Análisis Comparativo de los Frameworks FraMaS y Brainstorm/J

Martín Valacco, Analía Amandi

ISISTAN - Depto. Comp. y Sistemas, Fac. de Cs. Exactas - UNICEN
Tandil - Bs. As., Argentina
{mvalacco, amandi}@exa.unicen.edu.ar

Resumen
Los frameworks FraMas y Brainstorm/J han sido desarrollados para soportar el desarrollo de sistemas multi-agentes. Estos frameworks presentan varias diferencias en relación con su concepción y técnicas de diseño utilizadas. Estas diferencias deben ser contempladas por los desarrolladores de aplicaciones de sistemas multi-agentes para así aprovechar las características presentes en cada uno de ellos. Este artículo presenta una comparación entre estos frameworks con el objetivo de que desarrolladores de aplicaciones puedan realizar su mejor opción. Particularmente, se analiza los puntos de reutilización y las limitaciones resultantes de la utilización de diferentes técnicas de diseño.

Palabras clave: frameworks, agentes.

1-Introducción

Un agente inteligente es una entidad de software que evoluciona en un entorno, es capaz de percibir dicho entorno, actúa en él, se comunica con otros agentes y exhibe un comportamiento autónomo. Un sistema multi-agente es un conjunto de agentes posiblemente organizados que interactúan en un entorno común [Demazeau 90].

En los últimos años, el desarrollo de aplicaciones que involucran sistemas multi-agentes ha sido motivo de numerosas investigaciones. La complejidad de desarrollo de estas aplicaciones desde cero hace necesaria la creación de componentes genéricos de software que puedan ser reutilizados en diferentes aplicaciones. Sin embargo, las aplicaciones de este dominio son muy variadas e incluyen asistentes personales, agentes para filtrado de información, tutores inteligentes, buscadores y robots entre otras. Aún con esta gran variedad de aplicaciones, existen varias características comunes a todos los sistemas multi-agente, como percepción del entorno, comunicación entre agentes, selección de acciones, aprendizaje y movilidad que pudieron ser abstraídas para construir herramientas genéricas de software que puedan ser instanciadas con los componentes concretos que requiere cada aplicación.

En este contexto, artefactos especializados para el desarrollo de agentes fueron propuestos [Gutknecht 97], [Chauham 98] [Reticular 99] [Avancini 00] [Zunino 00]. Estos artefactos pueden ser clasificados como herramientas cerradas, herramientas basadas en composición de componentes y frameworks. Los dos primeros tipos de herramientas proveen funcionalidad fija que puede ser combinada dentro de los caminos explícitamente establecidos. Los frameworks permiten no sólo el desarrollo de agentes a partir de los componentes disponibles sino la adaptación de código, generalmente por especialización, para moldearlo a diferentes dominios.

Un framework orientado a objetos es un conjunto de clases, en general abstractas, que conforman un diseño reusable para un dominio particular. Un framework constituye un esqueleto para un conjunto de aplicaciones de dicho dominio. El usuario del framework deberá crear subclases que implementen la funcionalidad propia de la aplicación que se quiere desarrollar. Dentro de esta tercera categoría de clasificación, algunos frameworks fueron realizados. Específicamente, dos frameworks desarrollados en el Instituto de Sistemas ISISTAN de la Universidad Nacional del Centro serán analizados en este artículo. Estos dos frameworks tienen por

objetivo soportar la construcción de sistemas multi-agente, y fueron denominados FraMaS [Avancini 00] y Brainstorm/J [Zunino 00].

En este artículo se analiza y compara las características de los frameworks para sistemas multi-agente FraMas y Brainstorm/J, particularmente aspectos relevantes de diseño y del tipo de aplicación para el que resulta conveniente la utilización de cada uno de ellos. Estos frameworks han sido implementados en el lenguaje Java.

A pesar de que ambos frameworks han sido desarrollados para el mismo dominio de aplicación (sistemas multi-agente) existen notorias diferencias entre ellos, desde la metodología de desarrollo hasta el diseño y el tipo de agentes hacia el que han sido orientados.

El propósito de este artículo es analizar los puntos de reuso de los frameworks para captar la real funcionalidad y limitaciones de cada uno de ellos.

A continuación se describe como se encuentra organizado el resto del artículo. En la sección 2 se describe con mayor detalle las características de los frameworks orientados a objetos así como los beneficios de utilizar este tipo de tecnología. La sección 3 abarca la descripción de los frameworks mencionados y la comparación de los mismos. Finalmente, en la sección 4 se presentan las conclusiones de los estudios y comparaciones realizadas.

2- **Frameworks orientados a objetos**

Un framework es un conjunto de clases, en general abstractas, que conforman un diseño reusable para un dominio particular. Un framework constituye un esqueleto para un conjunto de aplicaciones de dicho dominio.

Las clases abstractas del framework poseen un número de métodos abstractos que pueden verse como huecos que el usuario deberá rellenar creando subclases concretas que implementen estos métodos con la funcionalidad específica de cada aplicación.

La figura 2.1 muestra como ejemplo la clase abstracta *Figura* y las subclases de esta. Es claro que el método *dibujar* es abstracto en la clase *Figura* ya que su implementación depende fuertemente del tipo de figura (cuadrado, círculo, etc).

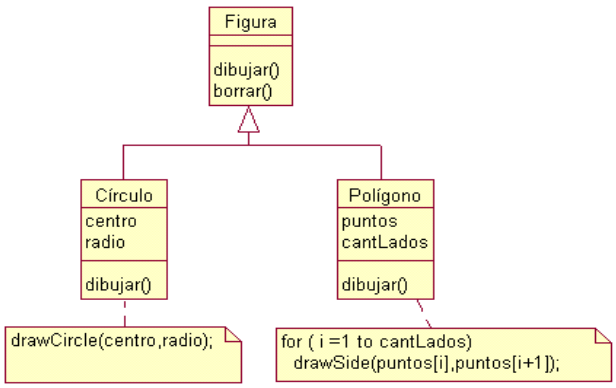

Figura 2.1- Ejemplo de método abstracto.

Además de los métodos abstractos, existen otros mecanismos muy importantes de la programación orientada a objetos que se utilizan en la construcción de frameworks. Estos mecanismos son: los métodos hook, template y base.

Los métodos hook son similares a los métodos abstractos con la diferencia que definen un comportamiento por defecto. Así, el comportamiento especificado por un método hook podrá ser utilizado directamente o redefinido por medio de herencia.

El flujo de control del framework es especificado por los métodos template. Un método template define comportamiento común al dominio de aplicación invocando a uno o más métodos

abstractos. El comportamiento específico de cada aplicación se implementa redefiniendo dichos métodos abstractos. El comportamiento final de un método template varía en función de la implementación de los métodos abstractos que éste invoca.

En la figura 2.2 se puede observar un ejemplo de método template en la clase abstracta *Compiler*. El método *compile* está completamente implementado ya que el algoritmo de compilación es común a todos los compiladores que son subclases de *Compiler*, sin embargo este método utiliza llamados a los métodos abstractos *lexical*, *parse* y *generateCode* que son dependientes del tipo de compilador y, por lo tanto, son implementados en las subclases.

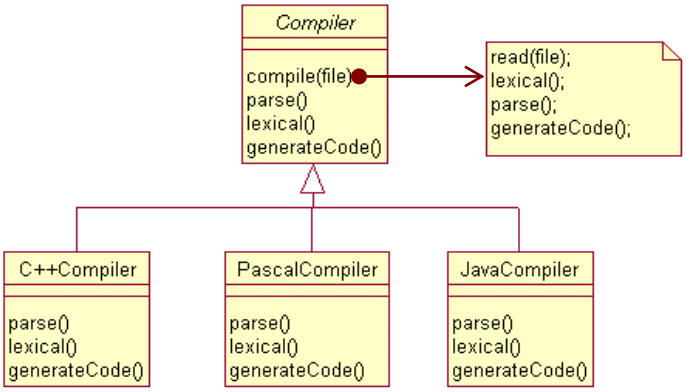

Figura 2.2- Ejemplo de método "template".

Finalmente, los métodos base proveen una implementación completa de un comportamiento común a cualquier aplicación del dominio. A diferencia de los métodos hook, los métodos base no deberían ser redefinidos por el programador.

El énfasis del desarrollo de frameworks está puesto en el reuso de diseño, no sólo en el reuso de código. Un framework captura las decisiones de diseño comunes del dominio de aplicación para el que fue construido. Tomando las decisiones de diseño por el desarrollador, este se puede concentrar en los detalles específicos de la misma.

El objetivo de los frameworks no solo es el de acelerar los desarrollos de aplicaciones. Las aplicaciones construidas con un mismo framework son consistentes y fáciles de mantener.

Por otra parte, como en los frameworks se especifican la mayoría de las decisiones acerca del dominio, buenos frameworks son muy difíciles de desarrollar. Un framework debe ser lo suficientemente simple como para ser aprendido. Además, un framework debe ser lo suficientemente flexible para satisfacer las demandas de nuevas aplicaciones. Un diseño debe ser reusado en varias ocasiones antes de que este pueda ser llamado verdaderamente un framework.

Es casi siempre imposible predecir todas las necesidades de las aplicaciones futuras. Como resultado, la construcción de un framework es un proceso iterativo. El framework evoluciona cuando se construyen nuevas aplicaciones y las debilidades de este son descubiertas y solucionadas.

Para diseñar un framework pueden adoptarse dos enfoques diferentes: conducido por ejemplos o conducido por un modelo de dominio de aplicación.

El primero consiste en abstraer las características comunes de un conjunto de aplicaciones pertenecientes al mismo dominio. La selección de un conjunto de aplicaciones representativas del dominio de aplicación es de fundamental importancia para que el framework posea abstracciones comunes a dicho dominio.

El segundo enfoque tiene sus orígenes en el concepto de arquitectura de software y estilos arquitectónicos. Así, otra manera de desarrollar un framework consiste en partir de un modelo de dominio abstracto y uno o más estilos arquitectónicos adecuados, de lo cual se obtiene una primer materialización arquitectónica. Luego, se refina y materializa la arquitectura mediante un conjunto

de clases que conforman el framework. La aplicación de patrones de diseño durante esta etapa puede mejorar la flexibilidad y reusabilidad del framework.

3- **FraMas y Brainstorm/J, descripción y comparación**

En esta sección se analizan dos frameworks para el desarrollo de sistemas multi-agente. Estos frameworks son conocidos como FraMas y Brainstorm/J y se describe cada uno de ellos junto con sus similitudes y diferencias. Para una mejor organización de la sección, se dividió a ésta en subsecciones, las cuales se detallan a continuación. En las secciones 3.1 y 3.2 se describen respectivamente FraMas y Brainstorm/J de manera individual, destacando las características propias de cada uno de ellos. Finalmente en la sección 3.3 se comparan estos frameworks componente por componente.

3.1- FraMas

FraMas [Avancini 00] ha sido diseñado con el objeto de construir sistemas multi-agente sin tener que empezar a programarlos desde cero. Para el desarrollo de FraMas se utilizó un enfoque bottom-up, es decir que primero se construyeron aplicaciones de agentes particulares y se fue abstrayendo comportamiento común de estas para generar las clases del framework.

La metodología de diseño antes descripta es la razón por la cual el framework no está basado en ninguna arquitectura de diseño particular, sino que consiste en un sistema orientado a objetos donde los problemas de diseño se solucionaron utilizando patrones.

En FraMas, un agente posee un comportamiento básico al cual se le agrega comportamiento inteligente. Dicho comportamiento es el que define al agente como inteligente. En este caso, el comportamiento inteligente de los agentes corresponde a la comunicación, selección de la próxima acción y aprendizaje de las preferencias del usuario. En este framework, dicho comportamiento inteligente se agrega por medio de decoradores o wrappers. Esta técnica está definida por el patrón de diseño “Decorator” [Gamma 95]tambien llamado “Wrapper”.

El primero de los wrappers que agrega comportamiento es el wrapper de comunicación que permite establecer una comunicación directa entre los agentes, sin necesidad de utilizar el SMA. Esta comunicación ha sido implementada mediante la invocación de métodos remotos (RMI) que provee el lenguaje Java.

El segundo wrapper es el analizador de preferencias el cual permite que el agente adquiera durante un tiempo de entrenamiento el conocimiento necesario para poder asistir al usuario.

La técnica de aprendizaje utilizada en el framework es la de razonamiento basado en casos (CBR). Esta técnica se fundamenta en que las situaciones ocurren con regularidad por lo que una acción exitosa llevada a cabo con anterioridad puede ser aplicado en una situación actual si esta es similar a aquella [Kolodner 97].

De la misma manera si en el pasado se cometió algún error, este se recuerda para no volver a cometerlo en una situación similar.

Finalmente, la razón por la que se dice que los agentes inteligentes son entidades autónomas es porque son capaces de decidir que tareas realizar y realizarlas sin que el usuario intervenga de manera directa. En FraMas, para seleccionar la acción a ejecutar, pueden utilizarse distintas técnicas tales como planning, redes bayesianas, etc. Con el objeto de dar flexibilidad a la elección de la técnica a utilizar, se aplicó el patrón de diseño *Strategy* [Gamma 95] que define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables.

En la figura 3.1 se ve un ejemplo de la interfaz de dos clases que representarán técnicas de planning y redes bayesianas. Estas clases no son partes del framework sino que se muestran como ejemplo de una posible instanciación del mismo.

FraMas define a los SMA como un conjunto de “entornos”. Un entorno pertenece a un determinado host, sin embargo, en un mismo host puede haber más de un entorno. En cada entorno habitan agentes que interactuan con el entorno y con el resto de los agentes inmersos en este.

Cada agente tiene asociado un contexto el cual es capaz de percibir para detectar situaciones de interés. Además, los agentes son capaces de comunicarse directamente con otros agentes, ya sea dentro del entorno o fuera de este.

Una característica importante de los agentes en FraMas es que estos pueden trasladarse de un entorno a otro con el objeto de interactuar con los agentes del entorno destino sin necesidad de intercambiar mensajes a través de la red, con el insumo de ancho de banda que esto implica cuando la interacción es grande. Para esto el framework provee un mecanismo de movilidad que permite que un agente detenga su ejecución y salve su estado para luego proceder a la transferencia.

Los SMA de FraMas proveen servicios a los agentes: para registrarse y salir del entorno, de movilidad, para que un agente se publique a fin de hacerse visible a los otros agentes en el entorno y el protocolo para la producción y percepción de eventos.

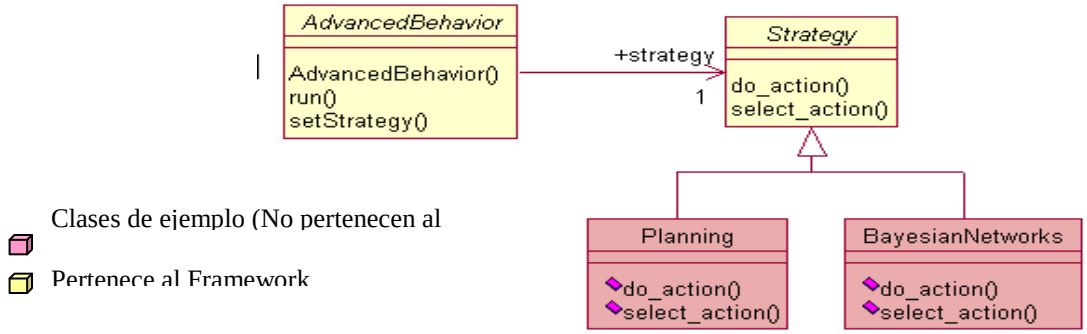

Figura 3.1 – Aplicación del patrón “Strategy” para la selección de acción.

Por último, la interfaz al usuario del SMA también esta incorporada al framework para permitir realizar un seguimiento de los agentes en el entorno aunque de una manera muy abstracta ya que los tipos de SMA y los agentes en si son muy diversos como para crear interfaces genéricas.

3.2- Brainstorm/J

Brainstorm/J [Zunino 00] es un framework creado con el objeto de construir aplicaciones de agentes inteligentes. Este framework se construyó materializando una arquitectura de software denominada Brainstorm [Amandi 97] que prescribe agentes con capacidades de percepción, comunicación, manejo de estado mental, reacción, deliberación y aprendizaje.

Brainstorm es una arquitectura que fue desarrollada con el objetivo de extender objetos para que soporte el comportamiento de agentes. En el contexto de Brainstorm, un sistema multi-agente se define como un sistema reflexivo, el cual está conectado a un sistema orientado a objetos en el nivel base. La arquitectura se ha construido utilizando meta-objetos los cuales, que junto al objeto base, definen un agente. Cada meta-objeto incorpora funcionalidad de agente a un objeto simple.

El nivel reflexivo de la arquitectura se divide a su vez en varios meta-niveles que se pueden observar en cada agente. Cada nivel esta conectado con el nivel inferior a través de reflexión.

El primer meta nivel está compuesto por los meta-objetos de comunicación, percepción y estado mental del agente. El meta-objeto de comunicación permite al agente enviar y reconocer mensajes de otros agentes en un lenguaje específico de comunicación. Por su parte la percepción permite reconocer situaciones en el ambiente que puedan tener algún tipo de interés para el agente. En cuanto al estado mental, este es el encargado de manejar los conocimientos, creencias, objetivos, intenciones, etc., que posee el agente en determinado momento. Además, el primer meta-nivel posee un objeto, el Situation Manager, que se encarga de observar las comunicaciones,

percepciones y el estado mental y decide cuando se producen situaciones de interés para el agente y de que tipo de situaciones se trata.

En el segundo meta-nivel se encuentra toda la responsabilidad de decidir las acciones a llevar a cabo en el futuro próximo. Básicamente estas acciones pueden ser decididas de dos maneras: reactiva y deliberativa o cognitiva. El meta-objeto reactivo decide de manera inmediata la acción a efectuar, a partir de una situación de interés observada por el Situation Manager. En cambio, en el meta-objeto cognitivo una situación puede activar mecanismos de deliberación que producen decisiones menos inmediatas pero mucho más analizadas.

Existe otro meta-nivel compuesto por un meta-objeto llamado "Learner", el cual es responsable de observar las reacciones y deliberaciones, aprender de sus decisiones y usar estas experiencias en decisiones futuras.

En la figura 3.2 se muestra un diagrama de la arquitectura Brainstorm.

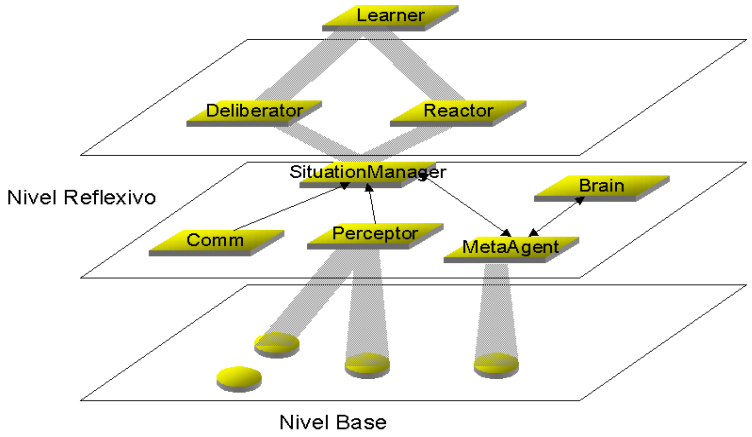

Figura 3.2- Estructura de la arquitectura "Brainstorm".

Los agentes construidos mediante los componentes abstractos que provee Brainstorm/J poseen las capacidades que se detallan a continuación.

- *Percepción*: los agentes pueden percibir otros objetos del entorno, ya sean objetos simples o agentes en si mismos. Esto lo hacen interceptando los mensajes que estos objetos reciben mediante el uso de meta-objetos reflexivos.
- *Comunicación*: los agentes creados con el framework pueden comunicarse entre si, ya sea a través del lenguaje KQML y el protocolo TCP/IP provistos por el framework, o bien mediante cualquier otro lenguaje y/o protocolo que se implemente extendiendo el componente de comunicación del framework. Además, el framework soporta conversaciones en lenguajes de alto nivel uniendo las capacidades de comunicación a las de deliberación.
- *Representación y manipulación de estado mental*: mediante el uso del lenguaje JavaLog [Amandi 99] que permite integrar cláusulas lógicas con Java, los agentes mantienen una representación del estado mental. Este estado esta definido por el modelo BDI (Belief Desire Intention).[Bratman 88] Además son capaces de actualizar este estado mental a medida que ejecutan acciones o perciben cambios en el ambiente, ya sea por la misma percepción o por alguna comunicación recibida. También se utiliza un mecanismo de revisión de creencias para mantener la coherencia del estado mental.
- *Administración de situaciones*: cuando se recibe una comunicación o se intercepta algún mensaje mediante la percepción entra en acción un componente del agente llamado administrador de situaciones. El administrador de situaciones analiza el evento ocurrido para

ver si este provoca alguna situación de interés para el agente. Si es así, comunica la situación al componente de reacción y/o deliberación.

- *Reacción*: si el agente ha sido definido como reactivo, el componente de reacción analiza las situaciones informadas por el administrador de situaciones e inmediatamente verifica las reacciones del agente. Si encuentra alguna cuyas condiciones se cumplen esta se ejecuta de inmediato.
- *Deliberación*: cuando el agente reconoce una situación de interés, este puede dar comienzo a un proceso deliberativo para decidir que acciones llevar a cabo en el futuro. El componente de deliberación se encarga de administrar los diferentes procesos deliberativos concurrentes que se producen en el agente. Cada proceso deliberativo utiliza alguna técnica, por ejemplo planning, para generar un plan consistente en una secuencia de acciones que el agente deberá ejecutar para intentar satisfacer sus objetivos. La generación de planes no es inmediata y el estado del mundo puede cambiar desde el momento del inicio al del final del plan. Para solucionar este problema, el framework soporta la reparación de planes, ya sea que estos estén en etapa de construcción o de ejecución.
- *Movilidad*: los agentes de Brainstorm/J soportan movilidad débil. Es decir, estos pueden trasladarse entre los diferentes hosts de la red aunque no pueden conservar algunos datos vitales de su estado de sus threads de ejecución, tales como el contador de programa y las variables de la pila. Por este motivo, antes de proceder al traslado, el agente deberá guardar estos datos para que puedan ser restaurados en el host de destino.

3.3- Comparación de los frameworks

En esta sección se analizan y comparan los frameworks FraMas y Brainstorm/J que fueron descriptos en las secciones anteriores. La comparación se divide como se detalla a continuación. En primer lugar se comparan las metodologías de desarrollo de ambos frameworks, esto se detalla en la sección 3.3.1. Luego se comparan los componentes del framework de la siguiente manera: en la sección 3.3.2 se compara el comportamiento básico de los agentes, en la sección 3.3.3 el componente de percepción, en la sección 3.3.4 el componente de comunicación, en la sección 3.3.5 la representación y manipulación de estados mentales, en la sección 3.3.6 el comportamiento reactivo, en la sección 3.3.7 el comportamiento deliberativo o cognitivo, en la sección 3.3.8 el componente de aprendizaje y en la sección 3.3.9 el componente de movilidad de agentes.

3.3.1- Metodología de desarrollo

La primer diferencia que presentan los frameworks está dada por el desarrollo de los mismos. Como se explicó en la sección 2, existen dos metodologías para construir frameworks: conducido por ejemplos y a partir de un modelo del dominio.

FraMas utilizó el primer enfoque para lo que se desarrolló una serie de aplicaciones de sistemas multi-agente para luego abstraer el comportamiento común de estas y proceder a la creación de las clases del framework. Para mejorar la flexibilidad se aplicaron patrones de diseño ante los diversos problemas que se presentaron.

Brainstorm/J, por su parte se basó en una arquitectura específica del dominio de agentes inteligentes llamada Brainstorm [Amandi 97]. Esto implica que no sólo se reutilizaron modelos y estilos arquitectónicos, sino que se aprovechó el resultado de un proceso de diseño.

Conclusión: FraMas solo utilizó algunos ejemplos de SMA para su desarrollo tales como un sistema de agentes de interfaz para una agenda que administra los compromisos del usuario y un sistema de robots que deben descargar un camión. Pero no ha sido probado para otros tipos de SMA. Brainstorm/J, en cambio, posee la madurez que le provee el uso de una arquitectura desarrollada específicamente para el dominio de los SMA más la utilización del framework en la implementación de ejemplos de SMA.

3.3.2- Comportamiento básico de los agentes

En todo sistema multi-agente, los agentes involucrados poseen un comportamiento básico al cual se le agrega comportamiento inteligente. Por ejemplo, un agente "robot" que tiene que descargar un camión tiene como comportamiento básico moverse hacia delante, girar a la izquierda, levantar una caja, etc. El comportamiento inteligente que podría agregársele es el de comunicarse con otros agentes para obtener la ubicación de una caja, planear la mejor manera de llegar hasta esa caja, etc.

En **Brainstorm/J** el agregado de comportamiento se realiza a través de meta-objetos reflexivos, de manera no intrusiva, por lo que el nivel base está totalmente desacoplado del resto del framework. Esto provee interesantes ventajas ya que, de este modo, se podría utilizar un conjunto de clases como nivel base aunque ni siquiera se posea el código fuente de las mismas. Para agregar comportamiento en Brainstorm/J deberá extenderse la clase abstracta *BasicAgent* la cual posee el método *handleMessage* que se encarga de interceptar los mensajes recibidos por el objeto base.

FraMas, en cambio, agrega comportamiento mediante el uso de decoradores lo que hace necesario que la clase del objeto base herede de *BasicAgentActions* ya que esta es la clase a la cual se le puedan agregar decoradores para extender su funcionalidad. Si la jerarquía de clases del nivel base estuviera definida con anterioridad, no solo es imprescindible modificar la clase del objeto base sino que, si esta heredara de alguna clase en particular , será necesario modificar toda la jerarquía ya que Java no permite herencia múltiple.

Conclusión: los meta-objetos utilizados en Brainstorm/J poseen la ventaja de ser transparentes por lo que no se necesita modificar el código de las clases a reflejar. El inconveniente es que el uso de meta-objetos aumenta el tiempo de respuesta por el sistema de intercepción de mensajes. Este tiempo de respuesta es mejor en FraMas con el uso de wrappers, pero estos no poseen la ventaja de la transparencia y fuerzan el tratamiento de todos los mensajes que reciba el objeto base, aún cuando estos no sean relevantes para el agregado de comportamiento.

3.3.3- Percepción

Una de las características principales de los agentes inteligentes es que estos son capaces de percibir su entorno y detectar cambios y/o eventos relevantes para su contexto.

Similarmente a lo que ocurre con el agregado de comportamiento, en **Brainstorm/J** la percepción se realiza a través de meta-objetos utilizando reflexión. Por esta razón se pueden percibir los mensajes que reciban objetos de cualquier clase sin necesidad de modificar en absoluto el código de las mismas. Esto termina de eliminar absolutamente la dependencia del nivel base respecto de los meta-niveles y por lo tanto permite que este sea definido sin necesidad de conocer el diseño del comportamiento de los agentes y del SMA. Para esto provee la clase *Perceptor,* cuyas instancias serán meta-objetos, que tiene como variables de instancia el objeto que será observado y luna lista de los métodos de este que deberán ser interceptados.

Por su parte, **FraMas,** implementa la percepción mediante el uso del patrón "Observer" [Gamma 95] utilizando pasaje de eventos desde el objeto observado a sus observadores. Por esta razón es necesario que cualquier clase que pudiera ser percibida implemente la interfaz *EventProducerI* que define los métodos *addAgentListener, removeAgentListener* y *notifyAgentListener*. De este modo, el objeto observado deberá llevar una lista de observadores y tendrá que notificarles cada vez que se produzca un evento de interés para estos.

Conclusión: la técnica utilizada en **Brainstorm/J** tiene la ventaja de desligar de toda responsabilidad al objeto observado. La desventaja es que deberá mantener un meta-objeto de percepción por cada clase diferente de la cual se desee observar sus instancias.

3.3.4- Comunicación

Aunque ambos frameworks soportan la comunicación entre agentes físicamente distribuidos, existen varias diferencias entre ellos en este punto.

En **FraMas,** el componente de comunicación entre agentes es un decorador que captura los métodos de la clase base y les agrega el comportamiento relativo a la comunicación. Esta comunicación se realiza mediante el mecanismo de invocación remota de métodos (RMI Remote Method Invocation) que provee el lenguaje Java.

Para implementar la comunicación se debe extender la clase abstracta *Communication* implementando el método *setReceive* que define la clase encargada de recibir los mensajes. Esta clase deberá ser subclase de *Receive* e implementar el método abstracto *receiveMessage*. Para enviar los mensajes, el framework provee una clase concreta *Send* la cual inicia la ejecución de un thread para el envío del mensaje. Cada vez que se quiera mandar un mensaje deberá crearse una nueva instancia de la clase *Send* con el mensaje (instancia de la clase *Message*) como parámetro.

Por su parte, **Brainstorm/J** define una clase abstracta *Communicator* que puede ser extendida para personalizar el componente de comunicación del agente. Esta clase posee un método template *receiveMessage* que invoca al método abstracto *processMessage* con el mensaje recibido como parámetro y notifica al administrador de situaciones sobre la recepción de un mensaje. El usuario del framework deberá extender la clase *Communicator* e implementar los métodos *processMessage* y *sendMessage* para procesar los mensajes recibidos y enviar nuevos mensajes respectivamente. Esto permite utilizar cualquier protocolo y mecanismo de comunicación

Brainstorm/J también provee una implementación por defecto para la comunicación entre agentes que utiliza un formato de mensajes KQML [Finin 97] bajo un protocolo TCP/IP.

En **FraMas,** el componente de comunicación puede ser extendido para soportar KQML o cualquier otro lenguaje, especializando la clase *Message.*

Si bien la implementación del componente de comunicación en **FraMas** no es muy flexible en cuanto al mecanismo que utiliza (fuerza el uso de RMI), presenta algunas características que son muy importantes para crear agentes inteligentes. En primer lugar tanto la recepción como el envío de mensajes funcionan en su propio thread, por lo que el agente no debe abandonar sus tareas (percepción, deliberación, ejecución de acciones; etc.) para enviar o recibir mensajes. Otro punto importante es que el componente de comunicación guarda un registro (log) de los mensajes enviados y de los que no se han podido enviar así como la causa del envío fallido. Además, el framework provee una clase, *PendingMessages*, que permite almacenar los mensajes pendientes para su posterior reenvío. Todas estas características proveen a los agentes de robustez ante los problemas que se puedan ocasionar en las conexiones de red.

Conclusión: ambos frameworks proveen la interfaz para el uso de cualquier formato de mensajes en la comunicación entre agentes pero mientras FraMas fuerza al uso de RMI, Brainstorm/J permite definir cualquier protocolo aunque por defecto usa TCP/IP.

3.3.5- Representación y manipulación de estados mentales

Una de las principales diferencias entre **Brainstorm/J** y **FraMas** está dada por la representación y manipulación de estados mentales.

En **Brainstorm/J** se provee el soporte para administrar el estado mental de cada agente a través de las tres nociones cognitivas básicas: creencias, deseos y objetivos. Esto lo hace manteniendo una base de datos, implementada en JavaLog [Amandi 99] que contiene cláusulas lógicas, las cuales representan dichas nociones cognitivas. Para mantener esta base de datos actualizada fuerza a que los componentes de percepción y comunicación la actualicen a medida que reconocen nuevas creencias. Además, cada acción del agente tiene asociadas precondiciones, que son verificadas antes de la ejecución de la acción, y efectos que modifican el estado de la base luego de que la acción fue ejecutada.

Por otra parte se utiliza un mecanismo de revisión de creencias para verificar la coherencia del estado mental. Esto significa, por ejemplo, que un agente “Robot” nunca creerá que está en dos lugares distintos en un momento dado.

Por su parte, **FraMas** solo maneja en el estado mental de los agentes todo lo referente a las preferencias del usuario.

Conclusión: Brainstorm/J provee un soporte mas amplio y flexible para la representación y manipulación de estados mentales aunque también más complejo.

3.3.6- Comportamiento reactivo

Tanto **FraMas** como **Brainstorm/J** soportan la creación de agentes reactivos aunque proveen distintas facilidades para definir las reacciones de los mismos.

En **FraMas,** el comportamiento reactivo no está definido como tal aunque puede implementarse utilizando selección de acción ante la ocurrencia de un evento. La clase *AdvancedBehavior* utiliza la estructura del patrón *Strategy* para especificar el mecanismo de selección de acciones del agente. De esta manera, para definir un comportamiento reactivo, se deberá extender la clase *Strategy* implementando los métodos *selectAction* y *doAction*. En estos métodos se indican las condiciones para que el agente reaccione ante un evento de una manera Para esto se debe combinar la administración de eventos percibidos (método *eventFired* de la clase *Agent*) con el mecanismo de selección de acción (método *selectAction* de la clase *Strategy*).

En Brainstorm/J, cada reacción está compuesta por cuatro elementos principales: la situación que dispara la reacción, las precondiciones, la acción que se ejecuta y los efectos que esta produce en el estado mental del agente. Cuando se produce una nueva situación se verifican las reacciones para encontrar alguna que pueda ser disparada. Si la reacción encontrada verifica positivamente su precondición, se ejecuta la acción correspondiente, actualizando luego el estado mental del agente con los efectos que produjo la reacción. De este modo, definir una reacción en **Brainstorm/J** implica solo definir el nombre, la situación que la provoca, las precondiciones y los efectos, y la acción que se ejecuta.

Conclusión: **Brainstorm/J** define una estructura de control para las reacciones que usualmente resulta muy útil pero que puede ser muy pesada cuando se requiere de gran cantidad de reacciones simples ya que se debe definir una clase por cada reacción. **FraMas** es más simple y flexible en este sentido aunque requiere mucho más esfuerzo de programación.

3.3.7- Comportamiento deliberativo o cognitivo

Al igual que la reacción, la deliberación también es una forma de decidir la próxima acción a ejecutar. Sin embargo la reacción es un mecanismo estímulo-respuesta que selecciona y ejecuta la acción inmediatamente después de ocurrido el evento. La deliberación en cambio, es un proceso mas complejo que puede involucrar técnicas de inteligencia artificial tales como planning, redes bayesianas, etc.Habitualmente el componente de deliberación no selecciona la próxima acción individualmente sino que genera planes compuestos por un conjunto de acciones que deben ser ejecutadas consecutivamente.

En **Brainstorm/J** se provee un componente deliberativo muy completo y extensible. Este componente actúa en un thread independiente por lo que no interrumpe la actividad del agente.

El componente de deliberación de **Brainstorm/J** utiliza una arquitectura blackboard[Corkill 91] donde las diferentes fuentes de conocimiento utilizan distintas técnicas para generar planes. Una vez generados estos planes se colocan en el blackboard de donde son tomados por una fuente de conocimiento especial llamada “Executor” que ejecuta las acciones de los planes.

El componente deliberativo en **FraMas** es muy simple y similar al reactivo. Mediante la clase *Strategy* se provee la interfaz para que el agente seleccione la próxima acción a ejecutar. En la clase *AdvancedBehavior* se encuentra el método template *run* que utiliza los métodos abstractos *selectAction* y *doAction* de dicha clase *Strategy* que el usuario deberá implementar.

Conclusión: **Brainstorm/J** provee una estructura de control muy amplia y compleja para la inclusión de algoritmos deliberativos. **FraMas**, por su parte, define una interfaz muy simple mediante un método template y dos métodos abstractos.

3.3.8- Aprendizaje

La arquitectura "Brainstorm" define al componente de aprendizaje en un tercer meta-nivel que refleja a los componentes de percepción y reacción. De este modo el componente de aprendizaje estudia el comportamiento de las reacciones y deliberaciones del agente para crear una base de casos a la que podrá recurrir en el futuro cuando se produzca una situación similar a la que ocurrió cuando se almacenó el caso.

A pesar de lo mencionado anteriormente, **Brainstorm/J** no implementa el componente de aprendizaje por lo que cualquier técnica de razonamiento basado en casos deberá definirse como una fuente de conocimiento y modificar todas las clases de deliberación y reacción para que generen los casos de entrenamiento, o bien aprovechar el mecanismo de reflexión para crear meta objetos sobre estas clases.

FraMas, por el contrario provee un diseño de un subsistema de CBR (Case-Based Reasoning) llamado analizador de preferencias que permite a los agentes aprender sobre las decisiones tomadas y el resultado de las mismas. Para ello mantiene una librería donde almacena casos. Cada uno de estos casos esta compuesto por: un identificador, la situación que lo produjo, la solución propuesta y el feedback obtenido del usuario. Para utilizar el analizador de preferencias en su aplicación el programador debe definir el algoritmo de matching entre casos y la adaptación de los casos recuperados. Asimismo puede extender las clases del analizador de preferencias para personalizarlo con los detalles propios del dominio en el que se trabaja.

Por lo explicado recientemente existen ciertas aplicaciones de agentes donde **FraMas** resulta más conveniente que **Brainstorm/J.** Por ejemplo, una aplicación que requiera agentes capaces de administrar las agendas de sus respectivos usuarios y que puedan aprender de las preferencias del mismo resulta más aplicable a **FraMas** ya que el aprendizaje involucra CBR.

Conclusión: **FraMas** provee soporte y flexibilidad para implementar aprendizaje cosa que no ocurre con **Brainstorm/J**.

3.3.9- Movilidad

Tanto **FraMas** como **Brainstorm/J** soportan movilidad débil de agentes de manera bastante similar ya que los problemas con los que se enfrentan son los del lenguaje de programación en el que están implementados que en ambos casos es Java.

El problema es que Java no provee ningún soporte para salvar y restaurar el estado de un thread (contador del programa y pila de ejecución) por lo que esto debe hacerse explícitamente en el momento de iniciar la transferencia.

4- **Conclusiones**

Aunque tanto FraMas como Brainstorm/J han sido desarrollados para construir aplicaciones en el dominio de los agentes inteligentes, existen varias y marcadas diferencias entre ellos.

La primer diferencia que se observa entre los frameworks se refiere a la metodología utilizada para su desarrollo. FraMas ha sido desarrollado abstrayendo comportamiento común de diversas aplicaciones de agentes construidas con ese propósito. Este hecho hace que el framework sea muy útil para SMA similares a los de las aplicaciones utilizadas para el desarrollo como sistemas de agentes de interfaz pero no ha sido probado en otro tipo de SMA.

Brainstorm/J, por su parte, se desarrolló a partir de una arquitectura específica del dominio, Brainstorm. Esta metodología implica la reutilización de un proceso de diseño que junto con algunos ejemplos desarrollados con el framework proveen a este de una mayor robustez.

La manera en que se agrega comportamiento inteligente a los objetos de nivel base es una diferencia importante entre los frameworks. Brainstorm/J utiliza meta-objetos que interceptan los mensajes recibidos por el objeto base de manera transparente por lo que no se requiere modificar el código de este aunque tiene el inconveniente del aumentar el tiempo de respuesta a causa de esta intercepción. FraMas, por su parte, utiliza decoradores que mejoran el tiempo de respuesta pero no proveen transparencia.

Otra diferencia entre estos dos frameworks es que Brainstorm/J maneja una representación del estado mental de los agentes, mientras que esto no ocurre en FraMas salvo con las preferencias del usuario almacenadas en una librería de casos. Esto representa una ventaja para Brainstorm/J ya que, por ejemplo, resulta muy simple agregar un algoritmo de planning, puesto que este se nutre de representaciones del estado mental para definir el estado inicial y los objetivos así como pre y post-condiciones de las acciones que involucra. Sin embargo para ciertas aplicaciones, mantener una representación del estado mental no resulta necesario. En estos casos, FraMas surge como más conveniente puesto que cumple ampliamente uno de los principales requisitos que se le exige a un framework: simplicidad. La simplicidad de FraMas hace que se reduzcan los tiempos de desarrollo de aplicaciones ya que disminuye el tiempo invertido en conocer el framework.

En las aplicaciones donde es necesario que los agentes aprendan sobre las preferencias de los usuarios FraMas provee un soporte superior a Brainstorm/J para implementar la técnica de razonamiento basado en casos. Si bien el aprendizaje fue contemplado por la arquitectura Brainstorm, este aun no ha sido implementado en su materialización: el framework Brainstorm/J.

5- Referencias

[Avancini 00] AVANCINI, FraMaS: Un Framework para Sistemas Multi-agente basado en Composición, Disertación de Maestría. Universidad Nacional del Centro, Facultad de Ciencias Exactas, Instituto de Sistemas (ISISTAN). Marzo 2000.

[Amandi 97] AMANDI, A.; PRICE, A. Object-Oriented Agent Programming through the Brainstorm System. Proceedings of the International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, 2. Londres, p.121-132. 1997.

[Amandi 99] AMANDI A.; ITURREGUI R.; ZUNINO A.; Multi-paradigm Languages Supporting Multi-Agent Development. Multi-Agent System Engineering, European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'99

[Bratman 88] BRATMAN, M. E., ISRAEL, D. J., AND POLLACK, M. E. Plans and resource-bounded practical reasoning. *Computational Intelligence* 4, 4 (1988), 349–355.

[Chauham 98] CHAUHAM, D.; BAKER, A. JAFMAS: A Multiagent Application Development System. Proceedings of the Second International Conference on Autonomous Agents. 1998.

[Corkill 91] CORKILL, D. D. Blackboard systems. *AI Expert* 6, 9 (Sept. 1991), 40–47.

[Demazeau 90] DEMAZEAU, Y.; Müller J.P. Decentralized AI - Proceedings of the Second European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-90)

[Finin 97] FININ, T., LABROU, Y., AND MAYFIELD, J. KQML as an agent communication language. In *Software Agents*, AAAI Press, Menlo Park, USA, 1997.

[Gamma 95] GAMMA, E. et al. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995.

[Gutknecht 97] GUTKNECHT, O.; FERBER, J. MadKit: Organizing heterogeneity with groups in a platform for multiple multi-agent systems. Technical Report RR.LIRMM 97188, Laboratorie d'Informatique, de Robotique et de Microélectronique de Montpellier.Université Montepellier. 1997.

[Kolodner 97] KOLODNER, J. Case-based reasoning. Morgan Kaufmann Publishers, Inc. 1997.

[Reticular 99] RETICULAR SYSTEMS INC. AgentBuilder: An integrated toolkit for constructing intelligent software agents. White Paper, Feb. 1999.

[Zunino 00] ZUNINO, A.; AMANDI A. Building Multi-Agent Systems From Reusable Software Components, in Proceedings of the~3rd Workshop in Distributed Artificial Intelligence and Multi-Agent Systems (3WDAIMAS) in the context of the 7th Iberoamerican Conference on Artificial Intelligence (IBERAMIA 2000) and the 15th Brazilian AI Symposium (SBIA 2000)