UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Infraestructura multi-agente para la argumentación con razonamiento basado en casos en Python

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Luis Serrano Hernández

Tutor: Dr. Vicent Botti Navarro
Dr. Jaume Jordán Prunera

Curso 2019-2020

Resum

La capacitat per argumentar dota els agents software d'una major intel·ligència. Esbrinar com aconseguir que un agent intel·ligent tinga qualitats pròpies de l'àmbit de l'argumentació requereix d'un extens recorregut per la història de la intel·ligència artificial, concretament per la branca que treballa la representació del coneixement. En aquest document es presenta un anàlisi de l'estat de l'art respecte a estos temes, des de la definició d'«argumentació» fins a l'estudi de plataformes ja existents amb un propòsit semblant. Este repàs general conduix a la realització i redacció d'un posterior anàlisi centrat a buscar formes de millorar sobre aquesta matèria, prenent com a base una infraestructura multi-agent que ja permet la realització de processos argumentatius entre els seus propis components. De tal investigació es deriva a una sèrie de propostes de millora que, de ser implementades, suposarien un benefici per a les investigacions dins de l'àmbit. Finalment s'expressa el desenvolupament d'estes millores, concloent amb un balanç dels objetius plantejats inicialment en contraposició a les metes aconseguides.

Paraules clau: Agents, intel·ligents, coneixement, casos, argumentació

Resumen

La capacidad de argumentar dota a los agentes software de una mayor inteligencia. Averiguar cómo lograr que un agente inteligente posea cualidades propias del ámbito de la argumentación requiere de un largo recorrido por la historia de la inteligencia artificial, más concretamente por la rama que estudia la representación del conocimiento. En esta memoria se presenta un análisis del estado de la cuestión al respecto de estos temas, desde la definición de «argumentación» hasta el estudio de infraestructuras ya existentes de propósito similar. Este repaso general conduce a que se realice y redacte un posterior análisis centrado en buscar formas de mejorar sobre esta materia, tomando como base una plataforma multi-agente que ya permite la realización de procesos argumentativos basados en casos entre sus componentes. De tal investigación se deriva a una serie de propuestas de mejora que, de ser implementadas, supondrían un beneficio para las investigaciones dentro del ámbito. Finalmente se expresa el desarrollo de estas mejoras, concluyendo con un balance de los objetivos planteados frente a las metas alcanzadas.

Palabras clave: Agentes, inteligentes, conocimiento, casos, argumentación

Abstract

The ability of reasoning provides more intelligence to software agents. Understanding how to achieve the goal of making an intelligent agent capable of reasoning requires of long travel across the artificial intelligence history or, more precisely, the history of the branch regarding knowledge representation. In this document an analysis of the state-of-the-art in terms of research around these histories is presented, from what "reasoning" means to the study of a currently existing infrastructures with a similar goal. From this research a new analysis is required, one that is focused on seeking ways of improving in the area, using as a base a multi-agent platform that already allows the execution of case-based argumentative processes between its components. And from this research several improvement ideas are suggested. Eventually, these new features are developed, concluding with a balance between the planned objectives and the reached goals.

Key words: Agents, intelligence, knowledge, cases, argumentation

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

La habilidad para analizar el entorno en el que se desenvuelven, captando y memorizando cada detalle, permite a los seres humanos almacenar conocimiento que pueden reutilizar en un futuro para resolver problemas a los que ya se hayan enfrentado. Pero el uso que los humanos le dan al conocimiento no termina ahí. La capacidad para argumentar es una cualidad que ha sido siempre atribuida a los seres humanos a lo largo de la historia, las personas son los animales racionales. Argumentar es una acción que puede realizar una persona en sociedad, o para sí misma. El proceso argumentativo permite partir del conocimiento que se posee e inferir nuevos conceptos desde ahí. La capacidad de almacenar e inferir conocimiento es un potente motor para la resolución de conflictos y problemas, un motor que se vuelve más poderoso cuando se combina con los de otras personas. Las capacidades cognitivas son limitadas, pero cuantas más personas se suman, más información poseen.

La investigación en el ámbito de la inteligencia artificial es inmensamente amplia y multidisciplinar. Con los años se han definido distintos modelos de tecnología que claman dotar de inteligencia y capacidad para la resolución de problemas complejos. Los agentes inteligentes nacen como un software que es capaz de valerse por sí mismo para cumplir la meta por la cual fue diseñado. Muchas veces estos agentes funcionan a base de aplicar una serie de reglas de su motor de inferencia sobre una serie de datos de su base de conocimiento. Son un logro excepcional de la evolución tecnológica pero, ¿qué sucedería si estos agentes obtuviesen la capacidad de argumentar? ¿Serían más capaces? ¿Más «inteligentes»? Estas son preguntas que se formulan dentro de este trabajo y a las que se pretende dar una respuesta orientativa, pero no definitiva, porque la tecnología y la investigación siempre avanzan.

1.1 Motivación

El desarrollo de este proyecto fue propuesto por los tutores del mismo como parte de las Becas de Colaboración para formarse en el grupo de investigación GTI-IA (Grupo de Tecnología Informática - Inteligencia Artificial) del Institut Valencià d'Investigació en Intel·ligència Artificial (VRAIN).

Inicialmente la propuesta consistía en retomar el proyecto «Una infraestructura para agentes argumentativos» [1] dotándole de nuevas características. El trabajo de Jordán et al. fue desarrollado con la intención de crear una infraestructura que todavía no existía. Dicha infraestructura proporcionaría un entorno en el que poder trabajar con sociedades de agentes argumentativos. Estos agentes estarían dotados de capacidad para aprender y por otro lado, para apoyar o desaprobar valores que otros agentes pudieran proponer.

El equipo defiende que dotar a un agente de capacidades para argumentar le permite adoptar comportamientos más inteligentes.

La primera versión de esta infraestructura fue desarrollada en Java y lanzada en enero de 2010. Los cinco años posteriores fue recibiendo revisiones y actualizaciones hasta abril de 2015. Actualmente la infraestructura ha logrado cumplir con los objetivos que motivaron su desarrollo, pero, con el fin de seguir expandiendo sus posibilidades y hacerla más accesible a todo tipo de desarrolladores, se ha decidido extender su funcionalidad permitiendo facilitar la creación de nuevos agentes argumentativos con comportamientos diferenciadores. Así mismo, una interfaz gráfica que permita visualizar las relaciones y comunicaciones dentro del entorno ayudaría en gran medida a monitorizar y depurar la ejecución del programa y, al mismo tiempo, se contaría con una potente herramienta gráfica con la que realizar comprobaciones teóricas.

1.2 Objetivos

Para dar un sentido a las motivaciones expuestas son varias las metas a lograr. En primer lugar, por medio de la genericidad que nos ofrece el lenguaje de programación, se debe permitir que los agentes que conforman las sociedades tengan distintas formas de afrontar los problemas y de relacionarse con el resto de agentes. Si bien esto es posible actualmente, la forma de hacerlo es muy poco eficiente. Para lograr esto se pretende generalizar la versión del agente argumentativo principal ya implementado.

También es necesario dedicar tiempo a las comunicaciones, las cuales se requiere reimplementar completamente para poder cumplir con los requisitos. La nueva versión de las comunicaciones debe trabajar con sockets Java y funcionar en varias fases guiadas por una máquina de estados finitos, de no proponerse una solución mejor.

Con el fin de facilitar el lanzar experimentos y obtener trazas de los argumentos lanzados, incluyendo los resultados generados tras las ejecuciones, se plantea como necesario crear una interfaz gráfica de usuario que haga más sencilla la interacción entre el usuario y la máquina. En el mismo camino de acomodar el uso para el personal menos especializado, se ha determinado que será necesario crear una documentación consultable sobre la implementación y un manual de manejo.

1.3 Estructura de la memoria

Este capítulo se presenta como un pequeño índice general comentado. En este se va a hacer un recorrido rápido sobre el contenido de los capítulos que conforman la memoria. La intención es servir de orientación para que con su simple lectura se obtengan nociones básicas sobre el contenido presente en cada capítulo.

El primer capítulo del trabajo comienza exponiendo un análisis del estado de la cuestión. Dicho análisis está conformado por dos partes, una primera en la que se introducen conceptos técnicos a modo de contextualización y una segunda en la que se explica desde que punto se parte en el desarrollo de este proyecto. Una vez establecida la casilla de salida se procede a realizar un análisis de la situación, que trata de determinar los aspectos menos óptimos y más contraproducentes susceptibles de ser mejorados, pero también teniendo en cuenta aquellos que funcionan perfectamente y cuya conservación puede traducirse en un ahorro de tiempo y energía. Se determina que los elementos más obsoletos son las comunicaciones, la interfaz, el diseño de algunos módulos y el lenguaje usado para el desarrollo. El contenido del siguiente capítulo se basa en proponer una serie de soluciones a los problemas detectados y en establecer la forma en la que van a ser

introducidas. El siguiente paso es seguir la planificación para llevar a cabo el desarrollo. De eso trata el sucesivo capítulo, en el que se describe como se ha pasado de la solución propuesta a la solución finalmente desarrollada, haciendo un pequeño inciso para hablar de las consecuencias de la planificación y el impacto del contexto social. El tramo final del trabajo se centra en hablar de la implantación del proyecto y de las pruebas realizadas para asegurar su correcto funcionamiento, cerrando con unas conclusiones en las que se hace hincapié en las diferencias y similitudes entre las metas propuestas inicialmente y los objetivos finalmente alcanzados, así como en los conocimientos empleados y adquiridos durante el desarrollo. Adicionalmente, se ha desarrollado un glosario de términos que se adjunta como anexo al final del proyecto por su gran extensión. Se recomienda darle una lectura rápida si no se poseen conocimientos técnicos del tema a tratar y tenerlo a mano para consultarlo en caso de olvidar el significado de alguna de las siglas empleadas, que son varias.

CAPÍTULO 2
Estado de la cuestión

En este capítulo se va a documentar que otros sistemas de funcionalidades iguales o similares al propuesto existen actualmente, o han existido en algún momento. De esta forma, al exponer distintas alternativas, se tratará de justificar los motivos que han llevado al proyecto a tomar el camino por el que ha sido llevado.

2.1 Contextualización

Para poder tratar el tema usando un léxico adecuado y ganar concisión, esta primera sección está dedicada a poner en contexto al lector haciendo un breve repaso sobre la terminología de más relevancia respecto al tema a tratar. Algunas de las fuentes bibliográficas [6] [7] empleadas para lograrlo han sido escogidas de entre la bibliografía de la asignatura «Agentes Inteligentes», también conocida como AIN, impartida en la Escuela Técnica Superior de Ingeniería Informática (ETSInf) de la Universitat Politècnica de València (UPV). Las versiones empleadas pueden encontrarse en la Biblioteca Digital de la UPV.

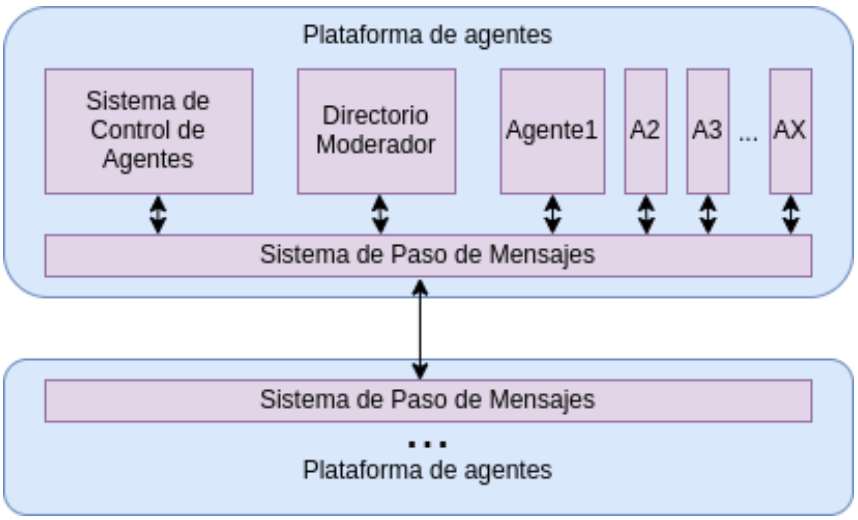
2.1.1. El plataforma de agentes estándar

El primer término a esclarecer es: «agente». Un agente es una entidad autónoma capaz de decidir qué acciones tomar para cumplir con el objetivo por el cual fue diseñado o contratado, sin la necesidad de estar preguntando qué debe hacer en cada momento. En palabras de otro autor, en el primer capítulo de su libro, Wooldrige [6] ya formula una definición para el término «agente inteligente», resumidamente sería así: un agente es un sistema computacional que cuenta con la capacidad de realizar acciones de parte de su usuario o propietario y de forma independiente. Cuando se emplea el adjetivo «inteligente» para hacer referencia a unos agentes es para remarcar que son sistemas computacionales, pero a nivel conceptual comparten la base de sus significado con cualquier otro tipo de agentes.

Cuando se juntan varios agentes para lograr una meta común, y estos se comunican y colaboran entre sí, hablamos de un sistema multiagente o MAS (del inglés «Multi-Agent System»). Esta definición también es muy relevante dentro del contexto del proyecto, dado que la plataforma base es un MAS. Para que un sistema de esta índole funcione es necesario que cuente con, a parte de los agentes, una construcción y un soporte de las comunicaciones, de manera que se puedan realizar consultas entre los distintos miembros de las organizaciones sobre el progreso en la resolución del problema, haciendo del trabajo una labor colectiva.

Estas definiciones no son arbitrarias, tal y como se enseña en «Ingeniería del Software», abreviada como ISW, no la forma de discernir distintos tipos de software es enmarcándolos dentro de un estándar. La FIPA es la organización de estándares para agentes y sistemas multiagente oficialmente aceptada por la IEEE el ocho de junio del año 2015. El objetivo de esta organización es dedicarse a promover la industria de los agentes inteligentes a base de desarrollar especificaciones que apoyen la interoperabilidad entre agentes y aplicaciones basadas en agentes. Es importante entender el estándar que esta organización ha desarrollado para MAS, ya que da las pautas de como debe ser una infraestructura en la que agentes, tanto software como hardware, puedan ser desarrollados. Es por este motivo que, a continuación, son presentadas las nociones básicas para facilitar la lectura de esta memoria.

Figura 2.1: Modelo Conceptual de Plataforma FIPA



El hecho de que sea un estándar abierto lo hace accesible a quien quiera usarlo, por supuesto las especificaciones son públicas [12]. Sin ir más lejos, en la figura 2.1 se muestra un esquema conceptual de lo que es, dentro del estándar FIPA, una plataforma de agentes[1] o AP. Los elementos que componen una plataforma FIPA son, pues:

- **Sistema de control de agentes**

  El AMS, del inglés «Agent Management System», es el elemento de gestión principal, desde el se controla los estados de la AP y los propios agentes. Es un elemento indispensable y único, por cada infraestructura de este tipo debe haber siempre uno y solo uno. El AMS supervisa el acceso a la plataforma y su uso. Este sistema ofrece servicios de páginas blancas a otros agentes, cada agente que desee formar parte del sistema debe pasar un proceso de registro para que el AMS le otorgue un identificador válido. El sistema almacenará estos identificadores en los que, entre otras cosas, guardará direcciones de transporte.

- **Directorio moderador**

  El DF, del inglés «Directory Facilitator», es un componente opcional pero, de ser implementado, debe cumplir con una serie de condiciones. Los DF ofrecen un servicio de páginas amarillas para otros agentes. Los agentes pueden realizar dos acciones sobre el DF principalmente, pueden registrar los servicios que ofrecen en él o realizar una consulta para conocer los servicios que ofrecen los otros agentes. Pueden

[1] La imagen ha sido adaptada al castellano desde el documento «FIPA Agent Management Specification» (número de documento: SC00023K) extraído del estándar público de la FIPA.

haber más de un directorio de esta clase por plataforma, pero todos ellos deben ser confiables, o lo que es lo mismo, deben garantizar que la información que ofrecen es lo más actual posible. Los casos en los que hay más de un DF, y estos realizan registros cruzados, se dice que forman federaciones.

- **Sistema de paso de mensajes**

  Un MTS, del inglés «Message Transport Service», es también obligatorio en cada plataforma. El MTS debe garantizar que los agentes pueden comunicarse entre sí vía paso de mensajes. Si el sistema no garantiza que los agentes puedan mantenerse en contacto, no podrán entablar relaciones sociales, ni mucho menos colaborar entre sí. Los mensajes que se envían por este sistema también están estandarizados, son los llamados mensajes FIPA-ACL. ACL son las siglas de «Agent Communication Language», una de las especificaciones más importantes del estándar es que los mensajes deben estar escritos usando un lenguaje ACL, esto es, un lenguaje de comunicación compartido por los agentes de la plataforma que asegure la comprensión del mensaje a cualquiera que lo reciba.

- **Agentes**

  Los agentes son procesos computacionales que implementan la autónoma y comunicativa funcionalidad de una aplicación, pero esto no es suficiente, para considerarlos agentes FIPA deben cumplir más requisitos. Para comunicarse los agentes se envían mensajes en ACL a través del MTS. Los agentes son los actores principales de la plataforma, estos combinan una o más cualidades que ofrecen como servicio dentro de un modelo de ejecución unificado e integrado. Cada agente tiene un propietario, este puede ser un usuario o una organización, entre otros y cada agente debe de contar con al menos una forma de representar su identidad. La noción de identidad es la etiqueta del agente, su propio «ID», que se emplea para poder diferenciarlo de otros agentes dentro del ecosistema. Los agentes emplearan sus identificadores para registrarse en el AMS y el DF para poder formar parte activa de la plataforma.

En sí misma, la AP es el soporte sobre el cual ejecutar los agentes, sin agentes no hay plataforma, pero sin la infraestructura los agentes no son funcionales. La fundación establece la siguiente definición para la AP: «La AP está formada por la máquina o máquinas, el sistema operativo, el hardware que da soporte a los agentes[2], los componentes de manejo de los agentes FIPA (DF, AMS y MTS) y los agentes».

2.1.2. Representación del conocimiento

La infraestructura con la que vamos a tratar la componen agentes argumentativos, estos son una especialización de los agentes anteriormente descritos. Un agente argumentativo es un agente que ha sido diseñado para cumplir un objetivo muy concreto: resolver un problema por medio de la argumentación. En la obra de Weston [8]m tal como proclama su título, se exponen las que son para el autor las claves de la argumentación. Weston, valiéndose de numerosos ejemplos para cada regla que dicta, trata de guiar al lector para convertirlo en un buen argumentador, enseña a diferenciar entre premisas y conclusiones, habla sobre el poder deductivo e indaga en numerosos tipos de argumentos. Sería conveniente que los agentes simularan estos comportamientos, no obstante, estos conceptos no pueden implementarse directamente, es necesario desengranar estas

» [2]El estándar se especifica también para MAS de agentes inteligentes físicos, lo cual no aplica para el caso, pues el proyecto a realizar es puramente digital. Para más información consultar la referencia [5].

ideas hasta obtener unas abstracciones que sí puedan implementarse con los medios con los que contamos, en este caso, los lenguajes de programación.

Para empezar es necesaria una forma de representar y almacenar lo que el agente sabe, la rama de la Inteligencia Artificial (IA) que estudia este ámbito se denomina Representación del Conocimiento (RC). La RC se puede enmarcar en tres preguntas fundamentales [2]:

1. ¿Cuál es el lenguaje de representación apropiado?

   Diseñar o, incluso, elegir un lenguaje de programación apropiado no es tarea sencilla, puesto que muchas de las funcionalidades deseadas son incompatibles entre sí. Un buen ejemplo son los lenguajes que permiten una elevada expresividad, porque estos suelen tener unas cualidades computacionales bastante pobres. Es por esto que, cuando la capacidad descriptiva del lenguaje es una prioridad, como mucho se suele optar por lenguajes que no llegan a la expresividad de una lógica de primer orden completa.

   Los llamados lenguajes conceptuales («concept languages») ofrecen la posibilidad de expresar el conocimiento sobre jerarquías de conceptos en forma de clases de objetos que comparten características. Al final, la peculiaridad que más diferencia a los sistemas basados en el conocimiento de otros sistemas computacionales de inteligencia artificial es que el conocimiento está explícitamente representado vía un lenguaje apropiado, y que de este conocimiento se pueden obtener nuevos datos mediante mecanismo de inferencia [3]. Lo cual nos permite enlazar con la siguiente pregunta.

2. ¿Qué inferencias deben extraerse de una base de conocimiento?

   El conocimiento almacenado por sí solo no es de gran utilidad. Lo que en realidad se pretende es usar el razonamiento sobre esta información para extraer las implicaciones que conlleva. Fijando la atención en los seres humanos, es común ver que las personas hagan deducciones que no tienen por qué ser ciertas y que son desechadas cuando aparece nueva información. Las inferencias no tienen por qué ser verdades absolutas, su utilidad reside en ampliar el conocimiento disponible y el valor de cada una oscila dependiendo de su certeza.

   Hay muchas y diversas formas de razonar, en la asignatura «Sistemas Inteligentes», o SIN, se imparten lecciones sobre algunas de ellas, principalmente el razonamiento inductivo, el difuso y el probabilístico. Por supuesto, hay muchas más, como por ejemplo el razonamiento abductivo, el razonamiento por analogía o el razonamiento no monótono. Este último encaja muy bien con el planteamiento del párrafo anterior, es por esto que es muy utilizado en RC.

   Dependiendo de la finalidad con la que se razona también podemos marcar una diferenciación. Cuando el razonamiento se emplea para discernir sobre la certeza de una creencia se está empleando un razonamiento teórico, propio de la filosofía. Por otro lado, si lo que se pretende lograr mediante el análisis y el uso de la razón es elegir entre todas las acciones disponibles, dada una situación y un contexto concretos, la más conveniente a realizar, entonces el término es razonamiento práctico. Ambos casos son claramente separables y obviamente necesitan de motores de inferencia con distintas cualidades.

   Para acabar con las formas de razonamiento, hay una clasificación más que interesa conocer, los sistemas basados en reglas frente a los sistemas basados en casos. Los motores de inferencia de los sistemas basados en reglas suelen tener una estructura similar, resumidamente: el motor revisa una a una todas las premisas o estados con

los que cuenta activando las reglas que tengan como precondición dichos estados, los estados visitados son apartados y cuando se activa una regla son creados otros nuevos. La ejecución termina cuando el problema converge, es decir, no quedan casos por visitar ni ninguna regla puede ser aplicada, puede darse el caso de no detenerse si no es interrumpido cuando se generan más casos de los que se rechazan. En cuanto a los sistemas basados en casos, el motor debe ser más complejo. Estos se basan en experiencias pasadas para obtener soluciones a situaciones presentes, por ejemplo, suponiendo el caso en el que se pretendiese decidir qué hacer cuando se funde una bombilla, el motor debería analizar la base de casos en busca de problemas similares ya resueltos para determinar cuáles fueron las soluciones que se aplicaron en su momento. Ha este tipo de razonamiento se le denomina basado en casos o CBR, del inglés «Case Base Reasoning»

La inferencia es una forma de obtener nuevo conocimiento, pero no es la única, la siguiente cuestión ronda este tema.

3. ¿Cómo podemos añadir nuevo conocimiento?

   Queda claro que las bases de almacenaje del conocimiento raramente son estáticas, es por esto que es necesario desarrollar métodos para modificar la información contenida en estas bases, ya sea añadiendo nuevos conceptos o retractando viejos. Ocasionalmente a este procedimiento se le llama revisión de las creencias («belief revision») y se vuelve especialmente complejo cuando la nueva información que se intenta añadir entra en conflicto con creencias con las que ya se contaba.

   Son muchos los posibles enfoques que se pueden dar ante estas situaciones. Algunos sistemas dan preferencia a la información más reciente, de esta forma cuando se añade nueva información que entra en conflicto con alguna premisa ya establecida, la antigua premisa es descartada, por otro lado tenemos el caso contrario, si priorizamos la información más antigua serían los nuevos conceptos los que se rechazarían. En un punto medio se puede optar por una actualización de las creencias, esto es, tomar la información que ya se conoce y ampliarla con la nueva información. Los conflictos entonces se conciben como cambios en el entorno percibido entre el momento en el que se obtuvo la primera información y el momento en el que se obtuvo la más reciente.

2.1.3. Argumentación

Esclarecida la situación actual en referencia a las formas de enfocar la representación del conocimiento es turno de hablar sobre argumentación. Para Weston, argumentar es «ofrecer un conjunto de razones o de pruebas en apoyo de una conclusión», claramente, las tres preguntas de antes son también claves a este respecto. Se necesitan pruebas almacenadas con las que argumentar, se necesita poder inferir de esas pruebas nuevos conceptos que usar en la defensa y se necesita poder asimilar los razonamientos que el oponente lance como argumentos. Son muchas las variantes a tener en cuenta, por esta razón es muy importante elegir bien las herramientas con las que trabajar.

En la asignatura «Lenguajes, tecnologías y paradigmas de la programación» (LTP), se enseña que cada lenguaje ofrece unas cualidades que lo hacen diferente al resto, el hecho de que coexistan varios lenguajes de programación al mismo tiempo puede verse como consecuencia al hecho de que distintas situaciones requieren distintos enfoques. En la primera cuestión se describen unas características que idóneamente tendría un lenguaje que vaya a ser usado para representar el conocimiento, o más concretamente, el formato que tendrán los conceptos almacenados. Estas características encajan perfectamente dentro del paradigma de la programación orientada a objetos (POO).

La segunda cuestión, sin embargo, no puede ser solventada con la misma solución, de hecho, sería mucho más conveniente para este caso tomar un lenguaje perteneciente al paradigma de la programación lógica [10] (PL), que es un tipo de programación declarativa. En este paradigma la lógica se usa como un lenguaje de programación; y un programa lógico es, básicamente, la expresión de una serie de relaciones representadas usando una notación lógica cuya base sea la lógica de predicados. En un lenguaje de programación puramente lógico, como Prolog, se cambia el más clásico concepto de computación como cálculo, en favor del pensamiento de computación como deducción.

Queda claro que no es posible tenerlo todo, en el momento de plantearse la solución será necesario definir las preferencias para poder determinar a que cualidad de las expuestas se le dará prioridad y cual será relegada a requerir de un mayor esfuerzo de implementación, arriesgándose incluso a tener que desprenderse de alguna cualidad.

2.1.4. Comunicaciones

Una vez dados los agentes argumentativos, lo único restante para constituir el sistema son las comunicaciones. En la descripción del estándar se ha anticipado el tipo de comunicaciones a emplear para este caso, sin embargo, es oportuno hacer una revisión de las alternativas para entender un poco mejor los motivos de la estandarización. Las opciones a considerar son muy distintas y el alcance de la infraestructura es crucial a la hora de decidir por cuál decantarse. Para este punto resulta muy útil remontarse a lo aprendido en asignaturas como «Tecnologías de los sistemas de información en la red» (TSR) y también «Concurrencia y sistemas distribuidos» (CSD). Además de las comunicaciones, en ellas también se tratan otros temas que pueden resultar muy apropiados para el diseño y desarrollo actual, como puede ser el despliegue de un programa en la red o la distribución de la carga computacional mediante la separación del trabajo en subtareas que se pueden ejecutar concurrentemente, sobre esto último se profundiza más en la asignatura «Computación Paralela» (CPA).

Pero ahora, como ya se ha introducido, la vista va a ser centrada en las comunicaciones casi exclusivamente. Desde una concepción un poco menos abstracta de lo que es un agente, podemos ver a estos entes como programas software que corren de forma individualizada en ámbitos de ejecución separados, en definitiva, a cada agente le corresponde un proceso de ejecución diferente. Esto tiene tanto lado positivo como negativo. Lo bueno es que es posible ejecutar el sistema de forma distribuida, esto es, en distintos lugares de la computadora e incluso en distintas máquinas. Por otra parte, lo malo es que la complejidad del programa se incrementa drásticamente, pues al ser procesos distintos no comparten la pila («stack») de ejecución, o lo que es lo mismo, un agente no tiene acceso al conocimiento de otro agente de forma sencilla.

Si ambos agentes se ejecutan en la misma máquina se pueden emplear ciertos mecanismos para eludir el último problema anteriormente expuesto, pues sí que es posible lograr que dos procesos distintos escriban y lean sobre la misma dirección de memoria. El montículo («heap»), al igual que la pila, es una abstracción empleada para representar y manejar cómo accede y modifica la memoria un programa. Mientras que la memoria a la que se puede acceder en la pila está limitada a un contexto de ejecución, el montículo tiene un alcance más general y es globalmente accesible. El truco, pues, es hacer que ciertas partes del montículo de los agentes coincidan para que estén accediendo a la misma parte de la memoria. A esta técnica se le llama comunicación por compartición de memoria y acarrea severas complicaciones, como la famosa condición de carrera.

Esta técnica es muy difícil de extrapolar a situaciones en las que los agentes no comparten la misma memoria física. Para estas ocasiones no suele quedar más remedio que

optar por la comunicación mediante envío de mensajes. Los mensajes son unidades de información que tienen, como mínimo: un receptor, un remitente y un contenido o cuerpo. Los sistemas de envío de mensajes requieren de un canal por el cual enviar los paquetes de información, en caso de compartir memoria el sistema operativo y la memoria principal conformarían el canal, pero en caso contrario sería necesario funcionar dentro de una red. Las comunicaciones de este tipo pueden ser diversas, las hay entre pares («peer to peer» o P2P), entre clientes y servidores, con o sin intermediario etc. Al igual que las comunicaciones por compartición de la memoria, estas presentan muchas dificultades con las que es preciso lidiar. En este caso son más si cabe, ya que hay que administrar muchas más variables como el estado de la red o las colas de envío.

A lo largo de esta sección se han puesto en contexto el panorama de los MAS, la investigación en el campo de la representación del conocimiento, el ámbito de la argumentación y las comunicaciones entre programas. El contenido de la siguiente sección se soporta sobre todas estas aportaciones para definir con claridad el punto de partida del proyecto.

2.2 La infraestructura base

En esta sección se va a hablar de la infraestructura de la que se parte para el desarrollo de este proyecto, esta es la presentada por Jordán et al., por lo que la información a continuación ha sido extraída principalmente del artículo en el que es propuesta [1] y las referencias del mismo. Como es común en los trabajos de investigación, el proyecto a describir también está construido sobre desarrollos e investigaciones previos, la infraestructura que constituyó las bases de este fue la propuesta por Heras et al. [4]. Los cimientos eran sólidos, puesto que se partía de una infraestructura que ya permitía la argumentación dentro de un MAS en el cual los agentes eran capaces de intercambiar argumentos entre ellos, siempre sujetos a un contexto social.

2.2.1. Aproximación a la argumentación mediante bases de casos

En el mencionado artículo [1] que inspira el título de esta subsección se presentaban ya varios conceptos importantes. Este trabajo comienza por mostrar la forma de representar el conocimiento que se va a emplear: los agentes contarán con bases de casos en las que tendrán los llamados «recursos del conocimiento» (KR del inglés «knowledge resources»). Esta ontología habilita un entendimiento común sobre los conceptos de la argumentación y permite que diversos tipos de agentes puedan enzarzarse en procesos argumentativos entre ellos. Seguidamente, en el artículo se explica que, tomando como base los KR propuestos, se ideó un proceso de razonamiento que, de ser seguido por los agentes, los dotaría de la capacidad para llegar a acuerdos en los que se tiene en cuenta tanto su experiencia en la argumentación, como su contexto social. Este proceso también les concedería el poder aprender de sus discusiones pasadas, ayudando a mejorar sus habilidades en la argumentación en base a la experiencia. Estos elementos combinados forman una infraestructura para la argumentación basada en casos cuya finalidad es alcanzar acuerdos dentro de sociedades de agentes.

Los KR utilizados para RC aquí pueden englobarse en dos: los casos de dominio[3] (DC, del inglés «domain cases») y los casos de argumentación (AC, del inglés «argumentation cases»). Los primeros representan problemas ya resueltos previamente y sus respectivas

[3]El término «dominio» se usa recurrentemente en este documento y, salvo indicios de lo contrario, siempre por su acepción: «ámbito real o imaginario de una actividad».

soluciones, mientras que los segundos representan argumentaciones finalizadas previamente y su correspondiente resultado. Otros KR relevantes a explicar son los tipos de argumentos, ataque y soporte, y los conjuntos de apoyo (SS del inglés «Support Set») que son los elementos que se emplean para reforzar el argumento.

Previamente se ha dado una definición general de lo que es un MAS, ahora toca especificar un poco más. Este sistema en concreto lo conforman sociedades que los autores definen como: «un conjunto de agentes que actúa un conjunto de roles, observa un conjunto de normas y un conjunto de relaciones de dependencia entre roles y usan un lenguaje de comunicación para colaborar y alcanzar las metas del grupo». Esta definición añade nuevas implicaciones que serán descritas más adelante, pero ahora es el momento de hablar de la evolución del sistema hacia la búsqueda de la solución a un problema en base a la resolución de conflictos.

En un MAS argumentativo de ámbito abierto, a saber, en el que es posible conocer los argumentos y posiciones del resto de agentes, es corriente que se generen conflictos. Los participantes de la conversación generan argumentos para apoyar su posición tomada con respecto al problema y es muy común que estos argumentos no sean compatibles con los argumentos de los otros participantes, produciéndose así una discrepancia. Estas discrepancias se deben resolver mediante diálogos argumentativos en los que los participantes intercambian una serie de argumentos con el fin de promover o invalidar distintas posiciones. Con la intención de dotar a los agentes con la capacidad para realizar todas estas acciones, cada una las entidades que vayan a participar de la argumentación cuentan con dos bases de casos distintas: una base con DCs con casos previos y sus soluciones y una base de ACs con los casos de conversaciones ya terminadas y el resultado de estas. Dichas soluciones se emplean para generar las posiciones (nuevas posibles soluciones) que defender, así como argumentos de apoyo para defenderlas, sin olvidar los argumentos de ataque para desafiar las posiciones de otros agentes.

La estructura de los DC consiste en un conjunto de cualidades descriptivas del problema resuelto y la solución aplicada. Los AC son más complejos que estos y para entenderlos es necesario saber como son los argumentos que se intercambian los agentes. Pues bien, estos argumentos son enviados como una tupla de tres elementos: el primero de ellos es la conclusión, el segundo es el valor del argumento (p. ej. ahorro, rapidez, calidad) y el tercer es un conjunto de apoyo. Los SS ya han sido mencionados como los elementos que acompañan al argumento para darle soporte, pero para una mejor comprensión requieren de una descripción más detallada. Un SS se compone de los siguientes elementos:

- Premisas: Cualidades del argumento que encajan con cualidades del problema. Son usadas para describir el problema y pueden relacionarse con la totalidad de las características del problema o con una parte de estas. Son utilizadas por los agentes para extraer de la su base de DCs casos similares al actual.
- Casos de dominio: Casos que representan problemas pasados y ya resueltos que comparten similitudes con algunas de las propiedades que definen el problema.
- Casos de argumentación: Casos que representan experiencias argumentativas pasadas del agente junto a su desenlace para poder extraer, de entre ellos, la mejor posición a adoptar en base al contexto del problema actual y la experiencia adquirida en debates previos.
- Premisas distintivas: Premisas que pueden impedir que otro agente genere una conclusión válida para un argumento a partir de un KR. Cuando un argumento propuesto es atacado se emplean estas premisas que son extraídas de DCs que proponen soluciones distintas a la propuesta por el argumento que está en duelo.

- Contra ejemplos: DCs cuyas descripciones comparten rasgos con la del caso, y por tanto son similares, pero que tienen conclusiones distintas.

Los agentes pueden generar dos tipos de argumentos según la situación, pero ambos comparten la misma estructura, eso sí, dependiendo del caso habrá elementos del SS que no serán empleados y por lo tanto se enviarán vacíos. El primer tipo son los argumentos de ataque y el segundo son los argumentos de apoyo. Para explicar a qué tipo de momento corresponde cada tipo de argumento hay que revisar los componentes que conforman un intercambio de argumentos entre agentes. Por un lado tenemos al oponente («opponent») que es el participante que decide desafiar un argumento y por otro al defensor («proponent») que tiene que defender su postura. Ambos pueden conversar porque pertenecen al mismo grupo («group») social. Bien, por defecto, los agentes no tienen que exponer ningún argumento para justificar su posición, sin embargo, si la posición es cuestionada, el defensor podrá mostrar su argumento para que el oponente pueda atacarlo con el suyo propio. Si el defensor decide apoyar su posición, entonces generará uno de los llamados argumentos de apoyo. Este argumento vendrá acompañado del SS con las características (premisas) que describen parcial o totalmente el problema y que encajan con los RCs (DCs) que ha empleado para inferir la posición que defiende. En caso de que después de esta acción el oponente decida desafiar el argumento que el defensor clama como justificación, deberá generar un argumento de ataque. Los argumentos se pueden esgrimir cargándolos con premisas distintivas o contra ejemplos, por lo que siempre dependen de los elementos del conjunto de apoyo que acompaña al argumento del defensor. Para ser más específicos, dependen de la justificación de las siguientes maneras:

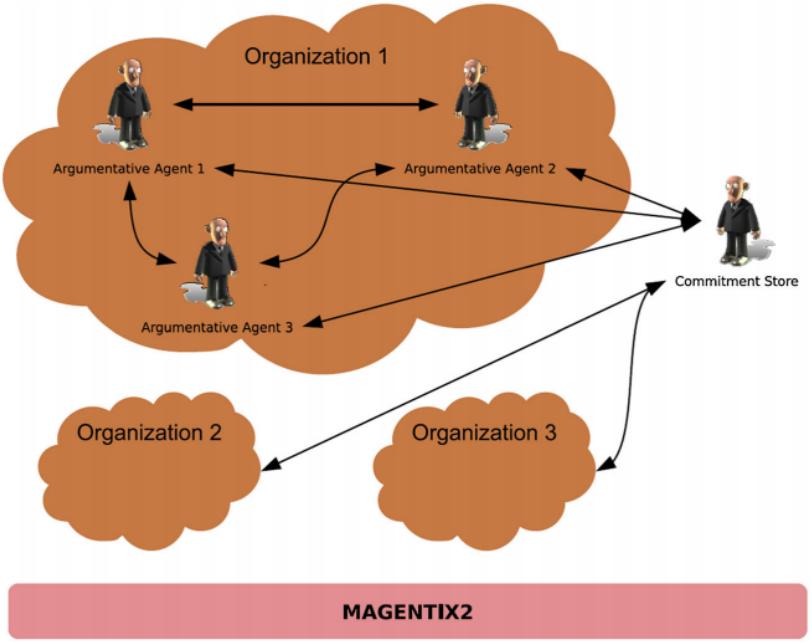
- La justificación es un conjunto de premisas: en este caso el oponente, si se encuentra en una situación privilegiada porque, por ejemplo, conoce información añadida sobre el problema que el defensor no expone, puede emplear esas premisas distintivas como arma metafórica cuando sirven para justificar su propia posición. También puede aprovechar si el problema está implícito dentro de algún caso que haya empleado para generar su propia posición, lanzando un argumento con este caso como contra ejemplo.

- La justificación es un caso de dominio o de argumentación: para esta situación, el agente podría recabar en su base de casos de dominio en busca de contra ejemplos para atacar a la defensa con un argumento que los contenga. De no ser así, también podría intentar generar un argumento basándose en premisas distintivas de entre las que el conoce que invaliden la justificación del defensor.

2.2.2. Soporte para el marco argumentativo

En el marco argumentativo presentado se han especificado los recursos de conocimiento que la infraestructura completa implementa, la sección a continuación define la parte de la misma sobre la que se sostiene el marco argumentativo. Para sintetizar, se va a proceder a exponer el sistema construido alrededor del marco expuesto en la subsección anterior, que ofrece herramientas para desarrollar agentes con dotes para la argumentación, incluidas las capacidades comunicativas y el protocolo de argumentación en MAS de ámbito abierto. El sistema también ofrece herramientas para crear sociedades, contexto social incluido, mediante la asignación de problemas a grupos compuestos por agentes, con las cualidades argumentativas descritas, que colaboran entre si con el fin de acordar una solución final.

Figura 2.2: La infraestructura propuesta por Jordán et al.



En la figura 2.2 tenemos una representación gráfica directamente extraída del artículo [1] de presentación de la infraestructura. En ella se aprecian los componentes de la plataforma y las relaciones entre los mismos. En ese artículo se diserta extensamente el funcionamiento y los detalles de implementación de la AP, pero hay muchos de esos conceptos que no son del interés de este trabajo, pues coparían más de lo necesario. El desarrollo de esta subsección se va a limitar a explicar lo necesario para comprender el resto de la memoria, aún con todo, es altamente recomendable acudir a la fuente si se desea profundizar. El funcionamiento va a ser descrito fijando la atención en sus componentes principales y cómo son estos en relación al estándar FIPA:

- **Comunicaciones**

  La infraestructura ha sido implementada usando la plataforma Magentix2[4]. Magentix2 es una plataforma abierta para el desarrollo de sitemas multiagente con soporte para establecer organizaciones (grupos, reglas y normas). Esta plataforma ofrece diversas herramientas para el manejo óptimo y seguro de MAS y es empleada para dar soporte a las comunicaciones. La figura 2.2 representa una visión más abstracta del modelo que, por ejemplo, el visto en la explicación del estándar FIPA, no obstante la plataforma sí funciona de acuerdo al modelo estándar. El esquema que vemos en la imagen representa la plataforma completa, las organizaciones no son otras plataformas, sino abstracciones que permite crear Magentix2 para crear sociedades de agentes dentro de la misma plataforma. Las flechas que relacionan a los agentes indican que pueden establecerse comunicaciones entre ellos, pero está implícito que estas comunicaciones requieren de una utilización del MTS, Magentix2 nuevamente, por parte de los agentes. En conclusión, las comunicaciones se acogen al estándar.

- **Agentes argumentativos**

  Los agentes argumentativos son el componente más importante y complejo de toda la infraestructura. Estos cuentan con capacidades argumentativas para entablar

[4] http://gti-ia.dsic.upv.es/sma/tools/magentix2/index.php

diálogos en los que, por medio de la argumentación, intentan alcanzar un acuerdo sobre cuál es la mejor solución a un problema. Sus principales componentes son:

- Módulo CBR del dominio: Módulo empleado para almacenar casos que contienen conocimiento del dominio sobre problemas resueltos previamente.
- Módulo CBR de la argumentación: Módulo empleado para almacenar casos que contienen conocimiento sobre experiencias argumentativas pasadas.
- Proceso de control de la argumentación: Este componente implementa el cómo las posiciones, los argumentos de apoyo y los argumentos de ataque son generados por los agentes a partir de sus propios recursos del conocimiento y sus dos módulos CBR.
- Protocolo argumentativo: Este protocolo define las acciones que los agentes argumentativos deben seguir para participar de los diálogos. Este componente es de gran relevancia para el desarrollo de este trabajo de fin de grado, por lo que más adelante se profundizará más.

Cada agente cuenta con su propio identificador exclusivo y es registrado por el directorio moderador de la plataforma, en este caso interpretado por la «commitment store», CS de ahora en adelante. Para entablar conversación con otros agentes se envían mensajes escritos en OWL2[5] (el ACL en esta plataforma) y pueden enviarlos gracias a que los agentes están implementados como una extensión de agente conversacional de Magentix2 (la clase «CAgent»). El propietario de los agentes es el usuario, programa o la organización que configura y ejecuta el programa, pues trabajan para resolver el problema que ha planteado. En retrospectiva es correcto decir que los agentes argumentativos de esta AP se adhieren al estándar FIPA.

- **Directorio moderador**

  El componente del sistema que hace las veces de DF es denominado «commitment store». El CS almacena la información de todos los agentes participes del proceso de resolución del problema (páginas blancas). Entre esta información se encuentran los identificadores de los agentes, los diálogos en los que participan, sus posiciones y sus argumentos. Si un agente lo desea puede realizar una consulta a este recurso para obtener toda la información acerca de los diálogos en los que está involucrado, podría decirse que implementa también el servicio de páginas amarillas, pero todos los agentes argumentativos ofrecen los mismos servicios, por lo que el resultado es una mezcla. En la figura 2.1 aparece representado como un agente, esto es porque el CS es también una extensión del agente conversacional de Magentix2. Esta decisión de diseño se tomó para simplificar las comunicaciones entre agentes evitando añadir más protocolos, que aumentarían irremediablemente la complejidad del sistema. El CS cuenta con algunas de las características que la FIPA dicta que debe implementar el DF de una AP, además, debido a que por su diseño todos los cambios han de pasar por él, es una fuente confiable pues siempre tiene la información actualizada, aún así, por sí solo no cumpliría con el estándar.

- **Sistema de control de la ejecución**

  El AMS está dividido en dos partes, una está en el programa principal, que prepara la ejecución del resto de componentes y la otra se encuentra en el CS. En la plataforma que propone el equipo de investigación no se plantea que agentes de otras plataformas pretendan interactuar con los suyos, en este sentido el AMS es menos complejo. Esto no implica que sea una infraestructura aislada, externamente

[5] OWL2 (https://www.w3.org/TR/owl2-overview/ es la abreviación de «OWL 2 Web Ontology Language», un lenguaje empleado para representar ontologías.

es posible configurar los problemas a resolver, los agentes y las sociedades, adicionalmente es posible obtener los resultados del consenso tras los procesos argumentativos. Al arrancar el sistema, el programa principal lanza a ejecución los agentes, incluido el CS. Posteriormente la gestión de los agentes pasará a estar regulada por el DF y por ellos mismos. Cuando se establezca que el proceso argumentativo a finalizado, los agentes serán llamados a concluir su ejecución de acuerdo a su protocolo y el resultado de la deliberación colaborativa será expuesto. Como ya se ha afirmado antes, la parte del AMS que maneja la información de los agentes registrados se encuentra en el CS. En este módulo se controla la ejecución de todos los diálogos, si un agente desea entablar un diálogo debe realizarle una petición que será procesada, almacenando toda la información necesaria. Como agente, el CS también tiene un protocolo de comportamiento, aunque no es de interés comentarlo extensamente ahora, es un protocolo sencillo por el cual atiende las peticiones de los agentes argumentativos y resuelve sus consultas. En relación al estándar, está es la parte que no se adapta tan plenamente a él. Las comunicaciones entre agentes de la plataforma y agentes de otras plataformas externas no está plenamente soportadas, desde afuera es posible acceder al resultado de las conversaciones, pero los agentes solo están preparados para comunicarse entre los de su mismo tipo y el CS.

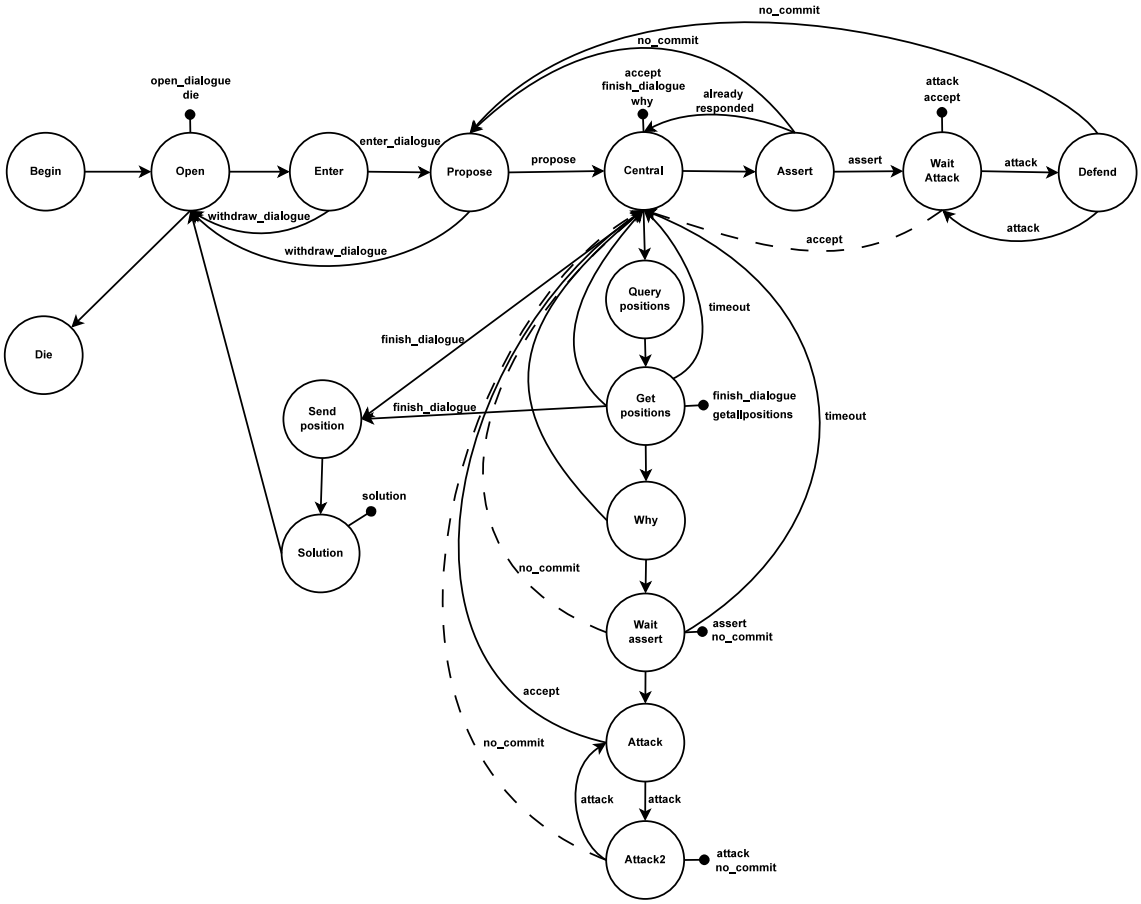
- **Protocolo argumentativo**

  El sistema funciona correctamente gracias a implementar un buen mecanismo para el intercambio de conocimiento. Este mecanismo es el que emplean los agentes y el CS para pasarse información, de aquí la importancia de emplear un ACL común. OWL2 es empleado para codificar los casos contenidos en las bases de casos del CBR de dominio (casos de dominio) y del CBR de argumentación (casos de argumentación), el formato empleado para almacenar dichos casos fue diseñado por el propio equipo. De esta forma, agentes dispares que tengan en común este lenguaje pueden intercambiarse soluciones y argumentos extraídos de sus bases de casos vía mensajes FIPA-ACL, mensajes que, gracias a haber usado ontologías, son fácilmente comprensibles.

  Toda esta construcción no podría ser aprovechada si no fuese por que los agentes siguen un cierto protocolo a la hora de argumentar, a partir de ahora referido como protocolo argumentativo. Este protocolo es altamente complejo, tal y como se muestra en la figura 2.3. La imagen muestra, específicamente, el esquema de una máquina de estados finitos, o FSM (del inglés «Finite States Machine»), representativo del protocolo argumentativo. En asignaturas como «Introducción a la Programación de Videojuegos», o IPV, se estudia este tipo de estructuras para emplearlas en personajes no jugables, también conocidos como PNJs, o NPCs (del inglés «Non-Player Character»). Los PNJs son, de hecho, un tipo de agentes software, por lo que es normal que se empleen mecanismos similares para ambas situaciones. Las FSM son mecanismos de escalable complejidad empleados para definir qué comportamientos se deben adoptar en función de los hechos que hayan sucedido previamente. El ente está siempre asociado a un estado, cada estado tiene una serie de enlaces a otros estados que representan transiciones condicionadas. Si se cumplen las condiciones de transición el ente pasa a tener asociado el estado al otro lado del enlace.

  En resumen, la figura 2.3 muestra la máquina de estados que define el protocolo argumentativo de un agente, este protocolo describe el comportamiento del agente cuando este se encuentra en un diálogo, así como el proceso que sigue para proponer sus posiciones, defenderlas y atacar las de sus oponentes. En el esquema, los estados cuya etiqueta es un acto performativo representan estados de espera. En estos estados el agente entra en un modo de «stand by» a la espera de recibir un mensaje por parte de otro agente o del CS. Además están las aristas de líneas

Figura 2.3: Protocolo de actuación de los agentes argumentativos



discontinuas, estas conexiones representan la transición entre estados cuando estos mensajes, conteniendo el acto performativo, son recibidos. Así pues, las transiciones entre estados están sujetas a una serie de reglas que especifican qué declaraciones pueden hacerse y bajo qué circunstancias dentro del diálogo. Todas las reglas y detalles del funcionamiento están descritos en el artículo [1], pero para el ámbito de este trabajo es suficiente con conocer lo ya mencionado y lo que viene a continuación. Cuando el agente se encuentra en un estado espera recibir mensajes con actos performativos específicos para el estado, sin embargo, hay tres actos que puede recibir en cualquier momento de su ciclo de vida una vez ha sido registrado. Estos actos son:

- FINISHDIALOGUE: Se le informa al agente que debe ejecutar, de tenerlas, las acciones programadas previas a abandonar un diálogo.
- DIE: Se le informa al agente que debe poner fin a su ejecución.
- GETALLPOSITIONS: Se le realiza una petición al CS para que le proporcione una lista con las posiciones disponibles en un cierto momento del diálogo. Este mismo acto performativo es usado por el CS para emitir la respuesta a la petición.

Por último, toda FSM tiene un único estado inicial, el que se le asocia al comienzo de su ejecución, y al menos un estado final, los que se le asocian cuando su ejecución va a concluir. En el protocolo argumentativo el estado inicial es el llamado «BEGIN», mientras que su único estado final es el llamado «DIE».

Con esto concluye el capítulo del estado de la cuestión, en él se han presentado el contexto en el que fue creada la infraestructura base a partir de la cual se ha realizado el trabajo de fin de grado y también se ha presentado dicha plataforma. Con estas nociones adquiridas se da paso al siguiente capítulo, la parte de la memoria en la que se realiza una evaluación del conflicto a resolver.

CAPÍTULO 3
Análisis del problema

En este capitulo se realiza un análisis del estado del proyecto teniendo en cuenta lo expuesto en el estado de la cuestión, pero también poniendo la vista en el futuro próximo. Esta evaluación desembocará en la necesidad de aplicar una serie de cambios y adiciones, la cual será el foco de atención del desarrollo y la razón de ser de este trabajo. Antes de continuar es necesario hacer una diferenciación entre el concepto de la AP que se va a tomar como base del desarrollo, y que ha sido presentado previamente, y su implementación[1]. La primera corresponde al ámbito teórico, son los conceptos con los que Jordán et al. [1] pretenden resolver la necesidad que se habían planteado. Una vez concebida la forma es necesaria una prueba empírica de que es correcta, de ahí viene la segunda. Para poder realizar las pruebas prácticas es necesaria una implementación del concepto. En las próximas secciones se hará hincapié en esta diferenciación. Y como última puntualización antes de continuar, en este capítulo se hacen muchas referencias a la infraestructura que se ha empleado como base, para hacer menos pesada la lectura cada vez que sea referida, de este punto en adelante, será como «la plataforma», «la infraestructura» o equivalentes a secas.

3.1 Una base consistente

Esta sección está dedicada a narrar las partes de la infraestructura que se conservan perfectamente actuales y no necesitan de ninguna revisión. Es una sección relevante, ya que, cuando hay menos tareas que realizar, se dispone de más tiempo para dedicar a las realmente importantes; lecciones como esta se enseñan en la asignatura «Gestión de Proyectos» (GPR), que junto a ISW aportan información y metodologías que han sido claves en el desarrollo de este y los siguientes capítulos.

Primeramente está el diseño y conceptualización de la AP, que, afortunadamente, no requiere ningún cambio; el trabajo dedicado a su concepción es suficiente para sostener todas las adiciones que se van a plantear, como se verá más adelante una vez estas sean planteadas. En la sección en la que ha sido presentada se ha puesto en comparación con las especificaciones dictadas por el estándar FIPA, y cumple con todas ellas. Como ya se ha avisado, en tal sección se trata indistintamente la plataforma teórica de la plataforma práctica, pero tal y como se ha explicado, son conceptos diferentes y, de hecho, en el caso de la versión implementada falta por cumplir con el requisito de permitir la interacción directa entre agentes de esta y otras plataformas. Pese a todo, y por el mismo motivo que esta habilitación no se implementó en su momento, no va a ser necesaria, pues el funcionamiento de la infraestructura puede ser probado sin ella. La plataforma fue im-

[1] https://github.com/gti-ia/magentix/tree/master/src/main/java/es/upv/dsic/gti_ia/argAgents

plementada para probar que las ideas concebidas estaban en lo cierto y en eso cumple perfectamente, porque se han demostrado como posibles y funcionales.

Los productos software libres y abiertos necesitan disponer de una documentación clara y útil, así como una buena explicación de la funcionalidad del sistema. Esto es, si cabe, más importante aún en los proyectos de investigación. Al respecto de este asunto hay poco que criticar, el artículo [1] en el que es propuesta la AP tiene todos los detalles sobre la funcionalidad que se puedan necesitar, y gracias al uso de herramientas de generación automática de la documentación, Javadoc[2] en este caso, esta es muy completa y accesible; el único punto mejorable sería la corrección de ciertos fallos tipográficos. Por lo tanto, seguir empleando la misma metodología que hasta este momento con los cambios venideros es una decisión razonable.

El hecho de contar con una base robusta facilita enormemente las labores de ampliación, puesto que no es necesario estar continuamente añadiendo parches a los cimientos de la infraestructura para evitar que se venga abajo. Gracias a que la plataforma de base es robusta, va a ser posible realizar el desarrollo del proyecto sin necesidad de aplicar grandes cambios y permitiendo una implementación más directa de las funcionalidades.

3.2 Obsolescencia y limitaciones

Explicando el origen de la plataforma se menciona que esta se fue revisando y actualizando hasta el año 2015. Cuando un programa deja de actualizarse corre el riesgo de quedarse obsoleto, son muchas las veces en las que es necesario recurrir a la emulación para ejecutar aplicaciones que ya no funcionan en los sistemas actuales. No es el caso de la infraestructura, al menos por el momento, pero es conveniente hacer una revisión a este respecto para asegurarse de que una vez el trabajo esté hecho pueda ser accesible a la mayor cantidad de usuarios posible. Ese es la dirección en la que se va a enfocar en esta sección; en los siguientes párrafos se van a analizar los efectos del tiempo en la infraestructura, pero no solo en términos de obsolescencia, también se tendrán en cuenta otro tipo de limitaciones derivadas.

3.2.1. Comunicaciones

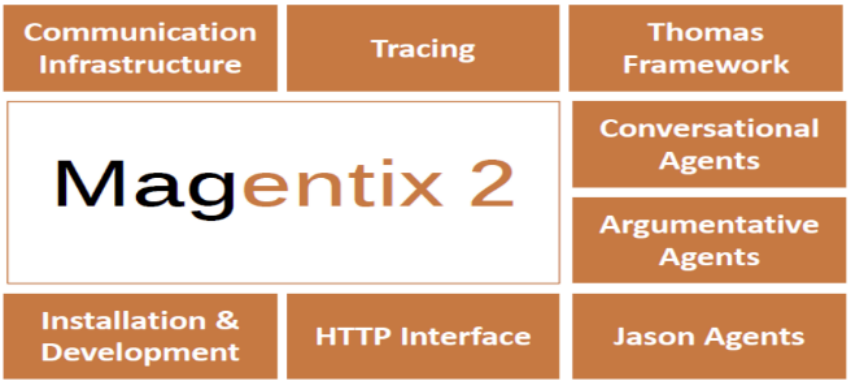
El paso de mensajes y las comunicaciones en general dentro de la plataforma están gestionadas mediante el uso de Magentix2, lo cual presenta tanto beneficios como limitaciones. Entre los beneficios se encuentran el poder funcionar con agentes programados en cualquier lenguaje de programación, el soporte para múltiples arquitecturas de agentes y la transparente interacción entre los agentes respecto al desarrollador. Por otro lado, también son varias las limitaciones, muchas de ellas derivadas de la necesidad de cumplir con un modelo base fijo de agente. Un agente Magentix2 tiene tres métodos principales: «init» se ejecuta al principio del ciclo de vida del agente, «execute» es el método que contiene el código principal del agente y «finalize» se ejecuta cuando el agente entra en el final de su ciclo de vida. De estos tres, sólo el segundo es obligatorio, por lo que, no es estrictamente limitante, sin embargo, no sucede lo mismo con los estados.

Volviendo a la figura 2.3 y los estados etiquetados como actos performativos, se pueden explicar mejor estas limitaciones. Pero antes de eso, un breve inciso: cuando la versión 1.0 de la infraestructura fue liberada, esta pasó a ser parte del proyecto Magentix2. Como se muestra en la figura 3.1, extraída de la página de Magentix2[3], los agentes ar-

[2] http://www.gti-ia.upv.es/sma/tools/magentix2/archivos/javadoc%20v2/index.html?es/upv/dsic/gti_ia/core/ACLMessage.html

[3] http://www.gti-ia.upv.es/sma/tools/magentix2/index.php

Figura 3.1: Esquema de contenidos de la plataforma Magentix2



gumentativos están listados como uno de sus componentes principales. Retomando la cuestión de las limitaciones, al extender el modelo de agente argumentativo del agente conversacional de Magentix2, se produce una dependencia sobre la forma en la que se pueden implementar las comunicaciones, siendo esta dependencia la causa de los estados de espera de los que se ha hablado en el pasado capítulo.

Las esperas son necesarias dado que las transiciones de unos estados a otros requieren del paso de mensajes. Efectivamente, esta limitación está ligada a lo comentado en el párrafo anterior, Magentix2 facilita en gran medida las comunicaciones, pero es necesario atenerse a las consecuencias de sus exigencias en la implementación. Adicionalmente, esta barrera de implementación conduce a un aumento en la complejidad de la FSM y, por ende, en la del protocolo argumentativo. Realmente el esquema podría simplificarse ya que las acciones realizadas en los estados de espera por exigencias del diseño podrían ser relegadas a los estados de los que parten, al no ser necesario el mencionado paso de mensajes.

En conclusión a este análisis, puede confirmarse que el sistema de comunicaciones de la plataforma actualmente es cómodo, salvo para definir protocolos sencillos. Aún así, tiene limitaciones en el diseño que pueden repercutir a la hora de escalar los protocolos que requieran de este sistema, a la vez que puede ser complicado de combinar con agentes de otras plataformas. Por lo tanto, la revisión de las comunicaciones debe ser un punto a tener en cuenta en futuras versiones, principalmente si se desean añadir nuevas funcionalidades o ampliaciones de los protocolos actuales.

3.2.2. Agentes impersonales

Según la Real Academia Española (RAE) el adjetivo «impersonal» se usa para referirse a un concepto «que no tiene o no manifiesta personalidad u originalidad». En la versión de la plataforma más reciente, la impersonalidad de los agentes argumentativos es una realidad. A continuación se va a justificar esta afirmación y a argumentar el por qué es un problema.

En la conceptualización del agente argumentativo se dan las pautas de cómo debe estar construido, qué funcionalidades tiene que implementar y a qué protocolo debe adherirse, no obstante, estas indicaciones no implican que solamente pueda existir un tipo de agente argumentativo, es todo lo contrario, partiendo de esa base se da pie a que haya agentes argumentativos que exhiban comportamientos distintos, como una mayor tendencia a aceptar las posiciones de los otros agentes, o actitudes escépticas hacia agentes que haya propuesto posiciones previamente que, por lo general, hayan sido poco aceptadas.

Las implementaciones de los agentes de esta infraestructura teórica en Magentix2 son solo dos, una del CS y otra de agente argumentativo. Esto implica que en todos los agentes argumentativos operantes, fruto de las ejecuciones realizadas con esta implementación, como los ejemplos disponibles[4], son instancias de la misma clase de agente argumentativo, es decir, todos presentan el mismo comportamiento impersonal, cadente de originalidad. En la versión *alpha* de la infraestructura se realizaron una serie de pruebas con agentes de distintas personalidades, pero su elaboración era muy costosa y poco elegante. Estas variantes de agente argumentativo se hicieron a base de tomar el código fuente del agente argumentativo original y modificar ciertas partes, comentando algunas otras. Esta es una buena solución para realizar comprobaciones puntuales, pero no es adecuada para una versión final, siendo que el procedimiento es aparatoso y muy poco intuitivo. La mejor solución en esta dirección sería tomar el agente argumentativo mencionado y extender de él, sobreescribiendo los métodos que interesase cambiar. Puede parecer una buena opción, pero es un uso artificial y sin premeditación de la genericidad. Es corriente en estas situaciones que al reemplazar ciertos métodos y conservar otros se produzcan errores causados por dependencias funcionales establecidas a raíz de partir de un código que no había sido planteado para tales propósitos.

En las versiones más recientes existe una forma extra de romper esta impersonalidad, al menos hasta cierto nivel. Esta técnica se basa en definir un factor de soporte o SF (del inglés «Support Factor») compuesto por seis valores normalizados o pesos, que sirven para ponderar ciertas actitudes del agente permitiéndole adoptar diferentes tácticas argumentativas. Este sistema es incorporado directamente de la infraestructura para argumentación definida en el artículo [4] y que, ya se ha comentado, fue la base en su momento de la infraestructura en la que se apoya ahora este proyecto. Dichos pesos está explicados con mucho detalle en el mencionado artículo, aquí se va a presentar un resumen de las tácticas que los relacionan. Teniendo en cuenta que en la nomenclatura la «w» hace referencia a que el valor es un peso normalizado, y que la fórmula para el cálculo del factor de soporte es la que se presenta a continuación, se muestra el resumen de estas tácticas seguidamente después.

$$SF = ((wPD * PD + wSD * SD + wRD * (1 - RD) + wAD * (1 - AD) + wED * ED + wEP * EP))/6$$

- Táctica persuasiva: El grado de persuasividad, o PD (del inglés «Persuasiveness Degree»), es un valor que representa el poder persuasivo estimado de un argumento basándose en cómo de persuasivo resultó ser un caso de argumentación con la misma descripción y conclusión al problema en el pasado. Cuando se emplean tácticas persuasivas el peso de la persuasividad (wPD) es elevado, dando mayor relevancia a argumentos con un elevado PD.
- Táctica de maximización del soporte: El grado de soporte, o SD (del inglés «Support Degree»), es un valor que representa una estimación aproximada de la probabilidad de que la conclusión del argumento actual vaya a ser aceptada al final del diálogo. Se calcula a partir del ratio de casos de argumentación aceptados de entre todos los casos de argumentación con la misma descripción y conclusión que el argumento actual, que hayan sido extraídos del CBR de argumentación. Cuando se emplean tácticas de maximización del soporte el peso del soporte (wSD) es elevado, dando mayor relevancia a argumentos con un elevado SD.
- Táctica de minimización del riesgo: El grado de riesgo, o RD (del inglés «Risk Degree»), es un valor que estima de forma aproximada el riesgo a ser atacado al que

[4] https://github.com/gti-ia/magentix/tree/master/src/examples/src/main/java/Argumentation_Example

se expone un argumento de ser propuesto en función de los ataques ya recibidos sobre argumentos con la misma descripción y conclusión que el problema actual. Se calcula a partir del ratio de argumentos atacados de entre todos los casos de argumentación con la descripción y conclusión iguales a las del argumento actual, que hayan sido extraídos del CBR de argumentación. Cuando se emplean técnicas de minimización del riesgo el peso del riesgo (wRD) es elevado, dando mayor relevancia a argumentos con un menor RD.

- Táctica de minimización del ataque: El grado de ataque de un argumento, o AD (del inglés «Attack Degree»), es un valor que, de forma similar al RD, evalúa las posibilidades de que un argumento pueda ser atacado en base a argumentaciones pasadas. La diferencia con el RD es que el AD es calculado teniendo en cuenta el número de ataques que recibieron dichos casos de argumentación con los que el argumento comparte descripción y conclusión, para posteriormente realizar una transformación linear con el fin de normalizar el valor. En tácticas de minimización del ataque, el peso de ataque (wSA) es elevado, dando mayor relevancia a argumentos con un menor AD.

- Táctica de maximización de la eficiencia: El grado de eficiencia, o ED (del inglés «Efficiency Degree»), es un valor que ofrece una estimación sobre el número de pasos que puede tardarse en alcanzar un acuerdo, basada en experiencias argumentativas pasadas. Se calcula en base a la profundidad de los nodos de diálogo en los casos de argumentación que comparten descripción y conclusión con el argumento actual. La normalización de este valor sigue los mismos pasos que para el AD. Cuando se emplean tácticas de maximización de la eficiencia el peso de la eficiencia (wED) es elevado, dando mayor relevancia a argumentos con un elevado ED.

- Táctica explicativa (wEP): El poder explicativo, o EP (del inglés «Explanatory Power»), es un valor que representa la cantidad de piezas de información que un argumento contiene. Se basa en el número de recursos del conocimiento que fueren empleados para generar los casos de argumentación similares que hayan sido extraídos del CBR de argumentación. La normalización de este valor sigue los mismos pasos que para el AD y el ED. Cuando se emplean tácticas explicativas el peso explicativo (wED) es elevado, dando mayor relevancia a argumentos con un elevado EP.

Estos valores, que se especifican durante la creación del agente, son sin duda una buena opción para otorgar personalidad a los agentes, pero sólo en los puntos mencionados, no más allá del proceso de gestión de la argumentación. Queda claro pues que una parte potencialmente mejorable de la implementación es la referente a la del agente argumentativo, más concretamente a la complicación que conlleva crear agentes de distintas personalidades. Contar en un futuro con una versión que facilitara la configuración de agentes con diversas personalidades y comportamientos sería realmente útil y aprovechable.

3.2.3. Interacción usuario-máquina

En la carrera hay una asignatura enteramente dedicada a la explicación del vínculo entre máquina y usuario y al diseño de interfaces. Esta materia es «Interfaces Persona-Computador», o IPC, y gracias a las herramientas y conocimientos obtenidos en ella ha sido posible realizar un mejor análisis para esta sección. De primeras, el nivel de entrada para trabajar con esta plataforma es ciertamente elevado. Este hecho entra en conflicto

con la filosofía planteada antes, ya que uno de los objetivos a alcanzar es el de la accesibilidad. Son varias las razones que hacen de esta infraestructura una herramienta difícil de tratar. En esta subsección se van a analizar unas cuentas con objeto de determinar por qué zonas se podría tratar de mejorar este aspecto.

Como bien se ha dicho la AP, como herramienta, cumple con el objetivo de ser una manera probada de realizar experimentos con agentes argumentativos, las posibilidades son incontables. Como tal, la plataforma posee un gran potencial instructivo y experimental muy grande, pero se ve un poco limitada en este aspecto en el sentido de ser bastante complicado para un usuario novicio el hecho de lanzar un experimento mediante esta herramienta tan técnica. Para poder reducir el hueco entre los conocimientos del usuario promedio y los requeridos por la plataforma es necesario contar con una interfaz usable. Jacobs et al. [11] exponen cinco formas de medir la usabilidad, normalmente no es posible sobresalir en todas, lo que hace casi obligatorio saber cuáles priorizar. Estas medidas son:

- Tiempo de aprendizaje: Representa el tiempo que le toma a los usuarios adecuarse y aprender cómo usar las acciones más relevantes de cara a realizar las tareas
- Velocidad de procesamiento: Representa el tiempo que cuesta realizar dichas acciones
- Ratio de errores por usuario: Representa el número de errores que se producen por cada usuario. Es importante conocer qué tipo de errores y en qué tipo de usuarios.
- Retención a lo largo del tiempo: Representa el conocimiento que los usuarios mantienen sobre el manejo de la interfaz después de estar un cierto periodo de tiempo (días, semanas, meses etc.) sin interactuar con ella
- Satisfacción subjetiva: Representa la popularidad y el contento con la aplicación por parte de los usuarios.

Todas estas medidas requieren de la realización de pruebas con usuarios durante varias sesiones separadas en el tiempo, no es algo admisible para este proyecto, sin embargo un repaso a las características de la actual interfaz puede ofrecer una visión aproximada sobre el nivel de usabilidad de esta. Realmente va a ser un repaso rápido, pues no hay demasiado sobre lo que estudiar. En primer lugar, las herramientas de la plataforma para la depuración. Es de gran interés para el desarrollador saber el estado del programa cuando este se está ejecutando. Cuanta más información posee el desarrollador, más fácilmente puede detectar errores. A este respecto el desarrollador está a merced del IDE que use para programar y del registro de ejecución que ofrece la plataforma. El registro de ejecución es una sucesión de mensajes, cuyo propósito es dejar constancia de una determinada transición en tiempo de ejecución. Este registro es creado con el apoyo de una biblioteca de código abierto desarrollada en Java llamada Log4j[5]. Dejando a un lado el desarrollo, el otro punto para el que es interesante contar con una interfaz usable es el de la creación de instancias de problemas. En este aspecto nos encontramos con la misma tesitura, ninguna interfaz ha sido específicamente diseñada para esta labor y la única forma de llevarla a cabo es escribiendo el código correspondiente mediante un IDE o un editor de texto.

En definitiva, la usabilidad de la interfaz de la implementación no es medible, porque no se ha diseñado ninguna interfaz específica para la plataforma. Las medidas de tiempo de aprendizaje, velocidad de procesamiento, etc., son muy negativas si el perfil del usuario no es el de un programador. La creación de una interfaz, tanto para el ámbito del desarrollo, como para el ámbito del uso, es una cuestión muy a tener en cuenta considerando los beneficios que reportaría.

[5] https://logging.apache.org/log4j/2.x/

3.2.4. Un lenguaje en declive

El lenguaje elegido para la implementación de la infraestructura fue Java. El objetivo de esta subsección va a ser analizar porque fue una decisión correcta, pero también preguntarse si lo sigue siendo a día de hoy. Para ello, y con el propósito de otorgar cohesión y conexión a este documento, se van a hacer múltiples referencias a ideas presentadas en el capítulo del estado de la cuestión (2). Antes de continuar, una advertencia: el análisis expuesto en esta sección fue realizado antes de empezar el desarrollo, pero la redacción es posterior, por lo que alguno de los datos representados corresponden a meses en los que el desarrollo estaba ya por finalizar. Esto no es algo negativo, al contrario, la realidad que se puede ver es que el análisis realizado sigue siendo vigente meses después de su concepción.

Java es un lenguaje de programación perteneciente al paradigma de la POO. Al igual que la mayoría de los lenguajes de programación, también incorpora características propias de otros paradigmas, como del paradigma declarativo, del cual toma, entre otros, los operadores lógicos (paradigma declarativo lógico) o el concepto de polimorfismo (paradigma declarativo funcional). En cuanto a la interpretación y estructuración del código, Java incorpora propiedades del paradigma imperativo, como la ejecución secuencial del código y del funcional, como la posibilidad de crear métodos o funciones. En cuanto a la forma en la que se ejecutan los programas Java, el lenguaje es de tipo voraz, en contraposición a los lenguajes perezosos. La última cualidad de Java a destacar, y a la que sin duda debe gran parte de su fama, es el hecho de que es un lenguaje interpretado. Para abreviar, los lenguajes interpretados y los compilados tienen ambos pros y contras; una de las principales virtudes de los lenguajes interpretados sobre las de los compilados es que, mientras que las compilaciones generan código objeto específico para una arquitectura, limitando las máquinas en las que dicho código se puede ejecutar, los interpretes son capaces de ejecutar un código intermedio, generado a partir de lenguajes interpretados, en cualquier máquina en la que estén instalados.

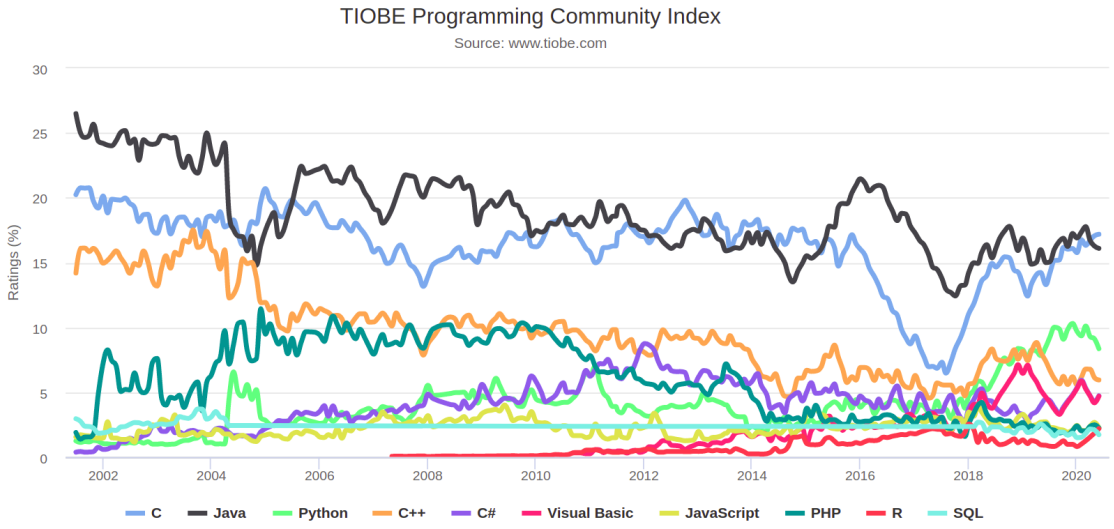
Después de leer esta información es posible empezar a intuir las razones por las que se escogió Java. En las especificaciones sobre infraestructuras en las que se utilizasen herramientas de KR era muy interesante contar con un lenguaje de la familia de la POO. Java cumple con esta condición, con el es posible definir clases que representen conceptos y realizar una jerarquización de las mismas. Una vez definidas estas se pueden manejar distintas instancias de ellas en tiempo de ejecución. Por contra, el lado de la programación lógica que sería conveniente para lidiar con la inferencia de nuevo conocimiento o con razonamientos argumentativos basados en lógicas de predicados, está un poco menos cubierto, aún así las herramientas disponibles son suficientes para cubrir las exigencias propuestas. También es una ventaja a tener en cuenta la alta accesibilidad que otorga el usar un lenguaje de programación interpretado, que tal y como se ha dicho, es mucho más sencillo de ejecutar y distribuir en muchos tipos de máquinas. En contra de esto, y haciendo referencia a «Lenguajes de programación y procesadores de lenguajes» (LPPL), los códigos objeto de los lenguajes interpretados tienden a ser menos eficientes, en términos de computación. No siempre se da el caso, pero es bastante común, y principalmente debido a que no es posible aplicar las avanzadas técnicas de optimización de código, como las basadas en autómatas finitos que sí son empleadas en los códigos compilados, puesto que el código en los lenguajes interpretados se traduce a código máquina en tiempo de ejecución. A pesar de todo, se pueden hacer aproximaciones, pero por lo general es difícil acertar el devenir de los lenguajes de programación y, pese a que todo parece indicar que la época dorada de Java ha concluido, podemos ver en la tabla 3.1 que al menos en lo que a este índice respecta, mayo fue un buen mes para Java.

Tabla 3.1: Lenguajes de programación más populares en Junio de 2020 según el índice TIOBE

Índice TIOBE Junio 2020				
Top Junio 2020	Top Junio 2019	Lenguaje	Valoración(%)	Cambio (%)
1	2	C	17.19	+3.69
2	1	Java	16.10	+1.10
3	3	Python	8.36	-0.16
4	4	C++	5.95	-1.43
5	6	C#	4.73	+0.24

Como último punto a favor de la elección de Java está su aceptación. La aceptación de un lenguaje se mide en base al número de desarrolladores que lo emplean y la cantidad de programas creados con él. En este asunto Java era el rey. Las formas de medir la aceptación del lenguaje pasan por la realización de diversos análisis, ninguna es definitiva y todas presentan aspectos a tener en consideración con respecto a las demás, pero son una fuente fiable de cara a obtener una buena aproximación. Uno de los índices más reconocidos es el que genera la compañía TIOBE [13]. Esta compañía saca una versión del «índice TIOBE» mensualmente, otorgando así datos actualizados. El otro gran motivo para tomar este índice como fuente es que los criterios aplicados para realizarlo son públicos[6], además es una fuente fiable y altamente reconocida.

Figura 3.2: Lenguajes de programación más populares a lo largo del tiempo según el índice TIOBE



Como se ilustra en la figura 3.2, cuando la plataforma comenzó a ser desarrollada, Java era no solo el lenguaje de POO más popular, sino el más aceptado en general entre todos los lenguajes. Son muchas las razones las que llevaron a este hecho, la portabilidad lo convertía en un lenguaje muy versátil y el auge del sistema operativo Android ayudó en gran medida a mantener esa popularidad. Pese a todo, se puede observar una tendencia descendente en la popularidad del lenguaje, no solo en este índice, sino que también en el número de nuevas aplicaciones y programas desarrollados en este sistema, por ejemplo, ya ga pasado a ser sólo el tercer lenguaje más usado en repositorios de GitHub, por debajo de JavaScript y Python [14]. El uso de Java es actualmente libre, pero es un lenguaje con propietario, y desde que Oracle[7] lo adquirió en 2010 se han sucedido di-

[6] https://www.tiobe.com/tiobe-index/programming-languages-definition/

[7] Oracle Coorporation es una empresa especializada en la nube https://www.oracle.com/es/index.html

versos sucesos en consecuencia. Una de estas consecuencias ha sido que Google a optado por dar cada vez más prioridad a Kotlin sobre Java en el desarrollo de aplicaciones Java.

De todos los puntos analizados en este capítulo este es, quizás, el que menos repercusión a corto plazo conlleva, aún así, la aceptabilidad y popularidad de un lenguaje son siempre factores a tener en cuenta, más aún si se pretende lograr que una biblioteca sea accesible al mayor número de desarrolladores posible. Con estas breves conclusiones se cierra el capítulo, en el próximo se van a exponer las distintas medidas que se proponen para solventar los problemas hallados en este capítulo, así como indicar cuáles son las prioridades principales.

CAPÍTULO 4
Solución propuesta

En este capítulo se expone el resultado de la deliberación alrededor de los anteriores dos capítulos. La solución elegida se presenta a lo largo de las siguientes secciones. En su presentación se empezará por explicar en qué consiste, posteriormente se expondrán las fases en las que consistirá su desarrollo y finalmente se expondrá cuál es el planteamiento de cara a la planificación.

4.1 Propuestas

Antes de mencionar cómo se va a enfocar el planteamiento del diseño, es necesario, obviamente, presentar la propuesta en sí misma. La solución propuesta son una serie de ideas que se van a exponer siguiendo el mismo orden con el que han sido mostrados los problemas a los que pretenden dar arreglo.

4.1.1. Comunicación por sockets

Los motivos que mueven a cambiar las comunicaciones deben compensar el hecho de tener que rehacer el sistema, pues es una gran inversión de tiempo. Ante las conclusiones extraídas del análisis se propone una solución que, pese a conllevar un coste temporal elevado, compensará al poder deshacerse de aquellas limitaciones.

En «Redes de Computación», abreviada «RED», se enseña que la unidad mínima de comunicación en Java son los «sockets». Estos elementos forman la interfaz entre el proceso y la red; los procesos envían y reciben mensajes mediante ellos. Son una solución interesante, teniendo en cuenta que al formar parte de la biblioteca «java.net» no crean ninguna dependencia que no sea fácilmente satisfecha, reduciendo así las ligaduras a bibliotecas de terceros. Esta no es la única propuesta a considerar, pues se contemplan más opciones. Hay más ideas extraíbles de TSR que son perfectamente aplicables en este trabajo. En dicha asignatura se presentan los «middleware» de mensajería, y en concreto uno llamado ZeroMQ[1]. Un «middleware» es un nivel (o niveles) de software y servicios entre las aplicaciones y el nivel de las comunicaciones (sistema operativo). Su uso introduce diversas transparencias, esto es, reduce el número de factores a tener en cuenta durante el desarrollo, solventándolos sin necesidad de la acción directa del desarrollador. Las ventajas de usar ZeroMQ son su gran aceptación y acogida, la gran variedad de «sockets» que ofrece de base y el hecho de que dispone de ligaduras para un gran número de lenguajes de programación, entre los cuales se encuentra Java.

[1] https://zeromq.org/

Entonces, las tres opciones que se plantean sobre el sistema de comunicaciones en Java son:

1. Conservar el sistema original
   - Ventajas:
     - No es necesario invertir tiempo en modificaciones
     - Magentix2 es una plataforma pensada específicamente para trabajar con agentes inteligentes
   - Desventajas:
     - Las comunicaciones quedan limitadas al ámbito de Magentix2, implicando limitaciones
     - Crear o modificar protocolos argumentativos ya existentes, que dependan de las comunicaciones, es una tarea que escala en complejidad
2. Sustituir el sistema original por un sistema de comunicaciones basado en «sockets» Java
   - Ventajas:
     - No requiere ligaduras a bibliotecas externas o de terceros
     - Es altamente compatible con la mayoría de las versiones de Java
   - Desventajas:
     - Es necesario invertir mucho tiempo en esta modificación
     - Añade cierta transparencia, pero sigue requiriendo mucha implicación de parte del desarrollador
3. Sustituir el sistema original por un sistema de comunicaciones basado en ZeroMQ
   - Ventajas:
     - Es altamente transparente y cuenta con varios tipos de «sockets» útiles de base
     - Sus «sockets» base, al ser más elaborados que los de Java, requieren menos tiempo de implementación
   - Desventajas:
     - Es necesario invertir tiempo en esta modificación
     - Es una biblioteca de terceros que requiere de una ligadura para funcionar correctamente en Java

Tras reflexionar sobre los pros y los contras de las tres posibilidades, tanto los tutores como el alumno estuvieron de acuerdo en proponer la última de ellas: rehacer la implementación de las comunicaciones para hacerlas funcionar con ZeroMQ. En contra de los pronósticos iniciales, por motivos que se explican en la sección 4.1.4, esta opción tuvo que ser reconsiderada, ya que tras un par de semanas de pruebas se decidió que el proyecto iba a ser reescrito en Python, otro lenguaje de programación. ZeroMQ también tiene ligaduras para Python, pero no entraron siquiera a considerarse, pues había una alternativa que se presentaba como mucho más interesante, véase el siguiente punto.

SPADE

SPADE[2] es una plataforma para soportar MAS basada en mensajería XMPP escrita en y para Python. Sus características principales son funcionar con mensajería asíncrona, dar soporte para facilitar el estándar FIPA, tener una interfaz basada en web y permitir al sistema conocer el estado de los agentes en cada momento. Además, los agentes se configuran en base a la definición de una serie de comportamientos que adoptan en función del tipo de mensaje que reciban. La distinción entre mensajes se lleva a cabo mediante la asignación de diferentes plantillas.

Para el ámbito de este proyecto, una de las pocas cualidades en las que ZeroMQ sería preferible sobre SPADE una vez determinado que Python va a ser el lenguaje a emplear es el hecho de que permite mucha más configuración de las comunicaciones, pero es una propiedad cuyo sacrificio no pesa tanto como las ventajas que aporta. Por ejemplo, al igual que Magentix2 es una plataforma pesada y orientada a dar soporte a AP, pero a diferencia de esta, al dar soporte a mensajería asíncrona, es posible desprenderse también de esos estados de espera que resultaban molestos para el caso. Otra ventaja importante y bastante exclusiva de la situación es que al ser SPADE una biblioteca mantenida por J.Palanca[3], es muy fácil comunicarse con él en caso de duda pues se encuentra cerca y es fácil de contactar.

Por lo tanto, dados los motivos expuestos en las sección 4.1.4 era necesario buscar una alternativa para las comunicaciones en Java. Tal y como se describe en los anteriores párrafos, SPADE es la plataforma de agentes a utilizar en este trabajo.

4.1.2. Genericidad

Para describir la propuesta ante el problema de la impersonalidad de los agentes se va a recurrir de nuevo a conocimientos obtenidos en LTP. El concepto fundamental es el polimorfismo. Este es una cualidad propia de algunos lenguajes de programación que se asocia al paradigma funcional, pero que no es única de este. Dicha característica permite manejar valores de tipos diferentes mediante una misma interfaz y es aplicable tanto a funciones como a tipos. Con un ejemplo se puede ilustrar mejor. Un tipo de función polimórfica es la suma, más concretamente el operador «+». Este puede aplicarse sobre diferentes tipos (enteros, reales, incluso cadenas). Por otro lado, un ejemplo de tipo de datos polimórfico son los diccionarios, pues pueden tener claves y valores de tipos arbitrarios. Los tipos de polimorfismo se clasifican en dos ramas: aparente y universal, en las cuales hay otra subdivisión dependiendo del tipo. En resumen, el polimorfismo se clasifica como:

- Aparente o *ad-hoc*: Cuando trabaja sobre un número finito de tipos no relacionado.
    - Sobrecarga: Cuando existen diversas funciones con el mismo nombre. Es el caso de los operadores aritméticos, como la suma mencionada anteriormente.
    - Coerción: Cuando se produce la conversión de valores de un tipo en otro, esta puede ser explícita si es introducida en el código o implícita si la realiza el compilador. La coerción se produce, por ejemplo, cuando se le asigna un valor entero a una variable de tipo real o cuando se concatena un entero a una cadena de caracteres.

[2] https://pypi.org/project/SPADE/

[3] Javier Palanca Cámara (http://www.upv.es/ficha-personal/japaca1) es un doctorado e investigador del Instituto Valenciano de Investigación en Inteligencia Artificial

- Universal o verdadero: Cuando trabaja sobre un número potencialmente infinito de tipos con cierta estructura común.
    - Paramétrico: Cuando la definición de una función o la declaración de una clase presenta una estructura que es común a un número potencialmente infinito de tipos. Se puede suponer, a modo de demostración, que existe un tipo de clase estudiante y un tipo de clase profesor. Existiría entonces una clase genérica de tipo persona que tendría la estructura que es común a la de las dos anteriores clases y a un potencial infinito número de clases más, como conserje, director o inspector. A este tipo de polimorfismo se le conoce también como genericidad.
    - Inclusivo: Cuando la definición de una función trabaja sobre tipos que están relacionados siguiendo una jerarquía de inclusión. Este tipo de polimorfismo es muy común en la POO. En este paradigma se le da mucho uso como mecanismo para la reutilización y la expansibilidad. Para ilustrar esta definición se puede pensar en una hipotética clase de tipo vehículo. Esta clase contaría con atributos como capacidad y velocidad de desplazamiento, así como una función para calcular cuándo debe pasar la revisión. Si quisiésemos crear una nueva clase de tipo bicicleta lo más sensato sería heredar todo de la clase vehículo y posteriormente añadir atributos o funciones propias de la bicicleta, como un atributo para almacenar el número de veces que se ha tocado el timbre. A este tipo de polimorfismo se le conoce también como herencia y está estrechamente ligado a la genericidad, ya que todas las clases que pudieran derivarse de vehículo, como automóvil, avión o tren, tendrían en común la estructura de la clase de la que heredan.

Todos y cada uno de los tipos de polimorfismo están presentes en Java y pueden ser empleados para facilitar la solución al conflicto planteado. La solución propuesta para este caso es la de crear una clase de tipo agente genérica. Esta clase podría simplemente consistir en ser el plano de la estructura que las clases que vayan a comportarse deban implementar, lo que se conoce como clase de tipo interfaz, o implementar en sí misma funciones y atributos que vayan a ser comunes en todos los agentes, lo que se conoce como clase abstracta, o directamente ser una implementación perfectamente funcional de agente de la que otras clases de agentes pudieran heredar para extender sus funcionalidades. De entre todas las opciones se ha considerado que la mejor solución es implementar una clase abstracta, pues hay funcionalidades que deben ser fijas en todos los agentes, como las relativas a los protocolos y otras que deben dejarse para que cada implementación realice acciones propias y personales, como qué hacer internamente cuando otro agente ataca su posición.

4.1.3. Interfaz web

El diseño de una interfaz gráfica para realizar las tareas relativas a la experimentación con la plataforma es un gran reto. Son muchas las opciones que ofrece la plataforma y en ocasiones bastante abstractas. El objetivo de una interfaz es hacer de nexo entre el usuario y la máquina, son tan relevantes que a lo largo de la joven vida de la informática han sido uno de los aspectos en los que más se ha fijado la atención. Desde las primitivas interfaces analógicas hasta las más recientes disponibles para la realidad mixta, todas tuvieron un motivo de ser. La inclusión de la psicología en la metodología del diseño de interfaces fue lo que derivó en adoptar conceptos como el de usabilidad. Las interfaces que han acompañado a los grandes programas a lo largo de los años han sido concebidas en un contexto acotado a unas limitaciones técnicas y unos avances en la teoría del diseño de interfaces persona-computador. Uno de los mayores saltos de dio cuando se pasó

de las interfaces basadas en caracteres a las interfaces basadas en píxeles. Esto permitió que, aparte de los textos descriptivos sobre los programas, se dispusiese de gráficos que hacían mucho más sencilla y agradable la interpretación de la información; un salto así se pretende imitar en este proyecto. Como ya se ha dicho, diseñar una interfaz para todos los aspectos de la plataforma es una labor titánica y más aún si es gráfica. Es por eso que hay que realizar un análisis de las prioridades para determinar que aspectos abarcar. A continuación se presenta la propuesta.

En la subsección 3.2.3 dentro del capítulo del análisis se ha diferenciado entre dos formas de enfocar la infraestructura. Esta división nace de la necesidad de separar entre los tipos de usuario que esta plataforma pueda potencialmente tener. Por un lado está el uso de cara al desarrollo, cuyo tipo potencial de usuario se acerca al perfil de un programador, o al menos, una persona con conocimientos técnicos sobre informática e IA. Al otro lado está el uso orientado a la experimentación, cuyo tipo potencial usuario abarca un espectro más amplío, desde el perfil de desarrollador hasta el de estudiante o curioso de la materia. Un usuario con conocimientos técnicos será capaz de desenvolverse sin demasiados problemas en ambos casos de uso, sin embargo, un usuario con conocimientos de nivel estudiantil se topará con serias dificultades para desenvolverse y verá que la curva de aprendizaje es muy pronunciada. No supone un problema que un estudiante no sea capaz de entenderse con las vías que ofrece la plataforma para realizar desarrollo, dado que no va a buscar hacerlo, por contra, sí es un problema que no sepa cómo se deben configurar y lanzar los experimentos, pues es lo que busca realizar con el programa.

En vista de lo expuesto y teniendo en cuenta lo complejo que es desarrollar herramientas para desarrolladores, la propuesta se centra en crear una interfaz gráfica para la configuración y despliegue de experimentos con agentes argumentativos. La interfaz debe ofrecer una forma de configurar fácilmente los agentes que van a conformar los grupos conversacionales, la distribución de los agentes por organizaciones y la carga de los casos de dominio y argumentación a partir de los cuales rondarán los diálogos argumentativos. En lo que respecta a la visualización de la información, es necesario que la interfaz muestre de alguna forma la evolución de las discusiones, los diálogos producidos y, por supuesto, la solución obtenida.

Para representar los diálogos en los agentes puede resultar de utilidad remontarse a lo aprendido en las unidades temáticas dedicadas al estudio de la teoría de grafos en la asignatura «Matemáticas Discretas» (MAD). Los procesos argumentativos pueden representarse mediante un grafo finito dirigido. Para justificar el por qué hay que volver a las explicaciones sobre el protocolo argumentativo del apartado 2.2.2. Los agentes serían representados por los nodos y las relaciones entre ellos serían las aristas. No es necesario representar al CS dentro de este grafo pues es únicamente consultado por los agentes argumentativos para hacer uso de sus servicios de páginas blancas, es decir, para poder contactar al resto de agentes argumentativos. El uso de aristas para representar los diálogos es viable gracias a que estos los componen únicamente dos agentes que se comunican como pares. La interfaz propuesta permitiría avanzar entre paso y paso de la ejecución representando en cada momento el estado de la ejecución, cada paso sería contado por cada petición al CS que supusiera un cambio en su base de diálogos. Los estados de la argumentación serían extraídos realizando una instantánea por cada paso contado. Para mostrar la solución hay muchas alternativas válidas, el uso de ventanas de diálogos sería una de ellas.

La otra parte crucial de la interfaz es la que da paso a la personalización de los agentes y permite lanzar los experimentos a ejecución. Por simplicidad se deberían omitir actividades como la del menú principal. El usuario debería acceder directamente a la ventana de configuración, con un enlace al manual o a la documentación por si pudiera tener dudas sobre el funcionamiento. La configuración de los agentes permitiría elegir

entre distintos valores que conformarían su personalidad. Para hacer más accesible y menos compleja la funcionalidad, estos valores deberían ser discretos en primera instancia, modificable a valores continuos si se desea. Los detalles de la implementación están reservados a la sección correspondiente del capítulo de diseño de la solución (4.1.3). En el se tomarán las propuestas de este capítulo como base y se diseñarán las que serán las interfaces a implementar.

4.1.4. Un lenguaje en auge

La decisión de cambiar el lenguaje de programación es la más polémica de todas. Esta decisión debe justificar el esfuerzo que conlleva traducir de un lenguaje a otro decenas de clases y millares de líneas de código. Se ha considerado como opción porque son varios los beneficios que se obtendrían de esta decisión. En esta subsección se van a enfrentar los beneficios contra las adversidades para culminar con la decisión tomada con respecto a este apartado.

El lenguaje en el que estarían escritas las nuevas versiones de la plataforma sería Python[4]. Volviendo al gráfico de la figura 3.2 puede parecer que no es el lenguaje más interesante en cuanto a aceptabilidad se refiere, pero hay más datos a tener en cuenta. En la revista Nature[5] se publicó un artículo [15] del autor Jeffrey M. Perkel. En este artículo del año 2015 el autor hace un repaso a las virtudes de este lenguaje de programación que, pese a no profundizar demasiado en los detalles, es bastante aclaratorio. Jeffrey introduce el artículo con el ejemplo de una ingeniera en agricultura y biosistemas que realiza su investigación apoyándose en Python como herramienta de análisis. Python es una herramienta que, aunque sus primeras versiones son más antiguas que Java, está experimentando un enorme crecimiento en los últimos años. Tal y como se describe en el artículo de Nature, es un lenguaje accesible para programadores novicios, pero es también una herramienta muy potente que puede ser empleada para tareas específicas y de rigor. Adicionalmente, mientras que Oracle mantiene como software privativo componentes importantes como Java SE[6] o el TCK de Java, Python es un lenguaje completamente abierto[7]. Para terminar con esta serie de ventajas de un lenguaje sobre otro se va a hablar de concisión. Los lenguajes con tipos explícitos, como Java, poseen la ventaja de ser más fácilmente legibles y la predisposición a cometer errores en la asignación de variables es bastante menor, por contra el número de lineas de código se vuelve mayor al no poder reutilizar variables y el tamaño del código en general aumenta, dificultando así su revisión. Python, en cambio, funciona con tipos implícitos, lo que elimina el anterior problema. Además cuenta con una sintaxis muy elaborada que permite realizar más acciones en menos código. Algunas de estas ventajas sintácticas son los generadores, los iteradores o sus funciones para trabajar con listas. Con esto se pierde una de las ventajas de la tipificación explícita de Java, la reducción de errores lógicos por una incorrecta escritura del código. Si bien es cierto que Python permite usar «duck typing» e incluso anotaciones manuales, estas acciones no funcionan a nivel de compilador o intérprete, por lo que son menos útiles.

En el párrafo anterior se han dado un conjunto de razones que podrían hacer de Python un mejor lenguaje de programación para la infraestructura, no obstante, es necesario asegurarse de que los motivos por los que Java fue elegido sigan siendo válidos para Python, es decir, el nuevo lenguaje debe de permitir hacer las mismas funciones que Java,

[4] https://www.python.org/

[5] Nature es una de las más prestigiosas revistas científicas a nivel mundial (https://www.nature.com/).

[6] Java «Standard Edition» es una colección de APIs para Java. Entre ellas están «java.security» que da soporte a sistemas de seguridad o «java.sql» que permite trabajar con bases de datos SQL.

[7] Todas las últimas versiones de Python son patentadas bajo una licencia GPL (https://www.python.org/doc/copyright/ certificada por la Open Source Initiative (https://opensource.org/)

al menos, como mínimo, las que han sido aprovechadas para el desarrollo. La respuesta es sí, Python es un lenguaje igual de válido que Java para este desarrollo. Python es un lenguaje de programación de carácter general que también se desempeña bien en labores específicas. Al igual que Java, pertenece al paradigma de la POO e incorpora características propias de otros paradigmas, en este aspecto es incluso más indicado que Java, pues su sintaxis está muy bien construida cuando a tratar listas se refiere. Finalmente, el tema de las comunicaciones también queda cubierto. La sección 4.1.1 tiene una descripción de las virtudes de SPADE que se considera, no sólo mejor que Magentix2 como alternativa, sino que preferible incluso a la reimplementación por «sockets» Java o ZeroMQ.

En base a los anteriores párrafos, la propuesta consiste en reimplementar la totalidad del código fuente original para que use Python. Esta decisión facilitaría futuras actualizaciones y daría paso a reconstruir todo el sistema de comunicaciones entorno a SPADE.

4.1.5. Documentación automática y manual de manejo

La generación automática de la documentación está marcado como uno de los objetivos principales, esto es así porque el hecho de contar con una documentación hace más accesible cualquier sistema y además existen actualmente muchos programas capaces de generar documentación muy elaborada y conectada en base a análisis del código y de sus comentarios.

Podemos contemplar en Java que los comentarios sobre los métodos y clases, también conocidos como cadenas de documentación o «docstrings», están escritos con la idea en mente de usar herramientas de generación automática de código. Esto es porque presentan palabras y caracteres reservados de estas herramientas, como por ejemplo «@param». El uso de la palabra «param» con el carácter «@» delante implica que herramientas como Doxygen[8] (y JavaDoc, que de hecho fue la herramienta usada en su día) interpreten el «token» siguiente como un parámetro del método y la oración siguiente a esta como su descripción. Doxygen es una herramienta de generación automática de la documentación muy extendida, se han dado seminarios sobre su uso en asignaturas como IPV. Una gran virtud de la documentación generada por Doxygen es que, si las cadenas de documentación son escritas siguiendo correctamente su sintaxis, permite crear enlaces entre elementos de la plataforma. Esto implica que si, por ejemplo, una descripción menciona que utiliza parámetros de un tipo de clase definido en otra parte de la plataforma, es posible pulsar sobre este tipo que, al estar enlazado con la clase a la que representa, te transporta a la descripción de su documentación.

Se asume que el lenguaje de programación a emplear va a ser Python. Este no es realmente ningún tipo de inconveniente, al menos puede parecerlo a simple vista. La realidad es que Doxygen puede emplearse para generar documentación automática de código Python, pero no directamente, es necesario pasar el código por un analizador sintáctico que pueda convertir el formato de los comentarios en Python a un formato aceptado por Doxygen. En primera instancia puede resultar difícil de comprender que Doxygen no dé soporte directo para este tipo de documentación en Python, pero hay dos motivos principales que hacen más fácil comprender la situación. El primero de todos es que Python no es un lenguaje de tipos explícitos como ya se ha dicho. Este problema se solventa con las anotaciones de tipos que permite realizar el lenguaje, pero esta característica es relativamente reciente en Python y todavía no está completa, como se ha dicho previamente algunas de sus funcionalidades requieren del uso de bibliotecas. El segundo motivo es que Python no tiene un formato de cadena de documentación estandarizado. Hay cua-

[8] https://www.doxygen.nl/index.html

tro formatos principales de para escribir comentarios sobre el código Python, todos ellos soportados por el IDE Pycharm, aunque los nombres puedan variar levemente:

- NumPy/SciPy: Normalmente usada sobre código en programas de ámbito científico que hacen uso de bibliotecas como NumPy[9]. Posee soporte para Sphinx[10].
- PyDoc: Es de todos los formatos el más cercano a considerarse un estándar. Posee soporte para Sphinx.
- EpyDoc: Se basa en el lenguaje de marcado Epytext[11], diseñado específicamente para documentar programas Python. A partir de este formato se generan una serie de documentos HTML con la documentación del proyecto.
- Google Docstrings: Es el formato para cadenas de documentación propuesto por Google. Está presente en el código fuente de los programas libres de la empresa.

Si bien en la «wiki» de Python es posible acceder a las herramientas recomendadas para la documentación del código[12], algunas están ya obsoletas o cumplen solo parte de la funcionalidad completa requerida. Curiosamente, Sphinx no aparece en el listado, lo cual es impactante teniendo en consideración que varios de los formatos principales tienen soporte bajo esta herramienta. En contra de lo que pueda parecer, Sphinx no es la herramienta propuesta para generar la documentación automática, sino la llamada «Read the Docs»[13]. Las razones para elegir esta herramienta son también dos. Siendo la primera el hecho de ser una recomendación de los tutores y la segunda la convicción de que para el ámbito de este proyecto ofrece lo mismo que Sphinx permite realizar. Para acompañar esta herramienta se ha hecho uso de un añadido para Pycharm que sí aparece en el listado de la página anterior. Este añadido es AutoDoc[14] y se encarga de corregir errores en el formato de la documentación y lanzar avisos en caso de estar olvidando algún parámetro.

Se ha propuesto emplear «Read the Docs» como herramienta en el proyecto porque se busca que la versión liberada al final del desarrollo de este sea lo más accesible posible. La elección deja atrás la opción de enlazar descripciones de elementos como las disponibles en Doxygen en favor de un desarrollo más rápido y ligero, además de un soporte directo para el almacenamiento y disponibilidad de la documentación.

4.2 Planteamiento de la planificación

Esta sección es una breve presentación de las partes de las que deberá constar el desarrollo del proyecto para poder implementar todas las propuestas anteriormente indicadas. Estas fases, sin orden definitivo, son:

1. Traducción del código fuente: El código fuente de la implementación original de la plataforma para agentes argumentativos será traducido íntegramente a Python. Esta fase sí debe ser la primera.
    - Reimplementación de las comunicaciones: Durante la traducción se aprovechará para sustituir las partes del código que funcionan bajo Magentix2 por un código equivalente que funcione sobre SPADE.

[9] https://numpy.org/
[10] https://www.sphinx-doc.org/en/master/#
[11] http://epydoc.sourceforge.net/epytext.html
[12] https://wiki.python.org/moin/DocumentationTools
[13] https://readthedocs.org/
[14] https://plugins.jetbrains.com/plugin/8561-autodoc

- Generalización del agente argumentativo: Una vez traducido todo el código aparte será el momento de traducir la implementación del agente argumentativo. Su puede traducir la clase primero completamente y luego extraer las genericidades o realizar la generalización antes de la traducción completa.

2. Diseño de la interfaz: Teniendo en cuenta las funcionalidades a cubrir se diseñará una interfaz usable para el sistema. Como es lógico, esta fase debe ir antes de la implementación de la interfaz.

3. Implementación de la interfaz: Los diseños realizados pasarán a ser implementados.

4. Pruebas: Esta fase son en realidad varias. Después de (y durante) la traducción del código se deberán realizar distintas pruebas que garanticen su correcta funcionalidad. También serán necesarias pruebas para la interfaz. En el siguiente punto se profundiza un poco más.

La forma en la que serán implementadas, así como los plazos estimados para cada una serán presentadas en el próximo capítulo. En él se verán con detalle los planteamientos iniciales sobre qué dirección debía tomar el proyecto y partiendo de las propuestas, qué camino se escoge finalmente.

CAPÍTULO 5
Diseño de la solución

Este es un capítulo altamente destacado. En él se identifican los grandes bloques o subsistemas en los que se dividirá la solución, concretando, a nivel de diseño, aspectos que van definiendo dicha solución propuesta, ofreciendo un alto grado de detalle. Visto de otra forma, consiste en tomar la sección 4.2 y llevar a cabo sus puntos explayándose sobre ellos.

5.1 Consideraciones

Esta sección está redactada a modo de advertencia y para dotar de sentido a la estructura y contenido de este capítulo para este trabajo en concreto, pues a lo largo del desarrollo se han dado circunstancias con las que no se contaba en un principio. Estas situaciones excepcionales han dado lugar a ciertas reconsideraciones y cambios en los planes originales. Estos cambios pueden dividirse en dos, según afectan al diseño o al desarrollo en si mismo. Sobre el segundo punto, el peso de la decisión y sus consecuencias se hablarán en el próximo capítulo, que trata, precisamente, del desarrollo de la solución. En cuanto al primero, ya que afecta directamente a este capítulo, se tratará a continuación.

Como se ha explicado en anteriores secciones, pese a que los objetivos y la motivación siguen siendo los mismos que cuando se concibió la idea del proyecto, en primera instancia el lenguaje de programación iba a seguir siendo el mismo y las únicas partes que necesitarían de un retoque serían por un lado las comunicaciones, que iban a ser reemplazadas por «sockets» y por el otro el agente argumentativo, que debía ser reimplementado para permitir acabar con la impersonalidad de sus instancias. Existe una gran diferencia en la cantidad de trabajo a realizar de un momento a otro.

Se ha considerado oportuno hablar de las primeras intenciones en capítulos anteriores, porque forman parte de la historia del proyecto, aún así, en este capítulo se va a concebir el diseño de la solución teniendo en cuenta únicamente la visión final del mismo. Esta decisión está enfocada a evitar confusiones y a mantener la coherencia de los siguientes puntos. De esta forma, los siguientes puntos están redactados de cara a definir una solución que implemente las propuestas, dadas como definitivas, propuestas anteriormente.

5.2 Métodos de traducción

La traducción en sí misma no tiene mucho misterio, pero son muchas las formas de llevarla a cabo. Al considerar este proceso una tarea repetitiva, pero difícil de mecani-

zar, se consideraron una serie de alternativas. Para las opciones a continuación hay que considerar que tras la traducción del código se pretenden conservar sus cualidades de documentación autogenerativa.

5.2.1. Automatización

La traducción automática de código pretende realizar una conversión del código fuente en un lenguaje de programación a código fuente escrito en otro lenguaje de programación distinto. En este caso la conversión a conseguir sería la de código Java a código Python. La herramienta indicada para esta acción era, aparentemente, java2python[1]. Esta herramienta permitiría traducir código en Java a código Python sin demasiadas complicaciones, pero con una serie de inconvenientes. El programa realiza las traducciones a Python 2.7, sin embargo, el proyecto debía ser desarrollado en Python 3.6 o superior para garantizar la compatibilidad con otros módulos, como SPADE, que requiere de Python 3.6 o superior. Por otro lado están los errores reportados, pero que todavía no han sido corregidos, ya que parece que el mantenimiento de la herramienta ha sido dejado de lado y también la dificultad para revisar las traducciones. En aclaración del último punto, para clases pequeñas no supone ningún problema, pero para las grandes, siendo el caso más extremo el agente argumentativo, puede ser realmente complicado asegurarse de que, pese a funcionar aparentemente, la traducción no haya generado errores lógicos. Todas estas razones condujeron a apartar esta alternativa, pero no es una mala opción combinarla con otras para lograr traducciones más rápidas.

5.2.2. Ingeniería inversa

La ingeniería inversa, también llamada retroingeniería es conocida popularmente por aparecer en relatos sobre OVNIS que se estrellaron en México o por ser empleada por compañías para desentrañar los secretos de la competencia. Más allá de los relatos fantasiosos y de sus usos considerablemente poco éticos, hay un uso de la ingeniería inversa que puede ser de gran ayuda en labores como la que concierne a esta sección. La técnica de la retroingeniería puede emplearse para, tras un análisis del código, generar un diagrama representativo de su estructura misma.

Pycharm, que es el IDE empleado para llevar a cabo el desarrollo del proyecto, incorpora la posibilidad de realizar diagramas pseudo-UML a partir del código fuente, pero es una característica muy limitada, pues al no ser un lenguaje con tipos explícitos no puede saber de qué tipo son las variables que se emplean en ciertas clases para marcar dichas relaciones. Esto limita a que las relaciones se marquen en función de las clases que sean importadas al archivo del programa y usadas como superclase Esta herramienta sirve para mostrar las relaciones de dependencia dentro del código, pero además puede emplearse en un único archivo simultáneamente, por lo que de tener el código separado en varios, lo cuál es lo recomendado en proyectos medios y grandes, la herramienta pierde bastante su utilidad.

Esto ya lo hace incompatible con su principal ventaja, que sería ir comparando el esquema en Python con el esquema en Java, pero el problema no se queda ahí. Como (no) se puede observar en las figuras 5.1 y 5.2 se han creado dos programas para generar los diagramas para ambos códigos. El primero ha sido generado mediante Pycharm, como ya se ha comentado y el segundo con un añadido a Eclipse llamado AgileJ. El problema se puede percibir a simple vista: los diagramas son demasiado grandes y los detalles demasiado pequeños. El diagrama sobre el código Python es más legible, pero no mues-

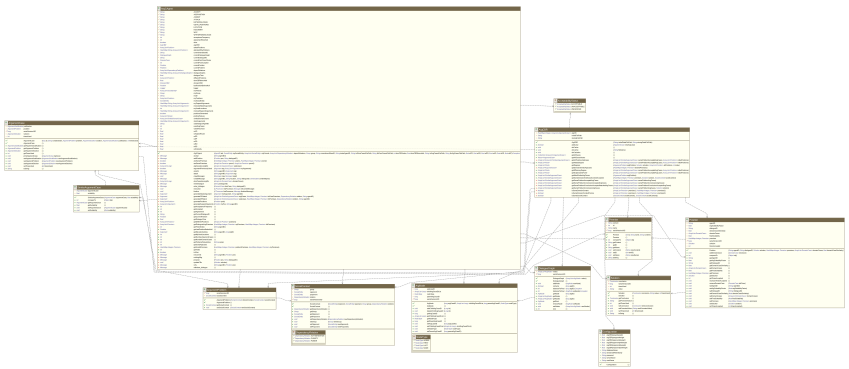
[1] https://github.com/natural/java2python

tra toda la información que debería, por otro lado, la versión de Java muestra toda la información de interés (puede mostrar más incluso) pero es tan grande que es ilegible, sin olvidar que representan una clase relativamente pequeña en comparación a todo el código de la infraestructura.

Las razones expuestas llevaron a aparcar a un lado esta metodología, pero fueron de mucha utilidad de cara a comprender ciertos mecanismos y para adquirir conocimientos sobre ingeniería inversa y sus posibles aplicaciones. Incluso, no es mala opción tener en cuenta estas herramientas para desarrollos de menor tamaño, sobretodo AgileJ que es una herramienta muy completa, aunque es necesario tener en cuenta que funciona por comandos, por lo que requiere un tiempo de aprendizaje.

Figura 5.1: Esquema de dependencias del CBR de argumentación ofrecido por Pycharm

Figura 5.2: Esquema de dependencias del CBR de argumentación ofrecido por AgileJ



5.2.3. Anotaciones de tipos

De nuevo, las consecuencias de emplear tipos implícitos en Python suponen un problema. En los lenguajes de tipos explícitos, como Java o C, al cometer un error semántico el compilador hace saber al usuario que está haciendo algo mal, incluso algunos IDEs modernos realizan un análisis en tiempo real de código y mediante el subrayado del código avisan sin necesidad de compilar. Esto en Python no sucede, puesto que las variables no están sujetas a un tipo cuando se definen, esto trae muchas ventajas, pero como se puede

ver también contrariedades. Esta circunstancia hace que, como se ve en LPPL, se reduzca mucho el valor del análisis sintáctico, reduciendo el número de ocasiones en las que el intérprete comunica que se están cometiendo errores de programación.

Lo ideal sería contar, al menos, con los subrayados de la sintaxis de los IDEs para el desarrollo. Sin ellos es muy difícil seguir el hilo de programas grandes en los que hay una alta reutilización de variables y métodos. En este proyecto no se contempla la reutilización de variables para emplear valores de tipos diferentes, al partir de un lenguaje con tipos explícitos es muy sencillo prescindir de ella. Para poder contar con dicho mínimo se puede optar por el anteriormente mencionado «duck typing» o por las anotaciones que ofrece Python. Con la intención de probar un poco de todo se pretende poner en práctica ambas. En Pycharm es bastante sencillo activar el «duck typing», para hacerlo hay que acceder a «Settings», seguidamente ir a «Build, Execution, Deployment» y, finalmente, en «Python Debugger» activar la opción que dice «Collect run-time types information for code insight». Esta funcionalidad por sí sola es útil, pero no es suficiente para abarcar todo lo que exige el mínimo marcado. La otra alternativa, las anotaciones, son también sencillas de aplicar, pero son una labor más complicada. Lo primero de todo es conocerse la documentación de la biblioteca «typing»[2], que es la que da soporte a estas anotaciones. Si no se conocen bien las anotaciones pueden introducirse indicaciones erróneas, aumentando la confusión del programador y ralentizando el desarrollo en lugar de agilizarlo. Es una labor ardua, pero una vez aprendido el sistema merece totalmente la pena. Lo segundo es prestar atención a las marcas en el código. Cuando el código va en contra de lo manualmente indicado es introducido, el IDE marca en amarillo las líneas en las que se encuentra el error.

En conclusión a esta sección, ni la traducción automática ni la ingeniería inversa formarán parte finalmente del desarrollo, pero si lo hará el uso del «duck typing» y las anotaciones. Estas últimas siguen en desarrollo, por lo que presentan algunos fallos. Afortunadamente es posible importar la biblioteca «__future__»[3] que incorpora correcciones para la mayoría de estos problemas.

5.3 Orden de traducción

En realidad, es injusto decir que las herramientas de retroingeniería para generar diagramas han servido para poco más que aprender un par de cosas. Definitivamente los diagramas generados para elementos grandes son muy difíciles de implementar, pero, si se configura para que únicamente muestre las clases y sus relaciones, sin métodos ni campos de variables, se pueden determinar con facilidad las relaciones de dependencia de las mismas. Conociendo estas relaciones de dependencia es fácil determinar que clases deben de traducirse primero. Puede parecer algo indiferente, porque si el objetivo es traducir todo el código, por dónde se empieza y dónde se acaba no se plantea como una cuestión relevante. Lo cierto es que sí es importante, traducir todo el código de una sentada y en un orden arbitrario y esperar que funcione correctamente puede ser comparable a pedir un milagro. La mejor decisión es traducir todo el sistema por partes e ir probando estos módulos de manera individual para poder solventar los errores de forma más localizada, que es, sin duda, una forma más eficaz de afrontar la situación.

Tan sólo con leer la sección 2.2.2 es posible figurarse cuáles van a ser las dependencias y, por lo tanto, que orden se debe seguir. El objetivo es hacer funcionar a los agentes argumentativos junto al CS. El CS es un módulo sencillo que puede funcionar de manera

[2] https://docs.python.org/3/library/typing.html

[3] Importar la biblioteca «__future__» es una forma de acceder a funcionalidades que no están presentes en las versiones actuales del lenguaje, pero que están preparadas para ser añadidas en un futuro próximo.

independiente a la mayoría de los elementos de la infraestructura, salvo el sistema de comunicaciones y algunos recursos del conocimiento. Por otro lado, el agente argumentativo está en lo más alto de la pirámide. Como se ha explicado, un agente argumentativo tiene dos módulos CBR, uno del dominio y otro de argumentación. Entre ambos el más sencillo es el componente CBR de dominio. Este componente CBR depende principalmente de los KR, del sistema de configuración y de las funciones métricas y de cálculo de similitud. A su vez, algunos KR dependen también de las funciones métricas. Los otros tres módulos son independientes. Por lo tanto, el orden lógico para comenzar sería tomar las funciones métricas y el modelo de configuración y pasarlos a Python. Posteriormente tocaría traducir todos los recursos del conocimiento, pese a que no todos son necesarios para el módulo CBR del dominio, hacerlos seguidos permitirá trabajar más rápido que segmentando esta parte, pues las partes similares se relacionan y entienden más rápido al no tener que estar cambiando el enfoque del problema. Una vez traducido lo anterior, toca el CBR del dominio y una vez finalizado este sería el momento de realizar pruebas. Tras las pruebas, estarían todas las dependencias para el módulo CBR de argumentación también resueltas, por lo que sería momento de convertirlo a Python y posteriormente probarlo. Una vez traducidos y probados los componentes CBR llegaría el turno del CS. Es preferible traducir primero este componente, pues se puede aprovechar su sencillez para empezar a manejar el sistema de comunicaciones. Finalmente, una vez implementado y probado el CS, llega el turno del agente argumentativo. Tras traducir el agente, tocaría probarlo y, de confirmar su correcto funcionamiento, proceder a su generalización.

En resumidas cuentas, el orden de traducción sería:

1. Funciones métricas y sistema de configuración
2. Recursos del conocimiento (KR)
3. CBR del dominio
4. CBR de argumentación
5. Commitment Store (CS) y el *sistema de comunicaciones*
6. Agente argumentativo
7. *Generalización* del agente argumentativo

Los elementos en cursiva indican que no son traducciones al uso, estas partes marcadas requieren de nuevo y distinto código. Como tal, estas partes van a ser tratadas más en profundidad en las secciones que siguen a esta. Y para finalizar, la traducción de las cadenas de documentación en funciones y clases se realizará al mismo tiempo que las mismas. El formato de los comentarios será el propuesto anteriormente en la sección **??** y la herramienta de apoyo también.

5.4 SPADE y las comunicaciones en MAS

Magentix2 está enfocado a MAS, pero su funcionalidad a base de mensajes síncronos suponían un problema para este proyecto en concreto, es por eso que se decidió optar por encontrar una alternativa. Dado ya el hecho de que la infraestructura iba a ser migrada a Python, surge una muy buena oportunidad para reemplazar el sistema de comunicaciones anterior por uno más moderno. SPADE es mejor alternativa para este proyecto en concreto que lo que podrían serlo los «sockets» de Java o de ZeroMQ. SPADE es también

una plataforma de MAS. De forma similar a Magentix2, SPADE cuenta con una clase agente de la cual heredar para implementar programas con esas características. Sin embargo, SPADE innova en partes en las que Magentix2 se queda detrás. Los protocolos usados para definir el funcionamiento de los agentes en Magentix2 toman la forma de los llamados comportamientos en SPADE y van un paso más allá. Esta plataforma escrita en Python permite construir la funcionalidad de los agentes bajo dos pilares: las plantillas para mensajes y los comportamientos. Para poder definir la solución al problema es necesario entender el sistema con el que va a ser construida, a continuación se van a explicar más en profundidad estos dos términos.

Un comportamiento es, tal y como indica su nombre, una forma de actuar. SPADE permite definir muchos tipos de comportamientos y asignarlos a diferentes agentes. En la documentación de la misma se define un comportamiento como «una tarea que un agente puede ejecutar siguiendo patrones de repetición». Para definir una clase de comportamiento es necesario extender de una de las especializaciones de la clase «Behaviour» de la misma plataforma. Estos comportamientos base tienen patrones de repetición distintos y son los que otorgan variedad al funcionamiento del sistema. Los hay cíclicos, periódicos, de una sola ejecución y varios más. Entre ellos el más relevante para la ocasión es el comportamiento FSM, que extiende del comportamiento cíclico y como se puede deducir por su nombre, permite definir una actitud basada en una máquina de estados finitos. Este es un tipo especial de comportamientos que permite la definición de estados y transiciones dentro del mismo. Es un comportamiento que viene como anillo al dedo para implementar el protocolo argumentativo, pues como se ha indicado, está definido en base a un autómata finito.

Un mismo agente puede tener varios comportamientos al mismo tiempo y SPADE ofrece una buena herramienta para facilitar esta gestión, las plantillas. Una plantilla vacía es como un mensaje en blanco, ante este tipo de plantillas todos los tipos de mensajes dan positivo. Cuando a una plantilla se le asocia un valor a cualquiera de sus atributos (remitente, destinatario, cuerpo o hilo) ante ella solo darán positivo los mensajes en los que coincidan los valores con los de la plantilla, excluyendo los que estén en blanco en esta. Para el caso de los metadatos es algo más complejo, si el mensaje lleva varios metadatos asociados, pero la plantilla tiene menos, es suficiente con que coincidan los metadatos presentes en la plantilla para que el mensaje de positivo. Las plantillas se asocian a los comportamientos, de esta forma cuando un mensaje da positivo significa que ese comportamiento podrá recibir el mensaje (si invoca al método de recepción). Combinando plantillas y comportamientos se pueden crear funcionalidades complejas y bien organizadas, es por este motivo que SPADE es tan buena opción para el caso actual.

Volviendo a los dos casos en los que este sistema es requerido, así es como se plantea su implementación:

- Commitment Store: El protocolo del CS es sencillo, podría resumirse en que está a la espera de recibir una solicitud de algún agente para poder contestarla, es un actor pasivo que actúa como pizarra para los agentes argumentativos. El comportamiento a implementar extenderá del cíclico, el CS estará constantemente ejecutando la misma tarea. Esta tarea consistirá en esperar a recibir los mensajes para procesar la acción correspondiente al acto performativo que se le indique. No es necesario el uso de plantillas, pues los mensajes que espera recibir son todos dentro del mismo ámbito, sería demasiado costoso crear una plantilla para cada acto performativo, aunque también posible. En su lugar se esperará a recibir un mensaje para comprobar sus metadatos y decidir la siguiente acción en consecuencia.

- Agente argumentativo: El protocolo del agente argumentativo es altamente complejo en comparación con el del CS. Debe asumir acciones distintas dependiendo

del estado en el que se encuentre, por lo tanto la mejor decisión es emplear un comportamiento que extienda del FSM. Para lograr que funcione correctamente será necesario que al arranque se inicialicen todos los estados y las posibles transiciones entre los mismos, de esta manera será posible pasar al agente de un estado a otro dentro del comportamiento. Al igual que en el CS el agente argumentativo define su reacción a los mensajes en base al acto performativo que acompaña al mensaje recibido. Esta respuesta será condicional al estado en el que se encuentre el agente. Por último sobre los estados, como se viene diciendo, los estados de espera ya no son necesarios al funcionar SPADE con mensajería asíncrona, por lo tanto serán retirados y, en caso de realizar alguna funcionalidad específica, dicha funcionalidad se realizará en el estado destino de la arista saliente del estado de espera. Hay ciertos tipos de mensajes, como se explica en la sección 2.2.2 que no se esperan en ningún estado, es por esto que es necesario definir al menos dos plantillas. Hay otra forma de ver por qué esto es así, cambiando la forma de interpretar el problema. Que no esperen ser tratados en ningún estado en específico implica que pueden ser recibidos en cualquier estado. Se definiría una plantilla que admitiese mensajes con este tipo de actos performativo de ámbito general y se asociaría al comportamiento correspondiente, cíclico como el del CS, que las manejase. Por otro lado, al comportamiento de la FSM se le asociaría la misma plantilla, pero negada, ya que esta es otra de las funciones útiles que incorpora SPADE: operadores sobre plantillas.

5.5 Generalizando el agente argumentativo

Las ejecuciones bajo el sistema actual pueden ser diferentes en la medida en que se cambien los casos de dominio y argumentación empleados como bases iniciales, así como el número de agentes, pero no el tipo de estos. Los agentes argumentativos de las ejecuciones son todos instancias del mismo agente a los que, para diferenciarlos, se les atribuyen casos distintos para inicializar sus bases CBR, promoviendo así el diálogo y la argumentación en busca de un consenso. Bajo esta premisa, los resultados son satisfactorios, pero sería preferible tener la capacidad de otorgar diferentes personalidades a los distintos agentes. Se ha discutido ya que la mejor solución a este problema es usar mecanismos de genericidad propios de los lenguajes de POO. Se ha concluido también que, para lograr este objetivo, es necesario tomar las partes del agente base que deben ser comunes a todos los agentes que se generen, para garantizar que opera tal y de acuerdo a la descripción que se ha dado sobre él, pero dando libertad para hacer modificaciones que permitan variar su personalidad sin interponerse en su funcionalidad básica. En lo que sigue se van a separar las partes que se van a generalizar de aquellas que pueden ser modificadas libremente para cada implementación.

5.5.1. Parte genérica y parte original

Como bien se ha dicho ya, la parte general a todos los agentes debe contar con la estructura que asume, a lo mínimo, el comportamiento básico de lo que se ha descrito que es un agente argumentativo. Revisando la definición proporcionada anteriormente se puede determinar que para tener un agente argumentativo se deben juntar al menos cuatro componentes: el módulo CBR del dominio, el módulo CBR de argumentación, el proceso argumentativo y el proceso de control de la argumentación. En mayor o menor medida estas partes se pueden generalizar para lograr la meta planteada. Las partes del agente que sean consideradas independientes serán las que más se presten a las modificaciones.

Protocolo argumentativo

Esta es la parte que menos se presta a tener distintas implementaciones. Para que todos los agentes puedan comunicarse correctamente deben ceñirse a un protocolo de comportamiento, en este caso, el argumentativo. De todas formas es necesaria una distinción, hay una estructura que se debe mantener obligatoriamente, pero de esta estructura se desprenden ciertas subtareas que permiten un alto grado de originalidad. La estructura inherente a cualquier agente argumentativo es la representada en la figura 2.3, es decir, los estados, sus relaciones y las transiciones entre los mismos, así como los detonantes de las mismas, deben mantenerse inmutables en una u otra implementación del agente argumentativo. Por otro lado, siempre y cuando no se modifique o entorpezca este comportamiento, se pueden asignar, o modificar las ya existentes, tareas relativas a la entrada o salida de un estado. La existencia de estos eventos ya se contemplaba en la versión de Java, pues, pese a que la mayoría son instrucciones vacías, la infraestructura da soporte a la ejecución de código tras recibir la señal de, por ejemplo, finalizar un diálogo. Cuando un agente es ordenado finalizar un diálogo está obligado a realizar las acciones asociadas en el protocolo, pero, siempre y cuando no deshaga dichas acciones puede realizar tareas extra, como almacenar con quién ha mantenido la conversación y quién se ha llevado el gato al agua.

Para concretar, el protocolo para la argumentación entre agentes se puede dividir en dos partes: la general, que debe formar parte de la implementación de todos y cada uno de los agentes argumentativos de la plataforma y la personal, que acepta añadidos que comprometan el funcionamiento de la estructura general, permitiendo así un alto grado de originalidad. Esta diferenciación se va a dar en todos los módulos a analizar. Una vez concretadas todas estas separaciones se ofrecerá el diseño definitivo para el agente argumentativo genérico.

Módulos CBR

Son dos los módulos CBR presentes en un agente argumentativo, el del dominio, usado para almacenar los casos que contienen conocimiento sobre problemas ya resueltos y el de la argumentación, usado para almacenar los casos que contienen conocimiento sobre experiencias argumentativas ya pasadas. Son elementos imprescindibles dentro de un agente argumentativo, por lo que sí deben ser un atributo general. Analizando ambos módulos se puede entrever que son, en esencia, bases de datos para almacenar los correspondientes casos junto a una serie de métodos auxiliares que sirven, principalmente, para gestionar estas bases de casos. La pregunta entonces gira entorno a si es posible o no otorgar personalidad a los agentes por medio de estos módulos y la respuesta es: no, al menos directamente. Esta negación entra en conflicto con la afirmación del punto anterior, por lo que es necesario extenderse un poco más.

Los métodos de gestión de las bases de casos de ambos módulos deben estar siempre presentes, pues son requeridos para hacer funcionar el proceso de control de la argumentación. Alguno de estos métodos, en el componente CBR del dominio en este caso, consiste en extraer de la base de casos una lista de casos similares al ofrecido, ordenados por orden de semejanza. Métodos como este deben estar presentes irremediablemente, por lo que no se puede decidir sobre su presencia, sin embargo, sí hay algo sobre lo que se puede decidir: los criterios. Algunas de las funciones métricas, entre las mencionadas como prioritarias en el orden de traducción en la sección anterior, definen diferentes formas de calcular las similitudes de los casos. Hay tres tipos de cálculo ya definidos en estas funciones: la similitud euclídea normalizada, la similitud euclídea ponderada y la similitud de Twersky normalizada. Cada una de ella ofrece cualidades diferentes y la

elección de una sobre la otra puede estar ligada a las circunstancias, o al simple gusto del programador. Estas variaciones en el criterio respecto al cálculo de las similitudes permiten dotar de personalidad a los agentes si se definen distintos criterios para estos cálculos en cada agente.

Los módulos CBR tienen una estructura sólida y son un componente crucial de los agentes, que permiten poca o ninguna variación de unos agentes a otros. Los módulos CBR serán, por lo tanto, generales a todos los agentes argumentativos, permitiendo, eso sí, pequeñas variaciones en los criterios tales como los que se encargan de determinar la similitud entre casos. De esta forma se permite una leve, pero existente, personalización de los agentes a este respecto.

Proceso de control de la argumentación

Puede ser que el proceso de control de la argumentación provoque confusión con respecto al protocolo argumentativo a causa de su nombre, pero son elementos completamente diferentes. El protocolo argumentativo ya ha tenido un hueco en esta sección, el proceso de control de la argumentación va a ser el centro de atención de este punto. Es el último de los puntos a tratar previa definición del diseño definitivo para la solución en lo que a crear un agente argumentativo genérico respecta.

Este módulo, más que englobar un proceso, se corresponde con una serie de procedimientos que definen la forma de argumentar de un agente argumentativo. Estos procedimientos son los encargados de generar las posiciones, los argumentos de apoyo y los argumentos de ataque que el agente esgrimirá en una argumentación, partiendo de sus recursos del conocimiento, establecidos en sus módulos de CBR del dominio y de la argumentación. Pese al énfasis en la distinción con el protocolo argumentativo, es posible encontrar similitudes a este respecto entre ellos. El proceso de control de la argumentación debe garantizar que el agente es capaz de generar argumentos de ataque y defensa, así como sus propias posiciones. Esta parte debe ser general a todos los agentes, pero al igual que con los módulos CBR y el protocolo argumentativo, la originalidad está en los detalles de implementación. Si el agente genera los argumentos y las posiciones cuando lo requiere, este puede usar el criterio que desee. Este es el punto de los tres vistos que más se presta a la variedad. Realizando modificaciones en estos métodos se pueden crear personalidades mucho más «palpables» que con cualquiera de los anteriores porque, por ejemplo, se puede lograr que un agente presente un comportamiento hostil si se da prioridad a que cuestione todas las posiciones de los otros agentes, o por el contrario, que sea un agente muy influenciable, que cuando su posición es atacada raramente la defienda. Este diseño da pie a crear actitudes muy variadas, además de las mencionadas sería posible añadir factores que los agentes pudieran tener en cuenta para definir si son rencorosos, permisivos o incluso que recuerden con que agentes han mantenido diálogos para partir de un prejuicio al comenzar con el diálogo. Las posibilidades son muy extensas.

5.5.2. Diseño del agente argumentativo genérico

La intención de esta subsección es tomar todo lo expuesto en la anterior y usarlo para definir el diseño definitivo del agente argumentativo genérico con el que comenzar su implementación. Para hacer más clara la representación se le va a dar utilidad a AgileJ, el añadido para Eclipse mencionado anteriormente. Representar una única clase, sin las relaciones de herencia o dependencia, mediante esta herramienta ofrece resultados que si son fáciles de interpretar. Por medio de la ingeniería inversa con la que funciona AgileJ se ha generado el diagrama de la figura 5.3. Este diagrama va a ser empleado para ilustrar información acerca de la solución propuesta. La figura se separa en dos partes bien

Figura 5.3: Estructura del agente argumentativo vía AgileJ

diferenciadas: la superior, en la que se encuentran los campos de las variables de la clase y la inferior, en la que se encuentran los métodos y funciones que implementa la clase. En la sección superior se han colocado una serie de marcas para remarcar algunas características sobre las variables señaladas y como serán tratadas en la solución diseñada. Su significado es el siguiente:

1. Variables del protocolo: Estas variables se utilizan como macros para evitar errores tipográficos. Son terminología empleada en el código del protocolo argumentativo y deben de encontrarse en el agente argumentativo genérico para que toda especialización del mismo las tenga. La forma de traducir esta parte será crear un archivo que contenga todas estas definiciones relativas al protocolo para que todo tipo de agente que las requiera pueda acceder a ellas con facilidad, incluido el CS.

2. Módulos CBR: Estas variables son las usadas para contener ambos módulos CBR. Tal y como se ha dicho ya, estos componentes deben forma parte de todos los agen-

te argumentativos sin excepción, por lo tanto formarán parte del agente argumentativo genérico. La personalización de los criterios de semejanza entre casos será gestionada por medio de ficheros de configuración, que se asocian en el momento de la construcción del agente a los módulos CBR.

3. Pesos: Estas variables incluyen aquellas usadas para determinar las tácticas argumentativas de los agentes, tal cual se ha explicado en la sección 3.2.2. También incluye otras variables de índole similar que aportan niveles de variabilidad al conjunto. Estas variables también deben ser comunes a todos los agentes, por lo tanto formarán parte del agente argumentativo genérico. El valor de estos pesos será asignado en la creación de la instancia del agente, no obstante, será posible modificar estos valores durante la ejecución del mismo, dando lugar a la posibilidad de emplear tácticas dinámicas.

4. Otras variables: El resto de variables formarán parte del agente argumentativo genérico porque son necesarias para su funcionamiento en general, ya sea porque son necesarias para el protocolo argumentativo, porque son utilizadas durante el proceso de gestión de la argumentación o porque sirven para mantener información sobre el sistema. Estas variables son asignadas en el constructor del agente y se utilizan para diferenciarlo o bien se inicializan en este y son modificadas por métodos de la clase. Son de ámbito privado y no se debe tratar de modificarlas externamente a la clase del agente.

En lo que a la sección inferior se refiere también hay ciertos puntos a detallar. Hay tres tipos de métodos según su uso: los métodos imprescindibles del protocolo argumentativo, los métodos opcionales del protocolo argumentativo y los métodos auxiliares. Cada uno de ellos requiere un enfoque concreto:

- Métodos imprescindibles del protocolo argumentativo: Son los métodos que deben estar sí o sí en el agente argumentativo genérico, dado que todos los agentes argumentativos tienen que ceñirse al protocolo argumentativo y estos métodos lo conforman. Estos métodos son los que comparten nombre con los estados de la FSM en la figura 2.3. Algunos de estos métodos contienen código que no es completamente indispensable para el correcto funcionamiento del protocolo. Este código será trasladado al siguiente tipo de métodos. Estos métodos tendrán una implementación fija, pero podrán ser sobreescritos por aquellas cases que extiendan del agente argumentativo genérico, siempre y cuando el protocolo se siga respetando.

- Métodos opcionales del protocolo argumentativo: Son los métodos que se ejecutan cuando el agente entra o sale de un estado del protocolo y que ejecutan código no esencial. Estos métodos también pueden ser vacíos o contener únicamente código de depuración, si así se desea. Estarán definidos como métodos abstractos en la clase raíz para un diseño más conciso.

- Métodos auxiliares: Estos métodos contienen código que es necesario para el funcionamiento general del agente. Es un caso similar al de las variables que no han sido marcadas en la sección superior, serán incluidos en la versión genérica del agente argumentativo y su ámbito será privado con respecto a la clase.

Este diseño escogido hace posible realizar una traducción global de la clase del agente al nuevo lenguaje y posteriormente crear la clase del agente genérico. Se va a optar por esta opción porque, de esta forma, resulta mucho más sencillo atender si se está realizando una traducción correcta. Una vez confirmado el correcto funcionamiento de la versión traducida tocará hacer la transformación.

5.6 Diseñando la interfaz

La interfaz propuesta en el capítulo anterior es sencilla, pero requiere igualmente de un proceso de diseño. En esta sección se van a mostrar los diseños previos a la implementación de la interfaz gráfica de usuario para la infraestructura. Para desarrollar el contenido que sigue se ha acudido a conocimientos adquiridos en IPC, como los relativos a la realización de «mockups» o bosquejos. La herramienta empleada para realizar estos ha sido Figma y los resultados son los mostrados en las figuras 5.4, 5.5, 5.6 y 5.7.

Figura 5.4: Bosquejo de la interfaz para la configuración de los experimentos (añadir agente)

Figura 5.5: Bosquejo de la interfaz para la configuración de los experimentos (modificar agente)

Los cuatro bocetos han sido creados en base a unas reglas establecidas para solventar una necesidad clásica. Esta necesidad ha sido el foco de muchas investigaciones a lo largo de la historia de la informática y lo sigue siendo hoy en día. Esta consiste hacer más accesible la interacción de los humanos con las máquinas. Las mencionadas investigaciones han adquirido carácter multidisciplinar al importar ideas de, por ejemplo, la psicología. Todas estas investigaciones pueden ser ahora aprovechadas para realizar mejores y más eficientes interfaces. Para el diseño de los cuatro bocetos se han aplicado algunos de los conocimientos más básicos extraídos de esta y van a ser comentados en adelante.

Figura 5.6: Bosquejo de la interfaz para la visualización de los experimentos (elementos contraídos)

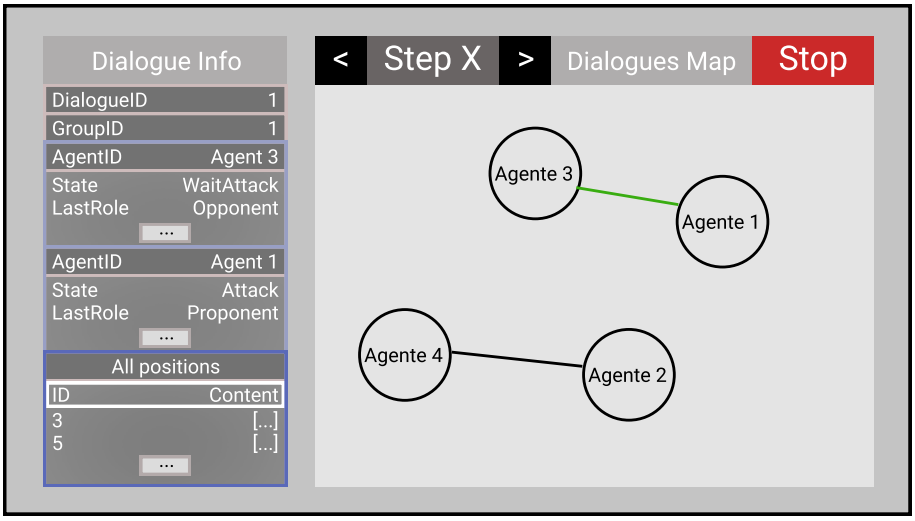
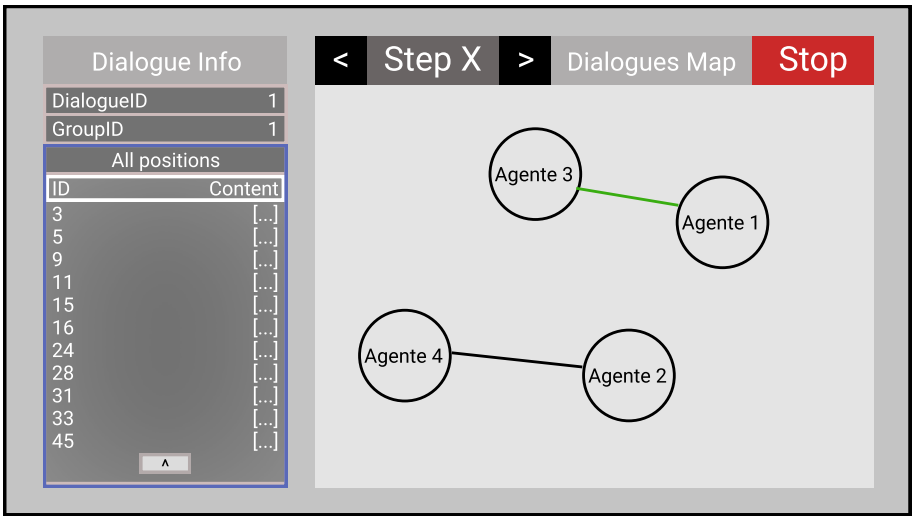


Figura 5.7: Bosquejo de la interfaz para la visualización de los experimentos (elemento desplegado)



Como ya se ha hablado, el programa debe ser fácil de manejar para usuarios avanzados, pero también para personas menos especializadas que simplemente busquen un uso educacional o recreativo. Dada la simplicidad del programa con respecto al número de actividades que permite hacer se ha optado por un diseño de ventana única que permite una navegación sin complicaciones. Esta decisión tiene un impacto con un ligero peso negativo en el diseño: la mayor parte de la ventana es ocupada por un gran recuadro que, si bien es imprescindible para el funcionamiento de la parte de la interfaz relativa a la visualización, para el caso de la configuración se podría haber simplificado, dejando un mayor hueco para distribuir mejor los distintos campos de configuración.

Una de las cuestiones que se han tenido en cuenta es la regla del «7+-2». Esta es una regla psicológica que dicta que una persona media es capaz de tener en mente hasta siete ideas al mismo tiempo, con un error admisible de dos ideas, es decir, una persona con pocas capacidades de concentración puede manejar alrededor de cinco conceptos al mismo tiempo, mientras que una persona que destaque en este aspecto puede ser capaz de gestionar hasta nueve conceptos a la misma vez. Esta regla es mucho más determinante en sistemas que requieran de un manejo en tiempo real, pero, pese a no ser el caso de

este programa, tenerla en cuenta es también beneficioso. Los cuatro bosquejos tratan de aplicar esta regla en busca de facilitar el aprendizaje sobre su manejo. Para explicar esto mejor, estos son los elementos a tener en cuenta de la interfaz en su conjunto:

- El panel de configuración: Este panel contiene los campos para rellenar la información de los agentes a incluir y está disponible cuando la interfaz se presta para la configuración de los agentes. Se encuentra en el lado izquierdo de la ventana y comienza estando vacío, pidiendo introducir los distintos valores. Estos campos para los valores están separados siguiendo una lógica que busca facilitar el trabajo al usuario. El primer campo es el correspondiente al valor único de los agentes, su identificador, y el segundo sirve para determinar a que grupo de argumentación pertenecerá el agente. Seguidamente están los campos que permiten definir la estrategia argumentativa a seguir por el agente, estos campos están agrupados por pertenecer al mismo ámbito. Los siguientes campos que se aprecian en los dos primeros bocetos son los correspondientes al tipo de agente y las rutas a los ficheros donde se encuentran los archivos que contienen las bases de casos para construir los módulos CBR del agente. Al igual que estos, el resto de campos son los requeridos para la construcción de un agente. Dependiendo del tipo de agente escogido pueden aparecer más campos en función de la especialización que se haya realizado del agente argumentativo genérico. Estos campos son accesibles a base de desplazarse hacia abajo, el llamado «scrolling». El panel de configuración tiene dos versiones, la que permite añadir nuevos agentes, cuyo título es «Add Agent» y que está representada en la figura 5.4 y la que permite modificar agentes seleccionados, cuyo título es «Edit Agent» y que está representada en la figura 5.5. Ambas versiones tienen la misma distribución, pero varían en el tipo de botón. El panel de edición permite borrar un agente, mientras que el de adición permite añadirlos. La ausencia de un botón de confirmación en el panel de edición implica que las modificaciones son guardadas automáticamente.

- El panel de información: Este panel muestra la información de los agentes durante un paso concreto de la ejecución y está disponible cuando la interfaz se presta para la visualización del proceso argumentativo, desencadenado a raíz de la configuración del problema. Sustituye al panel de configuración en la parte izquierda de la ventana bajo el título de «Dialogue Info». Se puede apreciar en las figuras 5.6 y 5.7 que se muestra información con los elementos desplegables contraídos y abiertos respectivamente. Todos los campos de información que no se encuentran a la vista se pueden acceder también desplazándose hacia abajo, el hecho de contraer los desplegables facilita la navegación cuando hay datos que son de más interés que otros. Este panel informativo muestra los datos del diálogo seleccionado en el panel de visualización.

- El panel de visualización: Este panel se encuentra a la derecha de la ventana, pero es tan grande que sobrepasa la mitad de la misma. El panel tiene funcionalidades distintas dependiendo de si el usuario está configurando el experimento, o si ya lo está visionando en ejecución. La primera situación se corresponde con las figuras 5.4 y 5.5, mientras que la segunda con las figuras 5.6 y 5.7. Durante la primera funcionalidad el panel de visualización puede ser usado para observar los agentes que ya han sido añadidos y también seleccionarlos. En cambio, en la segunda funcionalidad es posible observar los agentes y los diálogos establecidos entre ellos en un momento concreto de la ejecución, representado como paso. Estos diálogos, representados como aristas entre los agentes, pueden ser seleccionados para mostrar sus datos en el panel de información.

- El panel de control de los pasos: Este panel no es muy grande e incluye una funcionalidad muy simple, sirve para desplazarse por los distintos momentos de la ejecución. Las instantáneas del sistema que se muestran en el panel de visualización durante la ejecución del problema argumentativo son extraídas para cada paso, siendo un paso un cambio en el estado de alguno de los agentes. Este panel, consistente en el número del paso actual y dos botones en forma de flecha, permite cambiar el momento que se está mostrando. Presionar la flecha que apunta hacia la izquierda hace retroceder un paso, por otro lado, peesionar la flecha que apunta a la derecha provoca que se adelante un paso. Este panel solo es funcional en el modo de visualización del experimento, en el modo de configuración aparece deshabilitado y el paso de ejecución es el número cero.

- El botón para comenzar y concluir: Este es el panel más simple, sirve para cambiar de un modo a otro. Cuando la configuración ha sido terminada presionarlo provoca que comience la ejecución. Cuando se está visualizando la traza de la ejecución presionarlo provoca volver al modo de configuración. Tiene colores y etiquetas distintas dependiendo del modo, será rojo y «Stop» durante la visualización y verde y «Start» durante la configuración.

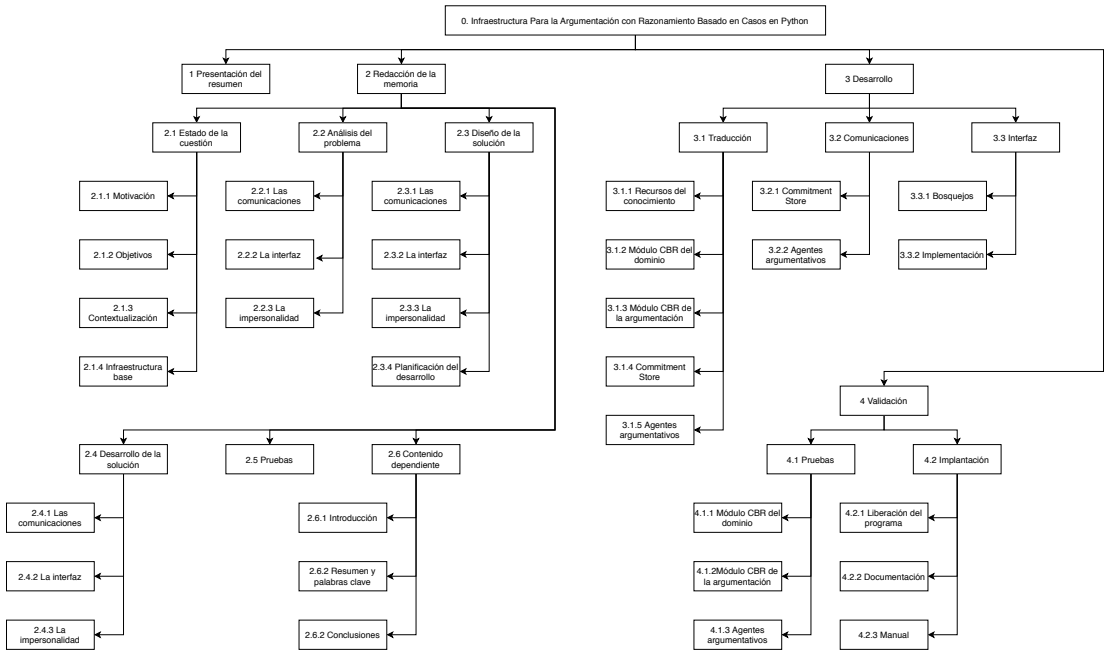
Son muchas más las formas de diseñar la interfaz, pero dada la simplicidad del objetivo se considera suficiente con la representación de estos bocetos. Otra opción interesante suele consistir en hacer una representación gráfica del flujo de la aplicación, representando las transiciones entre las distintas ventanas del programa, pero, como ya se ha comentado, por razones de simplificación del diseño, el programa funciona con una ventana única, por lo cual sería de poca utilidad esta representación. Eso sí, el programa mostrará ventanas de diálogo para informar de errores, datos no introducidos o ciertos detalles del interés del usuario. Por último, este diseño existe para marcar el camino a seguir, pero no es intocable, más bien todo lo contrario, los errores, apreciaciones o ideas de mejora que vayan surgiendo a lo largo del desarrollo deben ser tomados en consideración.

5.7 Definiendo los plazos

Tal y como enseña «Gestión de Proyectos», o GPR, la planificación es un punto clave que diferencia a un proyecto exitoso de uno de los tantos que no llegan a ningún puerto. Hasta ahora se han usado los conocimientos en planificación para enfocar el proyecto en la buena dirección. El capítulo en el que se propone la situación es en sí mismo la definición del alcance del proyecto y las secciones de este mismo capítulo previas a la actual son una definición globalizada de las actividades a realizar. En esta sección en concreto se va a definir una estructura de desglose de trabajo, o EDT, en la que se va a dividir el trabajo a realizar en base a las actividades definidas. La estructura va a consistir en una serie de paquetes de trabajo, representativo de las diferentes subactividades a realizar. Un conjunto de paquetes marcan un hito o entregable. Si bien el término entregable no está recogido en el diccionario, va a ser empleado para representar estos hitos al ser terminología de la asignatura mencionada. Estos entregables se reparten a lo largo del trabajo para marcar el ritmo del desarrollo, el final del proyecto viene marcado por el entregable final.

La EDT mostrada tiene como objeto enfrascar todo el proyecto en un único diagrama. De hecho, la estructura diseñada muestra también las actividades relacionadas con el desarrollo de la memoria del trabajo, no únicamente el proyecto software. Esta estructura se muestra en figura 5.8 y en ella se pueden ver todos los paquetes de trabajo. Si se observa en detalle, se pueden percibir ciertos aspectos que pueden no encajar com-

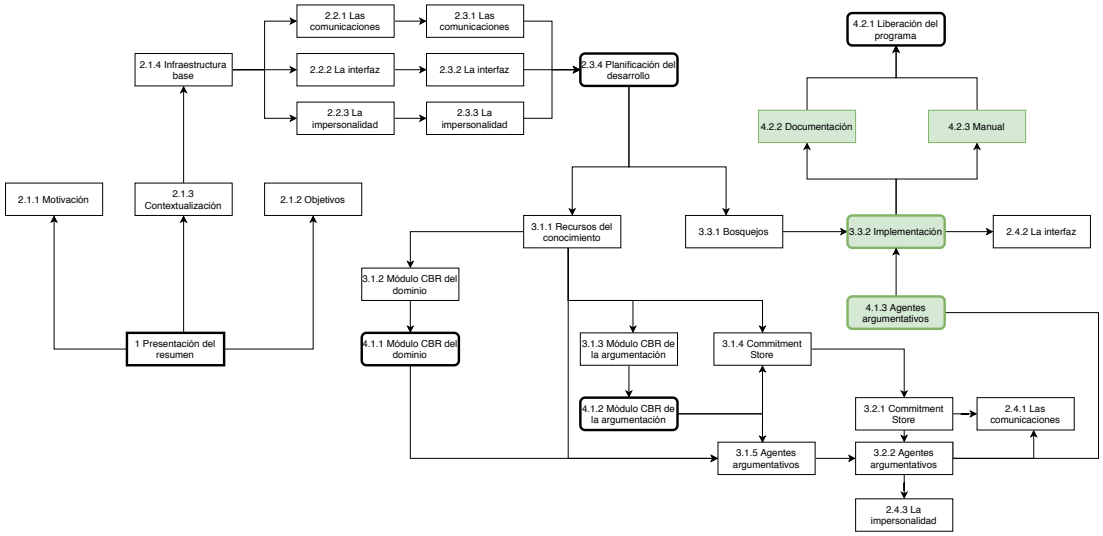
Figura 5.8: Estructura de desglose del trabajo para el trabajo de fin de grado



pletamente con los visto hasta ahora. Un paquete de trabajo se concibe como la unidad mínima a la que puedes reducir una parte de la tarea, por ejemplo, en un proyecto de viaje comprar los billetes y desplazarse al aeropuerto serían paquetes de trabajo. En las EDT, los paquetes de trabajo son las hojas del árbol. La circunstancia es, que en programación esta subdivisión puede no terminar nunca, es por eso que hay paquetes de trabajo, como el 3.1.1, que consisten en la implementación de ciertos módulos del código que bien podrían separarse por clases e incluso líneas, pero podría ser contraproducente. El otro tema a tener en cuenta tiene que ver con el contexto en el que fue diseñada esta EDT, más concretamente con el momento, que es el causante de los aspectos que se ha dicho, no encajan completamente. Esta estructura ha sido creada para ilustrar esta sección cuando el proyecto ya llevaba en marcha un buen tiempo, por lo que hay algunos paquetes que parecen no corresponderse. La gran diferencia se nota en los puntos relativos a la redacción de la memoria, que son tal y como se plantaron, pues se basan principalmente en la normativa a seguir y los puntos en relación al desarrollo que, como se ha comentado previamente, no pueden ser iguales a como se plantearon en un principio dado que al tiempo de comenzar el desarrollo se decidió realizar la migración del proyecto a otro lenguaje.

Una vez definida la EDT es el momento de definir cuáles son las dependencias entre las actividades. Esta es una parte importante pues hay actividades que no pueden ser realizadas antes que otras, siendo que existen unas relaciones de dependencia que es necesario respetar. Esta restricción es más importante cuando se trata de grupos colaborativos en los que participan varias personas e interesa trabajar en concurrencia. De todos modos, tener estas relaciones controladas puede prevenir quebraderos de cabeza y fuerza de trabajo desaprovechada. En la figura 5.9 se han reaprovechado las hojas del árbol de la ETS para definir dichas relaciones. Algunas actividades tienen nombres repetidos, pudiendo llevar a confusión, pero es muy sencillo de seguir el esquema fijando la atención en los índices de las hojas, que se corresponden con los de la figura 5.8, que está completamente estructurada. Se han añadido varios tipos de marcas para hacer énfasis

Figura 5.9: Dependencias entre las unidades de trabajo de la EDT



sobre unas actividades concretas. La marca con un grueso borde negro y rectangular es la que indica el inicio de todo el proceso, el desarrollo del proyecto dependía de la aceptación del resumen presentado. Luego está el caso de las actividades con borde grueso y contorno redondeado, Eestas están señaladas como los entregables del proyecto. Los plazos se establecen alrededor de los mismos, siendo el entregable final la liberación del programa. Por último, las actividades coloreadas son las que hacen referencia a los objetivos planteados al principio del documento, es decir, cuando una de estas actividades sea completada implicará haber alcanzado alguno de los objetivos.

Estas han sido, a grandes rasgos, las técnicas de gestión empleadas en el proyecto. Quedan algunas partes sin especificar y las dudas de si estas son suficientes para garantizar la estabilidad del desarrollo, dichas cuestiones serán respondidas al comienzo del próximo capítulo. Esta decisión esta orientada a dar justificación a ciertos sucesos que se han dado a lo largo del desarrollo y en los que se hace hincapié en el capítulo que sigue.

CAPÍTULO 6
Desarrollo de la solución

En este capítulo se detalla cómo se ha pasado de la propuesta inicial a la solución final en base a la aplicación de la solución diseñada. Los problemas y dificultades encontradas, las decisiones que se han tenido que tomar, particularidades de la solución final, todos estos temas son del interés del capítulo. El sistema completo no va a ser mostrado, por razones obvias, en cambio se va a prestar especial atención a los aspectos más destacables del sistema, estos son aquellos más relevantes. novedosos, complejos y (o) esenciales para entender el funcionamiento de la plataforma. En este capítulo se puede ver claramente que las cosas no siempre surgen como se plantean y que tal vez hubiese sido conveniente emplear más técnicas de las aprendidas en GPR, como la administración del tiempo con colchones. Sobre esto último trata la primera sección. En ella se contextualizan los efectos de la planificación y las circunstancias excepcionales sobre el desarrollo del proyecto, de esta forma en las siguientes secciones se podrá centrar mejor la atención sobre sus respectivos temas sin necesidad de irse deteniendo para explicar qué motivos llevaron a que ciertas partes no terminaran llevándose a cabo como fue primeramente planteado.

6.1 Efectos de la planificación y el contexto social en el desarrollo

Ya se ha mencionado en este documento la importancia de la planificación y lo relevante que es a la hora de determinar las posibilidades de un proyecto de ser concluido con éxito. En este proyecto en concreto se han concentrado problemas relativos a la planificación, contratiempos inesperados y errores de cálculo que han repercutido enormemente en el desarrollo del mismo. Para ilustrar uno de los mayores fallos en la planificación y sin duda, el que mayores consecuencias a traído, se ha creado la tabla 6.1.

Tabla 6.1: Planificación temporal sobre la traducción de la infraestructura a Python

Reparto del tiempo		
Componentes	Estimación (h)	Realidad (h)
Recursos del conocimiento	3	30
Módulo CBR del dominio	2	10
Módulo CBR de la argumentación	3	7
Commitment store	1	5
Agente argumentativo	5	10
Total	14	62

De esta forma, el problema es muy fácil de identificar, el tiempo requerido excedió de forma desmesurada el tiempo estimado. Además, las horas representadas en la columna del tiempo real invertido son una aproximación del esfuerzo que conllevó escribir el código en base al análisis de las clases en Java, pero no tiene en cuenta la energía invertida en buscar documentación sobre Python y las formas de sacar el mayor potencial posible mediante otras tecnologías. Cuando se buscan las causas se pueden encontrar una serie de ellas. Primero de todo, Python es un lenguaje sobre el que no se poseía demasiada experiencia, fue elegido con entusiasmo al considerarlo, aparte de beneficioso por las cualidades ya descritas en capítulos anteriores, una buena herramienta para ampliar el conocimiento sobre los lenguajes de programación. La idea era buena, pero las consecuencias son palpables. Por otro lado está la búsqueda de un mayor aprovechamiento de las cualidades del lenguaje, tratando de aplicar herramientas menos convencionales como las ya explicadas anotaciones de tipos y mecanismos de ingeniería inversa. Otra de esas circunstancias es el hecho de que, pese a ser lenguajes con una base sintáctica parecida, las estructuras de código entre Python y Java no son perfectamente intercambiables, lo cual implica que se requiere de un extenso análisis de las dependencias de las secciones a traducir, provocando la necesidad de revisar extensas cantidades de código para evitar cometer errores semánticos difíciles de detectar. Y para terminar, algunas partes del código original presentaban pequeños errores lógicos y a menudo se prestaban a recibir cambios, acciones que requieren, por supuesto, de tiempo extra.

Otro aspecto de interés que afectó durante el transcurso del proyecto fue el contexto en el que este fue desarrollado. Como se ha mencionado ya una vez, la propuesta del trabajo surge de los tutores del mismo. El alumno encargado de llevar a cabo el desarrollo estaba sujeto a un contrato de trabajo en el cual debía invertir quince horas semanales al trabajo. Esta condición no siempre se pudo cumplir, pues los horarios universitarios y el calendario de exámenes se llevaban parte de ese tiempo. Estos retrasos se tuvieron que compensar en los meses de mayo y junio en los cuales el tiempo disponible podía ser invertido en su totalidad en el proyecto, sin embargo, el trabajo se había acumulado y, claramente, no es igual de eficiente repartirse el trabajo en cortos periodos semanales que acumular toda la tarea pendiente en un par de meses. La perspectiva de gestión del tiempo se pierde y todo se vuelve una carrera contrarreloj.

Se dio también la situación excepcional de una pandemia que asoló al país derribando todos los sistemas de planificación planteados por prácticamente todos los trabajadores de casi cualquier disciplina, obligándoles a replantearse las estrategias. No obstante, no se considera que haya sido determinante en el desarrollo del proyecto, porque si bien fue un sonado inconveniente, el fallo principal reside en las erróneas expectativas creadas en la planificación. Teniendo en cuenta todas las circunstancias expuestas en este capítulo será posible tener un mejor entendimiento de las razones que llevaron a las siguientes partes a desenvolverse como lo hicieron.

6.2 Implementación del código

La columna vertebral del proyecto es el código de la infraestructura. Este ha sido escrito empleando Pycharm como IDE. Pycharm ofrece una serie de características altamente útiles para el desarrollo en Python, como la integración nativas de herramientas de control de versiones, un cómodo autoguardado y más aspectos en relación al lenguaje, como: herramientas de predicción de código, recopilación de los tipos de las variables en ejecución y una cómoda compatibilidad con Anaconda. Gracias a esta compatibilidad se hace posible gestionar muy cómodamente los paquetes incluidos en el proyecto, en

gran medida por la posibilidad que brinda de crear entornos específicos para proyectos concretos. Los tres paquetes externos principales son:

- SPADE: Este paquete contiene la plataforma que se empleada para dar soporte a las comunicaciones. Se utiliza en la implementación del CS y del agente argumentativo, ambos extensiones del agente SPADE. Los agentes se ciñen a unos comportamientos que permite definir este módulo. Desde estos comportamientos es posible enviar mensajes partiendo de un remitente hacia un receptor conteniendo un un cuerpo y los respectivos metadatos. Para dar soporte a mensajes de múltiples receptores, elemento propio de Magentix2 utilizado en la versión de Java, se ha realizado una especialización de la clase «Message» del paquete, que representa los mensajes que pueden ser enviados.

- Loguru: Este paquete contiene el sistema de gestión de registros de impresión que se emplea en el proyecto. Loguru viene a reemplazar al «Logger» de la versión de Java. Al igual que este último, permite imprimir mensajes de registro de diferentes tipos. Esto tipos son, principalmente: información, alerta, error y excepción. También permite aplicar filtros para los mensajes y capturar solo información específica. Se usa en todas las clases de las que interese obtener información y se emplea para depurar partes del programa que la interfaz no permitiría.

- Pytest: Este paquete contiene el sistema que se emplea para la realización de los tests unitarios. Pytest viene a sustituir a JUnit, que era la herramienta dedicada a los tests unitarios en las versiones de Java. El funcionamiento es muy similar en ambos, por lo que la traducción de los tests se vuelve una opción interesante. El uso que se le da es básico, se determinan una serie de condiciones a cumplir mediante comandos «assert». Si dichos comandos no reciben el valor esperado el test graba un error. Es empleado en la mayoría de los tests realizados sobre los entregables, especialmente los que consisten en módulos puramente software.

El entorno de «Conda» registra las dependencias de paquetes necesarias y en combinación con las herramientas de control de versiones de Pycharm permite la compartición de código instantánea y sin demasiadas complicaciones. Esto ha sido útil en este proyecto al hacer posible que los tutores tuviesen acceso rápido al código que el alumno iba desarrollando según este hacía las subidas a la rama de desarrollo del repositorio Git.

Para implementar el código el orden de traducción de los diferentes módulos fue el propuesto desde un principio. En correspondencia con lo planteado en el capítulo anterior, primero se tradujeron los KR, junto a las funciones métricas y de configuración. Posteriormente se tradujo el módulo CBR de dominio y se probó la estructura hasta ese punto, para seguir con la traducción del módulo CBR de argumentación y sus tests unitarios correspondientes. En penúltimo lugar, se escribió el código para el CS y finalmente el del agente argumentativo. Algunos errores de la versión original fueron corregidos durante la traducción, como, por ejemplo, el relativo al almacenamiento de las veces que un caso de argumentación había sido usado. Este error consistía en lo siguiente. Al crearse el componente CBR de argumentación como variable del agente a partir de un archivo con los casos iniciales, se van cargando uno a uno todos los casos. Si se detecta un caso repetido, este no es introducido, pero se aprovecha para incrementar la información disponible del mismo si cuenta con nueva información como, por ejemplo, con premisas que no están presentes en el que se encuentra ya en la base. El problema estaba en el momento de combinar las veces que un caso de argumentación había sido empleado. Mientras que en la versión de Java simplemente se incrementaba en uno el número de veces por caso combinado, en Python se corrigió para que el resultado fuese el valor de la suma de las veces que habían sido usados ambos casos (el de la base y el que se trataba

de introducir). Otros cambios con respecto al código original se han dado en forma de optimizaciones, sobretodo en operaciones con listas, en las que Python se desenvuelve muy bien, de hecho, es capaz de realizar más acciones en menos líneas de código fuente.

El orden establecido es como es por dos razones principales. La primera es la ineludible, las dependencias explicadas en el capítulo anterior. Hay módulos que requieren de otros para poder funcionar, de hecho, todos los módulos dependen de los KR, sin ellos no se pueden implementar. Y la segunda es por favorecer la adaptabilidad. Como se puede ver en la tabla 6.1 la tarea que más tiempo llevó finalizar fue la primera que se realizó. Aunque otras tareas fuesen más largas y complejas, como es el caso del agente argumentativo, tomaron menos tiempo porque ya había una fuerte exposición al lenguaje y el conocimiento sobre la infraestructura era considerablemente mayor. Por esta razón se priorizó traducir primero el CS sobre el agente argumentativo ya que presentaba una versión mucho más simple de las comunicaciones, adaptando la curva de aprendizaje para optimizar mejor los tiempos.

En conclusión, esta implementación ocupó una cantidad desmedida de tiempo provocando, como se va a ver más adelante, que otras partes del proyecto se resintieran. Aún así, hubo ciertos elementos beneficiosos que no se esperaba obtener, como la corrección de errores presentes en la versión original y la optimización de diversas operaciones y del tamaño del código, que ahora es más legible. Es decir, al final del proceso de traducción se obtuvo el resultado esperado, incluso mejor, pero el camino fue más empinado que el planteado.

6.3 Trabajo futuro

El capítulo se ha presentado de forma que podía parecer que tras la justificación de la planificación se iban a suceder tantas secciones como las que había en el capítulo de la solución propuesta, cada una conteniendo información de hasta dónde había llegado su desarrollo, pero, en cambio, se ha optado por combinar las secciones de trabajo inconcluso en una sola en la que, a su vez, se plantean las actividades restantes para su finalización.

6.3.1. La generalización

La parte de la generalización está prácticamente terminada, porque, como se ha justificado en varias ocasiones, la solución propuesta estaba planteada de manera que fuera posible desarrollar primeramente una traducción completa del agente argumentativo para después, con los esquemas ya creados de las partes de código traducido generalizables y propensas a su edición, crear la clase del agente argumentativo genérico.

Con el agente argumentativo traducido, el siguiente paso es, como se ha dicho, crear la clase genérica. A falta de tiempo para hacerlo se ha decidido no emprender en el intento ante la posibilidad de dejarlo como un trabajo a medias. Quizá podría no tomar demasiado tiempo, pero sería correr un riesgo demasiado grande puesto que, pese a que el agente argumentativo ha sido traducido, protocolo incluido, este no ha podido ser probado. El trabajo futuro a este respecto es, entonces, realizar las pruebas de funcionamiento correspondientes al agente argumentativo y posteriormente crear la clase general. Una vez terminado, podría pensarse en formas de ampliar el conjunto, como crear una serie de implementaciones del agente argumentativo que hiciesen las veces de ejemplos.

6.3.2. La interfaz

La interfaz es otro apartado que no se ha llegado a desarrollar en su plenitud. Este en concreto se ha quedado bastante lejos de ser finalizado. La parte de la conceptualización ha sido terminada, como ya se ha mostrado, pero, aunque es una parte de bastante carga en el desarrollo de la interfaz, esta lejos de ser la que más tiempo y esfuerzo requiere. En lo que a la interfaz web concierne, queda pendiente todo el trabajo relativo al desarrollo e implementación. Lo primero de todo será decantarse de forma definitiva por la herramienta con la cual trabajar los lenguajes correspondientes (HTML5, JavaScript y CSS) y lo segundo será realizar la implementación. Una vez implementada será momento de comprobar su funcionalidad y, si se considera oportuno, realizar tests de usabilidad, pero esto, como se va a explicar en una futura sección (8.3.2), no parece una idea razonable.

Los plazos han resultado ser apurados para la entrega del trabajo de fin de grado y su correspondiente memoria (este documento), motivo que, combinado con los fallos ya comentados de planificación, ha impedido llevar a cabo todos los objetivos dentro del plazo estipulado. Sin embargo, estas resoluciones son comunes en el mundo del desarrollo software como bien se ha comentado repetidas veces a lo largo de la carrera y gracias a la buena organización en el resto de los aspectos, retomar el trabajo en un futuro puede resultar muy sencillo, incluso para alguien que no haya estado directamente involucrado. A corto plazo, el proyecto seguirá recibiendo atención hasta finales de julio, que es cuando termina la beca formativa. Lo que sucederá después no está planificado todavía. Si hay algo claro y positivo que extraer es que terminar esta fase del proyecto dejando una versión estable de la plataforma facilita retomar el desarrollo en un futuro.

CAPÍTULO 7
Implantación

Este capítulo concentra su información en torno a la fase posterior al desarrollo, la fase de implantación. Esta fase suele ser conocida como aquella en la que se lleva el desarrollo realizado a la explotación. Para ello, suele ser necesaria la instalación y puesta en marcha (o en producción) del sistema desarrollado. En lo que refiere a este proyecto, al basarse principalmente en un trabajo de investigación, llevar el proyecto a explotación implica hacerlo público para que la gente interesada pueda aprovechar su potencial. Como no es un sistema que funcione de manera distribuida o que requiera de un acceso en línea para su utilización, no requiere de una puesta en marcha. Abreviando, la implantación del proyecto va a consistir en hacer pública y accesible su utilización y para ello es necesario habilitar una forma de obtener el software y las indicaciones de como emplearlo.

Al ser la fase final del proyecto requiere para su completitud total que todas las actividades que la preceden estén enteramente terminadas. Como se ha explicado en el capítulo anterior no se ha podido garantizar tal premisa, por lo que alguno de los objetivos se ha visto resentido. No es un problema grave, si se tiene en cuenta que el proyecto seguirá recibiendo actualizaciones a futuro, implicando nuevas versiones que suplan esta carencia. Este capítulo va a ser breve, pero no por ello menos importante.

Lo primero de todo es hacer el código fuente accesible. Las posibilidades son varias, desde crear una página web específica, a usar repositorios web. Por razones de tiempo y por ser la medida usualmente escogida por la mayoría de desarrolladores de programas de código y acceso abiertos, se ha optado por alojar el código en un repositorio. Este repositorio está en el ya mencionado GitHub[1] y es de hecho, el repositorio GIT sobre el cual se ha realizado el proyecto. En adición, para facilitar su incorporación como paquete externo a otros proyectos, ha sido subido al índice de paquetes de Python pypi[2].

La parte referente a la accesibilidad que más resentida se ha visto ha sido la creación de un manual, que era uno de los objetivos principales. Por lo tanto, un manual de uso propiamente dicho no va a estar disponible, sin embargo, todavía es posible crear la documentación. Tal y como se aclamaba en los objetivos y se ha demostrado a lo largo de los capítulos de análisis del problema y desarrollo de la solución, los comentarios de los métodos y clases empleados permiten la generación de documentación de manera automática. Esta documentación es accesible desde el GitHub[3].

En definitiva, hacer accesible el código fuente y la documentación sobre la infraestructura es lo que habilita definitivamente su aprovechamiento. La versión de código y documentación disponible se queda un poco por detrás de lo que se esperaba aportar, pero no hay que pecar de negatividad. Una de las cualidades más reseñables de la forma

[1] https://github.com/jaumejordan/pyargcbr
[2] https://pypi.org/project/pyargcbr/
[3] https://pyargcbr.readthedocs.io/en/latest/

en la que el proyecto ha sido implantado o lanzado, es que es perfectamente escalable. Con el sistema de control de versiones, el mismo repositorio GitHub, es posible continuar el desarrollo sin ensuciar lo ya disponible para, posteriormente, liberar nuevas versiones actualizadas que contengan las características que no se han podido alcanzar con la fecha límite establecida. Además, la generación automática permite crear documentación de forma dinámica, simplemente añadiendo comentarios a nuevas clases y métodos, o modificando los previos. Lo que quiere decir es que este ha sido solo el comienzo.

CAPÍTULO 8
Pruebas

En este capítulo se presentarán las pruebas realizadas para verificar que la solución funciona correctamente. Lo estipulado era realizar una prueba para cada entregable pautado en la planificación correspondiente a los eventos del desarrollo en sí mismo, es decir, sin incluir los dos primeros entregables (presentación del resumen y planificación del desarrollo). Es por este motivo que el capítulo se divide en las siguientes secciones, cada una en referencia a una de las pruebas realizadas.

8.1 El CBR del dominio

Las pruebas realizadas sobre el módulo CBR del dominio fueron las primeras formalmalmente realizadas. Esto es, previamente y a lo largo del resto del desarrollo, se realizaban pruebas menores para comprobar el funcionamiento de unidades compactas de código o la funcionalidad de alguno de los paquetes, nada que ver con la escala de las pruebas formales. El objetivo de estos tests no es únicamente asegurar la correcta implementación del componente CBR, también es importante asegurarse de que los KR están bien definidos. Para realizar los tests unitarios se hace uso de la biblioteca Pytest y se basans en los tests que se realizaron originalmente sobre el código Java[1]. Estos tests buscan comprobar que se pueden realizar con el módulo todas las acciones que son requeridas; el motivo de la necesidad de estas cualidades guarda relación a lo explicado en la sección 2.1.2. Estas características son precisión y consistencia en la recuperación, control de duplicados y operatividad. En las siguientes subsecciones se va a entrar más en detalle.

8.1.1. Precisión en la recuperación

El test de precisión está enfocado a probar los métodos de recuperación de casos desde la base. Tal y como se ha descrito en capítulos anteriores, el agente debe de ser capaz de, partiendo de un caso del dominio, extraer de su base de casos del dominio una serie de estos considerados según su similitud al caso dado. Este método está implementado en el código para devolver una lista ordenada de casos según su similitud. La similitud se indica como un valor normalizado en el que 1 implica que son casos completamente iguales y 0 que son totalmente diferentes. Los criterios, como se ha indagado antes (5.5.2), son dependientes de la métrica empleada, en este caso el algoritmo empleado es la similitud euclídea normalizada. La función permite marcar un listón de similitud para filtrar casos que no son deseados, es decir, si este valor es 1 entonces el método solo devolverá casos exactamente iguales al que se le pase, si es 0, todo vale.

[1] Los tests originales están alojados en un repositorio privado http://gitlab.gti-ia.upv.es/jjordan/argcbr/blob/master/trunk/ArgCBRGeneric/src/test/DomainCBRTests.java de GitLab.

Para cada elemento en la base de casos del dominio se comprueba la precisión en la recuperación y si es exitosa en todos se considera que pasa la prueba. La prueba realizada a cada caso de la base es la que se describe a continuación:

1. Se pide al módulo CBR que devuelva todos los casos que almacena ordenados por su similitud con el caso a analizar, el límite de similitud se establece en 1, por lo tanto solo serán devueltos los casos exactamente iguales al elemento bajo análisis.
2. Se toman uno a uno todos los casos devueltos por el CBR y se extraen las premisas asociadas a su contexto.
3. Estas premisas serán comparadas una a una y por orden con las premisas del caso a analizar. Se realiza un «assert» por cada comprobación de similitud entre premisas, por lo que de no ser iguales, no se pasará la prueba.

La precisión mide entonces la capacidad del algoritmo de similitud para determinar cuanto se parecen dos casos. Siempre tiene que haber por lo menos un caso (e idealmente solo uno) al estar contenido el mismo caso que está siendo analizado, por lo tanto si no se pasa esta prueba es porque existe un error de diseño o de implementación.

8.1.2. Consistencia en la recuperación

Con el test de consistencia se pretende comprobar que existe una estabilidad y coherencia en las recuperaciones de casos desde la base. El foco principal es definir y ejecutar un test que demuestre que llamadas al método de recuperación de casos con los mismos parámetros devuelve los mismos resultados. Esto es lo que el método científico define como experimentos replicables, mismas circunstancias y mismos parámetros ofrecen el mismo resultado.

Al igual que en el test anterior, para cada elemento en la base de casos del dominio se comprueba la consistencia en la recuperación y si es exitosa en todos se considera que pasa la prueba. En la base de casos del módulo los casos están almacenados por listas en un diccionario. Las claves para acceder a las listas son los identificadores de las premisas principales de los casos, siendo una premisa principal la primera premisa en la lista de premisas del primer caso en la lista de casos. Para determinar que un caso pasa la prueba se aplica un proceso similar al caso anterior, pero enfocado a la consistencia. Los pasos que componen el test son:

1. Se pide al módulo CBR del dominio que devuelva todos los casos que tiene almacenados, sin ningún criterio añadido. Para esta prueba estos casos son recibidos usando un método distinto que los devuelve en forma de listas de casos, una por cada premisa principal almacenada. El resultado sería el mismo que recibiendo todos los casos en una única lista plana, pero de esta forma se prueba también el otro método de recuperación que incluye a todos los casos sin criterios añadidos.
2. Se toma un caso a analizar y se realizan dos llamadas al método de recuperación de casos por similitud. Los parámetros son un valor 0 en el límite de similitud y, por supuesto, el caso a analizar como objeto de las comparaciones. El hecho de marcar el límite en 0 implica que todos los casos van a ser recuperados, desde los exactamente iguales hasta los completamente diferentes.
3. Las listas devueltas por las llamadas a la función son almacenadas en variables distintas. Esta estructura permite enfrentar uno a uno todos los casos de ambas listas.

4. Se recorren ambas listas con dos bucles anidados y se realiza un «assert» sobre la variable que guarda si el caso en la primera lista ha sido encontrado en la segunda, por lo tanto, si un caso de la primera lista no se encuentra en la segunda el test no será aprobado. La intención con esto es comprobar que todos los casos de la lista correspondiente a la primera llamada están presentes en la lista correspondiente a la segunda llamada.

5. Se regresa al punto 2 mientras queden casos por analizar.

La consistencia como propiedad del módulo CBR es, entonces, una cualidad del mismo que debe garantizar que llamadas idénticas al módulo para recuperar casos similares devuelven el mismo resultado. Por último, esta propiedad debe mantenerse mientras la base de casos no varía; una evolución en el contenido de esta base puede provocar cambios en el resultado de la ejecución del método y es lo esperable.

8.1.3. Control de duplicados

El método incluido en el componente CBR del dominio a cargo de añadir nuevos casos gestiona un intento de adición intentando evitar que se formen duplicados. Esto ya ha sido explicado en secciones anteriores, es una forma de aplicar la teoría de la representación del conocimiento que apoya la idea de que el nuevo conocimiento adquirido debe ser distintivo para ser añadido, pero no debe incluirse en sustitución del anterior, sino que debe servir para actualizarlo. Por ende, una base de casos de un módulo CBR del dominio formada a base de añadir casos uno a uno no debería contener casos duplicados. El test unitario sobre de control de duplicados aplicado sobre el módulo CBR del dominio pretende probar esto mismo y al igual que en los dos test anteriores se realiza la comprobación para todos los casos de la base de casos.

La forma de determinar si el módulo pasa la prueba para un caso concreto es la más sencilla de todas hasta el momento. Como es común en estos tests, se emplean recorridos sobre listas de casos, los pasos son:

1. En la primera de las listas se introducen todos los casos de la base del módulo, los cuales se irán extrayendo uno a uno para realizar el análisis sobre ellos.

2. La segunda lista se forma haciendo una llamada al método de recuperación de casos con el caso a analizar, extraído de la lista de casos del módulo, como primer parámetro y el límite de similitud al mínimo como segundo.

3. Se elimina el primer elemento de la lista, que al estar ordenada por similitud debe ser el homólogo al caso en análisis de la otra lista.

4. La comprobación requerida ahora es tan sencilla como recorrer la segunda lista en busca de un caso que sea igual al caso analizado, pues no debería de contener ninguno con estas características. En este test el «assert» se realiza sobre el resultado de estas búsquedas en recorrido, de forma que si se da que hay un caso igual al comparado, el test fallará.

5. Se regresa al punto 2 mientras queden casos por analizar.

En definitiva, el control de duplicados busca evitar la aglomeración de información basura en la base de casos del módulo CBR del dominio. El constante manejo de casos repetidos provocaría una desmesurada escalada en el tamaño de las bases de casos, lo cual no sólo ralentizaría las ejecuciones, sino que podría llevar a errores de procesamiento.

8.1.4. Operatividad

Según el DRAE la definición de operatividad es: «capacidad para realizar una función». La prueba de operatividad del módulo pretende, básicamente, comprobar que es capaz de realizar su función, o lo que es lo mismo, demostrar que el módulo puede ser usado para realizar el razonamiento basado en casos, que es la razón por la cual recibe su nombre. Esta afirmación requiere de más comprobaciones que los tests presentados anteriormente, de hecho, este test es considerablemente más complejo que sus predecesores.

Para determinar si el módulo pasa con éxito la prueba de operatividad se realiza un procedimiento más elaborado que un simple recorrido de listas. El test se divide en pequeñas subpruebas que deben ser satisfechas en su totalidad para conceder el aprobado. La articulación de la prueba al completo es la siguiente:

1. Se toma un caso arbitrario de la base, el primero para simplificar.
2. Se crea una lista en base a llamar al método de recuperación de casos con el caso escogido como parámetro y con el filtrado por similitud al mínimo.
3. Se hace «assert» sobre la comprobación de igualdad entre el caso escogido y el primero de la nueva lista. De esta forma, si la ordenación por similitud de la función de recuperación, o la función en si misma, no funciona correctamente, el módulo no pasa el test.
4. Se crea un nuevo caso de dominio.
5. Este caso de dominio será completamente igual al tomado arbitrariamente en un principio, pero cambiando el contenido de la primera lista de su lista de premisas asociadas al contexto. En el test implementado se busca una solución sencilla, por lo que simplemente se han añadido un par de caracteres extra al final del contenido.
6. Este nuevo caso es añadido a la base de casos del módulo y se garantiza que será introducido porque, pese a que cabe la ínfima posibilidad que el nuevo caso represente un duplicado, los casos de prueba han sido creados de forma que sea admitido.
7. Una vez introducido se realiza la una prueba como en el punto 3, se recupera información de la base en referencia al nuevo caso y se comprueba que la cabeza de la lista sea exactamente igual a este. Esta comprobación se hace también bajo el efecto de un «assert», implicando las mismas consecuencias que la subprueba anterior.
8. Se realiza el punto 3 una vez más, siendo el último de los tests parciales igual al primero, pero bajo un contexto ligeramente cambiado en el que la base de casos ya no es la misma.

Para resumir, la operatividad se comprueba para garantizar que el módulo puede desempeñar las funciones para las que ha sido diseñado e implementado. En su conjunto, el aprobado del test garantiza que el módulo funciona perfectamente para recuperar casos de la base similares al dado, que funciona para casos añadidos posteriormente y que después de cambiar el estado de la base de casos se siguen obteniendo los mismos resultados correctos.

Un módulo preciso, consistente, operativo y sin casos duplicados es el modelo a alcanzar. Con estos tests unitarios realizados sobre el módulo CBR del dominio es posible garantizar que si la implementación supera todos ellos es una versión válida para funcionar como componente de la infraestructura definitiva.

8.2 El CBR de argumentación

La estructura base de ambos módulos CBR es muy similar, por lo tanto es posible reutilizar la base de los test del anterior módulo para las pruebas del módulo CBR de la argumentación. Estas estructuras deberán recibir ciertos cambios diferenciadores para ser aplicables al módulo tratado en esta sección. Estas modificaciones son:

- Precisión de la recuperación: Un caso de argumentación es más complejo que un caso de dominio. Los casos de argumentación están conformados, entre otras cosas, por un caso de dominio. Esto permite dar un enfoque en perspectiva sobre la diferencia de complejidad entre ambos. El método de recuperación del módulo CBR de la argumentación, aparte de la similitud, comprueba que la relación de dependencia entre los casos de argumentación recuperados y el que es pasado como parámetro se correspondan. Gracias a la forma en la que están implementadas las funciones dentro de la clase del módulo, se garantiza la unicidad de los identificadores de los casos de argumentación similares retomados. Por este motivo y pese a las dificultades que aparentaba ofrecer, es suficiente con realizar una simple comparación de identificadores de los casos similares, en lugar de comparar todos los atributos del caso como en el módulo anterior.

- Consistencia de la recuperación: En este caso no es necesario realizar cambios estructurales, simplemente sustituir las funciones que hacen uso de los casos de dominio en el módulo CBR del dominio por aquellas que hacen uso de los casos de argumentación en el módulo CBR de la argumentación.

- Casos duplicados: Nuevamente, los cambios requeridos no son estructurales, únicamente son sobre los tipos y funciones aplicadas.

- Operatividad: Este test queda suprimido del módulo CBR de la argumentación.

Los tests correspondientes a este entregable son más sencillos y en menor cantidad que en el anterior entregable, pese a ser más complejos. El motivo detrás de esta aparente incongruencia viene dado por la necesidad de invertir el tiempo más eficientemente, se efectuó de igual manera en la versión original del código Java. No quiere decir, ni mucho menos, que las cualidades específicas del módulo CBR de argumentación no vayan a ser probadas. Estas pruebas se realizaran en el test del agente argumentativo, ya que este hace uso de todas estas funciones. La ventaja de realizar individualmente las pruebas para cada entregable ya se ha argumentado anteriormente, permite tratar los posibles errores de forma aislada, lo que facilita enormemente su corrección. Las consecuencias más directas de esta acción serán que, si se da el caso de fallar el agente argumentativo, los errores futuros estarán más distribuidos y llevarán un mayor coste de tiempo corregirlos. Puede verse como una pequeña apuesta contra el tiempo forzada por las circunstancias.

8.3 Planeadas

En esta sección de corta extensión se mencionan las pruebas que iban a ser realizadas a los distintos entregables una vez estos se considerasen aptos para las mismas. Las razones de por qué no se han llegado a aplicar han sido explicadas ya a lo largo de los dos anteriores capítulos, principalmente en la sección 6.1.

8.3.1. El agente argumentativo

El agente argumentativo es la pieza más compleja de la plataforma y como tal, requiere de los test unitarios más rigurosos. Además, cabe recordar que ciertas funciones del módulo CBR de la argumentación no fueron probadas en su momento porque se reservaban para este entregable. Existen test para la versión original del código en Java, pero son más bien ejemplos. Estos tests, consistían en crear escenarios artificiales en los que un programa principal configuraba el CS y una serie de agentes argumentativos que luego lanzaba a ejecución. Es una prueba dentro de un entorno controlado en la que se testan los distintos componentes. Lo normal en este tipo de pruebas es contar con la respuesta correcta para confirmar que los agentes dan en el clavo. Estos ejemplos podrían ser imitados, pero no traducidos, dado que al tener una fuerte implicación del sistema de comunicaciones las traducciones son todavía más incompatibles, pues dicho sistema ha sido repensado para la nueva versión.

8.3.2. La interfaz y plataforma

La interfaz es la capa de acceso que se sitúa sobre la capa de servicios. Para tener algo a lo que acceder desde la interfaz es necesario que exista un mecanismo por debajo que regule las interacciones. Es por este motivo que la interfaz es lo último que se monta y por ende lo último que se testea. Una vez dados todos los componentes y probado el agente argumentativo se desarrolla la interfaz gráfica de usuario y una vez esta se concluye es el momento de probarla.

Las pruebas sobre interfaces no pueden ser evaluadas de forma tan concisa y determinista como los otros módulos. Es necesario probar que todas las interacciones planeadas funcionan, que todos los avisos se muestran y que todo está donde tiene que estar, pero esto es solo para determinar que la interfaz es funcional. Si queremos que la interfaz sea accesible es necesario determinar su usabilidad. El problema subyace en la subjetividad de esta medida. Se ha visto que la usabilidad se mide en factores como el tiempo de retención o el esfuerzo que cuesta invertir en aprender el funcionamiento de la interfaz, estos son aspectos que claramente varían de persona a persona. Para realizar unas pruebas serias sobre este componente es necesario reunir a un grupo de gente, los más diverso posible, al cual entrevistar para poder determinar todos estos aspectos. Puede considerarse que es una actividad que requiere más esfuerzo incluso que los test unitarios del agente argumentativo.

En resumen, las pruebas que se han realizado se han llevado a cabo en los momentos correctos del desarrollo y han servido para corregir los problemas que presentaban las implementaciones de los componentes, mejorando así el resultado final. Desgraciadamente, no se han podido probar todos los componentes por lo que no se puede asegurar con certeza que módulos como el agente argumentativo ofrezcan la respuesta que deberían. Queda un largo camino, pero las pruebas restantes están bien planteadas y es solo cuestión de tiempo.

CAPÍTULO 9
Conclusiones

Al comienzo del documento se exponen los que eran los objetivos a alcanzar con el proyecto. Se pueden destacar tres principales, pero todos ellos se pueden englobar bajo la meta de conseguir mejorar la infraestructura existente en términos de manejo, accesibilidad y productividad. Con la entrega del trabajo de fin de grado y en vista de lo expuesto a lo largo de esta, su memoria, es objetivo confirmar que el desarrollo ha servido para dar pasos en la buena dirección, pero que no se han alcanzado todas las metas planteadas.

Comenzando por los objetivos que sí han sido cumplidos, tenemos:

- Reimplementación de las comunicaciones: Supuso un conflicto inicial el determinar la forma en la que las nuevas comunicaciones funcionarían, pero una vez resuelto fue de los módulos más rápidos de implementar.
- Generalización del agente argumentativo: Si bien no se ha llevado a cabo completamente, se puede considerar exitosa. Aunque la versión implementada liberada no posea estas cualidades, la parte más complicada y costosa ya ha sido realizada, que es determinar qué partes se pueden generalizar, cuáles no y cómo se llevará a cabo.
- Desarrollo de una interfaz de usuario: Es la propuesta inicial que más lejos a quedado de completarse, pero aún así, ha sido bien planteada y trabajada. Esto es así hasta el punto de tener las apreciaciones de diseño y los bocetos ya desarrollados, quedando como pendiente escoger una herramienta definitiva, ya que solo se han planteado marcos generales, para poder comenzar con su construcción e implementación.

Las razones que han llevado a la no compleción de algunas de las metas se explican por los problemas encontrados y los errores cometidos. El principal problema encontrado fue la necesidad de cambiar el lenguaje de programación en el que la infraestructura estaba implementada, se considera como problema y no como error porque fue a raíz del análisis del problema que se concluyo que la migración era necesaria. Por otro lado, el error más destacable fue sin duda el de planificación. Aún pudiendo achacar los fallos en la planificación al problema de la migración de un lenguaje a otro, la realidad es que el cálculo del tiempo que iba a requerir fue muy erróneo, provocando que las otras funcionalidades a implementar dispusieran de menos tiempo del que se suponía iban a necesitar.

No todo es malo y de los errores se aprende, el trabajo ha supuesto y requerido aprender muchas lecciones. Nuevas tecnologías no contempladas en la carrera, como SPADE, autodoc, Magentix2 y Pycharm han sido de gran utilidad para el desarrollo del proyecto y han requerido de un aprendizaje casi de cero. Este proceso ha permitido alcanzar

conocimiento suficiente para desenvolverse a un nivel básico, e incluso realizar acciones avanzadas como en el caso del protocolo argumentativo con SPADE. Además de las nuevas tecnologías, también se ha sacado mucho partido a los conocimientos adquiridos a lo largo de la carrera, regresar sobre los apuntes de años anteriores ofrece una visión distinta de la que se tuvo en su momento y se llega a apreciar que herramientas que se podía pensar no ofrecían más, ocultan un potencial oculto que, para este caso, ha resultado crucial.

9.1 Relación del trabajo desarrollado con los estudios cursados

Tal y como indica la oración que precede a esta sección y que, a su vez, como concepto ha estado presente a lo largo de casi todos los capítulos, el conocimiento adquirido a lo largo del transcurso de la carrera cursada ha sido esencial para el desarrollo del trabajo. Haciendo un repaso a todas las veces que se ha mencionado alguno de estos conocimientos rescatados se establece la siguiente lista:

- AIN: La asignatura más directamente en relación con el tema del trabajo. La IA, los agentes inteligentes, los MAS, todos son conceptos alrededor de los cuales se han desarrollado las ideas principales de los primeros capítulos. En esta materia se introduce también el estándar FIPA y las plataformas como Magentix2.

- CPA: Los temas de esta asignatura han servido para reforzar la argumentación sobre las ventajas de emplear un sistema de comunicaciones a nivel global. La materia no se centra plenamente en estos aspectos, pero sus temas son extrapolables al uso general, lo cual deja en muy buen lugar los conocimientos allí aprendidos.

- CSD: De forma similar a la anterior, esta asignatura ha servido para argumentar en relación al otro tipo de comunicaciones presentado, la de los sistemas distribuidos. Los conceptos adquiridos en esta asignatura son a un nivel introductorio, pero son una muy buena base para comenzar y desarrollar.

- GPR: Materia crucial en el desarrollo y más aún una vez vistas las consecuencias de la planificación. En esta asignatura se adquieren y revisan conocimientos sobre planificación, gestión e incluso contabilidad de los recursos. La EDT ha sido definida partiendo de esta asignatura como base.

- IPC: De cara al diseño de la interfaz y sus pruebas, ha sido la fuente más relevante. Desde la definición de la usabilidad, pasando por el diseño de bosquejos y culminando con las formas de testar la calidad de las interfaces, a grandes rasgos, casi todo el contenido de la asignatura se ha planteado útil en este desarrollo. Desafortunadamente, no todos estos conceptos han podido ser aprovechados a causa del tiempo.

- IPV: En contra de lo que pudiera parecer, el tema del trabajo tiene mucho interés desde el punto de vista de los videojuegos. Tal es así que se pueden trazar claras semejanzas entre el concepto de PNJ y el de agente inteligente. Otro aspecto de la asignatura al que se ha sacado partido es el hecho de que profundiza en el uso práctico de las FSM.

- ISW: Dado que la estandarización es una propiedad a la que se le presta mucha atención a lo largo del proyecto, era necesario remontarse a ISW. Además, en esta materia se adquieren conocimientos sobre distintas metodologías de trabajo y se presentan diversas herramientas que agilizan el desarrollo.

- LPPL: Los mismos redactores de la presentación de esta asignatura saben perfectamente que, pese a parecer un tema muy específico, en una rama específica de la carrera, LPPL trata temas que todo desarrollador debería conocer. Estos temas, por supuesto, han sido tenidos en cuenta durante el desarrollo. Han servido como baremo para medir las cualidades de los distintos lenguajes de programación que se planteaba usar y también para comprender el funcionamiento de algunas herramientas, como los analizador sintácticos o las herramientas de autocompleción.

- LTP: El enfoque de esta asignatura es ampliar el horizonte de los alumnos en lo que a los lenguajes de programación respecta. Se introducen los paradigmas y se presentan diversos lenguajes de cada uno, así como herramientas y tecnologías para desenvolverse mejor con ellos. Aparte del nuevo enfoque, LTP a influido mucho a la hora de analizar el problema y sobretodo, de proponer una solución.

- MAD: Es una de las asignaturas más tempranas dentro de la carrera y la razón de ello es que sedimenta las bases sobre las que asignaturas posteriores se apoyan. Entre esas bases se destacan en este trabajo los grafos, vitales a la hora de definir las interfaces y las relaciones de dependencia entre tareas.

- RED: Esta materia es la entrada a las redes de computación. En esta se introduce por primera vez en la carrera la idea de «socket», que ha sido un tema recurrente a la hora de decantarse por un nuevo sistema de comunicaciones.

- SIN: Es en esta materia que se empiezan a introducir temas más centrados alrededor de la IA. Muchas deducciones hechas a lo largo de esta memoria se basan en conceptos de la asignatura. Sin ir más lejos, el razonamiento es directamente la estrella de este trabajo. Aunque no sea el mismo razonamiento que el visto en la asignatura, conceptos como premisas, reglas e inferencia han sido determinantes en la conceptualización y desarrollo del proyecto.

- TSR: Los contenidos de esta asignatura han sido utilizados al principio para, junto a CSD, reflexionar entorno a las comunicaciones que emplearía el sistema, pero no ha sido su única aportación. Los «middleware» de mensajería han actuado como referente para contrastar las virtudes y defectos de las plataformas que han protagonizado las comunicaciones de las distintas versiones e implementaciones de las infraestructuras presentadas.

Estas son las asignaturas en las que es más fácil dilucidar sus aportaciones, pero, objetivamente, no son las únicas que han aportado contenido. Muchas de estas materias se sostienen sobre las bases de algunas de las primeras en ser introducidas. Las asignaturas de «Fundamentos de Computadores» y «Fundamentos de los Sistemas Operativos» aportan ideas sin las cuales habría sido muy difícil entender los conceptos que posteriormente se presentan en asignaturas como CSD, RED o ISW. Ahora bien, tras realizar una reflexión, se puede entender que el uso de conocimientos pasados ha sido un alivio sobre la carga de emplear tantas nuevas tecnologías sobre el proyecto, pero también que no ha sido suficiente para evitar que haya resultado siendo agotador y notablemente extenso.

Adicionalmente, las competencias transversales son siempre un tema de interés, ya que sirven para calificar cualidades de las personas que un simple expediente no podría reflejar. Comenzando por lo negativo, la planificación y gestión del tiempo, desgraciadamente, ha destacado por su ausencia y el pensamiento crítico no siempre ha sido bien empleado. Por otro lado, cabe destacar que el hecho de partir de un sistema complejo ya establecido demuestra una gran capacidad de comprensión e integración, además, la tendencia a decantarse por el uso de nuevas herramientas a las que se les pueda sacar alguna utilidad implica una gran capacidad de innovación, creatividad y emparedamiento.

Otras de las competencias no se pueden evaluar dentro del marco del trabajo porque no se ha podido demostrar su potencial. Algunos ejemplos son el trabajo en equipo y liderazgo y la responsabilidad ética y medioambiental. Aunque, pese a que no exista el trabajo en equipo, si que ha sido necesaria la comunicación entre alumno y tutores, la cual, al respecto del alumno, ha tratado de ser eficaz, pero ha resultado ser mejorable en diversos aspectos.

En conclusión, el proyecto se ha planteado bajo unos buenos puntos y con unos claros objetivos que se han visto interferidos por problemas externos e internos y por fallos en aspectos como la planificación, pese a todo, el balance es mucho más positivo que negativo, ya que se han puesto en uso conocimientos adquiridos a lo largo de los ocho semestres de la carrera, recordando viejas ideas y añadiendo conceptos nuevos. Además, la plataforma es potencialmente más útil, pues con la documentación creada y la liberación del código realizada se hace sencillo iniciar un nuevo proyecto que culmine los objetivos que este no pudo y que a su vez proponga nuevas ideas.

Bibliografía

[1] Jaume Jordán, Stella Heras, Soledad Valero y Vicente Julián. An infrastructure for argumentative agents. *Computational Intelligence*, 31:3:418–441, Noviembre, 2015.

[2] Gerhard Lakemeyer y Bernhard Nebel. Foundations of knowledge representation and reasoning. *Foundations of knowledge representation and reasoning*, 810:1:1–12, Enero, 1992.

[3] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi y Werner Nutt. The complexity of concept languages. *Information and computation*, 134:1–58, Febrero, 1997.

[4] Stella Heras, Vicente Botti, Jaume Jordán y Vicente Julián. Argue to agree: a case-based argumentation approach. *International Journal of Approximate Reasoning*, 82–108, 2013.

[5] Vicente Julián, Carlos Carrascosa, Juan Manuel Corchado, Javier Bajo, Esteve Acebo, Bianca Innocenti y Vicente Botti. Physical Agents. *Whitestein Series in Software Agent Technologies and Autonomic Computing*, 54:1:117–143, 2008.

[6] Michael Wooldrige. *An introduction to multiagent systems*. John Wiley & Sons, LTD., Chichester, primera edición, 2002.

[7] Stuart Russell y Peter Norvig. *Artificial intelligence: a modern aproach*. Pearson Education, LTD., Harlow, tercera edición, 2002.

[8] Anthony Weston. *Las claves de la argumentación*. Ariel, S.A., Barcelona, tercera edición (traducción oficial), 2006.

[9] Luis Enrique Colmenares Guillén. *Semántica operacional para lógicas no-monótonas*. Colmenares Guillén, L. E., Universidad de las Américas Puebla, 2000.

[10] María Alpuente Frosnedo y Pascual Julián Iranzo. *Programación lógica: teoría y práctica.*. Person Education, S.A., Madrid, 2007.

[11] Steven Jacobs, Maxine Cohen, Catherine Plaisant. *Designing the User Interface: Pearson New International Edition: Strategies for Effective Human-Computer Interaction.*. Pearson Education, LTD, Harlow, 2013.

[12] FIPA specifications grouped by category. *Consultado en* http://www.fipa.org/repository/bysubject.html

[13] index - TIOBE the sowtware quality company. *Consultado en* https://www.tiobe.com/tiobe-index/

[14] The state of the octoverse - Top languages. *Consultado en* https://octoverse.github.com/

[15] Programming: Pick up Python. *Consultado en* https://www.nature.com/news/programming-pick-up-python-1.16833

APÉNDICE A
Glosario de términos

Este apéndice esta orientado a definir conceptos empleados en la reducción del documento que puedan resultar ajenos a los lectores menos especializados en la materia. Ya sea el caso, o no, se recomienda la lectura, aunque sea de forma superficial, de este apéndice antes de entrar de lleno con el contenido de la memoria. El glosario va a estar dividido en dos secciones, una dedicada a los acrónimos y abreviaturas y otra dedicada a los términos más técnicos o nombres de herramientas y programas. En la primera sección se dará también una pequeña definición para acompañar al significado de las siglas. Se recomienda tener el apéndice localizado para buscar definiciones de términos que puedan aparecer durante la lectura completa. En favor de potenciar esta utilidad algunos términos que se definen en sus respectivas secciones aparecen también aquí y se respeta el orden alfabético para hacerlos más sencillos de localizar.

A.1 Acrónimos, abreviaturas y siglas

- AC («Argumentation Case»): Un caso de argumentación es una estructura creada a partir de KR que sirve para representar argumentaciones ya concluidas y su correspondiente resultado (sección 2.2.1). También es considerado parte de los KR.
- ACL («Agent Communication Language»): Lenguaje de comunicación común a los agentes de la organización (sección 2.1.1).
- AD («Attack Degree»): El grado de ataque es un valor que estima de forma aproximada el riesgo a ser atacado al que se expone un argumento de ser propuesto, basado en la cantidad de argumentos similares que fueron atacados en el pasado, teniendo en cuenta también la cantidad de ataques (sección 3.2.2).
- AIN (Agentes inteligentes): Asignatura de la carrera centrada en introducir el concepto de agentes inteligentes, explicar sus características y desarrollar sobre las tecnologías generadas con o sobre los mismos.
- AMS («Agent Management System»): Para el estándar FIPA es el sistema de control de agentes. Es un componente obligatorio y está a cargo de la gestión principal de una AP, desde el se controla los estados de la plataforma y los propios agentes (sección 2.1.1).
- AP («Agent platform»): Infraestructura software, hardware o mixta que da soporte a la ejecución de agentes inteligentes (sección 2.1.1).
- API («(Application Programming Interface)»): Véase ligadura.

- CBR («Case Base Reasoning»): El razonamiento basado en casos consiste en encontrar soluciones a los problemas buscando casos almacenados, para los que ya se posee una solución, que sean similares al del problema a resolver (sección 2.1.2).
- CPA (Computación Paralela): Asignatura de la rama de «Computación» centrada en expandir el conocimiento sobre las maneras de distribuir la carga computacional. Tiene un gran peso práctico.
- CS («Commitment Store»): Es el nombre del módulo de la infraestructura tomada como base que actúa como DF y que implementa funcionalidades del AMS, de acuerdo al estándar FIPA.
- CSD (Concurrencia y sistemas distribuidos): Asignatura de la carrera centrada en presentar distintas formas de distribuir la carga computacional. A parte de eso también se introducen los sistemas en tiempo real.
- DC («Domain Case»): Un caso de dominio es una estructura creada a partir de KR que sirve para representar problemas ya resueltos previamente y sus respectivas soluciones (sección 2.2.1). También es considerado parte de los KR.
- DF («Directory Facilitator»): Para el estándar FIPA es el directorio moderador. Es un componente opcional pero, de ser implementado, debe cumplir con una serie de condiciones. Entre estas condiciones está ofrecer un servicio de páginas amarillas y ser altamente confiable, es decir, ofrecer información actualizada (sección 2.1.1).
- ED («Efficiency Degree»): El grado de eficiencia es un valor que ofrece una estimación sobre el número de pasos que puede tardarse en alcanzar un acuerdo, basada en experiencias argumentativas pasadas (sección 3.2.2).
- EDT (Estructura de Desglose del Trabajo): Es la estructura resultante de dividir todo el trabajo de un proyecto en tareas mínimas.
- EP («Explaantory Power»): El poder explicativo es un valor representa la cantidad de piezas de información que un argumento contiene (sección 3.2.2).
- ETSInf (Escuela Técnica Superior de Ingeniería Informática): Centro pionero en la enseñanza de informática en la Comunidad Valenciana. Es parte de la UPV y esta es su página en la red: https://www.inf.upv.es/www/etsinf/es/.
- FIPA («Foundation for Intelligent Physical Agents»): Organización de estándares para agentes y sistemas multiagente oficialmente aceptada por la IEEE el ocho de junio del año 2015. El objetivo de esta organización es dedicarse a promover la industria de los agentes inteligentes a base de desarrollar especificaciones que apoyen la interoperabilidad entre agentes y aplicaciones basadas en agentes (sección 2.1.1). Su página en la red es: http://www.fipa.org/.
- FSM («Finite States Machine»): Las máquinas de estados finitos son estructuras compuestas por una serie finita de estados y las transiciones entre los mismos. Tienen un estado inicial y uno o varios estados finales (sección 2.2.2).
- GPR (Gestión de Proyectos): Asignatura de la carrera enfocada a enseñar la teoría de la planificación, estructuración y contabilidad aplicada a proyectos informáticos.
- GTI-IA (Grupo de Tecnología Informática - Inteligencia Artificial): Grupo de investigación que tiene como principal objetivo generar nuevo conocimiento en el área de los sistemas inteligentes, cooperación en el análisis de problemas socioeconómicos y entrenamiento de jóvenes investigadores. Página del grupo: http://www.gti-ia.upv.es/.

- IA (Inteligencia Artificial): Rama de la informática que investiga formas de dotar a los sistemas computacionales de inteligencia. Es una rama que bebe de múltiples y diversas disciplinas que también desarrolla sobre la ambigüedad del término «inteligencia» en busca de determinar que sistemas pueden o no considerarse inteligentes.
- IDE («Integrated Developement Enviroment»): Los entornos de desarrollo integrados son programas que proporcionan servicios al usuario para facilitarle la labor de desarrollar programas, bibliotecas u otro tipo de aplicaciones de carácter informático.
- IPC (Interfaces Persona Computador): Asignatura de la carrera dedicada a la explicación del vínculo entre máquina y usuario y al diseño de interfaces (sección 3.2.2).
- IPV (Introucción a la Programación de Videojuegos):Asignatura optativa de la carrera en la que se enseñan las nociones básicas de la programación de videojuegos mediante herramientas como Unity. También se ven conceptos más avanzados como las FSM.
- ISW (Ingeniería del SoftWare): Asignatura de la carrera centrada en introducir metodologías de trabajo y conceptos de programación centrados en al formalización y estandarizado del desarrollo software.
- KR («Knowledge Resources»): Los recursos del conocimiento son el conjunto de herramientas y datos que se emplean para representar el conocimiento poseído en términos computacionales (sección 2.2.1).
- LPPL (Lenguajes de programación y procesadores de lenguajes): Asignatura de la rama de «Computación» que se centra en explicar conceptos avanzados de los lenguajes de programación a nivel de diseño de los mismos, esto incluye su estructura y como generar compiladores para ellos.
- LTP (Lenguajes, tecnologías y paradigmas de la programación): Asignatura de la carrera que se imparte con la intención de ampliar la visión del alumno sobre los lenguajes de programación. Se presentan lenguajes de distintos paradigmas, explicando sus cualidades y mostrando herramientas para usarlos y comprenderlos.
- MAD (Matemáticas Discretas): Asignatura de la carrera que enseña los conceptos referentes a las matemáticas discretas, desde la lógica hasta los grafos.
- MAS («Multi-Agent System»): Conjunto de varios agentes que, por medio de la comunicación y la colaboración, tratan de resolver un problema común (sección 2.1.1).
- MTS («Message Transport Service»): Para el estándar FIPA es el sistema de paso de mensajes. Es un componente obligatorio y está a cargo de garantizar que los agentes pueden comunicarse entre sí vía paso de mensajes (sección 2.1.1).
- NPC («Non-Player Character»): NPC es la forma de decir PNJ en inglés.
- P2P («Peer to peer»): Tipo de comunicación en la cual los participantes son pares. Ser pares implica que están al mismo nivel, es decir, no hay comportamiento diferente entre ellos ni distinciones entre clientes y servidores. Un ejemplo de estas comunicaciones está en el sistema de videollamadas de Skype.
- PD («Persusiveness Degree»): El grado de persuasividad es un valor que representa el poder persuasivo estimado de un argumento (sección 3.2.2).

- PL: Paradigma de programación de tipo declarativo. En este paradigma la lógica se usa como un lenguaje de programación (sección 2.1.3.
- PNJ (Personaje No Jugable): Personaje de un videojuego que no puede ser controlado por el jugador.
- POO (Programación Orientada a Objetos): Paradigma de programación que se caracteriza por permitir la definición de clases de objetos que comparten cualidades similares. Algunas de sus cualidades distintivas son la reutilización y la expansibilidad del código.
- Pypi («Python Package Index»): Es el índice de paquetes de Python. Permite la fácil y rápida distribución e instalación de los módulos software registrados en su índice.
- RC (Representación del Conocimiento): Es una rama de la IA que estudia la forma de representar y almacenar conocimiento en el ámbito computacional.
- RD («Risk Degree»): El grado de riesgo es un valor que estima de forma aproximada el riesgo a ser atacado al que se expone un argumento de ser propuesto, basado en la cantidad de argumentos similares que fueron atacados en el pasado (sección (sección 3.2.2).
- RED (Redes de Computación): Asignatura de la carrera centrada en introducir las redes de comunicación entre sistemas computacionales.
- SD («Support Degree»): El grado de soporte es un valor que ofrece una estimación aproximada de la probabilidad de que la conclusión del argumento actual vaya a ser aceptada al final del diálogo (sección 3.2.2).
- SF («Support Factor»): Un factor de soporte es un conjunto de seis valores normalizados que representan pesos configurables. Estos pesos sirven para balancear la táctica argumentativa que usarán los agentes en la implementación actual del mismo (sección 3.2.2).
- SIN (Sistemas Inteligentes): Asignatura de la carrera centrada en introducir los primeros conceptos sobre inteligencia artificial. Entre otros temas, es estudian los sistemas basados en reglas, la inferencia y el razonamiento probabilístico.
- SPADE («Smart Python Agent Development Environment»): Plataforma con soporte para sistemas multiagente escrita en y para Python (sección 4.1.1).
- SS («Support Set»): Un conjunto de apoyo es una estructura creada a partir de KR que sirve para representar el conjunto de elementos que acompañan a un argumento de ataque (sección 2.2.1). También es considerado parte de los KR.
- TCK («Technology Compatible Kit»): Suite de compatibilidad tecnológica en español. El TCK de Oracle para Java es JCK («Java Compatibility Kit») y es utilizado para garantizar implementaciones compatibles de la plataforma.
- TSR (Tecnologías de los sistemas de información en la red): Asignatura de la carrera que enseña distintas herramientas de uso común en la red y la estructura de ciertos sistemas que la componen.
- UML: («Unified Modeling Language»): El lenguaje unificado de modelado es un lenguaje enfocado al modelado de sistemas software.
- UPV (Universidad Politécnica de Valencia): Institución pública valenciana dedicada a la investigación y a la docencia. Página en la red: https://www.upv.es/.

- VRAIN (Valencian Research Institute for Artificial Intelligence): Es un instituto Valenciano a cargo de investigaciones en el ámbito de la inteligencia artificial. Página del instituto: http://vrain.upv.es/.
- XMPP («Extensible Messaging and Presence Protocol»): Protocolo de mensajería que destaca por ser diverso, extensible, flexible, seguro, descentralizado, probado, estandarizado y abierto. Página en línea: https://xmpp.org/.
- wAD («Weigth of AD»): Valor ponderado, parte del SF, que representa el peso que se le da al AD de un argumento (sección 3.2.2).
- wED («Weigth of ED»): Valor ponderado, parte del SF, que representa el peso que se le da al ED de un argumento (sección 3.2.2).
- wEP («Weigth of EP»): Valor ponderado, parte del SF, que representa el peso que se le da al EP de un argumento (sección 3.2.2).
- wPD («Weigth of PD»): Valor ponderado, parte del SF, que representa el peso que se le da al PD de un argumento(sección 3.2.2).
- wRD («Weigth of RD»): Valor ponderado, parte del SF, que representa el peso que se le da al RD de un argumento (sección 3.2.2).
- wSD («Weigth of SD»): Valor ponderado, parte del SF, que representa el peso que se le da al SD de un argumento (sección 3.2.2).

A.2 Términos

- Acto performativo: «Performative», en inglés. Es aquel acto cuya sola enunciación realiza la acción que significa. Un acto performativo es, por ejemplo, un saludo.
- Agente FIPA: Agente inteligente que cumple con el estándar FIPA. Son autónomos, comunicativos y tienen un identificador único. Se comunican en un ACL mediante el MTS (sección 2.1.1.
- Agente inteligente: Sistema computacional que cuenta con la capacidad de realizar acciones de parte de su usuario o propietario y de forma independiente (sección 2.1.1).
- AgileJ: AgileJ StructureViews es una herramienta para Eclipse que genera diagramas de clases aplicando ingeniería inversa sobre código Java.). Página en línea: (http://www.agilej.com/.
- Anaconda: Plataforma abierta y de libre distribución muy extendida en su uso en ciencia de datos y aprendizaje automático. Funciona con Python y R. Página en la red: https://www.anaconda.com/.
- Assert: Nombre que reciben los comandos empleados para verificar condiciones en los tests unitarios del proyecto. Es una palabra reservada en herramientas como JUnit y Pytest.
- Autómata finito: Formalmente hablando, son estructuras consistentes en un conjunto de estados, un conjunto de símbolos, una función de transición para marcar el paso de un estado a otro mediante un símbolo (o conjunto de símbolos), un estado inicial y un conjunto de estados finales. Estas estructuras son también conocidas como FSM y estudiadas en profundidad dentro de la carrera en la asignatura «Teorías de autómatas y lenguajes formales».

- Cadena de documentación: Usualmente llamadas «docstrings», son los comentarios descriptivos de las clases y métodos a los que acompañan. Suelen estar precedidos de caracteres reservados.
- Código objeto: Código que se genera resultado de la compilación del código fuente.
- Conda: Véase Anaconda
- Condición de carrera: Evento no deseado que se produce cuando dos o más hilos de ejecución tratan de acceder al mismo tiempo a una dirección de memoria para modificarla, el resultado será que la memoria contendrá el valor aportado por el último modificador, sin tener en cuenta los cambios reportados por el resto.
- Diálogo: Tipo de ventanas que, por su baja complejidad, permiten una comunicación simple y rápida con el usuario.
- Duck typing: Estrategia de inducción del tipo basada en la prueba del pato. Esta prueba se define por la oración: «Si camina como un pato y grazna como un pato, entonces debe de ser un pato». Es un método de tipificación dinámica propio de los lenguajes de POO que se basa en aplicar razonamiento inductivo sobre métodos y propiedades en las que se haya usado previamente una variable para determinar su validez semántica.
- Eclipse: El IDE para desarrollo en Java más extendido. Página en la red: https://www.eclipse.org/downloads/.
- Figma: Software orientado al diseño de esquematizaciones gráficas, como grafos, diagramas o bosquejos de interfaces. Página en la red: https://www.figma.com/.
- GitHub: GitHub Servicio de alojamiento de repositorios GIT muy popular. GIT es el sistema de control de versiones más extendido; estos sistemas se emplean para facilitar la eficiencia y aumentar la fiabilidad del mantenimiento del código fuente.
- Instantánea: Representación del estado del sistema en un momento determinado.
- JUnit: Gupo de bibliotecas destinadas al diseño de pruebas unitarias en Java. Página en línea: https://junit.org/junit5/.
- Kotlin: Lenguaje de programación que puede ejecutarse sobre la máquina virtual de Java. Es desarrollado por JetBrains https://www.jetbrains.com/ y su objetivo es tener las características del lenguaje Scala (https://www.scala-lang.org/), que ofrece un híbrido entre POO y programación funcional a un alto nivel de abstracción, pero con un mejor tiempo de compilación.
- Ligadura: Del inglés «binding», son más comúnmente conocidos como APIs (Application Programming Interface) o interfaces de programación de aplicaciones. Las ligaduras consisten en un código, llamado código pegamento («glue code»), que sirve exclusivamente para adaptar ciertas partes del código fuente que harían incompatible la biblioteca con el lenguaje de uso.
- Lista plana: «Flat list», en inglés, es una lista que previamente contenía otras listas y que tras aplanarla pasa a estar formada directamente por los elementos que contenían estas listas internas.
- Lógica de primer orden: También llamada lógica de predicados, es un sistema formal diseñado para estudiar la inferencia en los lenguajes de primer orden

- Mensaje: En el contexto de la comunicación software por mensajería, los mensajes son unidades de información a transmitir y tienen, como mínimo: un receptor, un remitente y un contenido o cuerpo.
- Middleware: Software que asiste a un sistema para interactuar o comunicarse con otros sistemas, añadiendo distintas capas de transparencia..
- Mockup: Término referido en el ámbito del diseño software para referirse a los bosquejos
- Montículo: También conocido como «heap». Es una abstracción empleada para representar y manejar cómo accede y modifica la memoria un programa. La memoria a la que puede acceder el montículo no está limitada a un contexto y es globalmente accesible. Los accesos al montículo son más lentos que los accesos a la pila.
- Ontología: Red o sistema de datos que define las relaciones existentes entre los conceptos de un dominio o área del conocimiento.
- Operatividad: Capacidad para realizar una función
- Pila: También conocida como «stack». Es una abstracción empleada para representar y manejar cómo accede y modifica la memoria un programa. La memoria a la que puede acceder la pila está limitada al contexto de la ejecución. Lo accesos a la pila son muy rápidos.
- Plataforma FIPA: AP formada por la máquina o máquinas, el sistema operativo, el hardware que da soporte a los agentes, los componentes de manejo de los agentes FIPA (DF, AMS y MTS) y los agentes.
- Polimorfismo: Característica de los lenguajes que permite manejar valores de distintos tipos usando una interfaz uniforme. Puede ser *ad-hoc*, también conocido como aparente, si trabaja sobre un número finito de tipos relacionados o universal, también conocido como verdadero, sí trabaja sobre un número potencialmente infinito de tipos (con cierta estructura común).
- Pytest: Sistema empleado para la realización de tests unitarios en Python (sección 6.2).
- Razonamiento abductivo: Razonamiento que, a partir de la descripción de un hecho o fenómeno, ofrece o llega a una hipótesis, la cual explica las posibles razones o motivos del hecho mediante las premisas obtenidas.
- Razonamiento monótono: El razonamiento deductivo estándar es monótono. En este, dada la regla: «A implica b», si una premisa indica que A, entonces seguro que b. En cambio, en los razonamientos no monótonos esta propiedad no está garantizada.
- Realidad mixta: Mezcla de la realidad virtual con la realidad aumentada. Los Hololens de Microsoft (`https://www.microsoft.com/es-es/hololens`) son unos visores de realidad mixta.
- Socket: En Java, unidad mínima de comunicación.
- Superclase: Clase que está por encima de otras clases en la jerarquía. De una superclase pueden extenderse varias clases.
- Test unitario: Prueba que se realiza sobre un módulo de un proyecto para evaluar su correcto funcionamiento.

- Token: Es un elemento atómico del lenguaje. Normalmente se diferencian cinco tipos: constantes, identificadores, operadores, separadores y palabras reservadas.
- Transparencia: En informática, la cualidad de un sistema o componente de un sistema para desempeñar una función sin que se requiera conocer como la realiza.
- Usabilidad: Propiedad de las interfaces persona-computadora empleada para medir la calidad de las mismas (sección 3.2.3).