Diseño de una Arquitectura de Sistemas Multiagente para Videojuegos basada en el modelo de Creencias, Deseos e Intenciones en Unity

Jaime María Bas Domínguez
Álvaro Cuevas Álvarez
Alejandro García Montero
Juan Gómez-Martinho González
Irene González Velasco

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE INGENIERÍA DEL SOFTWARE
E INTELIGENCIA ARTIFICIAL

UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de Fin de Grado
Grado en Ingeniería Informática
Grado en Ingeniería del Software

Director: Prof. Dr. Federico Peinado Gil

Madrid, 31 de mayo de 2019

Agradecimientos

A mis amigos, que me han animado a seguir todos estos años y se han dejado la piel para sacar todo esto adelante. A mis padres, por no dejar nunca de animarme. Y a María, por estar ahí siempre que la necesito, incluso en mis peores momentos.

Jaime Bas

Me gustaría agradecer a todos mis amigos y compañeros que he conseguido durante todos estos años. Gracias a ellos esta etapa en la universidad ha sido de las mejoras de toda mi vida. Y gracias a mi familia porque sin ellos esto no hubiese sido posible.

Álvaro Cuevas

A mis padres, por sus consejos y opiniones sobre que debería hacer cuando tenía dudas, y por reírse conmigo de mis desventuras en este proyecto haciéndolo más llevadero y divertido. Y a mi perro, porque en el fondo, pero en el fondo, fondo, es un buen chico.

Alejandro García

A mis padres, por sus consejos y apoyo incondicional. Agradecer también al HUB de Innovación su apoyo y las facilidades que me han dado para completar éste trabajo, sin las cuales se me hubiera hecho imposible finalizarlo a tiempo. Por último, una mención especial a David Muñoz, cuya paciencia y apoyo durante incontables noches en vela han sido cruciales para llevar a buen término el proyecto.

Juan Gómez-Martinho

A mis padres y hermano, porque sin su apoyo nunca hubiera llegado hasta aquí. Para mis amigos, especialmente Rafa, Elena y Aza, por aguantarme en esas conversaciones eternas cada vez que tenía dudas o quería abandonar.

Irene González

Agradecer a los integrantes de Narratech Laboratories su conocimiento y disponibilidad a la hora de resolver dudas, ya que con ello han hecho posible conseguir un código robusto y funcional.

Los autores

Resumen

Si bien existen multitud de aproximaciones técnicas para desarrollar Inteligencia Artificial, la simulación de personajes con comportamiento realista sigue siendo un gran reto para los desarrolladores de videojuegos, sobre todo si se quieren evitar inabarcables máquinas de estados o árboles de comportamiento.

Desde el punto de vista del diseñador, sería más sencillo olvidarse de predecir y programar todas las situaciones posibles y diseñar en torno a tres aspectos fundamentales de cada personaje: qué es lo que cree sobre su entorno, qué metas desea alcanzar y cuáles son sus intenciones más inmediatas, representadas en forma de planes de actuación. Trabajar bajo este paradigma podría dar lugar a videojuegos dotados de una narrativa emergente con gran potencial.

Basándonos en dos Trabajos de Fin de Grado anteriores, tomamos como referencia para este proyecto a Jason, el intérprete más popular para implementar sistemas multiagente basados en el modelo de Creencias, Deseos e Intenciones (BDI) implementado en Java y apoyado en el lenguaje declarativo *AgentSpeak*. Hasta ahora, lo máximo que han logrado otros proyectos es usar Jason como un servidor externo de razonamiento para los agentes, pero esto sigue suponiendo muchos problemas para el programador, como son los tediosos protocolos de comunicación vía *sockets*, o el manejo, ejecución y depuración de varios entornos de desarrollo simultáneos.

El objetivo de nuestro proyecto es agilizar todo el trabajo del programador, permitiendo a la vez diseñar personajes más "humanos" con menor esfuerzo, buscando generar experiencias interactivas más ricas e interesantes. Para ello este trabajo pretende llevar las ideas de Jason a un sistema interno al propio videojuego, con una arquitectura BDI implementada en C# e integrada dentro del entorno de desarrollo de Unity.

Se ha llevado a cabo el análisis y la reingeniería de Jason, así como el posterior diseño e implementación de "Jasonity", un sistema multiagente BDI para Unity inspirado en Jason, comprobando mediante pruebas de integridad que el planteamiento es viable como propuesta para programar agentes sencillos, útiles para dar soporte al comportamiento de personajes de videojuegos y de uso más fácil y natural para el programador.

Palabras clave

Desarrollo de Videojuegos, Inteligencia Artificial, Sistemas Multiagente, Ingeniería del Software, Razonamiento Lógico, Diseño de Sistemas Interactivos.

Design of an Architecture for Multi-Agent Systems for Video Games based on the Belief, Desire and Intentions model in Unity

Abstract

Although there are many technical approaches to develop Artificial Intelligences, simulating characters with human-like behavior is still a huge challenge for video game developers, especially if you want to avoid unmanageable state machines or behavior trees.

From a designer's point of view, it would be easier to forget about predicting and programming every possible situation and designing around three fundamental aspects for each character: what they believe about their environment, which goals do they want to reach and what their most immediate intentions are, represented as acting plans. Working under this paradigm could mean having video games with emergent narrative with huge potential.

Taking two previous Final Degree Projects as a starting point, we took Jason as reference for this project. Jason is the most popular interpreter for implementing multiagent systems based on the Belief, Desire and Intention (BDI) model, implemented in Java and supported by the declarative language *AgentSpeak*. Until now, the further other projects have gone is using Jason as an external server for the agents' reasoning, but this is still troublesome for the programmer, just like the tedious communication protocols through *sockets* or the handling, execution and debugging of several, simultaneous environments.

The goal of our project is to speed up the programmer's work, allowing them to design more "human-like" characters with less effort, looking to generate richer and more interesting interactive experiences. To do this, this project strives to take the concepts in Jason to an internal system in the video game itself, with a BDI architecture implemented in C# and integrated inside the Unity environment.

We have conducted an analysis and reengineering process on Jason, as

well as the subsequent design and implementation of “Jasonity”, a multi-agent system based on BDI for Unity inspired by Jason, verifying through integrity tests that the proposal is viable for programming simple agents, useful to support character behavior in video games, and easier and more natural to use for the programmer.

Keywords

Video Game Development, Artificial Intelligence, Multi-Agent Systems, Software Engineering, Logical Reasoning, Interactive Systems Design

Índice

Índice de figuras

Capítulo 1

Introducción

"The cake is a lie"
— *Portal*. Valve (2007)

¿Realmente existe la Inteligencia Artificial (IA) en los videojuegos? ¿O simplemente se ha especializado el uso de otras técnicas más básicas que dotan a los personajes de una "inteligencia aparente"?

Si nos sumergimos en un mundo virtual por primera vez y **no nos fijamos demasiado en los detalles podemos llegar a pensar que todo lo que nos rodea está "vivo"**. Aunque en prácticamente todos los videojuegos actuales con personajes no jugables o NPCs (del inglés, *Non Playable Characters*) puede percibirse esa **apariencia de vida e inteligencia**, las probabilidades de que en realidad la implementación subyacente sea poco más que un conjunto de objetos de juego con un guión (*script*) predefinido son muy altas.

Por otro lado, cada año que pasa **los jugadores son más exigentes** con los productos que consumen, y los detalles se han convertido en un elemento clave que puede suponer el éxito o el fracaso de un título. Es por ello que los estudios creadores de videojuegos se esfuerzan por ofrecer **experiencias cada vez más sofisticadas e inmersivas**; y eso incluye generar un mundo y unos personajes que lo habitan que estén lo más vivos posible.

Sirva como ejemplo "The Last Guardian", genDesign; SCE Japan Studio (2016); tras casi diez años de desarrollo. Desde el minuto uno tras su esperado lanzamiento, críticos, usuarios y los propios desarrolladores coincidieron en alabar lo mismo: la IA detrás de Trico, la criatura mitológica que acompaña al protagonista a lo largo de su aventura, **hace que el jugador sienta que realmente está interactuando con una entidad con vida propia**.

Por otro lado, es cierto que hay títulos, como los de **Bethesda, que nunca han destacado por contar con unos NPCs especialmente inteligentes**. Si bien es cierto que en su más reciente título, *"Fallout 76"* (Bethesda Game Studios, 2006), no hay presencia de IAs que representen personajes "humanos", multitud de jugadores culpabilizan a los desarrolladores del comportamiento errático de las criaturas que habitan ese mundo. Si miramos a uno de sus títulos más icónicos, *"The Elder Scrolls IV: Oblivion"* (Bethesda Game Studios, 2018), los personajes que pueblan el enorme mapa del juego destacan por sus tremendas carencias en IA; cosa que lleva al jugador a vivir situaciones de lo más surrealistas. A pesar de los 12 años que distan entre ambos videojuegos, **no parece que este problema haya mejorado mucho**.

Y es que desarrollar una IA realista no es una tarea sencilla. A día de hoy sigue siendo **objeto de debates, estudios e investigaciones**. Implementar un comportamiento creíble y semejante al de un ser humano puede resultar muy caro: suele conllevar **muchísimas horas de trabajo** que no siempre van a verse recompensadas en términos de experiencia, porque existen otros apartados del juego (jugabilidad, narrativa, gráficos...) que resultan más atractivos para el consumidor y en cierto sentido, más fáciles de desarrollar.

En nuestro caso, creemos que **buena narrativa e IA son dos elementos que deberían ir de la mano**. Los personajes creíbles e inteligentes son elementos muy importantes para generar una historia rica en detalles capaz de atrapar al jugador desde el primer momento. Para ello debemos escapar de los guiones predefinidos y dejar **que sea el propio juego el que evolucione y se adapte al jugador** por sí mismo, permitiendo que sea este último el que escriba su propia historia y dejando hacer a los personajes que pueblan el mundo, para que respondan "por sí mismos" a dichas decisiones interactuando libremente con el entorno. Esto es lo que implica el abrazar el concepto de Narrativa Emergente y para lo que nos hemos embarcado en este proyecto.

Para permitir esa flexibilidad y autonomía en los personajes, nuestra propuesta se basa en una forma de construir agentes autónomos conocida como Arquitectura **BDI** (del inglés, *Belif-Desire-Intention*). Ésta permite definir cada agente con un **conjunto de Creencias** (información, cierta o no, percibida de forma directa o indirecta, que posee el agente sobre el entorno en el que habita), **Deseos** (metas que desea ver cumplidas) e **Intenciones** (generalmente planes, secuencias de acciones que debe realizar para alcanzar sus deseos teniendo en cuenta sus creencias sobre el entorno).

A través de este modelo, cada **Agente BDI es capaz de percibir hechos de su propio entorno, generar planes en tiempo real e incluso**

ir variando sus objetivos más inmediatos en función de la información que posee para satisfacer sus deseos principales. Siguiendo este paradigma cada NPC se diseñaría como un personaje **realmente vivo** con sus propios pensamientos, y el *technical designer* sería dotado de herramientas para conseguir esto, siempre dejando al "libre albedrío" de dicho personaje la manera exacta en que va a tratar de alcanzar sus objetivos; olvidándonos (como dijimos al principio) de programar todo al detalle, guionizar su comportamiento minuciosamente y generar una **máquina de estados o un árbol de decisión o de comportamiento totalmente inmanejable** (como llega a ocurrir en ocasiones) o incluso artefactos pesados e incapaces en última instancia de predecir todas las posibles situaciones que van a producirse en un juego real.

Según lo estudiado como parte de este trabajo, **no existe una herramienta estándar** para trabajar con éste tipo de IA para personajes, que esté disponible al menos en los principales entorno de desarrollo de videojuegos. Y de ahí que el **propósito de nuestro proyecto** sea contribuir al desarrollo de dicha herramienta.

1.1. Propósito del trabajo

Partimos de las premisas descritas anteriormente para desarrollar el siguiente **Trabajo de Fin de Grado**, que pretende **ofrecer una solución** a la carencia de plataformas de **desarrollo de sistemas multiagente con enfoque BDI** para videojuegos.

Para ello, llevamos a cabo una investigación sobre el **estado actual de los paradigmas de programación de IA** a día de hoy, sus ventajas y desventajas, posibles implementaciones alternativas y, finalmente, el **desarrollo de la herramienta** propiamente dicha. Así, todos los **conocimientos adquiridos** a lo largo de la carrera, algunos adquiridos por cuenta propia y otros tantos adquiridos durante la elaboración de éste trabajo **quedan reflejados y sintetizados** en el proyecto.

Como se desarrollará *a posteriori*, el trabajo consta de tres etapas:

1. **Investigacion**: Se ha realizado un análisis profundo de las **técnicas y herramientas** disponibles actualmente para la programación de Agentes BDI, seleccionando las mejores opciones para conseguir nuestro objetivo, al tiempo que analizamos si este es viable. Se ha partido de trabajos anteriores y artículos científicos que pueden encontrarse en la literatura, así como desarrollos de código abierto que pudieran, a primera vista, completar el objetivo deseado. Tras este análisis se ha

pasado a una etapa de desarrollo con el fin de obtener una solución que satisficiese las funcionalidades requeridas.

2. **Desarrollo**: Una vez cribadas nuestras opciones, se ha decidido **portar gran parte de la funcionalidad del sistema Jason** al lenguaje C# y al entorno tan usado en videojuegos de Unity. Esto ha supuesto un trabajo de reingeniería considerable, siendo trabajadas las más de 24.000 líneas de código y 250 clases no anónimas que podemos encontrar en la plataforma original. Dadas las sutiles diferencias entre ambos lenguajes de programación, las en ocasiones drásticas diferencias en los entornos de desarrollo y principalmente la falta de documentación sobre todo el sistema, ha sido necesario **crear también una guía para la migración de proyectos de Java a C# y Unity**. Dados los motivos que se expondrán en el capítulo 5, hubo que replantear el proceso de desarrollo dando lugar a una nueva fase de "desarrrollo 2.0".

3. **Pruebas**: Para ir realizando pruebas graduales del funcionamiento de la nueva plataforma implementada, se han ideo creando tests automáticos que chequean diversos aspectos operativos del sistema resultante y **verifican que los objetivos propuestos han sido alcanzados**. Además, se ha implementado una demo sencilla para mostrar visualmente la aplicación de nuesto sistema multiagente en un escenario de Unity.

1.2. Estructura del trabajo

Lo explicado hasta ahora y el flujo del trabajo realizado se desarrolla en profundidad a lo largo de los siguientes capítulos, incluyendo las conclusiones obtenidas y desgranando las aportaciones de cada miembro del equipo a este proyecto común.

Tras este capítulo introductorio, en el **Capítulo 2 se expone todo el trabajo de investigación** realizado para determinar qué capacidades tienen actualmente las herramientas de desarrollo de IA para personajes, en particular las que sigan una arquitectura BDI. Ofrece también una explicación sobre qué entendemos como razonamiento y como agente, qué es y cómo funciona la Arquitectura BDI, detalla la plataforma en la que nos basaremos para implementar nuestra solución y expondremos ejemplos de IA en videojuegos en la industria, junto con una breve explicación de los motores de desarrollo más populares.

En el **Capítulo 3**, teniendo ya claro cual es el estado de la técnica hoy por hoy, se detalla tanto el **alcance de nuestro proyecto como los objetivos concretos** que se persiguen, además de realizar una **especificación**

de nuestra herramienta y de los requisitos que esta tiene.

A lo largo del **Capítulo 4** se muestra **cómo** vamos a alcanzar los objetivos y cumplir con los requisitos a través de la **metodología** y de las **herramientas** que hemos escogido para esta tarea.

La descripción de nuestra plataforma, bautizada como **Jasonity** ocupa un lugar central en el **Capítulo 5**. En dicho capítulo se explica el diseño de la arquitectura software (tanto su *back-end* como su *front-end*), como parte de la documentación técnica sobre el **análisis, diseño e implementación** del sistema.

En el **Capítulo 6** se expone por qué hubo que **rediseñar nuestra herramienta**, y el trabajo de implementación de ese replanteamiento hasta obtener el sistema cerrado.

Por último, en el **Capítulo 7** recopilamos toda la información relevante del proyecto para presentar nuestras **conclusiones** y dar algunas pautas sobre el **trabajo futuro** que aconsejamos abordar para darle continuidad al mismo.

Capítulo 2

Estado de la Técnica

Como se ha explicado en el capítulo anterior, a continuación y durante todo este capítulo vamos a exponer el **trabajo de investigación que hemos llevado cabo** para tener una visión general de cómo se encuentra la cuestión que nos atañe.

Así, haremos una breve introducción en materia de técnicas de IA y por la construcción de sistemas multiagente basados en el paradigma BDI, así como herramientas que se encuentren disponibles para este mismo fin, seguido por un breve recorrido por la aplicación en estas técnicas en la industria de los videojuegos. De esta manera **quedará delimitado el ámbito de actuación de nuestro trabajo**, que permitirá establecer con precisión nuestros objetivos.

2.1. Razonamiento e Inteligencia Artificial

Si queremos desarrollar un sistema que implemente agentes capaces de razonar como lo haría un humano primero debemos **analizar qué es en sí el razonamiento**, así como qué tipos de razonamiento son los más relevantes y cómo se traducen en paradigmas de programación.

2.1.1. ¿Qué es el razonamiento?

El razonamiento, como se trata por ejemplo en "Historia y Evolución del Pensamiento Científico" (Ramón Ruiz, 2006), se define de la siguiente manera:

"El razonamiento es la operación lógica mediante la cual, partiendo de uno o más juicios, se deriva la validez, la posibilidad o la falsedad de otro juicio distinto".

Dichos juicios, que son la base del razonamiento, pueden ser conjuntos de información ya obtenida (por nosotros o por terceros), o que se plantean como una hipótesis. Si los juicios han sido obtenidos por nosotros mismos o por terceros sirven como base para obtener nueva información. Cuando se emplean hipótesis, la operación lógica no sólo busca obtener juicios nuevos, sino combinarlos con juicios ya comprobados como veraces para demostrar la veracidad de las propias hipótesis empleadas como base.

El proceso de deducción para obtener juicios nuevos a partir a partir de los ya verificados se denomina **inferencia**. La ejecución de una inferencia se lleva a cabo según las reglas establecidas y formuladas por la lógica.

Aunque habría mil maneras de definir qué es una deducción, la síntesis es que; dado uno o varios juicios previos llamados condiciones, se siguen una serie de operaciones lógico-matemáticas para obtener un nuevo juicio: la **conclusión**.

Si la conclusión obtenida de dicha inferencia es más precisa y concreta que los conocimientos previos de los que parte, entonces se habrá efectuado una **inferencia deductiva**.

De forma opuesta, si el resultado de la inferencia resulta en un conocimiento más general que los anteriores, la inferencia empleada se denomina **inductiva**.

Por último, cuando la conclusión es igual de genérica que sus premisas, la inferencia ejecutada habrá sido **transductiva**.

Para ordenar los conceptos referentes al razonamiento e inferencia, y sintetizarlos de forma clara y precisa en el contexto de nuestro proyecto nos hemos apoyado en "El Conocimiento Silencioso" (Euler Ruiz, 2000), y a continuación, describiremos los dos tipos de razonamientos empleados para poder llevar a cabo una inferencia.

2.1.1.1. Razonamiento lógico

Los resultados del razonamiento lógico son binarios: pueden ser **correctos** (aquellos válidos), o **incorrectos** (aquellos inválidos). Si bien ambos se obtienen a través de la inferencia, existen sutilezas en el método empleado para obtener sus conclusiones.

En el caso de los **razonamientos correctos** se pueden obtener mediante inferencia deductiva o no deductiva:

- **Inferencia deductiva**: las veracidad de las premisas convierte la conclusión en **verdadera de forma inequívoca**.
- **Inferencia no deductiva**: la veracidad de las premisas convierte la conclusión en **verdadera de forma probable**.

En cuanto a los **razonamientos incorrectos**, no es necesario distinguir entre los tipos de inferencia utilizadas en su desarrollo, ya que en todos los casos se definen por alcanzar una conclusión **falsa** a partir de las premisas empleadas.

También mencionar que existe un tipo de razonamiento no válido que puede parecer que ofrece una conclusión válida. Sin embargo, ésto lo logra a costa de violar alguna regla lógica. Este tipo de razonamiento se denomina **falacia**.

Mediante un correcto proceso de razonamiento podemos ampliar nuestros conocimientos sin la necesidad de una experiencia previa que pudiera servirnos como referencia y guía. Es la capacidad de razonar lógicamente la que distingue al ser humano de los demás seres vivos, y que se pretende simular en las técnicas de Inteligencia Artficial.

2.1.1.2. Razonamiento no lógico

Dado que éste tipo de razonamiento **parte de las experiencias, el contexto y la subjetividad del "razonador"** que la lleve a cabo, los procesos para realizar dichos razonamientos no están catalogados formalmente.

En el contexto de la IA, el razonamiento no lógico está más próximo a la psicología que a lógica matemática, ya que el sujeto que la realiza se vuelve un elemento clave en el razonamiento.

Otra diferencia a tener en cuenta con el razonamiento lógico, es que el no lógico nunca garantiza que la conclusión vaya a ser cierta pese a que las premisas lo sean. Por tanto, **el resultado de la inferencia siempre se considerará probable**, una síntesis incompleta de las premisas.

De modo que, en este tipo de razonamientos, **acertar en la conclusión será siempre una probabilidad, nunca una certeza**.

2.1.2. Programación lógica: del Razonamiento humano a la IA

Esta introducción sirve para comprender los **fundamentos de la llamada "programación lógica"**, que forma parte del paradigma de programación declarativa y gira en torno al concepto de *predicado*, o relación entre elementos.

A diferencia de los otros lenguajes de programación, donde las instrucciones son órdenes a ejecutar (paradigma imperativo), éste tipo de programación **trata de especificar cuál es el problema a resolver**, es decir, cuales son las condiciones que han de darse para considerar resuelto el problema.

Tal como se refleja en "Artificial Intelligence: A modern Approach" (Russell y Norvig, 1995), la Programación Lógica busca **solucionar el impedimento que planteaba el paradigma imperativo** de programación en problemas complejos que constaban de elementos propios de la lógica, como hipótesis o reglas. Por ello, conseguir un modo de programación que se basara en dichos elementos supondría una solución rápida a problemas complejos; aquellos donde una mente humana que razonara era esencial.

La programación lógica da paso a técnicas de IA empleados actualmente, como pueden ser:

- **Sistemas expertos**, donde un computador trata de emular la toma de decisiones de un humano.
- **Demostración automática de teoremas**, cuando un computador debe demostrar teoremas para probar la veracidad o falsedad de una teoría.
- **Reconocimiento de lenguaje natural**, donde se busca que el computador sea capaz de extraer la información que alberga el lenguaje humano.

2.1.2.1. Prolog

El lenguaje de programación lógica más conocido que hace uso de este tipo de programación es **Prolog**, ya que puede resolver problemas lógicos mediante la aplicación de diferentes estrategias (diferentes formas de programar la solución).

Como describen Russell y Norvig (1995), Los programas implementados en **Prolog** se componen en su totalidad de *cláusulas de Horn* invertidas:

primero se escribe la conclusión de la cláusula y después las condiciones a cumplir necesarias para que se dé dicha conclusión.

Un **ejemplo** muy simple para entender el funcionamiento de Prolog es el del árbol genealógico. Inicialmente partimos de una lista de hechos:

```
chico(mario).

chica(ana).

chica(luisa).

padreDe(mario, ana).

padreDe(mario, luisa).
```

Estas cláusulas nos indican que "mario" es un chico y "ana" y "luisa" son chicas, además de que "mario" es el padre de "ana" y que "mario" también es el padre de "luisa". A partir de aquí, podemos establecer una serie de **reglas** que el sistema recorrerá para establecer nuevos hechos. Un ejemplo de regla puede ser el siguiente:

```
hermanaDe(X,Y) :- chica(X), padreDe(Z,X) == padreDe(Z,Y).
```

Esta regla indica que un individuo X será hermana de un individuo Y si Y es una chica y el padre de X e Y (al que llamamos Z), es el mismo. El sistema recorrerá toda la lista de reglas e irá contrastando las condiciones con su lista de hechos hasta que encuentre una coincidencia. Si se da esa coincidencia, el sistema añadirá ese nuevo hecho (en este caso, el hecho de que X es la hermana de Y) a su lista de hechos y seguirá mirando las demás reglas hasta que recorra toda la lista.

2.2. Agentes racionales

Ya hemos visto qué es el razonamiento aplicado a IA y qué entendemos como Programación Lógica. A continuación debemos entender **qué es un agente racional y por qué es importante** para mejorar los sistemas y herramientas disponibles actualmente en la industria del videojuego, cuyos ejemplos veremos para finalizar el capítulo que nos ocupa.

2.2.1. ¿Qué es un agente?

En el contexto en el que nos ocupa, como se describe en la obra de Bordini, Hübner y Wooldrige (2007) y que sintetizamos a continuación, un Agente es una **entidad computacional inteligente** que percibe su entorno a través

de sensores y/o actúa a través de motores/actuadores. Por lo general, un Agente cuenta con las siguientes características:

- **Percibe** su entorno, o partes de él.
- **Responde** a esas percepciones de manera racional.
- Tiende a maximizar el efecto o la corrección de su **respuesta**.

Un agente completo (que posea las tres características) **ejecutará ciclos para la percepción, pensamiento y actuación**.

Tomemos a los seres humanos como ejemplo: percibimos nuestro entorno a través de nuestros cinco sentidos (sensores), pensamos sobre ello y luego actuamos utilizando las partes de nuestro cuerpo (actuadores). De manera similar, los agentes perciben el entorno a través de los sensores que les proporcionamos (pueden ser cámaras, micrófonos, detectores de infrarrojos, etc.), hacen algunos cálculos (piensan) y luego actúan utilizando varios motores / actuadores conectados para dicha función.

2.2.1.1. ¿Qué es un agente racional?

Un agente racional es un agente que a la hora de producir una respuesta **trata de maximizar la utilidad de dicha respuesta aplicando razonamiento**, en los términos en los que hemos hablado anteriormente. El agente racional es un concepto central para la IA clásica.

La racionalidad del agente se mide en última instancia por **su rendimiento**, según el conocimiento previo que tiene, el entorno que puede percibir y las acciones que puede realizar.

Para satisfacer los casos de uso en el mundo real, la IA utiliza un amplio espectro de agentes racionales, incluyendo los agentes lógicos (aquellos que utilizan el razonamiento lógico visto previamente).

En este trabajo nos centraremos en los agentes racionales para sistemas multiagente, los cuales están basados en objetivos. De esta forma, **los agentes combinan la información de proporcionada por el entorno con el estado del mismo para, de este modo, elegir las acciones que permitirán alcanzar dichos objetivos** al agente.

2.2.2. Sistemas y arquitecturas multiagente

Sabiendo ya lo que entendemos por agente racional, trataremos de ahora en adelante de definir qué es un sistema multiagente.

Un **Sistema Multiagente** (*Multi-Agent System* o MAS, en sus siglas en inglés) es, como su nombre indica, un **sistema compuesto de dos o más agentes racionales que interactúan entre sí en el contexto de un entorno**.

Tal y como ocurriría con sujetos humanos, cada agente tiene sus propios objetivos y motivaciones. El éxito a la hora de enfrentarse a ciertos objetivos requerirá un grado de interacción entre agentes. Por esto, **un sistema multiagente permite simular al mismo tiempo el comportamiento de un individuo y el comportamiento de un colectivo formado por múltiples individuos**, cada uno distinto del anterior.

Utilizando este tipo de sistemas con agentes basados en objetivos podemos plantear situaciones que deban ser resueltas por un solo agente a la vez que se plantean otras que requieren la participación de varios de ellos. Los agentes deberán decidir de manera independiente si perseguir su objetivo individual o cooperar con los demás agentes para resolver el objetivo conjunto.

El mundo real es, en cierto modo, un entorno multiagente. En muchas ocasiones, cumplir nuestros objetivos depende de interactuar con otros individuos, que también tienen sus propios objetivos. La interacción entre agentes, por tanto, es crucial en este tipo de sistemas. **Llamamos *habilidad social* a la capacidad que tienen los agentes de interactuar con otros agentes** mediante cooperación, coordinación y negociación. Para esto, lógicamente, es necesario contar con alguna clase de lenguaje de comunicación, que dependerá del tipo de agentes o de sistema multiagente que se desee implementar.

Existen múltiples arquitecturas abstractas que pueden emplearse a la hora de representar la "mente" de un agente. La idea es establecer cómo funciona la toma de decisiones y qué motiva a cada individuo a realizar una acción en su entorno. Seguiremos los conceptos expuestos en la asignatura de Enginyeria del Coneixement i Sistemes Distribuïts Intel.ligents (Universitat Politècnica de Catalunya, 2018).

Las ***arquitecturas reactivas*** **se basan en la conexión de sensores y actuadores con el entorno** que rodea al agente. El entorno lanza un estímulo sobre el agente y este responde en función del estímulo recibido. La conexión entre sensores y actuadores hace que la conducta del agente parezca inteligente. También puede añadirse al agente un modelo del entorno que

utilizará para decidir cómo reaccionar al estímulo. **Un agente reactivo no piensa en las consecuencias de sus actos. Reacciona a lo que sucede en su entorno sin pensarlo**, porque sabe lo que tiene que hacer ante cada situación.

Las ***arquitecturas deliberativas*** u orientadas por objetivos poseen una **representación interna del entorno**. Su aproximación al mundo es más simbólica y **su toma de decisiones está más basada en razonamientos lógicos** que en reacciones directas. Un agente deliberativo **puede decidir no actuar** ante un estímulo **si no es beneficioso** para sus objetivos. Un agente deliberativo sabe lo que tiene que hacer, pero no cómo hacerlo. Debe ser capaz de elegir sus objetivos y de descubrir la forma de alcanzarlos.

También puede implementarse una ***arquitectura híbrida*****, que combina reactivas y deliberativas** en un intento de obtener las ventajas de cada una evitando los inconvenientes. Es posible que una situación se resuelva mejor de manera reactiva, pero otra sea perjudicial si no se toma un camino más lógico.

2.3. Arquitectura BDI

El modelo BDI (o Creencia-Deseo-Intención), que es en el que nos vamos a centrar, **es en esencia una arquitectura deliberativa**. Nos apoyamos en la obra de Bordini, Hübner y Wooldrige (2007) para comprender la arquitectura BDI y posteriormente Jason.

2.3.1. Modelo Creencia-Deseo-Intención

Este modelo cognitivo se basa en el denominado ***razonamiento práctico.*** **Éste simula el proceso que realiza la mente humana** a la hora de decidir qué hacer en cada momento para alcanzar un fin específico. Podemos dividirlo, por tanto, en dos procesos: *deliberación*, o decidir qué objetivo queremos alcanzar, y *razonamiento*, o decidir cómo alcanzar ese objetivo.

Como su propio nombre indica, este modelo se basa en **basar el proceso de decisión de un agente en tres elementos**: **creencias**, cómo ve el agente el mundo; **deseos**, qué opciones tiene en base a esas creencias; e **intenciones**, qué objetivos va a perseguir.

Las creencias de un agente son, en esencia, el estado informativo en el que se encuentra. Es **todo lo que cada agente cree saber sobre su entorno**. Estas creencias pueden llevar a un agente a aceptar nuevas creencias en base a las que ya tiene o a rechazar ciertas acciones que se contradicen

con lo que sabe. La utilización del término "creencia" en lugar del más evidente "conocimiento" **acepta la posibilidad de que un agente pueda creer algo falso** para, quizá en el futuro, descubrir que no lo es y cambiar su creencia. Por lo general, las creencias se almacenan en una base de datos denominada *base de creencias*, aunque esto puede variar en función de la implementación de cada sistema.

Los deseos de un agente representan las motivaciones del mismo. Son **objetivos que el agente querría cumplir** o situaciones que el agente que el agente querría que se dieran. Ejemplos de deseos podrían ser: abrir la puerta, encender la luz, hablar con un agente específico, etc. **Puede que varios de los deseos de un agente se contradigan** los unos a los otros (por ejemplo, querer jugar y dormir al mismo tiempo), por lo que también **existe el concepto de *meta*, un deseo que el agente ha decidido alcanzar**. Si la meta de un agente es jugar, dormir no puede ser una de sus metas al mismo tiempo, ya que no pueden hacerse simultáneamente, aunque ambos sean deseos.

Las intenciones de un agente representan los actos inmediatos del agente o, dicho de otra forma, **lo que el agente ha decidido hacer en este momento**. Desde el punto de vista de la implementación, esto significa que el agente **ha comenzado a ejecutar un *plan***. Llamamos plan a una secuencia de acciones que el agente puede seguir para alcanzar una de sus intenciones. **Un plan puede estar formado por planes más pequeños**: abrir una puerta puede incluir un plan para conseguir la llave de esa puerta. Los planes y sus posibles "subplanes" se van desarrollando a medida que el agente descubre lo que necesita para llevarlos a cabo.

El modelo BDI también utiliza *eventos*. **Un evento es un suceso que ocurre en el entorno** y provoca una reacción de algún tipo en el agente. Es posible que el agente actualice sus creencias, modifique sus deseos o altere uno de sus planes para cumplir una intención en base a los cambios que se hayan podido producir. **Estos eventos pueden ser provocados por el propio entorno o por otros agentes** en el caso de un sistema multiagente.

Algunos modelos BDI también incluyen normas de conducta que impiden a los agentes realizar ciertas acciones. Es posible que no nos interese que uno de los agentes ataque a otro para conseguir la llave que abre la puerta que quiere cruzar, así que podemos evitarlo estableciendo ciertas leyes universales.

2.3.2. AgentSpeak

Posiblemente uno de los lenguajes de programación de Agentes individuales y MAS orientado a BDI para agentes autónomos (cognitivos) es *AgentSpeak*. Se basa en la programación lógica para construir una arquitectura capaz de manejar:

- Bases de creencias.
- Librerías de planes.
- Conjuntos de eventos.
- Conjuntos de intenciones.

La implementación más usada para AgentSpeak hoy día es la plataforma Jason, implementada en Java.

A continuación detallaremos el funcionamiento de la arquitectura BDI programada usando AgentSpeak.

2.3.2.1. Arquitectura

El agente recibe eventos que son o bien externos (del entorno, de datos percibidos), o bien generados internamente. Intenta encargarse de dichos eventos **buscando planes que encajen con ellos**. El conjunto de planes que encajan con los eventos son denominados opciones o deseos. **El agente elige un plan de entre sus deseos para ejecutarlos: se convierte en una intención.**

Según va ejecutando el plan **genera nuevos eventos** son los que tendrá que volver a lidiar en un futuro.

2.3.2.2. Creencias

Las **creencias representan información que el agente tiene sobre su entorno**. Se representan simbólicamente: *Ground terms* de lógica de primer orden. Algunos ejemplos pueden ser: `abrir(válvula32)`, `padre(Lily, Michael)` o `sobre(bloqueA, bloqueB)`.

2.3.2.3. Planes

Dan al agente **información sobre cómo responder a los eventos y como alcanzar los objetivos**.

La estructura de un plan se compone de esta forma: **Condición de activación : Contexto <- Cuerpo**

- **Condición de activación:** su entorno, o partes de él.
- **Contexto:** condiciones que especifican el uso de ese plan.
- **Cuerpo:** definen las acciones a ejecutar si el plan es escogido.

2.3.2.4. Eventos

- **+!P:** nuevo objetivo adquirido – "lograr P".
- **-!P:** objetivo P descartado.
- **+B:** nueva creencia B adquirida.
- **-B:** creencia B descartada.
- **+?P:** nuevo objetivo de prueba adquirido.
- **-?P:** objetivo de prueba descartado.

2.4. Herramientas para Desarrollo de Sistemas Multiagente

Sabiendo ya qué es un Agente Racional y un Sistema Multiagente, debemos saber **qué herramientas existen** para implementar dichas tecnologías. Así podremos analizarlas y aplicarlas a nuestro problema concreto.

2.4.1. Jason

Como partimos del empleo de *AgentSpeak* en nuestro sistema, debemos poner la vista en Jason (Jomi F. Hübner and Rafael H. Bordini, 2007), ya que (como se ha expuesto anterioremente) es **la herramienta más completa que implementa AgentSpeak** y se trata de la herramienta empleada en los trabajos previos en los que nos apoyamos: González (2018) y Sánchez-López et al. (2016).

Jason es un intérprete para una versión extendida de AgentSpeak. Implementa la semántica operacional de ese lenguaje y **proporciona una plataforma para el desarrollo de sistemas multiagentes, con muchas características personalizables** por el usuario. Jason, comparado con otros sistemas de agentes BDI, está implementado en Java (por lo tanto es multiplataforma), se encuentra disponible como código abierto y se distribuye bajo GNU LGPL e incluye características nuevas que no contempla su lenguaje base (por ejemplo, un sistema de comunicación de agentes basado en discursos).

2.4.2. jEdit para Jason

Si bien es cierto que podría considerarse un componente más de Jason, vamos a distinguirlo del mismo para desarrollar sus diferencias. Jason es el motor de procesamiento, usable como una librería; mientras que **jEdit es el editor de texto libre que ha sido extendido como herramienta de programación de *MAS***, incluyendo las funcionalidades propias de Jason (de ahora en adelante todas las menciones a "jEdit" harán referencia a la extensión específica de Jason).

Ofrece una **interfaz gráfica** (Figura 2.1) para la programación de agentes en ASL, la posibilidad de lanzar, pausar o lanzar en modo *depuración* los proyectos del usuario, explorar el razonamiento seguido en ejecución por todos los agentes o algunos en particular, ejecución por ciclos, etc.

Además, cuando se lanza el proyecto que estemos desarrollando, **ofrece una vista detallada del ciclo de razonamiento de todos los agentes** del mismo, ya sea todos los agentes en la misma consola o un agente por consola. Permite además una ejecución por ciclos: cada vez que el programador ordene la ejecución se realizará solamente un ciclo de razonamiento (desde la percepción hasta la ejecución), y quedará de nuevo a la espera de que el programador ejecute un nuevo ciclo.

No cabe duda del potencial de ésta herramienta, ya que **permite programar minuciosamente la lógica subyacente** en cada agente y explorar su ejecución en tiempo real, y es un gran referente a tener en cuenta a la hora de hablar de programación de sistemas multiagente BDI.

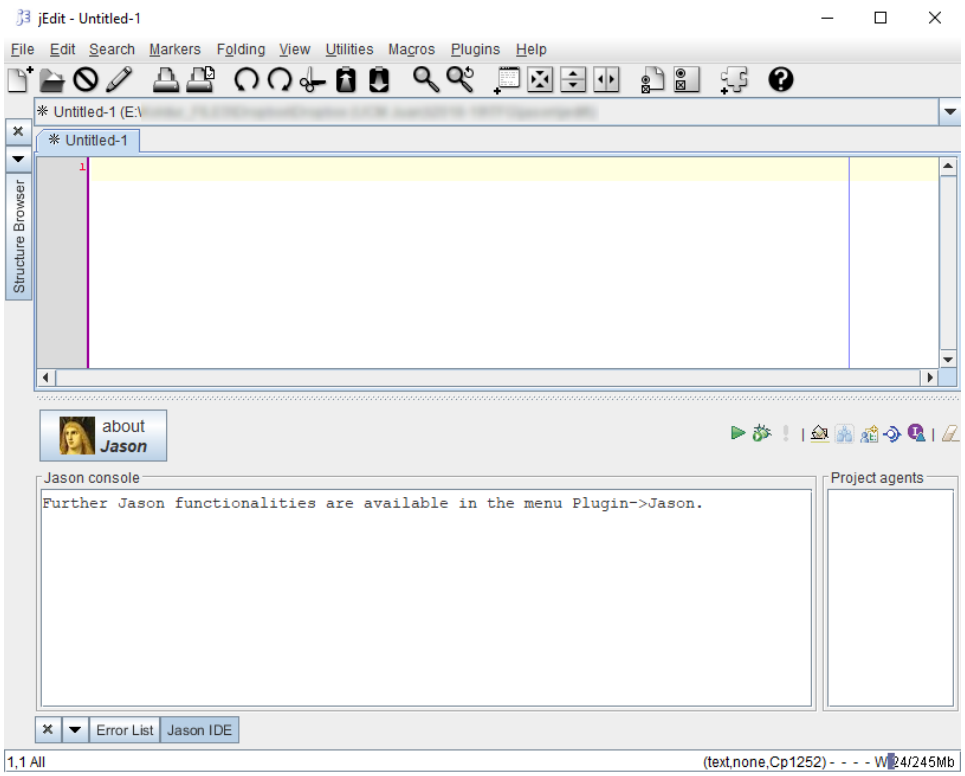

Figura 2.1: Interfaz gráfica de jEdit ejecutada en Windows. Se muestra la zona de edición de ASLs en la mitad superior, la consola de ejecución en la mitad superior y los controles de ejecución en el centro (entre otros elementos).

2.4.3. Unity

Unity3D (Unity Technologies, 2005), comúnmente denominado simplemente **Unity, es uno de los *Game Engines* más populares del momento** junto con Unreal Engine (Epic Games, 1998).

Ambos siguen un modelo de negocio similar: **cualquier usuario puede descargar y usar el motor de manera gratuita**, pero si el volumen de negocio derivado de un juego desarrollado con dicho motor es mayor a una cifra concreta, la desarrolladora percibe un porcentaje de los ingresos (y/o estás obligado a adquirir la versión premium del motor).

Unity está **basado en C#**, y ofrece una interfaz de desarrollo gráfica para la **composición de escenas y delega en los scripts** la programación del comportamiento de todos los elementos introducidos en escena.

Figura 2.2: Fotograma de "Ori and the Blind Forest" (Moon studios, 2015). Se trata de uno de los ejemplos de juegos desarrollados con Unity

Posee una **potente comunidad de desarrolladores** de elementos gráficos y programadores que comparten sus creaciones en la denominada *Asset Store*: una plataforma que funciona como tienda *on-line* donde pueden encontrarse complementos de todo tipo.

O *casi* todo tipo, ya que tras una investigación no se han encontrado herramientas de diseño de agentes BDI. Las herramientas encontradas (para la prgramación BDI o semejantes) fuera de la *Asset Store* han sido *Isomonks* y UnityProlog.

2.4.4. IsoMonks

Desarrollado a lo largo del TFG *Isomonks* (Sánchez-López, Romero y Martín-Solís, 2016); **supone la aproximación más cercana al objetivo que nos planteamos**. Se trata de un **conector Unity-Jason** para juegos isométricos.

Ésta herramienta **delega todo el procesamiento de los agente en el motor** de *Jason*, y se conecta a los *NPCs* de *Unity* a través de *Sockets*, que ejecutan los planes generados por el motor de razonamiento.

Si bien es cierto que **tiene mucho potencial**, ya que delegar el razonamiento en otro programa puede extenderse también a delegar el razonamiento mediante servicios web abiertos, **la implementación disponible está muy ligada a la demostración** que llevaron a cabo en el proyecto, por lo

que habría poco código reutilizable.

Así pues, se buscó una forma de implementar toda la lógica dentro de *Unity*.

2.4.5. Unity Prolog

UnityProlog (Ian Horswill, 2018) es el primer software con el que nos encontramos que **permite el uso de un lenguaje interpretado desde el propio Unity**. Se trata de un desarrollo personal de Ian Horswill, mantenido también por su persona, por lo que el alcance de la herramienta es relativamente pequeño.

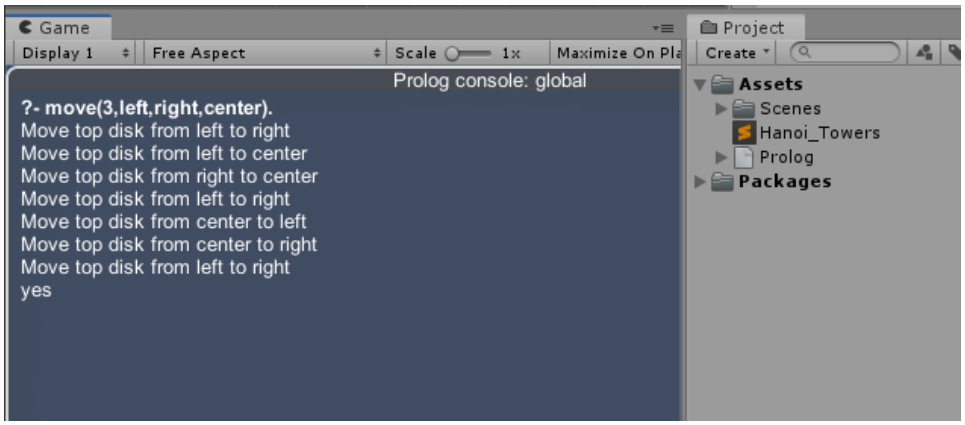

Figura 2.3: Consola de UnityProlog integrada en Unity. Muestra la depuración de una ejecución de prueba, concretamente en el caso del problema de las Torres de Hanoi con su resolución.

Ofrece una **interfaz integrada en el entorno de Unity**, como se puede observar en la figura 2.3, capaz de interpretar reglas en Prolog y alcanzar soluciones a un problema planteado de pequeña escala. No se encuentra disponible en la tienda de Unity y debe ser **descargada e instalada a mano desde su página de GitHub**.

Una vez hemos completado la instalación dispondremos de una consola de comandos como la mostrada en la figura 2.3 para realizar operaciones en Prolog sobre ella.

Como se puede comprobar, la herramienta es **capaz de leer una regla en Prolog desde la consola, procesarla y alcanzar una solución al pequeño problema** que se le plantea. Sin embargo, también querremos resolver problemas más complejos que requieran de más de una regla para poder plantearse, para los cuales ya no nos será posible escribirlos directa-

mente en la consola. Una vez ejecutada la consulta la herramienta trata de resolver el problema y, si lo consigue, muestra un "yes" al usuario una vez termina la ejecución.

Es aquí que entra en juego el concepto de **Base Global de Conocimientos, el cual será implementado mediante un "GameObject"** de Unity al que añadiremos un script proporcionado por el propio UnityProlog llamado KB, donde guardaremos los ficheros que contienen los problemas Prolog (los pasos a seguir para este proceso se encuentran también en la documentación).

Una vez que hemos creado la Base Global de Conocimientos, a la que denominaremos KB como en la documentación, **podemos realizar consultas en la consola referentes a los problemas** que tengamos guardados en la Base.

Pese al potencial que pudiera denotar la herramienta para resolver problemas de programación lógica, pronto se verá **descartada tanto por las limitaciones de la propia herramienta**, las cuales se ejemplifican en el próximo párrafo, **como por la mayormente imprecisa documentación**, que no hace más que complicarse o desaparecer a medida que nos aproximarnos a las funcionalidades más complejas que posee UnityProlog como por ejemplo: llamar al código Prolog desde CSharp o integrar las funcionalidades de los ficheros Prolog (scripts de la KB) en Unity sin la necesidad de tener que ejecutarlas desde la consola.

En referencia a las **limitaciones de la propia herramienta** mencionadas en el párrafo anterior, ahora citaremos algunas de las que fueron más notorias para nosotros de cara a implementar BDI con ellas:

- **No soporta reglas de más de 10 objetivos**: esto fue bastante restrictivo ya que muchos de los ejemplos de Agentes de Jason contienen más de diez objetivos y no se pueden subdividir ya que, de hacerlo, los planes y reglas de Jason perderían su sentido.
- **La sintaxis de Prolog difiere de la de Jason:** aunque ambos son lenguajes de programación lógicos, la sintaxis de Jason es diferente ya que ha sido personalizada por sus desarrolladores y contiene elementos como los planes y las acciones internas de las que Prolog carece y, puesto que nuestro sistema multiagente estaría basado en el de Jason, es una notoria restricción a la hora de desarrollarlo sobre esta herramienta.
- **Controlar la depuración es difícil con múltiples KB:** si pretendemos implementar un sistema multiagente donde cada agente tenga su propia base de conocimientos pero el propio autor de UnityProlog

> te pone por escrito que, literalmente, controlar el funcionamiento de los scripts es un dolor si hay varias bases de conocimiento a la vez, es un buen indicativo para descartar su herramienta.

Así pues no es escalable, la sintaxis es especializada y la depuración es complicada, por no decir irrealizable; por lo que **se descartó como herramienta** viable para cumplir nuestros objetivos.

2.4.6. Diseño de Agentes experimentando con robots que juegan al fútbol en ambientes reales y simulados

Es un proyecto de investigación realizado en la Universidad Nacional del Comahue, Argentina, por parte de Kogan, Parra y del Castillo (2009). En él usaron la estructura de programación de **BDI aplicada a Prolog para desarrollar cuatro robots "futbolistas"**: dos porteros y dos jugadores, que representaban de manera correcta sus roles dentro del campo. Sirvió como un primer vistazo a las posibilidades de Prolog para BDI, no obstante, debido a la **alta especialización en Prolog que se requería** para entender las complejas reglas que regían a los robots y poder extraer de ellas el patrón BDI, así como el hecho de que ninguno de los miembros del proyecto éramos duchos en Prolog ni podríamos contar con un posible experto en este lenguaje a la larga y que la idea del proyecto es crear una herramienta sencilla de usar, **se decidió descartar esta rama de la investigación**.

2.4.7. BDI Agents and Artifacts in Unity

Tesis doctoral de Poli (2018) en la que desarrolla un ejemplo de BDI en Prolog sobre unos pequeños **robots que recogen basura del escenario y la llevan a unos contenedores específicos según el color**. La gracia está en que un robot sabe donde está la basura de un color, otro sabe lo mismo para el segundo y un tercero no sabe nada. De esta forma **los robots deben comunicarse entre ellos para averiguar los datos que les faltan** cuando recogen la basura del color que no conocen.

Sin embargo, por muy didáctico y ejemplificador que sea este proyecto, no nos resultó útil debido a que, una vez se profundizó en el código, se trata de un **programa específico para resolver el problema explicado anteriormente** y sobre el que no se puede modificar nada. De modo que no puede construirse una herramienta sobre la que los usuarios de Unity puedan crear sus propios agentes.

2.5. IA en Videojuegos: breve recorrido por la última generación

Una vez tenemos claro qué es el razonamiento en el contexto de la IA, los Agentes Inteligentes y el modelo BDI vamos a hacer un **recorrido por los juegos más notables donde se puede apreciar la aplicación de dicho razonamiento** computerizado para darle vida a personajes virtuales.

Para empezar por lo básico debemos dejar claro que **IA en un videojuego**, en términos generales, la definimos como las técnicas/algoritmos/paradigmas (propios de la IA o simplemente una concatenación de instrucciones "if", si eso fuera suficiente) que **se usan para dotar de "vida" a un NPC**. Un NPC, como vimos, es un personaje virtual que el jugador no controla, y cuyo comportamiento define el sistema en el que el videojuego en cuestión se esté ejecutando (consola, ordendor u *on-line* vía servidores).

Sin embargo, existen ocasiones en las que una **IA demasiado optimizada y coherente sería antinatural e injusta para con el jugador** (injusta si el NPC juega contra el usuario) debido a que procesaría lo que ocurre, respondería siempre más rápido de lo que es capaz de hacer cualquier humano, y el proceso de razonamiento daría lugar a resultados demasiado perfectos. Por ejemplo, en los videojuegos de acción en primera persona, donde la puntería es crucial para el juego, una puntería perfecta y una capacidad de reacción que tiende a 0 supondría tener enemigos imposibles de vencer.

Así pues, **repasaremos ejemplos de la última generación de videojuegos** (juegos lanzados para los sistemas *PlayStation 4*, *Xbox One*, *Nintendo Switch* y/o PC desde el año 2013 hasta mayo de 2019) donde se haga notable el uso de la *IA* en el control de *NPCs*

2.5.1. The Last of Us

En "The Last of Us" (Naughty Dog, 2013) encontramos un ejemplo paradigmático que veremos en la generación de un **tipo de NPC muy controvertido: el acompañante**.

Si bien es cierto que no es estrictamente un juego de la última generación (fue lanzado en junio de 2013 para *PlayStation 3*, a unos meses del lanzamiento de *PlayStation 4*), es considerado por los fans como un juego de última generación de pleno derecho (posiblemente porque fue lanzada su versión mejorada para la siguiente generación al año siguiente, en 2014).

En este título, **el jugador toma el control de Joel**, uno de los dos protagonistas de la historia postapocalíptica, y **deja el control de su acom-**

Figura 2.4: Imagen promocional de "The Last of Us" (Naughty Dog, 2013). Aparecen Ellie (izquierda, chica de 13 años acompañante del jugador) y Joel (derecha, entrado en los 40 años y personaje controlado por el jugador) practicando su puntería en una granja abandonada.

pañante (Ellie) a la IA del juego. Juntos deben sobrevivir a la pandemia zombie, y por ello es importante que el comportamiento de Ellie no sea el de un simple "fardo" que el jugador deba proteger sin que se valga por sí misma.

Se trata de una **mecánica que a día de hoy resulta arriesgada**, ya que se necesita una **IA increíblemente realista** para que el jugador realmente entienda a su compañera como un ente humano, y no un objeto que portar. Además, éste caso en particular se ha hecho famoso por mostrar no sólo un NPC proactivo y "conveniente" para el jugador, sino que **se ha logrado dotarla de una personalidad única y realista**, capaz de realizar acciones imprevistas que resulten coherentes para el jugador que vive la historia.

Si bien es cierto que **la base es sencilla** (Ellie sería un objeto que sigue al jugador y realiza acciones básicas), los creadores **han refinado su IA en diversos puntos clave**. Existe un artículo publicado que detalla el funcionamiento de Ellie (Dyckhoff, 2015).

En primer lugar, **Ellie evalúa las posibles posiciones donde podría situarse** con respecto al jugador, se predice en qué dirección se moverá a continuación el mismo para descartar las que estén en medio de sus posibles trayectorias y que no existan obstáculos entre jugador y acompañante. Así se consigue un mayor realismo a la posición en la que se encuentra en todo

momento Ellie.

A la hora de ponerse a cubierto en combate, el sistema descrito anteriormente no es aplicable; ya que emplea sólo el jugador y los obstáculos para calcular la posición. Cuando **se introducen enemigos en la ecuación**, la posición a la que debe dirigirse Ellie para cubrirse es calculada también con variables como visibilidad de los enemigos, predicción de visibilidad futura, proximidad a grandes aglomeraciones o al líder del grupo, etc.

Además, **existen puntos de interés que el personaje de Ellie puede decidir o no explorar por su cuenta** (si las circunstancias se lo permiten) para dar la sensación de que el personaje posee una personalidad curiosa e independiente; humanizando aún más la IA subyacente.

Aunque existen más técnicas para más acciones, sólo estas tres **nos muestran que resolviendo problemas pequeños** (calcular dónde no se estorba al jugador, dónde es mejor cubrirse o la existencia de detalles interesantes) **se puede dotar de mayor realismo** a un objeto que podría ser bastante pasivo. El personaje de Yorda en "Ico" (Team ICO, 2001) es famoso por suponerle al jugador un inconveniente por encima de todo, más que un acompañante que genere empatía.

2.5.2. Assassin's Creed: Unity

Posiblemente la saga de "Assassin's Creed" (Ubisoft, 2007) merecería un capítulo entero, pero sintetizaremos lo más relevante para nuestro trabajo.

A lo largo de los años, desde el lanzamiento de su primer título en 2007, **ésta serie ha sido controvertida por la IA presente en los enemigos**. Si bien es cierto que los entornos, concretamente las ciudades, pueden ser vistos como lugares vivos con habitantes muy humanos; un simple vistazo a los detalles resquebraja esa sensación.

Aquí haremos una distinción entre dos tipo de NPC: los civiles y los enemigos.

Con respecto a **los civiles, siguiendo a cualquier NPC es suficiente para comprobar que sigue rutinas vacías**: se mueve en círculos alrededor de una manzana, sube y baja una calle, a lo sumo se para en algún puesto en el mercado, etc. Responde también a la amenaza visible por parte del jugador o un enemigo armado, pero es relativamente sencillo implementar ese comportamiento como una máquina de estados. Éste ha sido el comportamiento básico de los civiles en los más de diez años de franquicia; aunque con la ingente cantidad de humanos que pueden llegar a poblar una ciudad,

Figura 2.5: Fotograma de "Assassin's Creed: Unity" (Ubisoft, 2014). Aparece en el centro Arno Dorian (el protagonista de la historia) subido en lo alto de la cruz de la Notre Dame del Siglo XVIII.

tiene sentido que si a simple vista parezcan humanos no se realiza ninguna mejora significativa.

La mayor queja viene en cuanto nos enfrentamos a un grupo de enemigos.

Aunque **queda justificado por la experiencia de usuario** y para no hacer imposible la victoria al jugador, los **enemigos arremolinándose alrededor del protagonista y peleando de uno en uno**, un asesino de vestimenta bastante llamativa sentándose en un banco convirtiéndose en invisible para los guardias cuando minutos antes le han visto claramente acuchillando a un viandante o seguir el sonido de algo cayendo cuando han visto la roca/rama/ladrillo salir de una cobertura y cayendo a escasos metros **componen una IA muy conveniente, pero lejos de ser realista**.

En el caso concreto de "Assassin's Creed: Unity", no es difícil llegar a la situación en la cual el jugador asesina a un guardia colgado del exterior de una muralla perfectamente visible para los guardias que la patrullan, y sin embargo que ninguno reaccione a la presencia notable de un intruso, y que a la hora de detectar la ausencia de su compañero algunos guardias se lanzen al vacío intentando examinar su cadáver (con un final tan desastroso como cabe esperar).

2.5.3. The Last Guardian

Es el segundo ejemplo que disponemos de **IA acompañante, y posiblemente uno de los más famosos de los últimos años**.

En "The Last Guardian" (genDesign; SCE Japan Studio, 2016), el jugador se mete en la piel de un niño (de nombre desconocido) que acaba de despertar en unas ruinas ubicadas en El Nido. Ahí se encuentra con una criatura (mezcla de pájaro/perro/gato) a la que pone por nombre Trico, y que será su acompañante a la hora de resolver los puzles que le separan de la libertad fuera de ese lugar.

Figura 2.6: Fotograma de "The Last Guardian" (genDesign; SCE Japan Studio, 2016). En él aparece Trico (a la derecha, sobre cuatro patas) sujetando con el pico al protagonista (en el centro, sin nombre propio conocido pero coprotagonista de la historia y personaje jugable por el usuario).

En éste título, Team ICO logra **sobreponerse al fracaso de su anterior acompañante, Yorda**, en "Ico" Team ICO (2001). Usado como elemento clave en el marketing del título y corroborado por los jugadores, **la IA que le da vida a Trico logra realmente transmitir esa sensación de viveza** de un ser vivo independiente y con personalidad. Y es que un elemento clave, ya usado en Agro, el caballo del protagonista en "Shadow of the Colossus" (Team ICO; SIE Japan Studio, 2005) ha sido la independencia: **no obedecer incondicionalmente los reclamos u órdenes del jugador**.

Precisamente en el caso de animales o criaturas mitológicas, resulta evidente que **no en todo momento deseará seguir las órdenes de un humano**; y cuando las siga no tiene por qué ser de manera completa o precisa. Aunque existía el riesgo de ser visto como una característica tediosa,

éste rasgo ha sido recibido por los jugadores como un **rasgo clave a la hora de conectar con la criatura** y de sentir su realismo de forma novedosa.

Con esto queda constatado que una **IA realista no significa necesariamente una IA perfecta**, capaz de ejecutar sus acciones de forma impoluta y en el momento en el que le sea ordenado. Cabe destacar que el realismo es más fácil de conseguirse cuando sólo debes preocuparte de un agente: todos los esfuerzos quedan volcados en esa entidad y por ello es fácil conseguir avances significativos. Queda aún por asumir el reto de alcanzar un entorno altamente poblado con criaturas que se sientan tan vivas como los acompañantes.

2.5.4. Final Fantasy XV

En el caso de Final Fantasy XV (Square Enix, 2016) se puede ver un caso en el cual la **IA ha pasado desapercibida para el público general** a pesar de implementar un sistema bastante completo.

En éste título, el jugador se sumerge en la historia de Noctis, el príncipe de Insomnia (caída ante el imperio de Niflheim) que debe expulsar a los invasores con ayuda de sus tres amigos (Prompto, Gladiolus e Ignis).

Figura 2.7: Fotograma de FFXV, donde se muestra la colaboración en combate de los protagonistas. Noctis (centro) e Ignis (izquierda, agachado)

En éste juego, el **sistema de IA está muy refinado en el sistema de combate colaborativo** entre el jugador y sus compañeros. Al tratarse de *NPCs* que deben actuar compenetrados con el usuario, debe crearse un

sistema que no interfiera con las acciones del mismo pero las complemente.

Tal como se refleja en "Game AI Pro 3" (Miyake, Shirakami, Kazuya Shimokawa, Komatsu, Tatsuhiro, Prasertvithyakarn y Yokoyama, 2017) o en una publicación de Miyake y Hasegawa (2017), para crear la IA de los compañeros de Noctis, Square Enix **combinó árboles de comportamiento y máquinas de estados en una herramienta** de programación gráfica integrada dentro de Luminous Studio (el motor de juego de Square Enix) para dar lugar a lo que bautizaron como *AI Graph*.

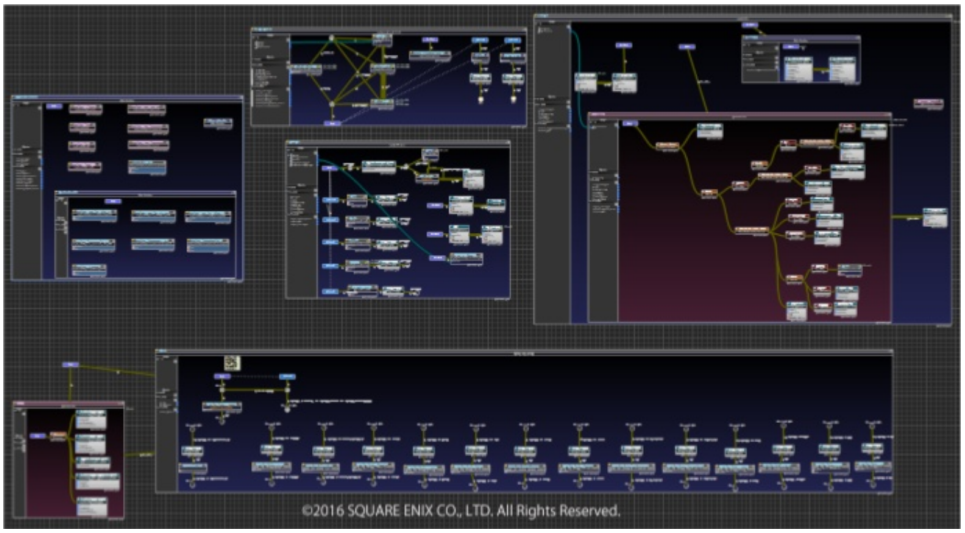

Figura 2.8: Captura de AI Graph. Se muestra la interfaz gráfica a través de la cual se pueden diseñar árboles de decisión (mitad superior) combinados con máquinas de estados (mitad inferior).

Ésta **permite crear módulos** (de combate, de paseo, de conducción, de posición estática) en los cuales la carga, uso y presencia de la máquina de estados y el árbol de comportamientos **se combinan de diferentes maneras**, independientemente unos módulos de otros. Así, las ventajas de cada sistema se pueden explotar cuando más convenga, y paliar las desventajas de los mismos reduciéndolas dependiendo de la situación.

Aunque existen características del *AI Graph* bastante interesantes (pensamiento paralelo, depuración visual, etc.), la que más nos interesa es *meta-AI, o AI Director*, usados indistintamente. **Este sistema es el encargada de orquestar el comportamiento de los personajes durante los combates**.

Permite analizar la situación para dar lugar a las siguientes acciones:

1. **Salvar** a un compañero en apuros.
2. Cuando un compañero esté rodeado, **crear una brecha** en los enemigos para que pueda escapar.
3. **Seguir al jugador** cuando huya.
4. **Seguir las tácticas** de equipo.

Así, en cada combate se puede sentir una **auténtica coordinación** entre los tres NPCs y el jugador, si bien es un detalle que ha pasado desapercibido para la mayoría de jugadores. Lo que no deja de indicar que ha logrado su objetivo: ser realista y no destacar por sus carencias o excesos.

2.5.5. Hitman 2

Del mismo modo que Assassin's Creed lo tomamos como ejemplo de una IA de enemigos demasiado limitada, los enemigos de Hitman 2 (IO Interactive, 2018) son un **ejemplo de IA demasiado inteligente**.

En éste título, el jugador toma el control del Agente 47 a lo largo de numerosas misiones donde deberá acabar con la vida de sus objetivos de manera sigilosa, empleando infinidad de herramientas disponibles a lo largo de cada mapa (armas, disfraces, distracciones, trampas, etc.) para planificar y ejecutar el crimen perfecto.

Aunque posiblemente haya pasado sin pena ni gloria, los enemigos (e incluso los viandantes) en el juego tienen una **percepción sobrehumana**: son capaces de ver un arma estando de espaldas, escuchar pasos a una distancia descomunal o (como usaron los desarrolladores como gancho publicitario) fijarse en espejos y reflejos del entorno para detectar la presencia de un intruso de una forma nunca antes vista en un videojuego.

Si bien es una **gran mejora en la IA con respecto a entregas anteriores, se comprueba que una IA más refinada no siempre se traduce en una IA más creíble** o atractiva para los jugadores. Éste aspecto del juego le ha costado críticas negativas que han impactado en el número de ventas del título.

Figura 2.9: Fotograma de "Hitman 2" (IO Interactive, 2018) con el Agente 47 en primer plano entrando en un recinto deportivo.

2.5.6. God Of War

"God of War" (SCE Santa Monica Studio, 2018) se trata del **ejemplo más reciente de juego "AAA" con presencia de un NPC acompañante: Atreus**.

En ésta entrega de la saga, el protagonista (Kratos) lleva años retirado tras los eventos acontecidos en el último título. Desplazado de la mitología griega, el juego se empapa de un ambiente nórdico para presentar la relación paternofilial de Kratos con Atreus, su hijo de once años. En ésta historia, Kratos (el jugador) debe enseñar a su hijo (el acompañante a lo largo del juego) cómo valerse por sí mismo sin caer en los errores de su padre.

Si bien es cierto que es una IA que ha pasado desapercibida en comparación con Trico en "The Last Guardian" (genDesign; SCE Japan Studio, 2016), **el sistema inteligente que da vida a Atreus también ha destacado no sólo por no interferir con la experiencia de juego, sino por ser bastante proactivo y eficaz en el momento del combate**.

Ha trascendido, como en éste artículo de Steven T. Wright (2018), que **desde The Coalition se consultó en numerosas ocasiones a Naughty Dog** para implementar la IA de Atreus basándose en la de Ellie de "The Last of Us" (Naughty Dog, 2013). Así, resulta natural comparar ambos personajes tanto por su relación de protector-protegido/a como por su comportamiento no invasivo. Para su programación, el equipo dedicado a su comportamiento

Figura 2.10: Imagen promocional de "God of War" (SCE Santa Monica Studio, 2018). Aparecen Atreus (izquierda, de unos 11 años, investigando un artefacto brillante) y Kratos (centro, de edad indeterminada, preparándose para el combate).

tomó la base de Ellie y añadió numerosas variables para su acción y diálogo como la dirección desde la cual se aproximan enemigos, la topología sobre la que se desenvuelve el combate, etc.

Además, incluyen **comportamientos no solamente orientados a la acción y al desarrollo del juego, sino puramente humanos** (actitudes con las que encarar situaciones, exploración, relación con animales salavajes, etc.) para generar esa apariencia de humanidad tan esencial para que el jugador sienta empatía por ese personaje y se forme una relación emocional con él, algo esencial para la narrativa del juego.

2.5.7. Resident Evil 2 (Remake)

Más de veinte años después de su lanzamiento original, Capcom ha relanzado de nuevo uno de sus títulos más famosos: "Resident Evil 2" (Capcom, 2018), dándole otra vuelta de tuerca a sus gráficos, a sus mecánicas, y también a su IA.

Dos meses después de la historia mostrada en "Resident Evil" (Capcom, 1996), el jugador se pone en la piel de Leon S.Kennedy (un oficial de policía recién llegado a su puesto) y Claire Redfield (una estudiante universitaria en busca de su hermano) quieres se verán envueltos en el desastre de Racoon

Figura 2.11: Fotograma de "Resident Evil 2" (Capcom, 2018). Aparece con vista de tercera persona desde la espalda Leon S.Kennedy (a la izquierda, de 21 años, uno de los personajes que podremos manejar a lo largo del juego).

City provocados por el Virus T, propiedad de Umbrella. En éste juego, el usuario deberá administrar sus recursos sabiamente (siguiendo el modelo del género *survival horror*) para superar las dificultades que se le presentan mientras intenta sobrevivir al ataque zombie.

A priori, **uno pensaría que en un juego donde no hay aliados y todos los enemigos son zombies descerebrados no existe una fuerte presencia de IA** más allá de un sensor visual para los enemigos y un algoritmo A^* que no tiene que estar demasiado pulido.

Sin embargo, aunque esto es cierto para el zombie de a pie, **existe un enemigo en concreto en el que se ha volcado una IA especialmente refinada: Mr. X**.

Este es especialmente sensible a las acciones del jugador, dotándole de una **capacidad de anticipación y predicción** que permite que siempre intente colocarse en las posiciones que más puedan incordiar al jugador para su progreso (bloqueando salidas, dificultando el paso por escaleras, etc.). Además, es realmente un agente **presente en la totalidad del escenario**, y no es usado como *jump scare* en situaciones predeciblemente sensibles del *gameplay*. En su lugar, aunque esté muy lejos del jugador, **seguirá donde quiera que esté Mr. X razonando la búsqueda de su objetivo (nosotros)**.

Supone una **gran mejora con respecto al "algoritmo" empleado en**

Figura 2.12: Fotogrma de “Resident Evil 2” (Capcom, 2018). Aparece amenazante entre sombras Mr.X mirando fijamente al jugador.

su análogo del 98 (que realmente sólo seguía unos raíles específicos, sólo estaba presente en lugares concretos para intimidar al jugador y cambiaba de sentido dependiendo de la ubicación del jugador).

Este caso nos muestra la importancia de dotar de cierta vida a los agentes presentes en los videojuegos, y que cuando se rehacen proyectos antiguos **merece la pena innovar** en aspectos fundamentales como la IA.

2.5.8. BDI en videojuegos: Black & White

El modelo BDI se ha utilizado muy poco en videojuegos, de ahí nuestro interés en estudiar el por qué. No obstante, encontramos un ejemplo relativamente conocido: **Black & White** (Lionhead Studios, 2001). Utilizaba un sistema basado en BDI para simular a sus criaturas, aprendían de tus acciones y tomaban decisiones en base a cada situación para salir lo más beneficiadas posible. A día de hoy, la inteligencia artificial del juego se sigue recordando por lo **sorprendente y avanzada que era**, aunque se temía que el jugador acabara en desventaja contra la máquina.

La utilización de **BDI en videojuegos es interesante por el enorme potencial que supone**. Tener a personajes no controlados por el jugador **reaccionando de forma orgánica y creíble** a lo que ocurre en su entorno daría un nuevo nivel de profundidad e inmersión.

Figura 2.13: Fotogrma de "Black & White" (Lionhead Studios, 2001). Se puede ver cómo los agentes que pueblan un entorno comunican el razonamiento actual.

Ahora bien, como se mencionaba anteriormente, se corre el peligro de que la alta capacidad de procesamiento de una IA suponga un pico de dificultad demasiado elevado para el jugador. Una máquina que es capaz de adaptarse a todo lo que haces y que sabe, al menos hasta cierto punto, cómo responder a cada uno de tus movimientos, puede resultar frustrante si no se limita de alguna forma. Por otro lado, **la posibilidad de enfrentarse a máquinas con un nivel de inteligencia similar, o incluso superior, al de un humano puede suponer un reto muy interesante para muchos jugadores**.

Capítulo 3

Objetivos y Especificación

Teniendo en cuenta el estado de la técnica, hemos comprobado que **no existe una herramienta potente** que permita programar un Sistema Multiagente BDI en un entorno de desarrollo de videojuegos como Unity. Si bien es cierto que "IsoMonks" (Sánchez-López, Romero y Martín-Solís, 2016) ofrece una propuesta de gran valor, **aunque se lograra un conector completo entre Jason y Unity se ha comprobado al usarlo que el tiempo dedicado a la programación es extremadamente alto**, ya que el programador debe cambiar constantemente de programa para compaginar *Front-End* (entorno en Unity) y *Back-End* (sistema multiagente Jason).

El **objetivo principal** de este trabajo es la creación de la infraestructura necesaria para acercar un Sistema Multiagente y **convertirlo en un componente nativo dentro del ecosistema de desarrollo de videojuegos de Unity**.

Para ello, analizamos el estado actual de la Inteligencia Artificial en los videojuegos como aproximación a los diferentes tipos de implementación utilizados para desarrollarlas. La idea es comprobar si los sistemas multiagente son utilizados con asiduidad en el desarrollo de estas IA's y, ya que han resultado no serlo, comenzar un proceso de traslación de dicha herramienta hacia Unity, como un software moderno y de de código abierto que permita crear estos agentes inteligentes de una manera cómoda e intuitiva en uno de los entornos de desarrollo de videojuegos más populares del mundo.

Se busca así **fomentar el uso de este tipo de IAs** en los videojuegos facilitando su implementación en un motor popular y de libre acceso para los desarrolladores.

3.1. Objetivos

Los objetivos propuestos para dar forma a este trabajo fueron los siguientes:

1. **Investigar** el estado de la técnica. Conociendo las bases de los sistemas multiagente y obteniendo una lista de ejemplos podemos delimitar mejor el margen de actuación de nuestro proyecto. Obtener una lista de **herramientas candidatas** para la programación de MAS en Unity o Unreal nos ofrece bastante información de las características comunes históricamente en software parecido al que queremos desarrollar, así como carencias a las que tenemos que prestar más atención para mejorar con nuestro trabajo el estado de la técnica.

2. **Realizar pruebas** conceptuales sobre otras herramientas de código abierto disponibles para el desarrollo de inteligencia artificial, concretamente de agentes BDI. Siguiendo con el trabajo de investigación teórica descrito más arriba, una **experiencia en primera persona de las herramientas candidatas** a solucionar nuestro problema nos proporcionarán un conocimiento más concreto de usabilidad, permitiéndonos ponernos en el papel de un desarrollador sin experiencia previa en el manejo de nuestra herramienta.

3. **Estudio de Jason** como sistema BDI aplicable al desarrollo de IA en videojuegos. Como mejor candidata para desarrollar agentes BDI, aunque no sea nativa de Unity, debemos estudiar el funcionamiento interno del sistema para realizar una correcta migración de sus funcionalidades.

4. **Reingeniería** de dicha herramienta y rediseño del código para poder migrarlo a Unity. La mejor opción para conseguir un sistema tan robusto parece **migrar el código de Java a C#**, manteniendo intactas las funcionalidades ofrecidas por el software. De éste modo, emplear nuestra herramienta para el diseño e implementación de agentes BDI no supondría una pérdida de características con respecto a la conexión de Jason y Unity, y permitiría ahorrar tiempo en realizar dicha conexión, además de eliminar el exceso de tiempo dedicado en cada depuración del juego a crear derivado de ejecutar separadamente y en orden el motor del MAS y el juego en cuestión.

5. **Implementación efectiva** del sistema multiagente nativo de Unity, que **permita llevar a cabo las funcionalidades básicas** de la arquitectura BDI sin depender de plataformas externas al motor de juego. A lo largo del proyecto (y por motivos detallados en el Capítulo 5),

el software producido en el cumplimiento de éste objetivo sustituyó al producido durante el objetivo anterior (aunque el conocimiento adquirido por el trabajo de reingeniería fue esencial para llevar a buen término éste). Dada la complejidad del sistema de Jason y, principalmente, la cantidad de software existente cuya utilidad no aplicaba a nuestra herramienta, conllevaba problemas o estaba incorrectamente implementada en su versión en Java, se optó en dedicar recursos a obtener el código mínimo para cumplir los requisitos esenciales.

6. **Implementación de una demostración** que permita demostrar que el sistema implementado **es funcional y permite el manejo de Agentes BDI**. Supone el objetivo final del proyecto, y supone la validación de requisitos. Obteniendo una demo funcional, se demuestra que el sistema desarrollado cumple con lo que se espera de ella.

3.2. Requisitos de la herramienta

Como comprobamos en el capítulo sobre el estado de la técnica, ya existen herramientas para manejar agentes inteligentes. También encontramos algunos ejemplos de software orientado a videojuegos basado en agentes inteligentes, entre los que se encontraban herramientas de desarrollo. Dichas herramientas fueron analizadas y expusimos el por qué descartarlas en las secciones correspondientes del Capítulo 2.

Así pues, como ninguno de los ejemplos descritos resultaba útil como herramienta genérica para ayudar a programar inteligencias artificiales a los desarrolladores de Unity, y ya que Jason es la mejor herramienta de programación BDI, **se decidió portar el código de Jason de Java a C#, cuyo entorno de actuación final es Unity**.

Aunque a lo largo del capítulo 5 detallaremos el proceso seguido, el tiempo dedicado a la reingeniería arrojó bastante luz sobre los requisitos esenciales en un sistema multiagente. Como ninguno de los presentes miembros del equipo habíamos tratado en profundidad con el código de Jason con anterioridad, los detalles técnicos de la misma nos eran ajenos, y hasta que no nos enfrentamos a la migración no perfilamos los detalles de nuestros objetivos.

3.2.1. Características de nuestra plataforma

La herramienta, a la que a partir de ahora llamaremos **Jasonity**, queda descrita a nivel funcional por nuestra Especificación de Requisitos.

Antes de comenzar a enumerar los requisitos de la herramienta Jasonity,

cabe remarcar que **el público al que está dirigida deberá tener una mínima base técnica** (conocimientos de C# y del paradigma BDI basado en ASL).

El objetivo es, en la medida de lo posible, que **el programador no deba ejecutar concurrentemente diversos programas** separando juego y lógica, diseñarlos empleando un exceso de entrornos de programación cuyo código se depura por separdo y facilitar la construcción de entornos que formen un Sistema Multiagente directamente en Unity.

Por esto deben cumplirse las siguientes **características**:

1. El código original en Java debe ser portado a C# para **evitar que el programador deba manejar a la vez demasiados entornos** de desarrollo y ejecución.
2. El código portado **no debe contener información específica del escenario, mecánicas ni de la implementación** del juego específico, para asegurar su generalidad y usabilidad en cualquier proyecto.
3. El **procesamiento del agente debe ser depurable**.
4. Debe realizarse el **procesamiento del agente sin realentizar** el desempeño del juego.
5. La herramienta debe disponer de una **interfaz que permita implementar los agentes de forma visual** para evitarle así al usuario el tener que aprender la sintaxis de Jason para hacerlo.

Si bien es cierto que la herramienta gráfica ha quedado fuera del alcance de nuestro desarrollo, los requisitos del 1 al 4 se han conseguido satisfacer con el resultado actual de *Jasonity*.

3.2.2. Especificación de Requisitos

Delimitados por las características arriba descritas, **los requisitos que debe satisfacer a nivel técnico** nuestra herramienta; fruto del conocimiento obtenido durante las priemras fases del proyecto y refinados por el proceso de reingeniería son los siguientes:

- **Manejador de ASL: El *parser***. Siguiendo el modelo de programación por capas, el analizador sintáctico (*parser*) es un requisito esencial en nuestro proyecto. Es el encargado de capturar los ficheros ASL específicos de cada agente y traducir toda la información obtenida del

archivo y escrita en su propio lenguaje en la representación de datos usada por las capas superiores de nuestro sistema. Por ello, un *parser* en nuestra plataforma debe cumplir los siguientes requisitos:

- Leer el contenido de ficheros con extensión .asl, contenidos en algún lugar del disco.
- Interpetar el string obtenido para distinguir lo descrito en el archivo en cuestión.
- Transformar lo interpretado en objetos usados por el sistema a nivel interno (literales, objetos plan, creencia o deseo) para que sea usable en la capa superior del sistema.

- **Implementación de agentes como conjuntos de deseos, creencias e intenciones**. Corresponde a uno de los elementos de la capa de negocio (de nuevo, siguiendo la analogía de la programación por capas). Como elemento central de un sistema multiagente, nuestra herramienta debe poseer una representación de un Agente BDI. Por ello, se deben cumplir los siguientes requisitos con respecto a los agentes:

  - Debe exisitir una representación de los deseos que puede poseer un agente, evitando tener preconstruida una librería de posibles deseos. Se debe evitar acoplar la herramienta del posible juego que el programador deseee implementar.
  - Debe existir una representación de las creencias que puede poseer un agente. Del mismo modo que con los deseos, no debe existir una colección previa de creencias que limite el uso de la herramienta.
  - Debe existir una representación de los planes que puede poseer un agente. Como en los dos casos anteriores, la definición de dichos planes debe estar plenamente definida por archivos asl propios de cada agene, evitando el uso de planes preconstruidos en el código de la herramienta.

- **Capacidad de los agentes de razonar**. Habiendo obtenido la implementación de agentes como un "contenedor de BDI", es imperativo que posea un razonador. Éste es el que verdaderamente llevará a cabo el proceso "inteligente" del agente:

  - Percibirá elementos del entorno para verificar o actualizar las creencias de las que dispone el agente.
  - Una vez percibido el entorno y acualizada una base de creencias, contrastará dichas creencias con los planes disponibles para ese agente y seleccionará el plan que mejor encaje para convertirlo en una intención.

  - Delegará las ejecuciones de las intenciones en un componente externo, ya que la información propia de Unity puede resultar muy pesada para concentrarla en un agente individual.
  - Mantendrá siempre un ojo en sus deseos, para intentar cumplirlos de forma óptima.

- **Capacidad de los agentes de realizar acciones**. Buscando un bajo acomplamiento en nuestro sistema, la representación del agente debe estar separada de la capacidad de actuación contra el entorno del mismo. Por ello, debemos satisfacer los siguientes requisitos:

  - Debe exisitr un manejador de tareas que lleve a cabo las intenciones seleccionadas y notificadas por los agentes.
  - El manejador de tareas debe incluir información de qué agente ha introducido cada tarea para poder manejarlas de forma apropiada.
  - El manejador de tareas debe disponer de una librería de acciones disponibles para satisfacer las mismas. Se trata de un módulo que debe implementar el desarrollador, ya que es la traducción de las acciones del agente sobre su entorno (pueden ser acciones como moverse en una dirección, moverse a un punto, interactuar con un interruptor, comunicarse con otro agente...)
  - El manejador de taeas debe poder notificar al agente generador de la tarea el efecto de la misma, de tal modo que el mismo pueda actualizar su base de creencias en base al evento o eventos derivados de la intención seleccionada y ejecutada.

Una vez quedan definidos los requisitos y los objetivos a cumplir, debemos exponer la metodología a seguir para alcanzar los objetivos (Capítulo 4); así como el diseño y análisis consecuencia de la búsqueda de satisfacer los requisitos (Capítulo 5 y Capítulo 6).

Capítulo 4

Metodología y Herramientas

Tras el capítulo 2 y 3, conocemos los objetivos que nos hemos propuesto derivados del estado de la cuestión. Por ello, podemos afirmar que sabemos **qué queremos** desarrollar.

A continuación, y a lo largo del presente capítulo, detallamos la metodología y herramientas que vamos a emplear para cumplir dichos objetivos, o lo que es lo mismo, **cómo** vamos a llevar a cabo el proyecto.

4.1. Metodología

Por la naturaleza de un Trabajo de Fin de Grado, **se ha seguido la Estructura de Descomposición del Trabajo (EDT)** para definir las diferentes etapas, hitos y tareas a realizar en el tiempo disponible.

Dado que la estimación de tiempo por nuestra parte es bastante imprecisa (falta de experiencia, uso de una tecnología nueva...) se ha seguido una **metodología ágil** para asegurarnos de que obteníamos una versión mínima lo antes posible que se fuera evolucionando a lo largo del tiempo.

Se ha seguido la estructura habitual por capítulos de la memoria para identificar las ramas de desarrollo que iban a tener lugar en nuestro trabajo.

Así, La división del trabajo y versiones queda reflejada en el diagrama presente en la figura 4.1.

Se puede ver a simple vista que existen tres ramas de desarrollo: *Front-End*, *Back-End* y *Demo*. Como queda reflejado en la Figura 4.1, existe una etapa temprana de investigación: antes de definir las etapas, los evolutivos

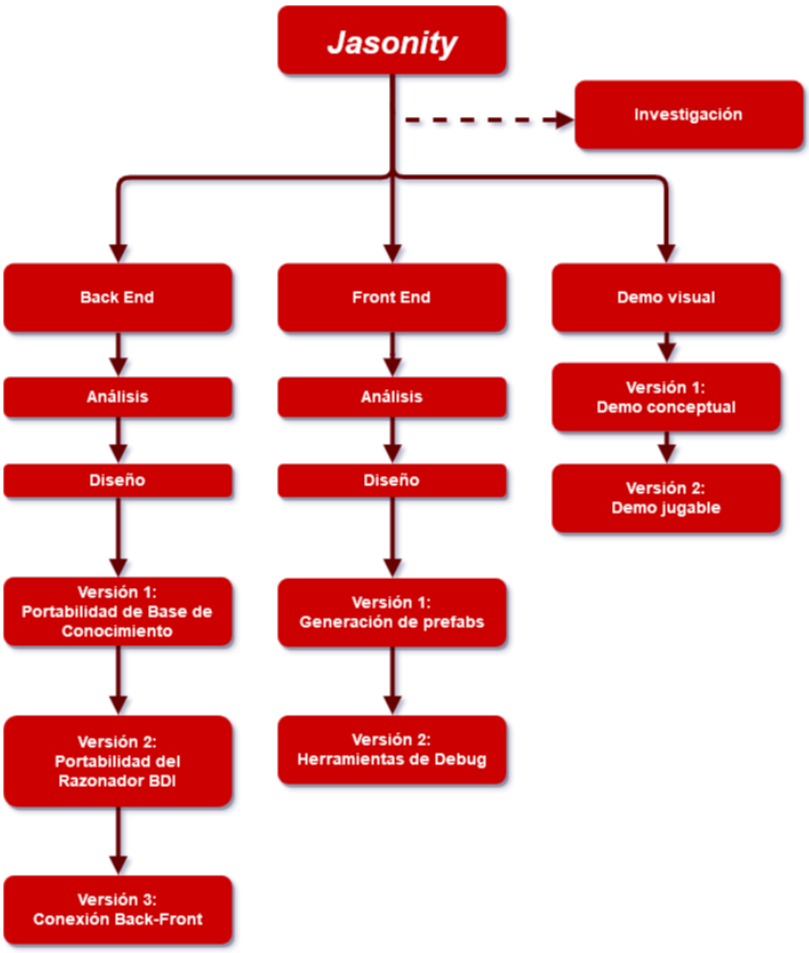

Figura 4.1: Diagrama EDT del TFG. Muestra las líneas de trabajo del proyecto y sus hitos evolutivos.

y qué funcionalidades y características componen el producto final, se han dedicado varios meses a investigar el estado de la técnica y al análisis del sistema software.

La rama del *back-end* se corresponde con el **desarrollo orientado a la arquitectura multiagente**, desarrollada nativamente en C# y encargada del procesamiento de los agentes.

La rama del *front-end* hace referencia a la **interfaz gráfica de nuestra herramienta**, que agiliza el proceso de diseño de agentes para el desarrollador.

Por último, la rama de la demo visual describe los hitos hasta alcanzar una **demostración gráfica que ilustre el funcionamiento** del software obtenido.

Una vez queda definido el flujo de entregables, se decide asumir el desarrollo **siguiendo una metodología ágil, concretamente *Scrum*, para dividir las etapas en *Sprints*** de dos semanas, coincidiendo con la frecuencia con la que el Director del Proyecto podía reunirse con el equipo.

Además, para satisfacer una clara necesidad de diversificación y especialización, **se ha dividido el equipo en varios grupos** con diferentes ámbitos de desarrollo.

4.1.1. Scrum: Descomposición del Desarrollo en Sprints

Como se ha mencionado en la sección anterior, se ha seguido la **metodología *Scrum* para el desarrollo del proyecto**.

En primer lugar, situamos al **Director del TFG como *Product Owner***. Las responsabilidades metodológicas del *Product Owner* en *Scrum* se ajustan al rol del mismo: Definir los items del *Backlog*, priorizar dichos items para alcanzar los objetivos, maximizar el valor del producto, asegurarse de que el *Product Backlog* está claro para todos los miembros del equipo y los motivos de la priorización.

También como *Cliente*, debe haber reuniones periódicas con él y debe estar presente a lo largo del proceso de desarrollo. Por ello, se establecieron reuniones cada fin de *Sprint* cada dos semanas para controlar los evolutivos, validar las tareas completadas y definir nuevas.

Además, dado el desconocimiento de las herramientas de BDI y los retos que puede suponer su traspaso a Unity, **la metodología ágil nos permite adaptarnos mejor a cambios** en los requisitos descubiertos en el propio proceso de desarrollo.

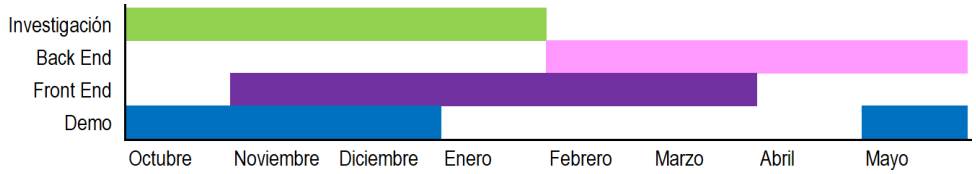

Figura 4.2: Diagrama Temporal de Tareas. Muestra la dedicación a cada una de las cuatro líneas de trabajo a lo largo del tiempo de proyecto (octubre-mayo).

4.1.2. Especialización: Descomposición del Equipo

Tal como se muestra en la Figura 4.1, existen 3 ramas del desarrollo relativamente paralelizables, por lo que resulta claro dividir el equipo por esas mismas ramas. Así, **se formaron los siguientes equipos**:

- **Investigación de lenguajes y herramientas**: Alejandro, Álvaro, Irene, Juan.
- ***Back-End***: Irene, Jaime, Álvaro.
- ***Front-End***: Juan.
- **Diseño e Implementación de la Demo**: Jaime, Irene, Juan.

De éste modo, **han podido aprovechar los conocimientos y destrezas de cada miembro del equipo**, dedicando el tiempo de desarrollo a uno o dos ámbitos muy concretos. Además, con equipos especializados, los mismos hemos podido focalizarnos en nuestros ámbitos y formarnos como "expertos" en ellos.

4.1.3. Desarrollo del trabajo

Si bien es cierto que algunos miembros pertenecieron a varios equipos a lo largo del desarrollo; y que los mismos no se solaparon en el tiempo, por lo general **la dedicación prioritaria de cada alumno fue la descrita**. La dedicación de tiempo ha sido descrita la descrita en la Figura 4.2.

La fase inicial se realizó **paralelamente entre la investigación de precedentes y el diseño de una demo** que mostrara el potencial de la arquitectura BDI. Sin embargo, tras estudiar varios casos y encontrarnos llegando al final del primer cuatrimestre, llegamos a la conclusión de que **el desarrollo de una demo jugable no era asumible**, y era redundante

con los trabajos realizados anteriormente por nuestros compañeros Sánchez-López et al. (2016) y González (2018) sobre BDI donde se muestran ejemplos válidos del uso. Sin embargo, en esos mismos trabajos quedaba latente la falta de una herramienta completa de BDI nativa de Unity. Así, el trabajo de demo se dejó hasta el final para construir un ejemplo práctico que empleara nuestro sistema.

Así, tras un primer cuatrimestre de investigación exhaustiva, y el diseño de una interfaz que pudiera dar las funcionalidades descubiertas, en el segundo cuatrimestre se pasó al desarrollo propiamente dicho de Jasonity.

Además, al formar parte del grupo de trabajo de Narratech Laboratories, se han añadido a los hitos del proyecto y a las fechas propias del TFG dos fechas adicionales propias del modo de trabajo del grupo. Tanto el martes 18 de diciembre de 2018 como el martes 9 de abril de 2019 y el miércoles 5 de junio de 2019 (antes de las vacaciones de Navidad, Semana Santa y Verano, respectivamente) se llevaron a cabo exposiciones dentro de Narratech que obligaron a todos los proyectos que se realizan bajo dicho grupo (TFGs, TFMs y Tesis Doctorales) a **cerrar el desarrollo hasta conseguir un producto mínimamente enseñable** al resto de compañeros.

Si bien es cierto que se han mantenido las reuniones bisemanales, la presencialidad de las mismas ha variado entre la reunión física o vía *Skype* dependiendo de las necesidades del equipo. De éste modo se ha podido cumplir rigurosamente la duración de todos los *Sprints*.

4.2. Herramientas del Proyecto

Para llevar a buen término el proyecto se emplearon diferentes herramientas, tanto productivas (como entornos de programación o de redacción de la memoria) como metodológicas (tablero de tareas, control de versiones, etc.).

4.2.1. Entornos de Programación, Tecnologías y Lenguajes

Forman el grueso de las herramientas productivas empleadas por el equipo para el desarrollo del proyecto. Aunque las principales son Unity con C# y Visual Studio / Code, hay otras que cabría mencionar.

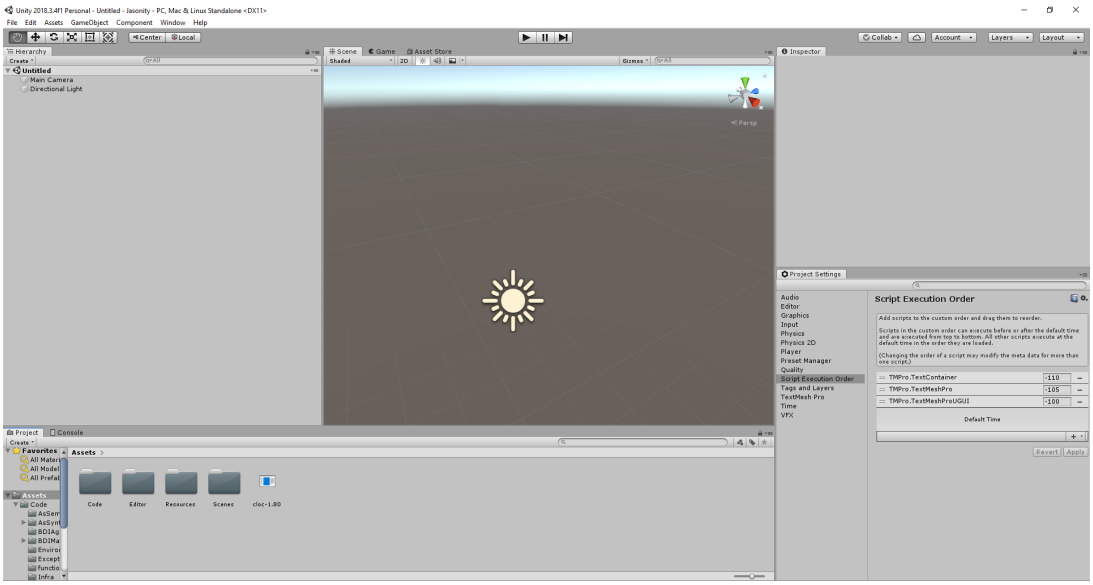

Figura 4.3: Interfaz general de Unity, ejecutado en Windows 10, con la configuración por defecto de ventanas.

4.2.1.1. Unity

Como ya se ha presentado en la sección 2.4.3, **Unity es junto con Unreal Engine uno de los motores de videojuegos más usados** a día de hoy (y por ello se ha escogido para éste proyecto). Se puede emplear para juegos de todos los espectros gráficos y jugables. Es gratuito para el uso académico/experimental, y la versión "PRO" no es de uso obligatorio hasta que la entidad que lo emplea tiene unos ingresos de $100K al año.

Además, **el editor (Figura 4.3) es personalizable y permite programar nuevas ventanas** (gracias a lo cual se puede diseñar la interfaz de Jasonity). Para poder realizar bastantes funciones para el manejo de los agentes se han empleado multitud de *prefabs*. A través de éste tipo de elemento el motor permite crear conjuntos de objetos y *scripts* preconstruidos, manejables como si fuera uno solo. Esos objetos son alojados como un recurso y se pueden crear instancias en la escena del juego.

En nuestro caso, las funciones propias de la programación de videojuegos no se han empleado hasta la implementación de la demo, y no se ha profundizado en todas las posibilidades que ofrece el motor. El mayor uso que se le ha dado a Unity es a su motor interno, escrito en C#.

Gracias al mismo **hemos podido portar las funciones de *Jason* en materia de BDI a Unity** sin necesidad de implementar los detalles más

mínimos (uso y manejo de hilos, actualización periódica del estado del juego, etc.).

4.2.1.2. Visual Studio / Visual Studio Code

Es el **entorno de desarrollo de código** por defecto de Unity. Gracias a éste, se pueden desarrollar los *scripts* que componen Jasonity en C#. Además, es el entorno de programación que se empieza a usar desde el primer año de carrera, por lo que su elección es más evidente. Su **fácil integración con Unity** para realizar una depuración paso por paso hace de Visual Studio el editor ideal para nuestro proyecto.

Sin embargo, se trata de un entorno que **puede llegar a consumir muchos recursos** de un ordenador, por lo que se ha complementado su uso con Visual Studio Code. Ésta es una **versión más ligera de Visual Studio**, que permite la edición de texto con una comprobación básica a nivel sintáctico. Sin embargo, existen carencias a la hora de depurar las soluciones, por lo que se trata de una herramienta puramente auxiliar.

4.2.1.3. Jason / jEdit

Se trata de la **herramienta de referencia para el manejo de agentes en BDI**. Lo componen Jason y jEdit. Jason es una librería basada en Java que sirve de intérprete de *AgentSpeak*. Su entorno de programación gráfica es jEdit. Éste permite editar los ficheros *ASL*, configurar los proyectos, ejecutarlos y depurarlos...

Al tratarse de un **proyecto de código abierto**, el código fuente de Jason ha servido de referencia para portar su código a nuestro proyecto.

4.2.1.4. Eclipse

Es el **entorno de programación** de código abierto **más extendido para el lenguaje Java**. Nos ha permitido navegar por las referencias existentes entre clases de Jason y depurar el código a fin de comprender el funcionamiento y las llamadas que se producen en el mismo.

4.2.1.5. JavaCC

Como se detallará en la sección 5.1.2.3 de éste mismo trabajo, JavaCC (*Java Compiler Compiler*) es un **generador de *parser* de código abierto**

de uso bastante extendido y que permite leer archivos que especifiquen la gramática de un lenguaje, y JavaCC lo convierte en un programa Java capaz de leer e interpretar dicha gramática.

Se propuso como herramienta que descargara algo de trabajo a la hora de realizar la interpretación de ficheros ASL.

4.2.2. Control de Versiones y Recursos Compartidos

Aunque es recomendable poseer siempre un repositorio donde alojar todos los archivos relevantes de un proyecto, al ser un equipo de cinco personas y un director, era estrictamente necesario que tanto código como cualquier tipo de información estuviera disponible en todo momento para todos los miembros. Para ello se emplearon GitHub y Google Drive de la forma descrita a continuación.

4.2.2.1. Git y GitHub

Se trata de una de las herramientas más extendidas de trabajo colaborativo software. **GitHub es una plataforma donde alojar proyectos** y trabajar concurrentemente usando el sistema Git.

Para llevar un control exhaustivo de todos los cambios y asegurarnos de que todos teníamos acceso a todo en cualquier momento, **hemos utilizado GitHub como plataforma de control de versiones**. Contamos con un repositorio compartido llamado ***Jasonity***[1] sobre el que todos los miembros del equipo y el director del proyecto podemos trabajar en paralelo.

Gracias al mismo, hemos podido mantener una rama principal del proyecto con las últimas características estables y diversas ramas paralelas, o bien con el trabajo de cada miembro o bien con diferentes funcionalidades independientes.

En nuestro caso, hemos dividido el proyecto en las siguientes ramas principales:

- **Master**: versión más estable disponible.
- **Desarrollo**: rama de funcionalidades más completas implementadas, a la espera de ser completamente depuradas para empezar las pruebas.
- **Editor**: rama dedicada a la implementación del entorno visual.

[1] https://github.com/Narratech/Jasonity

También permite mantener el **código abierto y accesible a cualquier usuario** ajeno al proyecto, y en nuestro caso está el respoitorio público bajo una licencia de software LGPL-3.0.

Además, nos ha permitido a lo largo del desarrollo restablecer versiones anteriores del código cuando se ha llegado a puntos muertos del mismo o cuando se han cometido errores que de otro modo hubieran sido fatales.

4.2.2.2. Google Drive

Para todos aquellos **archivos que no fueran estrictamente del software**, hemos empleado *Google Drive* como plataforma para alojar recursos compartidos.

En una carpeta compartida hemos guardado elementos como las actas de las reuniones que hemos tenido, referencias para la investigación y elaboración de la memoria, presentaciones, un horario común para organizar las reuniones bisemanales, documentos de diseño, etc. Las actas más relevantes y los *Sprint Backlogs* están disponibles en forma de apéndice al final de la memoria (Apéndice D).

Además, gracias a la cuenta UCM el **espacio disponible para éste fin ha sido ilimitado**.

4.2.3. Herramientas para la Comunicación y el Seguimiento del Proyecto

Cabe destacar que, siendo un grupo conformado por cinco alumnos y un director, el seguimiento de la metodología ha sido tan complicado como necesario. La coordinación en el equipo ha sido esencial para mantener un desarrollo sólido, coherente y eficiente.

Por ello, hemos empleado tantas herramientas como nos ha sido posible para evitar que el descontrol se adueñara del proyecto.

4.2.3.1. Slack

Se trata de un popular **servicio de mensajería (web y móvil) en el mundo del desarrollo** software.

Permite crear diversos entornos de trabajo con diferentes miembros de un grupo de usuarios, lo que da lugar a crear un servidor común de usuarios (en nuestro caso, los miembros de Narratech Laboratories) y crear salas de

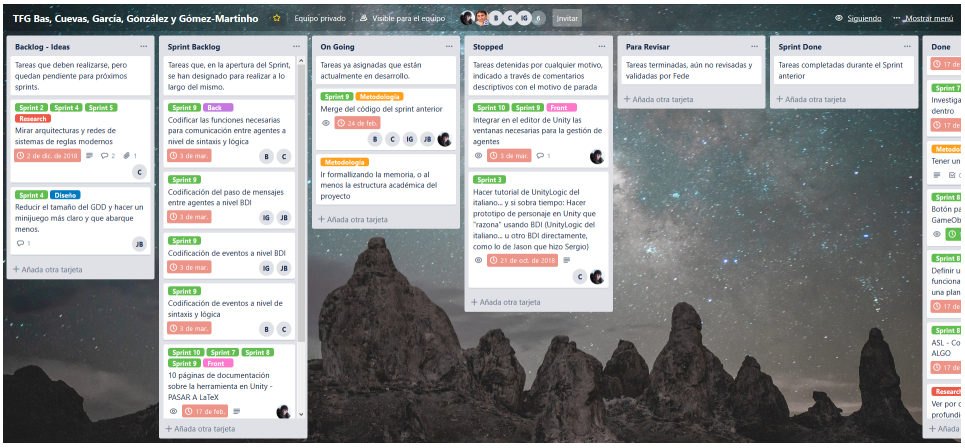

Figura 4.4: Captura de nuestro tablero de Trello. Muestra un estado intermedio de *Sprint* con tareas en el *Sprint Backlog*, *On Going* y *Stopped*, principalmente.

chat diferenciadas por proyectos (general, TFGs en general, nuestro TFG, desarrollo *Unity*, IA...). Además, permite la conexión con otros servicios (p.ej: Google Drive)

4.2.3.2. Trello

Es una plataforma (web y móvil) que **simula un tablero de equipo, a través del cual gestionar equipos y tareas**.

Siguiendo la metodología Scrum, hemos configurado nuestro tablero para tener una columna con el *Product Backlog*, el *Sprint Backlog*, tareas en desarrollo (*On Going*), tareas detenidas (por bloqueos por terceros, problemas en el desarrollo...), tareas para revisar por parte del director, las completadas en el sprint y las ya completadas a lo largo del proyecto.

Éste tablero **se revisaba en cada reunión bisemanal**: se comprobaban las tareas completadas en el último *Sprint* y se definían las tareas del *Sprint* siguiente, juntando así la Reunión de Cierre de *Sprint* con la de Apertura. Una atualización en tiempo real nos ha permitido tener un control en todo momento del estado de las tareas.

En la figura 4.4 se muestra el tablero usado en el proyecto a lo largo del noveno sprint.

4.2.3.3. Skype

Es el conocido **programa de videoconferencias** de Microsoft. Nos ha permitido tener sin falta las reuniones de equipo aunque algún miembro no estuviera disponible de forma presencial, permitiéndonos además compartir la pantalla de alguno de los usuarios para revisar el trabajo realizado de manera remota.

4.2.3.4. Google Docs

Como se ha mencionado en la sección referente a Google Drive, se han **redactado algunos documentos comunes** a través del editor de texto *on-line* de Google. Usando ésta herramienta, se ha podido redactar concurrentemente entre varios miembros del equipo; y queda inmediatamente disponible en la carpeta compartida.

4.2.4. Redacción de la Memoria

Aunque se trata de un lenguaje de difícil aprendizaje, se ha escogido LaTeX (lenguaje a su vez derivado de TeX) como maquetador de la memoria por la posibilidad de usar plantillas ofrecidas por la Universidad Complutense de Madrid para la elaboración de una memoria limpia, estandarizada y visual (TeXiS).

4.2.4.1. Overleaf

Para la redacción de la memoria hemos utilizado Overleaf, un **editor concurrente *on-line* de LaTeX**, basándonos en la **plantilla TeXiS**, diseñada para tesis y otros trabajos académicos.

Todos los participantes del grupo hemos participado en la redacción de la memoria, centrándonos cada uno en los aspectos que más hemos tratado durante el desarrollo del proyecto. Usando Overleaf, **hemos podido trabajar a la vez sobre el mismo documento, redactando secciones diferentes** de la misma de una forma práctica y cómoda. Además, facilita el trabajo de revisión y corrección **permitiendo añadir comentarios** como anotaciones sobre partes del texto, firmadas por el usuario que las redacta y abriendo un hilo de respuestas para poder debatir las correcciones o resolver dudas de cómo enfocarlas.

Capítulo 5

Jasonity: Diseño, Análisis y Reingeniería

Por las características propias de la herramienta que queríamos desarrollar, la herramienta se diseñó con una **arquitectura multicapa, con la intención de paralelizar** al máximo todo el trabajo posible y minimizar la dependencia entre las diferentes capas de la misma.

La arquitectura está dividida en:

- **Modelo lógico**: capa en la que se implementa toda la sintaxis y la semántica de AgentSpeak.
- **Sistema multiagente y razonador**: capa en la que se implementa el motor de razonamiento BDI, los deseos, creencias e intenciones y el agente.
- **Vista y control**: capa en la que se implementa la parte gráfica de la herramienta.

De todas las posibles implementaciones se optó por una **arquitectura centralizada** ya que, aunque Jason permite sistemas multiagente descentralizados usando diversos estándares para computación distribuida, debido al alcance de nuestro trabajo en Jasonity sólo tiene sentido cubrir la realización de sistemas multiagente en un sistema centralizado.

A lo largo de la Sección 5.1 detallamos el diseño y la implementación de la capa del modelo lógico, en la siguiente la arquitectura e implementación de la capa BDI y en la última sección la arquitectura e implementación de la capa de vista y control con el front-end del sistema.

5.1. Diseño y arquitectura de la capa lógica

En primer lugar nos centraremos en lo que podría considerarse la capa más baja: la Capa Lógica. Ésta **conecta el Parser con el sistema BDI**, para que se pueda ejecutar el razonamiento en base a elementos concretos y formateados para que dicho componente superior pueda usarlos.

5.1.1. Análisis

La capa lógica **está formada por todas las clases e interfaces que manejan la información recibida a través del parser que necesita el sistema BDI** para su funcionamiento; esto es, términos, expresiones lógicas y aritméticas, literales, variables y variables anónimas, planes y reglas. De igual forma, también contiene el parser generado automáticamente, ya que necesita las clases anteriormente citadas para funcionar correctamente. Todas estas clases e interfaces están contenidas dentro de la carpeta *AsSyntax*.

A continuación se enumerarán y describirán las **clases que componen esta capa**:

- **Term** Cualquier elemento que componga una instrucción de *AgentSpeak*.

  Es la interfaz que acabarán implementando casi todas las clases de esta capa.

  Es clonable, serializable y comparable. Define una función booleana por cada clase que implementa esta interfaz que devuelve si son un tipo de término en concreto o no.

- **Literal**. Representa cualquiera de los componentes que forman una regla o un plan.

  Clase abstracta que representa varios tipos de clases. Esencialmente, es un **Term** con más funcionalidad y menos genérico.

  A diferencia de este último, no abarca también reglas ni planes, sino sus componentes internos.

  Hereda de **DefaultTerm** e implementa **LogicalFormula**. Ya que es la clase padre de muchas otras de esta capa, como **Structure** o **Predicate**, contiene e implementa métodos comunes a todas ellas, como comprobar la presencia de variables o si el literal debería ser borrado de la base de datos o no.

  Además, implementa un iterador personalizado para recorrer las expresiones lógicas y contiene tres clases internas para crear los llamados **functors** del Literal, dependiendo de si el Literal es falso o cierto.

- **LiteralImpl**. Amplia un **Pred** añadiendo la fuerte "negación".

  Hereda de **Pred**.

  Contiene diferentes constructores según los tipos de parámetros recibidos para establecer el tipo de negación de este *Literal*.

- **AsSyntax**. La factoría empleada para la creación de objetos usados en la sintaxis *AgentSpeak* de *Jason*.

  Aquí es donde se llama al *Parser* para convertir el código de los ficheros **ASL** en clases que necesita la **capa BDI** para funcionar.

  Contiene las funciones encargadas de llamar al *Parser* por cada tipo de término que recibe como *string* por parámetro.

- **ArithExpr**. Clase empleada para representar y resolver expresiones aritméticas como sumas, restas, multiplicaciones, divisiones, porcentajes, módulos, etc.

  Hereda de **ArithFunctionTerm** e implementa **NumberTerm**.

  Contiene enumerados que implementan las operaciones aritméticas y su propia llamada al **Parser** para obtener el término que puede evaluarse como número.

- **ArithFunctionTerm**. Representa una función aritmética con dos argumentos, por ejemplo: "math.max(arg1, arg2)".

  Hereda de **Structure** e implementa **NumberTerm** y computa el valor de la función aritmética.

- **Atom**. Representa un **Literal** positivo sin argumentos ni anotaciones: "a", "run". Es el equivalente al concepto de constante en otros lenguajes lógicos, como **Prolog**, por ejemplo.

  Hereda de **Literal**.

  Contiene los constructores necesarios para crear este término y las funciones requeridas para las operaciones de comparación, clonación y devolución de atributos.

- **BinaryStructure**. Permite relacionar operadores unarios o binarios mediante operadores lógicos y relacionales.

  Hereda de **Structure**. Contiene los constructores y las funciones necesarias para devolver los atributos privados.

- **InternalActionLiteral**. Es un tipo particular de Literal usado para representar las **InternalActions**.

  Hereda de **Structure** e implementa **LogicalFormula**.

Implementa diferentes constructores según vayan a ser invocados desde el *Parser* u otras clases, así como un iterador para recorrer los resultados de las fórmulas lógicas.

- **DefaultTerm**. La clase base para todos los demás términos.

  Implementa **Term** y **Serializable**

- **ListTerm**. Interfaz para las listas de términos (**Terms**) de *AgentSpeak*. Hereda de **Term** e implementa de la propia interfaz *IList* de C#.

- **StringTerm**. La propia versión de un *string* implementada por *Jason*. Hereda de **Term** y es la interfaz que contiene la definición de las funciones que usará la clase que la implementa.

- **StringTermImpl**. Hereda de **DefaultTerm** e implementa **StringTerm**.

  Contiene su propia llamada al *Parser* para obtener los término *string* del término recibido y las funciones para comparar entre sí dichos términos una vez traducidos.

- **CyclicTerm**. Representa un término cíclico (recursivo) , creado por código del tipo: "X = f(X)".

  Hereda de **LiteralImpl** e implementa las funciones y métodos necesarios para clonar, comparar y editar este tipo de términos.

- **NumberTerm**. Interfaz para la interpretación de un número de *Jason*.

  Hereda de **DefaultTerm** y devuelve el valor numérico del término.

- **NumberTermImpl**. Clase inmutable que implementa el **Term** que representa un número.

  Hereda de **DefaultTerm** e implementa **NumberTerm**. Contiene las funciones y métodos necesarios para clonar, comparar y editar este tipo de términos.

- **LogicalFormula**. Interfaz que devuelve si el resultado de una fórmula lógica es consecuencia de una creencia de la base de datos.

  Hereda de **Term** e implementa la interfaz **ICloneable** de C#.

- **LogExpr**. Representa una expresión lógica con alguna clase de operador lógico, como: "&", "|" o ";".

  Hereda de **BinaryStructure** e implementa **LogicalFormula**.

  Contiene un iterador para recorrer los resultados de la fórmula lógica.

- **Plan**. Formado por un disparador, un contexto y un cuerpo del plan. Cuando se activa el disparador (**Trigger**), se comprueba si el contexto del plan se cumple y, de ser así, se ejecuta el cuerpo del plan (**PlanBody**).

  Es serializable, hereda de **Structure** e implementa la interfaz **IClonable** de C#.

  Tiene métodos y funciones para modificar y devolver los distintos elementos del plan.

- **PlanBody**. La interfaz para los elementos del cuerpo del plan.

  Hereda de **Term** e implementa un enumerado que devuelve un equivalente *string* a cada distinto valor que contiene.

- **PlanBodyImpl**. Compuesto a partir de una serie de términos (los cuales pueden ser objetivos y creencias) así como de *InternalActions* que el agente ejecutará si el plan se inicia.

  Hereda de **Structure** e implementa **PlanBody** e implementa la interfaz **IEnumerable** de C#.

  Ejemplo de un Plan: +!leave(home): not raining & not raining <- !location(window);?curtaintype(Curtains);open(Curtains).

- **PlanLibrary**. Representa un conjunto de planes utilizados por el agente. Es la clase donde el agente almacena dichos planes.

  Implementa la interfaz **IEnumerable** de C#.

  Contiene los métodos y funciones requeridas para añadir, borrar, recorrer, devolver; en resumen, gestionar, lo que es la "base de datos" del agente.

- **Structure**. Se utiliza para representar datos complejos (individuos u objetos). Es una clase formada por un átomo llamado *functor* y por otros literales comprendidos entre paréntesis a continuación de dicho átomo, llamados .ªrgumentos".

  Hereda de **Atom**.

  Ejemplo de una *Structure*: likes(john, music).

- **Pred**. Es una estructura (**Structure**) con anotaciones. Las anotaciones son términos complejos que dan detalles fuertemente asociados con una creencia en particular.

  Clase que consta de una *Structure* seguida de "anotaciones": términos complejos que dan detalles fuertemente asociados con una creencia en particular.; las cuales, ahora hablando de manera técnica, son otros literales comprendidas entre corchetes situados a continuación de los argumentos de las estructura.

Hereda de **Structure**.

Ejemplo de un *Pred*: busy(john)[expires(autumn)].

- **PredicateIndicator**. Representa el tipo de un predicado (**Pred**) según el primer término (**Atom**) por el que empieza, llamado *functor*, y su aridad.

  Es serializable e implementa la interfaz **IComparable** de C#.

- **RelExpr**. Es una expresión relacional del tipo: "10 >20".

  Hereda de **BinaryStructure** e implementa **LogicalFormula**.

  Contiene un enumerado que devuelve un *string* equivalente a cada valor que contiene.

- **Rule**. Permite al Agente adquirir nuevos conocimientos basándose en los que ya tiene almacenados en su base de conocimientos.

  Es una clase compuesta de un *Literal* más un cuerpo separados por este símbolo: ":-". Este cuerpo contiene otros *Literals* que, de cumplirse sus condiciones en el entorno o en el Agente, este adquiere la conclusión (el *Literal* a la izquierda de ":-") para su base de conocimientos.

  Hereda de **LiteralImpl**.

  Ejemplo de una *Rule*: likelycolour(C,B):- colour(C,B)[source(S)] & (S == self|S==percept).

- **SourceInfo**. Guarda información sobre el fichero fuente de algunos términos.

  Implementa la interfaz **Serializable**.

- **Trigger**. El disparador que pone en marcha el plan al que está ligado cuando se activa.

  Es una clase que recibe un tipo de operando, una creencia o deseo y un *Literal*. El operando indica si la creencia, o el deseo, recibido como un *Literal* debe ser borrado (si el oprenado es igual a "-") o añadido (igual a "+") a la base de conocimientos del agente en caso de que se cumpla el *Plan*.

  Es serializable.

  Ejemplo de un *Trigger*: +!leave(home):

- **Var**. Representa a cualquier variable no anónima.

  Es un término (**Term**) del tipo "X": todas empiezan por letra mayúscula independientemente de los caracteres que tengan, y carece de un valor predeterminado hasta que le es asignado uno y, entonces, mantiene ese valor para todo su ámbito. En *AgentSpeak*, el ámbito es el plan donde aparece.

Hereda de **LiteralImpl** e implementa **NumberTerm** y **ListTerm**.

- **UnnamedVar**. Representa la variable anónima "_".

  Funciona como una variable normal, pero no fija su valor a un ámbito. Esto quiere decir que se puede modificar desde otro plan.

5.1.2. Pruebas de concepto

Una vez explicadas las principales clases que componen la capa de lógica, iniciaremos un **recorrido por las distintas pruebas** que se llevaron a cabo con la finalidad de lograr reescribir esta capa en C#. Primero con una versión reducida de la misma, pensada para una posible primera versión también reducida de **Jason**, y, finalmente, una traslación total de las clases de Java a C# al descartarse la anteriormente citada versión simplificada de *Jason*.

5.1.2.1. Parser simplificado y clases reducidas

Lo primero que se hizo fue dividir el trabajo en dos partes que se irían desarrollando paralelamente: una **versión simplificada del *Parser* hecha manualmente y una segunda versión que se generaría automáticamente** a partir de unos ficheros modificados de Java para C# (la cual ya indicamos al comienzo de la sección 5.1 que describiríamos aparte).

El *Parser* manual se desarrolló de tal forma que pudiera leer línea a línea ficheros de texto con una sintaxis simplificada de *AgentSpeak* en su interior. Así pues, esta versión del lenguaje incluiría: **Atom**, **Pred**, **Structure**, **Var**, **InternalAction**, **Plan**, **Trigger**, y **PlanBody**.

Al finalizar esta primera prueba, el *Parser* manual era capaz de procesar las variables, átomos, reglas, estructuras y predicados del fichero y clasificarlas según fueran creencias, deseos o intenciones. Sin embargo, al finalizar esta etapa, debido al *feedback* recibido de los responsables de la capa BDI, se llegó a la conclusión de que **Jason no puede reducirse** y un **Parser simplificado no funcionaría** dentro del conjunto de Jasonity, además de que el traductor automático estaba pensado para generarse y funcionar a partir de las clases de "Syntax" originales de Jason. De modo que se apartó esta versión simplificada del *Parser* del proyecto y se decidió transcribir las clases originales a C# para Jasonity.

5.1.2.2. Reimplementación de AsSyntax en CSharp y creación de un manual de migración Java-C#

Su **implementación es equivalente a la versión original de Jason en Java**, con algunas diferencias en lo que respecta a la localización de ciertas clases como el *Parser*, que ahora se encuentra en una subcarpeta dentro de la propia carpeta AsSyntax y la exclusión de la clase BodyLiteral la cual, por propia advertencia en los comentarios de la misma, no se implementó y fue sustituida por un enumerado dentro de una subclase de PlanBodyImplm.

Cabe remarcar antes de comenzar que durante el desarrollo de los siguientes apartados se hablarán de forma recurrente del *Parser* automático, pero no se profundizara mucho en su implementación ya que, debido a que se fue desarrollando paralelamente durante todas las versiones de la implementación y a la magnitud de la misma, tiene su propia sección al final de esta sección 5.1, donde se describe apropiadamente.

Para esta etapa **se reimplementaron todas las clases de la carpeta AsSyntax en C#**. Durante este proceso **surgieron bastantes problemas** debido a las diferencias entre Java y C# y hubo que modificar y ajustar una gran cantidad de métodos y funciones debido a que eran su implementación original en Java era completamente inviable en C# (dichos errores se describen en un apartado propio fruto de la ingente cantidad de incompatibilidades entre ambos lenguajes, así como el desconocimiento de muchas de las clases de C# que pudieran servir de equivalente a las utilizadas en Java).

Esta situación **propició la creación de una *Wiki* que sirviera como una referencia de clases y funciones equivalentes entre Java y C#**, de cara a ahorrar tiempo en buscar de nuevo dichas equivalencias en la web y como guía para futuros trabajos de temática similar.

Al término de esta etapa, todas las clases de la carpeta AsSyntax estuvieron reimplementadas en CSharp, pero aún quedarían bastantes errores de sintaxis en el código que se repasarían durante varias revisiones posteriores, en muchas de las cuales se necesitó la ayuda del director del TFG para solventarlas; e incluso así, para fallos relacionados con multihilo y procesamiento en paralelo hubo que ponerse en contacto con expertos en este tipo de procesos en Unity, como los miembros del grupo de investigación Narratech, del que nuestro director es fundador.

5.1.2.3. JavaCC

Java Compiler Compiler es uno de los **generadores de *parsers*** de código abierto más conocidos y se puede utilizar para **crear sistemas completos, específicos y personalizados**. En el capítulo 4.2.1.5 hablamos de las principales virtudes de esta herramienta, en este apartado hablaremos de cómo hemos utilizado esta herramienta para nuestro proyecto: crear un parser personalizado capaz de interpretar *AgentSpeak*.

5.1.2.4. Primeros pasos con JavaCC

Uno de los primeros problemas que presenta la herramienta es la **falta de documentación** accesible a usuarios poco familiarizados con la herramienta. A pesar de ser uno de los generadores de *parsers* más populares, la gran mayoría de información disponible está dirigida a un público muy especializado.

Las capacidades de *JavaCC* no eran suficientes para nuestros objetivos: necesitábamos un parser en C# para que fuera compatible con el resto del código. La aplicación utiliza un archivo en Java y genera los archivos necesarios para el parser en Java. Aquí tuvimos que considerar **varias alternativas**:

1. Generar el parser en Java y traducir todo el código generado a C#
2. Buscar una alternativa a JavaCC que genere un parser similar en C# a partir de un archivo en Java.
3. Buscar una alternativa a JavaCC que genere un parser similar en C# a partir de un archivo en C#.

Se optó por **buscar una herramienta similar a JavaCC que nos generase el parser en C#**. Se investigó qué herramientas podrían ayudarnos a llegar a este fin y dimos con una herramienta llamada **CSJavaCC**: un *port* de la versión 4.0 de *JavaCC* que nos permitía generar el parser en C#. Debido a la falta de documentación para aprender a utilizar la herramienta y la falta de funcionalidades necesarias para la versión de Jason que utilizamos como modelo, también se descartó su utilización.

Finalmente encontramos la solución en el propio código de *JavaCC*. En las últimas versiones del código, **los creadores del repositorio de GitHub explicaban que estaban desarrollando una funcionalidad** que generaba el parser en C++, y estaban haciendo lo mismo para poder generar el parser en C#.

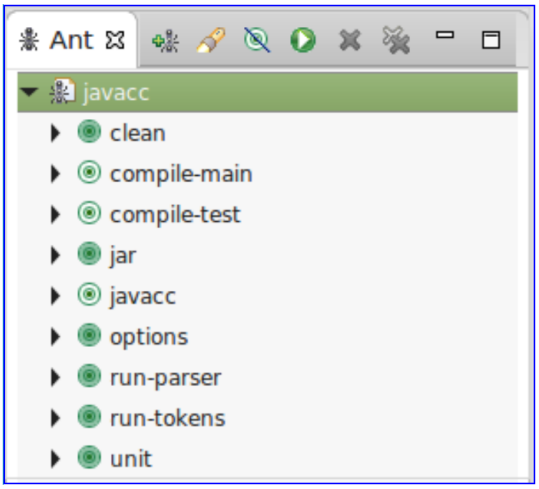

Figura 5.1: Ventana ant con el archivo build.xml introducido. Muestra los componentes de JavaCC

Contactamos con los creadores del repositorio para comprobar si se había llegado a algo con el traductor a C#. Tras unos días de espera, nos confirmaron que una de las ramas del repositorio **contenía todo lo necesario para poder generar el parser en C#**. Con esta nueva información, lo único que quedaba era aprender a utilizar la herramienta *JavaCC* y generar el parser en C#.

5.1.2.5. Uso de la herramienta JavaCC

En primer lugar, debemos tener instalado Eclipse para poder trabajar con el proyecto de *JavaCC*. Utilizaremos la ventana *ant* para generar el archivo *javacc.jar*, que nos servirá para generar los archivos que necesita el parser para funcionar.

Pasamos a cargar el proyecto de JavaCC al entorno de Eclipse, buscamos el archivo *build.xml* en el explorador de archivos de Eclipse, arrastramos este archivo hasta la ventana *ant* que acabamos de abrir y obtenemos algo como la Figura 5.1.

Ya podemos generar nuestro archivo *javacc.jar*, simplemente hacemos doble clic sobre la pestaña de *jar*. Nos aparecerá por consola los distintos procesos que realiza hasta generar lo que necesitamos, esperamos a que aparezca el mensaje de *BUILD SUCCESSFUL*. Sabemos que lo hemos bien y ha generado el archivo que queremos porque aparece la siguiente linea:
Building jar: *Ruta del proyecto*\javacc-master\target\javacc.jar

```
Java Compiler Compiler Version 7.0.5 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file ejemplo.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
```

Figura 5.2: Mensajes mostrados por el parser tras la ejecución de un comando de ejemplo.

Refrescamos el proyecto y nos aparecerá una nueva carpeta llamada *target* donde se encuentra el archivo que necesitamos para generar nuestro parser *javacc.jar*.

Solo queda generar nuestro parser a partir de nuestro archivo *ejemplo.jj* que contiene la descripción de nuestro lenguaje. Abrimos la pantalla de comandos y escribimos el siguiente comando (El archivo con nuestro lenguaje y el *javacc.jar* deben estar en la misma carpeta):
java -classpath "javacc.jar" org.javacc.parser.Main ejemplo.jj

Aparecerá por pantalla una serie de mensajes confirmando que si no existían los ficheros que forman el parser se han creado. (Figura 5.2).

Ya tenemos nuestro parser en lenguaje *Java* creado con las especificaciones que le hemos pedido en el archivo *ejemplo.jj*. Los archivos que forman el parser son:

1. ***DemoParser.java***: El nombre de las tres primeras clases viene determinado por el nombre que le hayamos dado en el archivo *ejemplo.jj*, posteriormente hablaremos del contenido de este fichero. Este archivo es la parte central del parser, contiene las distintas clases y métodos que permiten que al leer un archivo el parser sepa encontrar errores en su gramática.

2. ***DemoParserConstants.java***: Esta clase contiene los distintos símbolos o caracteres que reconoce nuestro parser.

3. ***DemoParserTokenManager.java***: Factoría que se encarga de interpretar la información que van recogiendo los distintos *tokens*.

4. ***ParseException.java***: Utilizamos esta clase para poder crear excepciones que se lancen cuando se encuentre un error en el parser o a la hora de leer la gramática de un fichero externo.

5. ***SimpleCharStream.java***: Implementación de la interfaz CharStream, lo usamos para recoger información de los distintos caracteres que vamos encontrando en la gramática.

6. ***Token.java***: Describe cada *token* que encontramos en el fichero de entrada de la gramática.

7. ***TokenMgrError.java***: Recoge los distintos errores que podemos encontrar al leer los distintos *tokens* de un fichero.

De esta forma podemos utilizar la herramienta de *JavaCC* para crear nuestro parser en lenguaje *Java*; pero, como ya hemos mencionado antes, necesitamos que la herramienta nos genere el código en C# para ello debemos realizar unos pequeños cambios en el proceso que hemos seguido hasta ahora.

A parte de generar el ejecutable *javacc.jar* tenemos que generar el ejecutable *csharp-codegenerator.jar* que nos permitirá generar nuestro parser en C#. Seguiremos el mismo sistema que hemos utilizado para generar el otro ejecutable, pero en este caso utilizaremos una rama específica del *GitHub* de *JavaCC*. Usaremos la rama csharp-codegen, que como su nombre indica nos generará el ejecutable que necesitamos para crear nuestro parser en C#.

Una vez tengamos ya los dos ejecutables, utilizando nuestro archivo *AS2JavaParser.jj* (Que recoge toda la información de la gramática de AgentSpeak) se generarán los archivos anteriormente mencionados, a excepción de la clase de constantes que se sustituye por una clase interna en la clase *TokenManager.java*.

5.1.2.6. Fichero AS2JavaParser.jj

Este archivo contiene toda la información necesaria para poder generar el *parser* que reconozca la gramática de *AgentSpeak*, describe si los archivos de entrada que utilicemos son válidos para la esta gramática. Lo primero que nos encontramos en este archivo es una parte de opciones que nos permitirá seleccionar cómo queremos generar los archivos del *parser*. En caso de que no se añade una opción siempre se utilizarán los valores de las distintas opciones por defecto. Utilizamos las siguientes opciones en nuestro fichero:

1. ***FORCE_LA_CHECK***: Controla la comprobación de ambigüedad de búsqueda realizada por *JavaCC*. Al estar a *true* se realizará una comprobación de ambiguedad de lookahead en todos los puntos de elección, independientemente de las especificaciones de lookahead en el archivo de la gramática. La valor por defecto es *false*.

2. ***STATIC***: Al estar en *false*, todos los métodos y variables de clase se especifican como no estáticos en el analizador y el administrador de *tokens* generados. Su valor por defecto es *true*.

3. ***IGNORE_ CASE***: Establecer esta opción a *true* hace que el administrador de *tokens* generado ignore el caso en las especificaciones de los distintos *tokens* y los archivos de entrada.

4. ***UNICODE_ INPUT***: Cuando establecemos esta opción en verdadero, el analizador generado utiliza un objeto de flujo de entrada que lee archivos *Unicode*.

5. ***CODE_ GENERATOR***: Esta opción nos permite seleccionar que generador de código vamos a utilizar. En nuestro caso, ya que queremos que el parser generado esté en C#, escribimos el valor personalizado para dicho lenguaje: "org.javacc.csharp.CodeGenerator".

6. ***NAMESPACE***: La opción nos permite seleccionar el nombre que le vamos a poner al *namespace* donde se encontrará el parser generado, en nuestro caso "Assets.Code.AsSyntax.parser";

Tras haber elegido las opciones que queremos utilizar para generar el parser, nos encontramos los distintos *tokens* usados en la gramática (Figura 5.3).

En nuestro caso nos encontramos con los espacios, saltos de línea, palabras reservadas, números, letras y cual es el contenido de estos *tokens*, por ejemplo un número puede estar formado por una cifra, puede ser un número negativo, estar compuesto por el símbolo de elevado.

Por último, nos encontramos las clases y reglas que debe seguir la gramática y que son usadas para especificar el código que se está ejecutado mientras parseamos. Todo el código siguiente debe estar escrito en C# para que el código generado este en ese mismo lenguaje. En estas reglas y clases utilizaremos los *tokens* que hemos creado en la parte superior de este archivo, así una vez generado el *parser* y por cada vez que entre un fichero con la gramática específica a este parser, podemos leer e interpretar su contenido.

Para poder interpretar cualquier gramática que siga el lenguaje de *AgentSpeak* necesitaremos las siguientes clases y reglas: *Agent, Directive, Belief, Initial_ Goal, Plan, Trigger, Plan_ Body, Plan_ Body_ Term, Plan_ Body_ Factor, Body_ Formula, Rule_ Plan_ Term, Literal, Pred, List_ Terms, Log_ Expr, Rel_ Expr, Arithm_ Expr, Term, Function.*

```
SKIP : {
 " "
|  "\t"
|  "\n"
|  "\r"
|  <"//" (~["\n","\r"])* ("\n" | "\r" | "\r\n")?>
|  <"/*" (~["*"])* "*" ("*" | ~["*","/"] (~["*"])* "*")* "/">
}

// Note: i do not why, but vars must be defined before TK_BEGIN and END
TOKEN : {
  <VAR : (<UP_LETTER> (<CHAR>)*) >
}

TOKEN : {
// Predefined
  <TK_TRUE:   "true">
| <TK_FALSE:  "false">
| <TK_NOT:    "not">
| <TK_NEG:    "~">
| <TK_INTDIV: "div">
| <TK_INTMOD: "mod">
| <TK_BEGIN:  "begin" >
| <TK_END:    "end" >
| <TK_LABEL_AT: "@"> // special chars

| <TK_IF:     "if" >
| <TK_ELSE:   "else" >
| <TK_ELIF:   "elif" >
| <TK_FOR:    "for" >
| <TK_WHILE:  "while" >

| <TK_PAND:  "|&|" >
| <TK_POR:   "|||" >

// Numbers
|  <NUMBER: ["0"-"9"] (["0"-"9"])*
          | (["0"-"9"])* "." (["0"-"9"])+ (["e","E"] (["+","-"])? (["0"-"9"])+)?
          | (["0"-"9"])+ (["e","E"] (["+","-"])? (["0"-"9"])+) >

// Strings
|  <STRING: "\"" ( ~["\"","\\","\n","\r"]
                 | "\\" ( ["n","t","b","r","f","\\","\'","\""]
                        | ["0"-"7"] (["0"-"7"])?
                        | ["0"-"3"] ["0"-"7"] ["0"-"7"]))* "\"">

// Identifiers
|  <ATOM      : (<LC_LETTER> | "." <CHAR>) (<CHAR> | "." <CHAR>)*
              | ("'" (~["'"])* "'" ) >
              { if (image.charAt(0) == '\'') matchedToken.image =
              image.substring(1, lengthOfMatch-1); }
|  <UNNAMEDVARID: ("_" (<DIGIT>)+ (<CHAR>)*) >
|  <UNNAMEDVAR: ("_" (<CHAR>)*) >
|  <CHAR      : (<LETTER> | <DIGIT> | "_")>
|  <LETTER    : ( <LC_LETTER> | <UP_LETTER> )>
|  <LC_LETTER : ["a"-"z"]>
|  <UP_LETTER : ["A"-"Z"]>
|  <DIGIT     : ["0"-"9"]>
}
```

Figura 5.3: Listado de Tokens de la gramática empleados en nuestro proyecto.

5.2. Diseño y arquitectura de la capa BDI

A continuación nos centraremos en la capa intermedia de nuestro sistema: la Capa BDI. **En ésta se ejecutan todas las operaciones necesarias para hacer funcionar** realmente nuestra IA.

5.2.1. Análisis

En esta capa se decidió **implementar toda la parte que hace funcionar el sistema BDI** propiamente dicho. Es decir, en esta capa están **los deseos, intenciones y creencias del agente, el propio agente, el razonador y el paso de mensajes entre agentes**. Además, se implementan también funcionalidades necesarias para el paso de información entre la lógica; es decir, la sintaxis de AgentSpeak y el BDI, y la funcionalidad necesaria para interactuar con el front y con Unity.

Al estar basados en Jason, se adaptó su arquitectura a nuestras necesidades específicas y se llegó a la conclusión de que **se necesitaba lo siguiente**:

1. Agente.
2. Intenciones.
3. Deseos.
4. Base de creencias.
5. Ciclo de razonamiento.
6. Unificador.
7. Entorno.

5.2.1.1. Agente

Un agente es el **actor** en el modelo BDI. Cada agente tiene su propia base de creencias, con sus propias percepciones sobre el entorno, su propia librería de planes, su propio conjunto de intenciones y su ciclo de razonamiento. Cada agente es una entidad independiente que puede comunicarse con las demás a través del paso de mensajes.

5.2.1.2. Intenciones

Representan las **acciones concretas** que el agente va a ejecutar para alcanzar sus deseos siguiendo un plan.

5.2.1.3. Deseos

Representan los **objetivos que el agente quiere alcanzar**.

5.2.1.4. Base de creencias

Almacena las creencias de un agente sobre el entorno en el que se encuentra. Las creencias **no son necesariamente verdaderas**, un agente también puede tener creencias falsas. **Se actualiza en cada iteración** del ciclo de razonamiento para añadir o eliminar creencias y también puede modificarse recibiendo mensajes de otros agentes.

5.2.1.5. Ciclo de razonamiento

Se encarga de ejecutar todo el ciclo de razonamiento del agente. Tiene las siguientes etapas:

1. **Percibir el entorno:** Obtiene las percepciones del agente acerca del entorno que le rodea.
2. **Actualizar la base de creencias:** Añade las percepciones obtenidas en el paso anterior al conjunto de creencias del agente.
3. **Recibir comunicación de otros agentes:** Se comprueba el buzón de mensajes del agente por si han llegado mensajes de otros agentes.
4. **Seleccionar mensajes:** Si se han recibido mensajes, se seleccionan aquellos en los que el destinatario sea el propio agente. Si hay mensajes para el agente, se leen y se actualiza de nuevo la base de creencias, ya que puede haber información nueva sobre el entorno.
5. **Seleccionar un evento:** Se comprueba si hay eventos por procesar. En cada iteración del bucle de razonamiento se procesa un único evento.
6. **Recuperar todos los planes relevantes:** Se busca en la librería de planes si existen planes relevantes para el evento actual. Se recuperan también todos los planes que tengan un *triggering event* que se pueda

unificar con el evento seleccionado. Si no existen planes relevantes, se descarga el evento seleccionado.

7. **Determinar los planes aplicables:** Los planes tienen un contexto que nos indican las condiciones bajo las que dicho plan se puede aplicar. Después de recuperar todos los planes relevantes, se comprueba si alguno de ellos puede aplicarse en ese momento y se recuperan todos.

8. **Seleccionar un plan aplicable:** Solo puede aplicarse un plan para cada evento, por lo que se selecciona uno de ellos de la lista de planes aplicables obtenida en el paso anterior.

9. **Seleccionar una intención para ejecutar:** Una vez seleccionado el plan, se tiene la intención de ejecutarlo. Un agente puede tener varias intenciones simultáneamente, por lo que hay que elegir una de entre todas ellas.

10. **Ejecutar la intención:** Se intenta ejecutar la intención en el entorno.

11. **Preparar la siguiente iteración del bucle del razonador:** El agente comprueba si quedan intenciones pendientes de *feedback* o de mensaje, las actualiza y devuelve al conjunto de intenciones. Elimina las intenciones que se hayan ejecutado hasta el final.

5.2.1.6. Unificador

A pesar de estar muy ligado a la capa lógica, el unificador se encuentra en la capa de BDI, ya que es el **encargado de coger las percepciones, creencias, deseos, intenciones... y sustituir las variables** de tal manera que el resultado sea algo que la sintaxis sea capaz de interpretar.

5.2.1.7. Entorno

El entorno se encarga de implementar la **interacción de los agentes con el mundo en el que se desarrolla la acción**; en nuestro caso, una escena de Unity. Está diseñado e implementado como parte del BDI aunque esté muy relacionado con la capa del Front, ya que el agente percibe los cambios en el entorno y ejecuta el ciclo de razonamiento con las percepciones obtenidas.

5.2.2. Diseño

Para simplificar el diseño y la implementación y paralelizar el desarrollo, se decidió separar en paquetes de código toda la arquitectura de la capa BDI de la siguiente manera:

- **BDIAgent:** Contiene las funcionalidades necesarias para crear el agente y el paso de mensajes entre agentes.
- **BDIManager:** Se divide a su vez en varios paquetes diferentes:
  - **Creencias:** Contiene las funcionalidades necesarias para implementar la base de creencias.
  - **Intenciones:** Contiene las funcionalidades necesarias para implementar las intenciones, las circunstancias bajo las que se puede aplicar un plan, y los eventos que desencadenan dichas intenciones.
  - **Deseos:** Contiene las funcionalidades necesarias para implementar los Deseos.
- **Razonador:** Contiene las funcionalidades necesarias para ejecutar el ciclo de razonamiento, el unificador y la ejecución de acciones.
- **Entorno:** Contiene las funcionalidades necesarias para que el agente interactúe con la escena en Unity.
- **Infra:** Contiene las funcionalidades necesarias para implementar la arquitectura centralizada del sistema multiagente.

5.2.3. Implementación

A continuación se detalla la implementación de las clases y su funcionalidad.

1. **BDIAgent.** Contiene las clases AgentArchitecture, Message y Agent.
   a) **Agent:** Esta clase implementa la funcionalidad básica de un agente dentro de un sistema multiagente. Un agente tiene su propia base de creencias, su propia librería de planes y un razonador asociado. Esta clase nos proporciona la funcionalidad necesaria para crear nuevas instancias de los agentes, añadir, actualizar y eliminar deseos, creencias e intenciones, gestionar el buzón de mensajes de cada agente, encolar tareas en un planificador que ejecutará dichas tareas de forma automática, percibir el entorno, actualizar

la base de conocimiento y seleccionar los eventos que se van a procesar.

b) **AgentArchitecture:** Esta clase proporciona la arquitectura concreta de un agente. Implementa un patrón cadena de responsabilidades, que encadena instancias de la arquitectura del agente de tal manera que delega en las siguientes instancias la tarea de responder a las peticiones y llamadas que recibe. La última instancia es de tipo CentralisedAgArch y se detalla más adelante.

c) **Message:** Proporciona los métodos necesarios para el envío y recepción de mensajes entre agentes.

2. **BDIManager.** Contiene tres paquetes: Beliefs, Desires e Intentions.

   a) **Beliefs:** Contiene todas las clases necesarias para el funcionamiento de las creencias.

      1) **BeliefBase:** Representa e implementa una base de creencias básica. Esta clase permite manipular una base de creencias añadiendo y eliminando literales de la misma, además de consultar la información disponible actualmente.
      2) **StructureWrapperForLiteral:** Utilizado para realizar comparaciones.

   b) **Desires:** Contiene la clase **Desire**, que representa el estado de los deseos de un agente. Genera eventos para notificar si un estado ha empezado, terminado, fallado, o si se ha suspendido o reanudado.

   c) **Intentions:** Contiene todas las clases relevantes al funcionamiento de las intenciones de un agente.

      1) **CircumstanceListener:** Interfaz general sobre la que implementar circunstancias.
      2) **Circumstance:** Controla todos los cambios en eventos, acciones e intenciones que tienen lugar. Contiene listas de eventos e intenciones, tanto pendientes como activas. Permite, además, añadir nuevas intenciones y eventos.
      3) **Event:** Permite modificar eventos que están teniendo lugar en el entorno. Controla los trigger e intenciones.
      4) **IntendedPlans:** Permite modificar y comprobar los planes actuales de un agente. Puede añadir y eliminar planes, modificar los pasos internos de un plan, sus triggers y comprobar si un plan ya se ha ejecutado o aún está en espera.
      5) **Intention:** Implementa una intención, que es esencialmente una pila de IntendedPlans. Permite añadir y quitar planes, así como comprobar el estado actual de la lista de intenciones.
      6) **Option:** Combina un Plan con el Unifier para hacerlo aplicable.

3. **Razonador**. Contiene las clases Reasoner, Unifier y ExecuteAction.
    a) **Unifier:** Contiene los métodos necesarios para unificar las variables.
    b) **Reasoner:** Proporciona las funcionalidades necesarias para llevar a cabo el bucle del ciclo de razonamiento explicado anteriormente.
    c) **ExecuteAction:** Proporciona la funcionalidad necesaria para que el agente ejecute acciones.
4. **Entorno.** Contiene las clases Environment y EnvironmentInfraTier.
    a) **Environment:** Proporciona funcionalidades para la interacción directa entre los agentetodo el sistema BDI y Unity.
    b) **EnvironmentI Proporciona funcionalidad para la interacción entre el entorno y el sistema BDI específico para la arquitectura centralizada que se está siguiendo.nfraTier:**
5. **Infra.** Contiene lMsgListener, CentralisedAgArch, BaseCentralisedMAS.
    a) **MsgListener:** Proporciona una interfaz diseñada para notificar a los agentes si los mensajes se han enviado.
    b) **CentralisedAgArch:** Proporciona la funcionalidad de interacción entre el agente y el entorno de manera específica siguiendo una arquitectura centralizada.
    c) **BaseCentralisedMAS:** Funciona como controlador del BDI de tal manera que proporciona la funcionalidad necesaria para la ejecución del BDI.

5.3. Entorno gráfico y funcionalidades

Por último, la capa superior en contacto directo con el programador es el Entorno Gráfico. Depende completamente de la implementación de las dos capas inferiores, y **facilita su uso para el usuario final**. Aunque no implementa características del sistema multiagente, las acerca al desarrollador al sistema y agiliza el diseño e implementación en su proyecto.

5.3.1. Análisis

Como el propósito de nuestra herramienta es simplificar la implementación del sistema, entendemos que la manera más sencilla de comprender y operar con agentes es a través de una Interfaz Gráfica. Para ello, **usaremos Jason como punto de partida**.

La herramienta jEdit, contenida en el paquete descargable de Jason, ofrece el entorno gráfico básico para el diseño de Sistemas Multiagente:

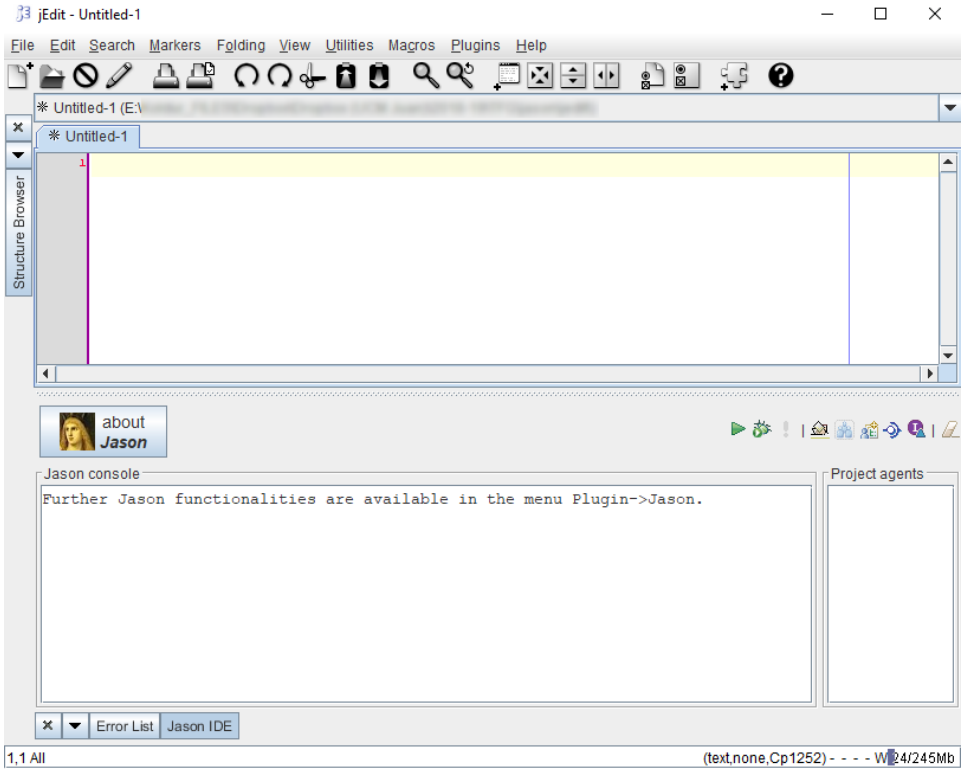

Figura 5.4: Interfaz gráfica de jEdit ejecutado en Windows. Se muestra la zona de edición de ASLs en la mitad superior, la consola de ejecución en la mitad superior y los controles de ejecución en el centro (entre otros elementos).

Empezando por lo más básico, **debe permitirse programar los aspectos más básicos de los Agentes Inteligentes basados en *BDI***: las creencias, deseos e intenciones. Para implementar dichos aspectos, *Jason* permite editar los ficheros fuente de los agentes (con extensión *'.asl'*) a través de su propio editor de texto.

Además, cuando se crea un proyecto nuevo se crea una carpeta para el mismo con **el siguiente archivo de configuración** en su interior:

```
/*
testing
- - - - - - - - - -
Jason Project File
This file defines the initial state of the MAS
Jason
enero 20, 2019 - 20:08:16
*/
MAS testing {
    infrastructure: Centralised
    agents:
}
```

El código anterior almacena el nombre del proyecto ("*testing*"), el tipo de infraestructura (*Centralised/Jade*) y la lista de agentes que contiene.

Un sistema centralizado **está pensado para operar en la misma máquina** y no permite la conexión con agentes no basados en *Jason*, pero un sistema multiagente de tipo *Jade* resuelve estas limitaciones.

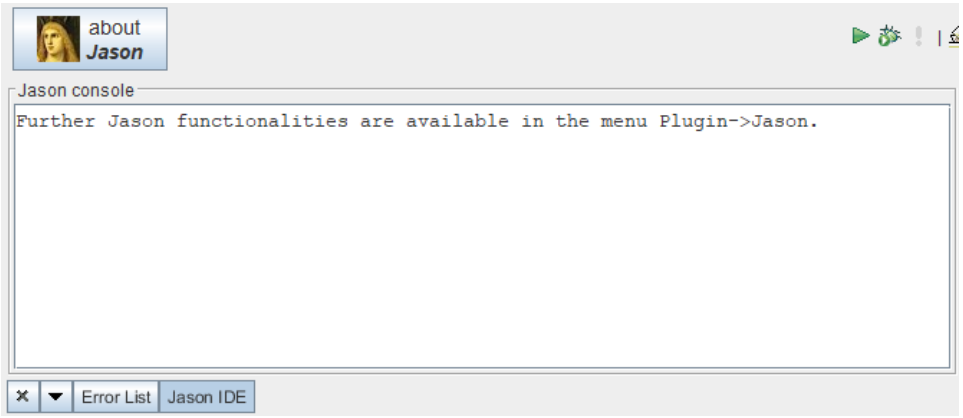

Figura 5.5: Consola de jEdit, a la espera de comenzar la ejecución.

Tras el análisis de la herramienta, concluimos que ya podríamos discernir las características que debemos incorporar a nuestra herramienta.

5.3.2. Diseño

Ya conocidas las funcionalidades más básicas de jEdit, podemos **diseñar una interfaz para nuestra herramienta** siguiendo los principios de Diseño de Sistemas Interactivos (DSI) siguiendo la metodología del Diseño Guiado por Objetivos.

5.3.2.1. DSI: Diseño Guiado por Objetivos

El Diseño Guiado por Objetivos **se centra en la psicología del usuario del sistema**: se deben considerar las posibles interacciones y los fines que puede perseguir el mismo a la hora de emplear la herramienta.

Los objetivos que se pueden identificar se clasifican en tres grados según la respuesta producida:

1. **Visceral**: apela al inconsciente del usuario. No es reflexiva y muy rápida.
2. **Conductual**: aunque sigue perteneciendo al inconsciente, el usuario le dedica un mayor tiempo de respuesta para ajustarse a un modelo mental que el usuario posee previamente.
3. **Reflexiva**: Se basa en la memoria, no en el inconsciente. El usuario realiza un razonamiento consciente a la hora de generar las interacciones.

De los tipos de respuesta se derivan tres tipos de objetivos:

1. **Objetivos de experiencia:** Apelan a lo visceral. Juegan con el subconsciente del usuario para generar sensaciones agradables como el de control, estética, no-frustración...
2. **Objetivos finales:** Siguiendo las respuestas conductuales, están directamente relacionados con la motivación del usuario para usar nuestro sistema (programar un agente, diseñar un icono, dar viveza a un *NPC*...)
3. **Objetivos vitales:** Relacionados con las respuestas reflexivas, apelan a las aspiraciones más profundas del usuario (ser un experto en X, conseguir reconocimiento por completar la tarea Y...)

Formalmente, el procedimiento para definir los objetivos (antes de la implementación) pasa por fases de Investigación, Modelado, Definición de

Requisitos, Definición del sistema y Refinamiento. Sin embargo, en el ámbito de nuestro proyecto las sintetizaremos en las fases de Investigación, Definición de requisitos y del sistema.

5.3.2.2. Diseño de Jasonity

Explicada la fase de Investigación previa, el análisis (mediante el *Análisis de la Competencia*) dio lugar a una definición de requisitos relativamente sencilla:

- El usuario debe poder **programar archivos ASL desde Unity**.
- El usuario debe poder **configurar el conjunto de agentes** desde un panel de Unity.
- El usuario debe poder **insertar modelos de agentes** en la escena vinculándolos a "GameObjects".
- El usuario debe poder **visualizar el proceso de razonamiento** de los agentes existentes en tiempo de ejecución.
- El usuario debe poder **configurar el entorno** de manera que se adapte a sus necesidades.

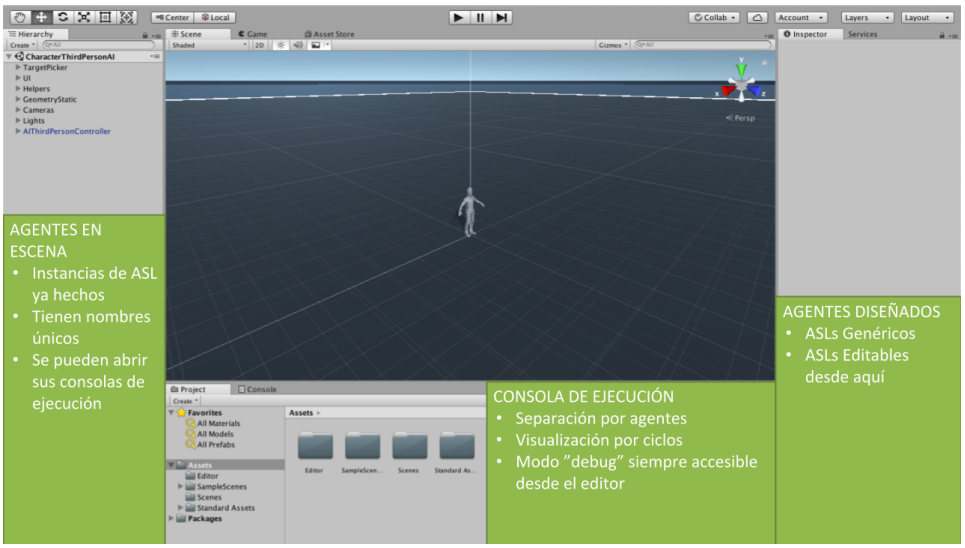

Figura 5.6: *Sketch* de la interfaz de Jasonity. Muestra los tres principales módulos: Instanciado de Agentes (izquierda), consola de ejecución exclusiva del sistema multiagente (centro) y listado de archivos ASL disponibles para el proyecto (derecha).

5.3.3. Implementación

Conocidos los requisitos del diseño, se propone la distribución de una interfaz gráfica interactiva integrada dentro de Unity reflejada en la Figura 5.6.

Sin embargo, la implementación de la herramienta gráfica, dependiente de la completitud del *Back-End*, **queda fuera del alcance de nuestro proyecto**. Es por ello que no disponemos actualmente ninguna herramienta completada, aunque en la rama "Editor" de nuestro respositorio de GitHub[1] están disponibles los primeros *Mockups* de funcionalidades gráficas.

5.4. Problemas encontrados durante la migración

Uno de los mayores obstáculos que han aparecido a lo largo de la implementación del código ha sido que la traducción de código en Java a C# no era directa y en gran cantidad de ocasiones no existía un equivalente a lo que se necesitaba traducir.

Los problemas más destacados y su resolución han sido los siguientes:

1. **Planificador:** En Java se utilizaba una clase llamada ScheduleExecutor que funcionaba como un planificador. En C# no existe ninguna clase con una funcionalidad similar por lo que se optó por diseñar e implementar un planificador. Este planificador consiste en una cola de tareas runnables que se ejecutan de manera secuencial y de una en una cuando el planificador es invocado.

2. **Interfaz IRunnable:** En Java existe una interfaz Runnable que se se utiliza para que las clases que la implementan tengan un método run, que es el que se invoca y ejecuta. Esta interfaz no existe en C#, por lo que se implementó una interfaz propia, IRunnable, que imita el comportamiento de la interfaz Java. Esta interfaz IRunnable se utiliza para crear las tareas que se añaden al planificador.

3. **Enumerados:** En Java los enumerados tienen un método toString que se puede sobreescribir. En C# no existe este método, por lo que la solución ha consistido en implementar una clase que implemente dicho enumerado y, en el método ToString de la clase, implementar el método ToString del enumerado.

4. **Implementación de interfaces anónimas:** En Java existe la implementación de interfaces anónimas, cosa que C# no permite ya que no

[1] https://github.com/Narratech/Jasonity

deja instanciar interfaces. Para solucionar este problema se decidió que lo mejor era realizar clases internas que implementasen las interfaces y sustituir la creación de instancias de las interfaces por esas clases internas implementadas.

5. **Concurrencia:** En Java la concurrencia está diseñada de manera que es muy simple implementarla. Mientras que en C# existe la posibilidad de implementar esa concurrencia, su integración en Unity es más compleja y genera algunos problemas a la hora de competir por recursos del sistema. Por este motivo se decidió que la mejor solución era abandonar la concurrencia en este proyecto, dejándolo como trabajo futuro.

Además de estos problemas destacados, hemos recopilado en una wiki de Github multitud de traducciones no triviales pero que no generaron tantos mismos problemas como los aquí detallados.

Como detallamos en el próximo capítulo, **los problemas encontrados durante la migración llevaron a un replanteamiento del problema y cómo solucionarlo, dando lugar a una nueva etapa de diseño e implementación**.

Capítulo 6

Jasonity: Diseño e Implementación de nuestro Sistema

Tras estudiar y analizar el diseño de Jason en el capítulo anterior, así como de transcribir una versión del código a C#; nos encontramos ante la tesitura de decidir si la arquitectura de dicha herramienta era adecuada para nuestro proyecto.

Para ello **se llevó a cabo el desarrollo y testeo de pruebas unitarias** sobre las clases transcritas a C#, principalmente de las funciones existentes en las clases de AsSyntax (capa Lógica), ya que estas, a través de llamadas al *Parser* (otra pieza clave en el funcionamiento de Jasonity), son las que crean las creencias, deseos e intenciones del agente. Por tanto, el correcto funcionamiento de dichas funciones resultaba clave para la viabilidad del proyecto.

Los resultados de las pruebas dieron lugar a la conclusión de que esta versión de Jasonity, de cara a uno de los objetivos del TFG de continuar actualizando esta herramienta, **era demasiado compleja de mantener e integrar en Unity**. Cualquier modificación realizada a futuros sobre ella, requeriría invertir de nuevo mucho tiempo en el aprendizaje de Jason, así como en la depuración, fruto del ingente tamaño del *Parser* original.

Por tanto, **se decidió implementar una versión propia de Jason, más pequeña y mantenible, como base** sobre la cual construir y extender nuevas funcionalidades en el futuro de forma más sencilla e intuitiva que en la versión anterior.

De éste modo, el sistema pasó de estar definido por un diagrama de clases totalmente inabarcable (más de doscientos cincuenta clases que se han

intentado insertar en ésta memoria, pero resulta ilegible) a disponer de una **arquitectura optimizada**: menos clases, más funcionales y cumpliendo su funcionalidad sin recargar los paquetes.

6.1. Diseño: La arquitectura de Jasonity 2.0

Aunque llegados a éste punto parezca que todo el esfuerzo dedicado a la migración de Jason ha sido en balde, dicha afirmación no resultaría del todo correcta. Es precisamente **la cantidad de horas dedicada a trabajar con el sistema**, leyendo el código original, entendiendo su uso y traduciéndolo al lenguaje C# **es la que nos ha dado el conocimiento** necesario para realizar una síntesis correcta, sin pasar por alto detalles esenciales para su funcionamiento.

Se ha optado por una arquitectura optimizada, teniendo en cuenta las particularidades de Unity, permitiendo una escalabilidad y facilidad de mantenimiento que con la versión anterior hubieran resultado altamente difíciles.

Así, pasamos de más **de 250 clases y 24.000 líneas a 11 clases y aproximadamente 560 líneas** de código distribuidas de la forma mostrada en la Figura 6.1

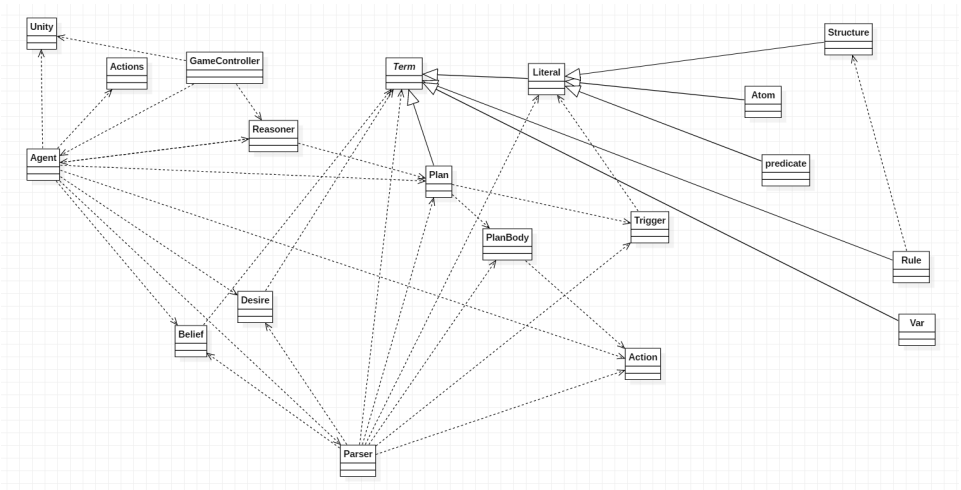

Figura 6.1: Diagrama de clases de la arquitectura propuesta para Jasonity 2.0, donde la clase Unity hace referencia al sistema conjunto de Unity

Como se aprecia a simple vista, hay **3 módulos principales que componen Jasonity**: El *parser*, el agente y el actuador. A continuación se describirán dichos módulos a nivel diseño e implementación.

6.2. El nuevo Parser

El diseño de la arquitectura del nuevo **Parser**, inspirado en el original de **Jason**, se compone de la siguiente manera:

- **Term**: Clase padre (**abstracta**) de la que heredan, directa o indirectamente, todas las demás clases del nuevo parseador. Contiene funciones "virtuales" booleanas que identifican el tipo de término que es cada clase del *Parser*, las cuales serán reimplementadas posteriormente por las clases herederas de esta.

- **Literal**: Clase de nexo entre **Term** y todas clases del parseador, a excepción de **Rule**, **Plan** y **Variable**. Contiene el constructor base al que llamarán las demás clases desde sus propios constructores, con el objetivo de guardar el nombre del término que contienen y establecer, en forma de atributo booleano, si dicho término es verdadero o falso para ellos.

- **Atom**: Clase que representa a las constantes del lenguaje. Contiene al constructor que invocará al de **Literal** y le pasará por parámetro el nombre de la constante y el valor booleano que indica si es cierta o no.

- **Structure**: Clase fruto de la combinación de un **Atom** y una serie de argumentos, que el *string* que representa la **Structure** contiene entre paréntesis, los cuales pueden ser a su vez, otros literales. Implementa el constructor que llamará al de **Literal** y también una lista para guardar en ella los posibles parámetros que tuviera el *string*.

- **Predicate**: Clase nacida de la conjunción entre una **Structure** y un nuevo conjunto de parámetros (denominadas **anotaciones**), que el *string* que representa el **Predicate** contiene entre corchetes, los cuales pueden ser, de nuevo, otros literales. Implementa el constructor que llamará al de **Literal** y también una lista para guardar en ella las posibles **anotaciones** que tuviera el *string*.

- **Rule**: Clase compuesta de un primer **Literal**, denominado **functor**, el cual es la consecuencia que se aplicará si se cumplen las condiciones que conforman el resto de literales de la regla, quienes se encuentran situados al otro de este caracter: `":-"`, el cual hace las veces de separador entre la primera parte de la regla y la segunda.

  Contiene un constructor el cual recibe una estructura, (**clase Dictionary de C#**), con todos los elementos de la regla, los cuales clasifica entre la consecuencia (primera entrada de la estructura) y condición (el resto).

- **Variable**: Clase formada por un único *string* el cual cumple siempre la condición de empezar por un caracter mayúsculo. Son los objetos que pueden alterar su valor a lo largo del programa. Contienen un constructor para establecer el nombre de la variable.
- **Action**: Clase formada por un *string* a modo de nombre de la acción interna del agente y una lista, también de *string*, como los argumentos que debe recibir dicha acción para funcionar. Las acciones internas son funciones invariables, propias del agente, que este invoca para realizar tareas en base a los datos de su base de conocimientos.
- **Trigger**: Es la clase que define el disparador del **Plan**. Compuesta por uno de estos dos caracteres, llamados **operadores**, (+) o (-), los cuales indican si el disparador del plan debe activarse cuando el **Literal** que da valor a dicho disparador, se añade o se borra de la base de datos del agente.
- **PlanBody**: Clase que contiene un lista de acciones (**Actions**) que conforman el cuerpo del **Plan**. Contiene un constructor que recibe la lista de acciones por parámetro y la guarda en su propio atributo.
- **Plan**: Clase formada en base a las clases **Trigger** y **PlanBody**, y una serie de **Literals** que recibe como una estructura, (**clase Dictionary de C#**), a través de los parámetros del constructor. Dichos literales conforman lo que se conoce como el **contexto** del plan, una serie de condiciones unidas mediante operadores lógicos que, de cumplirse, permiten ejecutar el cuerpo del plan.

  Contiene un constructor que guarda los parámetros descritos anteriormente en sus propios atributos.
- **Parser**: La clase que se encarga de leer la información contenida en el fichero de texto y transformarla en Intenciones, Creencias o Deseos; conformados a partir de las clases descritas en los apartados anteriores.

  Los planes están siempre precedidos por un **operador**, los deseos por los **símbolos**: "!" o "?"; y las creencias no están precedidas por nada.

  Contiene diversos métodos encargados de diseccionar los distintos *strings* contenidos en el fichero y de instanciar las clases correspondientes para moldearlos en objetos entendibles para el agente.
- **Belief**: Clase que guarda cada objeto que ha sido catalogado como creencia por el parser.
- **Desire**: Clase que guarda cada objeto que ha sido catalogado como deseo por el parser.
- **Intention**: Clase que guarda cada objeto que ha sido catalogado como intención por el parser.

6.2.1. Versión para la demo

Para esta primera demostración visual, **nos centramos en la parte del *Parser* encargada de traducir y generar los planes (Plan) a partir del fichero de texto**. Ya que, aunque se le restase alcance de esta forma, permitiría demostrar el funcionamiento de un agente con el ejemplo más complejo de razonamiento para él y extender esto a una versión más completa de la herramienta resultaría más sencillo que el ejemplo contrario.

De modo que se implementó una versión reducida del **Parser** con solo el propio parseador y las clases **Plan**, **PlanBody**, **Action**, **Trigger** y **Belief**, a las cuales hubo que modificar el tipo de los parámetros recibidos con respecto a los descritos en la sección anterior, ya que clases como **Literal** o **Structure** no se implementaron para esta versión.

La arquitectura del *Parser* para la demo sería la siguiente:

- **Plan**: Igual a la clase descrita en el diseño salvo por el contexto, el cual ha sido modificado para que consista en un *string* que indica si el contexto se cumple o no.
- **PlanBody**: En este caso tampoco difiere de la descripción del diseño, en ningún aspecto.
- **Trigger**: Para esta clase se modificó la forma de recibir el valor del disparador, en vez de un **Literal**, se recibe el nombre del *trigger* como un *string* por parámetro. De igual modo, se reciben los posibles argumentos como una lista de *strings* por parámetro, los cuales ya irían incluidos dentro del literal, si este fuera una **Structure** o un **Predicate**, en la versión del diseño.
- **Belief**: No se distingue en nada de la versión explicada en el diseño.

6.3. BDI: Agente y Controlador sobre Unity

Nuestra arquitectura BDI es original, aunque se basa en las clases utilizadas por el **Jason** original. Las clases que componen BDI son las siguientes:

- **Agent**: Clase principal que representa a los **agentes** utilizados por el sistema. Cada agente tiene su propia base de creencias, librería de planes y lista de deseos, además de un identificador y su propio razonador.

  Contiene métodos que informan sobre el **estado actual** del agente y permiten modificar sus valores internos a través del *parser* o de sus

percepciones. Así mismo, contiene métodos que le permiten **percibir el entorno** para obtener conocimiento y **actuar sobre el mismo** para llevar a cabo sus planes.

- **Belief**: Clase que representa las **creencias** de un agente. Cada creencia se identifica por un nombre y contiene un valor, ambos en forma de *string*. Contiene métodos para recibir nuevas creencias y actualizar y consultar la información correspondiente.
- **Desire**: Clase que representa los **deseos** de un agente. Permite modificar y consultar su valor interno, en forma de *string*.
- **Reasoner**: Clase que se encarga del proceso de **razonamiento** del agente. Indica al agente qué hacer en cada momento. Incluye métodos para percibir el entorno, actualizar las creencias, seleccionar un plan y actuar sobre el plan seleccionado.

6.4. *"I can't sleep"*: la demo de Jasonity

Como parte esencial del trabajo y sirviendo como **prueba y conclusión del desarrollo** de la nueva plataforma, hemos implementado una demostración visual donde se puede apreciar a los agentes razonando y actuando en su entorno.

Figura 6.2: Imagen de desarrollo de "I can't sleep". Muestra la lámpara en medio (encendida) y las camas donde, en tiempo de ejecución, aparecerán los hermanos mostrando en forma de diálogo su ciclo de razonamiento.

En "I can't sleep" dos hermanos comparten habitación por primera vez. ¿El problema? Uno de ellos sólo puede dormir en completa oscuridad, y

el otro necesita dejar la lámpara encendida para conciliar el sueño. Así, cada uno intentará que el dormitorio esté en las condiciones perfectas para descansar.

Como se puede ver en la figura 6.2, la escena es bastante sencilla, pero **permite ver de forma clara y directa la ejecución de cada agente**, y cómo sus deseos influyen en la escena directamente.

Traducido para nuestro sistema, **existe un entorno** (la escena del dormitorio) que contiene a los hermanos, dos camas y la lámpara. **Los hermanos están construidos como agentes BDI**, que quedan descritos mediante sus respectivos ficheros .asl, y **cuyo ciclo de razonamiento** pasará por percibir el entorno (donde comprobarán el estado de la lámpara), si la lámpara no se encuentra en el estado óptimo querrán cambiarlo y actuarán sobre la lámpara para cambiar su estado para ajustarse a sus deseos.

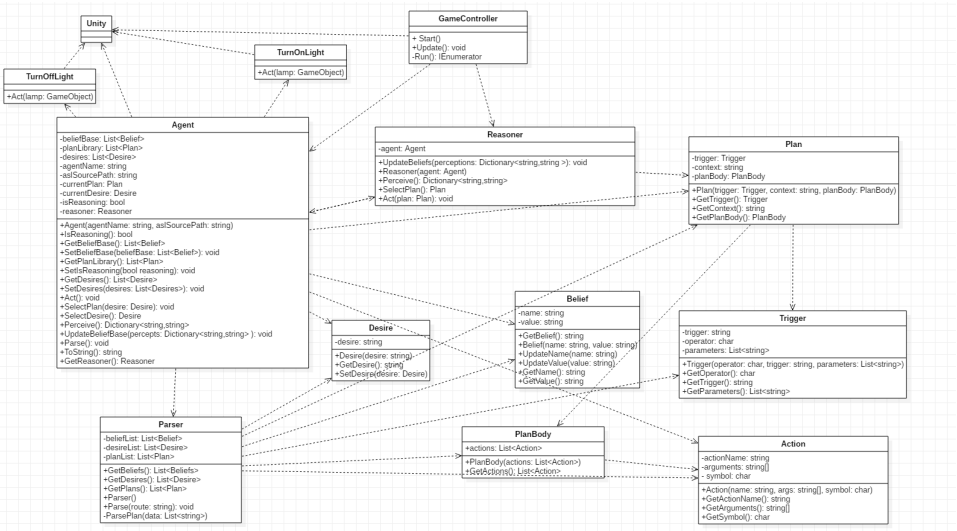

Figura 6.3: Diagrama de clases presente en la demo de "I can't sleep". Muestra la implementación concreta del módulo que actúa directamente sobre Unity.

En el diagrama mostrado en la figura 6.3 **se puede ver de un sólo vistazo los componentes técnicos** que conforman nuestra demo. Con apenas 13 clases óptimamente implementadas, se pueden crear múltiples agentes completos capaces de alcanzar sus objetivos y adaptarse a los cambios del entorno.

Así, **queda demostrado visualmente que Jasonity, nuesto sistema multiagente para Unity, cumple con sus funciones y permite programar NPCs basados en BDI de forma sencilla**.

Capítulo 7

Conclusiones

Tras una investigación inicial que conforma el Capítulo 2 y profundiza en la IA en videojuegos y las herramientas disponibles actualmente, en este trabajo **nos propusimos facilitar la programación de una IA creíble y con mayor apariencia humana en videojuegos**, siendo nuestro trabajo de desarrollo encaminado a dicho propósito.

Para ello, dada la robustez de Jason, trabajamos en la migración de esta plataforma Java a un entorno de desarrollo de videojuegos de última generación como es el caso de Unity/C#.

Una buena base de razonamiento permite el desarrollo de una herramienta completa a través de la cual alcanzar **dos objetivo**s: en primer lugar, que **un usuario no especializado pueda diseñar** sus propios agentes BDI; y en segundo, que **un usuario especializado reduzca el tiempo** de implementación de sus agentes significativamente.

Así es cómo ha comenzado el proyecto **Jasonity: una plataforma diseñada para desarrollar sistemas multiagente con arquitectura BDI para Unity**.

En éste documento se refleja detalladamente el proceso de investigación previo, así como los objetivos y la especificación que marcamos al comienzo del proyecto. Teniendo clara la metodología seguida, y las herramientas empleadas para asegurar su cumplimiento, hemos expuesto nuestro análisis, diseño e implementación de todos los aspectos de *Jasonity*.

A continuación, y por último, **nos disponemos a exponer cuáles han sido los resultados** obtenidos y **qué trabajo queda pendiente** para futuros evolutivos de la herramienta.

7.1. Resultados

Tal como se estableció en el capítulo 3, nuestros objetivos eran relativos a la Investigación, al Análisis, Migración del motor y la Creación de un entorno gráfico. Junto con la información disponible en los capítulos 5 y 6 obtenemos el siguiente estado final:

- **Investigar el estado de la técnica:** Es el objetivo, junto con el *testeo* de herramientas que más tiempo de proyecto ha consumido. Se ha realizado una exhaustiva investigación sobre la IA aplicada a la industria de los videojuegos, así como de la programación declarativa, el modelo de agentes y, en última instancia, de BDI y Jason. **El resultado del objetivo queda reflejado en el capítulo 2**, donde exponemos toda la información encontrada que nos ha servido para encauzar el desarrollo del proyecto.

- **Realizar pruebas conceptuales sobre otras herramientas**: Es posiblemente el **objetivo mejor cumplido**. Poseemos ahora una base de conocimiento amplia sobre no sólo el problema que queríamos abordar, sino de las soluciones probadas por otros investigadores/desarrolladores en la misma materia o semejante.

  Podemos afirmar que en contadas ocasiones se ha intentado permitir la programación con un lenguaje interpretado en Unity. Hemos realizado pruebas sobre dichas herramientas, intentando encontrar aquella que satisficiera el problema planteado, y ha resultado un verdadero quebradero de cabeza. Dichas herramientas (como UnityProlog) poseen una documentación escasa, errática o nula, y el coste de aprendizaje se vuelve inasumible.

  Así pues, **no hemos encontrado pruebas que parezcan indicar que se ha logrado subsanar el problema usando código abierto o software propietario**. Si bien se han realizado aproximaciones muy válidas, éstas suponen un coste de aprendizaje y uso demasiado alto. Por ello, la oportunidad y necesidad de llevar a cabo ese desarrollo queda constatado.

- **Estudio de Jason**: Tras una exhaustiva investigación, estábamos **suficientemente capacitados como para realizar un análisis** profundo de la mejor aproximación encontrada para la programación de un MAS en Java: *Jason*. Si bien es cierto que la mayor cantidad de conocimiento obtenido en la materia se deriva del cumplimiento del siguiente objetivo (reingeniería), hemos profundizado en el funcionamiento de Jason hasta sus capas más profundas, y obtenido una **visión**

de sus fortalezas y debilidades que podemos aprovechar en nuestro proyecto.

- **Reingeniería de Jason:** Se ha seguido un proceso de reingeniería para **obtener un análisis detallado del funcionamiento interno** del programa. Se han elaborado diagramas de clases en UML para representar la relación entre las diferentes clases y paquetes que conforman la estructura interna de *Jason*.

  A lo largo del **capítulo 3 y 5 se detalla el diseño** que consideramos lo suficientemente completo como para realizar una implementación correcta del sistema basándose en el mismo.

  Aunque se buscó la forma de realizar una **traducción automática de Java a C#, no encontramos ninguna herramienta** que realizara dicha función de forma íntegra sin generar problemas inabarcables. Es por ello que la traducción se ha tenido que llevar a cabo de forma manual, lo cual ha traído bastantes problemas.

  Hemos encontrado **bastantes dificultades a la hora de traducir elementos de Java** que no poseían una traducción directa a C#. Tanto es así, que hemos elaborado nuestra propia guía disponible en nuestro repositorio de GitHub[1].

  Aun así, tras más de 24.000 líneas de código traducidas; constatamos que se trata de una labor muy pesada, que puede volverse tediosa y cuyos cambios pueden ser poco intuitivos a veces.

  Tanto es así, que tras explorar las entrañas de Jason **se optó por descartar el proceso de migración literal** y desarrollar un sistema de cero (aprovechando todo el conocimiento obtenido de dicho proceso) con las funcionalidades básicas, con un código más limpio y optimizado que el que tomamos de referencia.

- **Implementación efectiva del sistema:** Como se adelanta en el capítulo 5 y se detalla en el capítulo 6, **se decidió optimizar el producto y el proceso de desarrollo realizando una programación limpia** del sistema multiagente en C# en base al conocimiento obtenido, sin realizar una migración de un sistema conocido. Esto nos ha permitido reducir drásticamente no sólo el número de clases y líneas de código, sino la complejidad del sistema, la comprensión del diseño del mismo y permite una mayor escalabilidad; facilitando el trabajo futuro por nuestra parte o por parte de terceros. Como se refleja en nuestra demo, ya que se puede observar el sistema funcionando sin problemas, **se considera el objetivo completamente cumplido** y demostrado con el próximo.

[1] https://github.com/Narratech/Jasonity

- **Implementación de una demostración:** Para probar el funcionamiento real de nuestra arquitectura, se ha construido una **sencilla pero potente demo que muestra el razonamiento** presente en nuestro sistema multiagente, el efecto sobre un entorno y en definitiva el funcionamiento de un MAS básico a través de Jasonity. Los detalles se desarrollan al final del capítulo 6 y **constatan que el objetivo ha sido cumplido**.

Así pues, la tapa de **diseño y análisis la consideramos satisfactoriamente completada**, y **la implementación (aunque problemática) ha derivado en un producto base con bastante potencial**. Aunque la implementación de la interfaz de Jasonity ha quedado fuera del alcance del proyecto, creemos haber sentado una más que buena base para el motor de razonamiento; y enriquecida con trabajo futuro puede sentar un referente en la programación de Agentes BDI.

7.2. Trabajo futuro

Como última sección de ésta memoria, sintetizaremos los flecos que han quedado pendientes por el tiempo de proyecto; así como posibles ideas que ni siquiera llegaron a asumirse en las etapas más tempranas pero que pueden suponer un valor añadido interesante de cara a futuro.

- **Refinamiento y completitud de funcionalidades**: De nuevo, como el código que había que traducir era tan extenso, hay funcionalidades de las que se ha prescindido o se han sintetizado en una versión escueta de las mismas.

  Por esa misma razón, para darle continuidad al proyecto, la primera tarea a asumir sería la escalabilidad del sistema para permitir numerosos agentes de forma concurrente (multihilo) sobre un entorno más complejo, donde la cantidad de eventos generados sea mayor. Además, la comunicación entre agentes ha quedado fuera del alcance de nuestro proyecto.

- **Creación de un entorno gráfico en Unity para facilitar la programación.** Aunque se haya realizado la especificación de la herramienta gráfica siguiendo los principios del Diseño de Sistemas Interactivos, los problemas encontrados con la traducción del *Back-End* han paralizado la implementación final de la herramienta.

  Al ser un componente completamente dependiente de un sistema de razonamiento terminado, no se puede asumir su implementación hasta que la tarea anterior quede completamente pulida.

De hecho, antes de asumir la tarea de la implementación debe realizarse un análisis del estado final del *Back-End* para definir detalladamente cómo crear la capa intermedia entre la interfaz y el motor de razonamiento.

- **Empaquetado, documentación y subida a la "Asset Store" de Unity.** Si se culmina el trabajo de *Back-End* y *Front-End*, el siguiente paso lógido es la distribución.

  Para ello, Unity ofrece una plataforma virtual de distribución de complementos para el motor, tales como recursos gráficos, herramientas... Así pues, una vez cerrado nuestro sistema, ésta es la vía más lógica de difusión de la herramienta, empleando nuestro repositorio para fomentar la evolución colaborativa del sistema y las contribuciones de los usuarios finales.

Y con éstas conclusiones y el trabajo futuro planteado definido damos por finalizada esta primera etapa del proyecto *Jasonity*, para el que auguramos un gran futuro y aplicabilidad a medio plazo en la industria del videojuego.

Apéndice A

Introduction

"The cake is a lie"
— *Portal*. Valve, 2007

Is there such a thing as artificial intelligence in video games? Or is the usage of other techniques that grant fake intelligence to entities more standardized?

If we dive into a virtual world for the first time and **we don't pay much attention to the details, we may end up thinking everything around us is "alive"**. Even if that **feeling of life and intelligence** can be perceived in most recent video games with *"NPCs"*, odds are they are, deep down, actually less more than objects with a predefined script.

On the other hand, as time goes by, **players become more demanding** with the products they consume, and details have become a key element that could doom a title or make it very successful. This is why studios are working hard to offer more **complete and immersive experiences** every time – and that includes generating a word as alive as possible.

As an example, *The Last Guardian*, genDESIGN; SCE Japan Studio (2016) was released after almost ten years of development. From the very first moment after its long-awaited release, critics, players and developers themselves agreed on one thing: The *AI* behind *Trico*, the mythical creature that travels alongside the protagonist in his adventure, **makes the player feel like they are actually interacting with a living, breathing entity.**

On the other hand, titles like ***Bethesda's* have never been known for having particularly intelligent *NPCs***. While it is true that, in their most recent title, *Fallout 76* (2018), there are no "human" *AIs*, many players

blamed the developers on the erratic behavior of the creatures within the world. If we take a look at one of their most iconic titles, *The Elder Scrolls IV: Oblivion* (2006), the characters inhabiting the huge world map are noticeable because of their hugely lacking AI, which makes the player have to go through very surreal situations. Despite the 12 years separating both video games, **it seems like nothing has improved much**.

And this is because developing a realistic *AI* isn't a simple task. Nowadays, it's still a subject of **research, study and debate**. Implementing believable, human-like behavior can be very expensive: it usually involves **many, many work hours** that will not always be rewarded, because there are other aspects in a game (gameplay, narration, graphics...) that are more attractive for the buyer and are, in a way, easier to develop.

In our case, we believe **good narration and artificial intelligence are two elements that should go hand in hand**. Realistic, intelligent characters are a very important element in developing a story filled with details that can hook the player from the beginning. To do this, we must avoid predefined scripts and **let the game evolve and adapt to the player**, allowing the latter to be the one to write his own story and letting the characters within the world do their thing, so they can respond "by themselves" to said choices, freely interacting with the environment. This is what embracing the concept of Emerging Narrative means and this is why we chose to embark on this project.

To allow that flexibility and autonomy in the characters, we are basing our proposal in a way to build autonomous agents known as **BDI** Architecture (*Belief-Desire-Intention*). This allows us to define each entity as a **set of Beliefs** (information, real or not, perceived directly or indirectly, that the agent has about the environment they live in), **Desires** (goals they wish to see fulfilled) and **Intentions** (mostly plans, sequences of action they must follow to reach their desires while considering their beliefs about the environment).

Through this model, each **BDI Agent is capable of perceiving their own environment, generating real-time plans and adapting their most immediate goals** based on the information they have to reach their main goals. Following this paradigm, each NPC would be designed as a **truly alive** character with their own thoughts, and the *technical designer* would be granted with tools to achieve this, always letting said character's "free will" choose the exact way to achieve their goals; we can forget (like we said in the beginning) about programming every detail, meticulously scripting their behavior and generating a **completely unmanageable state machine or behavior tree** (as it sometimes happens), or even heavy artifacts that ultimately are unable to predict all possible situations that will happen in a

real game.

According to what we studied as part of this project, **there is no standard tool** to work with this kind of character AI, which is also at least available in the main video game development environments. This is why the **purpose of our project** is to contribute to the development of said tool.

A.1. Purpose of the project

We start from the previously described premises to develop the following **Final Degree Project**, which is intended to **offer a solution** to the lack of platforms to **develop BDI-focused multiagent systems** for video games.

To do this, we did some research about the **current state of AI programming paradigms** available nowadays, their advantages and disadvantages, possible alternative implementations and, finally, the **development of the tool** itself. This way, all the **knowledge acquired** throughout the degree, some learned independently and others acquired through this very project, are **shown and synthesized** in the project.

As will be further explained afterwards, the project is split into three stages:

1. **Research**: We have performed a deep analysis of the **techniques and tools** available nowadays for programming BDI agents, choosing the best options to achieve our goal, and at the same time studying if it is viable to begin with. We used other papers and previous projects as a starting point, as well as open source tools that could, at least in theory, complete the desired goal. After this analysis, we moved on to a development stage to obtain a solution that would satisfy the required features.

2. **Development**: After screening our options, we decided to **port most of the features of the Jason system** to C#and the popular video game environment, *Unity*. This involved heavy re-engineering work, translating more than 24,000 lines of code and 250 non-anonymous classes we could find in the original platform. Given the subtle differences between both languages, the sometimes dramatic difference between environments and mostly the lack of documentation about the whole system, we also had to **create a Java to C# and Unity project migration guide**. Given the reasons explained in chapter 5, we had to rethink the development process, starting a new stage of "development 2.0".

3. **Testing**: To gradually test the features of the newly implemented platform, we created automatic tests that check different operative aspects of the resulting system and **verify the proposed goals have been achieved**. We have also implemented a simple demo to visually showcase the application of our multiagent system in a Unity scenario.

A.2. Project Structure

Everything explained up until now and the workflow followed is explained more in depth throughout the following chapters, including the conclusions and detailing the work done by each member of the team in this common project.

After this introductory chapter, **Chapter 2 explains the whole research work** done to determine the capabilities current character AI-development tools have, particularly those who follow a BDI architecture. It also offers an explanation on what we understand for reasoning and agent, what the BDI Architecture is and how it works, it details the platform we are basing our solution off of and showcases AI examples in already released video games, along with a brief explanation of the most popular development engines.

In Chapter 3, after learning how things are nowadays, we detail the **goals and range of our project**, alongside a **specification** of our tool and its requirements.

Throughout Chapter 4 we'll show **how** we're going to reach the goals and fulfill the requirements through our **methodology** and the **tools** we chose for this task.

The summary of our platform, which we have named ***Jasonity*** is right in the middle, in **Chapter 5**. In said chapter we explain the design of its software architecture (both its *back* and its *front*), as well as technical documentation regarding the **analysis, design and implementation** of the system.

In **Chapter 6** we explain why we had to redesign our tool, and the implementation work of that workaround until we achieved a closed system.

Finally, **Chapter 7** collects all the relevant information about the project to present our **conclusions and future work** that we suggest could be done to keep working on it.

Apéndice B

Conclusions

After an initial research, which shapes Chapter 2 and goes deeper into AI in video games and currently available tools, we decided to facilitate realistic, more human-like AI programming, and our development work has been building towards that goal.

To do so, given *Jason's* sturdiness, we worked on porting it from Java to a last generation video game engine, and Unity/C# was our choice. A good reasoning base allows development of a powerful tool with which to achieve two goals: first of all, making it so a non-specialized user can design their own BDI agents; and secondly, significantly reducing development time for a specialized user.

This is how ***Jasonity*** is born: a platform designed to implement BDI-based Multi-Agent Systems in Unity.

This document shows in detail the previous research process, as well as the goals and specification we set at the beginning of the project. Having a clear understanding of the methodology we followed, and the tools used to make sure it's fulfilled, we laid out our analysis, design and implementation of every aspect of *Jasonity*, followed by the tests performed to verify its usefulness.

Next, we are going to explain the obtained results and how much work is left for future instances of the tool.

B.1. Results

As was established in Chapter 3, our goals were relative to Research, Analysis, Migration of the engine and Creation of a graphic interface. Along with the information available in Chapters 5 and 6, this is our final state:

- **Researching the state of the art**: The goal, alongside the testing of tools, that took the most time in our project. We performed a thorough research on AI applied to the video game industry, as well as declarative programming, agent models and, finally, BDI and Jason. The results of this goal are present in Chapter 2, where we showcase all the information found that helped us set a path for the development of this project.

- **Perform concept tests on other tools**: Possibly our best-achieved goal. We now have a deep base of knowledge not only about the problem we wanted to solve, but the solutions other researchers/developers tried in the same or similar subjects.

  We can state there have been very few attempts of allowing programming with a *Unity*-interpreted language. We tested those tools, tried to find one that satisfied the problem, and it was very troublesome. Said tools (like UnityProlog) are very scarcely documented, or is confusing or there is no documentation at all, and the learning process becomes too big.

  So, we didn't find evidence that points to the problem being solved through open source or proprietary software. There have been some very valid aproximations, but the learning curve is too steep. So, the opportunity and necessity of this development is verified.

- **Studying Jason:** After thorough research, we were qualified enough to perform a deep analysis of the best approach we could find for MAS programming in Java: *Jason*. While it is true most of the knowledge obtained on the subject comes from fulfilling the next goal (re-engineering), we went through Jason's features down to the deepest layer, and obtained the whole scope of its strengths and weaknesses we could use in our project.

- **Re-engineering Jason**: We followed a process of re-engineering to obtain a detailed analysis of the inner workings of the program. We developed UML class diagrams to represent the links between all classes and packages that shape the inner structure of *Jason*.

  Throughout Chapters 3 and 5 we detail the design we consider complete enough to create a correct implementation of the system based on it.

Although we looked for a way to automatically translate from Java to C#, we couldn't find any tool that performed such task successfully without generating unmanageable issues. That is why translation has been done manually, and that brought a lot of problems.

We found it was quite difficult to translate Java elements that didn't have a direct translation to C#. So much so that we developed our own translation guide available in our GitHub repository.

Still, after translating more than 24,000 lines of code, we can state it's a very tedious, mechanic and not very intuitive task.

So much so that, after exploring the guts of Jason, we decided to discard the process of literal migration and develop a system from scratch (using all the knowledge obtained through that process) with basic functionality, cleaner code and optimizing the one we used as reference.

- **Effective implementation of the system**: As explained in Chapter 5 and detailed in Chapter 6, we decided to optimize the product and the development process doing a clean implementation of the multi-agent system in C# based on the obtained knowledge, without migrating from a known system. This allowed us to drastically reduce not just the number of classes and lines of codes, but the complexity of the system, the understanding of its design and it allows bigger scalability; it makes future work easier for us or anyone else.

  So, we consider this goal fully achieved and evidenced by the next one.

- **Implementation of a demo**: To test our architecture is truly functional, we built a simple yet powerful demo that shows the reasoning in our multi-agent system, its effect on an environment and the features of a simple MAS through Jasonity. Details are further explained at the end of Chapter 6 and prove the goal has been achieved.

So, design and analysis has been successfully completed, and implementation (although problematic) has ended up with a base product with enough potential. Although implementation of the *Jasonity* interface has been left out of the project's reach, we think we have set a more than good enough base for the reasoning engine; fine-tuned with future work, it could be huge in BDI Agent programming.

B.2. Future Work

As the last section of this project, we synthesize the loose ends that couldn't be tied due to the project's deadline; as well as possible ideas that

we didn't even consider on the earliest stages but could be very interesting in the future.

- **Polishing and finishing of features:** Again, due to the monumental size of the code we had to translate, some features were removed or synthesized into a lesser version of themselves.

  For that same reason, if someone kept working on the project, the first task to tackle would be finishing an accurate and complete translation of *Jason*.

- **Creation of a graphic interface in Unity to facilitate programming.** Although we did the specification of the graphic tool following the Interactive System Design principles, the issues found in the translation of the *back-end* have paralyzed the final implementation of the tool.

  Being a component that fully depends on a finished reasoning system, we can't begin implementation until the former task has been completely finished.

  In fact, before implementation, an analysis of the final state of the *back-end* must be performed to define in detail how to create the intermediate layer between the interface and the reasoning engine.

- **Packaging, documentation and upload to the "Asset Store" in Unity.** If the *back-end* and *front-end* work is done, the next logical step is distribution.

  For that, Unity offers a virtual distribution platform with plugins for the engine, such as graphic resources, tools... So, once our system is finished, this is the most logical way of broadcasting it, using our repository to encourage the collaborative evolution of the system and the final contributions of final users.

With these conclusions and future work defined, we conclude this first stage of the *Jasonity* project, which we think has a very promising future and could be used in the video game industry not so long into the future.

Apéndice C

Instalación de los entornos de programación y ejecución

C.1. Ejecución de la demo

Como aviso previo, indicamos que las instrucciones mostradas a continuación son aplicables en Windows 10 x64. En otros sistemas operativos o arquitecturas no se asegura el correcto funcionamiento.

En la raíz de nuestro repositorio de GitHub existe una carpeta llamada "Jasonity Demo". En su interior se encuentra el archivo "demo.exe". Haga doble clic sobre el icono para ejecutar la demo de Jasonity.

C.2. Instalación de los entornos de programación

En este apartado vamos a explicar como instalar los entornos de programación necesarios para ejecutar la demo final del proyecto.

C.2.1. Instalación de Unity

Primero debemos acceder a la página web oficial de Unity[1]. Una vez en ella seleccionamos el paquete "Personal", una vez aceptados los términos de uso y servicio, pulsamos en **Descargar** y esto iniciará la descarga de "Unity Hub".

[1] https://store.unity.com/es

Una vez descargado, arrancamos el instalador y seguimos las instrucciones que aparecen en pantalla.

C.2.1.1. Versión de Unity compatible con el proyecto

Una vez instalado el programa, seleccionaremos la versión de Unity que necesitamos para el correcto funcionamiento del proyecto: deberemos pinchar sobre **Installs** y allí dispondremos de un botón para agregar la versión que deseemos. El proyecto ha sido desarrollado en la versión **2019.1.3f1**, por lo que solo podemos garantizar su correcto funcionamiento en esta versión.

C.3. Cómo ejecutar el proyecto

Tras seleccionar la versión, deberemos agregar el proyecto a Unity. Esto se consigue pulsando sobre el botón **ADD**, en la sección **Projects**. De esta forma podremos seleccionar el proyecto de nuestro computador que queramos abrir en Unity.

Una vez Unity abra el proyecto elegido y estemos en su interfaz, pulsaremos sobre el botón "Play", situado en la parte de arriba de la **Escena**, junto al botón "Pause" y "Stop".

Con esto conseguiremos ejecutar la demo.

Apéndice D

División en Sprints: Actas y Sprint Backlog

A continuación se presenta la información generada como parte de la metodología con respecto al desarrollo del mismo en forma de *sprints*.

Se incluyen las tareas asumidas en cada lapso de dos semanas, y las actas recogidas en las reuniones de apertura/cierre de *sprint*, así como reuniones extraordinarias convocadas por necesidades de proyecto.

D.1. Sprint Backlogs

D.1.1. Primer Cuatrimestre

D.1.1.1. 1: 16/09-20/09

- *Research*: Iniciar Estado de la Técnica. Elaborar una lista de links con información

D.1.1.2. 2: 21/09-08/10

- *Diseño*: Diseñar el juego final que queremos mostrar
- *Research*: Revisión de juegos RPG/Estrategia por turnos/gestión de recursos relevantes para nosotros (indies y grandes)
- *Research*: Tirar del hilo de las referencias

- *Research*: Mirar arquitecturas y redes de sistemas de reglas modernos
- *Diseño*: Diseño de features de Jasonity

D.1.1.3. 3: 09/10-21/10

- *Diseño*: Escribir una primera versión del GDD "Space Prision"
- *Research*: Hacer tutorial de UnityProlog, y si sobra tiempo prototipo de personaje en Unity que "razona" usando reglas encadenadas (UnityProlog, y otro Prolog directamente)
- *Research*: Hacer tutorial de UnityLogic, y si sobra tiempo hacer prototipo de personaje en Unity que "razona" usando BDI (UnityLogic y otro BDI directamente, como el trabajo de Sergio)

D.1.1.4. 4: 22/10-21/11

- *Research*: Mirar arquitecturas y redes de sistemas de reglas modernos
- *Research, Front*: Super maratón de conceptos de DSI
- *Research, Back*: Investigar sobre Prolog
- *Diseño*: Reducir el tamaño del GDD y hacer un minijuego más claro y que abarque menos

D.1.1.5. 5: 22/11-02/12

- *Research*: Mirar arquitecturas y redes de sistemas de reglas modernos
- *Front*: Diseño simple siguiendo DSI para el front
- *Diseño*: Buscar juego de mesa en el que se oculte información

D.1.1.6. 6: 03/12-14/12

- *Research*: Preparar presentación de Narratech Laboratories

D.1.1.7. 7: 15/12-17/12

- *Research*: 10 páginas de documentación sobre agentes Racionales, lenguajes como prolog y finalmente AgentSpeak
- *Research*: 10 páginas de documentación sobre arquitectura BDI
- *Research*: 10 páginas de documentación sobre la herramienta en Unity
- *Research*: Investigar cómo funciona Jason por dentro
- *Research*: Investigar sobre BDI, Software Agents, arquitecturas, agentes cognitivos...
- *Research*: Investigar sobre Business Rules, Production Rules y Sistemas de Reglas

D.1.2. Segundo Cuatrimestre

D.1.2.1. 8: 07/02-17/02

- *Research*: Investigar cómo funciona Jason por dentro
- *Front*: Botón para insertar un GameObject master MAS en escena
- *Metodología, Diseño*: Definir unívocamente las funcionalidades del TFG y realizar una planificación basada en sprint para el desarrollo
- *Back*: Compilación primera

D.1.2.2. 9: 18/02-03/03

- *Metodología*: Merge del código del sprint anterior
- *Research*: Integrar en el editor de Unity las ventanas necesarias para la gestión de agentes
- *Back*: Codificar las funciones necesarias para comunicación entre agentes a nivel de sintaxis y lógica
- *Back*: Codificación del paso de mensajes entre agentes a nivel BDI
- *Back*: Codificación de eventos a nivel BDI
- *Back*: Codificación de eventos a nivel de sintaxis y lógica

D.1.2.3. 10: 04/03-17/03

- *Front*: Escena de prueba con agentes y environment
- *Front*: Integrar en el editor de Unity las ventanas necesarias para la gestión de agentes
- *Back*: Migración de código de Jason

D.1.2.4. 11: 18/03-31/03

- *Back*: Migración de código de Jason

D.1.2.5. 12: 01/04-14/04

- *Back*: Migración de código de Jason
- *Metodología*: Redacción de primer borrador de la memoria

D.1.2.6. 13: 15/04-28/04

- *Back*: Migración de código de Jason
- *Metodología*: Redacción de primer borrador de la memoria

D.1.2.7. 14: 19/04-20/05

- *Back*: Migración de código de Jason
- *Diseño*: Implementación de Jasonity Demo
- *Metodología*: Redacción de primer borrador de la memoria

D.1.2.8. 15: 21/05-31/05

- *Back, Metodología*: Corrección de errores

D.2. Actas

Todas las actas del proyecto están disponibles de forma pública en nuestro repositorio de Google Drive[1]. Los ficheros están nombrados con el formato YYYY-MM-DD para facilitar la visualización cronológica de las mismas, como se muestra en la figura D.1

Se corresponden a las reuniones de apertura y cierre de los *sprints*, además de algunas reuniones convocadas de forma extraordinaria para tratar problemas o aspectos del desarrollo que requerían ser tratados con urgencia.

Aunque no siempre ha sido posible, y en algunas ocasiones la acústica no acompañaba, se ha intentado grabar las reuniones presenciales para disponer de un archivo de registro más completo que la síntesis de las actas. Las pruebas se encuentran también en la misma carpeta que las actas.

[1] https://drive.google.com/open?id=1u2bCbLZnLiISfb1e6dC7vuhm8NG484zY

Nombre ↑	Propietario
grabaciones	yo
2018-07-02 - Primera previa	Federico Peinado
2018-07-16 - Segunda previa	Federico Peinado
2018-09-13 - Apertura_de_proyecto	yo
2018-09-24	IRENE GONZALEZ VELASCO
2018-10-11	IRENE GONZALEZ VELASCO
2018-11-08	yo
2018-11-22	yo
2018-12-14	IRENE GONZALEZ VELASCO
2019-01-07 - reunion skype	Caevias
2019-01-31.docx	JAIME MARÍA BAS DOMÍNGUEZ
2019-02-07	Caevias
2019-02-21 Skype Sprint 8	yo
2019-03-07	Caevias
2019-03-21 - Reunion Física	yo
2019-04-04	JAIME MARÍA BAS DOMÍNGUEZ
2019-05-05	Caevias
2019-05-25	IRENE GONZALEZ VELASCO

Figura D.1: Listado de las actas disponibles en la unidad de Google Drive del proyecto

Apéndice E

Aportaciones de los participantes

E.1. Jaime María Bas Domínguez

Durante la primera fase del proyecto, que abarcó todo el primer cuatrimestre hasta febrero, mi labor consistió en investigar sobre inteligencias artificiales utilizadas en videojuegos. Dediqué gran parte de mi tiempo a buscar juegos alabados por tener inteligencias complejas, así como otros que habían sido criticados por hacer justo lo contrario.

Para este fin, probé personalmente varios juegos con inteligencias llamativas y no tan llamativas, para observar qué destacaba tanto por lo bueno como por lo malo. Además, también recogí ideas de géneros y tipos de juegos en los que podría ser interesante aplicar tecnología BDI, y cómo podría impactar esto a la industria, a los desarrolladores y a los jugadores. Tras exponerlo en común con mis compañeros, llegamos a la conclusión de que BDI es una herramienta muy potente que permite hacer cosas hasta ahora (casi) nunca vistas y que podrían suponer uno de los siguientes grandes pasos en la industria.

Buena parte del primer cuatrimestre también estuvo dedicada al diseño de un juego, colaborando con Irene, que utilizaría BDI. Combinando sus conocimientos de Unity con mis conocimientos sobre videojuegos, hicimos unas primeras iteraciones con mucho potencial. Dicho juego, al que llamamos Project Space Prison, estaba centrado en una prisión espacial en la que el jugador controlaba al alcaide, la máxima autoridad del lugar. Los prisioneros pertenecían distintas especies alienígenas, por lo que cada uno tenía sus peculiaridades y capacidades especiales. Esto, combinado con el potencial de BDI, daba un punto de imprevisibilidad y aleatoriedad muy interesante

que podría hacer que cada partida contara una historia totalmente distinta. Cada uno de ellos contaba con un objetivo distinto, por lo que se explotaría al máximo todo el sistema de planes que nos brinda BDI.

En un diseño inicial del juego, se barajó la posibilidad de hacer que los guardias también tuvieran sus propios rasgos de personalidad y pudieran interactuar con los prisioneros, pero se decidió que era demasiado ambicioso y se recortó el proyecto, haciendo que los guardias fueran meros peones en las manos del jugador.

Finalmente, la idea del juego quedó apartada cuando vimos que el trabajo requerido en el desarrollo de la herramienta era sustancialmente mayor de lo previsto anteriormente.

Así pues, desde febrero en adelante, mi trabajo ha consistido en migrar una cantidad inmensa de código de Java a C#. Se nos encargó a Irene y a mí intentar hacer una versión reducida del sistema BDI para minimizar el trabajo necesario, pero pronto nos dimos cuenta de que era imposible reducir nada, ya que todo está muy interconectado.

Se nos dio también control sobre la arquitectura del proyecto y sobre la planificación del mismo. Todo el diseño general de la arquitectura, su división en paquetes y demás, es cosa nuestra. También establecimos ciertos "hitos"que debíamos alcanzar durante el desarrollo para fomentar el trabajo y que pudiéramos completar todo lo que debíamos hacer.

Durante este tiempo, aproveché para aprender y entender más en profundidad cómo funcionaba el sistema BDI por dentro. A base de leer código, buscar información para redactar presentaciones (como las que dimos para Narratech en diciembre y en abril) e investigar sobre todo el modelo psicológico sobre el que se basa este sistema, llegué a entenderlo y ser capaz de explicarlo con facilidad, por lo que me he encargado sobre todo de esta parte.

Mi trabajo de migración se ha centrado, como resulta lógico, en la capa de BDI. Toda la implementación de todos los elementos encargados de Creencias, Deseos e Intenciones que están contenidos en la carpeta BDIManager son trabajo propio. Debido a la gran interconexión entre clases que presenta Jason, parte de mi trabajo quedó colgando a la espera de que otros elementos del sistema estuvieran implementados.

En vista de que era un proceso muy lento y que muchas veces nos dejaba sin trabajo, decidimos hacer una sesión de crunch y terminar de implementar toda la capa de sintaxis, por lo que me trasladé también a esa sección del proyecto para colaborar. Un buen número de las clases incluidas en ese paquete (especialmente las que estaban más directamente relacionadas con BDI) son propias.

Durante el trabajo de migración observamos que, debido a las peculiaridades de C#, ciertas funcionalidades de la implementación en Java de Jason no iban a ser nada fáciles de llevar a cabo. Debido a su complejidad, decidimos eliminar aspectos como la mensajería (que también comencé a implementar yo en su momento) y la concurrencia (que aparecía en muchas de las clases que implementé, aunque siempre la dejaba de lado por no ser totalmente compatible).

Tras haber finalizado la migración de las cerca de 24.000 líneas de código, pasamos a la depuración. Junto a Irene y Alejandro, nos dedicamos a buscar y corregir todos los errores que encontramos en el proyecto. Tras varias sesiones de revisar todas las clases una y otra vez para corregir todo, conseguimos compilar el proyecto y pasamos a desarrollar y, una vez más, depurar los tests unitarios para probar cada uno de los aspectos del programa y asegurar su correcto funcionamiento.

Durante la implementación de la versión 2.0 de Jasonity, participé en la implementación de alguna clase menor para aliviar algo de peso a mis compañeros, que estaban más enfocados en las clases más importantes.

Por último, en cuanto a la redacción de la memoria, me he centrado específicamente en la parte de BDI, ya que ha sido lo que más he tratado durante el desarrollo; y en los apartados referentes al análisis, diseño e implementación de Jasonity y Jasonity 2.0, ya que fui uno de los responsables de estudiar y diseñar sus arquitecturas.

También me he encargado de traducir las dos secciones redactadas en inglés (introducción y conclusiones) y de revisar todo lo redactado por mis demás compañeros para buscar posibles errores o inconsistencias y asegurar que el resultado final es lo más homogéneo posible.

E.2. Álvaro Cuevas Álvarez

En un principio antes de ponerme a hacer nada, mi trabajo consistió en ponerme al día con todo lo que íbamos a utilizar durante todo el proyecto. Empecé investigando sobre BDI, agentes cognitivos y arquitecturas que pudiésemos utilizar durante el proyecto además de comprender los conceptos más básicos que íbamos a utilizar. Gran parte de mi participación en el primer cuatrimestre a consistido en esta búsqueda de información.

A la vista de que nuestro proyecto iba a estar enfocado a utilizar el lenguaje de AgentSpeak empecé a investigar sobre su gramática para comprenderla a la hora de intentar implementarla en el proyecto, de la misma forma cuando se decidió que crearíamos un juego me encargué de la búsqueda de juegos de mesa que tuviesen como mecánica ocultar información al resto de jugadores para intentar crear un juego en el que el usuario pudiese jugar contra distintos agentes y ser uno más de la partida.

Una vez se decidió que íbamos a crear una versión de Jason, donde pudiésemos trabajar con agentes inteligentes basados en el algoritmo de BDI en Unity, se nos encargó a Alejandro y a mí la parte de sintaxis y de crear el parser para leer archivos asl. Empezamos investigando esta parte de Jason para comprender como lo hacían en la herramienta, para posteriormente intentar simplificarlo todo lo posible . Se llegó a la conclusión de que era imposible reducir esta parte y se dividió nuestro trabajo en traducir las clases de sintaxis y entender y crear el parser que utilizaban en Jason.

Tras una reunión entre el director del proyecto y nosotros, se descubre que por debajo Jason se utiliza una herramienta llamada JavaCC para generar su parser. Recae sobre mi la tarea de poder crear el parser para poder unirlo con la otra parte de sintaxis. La primera parte de esta tarea consistió en aprender que era esa aplicación y de como usarla.

Consistía en una herramienta que desde un archivo con extensión .jj que contenía la gramática que el usuario quisiese crear(en lenguaje Java) generaba los archivos que se necesitaban para tener el parser completo(los archivos generados eran clases de Java). Esto supuso un problema ya que lo que necesitábamos era que nos generase el parser en C# por lo que tuve que pasar a buscar herramientas que hiciesen el mismo trabajo que JavaCC pero nos generase el parser en el lenguaje que queríamos.

De esta investigación encontré una herramienta muy parecida, de hecho en su documentación hablaban de que era la alternativa a JavaCC para poder generar el parser en C#, llamada CSJavaCC. Pero, aunque era la herramienta más prometedora, se descartó ya que utilizaban una versión de JavaCC muy antigua y, ante la falta de documentación de como utilizar su

herramienta, no teníamos tiempo de intentar comprender la herramienta ya que el resto de compañeros dependian de que yo terminase el parser.

Por suerte, tras más investigación, encontré que una de las últimas versiones de JavaCC tenía la posibilidad de generar el parser en C#. Mi siguiente cometido fue una serie de correos y mensajes con los creadores de JavaCC para conseguir información de si habían conseguido el generador en el lenguaje que necesitábamos. A la contestación positiva del dueño del repositorio de GitHub de la herramienta, me puse manos a la obra aprendiendo de como utilizar la herramienta para generar el parser, siguiendo las pautas que me habían dado. Esta fue la parte que más tarde ya que la información que me iban dando se iba demorando muchos días.

Tras crear el archivo AS2JavaParser.jj en lenguaje C#, que contenía todas las especificaciones de la gramática de AgentSpeak, conseguimos generar los primeros archivos que necesitábamos para el parser. Tras corregir una serie de errores y de la implementación de algunas clases que se necesitaban para el parser se dio, satisfactoriamente, esta tarea de general el parser por concluida.

En la última parte de mi participación en la implementación del código consistió en unir las distintas del código que habían implementado mis compañeros con el parser que había generado con la herramienta JavaCC. Para esta tarea nos encargamos Irene y yo, ya que ella tenia un conocimiento más extenso del código que habían implementado y yo de las clases que había creado para el parser.

Además de esto también me encargué de la corrección de errores ocasionados al migrar el código de Java en C#, añadiendo algunos aportes en la *Wiki*. También me encargué de la implementación de todas las clases de la carpeta functions, estas funciones representan todas las operaciones que se utilizan en la gramática de AgentSpeak. Además ayudé con la creación de algunos test de prueba para esta primera versión de Jason.

A la hora de crear nuestra propia versión de Jason, mi tarea consistió en crear algunas clases que pudiesen ayudar a mis compañeros que tenían un peso mayor en esta parte. Me encargue de una parte de revisar el código para su correcto funcionamiento.

Por último, mi aporte en la redacción de la memoria ha consistido en narrar todos los datos que he conseguido de la herramienta JavaCC, el proceso que he seguido para poder utilizarlo, cómo se utilizaba y todo lo que relacionado con los archivos que me generaba.

Toda esta información se recoge en el apartado de la memoria 5.1 del capítulo 5. En este apartado también se menciona el contenido del archivo

AS2JavaParser.jj que, como antes hemos mencionado, recogía todas las especificaciones de la gramática de AgentSpeak. Finalmente, a la hora de la entrega, revisé gran parte de la memoria para no encontrar sorpresas derivados de despistes.

E.3. Alejandro García Montero

Durante la primera fase del proyecto (el primer cuatrimestre) mi tarea consistió en buscar información sobre herramientas multiagente que pudieran existir ya en el mercado y pudieran servirnos tanto de referencia, como de posible alternativa para enfocar el TFG si resultaban funcionales para nuestro cometido.

Así pues, mucho de mi tiempo fue invertido en comunicarme con los autores de software potencial o de artículos sobre agentes inteligentes que iba hallando por la red, de modo que pudiera entender mejor su trabajo y averiguar la existencia de otras fuentes de información sobre arquitectura BDI que ellos pudieran haber utilizado en sus investigaciones.

Continué con el proceso descrito en el anterior párrafo hasta que hallé la herramienta para Unity denominada UnityProlog, creada Ian Horswill. Tras varios días de infructuoso esfuerzo por tratar de asociar la herramienta a Unity, ya que se trata de un software libre que solo está en disponible en el GitHub de Ian, contacté con Don Federico, director de este TFG, para tratar de ponerla en marcha con su ayuda. Después de toda una tarde intentando hacer funcionar la herramienta, logramos generar un archivo ".dll" el cual, al añadirse dentro de la carpeta "Assets" de un proyecto de Unity, hace que la interfaz de Unity muestre las funcionalidades que UnityProlog posee.

En los días posteriores, comencé a configurar un proyecto en Unity según los parámetros indicados en la documentación de UnityProlog y logré empezar a probar pequeños ejemplos de problemas de Prolog en la consola que incorpora la propia herramienta de Ian. Tras esto, también logré ejecutar pruebas de código Prolog sin necesidad de tener que escribir el problema completo en la consola, ya que esta no lo permite. Así pues, siguiendo la documentación, configuré un componente de Unity como base de conocimiento para cargar en ella los archivos Prolog y, así, tan solo tener que ejecutar las consultas en la consola para obtener la respuesta a los problemas planteados en dichos archivos.

No obstante, para la siguiente etapa de investigación sobre UnityProlog necesité la ayuda de mi compañera Irene González, ya que ella tenía mucha más experiencia que yo trabajando con Unity y esta fase requería invocar el código Prolog desde los scripts CSharp de Unity.

Tristemente, esta tarea resultó mucho más compleja de lo esperado ya que, pese a seguir los, en este apartado, confusos pasos de la documentación, no logramos invocar Prolog desde C# y consultar la intrincada implementación del código fuente de UnityProlog no ayudó a solventar las dudas sobre como realizar este apartado. Por tanto, se decidió abandonar esta herramien-

ta.

Al termino del primer cuatrimestre y comienzo de la segunda fase del proyecto, mi labor consistió inicialmente en desarrollar una versión simplificada del *Parser* de Jason para integrarlo en una versión, también simplificada, del propio Jason. Sin embargo, tras unas semanas de estudio de este software por parte de mis compañeros de equipo, se llegó a la conclusión de que no era posible reducir Jason a una versión más simplificada de si mismo, luego habría que reimplementar todo Jason en C#, con lo que mi *Parser* simplificado no encajaría en esta nueva versión. De modo que aparté este trabajo en pos de unirme a mis compañeros en la titánica tarea de migrar Jason.

Mi primera tarea en esta fase, fue la de migrar todas las clases que se encontraran en la carpeta AsSyntax ya que estas eran necesarias para el funcionamiento del *Parser* automático en el que estaba trabajando mi compañero Álvaro Cuevas y para el conjunto de Jasonity (la herramienta que da nombre a nuestro trabajo).

Durante esta etapa mantuve bastantes reuniones online con el director del trabajo, Don Federico, a modo de revisión de errores de sintaxis en el código, puesto que las diferencias de lenguaje entre Java y C# son notorias y hubo que replantear la implementación de algunas clases ya que su código original contenía funciones, métodos y referencias a clases del propio lenguaje Java que, o bien no existen en C#, o bien se denominan de otra forma y tuvimos que encontrar su equivalente. Este último problema propició la creación de una *Wiki* en la que todos los miembros del equipo íbamos apuntando y dejando constancia de las diversas diferencias de Java con respecto a la versión CSharp del código, de modo que tuviéramos una lista de referencias a la que acceder cuando nos encontrásemos de nuevo con estos problemas a lo largo de la implementación.

Una vez terminada la carpeta AsSyntax mi siguiente trabajo consistió en migrar las 97 clases contenidas en la carpeta StdLib, las cuales representan a las *InternalAction* de Jason. Pese a que son un número elevado de clases, su carga de código era bastante liviana en comparación a la de las clases de AsSyntax, por lo que no fueron demasiado pesadas de implementar. De igual forma, mantuve reuniones online con Don Federico para revisar y corregir errores de sintaxis que se produjeron durante el desarrollo de estas clases y como apunte final para esta fase, también participé en el desarrollo de test unitarios para la depuración del código.

Durante la etapa final del proyecto, que supuso la creación de nuestra propia versión de Jason, mi tarea fue la de terminar el diseño del Parser simplificado en el que comencé a trabajar a comienzos del cuatrimestre. Una vez terminado, desarrollé una versión funcional del mismo para la demo visual,

para lo cual conté una vez más con la inestimable ayuda de mi compañera Irene González.

A modo de aporte final, estuve encargado de redactar la sección sobre el **Razonamiento** y las distintas **Herramientas software** en el capítulo sobre el "Estado de la técnica", la redacción de la sección de la Capa Lógica en el capítulo 5: "Reingeniería, diseño e implementación de Jasonity" (excepto de la parte referente a JavacCC); y de desarrollar la descripción del nuevo Parser en el capítulo 6: "Diseño e implementación del sistema definitivo".

E.4. Juan Gómez-Martinho González

A lo largo del proyecto he realizado diferentes tareas para llevar a buen término el mismo.

Durante la primera mitad del primer cuatrimestre tres tareas principales ocupaban la mayor parte del tiempo: el encauce de la **metodología**, la **investigación** del estado de la técnica en lo referente a herramientas BDI y ejemplos en videojuegos y el **diseño de la interfaz** de la herramienta.

A nivel **metodológico**, dado que el muchos miembros del equipo tenían poca o ninguna experiencia en *Scrum*, era importante que todos los miembros se acostumbraran a seguir los procedimientos formales (el uso de Trello, mantener el propósito del cierre/apertura de Sprint, tomar acta...) Por ello, he adoptado a nivel funcional el rol de **Scrum Master** dentro del equipo. Como Scrum Master, una de mis tareas principales era facilitar las reuniones bisemanales (cuadrar los horarios de todos los miembros recabando sus horas libres, llevar el acta de la reunión, conducir los temas a tratar...) y priorizar las tareas del backlog junto con el Product Owner (en nuestro caso, el director del proyecto).

Como el resto de mis compañeros, los primeros meses la mayor parte de mi tiempo se reservó para el **análisis del estado de la técnica**. Ya que otros alumnos se encargaron de investigar herramientas y lenguajes genéricos, mi investigación se centró en las herramientas BDI de Unity (los TFGs previos). Por ello, **mi foco de atención caía sobre Jason y conectores con Unity**, así como herramientas nativas del motor de juego que pudieran ahorrarnos trabajo. Además, me encargué de buscar e investigar sobre los ejemplos de IA en videojuegos, encontrando numerosos artículos y publicaciones sobre las técnicas empleadas en los mismos. Como se trata de una industria muy cambiante y de avance muy rápida, **quise centrarme tan sólo en los juegos publicados en la última generación de consolas**, ya que nos ofrecerían una imagen general del estado de la IA más reciente y relevante (se consideró que encontrar ejemplos de hace más de 10-15 años no nos aseguraba que las técnicas empleadas hubieran mantenido una continuidad significativa).

Además, dado que estuve presente durante la elaboración del TFG de González (2018), tenía **experiencia previa en el uso de Jason**, por lo que también pude ayudar a mis compañeros a entender en la etapa inicial el funcionamiento de un sistema multiagente basado en BDI. Por ello, los primeros meses de investigación realizada por otros compañeros de herramientas BDI pudieron ser validadas y probadas en fases tempranas a través de dicha experiencia para no gastar mucho tiempo en probar soluciones que,

por la experiencia, no se adecuaban a las características de nuestro problema.

Una vez hubo acabado la fase de investigación por mi parte, y mientras el resto de miembros del equipo investigaban herramientas prometedoras, durante el primer cuatrimestre **seguí el Diseño Guiado por Objetivos** para elaborar la especificación del Sistema Interactivo subyacente en Jasonity. Para ello, dediqué varias semanas a adquirir los conocimientos de dicha rama; ya que en el plan de estudios de Ingeniería del Software no se cursa la asignatura correspondiente. Se me facilitó el material necesario (apuntes de alumnos, transparencias, una guía de la asignatura) para dicha fase de aprendizaje, que se llevó a cabo de forma maratónica en un tiempo más que reducido. Una vez se consideró la adquisición de conocimientos completada, **seguí la metodología de DSI hasta obtener un diseño base** con las funcionalidades que parecían adecuarse mejor a la herramienta que queríamos consturir.

Al concluir el primer cuatrimestre, se planteó el dilema acerca de la viabilidad de la implementación de la herramienta gráfica. A pesar de que había llegado a obtener diseños bastante concretos de las funcionalidades, sin el *back-End* funcionando iba a ser imposible asumir el desarrollo del *front-End*.

Es por ello que durante el segundo cuatrimestre, por mi experiencia profesional con Unity, mi aportación consistió principalmente en la **programación junto con Irene González de la capa intermedia que conecta el *Back-End* con el funcionamiento de los agentes en el entorno de Unity**, junto con la representación del *environment*. Por la experiencia en Unity anteriormente mencionada, también participé activamente en el diseño y desarrollo de "I can't sleep". Mi mayor aportación en ese aspecto fue el diseño y construcción de la escena, y el refinamiento del funcionamiento que mostrara el potencial de nuestros agentes BDI.

Durante los últimos meses ha sido **mi responsabilidad unificar todo el conocimiento adquirido por los miembros del equipo y el desarrollo del proyecto para generar una estructuración y división de la memoria óptimos**, evitando información duplicada por un lado o que se diera por supuesto que algún detalle fuera a ser explicado por otro compañero y fuera omitido a pesar de ser esencial. Crear la base de referencias supuso un reto dado el desconocimiento que envolvía la forma de especificarlas. Asumí numerosas relecturas de la memoria buscando posibles referencias perdidas que no se hubieran plasmado en el texto, para asegurarme de que todo texto referenciado quedara plasmado en la bibliografía.

Mi labor con la memoria también incluye **adquirir experiencia con LaTeX para mejorar el formato y asistir a mis compañeros** en las dudas y problemas que pudieran derivarse del uso de un lenguaje y forma de

redacción desconocido hasta ahora.

El mismo principio de unificación y formato es aplicable con la presentación del proyecto. Tras la experiencia resultante de las presentaciones frente a Narratech Laboratories se hizo notable un alto riesgo que podría aparecer en la defensa final del proyecto: la información común necesaria para exponer las aportaciones individuales se repetían a lo largo de la exposición; comiendo el tiempo limitado del que disponemos y creando una sensación de caos y descontrol que podría lastrar la evaluación del proyecto.

En ese aspecto, las últimas semanas de TFG estuve **encargado de unificar el formato de la memoria, diseñar una presentación coherente y con un uso del tiempo óptimo**. Con la presentación organizada por Narratech para la exposición final de nuestros trabajos el 5 de junio, que nos serviría como práctica para la defensa final del Trabajo de Fin de Grado.

E.5. Irene González Velasco

A lo largo de todo el proyecto he estado involucrada en multitud de tareas que buscaban la realización correcta del proyecto.

Durante el primer cuatrimestre me centré principalmente en la investigación de las tecnologías y herramientas ya existentes junto con mis compañeros, así como en el diseño de un pequeño juego con el que poner en práctica nuestra herramienta.

Al ser una de las pocas personas con experiencia en desarrollos con Scrum, serví de apoyo a Juan a la hora de explicar conceptos e implantar el uso de las herramientas propias de los desarrollos Scrum y participé en la división de tareas en los sprints del segundo cuatrimeste por mi conocimiento global de toda la herramienta.

Durante la fase de investigación me centré principalmente en Jason y en leerme la documentación disponible, principalmente el libro. Aunque no tenía conocimientos previos de Prolog, colaboré con Alejandro en la investigación de Unity Prolog, ya que para el correcto funcionamiento de dicha herramienta se necesitaban conocimientos de Unity que sí tenía. En esta etapa de la investigación me dediqué a leer la documentación disponible de Unity Prolog y, junto con Alejandro, intentar implementar una demo. Por diversos problemas que aparecieron en esta fase, se tomó la decisión de descartar Unity Prolog como herramienta.

Al contar con esos conocimientos en Unity, dediqué la otra mitad de mi tiempo a lo largo del primer cuatrimestre en diseñar junto a Jaime un pequeño juego. Dicho juego estaba pensado para implementarse utilizando la herramienta desarrollada, para que sirviese como demo y ejemplo de uso de la misma. Este juego consistía en una prisión espacial en la que el jugador era el alcaide y la máxima autoridad en la misma. El jugador tenía que gestionar a los prisioneros y a los guardias. Los prisioneros eran de especies alienígenas diferentes, lo que hacía que cada uno tuviese unas características y una forma de interactuar con su entorno únicas, mientras que los guardias tendrían todos las mismas características y servirían únicamente para llevar a cabo las órdenes del alcaide. Todos los prisioneros tendrían objetivos diferentes, para que, unido a sus características específicas, se pudiera emplear el modelo BDI en su implementación. Este juego quedó descartado después de decidir que era prioritario el desarrollo de la herramienta.

El segundo cuatrimestre empleé mi tiempo en el diseño e implementación del código, implementación de una pequeña demo, diseño e implementación de pruebas unitarias, apoyar a Juan en sus tareas de Scrum master y escribir la memoria.

A lo largo del diseño e implementación del código me dediqué principalmente a la capa de BDI a la vez que realizaba tareas de apoyo en la capa de lógica y en el front-end. En la capa de BDI desarrollé el ciclo de razonamiento y el agente y su arquitectura. Por los conocimientos previos del entorno Unity, Juan y yo diseñamos e implementamos el front-end y cómo los agentes y todo el sistema BDI debía interactuar con Unity. Gracias a toda la investigación sobre Jason que realicé en el primer cuatrimestre, contaba con conocimientos globales de la arquitectura de Jason así que apoyé a mis compañeros en el desarrollo de la capa de lógica y su integración con la capa BDI.

Relacionado con la implementación, dediqué parte de mi tiempo a buscar implementaciones alternativas a cosas que Java permitía pero que no tenían traducción exacta en C#, como por ejemplo un planificador de tareas, y la inclusión de concurrencia en la herramienta. En esta primera versión quedó descartado al final la concurrencia ya que se salía del alcance del trabajo.

Una vez estuvo el código completo migrado, comenzamos a diseñar e implementar pruebas unitarias utilizando la utilidad Test Runner, integrada en Unity, y el framework NUnit para pruebas unitarias en C#. Dichas pruebas consistieron en testear uno a uno los métodos más importantes y críticos del proyecto para comprobar su correcto funcionamiento.

Dado que la idea inicial de un juego como demo fue descartada porque se salía del alcance del proyecto, diseñé junto a Juan una nueva demo para poder demostrar el funcionamiento de la herramienta y su uso. Esta demo consiste en dos agentes que intentan dormir, uno no puede dormir con la luz apagada y otra no puede dormir con la luz encendida y ambos pelean por encender o apagar la luz y poder dormir.

Una vez tuvimos claro que la arquitectura original era demasiado grande como para integrarla correctamente en Unity y se optó a realizar una arquitectura reducida me encargue del diseño de la nueva arquitectura, apoyada siempre por mis compañeros.

Una vez la arquitectura quedó aprobada, me encargue de implementar la demo diseñada junto a Juan y revisar todo el código de mis compañeros para unificarlo y que funcionase.

Al haber estado centrada en el código, de la memoria tan sólo he rellenado el apartado 5.2 del capítulo 5, en el que cuento el detalle de la implementación de la capa BDI y el apartado 5.4 en el que hablamos de problemas surgidos a lo largo del desarrollo, como las implementaciones alternativas mencionadas anteriormente. Además de colaborar en la escritura de los apartados análogos en el capítulo 6. También he dedicado parte de mi tiempo a realizar el diagrama de clases con todo el diseño de la arquitectura 6 y el diagrama de

clases con el diseño de la demo, ya que tengo los conocimientos de UML más amplios de todos mis compañeros. También he colaborado en las labores de revisión y corrección de la memoria.

Las tareas de Scrum Master que realicé durante el segundo cuatrimestre consistieron en dividir las tareas de implementación del back-end, priorizarlas y su separación en los diferentes sprints, siempre siguiendo el consejo de Juan, que era realmente el Scrum Master, pero que al conocer menos detalles técnicos de la implementación necesitaba apoyo.

Bibliografía

Bethesda Game Studios. The elder scrolls iv: Oblivion. PC, Xbox 360, PlayStation 3, 2006.

Bethesda Game Studios. Fallout 76. PC, Xbox One, PlayStation 4, 2018.

Bordini, R. H., Hübner, J. F. y Wooldrige, M. *Programming multi-agent systems in AgentSpeak using Jason*. 2007.

Capcom. Resident Evil. PlayStation, Windows, Sega Saturn, 1996.

Capcom. Resident Evil 2. PlayStation 4, Xbox One, PC, 2018.

Dyckhoff, M. Ellie: Buddy AI in The Last of Us. *Game AI Pro*, vol. 1(2), Disponible en `http://www.gameaipro.com/GameAIPro2/GameAIPro2_Chapter35_Ellie_Buddy_AI_in_The_Last_of_Us.pdf`.

Epic Games. 1998.

Euler Ruiz. *El Conocimiento Silencioso*. 2000.

genDesign; SCE Japan Studio. The Last Guardian. PlayStation 4, 2016.

González, S. Desarrollo de un Sistema Multiagente Basado en Creencias Deseos e Intenciones para Modelar Prsonajes Autónomos en Videojuegos utilizando Jason y Unity. Disponible en `https://eprints.ucm.es/48883/1/114.pdf`.

Ian Horswill. UnityProlog. Disponible en `https://github.com/ianhorswill/UnityProlog`.

IO Interactive. Hitman 2. PlayStation 4, Xbox One, PC, 2018.

Jomi F. Hübner and Rafael H. Bordini. Disponible en `http://jason.sourceforge.net/wp/`.

Kogan, P., Parra, G. A. y del Castillo, R. Diseño de agentes experimentando con robots que juegan al fútbol en ambientes reales y simulados. Disponible en http://sedici.unlp.edu.ar/handle/10915/20729.

Lionhead Studios. Black & White. PC, Mac OS, 2001.

Miyake, Y. y Hasegawa, I. AI AND FUNDAMENTAL GAME TECHNOLOGIESIN FINAL FANTASY XV . https://es.slideshare.net/rakutentech/ai-and-fundamental-game-technologiesin-final-fantasy-xv, 2017.

Miyake, Y., Shirakami, Y., Kazuya Shimokawa, K. N., Komatsu, T., Tatsuhiro, J., Prasertvithyakarn, P. y Yokoyama, T. A Character Decision-Making System for FINAL FANTASY XVby Combining Behavior Trees and State Machines. *Game AI Pro*, vol. 1(2), Disponible en https://www.crcpress.com/rsc/downloads/Unprotected_K26804_Rabin_Chapter_11.pdf.

Moon studios. Ori and the Blind Forest. Xbox 360, PC, 2015.

Naughty Dog. The Last of Us. PlayStation 3, 2013.

Poli, N. Game engines and mas: Bdi & artifacts in unity. Disponible en https://amslaurea.unibo.it/15657/1/poli_nicola_tesi.pdf.

Ramón Ruiz. *Historia y Evolución del Pensamiento Científico*. 2006.

Russell, S. J. y Norvig, P. *Artificial Intelligence: A Modern Approach*. 1995.

SCE Santa Monica Studio. God of War 4. PlayStation 4, 2018.

Square Enix. Final Fantasy XV. PlayStation 4, Xbox One, PC, 2016.

Steven T. Wright . 'BOY'was Atreus tricky to implement, says God of War director. Disponible en https://www.gamasutra.com/view/news/318012/BOY_was_Atreus_tricky_to_implement_says_God_of_War_director.php.

Sánchez-López, A., Romero, F. y Martín-Solís, H. Un Sistema de Control Autónomo para Personajes de Videojuegos Basado en el Modelo Cognitivo Creencia-Deseo-Intención. Disponible en https://eprints.ucm.es/38416/1/Memoria.pdf.

Team ICO. ICO. PlayStation 2, 2001.

Team ICO; SIE Japan Studio. Shadow of the Colossus. PlayStation 2, 2005.

Ubisoft. Assassin's Creed. PlayStation 3, Xbox 360, Windows, 2007.

Ubisoft. Assassin's Creed: Unity. PlayStation 4, Xbox One, Windows, 2014.

Unity Technologies. 2005.

Universitat Politècnica de Catalunya. Sistemas multiagente. Disponible en `://www.lsi.upc.es/~bejar/ecsdi/Teoria/ECSDI02a-Agentes.pdf`.

Valve. Portal. PC, 2007.