



TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DO SOFTWARE



A.D.A.P.T. : Una herramienta de planificación de acciones orientada a objetivos para inteligencia artificial en Unity

Estudiante: Yago Mira Urdampilleta

Dirección: Enrique Fernández Blanco

Julián Alfonso Dorado De La Calle

A Coruña, septiembre de 2022.

Tatakael.

Agradecimientos

Primeramente agradecerles a mis tutores Enrique y Julián, por su esfuerzo, tiempo y dedicación, ya que sin ellos este proyecto no hubiera sido posible.

Gracias a Barry, por darme un futuro por el que luchar y hacerme feliz cada día de mi vida.

Gracias a mi hermana, por hacer de segunda madre y aguantarme en mis peores momentos.

Gracias a mi madre, por luchar a mi lado y no dejar que nunca me rindiera.

Gracias a Ana, porque sin ella no estaría donde estoy a día de hoy.

Gracias a Jorge y al resto de mis amigos y compañeros, por su ánimo y apoyo.

Y por último gracias a mi padre, por estar.

Resumen

Actualmente la [Inteligencia Artificial \(IA\)](#) es una tecnología que se encuentra en constante desarrollo y mejora. Utilizada en muchos ámbitos como puede ser el sector de los videojuegos, donde a través del análisis del entorno y la capacidad de adaptarse al mismo se pueden contemplar conductas similares a las de los humanos. Este proyecto se basa en esa idea, es decir, en desarrollar una herramienta que permita de manera sencilla poder crear un comportamiento inteligente y suplir aquellas carencias que el motor de videojuegos, [Unity](#) en este caso, no ofrece de una manera simple al usuario.

Para alcanzar ese objetivo se integró uno de los sistemas más novedosos en cuanto a la creación de [IA](#) en videojuegos, conocido como [Goal-Oriented Action Planning \(GOAP\)](#). Este permite dotar a los [Personajes No Jugadores \(PNJ\)](#) de un conjunto de acciones, de las cuales se ejecutarán unas u otras dependiendo de la situación, todo ello con la finalidad de conseguir alcanzar una meta prefijada.

Abstract

Currently [Artificial Intelligence \(AI\)](#) is a technology that is constantly developing and improving. Used in many areas such as the video game sector, where through the analysis of the environment and the ability to adapt to it, behaviors similar to those of humans can be seen. This project is based on that idea, on developing a tool that allows in a simple way to be able to create an intelligent behavior and fill those deficiencies that the [Unity](#) video game engine does not offer in a simple way to the user.

To achieve this goal, one of the most innovative systems was integrated in terms of the creation of [AI](#) in video games, known as [Goal-Oriented Action Planning \(GOAP\)](#). This will allow endow [Non-Playable Characters \(NPC\)](#) to be provided with a set of actions, and depends on the situation one or other will be executed, with the target of achieve a predetermined goal.

Palabras clave:

- GOAP
- Unity3D
- Herramienta IA
- C#
- A*

Keywords:

- GOAP
- Unity3D
- AI Tool
- C#
- A*

Índice general

1	Introducción	1
1.1	Motivación	1
1.2	Objetivos	2
1.3	Estructura de la memoria	3
2	Fundamentos Tecnológicos	4
2.1	Estado de la cuestión	4
2.1.1	Goal-Oriented Action Planning	5
2.2	Estudio de Mercado	7
2.2.1	SWOT	7
2.2.2	Estudio de mercado en la store	7
2.3	Tecnologías utilizadas	11
3	Metodología y Planificación	14
3.1	Metodología de desarrollo	14
3.1.1	Adaptación de la metodología al proyecto	14
3.1.2	Herramientas utilizadas para la aplicación de la metodología	15
3.2	Planificación	16
3.2.1	Recursos	17
3.3	Sprints	18
3.4	Planificación Inicial	19
3.4.1	Estimación de costes	21
4	Desarrollo	24
4.1	Sprint 1	24
4.1.1	Análisis	24
4.1.2	Diseño	26
4.1.3	Implementación	26

ÍNDICE GENERAL

4.1.4	Pruebas	26
4.2	Sprint 2	26
4.2.1	Análisis	27
4.2.2	Diseño	27
4.2.3	Implementación	29
4.2.4	Pruebas	30
4.3	Sprint 3	31
4.3.1	Análisis	31
4.3.2	Diseño	33
4.3.3	Implementación	34
4.3.4	Pruebas	35
4.4	Sprint 4	37
4.4.1	Análisis	37
4.4.2	Diseño	38
4.4.3	Implementación	43
4.4.4	Pruebas	44
4.5	Sprint 5	45
4.5.1	Análisis	45
4.5.2	Diseño	46
4.5.3	Implementación	49
4.5.4	Pruebas	49
4.6	Sprint 6	49
4.6.1	Análisis	50
4.6.2	Diseño	50
4.6.3	Implementación	54
4.6.4	Planificador de acciones	54
4.6.5	Pruebas	57
4.7	Sprint 7	58
4.7.1	Análisis	58
4.7.2	Diseño	59
4.7.3	Implementación	60
4.7.4	Pruebas	61
4.8	Sprint 8	61
4.8.1	Análisis	61
4.8.2	Diseño	62
4.8.3	Implementación	68
4.8.4	Pruebas	71

ÍNDICE GENERAL

4.9	Sprint 9	72
4.9.1	Análisis	72
4.9.2	Diseño	73
4.9.3	Implementación	74
4.9.4	Pruebas	75
5	Conclusiones	76
6	Trabajo Futuro	78
6.1	Mejoras Futuras	78
A	Tablas de Requisitos e Historias de Usuario	80
A.1	Tabla de requisitos, versión inicial	80
A.2	Tabla de requisitos, correcciones del Sprint 2	83
A.3	Tabla de requisitos, versión final	86
A.4	Historias de usuario, versión inicial	91
A.5	Historias de usuario, versión final	93
B	Mockups Iniciales	96
C	Mockups Finales	107
D	Solución desarrollada	122
E	Manual de usuario	138
Lista de acrónimos		161
Glosario		162
Bibliografía		165

Índice de figuras

2.1	Videojuegos que usan otros sistemas de IA	4
2.2	Videojuegos que implementan GOAP	5
2.3	Ánálisis SWOT	7
2.4	Quest Machine	8
2.5	Intense Shooter AI	8
2.6	SGOAP	9
2.7	Herramienta "Goal Oriented Action Planning"	9
2.8	ReGoap	10
2.9	Logo de A.D.A.P.T	11
2.10	Unity Editor	13
3.1	Ejemplo en Trello del Product Backlog	16
3.2	Diagrama de Gantt: Planificación Inicial (<i>Sin Fases</i>)	20
3.3	Diagrama de Gantt: Planificación Inicial detallada	21
4.1	Actor "Usuario"	26
4.2	Actores que podrán interactuar con el sistema	27
4.3	Diagrama de clases del Sprint 2	28
4.4	Ejemplo de la consola de Unity	30
4.5	Diagrama de clases del Sprint 3	33
4.6	Pruebas correspondientes a la HU-2	36
4.7	Pruebas correspondientes a la HU-6	36
4.8	Pruebas correspondientes a la HU-9	37
4.9	Diagrama de clases del Sprint 4	38
4.10	Diagrama de clases del Sprint 4, patrón template	42
4.11	Diagrama de clases del Sprint 4, patrón estrategia	42
4.12	Diagrama de clases del Sprint 4, interfaz gráfica	43
4.13	Interfaz gráfica de las acciones, Sprint 4	44

ÍNDICE DE FIGURAS

4.14	Pruebas correspondientes a la HU-2 del Sprint 4	45
4.15	Diagrama de clases del Sprint 5	46
4.16	Diagrama de clases del Sprint 5, interfaz gráfica	48
4.17	Diagrama de clases del Sprint 6	51
4.18	Un diagrama que compara la aplicación de GOAP de A* con pathfinding [1]	55
4.19	Funcionamiento del planner GOAP	56
4.20	Diagrama de clases del Sprint 7	59
4.21	Diagrama de clases del Sprint 8	63
4.22	Diagrama de clases del Sprint 8, interfaz gráfica	67
4.23	Añadir componente animator	70
4.24	Animator usado para las animaciones	70
4.25	Agente con modelado 3D ejecutando una animación	71
4.26	Diagrama de clases del Sprint 9	73
4.27	Diagrama de clases del Sprint 9, interfaz gráfica	74
4.28	DIRECTORIOS de géneros de videojuegos	75
5.1	Diagrama de Gantt: Planificación Final	77
B.1	Agente con acciones visualizado en el Unity Inspector	97
B.2	Agente con acciones y valores cubiertos visualizado en el Unity Inspector	98
B.3	Agente con acciones y valores modificados visualizado en el Unity Inspector	99
B.4	Agente con acciones y nuevos valores visualizado en el Unity Inspector	100
B.5	Menú de la herramienta	101
B.6	Menú de la herramienta con objeto seleccionado en Unity	102
B.7	Menú de la herramienta convirtiéndo a un objeto en agente	103
B.8	Árbol de acciones	104
B.9	Menú de acciones predefinidas	105
B.10	Menú de acciones predefinidas, añadir acción	106
C.1	Agente con acciones visualizado en el Unity Inspector	108
C.2	Agente con acciones y valores cubiertos visualizado en el Unity Inspector	109
C.3	Agente con acciones y valores modificados visualizado en el Unity Inspector	110
C.4	Menú de la herramienta	111
C.5	Menú de la herramienta con objeto seleccionado en Unity	112
C.6	Menú de la herramienta creando nuevo agente	113
C.7	Menú de la herramienta creando nueva acción	114
C.8	Menú de la herramienta con mensajes de error	115
C.9	Árbol de acciones	116

ÍNDICE DE FIGURAS

C.10	Árbol de acciones en tiempo de ejecución	117
C.11	Menú de acciones predefinidas	118
C.12	Menú de acciones predefinidas, añadir acción	119
C.13	Menú de acciones predefinidas: mensaje de error de acciones	120
C.14	Menú de acciones predefinidas: mensaje de error de agente	121
D.1	Toolbar de Unity, donde pueden seleccionarse los menús de la herramienta	122
D.2	Acción correspondiente al sistema GOAP	122
D.3	Recursos de una acción desplegados	123
D.4	Visualización de agente en inspector	124
D.5	Despliegue de propiedades de agente (Goals y acciones)	125
D.6	Propiedades relacionadas con las animaciones de un agente	126
D.7	Menú de A.D.A.P.T., ventana principal	127
D.8	Menú de A.D.A.P.T. cuando se selecciona un objeto	128
D.9	Añadir una nueva acción a través del menú	129
D.10	Mensaje de error en caso de que una acción ya exista	130
D.11	Mensaje de error en caso de que el objeto actual no sea un agente	131
D.12	Añadir un nuevo agente a través del menú	131
D.13	Mensaje de error en caso de que un agente ya exista	132
D.14	Mensaje de error en caso de que el objeto ya sea un agente	133
D.15	Árbol de acciones de un agente, sin ejecutar	133
D.16	Árbol de acciones en tiempo de ejecución	134
D.17	Menú de acciones predefinidas	135
D.18	Añadir acción predefinida a un agente	136
D.19	Mensaje de error en caso de que la acción ya esté agregada	136
D.20	Mensaje de error en caso de que el objeto no sea un agente	137

Índice de tablas

2.1	Tabla comparativa de los motores Unreal y Unity	12
3.1	Tabla de recursos humanos	17
3.2	Tabla con los salarios de los recursos (€/h)	22
3.3	Tabla con los costes finales de los recursos (en €)	22
3.4	Tabla con los costes materiales	22
3.5	Tabla con el coste total del proyecto	23
4.1	Tabla de requisitos funcionales del Sprint 3	31
A.1	Tabla de requisitos funcionales	80
A.2	Tabla de requisitos funcionales	83
A.3	Tabla de requisitos funcionales	86
A.4	Tabla con las historias de usuario del Sprint 1	91
A.5	Tabla con las historias de usuario del Sprint 2	93

Capítulo 1

Introducción

En este capítulo se expondrán las razones por las cuales se decidió realizar este proyecto así como también los objetivos del mismo.

1.1 Motivación

El sector de los videojuegos es en la actualidad una de las maneras de entretenimiento que más factura a nivel global, superando a otros sectores en su conjunto como son la música y el cine. Respectivamente en el año 2021 se facturaron más de 175.000 millones de dólares [2], a diferencia de la música facturando 26.000 millones [3] y el cine con una recaudación de 21.400 millones de dólares [4].

El desarrollo de videojuegos es un proceso largo el cual comprende múltiples fases diferentes entre sí, ya sea por ser procesos relacionados con un apartado artístico como, por ejemplo, el desarrollo narrativo, el diseño gráfico, la música y todas aquellas ramas artísticas que se abarcan. Así como también, contemplar aquellas fases de programación, animación, etc. Por ello es necesario en la mayoría de las empresas de videojuegos, un equipo multidisciplinario que permita llevar a cabo las diversas tareas del desarrollo para llegar a alcanzar una funcionalidad completa.

Se considera que la funcionalidad se logra a través de la implementación y corrección continuada de los distintos mecanismos. Estos permiten que un videojuego se diferencie del resto y permitan hacerlo atractivo, es lo que se conoce, de manera genérica como mecánicas. Estas mecánicas condicionan que todo aquel componente integrado dentro del entorno de un videojuego tenga que interactuar con el mismo. Aquí es donde entra en juego la inteligencia artificial, la cual permite que mediante la inclusión de los conocidos agentes o **Personajes No Jugadores (PNJ)** se genere una ilusión de inteligencia en el comportamiento comportándose de una manera similar a los humanos y permitiendo así, que el jugador en cuestión tenga una sensación de inmersión a la hora de relacionarse con los diferentes componentes del

videojuego.

Aunque existen diversos sistemas que permitan implementar este tipo de comportamiento en los componentes, bien es cierto que la mayoría de los mismos se tornan un tanto complicados al momento de permitir que un usuario común, es decir, con poco o nulo conocimiento sobre la inclusión de los [PNJ](#), haga uso de los mismos. Este es uno de los problemas que pretende solucionar el presente proyecto, mediante la implementación de uno de estos novedosos sistemas, más concretamente [Goal-Oriented Action Planning \(GOAP\)](#). Haciendo uso de su capacidad para dotar a un agente de un conjunto de acciones ejecutables y que, dependiendo de una situación u otra se ejecuten las respectivas acciones, todo ello con la finalidad de poder alcanzar una meta fijada previamente.

Por tanto, el objetivo principal que se pretende alcanzar con este proyecto es el desarrollo de una herramienta orientada a los motores gráficos de videojuegos, que facilite el proceso de creación y modificación de los agentes a través de una interfaz gráfica, pudiendo facilitar la inclusión de los mismos en las mecánicas actuales de un videojuego. Esto podrá realizarse de diversas maneras: ya sea a través la posibilidad de definir acciones o metas específicas que permitan a los agentes realizar los diversos comportamientos, o usar los prefijados por la propia herramienta y acelerar el proceso de desarrollo.

1.2 Objetivos

En base al objetivo expuesto al final del apartado [1.1](#) donde se concretaba que el objetivo principal del proyecto es utilizar un motor gráfico de videojuegos, más concretamente [Unity](#), para el desarrollo de una herramienta basada en el sistema [GOAP](#) para cumplir con los siguientes subobjetivos:

- Implementación de alternativas para algoritmos de planificación.
- Diseño e implementación de opciones por defecto que se puedan usar.
 - Facilidades a la hora de modificar las funcionalidades de los agentes.
- Creación de un interfaz que permita de una manera visual y sencilla la creación, modificación y eliminación de agentes creados con la herramienta, además de poder añadir/modificar/eliminar acciones y objetivos propios de estos mismos.
- Presentar a través de una interfaz gráfica y de manera similar a como lo haría un árbol de nodos, el conjunto de acciones que poseen cada uno de los agentes y resaltar de entre las mismas, aquellas que permiten llegar a cumplir con un objetivo concreto indicando el camino correspondiente.

1.3 Estructura de la memoria

Con la finalidad de facilitar la comprensión y lectura de la memoria, se detallan a continuación junto a una breve explicación, los diferentes apartados que podrán encontrarse en la misma:

1. **Introducción:** apartado que comprende aspectos básicos de la memoria como la motivación y los objetivos a alcanzar con el presente proyecto.
2. **Fundamentos tecnológicos:** en este apartado se contemplan herramientas similares. Realizando un breve estudio de mercado se contemplarán las ventajas y desventajas de las mismas respecto a la herramienta a desarrollar así como también las tecnologías utilizadas en el desarrollo.
3. **Metodología y planificación:** se detallará la metodología usada en el proyecto, así como también sus posibles modificaciones para su correcta adecuación en cuanto al tamaño del equipo de desarrollo. Además, se concretarán también otros aspectos como la planificación y seguimiento.
4. **Desarrollo:** se realizarán las distintas fases de la metodología para cada uno de los **sprints**: el análisis de requisitos (funcionales y no funcionales) y creación de diagramas, la fase de diseño contemplando los respectivos diagramas de clases y representaciones gráficas necesarias. Posteriormente la fase de implementación donde se detallará el trabajo realizado, además de la integración de los algoritmos y finalmente se contemplará la fase de pruebas donde se detallarán las pruebas realizadas para el correcto funcionamiento del sistema.
5. **Conclusiones:** se tendrá en cuenta la duración real del desarrollo y los posibles inconvenientes, así como también el aprendizaje y resultado obtenido del mismo.
6. **Trabajo futuro:** posibles aspectos futuros, importantes a tener en cuenta sobre el proyecto para posibles mejoras.

Capítulo 2

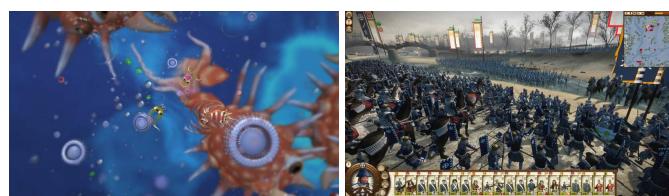
Fundamentos Tecnológicos

2.1 Estado de la cuestión

En este capítulo se expondrán los diferentes motivos que conllevaron al desarrollo del proyecto, obtenidos a través de la observación del panorama actual en cuanto a la industria del videojuego y la utilización del sistema [GOAP](#); así como también el breve estudio de mercado que se ha llevado a cabo, con el fin de poder reforzar esa primera idea del proyecto y ver las posibilidades de la herramienta.

Actualmente existen diversos sistemas además del utilizado en este proyecto para la implementación de [IA](#) en videojuegos. Cada uno con sus respectivas mejoras o inconvenientes respecto al resto, pero que tratan de ser usados en las situaciones correspondientes para obtener el máximo beneficio de sus capacidades.

Por una parte existen los [Behavior Tree](#), utilizados para definir comportamientos más complejos y que requieran cierto realismo en su toma de acciones. Un ejemplo claro del uso de este sistema puede verse en el videojuego Spore [5] (figura 2.1a), donde el comportamiento de los [NPC](#) evoluciona a la par que los propios individuos. Entre otros sistemas, también se contempla la utilización de [Redes Neuronales](#), más específicamente en videojuegos que requieren de cierta lógica y aprendizaje continuo en la toma de decisiones. Este último sería el caso de los juegos de estrategia, entre los cuales destaca el conocido videojuego Total War (figura 2.1b), el cual usa este sistema para el manejo de unidades de manera individual [6].



(a) Spore

(b) Total War: Shogun II

Figura 2.1: Videojuegos que usan otros sistemas de [IA](#)

Por otra parte, **GOAP** es un sistema que permite ser integrado en cualquier género y esto puede verse en como se ha usado en diferentes tipos de videojuegos a lo largo de los años. Ejemplos claros de su uso pueden verse en el primer videojuego que integró y dio origen a este sistema: F.E.A.R. (figura 2.2a). Incluso hasta casos más recientes como Middle Earth: Shadow of Mordor (figura 2.2b) o Tomb Raider (2013). En estos últimos ha sido confirmado por los propios desarrolladores [7] el margen de mejora de este sistema y versatilidad del mismo.



(a) F.E.A.R.



(b) Middle-Earth: Shadow of Mordor

Figura 2.2: Videojuegos que implementan GOAP

2.1.1 Goal-Oriented Action Planning

Esta sección está destinada a clarificar en qué consiste el sistema **GOAP**, comprendiendo desde su origen hasta el funcionamiento del mismo.

Siendo así, **Goal-Oriented Action Planning (GOAP)** fue creado alrededor del año 2003 por un programador de inteligencia artificial perteneciente a la empresa Monolith Productions, Jeff Orkin [8]. Más concretamente, no fue hasta más adelante en el año 2005 que se implementó de manera oficial en un videojuego completo (véase el caso del videojuego F.E.A.R. mencionado anteriormente). Previamente a una explicación en más profundidad sobre el sistema, cabe clarificar los siguientes conceptos para poder entender el funcionamiento que **GOAP** ofrece así como también, los componentes que lo conforman:

- **Agente**: equivalente a un **NPC**. Básicamente aquella entidad que no será controlada por el jugador de manera directa, usualmente suele verse como un enemigo o un elemento aliado dependiendo del tipo de videojuego y su implementación. Además, es aquel que se encargará de formular un plan con la finalidad de obtener una secuencia de acciones que le permitan alcanzar un objetivo concreto.
- **Estado** [9]: se conoce como una representación del mundo, en este caso, de un videojuego. Esto se realiza a través de una estructura de datos, esta estructura permitirá conocer en un momento concreto las propiedades de los **NPC** y los valores de los mismos: posiciones, armas, cantidad de munición, ...

- **Goal:** condición que un agente quiere alcanzar, más concretamente, un estado del mundo al cual llegar con la consecuente ejecución de acciones.
- **Plan:** nombre el cual recibe la secuencia de acciones que permita alcanzar un goal concreto. Un plan que satisface un objetivo se refiere a aquel conjunto de acciones que debe tomar un agente para poder llegar hasta ese estado concreto.
- **Acción:** consiste en llevar una actividad a cabo. Una acción está relacionada con dos componentes que permitirán saber o no, si esta se puede llevar a cabo y las consecuencias de hacerlo:
 - **Precondiciones:** son aquellos estados que han de cumplirse en el momento de saber si una acción se va a poder ejecutar o no.
 - **Efectos:** consecuencias de ejecutar una acción y las correspondientes modificaciones de los estados después de la finalización de la misma.

GOAP surgió ante la necesidad de implementar capacidades de planificación en aquellos videojuegos que basaban el comportamiento de sus **PNJ** en un sistema orientado exclusivamente a la toma de decisiones respecto a objetivos [10]. Esto es, basar la próxima acción a ejecutar en base únicamente a la prioridad que tenga el objetivo; esto tiene un alto coste, ya que se está constantemente reevaluando los objetivos con la finalidad de seleccionar el más prioritario. El sistema implementado en el proyecto en cuestión ofrece multitud de ventajas respecto a este último, ya que **GOAP** permite adaptar sus acciones a la situación actual pudiendo reutilizar comportamientos, así como también facilitar una mayor variedad y complejidad de los mismos, convirtiéndolo en un sistema escalable y fácilmente mantenible.

Implementación de GOAP

Aunque el algoritmo a utilizar para la implementación del sistema se detallará en apartados posteriores, se hará una mención a como usualmente se implementa de cara a poder entender mejor el funcionamiento del mismo (véase 4.6.3).

Por tanto, cabe destacar un concepto importante a la hora de implementar **GOAP**, este es el de *planner*. Ya que hay que tener presente que el sistema se implementa utilizando las ventajas de un **árbol de nodos**, donde se considera un nodo como una acción y las aristas que las unen como las respectivas precondiciones/efectos. Para poder navegar por ese árbol con la finalidad de encontrar un plan que satisfaga el llegar a un *goal* deseado, se necesita un proceso de navegación o **pathfinding** implementado por el *planner* mencionado con anterioridad. Por tanto el *planner* lo que nos permitirá es, que mediante el algoritmo de búsqueda elegido para navegar por el árbol, se pueda obtener un conjunto de acciones o nodos que satisfagan el *goal*.

2.2 Estudio de Mercado

A continuación, con el fin de poder concretar las ventajas que puede ofrecer el presente proyecto frente a la competencia actual así como también minimizar las posibles desventajas, se ha realizado un estudio de mercado que permitirá además, poder dar validez a la viabilidad del mismo.

2.2.1 SWOT

Previamente al estudio de mercado se ha realizado un análisis **Strengths-Weaknesses-Opportunities-Threats (SWOT)**, el cual justificará la elección del entorno en el cual se desarrollará la herramienta, así como también reforzar la viabilidad del proyecto antes de comenzar con su desarrollo, puede verse el análisis en la siguiente imagen:

Strengths	Weaknesses
Producto completamente gratuito. Se cubre una necesidad del mercado. Pocos o nulos casos reales de uso de herramientas competidoras	Poca experiencia de los desarrolladores. Límite de tiempo de desarrollo. Dificultad para cubrir todos los géneros posibles de videojuegos.
Opportunities	Threats
Pocos competidores Mercado en crecimiento Visibilidad de la herramienta Demostrar uso herramienta con un caso real (videojuego)	Desarrolladores competidores experimentados

Figura 2.3: Análisis SWOT

Con ello, se puede llegar a la conclusión de tomar en consideración la elección del motor **Unity** para el desarrollo de la herramienta (cuyas ventajas y características pueden verse reflejadas con mayor claridad en el apartado posterior: [2.3](#)). Ya que debido a la inexperiencia de los desarrolladores, así como también la posibilidad de un mercado creciente, junto a los precios competitivos de la herramienta presente, se convierte en la mejor opción para poder destacar frente a sus competidores.

2.2.2 Estudio de mercado en la store

El estudio de mercado se ha realizado en base a la propia tienda virtual de **Unity** conocida como **Unity Asset Store** [\[11\]](#), donde todos los usuarios de este motor pueden obtener los **assets** para sus videojuegos de una manera sencilla mediante una transacción económica o en algunos casos, de manera gratuita. Al tener esta tienda como modelo de referencia el estudio de mercado se simplifica, ya que se puede obtener una visión global de los precios, características, opiniones de los usuarios y todo tipo de información que podrá beneficiar de cara a la

planificación y mejora de la herramienta del presente proyecto. Siendo así, las herramientas relacionadas con **GOAP** presentes en la tienda:

- **Quest Machine** [12]: a pesar de aparecer como unos de los resultados de la búsqueda relacionada con el sistema GOAP, es realmente una herramienta orientada al desarrollo de **quest** de juegos estilo **RPG**. Donde indican al propio jugador que tareas debe realizar para cumplir unos objetivos propios, pero no trata en particular ningún comportamiento relacionado con **PNJ**.



Figura 2.4: Quest Machine

- **Intense Shooter AI** [13]: a pesar de implementar GOAP, es un caso muy específico y orientado al género **shooter** por lo que no podría extrapolarse de una manera sencilla a **IA**'s de otro tipo de videojuegos.



Figura 2.5: Intense Shooter AI

- **SGOAP** [14]: en comparación al resto de herramientas encontradas es la mejor debido a sus características y por tanto, la clara referencia a tener en cuenta para realizar el proyecto. Se trata de una herramienta que está bien optimizada, tiene documentación clara aunque no extensa, algunos ejemplos funcionales y permite implementar una **IA** relativamente sencilla. El gran inconveniente es que es una herramienta de pago a diferencia de otras como ReGoap (31€ aproximadamente).



Figura 2.6: SGOAP

- **Goal Oriented Action Planning** [15]: es una herramienta que contiene ejemplos funcionales en 2D. Aunque presenta diversos problemas, entre ellos: documentación escasa o inexistente, según algunos usuarios tiene problemas de rendimiento y las implementaciones de los agentes son exclusivamente a través de código. Este último punto dificultaría el desarrollo a usuarios con pocos conocimientos.



Figura 2.7: Herramienta "Goal Oriented Action Planning"

- **ReGOAP** [16]: a pesar de ser una herramienta presente en la tienda, no ofrece mucha información sobre sí misma a parte de un enlace externo a un repositorio de [GitHub](#). En el cual encontraremos documentación detallada pero orientada de manera exclusiva al código y no en una herramienta simple de usar a través de una interfaz como en el caso de SGOAP. Además, no se incluyen ejemplos claros de prototipos funcionales. Una gran ventaja de la misma es que es una herramienta completamente gratuita y que se encuentra en constante desarrollo y mejora.

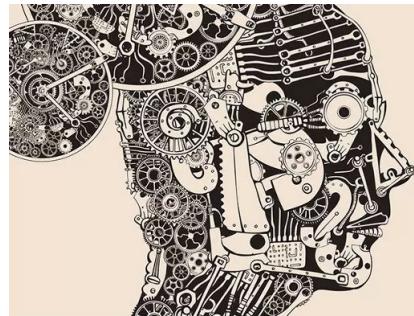


Figura 2.8: ReGoap

Después de poder obtener una ligera idea de como se presenta actualmente el mercado respecto a las herramientas que usan el sistema [Goal-Oriented Action Planning \(GOAP\)](#), se puede llegar a la conclusión de que la herramienta del presente proyecto basará su desarrollo en los siguientes aspectos:

- **Precio:** se ofrecerá una herramienta completamente gratuita, por lo que cualquier usuario podrá probarla sin ningún compromiso.
- **Interfaz:** a través de la propia interfaz del motor y con las modificaciones correspondientes se permitirá que el usuario defina, de una manera sencilla, el comportamiento de sus agentes.
- **Generalización:** en contraste a otras herramientas (como el caso de *Intense Shooter AI*) donde se centran en un único género de videojuego, se pretende que la herramienta sea lo suficientemente genérica como para que el usuario pueda desarrollar agentes de cualquier tipo.
- **Poco mercado:** al ser un mercado todavía en expansión se ofrecen multitud de oportunidades a la par que escasos competidores, por lo que la [Unity Asset Store](#) proporcionará la visibilidad requerida para que la herramienta sea utilizada por multitud de usuarios.
- **Animación 3D:** son pocos los casos en los que una herramienta de estas características muestre ejemplos funcionales con animaciones 3D en agentes, a ello que sea una clara funcionalidad a implementar en la herramienta.
- **Marca:** para darle un símbolo identificativo a la herramienta y que permita identificarla del resto, se usará un nombre y logo que facilite al usuario el reconocer la herramienta a simple vista. De aquí surge el nombre de la herramienta: **A.D.A.P.T.**



Figura 2.9: Logo de A.D.A.P.T

2.3 Tecnologías utilizadas

En este apartado se pretenden exponer aquellas tecnologías usadas en el desarrollo de la herramienta y en este caso particular solo cabe resaltar una única tecnología: el motor de videojuegos [Unity](#) [17].

Un motor de videojuegos consiste en una serie de rutinas o conjunto de librerías que permiten la creación y funcionamiento de los mismos. Esto es posible debido a las diferentes partes que contiene un motor de estas características: por una parte incluye un motor gráfico que permite renderizar aquellos modelos 2D y 3D, un motor físico que permita realizar simulaciones realistas de la física, también se incluyen módulos que permiten la implementación de otros apartados técnicos como sonidos, música, animación, redes, ... o el caso particular y que atañe a este proyecto, de módulos relacionados con la inteligencia artificial.

Uno de entre los muchos motores de videojuegos que se encuentran actualmente, además de ser el más usado en la industria debido al ser utilizado por las grandes compañías para el desarrollo de videojuegos, entre los que destacan títulos de alto presupuesto como es el caso de: Dragon Ball Fighter Z, Shenmue III, Kingdom Hearts III, ... es [Unreal Engine](#) [18]. El cual, a la hora de compararlo con el motor utilizado para el desarrollo del presente proyecto, presenta una serie de desventajas que podrán verse en la siguiente tabla comparativa, que permitirá aclarar las características de ambos motores con el fin de dar una justificación añadida a la elección de [Unity](#) para el desarrollo:

Unreal Engine	Unity
Limitaciones para crear videojuegos en 2D	Facilidad para creación de juegos 2D, 3D
Lenguaje C++	Lenguaje C#
Mayor exigencia de recursos y espacio	Ocupa poco espacio en disco
Documentación y comunidad limitada	Documentación extensa

Tabla 2.1: Tabla comparativa de los motores Unreal y Unity

Retomando el caso concreto de [Unity](#), es un motor que utiliza el lenguaje de programación [C#](#) (*aunque en versiones previas del motor se daba libertad al uso de [JavaScript](#)*) y que cuenta con varios módulos que se han utilizado a la hora de desarrollar la herramienta, entre los que destacamos:

- **AIModule:** módulo vital para la realización del proyecto así como también para la implementación de cualquier tipo de [IA](#) en Unity. Provee clases de utilidad que facilitan la inclusión de propiedades relacionadas con el [pathfinding](#), entre las que se destaca: [NavMesh](#).
- **AnimationModule:** módulo relacionado con las animaciones y que implementa el sistema de animación de Unity. Permite que los modelos tanto 2D/3D puedan efectuar sus animaciones a través de una interfaz conocida como [Animator](#) o lo que viene siendo una [Máquina de estados](#).
- **CoreModule:** es el módulo que implementa las clases más básicas y requeridas por el motor para funcionar. Dentro de este módulo encontramos la clase **MonoBehaviour**, que es aquella de la cual todo código de Unity deriva y permite añadir a los objetos los diversos componentes.
- **UnityEditor:** aunque realmente no es un módulo sino una API permite implementar las clases orientadas a modificar la interfaz del [Unity Inspector](#) y de las ventanas adyacentes al mismo, es decir, el [Unity Editor](#). Este módulo será usado para poder implementar toda la parte gráfica de la herramienta y editar las propiedades respectivas del inspector, con el fin de facilitar al usuario su uso (figura 2.10).

Todos estos módulos son utilizados en las correspondientes clases pero aplicados a los propios objetos de Unity, conocidos como **GameObject** [19] a los cuales se le añadirán los distintos componentes que permitirán modelar los distintos agentes de la herramienta.

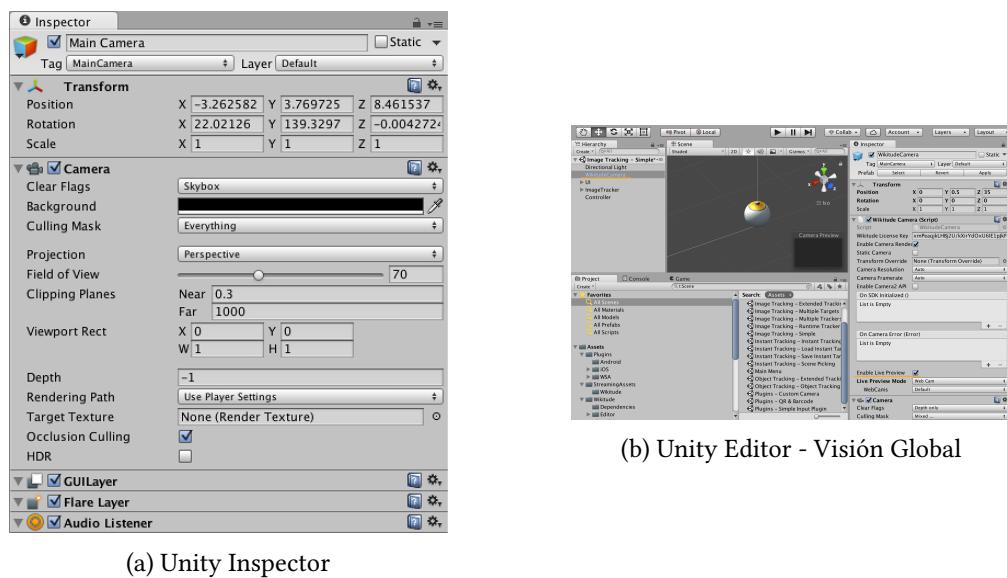


Figura 2.10: Unity Editor

Capítulo 3

Metodología y Planificación

3.1 Metodología de desarrollo

En base a las necesidades del proyecto a desarrollar donde se presenta una constante evolución de la herramienta y de sus capacidades, se ha decidido usar una metodología ágil. Más concretamente la metodología **Scrum**. Esta metodología es de gran utilidad para situaciones como la que se presenta en este desarrollo, en la que no se está entregando al cliente lo que necesita en todo momento, sino que el usuario final no podrá usar la herramienta hasta la finalización completa de la misma.

Eso sumado al hecho de que Scrum es una metodología flexible que permite la corrección e innovación de los requisitos, hace que sea una elección adecuada para llevar a cabo el proyecto. A diferencia de otras metodologías como pueden ser las tradicionales, Scrum nos presenta un conjunto de buenas prácticas también denominadas *prácticas ágiles*. Estas últimas proporcionan una solución óptima a diversos problemas software relacionados con diversos aspectos: como la velocidad de producción, entregas tardías y diversas problemáticas comunes entre otros muchos aspectos.

3.1.1 Adaptación de la metodología al proyecto

A pesar de no contar con un equipo de desarrollo compuesto por varios miembros se ha decidido adaptar esta metodología de una manera individual, prescindiendo de algunas prácticas de la misma como podrían ser el *pair programming* (donde se necesitaría de al menos un par de desarrolladores programando juntos) o modificándolas como en el caso de las *daily scrum* o reuniones diarias que se ha optado por simplemente realizar las revisiones al final de cada *sprint*, o lo que se conoce como **Sprint Review**. Además, para poder tomar ventaja del uso de esta metodología se hace uso de determinadas prácticas que podrán facilitar el avance del proyecto, entre las que destacan:

- **Product Backlog:** el uso de un espacio de trabajo permitirá a los desarrolladores entender de un primer vistazo las tareas a desarrollar, la prioridad de las mismas así como también las mejoras o correcciones que puedan sufrir (ver figura 3.1),
- **Control de versiones:** con el objetivo de poder obtener diferentes versiones del proyecto para finalmente y en caso de que fuera necesario, realizar las pertinentes correcciones. Se hace uso de un sistema que permita mantener el control sobre los constantes *sprints* del desarrollo.
- **Refactorización:** en la medida de lo posible se tratará de optimizar y simplificar el funcionamiento interno de la herramienta así como su propio código fuente, para con ello facilitar el entendimiento del mismo además de intentar obtener un mejor rendimiento.

Por tanto, una vez comprendida la adaptación de la metodología al proyecto a desarrollar cabe entender las fases en las que se divirán cada una de las iteraciones (también denominadas *Sprints*), que serán las detalladas a continuación:

- **1.-Análisis:** en base al análisis de mercado realizado en el apartado 2.2 con el objetivo de obtener una situación del entorno y entender lo que realmente se quiere aportar con el desarrollo de la herramienta, se detallan los requisitos que debe cumplir la misma así como también elaborar las diferentes **historias de usuario** expuestas en el *product backlog*.
- **2.-Diseño:** una vez se tengan las funcionalidades a desarrollar en el *sprint*, se confecionarán los diversos diagramas y representaciones gráficas que permitan entender la estructura del proyecto y los componentes necesarios que conformarán el funcionamiento deseado.
- **3.-Implementación:** se abordará el desarrollo de las funcionalidades propuestas en el *sprint*, realizando el trabajo de codificación correspondiente.
- **4.-Pruebas:** se comprueba el correcto funcionamiento de las funcionalidades y como estas se ajustan a los requisitos definidos en etapas anteriores.

Cabe resaltar que se realizará el correspondiente trabajo de **documentación** sobre todo en relación a la etapa de implementación, con el objetivo de cumplir con los requisitos no funcionales que se verán en apartados posteriores.

3.1.2 Herramientas utilizadas para la aplicación de la metodología

- **Github** [20]: herramienta que permite alojar código de manera remota, utilizada para el control de versiones.

- **Trello** [21]: software de administración de proyectos, herramienta visual que permite administrar y gestionar el flujo de trabajo. En este caso concreto permite gestionar el *product backlog* y las historias de usuario a desarrollar en cada uno de los *sprints*.
- **Draw.IO** [22]: software para la elaboración de diagramas de cualquier tipo. En este caso concreto ha sido utilizado para elaborar los diagramas de clases a través del lenguaje de modelado UML y también para realizar los diagramas Gantt a través de una extensión que proporciona la herramienta para ese mismo propósito [23].
- **Balsamiq** [24]: permite la realización de diagramas y esbozos de la interfaz de la herramienta. Utilizado para la realización de los mockups.
- **Visual Studio** [25]: IDE utilizado de manera conjunta con C#. Permite realizar las tareas de codificación así como también facilitar el control de versiones a través de su integración con GitHub.

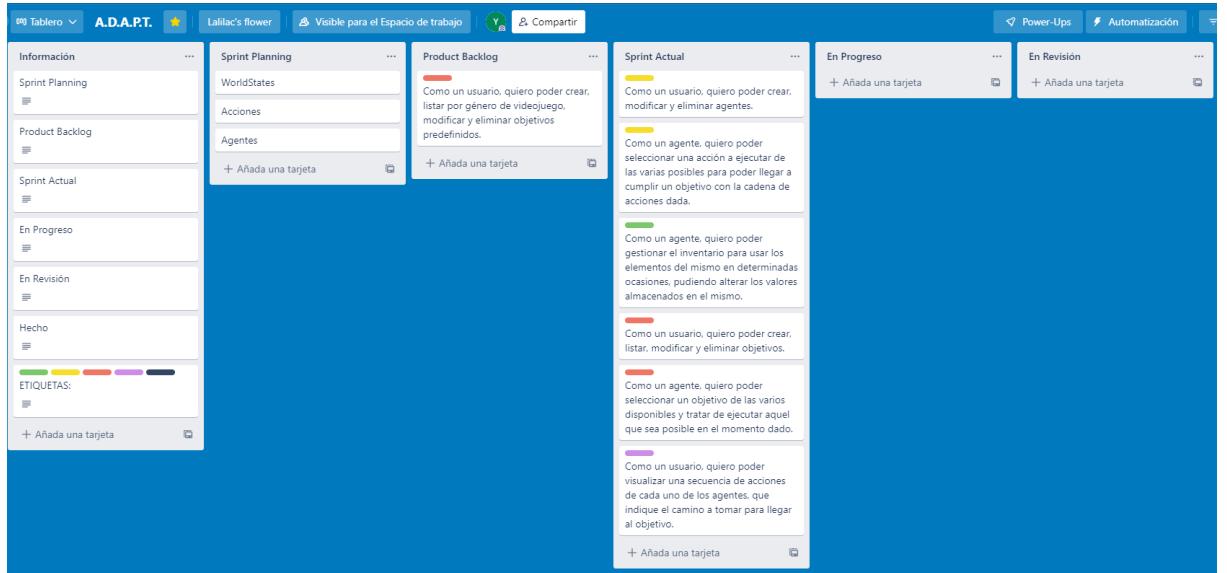


Figura 3.1: Ejemplo en Trello del Product Backlog

3.2 Planificación

En este apartado se detallarán los recursos necesarios para el desarrollo del proyecto, así como también mencionar los correspondientes *sprints* que lo componen aportando junto a los mismos y de manera resumida, su desarrollo. Además de realizarse una planificación inicial del proyecto con su respectiva duración estimada, para a continuación y con el objetivo de

reforzar la visión sobre la viabilidad del proyecto, relizar las estimaciones de costes correspondientes.

3.2.1 Recursos

Recursos Humanos

Este proyecto ha sido desarrollado por el siguiente conjunto de recursos humanos que se han de tener en cuenta a la hora de realizar la estimación de costes para poder obtener una previsión más realista:

Puesto	Recursos
<i>Analista/Programador</i>	1 (Alumno)
<i>Jefe de proyecto software</i>	2 (Tutores)

Tabla 3.1: Tabla de recursos humanos

Se contempla que el alumno dedique una jornada laboral completa ($8h/d$) y por otra parte, se estimará que el tiempo dedicado de los jefes de proyecto/tutores sea el sumatorio aproximado de las horas invertidas en las reuniones realizadas al final de cada *sprint*, también conocidas como **sprint review**. Las tareas del alumno contemplan el desarrollo y codificación de la herramienta a la vez que la elaboración de los diseños y correcciones pertinentes. Los tutores se encargaron de las tareas de supervisión del proyecto así como las correcciones y observaciones de las fases del proyecto que fueran necesarias.

Recursos Técnicos

- **Recursos Hardware:** se componen del equipo usado por parte del alumno, en este caso se corresponde a un ordenador portátil con las siguientes características: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, GTX 1050 Ti, RAM de 8,00 GB y un sistema operativo de 64 bits.
- **Recursos Software:** el software utilizado para el desarrollo ha sido detallado previamente en los apartados 2.3 y 3.1.2 así como las utilidades que puedan darse al mismo. A mayores cabe mencionar el uso de un sistema operativo *Windows* así como también el funcionamiento de la herramienta bajo dicho sistema.

3.3 Sprints

A continuación se detallarán de una manera resumida, los *sprints* llevados a cabo en el desarrollo:

- **Sprint 1:** esta primera fase se centrará sobre todo en las etapas de análisis y diseño, donde se especificarán los requisitos y historias de usuarios que a posteriori, formarán el *product backlog*. Se realizarán los primeros bocetos de la herramienta además de pequeñas implementaciones que permitan al alumno entender el funcionamiento del motor para facilitar su uso en el resto de *sprints*.
- **Sprint 2:** comprenderá las posibles modificaciones de los requisitos, diagramas e historias de usuario, así como establecer la granularidad de las mismas con el fin de realizar un correcto reparto de la carga de trabajo. Se realizará también la primera implementación de los agentes y su conocimiento del entorno además de las acciones que estos mismos ejecutarán.
- **Sprint 3:** en caso de ser necesarias, se realizarán nuevamente las correspondientes correcciones en la fase de diseño. Se implementarán además, los recursos usados por los agentes y los distintos tipos que pueden llegar a tener, obteniendo así una base para realizar las correspondientes pruebas que permitan validar las historias tanto del *sprint 2* como el presente.
- **Sprint 4:** en esta etapa intentará adaptarse la implementación a los estándares de los patrones de diseño, con el fin de obtener un mayor rendimiento. Además de obtener una herramienta legible para el usuario final. Se realizarán también, las primeras implementaciones de la interfaz gráfica que permita visualizar de una manera sencilla los agentes y sus propiedades.
- **Sprint 5:** nuevamente al tener previsto este *sprint* como uno un poco más extenso, se pretende obtener una buena base de cara a la implementación, realizando las correcciones en la misma si las hubiera. Este *sprint* se centrará sobre todo en implementar aquellas partes de la interfaz relacionadas con los recursos de las acciones así como también, estas mismas.
- **Sprint 6:** se orientará a implementar el planificador de acciones, es decir: la implementación del algoritmo A^* para poder tener una serie de acciones que lleven a los agentes a un objetivo concreto.
- **Sprint 7:** se contempla la implementación de los estados o inventarios propios de los agentes para poder contrastar el cumplimiento de las precondiciones y con ello, llegar

a ejecutarlas. Así como también, las posibles correcciones del planificador de acciones en caso de que las hubiera.

- **Sprint 8:** en este *sprint* se agrupan diversas tareas diferentes, por una parte la implementación de las animaciones 3D para los agentes y el correcto funcionamiento de las mismas, así como también la correspondiente [Máquina de estados](#) que permita ejecutarlas. Además, se tendrán en cuenta los estados o inventario global, donde todos los agentes podrán acceder a un mismo inventario de recursos y usarlo debidamente. Por otra parte, se añadirá el árbol de acciones/nodos que permitirá a los usuarios visualizar la secuencia de acciones a tomar para llegar a un objetivo. Cabe destacar también, que se realizarán las correcciones de interfaz necesarias con el fin de facilitar el entendimiento de la herramienta. Finalmente se procederá a comenzar con el desarrollo de la memoria.
- **Sprint 9:** se realizarán las últimas correcciones e implementación de las acciones predefinidas, que mediante una interfaz gráfica, podrá darse la posibilidad al usuario de añadir una acción a un determinado agente de una manera sencilla, además de poder listarlas por género de videojuego. Se procederán a realizarse también, los diversos ejemplos de uso reales, que permitirán a los usuarios tener una idea del funcionamiento de la herramienta. También se creará el manual de usuario y se ultimarán los pasos finales de la memoria.

3.4 Planificación Inicial

La planificación inicial del presente proyecto ha contemplado un conjunto de **9 sprints** en total. Cada uno de los cuales realizará las correspondientes fases detalladas en el apartado [3.1.1](#) donde en algunos casos, alguna de las mismas podría ver reducida su duración o ser prácticamente inexistente a medida que el proyecto ha ido avanzando. Esto es debido al hecho de poder no ser necesarias y haberse dado el caso de ya haber completado todas las correcciones, diseños, diagramas, ... necesarios para el proyecto en *sprints* anteriores. Esto último podrá verse reflejado en apartados posteriores donde se obtendrá la duración real del proyecto, que servirá de retrospectiva para comparar con la planificación inicial.

En un primer momento se contempló que la duración de los *sprints* debía de ser de 2 semanas, esto permitiría abordar varias historias de usuario en un único *sprint* y en caso de necesitarse, poder realizar correcciones de historias de usuario incompletas o aquellas que su implementación ha ocasionado fallos. Aun así, se daba la posibilidad a contemplar un retraso de 1 semana extra en el *sprint*, es decir, un máximo de 3 semanas para aquel que pudiera tener una cantidad de trabajo mayor que el resto, o inclusive también, para prevenir algún tipo de imprevisto.

Al final de cada *sprint* se realizarían las *sprint review* con el objetivo de que el **scrum master**, aquel encargado de enfocar el desarrollo manteniendo los principios y prácticas de la metodología Scrum y donde en este caso tomara este rol el alumno. Además, formará parte del equipo de desarrollo y comunicará el feedback correspondiente a los **product owner**, es decir, los tutores/jefes de proyecto. Estas reuniones tienen la finalidad de demostrar que el incremento del proyecto a desarrollar se ha llevado con éxito, y en caso de no ser así, los *product owner* reorganizarán el *product backlog* para futuros *sprints*.

A continuación se muestra un diagrama de Gantt que permitirá visualizar de una manera gráfica la planificación inicial que se tenía del proyecto de una manera sencilla. Este diagrama muestra las semanas de los *sprints*, las cuales podrán alternarse entre 2-3 semanas en función del trabajo a desarrollar:

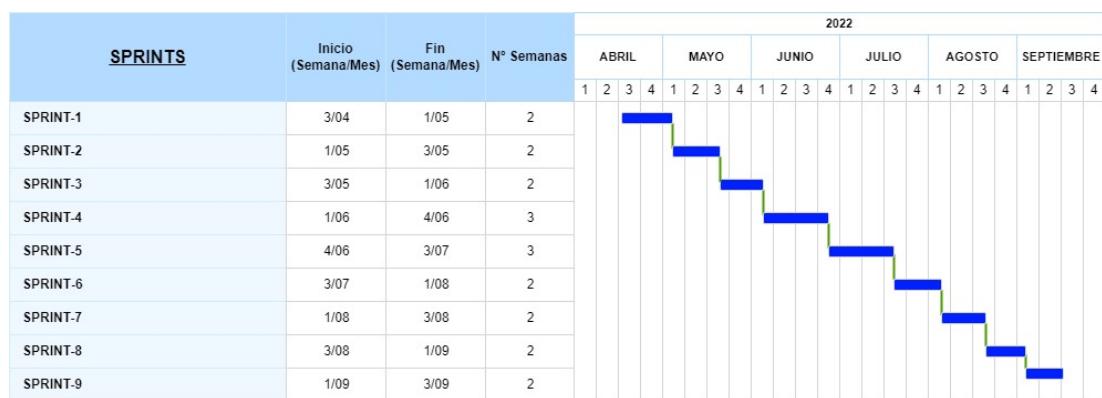


Figura 3.2: Diagrama de Gantt: Planificación Inicial (Sin Fases)

Como se puede ver en la figura 5.1 se presenta un cómputo total de aproximadamente 20 semanas para los 9 *sprints* del proyecto. En estas se contemplan dos posibles *sprints* de una extensión de 3 semanas, ambas situadas en la mitad del proyecto, para prever posibles correcciones e integraciones importantes de la herramienta.

Para poder visualizar con mayor claridad la implicación de cada *sprint*, se expondrá a continuación (figura 3.3) otro diagrama de Gantt, similar al anterior. Este nuevo diagrama contemplará las distintas fases a emplear por la metodología.

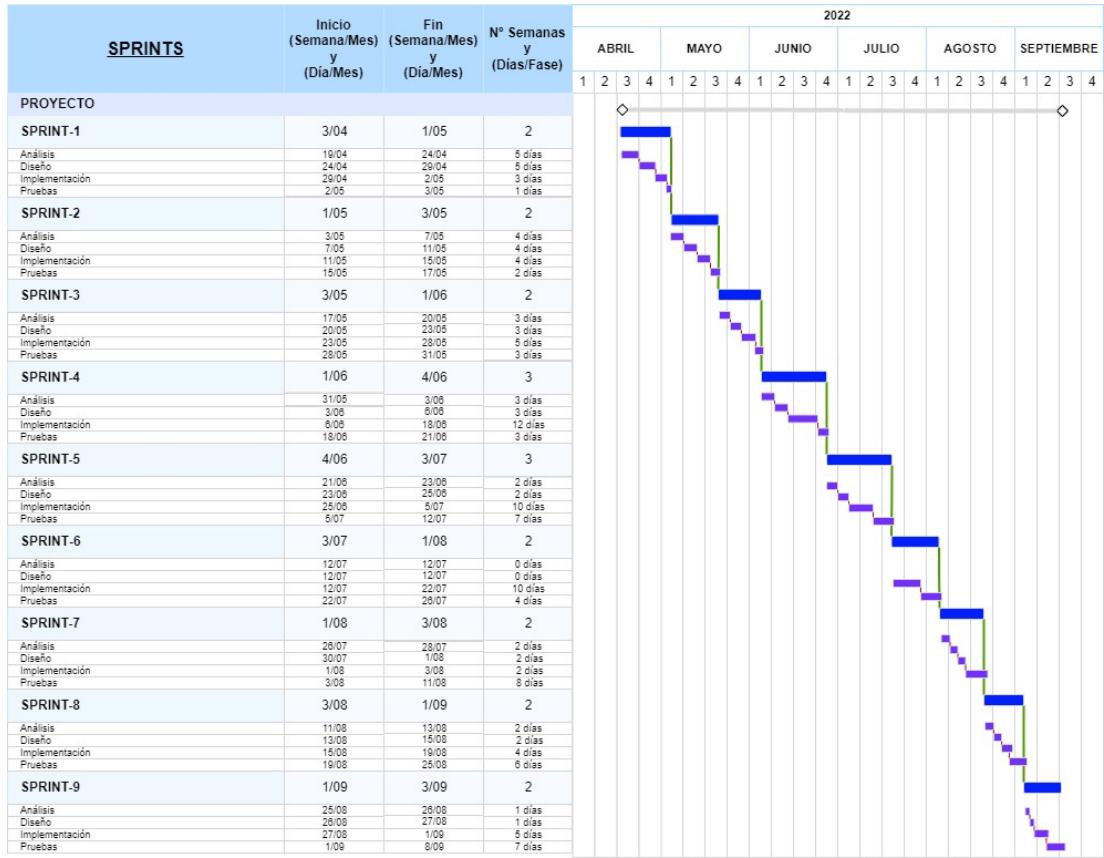


Figura 3.3: Diagrama de Gantt: Planificación Inicial detallada

Cabe resaltar que en ambos diagramas se contempla que la duración del último *sprint* se aproxime a la fecha de entrega del proyecto, con la finalidad de poder obtener el mayor tiempo posible para el desarrollo del mismo. Además, se contempla que en los dos últimos *sprints* se incluya otra parte importante para la finalización del trabajo: el desarrollo de la memoria y revisión de la misma.

3.4.1 Estimación de costes

En este apartado se realiza una estimación de los costes del proyecto, más concretamente del salario de los recursos. Se tendrán en cuenta las jornadas y el cómputo de horas comentado anteriormente en el apartado 3.2.1. Además se tendrá en cuenta la información recogida en la web [26]. Se tomará como base el salario medio de cada uno de los puestos de trabajo, para así poder de valorar el salario/hora correspondiente. Por tanto, se obtiene la siguiente tabla:

Puesto	€/h
Analista/Programador	15,90 €/h
Jefe de proyecto software	24,36 €/h

Tabla 3.2: Tabla con los salarios de los recursos (€/h)

Una vez obtenido el valor €/h para los respectivos puestos de trabajo, se procede a realizar el cómputo de horas en base a la figura 3.3 para el tiempo de desarrollo por parte del alumno. En este caso al realizar una jornada laboral completa de 8h/d sería realizar el cálculo en función del sumatorio de los días previstos en el diagrama de Gantt. Por otra parte, para los jefes de proyecto se tendrán en cuenta las 9 *sprint reviews*, estimando 1 hora para cada una de las mismas. Con esto en cuenta, se puede obtener la tabla con los costes finales de los recursos:

Puesto	Horas	Coste
Analista/Programador	1120	17.808€
Jefe de proyecto software	9	219,24€
TOTAL:	1129	18.027,24€

Tabla 3.3: Tabla con los costes finales de los recursos (en €)

A continuación, se detallaran los costes materiales, los cuales englobaran tanto el equipo hardware detallado anteriormente en el apartado 3.2.1, como los recursos software detallados también, en ese mismo apartado.

Hardware/Software	Coste
Ordenador personal	140€
Licencia "Personal" Unity	0€
Licencia "Free" Balsamiq	0€
Licencia "Community" Visual Studio	0€
TOTAL:	140€

Tabla 3.4: Tabla con los costes materiales

En cuanto al *ordenador personal* utilizado por el alumno para el desarrollo de la herramienta, su coste se ha calculado en función de la vida útil del mismo, para la cual se han asumido un total de 9600 horas. Por otra parte, se tendrán en cuenta el precio del portátil, el cual asciende a un total de 1200€, así como también el número de horas dedicadas por parte del alumno (1120 horas). El total correspondiente al coste aparecerá reflejado en la tabla anterior [3.4](#).

En conclusión, se pueden obtener los gastos totales siguientes, obtenidos del sumatorio de las tablas de los salarios con los costes finales de los recursos (véase [3.3](#)) y la tabla con los costes materiales (véase [3.4](#)):

Salarios	18.027,24€
Materiales	140€
TOTAL:	18.167,24€

Tabla 3.5: Tabla con el coste total del proyecto

Capítulo 4

Desarrollo

En este capítulo se contempla la elaboración y evolución de la propia herramienta a través del uso de la metodología designada. Podrán verse las diferentes implementaciones que aportan cada uno de los *sprints* al presente proyecto, así como las posibles correcciones que pudiera sufrir el mismo en su desarrollo.

4.1 Sprint 1

Este primer *sprint* del proyecto no consta de alguna implementación propiamente dicha, así como tampoco ningún período de pruebas. Esto es debido a que esta fase se centró en su debido momento, en establecer una correcta base para facilitar la posterior implementación de la herramienta. Esto es, formalizar de manera adecuada los requisitos que compondrán en un futuro, las funcionalidades del proyecto a desarrollar, así como también la creación de las correspondientes *historias de usuario* para que puedan ser usadas a posteriori en el *product backlog*.

Cabe destacar que la creación de las *historias de usuario* suele realizarse al inicio del proyecto ágil, es decir, al comienzo del desarrollo del mismo. Justificando así, el motivo de este *sprint*.

4.1.1 Análisis

En esta primera fase del *sprint* se realizó la especificación de requisitos (tanto funcionales como no funcionales) para a continuación, pasar a la creación de las *historias de usuario*.

Requisitos funcionales

Primeramente y con la finalidad de obtener una mejor comprensión de los requisitos y funcionalidades que se llevarán a cabo, se presentarán en la tabla visible en el **Apéndice A.1**

los **requisitos funcionales** junto a una breve descripción. Los cuales se han podido obtener en base a lo explicado en apartados anteriores (véase [1.2](#)).

Requisitos no funcionales

A continuación, se detallaron los **requisitos no funcionales** de la herramienta, los cuales permitirán identificar la operabilidad de la misma así como su posible comportamiento, siendo así:

- **Usabilidad:**
 - *Intuitiva y fácil de usar*: la interfaz gráfica será lo suficientemente sencilla como para que cualquier usuario con un mínimo de conocimientos pueda interactuar con ella.
 - La herramienta tendrá un *manual de usuario* que permita ayudar al usuario en el manejo de la misma, puede verse en el [Apéndice E](#).
- **Capacidades funcionales de la IA** (Requisito de Funcionalidad): la IA implementada con el algoritmo correspondiente, debe ser capaz de alcanzar sus objetivos de manera adecuada y de la forma más eficiente posible.
- **Adaptabilidad**: la herramienta debe poder funcionar en varias versiones de Unity diferentes (2019.x-2020.x).
- **Testeabilidad**: se podrá probar que la herramienta funciona mediante algún prototipo que demuestre las diferentes funcionalidades.
- **Soporte**: código bien documentado y explicado para un buen entendimiento y dar facilidades en cuanto a la ampliación de funcionalidad por parte del usuario.
- **Rendimiento**: se desea obtener el mejor rendimiento de la herramienta, optimizando y mejorando el algoritmo de [pathfinding](#) lo mejor posible

Historias de Usuario

En base a los requisitos establecidos y con el fin de poder concretar las funcionalidades del sistema de cara a su desarrollo. Se han definido las historias de usuario utilizadas en cada uno de los [sprints](#) para poder establecer que elementos de la herramienta se implementarán, dándolos a conocer además, a través del *product backlog*. Las correspondientes historias de usuario pueden verse en el [Apéndice A.4](#).

4.1.2 Diseño

Como se mencionó anteriormente, este *sprint* estaba centrado en concretar los requisitos de la herramienta, así como también las posibles historias que pudieran surgir de los mismos. Lo único que se ha podido obtener en esta fase ha sido mediante la descripción de las [historias de usuario](#), las cuales han permitido obtener a uno de los agentes que interactuará con el sistema, en este caso un único actor, denominado **Usuario**:



Figura 4.1: Actor "Usuario"

4.1.3 Implementación

En este *sprint* no se ha contemplado ninguna tarea o actividad relacionada con la fase de implementación.

4.1.4 Pruebas

En este *sprint* no se ha contemplado ninguna tarea o actividad relacionada con la fase de pruebas.

4.2 Sprint 2

En este *sprint* se llevaron a cabo, después de la correspondiente *sprint review*, las correcciones relacionadas con los requisitos funcionales con el objetivo de poder concretar las funcionalidades de la herramienta de una manera más clara. Además, también se han realizado las correspondientes correcciones en las historias de usuario así como también establecer la granularidad de las mismas. Por otra parte, y una vez realizadas las correcciones de las historias, se ha seleccionado un conjunto de las mismas para poder usarlas en el producto backlog y poder comenzar así, con el desarrollo de las demás fases de este *sprint*.

4.2.1 Análisis

Correcciones de los requisitos funcionales

Se detallarán en el [Apéndice A.2](#) las correcciones realizadas sobre los requisitos funcionales mostrados con anterioridad en la tabla [A.1](#). Para poder facilitar la lectura de los mismos, se expondrán solamente los requisitos funcionales modificados.

Historias de Usuario

Las pertinentes modificaciones realizadas anteriormente en los requisitos funcionales obligan a que las historias de usuario sean modificadas, con ello, se detallan en el [Apéndice A.5](#) las historias de usuario finales y que se usarán en este y todos los demás [sprints](#) a la hora de tener en cuenta la funcionalidad a desarrollar mediante el uso del *product backlog*.

En este *sprint* puede comprobarse el menor número de historias de usuario, esto es debido a la elección de la granularidad con la que se establecerían estas mismas, es decir, una *granularidad alta*. Esto último con el objetivo de evitar la existencia de un gran número de historias que puedan provocar problemas al momento de gestión de las mismas así como su entendimiento. Por tanto, se ha utilizado una representación de las funcionalidades de tipo [Create, Read, Update, Delete \(CRUD\)](#) en aquellas historias de usuario que así lo permitan, más concretamente: HU-1, HU-2 y HU-3.

4.2.2 Diseño

Una vez establecidas las historias de usuario y sus debidas correcciones, se ha podido comprobar la existencia de otro actor, que interactuará con el sistema a partir de las debidas funcionalidades. A continuación se presentan los actores que podrán utilizar la herramienta, eso sí, cada uno a través de sus respectivas funciones:

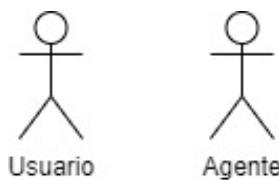


Figura 4.2: Actores que podrán interactuar con el sistema

Con el fin de poder avanzar en el desarrollo y establecer el *product backlog*, se eligieron las siguientes historias de usuario para el desarrollo de este *sprint*: **HU-2, HU-6, y HU-9** (véase tabla [A.5](#)). La elección de estas historias se ha realizado para poder favorecer al alumno el entendimiento del sistema [GOAP](#) y sus componentes: con ello se facilita el poseer una curva de dificultad cuyo requisito en cuanto a nivel de conocimiento vaya en incremento a

medida que avanza el proyecto. Con ello además, también se pretende seguir un desarrollo basándose en procedimientos que han usado desarrolladores de herramientas pertenecientes a la competencia, para poder sobrellevar el desarrollo de una manera cómoda y ágil.

A continuación, se confeccionaron los mockups o bocetos. Estos permitirían el poder tener una idea de como se vería finalmente la interfaz de la herramienta. Aun así, han sido expuestos a cambios a lo largo de las iteraciones hasta llegar a su versión final (véase 4.9.2), debido a las limitaciones de la propia herramienta en cuanto al entorno gráfico. Esto sumado a la inexperiencia del desarrollador en cuanto al tema, han sido factores los cuales han condicionado estas sucesivas correcciones. Los mockups iniciales pueden verse en el **Apéndice B**.

El paso siguiente consistió en el desarrollo de los diagramas de clases, basándose en las historias de usuario elegidas para el *sprint* actual (véase 4.2.1):

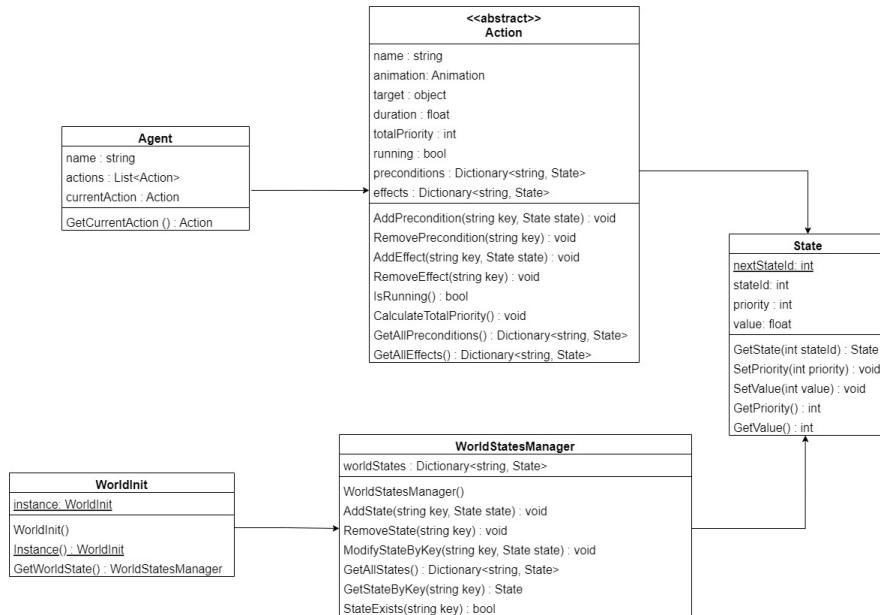


Figura 4.3: Diagrama de clases del Sprint 2

Este diagrama se centra sobre todo, en la implementación de la clase **Action** haciendo referencia a las acciones del sistema **GOAP**. En este diagrama y en relación a las historias escogidas para llevarse a cabo en este *sprint*, se pueden destacar las siguientes variables y funciones junto a sus respectivas clases:

- **Action**:

- **TotalPriority**: será la prioridad total que posee una acción en concreto. Aunque en este *sprint* no esté implementado, consistirá en el sumatorio reiterado de las prioridades, pertenecientes a las precondiciones y efectos de una acción.

- **Preconditions**: estados o condiciones necesarias para poder satisfacer los criterios mínimos de dar cabida a la ejecucción de una acción.
- **Effects**: conjunto de estados o recursos que sucederán de la ejecucción de una acción.
- **CalculateTotalPriority()**: al no estar implementados todavía los estados o recursos necesarios para la creación de precondiciones y efectos, esta función realiza un cálculo simulado de como sería el sumatorio de prioridades.
- **State**: clase usada para simular los estados del mundo, que permitirán que en función del estado en el que un agente se encuentre, pueda cumplir o no con una serie de precondiciones y con ello, ejecutar una acción. Esta clase al igual que **WorldInit** y **WorldStatesManager** son clases implementadas para realizar una simulación en relación a capturar el estado del entorno. No representan la versión final del mismo ya que en este punto del desarrollo y debido a los pocos conocimientos del alumno, no son una representación adecuada de cada uno de los estados que las entidades o el mundo pueden llegar a tener.
- **Agent**: clase usada para simulaciones de los agentes, ya que al tener una amplia relación con las acciones, es necesario implementarla. No obstante, las versiones correctas de la misma se verán en los próximos *sprints*.

4.2.3 Implementación

Una vez realizados los diagramas se continuó con la fase de implementación de las diversas clases. Se expondrán por tanto los diversos aspectos de relevancia que se han podido encontrar de cara a la integración de la herramienta en el motor, además de la consecuente toma de decisiones. Para poder implementar la clase **Action** surgían dos alternativas:

- Por una parte, el contemplar de que se pudieran añadir las acciones mediante código, es decir, creando una nueva clase hijo con sus respectivos valores, obsérvese en el diagrama 4.9 que **Action** es una clase abstracta, por lo que no se podrían crear objetos de la misma. Esto se ha realizado así con la finalidad de poder obtener diversas acciones, diferentes entre sí, y que cada una contemple su propio algoritmo o en este caso, su propia acción a llevar a cabo: ya sea moverse por el mapa, saltar, ...
- Por otro lado, contemplar que una nueva acción se pudiera añadir mediante un botón, es decir, de una manera gráfica. Con la finalidad de acelerar el proceso de desarrollo, se eligió la primera de las dos opciones, postponiendo la segunda para los futuros *sprints*.

En cuanto a la clase **Action** además, cabe resaltar una propiedad importante, y esa es la de **MonoBehaviour** [27]. Ya que como se mencionaba en apartados anteriores, es la clase base

de la que todo componente de [Unity](#) deriva. Todo aquel objeto o instancia que esté presente dentro del mundo de un videojuego y que use el mencionado motor, debe incluir esta clase. Esto es hecho así ya que Unity tiene una particular manera de tratar a los objetos y todas aquellas instancias "físicas" presentes dentro de una [scene](#), ya que para facilitar el poder hacer referencia a las mismas o añadirles propiedades relacionadas con el motor de físicas deben por tanto heredar de `monobehaviour`.

Por otra parte, y para tener una representación abstracta de como tratar a los recursos o estados dentro del mundo, se ha creado una única instancia de la clase **WorldInit**. Esta instancia, no es modificable por otras clases, pero pueden añadirse los estados que se requieran mediante el uso de las otras dos clases **WorldStatesManager** y **State**. Esta última de gran importancia, ya que serviría como base para la implementación propia de los recursos que se verán en los siguientes *sprints*. Los *states* en el caso de esta implementación, son demasiado genéricos ya que admiten un valor de tipo *float* con la finalidad de que sea el propio usuario el que se encargue de gestionar los diferentes objetos que existen: posiciones, inventarios, ... Realmente estas últimas 3 clases han sido usadas para experimentación y diferentes pruebas para comprobar el correcto funcionamiento del resto de clases y funcionalidades. Estas no pueden llegar a estar completas sin el resto del sistema, por tanto las historias de usuario no pueden verse completadas en un único *sprint* al ser [GOAP](#) un sistema el cual depende de varios componentes que simultáneamente se necesitan entre sí.

4.2.4 Pruebas

En este *sprint* las pruebas han sido básicas, se ha comprobado simplemente la creación de los diferentes objetos a partir de la clase **State** y observar el manejo de datos por parte de la clase **Action**. Esto último, con el objetivo de comprobar el correcto funcionamiento de sus respectivas funciones, así como también, de visualizar aquellos datos a través de la consola de [Unity](#).

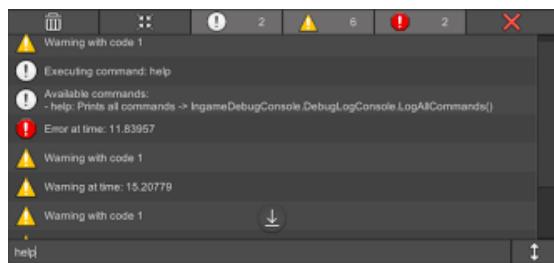


Figura 4.4: Ejemplo de la consola de Unity

4.3 Sprint 3

Nuevamente, y en un paso posterior a la *sprint* review realizada. Se han contemplado las posibles correcciones a realizar en las diferentes fases de este *sprint*, incluyendo por tanto: correcciones en el diagrama de clases, así como también aquellos requisitos funcionales que quizás su descripción no fuera de todo acertada de cara a tener cabida para todos los conceptos que podría abarcar. Además en este *sprint*, se comienza a intentar implementar los diagramas de clases teniendo en cuenta los conocidos patrones de diseño, con el objetivo de poder facilitar el entendimiento al usuario final de la herramienta y también, mejorar la escalabilidad de la misma. Por otra parte, se aclaran varios conceptos sobre las historias de usuario, relacionados con la división en tareas de las mismas.

4.3.1 Análisis

Correcciones de los requisitos funcionales

Se detallarán a continuación las correcciones realizadas sobre los requisitos funcionales mostrados con anterioridad en la tabla A.1 y sujetos a las demás modificaciones realizadas en la tabla A.2. Para poder facilitar la lectura de los mismos, se expondrán solamente los requisitos funcionales modificados.

ID	Corrección
RF-21	Cambio de requisito (nombre y descripción): Este requisito ahora se considerará como "Replanificación de acciones" y contará con la siguiente descripción: <i>Mientras un agente ejecuta una acción pueden darse situaciones externas en las que aparezcan nuevas acciones con más prioridad que la actual, el objetivo del agente será poder seleccionar estas nuevas y diferentes acciones a ejecutar teniendo en cuenta la prioridad de las mismas.</i>

Tabla 4.1: Tabla de requisitos funcionales del Sprint 3

En cuanto al requisito modificado **RF-21: Replanificación de Acciones**, se debe aclarar el funcionamiento del mismo, por tanto consistirá:

- A la hora de que un agente seleccione una acción, se pueden seleccionar varias y no solo una de manera individual. Es decir, formar un plan o conjunto de acciones que conlleven el poder llegar a un objetivo deseado.

- Mientras una acción ocurra, hay que tener en cuenta que pueden surgir condiciones que hagan que una acción deje de ejecutarse y se lleve otra a cabo, conocidas como una serie de “interrupciones”. Es decir, el agente tendrá que replanificar las acciones en caso de que llegue a aparecer una de más prioridad.
- Tener en cuenta que las acciones son atómicas.

Una vez finalizadas las respectivas correcciones de los requisitos funcionales, podrá verse la versión final de la misma en el [Apéndice A.3](#).

Historias de Usuario

- **Historias Heroicas** Con el objetivo de realizar una serie de declaraciones sobre el reparto de tareas en cada una de las historias de usuario. Cabe resaltar el concepto de lo que se conoce como *historias heroicas* o *epic*: estas son aquellas historias de usuario demasiado grandes, tanto que se tienen que descomponer en otras historias de un tamaño adecuado. Es decir, aquellas historias que es difícil abarcarlas en un único *sprint* y que para apalilar esto, se suelen dividir con la finalidad de realizar un mejor seguimiento de la implementación de sus tareas así como también, mejores estimaciones. Básicamente consisten en agrupaciones de historias de usuario que están relacionadas de alguna manera, ya sea por funcionalidad o por subsistemas con los que trabajan. En el caso de este proyecto concreto, se consideran historias épicas a aquellas definidas previamente al final del apartado [4.2.1](#). Es decir, aquellas historias que ya de por sí, eran reconocidas como [CRUD](#): HU-1, HU-2 y HU-3. Por tanto, se considerarán estas últimas como historias heroicas/epic ya que realmente se está utilizando el método de *división por operaciones*, o lo que es lo mismo, reconocerlas como que propiamente son: [CRUD](#). Por otro lado, y con el motivo de no entorpecer el desarrollo, el resto de historias de usuario se ha decidido considerarlas tal y como están descritas, sin realizar ninguna agrupación de las mismas. El motivo principal de no hacerse es por lo explicado en apartados anteriores donde se comentaba la metodología y la planificación elegida (véase [3.4](#)) y como la posibilidad de extender el desarrollo a 1 semana más de margen, posibilita el poder incluir más historias a desarrollar dentro de un mismo *sprint* y al ser estas de una menor duración que las historias épicas, no supondría ningún problema añadirlas a un *sprint* concreto.

En base a los requisitos establecidos y a las correcciones de los mismos vistos en la tabla anterior (véase [4.1](#)) y con el fin de poder concretar las funcionalidades del sistema de cara a su desarrollo, se utilizan las historias de usuario. En el caso de este *sprint* concreto, se continuará con el desarrollo de las historias del *sprint* anterior: **HU-2, HU-6, y HU-9** (véase [4.2.2](#)).

4.3.2 Diseño

A continuación y en base a las historias de usuario previamente mencionadas, se confecionó el correspondiente diagrama de clases, mostrado en la imagen 4.5.

En este diagrama se continua el desarrollo de la clase **Action**, más concretamente con los precondiciones y efectos que estas pueden poseer. Como se mencionó anteriormente en varias ocasiones, ambos son conocidos como estados o representaciones en el mundo de las propiedades concretas de los agentes. Estas representaciones suelen ser bastante abstractas a la par que subjetivas, ya que realmente depende del propio desarrollador intepretarlas y usarlas [9]. Como puede observarse en el diagrama, las clases utilizadas para las pruebas y que de alguna manera simulaban estos estados o **recursos** han sido eliminadas respecto al diagrama del *sprint* anterior (véase 4.9), debido al principal motivo de ser representaciones del mundo demasiado genéricas. Para solventar esto último se ha añadido la clase **Resource**. Por tanto, en este diagrama y en relación a las historias escogidas para llevarse a cabo en este *sprint*, se pueden destacar las siguientes variables y funciones junto a sus respectivas clases:

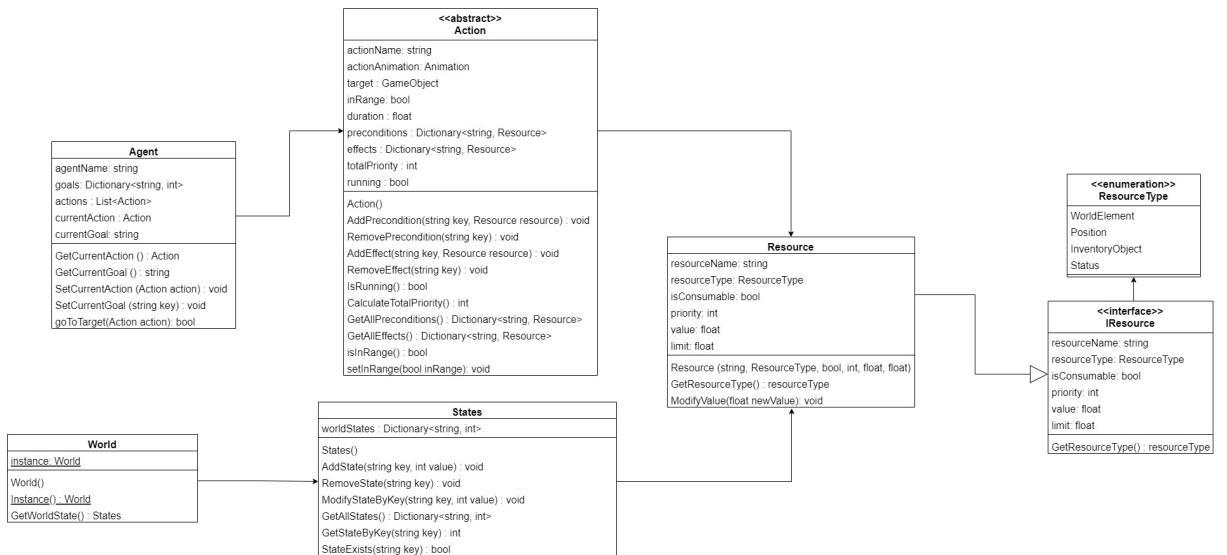


Figura 4.5: Diagrama de clases del Sprint 3

- **Resource**: también conocidos como recursos, serán la representación de aquellos elementos que un agente puede tener en el mundo y en un momento concreto del mismo. Se utilizarán en relación (y como podrá verse en futuros *sprints*) a los estados, con el fin de poder saber que conjunto de recursos posee un agente al encontrarse en una representación del mundo concreta.
 - **ResourceType**: sirve para identificar el tipo de recurso de manera sencilla. Será usado sobre todo de cara a la interfaz gráfica.

- **Value**: valor que puede tener un recurso, en este *sprint* concreto se considera un tipo *float* pero realmente trata de representar cualquier tipo de objeto: posición, cantidad de objetos en el inventario, ...
- **Limit**: limitaciones de cara a la cantidad de recursos que un agente puede tener o distancia que puede recorrer (dependiendo del tipo de recurso que sea).
- **Resource «interface»**: permitirá de manera sencilla implementar diferentes tipos de recursos a través de sus clases hijas.
- **ResourceType**: enumerado que contiene los posibles nombres de los recursos para asignar a un objeto concreto del mundo.

En este *sprint* concreto, a pesar de intentar implementar los posibles *patrones de diseño*, no ha sido posible. Se verá el desarrollo de los mismos en la siguiente iteración.

4.3.3 Implementación

Recursos

La parte más importante de la implementación de este *sprint* consiste en la clase **Resource**, por tanto, con la finalidad de poder clarificar el funcionamiento de los mismos así como también el poder ofrecer una mejor comprensión del concepto, se procederá a detallarlos a continuación.

- **Recurso**: se considera un recurso como aquel elemento que puede poseer un agente o algo que quiere alcanzar (una posición por ejemplo). Y que servirán de base para manipular los estados y el cumplimiento de los mismos, tanto precondiciones como efectos. Obviamente, representar todos los tipos de elementos que pueden existir dentro de un videojuego es una tarea complicada, ya que como se mencionó con anterioridad, es una tarea meramente subjetiva. Aun así, si se tiene en cuenta el funcionamiento de un **NPC** donde por lo usual sus acciones requieren el desplazarse a un lugar o llevar una carga consigo, y esto último puede verse en videojuegos como los expuestos en el apartado [2.1](#), ya podrían obtenerse dos tipos de recursos: relacionados con las posiciones y otros con la cantidad de elementos que se pueden llevar o que es lo mismo, un inventario. Por tanto, para poder establecer un manera clara de representar todos los elementos del mundo se siguen los siguientes **4 tipos** representados:

- **WorldElement**: hace referencia a cualquier objeto de Unity, es decir, a los **gameobject** [\[19\]](#). Usualmente será utilizado para edificios, agentes, o posiciones que quieran definirse mediante la referencia a un objeto y no a una posición concreta.

- **Position:** serán utilizados de manera similar que los **WorldElement**, con la gran diferencia de que estos recursos usan un tipo de objeto diferente, los **Transform** [28]. Que consisten en representar posiciones del mundo a través de un vector de 3 coordenadas (X,Y,Z), o también conocido como **Vector3** [29].
- **InventoryObject:** utilizado para sistemas de inventario, puede ser cualquier elemento del mundo que pueda almacenarse en cantidades. Ejemplos claros de estos recursos son: madera, comida, ... Utilizado usualmente en videojuegos de estrategia.
- **Status:** utilizado para representar aquellos estados abstractos en los que un agente puede encontrarse por diversas situaciones o hechos. Ejemplos claros de estos son: cuando un agente está enfermo, cuando recibe daño de un enemigo, ... Serán representados mediante variables de tipo *bool*.

Una vez explicados los recursos cabe resaltar que en esta implementación, a pesar de tener en cuenta el tipo de valor debe de tener cada uno de los recursos: gameobject, transform, float o bool. Se ha considerado utilizar de manera genérica el tipo *float*, con la finalidad de facilitar las pruebas de esta iteración e implementar este comportamiento en las iteraciones posteriores.

4.3.4 Pruebas

Para poder representar de manera adecuada las historias de usuario a abordar en esta iteración, se han realizado las siguientes pruebas/simulaciones a pesar de no contar con todo el sistema **GOAP** al completo. Esto último repercutirá en varios *sprints*, ya que no podrá hacerse hasta un punto futuro, una versión realista de las mismas. Pese a ello, se han realizado las siguientes pruebas:

- **HU-2** (véase A.4): Para esta historia se han simulado varias partes:
 - Por una parte, el comportamiento del planificador de acciones, que si bien no se ha implementado de una manera realista en este *sprint*, se han introducido los valores correspondientes al agente para que haga las consideraciones oportunas.
 - En combinación con el planificador anterior, se simulará una acción de "ir a...", que básicamente se encargará de comprobar si respecto al límite impuesto por el recurso, el agente puede o no desplazarse hasta y en este caso: una mena de oro. Es decir, se comprobaría si la distancia entre el destino y la posición actual del agente es menor al límite impuesto por la precondición.
 - En caso de que el agente se encuentre a una distancia suficiente como para saber que la mena de oro se encuentra cerca, el estado que posee el agente y previamente definido "isMining" pasaría a tornarse a un valor de verdadero.

- Esto último hará que se cumplan las precondiciones respecto a ejecutar la acción de "picar mineral" y con ello, obtener oro y cumplir con el objetivo impuesto al agente.

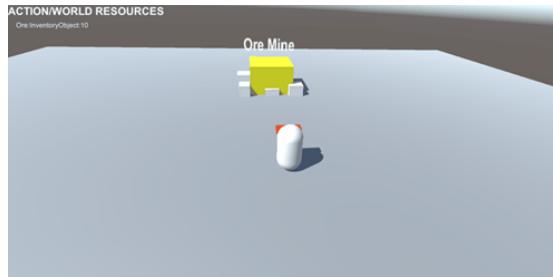


Figura 4.6: Pruebas correspondientes a la HU-2

- **HU-6** (véase A.4): Para esta historia se ha simulado un elemento del escenario (una plataforma) donde comienza con un estado de un valor a 0. Para comprobar el correcto funcionamiento de que se está obteniendo el estado del entorno, se ha simulado la interacción de un jugador a la hora de acercarse a la plataforma, alterando el estado de la misma a estado de un valor igual a 1. A consecuencia, se muestra en todo momento el estado del mundo para ese objeto en concreto, viendo como se altera su valor en tiempo real siempre y cuando el jugador esté interactuando como el mismo.

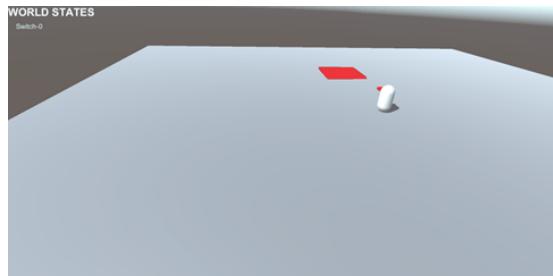


Figura 4.7: Pruebas correspondientes a la HU-6

- **HU-9** (véase A.4): Para esta historia se ha simulado una acción donde el jugador irá tomar un recurso en concreto (oro). Como se han definido las precondiciones y efectos como un tipo de recurso, y estos a su vez serán de diferentes tipos (consumibles, posiciones del mundo, valores booleanos,...) se tendrá que tener en cuenta si se cumple que los respectivos recursos tienen las cantidades o condiciones que se necesitan para realizar las acciones de los agentes. En este caso concreto se ha simulado (creando un nuevo recurso): un efecto. El jugador irá a la mena de oro a obtener el recurso (se simula que hace una acción de "ir a..."), una vez en la mena se podría suponer que realiza otra

acción (picar oro por ejemplo) donde el efecto de esta acción será aumentar el valor del oro que posee el jugador. Y una vez el jugador sale del rango de la mena de oro y a continuación, se podría suponer que se ejecuta otro tipo de acción que le da un bonus por el hecho de haber picado (realizar un incremento del material almacenado por una cantidad determinada, aunque esto es un añadido extra).

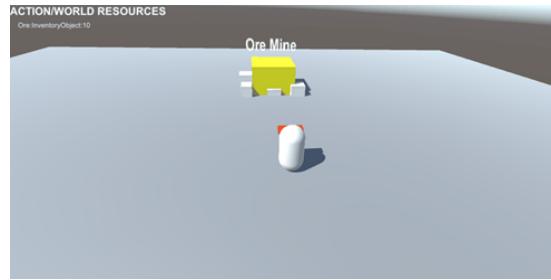


Figura 4.8: Pruebas correspondientes a la HU-9

Cabe recordar que los estados utilizados para las pruebas son realmente simulaciones para poder realizar con éxito las mismas, ya que al no poder obtener en este punto todas las funcionalidades del sistema los estados son representados con las clases auxiliares de **World** y **States** vistas en [4.5](#).

4.4 Sprint 4

En este *sprint* se han realizado las correcciones pertinentes a los diagramas y entre las mismas, el poder contemplar la implementación de patrones de diseño. Así como también, continuar con el desarrollo de las historias de usuario pendientes del *sprint* anterior. En este caso además, se implementará la primera versión de la interfaz gráfica.

4.4.1 Análisis

En este fase no se han contemplado correcciones de requisitos o derivados. Se contempla realizar las correcciones respectivas al anterior *sprint*, así como continuar con el desarrollo de las historias de usuario: **HU-2**, **HU-6**, y **HU-9** (véase tabla [A.5](#)) que se corresponden con los requisitos funcionales especificados a continuación (ver **Apéndice A.3** para más detalles).

- **HU-2**
 - **RF-4:** Añadir nueva precondición.
 - **RF-5:** Modificar precondiciones.
 - **RF-6:** Eliminar precondiciones.

- **RF-7:** Listar precondiciones de una acción.

- **RF-8:** Añadir nueva Acción.

- **RF-9:** Modificar Acciones.

- **RF-10:** Eliminar Acción.

- **RF-17:** Añadir Nuevo Efecto.

- **RF-18:** Modificar Efecto.

- **RF-19:** Eliminar Efecto.

- **HU-6:**

- **RF-20:** Capturar estado entorno.

- **HU-9:**

- **RF-23:** Conocimiento de ejecución de acciones.

4.4.2 Diseño

Continuando con el desarrollo de *sprints* anteriores, en esta fase de diseño se pretende dar cabida a la implementación de **patrones de diseño** [30] siempre y cuando sea posible. Se pretende la integración de los mismos con la finalidad de obtener una herramienta que permita su fácil mantenimiento, así como también la escalabilidad de la misma en un futuro. Además, los patrones de diseño facilitan al usuario final el entendimiento del código, así como también la estructura del sistema. Se detallará a continuación, el diagrama de clases del presente *sprint*:

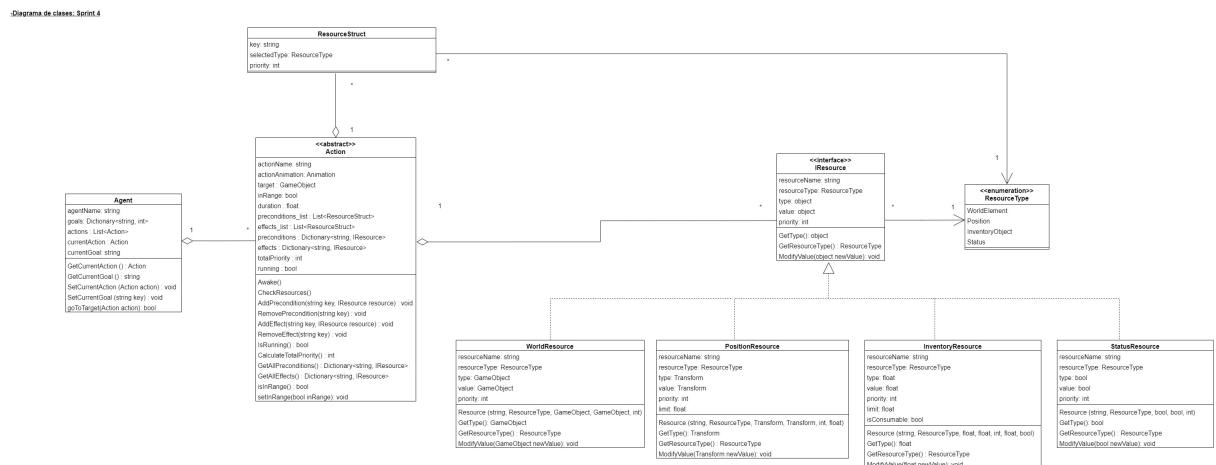


Figura 4.9: Diagrama de clases del Sprint 4

Este diagrama se centra en adaptar los ya conocidos recursos a través de la implementación de la interfaz **IPResource**, donde en este caso concreto, puede verse como el uso de las

clases hijas permite concretar la especificación del recurso en cuestión. Por tanto, en relación a las historias elegidas para la elaboración de este *sprint*, se pueden destacar las siguientes variables y funciones junto a sus respectivas clases:

- **Action**: caben destacar las siguientes funciones y variables de la misma.
 - **inRange**: valor booleano que será usado para los recursos de tipo **WorldResource** y **Position** de cara a poder saber si un recurso se encuentra dentro del rango de un agente, esto es, a una distancia menor que el *limit* del propio recurso. En el caso de los recursos de tipo **Inventory** se comprueba que la cantidad del recurso en cuestión sea menor que la del límite, de no ser así, se avisa mediante la consola al usuario.
 - **isInRange()** y **setInRange(bool inRange)**: funciones que permiten calcular las distancias entre agente y recurso a través de un **Vector3** [29].
 - **preconditions_list** y **effects_list**: listas del tipo **ResourceStruct**, utilizadas para poder facilitar al usuario el agregar precondiciones o efectos a través del **Unity Editor**, a posteriori, se usarán estos valores en las respectivas colecciones de tipo *dictionary*.
- **IResource «interface»**: ya mencionada con anterioridad en el apartado 4.3.2. Además puede encontrarse toda la explicación relacionada con los recursos en el apartado 4.3.3, aunque ahora se realizarán algunas concreciones más específicas.
 - **WorldResource**: es la clase que permitirá la integración de los recursos con tipo *gameobject* [19].
 - * **Type**: solo acepta valores del tipo *gameobject*, permitirá identificar el tipo de recurso en cuestión. Se usará el tipo de datos *Type* y el método *getType()* de **Unity**, los cuales permiten identificar el tipo de datos de un objeto concreto.
 - * **Value**: valor del recurso en cuestión, en caso de ser una precondición valor que deben cumplirse para poder ejecutar la acción. Y en caso de ser un efecto, valor final que espera obtener después de la ejecución de la acción y que afectará a los estados del agente. Véase que ambas variables de esta clase utilizan el tipo de objetos **gameobject** de cara a poder representarlos en el mundo.
 - **PositionResource**: es la clase que permitirá la integración de los recursos con tipo *transform* [28].
 - * **Type**: solo acepta valores del tipo *transform*, permitirá identificar el tipo de recurso en cuestión. Y utilizará los mismos métodos y tipos de variables que los recursos anteriores.

- * **Value**: valor del recurso en cuestión, en caso de ser una precondición valor que deben cumplirse para poder ejecutar la acción. Y en caso de ser un efecto, valor final que espera obtener después de la ejecución de la acción y que afectará a los estados del agente. Véase que ambas variables de esta clase utilizan el tipo de objetos **transform** de cara a poder representarlos en el mundo.
- **InventoryResource**: es la clase que permitirá la integración de los recursos relacionados con algún tipo de sistema de inventario por parte de los agentes. Permitirá agrupar los mismos tipos de objetos y la modificación de la cantidad de los mismos.
 - * **isConsumable**: variable utilizada para identificar si el recurso de tipo inventario en cuestión puede gastarse o no. Puede verse con el ejemplo de recursos relacionados con alimentos o aquellos necesarios para poder crear otros, por ejemplo: el uso de madera para la creación de una casa.
 - * **Type**: solo acepta valores del tipo *float*, permitirá identificar el tipo de recurso en cuestión. Y utilizará los mismos métodos y tipos de variables que los recursos anteriores.
 - * **Value**: valor del recurso en cuestión, en caso de ser una precondición valor que deben cumplirse para poder ejecutar la acción. Y en caso de ser un efecto, valor final que espera obtener después de la ejecución de la acción y que afectará a los estados del agente. Véase que ambas variables de esta clase utilizan el tipo de objetos **float** de cara a poder representar a los recursos que tengan relación con algún tipo de inventario o agrupación de recursos en el mundo, véase el caso de materiales como los ya mencionados: madera, oro, comida, ...
- **StatusResource**: es la clase que permitirá la implementación de aquellos conceptos abstractos por parte de los agentes, tipos de estados a los cuales pueden llegar después de ejecutar una acción como: estar herido, estar cansado, ... o necesarios para la ejecución de la misma, como puede ser el caso de casos concretos en los cuales, por ejemplo, se necesitaría que un agente estuviera en un estado *"enfermo"* para poder curarlo.
 - * **Type**: solo acepta valores del tipo *bool*, permitirá identificar el tipo de recurso en cuestión. Y utilizará los mismos métodos y tipos de variables que los recursos anteriores.
 - * **Value**: valor del recurso en cuestión, en caso de ser una precondición valor que deben cumplirse para poder ejecutar la acción. Y en caso de ser un efecto, valor final que espera obtener después de la ejecución de la acción y que

afectará a los estados del agente. Véase que ambas variables de esta clase utilizan el tipo de objetos **bool** de cara a poder representar los ya denominados recursos abstractos en el mundo.

Cabe destacar que esta interfaz utiliza un tipo de datos **object** en las variables **Type** y **Value**, este tipo hace referencia al manejo de datos que implementa el propio lenguaje C# o otros lenguajes como Java, donde se acepta como parámetro cualquier tipo de dato de manera genérica. Y que por tanto, no debe confundirse con el tipo de datos **Object** en mayúscula, el cual trata sobre el propio manejo de datos de [Unity](#), donde se corresponde al uso genérico de los gameobject y transform.

- **ResourceType**: ya mencionada con anterioridad en el apartado [4.3.2](#).
- **ResourceStruct**: permite que las colecciones de estados de la clase **Action**, más concretamente los efectos y las precondiciones integren los recursos a la hora de poder realizar las comprobaciones necesarias de cada a la ejecución de una acción. Es decir, el cumplimiento de las condiciones necesarias para llevar a cabo la misma, así como las consecuencias de hacerlo.

Patrones de diseño

Como también se mencionó con anterioridad, se han implementado en la medida de lo posible, los patrones de diseño que permitan favorecer el entendimiento y mejorar del sistema. A continuación se expondrán a detalle los patrones implementados:

- **Patrón template o plantilla** [\[31\]](#): en la clase **Action**, a partir de esta clase abstracta se pueden implementar todas las acciones que el usuario desee. Este patrón ha sido usado con el objetivo de dar facilidades a la hora de querer implementar cambios en alguna de las funciones, ya sea en la búsqueda de elementos que estén en rango o no, como añadir o eliminar precondiciones/efectos bajo ciertas condiciones, etc. Es decir, se le da cierta flexibilidad al usuario final para que pueda implementar pequeños cambios en los algoritmos en caso de que fueran necesario. Para poder visualizar de una manera más clara este patrón, se representa a continuación un posible ejemplo de la utilización del mismo, donde se consideran dos acciones: "GetOre" y "GoToOre" las cuales pueden realizar modificaciones en las funciones *isInRange()* y *setInRange(bool inRange)* pudiendo necesitar que un recurso se encuentre a una distancia diferente o no.

CAPÍTULO 4. DESARROLLO

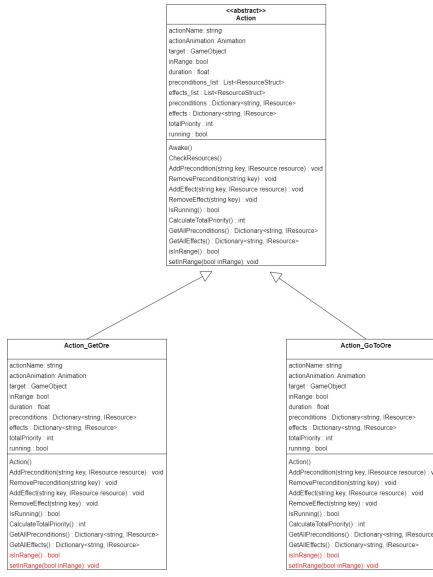


Figura 4.10: Diagrama de clases del Sprint 4, patrón template

- **Patrón estrategia** [31]: por otro lado se ha implementado el patrón estrategia entre la interfaz **IResource** y las clases herederas de la misma, junto a la relación que esta posee con la clase **Action**. Esto se ha hecho con el objetivo de que no se le dé importancia al tipo de recurso que se esté utilizando ya que el consumidor (Action) no se percatará de los cambios que se puedan hacer a la hora de elegir un recurso u otro, o lo que es lo mismo, cambiar de estrategia. Y esto último es debido al uso de una misma interfaz; los distintos comportamientos que puedan tener los recursos se encapsularán dentro de sus propias clases. Para poder integrar este patrón se han cambiado aquellas funciones respecto al *sprint* anterior donde se tenía en cuenta la interfaz como un parámetro, por el atributo *resource* declarándolo previamente como un tipo de *IResource*.

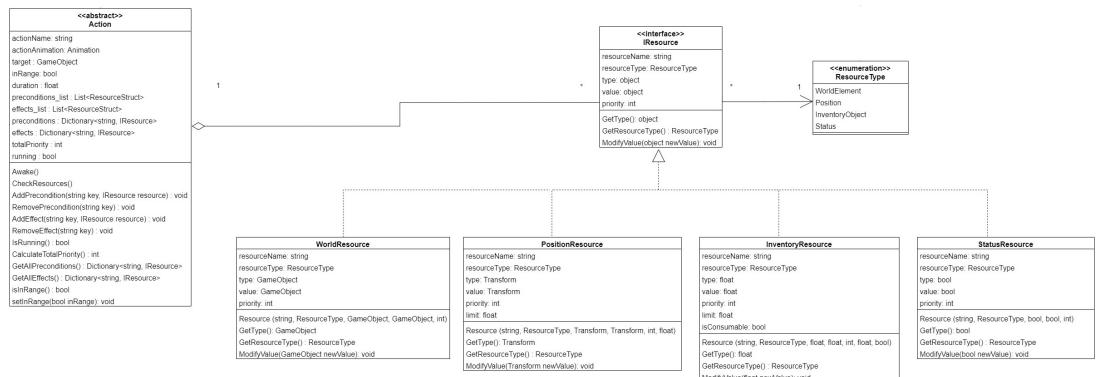


Figura 4.11: Diagrama de clases del Sprint 4, patrón estrategia

Por otra parte, se han realizado diversas implementaciones de la interfaz gráfica, las cuales servirán de base para los posteriores *sprints* y cuyo diagrama se encuentra detallado a continuación:

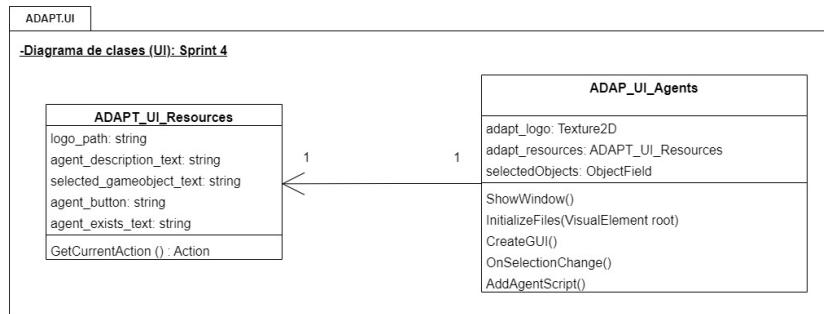


Figura 4.12: Diagrama de clases del Sprint 4, interfaz gráfica

- **ADAPT UI Resources:** realmente es una clase contenedora, que se encargará de tener los respectivos *string* y valores de texto referentes a los botones, paneles, etc. Se agrupan en una clase individual con la finalidad de poder favorecer tareas de **internacionalización** en un futuro, en caso de que fueran necesarias.
- **ADAPT UI Agents:** parte de la interfaz gráfica relacionada con mostrar las variables relacionadas con los agentes. En este *sprint* en concreto solo se contempla la representación de las colecciones de elementos así como también, las demás variables que las acciones pudieran poseer.

4.4.3 Implementación

Una vez realizados los diagramas de clases se continuó con la fase de implementación y con ello, la integración de las las diversas clases y métodos. La parte a destacar en este *sprint* es aquella relacionada con el **polimorfismo** de la interfaz, así como los recursos herederos, cuya explicación se expone a continuación:

La integración del polimorfismo en la interfaz **IResource** se ha realizado con la finalidad de poder tener los respectivos recursos con sus métodos y variables correspondientes. Ya que, como cada recurso va a tratar con un tipo diferente de dato, pues se contempla que el usuario pueda usarlos de una manera sencilla al implementarlos, en vez de acoplar un único recurso en una clase genérica, lo cual implicaría que el usuario tuviera que conocer el tipo del recurso de antemano y pudiera dificultar la creación de los mismos. Es decir, se aplica un **polimorfismo dinámico** o más bien, tratar de primera mano, los recursos de una manera genérica y que sea el propio usuario el que, mediante la creación de un nuevo recurso del tipo que deese, se pueda contemplar la utilización del mismo de cara a usarlos para las precondiciones/efectos correspondientes. Al fin y al cabo, todos los recursos se comportarán de maneras similares, pero no

quiere decir que todas las funciones y el manejo de datos dentro de las mismas tenga que ser idéntico. Por otro lado, en este *sprint* en concreto se han realizado diversas implementaciones de la interfaz gráfica, más concretamente de las primeras implementaciones relacionadas con las variables de tipo *Dictionary* dentro de la clase **Action**, más concretamente, de las listas de efectos y precondiciones, puede verse la versión implementada de la interfaz en la siguiente imagen:



Figura 4.13: Interfaz gráfica de las acciones, Sprint 4

Puede observarse que en este punto se contempla la visualización de las listas de precondiciones y efectos, que debido a su tipo *Dictionary* se han implementado las ya mencionadas variables de tipo *List*: *preconditions_list* y *effects_list*. Ya que *unity* de por sí, no permite la visualización de las colecciones en el *Unity Inspector*. En este caso, no se han llegado a implementar todas las variables dentro de las colecciones de tipo **IResource** y se espera que se implementen en los *sprints* posteriores.

4.4.4 Pruebas

Para poder representar de manera adecuada las historias de usuario a abordar en esta iteración, se han realizado las siguientes pruebas/simulaciones a pesar de no contar con todo el sistema *GOAP* al completo. Esto último repercutirá en varios *sprints*, ya que no podrá hacerse hasta un punto futuro, una versión realista de las pruebas. Pese a ello, se han realizado las siguientes:

- **HU-2** (véase A.4): en relación a esta historia y en concreto en base a la necesidad actual (y a pesar de ser algo relacionado con la HU-1), se ha contemplado la simulación de la integración de agentes, que se desarrollara al completo en *sprints* futuros.
 - Se ha facilitado por tanto, una interfaz que facilite al usuario poder convertir cualquier objeto del entorno en un Agente. Es decir, permitir el añadir un script del tipo **Agent** a un objeto concreto. Como se ha mencionado, no se corresponde a

la creación de un agente como tal, sino a una breve simulación del mismo en base a las necesidades del sistema.



Figura 4.14: Pruebas correspondientes a la HU-2 del Sprint 4

La versión final de esta parte gráfica puede verse en el [Apéndice D](#), más concretamente en la figura [D.8](#).

- Las pruebas relacionadas con el resto de historias de usuario se han realizado de una manera manual. Es decir, a través de la consola de [unity](#) en vista a proseguir con la expansión del sistema en futuros *sprints*, cuyas mejoras permitirán realizar pruebas más precisas y de las cuales puedan obtenerse resultados concretos.

4.5 Sprint 5

En este *sprint* se ha contemplado continuar con el desarrollo de las historias de usuario presentes en los anteriores *sprints*. Todo ello con la finalidad de terminar en este mismo la implementación de estas historias y poder obtener una base sobre la cual ampliar la herramienta en el resto de *sprints*. Se han realizado las correcciones pertinentes en las fases de diseño y sobre todo, en la parte de la interfaz gráfica.

4.5.1 Análisis

Nuevamente, no se contemplan correcciones de requisitos o derivados en esta fase. Se contempla la continuación y corrección de las historias de usuario del *sprint* anterior (véase [4.4](#)), más concretamente: **HU-2**, **HU-6**, y **HU-9** (véase tabla [A.5](#)) que se corresponden con los requisitos funcionales especificados a continuación (ver [Apéndice A.3](#) para más detalles).

- **HU-2**
 - **RF-4:** Añadir nueva precondición.
 - **RF-5:** Modificar precondiciones.

- **RF-6:** Eliminar precondiciones.
 - **RF-7:** Listar precondiciones de una acción.
 - **RF-8:** Añadir nueva Acción.
 - **RF-9:** Modificar Acciones.
 - **RF-10:** Eliminar Acción.
 - **RF-17:** Añadir Nuevo Efecto.
 - **RF-18:** Modificar Efecto.
 - **RF-19:** Eliminar Efecto.
- **HU-6:**
 - **RF-20:** Capturar estado entorno.
 - **HU-9:**
 - **RF-23:** Conocimiento de ejecución de acciones.

4.5.2 Diseño

Continuando con el desarrollo de los *sprints* anteriores, en esta fase de diseño se pretende dar cabida a aquellas parte de la herramienta donde, por motivos del propio motor **Unity**, se han tenido que adaptar para el correcto funcionamiento de las mismas. Así como también, el poder visualizar correctamente aquellos datos o información necesaria en la interfaz gráfica. Esto puede entenderse de una manera más clara observando el diagrama de clase a continuación:

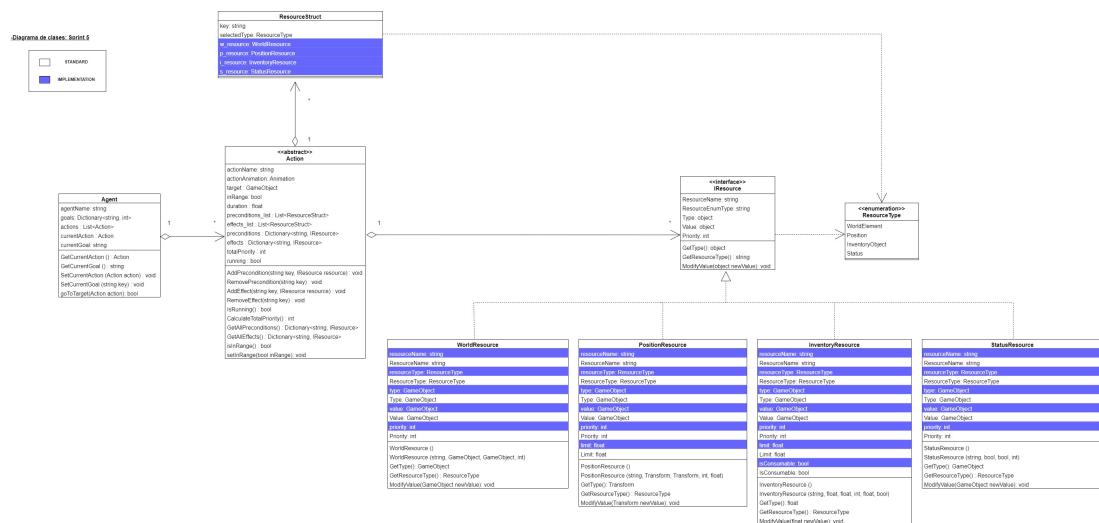


Figura 4.15: Diagrama de clases del Sprint 5

Donde se representan en varios colores, las diferentes variables y métodos necesarios para la implementación de la herramienta. Por una lado, se destacan en color azul aquellas partes necesarias para la interfaz gráfica y que por motivos ajenos, se han tenido que replicar variables o añadir algunas que permitan la correcta visualización de la herramienta. A su vez, también se contemplan aquellas funciones propias de Unity, que serán detalladas en los próximos *sprints* al no haber uso de las mismas en el *sprint* actual. Entre estas variables de la parte orientada a la implementación pueden destacarse las siguientes:

- **ResourceStruct:**

- **w_resource:** hace referencia a los recursos de tipo **WorldElement**. Permite la correcta visualización de los mismos en el inspector de Unity.
- **p_resource:** hace referencia a los recursos de tipo **Position**. Permite la correcta visualización de los mismos en el inspector de Unity.
- **i_resource:** hace referencia a los recursos de tipo **Inventory**. Permite la correcta visualización de los mismos en el inspector de Unity.
- **s_resource:** hace referencia a los recursos de tipo **Status**. Permite la correcta visualización de los mismos en el inspector de Unity.

Cabe destacar que la implementación de estos recursos y su correcta visualización es debido a, como ya se ha mencionado, limitaciones de **Unity**. Más concretamente a como este motor trata a las clases abstractas e interfaces, que no permite mostrar los valores de las mismas en la propia interfaz gráfica. Esto último es debido a que ambas, tanto las interfaces como las clases abstractas, no tienen un tipo especificado de datos (es decir, de base se consideran como genéricas) a ello que el propio motor no sepa como representarlas y sea el propio desarrollador el que tenga que indicarle explícitamente como ha de hacerlo y en función de que tipo de datos (en caso de esta herramienta pues sería representar los *recursos* y sus clases derivadas).

- **Clases herederas de IResource:**

- **resourceName:** nombre del recurso, tiene que implementarse de esta manera para ser visualizado correctamente por la parte de la interfaz gráfica de Unity.
- **resourceType:** tipo de recurso, enumerado concretamente. Tiene que implementarse de esta manera para ser visualizado correctamente por la parte de la interfaz gráfica de Unity.
- **type:** tipo de recurso en función de los métodos propios de Unity (derivados de la clase *Type*). Tiene que implementarse de esta manera para ser visualizado correctamente por la parte de la interfaz gráfica de Unity.

- **value**: valor que podrá tener el recurso, ya sea precondición o efecto. Tiene que implementarse de esta manera para ser visualizado correctamente por la parte de la interfaz gráfica de Unity.
- **priority**: prioridad del recurso. Tiene que implementarse de esta manera para ser visualizado correctamente por la parte de la interfaz gráfica de Unity.
- **limit**: valor límite del recurso. Tiene que implementarse de esta manera para ser visualizado correctamente por la parte de la interfaz gráfica de Unity.

Por otra parte, también se han contemplado modificaciones en el diagrama de clases relacionando íntegramente con la parte de la interfaz. Esto es debido, al querer representar correctamente los recursos así como todas las variables respectivas a las acciones, además, se añadirán los botones correspondientes así como también los efectos que pueden causar los mismos (añadir precondiciones/efectos), el diagrama se muestra a continuación:

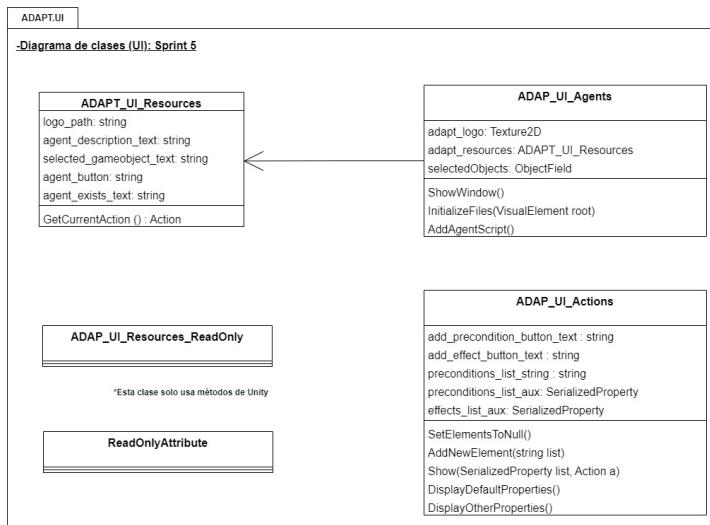


Figura 4.16: Diagrama de clases del Sprint 5, interfaz gráfica

En este *sprint* en concreto puede destacarse la clase **ADAPT UI Actions**, la cual se detalla a continuación:

- **Preconditions_list_aux** y **Effects_list_aux**: son variables utilizadas como soporte para la correcta visualización de las listas de precondiciones y efectos.
- **SetElementsToNull()**: función utilizada para reiniciar todas las variables necesarias por temas de implementación, implementadas en el diagrama anterior 4.15.
- **AddNewElement(string list)**: añadirá de manera dinámica, todos los elementos de los diccionarios correspondientes, así como también, de las listas de recursos a la lis-

ta auxiliar correspondiente. Cabe destacar que Unity, a la hora de representar los correspondientes elementos gráficos utiliza un tipo de datos denominado **SerializedProperty** [32]. Esto es, el permitir editar los objetos de una manera genérica y permitir el leer o cambiar el valor de una propiedad concreta, sobreescribiendo aquellas modificaciones gráficas que otros scripts o clases pudieran realizar sobre esa propiedad en concreto.

4.5.3 Implementación

En cuanto a la implementación de este *sprint* en concreto, ha consistido meramente en las modificaciones respectivas a la interfaz gráfica. Es decir, la correcta visualización de los recursos y la facilidad de uso de los mismos de cara al usuario. Cuya versión final puede verse en el **Apéndice D**, más concretamente en la figura D.3. Se incluye la visualización de cada una de los propiedades de los recursos, expuestas en cada uno de los diagramas, además, en función del tipo de recurso seleccionado en el desplegable **Selected Type** podrá aparecer unos campos u otros, dependiendo del mismo. Tanto la lista de precondiciones como la de efectos funciona de manera similar, además, se incluirá un botón que permita eliminar el elemento específico de cada una de las listas. Por otra parte, el elemento **Key** que puede verse en la representación final de la interfaz, hace alusión al nombre propio de recurso. Se añadirán además, los ya mencionados botones que permitan, en función de con cual se interaccione, añadir un recurso a la lista de precondiciones o a la de efectos. Los campos de **Total Priority** y **Total Cost** serán calculados en tiempo de ejecución, con el correspondiente sumatorio de las prioridades y costes de las precondiciones y efectos.

4.5.4 Pruebas

Las pruebas de este *sprint* han consistido en comprobar la correcta visualización de los recursos. Así como también, comprobar la inexistencia de posibles valores a nulo para evitar posibles errores en el manejo de la herramienta. Todas las pruebas realizadas están basadas en revisar los desplegables y desplegar el recurso en concreto elegido, así como también todas las propiedades relacionadas a los mismos. Por otra parte, la comprobación de los cálculos en los campos de prioridad y coste total se ha realizado en base a cálculos simples de manera manual y con ayuda nuevamente, de la consola de Unity.

4.6 Sprint 6

Este junto a los siguientes *sprints* son clave para la implementación de la herramienta, ya que a diferencia de como se ha realizado en *sprints* anteriores donde se revisaba el diagrama de clases y posibles correcciones de las historias de usuario. En este caso concreto se abarca la

implementación del planificador de acciones, el cual además, será explicado en detalle para entender el funcionamiento del mismo. Se conciben también la implementación y desarrollo de otras propiedades, que servirán como base para las funcionalidades de posteriores *sprints*.

4.6.1 Análisis

No se contemplan correcciones en requisitos funcionales ni derivados. Además se dan por finalizadas las historias de usuario mencionadas en *sprints* anteriores y cuya corrección se ha mantenido durante un largo período. Por ello y después de la correspondiente *sprint review*, se ha decidido continuar con el desarrollo de la herramienta en base a las siguientes historias de usuario: **HU-1, HU-3, HU-7 y HU-10** (véase tabla A.5) que se corresponden con los requisitos funcionales especificados a continuación (ver [Apéndice A.3](#) para más detalles).

- **HU-1**
 - **RF-1:** Añadir agente [GOAP](#).
- **HU-3:**
 - **RF-11:** Listar acciones de objetivo.
 - **RF-12:** Añadir nuevo objetivo.
 - **RF-13:** Modificar objetivo.
 - **RF-14:** Eliminar objetivo.
 - **RF-15:** Listar objetivos de agente.
- **HU-7:**
 - **RF-21:** Replanificación de acciones.
- **HU-10:**
 - **RF-24:** Selección de objetivo.

4.6.2 Diseño

En relación a las historias de usuario planeadas para este *sprint* concreto, se muestra el diagrama de clases desarrollado en esta fase. En concreto, el desarrollo de los agentes y los objetivos que los mismos pueden tener. Así como también el planificador de acciones, cuyo funcionamiento será explicado en detalle en la fase de implementación 4.6.3.

CAPÍTULO 4. DESARROLLO

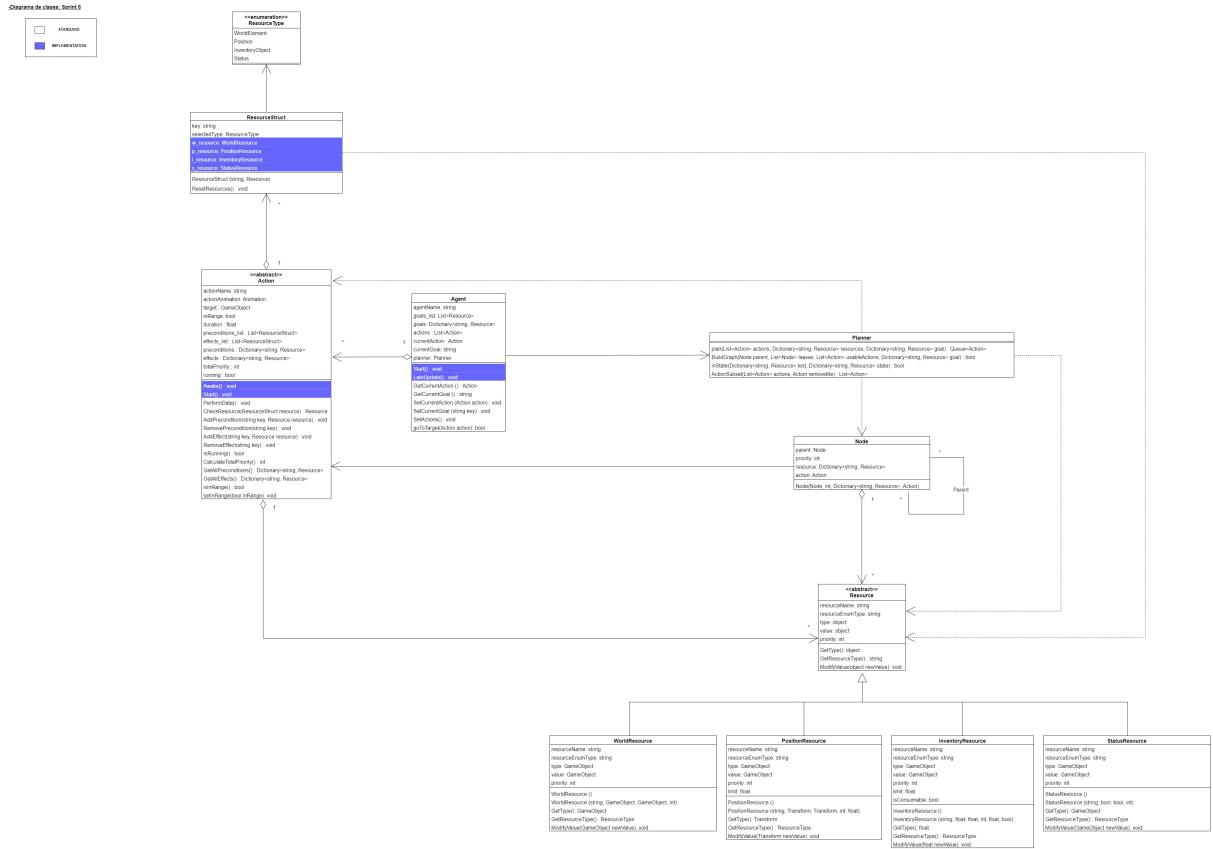


Figura 4.17: Diagrama de clases del Sprint 6

Una vez expuesto el diagrama de clases corresponde mencionar clase por clase, las variables de cada una así como sus funciones y la justificación de las mismas:

- **Action:** como se mencionó en *sprints* anteriores, se corresponde a las acciones que pueden tomar los agentes.
 - **Awake()** [33]: al igual que también se comentó en otros *sprints*, esta es una función propia de la implementación del motor. Consiste en una función orientada a los tiempos de ejecución, concretamente se inicializa cuando se carga lo que se conoce como una **scene** de Unity o lo que viene siendo, un mundo. Esta función en este caso es utilizada para inicializar las correspondientes variables, por ejemplo: los diccionarios y las listas.
 - **Start()** [34]: ocurre de manera similar que con la función anterior, en este caso `Start()` se ejecuta también al cargar una escena o **scene** pero al momento después del `Awake()`. Se suele utilizar también para la inicialización de variables o aquellas funciones que requieran una única llambda, en este caso se utiliza para las funciones **PerformData()**, **CheckResource()**,

- **PerformData()**: función utilizada para el manejo de diccionarios y lista. Realiza un volcado de datos de las listas que se usan en la interfaz gráfica hacia los diccionarios, para su uso posterior en otras clases.
 - **CheckResource(ResourceStruct resource)**: realiza una serie de comprobaciones en función del recurso recibido, se utiliza de manera auxiliar a la interfaz gráfica, de modo que dependiendo del recurso que tenga una lista o diccionario sepa de que tipo es, y con ello, realizar las operaciones pertinentes.
 - **Agent**: clase que hace referencia a los propios agentes o entidades que serán el núcleo del presente proyecto. Podrán ejecutar una serie de acciones en caso de tener asignado un plan, es decir, un conjunto de las mismas que permita llegar hasta uno de los objetivos que estos mismos poseen.
 - **goals_list**: de manera similar a como ocurría con las listas de precondiciones y efectos en la clase **Action**, esta variable será utilizada para el manejo de los objetivos en la interfaz gráfica, que a su vez, serán recursos. Por tanto a la hora de realizar las debidas representaciones se hará de una manera similar a como se ha visto con los recursos y sus respectivas listas.
 - **goals**: diccionario que contendrá los respectivos objetivos del volcado de datos de **goals_list**. Permitirá usar los objetivos en el planificador de acciones y donde se requiera.
 - **planner**: instancia de la clase **Planner**, planificador de acciones que permitirá obtener un conjunto de las mismas con la finalidad de que a través de su ejecución, pueda cumplirse un objetivo. En vista a *sprints* anteriores, cabe mencionar que las acciones deben cumplir con las precondiciones para su posterior ejecución, y que esto último recae en la consecución de unos efectos aplicados sobre los estados del mundo.
 - **Start()** [34]: se utiliza de manera similar que en las clases anteriores, en este caso concreto, para la inicialización de los diccionarios y listas.
 - **LateUpdate()** [35]: es utilizado de manera similar a **Start()**, con la gran diferencia de que en este caso concreto se ejecuta al menos una vez por cada **frame**, usualmente suele ser al finalizar este mismo. En el caso concreto de esta clase se utiliza esta función para iniciar el **planner**, con la finalidad de reiniciarlo al final de cada uno de los *frames*.
 - **Node**: cada uno de los nodos que conformará el árbol de nodos utilizado en el **pathfinding**. Cada uno de ellos corresponde a una acción.
 - **parent**: nodo padre, del cual se extiende el nodo en cuestión como una hoja.

- **priority**: prioridad que tendrá cada uno de los nodos, esta debe ser calculada en base al algoritmo a utilizar.
 - **resource**: estado actual en el que puede encontrarse un nodo, este punto será detallado en *sprints* posteriores con sus correspondientes modificaciones.
 - **action**: acción que formará el nodo en sí y que permitirá a un agente ejecutarla llegado el momento dado.
- **Planner**: planificador de acciones, el cual permitirá que en función del estado actual en el que se encuentre el agente (es decir, el conjunto de recursos que tenga un agente en un momento dado) pueda cumplir con una serie de precondiciones. Según el cumplimiento, o no, de estas precondiciones se podrá ejecutar una acción cuyos efectos podrán hacer que se ejecute otra y así consecuentemente hasta llegar a alcanzar uno de los objetivos deseados del agente.
 - **Plan(actions, resources, goal)**: será la cola de acciones que devolverá el planificador en base a unas *actions* o lista de acciones que deben comprobar el cumplimiento de sus precondiciones y la consecución de sus efectos, unos *resources* o conjunto de recursos en un momento dado, los cuales se usarán para comprobar el cumplimiento de las precondiciones. Y también en base a *goal* o un diccionario que tenga el conjunto de objetivos a alcanzar por el agente.
 - **BuildGraph(parent, leaves, usableActions, goal)**: devolverá un valor de tipo *bool* el cual indicará si se ha podido crear un plan de acciones que satisfaga un objetivo. Esta función consiste en la creación del árbol de nodos.
 - **InState(test, state)**: recibe dos parámetros entre los cuales se tiene a **test** o el conjunto de precondiciones a contrastar con **state** o el conjunto de recursos que forma un estado. Función usada por **BuildGraph** para comprobar si un conjunto de recursos (considerado un estado) satisface la precondición de una acción concreta.
 - **ActionSubset(actions, removeMe)**: función que realiza sucesiva iteraciones sobre la construcción del árbol. Trabaja de manera similar a las listas de lenguajes funcionales, donde a partir de una lista o una lista de **actions**/acciones en este caso y la iteración sobre la misma, se van descartando elementos sucesivamente.
- **Resource**: aunque se ha explicado con anterioridad en pasados *sprints*, cabe resaltar que en este caso se ha pasado de tener una interfaz (*IResource*) a formar una clase abstracta. Aplicar esto tiene una gran ventaja y es el poder hacer uso de uno de los patrones de diseño también ya comentados: el **patrón plantilla**. El cual estará conformado por esta

clase y las herederas de la misma y permitirá el poder crear diversos recursos así como también la ampliación de los mismos en el futuro si se requiriera.

4.6.3 Implementación

4.6.4 Planificador de acciones

A continuación se va a explicar el núcleo de estas y de las posteriores iteraciones, así como uno de los pilares que sostiene la herramienta a desarrollar en el presente proyecto, este es: el planificador de acciones.

Primeramente cabe redefinir en lo que consiste este término, el planificador consiste en aquel sistema que busca un plan de acciones adecuado para poder llegar a alcanzar un objetivo. Más concretamente, un agente que poseerá un conjunto de acciones y a su vez, un conjunto de metas. Ambos serán usados como parámetros de un proceso o sistema que su principal función es, que mediante los estados (o considerados también conjuntos de recursos en un momento concreto) que un agente tenga, se pueda contrastar el cumplimiento de las precondiciones. Cumplidas estas, las acciones pueden ejecutarse ocasionando unos efectos que a su vez, serán las precondiciones de otra acción. Esto es un proceso en cadena que será realizado hasta que las acciones completen una secuencia que les permita llegar desde el estado inicial hasta el *goal* del agente. Es lo que se conoce como formular un **plan** (véase 2.1.1 para más información).

Algoritmos de búsqueda

Para poder representar un algoritmo que realice lo que un planificador de acciones realiza se necesita algún tipo de representación o mecanismo. Aquí es donde entra la similitud de la funcionalidad de este sistema con un algoritmo de búsqueda o un **pathfinding**. Ya que estos al fin y al cabo, se basan en la búsqueda del camino menos "costoso" o básicamente el de mayor prioridad de nodos con tal de alcanzar una meta fijada, pudiendo intercambiar el nodo elegido o no, esperando encontrar el mejor camino posible. Existen varias implementaciones de **GOAP** con diferentes algoritmos de búsqueda, pero la que usualmente se suele utilizar es **A***. Podría surgir la duda del porqué usar este algoritmo y no otro como **Dijkstra**, la diferencia es clara. Dijkstra y de manera similar a lo que lo hace A*, trata de buscar el camino más corto a partir de un nodo origen, esto es mediante la construcción de un conjunto de nodos entre los cuales, existirá una distancia mínima. A* añade un nuevo concepto a esta versión y es el uso de la **heurística**. En el caso de un videojuego, la heurística es algo que proporciona grandes ventajas, ya que consiste en dar un coste estimado de lo que podría llegar a suponer usar un nodo u otro para obtener una ruta o camino hacia la meta. Además, la heurística suele ser algo subjetivo realmente, cualquier desarrollador puede aplicar el concepto que deseé para poder

aplicarla y esto es algo que da gran flexibilidad al algoritmo, este último concepto no obstante, se detallará en el siguiente *sprint*, donde se verá el cálculo de la heurística elegida. No obstante, cabe resaltar que esa flexibilidad que proporciona este algoritmo es demostrable, ya que en caso de considerar una heurística de un coste igual a 0, pasaría a transformarse nuevamente en un algoritmo de Dijkstra (cabe mencionar que existen otros estándares de estimación de la misma como la distancia de **Manhattan** [36]).

Funcionamiento del planner

Para poder visualizar mejor el funcionamiento del planificador y el algoritmo, se van a presentar las siguientes imágenes ilustrativas del concepto:

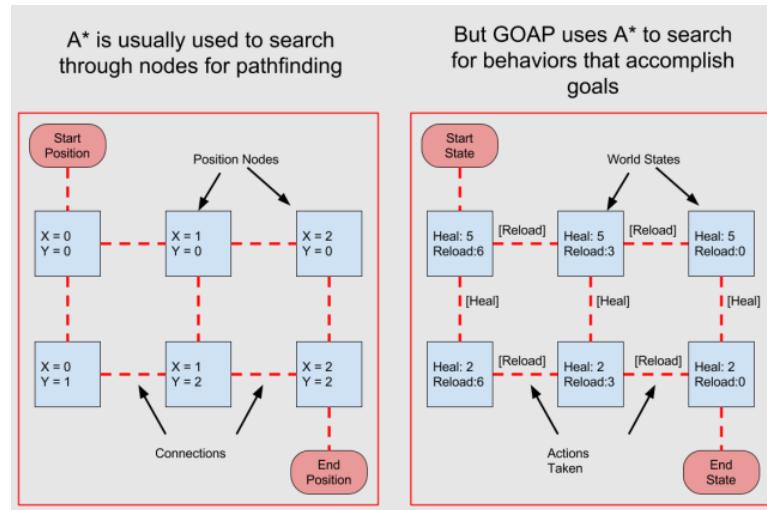


Figura 4.18: Un diagrama que compara la aplicación de GOAP de A* con pathfinding [1]

En esta figura en concreto puede visualizarse la manera que tiene el sistema GOAP de extrapolar el algoritmo de búsqueda de cara a orientarlo a las ya denominadas acciones y estados. Si bien en un algoritmo A* se tratarían cuestiones donde se tiene un valor de coste "x=0" y por otro lado una heurística de "y=0" en **Goal-Oriented Action Planning (GOAP)** se tendrían en cuenta el estado actual del agente donde tendría los recursos tales: "Heal: 5" y "Reload: 6" y se trataría de buscar la acción que junto a sus precondiciones pudiera satisfacer el estado.

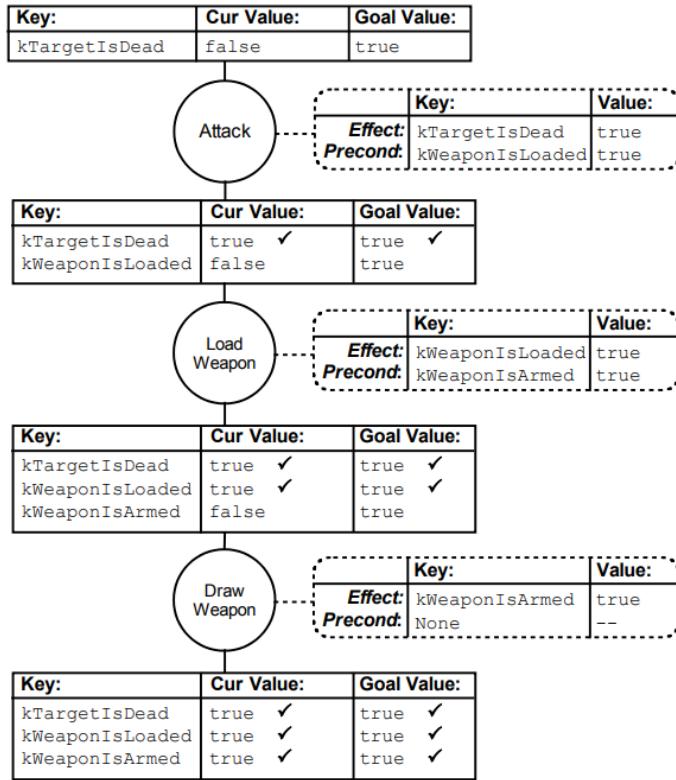


Figura 4.19: Funcionamiento del planner GOAP

Continuando con la imagen anterior (4.19), puede verse un ejemplo real en donde un agente pudiera estar en un estado tal que: "kTargetIsDead", es decir, buscar que otro agente pudiera estar o no eliminado. Puede verse que la acción para que eso ocurra sería la de "Atacar/Attack" cuyo efecto sería que el estado anterior de "kTargetIsDead" se hiciera cierto. Pero previamente a ejecutar esa acción de "Atacar/Attack" el agente tendría que cumplir con las precondiciones necesarias, es decir, tener un recurso (un recurso de tipo **Status** si se extrapolara a la herramienta del presente proyecto) que cumpliera que la condición "kWeaponIsLoaded" tuviera un valor "verdadero". Estas comprobaciones son reiteradas, es decir, constantemente se está comprobando si en algún momento algún estado se cumple para poder comprobar si se adaptan a alguna de las precondiciones de una acción. Esto último puede realizarse mediante los repetidos reinicios o interrupciones del *planner* provocadas por la prioridad de las acciones, ya que, en el momento de encontrar una acción/objetivo que se pueda ejecutar y esta tenga una mayor prioridad que la actual o la que se esté ejecutando en ese momento, el *planner* interrumpirá la ejecución y formará un nuevo conjunto de acciones.

Implementación del planner

En cuanto a la propia implementación de código del planificador de acciones, si bien se ajusta a un estándar A^* al uso, cabe resaltar una importante modificación. Ya que normalmente, la mayoría de implementaciones **GOAP** así como parte de los ejemplos de algoritmos de búsqueda relacionados con el que se comenta. Implementan una búsqueda de cara a encontrar el nodo con el "menor coste" asociado, o lo que es lo mismo, buscar el camino más "barato" en cuanto a costes asignados a cada uno de los respectivos nodos. Esta herramienta pretende diferenciarse en este punto al darle control total al usuario, ya que, la búsqueda se realiza por prioridad y no por coste, es decir, a cuanta mayor prioridad mejor será para el algoritmo de búsqueda. A ello que, se le permita al usuario el poder integrar su propia prioridad tanto en precondiciones como efectos (y los respectivos cálculos para poder obtener así, la prioridad total de la acción).

```
1 //Get the highest priority Leaf
2     foreach (Node leaf in leaves)
3     {
4         if (highest == null)
5         {
6             highest = leaf;
7         }
8         else
9         {
10            if (leaf.priority > highest.priority)
11                highest = leaf;
12        }
13    }
```

Por último y en relación a la fase de implementación de este *sprint*, cabe resaltar la utilización de una cola o **Queue** como manera de almacenar el plan de acciones (puede verse en la clase **Planner** en el método *Plan*). Esto se ha realizado así por una parte, debido a la inexperiencia del desarrollador a la hora de desarrollar un sistema de estas características y con el objetivo de facilitar la implementación y por otra parte, a la facilidad de poder ejecutar las acciones, ya que al ser una cola, las acciones podrán venir ordenadas y con ello ejecutarse de manera secuencial.

4.6.5 Pruebas

La fase de pruebas de este *sprint* ha consistido en comprobar el funcionamiento del planificador mediante la **consola de Unity**. Donde a través de la cual, se intentaba obtener el conjunto de nombres de acciones que podría ofrecer el *planner* ante los respectivos *goals* ofrecidos. Con ello además, todas aquellas historias de usuario y con ello sus funcionalidades,

respectivas de este *sprint* y que no se hayan contemplado a través de una interfaz gráfica, quedaron relegadas a implementarse a través de código, pudiendo crear agentes con sus respectivos objetivos a través del mismo y usando la consola para verificar el funcionamiento correcto.

4.7 Sprint 7

4.7.1 Análisis

En este *sprint* tampoco se contemplaron correcciones de requisitos o derivados, al igual que en el resto de los mismos. Continuando con las funcionalidades del *sprint* anterior, este pretende terminar esas historias de usuario pendientes así como también añadir algunas funcionalidades extras para poder aprovechar mejor el tiempo de desarrollo. Con estas funcionalidades también surgen cambios en el diagrama de clases así como sus respectivas correcciones. Por tanto se continuará el desarrollo de la herramienta en base a las siguientes historias de usuario: **HU-1, HU-3, HU-7, HU-8 y HU-10** (véase tabla A.5) que se corresponden con los requisitos funcionales especificados a continuación (ver **Apéndice A.3** para más detalles).

- **HU-1**
 - **RF-1:** Añadir agente **GOAP**.
- **HU-3:**
 - **RF-11:** Listar acciones de objetivo.
 - **RF-12:** Añadir nuevo objetivo.
 - **RF-13:** Modificar objetivo.
 - **RF-14:** Eliminar objetivo.
 - **RF-15:** Listar objetivos de agente.
- **HU-7:** Cabe resaltar que de esta historia de usuario solo se estableció la base para poder implementarla en su totalidad en el *sprint* siguiente.
 - **RF-21:** Replanificación de acciones.
- **HU-8:**
 - **RF-22:** Gestionar Inventario.
- **HU-10:**
 - **RF-24:** Selección de objetivo.

4.7.2 Diseño

En el caso de este *sprint* concreto, no ha habido muchas modificaciones de cara al diagrama de clases. Esto es debido a que la mayoría de las implementaciones a realizar tienen relación con las correcciones de historias de usuario pasadas. Por tanto, se hará hincapié a las funcionalidades relacionadas con el planificador de acciones, más concretamente, el apartado de prioridades y heurística del mismo (desarrollado este último en el apartado correspondiente), así como también aquellas funcionalidades extras que hayan podido ser añadidas en este período de desarrollo. A continuación se muestra el diagrama perteneciente al desarrollo de este mismo *sprint*:

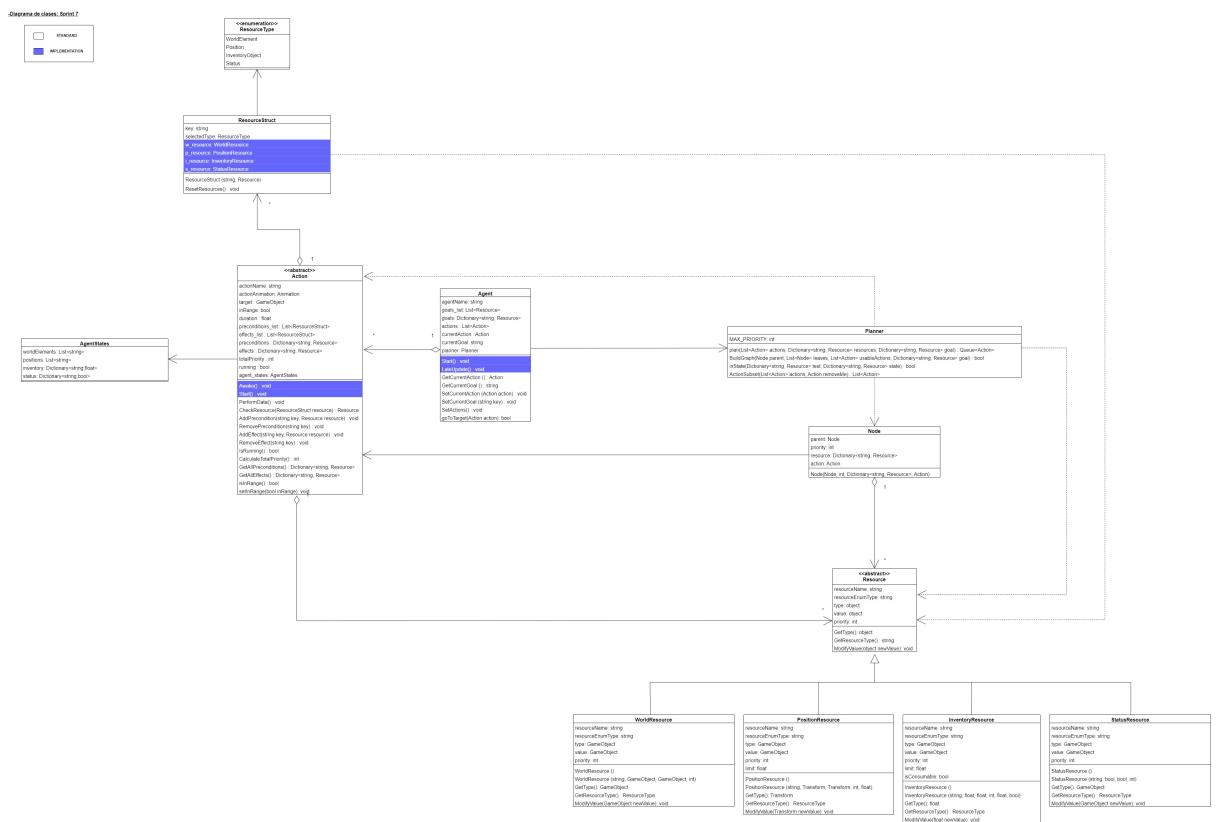


Figura 4.20: Diagrama de clases del Sprint 7

Una vez expuesto el diagrama de clases corresponde mencionar aquellas clases que tengan relación con las historias de usuarios pertenecientes a este período del desarrollo, así como también, las variables de cada una, sus funciones y la justificación de las mismas:

- **AgentStates**: componen el inventario de los agentes. Es decir, el conjunto de recursos que poseeran aquellos **NPC** en un momento dado, que son los que se conocen también como **estados**. Estos serán los elementos esenciales de cara a comprobar que condicio-

nes se cumplen y cuales pueden ser añadidas o no, al plan (o conjunto de acciones). De manera similar a lo que se ha llevado a cabo hasta el momento, donde las precondiciones y efectos se han considerado como **4 tipos** diferentes de recursos, siendo así: *WorldElement*, *Position*, *Inventory* y *Status* se tendrán en cuenta las mismas consideraciones de cara a la implementación de estos estados. Los cuales, serán desarrollados con más detalle en el próximo *sprint*.

- **WorldElements**: conjunto de estados relacionado con recursos de tipo *WorldElement*. Se necesita solo una referencia de nombre para poder referirse a los mismos.
- **Positions**: de manera similar a la variable anterior, conjunto de estados relacionado con recursos de tipo *Position*. Se necesita solo una referencia de nombre para poder referirse a los mismos.
- **Inventory**: consiste en el conjunto de recursos de tipo *Inventory*. Se necesita una referencia de nombre además de un valor inicial para poder inicializarlos.
- **Status**: consiste en el conjunto de recursos de tipo *Status*. Se necesita una referencia de nombre además de un valor inicial para poder inicializarlos.

4.7.3 Implementación

Heurística en el planner

Una vez realizado y explicado el diagrama de clases del presente *sprint* se procederá a entrar en detalle con la gestión de la heurística respecto al planificador de acciones. Para ello, cabe resaltar una variable que no se mencionó anteriormente para poder detallarla ahora en la explicación, y esta es la constante *MAX_PRIORITY*. Esta variable servirá para establecer la prioridad máxima (es decir, el valor que permitirá elegir entre una acción u otra) que puede asignarse. Aunque en este *sprint* en concreto se usa como inicialización del nodo padre, es decir, el primer nodo de todos, que en sí no es una acción, sino que sirve de nodo auxiliar para crear el árbol de nodos al completo. Por tanto, la heurística aplicada en el desarrollo de la siguiente herramienta necesita la explicación del siguiente fragmento de código:

```
1 Node node = new Node(parent, parent.priority +
    a.CalculateTotalPriority(), currentState, a);
```

La variable *node* en este caso puede obviarse, ya que serán los nodos abiertos, es decir, acciones sin todavía navegar del algoritmo de búsqueda y que serán usadas en otras partes del código. La parte importante de este fragmento es la asignación que se realiza: *parent.priority + a.CalculateTotalPriority()*. Ya que, en este *sprint* en particular, consiste en el sumatorio de la prioridad del nodo padre, es decir, del que tenga el nodo en cuestión antes que él mismo en el

árbol de nodos, además de su propia prioridad, que sería la asignada por el propio usuario. Esto realmente es una implementación auxiliar pero que sirve como una simulación de pasar de realizar un algoritmo A* a un algoritmo del mismo tipo, pero especializado en una **búsqueda en anchura**. Realmente esto último ha sido implementado así ante la necesidad de otra parte importante para el cálculo de la heurística, los **costes**. Que serán implementados en el *sprint* siguiente, más concretamente en el apartado 4.8.2, para dar así por finalizada la explicación de la heurística.

Por otra parte, la implementación en general ha consistido en correcciones relacionadas con el planificador, como se había mencionado anteriormente. Además, la implementación referente a los estados del agente o más concretamente a la clase **AgentStates**, no requiere de una explicación a mayores, ya que realmente se ha implementado como base para el *sprint* siguiente.

4.7.4 Pruebas

Nuevamente las implementaciones llevadas a cabo han sido a través de la consola de **Unity**. En este caso concreto se ha contemplado el manejo de los nodos dependiendo de la prioridad de los mismos y a su vez, el presenciar como las asignaciones respecto al límite impuesto a la constante mencionada con anterioridad, se gestionan. Además, se han realizado implementaciones básicas de los estados, creando a través de código los diccionarios de los mismos, a los cuales se les ha añadido valores para poder ser presentados por la consola. Comprobando así, el buen funcionamiento de los mismos.

4.8 Sprint 8

Respecto al desarrollo de este *sprint*, cabe resaltar que es uno de los cuales más tiempo de desarrollo ha conllevado y a causa de ello, uno de los que más historias de usuario comprende. Siendo así, se contemplan en esta etapa del desarrollo las correcciones del *sprint* anterior y todas aquellas correcciones necesarias que pudieran aparecer (en el caso de haber algún *bug* de la herramienta). Además, también se incluirán nuevas funcionalidades referentes a las historias de usuario pendientes del proyecto.

4.8.1 Análisis

Por tanto, este *sprint* comprende las correcciones de las historias de usuario pasadas así como también la inclusión de las nuevas, después de realizarse nuevamente, la correspondiente *sprint review*. A continuación se presentan todas las historias de usuario a desarrollar en este período, donde se continuará el desarrollo de la herramienta en base a las siguientes historias

de usuario: **HU-1, HU-3, HU-7 , HU-8 y HU-10** (véase tabla A.5) que se corresponden con los requisitos funcionales especificados a continuación (ver **Apéndice A.3** para más detalles).

- **HU-1**
 - **RF-1:** Añadir agente [GOAP](#).
- **HU-3:**
 - **RF-11:** Listar acciones de objetivo.
 - **RF-12:** Añadir nuevo objetivo.
 - **RF-13:** Modificar objetivo.
 - **RF-14:** Eliminar objetivo.
 - **RF-15:** Listar objetivos de agente.
- **HU-5:**
 - **RF-16:** Ver Árbol de Acciones
- **HU-7:** Cabe resaltar que de esta historia de usuario solo se estableció la base para poder implementarla en su totalidad en el *sprint* siguiente.
 - **RF-21:** Replanificación de acciones.
- **HU-8:**
 - **RF-22:** Gestionar Inventario.
- **HU-10:**
 - **RF-24:** Selección de objetivo.

4.8.2 Diseño

Este *sprint* tiene un amplio desarrollo, tanto en la parte de las funcionalidades como en todas aquella conjunto de interfaces de las diferentes historias de usuario a desarrollar. Por tanto, se intentará abarcar por partes, la explicación de cada uno de los diagramas de clases. Primeramente podrá observarse el diagrama de clases relacionado con las implementaciones y funcionalidades de la herramienta:

CAPÍTULO 4. DESARROLLO

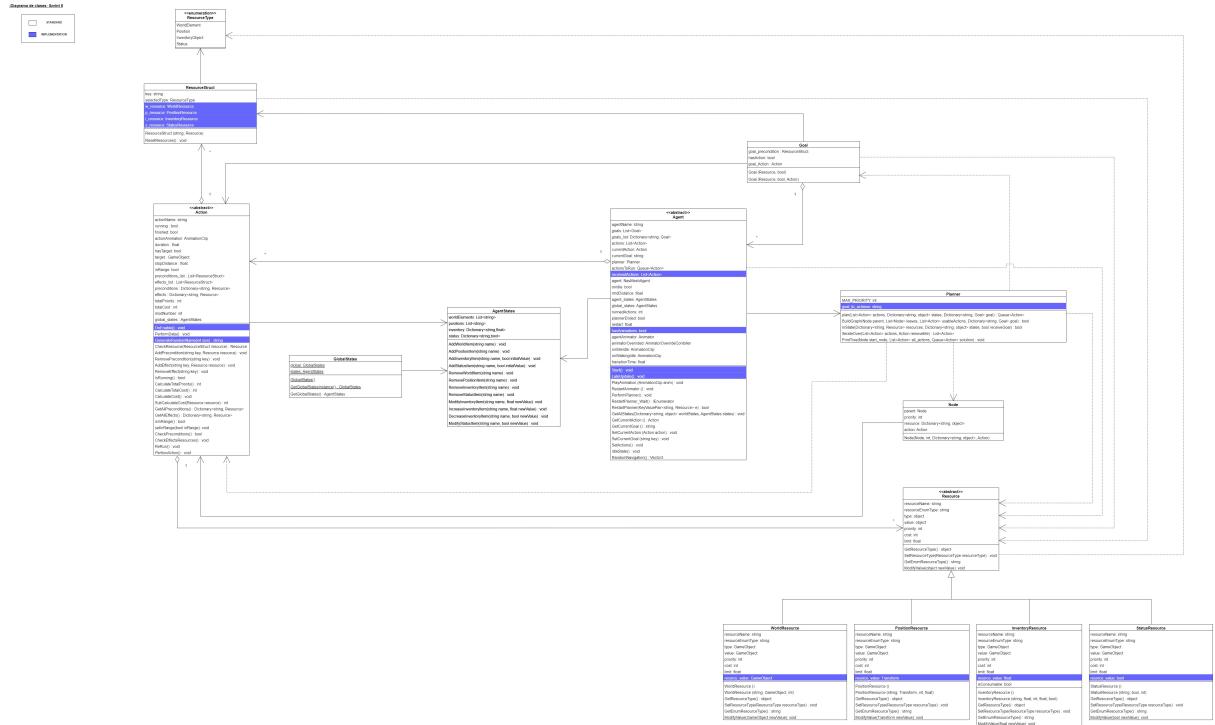


Figura 4.21: Diagrama de clases del Sprint 8

Una vez expuesto el diagrama de clases corresponde mencionar las funciones y variables correspondientes a las historias de usuario a desarrollar dentro del actual *sprint*, siendo así:

- **Action:** clase mencionada en anteriores *sprints*. Se corresponde a las acciones de los agentes.
 - **ActionAnimation:** en este parte del desarollo se contempla la integración de animaciones en 3D para los agentes (podrá verse la extensión sobre este mismo apartado en la subsección 4.8.3). Por ello, se necesita de esta variable del tipo **AnimationClip** para poder usar las respectivas animaciones a la hora de ejecutar una acción. Cabe destacar que este tipo de variable forma parte del motor de animaciones de **Unity**: **motion** y más concretamente del componente **Animation** [37].
 - **Duration:** duración de la animación. El usuario puede introducir un valor y en caso de ser superior al cálculo de la duración de la animación, esta se repetirá en bucle.
 - **StopDistance:** en relación al movimiento de los agentes en el mapa, distancia a la que como máximo, pueden acercarse.
 - **TotalCost:** será el sumatorio de los costes de cada una de las precondiciones y efectos. Se detallará el cálculo de estos en el apartado 4.8.3 .

- **modNumber**: variable auxiliar utilizada para el cálculo de los costes. Permite realizar divisiones a partir de un módulo e incrementar el mismo si fuera necesario.
 - **Global_states**: variable que hereda de la clase **AgentStates**, será el inventario local de cada uno de los agentes, es decir, los estados de cada uno o conjunto de recursos en un momento dado.
 - **CalculateTotalCost()**: permite calcular el coste total de la acción en particular, esta función es la que realiza el sumatorio.
 - **CalculateCost()**: asigna los costes a cada una de las precondiciones y efectos después de realizar los cálculos correspondientes.
 - **SubCalculateCost()**: función que auxiliar a la anterior que se encarga de calcular el coste real de cada recurso (precondición o efecto), esto es, calcular las distancias respectivas, cantidad de elementos del inventario, ... y en función de esto, asignar un coste específico.
 - **ReRun()**: función que permite aplicar cambios sobre los recursos en tiempo de ejecución si fuera necesario.
 - **PerformAction()**: función de vital importancia, ya que se mantiene como una función abstracta con el objetivo de que sea implementada por todas sus clases herederas a través del **patrón plantilla**. En esta función es donde se implementará el comportamiento de la acción en sí mismo.
- **GlobalStates**: esta clase se corresponde a los estados globales a los cuales podrán acceder todos los agentes. En este caso se presenta un patrón **singleton**, el cual consiste en que al momento de ejecutar **Unity** esta clase creará una única instancia, inmodificable por terceros, que permita que todos los agentes accedan y modifiquen los respectivos estados con tal de usarlos para el planificador de acciones.
 - **Global**: será la instancia única que podrán usar los agentes a posteriori para hacer referencia al inventario global.
 - **Global**: variable que hará referencia a los estados locales, es decir a la clase **AgentStates** y que servirá para poder indicarle a la clase en cuestión, como gestionar el inventario.
 - **GetGlobalStatesInstance()**: permite a terceros referenciar a la instancia creada para el inventario local.
- **AgentStates**: como se mencionó en el *sprint* anterior, esta clase permitirá implementar los inventarios o conjuntos de recursos de manera local a los agentes. Se implementan los respectivos métodos que permitirán modificar, añadir nuevos elementos o eliminar los existentes de los diversos diccionarios de estados creados con anterioridad.

- **Agent**: clase genérica que permitirá identificar a cada uno de los agentes y que mediante un **patrón plantilla** se pretende que las clases sucesoras implementen los comportamientos deseados.
 - **CurrentAction**: variable que permite, a través del planificador de acciones, saber que acción se está ejecutando en el momento actual.
 - **CurrentGoal**: permitirá conocer el objetivo actual a alcanzar por el planificador de acciones.
 - **ActionsToRun**: será la ya mencionada **cola** de acciones que se recibirá como resultado del *planner*.
 - **ReceivedActions**: servirá como variable auxiliar para cuestiones de la interfaz gráfica, más concretamente, permitirá saber al *Action tree* implementado en este *sprint*, el conjunto de acciones que se ejecutarán.
 - **Agent**: permitirá implementar el movimiento en los agentes. Es una variable de tipo **NavMesh**, lo que implica que será necesario realizar la correspondiente función de "bake" o lo que es lo mismo, habilitar la **scene** para que pueda ser navegable por este tipo de componentes (para saber como realizar esta última función, véase el manual de usuario especificado en el **Apéndice E**).
 - **OnIdle**: variable booleana que permitirá a los agentes establecer dos estados diferentes. Y utilizará la función **IdleStates()** para ejecutar las pertinentes animaciones, así como también, poder reiniciar aquellos caminos que tenga asignado el agente (posiciones a las que navegar con el *navmesh* respectivamente).
 - * En caso de habilitar esta opción, se hará que el agente mantenga la posición y en caso de que la hubiere, ejecutar la animación correspondiente.
 - * En caso de deshabilitar esta opción, se hará que el agente navegue por el mundo designado de manera aleatoria, a una distancia establecida por la función **RandomNavigation()** y asignando ese valor a la variable **RmdDistance**. Y en caso de que la hubiere, se ejecutaría la animación correspondiente.
 - **Agent_states** y **Global_states**: se corresponden al inventario o conjunto de recursos local y la instancia del inventario global respectivamente.
 - **Restart**: valor booleano que indicará cuando se puede reiniciar el *planner*.
 - **HasAnimations**: variable booleana utilizada para indicar si la acción en concreto tendrá o no acciones, en caso de un valor afirmativo se desplegarán en la interfaz del agente las variables siguientes:
 - * **AgentAnimator**: el cual se corresponde a una **Máquina de estados** que permitirá ejecutar las respectivas animaciones de cada una de las acciones, será detallado en el apartado **4.8.3**.

- * **AnimatorOverrided**: permite el intercambio de animaciones en los estados del **Animator** [38]. Esta variable no será desplegada en la interfaz gráfica.
- * **onSiteIdle** y **onWalkingIdle**: serán **AnimationsClips** correspondientes a las animaciones que se pueden ejecutar dependiendo del valor de la variable **OnIdle**.
- **PlayAnimation(AnimationClip anim)**: permite que mediante el uso de un **AnimationOverrideController** se puedan ejecutar las animaciones de las respectivas acciones, pudiendo modificar la que está actualmente en el estado actual de la **Máquina de estados** o **Animator**.
- **PerformPlanner()**: función encargada de llamar a la clase **Planner** y crear o reiniciar un plan en los casos que se requiera.
- **RestartPlanner_Wait()**: antes de reiniciar el *planner*, realiza una espera de unos cuantos segundos con la finalidad de evitar posibles errores en el reinicio del mismo. Esto se realiza mediante el uso de corrotinas, más concretamente la función *StartCoroutine()* [39] y la combinación de la misma con **IEnumerator** [40] el cual permite pausar una iteración determinada.
- **RestartPlanner(KeyValuePair<string, Resource> e)**: se comprueba si el estado actual cumple con los efectos de la acción que se está ejecutando, en caso de no ser así, se reinicia el *planner* y se repite todo el conjunto de acciones hasta que se consiga.
- **Planner**: comentado en *sprints* anteriores, hace referencia al planificador de acciones.
 - Cabe mencionar sobre esta clase el cambio de variables respecto a los *sprints* anteriores, donde anteriormente se recibían listas o diccionarios del tipo **Resource** y en este caso concreto se admite el tipo propio del lenguaje **object**. Esto está hecho con la finalidad de poder admitir también los estados, ya sean inventarios locales o globales, para poder compararlos a la hora de contrastar las precondiciones y efectos a cumplir, esto no excluye el seguir pudiendo hacer los tipos **Resource** como parámetros de las funciones. Ocurre lo mismo en la clase **Node** donde todas aquellas referencias a los tipo **Resource** han sido cambiadas.
- **Goal**: permite la implementación de los objetivos que desea cumplir el agente, así como también la inclusión de animaciones propias de los *goals*, que serán ejecutadas una vez se alcancen los mismos.
 - **HasAction**: en caso de que el objetivo actual tenga una acción, permite desplegar en la interfaz la variable definida a continuación.

- **Goal_Action**: acción que ejecutará un agente en caso de tenerla y haber llegado a cumplir el objetivo.

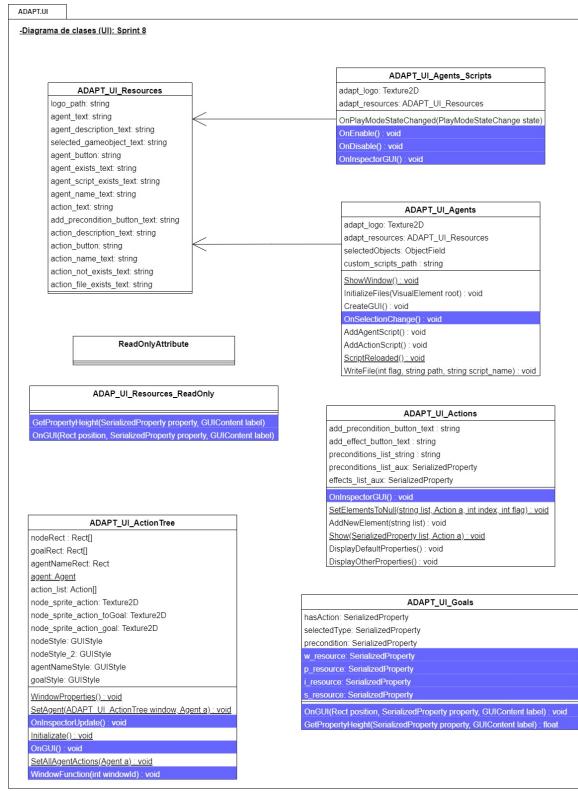


Figura 4.22: Diagrama de clases del Sprint 8, interfaz gráfica

En cuanto al diagrama de clases referente a la interfaz gráfica, en este *sprint* concreto se realizan múltiples modificaciones, ya que se ha incorporado la misma casi en su totalidad a lo largo de este período de desarrollo. Esto último es debido al haber llegado a un punto del desarrollo donde ya se puede contar prácticamente con la totalidad del sistema **GOAP** y permitir así la implementación de las funciones mediante un entorno gráfico de manera mucho más sencilla. De las determinadas clases y variables se destacan:

- **ADAPT UI Agents y ADAPT UI Agents Scripts**: las cuales permiten las respectivas implementaciones de las interfaces relacionadas con el menú de los agentes. Así como también la modificación de la visualización en cuanto al **Unity Inspector**, de los respectivos scripts de los agentes agregados a los gameobject correspondientes.
- **ADAPT UI Actions**: permitirá implementar la interfaz relacionada a la disposición de los elementos de las acciones en el **Unity Inspector**.

- **ADAPT UI Goals**: de manera similar a las acciones, esta clase se encargará de implementar la disposición de los elementos de la clase **Goal**, es decir, de los objetivos de un agente. Y permitir que estos se puedan visualizar de manera similar a como lo haría un recurso.
- **ADAPT UI ActionTree**: permite implementar el árbol de nodos o el conjunto de acciones a ejecutar por el agente. Se utilizan los denominados elementos **Rect** [41] para poder implementar la apariencia de cada una de las acciones simulando ser un nodo de un árbol de navegación. Caben destacar las funciones propias de **Unity** utilizadas para la implementación del apartado gráfico y utilizadas en varias de las determinadas clases. Usualmente, para la inicialización de diccionarios o variables que se usarán a posteriori, o controlar que elementos se están visualizando en el momento, siendo así:
 - **OnEnable()** y **OnDisable()**: se ejecutarán siempre y cuando la interfaz deseada esté mostrada en pantalla o no, respectivamente
 - **OnSelectionChange()**: en caso de seleccionar un elemento distinto del ya seleccionado en la jerarquía de elementos de **unity**, también llamada **hierarchy** [42], se ejecutará la función.
 - **OnInspectorUpdate()**, **OnInspectorGUI()** y **OnGUI()**: trabajan de manera similar, ya sea, llamando a la función cuando se actualiza el **Unity Inspector** por alguna causa, o simplemente cuando este aparece por primera vez en pantalla, respectivamente. Por otra parte, **OnGUI()** se ejecutará siempre que la ventana esté en pantalla, de manera similar al **Update()**.

4.8.3 Implementación

Cálculo de heurística

Como se comentaba anteriormente en la explicación inicial de la heurística elegida (véase 4.7.3). Se seguirá con la justificación de su elección, así como también visualizar a continuación, las correcciones que la misma ha sufrido. Por tanto y en base al siguiente código:

```
1 Node node = new Node(parent, ((a.totalPriority - a.totalCost) <= 0)
2     ? 1 : a.totalPriority - a.totalCost, currentState, a);
3
4     if (node.priority > MAX_PRIORITY)
5     {
6         node.priority = MAX_PRIORITY;
7     }
```

Se puede obviar nuevamente la variable *node* por las razones explicadas en capítulos anteriores. Por tanto la parte de código a explicar será: $((a.totalPriority - a.totalCost) <= 0) ? 1 :$

$a.totalPriority - a.totalCost$. Ya que esta es la heurística aplicada para poder obtener el camino con mayor prioridad. Básicamente consiste en tomar todas aquellas prioridades asignadas a las precondiciones y efectos, las cuales están asignadas de una manera completamente a elección del usuario, el sumatorio de las mismas se realiza y se obtiene con la variable **totalPriority** y por otra parte se realiza el cálculo de los costes y se almacena en la variable **totalCost**, el valor de ambas variables se resta y se comprueba que en ningún momento el valor resultante sea menor que 0 o mayor que la constante **MAX_PRIORITY**, la cual puede ajustar el usuario en caso de necesidad. La parte más importante de toda esta asignación de heurística es el cálculo de los costes, porque se hace de una manera dinámica teniendo en cuenta el tipo del recurso del que se va a realizar el cálculo, es decir:

- **WorldElement**: se toma en consideración la variable *value* del tipo de recurso en cuestión y se comprueba la distancia de ese mismo objeto respecto al agente. Esta distancia será el costo total de ese efecto en concreto o precondición.
- **Position**: funciona de manera similar a los WorldElement, en función de la distancia del agente concreto hasta el punto destino, se calcula una distancia y se asigna como coste.
- **Inventory**: este es un caso particular ya que trabajará en función de la cantidad de elementos del campo *value*. Es decir, limitará el coste máximo que una cantidad exagerada de recursos podría dar como coste, a cuanto mayor cantidad, se aplica mayor número de una operación de división. Es decir, en el caso de tener un valor de una cantidad de "1000" se dividiría por "100" teniendo una prioridad de "10", en el caso de una cantidad de "10000" se dividiría por "1000" y así de forma reiterada siempre y cuando la cantidad de elementos en el inventario supere un umbral.
- **Status**: al ser elementos abstractos o consideraciones subjetivas del mundo, se ha añadido por defecto un coste igual a "1".

Se ha de tener en cuenta que al tener distintas prioridades para cada una de las acciones, puede llegar un momento el cual se cumplan las precondiciones de una acción con más prioridad de la que se está ejecutando, para ello, el *planner* debe poder reiniciarse cada cierto tiempo con el objetivo de revisar estas posibles acciones prioritarias (a ello que exista la función *RestartPlanner()* y *derivadas de la misma*). Este reinicio puede ocurrir de varias maneras: ya sea al haber ejecutado un número máximo de acciones en cadena y por tanto, se necesite reiniciar para evitar posibles acciones prioritarias entrantes. O cuando se ha conseguido alcanzar una meta y el *planner* necesita buscar algún nuevo objetivo.

Animaciones de los agentes

Como se mencionó con anterioridad, en este *sprint* se llevó a cabo la implementación de las animaciones para aquellos agentes que posean un modelo en 3D. Esto se ha realizado mediante el uso de la clase **Animator** de **Unity**. La cual es una implementación de una **Máquina de estados** al uso. Para poder hacer que un agente en concreto pueda usar una animación debe añadirse el componente especial del **Animator**, para ello, se utiliza el propio **Unity Inspector** y el resultado tendría que ser algo similar a lo mostrado en las siguientes imágenes:



Figura 4.23: Añadir componente animator

Una vez añadido el componente a aquel agente que se desee que ejecute animaciones, se deben añadir las mismas a las variables *ActionAnimation* correspondientes. Estas serán ejecutadas según la ya mencionada máquina de estados, la cual estará formada por dos de ellos: **Idle** que es aquel estado en el que siempre se estará por defecto y **RunnableAction** que es aquel en el que solo se entrará cuando haya una animación que ejecutar y con ello, una acción posible para poder llevar a cabo, se muestra a continuación el animator de la herramienta desarrollada:

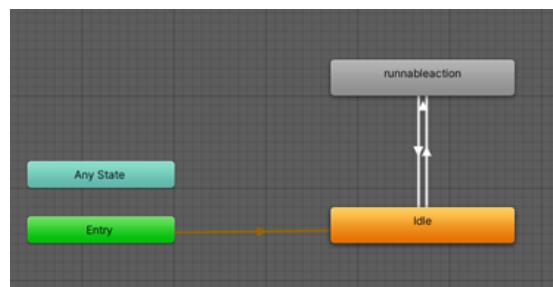


Figura 4.24: Animator usado para las animaciones

Action Tree

En base a otra de las historias de usuario a implementar en este *sprint*, se ha llevado a cabo el desarrollo de la pantalla de visualización referente a el plan, o lo que es lo mismo,

el conjunto de acciones a ejecutar. Puede verse el planificador de acciones representado en las imágenes referentes al **Apéndice D**. Para poder representar las acciones y que con ello, se asemeje a un árbol de nodos, se han implementado con una apariencia de los mismos, además, el color de cada uno de estos nodos indica diferentes situaciones:

- **Azul**: para aquellas acciones que no han podido añadirse a un plan; o que por el contrario, cuando todas presentan este mismo color, no se ha inicializado aún el *planner* en tiempo de ejecución.
 - **Verde**: aquellas acciones que forman parte de un plan para poder llegar al objetivo.
 - **Rojo**: para los *goals* que el agente puede alcanzar, pueden aparecer más en tiempo de ejecución.

Cabe resaltar que junto a las acciones también se han representado los respectivos valores de las prioridades, así como su costes. Ambos valores calculados en el momento de ejecución del videojuego o programa.

4.8.4 Pruebas

Las pruebas de este *sprint* constan de varias partes:

- Por una lado, las pruebas relacionadas con el planificador de acciones y las animaciones. Para comprobar ambas situaciones se ha hecho uso de modelados 3D de libre uso encontrados en la web [43]. Pudiendo ver una representación de los mismos y la ejecución de sus respectivas acciones en la siguiente imagen:

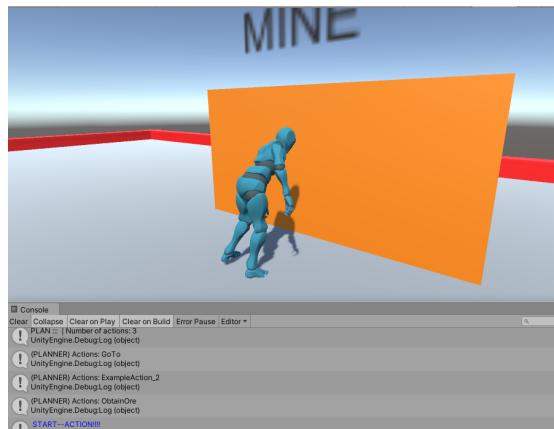


Figura 4.25: Agente con modelado 3D ejecutando una animación

Donde puede verse como un agente con su respectivo modelado llega a ejecutar la acción y su correspondiente animación, además, el movimiento del mismo a través del

mapeado de la escena se realiza mediante el componente *navmesh* así que se ha podido corroborar el funcionamiento del mismo. Nuevamente se ha utilizado la consola de Unity para comprobar la correcta salida de las acciones así como la modificación de los estados.

- Y por otra parte, todas aquellas implementaciones relacionadas con la interfaz gráfica se han probado mediante la propia visualización de las mismas. Es decir, mediante la comprobación de que los elementos se mostraban correctamente en la interfaz, así como también, comprobar el correcto funcionamiento de los botones, desplegables, ... y todo elemento gráfico que permitiera mostrar de alguna manera los elementos de la herramienta. Además, se implementaron diversas funciones que permitieran la creación de cadenas de texto completamente aleatorias (función `GenerateRandomName()` en la clase **Action**) para impedir que el usuario final dejará valores nulos en las respectivos componentes que forman la herramienta. Entre las diversas implementaciones gráfica se destaca la ventana correspondiente al planificador de acciones y el menú principal de la herramienta, junto a la comprobación del funcionamiento de crear agentes y acciones a la par que añadir los mismo a un `gameObject` seleccionado en la `hierarchy`. Todas estas pantallas, así como la solución final, pueden verse en el **Apéndice D**.

4.9 Sprint 9

Este *sprint* comprende la implementación de las últimas historias de usuario y sin apenas correcciones respecto al resto de las mismas, al haber sido implementadas correctamente en el resto de *sprints*.

4.9.1 Análisis

Por tanto, en la fase de análisis se concretarán las historias de usuario a desarrollar en este período del desarrollo, más concretamente **HU-13**, **HU-14** (véase tabla A.5) que se corresponden con los requisitos funcionales especificados a continuación (ver **Apéndice A.3** para más detalles).

- **HU-13:**
 - **RF-3:** Añadir Acción predefinida.
- **HU-14:**
 - **RF-2:** Listar Acciones predefinidas.

CAPÍTULO 4. DESARROLLO

4.9.2 Diseño

A continuación se mostrarán los diagramas de clases referentes a las implementaciones realizadas en el presente *sprint*:

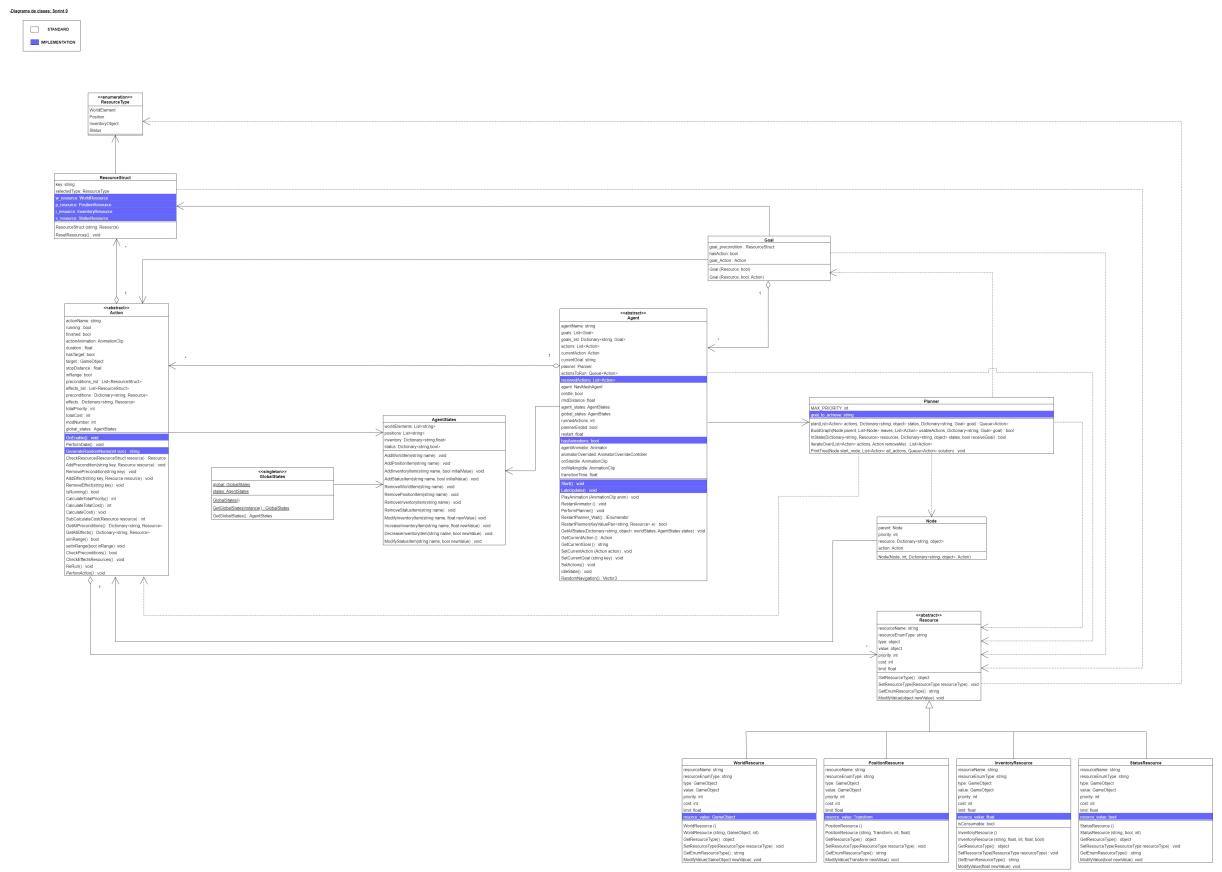


Figura 4.26: Diagrama de clases del Sprint 9

En el diagrama de clases anteriormente representado no surgen modificaciones relevantes respecto a los diagramas vistos con anterioridad en estas iteraciones, esto es debido a que las historias de usuario a desarrollar en este *sprint* son meramente estéticas. Por otra parte cabe mencionar, la finalización de los mockups al completo ya que, al existir ciertas limitaciones por parte del motor usado y vistas a lo largo del desarrollo de este proyecto, se han ido realizando las correcciones oportunas con la finalidad de poder ofrecer la mejor versión del mismo y de la interfaz, al usuario. Pueden visualizarse los mockups finales en el **Apéndice C**.

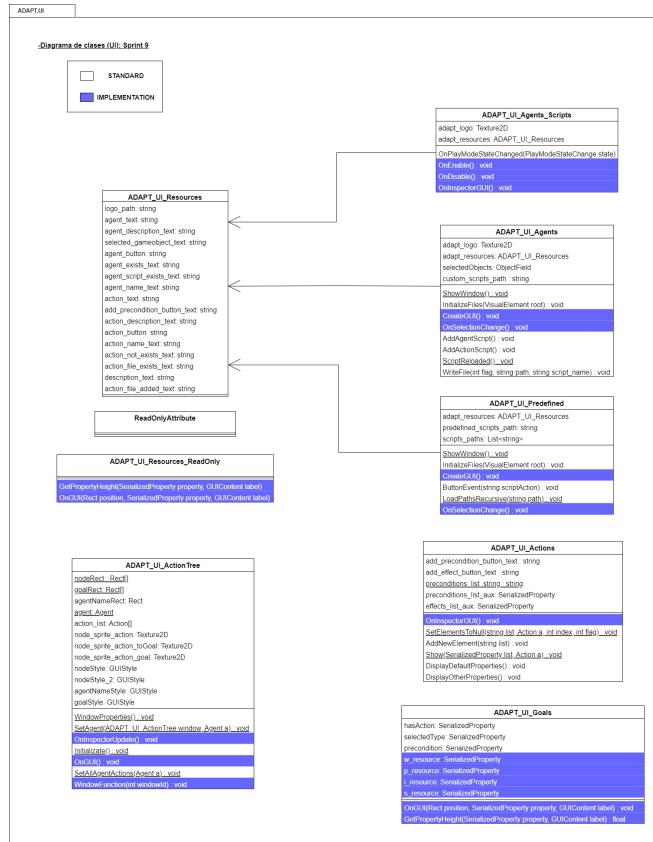


Figura 4.27: Diagrama de clases del Sprint 9, interfaz gráfica

Para este diagrama cabe destacar la clase **ADAPT UI Predefined**, la cual se encargará de poder llevar a cabo la correcta visualización del menú de acciones predefinidas, así como también listar aquellas acciones que se encuentre, por género de videojuego. Dentro de esta clase caben destacar las siguientes variables y funciones:

- **Scripts_paths**: será el conjunto de paths o básicamente, rutas de ficheros. Que permitan obtener aquellas carpetas con nombre de un género de videojuego, se guardará el nombre y dirección de las mismas con el fin de poder recorrerlas recursivamente.
- **LoadPathsRecursive(string, path)**: función que se encargará de la búsqueda recursiva de las acciones así como también, de poder mostrarlas en función del género de videojuego que posean.

4.9.3 Implementación

La implementación de esta funcionalidad consiste en tener una serie de directorios con los correspondientes nombres de géneros de videojuegos como se ve en la siguiente imagen:



Figura 4.28: Directorios de géneros de videojuegos

A partir de estos directorios se realiza una función que recorra todos y cada uno de ellos buscando los scripts o acciones predefinidas, al estar dentro de un directorio con un nombre de género particular ya pueden filtrarse de manera sencilla. Una vez listadas todas las acciones se muestran en el menú de acciones predefinidas, a partir del cual, y de manera similar al menú de acciones/agentes. Una vez se tenga un `gameObject` seleccionado en la [hierarchy](#) de Unity podrán añadirse pulsando el respectivo botón de la interfaz. Si el objeto seleccionado es un agente no habrá problemas y por tanto se añadirá como un componente más a su instancia. Estas ahora pasarán a formar parte de las acciones de un agente y pueden contemplarse a formar parte de un respectivo plan si se dieran las condiciones necesarias.

4.9.4 Pruebas

Las pruebas realizadas para esta interfaz gráfica han sido mediante la propia visualización de la misma, es decir, su correcto despliegue tanto de los textos, botones, ... como la posibilidad de añadir las acciones predefinidas a los agentes deseados. Y comprobar el correcto filtrado por género de videojuego de estas últimas. Puede verse la interfaz desarrollada relativa a las acciones predefinidas en el [Apéndice D](#).

Capítulo 5

Conclusiones

En base al desarrollo realizado se ha podido concluir que ha sido un proyecto el cual ha cumplido todos los objetivos establecidos en el apartado 1.2 así como también, todos los requisitos de su tabla correspondiente, la cual puede visualizarse en el [Apéndice: A.3](#). Es un proyecto que ha requerido estudio y entendimiento por parte del alumno ya que, un sistema [GOAP](#) comprende algunos conceptos que no suelen verse con frecuencia en un ámbito educativo. No obstante, ha sido un proceso lucrativo y educativo ya que ha permitido a este mismo el obtener conocimientos nuevos, así como también profundizar en muchos otros. Si bien la utilización de [Unity](#) no era algo nuevo para el alumno, si se ha podido ver afectado el desarrollo del proyecto de cara al entendimiento del sistema a implementar, debido a los conceptos específicos que este posee, y que requieren de una buena base sobre la que poder actuar a posterior, entre estos conceptos pueden destacarse algunos como: [pathfinding](#), [A*](#), [Máquina de estados](#), ... y como ha de llevarse la correcta implementación de los mismos al entorno de los videojuegos.

Teniendo en cuenta lo explicado, se considera que el desarrollo de la herramienta en cuanto a tiempo y costes ha sido satisfactorio, ya que la estimación realizada al inicio del proyecto ha sido ajustada y casi igual al resultado final obtenido, el cual puede observarse en el siguiente diagrama de Gantt:

CAPÍTULO 5. CONCLUSIONES

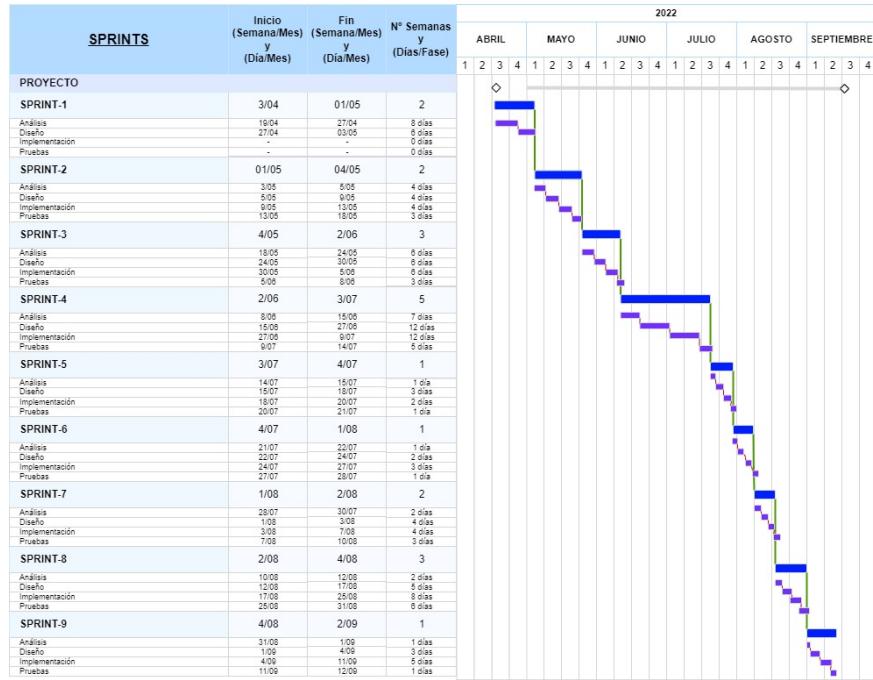


Figura 5.1: Diagrama de Gantt: Planificación Final

En base a las estimaciones realizadas, puede observarse que los costes iniciales también estimados al inicio del proyecto y más concretamente en la tabla 3.5 prácticamente no verán modificación alguna ya que, la estimación en cuanto a tiempos de desarrollo ha sido casi exacta. No obstante, a pesar de que el tiempo de desarrollo empleado haya sido de cierta manera, similar a lo que se tenía planeado en un primer momento, si han existido variaciones en cuanto a la duración de los *sprints* entre las cuales destacan:

- **Sprint 4:** el tiempo estimado inicial para este *sprint* era de aproximadamente 3 semanas (21 días) el cual ha visto postergada su finalización a un total de 5 semanas (36 días), esto ha sido debido a la complejidad a la hora de definir los recursos de una manera adecuada y la implementación correcta de los patrones de diseño.
- **Sprint 5:** el tiempo estimado inicial para este *sprint* era de aproximadamente 3 semanas (21 días) el cual ha visto postergada su finalización a un total de 1 semanas (7 días). Debido al gran conjunto de funcionalidades integradas en el *sprint* anterior, las implementaciones en este *sprint* se facilitaron así como también se ha visto reducido la duración prevista del mismo, debido a incluir modificaciones meramente estéticas.

Estos dos *sprint* han sido por tanto, los causantes de las modificaciones correspondientes en la duración semanal de los *sprints* a posteriori.

Capítulo 6

Trabajo Futuro

A pesar de que la herramienta desarrollada en el presente proyecto ha cumplido con todos los objetivos designados, en un futuro podrían tratar de mejorarse ciertos aspectos que permitieran aumentar la calidad de la misma, en este apartado se pondrán las posibles modificaciones que ayudarían a la herramienta en esto último.

6.1 Mejoras Futuras

A continuación se mencionarán algunas de las características que podrían incluirse en la herramienta de cara a su evolución:

- **Paquetes de expansión:** en relación a la integración de acciones predefinidas, añadir un conjunto extenso de las mismas orientado de cara a un género específico de videojuegos. Esto es, añadir conjuntos de acciones predefinidas en base a las necesidades de un videojuego muy concreto, para poder ahorrarle tiempo al usuario final.
- **Implementación de prototipos extensos:** a pesar de incluirse algunas demostraciones que prueban el correcto funcionamiento de la herramienta, la inclusión de prototipos más complejos podrían favorecer al usuario de cara a integrar la herramienta de la mejor forma posible, pudiendo sacar el máximo provecho de la misma al tener una base compleja a partir de la que guiarse.
- **Animator en 2D:** a pesar de implementarse un **Animator** que permite el correcto uso de animaciones 3D es cierto que, de cara a aquellos sistemas que usen animaciones de dos dimensiones tiene sus limitaciones, ya que estos últimos usan además sus propias físicas que poco o nada tienen que ver con las tridimensionales.
- **Multithreading:** permitir la integración de la concurrencia y hacer uso de la misma para mejorar el rendimiento de la herramienta, sobre todo relacionado a la interfaz gráfica ya que al ejecutar todo bajo un mismo hilo, se pierde rendimiento de la misma.

Apéndices

Apéndice A

Tablas de Requisitos e Historias de Usuario

En este apéndice se mostrarán las diferentes versiones de las tablas de requisitos, tanto las iniciales como sus posteriores modificaciones y correcciones hasta llegar a la versión final de la misma. Por otra parte, se expondrán las historias de usuario que se han llevado a cabo en el desarrollo como también las modificaciones que pudieran darse en las mismas.

A.1 Tabla de requisitos, versión inicial

Tabla A.1: Tabla de requisitos funcionales

ID	Nombre	Descripción
RF-1	Añadir Agente GOAP	Permitirá seleccionar un objeto del entorno y pulsando un botón, hacer que se convierta en un Agente GOAP añadiéndole los correspondientes scripts.
RF-2	Listar Acciones predefinidas	La herramienta tendrá una lista con todas aquellas acciones predefinidas que puedan añadirse a un agente, cada una con su respectivo género de videojuego.

..... (*continúa na páxina seguinte*)

Tabla A.1 – (*vén da páxina anterior*)

ID	Nombre	Descripción
RF-3	Añadir Acción predefinida	Teniendo seleccionado un agente, se podrán añadir acciones predefinidas al mismo al pulsar un botón.
RF-4	Añadir nueva precondición	El usuario podrá añadir nuevas precondiciones (que irán vinculadas a una acción), además tendrá que asignar varios parámetros como: elegir el nombre de la variable a usar, valor asignado, etc.
RF-5	Modificar Precondiciones	Se podrán modificar las precondiciones que ayudarán a los agentes a cumplir con un objetivo, es decir: añadir, eliminar, cambiar nombre de precondiciones o parámetros de las mismas (variables, componentes, ...).
RF-6	Eliminar Precondición	Se podrán eliminar precondiciones añadidas a un agente.
RF-7	Listar Precondiciones de Acción	Podrá verse una lista con todas las precondiciones añadidas a una acción de un agente en concreto, así como información derivada de las mismas.
RF-8	Añadir nueva Acción	El usuario podrá añadir nuevas acciones, además tendrá que asignar varios parámetros como: elegir el nombre de la acción, ...

..... (*continúa na páxina seguinte*)

Tabla A.1 – (*vén da páxina anterior*)

ID	Nombre	Descripción
RF-9	Modificar Acciones	Se podrán modificar las acciones que ayudarán a los agentes a cumplir con un objetivo, es decir: añadir, eliminar, cambiar nombre de acciones o implicaciones de las mismas (variables, componentes, ...), inclusive se podrán añadir o eliminar animaciones para modelos 3D que se ejecuten al momento de realizar la acción.
RF-10	Modificar Prioridad	Se podrán modificar la prioridad asociada a la ejecución de una acción.
RF-11	Eliminar Acción	Se podrán eliminar acciones añadidas a un agente.
RF-12	Listar Acciones de Objetivo	Podrá verse una lista con todas las acciones añadidas a un objetivo de un agente en concreto, así como información derivada de las mismas.
RF-13	Añadir Nuevo Objetivo	El usuario podrá añadir nuevos objetivos, además tendrá que asignar varios parámetros como: elegir un nombre para los mismos, asignarles acciones y precondiciones, ...
RF-14	Modificar Objetivo	Se podrán modificar los objetivos y sus respectivos parámetros (variables, componentes, ...). Así como añadir o eliminar acciones vinculadas a los mismos.

..... (*continúa na páxina seguinte*)

Tabla A.1 – (*vén da páxina anterior*)

ID	Nombre	Descripción
RF-15	Eliminar Objetivo	Se podrán eliminar objetivos añadidos a un agente.
RF-16	Listar Objetivos de Agente	Podrá verse una lista con todos los objetivos añadidos a un agente. Así como sus precondiciones y acciones.
RF-17	Ver Árbol de Acciones	Una vez seleccionado un agente, el usuario podrá pulsar un botón y ver el árbol de nodos que identificará la secuencia de acciones a tomar por el propio agente con sus respectivas precondiciones y el objetivo a alcanzar.

A.2 Tabla de requisitos, correcciones del Sprint 2

Tabla A.2: Tabla de requisitos funcionales

ID	Corrección
RF-5	Modificaciones en descripción: Se podrán modificar las precondiciones que ayudarán a los agentes a cumplir con un objetivo, es decir: añadir, eliminar, cambiar nombre de precondiciones o parámetros de las mismas (variables, componentes, ...). Además se podrán modificar los valores referentes a la prioridad de las precondiciones.

..... (*continúa na páxina seguinte*)

Tabla A.2 – (*vén da páxina anterior*)

ID	Corrección
RF-7	<p>Modificaciones en descripción: Podrá verse una lista con todas las precondiciones añadidas a una acción de un agente en concreto, así como información derivada de las mismas incluyendo la prioridad de cada una de las precondiciones.</p>
RF-10	<p>Requisito eliminado: Con ello, los identificadores de los requisitos posteriores pasarán a ser modificados al final de estas correcciones.</p>
RF-12	<p>Modificaciones en descripción: Podrá verse una lista con todas las acciones añadidas a un objetivo de un agente en concreto, así como información derivada de las mismas(incluyendo las animaciones vinculadas) además de poder visualizar la prioridad total, que consistirá en la suma de las prioridades de las precondiciones referentes a una acción.</p>
RF-18	<p>Añadido nuevo requisito: <i>Añadir Nuevo Efecto</i> Despues de eliminar el RF-10 este requisito pasará a tomar el id "RF-17". La descripción del mismo detallará: El usuario podrá añadir nuevos efectos que irán vinculados a una acción. Definirá un nombre para los mismos así como las implicaciones que pueden darse de los mismos: *en el inventario (con elementos consumibles o no), en el entorno o en los propios agentes.</p>
RF-19	<p>Añadido nuevo requisito: <i>Modificar Efecto</i> Despues de eliminar el RF-10 este requisito pasará a tomar el id "RF-18". La descripción del mismo detallará: Se podrán modificar los efectos: añadir, eliminar, cambiar nombre de efectos o parámetros de los mismos.</p>

..... (*continúa na páxina seguinte*)

Tabla A.2 – (vén da páxina anterior)

ID	Corrección
RF-20	<p>Añadido nuevo requisito: <i>Eliminar Efecto</i> Despues de eliminar el RF-10 este requisito pasará a tomar el id "RF-19". La descripción del mismo detallará: Se podrán modificar los efectos: Se podrán eliminar efectos añadidos a una acción determinada.</p>
RF-21	<p>Añadido nuevo requisito: <i>Capturar estado en-torno</i> Despues de eliminar el RF-10 este requisito pasará a tomar el id "RF-20". La descripción del mismo detallará: Los agentes debe poder capturar el entorno que le rodea, que elementos se encuentran disponibles y cuales no, a que lugares quiere llegar, si un objetivo se encuentra cerca del mismo, etc.</p>
RF-22	<p>Añadido nuevo requisito: <i>Seleccionar acción a ejecutar</i> Despues de eliminar el RF-10 este requisito pasará a tomar el id "RF-21". La descripción del mismo detallará: Los agentes podrán obtener a través del planificador y el algoritmo A* una secuencia de acciones que les lleve de la manera más óptima, a obtener el objetivo que desean.</p>
RF-23	<p>Añadido nuevo requisito: <i>Gestionar Inventario</i> Despues de eliminar el RF-10 este requisito pasará a tomar el id "RF-22". La descripción del mismo detallará: Todos los agentes poseerán un inventario propio, el cual podrán usar para visualizar los elementos que poseen y usarlos en caso de que fuera necesario para cumplir con ciertas precondiciones. Además, el inventario puede sufrir modificaciones causadas por la ejecución de acciones.</p>

..... (continúa na páxina seguinte)

Tabla A.2 – (vén da páxina anterior)

ID	Corrección
RF-24	Añadido nuevo requisito: <i>Conocimiento de ejecución de acciones</i> Despues de eliminar el RF-10 este requisito pasará a tomar el id "RF-23". La descripción del mismo detallará: Los agentes deben de ser capaces de poder tener en cuentas las precondiciones actuales para poder ejecutar una acción o no.
RF-25	Añadido nuevo requisito: <i>Selección de objetivo</i> Despues de eliminar el RF-10 este requisito pasará a tomar el id "RF-24". La descripción del mismo detallará: Los agentes deben poder seleccionar uno de los posibles objetivos de la lista definida por el usuario. En caso de no poder satisfacer uno en concreto, tratarán de satisfacer otro de la lista y así consecuentemente.

A.3 Tabla de requisitos, versión final

Tabla A.3: Tabla de requisitos funcionales

ID	Nombre	Descripción
RF-1	Añadir Agente GOAP	Permitirá seleccionar un objeto del entorno y pulsando un botón, hacer que se convierta en un Agente GOAP añadiéndole los correspondientes scripts.
RF-2	Listar Acciones predefinidas	La herramienta tendrá una lista con todas aquellas acciones predefinidas que puedan añadirse a un agente, cada una con su respectivo género de videojuego.

..... (continúa na páxina seguinte)

Tabla A.3 – (*vén da páxina anterior*)

ID	Nombre	Descripción
RF-3	Añadir Acción predefinida	Teniendo seleccionado un agente, se podrán añadir acciones predefinidas al mismo al pulsar un botón.
RF-4	Añadir nueva precondición	El usuario podrá añadir nuevas precondiciones (que irán vinculadas a una acción), además tendrá que asignar varios parámetros como: elegir el nombre de la variable a usar, valor asignado, etc.
RF-5	Modificar Precondiciones	Se podrán modificar las precondiciones que ayudarán a los agentes a cumplir con un objetivo, es decir: añadir, eliminar, cambiar nombre de precondiciones o parámetros de las mismas (variables, componentes, ...) Además se podrán modificar los valores referentes a la prioridad de las precondiciones.
RF-6	Eliminar Precondición	Se podrán eliminar precondiciones añadidas a un agente.
RF-7	Listar Precondiciones de Acción	Podrá verse una lista con todas las precondiciones añadidas a una acción de un agente en concreto, así como información derivada de las mismas incluyendo la prioridad de cada una de las precondiciones.
RF-8	Añadir nueva Acción	El usuario podrá añadir nuevas acciones, además tendrá que asignar varios parámetros como: elegir el nombre de la acción, ...

..... (*continúa na páxina seguinte*)

Tabla A.3 – (*vén da páxina anterior*)

ID	Nombre	Descripción
RF-9	Modificar Acciones	Se podrán modificar las acciones que ayudarán a los agentes a cumplir con un objetivo, es decir: añadir, eliminar, cambiar nombre de acciones o implicaciones de las mismas (variables, componentes, ...), inclusive se podrán añadir o eliminar animaciones para modelos 3D que se ejecuten al momento de realizar la acción.
RF-10	Eliminar Acción	Se podrán eliminar acciones añadidas a un agente.
RF-11	Listar Acciones de Objetivo	Podrá verse una lista con todas las acciones añadidas a un objetivo de un agente en concreto, así como información derivada de las mismas(incluyendo las animaciones vinculadas) además de poder visualizar la prioridad total, que consistirá en la suma de las prioridades de las precondiciones referentes a una acción.
RF-12	Añadir Nuevo Objetivo	El usuario podrá añadir nuevos objetivos, además tendrá que asignar varios parámetros como: elegir un nombre para los mismos, asignarles acciones y precondiciones, ...

..... (*continúa na páxina seguinte*)

Tabla A.3 – (*vén da páxina anterior*)

ID	Nombre	Descripción
RF-13	Modificar Objetivo	Se podrán modificar los objetivos y sus respectivos parámetros (variables, componentes, ...). Así como añadir o eliminar acciones vinculadas a los mismos.
RF-14	Eliminar Objetivo	Se podrán eliminar objetivos añadidos a un agente.
RF-15	Listar Objetivos de Agente	Podrá verse una lista con todos los objetivos añadidos a un agente. Así como sus precondiciones y acciones.
RF-16	Ver Árbol de Acciones	Una vez seleccionado un agente, el usuario podrá pulsar un botón y ver el árbol de nodos que identificará la secuencia de acciones a tomar por el propio agente con sus respectivas precondiciones y el objetivo a alcanzar.
RF-17	Añadir Nuevo Efecto	El usuario podrá añadir nuevos efectos que irán vinculados a una acción. Definirá un nombre para los mismos así como las implicaciones que pueden darse de los mismos: en el inventario (con elementos consumibles o no), en el entorno o en los propios agentes.
RF-18	Modificar Efecto	Se podrán modificar los efectos: añadir, eliminar, cambiar nombre de efectos o parámetros de las mismos.

..... (*continúa na páxina seguinte*)

Tabla A.3 – (*vén da páxina anterior*)

ID	Nombre	Descripción
RF-19	Eliminar Efecto	Se podrán modificar los efectos: Se podrán eliminar efectos añadidos a una acción determinada.
RF-20	Capturar estado entorno	Los agentes debe poder capturar el entorno que le rodea, que elementos se encuentran disponibles y cuales no, a que lugares quiere llegar, si un objetivo se encuentra cerca del mismo, etc.
RF-21	Replanificación de acciones	Mientras un agente ejecuta una acción pueden darse situaciones externas en las que aparezcan nuevas acciones con más prioridad que la actual, el objetivo del agente será poder seleccionar estas nuevas y diferentes acciones a ejecutar teniendo en cuenta la prioridad de las mismas.
RF-22	Gestionar Inventario	Todos los agentes poseerán un inventario propio, el cual podrán usar para visualizar los elementos que poseen y usarlos en caso de que fuera necesario para cumplir con ciertas precondiciones. Además, el inventario puede sufrir modificaciones causadas por la ejecución de acciones.
RF-23	Conocimiento de ejecución de acciones	Los agentes deben de ser capaces de poder tener en cuentas las precondiciones actuales para poder ejecutar una acción o no.

..... (*continúa na páxina seguinte*)

Tabla A.3 – (*vén da páxina anterior*)

ID	Nombre	Descripción
RF-24	Selección de objetivo	Los agentes deben poder seleccionar uno de los posibles objetivos de la lista definida por el usuario. En caso de no poder satisfacer uno en concreto, tratarán de satisfacer otro de la lista y así consecuentemente.

A.4 Historias de usuario, versión inicial

Tabla A.4: Tabla con las historias de usuario del Sprint 1

ID	Funcionalidad	Descripción
HU-1	Agentes	Como un usuario, quiero poder crear agentes: pudiendo seleccionar un elemento del entorno y convertirlo en un agente
HU-2	Agentes	Como un usuario, quiero poder modificar los parámetros respectivos a agentes ya creados.
HU-3	Agentes	Como un usuario, quiero poder eliminar a agentes ya creados.
HU-4	Acciones	Como un usuario, quiero poder crear nuevas acciones: añadiéndoles además una serie de precondiciones a la hora de crearlas.

..... (continúa na páxina seguinte)

Tabla A.4 – (*vén da páxina anterior*)

ID	Funcionalidad	Descripción
<i>HU-5</i>	Acciones	Como un usuario, quiero poder modificar acciones ya creadas, así como la posibilidad de añadirles o eliminarles precondiciones.
<i>HU-6</i>	Acciones	Como un usuario, quiero poder eliminar acciones ya creadas.
<i>HU-7</i>	Acciones	Como un usuario, quiero poder visualizar la lista de precondiciones que se tienen que dar para poder realizar una acción
<i>HU-8</i>	Acciones	Como un usuario, quiero poder visualizar una lista con todas las acciones añadidas a un agente concreto para llegar a cumplir un objetivo particular.
<i>HU-9</i>	Objetivos	Como un usuario, quiero poder crear nuevos objetivos.
<i>HU-10</i>	Objetivos	Como un usuario, quiero poder modificar objetivos ya creados, pudiendo modificar sus parámetros.
<i>HU-11</i>	Objetivos	Como un usuario, quiero poder eliminar objetivos ya creados.
<i>HU-12</i>	Objetivos	Como un usuario, quiero poder visualizar una lista con todos los objetivos añadidos a un agente concreto

..... (*continúa na páxina seguinte*)

Tabla A.4 – (*vén da páxina anterior*)

ID	Funcionalidad	Descripción
<i>HU-13</i>	Acciones predefinidas	Como un usuario, quiero poder asignar acciones predefinidas a los agentes.
<i>HU-14</i>	Acciones predefinidas	Como un usuario, quiero poder listar las acciones predefinidas así como su género de videojuego.
<i>HU-15</i>	Árbol de acciones	Como un usuario, quiero poder visualizar una secuencia de acciones de cada uno de los agentes, que indique el camino a tomar para llegar al objetivo.

A.5 Historias de usuario, versión final

Tabla A.5: Tabla con las historias de usuario del Sprint 2

ID	Funcionalidad	Descripción
<i>HU-1</i>	Usuario	Como un usuario, quiero poder crear, modificar y eliminar agentes
<i>HU-2</i>	Usuario	Como un usuario, quiero poder crear, listar, modificar y eliminar acciones. Así como sus respectivas precondiciones y efectos.
<i>HU-3</i>	Usuario	Como un usuario, quiero poder crear, listar, modificar y eliminar objetivos.

..... (*continúa na páxina seguinte*)

Tabla A.5 – (*vén da páxina anterior*)

ID	Actor	Descripción
<i>HU-4</i>	Usuario	Como un usuario, quiero poder crear y listar por género de videojuego acciones predefinidas.
<i>HU-5</i>	Usuario	Como un usuario, quiero poder visualizar una secuencia de acciones de cada uno de los agentes, que indique el camino a tomar para llegar al objetivo.
<i>HU-6</i>	Agente	Como un agente, quiero poder capturar el estado del entorno para posteriormente usar esta información en la toma de decisiones
<i>HU-7</i>	Agente	Como un agente, quiero poder seleccionar una acción a ejecutar de las varias posibles para poder llegar a cumplir un objetivo con la cadena de acciones dada.
<i>HU-8</i>	Agente	Como un agente, quiero poder gestionar el inventario para usar los elementos del mismo en determinadas ocasiones, pudiendo alterar los valores almacenados en el mismo
<i>HU-9</i>	Agente	Como un agente, quiero poder tener conocimiento de las precondiciones para saber si puedo llevar a cabo una acción o no.

..... (*continúa na páxina seguinte*)

Tabla A.5 – (*vén da páxina anterior*)

ID	Actor	Descripción
<i>HU-10</i>	Agente	Como un agente, quiero poder seleccionar un objetivo de las varios disponibles y tratar de ejecutar aquel que sea posible en el momento dado

Apéndice B

Mockups Iniciales

En este apéndice se recogen todos los mockups/bocetos correspondientes a la primera versión de la herramienta. Los cuales han sido realizados en las respectivas fases de diseño a lo largo de los sucesivos [sprints](#) de los que se compone este proyecto. Estos prototipos no se corresponden con la versión final de la herramienta, sino que sirvieron como base para las posteriores correcciones de la interfaz.

APÉNDICE B. MOCKUPS INICIALES

Initial_AGENT/ACTION_UI

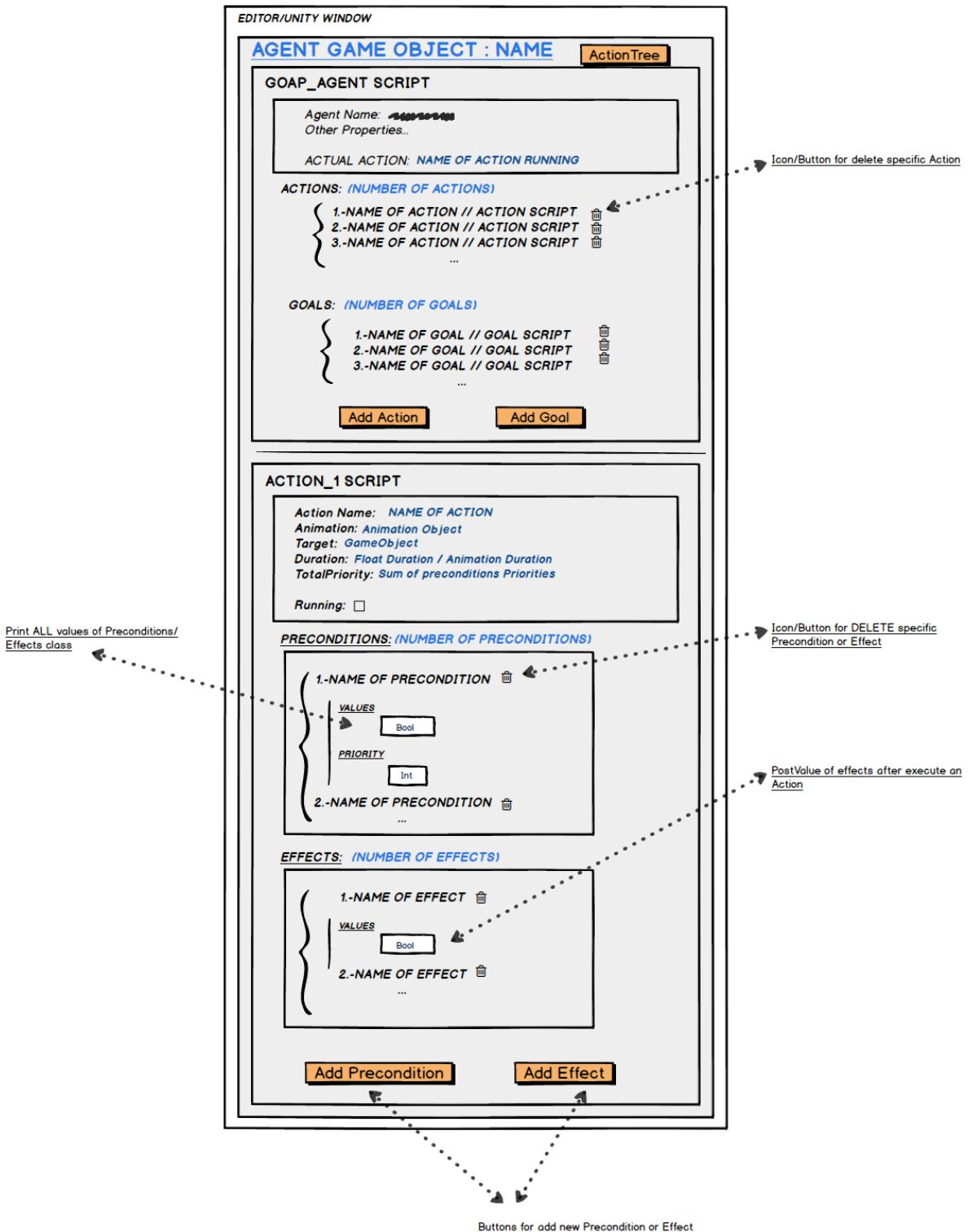


Figura B.1: Agente con acciones visualizado en el Unity Inspector

APÉNDICE B. MOCKUPS INICIALES

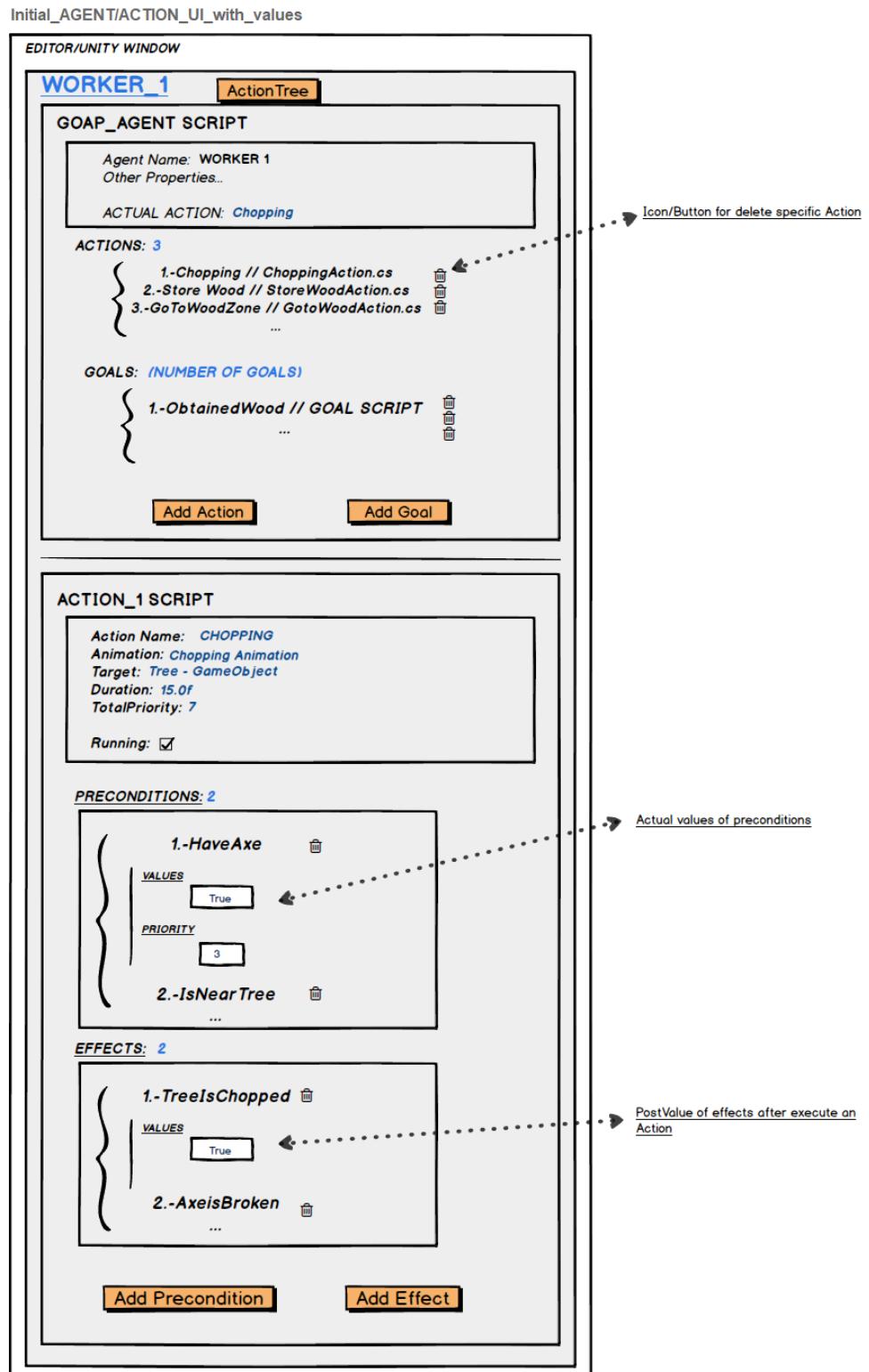


Figura B.2: Agente con acciones y valores cubiertos visualizado en el Unity Inspector

APÉNDICE B. MOCKUPS INICIALES

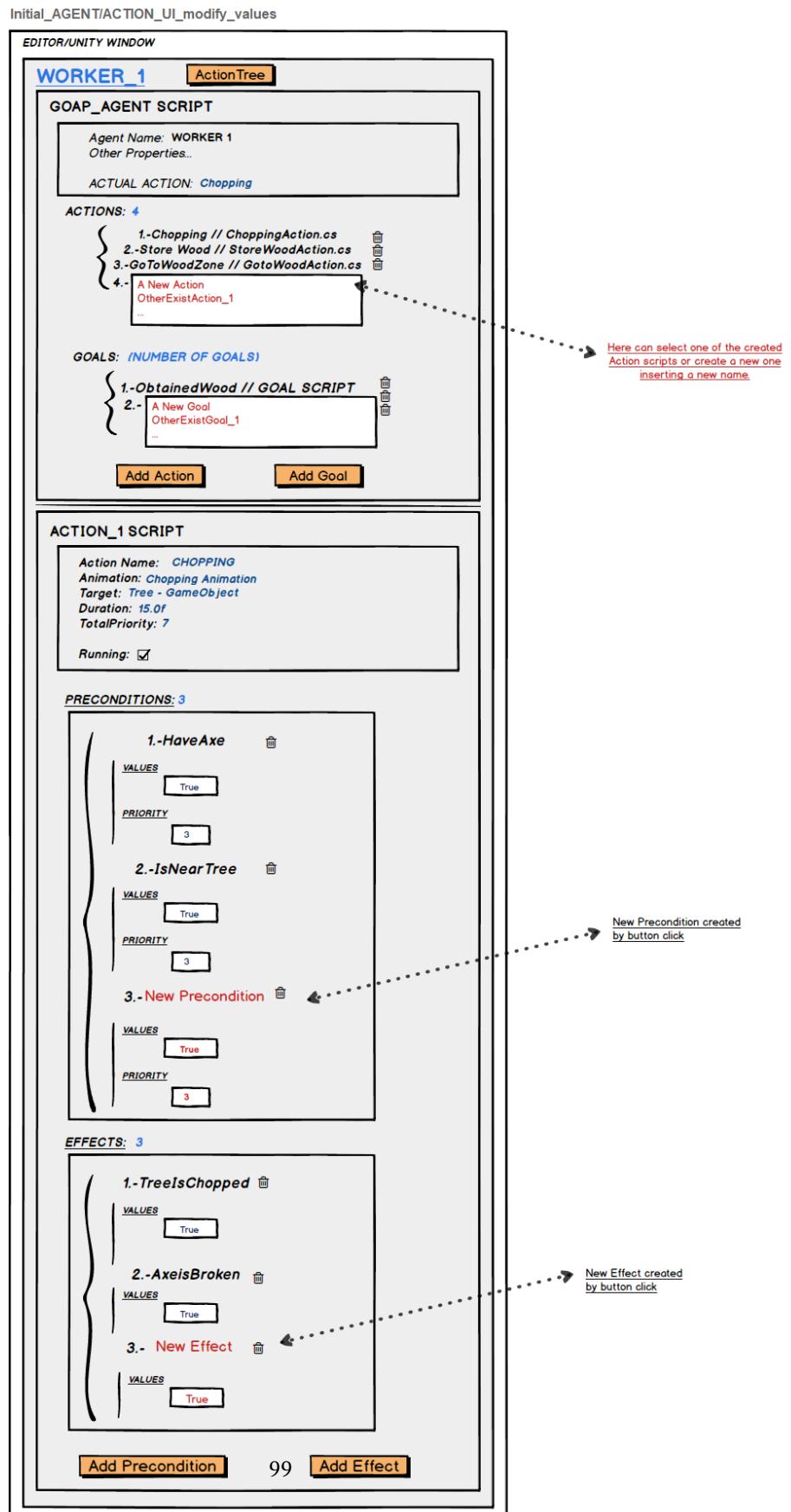


Figura B.3: Agente con acciones y valores modificados visualizado en el Unity Inspector

APÉNDICE B. MOCKUPS INICIALES

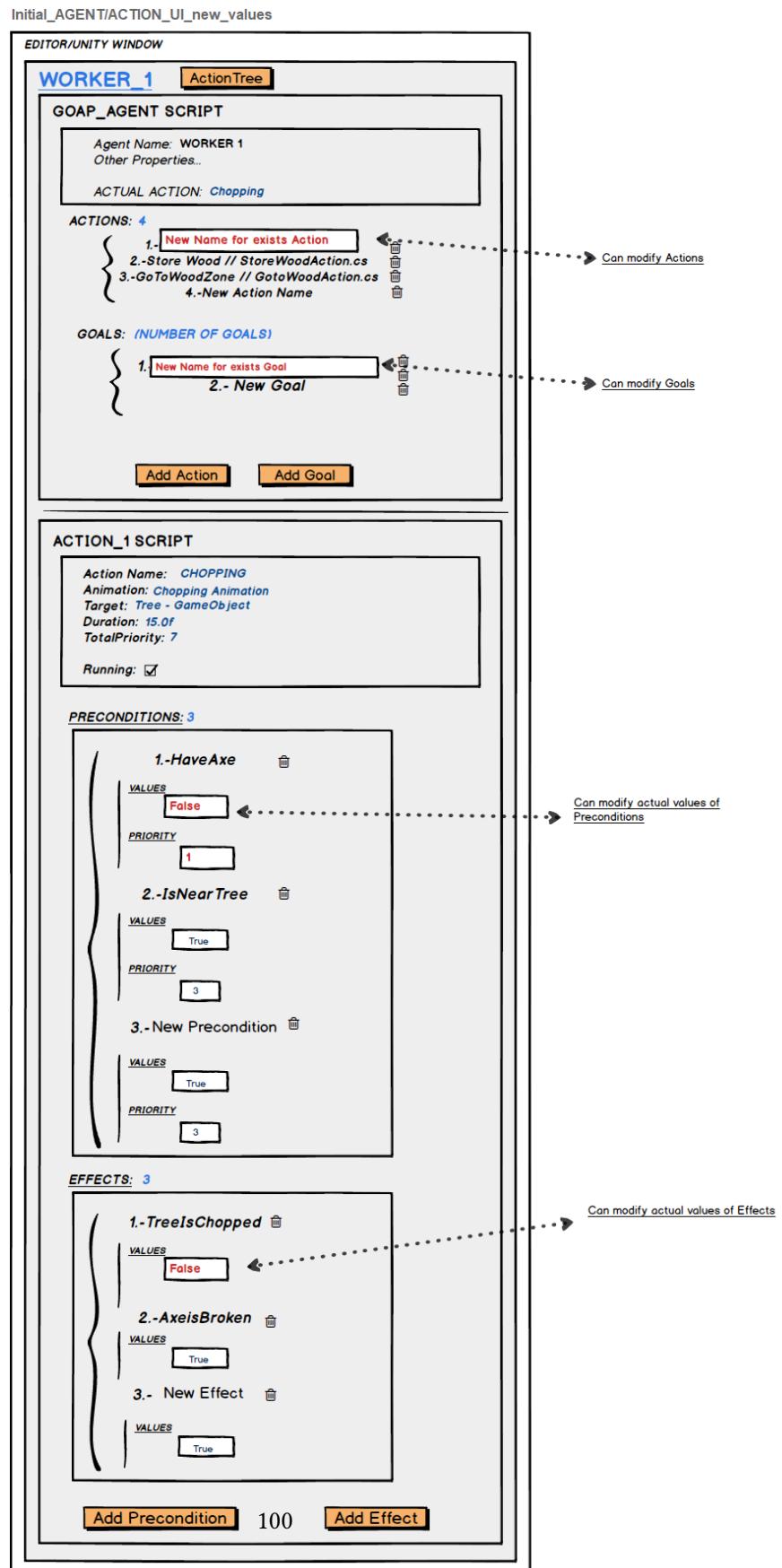


Figura B.4: Agente con acciones y nuevos valores visualizado en el Unity Inspector

APÉNDICE B. MOCKUPS INICIALES

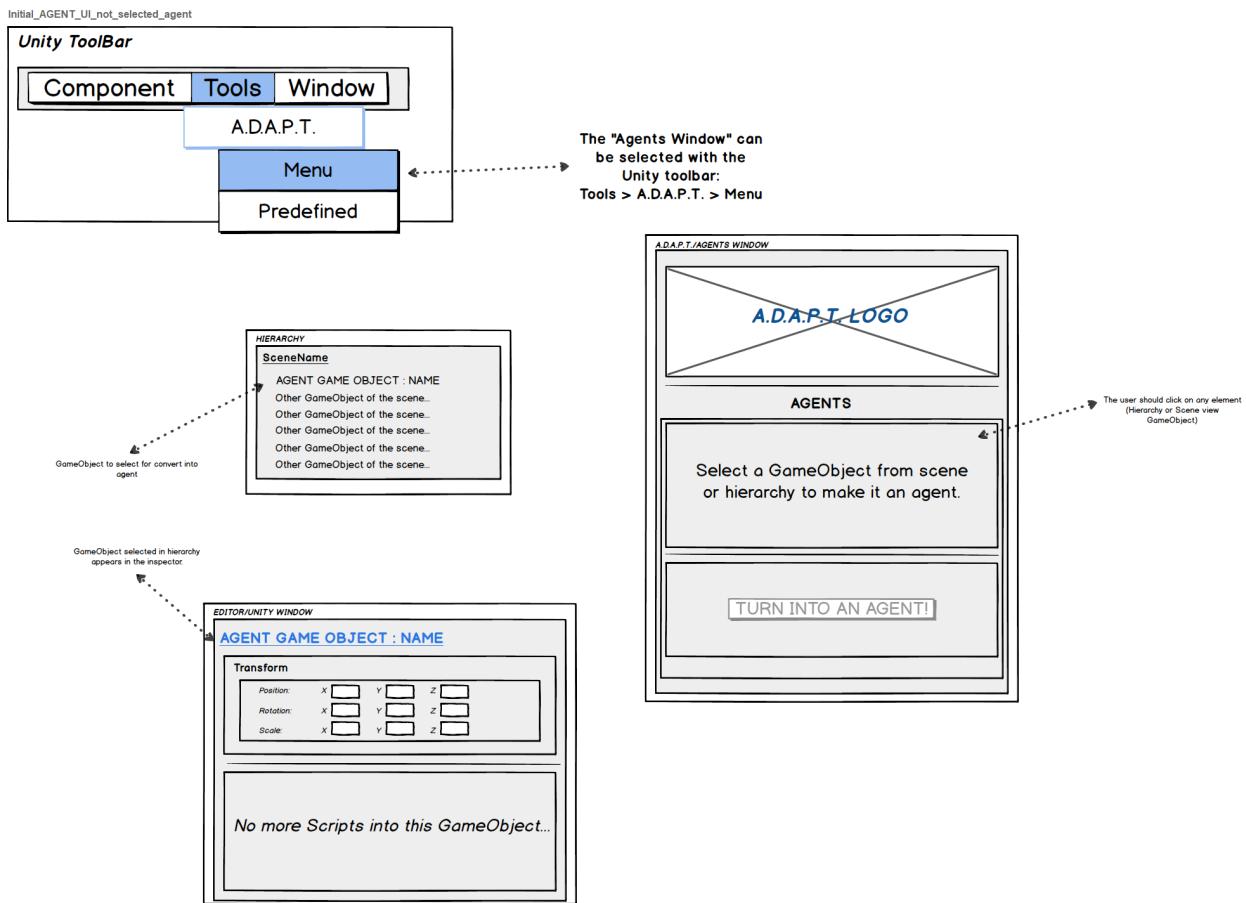


Figura B.5: Menú de la herramienta

APÉNDICE B. MOCKUPS INICIALES

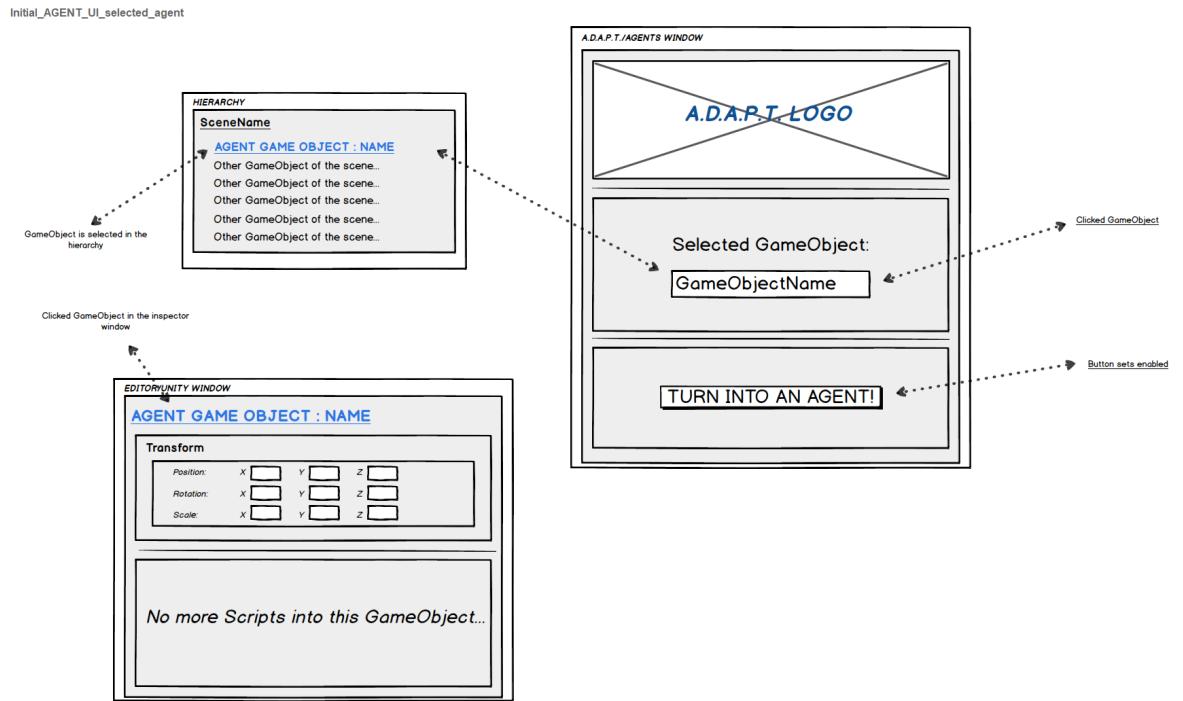


Figura B.6: Menú de la herramienta con objeto seleccionado en Unity

APÉNDICE B. MOCKUPS INICIALES

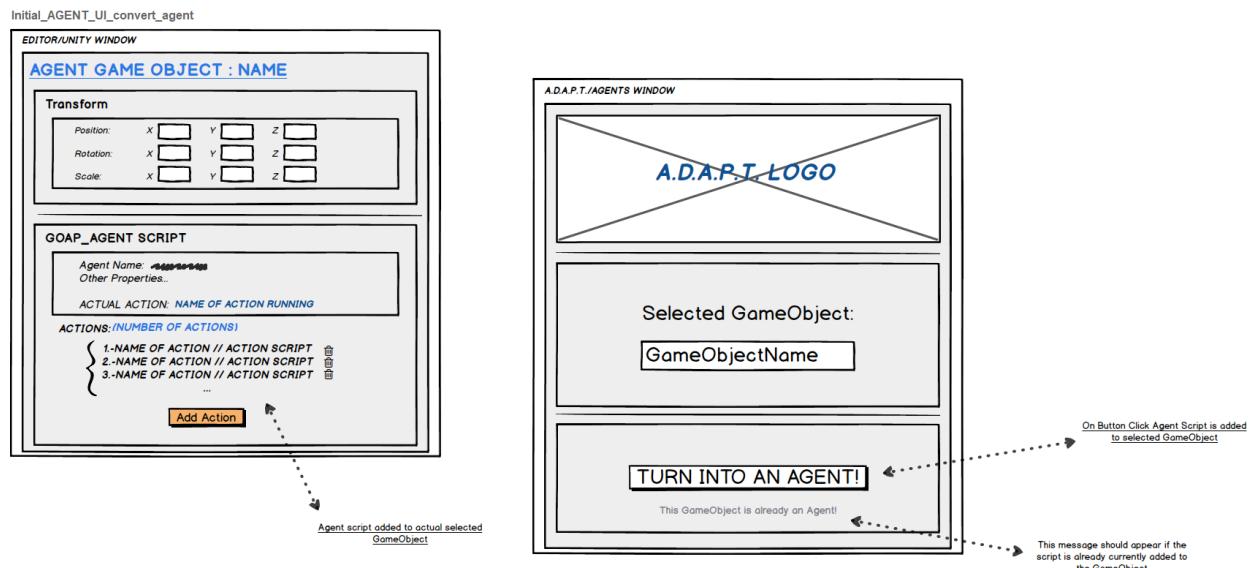


Figura B.7: Menú de la herramienta convirtiéndo a un objeto en agente

APÉNDICE B. MOCKUPS INICIALES

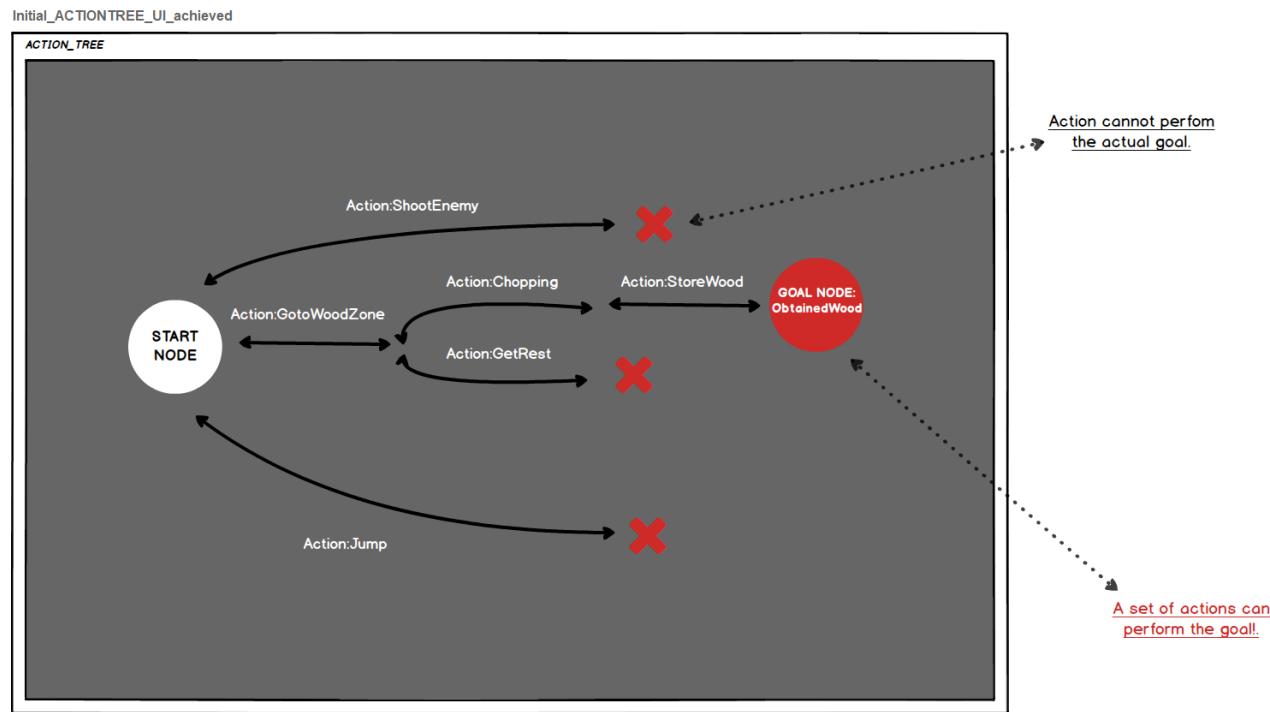


Figura B.8: Árbol de acciones

APÉNDICE B. MOCKUPS INICIALES

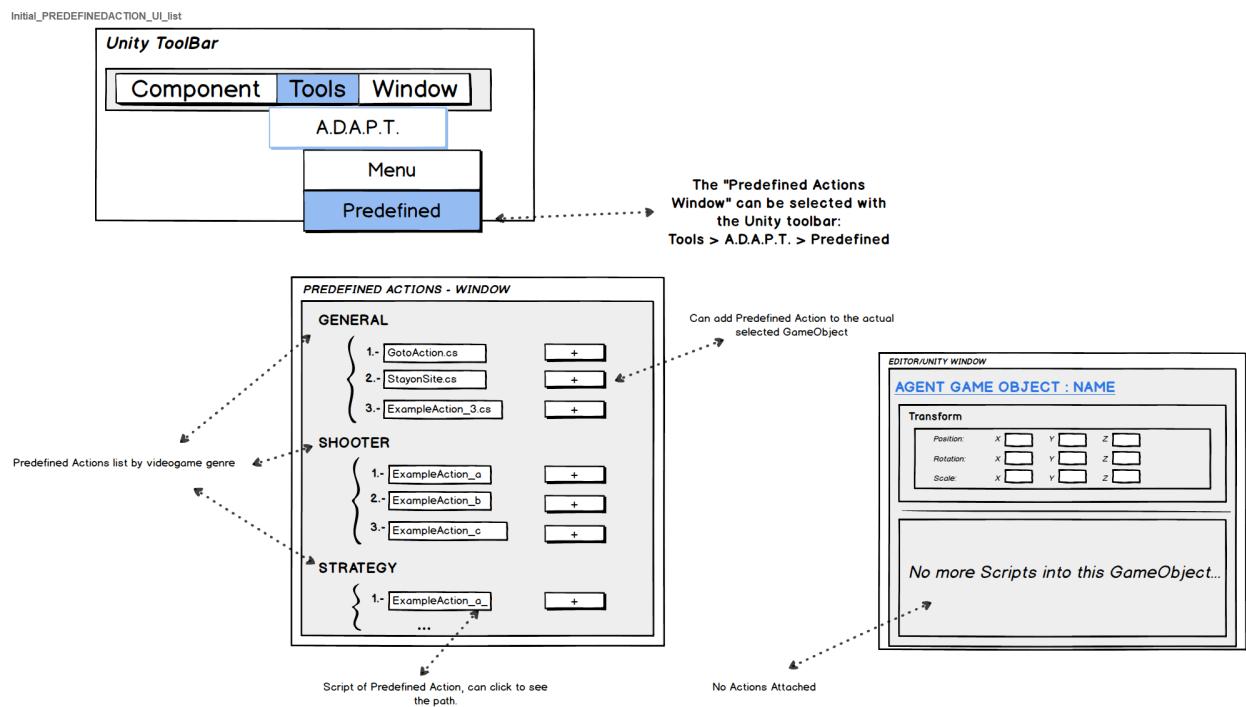


Figura B.9: Menú de acciones predefinidas

APÉNDICE B. MOCKUPS INICIALES

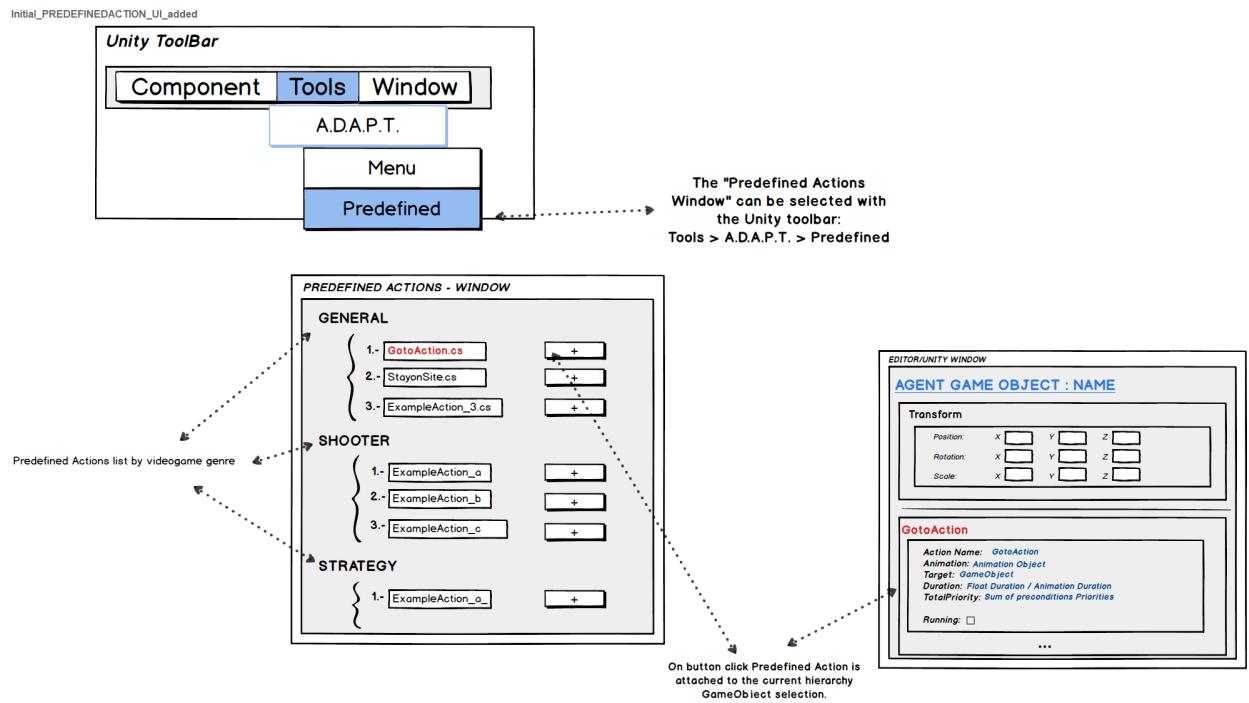


Figura B.10: Menú de acciones predefinidas, añadir acción

Apéndice C

Mockups Finales

En este apéndice se recogen todos los mockups/bocetos correspondientes a la última y definitiva versión de la herramienta. Los cuales han sido realizados en las respectivas fases de diseño a lo largo de los sucesivos [sprints](#) y a partir de los mockups iniciales basándose en la corrección de los mismos.

APÉNDICE C. MOCKUPS FINALES

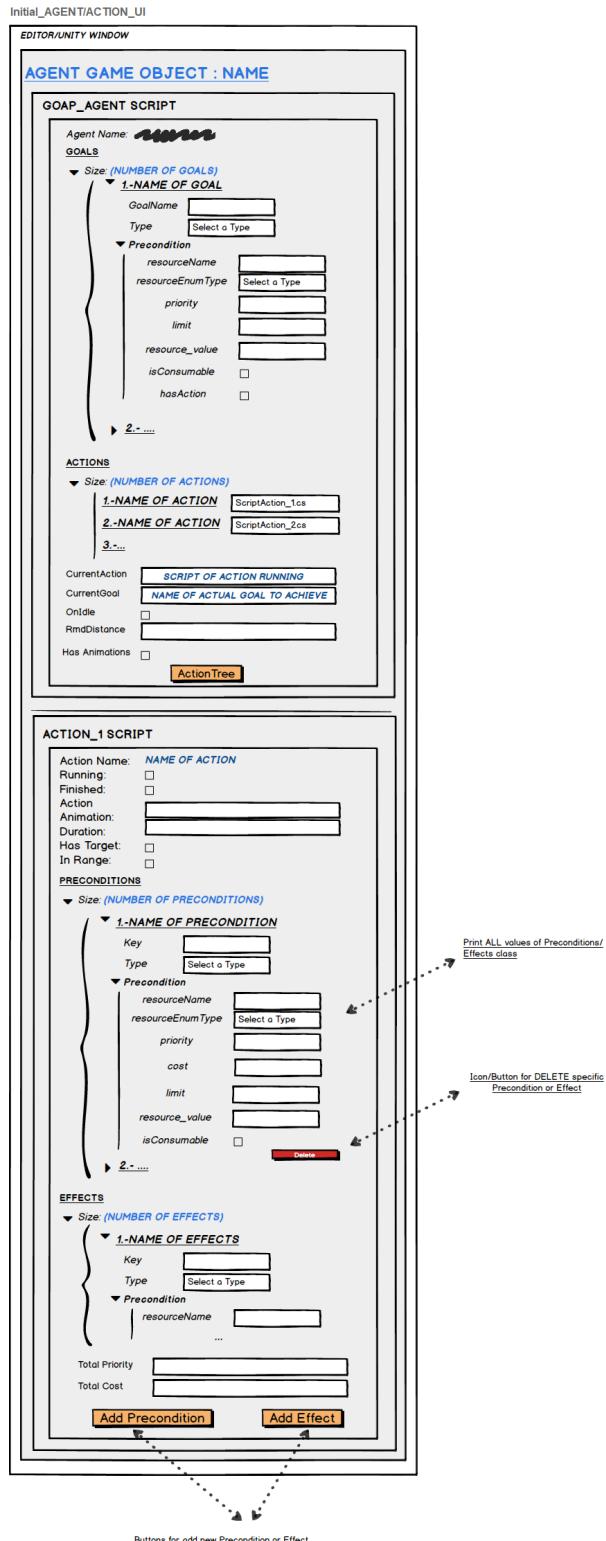


Figura C.1: Agente con acciones visualizado en el Unity Inspector

APÉNDICE C. MOCKUPS FINALES

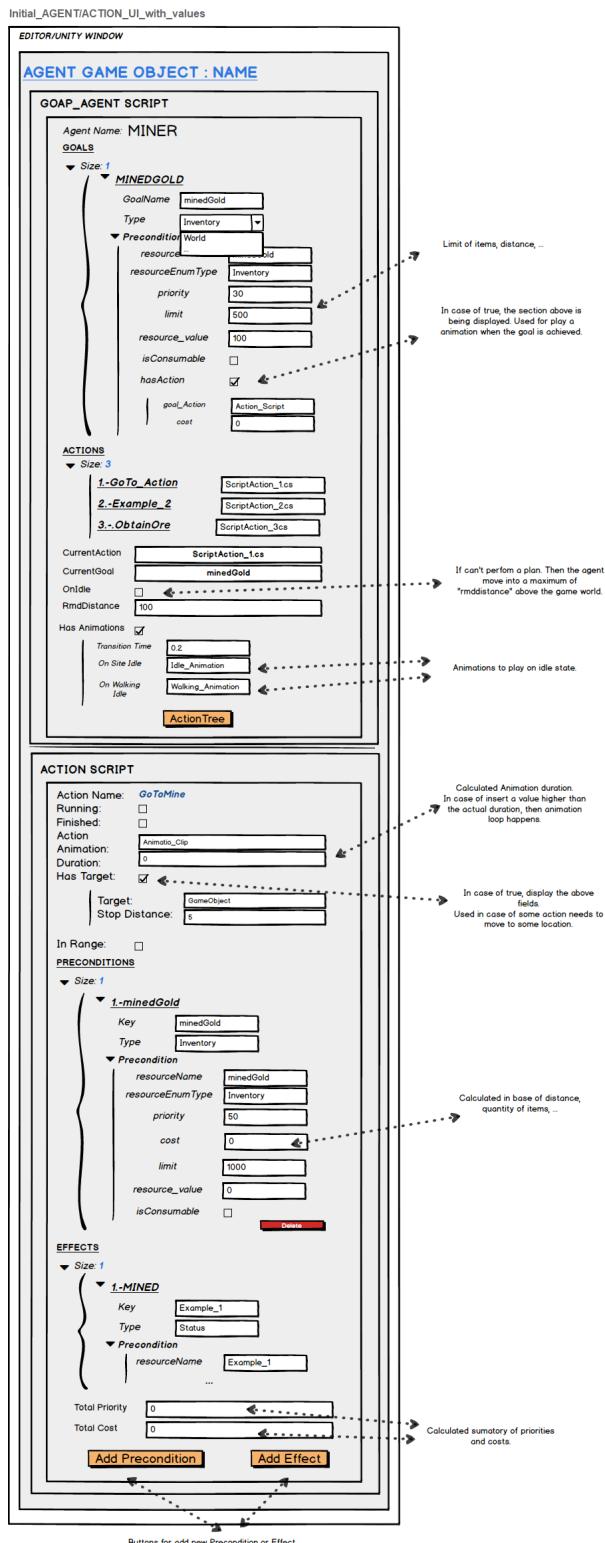


Figura C.2: Agente con acciones y valores cubiertos visualizado en el Unity Inspector

APÉNDICE C. MOCKUPS FINALES

Initial_AGENT/ACTION_UI_modify_values

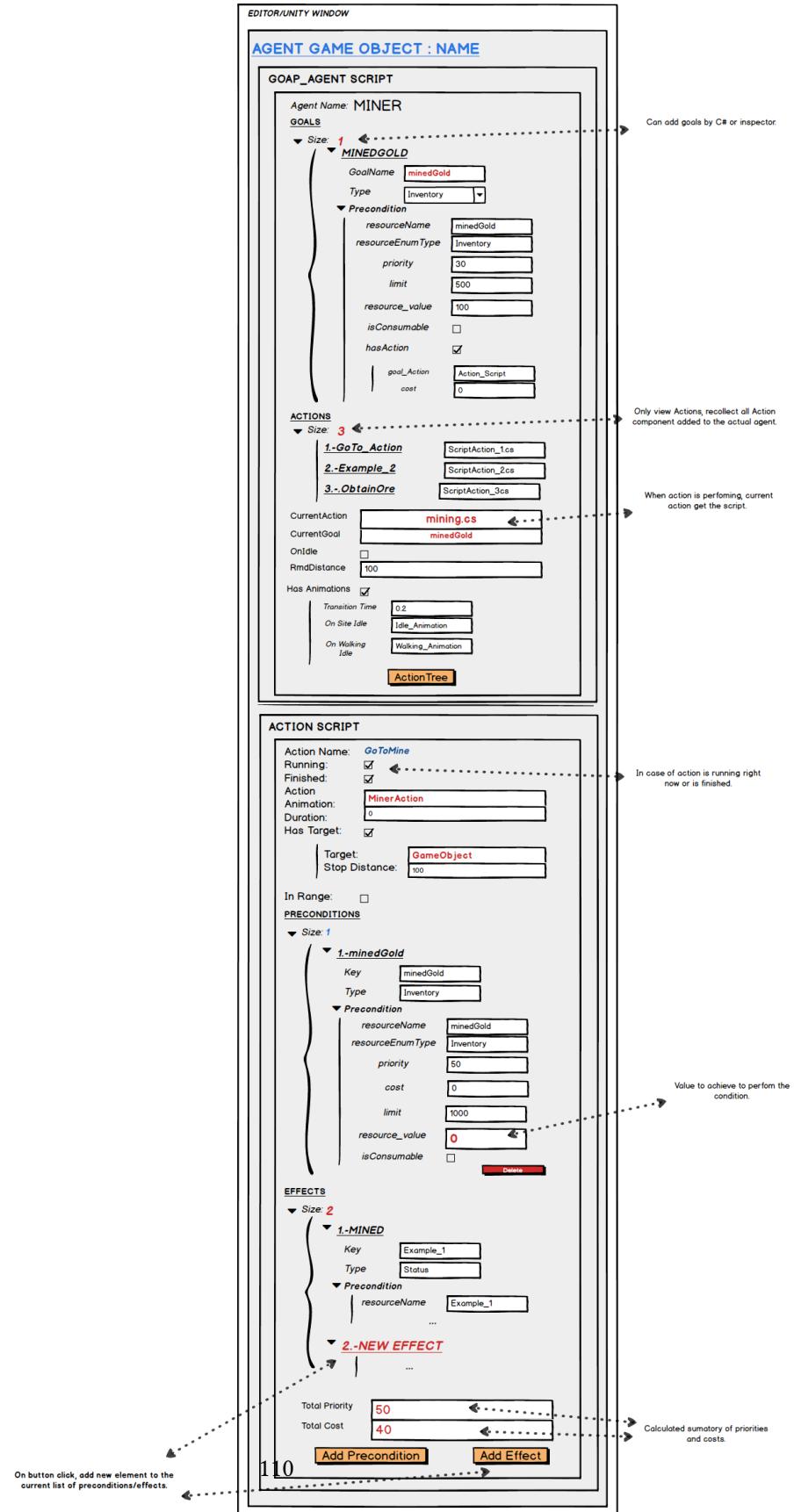


Figura C.3: Agente con acciones y valores modificados visualizado en el Unity Inspector

APÉNDICE C. MOCKUPS FINALES

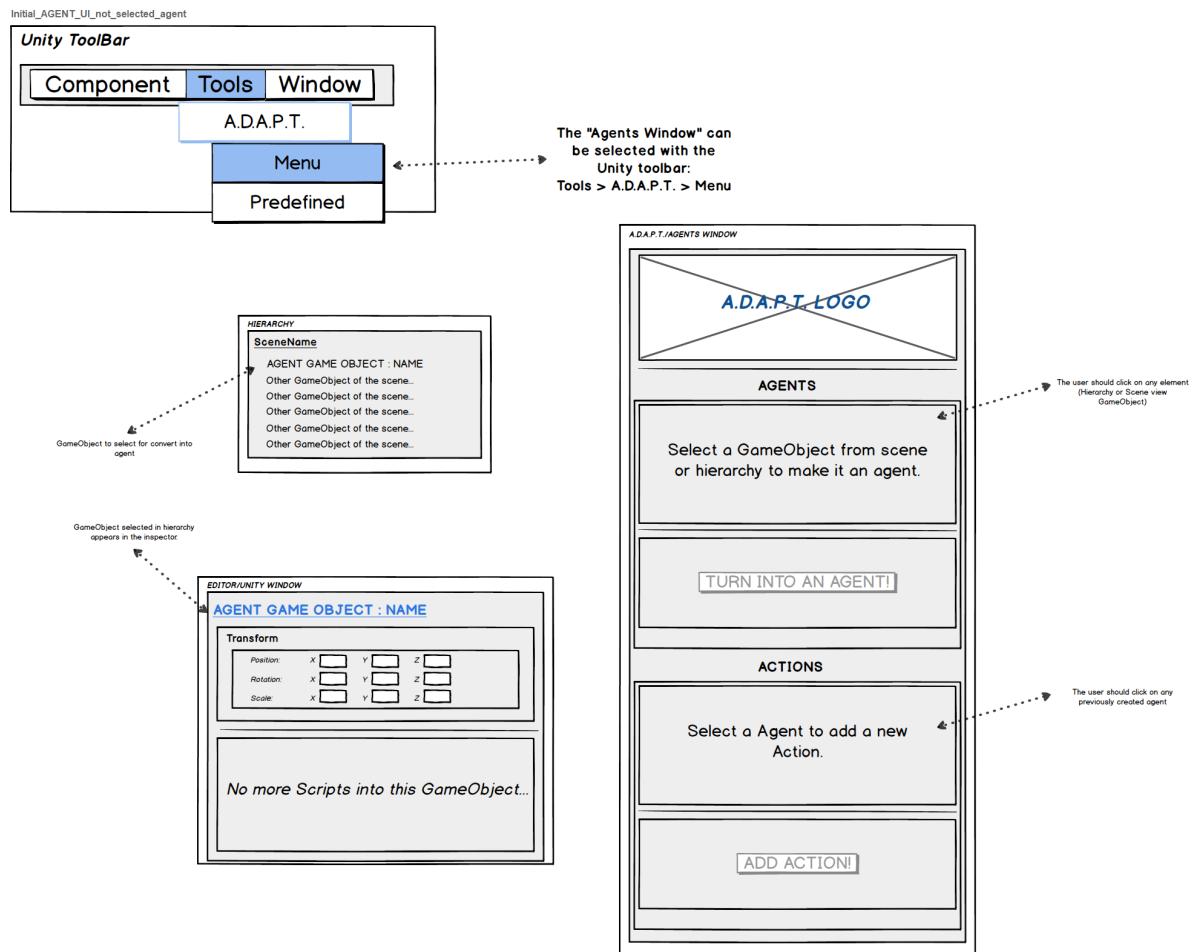


Figura C.4: Menú de la herramienta

APÉNDICE C. MOCKUPS FINALES

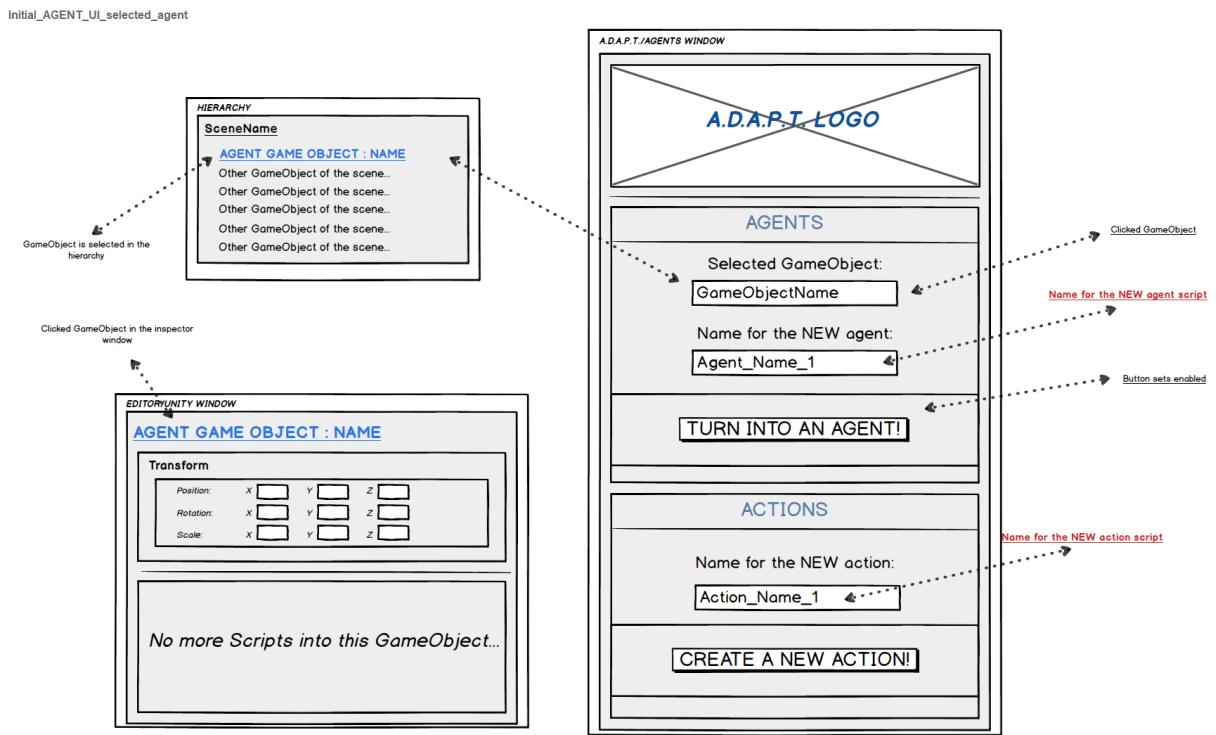


Figura C.5: Menú de la herramienta con objeto seleccionado en Unity

APÉNDICE C. MOCKUPS FINALES

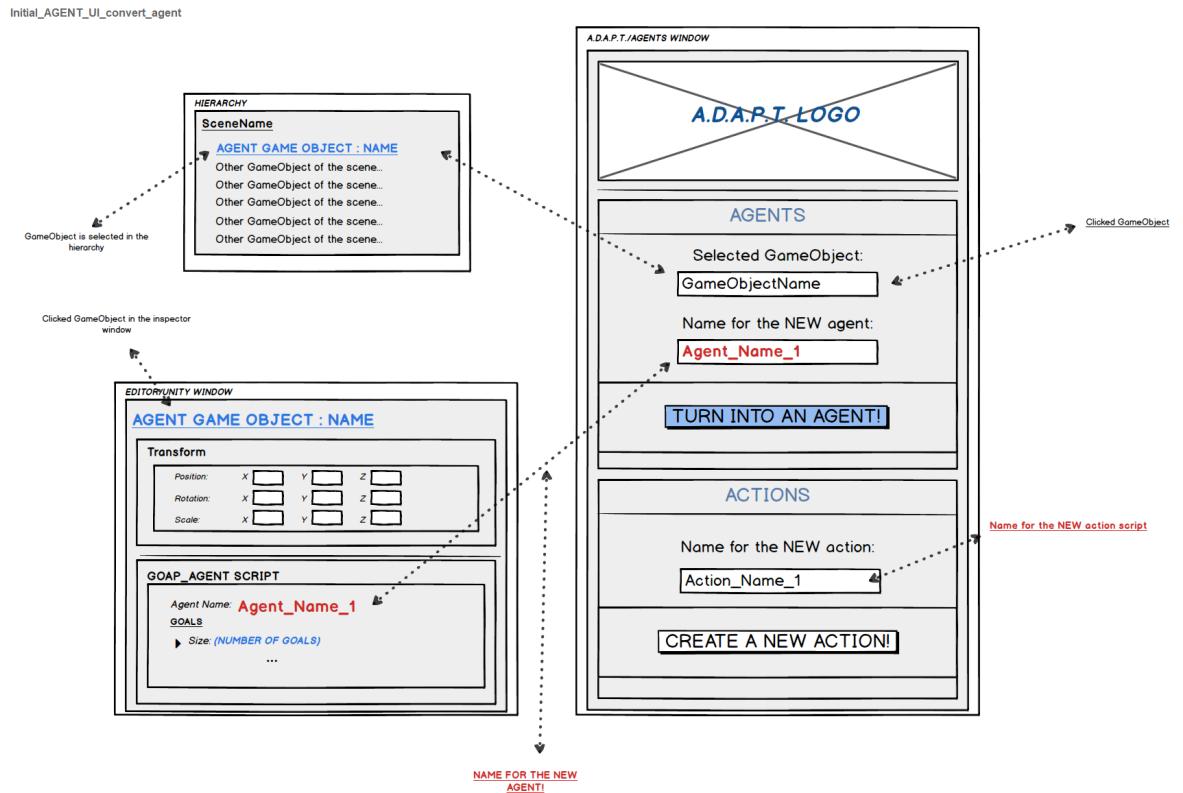


Figura C.6: Menú de la herramienta creando nuevo agente

APÉNDICE C. MOCKUPS FINALES

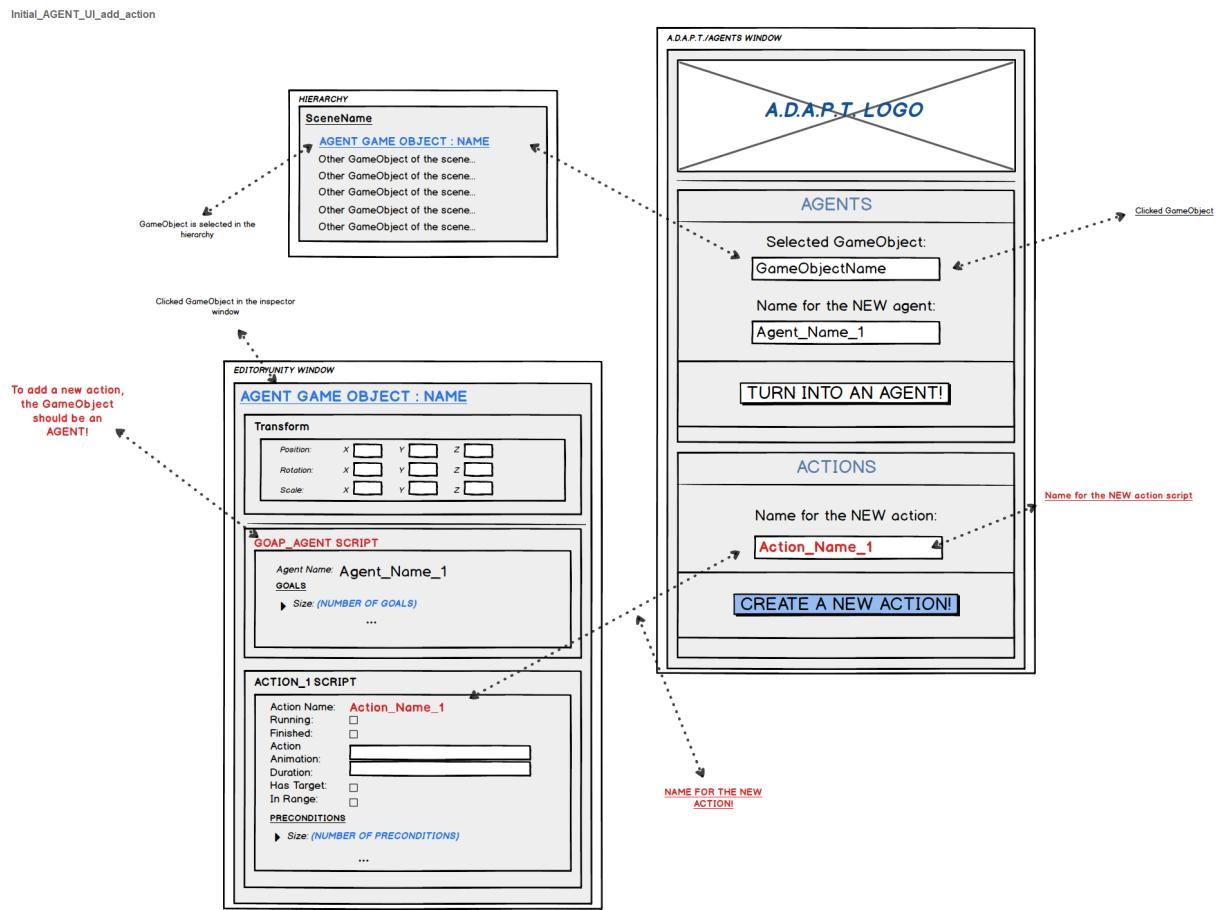


Figura C.7: Menú de la herramienta creando nueva acción

APÉNDICE C. MOCKUPS FINALES

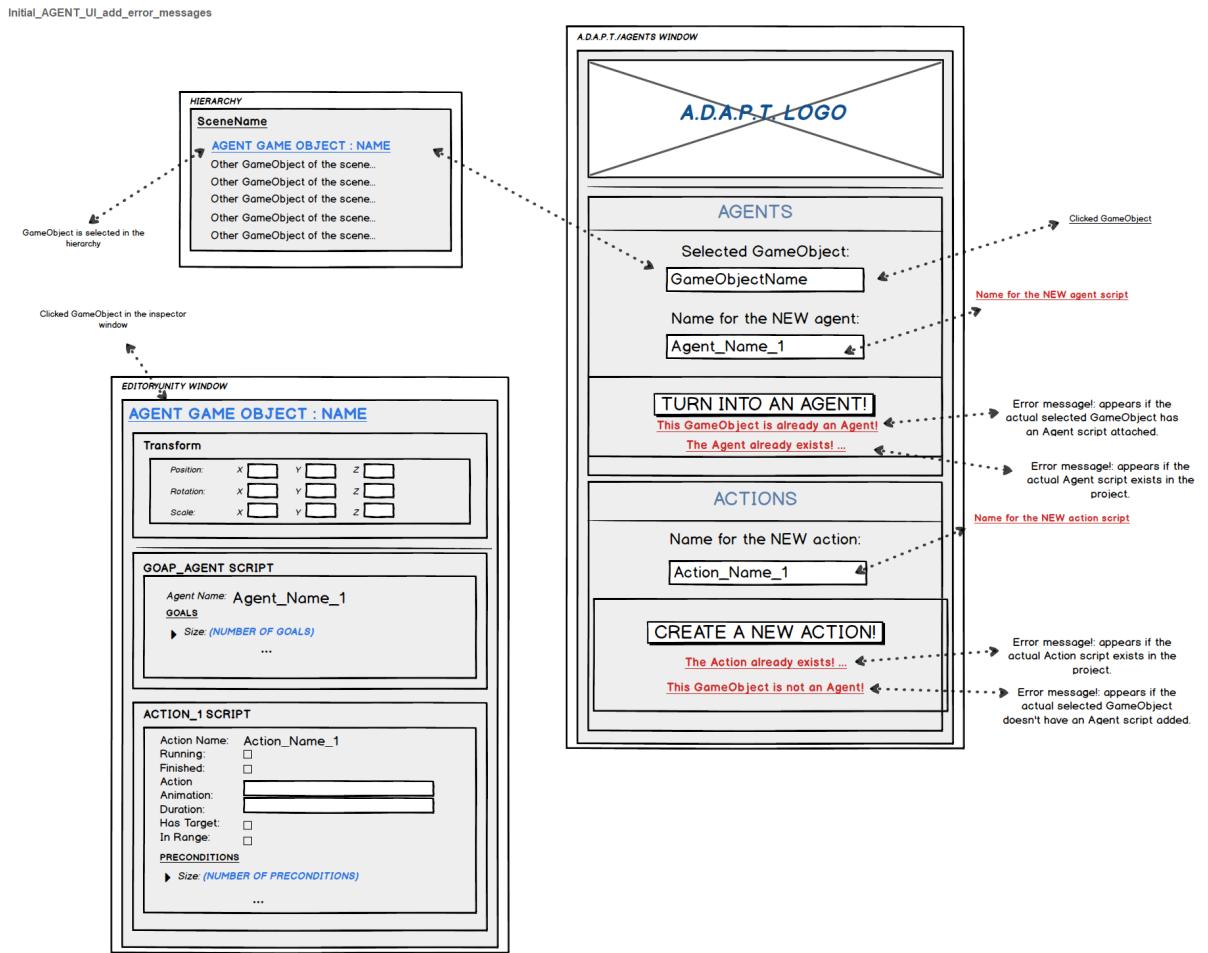


Figura C.8: Menú de la herramienta con mensajes de error

APÉNDICE C. MOCKUPS FINALES

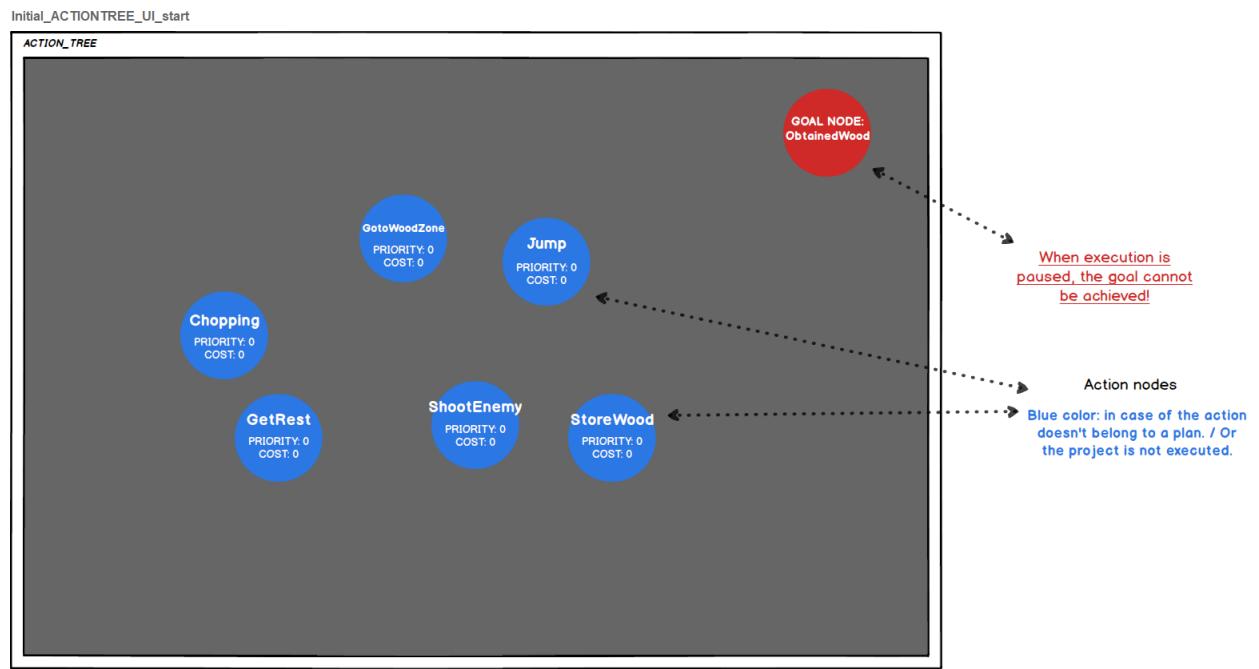


Figura C.9: Árbol de acciones

APÉNDICE C. MOCKUPS FINALES

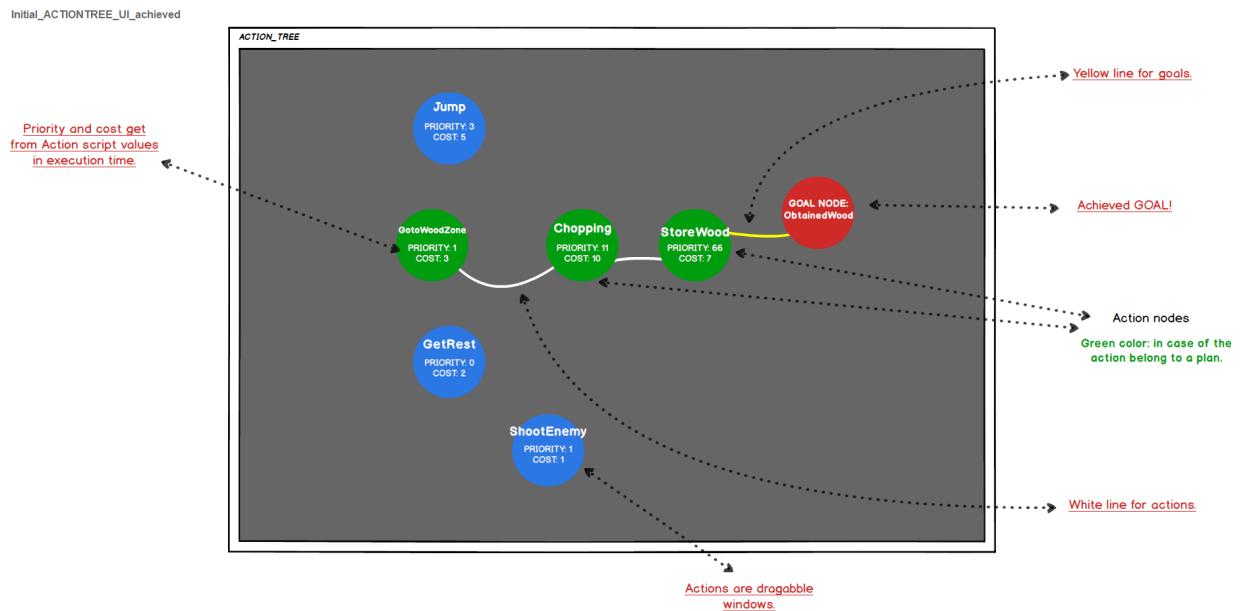


Figura C.10: Árbol de acciones en tiempo de ejecución

APÉNDICE C. MOCKUPS FINALES

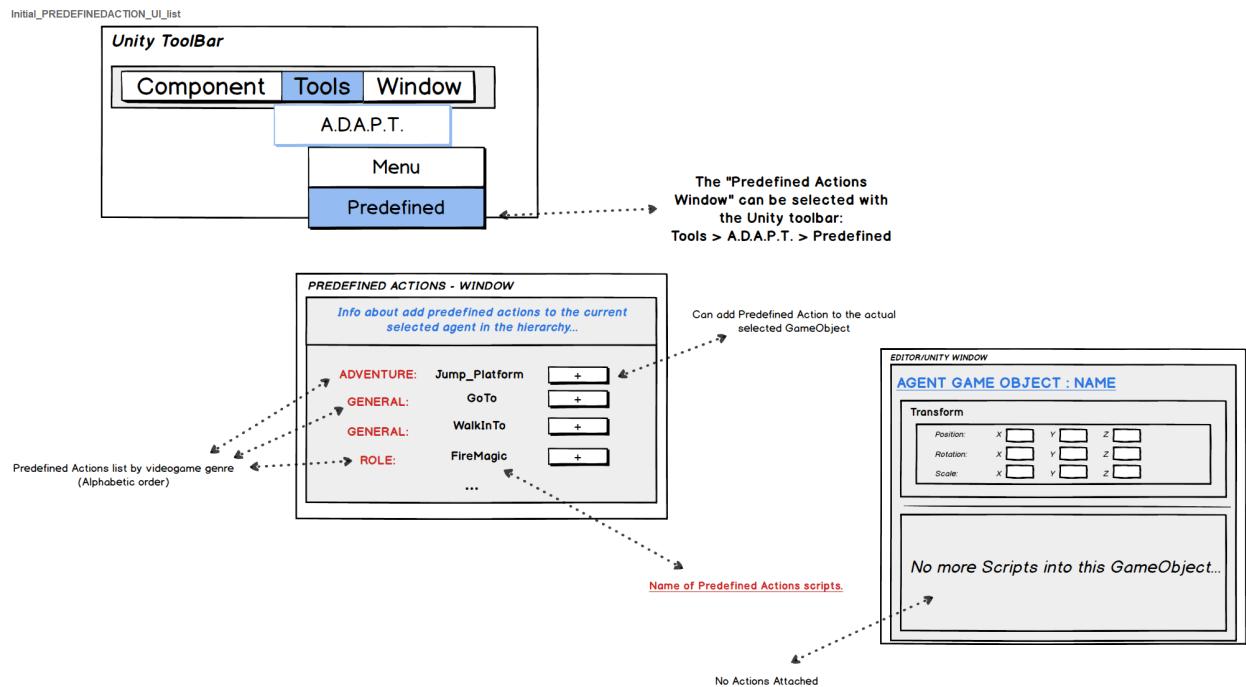


Figura C.11: Menú de acciones predefinidas

APÉNDICE C. MOCKUPS FINALES

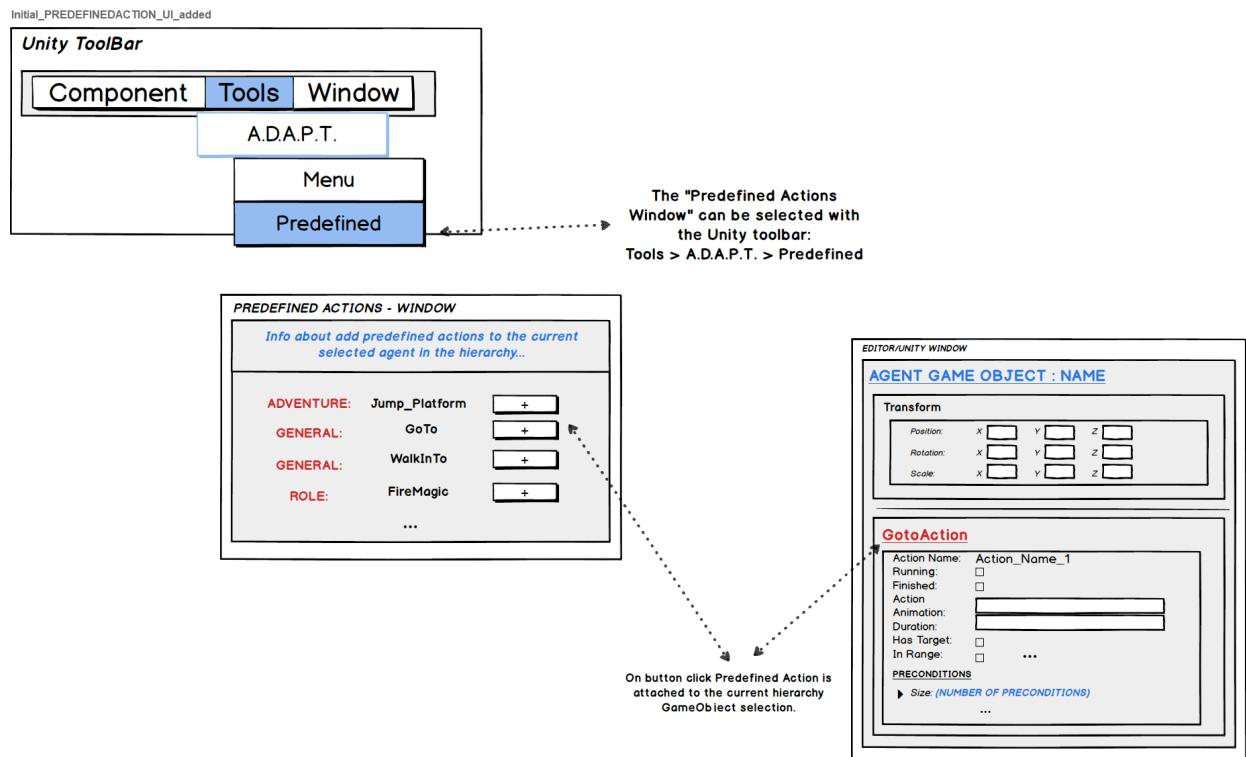


Figura C.12: Menú de acciones predefinidas, añadir acción

APÉNDICE C. MOCKUPS FINALES

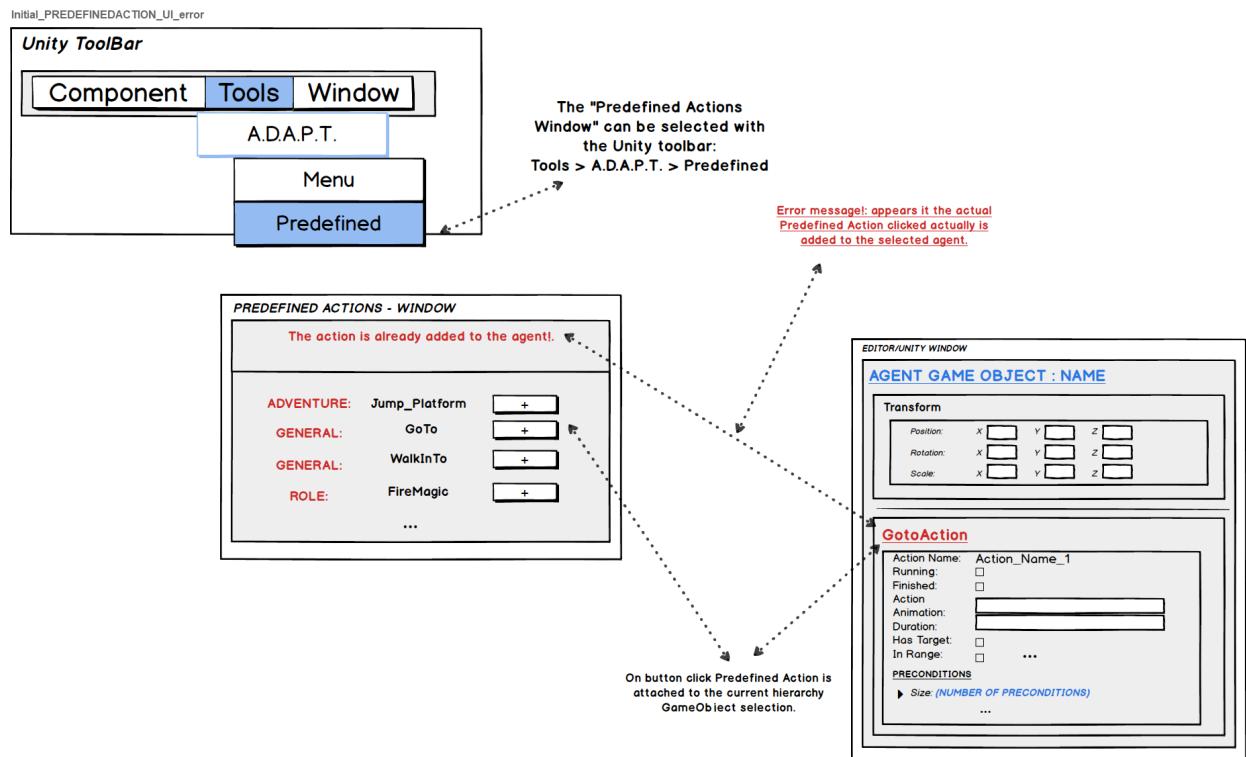


Figura C.13: Menú de acciones predefinidas: mensaje de error de acciones

APÉNDICE C. MOCKUPS FINALES

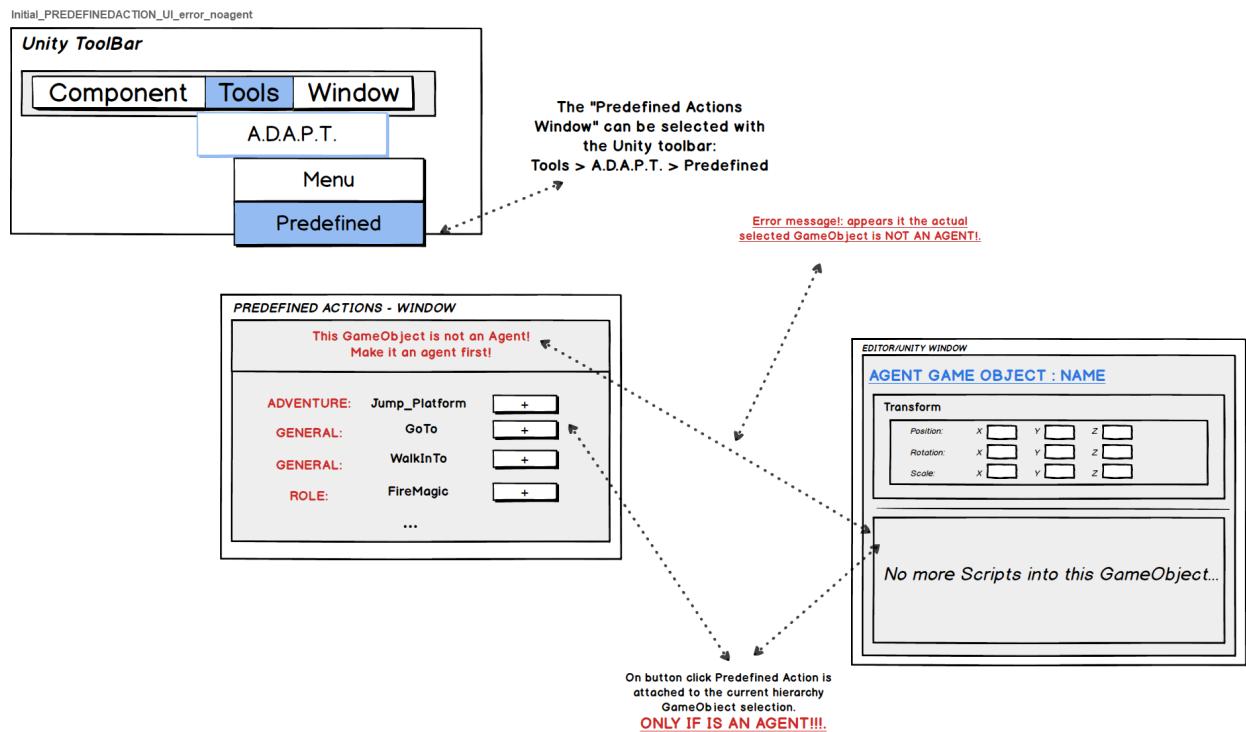


Figura C.14: Menú de acciones predefinidas: mensaje de error de agente

Apéndice D

Solución desarrollada

En este apéndice se recogen todos las imágenes relacionadas con el aspecto final de la herramienta, es decir, toda la parte de la interfaz gráfica que permita el llevar a cabo, las diferentes funcionalidades de la misma.



Figura D.1: Toolbar de Unity, donde pueden seleccionarse los menús de la herramienta

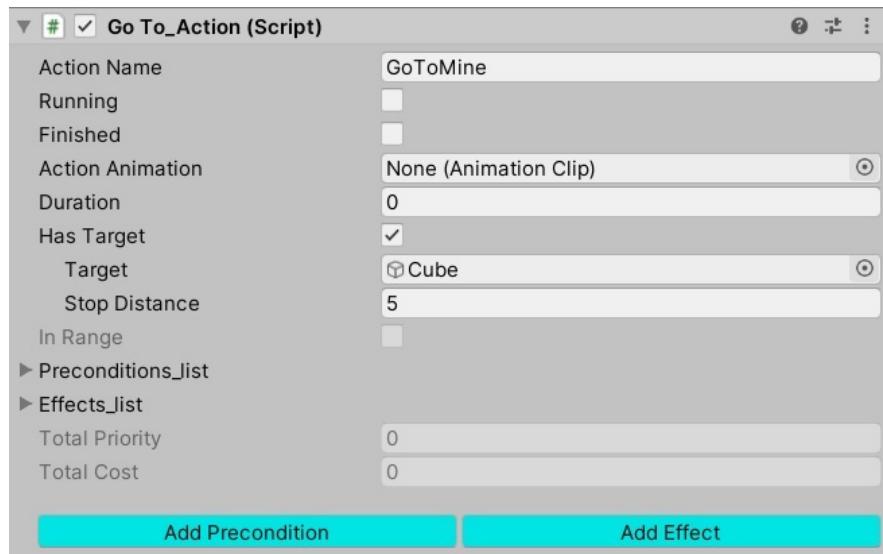


Figura D.2: Acción correspondiente al sistema GOAP

APÉNDICE D. SOLUCIÓN DESARROLLADA

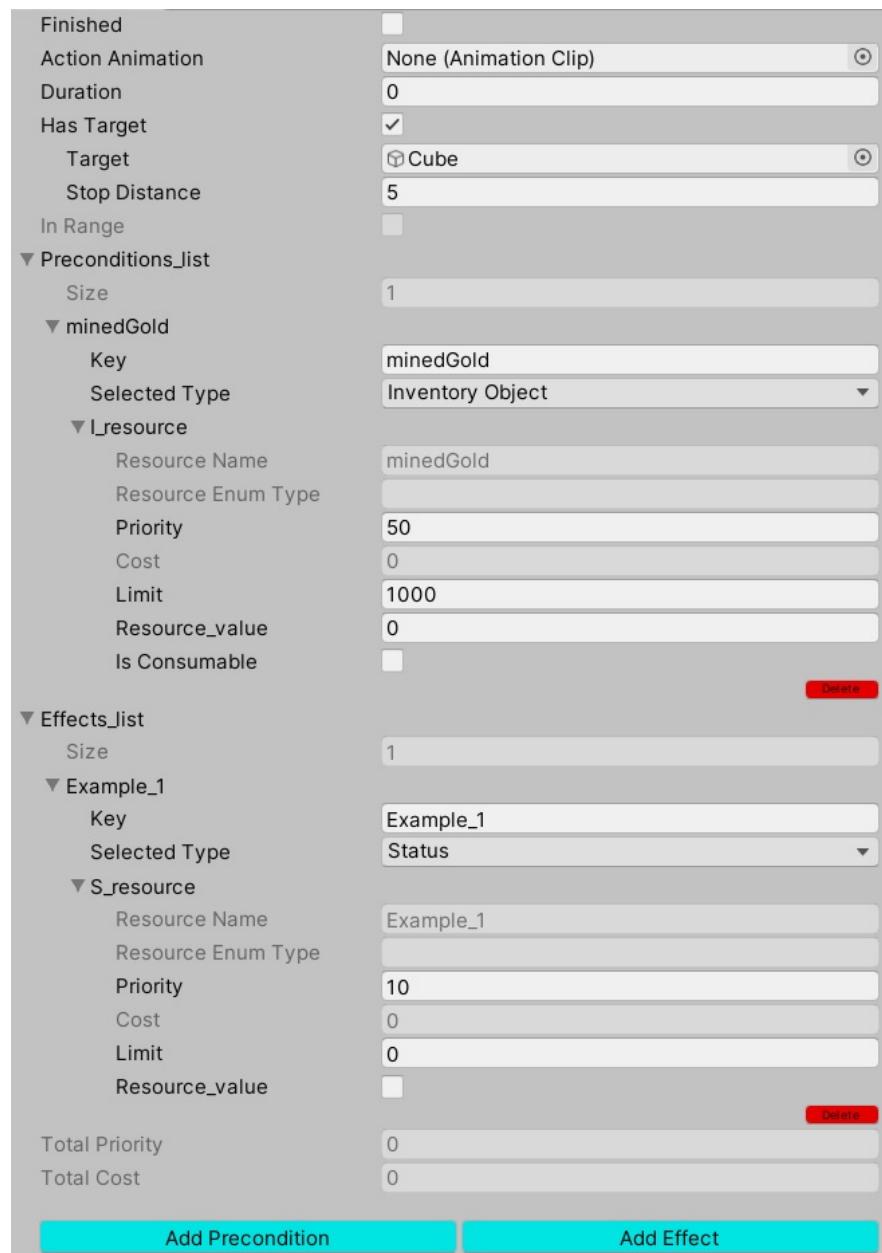


Figura D.3: Recursos de una acción desplegados

APÉNDICE D. SOLUCIÓN DESARROLLADA

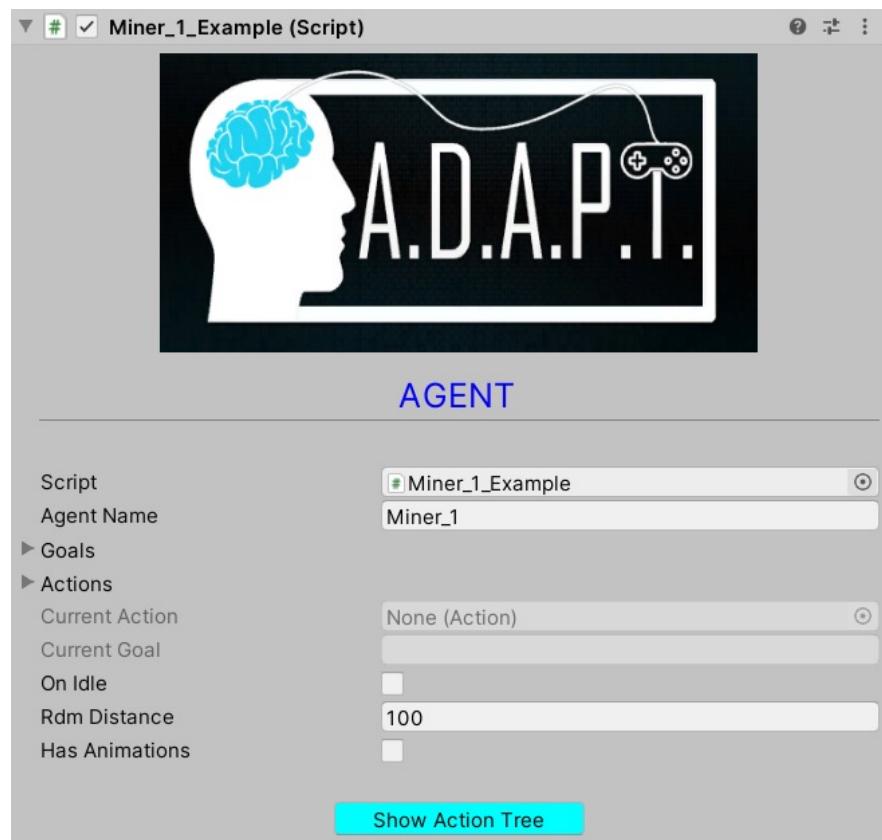


Figura D.4: Visualización de agente en inspector

APÉNDICE D. SOLUCIÓN DESARROLLADA

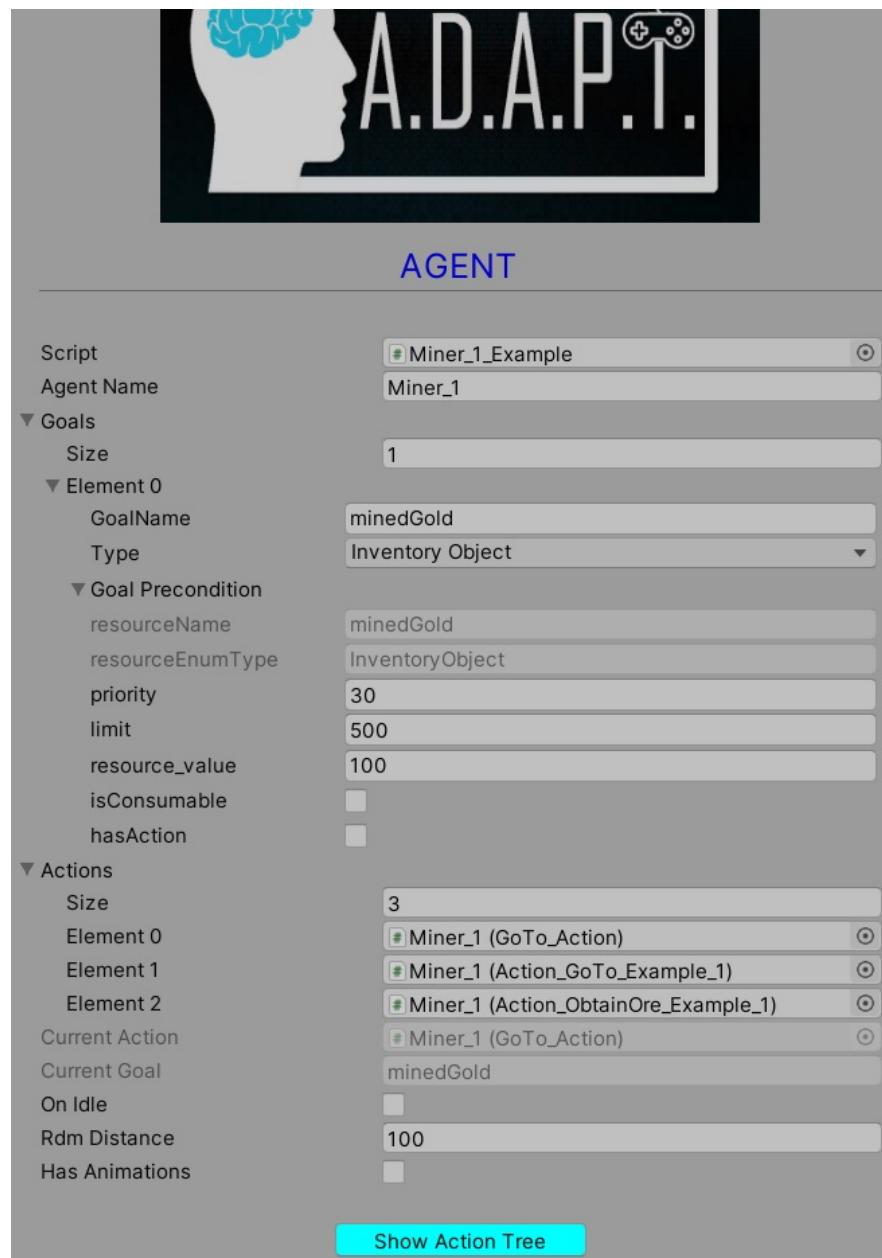


Figura D.5: Despliegue de propiedades de agente (Goals y acciones)

APÉNDICE D. SOLUCIÓN DESARROLLADA

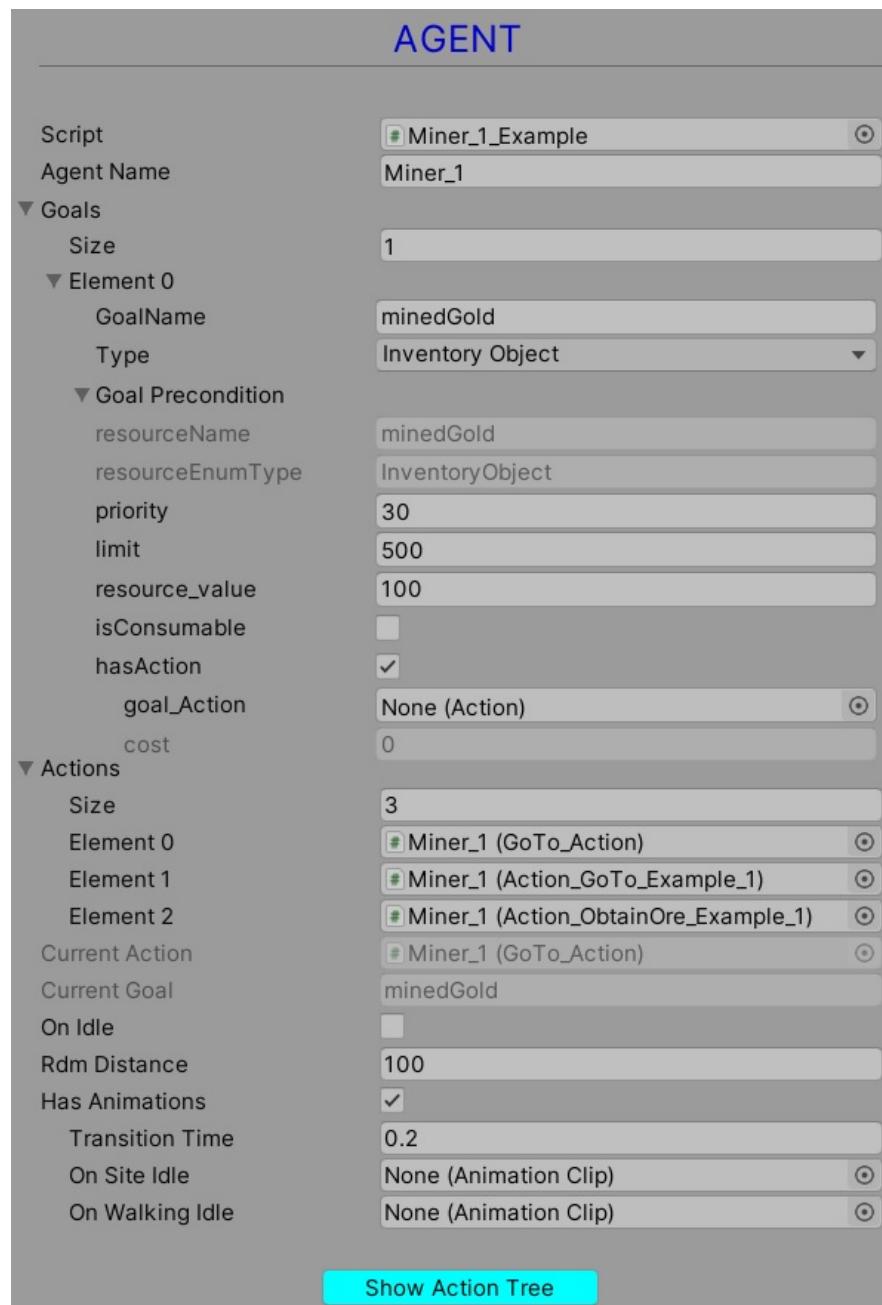


Figura D.6: Propiedades relacionadas con las animaciones de un agente



Figura D.7: Menú de A.D.A.P.T., ventana principal

APÉNDICE D. SOLUCIÓN DESARROLLADA

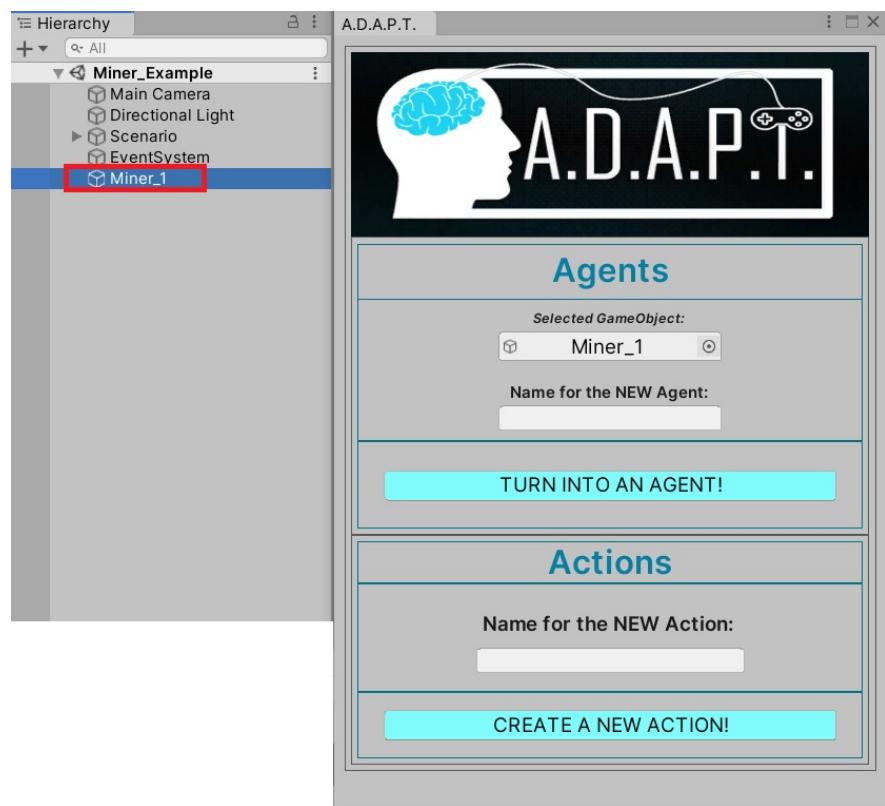


Figura D.8: Menú de A.D.A.P.T. cuando se selecciona un objeto

APÉNDICE D. SOLUCIÓN DESARROLLADA

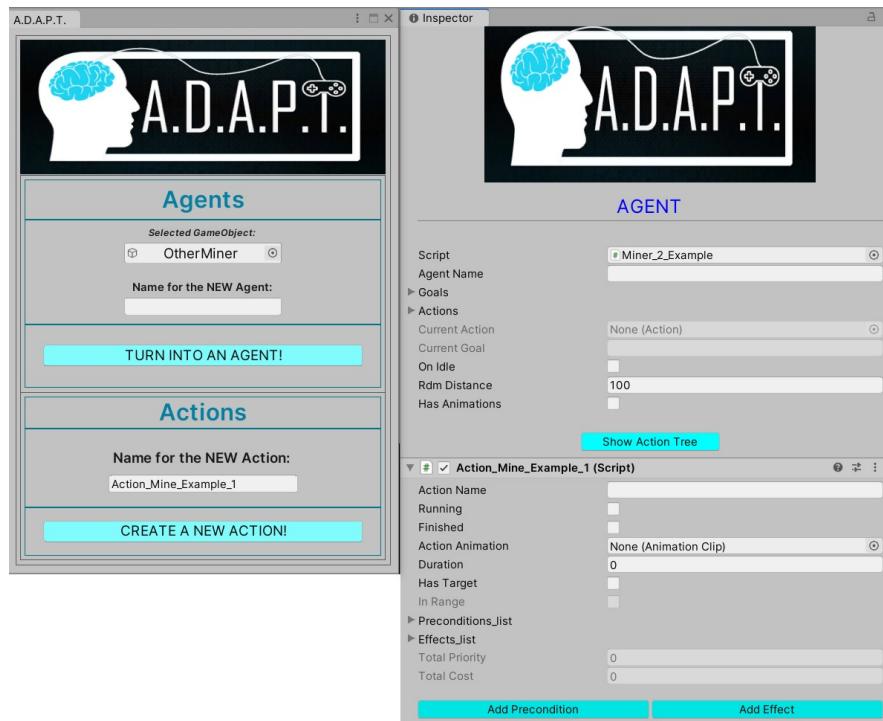


Figura D.9: Añadir una nueva acción a través del menú



Figura D.10: Mensaje de error en caso de que una acción ya exista

APÉNDICE D. SOLUCIÓN DESARROLLADA

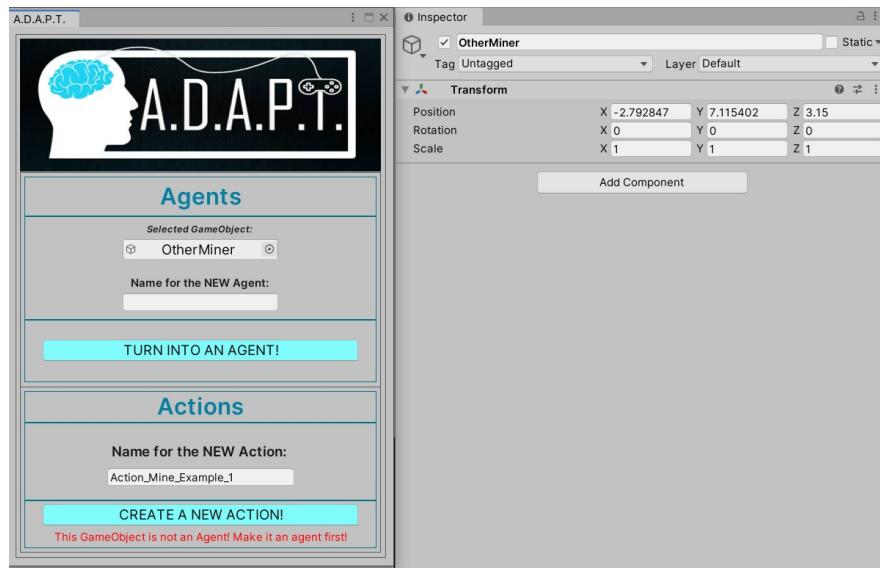


Figura D.11: Mensaje de error en caso de que el objeto actual no sea un agente

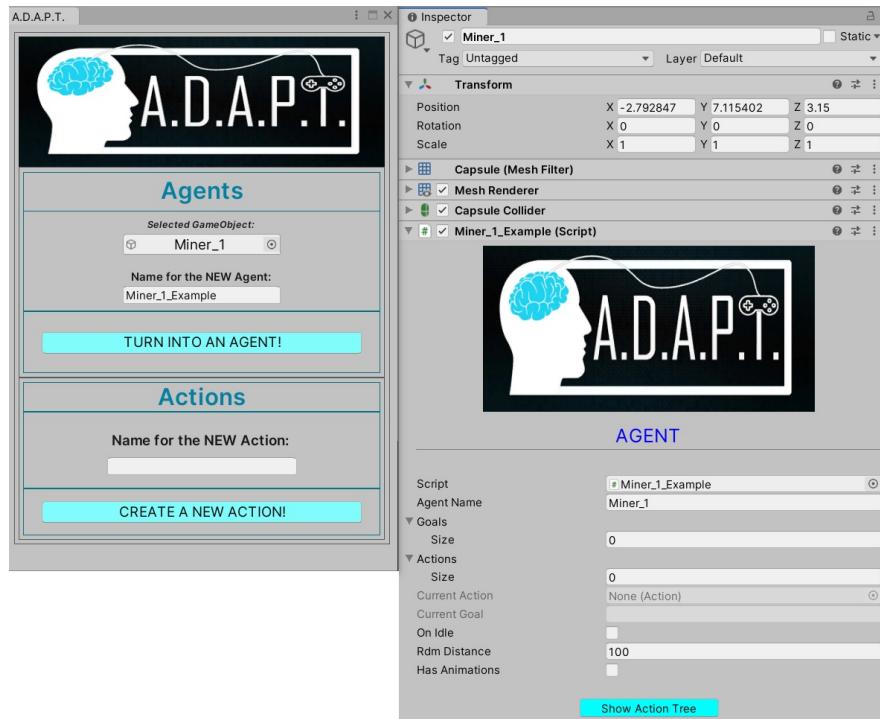


Figura D.12: Añadir un nuevo agente a través del menú

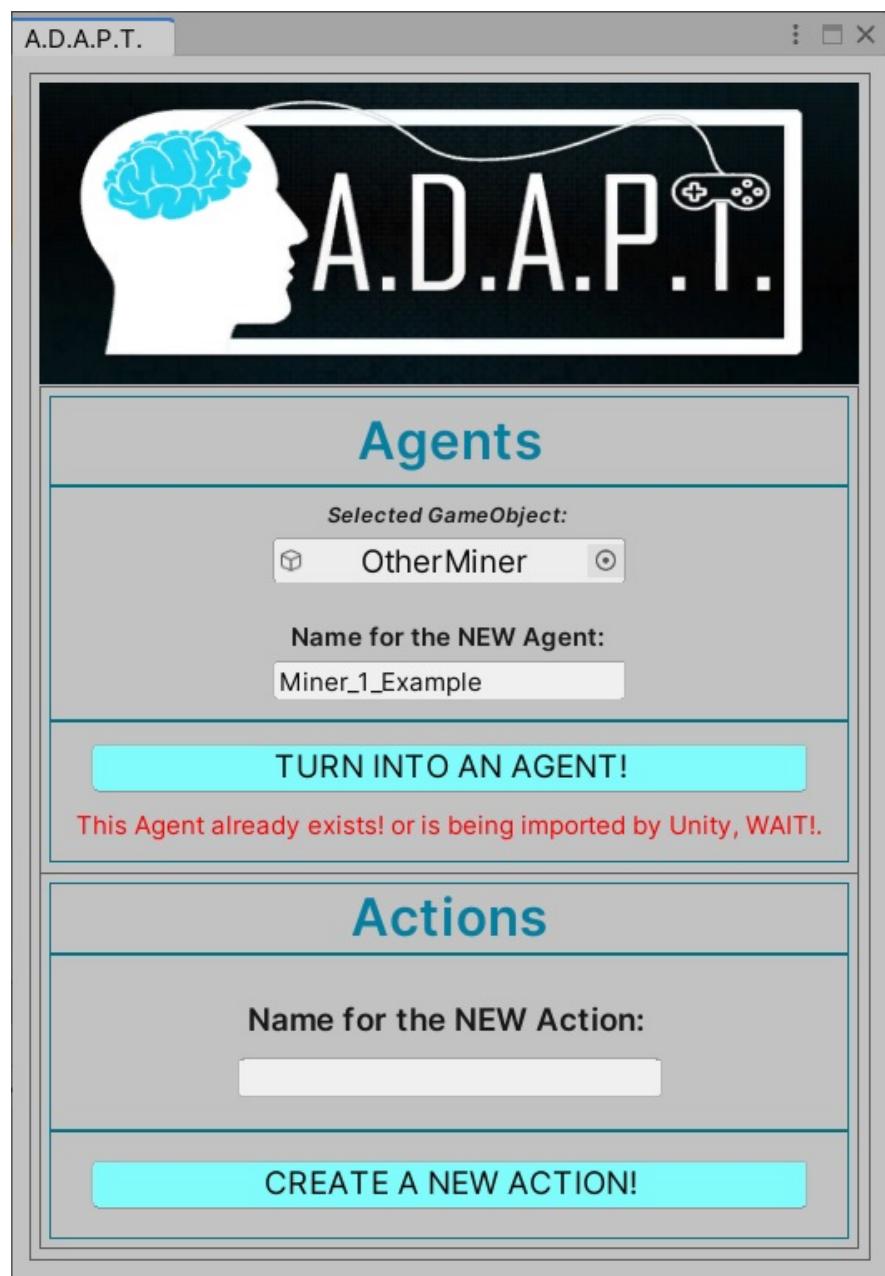


Figura D.13: Mensaje de error en caso de que un agente ya exista

APÉNDICE D. SOLUCIÓN DESARROLLADA

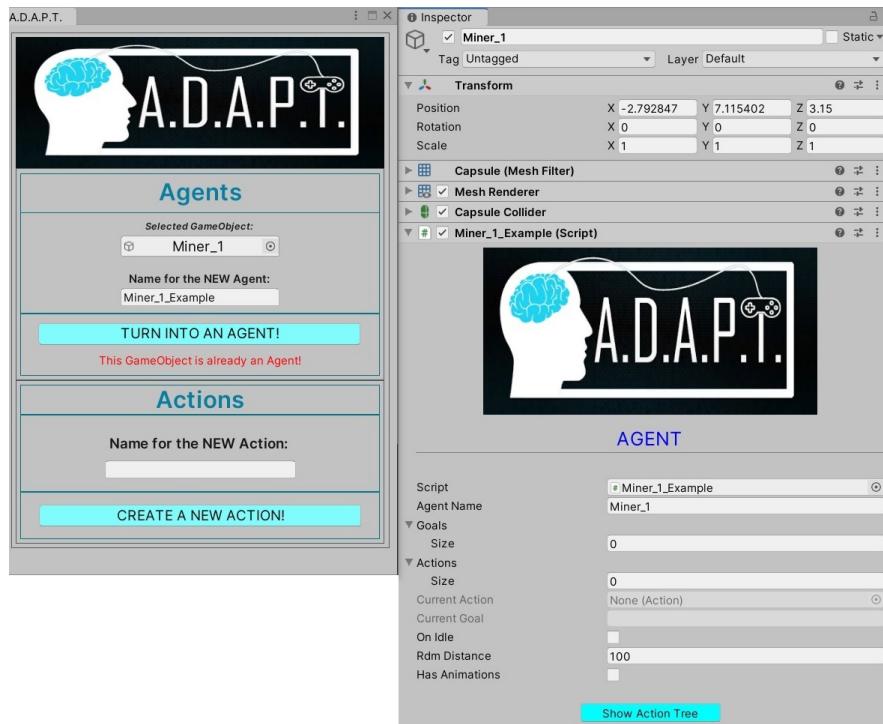


Figura D.14: Mensaje de error en caso de que el objeto ya sea un agente

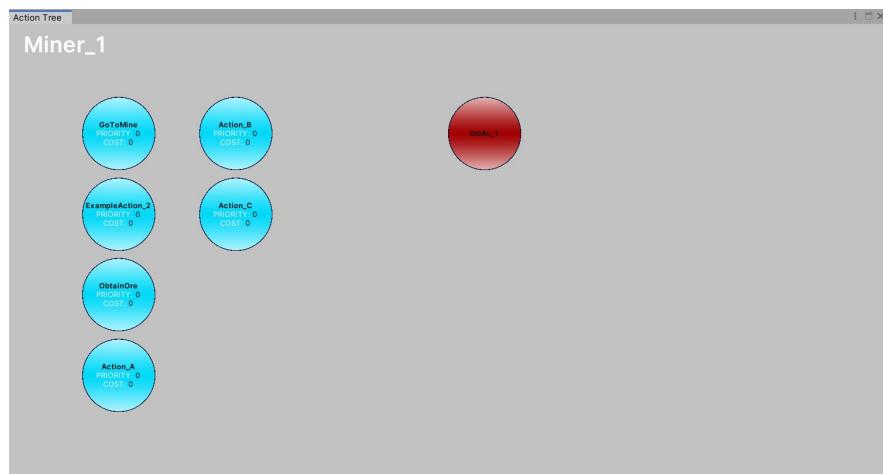


Figura D.15: Árbol de acciones de un agente, sin ejecutar

APÉNDICE D. SOLUCIÓN DESARROLLADA

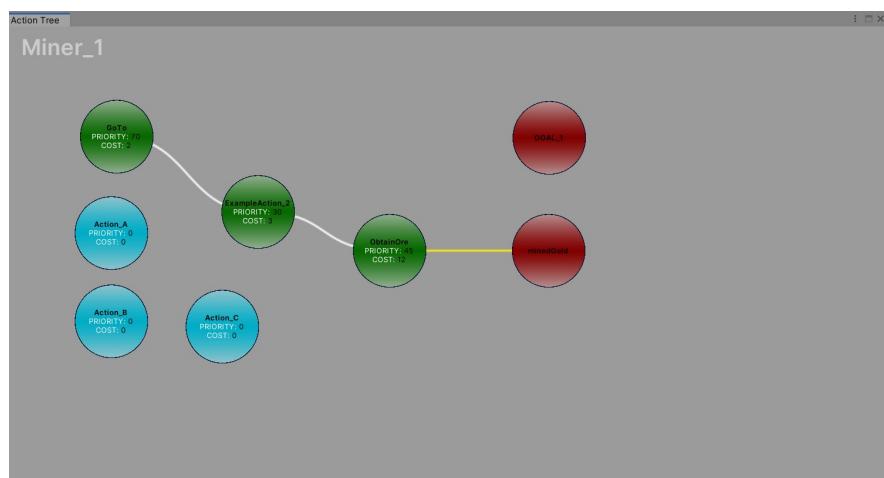


Figura D.16: Árbol de acciones en tiempo de ejecución

APÉNDICE D. SOLUCIÓN DESARROLLADA

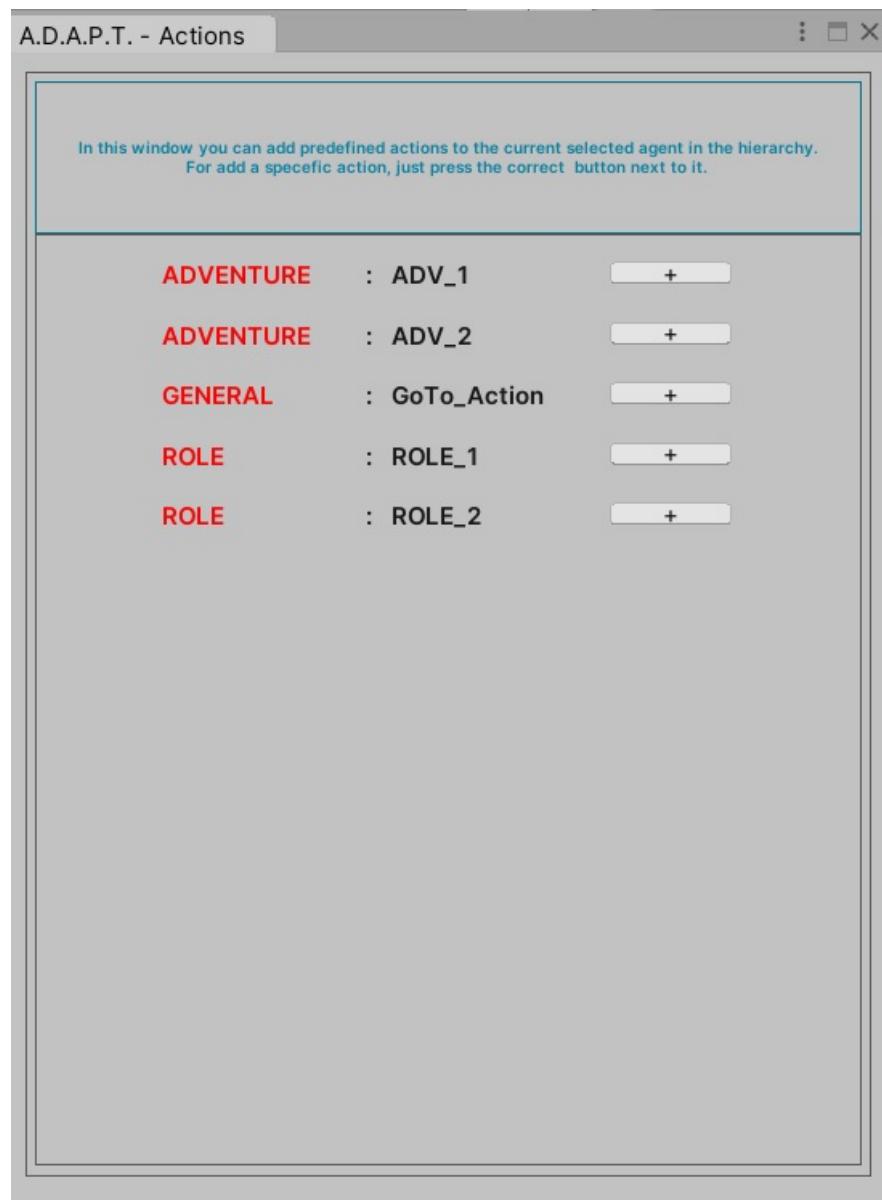


Figura D.17: Menú de acciones predefinidas

APÉNDICE D. SOLUCIÓN DESARROLLADA

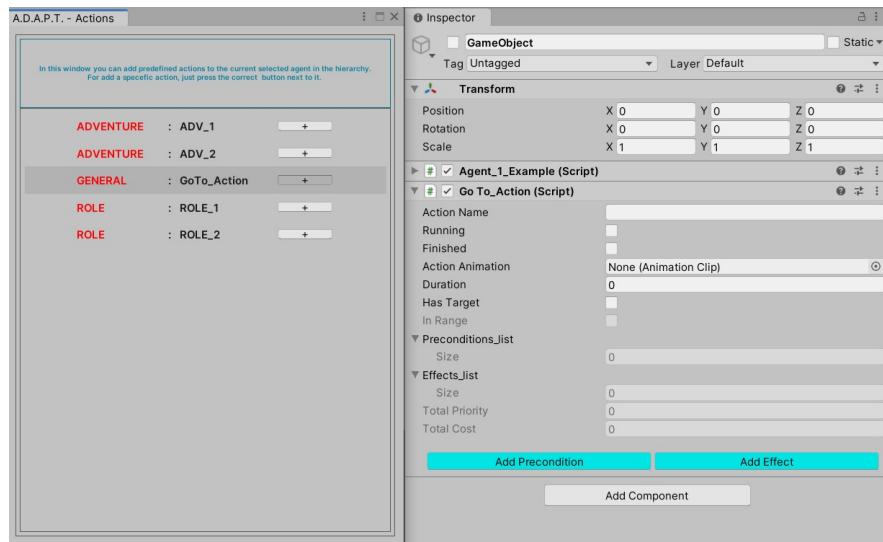


Figura D.18: Añadir acción predefinida a un agente

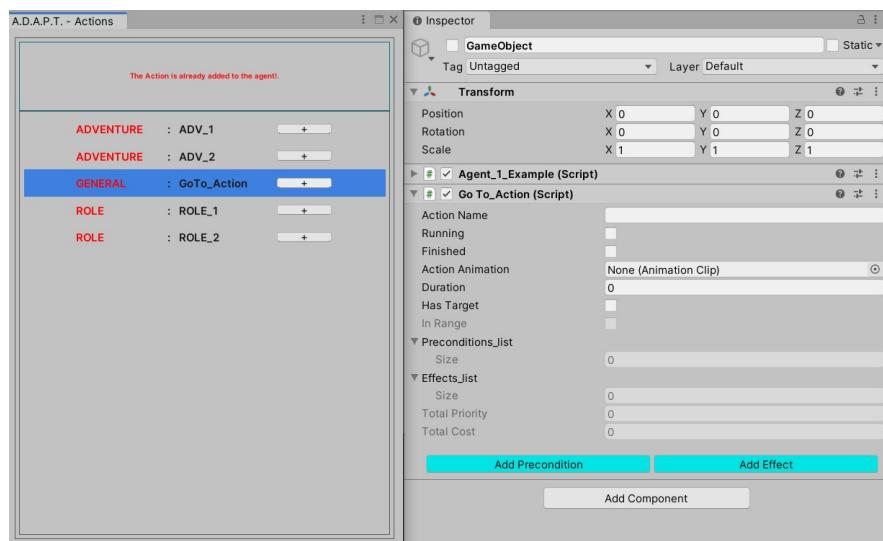


Figura D.19: Mensaje de error en caso de que la acción ya esté agregada

APÉNDICE D. SOLUCIÓN DESARROLLADA

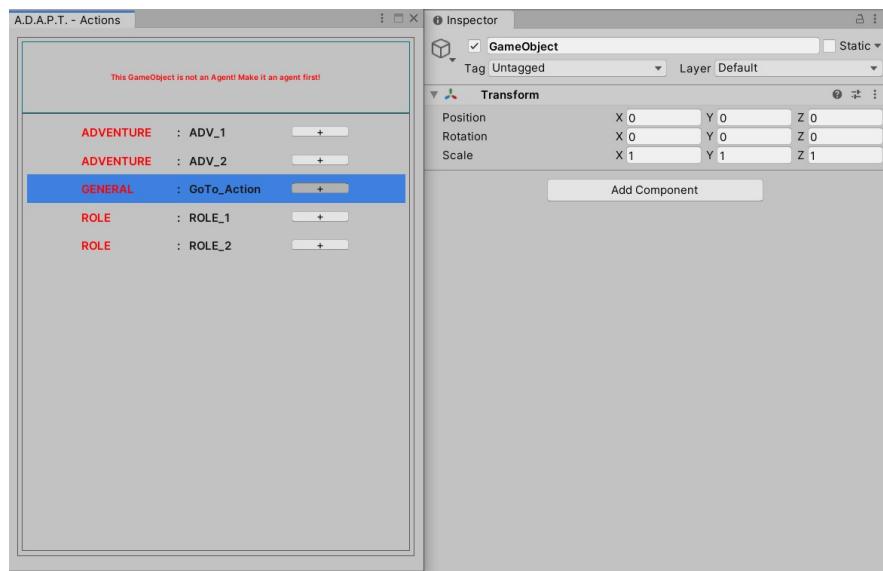


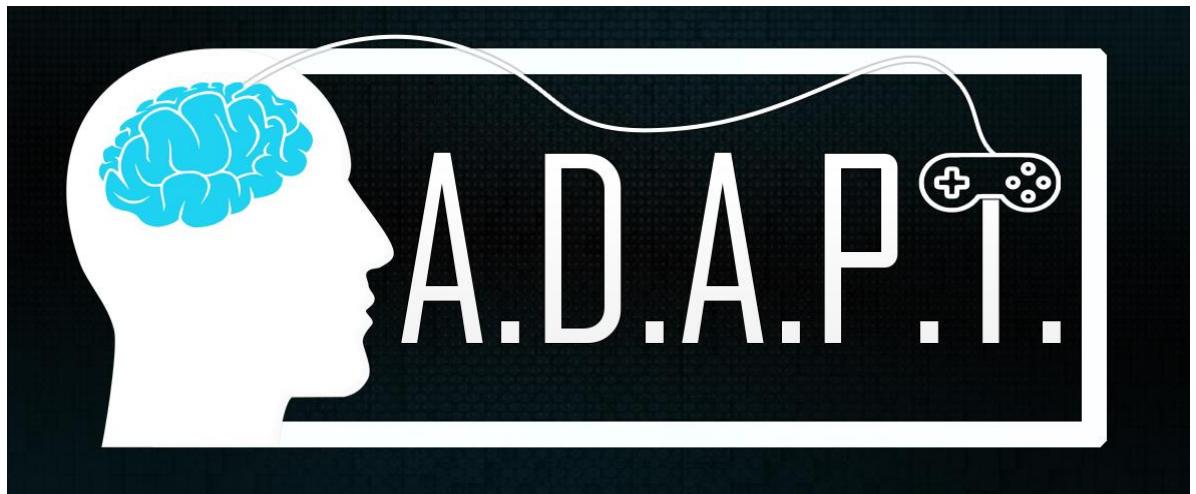
Figura D.20: Mensaje de error en caso de que el objeto no sea un agente

Apéndice E

Manual de usuario

En este apéndice se recoge el documento del manual de usuario. El cual está enfocado a poder ayudar al usuario final de cara a la utilización de la herramienta.

V. 1.0.



A.D.A.P.T. USER GUIDE

GOAL-ORIENTED ACTION PLANNING TOOL FOR ARTIFICIAL INTELLIGENCE IN UNITY

A.D.A.P.T. – UNITY AI TOOL
YAGOMIRA@GMAIL.COM

-INDEX:

1.- A.D.A.P.T. as GOAP tool.....	2
2.- Some concepts.....	2-3
3.- Create a new agent from zero.....	3-6
4.- About Agents.....	7-9
5.- Create new actions.....	9-12
6.- By code or by inspector.....	12-17
7.- Predefined Actions.....	17-19
8.- Action Tree.....	19-21
9.- In case of bug.....	21

1.- A.D.A.P.T. as GOAP tool :

Currently Artificial Intelligence (AI) is a technology that is constantly developing and improving. Used in many areas such as the video game sector, where through the analysis of the environment and the ability to adapt to it, behaviors similar to those of humans can be seen. This project is based on that idea, on developing a tool that allows in a simple way to be able to create an intelligent behavior and fill those deficiencies that the Unity video game engine does not offer in a simple way to the user.

To achieve this goal, one of the most innovative systems was integrated in terms of the creation of AI in video games, known as Goal-Oriented Action Planning (GOAP) which will allow those Non-Playable Characters Non-Playable Characters (NPC) to be provided with a set of actions, and depends on the situation one or other will be executed, with the target of achieve a predetermined goal.

2.- Some Concepts:

-Resources: the resources will be the actual Preconditions/Effects/Goals, A.D.A.P.T. use this for do more easy the concept of the actual states to achieve. Actually A.D.A.P.T. has **4 types** of resources:

1.-WorldResource: used for the desired locations, like some building, other Agent, ... is the same as PositionResource but use GameObject.

2.-PositionResource: used for desired locations, use Transforms.

3.-InventoryResource: used for items can stack, like wood, minerals, ...

4.-StatusResource: use for abstract situations for example, when a character is sick.

-States:

-Agent_States: used as local inventory for the actual agent, only this agent can access to the inventory.

-Global_States: global inventory, shared between all agents, for example: a gold storage of the town (*strategy games like Age of Empires*).

-¿How to perform a plan?: actually to perform a plan with the GOAP system you should achieve the preconditions when the actual state of the agent, for example:

-Agent has a state of inventory where: "Wood = 100"

-And a precondition to "Craft Wood Table" is "Wood = 100" then the Action can be performed because the preconditions are achieved.

-For other side, the Action has some effects of its execution, this effects are for example, after play the "Craft Wood Table" Action, then the actual state of the Agent: "Wood = 0" because the Agent use all of the wood in the Action.

-And a consequence of possible actions can achieve a goal, the goal can be anything you want, for example:

-"Stop work", and this goal makes the agent cannot do more actions.

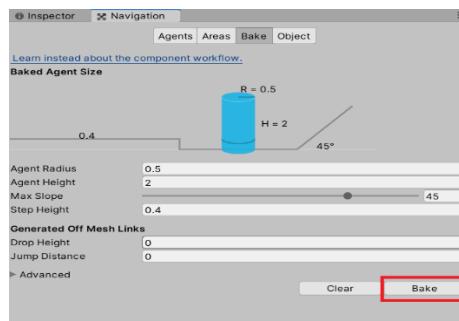
-And you can achieve this goal if you for example has a consequence of actions you can achieve because you fulfill with all preconditions:

"Make Wood Table" > "Go to Bed" > "Stop Work"

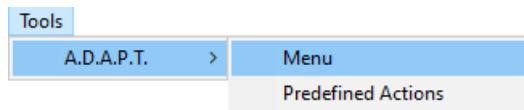
3.- Create a new agent from zero :

For transform your desired GameObject into an GOAP Agent you should follow the next steps:

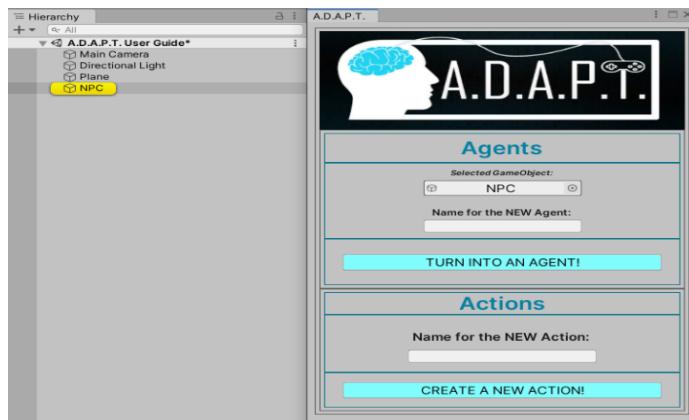
-FIRST OF ALL: IF YOU WANT TO CREATE A NEW AGENT, YOU SHOULD BAKE THE MESH TO NAVIGATE AS YOU DO WITH NAVMESH:



1.- Open A.D.A.P.T. Menu: Tools > A.D.A.P.T. > Menu



2.- With the Menu open, select your desired GameObject in the hierarchy:



At the moment into the “Selected GameObject” box should appear your GameObject name.

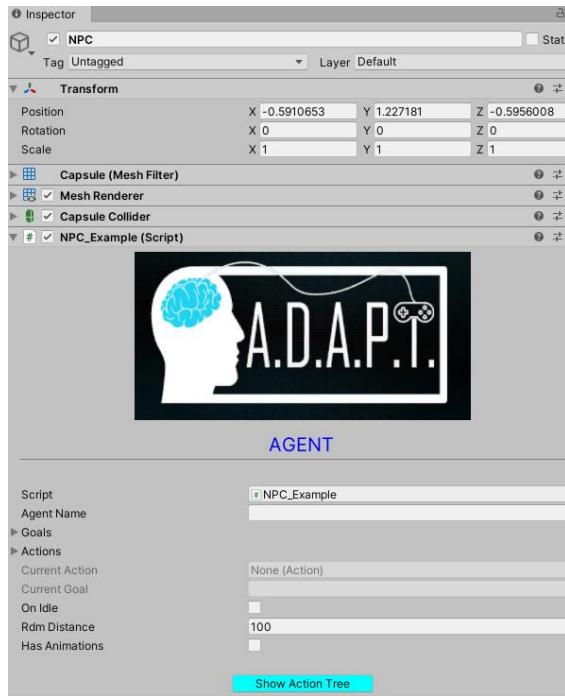
3.- Now in the “Name for the NEW Agent” box, insert the name of the new agent:



*Take care with agents that exists, the tool will not duplicate the scripts, only check if exists.

4.-Click the button “TURN INTO AN AGENT!” and wait some seconds to the new script is being attached to your selected GameObject.

-Your agent should look like these:



5.- Now you have your new agent, with the correct code attached. The new script should look like these in the code editor:

```
1  using UnityEngine;
2  public class NPC_Example : Agent
3  {
4      new void Start() //DON'T MODIFY ANY LINE OF THIS FUNCTION!!!
5      {
6          AddGoals();
7          *****
8          base.Start(); //DON'T DELETE THIS LINE!!!
9          *****
10         ManageStates();
11     }
12
13     public void AddGoals() { }
14
15     public void ManageStates() { }
16
17 }
```

*Format the document if the tabulation is incorrect.

ABOUT AGENT-SCRIPT FUNCTIONS:

-Start() : **Shouldn't be modified.** Initiates the goals, states and all necessary variables from the base class.

-AddGoals() : here you can add your goals to perform by the agent.

Example:

```
InventoryResource goal_2_resource =  
(new InventoryResource("minedGold", 100.0f, 30, 500, false));  
//InventoryResource(GoalName, ValueToAchieve, Priority, Limit,  
isConsumable)  
goals.Add(new Goal(goal_2_resource, false)); //Add the new goal.
```

* Check the actual predefined agent and actions to see the details about add new goals.

-ManageState() : here you can add your agent states. Global or local (like a shared or individual inventory).

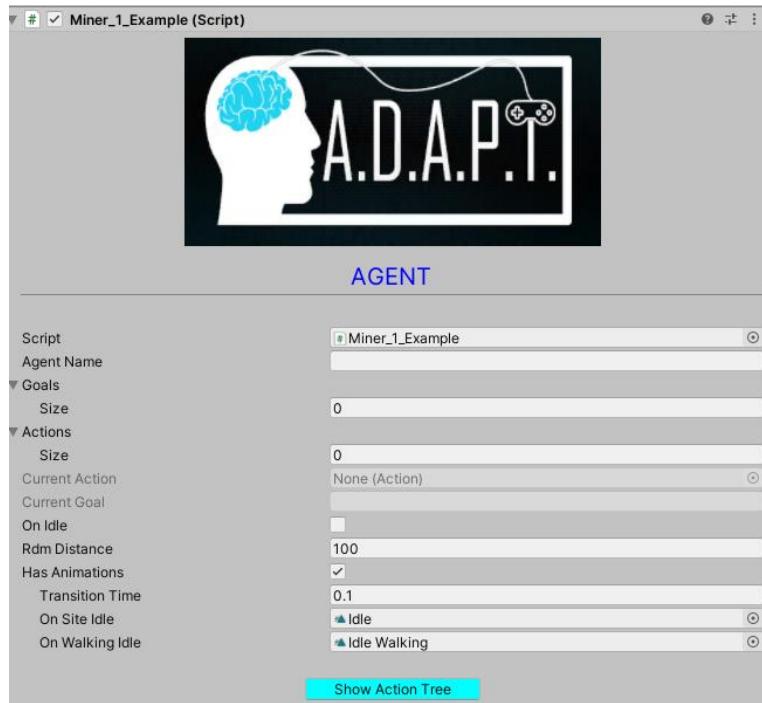
Example:

```
global_states.AddInventoryItem("minedGold", 0f);  
//Add as inventory the actual (Name, initialValue)  
//Use: "global_states" for shared states and "agent_states" for  
individual ones.
```

* Check the actual predefined agent and actions to see the details about add new goals.

4.- About Agents:

Here are some concepts you should know about the Agent:



Goals: check the point “**By code or by inspector**” for see the information relative.

Actions: **YOU SHOULDN'T MODIFY IT!**. This actions are the actions you will attach to the agent, and will appear here as summary.

Current Action & Current Goal: current action performing and goal to achieve.

OnIdle: in case of agent cannot find some plan to achieve, then the agent will enter in a “Idle” mode. If this variable is **true**, the agent will remain on the same site. If its **false**, the agent will move randomly across the map with a maximum of “**Rmd Distance**”.

Rmd Distance: distance Agent can achieve randomly.

ABOUT AGENT ANIMATIONS:

HasAnimations: if you want to have a character with animations, set to **true**.

Transition Time: time between play one animation and the next one.

-OnSiteIdle: Idle on site animation.

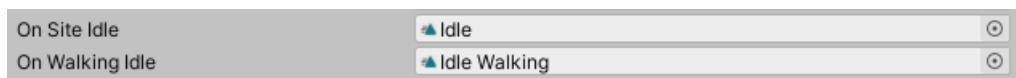
-OnWalkingIdle: walking idle state animation.

* If you use the same animation for the idle states (on the agents inspector) and the Action animation, change the name of one of them.

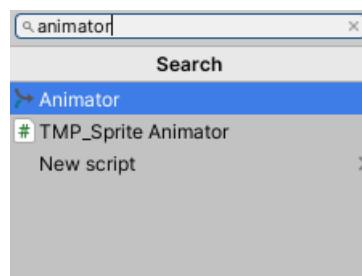
-ADD ANIMATIONS TO AN AGENT:

-If you want to play animations with the actions of the agent then follow the next steps:

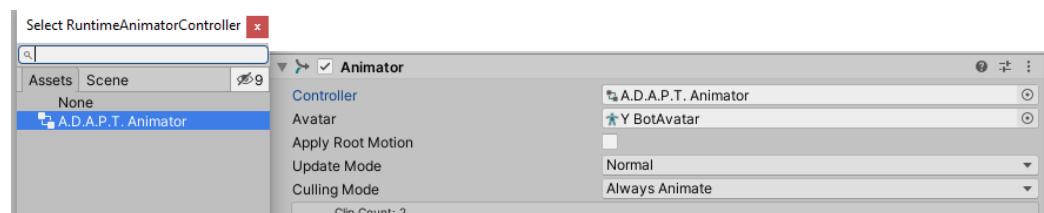
1.- Add Idle animations to the Agent:



2.- Add an Animator to the Agent:



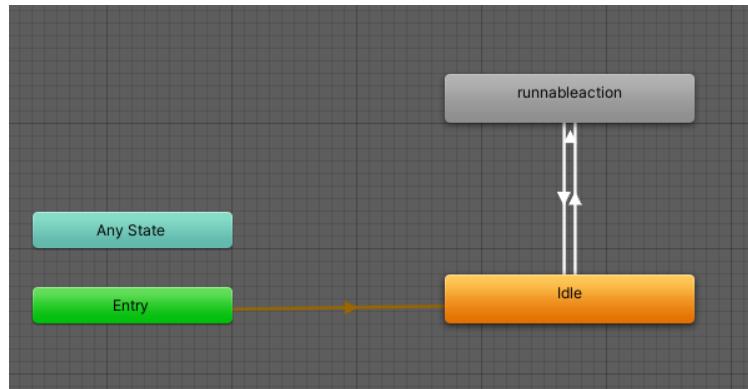
3.- Select the A.D.A.P.T. Animator controller:



Now you have your agent with animations!.

*In the action “Duration” field, you can add a value of duration if you want. By default, this field takes the duration of the animation, but if you insert a higher value, then the animation start a loop.

-ADVICE: DON'T MODIFY THE ANIMATOR. DON'T TOUCH THE ANIMATIONS INSIDE THE STATES. ONLY ADD IT VIA “Action Animation” OF THE DESIRED ACTION TO PLAY!

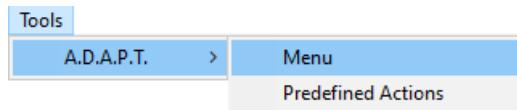


*The Animator use a simple state machine with two states and one loop. So you shouldn't worry about push the new states, A.D.A.P.T. will do for you!.

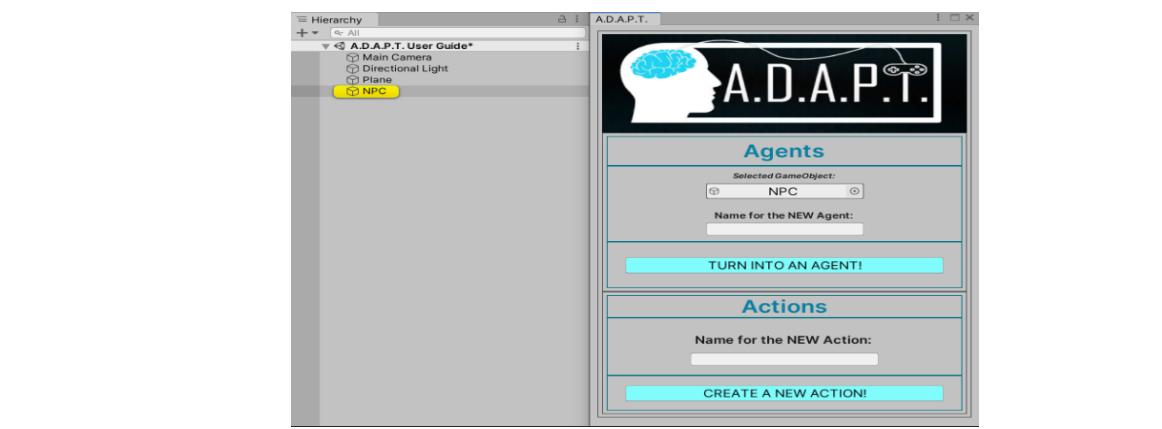
5.- Create new actions :

In case of need the creation of some new actions, you should follow the next steps:

1.- Open A.D.A.P.T. Menu: Tools > A.D.A.P.T. > Menu

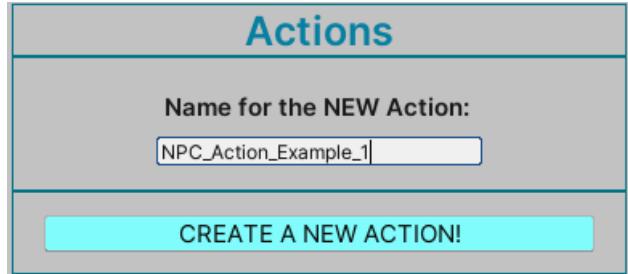


2.- With the Menu open, select your desired GameObject in the hierarchy:



3.-Check if the actual selected GameObject is an Agent. If is not, you should follow the previous steps of the previous chapter.

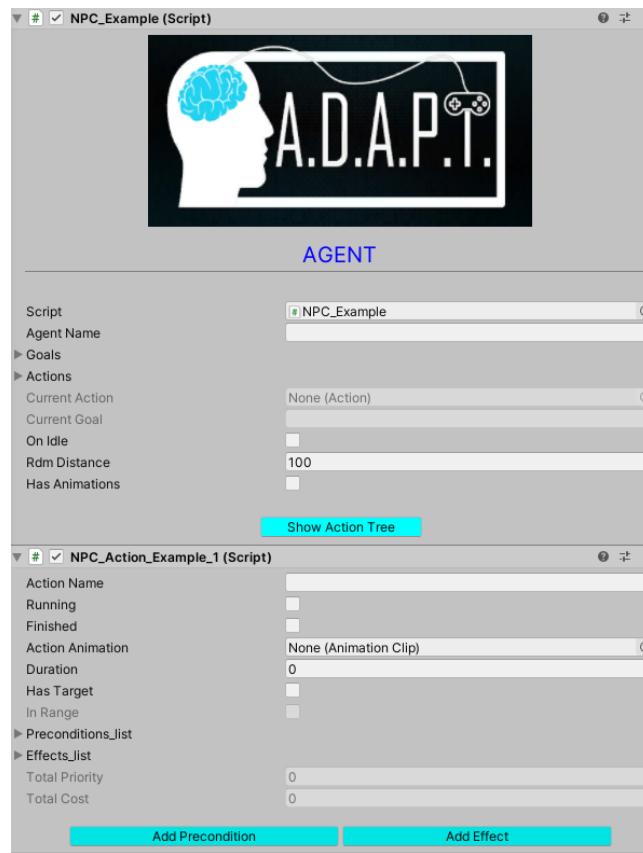
4.- Now in the “Name for the NEW Action” box, insert the name of the new action:



*Take care with actions that exists, the tool will not duplicate the scripts, only check if exists.

4.- Click the button “CREATE A NEW ACTION!” and wait some seconds to the new script is being attached to your selected Agent.

-Your agent should look like these with the new Action:



5.- Now you have your new action, with the correct code attached. The new script should look like these in the code editor:

```

1  using UnityEngine;
2  using UnityEngine.AI;
3  public class NPC_Action_Example_1 : Action
4  {
5      string a_name = "ActionName";
6      Agent agent;
7      NavMeshAgent actual_agent;
8
9      void Awake()
10     {
11         //*****DON'T DELETE THIS LINES!!!*****
12         actionName = a_name;
13         agent = gameObject.GetComponent<Agent>();
14         //WARNING MESSAGE!
15         Debug.Log("<color=blue> Action: </color> " + actionName + " <color=blue> has preconditions / effects added by code,</ color > <color=red> DON'T ADD MORE VIA INSPECTOR!.</color>");
16         //*****
17
18         //HERE YOU CAN ADD YOUR PRECONDITIONS // EFFECTS
19         //*****
20         //In case of add preconditions/effects, uncomment the next lines:
21         //preconditions_list.Add(GoTo_preconditions);
22         //effects_list.Add(GoTo_effects);
23     }
24
25
26     public override void PerformAction()
27     {
28         //Uncomment next line if you need some navmesh:
29         //actual_agent = gameObject.GetComponent<NavMeshAgent>();
30         //Use 'finished = true;' when finish the action.
31     }
32
33
34

```

*Format the document if the tabulation is incorrect.

*You shouldn't delete the lines between comments.

-ABOUT ACTION:



-HasTarget: use it if the actual action need to move to desired location.

***IF ITS TRUE, THEN TARGET CANNOT BE NULL!.**

-Target: desired location to achieve.

***You should add some Precondition of WorldPosition where the resource_value == Target.**

*In case of don't do it, and exists some precondition with resource_value == null and is the tipo of World/Position resource, then it will take the actual Target as value.

-Stop Distance: desired distance from target to stop the Agent navmesh.

-ABOUT AGENT-SCRIPT FUNCTIONS:

-Awake(): only initializes the necessary variables and components.

-PerformAction(): HERE SHOULD BE YOUR CODE TO PERFORM!.

Example:

```
ResourceStruct precondition_1 =  
  
new ResourceStruct("isNear", new WorldResource("isNear", target,  
5, 50.0f  
  
//new ResourceStruct(NameOfResource, new...  
  
//...TypeOfResource(NameOfResource, GameObject, Priority, Limit))  
preconditions_list.Add(GoTo_preconditions);  
  
  
//Preconditions -> preconditions_list // Effects -> effects_list
```

* Check the actual predefined agent and actions to see the details about add new goals.

6.- By code or by inspector :

Actually you can use code or inspector to add the preconditions/effects and goals.

-Preconditions/Effects via inspector:

1.- Click the button “**Add Precondition**” or “**Add Effects**” respectively.

2.- Insert the desired values in the correspondent rows:

▼ minedGold	
Key	minedGold
Selected Type	Inventory Object
▼ I_resource	
Resource Name	minedGold
Resource Enum Type	
Priority	30
Cost	0
Limit	500
Resource_value	100
Is Consumable	<input type="checkbox"/>
Delete	

-Key: name for the desired resource (Effect/Precondition).

-Selected Type: type of the desired resource (*World, Position, Inventory or Status*).

In function of the Selected Type, one resource or another will be displayed: (**W_resource : World, P_resource : Position, I_resource : Inventory, S_resource : Status**).

-Inside resource:

-Resource Name: will copy the text of key.

-Resource Enum Type: type of resource loaded on execution.

-Priority: priority to achieve the actual precondition/effect, **assign more priority for the most important resources.**

-Cost: will be calculated. In case of **World/Position** resources: calculate the distance between the agent and the Target. In case of **Inventory** resources: calculate the cost in function of *resource_Value* amount of inventory, in case of a value of 1000, the cost will be 10 and so on. In case of **Status** resources: the cost will be 1 by default.

-Limit: used for the distance or inventory maximum amount to reach. In case of **World/Position** resources: **the target cannot be more far of the limit distance, or the action will not be part of the planner.** In case of **Inventory** resources: error message will appear if the *resource_Value* is bigger than the limit. In case of **Status** resources: the limit will be 0 by default, not use.



Inventory minedGold in ObtainOre is full!.(Reduce value or increase limit.).
UnityEngine.Debug:Log (object)

-Resource Value: value to reach by states.

-isConsumable: you can use this variable for the consumable inventory resources: like potions, ammo, etc (**You should call it in the PerformAction code**).

* You can use the button “Delete” if you want to delete this specific resource.

-Preconditions/Effects via code:

-For add preconditions/effects via code, you should use the next structure:

ResourceStruct nameOfResource =new ResourceStruct(...);

-And inside the new ResourceStruct:

-Key: the name of precondition/effect.

-Resource:

-In case of World Resource: **new WorldResource:**

new WorldResource("Same as Key" (string), Target (GameObject), Priority (int), Limit (float))

-In case of Position Resource: **new PositionResource:**

new PositionResource("Same as Key" (string), Target (Transform), Priority (int), Limit (float))

-In case of Inventory Resource: **new InventoryResource:**

new InventoryResource("Same as Key" (string), Value (float), Priority (int), Limit (float), isConsumable (bool))

-In case of Status Resource: **new StatusResource:**

new StatusResource("Same as Key" (string), Value (bool), Priority (int))

-The result should view like this:

```
new ResourceStruct(new ResourceStruct("isNear", new  
WorldResource("isNear", target, 5, 50.0f));
```

-Finally add the new precondition/effect, to the actual list for view the new element in the inspector:

```
preconditions_list.Add(nameOfResource); //PRECONDITIONS*  
effects_list.Add(nameOfResource); //EFFECTS*
```

* Add to the list in the [Awake\(\)](#) function.

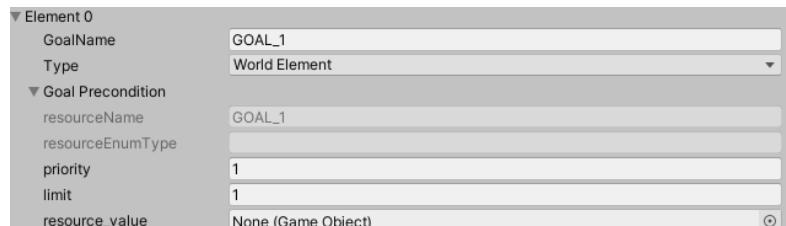
****ADVICE:** IN CASE OF ADD PRECONDITIONS/EFFECTS ONLY BY CODE AND NOT BY INSPECTOR, YOU SHOULD INSERT SOME ACTION THAT IMPLIES TO CHANGE THE VALUES OF STATES TO REACH THE PRECONDITION/EFFECT.

-Goals via inspector:

1.- Increase the size of Goals by +1.



2.- Insert the desired values in the correspondent rows as you do with preconditions/effects



-Extra rows:

[-hasAction:](#) in case of the goal should play some animation when is reached.

[-goal Action:](#) animation to play.

[-cost:](#) calculated same as preconditions/effects.

-Goals via code:

-In the Agent script ([inside function AddGoals\(\)](#)), you should use the next structure:

```
Resource nameOfResource = new Resource(...);
```

*Where resource can be: **WorldResource**, **PositionResource**, **InventoryResource**, **StatusResource**.

-And inside the new Resource:

-Resource:

-In case of World Resource: **new WorldResource:**

new WorldResource("Same as Key" (string), Target (GameObject), Priority (int), Limit (float))

-In case of Position Resource: **new PositionResource:**

new PositionResource("Same as Key" (string), Target (Transform), Priority (int), Limit (float))

-In case of Inventory Resource: **new InventoryResource:**

new InventoryResource("Same as Key" (string), Value (float), Priority (int), Limit (float), isConsumable (bool))

-In case of Status Resource: **new StatusResource:**

new StatusResource("Same as Key" (string), Value (bool), Priority (int))

-The result should view like this:

```
InventoryResource goal = (new InventoryResource("minedGold",  
100.0f, 30, 500, false));
```

For other side, **YOU CAN ONLY USE CODE TO ADD THE STATES:**

-In the Agent script (**inside function ManageStates()**), you should use the next structure:

-For shared states (GLOBAL): global_states

-For individual states (LOCAL): agent_states

-In this case, you have 4 type of items:

-worldElements: for World resources.

-positions: for Position resources.

-inventory: for Inventory resources.

-status: for Status resources.

-And you can use the next functions for **Add, Remove, Modify or Increase/Decrease** the values of the states:

**AddWorldItem(string state),
AddPositionItem(string state),
AddInventoryItem(string state, float initialValue),
AddStatusItem(string state, bool initialValue):** for initialize a state.

**RemoveWorldItem(string state),
RemovePositionItem(string state),
RemoveInventoryItem(string state),
RemoveStatusItem(string state):** for remove a state.

**ModifyInventoryItem(string state, float newValue),
ModifyStatusItem(string state, bool newValue) :** for modify the current value of a state.

IncreaseInventoryItem(string state, float newValue), DecreaseInventoryItem (string state, bool newValue) : subtract/add operations for inventory resources only.

* Except "Add" functions the other should be use in the PerformAction() function to modify the states as you desire.

-The result should view like this:

```
global_states.AddInventoryItem = ("minedGold", 0f);
```

* This is because the use of a unique instance to generate the global world state.

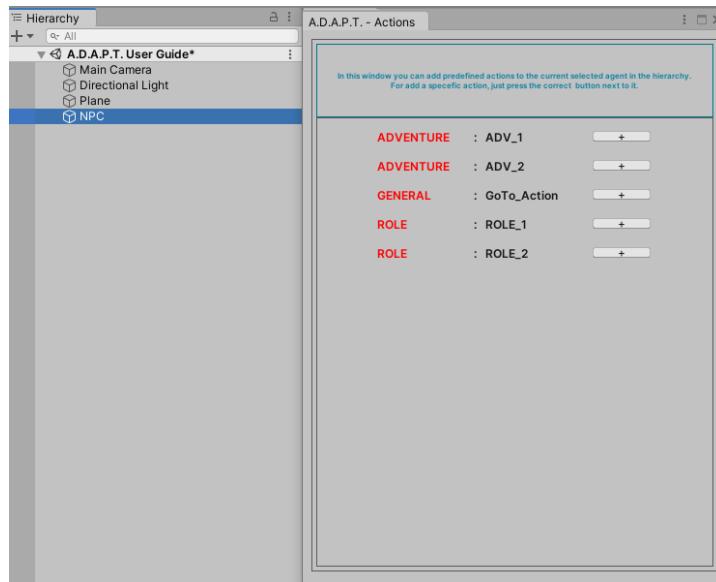
7.- Predefined Actions :

To add some of the predefined action to an actual agent (*the selected GameObject should be an Agent, if is not, then convert into one*), you should follow the next steps:

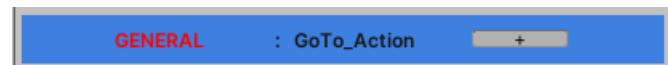
1.- Open A.D.A.P.T. Menu: **Tools > A.D.A.P.T. > Predefined Actions**



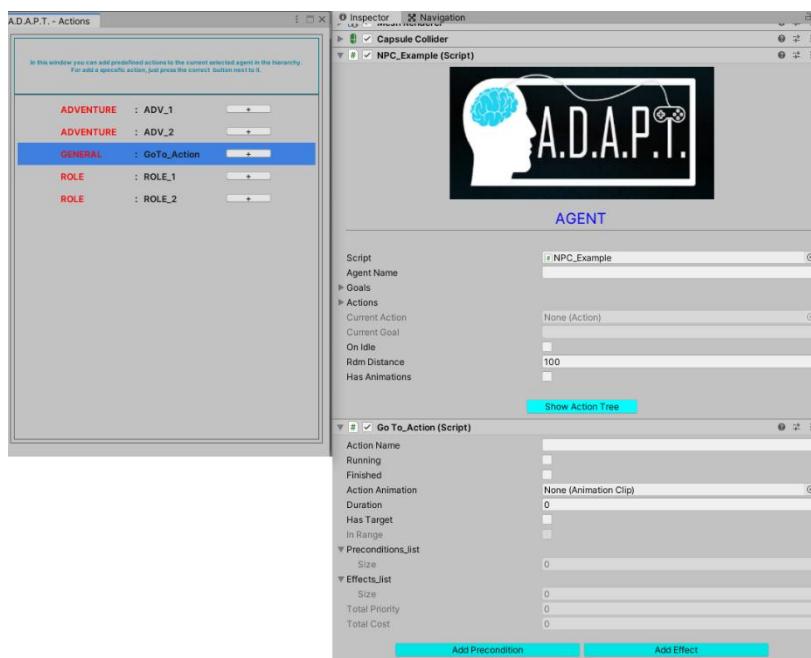
2.-With the Menu open, select your desired GameObject in the hierarchy:



3.-Click on the button “**+**” of the desired Predefined Action you want to add to the actual selected agent.



4.-Wait to the action is being added to the agent:

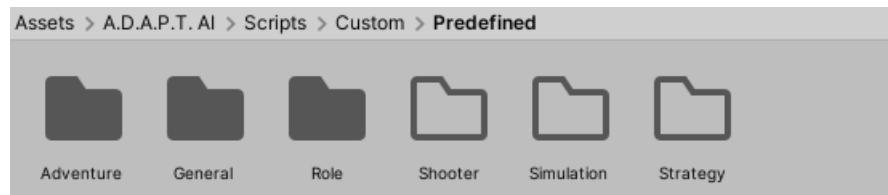


-Add your own Predefined Actions!:

You can add your own Predefined Actions to the Menu by moving your desired action to the path:

Assets/A.D.A.P.T. AI/Scripts/Custom/Predefined

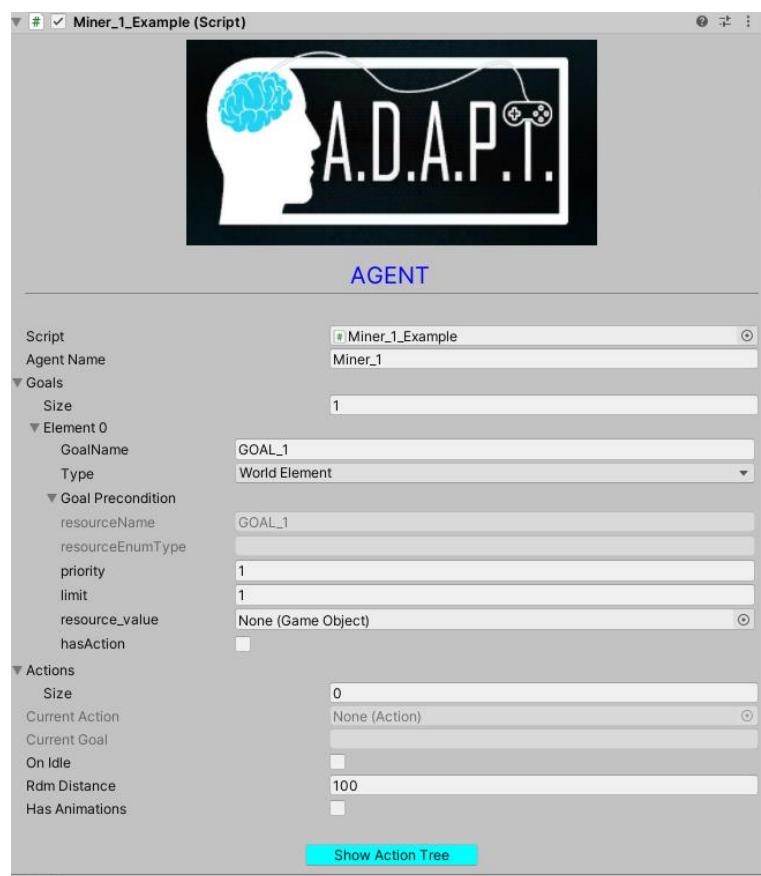
Then just throw your Predefined Action to one of the Game Genres:



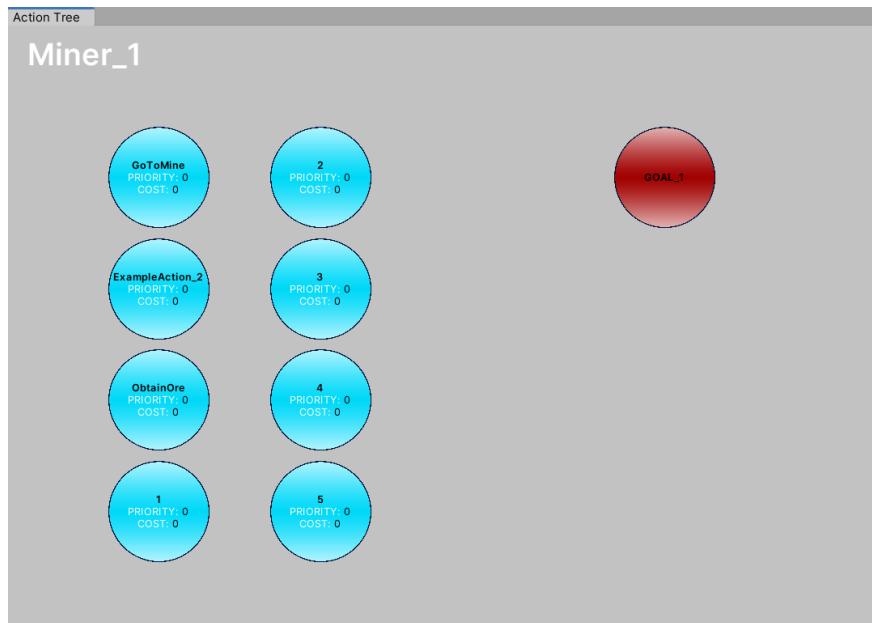
8.- Action Tree:

If you want to see the actual Action Tree of some specific created Agent (who you add some actions previously), follow the next steps:

1. Select some created Agent to see it in the inspector:



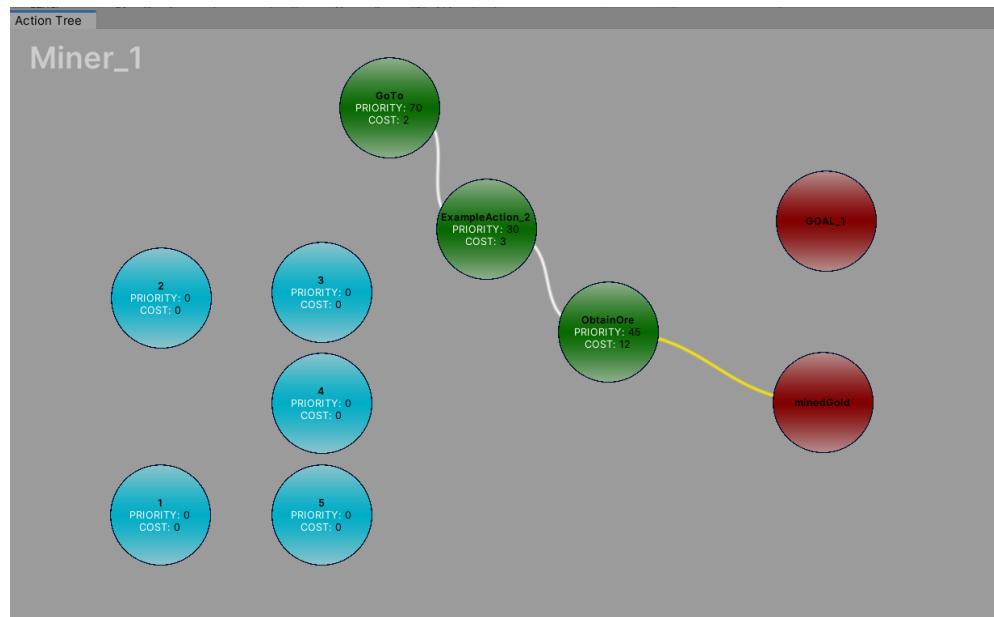
2.- Click the “Show Action Tree” button:



-In this case you will see the added actions and the goals added by inspector.

* If a goal is added by code it will be show in execution time.

3.- On execution time:



-You will see the actual Action plan to reach the possible goal:

-White lines: relationates actions.

-Yellow lines: relationates actions with the goal.

-Blue nodes: not achieve actions.

-Green nodes: actions who perfom the plan.

-Red nodes: goals.

* Nodes are draggable windows, so you can move it to any position if you want.

9.- In case of bug:

In case of any bug you can report it to my email: yagomira@gmail.com

Otherwise, here are some frequently bugs and solutions:

1.-In case of some error of “Index 0” caused by some relationated with Animations:

Close the Inspector window (A.D.A.P.T. and Unity also).

If the bug persists, restart the engine.

2.- Animation of the agent/action don’t play:

-Maybe you use the same animation for the idle states (on the agents inspector) and the Action animation, change the name of one of them.

-Add animator if you don’t have one (remind to select the specific **A.D.A.P.T. Controller**).

-Otherwise, check if you modify the “T-Pose” of the Agent Animator, **YOU CANNOT CHANGE THIS ANIMATION!**

I hope this User Guide will be useful for you ☺. Thanks for use the tool!

Sincerely,

Yago, developer of A.D.A.P.T. - AI Tool.

Lista de acrónimos

AI Artificial Intelligence. [1](#)

CRUD Create, Read, Update, Delete. [27](#), [32](#)

GOAP Goal-Oriented Action Planning. [1](#), [2](#), [4–6](#), [8](#), [10](#), [27](#), [28](#), [30](#), [35](#), [44](#), [50](#), [54](#), [55](#), [57](#), [58](#), [62](#), [67](#), [76](#)

IA Inteligencia Artificial. [iv](#), [1](#), [4](#), [8](#), [12](#)

NPC Non-Playable Characters. [1](#), [4](#), [5](#), [34](#), [59](#)

PNJ Personajes No Jugadores. [1](#), [2](#), [6](#), [8](#)

RPG Role-Playing Game. [8](#)

SWOT Strengths-Weaknesses-Opportunities-Threats. [7](#)

Glosario

A* algoritmo de búsqueda de tipo heurístico o informado. Trata de buscar el camino de menor coste entre un nodo origen y uno objetivo. [2](#), [18](#), [54](#), [55](#), [57](#), [61](#), [76](#)

Animator interfaz que forma parte del módulo AnimationModule de Unity y que permite controlar su sistema de animación. [12](#)

assets todo aquel objeto que puede ser usado en un videojuego, comprende tanto: ficheros de audio como modelos 3D, sprites, texturas, etc... [7](#)

Behavior Tree modelo matemático de ejecución de planes que permite modelar el comportamiento de los personajes que no son jugadores mediante la utilización de un árbol de nodos dirigido. [4](#)

búsqueda en anchura algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo. Se comienza en la raíz y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol. [61](#)

frame fotograma o imagen completa presentada en la pantalla del jugador. Usualmente se relaciona con el término framerate que consiste en la cantidad de fotogramas por segundo que se llegan a visualizar. [52](#)

GitHub servicio de alojamiento para el desarrollo de software y el control de versiones mediante Git. [9](#)

hierarchy ventana de Unity que despliega todos y cada uno de los GameObject instanciados en una Scene. [68](#), [72](#), [75](#)

historias de usuario artefactos utilizados en prácticas ágiles para levantar los requerimientos. Se entiende por requerimiento aquellas funcionalidades que un producto software debe cumplir. [15](#), [24](#), [26](#)

Máquina de estados también denominada autómata finito, es un modelo computacional que permite realizar cálculos de manera automática sobre una entrada para producir una salida. [12](#), [19](#), [65](#), [66](#), [70](#), [76](#)

NavMesh clase del módulo AIModule de Unity que permite dotar a los objetos de propiedades como pathfinding y movimiento a través de un espacio. [12](#), [65](#)

pathfinding consiste en el trazado realizado por una aplicación de computadora, del camino más corto entre dos puntos, usualmente haciendo referencia a un ábol de nodos. [v](#), [6](#), [12](#), [25](#), [52](#), [54](#), [55](#), [76](#)

quest también conocidas como misiones, son tareas dentro de un videojuego que un jugador o un grupo de ellos debe completar para obtener una recompensa. [8](#)

Redes Neuronales método computacional que se basa en la interconexión de nodos pero realizada de una manera similar a la que tendrían las neuronas en el cerebro humano. [4](#)

scene Las escenas contienen los objetos de un videojuego que use el motor Unity. Se pueden usar para crear un menú principal, niveles individuales y cualquier otra cosa. Cada escena consta de un único archivo.. [30](#), [51](#), [65](#)

shooter género de videojuegos que engloba a un amplio subgrupo con la característica particular de que todos ellos controlan a un personaje que dispone de un arma que puede ser usada a voluntad del jugador. [8](#)

sprints hacen referencia a cada uno de los ciclos o iteraciones que tendrá un proyecto que utilice la metodología Scrum. Permite obtener un ritmo de trabajo con un tiempo prefijado. [3](#), [15](#), [16](#), [18](#), [20](#), [24](#), [25](#), [27](#), [49](#), [77](#), [96](#), [107](#)

unity es un motor de videojuegos multiplataforma creado por Unity Technologies. [1](#), [2](#), [7](#), [11](#), [12](#), [30](#), [39](#), [41](#), [44–47](#), [61](#), [64](#), [68](#), [70](#), [76](#)

Unity Asset Store librería de recursos gratuitos y comerciales para videojuegos, creada tanto por miembros de la comunidad como la propia Unity Technologies. [7](#), [10](#)

Unity Editor se le denomina editor a la propia interfaz del motor, es decir, al conjunto de todas las pantallas, permite que se le añadan nuevas ventanas y menús con el objetivo de añadir nuevas funcionalidades. [12](#), [39](#)

Unity Inspector es parte de la interfaz del motor de Unity, permite ver y editar las propiedades de los diferentes objetos del mundo incluyendo los GameObjects, assets, etc. [12](#), [44](#), [67](#), [68](#), [70](#)

árbol de nodos tipo abstracto de datos que imita la estructura jerárquica de un árbol, con un valor en la raíz y subárboles con un nodo padre, representado como un conjunto de nodos enlazados. Se considera un nodo como una estructura de datos con un valor. [6](#)

Bibliografía

- [1] “Goal-oriented action planning research,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://jaredemitchell.com/goal-oriented-action-planning-research/>
- [2] “Libro blanco del desarrollo español de videojuegos 2021,” 2021, consultado el 2022-09-10. [En línea]. Disponible en: <https://dev.org.es/images/stories/docs/libro%20blanco%20del%20desarrollo%20espanol%20de%20videojuegos%202021.pdf>
- [3] “Ingresos totales de la industria musical en 2021,” 2021, consultado el 2022-09-10. [En línea]. Disponible en: <https://forbes.es/economia/154190/el-streaming-impulsa-la-musica-los-ingresos-del-sector-crecen-un-185-en-2021/>
- [4] “Recaudación de taquilla a nivel mundial en 2021,” 2021, consultado el 2022-09-10. [En línea]. Disponible en: <https://es.statista.com/estadisticas/600690/ingresos-de-taquilla-a-nivel-mundial/>
- [5] “Behavior trees: Three ways of cultivating game ai,” 2010, consultado el 2022-09-10. [En línea]. Disponible en: https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc10/slides/ChampandardDaweHernandezCerpa_BehaviorTrees.pdf
- [6] “Breakdown: The ai of total war,” 2018, consultado el 2022-09-10. [En línea]. Disponible en: <https://80.lv/articles/breakdown-the-ai-of-total-war/>
- [7] “Exposición de la gdc : Ten years of ai programming,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://www.gdcvault.com/play/1022020/Goal-Oriented-Action-Planning-Ten>
- [8] “Web de jeff orkin,” 2011, consultado el 2022-09-10. [En línea]. Disponible en: <https://alumni.media.mit.edu/~jorkin/goap.html>
- [9] “Symbolic representation of game world state,” 2004, consultado el 2022-09-10. [En línea]. Disponible en: <https://alumni.media.mit.edu/~jorkin/WS404OrkinJ.pdf>

- [10] “Applying goal-oriented action planning to games,” 2003, consultado el 2022-09-10. [En línea]. Disponible en: https://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf
- [11] “Unity asset store,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://assetstore.unity.com/>
- [12] “Quest machine asset,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://assetstore.unity.com/packages/tools/game-toolkits/quest-machine-39834>
- [13] “Intense shooter asset,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://assetstore.unity.com/packages/tools/ai/intense-shooter-ai-63221>
- [14] “Sgoap asset,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://assetstore.unity.com/packages/tools/ai/s-goap-ai-solution-167167>
- [15] “Goal oriented action planning asset,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://assetstore.unity.com/packages/tools/ai/goal-oriented-action-planning-artificial-intelligence-72912>
- [16] “Regoap asset,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://assetstore.unity.com/packages/tools/ai/regoap-77376>
- [17] “Página web del motor unity,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://unity.com/es>
- [18] “Videojuegos que usan unreal engine,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://www.unrealengine.com/en-US/solutions/games>
- [19] “Api de unity sobre los gameobject,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/es/530/Manual/class-GameObject.html>
- [20] “Github,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://github.com/>
- [21] “Trello,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://trello.com/>
- [22] “Draw.io,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://draw.io/>
- [23] “Draw.io extensión diagramas gantt,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://drawio-app.com/increase-productivity-with-gantt-charts-in-draw-io/#>
- [24] “Balsamiq,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://balsamiq.com/>

BIBLIOGRAFÍA

- [25] “Visual studio,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://visualstudio.microsoft.com/es/>
- [26] “Salario medio de un analita programador,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://es.talent.com/salary?job=analista+programador#:~:text=El%20salario%20analista%20programador%20promedio,hasta%20%E2%82%AC%2039.000%20al%20a%C3%B1o.>
- [27] “Api de unity sobre los monobehaviours,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>
- [28] “Api de unity sobre los transform,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/ScriptReference/Transform.html>
- [29] “Api de unity sobre los vector3,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/ScriptReference/Vector3.html>
- [30] “Gangs of four (gof) design patterns,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://www.journaldev.com/31902/gangs-of-four-gof-design-patterns>
- [31] R. Nystrom, “Game programming patterns,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <http://gameprogrammingpatterns.com/>
- [32] “Api de unity sobre las serializedproperty,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/ScriptReference/SerializedProperty.html>
- [33] “Api de unity sobre awake,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html>
- [34] “Api de unity sobre start,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html>
- [35] “Api de unity sobre lateupdate,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.LateUpdate.html>
- [36] “Taxicab geometry (manhattan),” 2022, consultado el 2022-09-10. [En línea]. Disponible en: https://en.wikipedia.org/wiki/Taxicab_geometry
- [37] “Api de unity sobre las animations,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/es/530/ScriptReference/Animation.html>
- [38] “Api de unity sobre animatoroverridecontroller,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/ScriptReference/AnimatorOverrideController.html>

BIBLIOGRAFÍA

- [39] “Api de unity sobre coroutine,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/ScriptReference/Coroutine.html>
- [40] “Api de unity sobre información relativa al uso de los ienumerator,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.StartCoroutine.html>
- [41] “Api de unity sobre rect,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/ScriptReference/Rect.html>
- [42] “Api de unity sobre hierarchy,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://docs.unity3d.com/Manual/Hierarchy.html>
- [43] “Mixamo: web de modelos 3d,” 2022, consultado el 2022-09-10. [En línea]. Disponible en: <https://www.mixamo.com/>