Modelo de agente interfaz basado en lógica y especificado como componente de software reutilizable para computación científica

Ing. Diego Mosquera Uzcátegui

Tutor: PhD. Jacinto Dávila

Trabajo presentado como requisito parcial para la obtención del grado de
MAGISTER SCIENTIAE EN MODELADO Y SIMULACION DE SISTEMAS

UNIVERSIDAD DE LOS ANDES
MERIDA, VENEZUELA
Mérida, 30 de Septiembre de 2004.

Índice general

Índice de figuras

Índice de tablas

Resumen

Este documento presenta una implementación de un agente inteligente basado en lógica y especificado como componente de software reutilizable: SIMULANTE, programado para ayudar a usuarios en el uso de herramientas proporcionadas por una aplicación Web distribuida y cooperativa.

SIMULANTE es una propuesta para integrar, en una misma tecnología, la especificación lógica del agente GLORIA [5], la implementación de ese agente escrita en Java [4] y el despliegue de esa integración como un componente reutilizable Enterprise Java Bean.

SIMULANTE es caracterizado como un agente interfaz basado en el conocimiento [17] cuyo dominio de aplicación está dirigido a la simulación de modelos escritos en GALATEA [2]. El agente cumple dos roles primarios dentro de ese dominio: el rol de asistente de la aplicación y el rol de consultor.

En el rol de asistente, SIMULANTE está programado para realizar tareas de compilación y ejecución de los modelos GALATEA, al mismo tiempo que es capaz de presentar los resultados de una simulación.

En el rol de consultor, SIMULANTE está programado para prestar servicios basados en conocimiento y proponer material didáctico e informativo sobre el uso de GALATEA como lenguaje de simulación.

SIMULANTE se propone como una estructura de componente reutilizable que puede ser integrado a cualquier plataforma Web distribuida y cooperativa que siga las especificaciones de diseño J2EE [19].

En este trabajo se describe la especificación de SIMULANTE, la implementación y el despliegue como componente reutilizable en una aplicación Web demostrativa, y los aspectos de integración a una plataforma ya construida llamada BIOINFORMANTES [3].

Palabras clave:

Agentes Inteligentes, GLORIA, GALATEA, BIOINFORMANTES, Componente de Software Reutilizable, Enterprise Java Beans, J2EE, Aplicación Web Distribuida, Aplicación Web Cooperativa.

Agradecimientos

A Dios todo poderoso: por otorgarme las fuerzas necesarias.

A mis hijos Diego Alejandro y María Fernanda: principal fuente de motivación en el esfuerzo para cumplir cada uno de los objetivos que me he propuesto.

A mi madre: quién me ha estimulado siempre para seguir en la búsqueda de nuevas metas.

A mi esposa Marié: quién por mucho tiempo me ha apoyado y con mucho sacrificio asumido cada una de mis motivaciones de manera incondicional.

A mi tutor Jacinto Dávila: quién confió en mí y con paciencia hizo posible la culminación de este paso tan importante.

A los miembros de las dependencias que colaboran con el desarrollo y el financiamiento del proyecto BIOINFORMANTES: Centro de Simulación y Modelos (CESIMO).

Este trabajo fue parcialmente financiado por FONACIT bajo el proyecto: S1 2000000819.

Introducción

En los últimos años, la tecnología de agentes se ha convertido en el pilar de una gran cantidad de aplicaciones que, por su naturaleza o complejidad, requieren la incorporación de un ente basado en conocimiento. Esta tecnología, que proviene del campo de la Inteligencia Artificial (AI, Artificial Intelligence), se ha utilizado en la aplicación de modelos estructurados de conocimiento para afrontar con éxito la complejidad de problemas reales. Esta evolución ha permitido el desarrollo de la Ingeniería del Conocimiento, área de la informática en la que se diseñan y desarrollan arquitecturas cognitivas, esto es, arquitecturas en las que se emplean símbolos y abstracciones que representan el conocimiento y cuyos métodos de gestión, si son ejecutados convenientemente, permiten al sistema alcanzar un comportamiento inteligente [13]. De esta manera aparece, a principio de los ochenta, el concepto de *agente inteligente* definido como una entidad de software que percibe su entorno y es capaz de actuar sobre él para lograr sus objetivos [1]. Este paradigma de agente permite no sólo una nueva metodología de desarrollo, sino también una nueva forma de entender y enseñar lo que es la Inteligencia Artificial. Esto coincide con la evolución general hacia la modularidad y autonomía de componentes de las aplicaciones que permitan 1) su fácil integración en sistemas con objetivos complejos, 2) la inspección y mantenimiento independiente de los diversos componentes y 3) la reutilización de módulos, incluyendo todo ello en la fiabilidad del software y la eficacia en su desarrollo [13].

Una de las áreas en la que los agentes de software han alcanzado un gran nivel de impacto es el área de la enseñanza-aprendizaje. En este contexto la AI se ha utilizado en la búsqueda de nuevos métodos de enseñanza-aprendizaje asistidos por agentes artificiales inteligentes, dando lugar a lo que se conoce hoy como Tutores Inteligentes (ITs, Intelligence Tutors) [15].

Estos ITs son utilizados en dominios de conocimiento especializados y sirven para inferir, a través de las interacciones con el usuario y un mecanismo de razonamiento, una estrategia de enseñanza apropiada que permita aumentar el rendimiento en la enseñanza haciendo uso de software orientado a la educación [16].

En la actualidad, estos tutores inteligentes no han recibido todavía una aceptación general debido, principalmente, a la complejidad de su diseño lo que limita su aplicabilidad en la práctica.

Una de las aproximaciones actuales, en los campos vinculados a los ITs y la AI, basada agentes de software, es el diseño de asistentes personales que supervisen las acciones del usuario en un entorno informático para proporcionar ayuda. De forma más específica, los asistentes personales son llamados agentes de interfaz programados para cooperar con el usuario y alcanzar un objetivo común [16].

Así, el campo de la AI se ha enfocado en el desarrollo de estrategias tecnológicas para crear sistemas o programas inteligentes basados en conocimiento. Mientras tanto, la Ingeniería del Software ha ido desarrollando tecnologías para facilitar la construcción de ese tipo de programas siguiendo el enfoque del desarrollo basado en componentes.

En este sentido, los componentes de software reutilizables han sido, por omisión, la estrategia de desarrollo e integración que se utiliza hoy por hoy en la construcción de sistemas y aplicaciones de software. Esta tecnología permite, entre otras cosas, la creación de software a partir de software ya existente en lugar de desarrollarlo desde el comienzo [7,9].

El objetivo principal de este trabajo es el diseñar e implementar un agente de software, como componente de software reutilizable, que pueda desempeñarse como asistente y consultor de los usuarios de una aplicación Web distribuida y

cooperativa. Para alcanzar este objetivo, se integran un conjunto de tecnologías ya desarrolladas, o en desarrollo, que facilitan: la especificación del agente basado en lógica [5], la caracterización de una base de conocimientos para el agente [4,5], la implementación del agente en un lenguaje de programación [4] y la especificación de ese agente como componente de software reutilizable. Estas características conjugadas en un mismo dispositivo de software se denomina SIMULANTE.

En este sentido, SIMULANTE pretende aprovechar, específicamente, tecnologías como:

- GLORIA [5]: un modelo de agente basado en lógica, desarrollado en el lenguaje de programación PROLOG [17], que utiliza la lógica proposicional, y de predicados, para la caracterización de la base de conocimientos del agente;

- La implementación de ese agente en el lenguaje de programación Java [18] y

- Las especificaciones de la Versión Empresarial de Java (J2EE, Java 2 Enterprise Edition) para el desarrollo de componentes de software reutilizables con Java Beans Empresariales (EJB, Enterprise Java Beans) [14,19].

Este agente, como componente de software, es desplegado en una aplicación Web demostrativa que permite el uso del agente en un dominio de conocimiento orientado a la simulación de modelos escritos en GALATEA [2].

Finalmente, la descripción del agente, incluye los aspectos de su integración en aplicaciones J2EE. Para ello es necesario utilizar, como plataforma, una aplicación Web construida sobre las especificaciones de diseño J2EE y que además implemente componentes característicos de un sistema cooperativo y distribuido.

En este sentido, utilizamos la arquitectura de una aplicación Web ya construida llamada BIOINFORMANTES [3]: una plataforma orientada al procesamiento de data científica en un entorno virtual, distribuido y cooperativo.

En resumen, para cumplir el objetivo general de este trabajo, se proyectan tres objetivos específicos:

- Especificar un tipo de agente interfaz utilizando un modelo de agente basado en lógica.
- Especificar los requisitos de diseño e implementación para que ese agente interfaz pueda ser desplegado como componente de software reutilizable.
- Definir la ingeniería de conocimiento y métodos de integración del agente en una aplicación Web cooperativa.

Este documento está organizado de la siguiente manera:

En el capítulo 1 proporcionamos un marco teórico en el que incluimos algunos conceptos relacionados con: agentes de software; especificación de GLORIA como un agente lógico, reactivo y racional; características de los agentes interfaz o agentes asesores, componentes de software reutilizables; la tecnología EJB y las aplicaciones multicapas J2EE.

En el capítulo 2 mostramos la arquitectura de BIOINFORMANTES como una plataforma J2EE, mencionamos aspectos específicos de su diseño arquitectónico y los componentes de aplicación.

En el capítulo 3 damos una breve descripción de la plataforma de simulación GALATEA, mostramos como se construye un modelo de simulación GALATEA y cual es la implementación actual de esta plataforma.

En el capítulo 4 mostramos el diseño y la implementación de SIMULANTE como

componente de software reutilizable, describimos el dominio de conocimiento y los requerimientos funcionales del agente; caracterizamos la base de conocimiento; describimos la interfaz de interacción agente-ambiente y especificamos al agente como un componente EJB.

Por último, en el capítulo 5, mostramos la implementación de nuestro agente en una aplicación Web y cómo nuestro agente puede ser integrado a una plataforma Web multicapa basada en J2EE.

Marco teórico

1.1. Agentes en inteligencia artificial

Los sistemas de agentes constituyen un paradigma de programación para el desarrollo de aplicaciones de software [7]. Actualmente los agentes son el centro de interés en muchas ramas de la ingeniería informática e inteligencia artificial, y están usándose en una amplia y creciente variedad de aplicaciones. Algunas de las ventajas que se persiguen con la consolidación de la tecnología de agentes es la resolución de problemas de una forma significativamente mejor que la forma utilizada hasta el momento y, en algunos casos, problemas que no habían podido resolverse por no disponerse de la tecnología adecuada, o porque la utilización de la existente implicaba un costo excesivo [Posadas, citado en 7]. En este sentido el concepto de agente se ha convertido en una alternativa de programación para la resolución de problemas, siendo la inteligencia artificial el área que se ha encargado de su evolución desde el punto de vista tecnológico, adecuando conceptos propios de agentes reales en el ámbito artificial y definiendo herramientas para su diseño, desarrollo e implementación.

1.1.1. Concepto de agente

La palabra “Agente” (del latín *agens*) se refiere al productor de un efecto, una sustancia activa, una persona o cosa que ejecuta una acción, un representante [20]. Según el diccionario hispánico universal un agente “es el que obra o tiene virtud de operar; tradicionalmente llámese así a la persona, animal o cosa que realice la acción del verbo; persona o cosa que produce un efecto; persona que obra con poder de otra; persona que tiene a su cargo una agencia para gestionar asuntos ajenos o prestar determinados servicios”.

La tecnología de agentes en inteligencia artificial ha adoptado algunos términos de estas definiciones para llegar a conceptualizar lo que se conoce hoy como “agentes artificiales inteligentes”. En este contexto podemos encontrar un gran número de definiciones del concepto de agente, sin que ninguna de ellas haya sido plenamente aceptada por la comunidad científica, siendo quizás la más simple la de Russell [1], que considera que un agente es todo aquello que percibe su ambiente mediante sensores y que responde o actúa en tal ambiente mediante efectores. Basándose en esta definición, se pueden caracterizar distintos agentes de acuerdo a los atributos que posean (y que van a definir su comportamiento) [11] para resolver un determinado problema.

En la actualidad se usan diversas palabras para describir agentes: agentes inteligentes, interfaces inteligentes, interfaces adoptivas, knowbots (knowledge based robots), softbots (software robots), userbots, taskbots (task based robots), agentes personales, agentes autónomos, asistentes personales y agentes de red, solo por mencionar algunos. Cada uno programado con objetivos diferentes y prestando servicios diferentes.

La figura 1.1.1 muestra la relación que existe entre un agente y su ambiente [1].

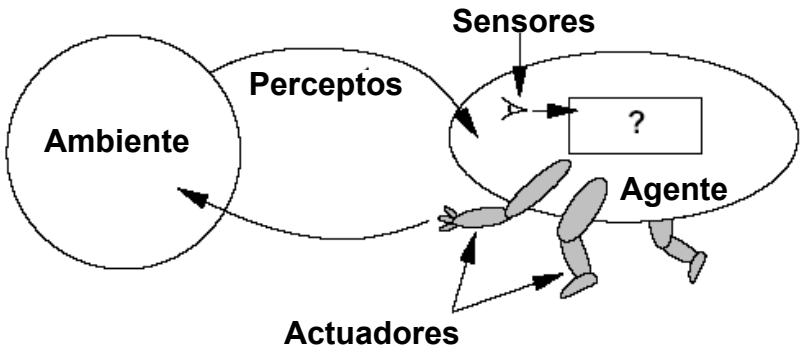

Figura 1.1.1: El agente y su ambiente

El *agente,* en su estructura interna, posee uno más sensores y una base de

conocimientos que describe el comportamiento del agente. Los *sensores* le permiten obtener, de su ambiente, un conjunto de observaciones, cada una de ellas llamadas *perceptos,* que son relacionadas con un conjunto de *acciones*. Estas acciones son ejercidas sobre el *ambiente* y vistas como el conjunto de influencias que ese agente causa sobre tal ambiente. De esta manera podemos definir a la *base de conocimiento* de un agente como la función que describe la relación entre las percepciones y las acciones.

Un agente posee ciertas propiedades para cumplir su rol de agente en un dominio de aplicación. Estas propiedades, en su conjunto, no necesariamente deben estar presentes en todos los tipos de agentes. Son propiedades deseables que un agente puede tener [1]:

- **Autonomía**: Los agentes actúan sin intervención humana directa o de otros agentes y tienen alguna clase de control sobre sus acciones y estado interno.

- **Sociabilidad**: Capacidad de interaccionar con otros agentes (incluso humanos) utilizando alguna clase de lenguaje de comunicación de agentes. Los agentes colaboran entre si para la ejecución de tareas.

- **Reactividad**: Percibe el entorno en el que está inmerso y responde de manera oportuna a cambios que tienen lugar en él (para actuar adecuadamente un agente debe de poder conocer en todo momento el "mundo" que le rodea).

- **Iniciativa** (proactividad): Tiene que tener un carácter emprendedor y tomar la iniciativa para actuar guiado por los objetivos que debe satisfacer.

- **Veracidad**: Un agente no comunica información falsa.

- **Benevolencia**: Un agente que está dispuesto a ayudar a otros agentes.

- **Racionalidad**: Un agente actúa de forma racional, intentando cumplir sus objetivos solo si son viables.

1.1.2. Tipos de agentes

Existen varios tipos de agentes, clasificados de acuerdo a la forma de relacionar las percepciones con las acciones [1]. Sin embargo, estos tipos de agentes tienen en común la teoría clásica de planificación de la inteligencia artificial simbólica: dado un estado inicial, un conjunto de operadores/planes y un estado objetivo, la deliberación del agente consiste en determinar qué pasos debe encadenar para lograr su objetivo [10]. De esta manera podemos encontrar los siguientes tipos de agentes [1]:

- Un **agente de reflejo simple**, tiene un conjunto de reglas condición/acción que le permite establecer una conexión entre las percepciones y las acciones. Este tipo de agente es **reactivo** y no guarda ningún tipo de estado interno, lo quiere decir que una misma entrada al agente siempre produce la misma salida.

- Un **agente con memoria,** basado en un agente reactivo pero que es capaz de guardar cierto estado interno que le permite discernir entre estados del mundo que generan la misma entrada de percepciones pero que necesitan acciones distintas. La actualización de esta información exige la codificación de dos tipos de conocimiento en el programa de agente. En primer lugar, se necesita cierta información sobre cómo está evolucionando el mundo, independientemente del agente. En segundo lugar, se necesita información sobre cómo las acciones del agente mismo afectan al mundo.

- Un **agente basado en metas** incorpora otro tipo de información que tiene que ver con las situaciones deseables. El programa de agente podría combinar esta información con la información relativa al resultado que

producirían las posibles acciones que se emprendan y elegir las acciones que permitan alcanzar la meta. En ocasiones esto es muy sencillo, cuando alcanzar la meta depende de responder con una sola acción; otras veces es más complicado, cuando el agente tiene que considerar largas secuencias de acciones para encontrar la vía que le lleve a cumplir la meta deseada. Para ello, la planificación juega un papel importante al buscar las acciones que permiten alcanzar las metas de un agente. De esta manera, los agentes basados en metas incorporan un tipo de razonamiento para seleccionar la (s) acción (es) adecuada (s). Si bien esto le resta eficiencia, en términos de procesamiento, comparado con un agente reflejo, también es cierto que les da más flexibilidad.

- Por último, tenemos los **agentes basados en utilidad** los cuales agregan una medida de desempeño a cada conjunto de acciones que permiten al agente conseguir su meta. Esto está relacionado con los diferentes estados del mundo que se pueden producir cuando un agente ejecuta uno u otro conjunto de acciones para encontrar su meta. Aquel conjunto de acciones que le otorgue al agente mayor utilidad, será la elegida para conseguir la meta. Por lo tanto, la utilidad es una función que correlaciona un estado y un número real mediante el cual se caracteriza el correspondiente grado de satisfacción.

A esta clasificación es conveniente agregar las consideraciones fundamentales para obtener una medida de desempeño que maximice la utilidad de las acciones que ejerce un agente sobre su ambiente. Esta medida de desempeño no es fácil de establecer de manera objetiva y debe ser delegada a una autoridad externa que defina la norma de lo que se considera un satisfactorio desempeño en un ambiente y emplearlo en la medición del desempeño de los agentes [1]. Una manera de establecer esto es definiendo un carácter de racionalidad en función a las percepciones y acciones del agente en un momento determinado. Este

carácter de racionalidad define el grado de éxito logrado por el agente y depende de la secuencia de percepciones del agente (todo lo que el agente ha percibido en un momento dado), del conocimiento que posea el agente acerca del medio y de las acciones que el agente pueda emprender [1].

En principio, es posible determinar qué mapeo describe acertadamente a un agente, ensayando todas las secuencias posibles y llevando un registro de las acciones que en repuesta emprende el agente. (Si el agente utiliza cierto azar en sus cálculos, será necesario probar algunas secuencias de percepciones varias veces, a fin de formarse una idea adecuada de la conducta promedio del agente.). Especificar qué tipo de acción deberá emprender un agente como respuesta a una determinada secuencia de percepciones constituye el diseño de un agente ideal [1].

De esta manera, se puede definir a un agente racional ideal como aquel agente que en todos los casos de posibles secuencias de percepciones emprende todas aquellas acciones que favorezcan obtener el máximo de su medida de rendimiento, basándose en las evidencias aportadas por la secuencia de percepciones y en todo conocimiento incorporado en tal agente [1].

1.1.3. Dominio y representación del conocimiento

El dominio de conocimiento para un agente está relacionado directamente con el área de aplicación externa del agente y sirve para construir su base de conocimientos. Todo agente está programado para hacer “algo”, ejecutado sobre el entorno de operación y basado en un conocimiento acotado que le permite discernir o razonar, de manera adecuada, las acciones correspondientes. Un dominio de conocimiento puede ser visto como el área de experticia del agente para el cual fue programado y sobre el cual dicho agente puede aprender. El conocimiento acotado del agente es función de la representación simbólica que se

le dé al dominio de aplicación.

Por su parte, la representación del conocimiento se refiere a la correspondencia entre el dominio de aplicación externo y el sistema de razonamiento simbólico. Esta representación está conformada por una estructura de datos para almacenar la información y los métodos que permiten manipular dicha estructura. De esta forma, para cada elemento relevante del dominio de aplicación del agente existe una expresión en la base de conocimiento del agente que representa dicho elemento.

Esta correspondencia entre los elementos del dominio de aplicación y del dominio del modelo permite a los agentes razonar acerca del dominio de aplicación al ejecutar procesos de razonamiento en el dominio del modelo y transferir las conclusiones de vuelta al dominio de aplicación [2].

1.1.4. Base de conocimientos

La base de conocimientos contiene la información disponible para el agente. Existen muchas alternativas de implementación de este almacén de conocimientos. Sin embargo, en el desarrollo de una base de conocimientos se pueden distinguir tres fases [2]:

- En la **incorporación** se define el conocimiento básico del agente. Es importante seleccionar el esquema de representación de conocimiento y desarrollar la terminología básica y la estructura conceptual de la base de datos. El resultado de esta fase es una base de conocimientos inicial, generalmente incompleta y compuesta por conocimiento parcialmente incorrecto que posteriormente será refinado y mejorado durante las siguientes etapas del desarrollo [2].

- En el **refinamiento** se extiende y se depura la base de conocimientos. El

resultado de esta fase debe ser una base de conocimientos sustancialmente completa y correcta que permita proporcionar soluciones correctas a los problemas que el agente debe resolver [2].

- En la **reformulación** se organiza la base de conocimientos para optimizar el proceso de contraste de reglas por parte de motor de inferencia del agente [2].

1.1.5. Razonamiento

La racionalidad de un agente se muestra como la capacidad de cada agente para asimilar los datos que recibe de su ambiente y, con esa información y con conocimiento válido de las reglas de cambio de su entorno, decidir como actuar para alcanzar sus metas e intenciones. Dotar a un agente con "racionalidad" significa proporcionarle los medios de representación de aquel conocimiento e información de su entorno y de los mecanismos para que pueda planificar (decidir sus acciones) para alcanzar sus metas [3].

Un proceso de razonamiento le permite a los agentes anticipar las consecuencias de las posibles acciones a ejecutar y así escoger la acción más "racional" [2], lo que le permite actuar de manera lógica ante el conjunto de percepciones recibidas. Este tipo de agente tiene una conducta programada en su base de conocimiento descrita como un conjunto de reglas que le permiten al agente establecer las relaciones entre sus percepciones y sus acciones. Así, el agente se comporta como un sistema intencional: un sistema con intenciones y actitudes tales como metas y creencias, describiendo una entidad autónoma que persigue las metas para las cuales fue programado [4,5]. Esta búsqueda define un plan de acción que le permite al agente modificar el mundo que lo rodea a través de sus mecanismos efectores.

Existen muchas estrategias para dotar de racionalidad a un agente. Una de las

estrategias más empleadas, dado que relaciona incluso ciertos mecanismos de aprendizaje, es el razonamiento abductivo [5], el cual se basa en que el agente adopte una hipótesis que explique sus observaciones y proponga el conjunto de acciones para alcanzar sus metas.

En el proceso de razonamiento abductivo puede utilizarse la representación clausal del conocimiento, facilitando así la exploración que se realiza para explicar las observaciones.

Nuestro trabajo busca definir un agente de software basado en el concepto propuesto por Russell [1], con propiedades de reactividad y racionalidad. Estas propiedades conjugadas en un mismo tipo de agente nos exige combinar dos modelos de agente aparentemente opuestos: un modelo de agente reactivo y un modelo de agente racional. Para lograr esto, nuestra especificación de agente se basa en el modelo de agente propuesto por Dávila [5], cuya concepción de agente integra componentes de reactividad y racionalidad en un mismo comportamiento, tomando como base el concepto de racionalidad acotada [5]. Estas propiedades permiten establecer la relación que existe entre las percepciones del agente y sus acciones.

Propiedades como autonomía, sociabilidad, iniciativa y veracidad definen el desempeño de nuestro agente en su entorno, lo que nos permite establecer qué tan bien se adapta nuestro agente y qué tan bien cumple con sus objetivos. Para lograr esto, es necesario establecer un carácter de racionalidad apropiado basándonos en la propuesta de Russell [1], lo cual nos permite llevar un registro de las acciones que emprende el agente como respuesta a una determinada secuencia de percepciones y, en función de ello, establecer una medida de rendimiento para el agente. Esta medida de rendimiento podrá ser constituida de acuerdo a las facultades del agente para incorporar, en sus acciones, niveles de autonomía, sociabilidad, iniciativa y veracidad en un dominio de aplicación especificado.

En este sentido, nuestro dominio de aplicación incorporará un tipo de conocimiento que especializa a nuestro agente como un agente interfaz que coopera con los usuarios de una aplicación orientada al modelado y simulación de sistemas. La base de conocimientos del agente se construye de acuerdo a la especificación propuesta por Dávila [5], programada en lenguaje lógico proposicional y de predicados. Esta base de conocimiento contiene el conjunto de observaciones que el agente puede percibir de su ambiente, el conjunto de reglas que definen el comportamiento del agente, los planes de acción del agente y el conjunto de influencias que, de acuerdo al comportamiento, el agente puede ejercer sobre su ambiente.

En la siguiente sección, se explica en detalle el modelo de agente de tipo reactivo y racional propuesto por Dávila [5], su descripción lógica, su mecanismo de razonamiento o motor de inferencia, la forma expresiva de una base de conocimientos para un agente de este tipo y la implementación de ese agente en un lenguaje de programación.

1.2. GLORIA: un agente lógico, reactivo y racional

En la sección 1.1.2 se presentaron los tipos de agentes y la manera en que éstos relacionan las percepciones con las acciones. En la sección 1.1.5 se presentó de manera general la forma de dotar a un agente de racionalidad y se mostró una estrategia de razonamiento llamada abducción que puede incorporar la representación clausal del conocimiento. Por último, se caracterizó, de manera general, la especificación de nuestro agente, basada en dos teorías: La teoría de agentes propuesta por Russell [1] y el modelo de agente reactivo y racional propuesto por Dávila [5]. En esta sección se presenta la especificación de ese modelo de agente en lógica clásica, que logra la reconciliación entre la reactividad y la racionalidad, ésta última basada en un proceso abductivo.

1.2.1. Definición

GLORIA o General-purpose, Logic-based, Open, Reactive and Intelligent Agent [descrito formalmente en 2 y 5] es un modelo de agente, propuesto por Dávila [5], que integra componentes de reactividad y de racionalidad en un mismo comportamiento. Un aspecto de esta concepción de agente cuya importancia es cada vez más apreciada, es el hecho de que la racionalidad de cualquier agente real es “acotada”. Hay restricciones concretas en tiempo de procesamiento, espacio de almacenamiento y estructuras cognitivas, que impiden que el agente pueda deducir todas las consecuencias de sus creencias y conocimiento [2].

Esta especificación de agente solapa el razonar con el actuar. Para ello, existen límites para el tiempo en el proceso de razonamiento y, al alcanzar esos límites, el agente actúa siguiendo planes que no han sido completamente elaborados. Es decir, de haber tenido más tiempo, el agente podría haber tomado otro curso de acción. El detalle clave en esta reconciliación de reactividad con racionalidad es que el agente va a estar siempre listo para la reacción, en comparación con un agente que trata de completar todo su racionamiento antes de actuar.

Para razonar, el agente utiliza descripciones lógicas de su entorno, metas y conocimiento, en forma acotada [5] produciendo “planes” a partir de una representación de la situación actual y de las metas del agente [4,5].

GLORIA establece que la relación del ambiente es, no sólo función de las influencias, y del estado actual del ambiente sino también de las leyes de cambio del sistema y en general del conocimiento que se tenga sobre el ambiente [4].

1.2.2. Descripción lógica del agente

GLORIA utiliza la lógica modal para representar nociones intencionales como conocimiento, creencias e incluso metas [4,5]. Utiliza la lógica clásica para

establecer una especificación de un agente de tipo reactivo-racional. El tipo de razonamiento incorporado al agente sigue el proceso abductivo explicado en la sección 1.1.5, aunque aplicado sobre un contexto diferente, en este caso la abducción es utilizada para explicar las metas [2].

En este sentido, la descripción del agente contiene:

- Una base de conocimiento con definiciones de la forma *si y solo si*, que es utilizada para reducir metas a sub-metas.

- Un conjunto de metas dadas en forma de átomos o implicaciones que hacen posible representar metas condicionales como reglas de condición-acción generalizadas y prohibiciones, acciones complejas o atómicas y que las observaciones del agente se incorporen a las metas.

- Una representación explícita del tiempo que utiliza el cálculo de eventos de la forma: *se tiene(paso libre,25/07:10:10)* y una especificación de la forma en que este tiempo avanza. En cada paso de la iteración, el agente limita su razonamiento a pequeñas porciones de tiempo y permite que se pueda reanudar en la próxima iteración.

En el proceso de reducción de metas a sub-metas, se arriba a una sub-meta atómica que no se puede reducir, pero que si se puede postular como condiciones que, de cumplirse, implicarían el logro de las metas superiores de donde se obtuvieron. Estas sub-metas atómicas son las acciones y un conjunto de estas acciones obtenidas para una meta, constituyen un plan. De esta manera es como la abducción se convierte en el mecanismo de planificación del agente. Así, lo que el agente conoce de su ambiente determina sus planes de acción [2].

La evolución en el tiempo T que describe la vida de un agente esta determinada por los siguientes cuatro pasos [2]:

- Observar la porción del estado del ambiente a la cual tiene acceso en T, asimilar la observación como una meta y registrar dicha observación en la base de conocimientos.

- Ejecutar el procedimiento de reducción de metas a sub-metas en T + 1. Este procedimiento permite propagar las observaciones, durante R unidades de tiempo.

- Seleccionar entre las metas que se ofrecen como alternativa de acciones atómicas que puedan ser ejecutadas y postular dichas acciones en el tiempo T + R + 2.

- Repetir el ciclo en el tiempo T + R + 3.

Estos cuatro pasos están incluidos en un predicado llamado *cycle()*, que describe la evolución del tiempo y describe la especificación básica en lógica del agente reactivo-racional tal como se muestra en le figura 1.2.1. Esta descripción, denominada GLORIA, explica el comportamiento de un agente a través de los siguientes predicados a partir del predicado *cycle() [2,5]*:

- *demo(),* corresponde al paso 2 del predicado *cycle()*. Permite reducir metas (Goals) a sub-metas (Goals') a partir de la base de conocimientos KB controlando además el uso de la cantidad de recursos R disponibles para llevar a cabo este proceso de razonamiento.

- *act()* corresponde al paso 3 del predicado *cycle()*. Cambia la estructura mental del agente, si en T_a el plan preferido del agente **PreferredPlan** contiene acciones **TheseActions** que puedan ser ejecutadas en T_a, estas acciones se postulan en paralelo y se incorpora nuevas metas con la información obtenida como respuesta.

- *executables()* permite obtener el grupo de acciones **NextActs**

correspondiente a la lista de intenciones **Intentions** que pueden ser ejecutadas en T_a.

GLORIA's cycle

$$cycle(KB, Goals, T)$$
$$\leftarrow demo(KB, Goals, Goals', R)$$
$$\wedge\ R \leq n$$
$$\wedge\ act(KB, Goals', Goals'', T+R)$$
$$\wedge\ cycle(KB, Goals'', T+R+1) \qquad [\mathbf{GLOCYC}]$$

$$act(KB, Goals, Goals', T_a)$$
$$\leftarrow Goals \equiv PreferredPlan \vee AltGoals$$
$$\wedge\ executables(PreferredPlan, T_a, TheseActions)$$
$$\wedge\ try(TheseActions, T_a, Feedback)$$
$$\wedge\ assimilate(Feedback, Goals, Goals') \qquad [\mathbf{GLOACT}]$$

$$executables(Intentions, T_a, NextActs)$$
$$\leftarrow \forall A, T(\ do(A,T)\ is\_in\ Intentions$$
$$\wedge\ consistent((T = T_a) \wedge Intentions)$$
$$\leftrightarrow\ do(A, T_a)\ is\_in\ NextActs\ ) \qquad [\mathbf{GLOEXE}]$$

$$assimilate(Inputs, InGoals, OutGoals)$$
$$\leftarrow \forall A, T, T'(\ action(A, T, succeed)\ is\_in\ Inputs$$
$$\wedge\ do(A, T')\ is\_in\ InGoals$$
$$\rightarrow\ do(A, T)\ is\_in\ NGoal\ )$$
$$\wedge\ \forall A, T, T'(\ action(A, T, fails)\ is\_in\ Inputs$$
$$\wedge\ do(A, T')\ is\_in\ InGoals$$
$$\rightarrow\ (\mathbf{false} \leftarrow do(A, T))\ is\_in\ NGoal\ )$$
$$\wedge\ \forall P, T(\ obs(P, T)\ is\_in\ Inputs$$
$$\rightarrow\ obs(P, T)\ is\_in\ NGoal\ )$$
$$\wedge\ \forall Atom(\ Atom\ is\_in\ NGoal$$
$$\rightarrow\ Atom\ is\_in\ Inputs$$
$$\wedge\ OutGoals \equiv NGoal \wedge InGoals \qquad [\mathbf{GLOASSI}]$$

$$A\ is\_in\ B \leftarrow B \equiv A \wedge Rest \qquad [\mathbf{GLOISN}]$$

$$try(Output, T, Feedback) \leftarrow\ tested\ by\ the\ environment.. \qquad [\mathbf{TRY}]$$

Figura 1.2.1: Especificación lógica de GLORIA

- *try()* Intenta ejecutar la lista de acciones **Output** en T obteniendo como respuesta **Feedback**.

- *assimilate()* corresponde al paso 1 del predicado *cycle()*. Permite actualizar

la base de conocimientos ya que asimila las entradas **Inputs** las incorpora con las metas del agente **InGoals** para obtener el nuevo conjunto de metas **OutGoals** que incorpora la información correspondiente a la entrada dada.

1.2.3. Proceso de razonamiento

Los límites en el razonar son restricciones numéricas sobre el tiempo de razonamiento. Como se muestra en la figura 1.2.2, un agente con tiempo limitado para razonar está caracterizado por una función de conducta [4,5].

La función *actualiza$_a$()* describe cómo la base de conocimientos del agente K_a se actualiza con el registro de un conjunto de perceptos. Estas percepciones son obtenidas del entorno con la función *percibe$_a$()*, donde la representación de los perceptos se describe como *Obs(P,t)* para indicar que el agente observa la propiedad P en el tiempo t.

$$\begin{aligned} Conducta &: \mathcal{T} \times R_a \times K_a \times \Gamma \rightarrow K_a \times \Gamma \\ < k'_a, g'_a, \gamma'_a > &= conducta_a(t, r_a, k_a, g_a, \gamma) \end{aligned}$$

con

$$k_a, g_a \in K_a, \qquad \gamma_a \in \gamma, \qquad r_a \in R_a$$

t : tiempo actual.
r_a : cantidad de tiempo para razonamiento del agente.
k_a : base de conocimientos del agente.
g_a : conjunto de metas del agente.
γ_a : conjunto de influencias que postula el agente.

donde $conducta_a()$ hace referencia a

$$\begin{aligned} Actualiza &: \mathcal{T} \times \Gamma \times K_a \rightarrow K_a \\ k'_a &= actualiza_a(t, percibe_a(\gamma), k_a) \end{aligned}$$

y

$$\begin{aligned} Planifica &: \mathcal{T} \times R_a \times K_a \rightarrow \Gamma \times K_a \\ < \gamma'_a, g'_a > &= planifica_a(t, r_a, k'_a, g_a) \end{aligned}$$

Figura 1.2.2: Caracterización de un agente con razonamiento acotado

El planificador de tareas del agente, representado por la función $planifica_a()$,reduce las metas del agente g_a a un nuevo conjunto de metas g'_a y las influencias que serán postuladas al ambiente γ'_a empleando para ello las reglas de conocimiento e información sobre su entorno k'_a. El proceso de planificación toma en cuenta el tiempo actual t y no puede prolongarse por más de r_a unidades de tiempo. Alterando el factor r_a, podemos forzar una conducta más reactiva (un r_a más pequeño hace que el agente observe su entorno con mucha frecuencia y por ende tardaría más en llegar a "conclusiones profundas" acerca de su forma de actuar) ó más racional (un r_a más grande hace que el agente razone más profundamente antes de revisar su conocimiento del ambiente).

$planifica_a()$ equivalente al predicado *demo()* [5].

1.2.4. GLORIA en programación lógica

La especificación lógica de GLORIA, mostrada en la sección 1.2.3, fue traducida, casi literalmente, por Dávila [5], al lenguaje de programación lógica Prolog [12], como un motor de inferencia para agentes programados en lógica proposicional y lógica de predicados[5].

La base de conocimientos de un agente particular, que utiliza este motor de inferencia programado en Prolog, puede ser escrita haciendo uso de lógica proposicional y de predicados. Esta base de conocimiento debe poseer el conjunto de observaciones que el agente puede percibir de su ambiente, el conjunto de reglas que definen el comportamiento del agente, los planes de acción del agente y el conjunto de influencias que, de acuerdo al comportamiento, el agente puede ejercer sobre su ambiente.

En este contexto, un agente GLORIA ejecuta un ciclo que se basa en la obtención de observaciones, un proceso de razonamiento que permite la selección de un plan de acción de acuerdo a sus metas, y con esto el conjunto de acciones que el

agente efectuará sobre su ambiente.

Las observaciones, son un conjunto de átomos, que forman parte de la base de conocimientos del agente y sirven para describir qué cosas puede observar el agente desde su ambiente.

Para seleccionar un plan de acción, el agente posee, en su base de conocimientos, un conjunto de reglas. Estas reglas relacionan observaciones con planes de acción.

Un plan de acción puede corresponder a una sola acción en una regla de condición/acción. Sin embargo, existen planes más complejos donde interviene un conjunto de acciones que deben ser ejecutadas en su totalidad.

La figura 1.2.3 [5] muestra una descripción de un agente GLORIA muy simple que tiene un solo plan asociado. El agente representa a un “político” cuya meta es conseguir votos en unas “elecciones”.

```
executable(solicita_apoyo).
executable(ofrece_tierra).

observable(tienes_tierra).
observable(observo_colono).

% Integrity constraints.
si observo_colono, tienes_tierra entonces negocia_tierra_por_voto.

% Definitions
para negocia_tierra_por_voto haga
        solicita_apoyo,
        ofrece_tierra.

observe tienes_tierra.
observe observo_colono.
```

Figura 1.2.3: Ejemplo de un agente en GLORIA

Las posibles observaciones del agente están descritas en el predicado “observable”. Las acciones que el agente puede ejecutar están descritas en el predicado “executable” y se relacionan directamente con el plan de acción llamado

“negocia_tierra_por_voto”.

1.2.5. Implementación del agente en Java

La figura 1.2.4 [2] muestra la implementación actual del agente [2] donde se reflejan los métodos que corresponden a los predicados mostrados en la descripción lógica del agente y el proceso de razonamiento del agente (sección 1.2.3 y 1.2.4 respectivamente) [4]:

- cycle(). Este método corresponde al predicado cycle(), y a la función Evolución
- observe(). Informa al agente el estado del ambiente. Este método esta asociado al predicado act() y a la función *percibe$_a$().*
- execute(). Comunica al ambiente las intenciones del agente. Este método esta asociado al predicado act() y a la función Acción.
- reason(). Implementa los mecanismos de adquisición de conocimiento del agente. Este método corresponde al predicado demo() y a la función Planifica.

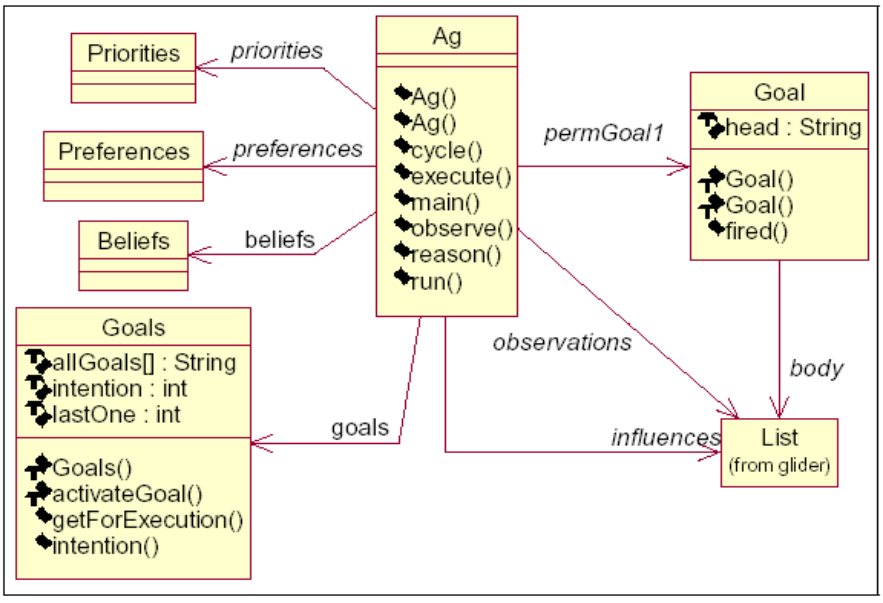

Figura 1.2.4: Diagrama de clases de la implementación del agente

Por otro lado, en la figura 1.2.4 se muestran también las estructuras de datos para almacenar las metas (Goals), las creencias (Beliefs), las preferencias (Preferences), las prioridades (Priorities) y una meta permanente (permGoal1) del agente. Además se evidencia que el manejo de la información se hace a través de listas: *observations* contiene las percepciones del agente mientras que *influences* contiene las influencias que postula el agente.

La figura 1.2.5 muestra un código fuente en Java que implementa la especificación del agente GLORIA, basado en el diagrama de clases de la figura 1.2.4:

```
public class Ag {
    List observations;
    List influences;
    Goals goals;
    Beliefs beliefs;
    Goal permanentGoal1;
/** Agent constructor initiates all the structures. */

    public Ag() {
        observations = null;
        influences = null;
        goals = new Goals();
        beliefs = null;
            // As a test, consider this "permanent goal
        List rains = new List("It rains");
        permanentGoal1 = new Goal("carry umbrella", rains); }
/** The main cycle/locus of control of the agent */
    public void cycle() {
        observe();
        reason();
        List result = execute();
        result.writeAll();
        cycle(); }
/** It's used to try execute the intentions. */
    public List execute() { return influences; }
/** It's used to update the knowledge of the environment. */
    public void observe() {
            // Get inputs from the environment.. somehow..
        List obs = new List("it rains");
            // Update its records.
        observations = obs; }
/** Very simple implementation of the reasoning engine. */
    public void reason() {
            // Every goal must be checked against observations..
        if (permanentGoal1.fired(observations)) {
            goals.activateGoal(permanentGoal1); };
        influences = new List(goals.allGoals[goals.intention]); }
/** Auxiliary method to test the agent. */
    public static void main(String argv[]) {
        Ag agent = new Ag();
        agent.permanentGoal1.body.writeAll();
        agent.cycle(); }}
```

Figura 1.2.5: Implementación del agente en Java

Nuestro trabajo, busca implementar un agente de software basado en la descripción de GLORIA como agente reactivo-racional, que nos permita la producción de planes a partir de una representación de la situación actual del ambiente y de las metas del agente. Utilizaremos la implementación del agente escrita en Java mostrada en la sección 1.2.5 y una base de conocimientos escrita en lógica proposicional y de predicados como la mostrada en la sección 1.2.4.

Las posibles observaciones, planes y acciones de nuestro agente serán escritas de acuerdo al dominio de conocimiento y de aplicación. Nuestro agente será implementado en una aplicación de software orientada al procesamiento de datos y, su principal función, es servir como un agente asistente y consultor que facilite el uso de las diferentes herramientas proporcionadas por la aplicación a usuarios potencialmente inexpertos. Para lograr esto, es importante caracterizar a nuestro agente. Una vez especificado el modelo lógico del agente a implementar, es necesario definir el tipo de agente de acuerdo al dominio de aplicación. Esto nos permite acotar el alcance de aplicación del agente de acuerdo a descripciones funcionales implementadas en el ámbito de la Inteligencia Artificial.

En la siguiente sección, se describen algunos tipos de agentes artificiales que pueden ser adecuados para caracterizar a nuestro agente como un agente inteligente orientado a procesos de enseñanza, específicamente de asistente o consultor dentro de un dominio de aplicación. En la sección 1.3.1, se describe lo que es un agente interfaz y cómo puede ser aplicado en nuestro dominio de conocimiento; en la sección 1.3.2 se definen algunos tipos de agentes interfaz y cómo han sido utilizados en el área de la enseñanza-aprendizaje.

1.3. Agentes programados para asesorar

Introducción

En la sección 1.1 se explicó de manera conceptual las características de los agentes artificiales, los tipos de agentes y las funciones que éstos pueden cumplir en el mundo de la informática. Dijimos que los agentes son un paradigma de programación centro de interés en muchas ramas de la computación.

Hoy en día son muchas las aplicaciones que se han dado a los agentes artificiales y el área del enseñanza-aprendizaje no ha escapado a ello. En este contexto la inteligencia artificial se ha utilizado en la búsqueda de nuevos métodos de enseñanza-aprendizaje asistidos por agentes artificiales inteligentes, dando lugar a lo que se conoce hoy como Tutores Inteligentes (ITs, Intelligence Tutors) [15].

Estos ITs son utilizados en dominios de conocimiento especializados y sirven para inferir, a través de las interacciones con el usuario, una estrategia de enseñanza apropiada que permita aumentar el rendimiento en la enseñanza haciendo uso de software orientado a la educación [15].

En la actualidad, estos tutores inteligentes no han recibido todavía una aceptación general debido, principalmente, a la complejidad de diseño y definición de los mismos que limitan su aplicabilidad en la práctica.

Una de las aproximaciones actuales, en los campos vinculados a los ITs y la AI, basada agentes de software, es el diseño de asistentes personales que vigilen las acciones del usuario en un entorno informático con el objetivo de proporcionar ayuda. De forma más específica, los asistentes personales son agentes de interfaz que cooperan con el usuario para alcanzar un objetivo. Esta cooperación estrecha entre el usuario y el agente permite al usuario aumentar considerablemente su rendimiento [15].

Un agente interfaz (también llamado personal) es un agente que provee a su usuario de la asistencia necesaria para facilitar su interacción con una determinada aplicación [17]. Según este planteamiento, y por la definición básica de agente discutida en la sección 1.1, un agente interfaz estaría situado en un entorno. Este entorno, para nuestro agente, sería una aplicación Web colocada en Internet. Por otra parte en la definición se indica que debe percibir y actuar en dicho entorno. Para un agente interfaz, la percepción se puede ver en el hecho de que el agente va recibiendo e instruyéndose de las peticiones que se le realizan, mientras que la acción queda patente cuando el agente muestra información que él mismo ha podido procesar desde y hasta la aplicación.

En nuestra propuesta el agente interfaz se encarga de ayudar al usuario en un conjunto de acciones para la manipulación de la aplicación, al mismo tiempo que sirve como consultor de la aplicación en un dominio de conocimiento especificado.

1.3.1. Definición de un agente interfaz

Un agente interfaz es un programa que utiliza técnicas de AI para proveer asistencia al usuario de alguna aplicación de cómputo. El agente busca continuamente la oportunidad de realizar tareas que ayuden a la satisfacción de los objetivos del usuario. Agente y usuario establecen una especie de contrato donde el usuario da a conocer sus objetivos y el agente ofrece sus capacidades para el logro de estos objetivos. Un agente interfaz “conoce” los intereses del usuario y pueden actuar con autonomía en beneficio de éste [17].

La **comunicación** entre el agente y sus usuarios debe ser lo suficientemente compleja como para permitir a ambas partes dar a conocer sus intenciones y habilidades. Además debe permitir una retroalimentación en ambas direcciones [17].

1.3.2. Aproximaciones de agentes interfaz

Existen varias aproximaciones al diseño de agentes interfaz [MAE93 citado en 17], entre las cuales conviene destacar:

- **La aplicación como agente interfaz**. Esta aproximación consiste en hacer que la aplicación misma tenga la funcionalidad de un agente interfaz. El ejemplo que revisaremos brevemente es Object Lens (Oval) [LAI88 citado en 17]. En Oval se utilizan agentes semi-autónomos que están constituidos por un conjunto de reglas programadas por el usuario. Estas reglas determinan cómo se procesa la información relativa a una tarea particular.

  La principal desventaja de este enfoque se debe a que la programación del agente recae sobre el usuario de la aplicación, de tal forma que:

  - El usuario debe reconocer la oportunidad de utilizar el agente.
  - Tomar la iniciativa de crear el agente.
  - Proveer al agente de conocimiento explícito en forma de reglas (lenguaje abstracto).
  - Dar mantenimiento a las reglas del agente.

- **Basado en el conocimiento**. La idea central de este enfoque es utilizar un modelo del usuario y un modelo de la aplicación, de donde el agente pueda extraer el conocimiento necesario para reconocer los planes del usuario y encontrar la oportunidad de contribuir con ellos. Este es el enfoque más utilizado en trabajos sobre interfaces inteligentes [SUL91 citado en 17]. Como ejemplo revisaremos el caso de UCEgo [CHI91 citado en 17]. UCEgo es un agente que forma parte del sistema UC (UNIX Consultant) cuyo objetivo es auxiliar a un usuario del sistema operativo UNIX en sus tareas. El usuario mantiene un dialogo en lenguaje natural (inglés) con UC por

medio de un analizador de lenguaje llamado ALANA que produce una representación semántica (red KODIAK). La representación es refinada y sirve de entrada a un componente analizador de metas llamado PAGAN, el cual entrega los planes y metas del usuario al agente UCEgo. Con esta información, UCEgo genera objetivos en UNIX que entrega a UCPlanner, un planificador en el dominio del sistema UNIX que regresa planes UNIX a UCEgo para que éste los comunique a UCExpress que se encarga de generar una red de KODIAK que es utilizada por un módulo UCGen para generar una respuesta al usuario en lenguaje natural.

Las desventajas de este enfoque son las siguientes:

- El diseño es muy dependiente del dominio. Al cambiar de dominio es difícil aprovechar la arquitectura de control desarrollada y el trabajo en ingeniería del conocimiento llevado a cabo.

- El mantenimiento de los modelos del usuario y aplicación es muy complicado. Generalmente los modelos permanecen fijos una vez terminado el trabajo de ingeniería de conocimiento. Esto constituye una desventaja seria en dominios de aplicación dinámicos y difíciles de predecir.

Para dominios donde los usuarios realizan su trabajo de forma muy similar, este enfoque es el más apropiado. [ETZ95] reporta el trabajo sobre agentes inteligentes en el Internet, generalmente desempeñando el papel de guía de visita o manteniendo índices. Su equipo de trabajo también ha trabajado sobre agentes para UNIX.

- **Basado en aprendizaje**. La tercera alternativa para la construcción de interfaces inteligentes descansa fuertemente en las técnicas de aprendizaje automático. La idea central es que a partir de un conocimiento mínimo de la aplicación, el agente aprenda a realizar su trabajo observando al usuario y

a otros agentes.

Existen dos condiciones necesarias para poder aplicar este enfoque:

- El uso de la aplicación involucra una cantidad considerable de comportamiento repetitivo. La repetición hace posible el aprendizaje.
- Este comportamiento repetitivo es potencialmente diferente para cada usuario. Si todos los usuarios se comportan de la misma manera, el enfoque basado en el conocimiento es más apropiado.

El agente tiene cuatro fuentes posibles de aprendizaje:

- El usuario. El agente “observa” las acciones del usuario guardando un registro de estas. Sobre este registro, el agente busca regularidades y patrones recurrentes que pueda automatizar.
- Retroalimentación indirecta. El usuario rechaza la sugerencia del agente y lleva a cabo una acción diferente a la propuesta. Puede haber también una retroalimentación negativa explícita (no vuelvas a hacer esto).
- Entrenamiento. El usuario puede dar ejemplos de situaciones e indicar al agente que hacer en cada caso. El agente guardaría los ejemplos en su registro y procedería como en el aprendizaje por medio del usuario.
- Otros agentes. Si un agente no sabe que hacer ante una situación, puede presentar el caso a otros agentes en el mismo dominio.

Este enfoque presenta las siguientes ventajas:

- El usuario final y el programador de la aplicación tienen menos trabajo.

- El agente se adapta a la forma de trabajar de distintos usuarios.

Actualmente existen agentes basados en aprendizaje trabajando en correo electrónico, agenda, filtrado de información en los “news groups”, recomendaciones musicales y literarias [MAE94].

Esta clasificación o aproximaciones de agente interfaz, nos permite definir a nuestro agente como un agente basado en el conocimiento. La descripción del modelo de usuario se basa en el uso de una herramienta común para todos los usuarios. El trabajo de estos usuarios está estandarizado y el agente funciona como asistente de la herramienta para facilitar su uso. El modelo de aplicación está basado en el uso de un lenguaje de simulación llamado GALATEA [2] que funciona como una herramienta de uso cooperativo asistida por agentes de software. En este sentido, nuestro agente interfaz conjuga dos roles: el rol de asistente y el rol de consultor en el modelo de aplicación.

De esta manera, hemos completado las bases que describen, especifican y caracterizan nuestro agente de software dentro de un dominio de aplicación especificado. En resumen, nuestro agente estará basado, principalmente, en lo siguiente:

- La descripción lógica de GLORIA, definida en la sección 1.2, para obtener: un modelo lógico de agente, un mecanismo de razonamiento, un motor de inferencia, un marco para la definición de la base de conocimientos del agente y la implementación de ese agente en Java.

- La descripción de un agente interfaz basado en el conocimiento, definido en la sección 1.3, para caracterizar a nuestro agente dentro de un dominio de aplicación orientado al proceso de enseñanza-aprendizaje, específicamente, con roles de asistente y de consultor en ese dominio.

- Un modelo de pruebas y producción de nuestro agente basado en el carácter de racionalidad propuesto por Russell y descrito en la sección 1.1.

1.4. Componentes de software reutilizables

1.4.1. Reutilización

El reuso es, por omisión, la estrategia de resolución de problemas en la mayoría de las actividades del ser humano. Los científicos cognitivos coinciden en afirmar que lo primero que hacemos cuando enfrentamos un problema es determinar si éste ha sido resuelto antes; en caso contrario, buscamos en nuestro espacio mental problemas análogos que ya se han resuelto y adaptamos su solución al problema actual; cuando ninguna de las anteriores situaciones se cumple, utilizamos entonces las habilidades y el conocimiento general analítico para la resolución de problemas. Se puede hablar de reuso en diferentes grados: reuso informal e intuitivo que se basa en el conocimiento individual de las personas, reuso esquematizado por medio de la utilización de procedimientos que guían el desarrollo de actividades y reuso orientado a productos cuando la reutilización se concreta en artefactos que representan el conocimiento y que facilitan la labor de reutilizar [Anaya de Páez, citado en 7].

1.4.2. Reutilización de software

En el campo de la ingeniería del software el reuso ofrece un gran potencial en términos de productividad y calidad del software. Productividad, porque amplifica la capacidad de programación, en el sentido de escribir menos código, reduce la cantidad de documentación y pruebas que se deben realizar y genera un efecto de sinergia sobre la funcionalidad del sistema completo a partir de la funcionalidad de sus componentes. Calidad, porque el diseño de componentes se realiza pensando

en su posterior utilización, con una documentación precisa, con procesos certificados de prueba y validación y con una estructuración adecuada de las partes del componente para que sea entendible por el usuario.

La demanda de sistemas de información cada vez más complejos, la dinámica de cambio en las organizaciones y la globalización de las operaciones del negocio bajo unas restricciones de tiempo, costo y calidad, hacen de la reutilización, una necesidad imperiosa dentro de todo proceso de ingeniería de software. En este contexto la reutilización puede ser definida como la propiedad de utilizar conocimiento, procesos, metodologías o componentes de software ya existente para adaptarlo a una nueva necesidad, incrementando significativamente la calidad y productividad del desarrollo [Frakes & Ferry, citado en 7].

1.4.3. Definiciones de reutilización de software

La reutilización de software "es el proceso de crear sistemas de software a partir de software existente, en lugar de desarrollarlo desde el comienzo" [8].

"...es el uso de componentes de software en un nuevo contexto, bien en cualquier parte del mismo sistema o en otro sistema" [Braun, 1994, citado en 7].

"... la capacidad de un componente de software desarrollado de ser utilizado de nuevo, parcial o totalmente, con o sin modificación" [Lim, 1994, citado en 7].

"La reutilización de software es el proceso de implementar o actualizar sistemas de software usando activos de software existentes" [Sodhi & Sodhi, 1999, citado en 7].

La reutilización de software va más allá de la reutilización de componentes de software. Involucra el reuso de otros elementos de software, tales como [7]:

- Algoritmos,

- Diseños,
- Arquitecturas de Software,
- Documentación y
- Especificación de requerimientos.

La reutilización de software es un proceso de la Ingeniería de Software que conlleva el reuso de activos de software durante las diferentes fases del proceso de desarrollo, como son [7]:

- La especificación,
- El análisis,
- El diseño,
- La implementación y
- Las pruebas de una aplicación o sistema de software.

1.4.4. Componentes de software reutilizables

En el desarrollo de sistemas programados se pueden reutilizar muchas cosas, por ejemplo, algoritmos, diseños, especificación de requerimientos, procedimientos, módulos, aplicaciones, ideas, patrones de diseño, arquitecturas, etc. [8].

Un componente es una pieza identificable de software que describe o libera un conjunto manejable de servicios que sólo pueden ser usados a través de una interfaz bien definida [7].

Un componente es parte de una estructura de mayor tamaño, un elemento que contribuye a la composición de un todo, con una interfaz bien definida.

Un componente de software reutilizable es una unidad de software auto contenida, modular y reemplazable que encapsula su implementación, hace visible su funcionalidad a través de un conjunto de interfaces, se integra con otro componente mediante sus interfaces para formar un sistema u otro componente mayor, puede ser desplegado (instalado y ejecutado) independientemente de los otros componentes en una aplicación [8,9].

Cualquier componente que exista, bien sea físico, lógico o conceptual debería siempre exhibir las siguientes características esenciales [9]:

- Ser identificable: Debe tener una identificación clara y consistente que facilite su catalogación y acceso.

- Es autocontenido: Un componente no debe requerir la reutilización de otros componentes para cumplir su función. Si el componente requiere de otros, el conjunto de componentes o funciones, visto como un todo, debe ser considerado como un componente de más alto nivel. Por ejemplo, los paquetes en Java permiten agrupar varias clases en un componente al cual se le asocia una funcionalidad bien definida.

- Debe ser genérico (dentro de su dominio de aplicación).

- Es rastreable: Que se le pueda hacer un seguimiento a lo largo de la vida del componente.

- Reemplazable: que pueda ser remplazado por una nueva versión o por otro componente que ofrezca la misma funcionalidad.

- Debe poseer una interfaz claramente definida y ser accesible solo a través de su interfaz.

- El servicio que se ofrece a través de la interfaz no debe cambiar. La implementación de los servicios puede cambiar, pero el servicio

propiamente dicho no debe hacerlo.

- Los servicios que ofrece el componente deben ser documentados de forma clara y precisa.

Además de las características antes mencionadas es deseable que un componente de software posea otras características [9].

- Encapsulado. El término encapsulado se refiere a la combinación de información y comportamiento dentro de un objeto. El encapsulado protege la información interna de un objeto y oculta la complejidad interna de su comportamiento.

- El reuso del servicio ofrecido no debe estar restringida por la implementación física. (1) el reuso debe ser independiente del lenguaje y de la herramienta de desarrollo, (2) debe ser reutilizable desde diferentes plataformas y Sistemas Operativos.

- Que pueda ser reusado y ensamblado dinámicamente.

- Ofrecer servicios genéricos.

- Técnicas de certificación y autenticación que habiliten aplicaciones para asegurar que el servicio correcto está siendo usado.

Es importante prestar atención a que es lo que el componente hace y no como lo hace. La implementación física de un componente es oculta. De esta forma los cambios a la implementación del componente son aislados de la aplicación que la usa [7,9].

Un componente puede proveer diversos servicios, pero cada servicio debe tener su propia interfaz, para facilitar y hacer más rápido el uso del servicio.

Las interfaces son fundamentales habilitadores críticos de la composición. Son el

enlace entre los requerimientos de negocios y la implementación física. También es el contacto entre el consumidor y el proveedor del servicio [7].

Como se indicó en las secciones 1.2 y 1.3, nuestro trabajo se basa en la especificación de un agente inteligente de software basado en el modelo lógico de agente GLORIA y caracterizado como un agente interfaz de conocimiento. Sin embargo, esa especificación de agente parece estar restringida al uso de aplicaciones de software que impliquen la implementación del agente en ambientes no distribuidos y monousuarios. Además, su aplicabilidad practica está completamente ligada a un domino de conocimiento específico, sin un marco de desarrollo que permita, de manera esquematizada, la definición de nuevas bases de conocimiento que puedan ser empleadas en otros dominios de aplicación.

Para solucionar esto, y apoyándonos en técnicas de reutilización e ingeniería del software, proponemos transformar nuestra especificación de agente en un componente de software reutilizable que cumpla con los aspectos de productividad y calidad de software descritos en la sección 1.4.2.

Igualmente, basándonos en algunas de las definiciones de componentes de software reutilizables de la sección 1.4.3, podemos resaltar, como las más acopladas a nuestra aplicación, las de Braun y Lim, los cuales convergen en que la reutilización de software tiene que ver con la capacidad de un componente de software para ser utilizado y reutilizado en uno o varios sistemas, parcial o totalmente, con o sin modificaciones. Según esa definición y la teoría de Montilva [9], podemos definir qué elementos de nuestro agente son potenciales de software que pueden ser caracterizados como componentes o elementos de software reutilizables.

En este sentido, proponemos definir un marco de desarrollo, apoyado en técnicas de reutilización de software, para representar la base de conocimiento de nuestro

agente y la especificación de sus requerimientos. Este marco de desarrollo, debe definir una técnica para escribir, de manera esquematizada, la base de conocimientos de nuestro agente de acuerdo a un dominio de aplicación particular, relacionado con agentes de tipo interfaz. De igual manera, este marco de desarrollo debe incluir una descripción detallada de los pasos a seguir para incorporar e implementar la nueva base de conocimientos en el nuevo agente interfaz y la puesta en producción de ese agente.

Finalmente, proponemos la especificación completa de nuestro agente como un componente de software reutilizable. Utilizaremos la implementación del agente en Java, descrita en la sección 1.2.5 y, apoyados en los conceptos y características de los componentes de software reutilizables, descritos en la sección 1.4.4, definimos nuestro agente como: una pieza de software que contribuye a la composición de un todo, que describe o libera un conjunto manejable de servicios que pueden ser usados a través de una interfaz definida.

De esta manera, nuestra especificación de agente puede ser completamente transformada y envuelta como un componente de software reutilizable, con las características esenciales de los componentes de software reutilizables descritas en la sección 1.4.4.

Esta descripción nos proporciona una modelo de desarrollo conceptual de nuestro agente para ser especificado como componente de software reutilizable. En la siguiente sección, se presenta una tecnología concreta de implementación de estos componentes basada en una especificación formal: la Tecnología Java Beans Enterprise.

1.5. Tecnología Java Beans Enterprise

El desarrollo basado en componentes se encuentra en permanente evolución. En particular, dicha evolución se experimenta en dos mundos diferentes [21]. Por un

lado, pueden encontrarse estudios del desarrollo basado en componentes a nivel conceptual y/o metodológico, en los que, fijando la noción de componente, se busca definir la arquitectura lógica de ciertos tipos de sistemas y los pasos para probarla. Por otra parte, se encuentran las tecnologías que proporcionan plataformas para el desarrollo y ejecución de componentes (o sistemas basados en componentes), como la tecnología Enterprise Java Beans (EJB), propuesta por SUN Microsystems con la versión empresarial de Java 2 (J2EE, Java 2 Enterprise Edition). En este mundo la noción de componente se encuentra definida como una forma de implementación. De esta manera los EJBs son una propuesta tecnológica para el desarrollo de componentes de software reutilizables [21].

1.5.1. Gestión de EJBs en aplicaciones Web

Un EJB es un componente distribuido que se ejecuta dentro de un contenedor que proporciona un servidor de aplicaciones. El EJB es el componente distribuido básico de J2EE [12].

Un contenedor proporciona varios servicios básicos al componente EJB, incluyendo servicios de gestión de ciclo de vida, seguridad, transacciones, persistencia y nombres [12].

La gestión de ciclo de vida que proporciona el contenedor gestiona, esencialmente, la creación y potencialmente la compartición del objeto remoto EJB. El contenedor toma decisiones respecto a la creación de objetos EJB necesarios y gestiona la compartición del objeto entre clientes, si es que esto se permite [12].

La mayoría de los servidores de aplicaciones que incluyen contenedores EJB proporcionan pools de beans. En ellos se crean varias instancias del bean cuando arranca el servidor, las cuales se proporcionan a las sesiones de los clientes según se van solicitando. Cuando un bean ya no es necesario, puede devolverse

al pool. El movimiento de un bean del estado activo al estado pasivo se conoce como *pasivación,* mientras que el movimiento de un bean del estado pasivo al estado activo se conoce como *activación*. Las clases con las que se implementan los beans incluyen métodos *callback* para la pasivación y activación [12].

La gestión del ciclo de vida incluye también la gestión general de recursos de memoria. El contenedor puede destruir componentes para liberar los recursos que utilizan [12].

1.5.2. EJBs de entidad

Los EJBs de entidad modelan conceptos de negocios que representan “cosas”, como objetos del mundo real, e incluso cosas abstractas. Los beans de entidad describen tanto el estado como la conducta de estos objetos y permiten a los desarrolladores encapsular las reglas de datos y de negocio asociadas a un concepto específico. Esto hace posible manejar de forma consistente y segura los datos asociados a un concepto [22].

Los beans de entidad se corresponden con datos en una unidad de almacenamiento persistente. Las variables de instancia del bean representan los datos en las columnas de la base de datos. El contenedor debe sincronizar las variables de instancia del bean con la base de datos [22].

Los beans de entidad tienen dos tipos de persistencia: Persistencia Gestionada por el Bean (BMP, Bean-Managed Persistence) y Persistencia Gestionada por el Contenedor (CMP, Container-Managed Persistence). En el primer caso (BMP) el bean de entidad contiene el código que accede a la base de datos. En el segundo caso (CMP) la relación entre las columnas y la base de datos del bean se describen en un archivo de propiedades del bean y el contenedor de EJB se ocupa de la implementación [22].

CMP significa que el contenedor EJB maneja los accesos a la base de datos, es decir, el código del Bean no contiene ninguna llamada a la base de datos (sentencias SQL). Como resultado, el código del Bean no está atado a un tipo específico de almacenamiento persistente. A causa de esa flexibilidad, incluso si se vuelve a desplegar el Bean en un servidor J2EE distinto, que use una base de datos distinta, no hará falta recompilar el código del Bean, lo que los hace muy portables [22]. En la especificación 1.0 de CMP, el desarrollador debía definir en el EJB de entidad los atributos de la clase de manera explícita. Igualmente debía implementar cada uno de los métodos para acceder a esos atributos una vez desplegado el Bean [22]. En CMP 2.0 el desarrollador describe los atributos y las relaciones de un Bean de entidad usando campos de persistencia virtuales y campos de relación. Se llaman campos virtuales porque el desarrollador no declara estos campos explícitamente, sino que se definen métodos abstractos de acceso (get y set) en una clase abstracta del Bean de entidad. La implementación de estos métodos se genera en tiempo de despliegue por las herramientas del servidor de aplicaciones. A este tipo de implementación de Beans de entidad se le conoce como esquema abstracto de persistencia [22,23].

El esquema abstracto de persistencia es un conjunto de elementos XML que describen los campos de relación y los campos de persistencia. Estos elementos son utilizados por el contenedor EJB para hacer corresponder la entidad y sus relaciones con otros Beans en la base de datos [22,23].

1.5.3. EJBs de sesión

Los Beans de sesión representan sesiones interactivas con uno o más usuarios de la aplicación. Los Beans de sesión pueden mantener su estado, pero solo durante el tiempo que el usuario interactúa con el Bean.

A diferencia de los Bean de entidad, los Bean de sesión no se comparten entre

más de un usuario, sino que existe una correspondencia una a uno entre el Bean de sesión y el usuario [22, 24].

Los Bean de sesión pueden ser de dos tipos: Beans de sesión sin estado y Beans de sesión con estado [12,22].

- Un **Bean de sesión sin estado** no se modifica con la llamada de los usuarios. Los métodos que ponen a disposición de las aplicaciones son llamadas procedurales que reciben datos y devuelven resultados, pero que no modifican internamente el estado del Bean. Esta propiedad permite que el contenedor EJB pueda crear un almacén (pool) de instancias, todas ellas del mismo Bean de sesión sin estado y asignar cualquier instancia disponible a cualquier usuario. Es necesario utilizar un Bean de sesión sin estado cuando una tarea no está ligada a un usuario específico.

- Un **Bean de sesión con estado** posee variables de instancia que manejan datos específicos obtenidos durante la conexión con el usuario. Cada Bean de sesión con estado, por tanto, almacena el estado conversacional de un cliente que interactúa con el Bean. Este estado conversacional se modifica conforme el usuario va realizando llamadas a los métodos de negocio del Bean. El estado conversacional no se guarda cuando el usuario termina la sesión.

1.5.4. Despliegue de EJBs

Como los EJB operan dentro de un contenedor, es necesario informar al entorno del contenedor su existencia. Esto se hace a través de un proceso de despliegue. El despliegue no solo informa al contenedor de las clases que se utilizan para implementar los componentes remotos, sino que proporciona también instrucciones sobre cómo estarán disponibles los componentes, qué recursos utilizaran y qué propiedades exhibirán [12].

El despliegue de EJB requiere de un archivo JAR, el cual incluye un archivo de configuración conocido como *descriptor de despliegue*. Este es un archivo XML que contiene entradas que describen el proceso de despliegue de los componentes que contiene el archivo JAR. No se pueden desplegar los componentes EJB sin entradas correctas en el descriptor de despliegue [12].

La especificación de los EJB está pensada para proporcionar a los componentes distribuidos flexibilidad, portabilidad, interoperabilidad y funcionalidad de tipo *plug-and-play*. Para lograr esta meta, muchas de las propiedades de los componentes distribuidos no se gestionan dentro del programa, sino que se configuran durante el proceso de despliegue. Esto se conoce como *programación declarativa* y proporciona al proceso de desarrollo una nueva dimensión que al final puede proporcionar oportunidades adicionales [12].

La figura 1.4.1 muestra los dispositivos o componentes generados una vez desplegado un EJB.

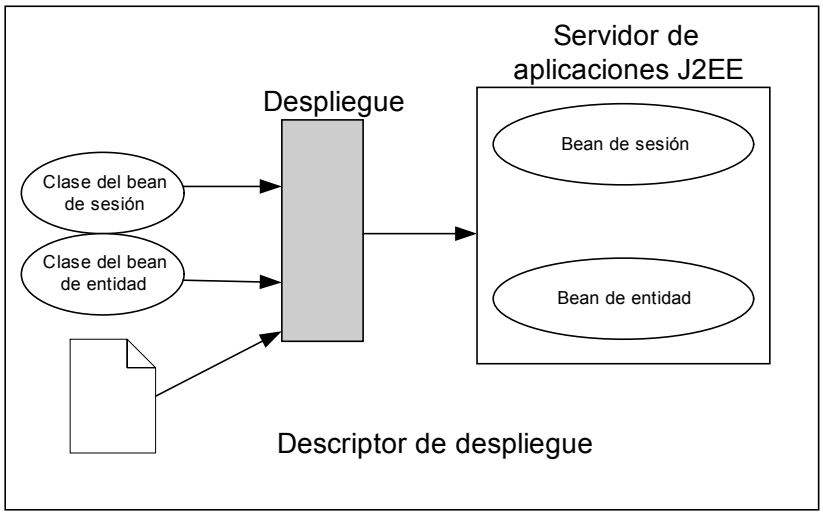

Figura 1.4.1: Despliegue de un EJB

1.5.5. Desarrollo de EJB

Para crear un EJB es necesario crear primero el bean y determinar los métodos que expondrá. Se deben crear después de un par de interfaces que definen cómo interactuar con el componente en forma remota. En total es necesario crear tres archivos para cada componente. La interfaz *home*, la interfaz *remota* y la implementación del bean (del componente) [12].

La **interfaz *home*** define los métodos que se pueden utilizar para crear una instancia del bean. El desarrollador escribe la interfaz *home* de forma que extienda la interfaz *EJBHome* del API J2EE. Esta interfaz debe declarar una o más firmas de versiones potencialmente sobrecargadas de los métodos que gestionan el ciclo de vida del bean. Uno de éstos métodos es el método *create*, el cual devuelve una referencia remota al componente.

La **interfaz remota** define los métodos que el EJB expondrá a sus clientes. La clase del EJB puede implementar varios métodos, pero no todos ellos tienen que ser accesibles al cliente. Limitar el número de métodos que expone el bean a un conjunto conciso de métodos que realizan las tareas requeridas por el cliente es una buena práctica, que cumple con la meta de limitar la información disponible, lo cual es un aspecto básico de seguridad. El desarrollador escribe la interfaz *remota* de forma que extienda la interfaz *EJBObject* del API J2EE.

La **implementación del bean** proporciona la implementación en sí del componente distribuido. Este es el código que se ejecutará dentro del contenedor de EJB y realizará los servicios que solicite el cliente. El código de la implementación debe implementar la interfaz correcta del paquete *javax.ejb* del API J2EE, que actualmente puede ser S*essionBean, EntityBean* y *MessageDrivenBean*; para beans de sesión, de entidad y de mensajería, respectivamente. Estas interfaces definen un conjunto de métodos callback. Estos métodos son invocados por el contenedor durante los distintos eventos del ciclo de

vida del bean.

De hecho el cliente nunca se comunica directamente con el bean. En vez de ello interactúa con él a través de la interfaz remota. Las firmas de los métodos que se identifican en la interfaz deben corresponder a los métodos que se declaran en este archivo.

Como se describió en la sección anterior, nuestra propuesta persigue construir la especificación de nuestro agente de software como un componente de software reutilizable. Buscamos implementar nuestro componente de acuerdo a una arquitectura definida, estándar y abierta como la propuesta por Sun Microsystems, descrita en la sección 1.5. Esta especificación, define dos formas o modelos de componentes que pueden ser utilizados para implementar nuestro agente como componente de software; implementación que depende de los requerimientos funcionales de nuestro componente y su alcance dentro de la aplicación practica.

Como describimos en la sección 1.3, nuestro agente debe estar caracterizado como agente interfaz basado en el conocimiento; el cual utiliza un modelo de usuario y un modelo de aplicación para extraer el conocimiento necesario para reconocer los planes del usuario y encontrar la oportunidad de contribuir con ello. Este tipo de interacción puede ser enfocada como sesiones interactivas que un usuario de la aplicación establece con un agente de software cada vez que sea requerido. Los datos que se transportan en estas sesiones sirven como información para que el agente coopere con el usuario en la resolución de sus tareas.

Basado en esto, es posible definir un agente interfaz, especificado como componente de software reutilizable e implementado como un EJB de sesión, descrito en la sección 1.5.3. Un EJB de sesión, permite mantener una sesión interactiva con uno o más usuarios de la aplicación a partir de una o más

instancias del componente proporcionadas por el servidor de aplicaciones, específicamente, por el contenedor de beans. Cuando un usuario solicita el servicio del bean, el contenedor selecciona una nueva instancia y abre la sesión con el usuario a través de las interfaces del bean.

Como se describió en la sección 1.5.3, un bean de sesión puede o no mantener un estado de la interacción con el usuario; esto depende del tipo de transacción que coexista entre el usuario y el bean, y de la relación que exista entre un tipo de servicio y otro. Cuando un servicio del bean depende de variables de instancia que son modificadas durante la sesión, es necesario utilizar un bean de sesión con estado. Si estado “conversacional” no utiliza variables de instancia que puedan ser modificadas durante la sesión, es más útil implementar un bean de sesión sin estado.

La especificación de nuestro agente y su dominio de aplicación nos permite definir a nuestro componente como un Enterprise Java Beans de Sesión, primeramente implementado como un EJB de sesión sin estado, que nos permita facilitar el uso de herramientas en una aplicación Web, a través de un agente interfaz implementado como componente de software reutilizable.

Para lograrlo, utilizaremos la implementación del agente descrita en la sección 1.3.6; la cual, y por estar escrita en Java, es totalmente transportable a un EJBs sin modificar la estructura de datos correspondiente a la programación y control del agente. Debemos re-programar esta nueva implementación como un envoltorio de la especificación lógica de ese agente y su interfaz con el ambiente, al mismo tiempo que definimos la estructura de datos para crear las interfaces del componente y su especificación como un EJB de sesión sin estado tal como se describe en la sección 1.5.5.

Una vez desplegado el componente, es posible transportar la implementación y re-desplegarlo con el manejo de estado. Esto, siempre que el dominio de aplicación,

así lo requiera.

Por último, es importante destacar que esta tecnología de EJBs forma parte de una arquitectura estándar y abierta llamada J2EE, la cual es un conjunto de especificaciones que proveen tecnologías para la construcción de aplicaciones empresariales basadas en capas. Cuando una aplicación Web implementa la tecnología de EJBs, agrega una capa denominada “capa de lógica del negocio” a la aplicación, lo que convierte a la aplicación en cuestión en una aplicación multicapa J2EE.

En la siguiente sección, se describe este conjunto de especificaciones y se ubican a los EJBs en función a ese número de capas y el papel que juegan en las aplicaciones Web J2EE.

1.6. Aplicaciones multicapas en J2EE

J2EE se define como un conjunto de especificaciones APIs Java que provee un conjunto de tecnologías para la construcción de aplicaciones empresariales. Una aplicación empresarial es un sistema informático basado en capas. Cada una de las capas identifica un conjunto de componentes software que tienen funcionalidades específicas una vez que la aplicación es instalada y desplegada en uno o varios servidores [12,22].

La tabla 1.6.1, muestra un resumen de las capas de una aplicación multicapa J2EE, las tecnologías asociadas a cada una de ellas y una breve descripción de sus funciones [12].

Capa	Tecnologías	Descripción
Cliente	Navegador Web, aplicación cliente.	Interactúa con el usuario y con la capa de presentación
Presentación	Servidor Web (Tomcat, BEA, Apache, IIS, SunOne)	El servidor Web. Crea la presentación de la aplicación e interactúa con la capa de negocios.
Negocio	Servidor de aplicaciones (WebLogic, Webshere, Jboss, SunOne)	Ejecuta la lógica de negocio de la aplicación. Interactúa con la capa de integración.
Integración	Servidor de aplicaciones	Responsable de interactuar con los distintos recursos de datos de la aplicación.
Recursos	Bases de datos relacionales, sistemas o bases de datos heredados (Oracle, PostgreSQL, SQL Server, mainframe), sistemas de mensajería empresarial (MQSeries).	Representa los datos – la información que utilizan las aplicaciones.

Tabla 1.6.1: Descripción de las capas de una aplicación multicapa J2EE

1.6.1. Arquitecturas

J2EE o Java 2 Enterprise Edition es una arquitectura estándar orientada al desarrollo y despliegue de aplicaciones Web usando el lenguaje de programación Java. Una aplicación multicapa J2EE consta de dos o más capas de componentes de software. La figura 1.6.1 muestra una arquitectura típica J2EE que incorpora un contenedor para cada grupo de componentes desplegados [22].

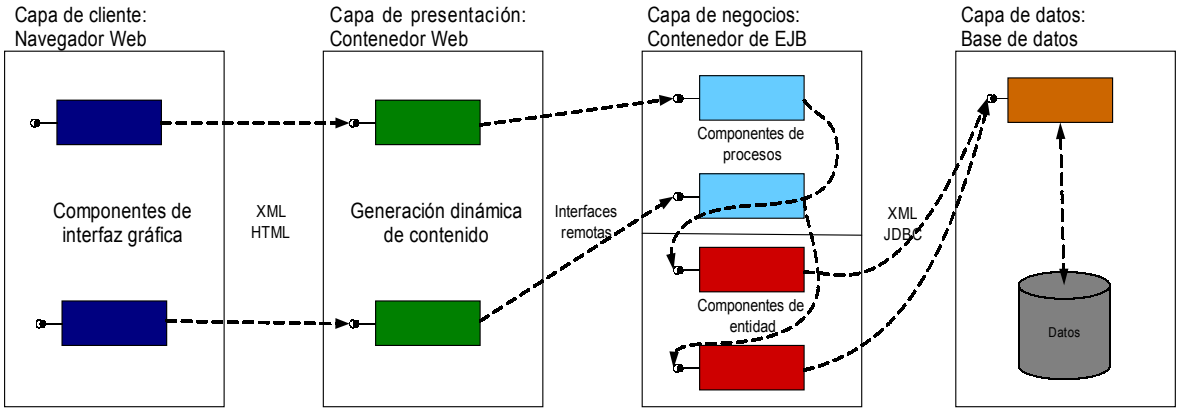

Figura 1.6.1: Modelo de aplicaciones multicapas

1.6.2. Capa de cliente

En la capa de cliente, típicamente existe un contenedor de applets Java embebido en el navegador Web. Esta capa despliega la interfaz de usuario haciendo uso de páginas HTML y páginas JSP para la generación de contenido Web [22,24].

1.6.3. Capa de presentación

La capa de presentación mantiene un contenedor Web que proporciona el entorno para el desarrollo, despliegue y manejo en tiempo de ejecución de servlets y páginas JSPs. Los servlets y páginas JSPs se agrupan en unidades desplegables llamadas aplicaciones Web. Estos componentes interactúan directamente con la capa de cliente y se encargan de la generación dinámica de contenido y de la interacción con la capa que controla la lógica de la aplicación [22,24].

1.6.4. Capa de lógica de negocios

La capa de lógica de negocios mantiene un contenedor EJB que proporciona el entorno de desarrollo, despliegue y manejo en tiempo de ejecución de EJBs. Los

EJBs son componentes que implementan los procesos y las entidades de negocio de la aplicación. Controla la lógica de la aplicación y la interacción con la capa de datos [22,24].

1.6.5. Capa de datos

La capa de datos se encarga de manejar la persistencia de datos de la aplicación. En J2EE y, específicamente, en la tecnología EJBs, la capa de datos corresponde al conjunto de EJBs que manejan las entidades de negocio de la aplicación. Estos EJBs guardan su estado en algún dispositivo secundario, típicamente una base de datos, que almacena el estado de las instancias de cada Bean en una tabla. Cuando se incorpora esta capa a la aplicación, el desarrollador debe decidir a qué componente delega el manejo de la persistencia de los datos. Para ello existen dos posibilidades: 1) delegar el manejo de la persistencia a cada EJB de entidad, para lo cual debe programarse, en el EJB, el código fuente necesario para crear y manipular la base de datos, o 2) delegar el manejo de la persistencia al contenedor de EJBs, lo cual evita la programación explícita en el código fuente del EJB para manipular la base datos, dejando que esta forme parte, de manera declarativa, del descriptor de despliegue de la aplicación [22,24].

Este modelo de aplicación, basado en capas, nos permite desarrollar cada uno de los componentes de cada capa independientemente de la aplicación practica; esto facilita la portabilidad y el reuso de los componentes entre aplicaciones. En este sentido, nuestro agente puede ser desarrollado como un componente de software reutilizable EJB independiente de la aplicación. Para cada aplicación se puede utilizar un marco de desarrollo como el propuesto en la sección 1.4, y programar al agente para un dominio de conocimiento específico.

Una vez que el agente posea la base de conocimientos y programada su interfaz

con el ambiente, es posible desplegar el componente como un EJB de sesión. Este despliegue, como se describe en las secciones 1.5.4 y 1.6, se hace en la capa de lógica de negocios de una aplicación Web multicapa. De esta manera nuestro componente se convierte en un componente distribuido que puede ser utilizado en aplicaciones Web cooperativas y distribuidas.

La siguiente sección muestra un modelo de aplicación multicapa basada en J2EE, que pudiera implementar nuestro modelo de agente con solo definir el dominio de conocimiento y la interfaz agente-ambiente dentro de la aplicación.

2. Bioinformantes como aplicación Web cooperativa

Introducción

Bioinformantes [3], fue desarrollado como una plataforma de computación científica sobre la Web que integra, en un ambiente virtual, funciones y servicios proporcionados por programas para el procesamiento de data científica, específicamente biológica. El objetivo fundamental de esta plataforma es el de proveer un ambiente cooperativo asistido por agentes de software que se comportan como tutores sobre un dominio particular de la biología [3].

En su estado actual, Bioinformantes incorpora un conjunto de clases APIs Java para proveer la aplicación Web cooperativa, incluyendo la interfaz gráfica de usuario, componentes para la generación dinámica de contenido Web y seguridad.

Igualmente, la especificación de Bioinformantes incorpora interfaces de usuarios y un sistema para el manejo de sesiones que pueden ser utilizadas para facilitar la implementación de un agente interfaz que sirva como tutor de la aplicación.

2.4. Arquitectura

En su diseño arquitectónico, Bioinformantes sigue las especificaciones básicas de aplicaciones multicapa basadas en J2EE. Como se muestra en la figura 2.1.1 existen dos capas implementadas funcionando como una aplicación cliente/servidor [3].

El sistema operativo sobre el cual se ejecuta el servidor de Bioinformantes está basado en UNIX, corriendo aplicaciones tales como Philyp. La máquina es también un servidor Web extendido que soporta aplicaciones Web desarrolladas

en Java (tecnologías como servlets y JSPs). Estas aplicaciones que funcionan en el servidor proporcionan al usuario una seudoterminal (una consola remota del sistema operativo) para que ejecuten sus programas desde la aplicación Web; además de un soporte para integrar al agente Bioinformante que sirva al usuario como asistente Web. Existe, por supuesto, una página Web que puede ser descargada desde cualquier navegador Web. Un applet, en esa página Web, es la interfaz del usuario para utilizar a Bioinformantes. Como primera aproximación los agentes Bioinformates son agentes muy simples. Son lanzados por el servidor de Bioinformantes simultáneamente con la seudoterminal para cada usuario. Un Bioinformante toma los mensaje que, explícitamente, van dirigidos a él, expresados en un lenguaje natural controlado, y los procesa según su propio conjunto de metas, reglas y creencias. Al mismo tiempo el usuario es libre de enviar mensajes, típicamente comandos Unix, pidiendo al sistema operativo su ejecución, cuya salida es enviada a la interfaz Web del usuario [3].

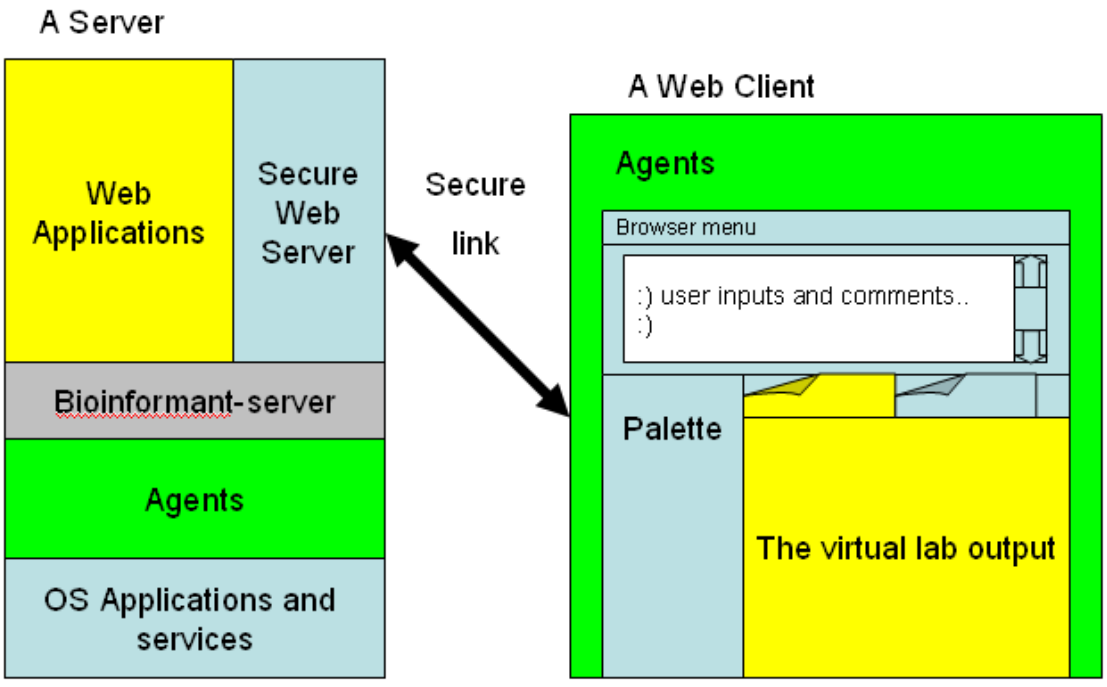

Figura 2.1.1: Arquitectura de Bioinformantes

2.5. Componentes de la aplicación

2.5.1. Seguridad y acceso

En Bioinformantes, los aspectos de seguridad y control de acceso son delegados al servidor de aplicaciones. El servidor mantiene una base de datos de los usuarios registrados que pueden hacer uso de los componentes de la aplicación. De igual manera, la transferencia de datos entre el servidor y el navegador Web del usuario se lleva a cabo por un puerto de conexión segura SSL (Securiry Socket Layer), el cual permite que los datos viajen cifrados por la red con autentificación y certificados de conexión.

Cuando el usuario accede a Bioinformantes, el servidor envía una página de acceso JSP que toma, en campos de texto, el nombre de usuario y contraseña de acceso. Estos campos son entregados al servidor de aplicaciones para la autentificación y validación del usuario. Si un usuario no está registrado en el sistema, o su nombre de usuario y contraseña no coinciden, entonces el servidor envía una página JSP indicando el error de acceso [3]. Este tipo de control de acceso se conoce como “autentificación basada en formulario” [22].

2.5.2. Interfaz gráfica de usuario

Una vez que el usuario es autenticado por el servidor y se acepta su acceso a la aplicación, el servidor envía una pagina HTML que contiene una llamada a un applet Java que sirve como interfaz entre el usuario y Bioinformantes. Desde este applet el usuario podrá realizar diversas tareas: establecer una sesión de chat con los demás usuarios de la aplicación, enviar mensajes solicitando la ejecución de comandos del sistema operativo y aplicaciones del lado del servidor e interactuar con el agente de software de su sesión [3].

2.5.3. Flujo de información

Cuando un usuario envía un mensaje lo hace escribiendo directamente en el applet. Este mensaje debe ser construido siguiendo ciertas estructuras, de tal manera que el applet pueda discriminar entre un mensaje para el chat y otro tipo de mensaje. En el primer caso el control de flujo del mensaje queda suscrito al applet. En el segundo, el applet envía el mensaje a los componentes del servidor para su procesamiento [3].

En la tabla 2.2.1, en la columna “forma del mensaje”, se muestra, respectivamente, un mensaje de chat dirigido a todos los usuarios que se encuentren en estado de sesión iniciada, un mensaje dirigido de manera personal a otro usuario con solo especificar su nombre de usuario, y un mensaje típico que solicita la ejecución de un comando en el sistema operativo. En este caso el applet envía el mensaje a un servlet, desplegado en el servidor, quién finalmente determina el flujo del mensaje.

De esta manera, los mensajes en la GUI de Bioinformantes son escritos en un lenguaje natural controlado, el cual, y obedeciendo a la aparición de ciertos términos o caracteres, es enviado a uno u otro componente de la aplicación para su procesamiento. La tabla 2.2.1 muestra la relación entre los mensajes escritos por el usuario, la aparición de términos y caracteres en esos mensajes y los componentes de la aplicación asociados a su flujo y procesamiento.

Tabla 2.2.1: Formato de mensajes en Bioinformantes.

Forma del mensaje	Destino del mensaje	Tipo del mensaje	Procesador del mensaje
*, Estoy procesando nueva data para analizar	Todos los usuarios con sesión activa en Bioinformantes	Sesión de chat	Applet
jdavila, Quisiera obtener tus resultados	Un usuario de Bioinformantes registrado como: jdavila	Sesión de chat	Applet
ls	Seudoterminal	Ejecución de comandos UNIX	Servlet desplegado en el servidor
BioTutor, quisiera aprender sobre philyp	Agente BioTutor	Solicitud de tutoría a bioinformante	Servlet desplegado en el servidor

2.5.4. Componentes desplegados en el servidor

Cuando el applet determina que el mensaje debe ser manejado por los componentes desplegados en el servidor, lo envía a un servlet. Este servlet discrimina entre mensajes dirigidos a la seudoterminal y mensajes dirigidos al agente. Si el mensaje va dirigido a la seudoterminal, se solicita la intervención de otro servlet encargado de ejecutar una clase middleware que enlaza la aplicación Java con la seudoterminal escrita en C. Si el mensaje va dirigido al agente, entonces se convierte en una lista de términos que puedan ser entendidos por el agente como un conjunto de átomos observables. En ambos casos, y luego del procesamiento del mensaje, el servlet envía la respuesta al applet para que finalmente sea desplegada al usuario [3].

En esta sección se describió la arquitectura de una aplicación J2EE llamada Bioinformantes. Esta plataforma posee características importantes para servir como una aplicación cooperativa, que integre agente inteligentes que sirvan como asistentes y consultores dentro de la aplicación. Las herramientas que incorpora no parecieran ser de fácil manejo por los usuarios, y los programas que utiliza en el procesamiento de datos son programas no distribuidos que corren en equipos UNIX. Los agentes pudieran funcionar en ese ambiente cooperativo y ayudar a los usuarios de la aplicación a realizar las tareas complejas y en el manejo de los programas destinados para tal.

En este sentido, es imperativo agregar a Bioinformantes una capa de lógica de negocios, tal como se describió en las secciones 1.5 y 1.6. Una vez desplegado Bioinformantes, como una aplicación multicapa, es posible agregar la implementación de nuestro agente de acuerdo al dominio de conocimiento específico de la aplicación.

Esta agregación e implementación de nuestro agente a Bioinformantes, podrá hacerse, progresivamente, de acuerdo a lo descrito en los capítulos 4 y 5. Allí se utiliza a nuestro agente en un modelo de aplicación multicapa, la manera de integrarlo a la capa de lógica de negocios de la aplicación y la implementación de ese agente en un dominio de conocimiento específico. Igualmente, en capítulos 4 y 5, se describe un marco de desarrollo, basado en reutilización de software, que facilita la definición de nuevas bases de conocimientos y nuevas interfaces agente-ambiente para que nuestro agente interfaz pueda ser incorporado en una aplicación particular y diferente.

El siguiente capítulo, describe un modelo de herramienta complejo que una aplicación puede integrar, y que su uso puede verse asistido por nuestro agente interfaz. Se trata de GALATEA [2], un lenguaje de simulación orientado al modelado de sistemas descompuestos como una red de nodos, y que pueden integrar agentes inteligentes para definir la dinámica de cambio en una simulación.

Este lenguaje fue construido en la Universidad de Los Andes y es utilizado con fines de aprendizaje en materias de pregrado y postgrado de la misma Universidad. GALATEA es un proyecto que, si bien es cierto que no está culminado aún, también es cierto que ofrece un conjunto de especificaciones APIs que pueden ser utilizadas para desarrollar, ejecutar y analizar modelos de simulación haciendo uso del compilador de Java y su máquina virtual.

Por último, es importante destacar que el API de GALATEA está siendo actualmente implementado en Bioinformantes como un lenguaje de simulación y una herramienta de la aplicación para sus usuarios. Esto facilita aún más la integración de nuestro agente interfaz a Bioinformantes.

3. Lenguaje de simulación Galatea

Introducción

Galatea [2] fue desarrollado como una plataforma de simulación, con varios lenguajes de programación asociados, que permite el modelado de sistemas en donde existen, destaca o influyen agentes inteligentes.

En Galatea, se plantea una extensión del lenguaje de programación Java para escribir los modelos de simulación. Como Java, Galatea se considera un lenguaje de simulación orientado a objetos que incorpora además tecnologías de agentes de software inteligentes.

Los modelos de simulación Galatea son escritos con una estructura orientada a red, donde los componentes del sistema modelado son organizados en redes de nodos que intercambian mensajes. Galatea fue pensado como un lenguaje de simulación de modelos continuos y por eventos discretos, que pueden incorporar agentes que influyen de manera autónoma en la simulación.

3.4. Modelos de simulación

Los modelos de simulación Galatea son escritos con una estructura orientada a red, donde los componentes del sistema modelado son organizados en redes de nodos que intercambian mensajes. La figura 3.1.1 muestra la estructura básica de un programa (modelo de simulación) escrito en el lenguaje de simulación Galatea [2].

- TITLE, especifica un título para el modelo a simular.

- NETWORK, Representa el conjunto de subsistemas (nodos) presentes en el sistema a simular y por lo tanto, tiene como función controlar la interacción y activación de los nodos. Lleva a cabo las siguientes actividades:

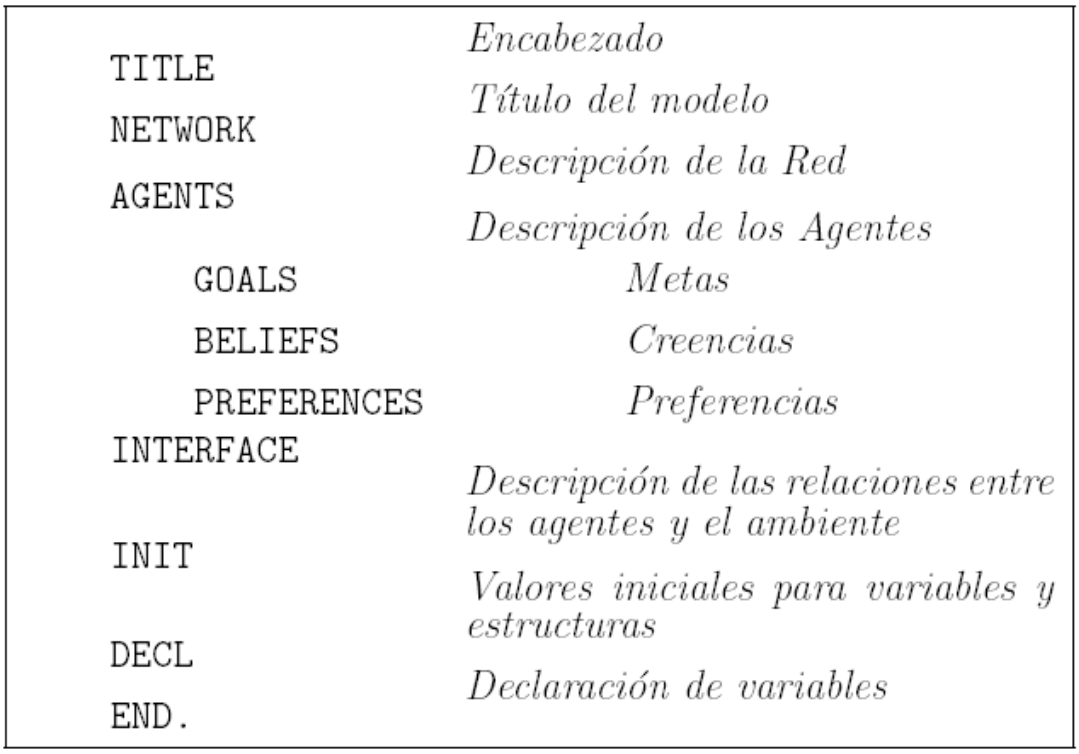

Figura 3.1.1: Estructura básica de un modelo escrito en Galatea

  - Controlar la activación del nodo actual.
  - Controlar la revisión de la red.
  - Controlar la activación de los agentes.

- INTERFACE, Sirve de intermediario entre los agentes y el simulador.
- AGENTS, Describe la interacción entre el grupo de agentes presentes en el

sistema. Este componente tiene la función de controlar las actividades propias del agente:

  - razonar. Implica asimilar percepciones, observaciones, recuerdos, metas, creencias y preferencias para tomar decisiones sobre su conducta, lo cual es realizado por el proceso simulado de cada agente.

  - observar. Percibir el ambiente,

  - actuar. Intentar modificar el ambiente de acuerdo a las decisiones tomadas.

- INIT, Mantiene los valores iniciales de las variables de estado del modelo y de las estructuras.

- DECL, sección que permite la declaración de las variables del modelo y las especificaciones para la generación de estadísticas.

La figura 3.1.2 muestra, como el ejemplo típico, de un modelo escrito en Galatea un sistema simple de tres taquillas de un banco con agentes [2].

En la sección NETWORK del modelo están representados el conjunto de nodos del sistema, en este caso, el sistema tiene un nodo de *entrada*, un nodo *taquilla* y un nodo *salida*. En la sección AGENTS se declaran dos tipos de agentes, uno correspondiente al cliente del banco quién debe revisar la cola de las taquillas y "quejarse" si percibe que estas colas sobrepasan un número especificado de "clientes". El segundo agente corresponde al gerente del banco, quién decide la apertura o cierre de una taquilla dependiendo de la longitud de cola. En la sección INTERFACE se implementan los métodos que son llamados por los agentes en la sección AGENTS. Estos métodos contienen programadas las acciones que el agente ejecuta una vez llamado el método; por ejemplo, la acción *queja* del agente

cliente llama al método *quejar()* que imprime un mensaje en la salida estándar del lenguaje que dice: “Cola larga” en el tiempo *time* de la simulación.

```
TITLE
   Sistema simple de tres taquillas
NETWORK
   Entrada (I) ::
      setAgent(Cliente);
      IT:= 10;
      SENDTO(Taquilla[MIN]);
   Taquilla [1..nCaj] (R) ::
      STAY:= 45;
   Salida (E) ::
AGENTS
   Cliente (AG) ::
      GOALS
         revisa_cola and
         if cola_larga then (queja)
   Gerente (AG) Static ::
      GOALS
         revisa_cola and
         if cola_larga then (crear_taquilla) and
         if taq_vacias then (elim_taquilla)
INTERFACE
   revisa_cola() {
      if ((where.getName() == "Taquilla") && (where.el.ll() > 5))
         input.add("cola_larga", time);}
   queja() {
      System.out.println(name + ": Cola larga!!"); }
   crear_taquilla() {
      if (Taquilla.mult > Taquilla.maxMult)
         System.out.println("Banco lleno!!");
      else Taquilla.addInstance(); }
   taq_vacias() {
      for (int i = 0, j = 0; i < Taquilla.mult; i++) {
         if (Taquilla[i].il.ll() + Taquilla[i].el.ll() = 0) {
            j++;
            if (j > 1) input.add("taq_vacias", time); }}}
   elim_taquilla() {
      int i = 0;
      while ((i < Taquilla.mult  - 1) &
              (Taquilla[i].il.ll() + Taquilla[i].el.ll() > 0)) {i++;}
      if (Taquilla[i].il.ll() + Taquilla[i].el.ll() = 0)
          Taquilla[i].delInstance(); }
 INIT
    TSIM:= 300;
    nCaj:= 3;
    ACT(Entrada, 0);
    INTI nCaj:3:0 :cantidad de cajeros;
 DECL
    VAR nCaj: INTEGER;
    STATISTICS ALLNODES;
```

Figura 3.1.2: Modelo de simulación escrito en Galatea

3.5. Implementación actual

Galatea es un proyecto en desarrollo que actualmente cuenta con un conjunto de clases escritas en Java que forman la Interfaz de Programación de Aplicación API de Galatea. Estas clases permiten la creación de modelos de simulación siguiendo las estructuras de programación de Java. La estructura básica de un programa Galatea, presentada en la sección 3.1, sería la forma definitiva de programar un modelo Galatea. Sin embargo, el compilador de Galatea se encuentra aún en desarrollo, lo que impone el uso del lenguaje Java y su compilador para la creación y compilación de este tipo de modelos.

Como se muestra en la figura 3.2.1 cada una de las clases creadas extienden de una clase base llamada *Node*. En esta clase se define y describe el comportamiento de un tipo de componente en el sistema simulado; para ello, existen diferente tipos de nodos los cuales deben ser especificados en el constructor de la subclase de *Node*. Todo nodo en un modelo debe ser creado, especificado, programado y agregado a la red de composición de nodos. Una vez que se definen las clases que describen los nodos de la red, se debe crear una clase principal que instancia, en un objeto Java, cada uno de estos nodos, guardando cierta relación de dependencia. Finalmente, se debe implementar el método *main*() de Java para inicializar variables, programar eventos iniciales, definir el tiempo de simulación, ordenar la ejecución del modelo, generar archivos de trazas y estadísticas de la simulación, entre otros parámetros [2].

La figura muestra un programa escrito en Java que importa el conjunto de clases Galatea del paquete galatea.glider. Las tres primeras clases implementadas en este código se refieren a la red de nodos del modelo a simular, equivalente a la sección NETWORK de la estructura básica de un programa en Galatea. La clase llamada *Taquilla3*, es la clase principal del modelo que instancia cada una de las clases referidas a un nodo, guarda la estructura de la red de composición de nodos y la dependencia entre cada uno de ellos. Además, ésta clase contiene el

método *main()* que sirve programar los eventos, establecer la inicialización de los parámetros de la simulación, generar archivos de trazas y estadísticas, entre otros [2].

```
import galatea.glider.*;
class Entrada extends Node{
  Entrada(Node s[]){
    super("Entrada",'I',s);
    Glider.nodesl.add(this); }
  public void sendto(Message m){
    sendto(m,Taquilla3.taquilla,Glider.MIN); }
  public boolean fact(){
    it(10);
    return true; } }
class Taquilla extends Node{
  private static int mult = 1;
  Taquilla(Node s){
    super("Taquilla",mult,'R',s);
    Glider.nodesl.add(this);
    mult++;}
  public boolean fscan(){
    stay(45);
    return true; } }
class Salida extends Node{
  Salida(){
    super("Salida",'E');
    Glider.nodesl.add(this); } }

public class Taquilla3 {
  // construccion de la red
  public static Salida salida = new Salida();
  public static Taquilla taquilla[] = { new Taquilla(salida),
                                        new Taquilla(salida),
                                        new Taquilla(salida) };
  public static Entrada entrada = new Entrada(taquilla);
  // Simulador
  public static void  main(String args[]) {
            // Inicia las variables globales
    Glider.setTitle("Sistema simple de tres taquillas");
    Glider.setTsim(300);
            // Traza de la simulacion en archivo
    Glider.trace("Taquilla3.trc");
            // programa el primer evento
    Glider.act(entrada,0);
            // Procesamiento de la red
    Glider.process();
            // Estadisticas de los nodos en archivo
    Glider.stat("Taquilla3.sta"); } }
```

Figura 3.2.1: Modelo de simulación escrito en Java con API de Galatea

Nótese que el programa mostrado en la figura 3.2.1 no es totalmente equivalente al modelo presentado en la figura 3.1.2. En este caso, el modelo no incluye al conjunto de agentes en el proceso de la simulación.

3.5.1. Compilación y ejecución de modelos

En la actualidad, todo modelo de simulación escrito en Galatea es compilado con el propio compilador de Java y ejecutado con la Máquina Virtual de Java, lo que define una relación de dependencia entre el API de Galatea y el Java 2 Standard Edition.

Para compilar un modelo Galatea, es necesario llamar al compilador de Java, el cual genera un archivo en código de byte que solo puede ser interpretado por la JVM.

De esta manera GALATEA se convierte en una herramienta de simulación que, a pesar de no estar completamente desarrollada, proporciona un API que permite la programación, ejecución y análisis de modelos de simulación.

Su aplicabilidad practica hecha a un lado su complejidad de uso y, es actualmente utilizado, para desarrollar modelos de simulación escritos con estructuras de datos basadas en Java, tal como se muestra en la sección 3.2.

Por otro lado, GALATEA esta siendo implementado en Bioinformantes como el lenguaje de simulación de la aplicación, lo que agrega a esta plataforma nuevas herramientas, además de las descritas en el capítulo 2, que pueden ser utilizadas por los usuarios de la aplicación por medio de una interfaz Web.

En este sentido, nuestra propuesta se basa en dotar, a nuestro agente interfaz, de un conocimiento básico, que ayude a los usuarios de la aplicación Web a

desarrollar, compilar, ejecutar y analizar sus modelos de simulación haciendo uso de GALATEA. Esto nos proporciona un dominio de aplicación practico de nuestro agente, al mismo tiempo que nos permite establecer los parámetros de integración del agente en una aplicación Web. Igualmente, podemos establecer una medida de rendimiento de nuestro agente haciendo uso de la propuesta de Russell, descrita en el capítulo 1. Finalmente, al dotar a nuestro agente interfaz de un conocimiento específico, podemos establecer los parámetros que permitirán construir un marco de desarrollo para proporcionarle al agente nuevo conocimiento, o para implementar una nueva base de conocimientos en función a un nuevo dominio de aplicación.

En el siguiente capítulo se especifica a nuestro agente interfaz como un componente de software reutilizable EJB, se define un marco de desarrollo basado en técnicas de reutilización para implementar la base de conocimientos del agente, y se mostrará cómo dotar de conocimiento al agente a partir de un dominio de aplicación, el cual, en nuestro caso, será aquel orientado al modelado y simulación de sistemas escritos en GALATEA. A este agente lo llamamos SIMULANTE.

4. Especificación de un agente Simulante

Introducción

Un Simulante se define como un agente de software con capacidades específicas para servir como asistente y consultor de una aplicación Web dirigida al modelado y simulación de sistemas. Simulante ayudará a simulistas a construir, compilar, ejecutar y procesar sus modelos de simulación y analizar los resultados desde una aplicación Web.

Simulante está basado en un modelo lógico de agente que implementa el motor de inferencia GLORIA descrito en la sección 1.2. La interfaz agente-ambiente está programada en Java y se conecta con un "envoltorio" de comunicación, también programado en Java, que ejecuta el ciclo de *observar, ejecutar* y *razonar* (descrito en la sección 1.2.6) y que permite llamar al motor de inferencia programado en Prolog [18]. La base de conocimientos utiliza la estructura presentada en la sección 1.2.5 y está programada para caracterizar a un agente reactivo-racional que es capaz de realizar labores de asistente para la compilación y ejecución de modelos de simulación escritos en Galatea. Igualmente, un Simulante es capaz de servir como consultor de la aplicación en el dominio de conocimiento orientado al desarrollo de modelos en Galatea.

Todas las acciones (salidas) que el agente puede causar sobre el ambiente son obtenidas del motor de inferencia de acuerdo al conjunto de observaciones (entradas) que el usuario proporcione en un momento dado. Una vez inferida la acción, el "envoltorio" Java la recibe desde el motor de inferencia Prolog y luego se comunica con la interfaz agente-ambiente para ejecutar la acción. De esta manera, el "envoltorio" Java con la interfaz agente-ambiente, se convierten en la

estructura de datos que caracteriza al agente, permitiéndole obtener, desde el ambiente, las observaciones necesarias, llamar al motor de inferencia, obtener el conjunto de influencias y ejecutar las acciones correspondientes.

Esta estructura de datos, totalmente programada en Java, puede ser adaptada e implementada como un componente de software reutilizable EJB (descritos en la sección 1.5). El objetivo principal de esto es que nuestro agente Simulante pueda funcionar como un agente interfaz basado en conocimiento (descrito en la sección 1.3) dentro de una aplicación Web J2EE cooperativa, distribuida y multiusuario. De esta manera, cada usuario de la aplicación puede utilizar una instancia del componente (que es un agente Simulante) y emplearlo como su asistente en la aplicación. Esta instancia es obtenida desde el pool de instancias del contenedor EJB una vez que el usuario "codifique", desde la GUI, las observaciones necesarias para el agente.

Simulante, como componente de software reutilizable EJB, es especificado como un EJB de sesión sin estado (descrito en la sección 1.5.3). Así, el componente está atento a las llamadas del usuario y sirve como agente de interfaz mientras recibe datos y devuelve resultados.

Otra característica importante de nuestro agente como componente de software reutilizable, es que puede ser adaptado a cualquier tipo de aplicación Web distribuida que amerite el uso de un agente interfaz. Esto puede lograrse con solo cambiar la base de conocimiento del agente y las acciones programadas en la interfaz agente-ambiente. Simulante es un agente orientado al dominio de aplicación del modelado y simulación en Galatea que sirve como piloto para pruebas de implementación y funcionalidad en la aplicación. Sin embargo, un experto en un dominio de aplicación diferente puede reprogramar al agente y aplicarlo para propósitos diferentes.

4.1. Dominio de conocimiento y requerimientos funcionales

Simulantes es un agente especificado y programado para funcionar como agente interfaz basado en el conocimiento sobre una aplicación Web J2EE cooperativa, distribuida y multiusuario. El dominio de conocimiento de nuestro agente está relacionado con la simulación de modelos escritos en Galatea. Un Simulante tendrá la capacidad de servir como agente interfaz para asistir a usuarios en el aprendizaje y uso de Galatea como lenguaje de simulación.

4.1.1. El agente cumpliendo un rol de asistente

Este agente interfaz, cumpliendo un rol de asistente, podrá interactuar con el sistema operativo que mantiene la aplicación Web en el lado del servidor y solicitarle servicios de compilación y ejecución de modelos Galatea haciendo uso del compilador y la máquina virtual de Java [19]. El agente debe ser programado para prestar servicios de búsqueda de archivos (que son modelos Galatea) en un directorio de trabajo y determinar la existencia de ellos. Los archivos que el agente puede explorar en el directorio de trabajo son los siguientes:

- Archivos con extensión *.java*, que refieren al código fuente del modelo.
- Archivos con extensión *.class*, que se refieren al código intermedio (bytecode) que genera el compilador de Java. Este es el archivo ejecutable por la máquina virtual de Java.
- Archivos con extensión *.trz*, que refieren al archivo de trazas generado durante la corrida de un modelo Galatea.
- Archivos con extensión *.sta*, que refieren al archivo de estadísticas generado durante la corrida de un modelo Galatea.

Cuando un usuario de la aplicación le pide al agente que compile o ejecute un

modelo de simulación, el agente debe realizar acciones de búsqueda de ese modelo en el directorio de trabajo. En el caso de la compilación, el agente se encargará de buscar el archivo especificado con extensión .java, comunicarse con el sistema operativo para llamar y ejecutar el compilador de Java quién finalmente se encargará de generar el archivo compilado del modelo. En el caso de la ejecución, el agente se encargará de buscar el archivo especificado con extensión .class, comunicarse con el sistema operativo para llamar y ejecutar la máquina virtual de Java quién finalmente se encargará de ejecutar el modelo. De igual manera, si el usuario solicita la presentación o visualización de los archivos de traza o estadísticas, el agente buscará el archivo especificado con extensión .trz o .sta, respectivamente y ejecutará las acciones necesarias para que la aplicación visualice la información. Si el agente no puede determinar la existencia de un archivo, u ocurre algún error de compilación o ejecución, éste ejecuta las acciones necesarias para dar por enterado al usuario, ofreciéndole una solución parcial o total al problema.

4.1.2. El agente cumpliendo el rol de consultor

De igual manera, este agente interfaz puede prestar servicios basados en conocimiento para servir como consultor del usuario en la aplicación. En este caso, el agente está programado para proponer material didáctico e informativo sobre el uso de Galatea como lenguaje de simulación. Debe ofrecer un conocimiento básico que ayude a la construcción de modelos Galatea, siendo capaz de explicar las sentencias de programación, estructuras de datos y métodos que deben existir en un modelo escrito en este lenguaje. Para lograr esto, el agente debe contar con una “biblioteca documental”, que pueden ser: archivos locales al servidor, frases programadas, referencias a direcciones Web, textos digitales, archivos con modelos programados para ejemplificar, archivos de modelos ya compilados y ejecutados, etc.

Cuando un usuario de la aplicación solicita este tipo de servicios, el agente debe determinar el tipo de documentación a presentar. Este discernimiento está estrechamente relacionado con la frase que construya el usuario para enfocar su solicitud. Si el agente no puede relacionar con precisión las observaciones con las posibles acciones, entonces ejecuta las acciones necesarias para dar por enterado al usuario, ofreciéndole una solución al problema. Esto último dependerá del módulo de procesamiento de lenguaje utilizado en la aplicación.

Es importante destacar que el agente puede determinar, como consultor, que el usuario revise documentación o modelos de simulación que son archivos locales en el servidor. En este caso el agente, de nuevo, debe emprender acciones de búsqueda en un directorio de trabajo y ejecutar las acciones necesarias para que la aplicación presente al usuario la información.

4.2. Base de conocimientos

La base de conocimiento de nuestro agente interfaz sigue las especificaciones descritas en la sección 1.2.5 y es programada de acuerdo a los roles especificados en la sección 4.1.

La figura 4.2.1 muestra la base de conocimiento, escrita en lógica, para el agente cuando percibe su ambiente cumpliendo un rol de asistente. La figura 4.2.2 muestra la base de conocimientos del agente cumpliendo el rol de consultor.

Como se observa en las figuras 4.2.1 y 4.2.2, el conjunto de átomos observables (especificados en el predicado *observable*) para el agente, se refiere al conjunto de observaciones que el agente puede percibir de su ambiente.

Al conjunto de observaciones que son obtenidas directamente del mensaje enviado por el usuario las llamamos **efecto de orden** para el agente. Son ejemplos de este tipo de observaciones los átomos compilar, ejecutar,

traza_interpretado y estadisticas_interpretado de la figura 4.2.1; y los átomos aprender_lenguaje, doc_basica, estructura_modelo, aprender_programacion, ver_api, ver_metodos, ejemplo_codigo y ejemplo_ejecucion de la figura 4.2.2.

Al conjunto de observaciones que el agente obtiene (o deja de obtener) una vez que intenta ejecutar el plan (es) para cumplir con un efecto de orden las llamamos **efecto de control** para el agente, y son observaciones que no están directamente relacionadas con el mensaje del usuario, pero que deben ser tomadas en cuenta por el agente para cumplir con el plan formulado. Son ejemplos de este tipo de observaciones los átomos existe_archivo_java, existe_archivo_class, existe_archivo_trz y existe_archivo_sta. Por ejemplo, si el usuario solicita la compilación de un archivo (efecto de orden), el agente debe asegurarse que tal archivo exista en el directorio explorable. Si este archivo existe, la observación *existe_archivo_java* (efecto de control) es colocada por el motor de inferencia en el conjunto actual de observaciones, permitiendo que el agente ejecute las acciones de compilación y dé respuesta al usuario.

```
%Conjunto de acciones que el agente puede ejercer sobre su ambiente en el rol de ASISTENTE
executable(buscar_archivo_java).
executable(buscar_archivo_class).
executable(buscar_archivo_trz).
executable(buscar_archivo_sta).
executable(agente_compilar).
executable(agente_ejecutar).
executable(agente_leer_traza).
executable(agente_leer_estadisticas).
executable(agente_leer_fecha).
executable(agente_mostrar_traza).
executable(agente_mostrar_estadisticas).
executable(no_existe_archivo_java).

%Efectos de control del agente cumpliendo el rol de asistente
observable(existe_archivo_java).
observable(existe_archivo_class).
observable(existe_archivo_trz).
observable(existe_archivo_sta).

%Efectos de orden del agente cumpliendo el rol de asistente
observable(compilar).
observable(ejecutar).
observable(traza_interpretado).
observable(estadisticas_interpretado).


% Reglas de Integridad para el agente cumpliendo el rol de Asistente
si compilar entonces buscar_archivo_java.
si existe_archivo_java, compilar entonces agente_compilar.
%si not(existe_archivo_java) entonces no_existe_archivo_java.
si ejecutar entonces buscar_archivo_class.
si existe_archivo_class entonces completar_proceso_ejecutar.
si compilar, ejecutar entonces buscar_archivo_java.
si existe_archivo_java, compilar, ejecutar entonces completar_proceso_compilar.
si compilar, ejecutar, not(existe_archivo_java) entonces buscar_archivo_class.
si existe_archivo_class, compilar, ejecutar, not(existe_archivo_java) entonces forzar_proceso.
si traza_interpretado entonces buscar_archivo_trz.
si existe_archivo_trz entonces agente_mostrar_traza.
si estadisticas_interpretado entonces buscar_archivo_sta.
si existe_archivo_sta entonces agente_mostrar_estadisticas.


%Planes concretos del agente en el rol de Asistente
para completar_proceso_ejecutar haga
    agente_ejecutar,
    agente_leer_traza,
    agente_leer_estadisticas.
```

Figura 4.2.1: Base de conocimiento para Simulante en su rol de asistente.

Para el agente en el rol de asistente, todo efecto de orden va acompañado de un efecto de control. El efecto de orden siempre es verdadero en el conjunto de observaciones actuales del agente. El agente se encarga de asignar el valor de verdad para el efecto de control una vez que culmina una acción de búsqueda.

```
%Conjunto de acciones que el agente puede ejercer sobre su ambiente en el rol de CONSULTOR
executable(preguntar_nivel1).
executable(preguntar_nivel2).
executable(preguntar_nivel3).
executable(buscar_archivo_doc).
executable(seleccionar_info).
executable(mostrar_ayuda).
executable(recopilar_ref).
executable(leer_estructura).
executable(presentar_estructura).
executable(referenciar_api).
executable(leer_metodos).
executable(presentar_metodos).
executable(seleccionar_archivo_java).
executable(mostrar_archivo_java).

%Efectos de orden del agente cumpliendo el rol de consultor
observable(aprender_lenguaje).
observable(doc_basica).
observable(estructura_modelo).
observable(aprender_programacion).
observable(ver_api).
observable(ver_metodos).
observable(ejemplo_codigo).
observable(ejemplo_ejecucion).

% Reglas de Integridad para el agente cumpliendo el rol de Consultor
si aprender_lenguaje entonces preguntar_nivel1.
si doc_basica entonces mostrar_doc.
si estructura_modelo entonces mostrar_estructura.
si aprender_programacion entonces preguntar_nivel2.
si ver_api entonces referenciar_api.
si ver_metodos entonces explicar_metodos.
si ver_ejemplos entonces preguntar_nivel3.
si ejemplo_codigo entonces mostrar_codigo.
si ejemplo_ejecucion entonces mostrar_ejecutado.

%Planes concretos del agente en el rol de Consultor
para mostrar_doc haga
    buscar_archivo_doc,
    seleccionar_info,
    recopilar_ref,
    mostrar_ayuda.

para mostrar_estructura haga
    leer_estructura,
    presentar_estructura.

para explicar_metodos haga
    leer_metodos,
    presentar_metodos.

para mostrar_codigo haga
    seleccionar_archivo_java,
    buscar_archivo_java,
    mostrar_archivo_java.

para mostrar_ejecutado haga
    seleccionar_archivo_java,
    buscar_archivo_java,
    agente_compilar,
    agente_ejecutar,
    agente_leer_traza,
    agente_leer_estadisticas.
```

Figura 4.2.2: Base de conocimiento para Simulante en su rol de Consultor.

El conjunto de influencias buscar_archivo_java, buscar_archivo_class, buscar_archivo_trz y buscar_archivo_sta son acciones que el agente ejerce una vez obtenido algún efecto de orden. El resto de las influencias son acciones que el agente ejecuta una vez obtenido algún efecto de control y ese efecto de control es verdadero (es decir, se convierte en una observación).

Todas las acciones que el agente emprende son calculadas por el motor de inferencia del agente de acuerdo al conjunto de reglas especificadas en la base de conocimiento (figura 4.2.1). Estas reglas pueden concluir en una **acción simple** o una **acción compuesta**. Una acción simple se refiere a un consecuente de la regla que no referencia a un plan de acción en la base de conocimiento, es decir, que el agente ejecuta una única acción en el momento T. Una acción compuesta se refiere a un consecuente de la regla que referencia a un plan de acción, en este caso, el agente debe ejecutar dos o más acciones en el momento T. Por ejemplo, cuando el conjunto de observaciones conjugados son *existe_archivo_java, compilar* y *ejecutar*, el agente debe emprender una acción compuesta que referencia al plan de acción *completar_proceso_compilar*, el cual conjuga un conjunto de acciones que el agente debe ejecutar secuencialmente hasta cumplir el plan.

4.3. Interfaz agente-ambiente

La interfaz agente-ambiente, se refiere al medio que utiliza el agente para percibir y actuar de manera concreta. Esta interfaz es capaz de conjugar las observaciones del agente y entregarlas al motor de inferencia para su procesamiento. Una vez que el motor de inferencia determina, en función de esas observaciones, el conjunto de acciones ejecutables para el agente, la interfaz agente-ambiente ejecuta cada una de esas acciones ofreciendo una dinámica de cambio en el ambiente (en la aplicación) causada por el agente.

Esta interfaz entre el agente y su ambiente está programada en el lenguaje Java y sus métodos contienen la implementación concreta de las acciones del agente. Cada acción que el agente deba ejercer debe estar programada en esta interfaz [4].

La figura 4.3.1 muestra la implementación de uno de los métodos que especifica una acción concreta del agente.

```
/*Método al que llama exec para ejecutar la acción del agente de compilar un archivo .java en el directori
  actual, consecuente de una regla en el cerebro*/
public void agente_compilar(){
    String salidaM = "";
    Runtime now = Runtime.getRuntime();
    try {
        Process pp = now.exec("javac " + galatea + ".java");
        InputStream s = pp.getInputStream();
        BufferedReader in = new BufferedReader(new InputStreamReader(s));
        salidaM = in.readLine();
        if (salidaM == null){
            acciones [i] = "Simulante: He terminado de compilar el archivo " + galatea + ".java";
            i++;
        }
        else{
            acciones [i] = "Simulante: No pude compilar el archivo " + galatea + ".java";
            acciones [i] = acciones [i] + " es posible que no exista en el directorio de trabajo o que
                           tenga un error de compilación";
            i++;
        }
    }catch(Exception e){
        salida = "Simulante: Disculpe, pero ocurrió un problema mientras intentaba compilar el archivo " +
                 galatea + ".java,";
        salida = salida + " la excepción provino de la JVM denominada: " + e.toString();
    }

}
```

Figura 4.3.1: Implementación de una acción concreta del agente

Para recolectar el conjunto de observaciones del agente y entregarlas al motor de inferencia, la interfaz agente-ambiente implementa un método que almacena, en

una estructura de datos, los efectos de orden y de control obtenidos del ambiente en un momento determinado [4].

La figura 4.3.2 muestra el código Java que implementa el método de recolección de observaciones para el agente en su rol de asistente.

Por otro lado, es importante destacar, que el motor de inferencia, la base de conocimiento y la interfaz agente-ambiente son programas independientes escritos en lenguajes de programación diferentes. Sin embargo, el motor de inferencia está programado para entender literalmente el lenguaje utilizado en la base de conocimientos y, a partir de ello, inferir las acciones del agente. Como la interfaz agente-ambiente fue programada en el lenguaje de programación Java, se necesita un componente adicional que envuelva al motor de inferencia del agente (programado en Prolog). Este componente está programado en Java y sirve para interactuar con el motor de inferencia [4]. Sus funciones básicas son:

- Recibir observaciones de la interfaz agente-ambiente,
- Entregar tales observaciones al motor de inferencia,
- Recibir el conjunto de acciones calculadas por el motor de inferencia y
- Entregar tales acciones a la interfaz agente-ambiente para ser ejecutadas.

Este envoltorio del motor de inferencia fue programado por Dávila en la implementación del agente descrita en la sección 1.2.6 y mostrada en la figura 1.2.4. Como se comentó, esta estructura de datos, programada en Java, es una implementación de GLORIA que permite la ejecución del ciclo *observar, ejecutar* y *razonar* para el agente, al mismo tiempo que sirve como medio de comunicación entre: el motor de inferencia, la base de conocimiento y la interfaz agente-ambiente.

```
/** This are the "objective" fluents */

/** This methods generates the actual observation as they
 *  are processed by the demo program, i.e. una lista de
 *  atomos en Prolog. It is worth noting that all the observations
 *  fluents are taking into consideration when it comes to
 *  process inputs to to the agent.
 */
public String processRawInputs(String o) {
    String r = "";
    r = r + o;
    if (buscar_archivo_java(nombre_archivo)) {
        r = r + ",existe_archivo_java" ;
    }
    if (buscar_archivo_class(nombre_archivo)) {
        r = r + ",existe_archivo_class" ;
    }
    if (buscar_archivo_class(nombre_archivo)) {
        r = r + ",existe_archivo_trz" ;
    }
    if (buscar_archivo_class(nombre_archivo)) {
        r = r + ",existe_archivo_sta" ;
    }
    r = r + ",nulo";
    return r;
}
```

Figura 4.3.2: Método que implementa la agrupación de observaciones para el agente

La figura 4.3.3 muestra el método agregado a la implementación del agente que sirve como envoltorio para el motor de inferencia programado en Prolog [4].

```
/** Este método hace el llamado al sistema operativo para ejecutar el brain
 * o motor de inferencia del agente. Es decir el wrapper.
public void think(){
    synchronized (this) {
        // it gets a virtual machine
        Runtime now = Runtime.getRuntime();
        try {
            //  it launches a process
            String [] env = new String[1];
            env[0]="PATH=$PATH;"+demoHome;
            Process pp = now.exec(demoHome + "simulante" + " -- " + observations + " " + goalsIn, env);
            InputStream s = pp.getInputStream();
            BufferedReader in = new BufferedReader(new InputStreamReader(s));
            try {
                // First line is the set of pending goals
                goalsIn = in.readLine();
                // First line is the set of pending goals
                goalsIn = in.readLine();
                // the rest are influences..
                String b = " " ;
                while((b = in.readLine())!= null) {
                    System.out.println("Agente "+brain+": attempting "+b);
                    influences.add(b);
                }
            } catch (Exception e) {
                System.out.println("Agente "+brain+": I can't read the influences, " + e); }
         } catch (Exception e) {
            System.out.println("Agente "+brain+": I couldn't think " + e);
     | }

    }
}
```

Figura 4.3.3: Implementación del envoltorio Java-Prolog para el motor de inferencia del agente

4.4. Implementación del agente como un EJB

4.4.1. EJB de sesión sin estado

Como se explicó en secciones anteriores, Simulante es un modelo de agente que se implementa en Java de acuerdo a las especificaciones de GLORIA. Esta implementación del agente tiene las siguientes propiedades [2,4]:

- Extiende de una clase base llama *Agen* del API de Bioinformantes, que contiene, básicamente, las estructuras de datos para almacenar las observaciones (entradas) e influencias (salidas) para el agente.

- Posee un método recursivo llamado *cycle()* que ejecuta el ciclo de razonamiento del agente: El agente observa, razona y ejecuta. Cada proceso de ese ciclo es implementado en los métodos: *observe(), reason()* y *execute()*, respectivamente.

- Contiene la implantación de un método *Think()* que sirve como envoltorio para el motor de inferencia (programado en Prolog) y su base de conocimientos. Este método fue explicado en la sección 4.3.

- Incorpora la interfaz agente-ambiente que contiene implementados los métodos de las acciones concretas del agente, también explicada en la sección 4.3.

Esta estructura de datos, en su conjunto, forma un archivo codificado completamente en el lenguaje Java que depende de dos archivos adicionales:

- el archivo compilado en Prolog que contiene el motor de inferencia del agente y su base de conocimiento y

- la clase base *Agent* que define las estructuras de almacenamiento para las

observaciones y acciones del agente.

La figura 4.4.1 muestra la relación que existe entre estos tres archivos que componen al agente especificado.

Como se muestra en la figura 4.4.2, la clase especializada contiene todos los métodos necesarios para que el agente pueda recibir y almacenar el conjunto de átomos observables de su ambiente, enviar ese conjunto de átomos al motor de inferencia, recibir de ese motor de inferencia el conjunto de acciones a ejecutar y ejecutar de manera concreta cada acción.

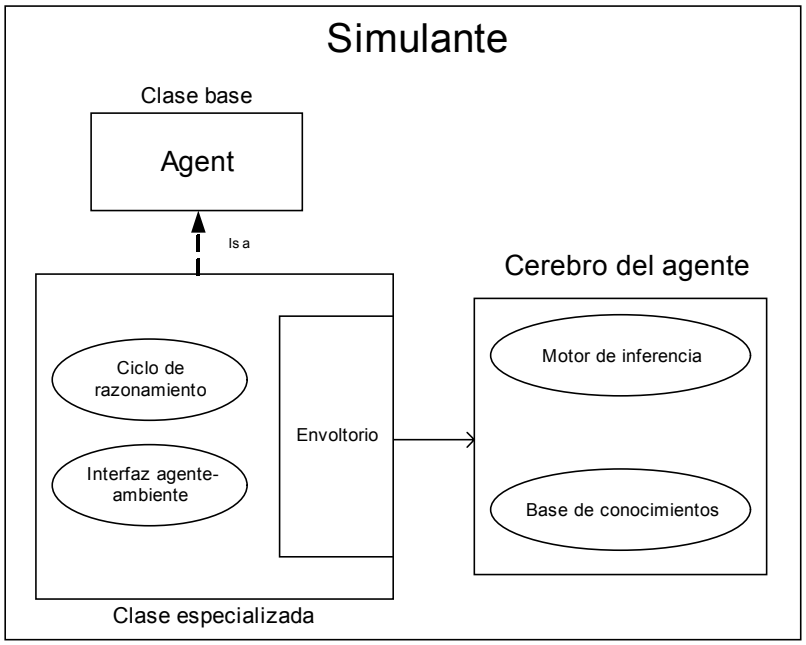

Figura 4.4.1: Implementación del agente en Java

Esta clase especializada, mostrada en la figura 4.4.2, puede ser especificada como un componente de software reutilizable. La implementación de la clase, mostrada en la figura 1.2.4, está programa en el lenguaje Java y podrá ser especificada como un EJB.

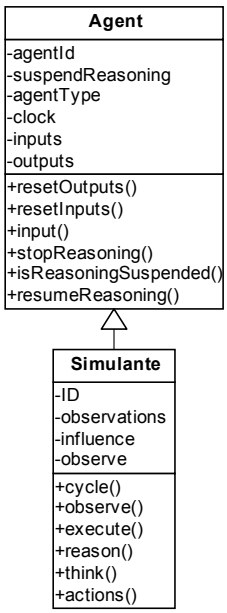

Figura 4.4.2: Especificación de la clase especializada para el agente

Nuestro agente será implementado en una aplicación Web como un EJB de sesión sin estado. Los usuarios de la aplicación pueden obtener una referencia remota del agente e invocar los métodos necesarios para entregarle observaciones y recibir respuestas de acciones concretas. Como el agente es un EJB de sesión sin estado, el trabajo completo del agente, para un conjunto de observaciones dadas, se realiza dentro de una sola llamada a un método (en este caso el que recibe las observaciones). Sin embargo, si se requiere mantener estado, se mantiene en el cliente (lado del usuario) y se pasa a los métodos del bean de sesión sin estado en cada llamada a un método. Esto último depende de la aplicación donde se ejecute el agente y los requerimientos funcionales de tal aplicación.

Nuestro agente es especificado como un EJB de sesión sin estado porque proporciona componentes más eficientes en la capa de lógica de negocios de la aplicación, donde la velocidad y eficiencia son importantes [12], pero exige

mantener la información del estado en algún otro lugar, probablemente en la capa de presentación. Sin embargo, un Simulante no amerita el uso de recursos externos para el mantenimiento del estado de la sesión, pues la ejecución de una acción o un plan de acciones deben concretarse en cada llamada del bean, es decir, una solicitud de servicio al agente no depende de una solicitud de servicio anterior.

Es importante destacar que al momento de desplegar nuestro agente como EJB en un contenedor, el contenedor crea un almacén (pool) de instancias, todas ellas del mismo Bean. Si un usuario requiere el uso de un agente, el contenedor asigna cualquier instancia disponible al usuario. Este pool de instancias es administrado por el contenedor según lo descrito en la sección 1.5.1.

4.4.2. Especificación del componente

En la sección 1.5, se describieron los conceptos asociados a la tecnología EJB. En la sección 1.5.3, se definieron las características esenciales de lo que son los EJB de sesión sin estado. En la sección 1.5.5 se describió la manera en que se desarrolla un EJB y cuáles son los componentes que se encapsulan para especificar el ciclo de vida y las funcionalidades de un bean.

En esta sección se especifica nuestro agente como un componente de software reutilizable haciendo uso de la tecnología EJB. Como se dijo en la sección anterior, nuestro agente será un EJB de sesión sin estado.

La figura 4.4.3 muestra la especificación de Simulante como un componente de software reutilizable EJB.

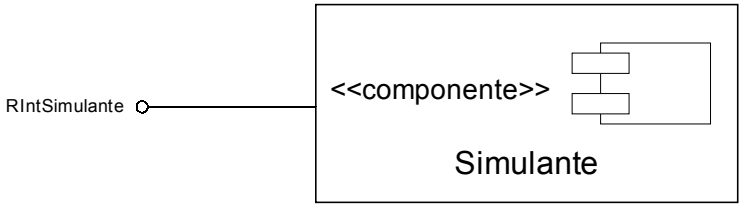

Figura 4.4.3: Especificación de Simulante como EJB

El nombre del componente bean es *Simulante*, la interfaz remota del bean recibe el nombre de *RIntSimulante* y mantiene los métodos que el agente como componente expondrá a los usuarios. La interfaz remota del bean debe extender de la interfaz *EJBObject*. Esta propiedad se muestra en la figura 4.4.5.

La figura 4.4.4 muestra la relación que existe entre el componente y su interfaz remota.

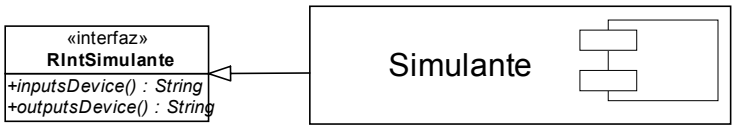

Figura 4.4.4: Interfaz remota de Simulante

El método publico *inputsDevice()* presente en la interfaz remota del componente, devuelve un objeto tipo *String* que contiene el conjunto de observaciones iniciales que el agente percibe directamente de su ambiente. Según lo descrito en la sección 4.2, estas observaciones son las llamadas *efecto de orden* para el agente. Las observaciones son entregadas al componente desde la capa de presentación. Un módulo de procesamiento de lenguaje se encargará de convertir el mensaje del usuario en un conjunto de observaciones atómicas. Estas observaciones son capturadas por el metodo *inputsDevice()* y entregadas al componente para su procesamiento.

Finalmente el método *outputsDevice()* reúne las influencias del agente luego del procesamiento de sus acciones. Este método devuelve un objeto tipo *String* que contiene información de respuesta en un lenguaje entendible por el agente. El módulo de procesamiento de lenguaje se encargará de traducir la semántica de esa información para ser entendida por la aplicación o el usuario.

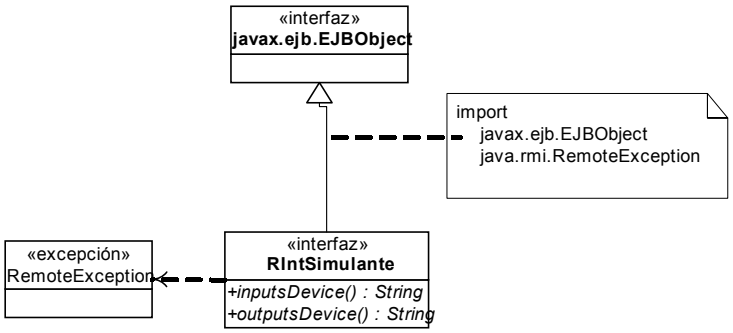

Figura 4.4.5: Especificación de de la interfaz remota de Simulante

La figura 4.4.6 muestra la interfaz *home*, llamada *HIntSimulante*, para el componente *Simulante*, la cual define el método *create()*, que devuelve un objeto de tipo *RIntSimulante* y permite al usuario obtener una referencia remota del componente. Este método no necesariamente implica la creación del componente. El contenedor EJB puede tomar decisiones acerca del pool de componentes. El método *create()* permite tanto crear una instancia del componente y colocarla en el pool como obtener un componente ya instanciado del pool.

Al igual que la interfaz remota, la interfaz home de un componente debe extender de una interfaz base. Como se muestra en la figura 4.4.6, la interfaz home extiende de la interfaz EJBHome.

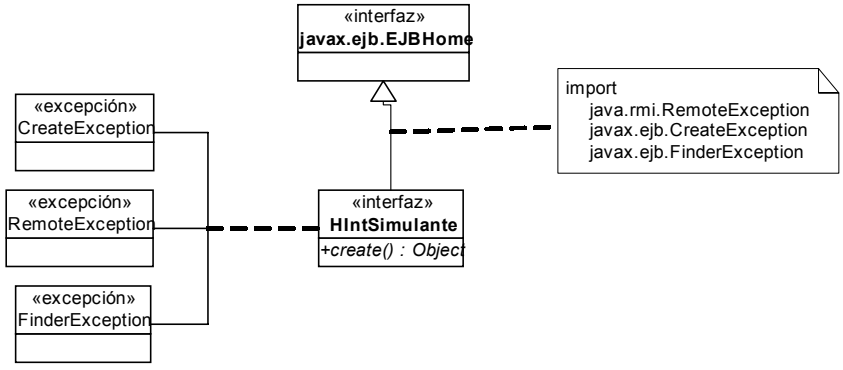

Figura 4.4.6: Especificación de la interfaz home de Simulante

Una vez definidas las interfaces del componente, es necesario especificar el componente en sí. La figura 4.4.7 muestra la especificación de nuestro componente como un EJB de sesión sin estado.

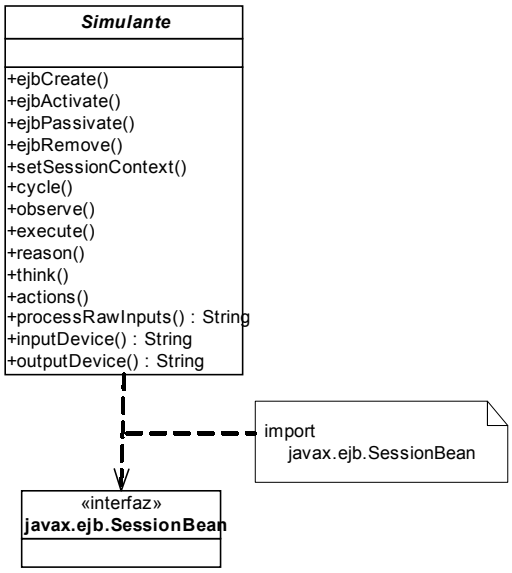

Figura 4.4.7: Especificación de Simulante como EJB de sesión sin estado

Los métodos *cycle(), observe(), execute(), reason()* y *think()* corresponden a los métodos explicados en la sección 4.3. El método *actions()* es un método simbólico para representar al conjunto de métodos que implementan las acciones concretas del agente (es la interfaz agente-ambiente). El método *processRawInputs()*, se encarga de procesar las observaciones para el agente obtenidas desde el ambiente o durante el proceso de razonamiento. Los métodos *inputsDevice()* y *outputsDevice()* corresponden a la implementación de los métodos públicos cuya firmas se encuentran en la interfaz remota del componente y sirven para comunicar al componente con la aplicación. El método *inputsDevice()* recibe las observaciones desde el ambiente para el agente y el método *outputsDevice* devuelve el conjunto de influencias que el agente causa sobre ese ambiente.

Los métodos *ejbCreate(), ejbActivate(), ejbRemove() y setSessionContext()* corresponden a los métodos cuyas firmas se encuentran en la interfaz *SesionBean* y sirven para que el contenedor EJB gestione el ciclo de vida de los componentes.

De esta manera obtenemos la implementación de nuestro agente interfaz como un componente de software reutilizable EJB de sesión sin estado, un dominio de conocimiento específico y una aplicabilidad practica que nos permitirá establecer los parámetros de desempeño del agente y un marco de desarrollo para nuevas aplicaciones. En resumen, nuestro trabajo integra los siguientes conceptos, tecnologías y servicios:

- Un agente de software basado en el concepto propuesto por Russell [1] y descrito en la sección 1.1, con propiedades de autonomía, sociabilidad, iniciativa y veracidad;

- La especificación de ese agente como un agente reactivo y racional basada en una técnica de razonamiento propuesta por Dávila [5], descrita en la sección 1.2;

- La descripción lógica de GLORIA [5], definida en la sección 1.2, para obtener: un modelo lógico de agente, un mecanismo de razonamiento, un motor de inferencia, un marco para la definición de la base de conocimientos del agente y la implementación de ese agente en Java.

- La descripción de un agente interfaz basado en el conocimiento, definido en la sección 1.3, para caracterizar a nuestro agente dentro de un dominio de aplicación orientado al proceso de enseñanza-aprendizaje, específicamente, con roles de asistente y de consultor en ese dominio.

- Una interfaz agente-ambiente programada en Java donde se programan las

acciones que el agente puede ejercer sobre ese ambiente;

- La implementación de ese agente en Java como un envoltorio del planificador o motor de inferencia y de la interfaz agente-ambiente;

- La especificación de ese envoltorio como un componente de software reutilizable EJB de sesión sin estado, con dos interfaces: una para la creación y gestión del componente (interfaz home) y la otra para la interacción con el usuario y la aplicación (interfaz remota);

- La definición de una base de conocimiento específica para el agente orientada a la simulación de modelos escritos en Galatea [2];

- La especificación de la capa de lógica de negocios para aplicaciones cooperativas como Bioinformantes [3];

- Un marco de desarrollo para la generación de la base de conocimientos del agente de acuerdo a una estrategia de clasificación de observaciones: efectos de orden y efectos de control; planes: simples y compuestos e influencias: las cuales se describen en función a los efectos de orden y/o de control para el agente.

- Un modelo de pruebas y producción de nuestro agente basado en el carácter de racionalidad propuesto por Russell y descrito en la sección 1.1. Estas pruebas se relacionan directamente con el dominio de aplicación del agente.

Estos elementos integrados constituyen lo que nosotros llamamos un Simulante: un modelo de agente interfaz basado el lógica y especificado como componente de software reutilizable.

En el siguiente capítulo, se detalla la forma de integrar a nuestro agente a una aplicación Web cooperativa y distribuida, se utiliza una aplicación J2EE

demostrativa para la implementación del agente, y se muestra cómo realizar el proceso de despliegue de toda la aplicación incluyendo el despliegue del componente en un servidor de aplicaciones. Finalmente, se detallan los aspectos de integración de nuestro agente para aplicaciones ya desarrolladas, colocando como plataforma de implementación a Bioinformantes.

5. Integración de Simulantes a una aplicación

Introducción

En esta sección, se describe la forma de implementación de nuestro agente como componente de software reutilizable EJB en una aplicación J2EE demostrativa que incluye: una aplicación Web que sirve como interfaz de usuario para solicitar los diferentes servicios del agente, un componente Web que permite la interacción de la aplicación con el agente componente a través de sus interfaces y un componente EJB de sesión sin estado que encapsula al agente y lo describe como un componente de software reutilizable exponiendo sus interfaces. Estos tres componentes definen, respectivamente, la capa de cliente, la capa de presentación y la capa de lógica de negocios de una aplicación Web J2EE.

5.1. Arquitectura de la aplicación

La figura 5.1.1, muestra la arquitectura de la aplicación que implementará al agente interfaz sobre un dominio de conocimiento orientado a la simulación de modelos escritos en Galatea[2], la cual es definida en función de la arquitectura de Bioinformantes, presentada en la sección 2.1:

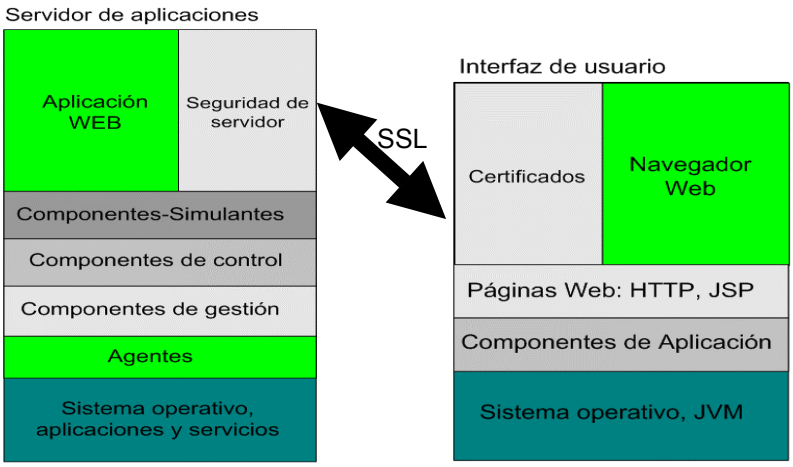

Figura 5.1.1: Arquitectura de la aplicación

Del lado del cliente, se despliega, en un navegador Web, la interfaz gráfica de usuario. Un usuario accede a la aplicación por un Puerto de Conexión Segura (SSL, Security Socket Layer) obteniendo un certificado de conexión. Los componentes de la capa de cliente son páginas HTML y JSPs, estos últimos desplegados en el lado del servidor. En el nivel más bajo, está el sistema operativo del cliente y la máquina virtual de Java. La interfaz del usuario se conecta a los componentes de la capa de presentación desplegados en el servidor, los cuales son encargados de la comunicación con el agente a través de sus interfaces.

Del lado del servidor, se despliegan varios tipos de componentes agrupados en las capas de presentación y lógica de negocio de la aplicación. Los primeros son componentes de aplicación que se comunican con la capa del cliente y sirven para la generación dinámica de contenido y el control de flujo de información. La capa de lógica de negocio ejecuta a nuestro agente Simulante como un componente de software reutilizable EJB de sesión sin estado. En el nivel *Componentes-Simulantes* mostrado en la figura, se almacena el conjunto de instancias del EJB, es decir, el pool de instancias del contenedor. Los componentes de control son módulos especializados para procesar el lenguaje de interacción entre el usuario y el agente. Los componentes de gestión es el agente en sí, el cual se comunica con el motor de inferencia y la base de conocimiento para el procesamiento de las observaciones y la generación de influencias del agente.

5.2. Integración del agente Simulante

Como se explicó en la sección 4.1 Simulante es un agente programado para cumplir dos roles en una aplicación Web orientada a la simulación de modelos escritos en Galatea.

Las tablas 5.2.1 y 5.2.2 muestran los casos de uso de interacción usuario/agente y el momento en que el agente debe intervenir como asistente o consultor de acuerdo al flujo de datos dentro de la aplicación.

Tabla 5.2.1:

Caso de uso <<Solicitud de asistencia al sistema>>
• El **usuario** introduce un mensaje dirigido al **agente.** • La *aplicación* valida el mensaje de entrada y determina el flujo de información. • La *aplicación* construye, a partir del mensaje de entrada, el conjunto de observaciones que manipulará el **agente**. • El **agente** percibe el conjunto de observaciones y efectúa las acciones correspondientes. • La *aplicación* construye los mensajes de respuesta al **usuario** de acuerdo a las acciones del **agente**. • El **usuario** valida los resultados

Tabla 5.2.2:

Caso de uso <<Solicitud de consultoría al sistema>>
• El **usuario** introduce un mensaje dirigido al **agente** solicitando asesoría. • La *aplicación* valida el mensaje de entrada y determina el flujo de información. • La *aplicación* construye, a partir del mensaje de entrada, el conjunto de observaciones que manipulará el **agente**. • El **agente** percibe el conjunto de observaciones. • El **agente** determina el plan de acción como consultor. • El **agente** efectúa las acciones correspondientes sobre el plan de acción seleccionado. • La *aplicación* construye los mensajes de respuesta al **usuario** de acuerdo a las acciones del **agente**. • El **usuario** manipula la información de salida.

En ambos casos, una vez que el usuario introduce un mensaje dirigido al agente, la aplicación utiliza una forma controlada de lenguaje natural y construye el conjunto de átomos observables para el agente. El agente recibe este conjunto de átomos y los procesa de acuerdo a su base de conocimientos. Una vez que el motor de inferencia del agente determina el plan de acción, el agente ejecuta las acciones correspondientes. El resultado de cada acción ejecutada por el agente es entregado a los componentes Web para su presentación y visualización al usuario.

La figura 5.2.1 muestra el flujo de datos que se produce una vez que la aplicación detecta un mensaje para el agente en su rol de asistente o consultor. Es importante destacar que las tres líneas verticales hacen referencia, respectivamente, a la capa de cliente, presentación y negocios de la arquitectura

multicapa de Simulantes.

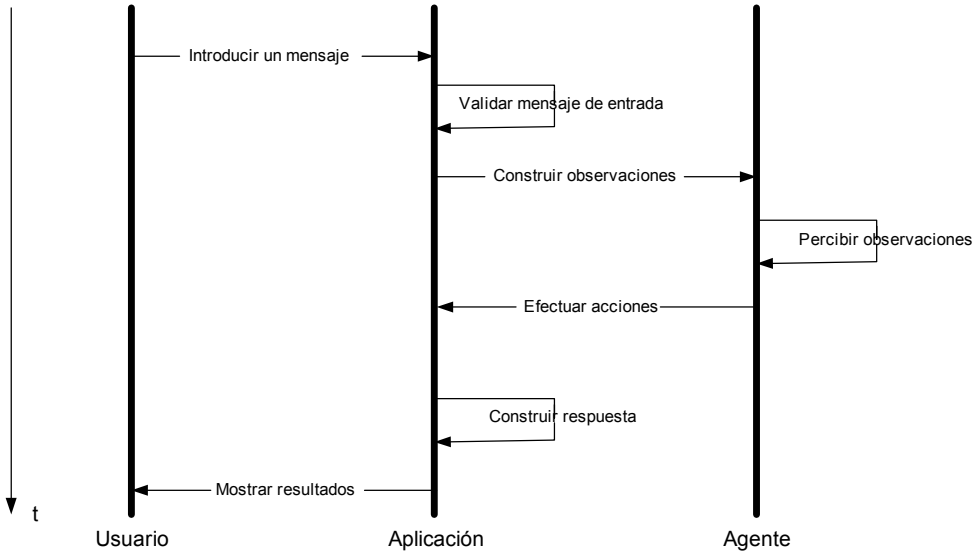

Figura 5.2.1: Diagrama de actividad,
flujo de datos usuario/agente

El flujo de datos existe entre el usuario y el agente. El usuario puede introducir un mensaje en lenguaje natural, cumpliendo con cierta estructura, para solicitarle al agente que efectúe alguna acción. Este mensaje es procesado por la aplicación, el cual valida el flujo de la información y realiza una representación abstracta del mensaje para que sea entendido por el agente. El agente recibe las observaciones desde su ambiente y, de acuerdo a sus metas, reglas y creencias efectúa una o un conjunto de acciones acordes con los requerimientos del usuario. Una vez que la intervención del agente termine, la aplicación construye un mensaje de respuesta indicándole al usuario el resultado de la intervención del agente.

5.3. Despliegue de la aplicación en un servidor

Las aplicaciones J2EE, deben ser instaladas en un servidor de aplicaciones que proporcione un contenedor para desplegar los componentes de cada una de las capas de la aplicación. Existe una gran cantidad de proveedores que

proporcionan servidores de aplicaciones para desplegar cada una de las capas de la aplicación. Otros servidores sirven para desplegar una sola de las capas J2EE, lo que obliga a correr varios servidores de acuerdo al número de capas de la aplicación.

Una de las propuestas de servidor de aplicaciones más utilizada es la desarrollada por Sun Microsystems, llamada Sun One Application Server (SOAS) [25]. Entre las características más importantes del SOAS se destacan las siguientes [25]:

- Incluye un monitor de procesamiento de transacciones integrado que permite distribuir transacciones de forma segura entre varias fuentes de datos.

- Proporciona varias opciones de equilibrio de carga.

- Amplía el equilibrio de carga a la tecnología JSP y las interfaces RMI/IIOP como "clientes enriquecidos" como aplicaciones Java de cliente.

- Ofrece tecnología de memoria caché avanzada para mejorar el rendimiento de acceso a la base de datos y Java Servlets.

- Sun ONE Web Server y Sun ONE Directory Server monitorizan automáticamente cualquier actualización realizada en Sun ONE Application Server, los clústers o aplicaciones de Enterprise Edition. Esto reduce la carga de trabajo de los administradores sistema cuando se agregan o modifican las aplicaciones J2EE y ayuda a que estén disponibles las aplicaciones más recientes y eficaces.

Este servidor de aplicaciones proporciona un contenedor para cada una de las capas de la aplicación que corren del lado del servidor. La capa de presentación utiliza un contenedor Web llamado Tomcat [26]. Este contenedor se encarga del despliegue de los componentes Web, control de acceso, seguridad y

transacciones en la aplicación. La capa de lógica de negocios proporciona un contenedor de componentes J2EE que se encarga del despliegue de EJBs y su interacción con el resto de las capas de la aplicación. La capa de datos proporciona un sistema manejador de base de datos para manejar la persistencia de datos de la aplicación.

El SOAS proporciona servicios de seguridad como Securiry Socket Link (SSL) [27], el cual permite que todos los datos que se envíen, entre el usuario y el servidor, viajen cifrados por la red. Igualmente, el SOAS, proporciona, al administrador del servidor de aplicaciones, una interfaz vía Web, para el control y monitoreo de las aplicaciones desplegadas en el servidor. De igual manera permite la administración de cuentas de usuarios y grupos para cada una de las aplicaciones y sus componentes.

Para desplegar nuestra aplicación en el servidor de aplicaciones SOAS es necesario cumplir una serie de pasos que constituyen la integración de componentes desplegados en diferentes contenedores, pero instalados en un mismo servidor de aplicaciones.

Para ello es importante establecer qué componentes serán desplegados en el contenedor Web, y qué componentes serán desplegados en el contenedor EJB. Esta clasificación es proporcionada por la arquitectura de nuestra aplicación y por la especificación de nuestro agente como componente de software reutilizable.

La figura 5.1 muestra la composición del nuestro agente como componente de software reutilizable EJB una vez desplegado en un servidor de aplicaciones. De acuerdo con la especificación de nuestro agente, descrita en el capítulo 4, el descriptor de despliegue es un archivo XML que contiene las entradas necesarias para gestionar el ciclo de vida del bean (llamado *Simulante*) por medio de su interfaz home (llamada *HIntSimulante*) y desplegar los métodos públicos a otros componentes de aplicación (Ej. Un servlet o una página JSP) a través de su

interfaz remota (llamada *RIntSimulante*). Una instancia del bean es creada en el contenedor EJB una vez invocado el método *create*() de la interfaz home. Cuando el contenedor Web amerita el uso del EJB, utiliza cualquiera de los métodos expuestos en la interfaz remota del bean (*inputsDevice*(*String a, String o*) para solicitar la intervención del agente dado un conjunto de observaciones y *outputsDevice*() para recibir el conjunto de influencias). La interfaz remota se comunica con el bean utilizando los mismos métodos públicos. El bean es una subclase de la clase base *Agent* que ejecuta el ciclo *observar*, *razonar* y *actuar* del agente. Mientras este ciclo ocurre, el bean utiliza el método *think*() con el que lee la base de conocimiento y calcula el plan de acción a ejecutar utilizando un Planificador o Cerebro (*simulante*). El agente postula sus influencias a través de su Interfaz Agente Ambiente utilizando el método *proccessRawsInputs*() y las escribe en una estructura de datos que puede ser leída por el método público *outputsDevice*(). El componente Web encargado de la comunicación entre la interfaz remota del bean y el contenedor Web, es un servlet que debe ejecutar un Módulo de Procesamiento de Lenguaje Natural (MPDN) para calcular las observaciones del agente como un conjunto de átomos que puedan ser interpretados por el planificador. Igualmente el MPDN debe interpretar las influencias del agente y devolverlas en lenguaje natural para que puedan ser entendidas por el usuario.

De igual manera, la figura 5.1 muestra qué componentes serán desplegados en el contenedor Web y qué componentes serán desplegados en el contenedor EJB.

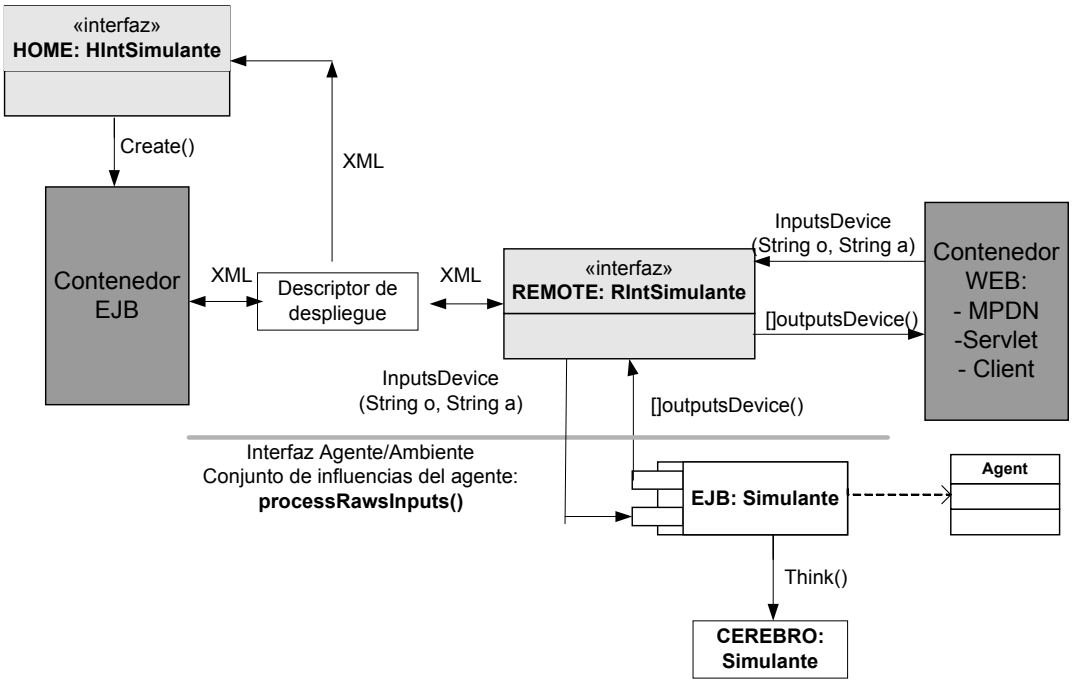

Figura 5.1: Diagrama de colaboración del Agente como Componente de Software Reutilizable desplegado en un servidor de aplicaciones.

5.4. Instalación y despliegue de la aplicación en el servidor Sun One Applications Server

Lo siguiente describe la forma de instalar y desplegar la aplicación en el servidor de aplicaciones de Sun Microsystems, el Sun One Application Server. Para ello, es importante configurar ciertas variables de entorno del sistema operativo y ubicar ciertos archivos y controladores en directorios específicos que son utilizados por el servidor de aplicaciones, por la aplicación en sí y por el agente como componente de software.

5.4.1. Configuraciones generales y variables de entorno

Para que nuestra aplicación pueda ser desplegada y ejecutada correctamente, es necesario configurar, establecer, definir y re-definir algunas de las variables de entorno del sistema:

Para simplificar la explicación, llamaremos JAVA_HOME a la ruta donde se encuentra el lenguaje Java, PROLOG_HOME a la ruta donde se encuentra el lenguaje Prolog, y así sucesivamente. Las siguientes son las variables de entorno que se deben configurar:

- PATH para que apunte a \JAVA_HOME\bin
- CLASSPATH para que apunte a \SUNONEAPPSERVER_HOME\bin
- PATH para que apunte a \PROLOG_HOME\bin
- CLASSPATH para que apunte a \SUNONEAPPSERVER_HOME\lib\j2ee.jar
- CLASSPATH para que apunte a \AGENTE_HOME\
- CLASSPATH para que apunte a \GALATEA_HOME\

Es importante cambiar las políticas de seguridad en el servidor de aplicaciones. Esto se debe a que nuestra aplicación hace uso de un conjunto de comandos del sistema y aplicaciones que son ejecutados en el lado del servidor. Algunos de ellos son, por ejemplo, el compilador de Java, el binario del Cerebro, la máquina virtual de Java, entre otros.

Para lograr esto, es necesario seguir los siguientes pasos:

- Editar el archivo de políticas, llamado server.policy, ubicado en el directorio \SUNONEAPPSERVER_HOME\domains\domain1\config

- Cambiar la permisologia de ejecución en la línea:

  permission java.io.FilePermission "<<ALL FILES>>", "read,write";

  por otra que diga:

  permission java.io.FilePermission "<<ALL FILES>>", "read,write,execute";

5.4.2. Archivos necesarios para el despliegue

Estos archivos se han almacenado en una estructura de directorios de fácil acceso, la cual se muestra en la figura 5.4.1:

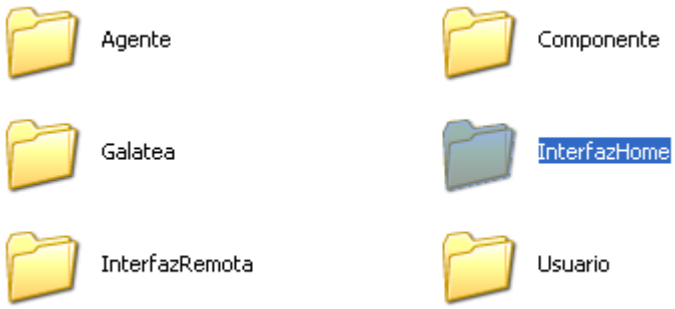

Figura 5.2: Composición d la aplicación en un directorio de trabajo

Los directorios *Agente*, *Componente*, *InterfazHome* e *InterfazRemota*, contienen dos subdirectorios adicionales: uno llamado *src* que contiene los archivos fuentes y otro llamado *sbin* que contiene el bytecode de cada elemento de la aplicación. El directorio *Usuario*, contiene dos subdirectorios: *Cliente* y *Servidor*, los cuales, a

su vez contienen dos subdirectorios más con los archivos fuente y bytecode respectivamente. El directorio *Galatea* contiene el API de Galatea.

5.4.3. Ubicación del motor de inferencia y fuentes Galatea

El directorio por defecto al que apunta el servidor de aplicaciones Sun One Applications Server es el siguiente: a \SUNONEAPPSERVER\ domains\domain1\config. Mientras esto no sea cambiado, el motor de inferencia, la base de conocimientos del agente y los archivos Galatea de los usuarios deben estar almacenados en este directorio. Se delega al sistema operativo el control de acceso y seguridad a este directorio.

5.4.4. Despliege del componente EJB

Como dijimos anteriormente, para desplegar el bean es necesario instalar un Servidor de Aplicaciones. La descripción de despliegue que daremos a continuación para nuestro componente (llamado *Simulante*), es utilizando el Servidor de Aplicaciones Sun One Applications Server. Este servidor puede ser instalado en cualquier sistema operativo (Windows, Unix, Solaris y MacOS) y proporciona una herramienta de despliegue para generar los descriptores de despliegue y organizar la aplicación en un archivo único Enterprise Applications Server (.EAR). Los componentes EJB son ubicados en un archivo .JAR y los componentes de aplicación (componentes Web) en un archivo .WAR.

Una vez instalado el servidor, es necesario construir la aplicación por medio de la herramienta de despliegue. La figura 5.3 muestra la interfaz de integración proporcionada por el Sun One Applications Server.

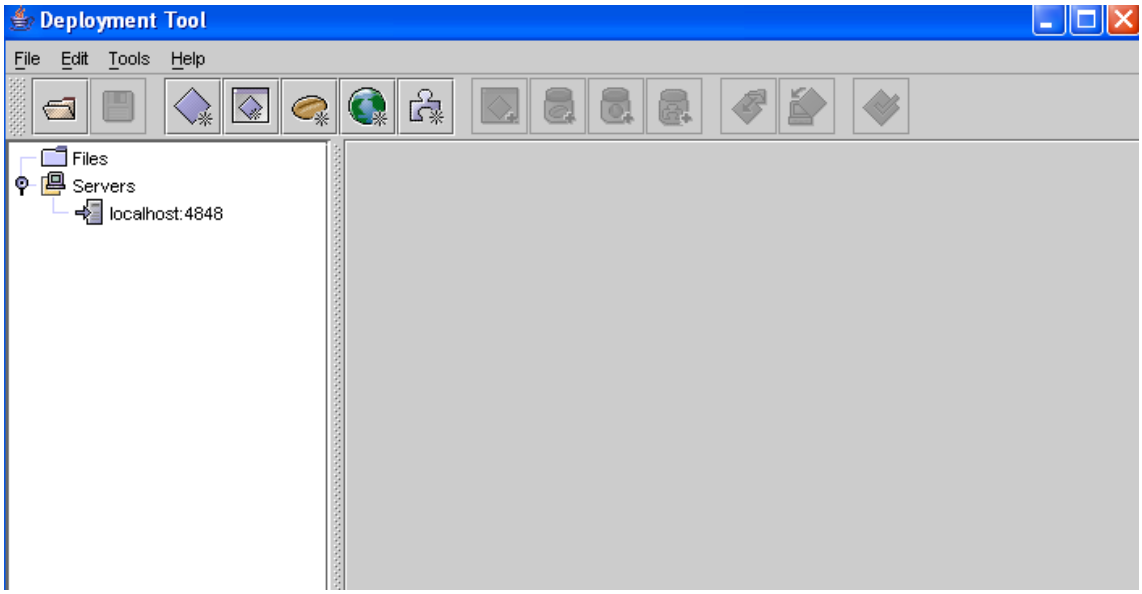

Figura 5.3: Herramienta de despliegue del Sun One Applications Server

La figura 5.4 muestra cómo agregar una nueva aplicación haciendo uso de la herramienta de despliegue del Sun One Applications Server.

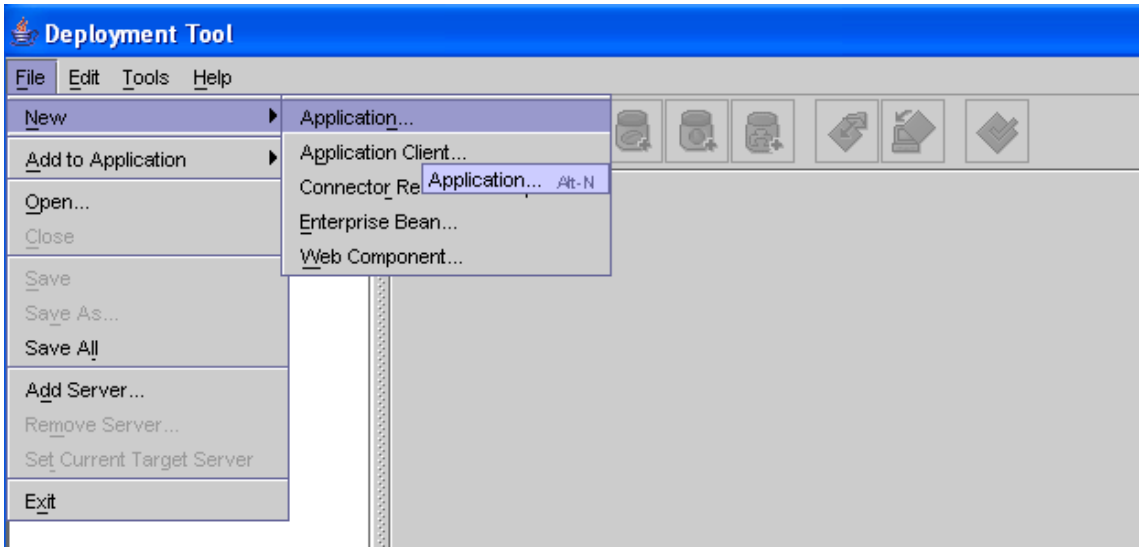

Figura 5.4: Agregar una nueva aplicación desplegable

En el cuadro de diálogo que se despliega, se debe definir la ruta donde se almacenará el archivo único .EAR. Para ello se utiliza el cuadro de texto *Application File Name* y se debe escribir *simulante* en el cuadro de texto *Application Display Name,* tal como se muestra en la figura 5.5.

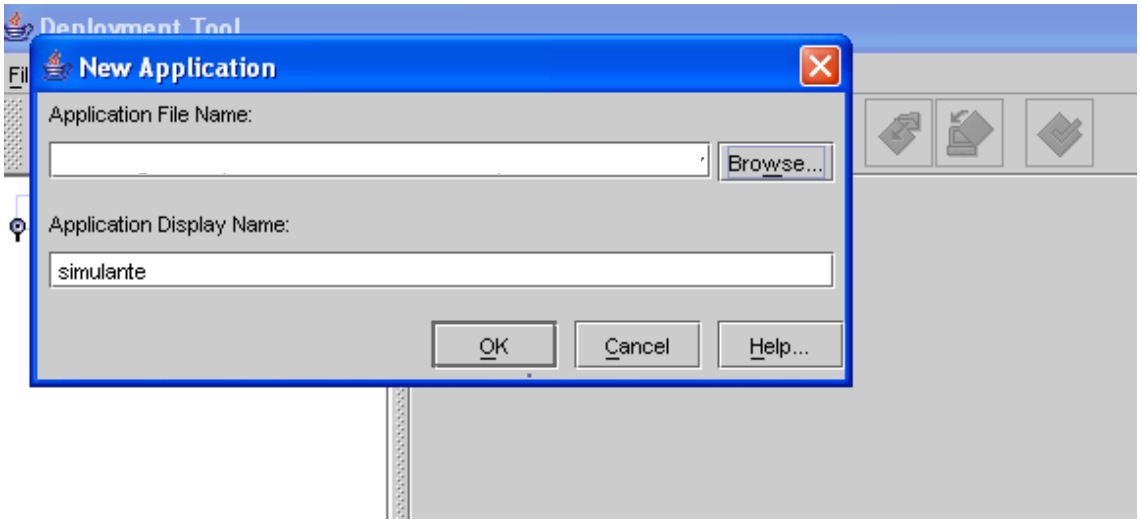

Figura 5.5: Definición del nombre de la aplicación

Una vez definido el nombre de la aplicación, es necesario agregar el componente Web que se desplegará en el servidor de aplicaciones, tal como se muestra en la figura 5.6.

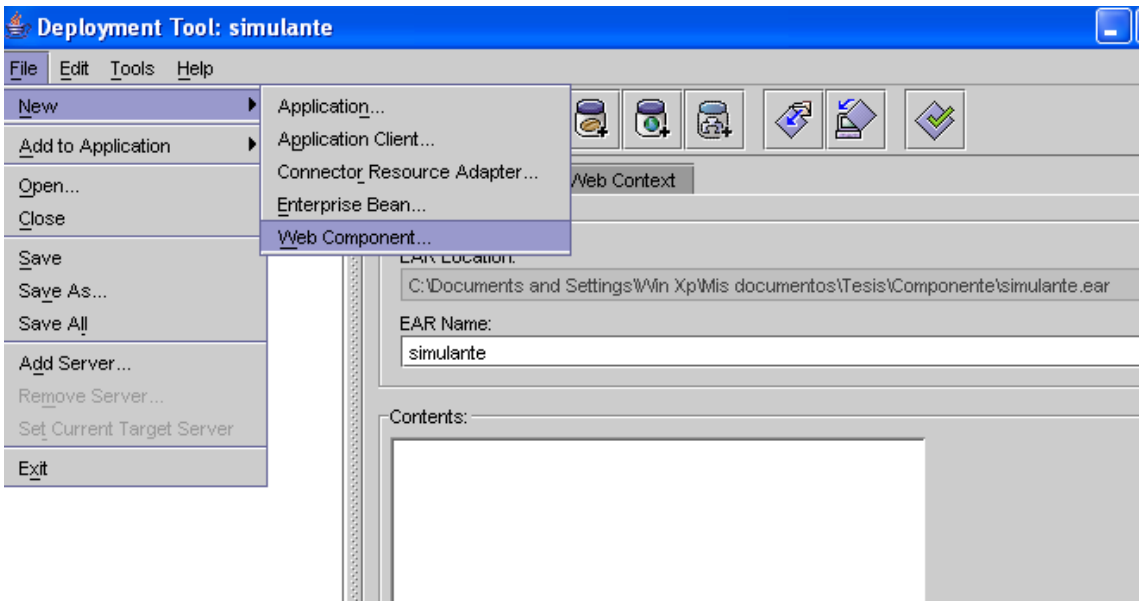

Figura 5.6: Agregar un componente Web a la aplicación

El componente Web es el archivo .WAR que se despliega en el contenedor Web. Este debe incluir el servlet, que se encarga de comunicar al usuario de la aplicación con el agente, y las páginas que se despliegan en el navegador del usuario.

Una vez invocada esta opción en la herramienta de despliegue, aparecerá un asistente para estructurar y configurar los componentes Web de la aplicación, tal como se muestra en la figura 5.7.

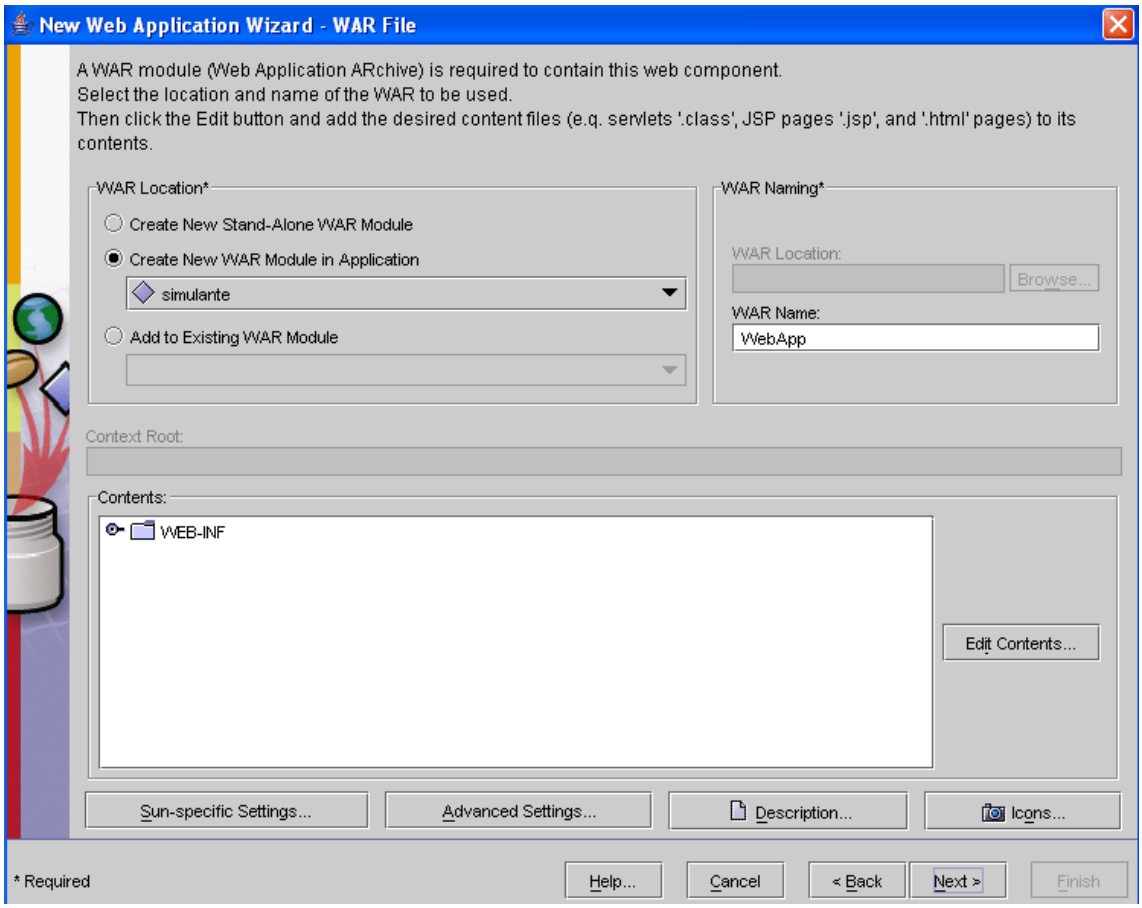

Figura 5.7: Asistente para integrar un componente Web a la aplicación

Pulsando el boton *Edit Contents…,* aparecerá una nueva ventana donde se podrá incluir los archivos del componente Web que serán almacenados en el archivo .WAR. La imagen 5.8 muestra la estructura de directorios una vez incluidos los componentes Web:

Estos componentes se encuentran en el directorio *Usuario* del conjunto de archivos de composición de la aplicación.

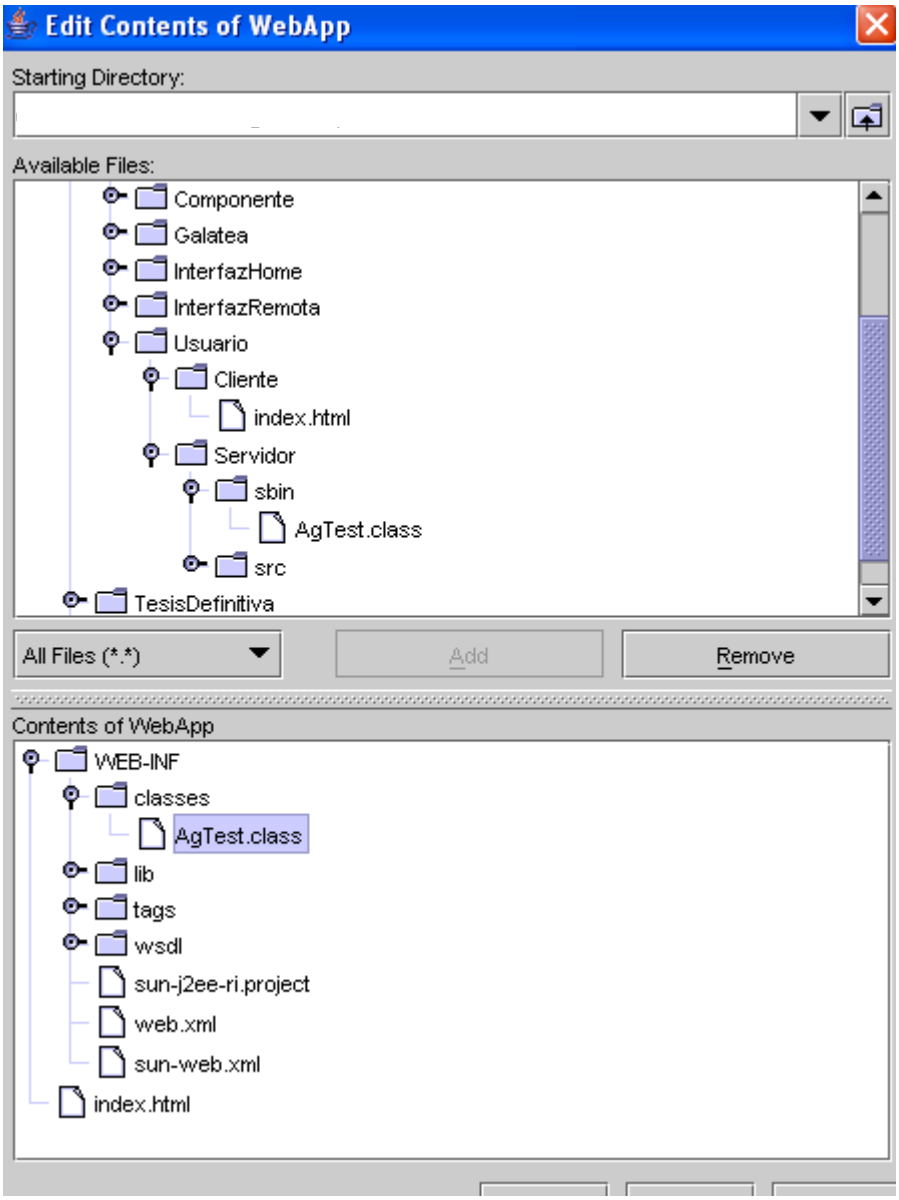

Figura 5.8: Ubicación de los componentes Web de la aplicación

Una vez concretado este paso, el siguiente requerimiento del asistente es el de definir qué tipo de componente Web será utilizado en la aplicación para comunicarse con el agente, el cual, tal como se muestra en la figura 5.9 puede ser un servlet, una página JSP o un servicio Web. Nuestra aplicación utiliza un servlet.

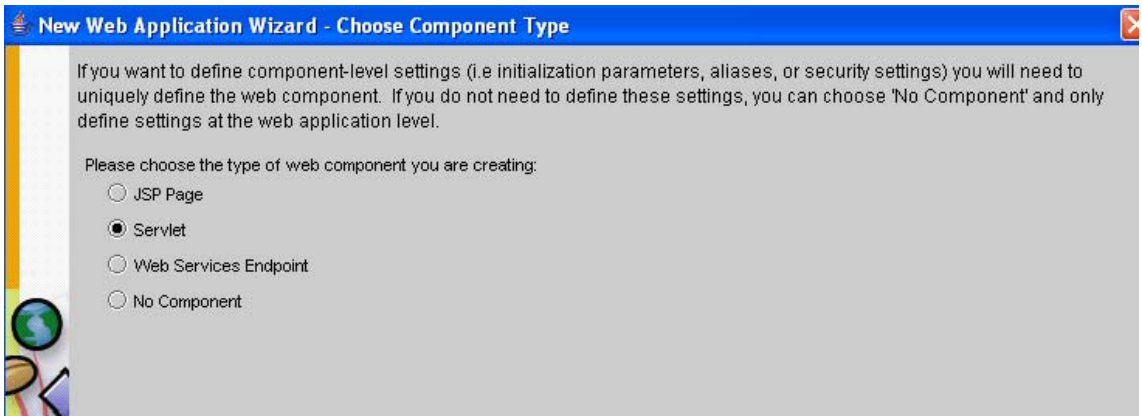

Figura 5.9: Definición del componente Web de la aplicación

El siguiente requerimiento permite definir el nombre del servlet que será utilizado. Esto puede colocarse en la pantalla del asistente que se muestra en la figura 5.10.

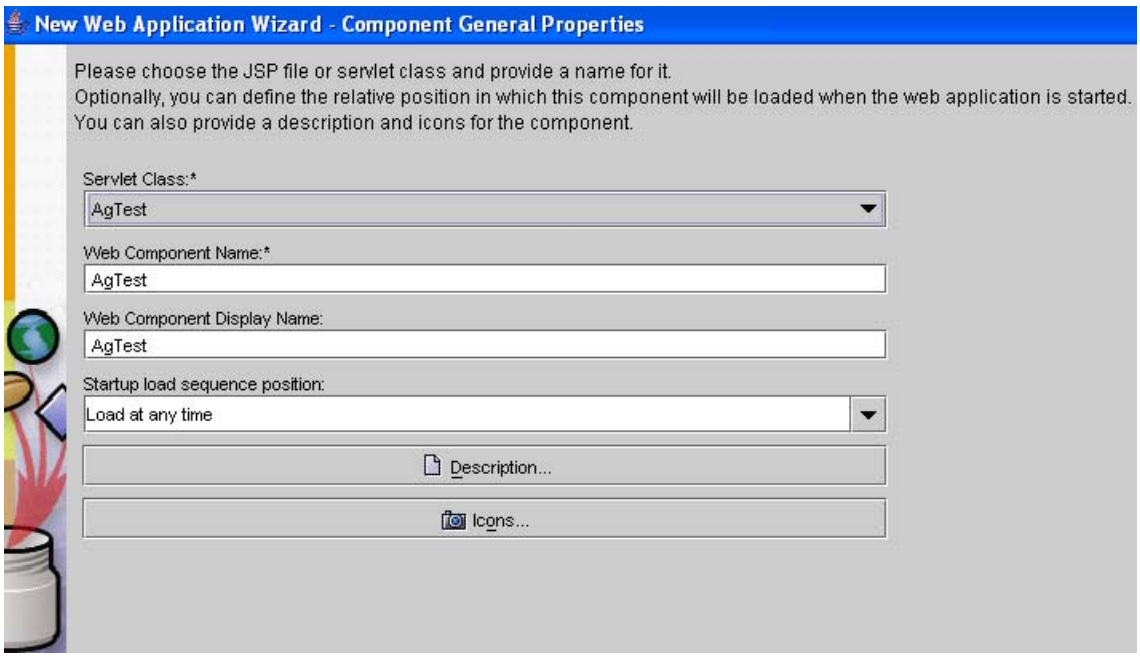

Figura 5.10: Nombre del componente Web de la aplicación

Una vez finalizado el asistente, se debe desplegar la siguiente pantalla: Nótese que en el cuadro de texto *Context Root* se debe especificar el nombre del contexto de la aplicación.

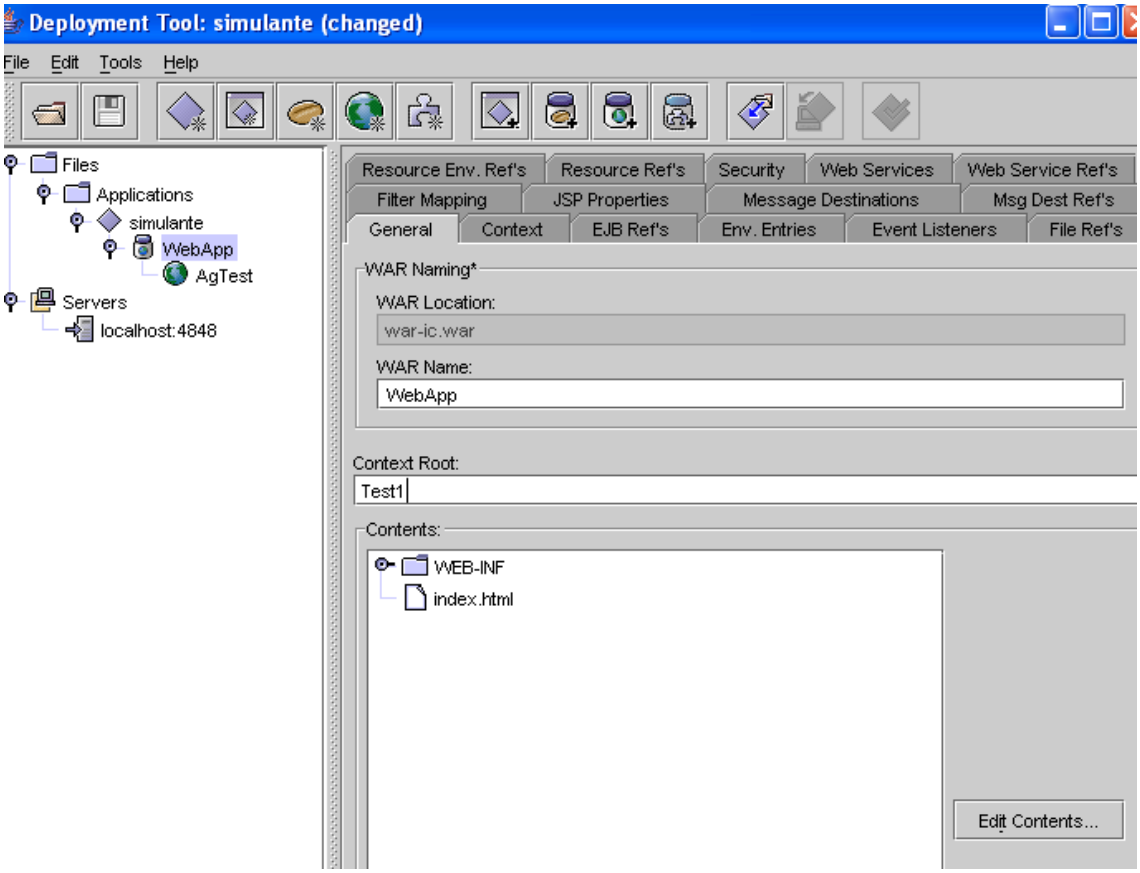

Figura 5.11: Definición del contexto de la aplicación

Haciendo clic en el componente Servlet *AgTest* y luego en la pestaña *Aliases*, se debe definir el nombre lógico del servlet, que será utilizado por el contenedor Web para ubicarlo y ejecutarlo. En nuestro caso el nombre es *AgMapping*, tal como se muestra en la siguiente figura.

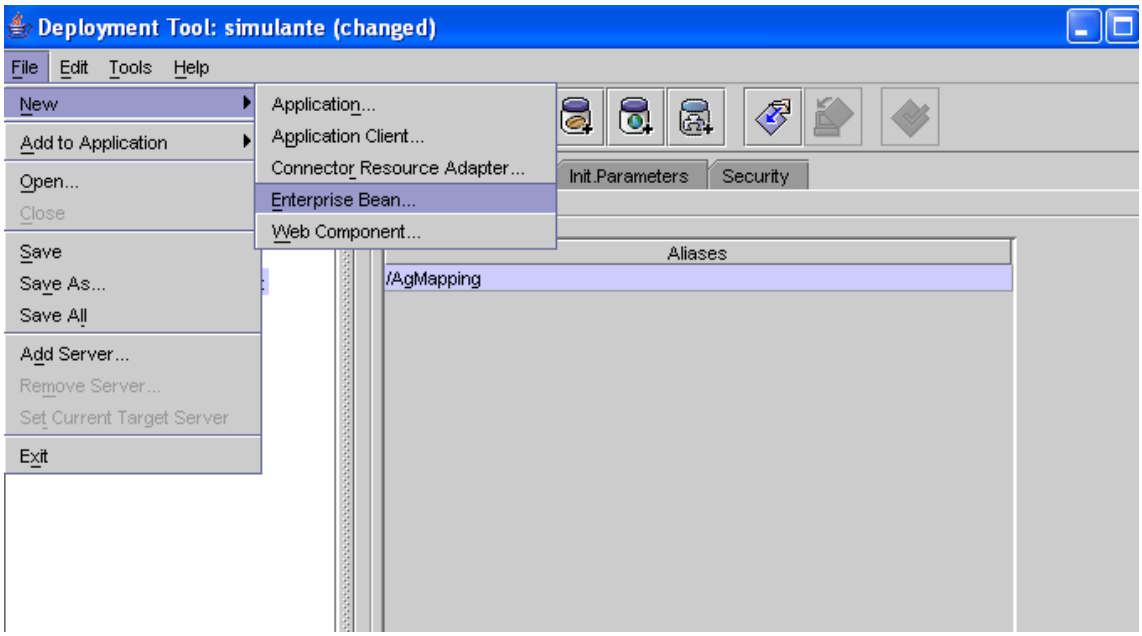

Figura 5.12: Definición del alias para el servlet de comunicación

Una vez instalados los componentes Web de la aplicación, es necesario agregar el componente de software reutilizable. Tal como se muestra en la figura 5.13. Una vez invocada esta opción en la herramienta de despliegue, aparecerá un asistente para estructurar y configurar los componentes de software reutilizables EJBs de la aplicación.

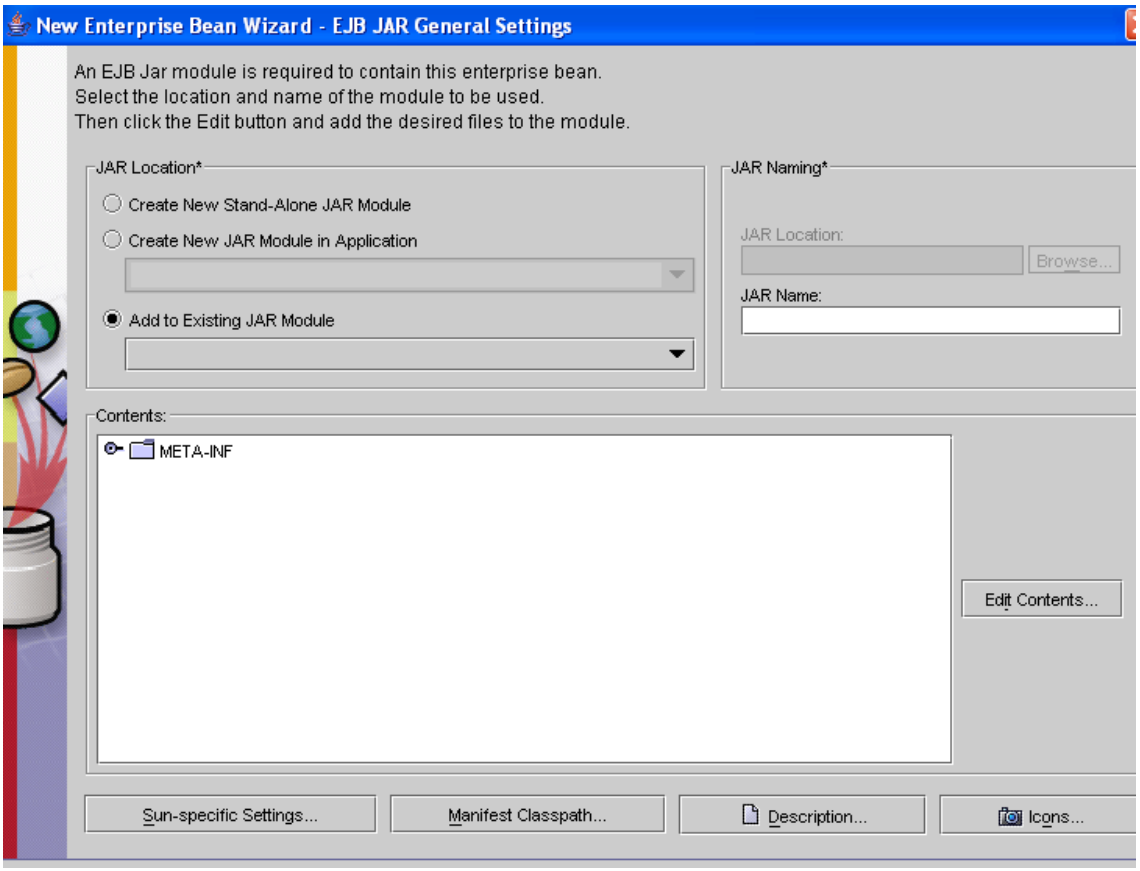

Figura 5.13: Agregar un componente EJB a la aplicación

Pulsando el boton *Edit Contents*…, aparecerá una nueva ventana donde se podrá incluir los archivos del componente EJB que serán almacenados en el archivo .JAR. La figura 5.14 muestra la estructura de directorios una vez incluidos el EJB:

Este componente y sus interfaces se encuentran en el directorio componente, *InterfazHome* e *InterfazRemota*, respectivamente, en el conjunto de archivos de composición de la aplicación:

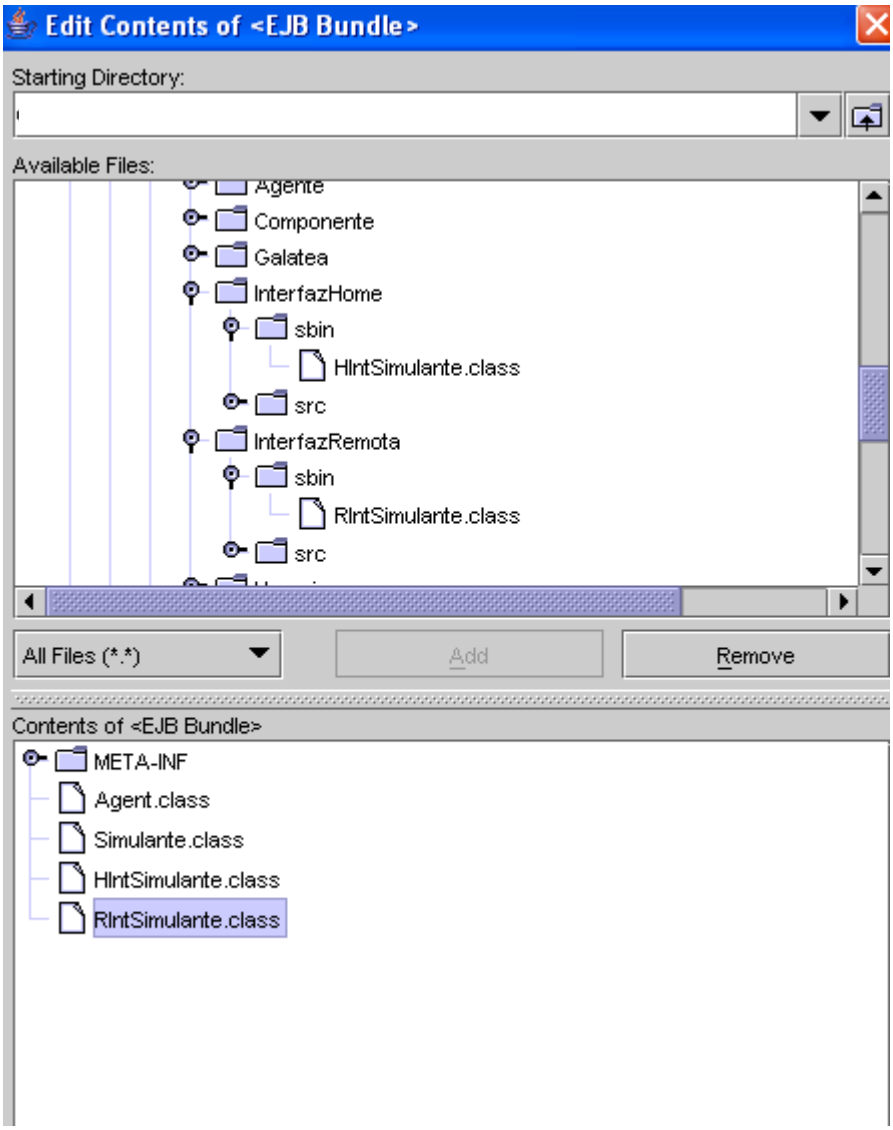

Figura 5.14: Ubicación de los componentes EJB para la aplicación

Una vez concretado este paso, el siguiente requerimiento del asistente es el de definir el bean y sus interfaces. La figura 5.15 muestra la elección correcta para nuestra aplicación:

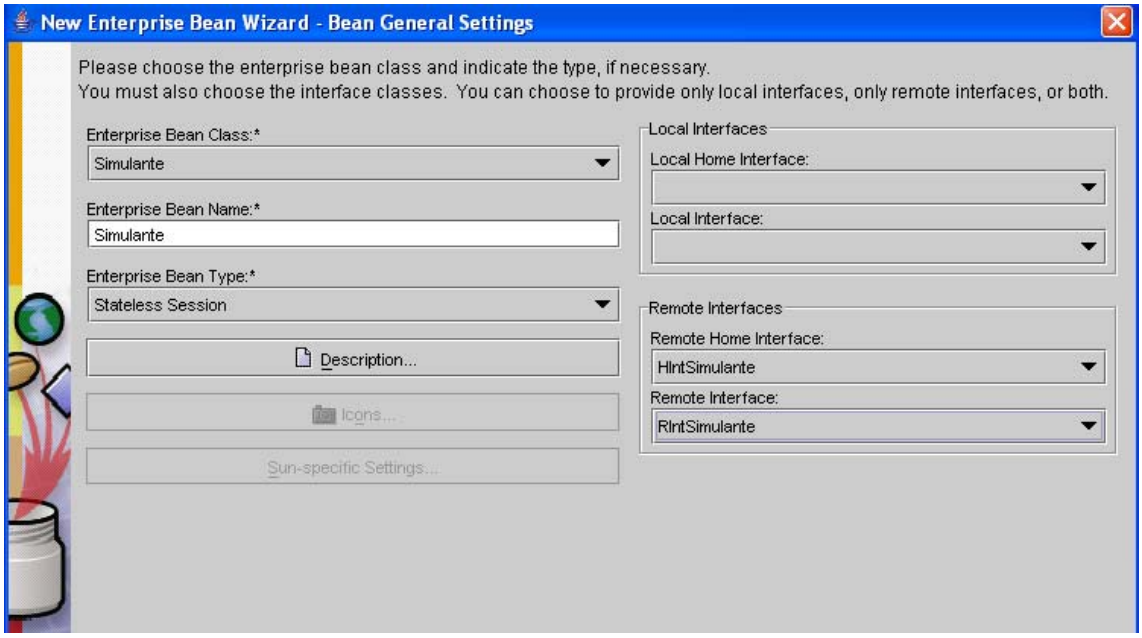

Figura 5.15: Definición del bean y sus interfaces

Con este paso se completa el ensamblaje de la aplicación.

Una vez ensamblada la aplicación y seguros de haber salvado todo lo hecho hasta ahora, procedemos al despliegue. La figura 5.16 muestra cómo comenzar a desplegar la aplicación.

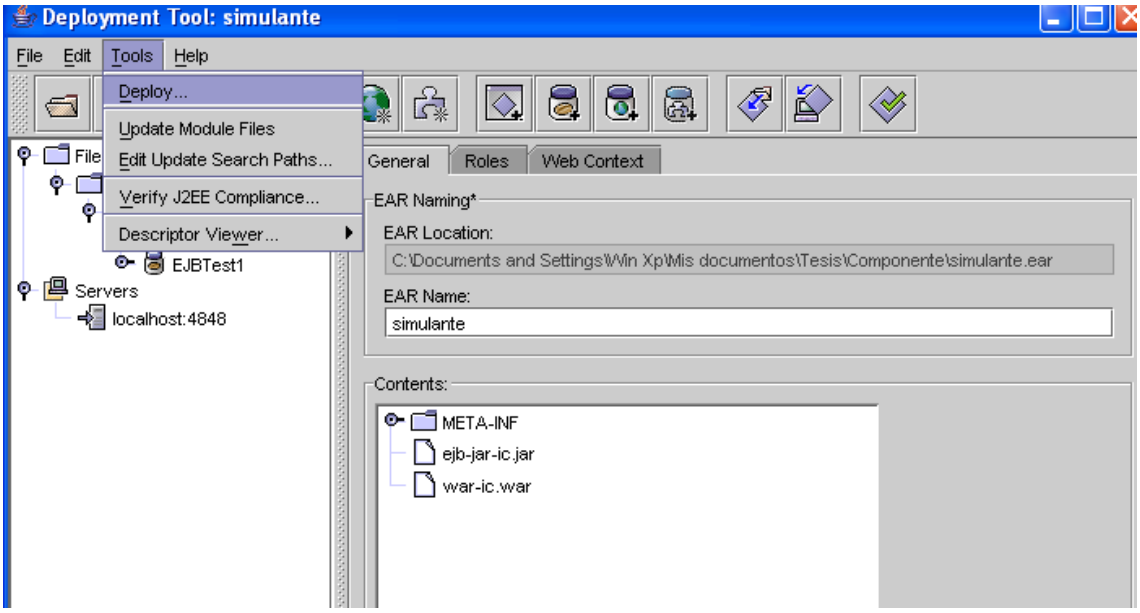

Figura 5.16: Inicio del despliegue de la aplicación en el servidor

Es importante que, para el momento del despliegue, el servidor de aplicaciones Sun One Application Server esté corriendo.

Una vez ingresado el login y password como administrador del servidor, la herramienta de despliegue comienza con el proceso de despliegue. La figura 5.17 muestra que el proceso de despliegue se ha ejecutado con éxito.

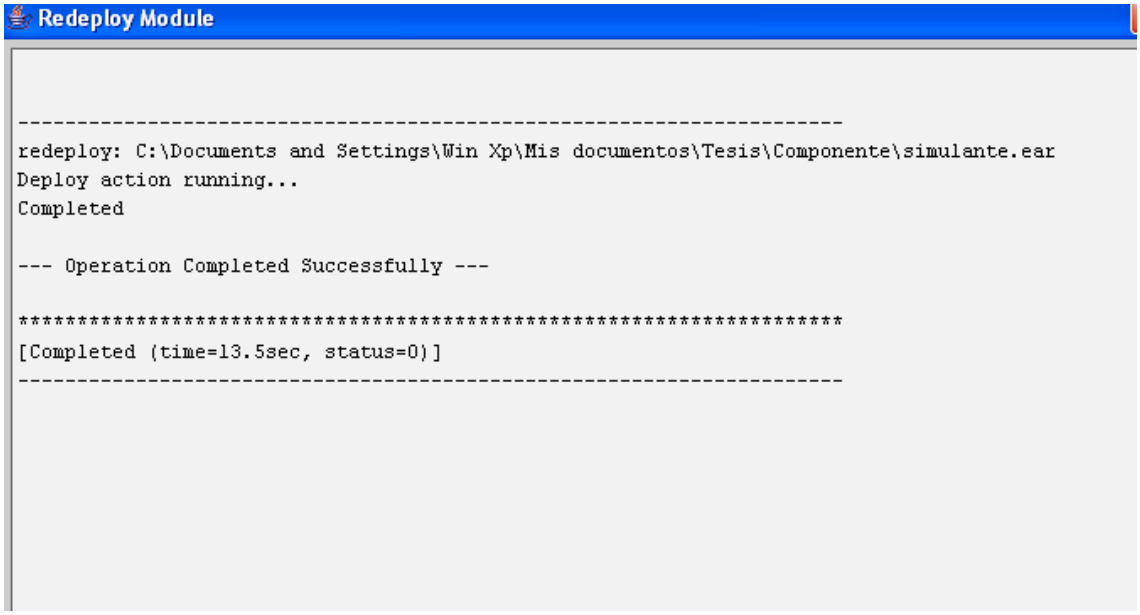

Figura 5.17: Finalización del proceso de despliegue de la aplicación

Con estos pasos hemos completado el proceso de instalación, configuración, despliegue y producción de una aplicación J2EE demostrativa, que incorpora a nuestro agente interfaz como un componente de software reutilizable EJB. Desde esta aplicación, hemos logrado hacer las pruebas necesarias para establecer una medida de rendimiento del agente en función a los servicios que presta, tanto en su rol de asistente como su rol de consultor.

La siguiente sección muestra cómo incorporar nuestro agente interfaz a una aplicación en producción. Para ello utilizaremos la arquitectura de Bioinformantes,

descrito en el capítulo 2, como una herramienta de uso cooperativo. Bioinformantes ofrece, no solo la tecnología como una aplicación J2EE multicapa, sino también todo una plataforma y componentes Web, que pueden ser utilizados y re-programados para servir como interfaz de comunicación entre el agente, la aplicación y el usuario.

5.5. Integración del agente a una aplicación en producción

En esta sección se describen los aspectos fundamentales de integración, programación, configuración, instalación y despliegue de nuestro agente interfaz en una aplicación Web J2EE en producción. Comenzaremos nuestra explicación, con la definición de un marco de desarrollo, propuesto en el capítulo 1, para definir y re-definir la base de conocimientos de nuestro agente en función a un dominio de aplicación particular. Luego mostramos cómo puede ser integrado ese agente a Bioinformantes [3] utilizando los componentes Web que esta plataforma provee.

5.5.1. Marco de desarrollo de una base de conocimientos e interfaz agente-ambiente

Como se explicó en la sección 4.2, la base de conocimientos del agente, esta programada en un lenguaje lógico proposicional y de predicados, definido en la sección 1.2.4. Este modelo de base de conocimientos debe ser construido tomando en cuenta cuatro estructuras de datos, que son utilizadas por el razonador para inferir los planes de acción del agente. Estas estructuras de datos están compuestas por los siguientes elementos:

- El conjunto de observaciones que el agente puede percibir de su ambiente.
- El conjunto de acciones que el agente puede ejecutar en su ambiente, las cuales son átomos que componen uno o más planes de acción.

- El conjunto de reglas que establecen la conducta del agente, las cuales le permiten al razonador seleccionar un plan de acción particular en un momento dado.

- El conjunto de planes compuestos por una o más acciones de acuerdo a las metas del agente.

Para desarrollar una base de conocimientos para un agente particular, el diseñador del agente debe generar, de acuerdo a un dominio de conocimiento, estos cuatro elementos, y escribirlos en la base de conocimientos. Un aspecto importante que identifica a un agente interfaz es que éste tipo agente posee un comportamiento similar en todas sus aplicaciones; el de servir como asistente de la aplicación.

Como se explicó en la sección 1.3.1, un agente interfaz busca continuamente la oportunidad de realizar tareas que ayuden a la satisfacción de los objetivos del usuario, por lo que debe estar atento a las peticiones y acciones del usuario para determinar su intervención como asistente de la aplicación.

La forma en que el usuario solicita y utiliza los servicios de la aplicación depende de la interfaz de usuario que la aplicación proporcione, y del propósito de la aplicación. Algunas técnicas utilizadas para ello son: la solicitud de un servicio especificado en una lista de servicios; la ejecución de eventos asignados un control de la interfaz: botones, imágenes, etiquetas, enlaces, etc.; la solicitud de servicios por herramientas de comunicaciones síncronas o asíncronas: salas de chat, correo electrónico, foros, pizarras virtuales, etc.; o la solicitud de un servicio haciendo uso de un cuadro de texto y, por medio de un módulo de procesamiento de lenguaje natural, determinar a que tipo de servicio se hace referencia.

Cualquiera que sea el caso, si se incorpora un agente interfaz a la aplicación, éste debe recibir el conjunto de observaciones tal cual como está especificado en la base de conocimientos, y así determinar su plan de acción.

De esta manera, un agente interfaz es activado una vez que la aplicación determina que el usuario ha solicitado la intervención del agente. Esta solicitud tiene un **efecto de orden** que el usuario hace saber al agente para que éste último le ayude a resolver un problema. Este efecto de orden se convierte, entonces, en la meta que el agente debe cumplir; cumplimiento que dependerá, en la mayoría de los casos, de un conjunto de sub-metas de acuerdo a un tipo de plan específico. Este conjunto de sub-metas pueden verse como un conjunto de condiciones o **efectos de control** que deben ser verdaderos para que el agente pueda cumplir con el plan propuesto. Así, se puede discriminar las observaciones que deben ser explícitamente escritas en la base de conocimientos del agente. Los efectos de control son todas aquellas acciones que el agente debe hacer cumplir para satisfacer uno o más efectos de orden. Estos efectos de control son el resultado de un conjunto acciones ocultas (al usuario) del agente que generan nuevas observaciones independientes del mensaje del usuario, las cuales, conjugadas, permiten definir si es posible o no hacer cumplir un plan de acción.

Esta clasificación de las observaciones, ayuda a separar los servicios que un usuario de la aplicación puede solicitar y, en función a ello, determinar qué tipo de servicios el agente debe prestar para cooperar con el usuario en la resolución de sus tareas.

Una vez obtenido este conjunto de servicios, el diseñador de la base de conocimiento debe determinar qué tipos de **conducta** interviene en el cumplimiento de ese servicio, y con ello construir las reglas de integridad y los planes de acción para el agente. El conjunto de servicios equivalen al conjunto de efectos de orden (observaciones) en la base de conocimiento, y el resultado de una conducta particular equivale a los efectos de control (otras observaciones) en la base de conocimiento. Las reglas de integridad son condiciones de causa y efecto que determinan la conducta del agente y deben ser escritas como reglas de condición y acción.

Con esto es posible completar la definición de las observaciones, la construcción de las reglas de integridad y la generación de los planes de acción del agente.

El conjunto de acciones ejecutables del agente, son obtenidas de dos fuentes dependiendo del tipo de plan al que se haga referencia. Existen dos tipos de planes: **planes simples** y **planes compuestos**.

Un plan compuesto es la referencia a un conjunto de acciones ejecutables por el agente. Este tipo de planes tienen un nombre y son explícitamente estructurados como la conjunción de dos o más acciones que el agente debe realizar para cumplirlos.

Un plan simple es la referencia a una sola acción ejecutable del agente, obtenida del cuerpo de las reglas de integridad y que no hacen referencia a un plan compuesto.

De esta manera, el conjunto de acciones ejecutables del agente son aquellas que pertenecen a los planes simples y aquellas que componen a un plan compuesto.

5.5.2. Integración del agente a Bioinformantes

Como se explicó en el capítulo 2, Bioinformantes[3] posee un conjunto de componentes Web que pueden ser utilizados como una aplicación cooperativa en producción, y que puede incorporar un tipo de agente interfaz como el propuesto en el capítulo 4. Para ello, es necesario determinar qué componentes de la aplicación pueden ser utilizados como interfaz entre el usuario, la aplicación y el agente; que técnica (s) será (n) utilizada (s) por el usuario para comunicarse con ese agente; y qué tipo de conocimiento debe incorporar el agente para formar parte de la aplicación.

En la sección 2.2.2, se detalló el tipo de interfaz gráfica de usuario que ofrece Bioinformantes para utilizar la aplicación, la cual está compuesta por un applet de

Java que funciona como una sala de chat y una seudoterminal para ejecutar comandos y programas del lado del servidor.

De acuerdo a las especificaciones de Bioinformantes, descritas en la sección 2.1, es posible la incorporación de un agente interfaz que pueda servir como asistente y consultor de usuarios sobre las herramientas que incorpora esta aplicación. Igualmente, se especifica que este agente puede recibir órdenes o solicitudes del usuario haciendo uso de la interfaz gráfica presentada en la sección 2.2.2, las cuales son discriminadas de acuerdo al flujo de información descrito en la sección 2.2.3.

Este flujo de mensajes, incorpora, además, el formato de los mensajes que puede escribir un usuario, dirigido al agente de la aplicación. En la tabla 2.2.1 se muestra un ejemplo de mensaje dirigido al agente, al cual, en Bioinformantes se le denomina *BioTutor*.

Como se puede notar, el mensaje está escrito en lenguaje natural, dirigido al agente BioTutor, pero debe ser antes recibido y procesado por un servlet, quién debe agrupar, en una estructura de datos, las observaciones para el agente tal como se encuentran en su base de conocimientos. Una vez realizado este procesamiento se debe invocar al agente de acuerdo a los métodos descritos en su interfaz presentada en el capítulo 4 y sección 5.3. El resultado de la ejecución de las acciones del agente deben ser entregados al servlet para su procesamiento y presentación. Esto último se logra haciendo uso de las interfaces del agente tal como se explica en la sección 5.3.

Al servlet que conjuga estas propiedades en un componente Web desarrollado y desplegado en Bioinformantes, debe agregársele las líneas de código necesaria para que pueda lograr una comunicación bi-direccional con el agente.

Igualmente, la plataforma debe incorporar un modulo de procesamiento de lenguaje natural que pueda determinar y agrupar las acciones del agente tal como

se encuentran escritas en la base de conocimientos. Si bien esto no es una tarea sencilla, es importante destacar que el marco de desarrollo descrito en la sección 5.5.1 puede servir como base para el diseño de este módulo, pues existe una relación estrecha entre el mensaje del usuario y los efectos de orden del agente.

Por otro lado, es importante destacar que el agente descrito en el capítulo 4 está programado para un dominio de conocimiento específico, orientado a la simulación de modelos Galatea [2]. Este agente puede ser integrado a la plataforma de Bioinformantes sin realizar cambios en la estructura interna del agente, pues esta aplicación incorpora a Galatea como una herramienta más disponible a sus usuarios.

Sin embargo, éste agente estaría limitado a servir como asistente y consultor de la aplicación en el dominio de uso de Galatea como leguaje de simulación. Para agregar nuevas funcionalidades al agente, es necesario re-programar la base de conocimientos del agente de acuerdo al marco de desarrollo descrito en la sección 5.5.1, y re-compilar, esa estructura, para que la aplicación disponga de estos nuevos servicios por parte del agente.

Otra forma de lograr esto y, tal vez una manera más organizada, es implementando una base de conocimiento para cada dominio de aplicación del agente y utilizar las mismas interfaces para la comunicación. En este caso, es importante que el servlet que recibe el mensaje del usuario pueda discriminar entre las bases de conocimientos que tiene disponibles y, en función a ello, solicitar la intervención del agente.

En cuanto a la instalación y despliegue de Bioinformantes y, en contraste con el despliegue de la aplicación mostrada en la sección 5.4, Bioinformantes incorpora no solo un componente Web para la comunicación con el agente, sino también un conjunto de éstos componentes para su funcionalidad específica. Todos los componentes de Bioinformantes deben ser desplegados en el servidor de

aplicaciones según lo descrito en la sección 5.4.2 y la integración del agente como un componente EJB debe llevarse a cabo según lo descrito en esa misma sección.

Con este capítulo se completa la descripción de nuestro agente interfaz y su especificación como un componente de software reutilizable, que posee características de adaptabilidad en aplicaciones de cualquier dominio, orientadas al procesamiento de datos en entornos cooperativos.

Conclusiones

En este documento hemos presentado la estructura general y los requerimientos de implementación de un tipo de agente de software basado en lógica extendido como un componente de software reutilizable. Este tipo de agente, llamado SIMULANTE, está basado las especificaciones del modelo de agente GLORIA [5] y en su implementación en Java, con un dominio de conocimiento orientado a la simulación de modelos escritos en GALATEA.

Nuestra extensión de esa especificación utiliza la tecnología Enterprise Java Beans de la plataforma Java J2EE. Nuestro agente es caracterizado como un componente reutilizable del tipo sesión sin estado. Esta implementación permite que el agente pueda funcionar de manera independiente en una aplicación Web. Cada usuario de la aplicación puede obtener una referencia remota de un agente (instancia del EJB) e interactuar con él de tal manera que le sirva como asistente o consultor en la aplicación.

Por otro lado, nuestro modelo de agente puede ser adaptado a cualquier dominio de aplicación con solo caracterizar una nueva base de conocimiento utilizando un marco de desarrollo. La idea es que SIMULANTE cumpla con el objetivo de validar la arquitectura propuesta a la vez estimule el desarrollo de aplicaciones Web orientadas al trabajo cooperativo y soportadas por agentes inteligentes.

Trabajos futuros

Para complementar el desarrollo de este modelo de agente interfaz como componente de software reutilizable, los trabajos inminentes son:

- Implementar un módulo para el manejo de los perfiles de usuarios que haga posible que el agente pueda emprender planes de tutoría acordes con el perfil de cada uno de los usuarios.

- Agregar al modelo de aplicación del agente una capa de datos que permita el manejo de la persistencia de los datos de la aplicación. Esto permitiría que el agente pueda retomar planes inconclusos que por algún motivo no pudo terminar en espera de algún evento. Con ello se busca que el agente pueda ser desplegado en aplicaciones sicrónicas o asíscrónicas. Este modelo de datos permitiría, además, implementar planes de tutoría para la enseñanza de algún tópico particular. Esto último de acuerdo a un número de clases que deben ser programas y retomadas de acuerdo a niveles de conocimiento y aprendizaje de los usuarios.

- Separar la interfaz agente-ambiente de la implementación del agente. Esto daría flexibilidad para redefinición de bases de conocimientos y la creación de otras para nuevas aplicaciones.

- Diseñar e implementar un módulo de procesamiento de lenguaje natural que pueda ser fácilmente integrado a una aplicación cooperativa y aplicada al modelo de agente propuesto.

Referencias

[1] Stuart Russell, Meter Norvig. Inteligencia Artificial: un enfoque moderno. Prentice Hall, 1995.

[2] Mayerlin Uzcátegui. Diseño de la plataforma de simulación de sistemas multiagentes: Galatea. Tesis de Maestría, 2002.

[3] Jacinto Dávila, José López, Almathely Vivas. Bioinformants: BIOLogical, INFORMAtional ageNTS on the Internet. White paper, 2003.

[4] Jacinto Dávila, Mayerlin Uzcátegui. Agents' executable specifications. White paper, 2002.

[5] Jacinto Dávila. Agents in Logic Programming. Tesis PhD, 1997.

[7] Alexander Barrios. Agentes de Software como Componentes de Software Reutilizable para Computación Científica. Tesis de Maestría. Universidad de Los Andes. Mérida-Venezuela, 2002.

[8] Software Engineering with Reusable Components. J. Samentinger. Editorial Springer. 1997.

[9] Montilva Jonás. Notas del curso: Ingeniería de Componentes Reulilizables e Integración de Software. Postgrado en Computación, Universidad de Los Andes. Mérida – Venezuela. 2002.

[10] Iglesias Fernández, Carlos Ángel. Definición de una Metodología para el desarrollo de Sistemas Multiagente. Tesis Doctoral. Universidad Politécnica de Madrid, Enero 1998.

[11] Botti, V.; Carrascosa, C.; Julian, V.; Soler, J. (1999). The ARTIS Agent

Architecture: Modelling Agents in Hard Real-Time Environments. Valencia: Springer-Verlag (ed.). Proceedings of the MAAMAW'99. LNCS, 1647, p. 63-76. Apéndices

[12] Art Taylor; Brian Buege; Randy Layman. Hackers de Java y J2EE. McGrawHill. 2002.

[13] Ana García Serrano; Josefa Z. Hernández; Paloma Martínez. Dialogos en lenguaje natural con asistentes virtuales. Universidad Politécnica de Madrid. White paper, 2003.

[14] Pedro Javier Espeso Martínez. Diseño de componentes de software en tiempo real. Trabajo de grado. Universidad de Cantabria. 2002.

[15] Sergio Bermejo; Aniceto Saboya. TUTORES INTELIGENTES BASADOS EN ASISTENTES PERSONALES. Universidad Politécnica de Cataluña. White paper, 2003.

[16] Rosenberg, M. J. (2000). E-Learning: Strategies for Delivering Knowledge in the Digital Age.New York: McGraw-Hill Professional Publishing.

[17] Alejandro Guerra Hernández. Agentes Interfaz Inteligentes. Universidad Veracruzana - LANIA, A.C.

[18] Ulf Nilsson and Jan Ma luszy nski. LOGIC, PROGRAMMING AND PROLOG (2ED). Noviembre de 2000.

[19] Luis Joyanes Aguilar, Matilde Fernández Azuela. JAVA 2 Manual de Programación. McGrawHill. Universidad Pontificia de Salamanca. Madrid-España, 2001.

[20] Pérez Lezama. Agentes móviles en bibliotecas digitales. Tesis de Maestría. Universidad de Las Américas-Puebla. Mayo 1998.

[21] Daniel Perovich, Leonardo Rodríguez, Andres Vignaga. Arquitecturas de Sistemas de Información basados en Componentes sobre la Plataforma J2EE. Universidad de la República Julio Herrera y Reissig. Uruguay 2002.

[22] The J2EE Tutorial for J2EE 1.3 (http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html).

[23] Enterprise JavaBeans Specification 2.0 (http://java.sun.com/products/ejb/docs.html#specs).

[24] Richard Monson-Haefel: Enterprise JavaBeans, 3rd Edition. Ed. O'Reilly,September 2001.

[25] Servidores de aplicaciones e integración. Productos y Servicios de Sun. http://es.sun.com/tecnologia/software/servidores_aplicacion/servidores/

[26] http://www.jakarta.apache.org/tomcat

[27] http://www.iec.csic.es/criptonomicon/comercio/ssl.html

Glosario

Agente

Entidad que puede percibir su ambiente, asimilar dichas percepciones y almacenarlas en un dispositivo de memoria, razonar sobre la información almacenada en dicho dispositivo y adoptar creencias, metas e intenciones por sí mismo dedicándose al alcance de dichas intenciones a través del control apropiado de sus efectores.

Agente interfaz

Programa que utiliza técnicas de inteligencia artificial para proveer asistencia al usuario de alguna aplicación de cómputo. El agente busca continuamente la oportunidad de realizar tareas que ayuden a la satisfacción de los objetivos del usuario. Agente y usuario establecen una especie de contrato donde el usuario da a conocer sus objetivos y el agente ofrece sus capacidades para el logro de estos objetivos.

Ambiente

Entorno en donde se desenvuelven los agentes que se representa como un conjunto de estados posibles del mundo. El ambiente incluye todo lo que no forma parte de un agente dado, incluido otros agentes.

API (Application Programming Interface)

Interface para Programación de Aplicaciones. Se refiere al conjunto de librerías (procedimientos, módulos, clases, objetos, etc) que tiene disponible un lenguaje de programación para la construcción y desarrollo de aplicaciones.

Aplicación Web

Sistema de información donde una gran cantidad de datos volátiles, altamente estructurados, son consultados, procesados y actualizados mediante navegadores.

Aplicación Web cooperativa

Aplicación Web diseñada para asistir a un grupo de personas dedicadas a una tarea común, cada una de ellas trabajando en su propia computadora, pero intercambiando datos o programas a través de una interfaz Web.

Aplicación Web distribuida

Tipo de aplicación Web que utiliza o accede a recursos de varios sistemas distribuidos en múltiples computadoras de una red. Las aplicaciones distribuidas se implementan típicamente como sistemas cliente/servidor organizados en conformidad con la interfaz del usuario, el procesamiento de la información y las capas de procesamiento de la información.

Base de conocimientos

Una estructura de datos que contiene reglas y heurísticas específicas a cierta área

del conocimiento.

BIOINFORMANTES

Aplicación Web cooperativa para el procesamiento de data científica, específicamente, biológica.

Componente

Pieza identificable de software que describe o libera un conjunto manejable de servicios que sólo pueden ser usados a través de una interfaz bien definida.

Componentes de Software Reutilizables (CSR)

Unidad de software auto contenida, modular y reemplazable que encapsula su implementación, hace visible su funcionalidad a través de un conjunto de interfaces, se integra con otro componente mediante sus interfaces para formar un sistema u otro componente mayor, puede ser desplegado (instalado y ejecutado) independientemente de los otros componentes en una aplicación.

Contenedor

Ambiente de ejecución que provee la conectividad con la red, la gerencia del ciclo de vida de las componentes, seguridad, y servicios especiales como transaccionalidad, persistencia, pooling, etc. En general los contenedores proveen un ambiente estándar de ejecución para los componentes, una vista federada de las APIs para los componentes y servicios específicos para los componentes. Los contenedores interactúan con las componentes invocando a los métodos call-back. Estos métodos definen la interface entre el Contenedor y las componentes.

Cooperación

Proceso de obrar o ejecutar ciertas tareas juntamente con otro u otros para un mismo fin.

Creencias

Un concepto que describe la información que el agente posee del ambiente, que no necesariamente es cierta.

Descriptores de despliegue

Son archivos de configuración XML que contienen toda la información necesaria para desplegar los componentes de un módulo así como las instrucciones para combinar los diversos componentes dentro de una aplicación.

Despliegue

Es una de las etapas finales del desarrollo de software. En ella, se generan todos los archivos que se necesitan para ejecutar una aplicación y pasar del entorno de desarrollo al entorno de producción. El resultado de la etapa de despliegue es una aplicación ejecutable ubicada en un entorno de producción.

Dominio de conocimiento

Conocimiento requerido para llevar a cabo los procesos que apoyan una estrategia especifica de acción sobre un área de aplicación.

Enterprise Java Beans (EJB)

Componente distribuido que se ejecuta dentro de un contenedor que proporciona un servidor de aplicaciones. El EJB es el componente distribuido básico de J2EE.

GALATEA (GLIDER with Autonomous, Logic-based Agents, TEmporal reasoning and Abduction)

Plataforma de simulación multiagente.

GLORIA (General-purpose, Logic-based, Open, Reactive and Intelligent Agent)

Descripción lógica de un tipo de agente de propósito general que es, tanto reactivo como racional, de acuerdo a las circunstancias de operación.

Influencias

Representan las intenciones de acción de cada agente, con las que intenta modificar la historia si se convierten en acciones.

Ingeniería de conocimiento

Proceso de adquirir el conocimiento del área determinada y estructurarlo en una base de conocimientos

Ingeniería del software

Disciplina tecnológica y administrativa dedicada al tratamiento sistemático de

todas las fases del ciclo de vida del software, aplicando principios de la ingeniería para obtener software de calidad.

Inteligencia Artificial (AI)

Ciencia e ingeniería de los sistemas inteligentes (artificiales principalmente), enfocado en el estudio sistemático del comportamiento inteligente y de los procesos de aprendizaje de los seres humanos con la finalidad de que las máquinas y computadoras imiten las habilidades humanas.

Inteligencia Artificial Distribuida

Rama de la AI que se centra en comportamientos inteligentes colectivos que son producto de la cooperación de diversos agentes". Estos agentes, son las entidades que colaboran.

JAVA

Lenguaje de programación orientado a objetos diseñado para generar aplicaciones que puedan ejecutarse en cualquier plataforma de hardware, ya sea pequeña, mediana o grande, sin modificaciones. Desarrollado por Sun, se ha promovido y fomentado el uso de Java para la Web, tanto para sites públicos como para intranets.

Java 2 Enterprise Edition (J2EE)

Conjunto de especificaciones de APIs Java para la construcción de aplicaciones empresariales.

Metas, objetivos, propósitos

Un concepto que describe la predisposición del agente en ver realizadas ciertos deseos. Las metas deben ser mutuamente consistentes.

Modelado

Proceso de construir modelos que permiten describir un sistema.

Planificación

El proceso por medio del cual un agente plantea la solución a un problema. Normalmente equivale a derivar los pasos y sub-tareas que deberán cumplir uno o varios agentes, para alcanzar la consecución de una tarea mayor o un estado deseado.

Planificación abductiva

El uso del razonamiento hipotético y la abducción para generar planes dirigidos a alcanzar ciertas metas pre-establecidas.

Preferencias

Un conjunto de heurísticas que sesgan las decisiones de un agente en dirección a ciertos conjuntos de acciones. Le permiten al agente incorporar criterios de urgencia e importancia en el proceso de planificación.

PROLOG

Lenguaje de programación hecho para representar y utilizar el conocimiento que se tiene sobre un determinado dominio. Más exactamente, el dominio es un conjunto de objetos y el conocimiento se representa por un conjunto de relaciones que describen las propiedades de los objetos y sus interrelaciones. Un conjunto de reglas que describa estas propiedades y estas relaciones es un programa Prolog. Prolog es un lenguaje de programación que es usado para resolver problemas que envuelven objetos y las relaciones entre ellos.

Secure Sockets Layer (SSL)

El protocolo SSL permite la autenticación de servidores, la codificación de datos y la integridad de los mensajes. Con SSL tanto en el cliente como en el servidor, sus comunicaciones en Internet serán transmitidas en formato codificado. De esta manera, puede confiar en que la información que envíe llegará de manera privada y no adulterada al servidor que usted especifique. Los servidores seguros suministran la autenticación del servidor empleando certificados digitales firmados emitidos por organizaciones llamadas "Autoridades del certificado". Un certificado digital verifica la conexión entre la clave de un servidor público y la identificación del servidor. Las verificaciones criptográficas, mediante firmas digitales, garantizan que la información dentro del certificado sea de confianza.

Semántica

Las estructuras que otorgan significado a un lenguaje.

Servidor de aplicaciones

Servidor que incluye un servidor HTTP (también llamado servidor Web); suele estar diseñado para que accedan a él los clientes HTTP y para alojar lógica de presentación y lógica empresarial.

Simulación

Proceso de diseño de un modelo lógico o matemático de un sistema real y realización de experimentos basados en la computadora con el modelo, al objeto de describir, explicar o predecir el comportamiento del sistema real

SIMULANTE

Agente de software caracterizado como asistente y tutor de una aplicación Web orientada al modelado y simulación de sistemas, cuyo diseño y prototipo funcional se describen en este documento.

TOMCAT

Contenedor de Servlets con un entorno JSP. Un contenedor de Servlets es un shell de ejecución que maneja e invoca servlets por cuenta del usuario.

Tutores Inteligentes (ITs)

Sistema de enseñanza asistida por computadora, que utiliza técnicas de Inteligencia Artificial, principalmente para representar el conocimiento y dirigir una estrategia de enseñanza; y es capaz de comportarse como un experto, tanto en el dominio de conocimiento que enseña (mostrando al alumno cómo aplicar dicho

conocimiento), como en el dominio pedagógico, donde es capaz de diagnosticar la situación en la que se encuentra el estudiante y de acuerdo a ello ofrecer una acción o solución que le permita progresar en el aprendizaje.