee cubrir.

En lugar de pasar el nombre de la necesidad directamente como parámetro, se ha encapsulado en la clase *RequestData*. De esta forma, es más fácil modificar que datos se envían en función de las necesidades concretas, y en caso de querer enviar únicamente la necesidad, se proporciona un operador de casteo implícito de *string* a *RequestData* para facilitar el uso.

3.6.5 Interacción

Esta fase engloba todo el proceso desde que el *SmartObject* recibe la petición hasta que se termina de ejecutar el comportamiento proporcionado.

Además del propio comportamiento, se deben devolver otros datos relevantes para la ejecución, por lo que se ha creado la clase *SmartInteraction* para contener todos esos datos, y en lugar de devolver un objeto de la clase *Action* a la petición se devolverá un objeto de esta clase.

El siguiente diagrama muestra uno de los problemas que pueden surgir a la hora de ejecutar una acción proporcionada por un objeto inteligente (ver fig. 3.24). En dicho diagrama aparece un árbol de comportamiento formado por una secuencia con tres nodos hojas. El objeto establece que solo puede ser usado por un agente simultáneamente, por lo que define una variable booleana llamada *"Owned"* cuyo valor será *true* mientras dure la acción. En una ejecución normal la acción terminaría y no habría ningún error, pero el problema aparece si la secuencia es interrumpida antes de llegar al último nodo, ya que el valor de *owned* se mantendría en *true* tras salir de la acción.

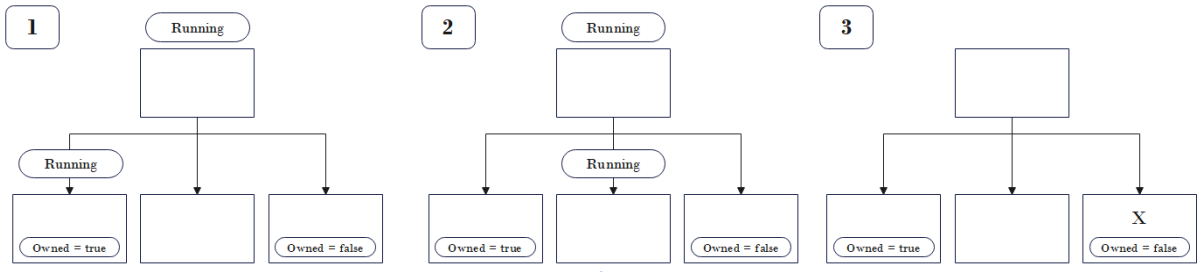


Fig. 3.24 Ejemplo de interrupción de un comportamiento proporcionado por un Smart Object.

La solución a este problema es definir una serie de eventos en la clase *SmartInteraction* que se llamarán en ciertas situaciones:

- *OnInitialize:* Se llama cuando comienza la ejecución de la acción.
- *OnComplete:* Se llama cuando la ejecución de la acción termina con *Success* o *Failure*. El evento recibe como parámetro el valor *Status* concreto.
- *OnRelease:* Se llama cuando la ejecución de la acción se detiene, haya terminado o no.
- *OnPause:* Se llama cuando la acción se pausa.
- *OnUnpause:* Se llama cuando la acción se despausa.

El problema presentado anteriormente queda resuelto si en lugar de modificar el valor de la variable *owned* dentro del propio árbol, se hace usando los eventos *OnInitialize* y *OnRelease*.

Si la acción de la interacción se completa con éxito, se aplican las capacidades a las necesidades del agente. Cuando el objeto crea la interacción, establece las capacidades en el constructor mediante un diccionario *string-float*, que relaciona cada necesidad a cubrir con su valor. Esto permite que las capacidades del objeto no sean estáticas y además que puedan variar en función de la interacción concreta. Volviendo al ejemplo de la fase de petición, si la *RequestAction* ha especificado en la petición que la necesidad que quiere cubrir es "aprendizaje", cuando el objeto genere la interacción, establecerá que si se completa con éxito la necesidad que se cubrirá será la de aprendizaje (ver fig. 3.25).

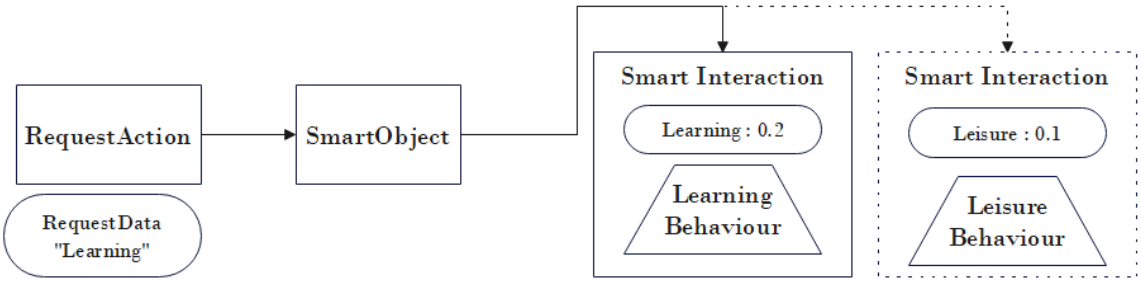


Fig. 3.25 Ejemplo de SmartObject con dos interacciones posibles dependiendo de la necesidad a cubrir.

3.6.6 Smart Objects enlazados

La arquitectura desarrollada permite crear *Smart Objects* que pueden usar otros *SmartObjects* para generar el sistema de comportamiento de sus interacciones. Para hacer esto hay dos posibilidades.

- Al generar la interacción en el objeto principal, establecer la acción de dicha interacción como una *RequestAction* que buscará el objeto secundario. En este esquema el primer objeto solo actúa de intermediario entre el agente y el segundo objeto.
- Incluir una o varias *RequestActions* dentro del sistema de comportamiento proporcionado por el objeto principal. De esta forma es posible anidar cualquier cantidad de objetos, aunque es importante evitar referencias cíclicas en las peticiones.

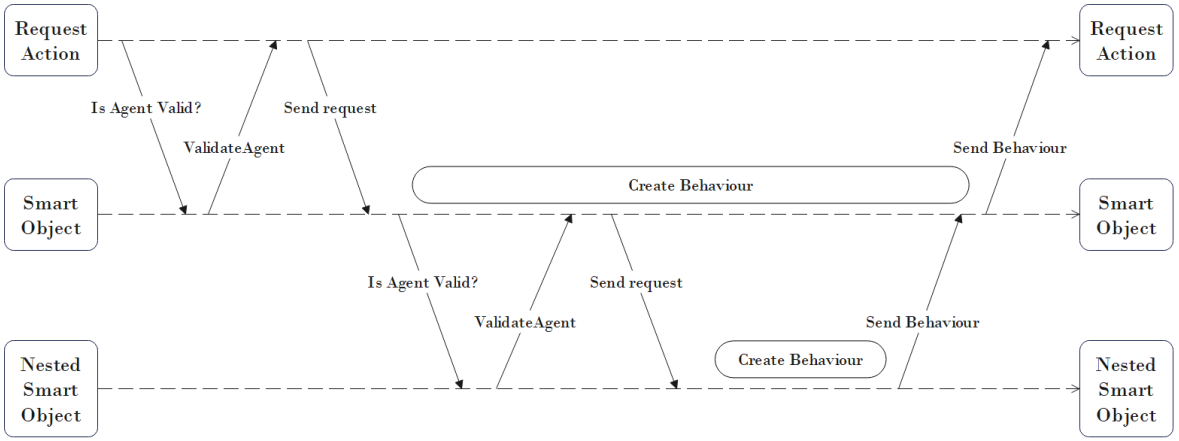


Fig. 3.26 Esquema de petición-respuesta con un Smart Object enlazado

El esquema anterior (ver fig. 3.26) muestra un ejemplo de ejecución de un *Smart Object* anidado.

3.6.7 Diagrama de clases de Smart Object

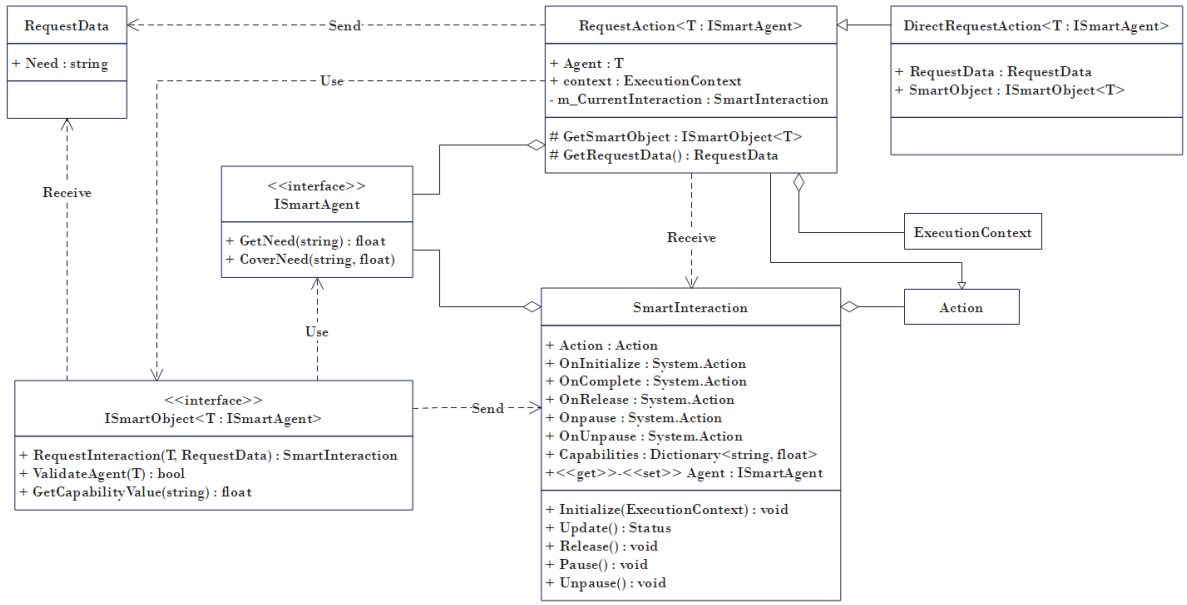


Fig. 3.27 Diagrama de clases de Smart Objects

4

Pruebas y documentación

4.1 Pruebas unitarias

Las pruebas unitarias se han desarrollado usando *MSTest* [34], un framework de testing integrado en Visual Studio y enfocado a la plataforma *.NET*. Para cada subproyecto de la *API* completa se ha creado una carpeta con uno o varios archivos para organizar los distintos tests. Tras implementar un nuevo tipo de nodo o una funcionalidad concreta, se han ido añadiendo uno o varios tests para probar los distintos casos de uso de cada uno de ellos.

Para complementar los test de Visual Studio se ha construido un sistema de integración continua en el repositorio de *GitHub*, a través de *GitHub Actions* [35], un mecanismo que permite automatizar ciertas tareas en respuesta a eventos específicos del repositorio. Estas acciones se definen mediante ficheros de configuración con extensión "*.yml"* guardados en una carpeta especial llamada *workflows*.

Para los test se ha creado un archivo "*build-tests.yml"* que define un flujo de trabajo que consiste en utilizar uno de los servidores de *GitHub* para subir todo el contenido del repositorio y ejecutar los test. Una vez terminado el proceso, es posible ver el resultado en detalle en la pestaña *Actions* (ver fig. 4.1) y en caso de que haya habido algún error, se envía un correo electrónico al dueño del repositorio.

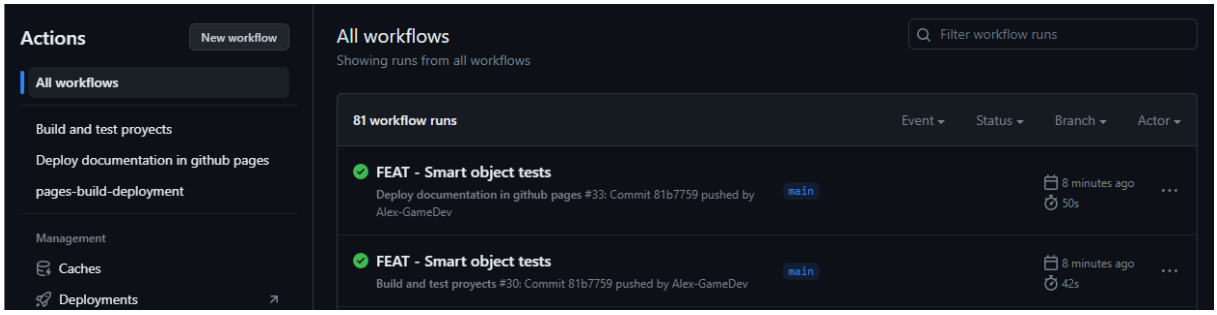


Fig. 4.1 Sistema de integración continua en GitHub

4.2 Demo en C#

Además de los test unitarios, se ha creado una demo para probar los distintos sistemas de comportamiento, que consiste en una aplicación de consola que simula un videojuego de combate por turnos.

En esta aplicación el jugador se enfrenta a un personaje controlado mediante un sistema de comportamiento. En cada turno del juego, tanto el jugador como el personaje enemigo escogen una acción y la ejecutan. El juego termina cuando la vida de uno de los dos llegue a 0.

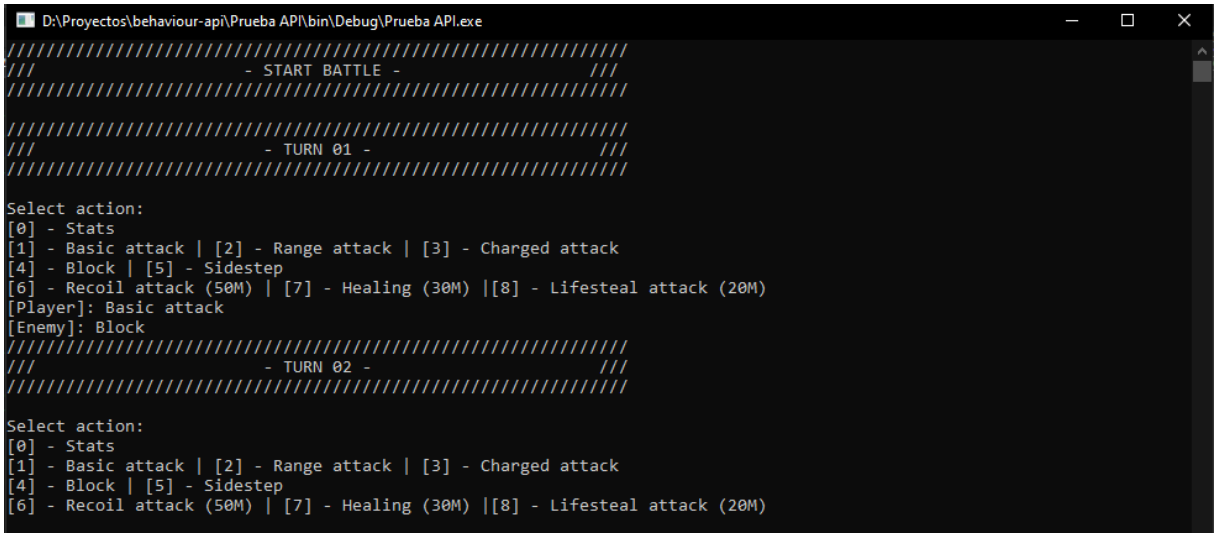


Fig. 4.2 Salida por consola de la demo en C#

Para esta demo, se han desarrollado dos sistemas de comportamiento, para el personaje y para el jugador.

4.2.1 Sistema de comportamiento del jugador.

El sistema de toma de decisiones del jugador se construye como una máquina de estados con un estado inicial conectado a otro estado por cada una de las acciones que el jugador pueda realizar. Cuando el estado inicial es el estado activo se pedirá al jugador que escoja una de las acciones disponibles usando las teclas numéricas. El valor introducido se guardará en una variable interna y al comprobar las transiciones de salida se cambiará al estado correspondiente. Cuando la acción termine (en ese turno, o en el siguiente dependiendo de la acción) se volverá al estado inicial para repetir el proceso.

4.2.2 Sistema de comportamiento del personaje enemigo

El grafo principal del personaje enemigo es una máquina de estados. El estado “combate” ejecuta un subsistema de utilidad para elegir una de las acciones. Si se escoge la acción “run” del sistema de utilidad, se pondrá el valor de la variable “*isRunning*” a true, lo que activará una transición al estado “*escapeState*”. La acción de este estado termina aleatoriamente en Success o Failure y dependiendo del resultado se transiciona al estado “*Trapped State*” o se volvería al estado de combate.

Cuando el personaje pasa al estado “*Trapped”* queda bloqueado durante dos turnos y no puede realizar acciones. Si el subsistema de utilidad escoge la acción de ataque especial se entra en un subárbol de comportamiento para escoger una de las tres acciones especiales en función de las estadísticas del personaje.

4.3 Documentación de la API

Para facilitar a los usuarios el uso de la librería y tener un registro de todas las funcionalidades se proporciona varios documentos con guías e información útil de la herramienta. Además, mediante el uso de la herramienta *Docfx* [7] se proporciona la documentación completa de la *API*, con todas las clases, propiedades y métodos.

Al igual que los test unitarios, se ha creado una *Github Action* para generar la documentación de forma automática cada vez que se actualiza la rama principal del repositorio. Además, esta documentación se ha generado en formato *html* y se ha subido a las *Github Pages* del repositorio para que cualquier usuario pueda acceder a ella.

5

Conclusión y trabajo futuro

5.1 Trabajo realizado

En este proyecto se ha desarrollado una librería en C# para crear distintos tipos de sistemas de comportamiento como máquinas de estados, árboles de comportamiento o sistemas de utilidad en forma de grafos.

Esta librería no se ha creado desde 0, sino que consiste en una refactorización de una versión previa. Aunque la mayor parte del código de la API actual se ha desarrollado en este proyecto, el hecho de contar con una versión anterior ha servido como guía para entender que funcionalidad había que implementar y como debían funcionar los distintos tipos de sistemas.

El primer paso del desarrollo ha sido resolver el principal problema que tenía la librería, la coherencia entre los distintos sistemas. Por ello, se planteó un modelo que consistía en construir los distintos tipos de sistemas como grafos dirigidos y ejecutarlos en base a eventos que alteran el estado interno de los grafos y sus nodos. A pesar de que surgieron algunos problemas debido a las diferencias entre los sistemas, se ha logrado completar este objetivo.

Una vez establecido el núcleo del proyecto, se han realizado la refactorización de cada uno de los sistemas de comportamiento en base al modelo planteado. La mayoría de los problemas en este apartado han surgido a la hora de crear sistemas jerárquicos y establecer que elementos de cada tipo de sistema usar para manejar los subsistemas. Aunque se han realizado cambios que no estaban previstos en un principio, como el diseño de las transiciones de salida o los nodos acción en sistemas de utilidad, el resultado cumple con el objetivo de mantener independiente el código de los distintos sistemas. Después de adaptar cada sistema de comportamiento a la nueva implementación de grafos y eventos, se han implementado las distintas extensiones y nuevas funcionalidades.

En las máquinas de estados, los cambios a las transiciones permiten definir acciones dentro de las mismas y darles comportamiento de “FSM Mealy”, además se añadió la posibilidad de definir cuando serán comprobadas en base al estado origen para tener aún más control sobre su funcionamiento.

En árboles de comportamiento, se ha desarrollado una jerarquía de nodos en base a los patrones *decorator* y *composite*, facilitando así la inclusión de nuevos nodos. Esto ha permitido añadir de forma simple nuevos tipos de nodos decoradores y compuestos como el nodo condicional reactivo, los nodos compuestos paralelos y los nodos de selección de rama, manteniendo todos los tipos que ya incluía la API original.

En sistemas de comportamiento, se ha diseñado la jerarquía de nodos, diferenciando entre factores de decisión y nodos seleccionables. La jerarquía de factores se ha basado en el modelo utilizado en árboles de comportamiento, distinguiendo los tipos de factor en base a la cantidad de nodos hijos, y facilitando así al usuario crear sus propios nodos. Además, se ha añadido un nodo especial llamado *bucket* que permite agrupar acciones y darles un nivel de prioridad, y se han añadido distintas herramientas para optimizar el cálculo de la utilidad de los distintos nodos, como la posibilidad de activar y desactivar la actualización automática.

Una vez desarrollada la refactorización de la API y añadidas nuevas funcionalidades a cada tipo de sistema de comportamiento, se decidió trabajar en una nueva funcionalidad para la librería, y tras plantear varias ideas se decidió desarrollar un sistema de *SmartObjects*. Desarrollar este sistema ha sido considerablemente más complejo que el resto de las partes del proyecto. En un primer momento, se intentó crear una API cerrada y que el usuario pudiera usar directamente sin tener que crear ninguna clase propia, pero uno de los objetivos principales era que la herramienta se pudiese usar en distintas disciplinas, no solo en desarrollo de personajes para videojuegos. Por esto, se cambió el enfoque del diseño de la API a un modelo abierto que el usuario adaptase a las necesidades concretas de cada entorno. De esta forma, aunque requiere algo más de trabajo por parte del usuario, se encontró una solución válida para varios entornos.

El uso de los *Smart Objects* se basa en un esquema de petición-respuesta. Esto, además de ser relativamente sencillo de implementar, facilita al usuario entender el proceso de uso de los objetos, cumpliendo el objetivo de hacer que el funcionamiento interno de la API sea comprensible para los usuarios.

5.2 Trabajo futuro

En esta sección se proponen funcionalidades que se pueden añadir a la librería en un futuro.

5.2.1 Nuevos sistemas de comportamiento

Ya que una de las metas conseguidas en el proyecto ha sido crear un modelo común para todos los sistemas de comportamiento, una posible extensión para la API actual podrá consistir en añadir más tipos de sistemas, ya sean basados en grafos o no.

Algunos de los sistemas que se podrían desarrollar serían redes neuronales, árboles de decisión o sistemas de Q-Learning. Aunque se tratan de técnicas de aprendizaje por refuerzo, podría ser interesante su uso en combinación con otras técnicas.

En cuanto a modelos no basados en grafos, se podrían implementar sistemas de subsunción o *GOAP (Goal Oriented Action Planning)*, aprovechando el sistema de acciones incluido en la *API*.

5.2.2 Árboles de comportamiento dinámicos

Una de las limitaciones en la implementación de árboles de comportamiento es que no es posible eliminar y crear ramas en tiempo de ejecución, por lo que sería interesante crear o bien un tipo de nodo o bien una variante de árbol de comportamiento que si lo permita.

5.2.3 Sistemas basados en lógica borrosa

La lógica borrosa o difusa [36] se basa en expresiones cuyo resultado no tiene por qué ser cierto o falso, sino que posee un cierto grado de verdad, normalmente establecido como un valor entre 0 y 1.

Este modelo podría servir para crear agentes que no ejecuten solo una acción en cada instante sino varias, cada una de ellas con un valor que definiría la "intensidad" con la que el agente realiza la acción. Por ejemplo, si un agente en un videojuego tipo FPS (First person shooter) quiere moverse hacia una posición concreta y a la vez recargar su arma, puede hacer ambas acciones simultáneamente, aunque se mueva más despacio de lo normal y tarde más en recargar.

Una posible integración de la lógica difusa sería diseñar máquinas de estados difusas (*Fuzzy Finite State Machines*) [37]. En este tipo de máquinas de estados, es posible estar en varios estados a la vez, con un valor específico, y las transiciones propagan dicho valor entre estados.

Otro elemento más sencillo de implementar podría ser los sistemas de utilidad con acciones difusas, en los que, en lugar de escoger la mejor acción en base a la utilidad, se usa la propia utilidad para definir con que intensidad se realiza cada acción.

5.2.4 Guardar grafos en ficheros

Aunque actualmente la herramienta puede usarse en cualquier entorno que soporte .NET Standard 2.1, podría ser interesante añadir la posibilidad de guardar los grafos de comportamiento en ficheros en algún lenguaje de representación de datos como *json* o *xml*. Esto permitiría crear otras implementaciones de la API en otros lenguajes como *Java* o *Python*, además de crear una forma cómoda de compartir sistemas de comportamiento sin tener que compartir el código directamente.

5.3 Conclusiones

En definitiva, se ha logrado crear API que mantiene los puntos fuertes de la versión anterior, pero mejorando aspectos como el funcionamiento interno de los sistemas, y añadiendo funcionalidad que aporta flexibilidad a la herramienta, manteniendo su simplicidad.

El mayor problema actualmente es no haber podido probar la API en usuarios reales antes de terminar el proyecto, pero se espera que tenga buenos resultados ya que la interfaz se ha mantenido muy similar a la versión anterior.

A nivel personal, este proyecto me ha servido para poner a prueba los conocimientos de programación que he adquirido durante la carrera, desde uso de estructuras de datos hasta patrones de diseño o complejidad de algoritmos. Además, considero que desarrollar una API con una implementación base y con ciertas limitaciones impuestas es una experiencia más enriquecedora que desarrollar un proyecto libre desde cero.

6

Bibliografía y referencias

[1] IBM. ¿Qué es el Deep Learning? [Online] Disponible en: https://www.ibm.com/es-es/topics/deep-learning

[2] C. V. Nicholson. A Beginner's Guide to Neural Networks and Deep Learning. [Online]. Disponible en: https://wiki.pathmind.com/neural-network

[3] Unity Technologies. Unity. [Online]. Disponible en: https://unity.com/es

[4] Notion Labs. Your wiki, docs & proyects together. [Online]. Disponible en: https://www.notion.so/

[5] Microsoft. Visual Studio. [Online]. Disponible en: https://visualstudio.microsoft.com/es/

[6] Github. Github. [Online]. Disponible en: https://github.com

[7] DocFx. Quick Start. [Online]. Disponible en: https://dotnet.github.io/docfx/

[8] Toal, Daniel & Flanagan, Colin & Jones, Caimin & Strunz, Bob. (1995). Subsumption Architecture for the Control of Robots. Proceedings Polymodel-16.

[9] Brooks, Rodney (1999). Cambrian Intelligence: The Early History of the New AI. Cambridge, Massachusetts: The MIT Press. ISBN 978-0-262-02468-6.

[10] Adservio (2022, Jun). Introduction to state machines and use cases. [Online]. Disponible en: https://www.adservio.fr/post/introduction-to-state-machine

[11] Plazas, David & Cárdenas-Rodríguez, Juan. (2018). Bayesian Statistical Analysis in PacMan. 10.13140/RG.2.2.10783.30888.

[12] C. Simpson (2014, Jul). Behavior trees for AI: How they work. [Online]. Disponible en: https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work

[13] Bruno, Mario & Aldunate, Roberto & Melendez, Jaime. (2014). Personalization of Serious Video Games for Self Care in Aging. Latin America Transactions, IEEE (Revista IEEE America Latina). 12. 484-490. 10.1109/TLA.2014.6827877.

[14] V. Chaudhari (2017, Dec). Goal Oriented Action Planning. [Online]. Disponible en: https://medium.com/@vedantchaudhari/goal-oriented-action-planning-34035ed40d0b

[15] D. Graham. An Introduction to Utility Theory. [Online]. Disponible en: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter09_An_Introduction_to_Utility_Theory.pdf

[16] M. Černý, T. Plch, M. Marko, J. Gemrot, P. Ondráček and C. Brom (June, 2017). Using Behavior Objects to Manage Complexity in Virtual Worlds, in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 9, no. 2, pp. 166-180, doi: 10.1109/TCIAIG.2016.2528499.

[17] Gaia-ucm. JBT Github repository. [Online]. Disponible en: https://github.com/gaia-ucm/jbt

[18] Auryn Robotics. BehaviourTree.CPP. [Online]. Disponible en: https://www.behaviortree.dev/

[19] Auryn Robotics. Groot 2. [Online]. Disponible en: https://www.behaviortree.dev/groot/

[20] PyTrees. PyTrees. [Online]. Disponible en: https://py-trees.readthedocs.io/en/devel/

[21] A. Davis (2016, Feb). Fluent behavior trees for AI and game-logic. [Online]. Disponible en: https://www.the-data-wrangler.com/fluent-behavior-trees-for-ai-and-game-logic/

[22] G. Gasking (2021, Jul). C# fluid interfaces: the good and (maybe) the bad and the ugly. [Online]. Disponible en: https://www.linkedin.com/pulse/c-fluid-interfaces-good-maybe-bad-ugly-george-gaskin

[23] M. Hjort (2017). Aivo. [Online]. Disponible en: https://github.com/mhjort/aivo

[24] F. Ziegelmayer. Behaviour Tree API Reference [Online]. Disponible en: https://hexdocs.pm/behavior_tree/api-reference.html

[25] Elixir team. Elixir. [Online]. Disponible en: https://hexdocs.pm/behavior_tree/api-reference.html

[26] Dotnet State Machine. Stateless. [Online]. Disponible en: https://github.com/dotnet-state-machine/stateless

[27] Desarrollo de un paquete para programación de comportamiento de personajes en videojuegos. Autor: Pablo Manuel Sánchez Trujillo. Tutor: Carlos Garre del Olmo. Trabajo Fin de Grado, Grado en Diseño y Desarrollo de Videojuegos. Universidad Rey Juan Carlos, 2019

[28] Motor de sistemas de utilidad para la extensión de una api de comportamiento de personajes en videojuegos. Autor: Rafael Tomé Ruiz. Tutor: Carlos Garre del Olmo. Trabajo Fin de Grado, Grado en Diseño y Desarrollo de Videojuegos. Universidad Rey Juan Carlos, 2020

[29] desarrollo de una herramienta gráfica para la implementación de comportamientos de personajes de videojuegos. Autor: Diego Sagredo de Miguel. Tutor: Carlos Garre del Olmo. Trabajo Fin de Grado, Grado en Diseño y Desarrollo de Videojuegos. Universidad Rey Juan Carlos, 2020

[30] C. Escoffier (2017, Jun). 5 Things to Know About Reactive Programming. [Online]. Disponible en: https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming

[31] Epfl (2008, Jun). Finite state machine with epsilon transitions. [Online]. Disponible en: https://lara.epfl.ch/w/finite_state_machine_with_epsilon_transitions

[32] Kalihari, R. (2017). Comparison Between Mealy and Moore Using Automated Machine. *Journal of Science & Engineering Education (ISSN 2455-5061)*, *2*, 49-55.

[33] Refactoring Guru. Composite Pattern. [Online] Disponible en: https://refactoring.guru/es/design-patterns/composite

[34] Microsoft. Prueba unitaria de C# con MsTest y .NET. [Online] Disponible en: https://learn.microsoft.com/es-es/dotnet/core/testing/unit-testing-with-mstest

[35] Github. Github Actions. [Online]. Disponible en: https://github.com/features/actions

[36] E. A. Colino (2009, Ene). Conceptos y aplicaciones de la lógica borrosa. [Online]. Disponible en: https://www.tecnicaindustrial.es/conceptos-y-aplicaciones-de-la-logica-borrosa/

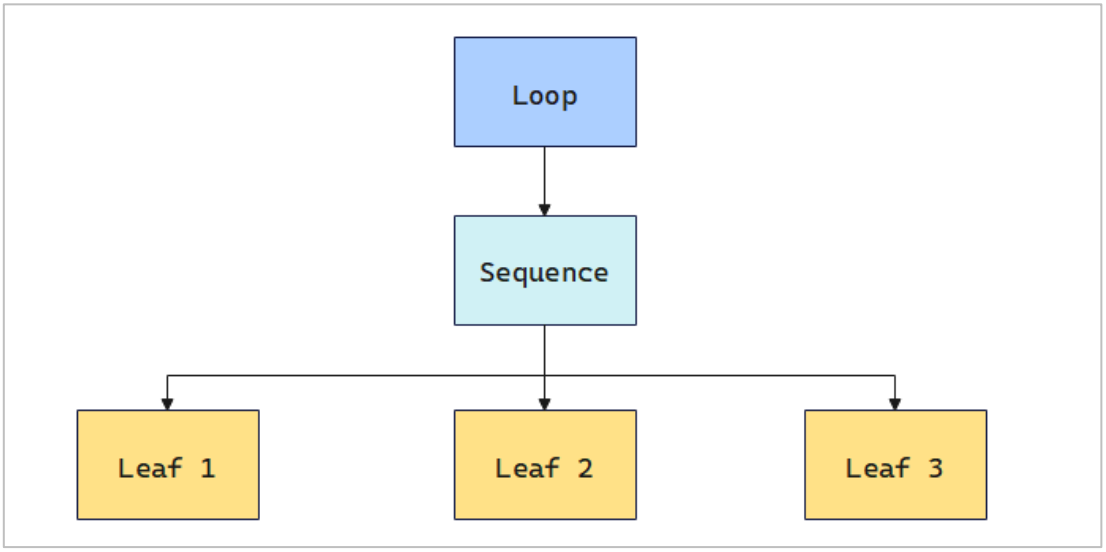
[37] Reyneri, L.M. (1997). An Introduction to Fuzzy State Automata. 1240. 273-283. 10.1007/BFb0032485.

7

Anexos

7.1 Anexo 1: Guía 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 función 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 `FunctionalAction` 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 seq = 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 sel = 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>(childNode);
```

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.

Extensión: Nodo condicional reactivo

Los nodos condicionales reactivos sirven para interrumpir la ejecución de su nodo hijo cuando una percepción devuelve *false* y reanudarla cuando devuelve *true*. Para crear un nodo reactivo, se usa el método para crear decoradores:

```
BTNode childNode = …;
Perception p = …;
ReactiveConditionNode loop = bt.CreateDecorator<ReactiveConditionNode>(childNode);
loop.Perception = p;
```

La ejecución del nodo hijo no comenzará hasta que la condición de la percepción se cumpla. Si ya se ha iniciado y la condición deja de cumplirse, la ejecución del nodo se parará hasta que vuelva a cumplirse.

Extensión: Nodo compuesto paralelo

Los nodos paralelos son un tipo de nodos compuestos que ejecutan todos sus hijos a la vez en lugar de uno a uno. Para crear un nodo paralelo, se usa el método para crear compuestos:

```
BTNode child1 = …;
BTNode child2 = …;
…
ParallelCompositeNode p = bt.CreateComposite<ParallelCompositeNode>(false,child1,child2, …);
```

Por defecto la ejecución terminará cuando todos los hijos hayan terminado y se devolverá el resultado del último en terminar, pero puede cambiarse para que termine cuando cualquiera de los hijos devuelva *Success*, *Failure* o cualquiera de los dos.

```
p.FinishOnSuccess = true; // Termina cuando un hijo devuelva Success
p.FinishOnFailure = true; // Termina cuando un hijo devuelva Failure
```

Extensión: Nodos de selección de rama

Los nodos de selección de rama son otro tipo de nodos compuestos que ejecutan sólo uno de sus hijos en lugar de todos. Existen varios tipos según la forma de seleccionar el hijo que ejecutan.

Nodo de selección aleatoria

Este nodo elige uno de los hijos aleatoriamente al principio de la ejecución

```
RandomBranchNode node = bt.CreateComposite<RandomBranchNode>(false, nodo1, nodo2, ...);
```

Nodo de selección por probabilidad

Este nodo elige uno de sus hijos en base a una probabilidad asignada a cada uno de ellos.

```
ProbabilityBranchNode node = bt.CreateComposite<ProbabilityBranchNode>(false, nodo1, nodo2, ...);
node.probabilities = new List<float>(){...};
```

Nodo de selección por función

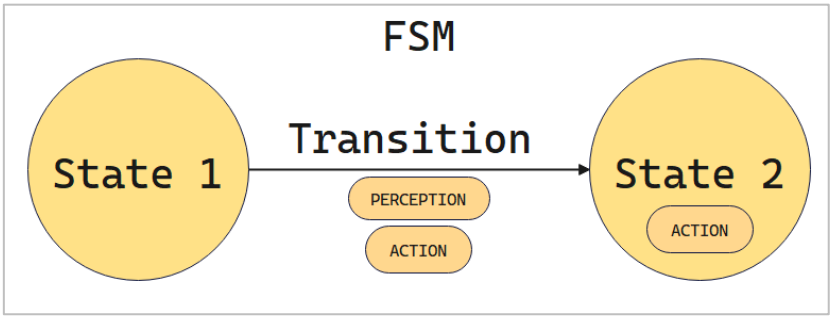
Este nodo permite especificar un método que devuelva el índice del hijo deseado.

```
Func<int> function = ...;
FunctionBranchNode node = bt.CreateComposite<FunctionBranchNode>(false, nodo1, nodo2, ...);
node.nodeIndexFunction = function;
```

7.2 Anexo 2: Guía de máquinas de estados

Crear y ejecutar una máquina de estados

Esta guía muestra los pasos a seguir para crear la siguiente máquina de estados.



Antes de crear la máquina de estados hay que importar los espacios de nombres `BehaviourAPI.Core` y `BehaviourAPI.StateMachines`.

Para crear una máquina de estados se crea un objeto de la clase `FSM`:

```
FSM fsm = new FSM ();
```

En las máquinas de estados primero se crean los estados y después las transiciones. Después de crear todos los elementos se especifica cual es el estado inicial. Si no, será el primero en haber sido creado.

```
fsm.SetEntryState(startNode);
```

Por último, para ejecutar la máquina de estados se debe lanzar el método Start al comienzo, y después ejecutar el método Update en cada iteración.

```
FSM fsm = new FSM (); // Al principio de la ejecución se crea el grafo (p.e. Awake en Unity)
fsm.Start();  // En el primer frame se inicia la ejecución del grafo (p.e. Start en Unity)
fsm.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.*

Estados

Los estados son los elementos principales de la máquina de estados. Para crear un estado se usa el método `CreateState` y se pasa opcionalmente una acción.

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 estado (o directamente si no tiene acción). Si el estado tiene una acción asignada, su valor `Status` se actualizará con el de la acción, mientras que si no la tiene se mantendrá en `Running`.

En el ejemplo de la guía se crean dos estados, el segundo de ellos con una acción asignada.

```
State state1 = fsm.CreateState();
State state2 = fsm.CreateState(action);
```

Transiciones

Las transiciones permiten a la máquina de estados cambiar su estado actual.

Transiciones entre estados

Para crear una transición entre dos estados se usa el método `CreateTransition`, pasando como argumentos el estado origen y el estado destino. Opcionalmente se pasan también una percepción y una acción.

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 siempre 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);
```

Si la percepción no se especifica, la transición siempre devolverá true al comprobarse. La acción solo se ejecutará en el frame en el que se lance la transición.

En el ejemplo inicial se crean una transición del estado “State1” al estado “State2” con una percepción y una acción asignadas.

```
StateTransition transition = fsm.CreateTransition(state1, state2, perception, action);
```

Especificar cuando se comprueba una transición

Por defecto las transiciones se comprueban siempre que el estado origen sea el estado actual de la FSM, pero en algunos casos puede ser necesario que la transición solo se active cuando el estado origen haya terminado de ejecutarse. Para esto se añade un parámetro de tipo `StatusFlags` a los métodos que crean transiciones (de cualquier tipo), de forma que la transición solo se comprobará cuando el valor status del estado origen coincida con este parámetro. A continuación, se muestran algunos casos de uso de esta funcionalidad:

```
// La transición se lanzará directamente cuando la acción de s1 haya terminado:
StateTransition t = fsm.CreateTransition(s1, s2, statusFlags: StatusFlags.Finished);
```

```
// La transición se lanzará directamente cuando la acción de state1 haya terminado (con éxito):
StateTransition t = fsm.CreateTransition(s1, s2, statusFlags: StatusFlags.Success);

// La transición se lanzará directamente cuando la acción de state1 haya terminado (con fracaso):
StateTransition t = fsm.CreateTransition(s1, s2, statusFlags: StatusFlags.Failure);

// La transición sólo se comprobará cuando la acción de state1 haya terminado y se lanzará cuando se cumpla la condición de la percepción asignada.
StateTransition t = fsm.CreateTransition(s1, s2, perception, statusFlags: StatusFlags.Finished);

// La transición no se comprobará nunca (útil para hacer transiciones que solo se activen externamente con percepciones push)
StateTransition t = fsm.CreateTransition(s1, s2, statusFlags: StatusFlags.None);
```

Activar transiciones de forma externa

Para activar una transición desde fuera de la FSM se debe crear un objeto de la clase `PushPerception` y pasarle como argumento la transición o transiciones que se quieren activar.

```
PushPerception push = new PushPerception(transition1, transition2, ...);
```

Una vez creada, se puede usar el método Fire para lanzar las transiciones desde cualquier parte del código, evitando las comprobaciones en cada iteración que supondría una percepción normal.

```
push.Fire();
```

Para que una transición pueda ser activada, su estado origen debe ser el estado actual de la `FSM`.

Crear subgrafos con máquinas de estados

Crear un subgrafo dentro de una FSM

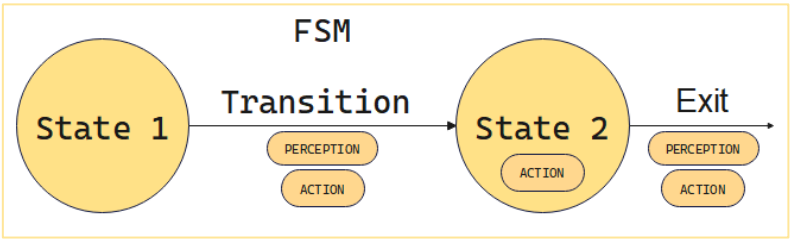
En las FSM se pueden crear subgrafos dentro de los estados usando acciones de tipo `SubsystemAction`:

```
FSM fsm = new FSM ();
BehaviourGraph subgraph = ...;
SubsystemAction action = new SubsystemAction(subgraph);
State state = fsm.CreateState(action);
```

NOTA: *Aunque también es posible, no es recomendable pasar acciones con subgrafos a transiciones, ya que estas sólo ejecutan su acción en el instante en el que son activadas.*

Salir de una submáquina de estados

Si la `FSM` es el subgrafo, para salir hay que lanzar una transición de salida.

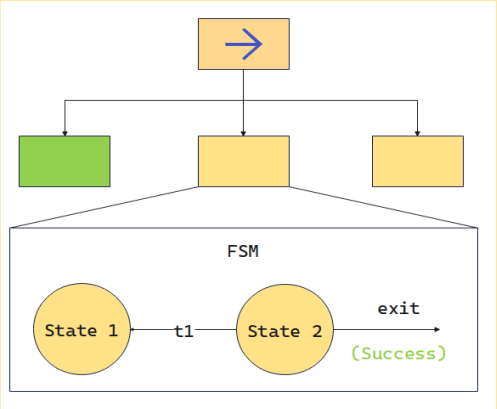


Las transiciones de salida permiten terminar la ejecución de la máquina de estados. Para crear una transición de salida se usa el método `CreateExitTransition` y se pasa como argumentos el nodo origen y el valor `Status` con el que terminará la ejecución de la `FSM` cuando la transición se active.

El valor de salida debe ser `Status.Success` o `Status.Failure`. Al igual que las transiciones entre estados, se puede pasar de forma opcional una percepción y una acción.

```
ExitTransition exit = fsm.CreateExitTransition(state2, Status.Success, perception, action);
```

El nodo que contiene la acción de submáquina actualizará su valor status al valor con el que la submáquina haya terminado su ejecución permitiendo al grafo padre continuar su ejecución.



En el ejemplo, cuando se lance la transición de salida la FSM terminará su ejecución con éxito, por lo que el nodo que la contiene también cambiará a éxito y la secuencia pasará al siguiente nodo.

Extensión: Máquinas de estados de pila

Las máquinas de estados de pila permiten almacenar en una estructura de datos de tipo pila los estados que se van recorriendo.

Para crear una fsm de pila se añade el espacio de nombres `BehaviourAPI.StateMachines.StackFSMs` y se crea un objeto de la clase `StackFSM`.

```
StackFSM stackFSM = new StackFSM();
```

Esta clase tiene los mismos métodos que una FSM normal, pero incluye dos tipos de transiciones más:

Transiciones push

Las transiciones push guardan el estado origen en la pila al lanzarse. Para crear una transición push se usa el método `CreatePushTransition`, pasando como argumentos el estado origen y el estado destino.

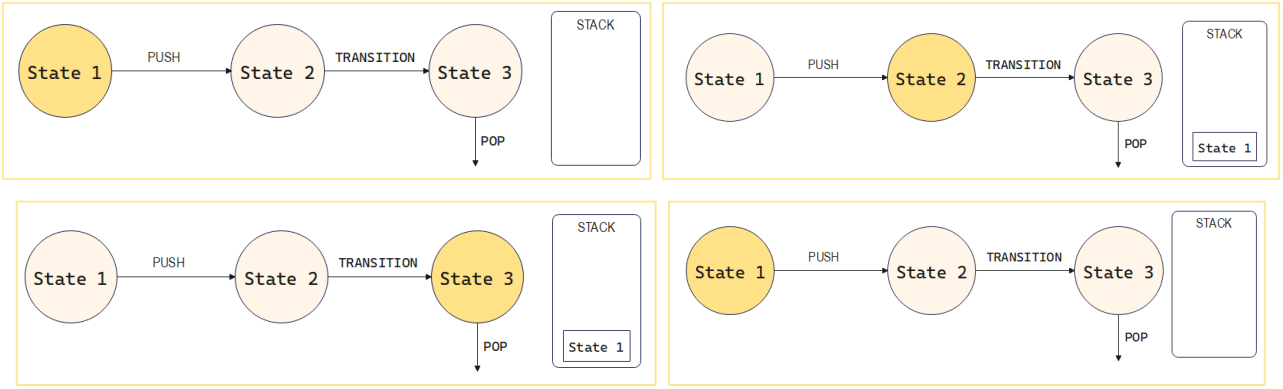
```
PushTransition push = stackFSM.CreatePushTransition(state1, state2, perception, action, ...);
```

Transiciones pop

Las transiciones pop permiten volver al último estado guardado en la pila. Para crear una transición pop se usa el método `CreatePopTransition`, pasando como argumento el estado origen.

```
PopTransition pop = stackFSM.CreatePopTransition(state1, perception, action, ...);
```

A continuación, se muestra cómo funcionan los nuevos tipos de transiciones. Cuando la FSM pasa del estado 1 al 2, pasa el estado 1 a la pila. Cuando se lanza la transición pop desde el estado 3, se saca el ultimo estado introducido a la pila y se transiciona a él.

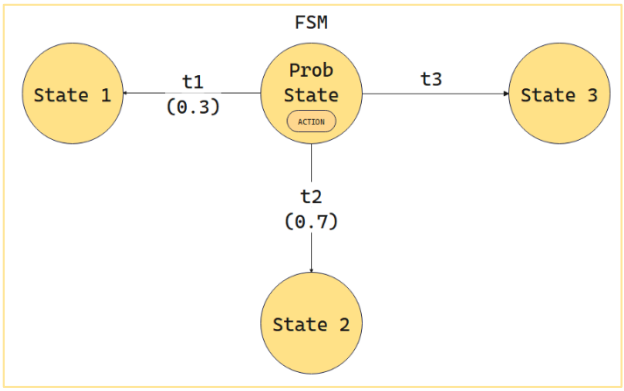


Extensión: Estados probabilísticos

Los estados probabilísticos permiten asignar probabilidades a sus transiciones. En cada iteración, el estado elegirá aleatoriamente una de las transiciones que tengan asignada una probabilidad y solo esa será comprobada.

Para crear un estado probabilístico se usa el método `CreateProbabilisticState` pasando como parámetro opcional la acción que ejecuta el estado.

Una vez creadas las transiciones del estado, se puede especificar la probabilidad de una transición con el método `SetProbability`.



Si la probabilidad especificada es 0 o menos, se considera que esa transición no tiene probabilidad asignada y se comprobará siempre, teniendo prioridad con respecto a las transiciones que si tienen.

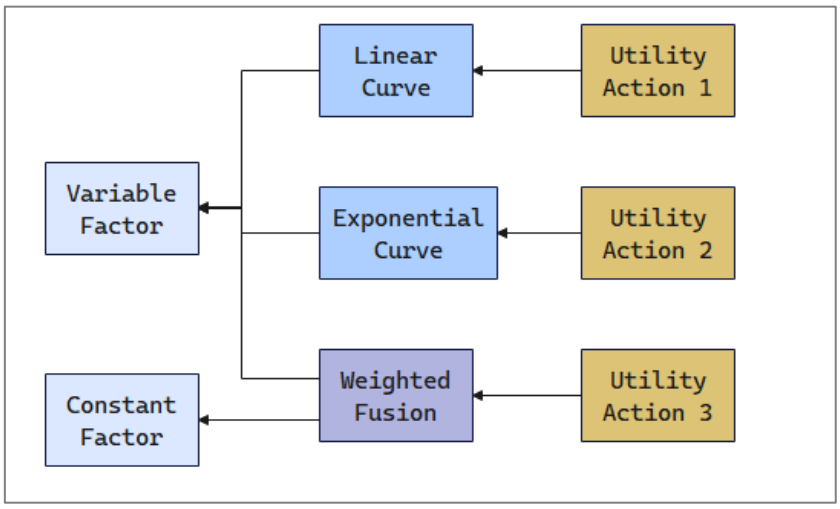
```
ProbabilisticState probState = fsm.CreateProbabilisticState(action);
StateTransition t1 = fsm.CreateTransition(probState, s1);
StateTransition t2 = fsm.CreateTransition(probState, s2);
StateTransition t3 = fsm.CreateTransition(probState, s3);

probState.Setprobability(t1, 0.3f);
probState.Setprobability(t2, 0.7f)
```

7.3 Anexo 3: Guía de sistemas de utilidad

Crear y ejecutar un sistema de utilidad

Esta guía muestra los pasos a seguir para crear el siguiente sistema de utilidad.



Para crear un sistema de utilidad se crea un objeto de la clase `UtilitySystem`. Se pasa como parámetro opcional la inercia del sistema, que es un multiplicador para la utilidad del elemento seleccionado para facilitar que este se mantenga estable. Por defecto es 1.3.

```
UtilitySystem us = new UtilitySystem();       // Inercia 1.3
UtilitySystem us = new UtilitySystem(1.5f);   // Inercia 1.5
```

En los sistemas de utilidad se crean primero los factores de izquierda a derecha y luego el resto de los nodos.

Para ejecutar el sistema de utilidad se debe lanzar el método Start al comienzo, y después ejecutar el método Update en cada iteración.

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

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

Factores hoja

Los factores hoja sirven para obtener un valor de utilidad del entorno y el resto de los factores calculan su utilidad en base a estos. Todos los tipos de factores limitan su valor al intervalo [0 - 1].

Factores hoja variables

Estos factores obtienen su utilidad de una variable o función y normalizan su valor en una escala 0-1.

Para crear un factor variable se usa el método `CreateVariable` del sistema de utilidad y se pasa como parámetros una función que devuelva un valor `float`, y dos valores `floats` que representan el valor mínimo y máximo que puede devolver la función. Por defecto estos valores son 0 y 1.

En el ejemplo inicial se crea un factor variable. La utilidad del factor cambiará cuando cambie el valor de "value".

```
float value = 5f;
VariableFactor v = us.CreateVariable(() => value, 0f, 10f); // Devolverá (value-0)/(10-0) = 0.5
```

Factores hoja constantes

Para crear un factor constante se usa el método `CreateConstant` y se pasa como parámetro el valor deseado. La utilidad del factor siempre será un valor entre 0 y 1, aunque el valor introducido sea mayor o menor.

En el ejemplo inicial se crea un factor constante. Este tipo de factores es útil para crear una "acción por defecto" que se seleccione si el resto no llegan al valor.

```
ConstantFactor v = us.CreateConstant(0.7f); // Devolverá siempre 0.7
```

Factores curva

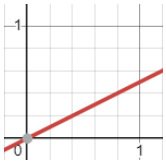
Los factores curva o curvas de utilidad sirven para modificar la utilidad de un factor hijo con una función. Para crearlos se usa el método create curve pasando el nodo factor hijo.

En el ejemplo inicial se crean dos factores curva de tipo lineal y exponencial respectivamente, ambos modificando la utilidad del factor hoja variable.

```
LinearCurveFactor lf = us.CreateCurve<LinearCurveFactor>(variableFactor);
ExponentialCurveFactor ef = us.CreateCurve<ExponentialCurveFactor>(variableFactor);
```

Factor curva lineal

Modifica la utilidad de su factor hijo con una función lineal. Se puede personalizar la pendiente (por defecto 1) y la ordenada en el origen (por defecto 0) de la función.

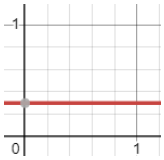


Slope = 0.5

YIntercept = 0

Slope = -0.5

YIntercept = 0.8



Slope = 0

YIntercept = 0.3

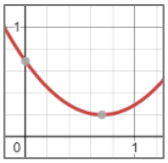
```
LinearCurveFactor lf = us.CreateCurve<LinearCurveFactor>(factor);
lf.Slope = -0.5f
lf.YIntercept = 0.8f;
```

Factor curva exponencial

Modifica la utilidad de su factor hijo con una función exponencial. Se puede personalizar el exponente (por defecto 1), el desplazamiento horizontal y el desplazamiento vertical de la función (por defecto 0).

Exponent = 0.5
DespX = -0.2
DespY = -0.3

Exponent = 1.8
DespX = 0.5
DespY = 0.4

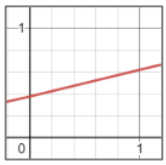


Exponent = 2
DespX = 0.7
DespY = 0.2

```
ExponentialCurveFactor ef = us.CreateCurve<ExponentialCurveFactor>(factor);
ef.Exponent = 0.5f
ef.DespX = -0.2f;
ef.DespY = -0.3f;
```

Factor sigmoide

Modifica la utilidad de su factor hijo con una función sigmoide. Se personaliza el coeficiente de crecimiento (por defecto 1) y el punto medio (por defecto 0.5).



GrownRate = 1
MidPoint = 0.5

GrownRate = 10
MidPoint = 0.2

GrownRate = -7
MidPoint = -0.3

```
SigmoidCurveFactor sf = us.CreateCurve<SigmoidCurveFactor>(factor);
sf.GrownRate = -7f
sf.MidPoint = -0.2f;
```

Factor curva a puntos

Crea la función con la que modifica la utilidad de su factor hijo mediante puntos que forman una función lineal por partes. Se puede especificar la lista de puntos que forman la función. Para evitar errores estos puntos deben pasarse ordenados en su coordenada x.

P1(0, 0.5)
P2(0.5, 0.8)
P3(1, 0.2)

```
PointedCurveFactor pf = us.CreateCurve<PointedCurveFactor>(factor)
pf.Points = new List<CurvePoint>(){
          new CurvePoint (0,0.4f), new CurvePoint (0.5f,0.8f), new Vector2(1,0.2f)
};
```

Factor curva personalizado

Modifica la utilidad de su factor hijo con una función especificada por el usuario. Se especifica la función con la variable `Function`.

```
CustomCurveFactor cf = us.CreateCurve<CustomCurveFactor>(factor);
cf.Function = (x) => x * x + 0.5f;
```

Factores fusión

Los factores fusión permiten combinar la utilidad de varios factores en uno solo. Para crear un factor fusión se usa el método `CreateFusion`, pasando como parámetro genérico el tipo de nodo fusión y como argumentos los factores hijos. Los factores hijos pueden pasarse por separado o como una lista directamente. En el ejemplo inicial se crea un factor fusión que calcula la suma ponderada de las utilidades de los dos factores hoja.

```
WeightedFusionFactor weightedFusion = us.CreateFusion<WeightedFusionFactor>(factor1, factor2);
weightedFusion.Weights = new float[]{0.7f, 0.3f};
```

MinFusionFactor

Devuelve la utilidad mínima de los factores hijos.

```
MinFusionFactor min = us.CreateFusion<MinFusionFactor>(factor1, factor2, ...);
```

MaxFusionFactor

Devuelve la utilidad máxima de los factores hijos.

```
MaxFusionFactor min = us.CreateFusion<MaxFusionFactor>(factor1, factor2, ...);
```

WeightedFusionFactor

Devuelve la media ponderada de la utilidad de los factores hijos. Se especifican los pesos en la variable `Weights`.

```
WeightedFusionFactor weighted = us.CreateFusion<WeightedFusionFactor>(factor1, factor2, ...);
weighted.Weights = new float[]{0.2f, 0.3f, …};
```

Acciones de utilidad

Las acciones de utilidad son nodos que el sistema de utilidad puede seleccionar para ejecutar una acción. Para crear una acción de utilidad se usa el método `CreateAction` y se pasa como parámetros el factor para calcular su utilidad y la acción que ejecutan.

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.

En el ejemplo de esta guía se crean tres acciones de utilidad, cada una asignada a un factor concreto.

```
UtilityAction utilityAction1 = us.CreateAction(linearCurve, action1);
UtilityAction utilityAction2 = us.CreateAction(exponentialCurve, action2);
UtilityAction utilityAction3 = us.CreateAction(weightedFusion, action3);
```

Crear subgrafos con sistemas de utilidad.

Crear un subgrafo dentro de un sistema de utilidad

Para ejecutar un subgrafo en un sistema de utilidad hay que crear una acción de tipo `SubsystemAction` pasando como parámetro el subgrafo y asignar la acción a un nodo de tipo `UtilityAction`.

```
UtilitySystem mainUS = new UtilitySystem();
BehaviourGraph subgraph = ...;
SubsystemAction action = new SubsystemAction(subgraph);
UtilityAction uAction = mainUS.CreateLeafNode(factor, action);
```

Salir de un subsistema de utilidad

Hay dos formas de salir de un sistema de utilidad.

- Al acabar de ejecutar un `UtilityAction`, es posible hacer que el sistema de utilidad termine su ejecución con el mismo valor con el que ha terminado la acción. Para ello se pasa el valor `true` al último argumento del método.

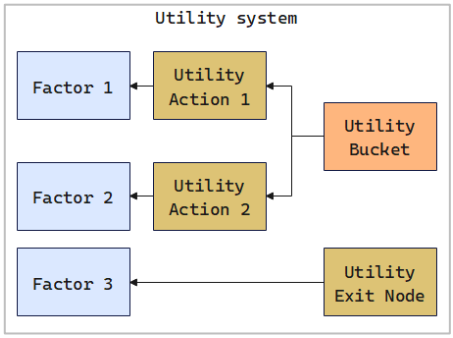
```
UtilityAction uAction = us.CreateAction(f, action, true);
```

- También se pueden crear nodos que terminen la ejecución del sistema de utilidad cuando son seleccionados. Para crear un nodo de salida se usa el método `CreateExitNode` y se pasa como parámetros el factor para calcular la utilidad y el valor `Status` de salida.

```
UtilityExitNode exitNode = us.CreateExitNode(factor, Status.Success);
```

Extensión: Buckets o grupos de elementos

Los grupos o buckets permiten agrupar acciones, nodos de salida u otros buckets. Para agrupar elementos primero se debe crear el bucket con el método `CreateBucket`. A este método se pasan dos argumentos, la inercia y el umbral de utilidad.



La inercia funciona igual que en los sistemas de utilidad y el umbral determina la utilidad que debe tener una acción del grupo para que este tenga prioridad y se ejecute, aunque otros elementos posteriores de fuera del grupo tengan más utilidad.

```
UtilityBucket bucket = us.CreateBucket(1.3f, 0.3f);
```

Para crear un elemento dentro de un bucket se debe añadir como parámetro el bucket en el método que crea el elemento.

```
UtilityBucket bucket = us.CreateBucket(1.3f, 0.3f);

UtilityAction groupedAction = us.CreateAction(factor, action, false, bucket);
UtilityExitNode groupedExitNode = us.CreateExitNode(factor, Status.Success, bucket);
UtilityBucket subBucket = us.CreateBucket(1.3f, 0.3f, bucket);
```

***NOTA**: El sistema de utilidad escogerá un elemento de aquellos que no tengan grupo asignado. El resto de los elementos solo podrán ser escogidos dentro de su propio grupo.* En el ejemplo anterior, el sistema de utilidad escoge entre "*UtilityBucket*" y "*UtilityExitNode*", y si escoge el primero, este escogerá a su vez entre las dos acciones.

Extensión: Optimizar cálculo de utilidad

Es posible optimizar el cálculo de utilidad en los casos es los que sabemos cuándo el valor va a cambiar. Para ello se asigna el valor *false* a la variable *PullingEnabled* para hacer que la utilidad no se recalcule en cada frame.

```
float v = 1f;
UtilitySystem us = new UtilitySystem();
Action action = …
Factor leafFactor = us.CreateLeaf(() => v);
UtilityAction uAction = us.CreateAction(leafFactor, action1);
uAction.PullingEnabled = false;
```

Después solo tenemos que llamar al método UpdateUtility pasando como parámetro el valor *true* cuando queramos que la utilidad se actualice. En el ejemplo anterior, esto sería cada vez que cambia el valor de v.

```
uAction.UpdateUtility(true);
```