UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS
MAGISTER EN INGENIERÍA DE SISTEMAS

FraMaS: Un Framework para Sistemas Multi-agente basado en Composición

Henri Héctor Avancini

Prof. Dra. Analía Adriana Amandi
Director

Tandil, Junio del 2000

UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES
Rector: Carlos Alberto Nicolini
Vice-Rector: Eduardo Miguel
Facultad de Ciencias Exactas
Instituto de Investigación de Sistemas de Tandil (ISISTAN)
Director ISISTAN: Marcelo Campo
Junio 2000

RESUMEN

La *Ingeniería del Software basada en Agentes* ha surgido en respuesta a la demanda de los requerimientos de los Sistemas Multi-agente (SMA). Éstos sistemas son esencialmente distribuidos y están compuestos por un conjunto de *agentes de software* que interactúan a fin de satisfacer sus objetivos. Si bien los SMA son aplicados a una diversidad enorme de sistemas de computadoras (hardware o software) comparten una serie de características, entre las que se mencionan: la información es distribuida, las entidades tienen conocimiento parcial, la computación es asincrónica, no existe un sistema de control central.

Los *agentes de software* (ó *agentes inteligentes* ó simplemente *agentes*) son entidades computacionales autónomas que están dirigidas por objetivos e insertas en un entorno que pueden percibir y actuar sobre él. *Autonomía* significa que las acciones ejecutadas por el agente no son explícitamente iniciadas por el usuario u otro agente; además de que el agente persiste en el tiempo, es decir que no termina su ejecución cuando ha finalizado una tarea, sino que continúa observando su entorno, decidiendo qué acción ejecutar en el próximo instante de tiempo.

Éstas y otras características de los SMA hacen que su diseño, implementación, mantenimiento, etc. sea una tarea poco trivial. Por ello es deseable contar con herramientas que asistan en las etapas involucradas en la construcción de un SMA. Sin embargo, no existe al momento tal herramienta, más allá del gran esfuerzo en investigación dirigido en este sentido.

Una alternativa propuesta desde hace tiempo en la Ingeniería del Software consiste en reutilizar los componentes previamente construidos. Entre las técnicas de reuso empleadas en el paradigma Orientado a Objetos están la reutilización de librerías de clases, los patrones de diseño y los frameworks. Éstos últimos son el esqueleto de un conjunto de aplicaciones que pertenecen a un dominio concreto.

En particular se propone un framework para el dominio de los Sistemas Multi-agente, denominado FraMaS, que incorpora las características de los SMA. Este framework está formado por una estructura de clases e interfaces Java que modela a los SMA a través de un conjunto de *entornos multi-agente*. Cada uno de éstos entornos contiene un conjunto de agentes que interactúan autónomamente a fin de satisfacer sus objetivos usando los servicios brindados.

Cada agente diseñado usando FraMaS tiene un conjunto de *acciones básicas* que son cubiertas por la *funcionalidad avanzada*. Las acciones básicas son aquellas inherentes a la funcionalidad de agente en si y que no involucra técnicas de deliberación, negociación, etc. La idea detrás del diseño de cada agente es contar con una serie de "*decoradores*" sobre las acciones básicas que pueden ser adicionados (y eliminados) dinámicamente.

Cada uno de estos agentes está inserto en un solo entorno simultáneamente, aunque puede también mudarse de uno a otro. Los agentes perciben eventos que ocurren en otros agentes y en el entorno. La comunicación entre agentes es directa.

El framework presentado es viable para la construcción de Sistemas Multi-agente en función de las implementaciones realizadas: asistente personal, sistema de robots que interactúan a fin de satisfacer un objetivo global y la estructura de agentes para comercio electrónico.

FraMaS fue desarrollado con un enfoque *bottom-up*, es decir desde las aplicaciones al diseño. El mismo permite la *adición de nuevos componentes* a través de una serie de métodos incorporados, que simplifican la extensión de la funcionalidad del framework. Luego, otras aplicaciones construidas usando el framework pueden hacer uso de estos componentes.

Palabras claves: Sistemas Multi-agente, framework, Internet, agentes inteligentes, Ingeniería del Software Basada en Agentes.

ABSTRACT

Agent-based Software Engineering has been developed in order to satisfy the requirements of Multi-agent Systems (MAS). MAS basically are distributed and are composed of a set of *software agents* that interact in order to satisfy their objectives. Although MAS are applied to a great variety of different computer systems (hardware and/or software), they share a number of characteristics and components, i.e. the distributed information, entities with partial knowledge, no central control system.

Software agents (or *intelligent agents* or simply *agents*) are autonomous computational entities driven by objectives and inserted in an environment, which can be perceived and in with they can act. By *autonomy* we mean the set of actions performed by the agent without explicit user indication or other entity indication. Moreover, an agent is autonomous if it persists over time, i.e. it does not complete its execution when finished with an action; in fact, it continues perceiving its environment, deciding what action to carry out next, etc.

These and other characteristics of MAS make their development a non-trivial task. It is therefore desirable to have tools to assist with the different stages involved in the construction of MAS. However, despite the research efforts in that area, there are no tools at present.

The reuse of previously constructed software components is a common proposal in Software Engineering. The most common reuse techniques, in the Object Oriented paradigm, are the reuse of libraries of classes, design patterns, and frameworks. Frameworks are the skeleton of a set of applications that belong to a specific domain.

In particular, we propose a framework for the Multi-agent Systems domain called FraMaS, implemented in Java, which incorporates the characteristics of MAS systems. The FraMaS structure is made up of a set of Java classes and interfaces that model Multi-agent Systems by means of a set of multi-agent environments. Each one of these environments contains a set of agents that interact autonomously in order to satisfy their objectives through the use of the services provided by the environment.

Each agent designed under FraMaS has a set of *basic actions* covered by the *advanced functionality*. Actions that do not use any kind of deliberation techniques, learning, negotiation, etc. are basic agent actions; e.g. robot move action is a basic action, whereas the planning of the move action is an advanced functionality. The idea behind the design of each agent is to endow the agent with a set of wrappers over the basic actions that can be added (and deleted) dynamically.

Agents are inserted in only one environment in each time stamp, though they can move from one environment to another. Events that occurs in the agent context (i.e. in other agents or the environment) are perceived by sensors. The communication between agents is direct.

The presented framework (FraMaS) is viable for the construction of multi-agent systems according to our experience with the design, implementation and testing of the following software systems: Meeting Scheduling Agent, Forklift agents' system, and e-commerce agent structure.

FraMaS was developed using a bottom-up approach, i.e. from applications to the design. This permits the inclusion of new components by means of a series of implemented methods that simplify the extension of the framework functionality. Therefore, other applications built using the framework can make use of these components.

Key-words: Multi-agent Systems, Framework, Internet, Intelligent Agents, Agent-based Software Engineering.

Agradecimientos

Quiero expresar mi más sincero agradecimiento a la persona que me ha conducido en los últimos dos años en mi formación, guiándome desde el comienzo y manifestando en cada oportunidad su predisposición ante los innumerables requerimientos que le he realizado: Analía, Gracias!.

Así también extiendo mi agradecimiento a la primera persona que conocí del ISISTAN: Marcelo Campo. Recuerdo perfectamente que en aquel momento se tomó el tiempo para explicarme cuáles serian mis obligaciones y derechos si comenzaba mi investigación aquí en Tandil. Hoy, a poco más de dos años de aquel momento, sólo puedo reconocer con franqueza (y no poca sorpresa) que aquello se ha respetado por completo.

Mi gratitud a los miembros del ISISTAN en general y en particular a Claudia, Daniela, Jane, Silvia, Alejandro, Alfredo, Alvaro, Andrés, Edgardo, Federico y Ramiro por su colaboración en distintos temas de la tesis, por conformar un grupo con el ambiente adecuado para el trabajo y, especialmente, por los momentos compartidos.

En cuanto a Instituciones agradezco a los responsables de la Universidad Nacional del Comahue la beca de investigación que me ha permitido cursar mis estudios de postgrado y a los responsables de la Universidad Nacional del Centro de la Provincia de Buenos Aires por las excelentes instalaciones disponibles para la investigación.

Mi más sentido reconocimiento para mi Madre y mi Padre, quienes desde que puedo recordar han sabido (aun no se cómo!) ceder y ser duros en la justa medida. Comprendiendo con paciencia infinita mis errores e inculcando en mí con el ejemplo un firme optimismo, no de aquel que establece un punto de vista sobre una situación actual, sino aquel optimismo que es una fuerza vital que nos permite mantener erguida la cabeza cuando todo lo demás parece fracasar. Mamá, Papá sinceramente Gracias!.

Me siento complacido de reconocer lo importante que son mis tíos, primos, sobrinos y amigos más cercanos que me han acompañado y escuchado, para ellos mi agradecimiento también.

Henri

ÍNDICE

Índice de Figuras

Índice de Ejemplos

Abreviaturas

ABSE	Agent-based Software Engineering
ACL	Agent Communication Language
API	Application Programming Interface
CBR	Case Based Reasoning
COOL	Coordination Language
DAI	Distributed Artificial Intelligence
DPS	Distributed Problem Solving
FIPA	Foundation for Intelligent Physical Agents
FraMaS	Framework for Multi-agent Systems
FTP	File Transfer Protocol
HTML	Hypertext Markup Language
IA	Inteligencia Artificial
IAD	Inteligencia Artificial Distribuida
IP	Internet Protocol
ISO	International Organization for Standardization
JVM	Java Virtual Machine
KQML	Knowledge Query and Manipulation Language
LAN	Local Area Network
LCA	Lenguaje de Comunicación entre Agentes
MAS	Multi-agent System
OO	Orientado a Objetos
PC	Personal Computer
PDA	Personal Digital Assistant
POA	Programación Orientada a Agentes
POO	Programación Orientada a Objetos
RMI	Remote Method Invocation
RPD	Resolución de Problemas Distribuidos
SMA	Sistemas Multi-agente
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
TDA	Tipo de Dato Abstracto
UDP	User Datagram Protocol
WWW	World Wide Web

1. Introducción

FRaMaS es un framework para Sistemas Multi-agente (SMA) basado en composición, es decir un conjunto de clases[1] en general abstractas, relaciones[2] y métodos[3]; implementado en el lenguaje Java que permite la construcción de Agentes de Software. El framework modela al SMA en "entornos multi-agente", cada uno de ellos contiene a los agentes y es responsable de proveer los servicios para que éstos puedan interactuar.

Cada Agente está formado por un conjunto de *acciones básicas* que son cubiertas por un número de *decoradores* que adicionan la funcionalidad avanzada que el agente tendrá. Los decoradores agregan funcionalidad a los objetos dinámicamente, además es posible cambiar estos decoradores en tiempo de ejecución.

El framework tiene definidos una serie de decoradores para analizar las preferencias del usuario y para la comunicación, como así también tiene definido un protocolo para la incorporación de estrategias de decisión que le permiten al agente decidir que acción ejecutar en el próximo instante de tiempo. Por último, el diseño del framework permite que la incorporación de nuevos decoradores y estrategias de decisión sea de escasa complejidad.

Se presenta en este capítulo una descripción de los agentes de software, sus características principales, y tipos de agentes más nombrados. Así también es presentado el concepto de sistema multi-agente y su relación con la Inteligencia Artificial Distribuida; además de los frameworks como técnica de reuso.

A continuación es descripto el campo de aplicación de los sistemas multi-agente y las principales dificultades encontradas durante su desarrollo. Se propone un framework para sistemas multi-agente a fin de permitir el reuso de componentes previamente construidos y evitar la construcción de este tipo de sistemas desde cero. Luego se introduce la metodología empleada para el desarrollo del framework mencionado y, por último, se describe la organización del presente trabajo.

Los sistemas multi-agente (SMA) son sistemas de computadoras (hardware y/o software) compuestos por un conjunto de entidades (llamadas agentes) que cooperan a fin de satisfacer sus objetivos [Jennings 98]. Las características de estos sistemas (cuyas entidades tienen conocimiento parcial, son distribuidas, etc.) hacen que su construcción sea inherentemente compleja.

La Inteligencia Artificial Distribuida (IAD) se ha ocupado de la resolución de sistemas de este tipo, entendiendo que las entidades mencionadas exhiben algún grado de *inteligencia*. Esencialmente se considera un acto inteligente la agrupación de operaciones con arreglo a ciertas estructuras definidas. La inteligencia es concebida así como la forma de equilibrio hacia la que tienden todos los procesos cognoscitivos. Ésta engendra los problemas de las relaciones entre las estructuras, la percepción y el hábito; además, suscita todas las cuestiones relativas a su desarrollo y a su socialización [Jean-Piaget 92].

Sin embargo, y más allá de las definiciones de inteligencia, la IAD ha sido subclasificada en dos áreas: la resolución de Problemas Distribuidos (módulos que cooperan) y la resolución de Sistemas Multi-agente [O'Hare 96]. Entre las características de los SMA se encuentra que cada

[1] Clases *concretas* y *abstractas*, como así también *Interfaces*
[2] Relaciones de *herencia*, *asociación* y *agregación*
[3] Métodos *base*, *hook*, *template* y *abstractos*

agente tiene información incompleta, que no existe un sistema de control global, que la información es descentralizada.

Los agentes de software (o agentes inteligentes) son entidades computacionales autónomas, insertas en un entorno que pueden percibir los eventos que se producen y actuar sobre él. Esta breve definición sin embargo no es única, diversos autores [Russell 95] [Maes 95] [Wooldridge 95] han propuesto otras varias definiciones según el tipo de sistema que desarrollan. Lo importante, es notar que los agentes de software son una forma de analizar, desarrollar e implementar sistemas de computadoras.

Una *taxonomía* de agentes propuesta por [Nwana 96] establece características que deberían exhibir los agentes: cooperación, autonomía y aprendizaje; logrando distinguir agentes de aprendizaje colaborativos, agentes de interfaz, agentes colaborativos, y agentes inteligentes si tienen las tres características. Existen otras clasificaciones que identifican agentes de información [Etzioni 94], agentes híbridos [Ferguson 92a], agentes móviles [Odyssey 97], etc.

El desarrollo de software en general se ve significativamente facilitado si se reutilizan partes previamente construidas. Idealmente, la construcción de un sistema estaría formado por la unión de los componentes apropiados. En este contexto la palabra "componente" significa una porción de código, una clase, una jerarquía y asociaciones entre clases, etc. Usualmente estos componentes están disponibles desde el código fuente y/o una especificación en algún lenguaje formal o semi-formal.

Los framework son una técnica de reuso orientada a objetos. Permiten el reuso de diseño y código [Johnson 97a]. Básicamente están compuestos por un conjunto de clases y relaciones entre estas clases que definen un dominio de aplicación. La estructura de los frameworks es la base para la construcción de aplicaciones que pertenecen al dominio de aplicación.

1.1. Fundamentación

Los sistemas multi-agente son una forma de construir sistemas de computadoras (hardware o software) que tiene un auge importante porque responden a varios de los requerimientos que se plantean a las aplicaciones actuales.

El aumento de computadoras interconectadas usando redes (LAN, Intranet, Internet) demanda aplicaciones distribuidas, que compartan información, se comuniquen, etc. Las estaciones de trabajo son heterogéneas y los sistemas operativos diferentes (Solaris, Linux, Windows, Unix, etc.). Este número de sistemas debe convivir y, más aún, colaborar a fin de satisfacer objetivos comunes de las entidades que lo componen.

Existen desarrollos que ayudan en este enfoque, como por ejemplo el lenguaje Java con características apropiadas a las necesidades planteadas, la arquitectura CORBA [Orfali 98] diseñada para permitir que componentes inteligentes se localicen y operen entre si; sin embargo, no existen desarrollos que sean aplicables al amplio dominio de los SMA.

Los problemas concretos que plantean este tipo de sistemas (cómo formular, describir, descomponer y distribuir problemas, cómo sintetizar los resultados entre un grupo de agentes, cómo permitir que los agentes se comuniquen e interactúen, qué lenguaje y protocolo usar) hacen necesario contar con herramientas que asistan en la construcción a fin de evitar desarrollos costosos desde cero.

En resumen, la construcción de sistemas multi-agente es un problema de ingeniería, y como tal requiere de la correcta aplicación de las técnicas existentes. Intentar construir cada sistema multi-agente desde cero implica altos costos de producción y manutención, por esto es deseable emplear técnicas de reuso (utilización de lo existente –o previamente construido) para el desarrollo de SMA.

Los sistemas multi-agente comparten un conjunto de componentes y relaciones[4], que pueden ser identificadas y representadas en, por ejemplo, un framework. Los frameworks de aplicación orientados a objetos son una tecnología que intenta reducir los costos y mejorar la calidad del software.

El desarrollo de un framework es un proceso iterativo. En cada paso el framework se puede enriquecer con nuevos componentes y relaciones de los SMA desarrollados. Es así que la construcción del mismo es hecha desde esos sistemas de agentes.

En consecuencia se propone construir un Framework para Sistemas Multi-agente basado en Composición, el cual ha sido denominado FraMaS, y hace posible la incorporación de componentes previamente construidos. Se brinda originalmente un conjunto de componentes que permiten construir agentes de software y la estructura necesaria para la incorporación de nuevos componentes.

1.2. Metodología

La construcción de un framework es un proceso iterativo. Se aplica este principio cuando no se conoce lo suficiente sobre el dominio de las aplicaciones como para hacerlo en una iteración.

En general, existen dos enfoques para la construcción de un framework: desde las aplicaciones o desde una arquitectura. El primero identifica los componentes comunes, y sus relaciones, de las aplicaciones pertenecientes a un dominio, desarrollando desde éstas el framework. El segundo, parte desde una arquitectura de software que representa un modelo de las partes del sistema y sus interacciones.

Se adopta en este trabajo la construcción del framework para sistemas multi-agente FraMaS desde ejemplos, con el objetivo de construir un modelo que permita desarrollar SMA además de permitir que nuevos componentes sean adicionados a través de un protocolo definido.

1.3. Organización

El capítulo 2 define los conceptos involucrados en el presente trabajo de tesis, es decir: los agentes, sus propiedades y diferencias con otros sistemas de computadoras; los sistemas multi-agente, sus características e inconvenientes que presenta su diseño; los frameworks, su relación con otras técnicas de reuso, su desarrollo, etc. Se mencionan en la sección 2.3 las características del lenguaje de programación Java y en que consiste la Ingeniería del Software basada en Agentes.

El capítulo 3 introduce las características de algunos frameworks para agentes. Luego a partir de la sección 3.2 define qué es una teoría de agentes, una arquitectura de agentes y un lenguaje de agentes y, por último se describen tres ambientes para la programación de agentes.

Mientras que el capítulo 4 describe el problema a fin de introducir el dominio de aplicación y limitaciones del framework presentado (sección 4.3), la metodología empleada para su desarrollo se describe en la sección 4.5.

La descripción del framework se encuentra en el capítulo 5. Primero se explica el concepto de decoradores y de la reflexión estructural, haciendo hincapié en las diferencias entre usar decoradores y usar herencia para disponer de mayor funcionalidad en los objetos. Luego, en la sección 5.2 se describe la estructura de un agente usando FraMaS describiendo cómo percibe su entorno, cómo se comunica y cómo realiza la selección de la próxima acción. La sección 5.3 hace lo propio con los entornos multi-agente usado FraMaS, presentando cómo se crean los agentes y la interfaz al SMA, como así también los servicios que el SMA provee a fin de permitir la interacción entre agentes.

[4] Que forman un domino de aplicación.

El capítulo 6 presenta una serie de guías para instanciar un SMA usando FraMaS y la descripción de tres sistemas desarrollados desde el framework. Por otro lado, el capítulo 7 está compuesto por la especificación formal del framework usando Object-Z.

Por último, las conclusiones y el trabajo futuro se introducen en el capítulo 8. Donde se resumen las características principales, contribuciones y limitaciones del framework.

2. Contexto

Las técnicas orientadas a agentes representan una forma de diseñar sistemas de software complejos. Estos sistemas, inherentemente distribuidos, surgen por el aumento de las redes de computadoras y de la información disponible; como así también para hacer frente al desafío de interconectar arquitecturas heterogéneas, entre otras.

Este capítulo presenta los conceptos involucrados en el desarrollo del framework para sistemas multi-agente. Primero se define qué es un agente y qué es un sistema multi-agente, analizando las posturas existentes, características de estos sistemas, las clases de agentes, las motivaciones para el desarrollo de los mismos y los inconvenientes que presentan.

A continuación se describen técnicas de reuso, tales como patrones de diseño, componentes y principalmente los frameworks. Se explican las ventajas y desventajas del uso de frameworks, los tipos de frameworks y la forma en que pueden ser construidos. Luego se describen las características del lenguaje Java, utilizado para la implementación del framework. Introduciendo finalmente la Ingeniería del Software basada en Agentes.

2.1. Agentes y Sistemas Multi-agente

Debido al crecimiento en el número de computadoras y las redes de computadoras, surge la necesidad de hacer que los datos almacenados en ellas estén disponibles en cualquier momento y lugar. Además, como los sistemas de información son más grandes, abiertos y heterogéneos se tornan más complejos. Los agentes de software y los sistemas multi-agente (SMA) representan una nueva forma de analizar, diseñar e implementar estos sistemas de software de aplicación distribuídos [Jennings 98].

Los agentes se vienen usando desde algunos años en una gran variedad de aplicaciones, que van desde pequeños asistentes personales [Mitchell 94] hasta complejos sistemas de control de tráfico aéreo [Ljungberg 92]. Estas aplicaciones (aparentemente tan diferentes) comparten entidades similares que pueden ser abstraídas en una misma arquitectura de software, aun así estos sistemas fueron desarrollados desde cero y no existe actualmente una técnica unificada para su diseño.

El término *agente* ha sido usado desde hace tiempo en Inteligencia Artificial y en Computación Distribuida. Sin embargo, es aún hoy complicado encontrar una única definición que incluya todas las características que los investigadores toman en consideración al hablar de un agente y excluya todas las otras que no tiene que tener.

La tecnología de *agentes* ha mantenido un alto interés comercial y un potencial tecnológico importante para solucionar problemas actuales de los sistemas de computadoras. Entre estos problemas se encuentran: facilitar la interfaz al usuario, reducir el problema del ancho de banda en las redes, solucionar los inconvenientes por el uso de redes no seguras, como así también hacer posible la vieja promesa de diseñar sistemas de computadora que sean adaptables, que aprendan de sus experiencias y autónomos; es decir, que exhiban algún grado de inteligencia.

A continuación se presenta la definición de agente, su relación con los programas y actores, como así también la definición de sistemas multi-agente y las distintas clases de agentes. La sección 2.2 describe los frameworks en general, su relación con los patrones de diseño y las técnicas aplicadas para su desarrollo. Luego, la sección 2.3, introduce las características de Java. Por último, la sección 2.4 describe la Ingeniería de Software basada en Agentes.

2.1.1. Agentes, programas y actores

Se han propuesto diversas definiciones de qué es un agente, hasta el punto que cada investigador en agentes a propuesto qué significa la palabra "agente". Entre estas definiciones:

[FIPA 97] (Foundation for Intelligent Physical Agents) *"...una entidad que reside en un entorno dónde recibe datos a través de sensores que reflejan eventos en el entorno y ejecuta comandos que producen efectos en el entorno. Un agente puede ser solamente de software o de hardware. Un agente es el actor fundamental en un dominio y combina uno o más capacidades de servicio en un modelo unificado que puede incluir acceso a software externo, usuarios humanos y comunicación."*

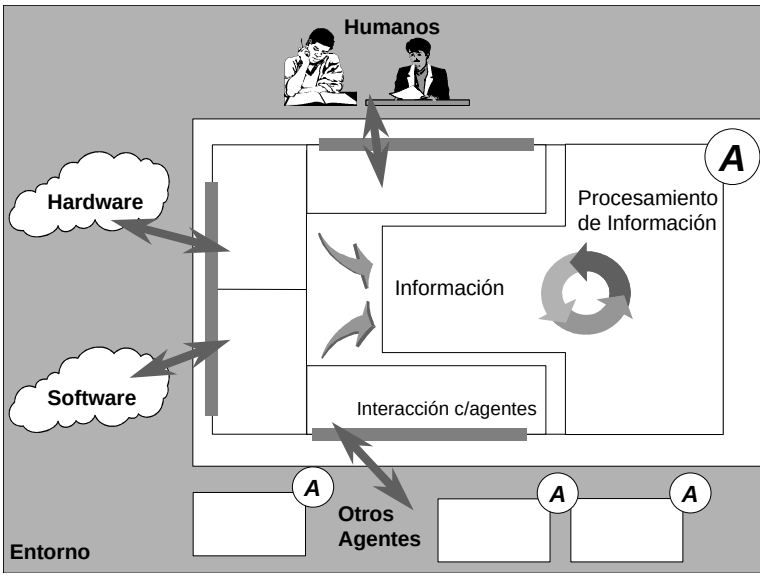


Figura 2 - 1 – Interacción de agentes [FIPA 97]

FIPA especifica los diferentes componentes de un entorno con los que un agentes puede interactuar, es decir: humanos, otros agentes (indicados con una "A" en la figura), sistemas de software (que no son agentes) y elementos de hardware del mundo físico (Figura 2 - 1).

[Wooldridge 95] *"...un sistema de hardware o (más usualmente) un sistema de computadora basado en software que tiene las siguientes propiedades:*

- *Autonomía: los agentes operan sin la directa intervención del usuario, y tiene cierto control sobre sus acciones y su estado interno.*
- *Habilidad social: los agentes interactúan con otros agentes (y posiblemente humanos) usando algún tipo de lenguaje de comunicación de agentes.*
- *Reactividad: los agentes perciben su entorno (que puede ser el mundo físico, el usuario a través de una interfaz gráfica, una colección de otros agentes, INTERNET, o una combinación de estos), y responden a los cambios que ocurren en él.*
- *Pro-actividad: los agentes no actúan solamente en respuesta a su entorno, sino que son capaces de exhibir un comportamiento dirigido por objetivos tomando la iniciativa."*

[IBM] *"Los agentes inteligentes son entidades de software que realizan algunas operaciones en lugar de su usuario o de otro programa con algún grado de independencia y autonomía, empleando algún tipo de conocimiento o representación de los objetivos o deseos del usuario."*

[Russell 95] (AIMA, Artificial Intelligence: a Modern Approach) *"Un agente es cualquier cosa que pueda ser vista como percibiendo su entorno a través de sensores y actuando en ese entorno a través de efectores."*

[Maes 95] *"Los agentes autónomos son sistemas computacionales que habitan en entornos dinámicos complejos, sienten y actúan autónomamente en éste entorno; realizando un conjunto de objetivos o tareas para la cual fueron diseñados.*"

Estas definiciones comparten algunas características, como que los agentes son *autónomos* y están en un *entorno*, dejando otras sin especificar. Sin embargo la noción de *agente* intenta ser una herramienta para desarrollar sistemas, más no una caracterización absoluta que divida al mundo en agentes y no-agentes.

Se coincide en que los agentes están en (y son parte de) un entorno. Pueden percibir y actuar autónomamente sobre este entorno (sin la intervención de otra entidad). Cada agente persigue sus objetivos realizando tareas sobre el entorno que afectarán lo que perciba en el futuro (las tareas tienen efectos sobre el entorno). Finalmente, actúa continuamente sobre un período de tiempo, una vez que un agente fue invocado se ejecuta autónomamente hasta que decide no hacerlo más [Franklin 96]. Lo que puede resumirse en:

> *"Un agente es un sistema situado en un entorno, del que forma parte; puede percibir dicho entorno y realizar operaciones sobre éste, a lo largo del tiempo a fin de satisfacer sus objetivos, y como consecuencia de esto afecta lo que percibirá en el futuro."*

Análogamente, un *programa* de computadoras en un entorno real –por ejemplo un sistema bancario, puede ser visto como que percibe el mundo a través de sus entradas y actúa a través de sus salidas. Pero no es un agente, porque sus salidas podrían no afectar que va a percibir en el futuro, ya que no realiza un cambio en el estado del mundo; además se ejecuta una vez y después deja de hacerlo hasta que es llamado otra vez, mientras que los agentes, una vez iniciados, se ejecutan autónomamente. Todos los agentes son programas, pero no todos los programas son agentes.

Por otro lado, está el concepto de *actor*. Un actor es una entidad que procesa los mensajes entrantes a su casilla de correo, cuya dirección nombra al actor [Kafura 98]. La reacción de un actor ante la llegada de un mensaje es determinada por su comportamiento (definido en un script que puede modificarse en su ciclo de vida). El modelo de actor ha sido ampliamente usado en la programación orientada a objetos concurrente y comparte con los agentes las características de identidad, comunicación y coordinación. Sin embargo, los agentes poseen atributos tales como: autonomía (para decidir que acción ejecutar en el próximo instante de tiempo), adaptabilidad (a través del aprendizaje) y movilidad. Los actores no son agentes pero pueden ser el soporte de implementación de agentes.

Si bien existen otras definiciones de agente, cabe mencionar por último una noción de agente "más fuerte" propuesta por [Wooldridge 95], particularmente usada en Inteligencia Artificial. Se agregan a las características anteriores otras asociadas usualmente con los humanos, tales como: conocimiento y su representación, creencias, intenciones y obligaciones.

2.1.2. Sistemas Multi-agente

Usualmente los agentes son desarrollados como parte de un sistema multi-agente (SMA). La tendencia a la interconexión total entre computadoras hace evolucionar las arquitecturas computacionales hacia un conjunto de agentes que representan a usuarios, servicios o datos [Huhns 97]. Un paradigma usado consiste en la publicación de los servicios por parte del agente de

recursos; mientras que el agente de usuario usa los servicios para encontrar lo que necesita y realizar consultas en consecuencia.

Históricamente, la Inteligencia Artificial Distribuida (IAD) fue dividida en dos áreas principales: la Resolución de Problemas Distribuidos (RPD) y los Sistemas Multi-agente (SMA). Sin embargo, el término SMA ha tomado un significado más amplio, abarcando así a todos los sistemas compuestos por múltiples componentes (semi-)autónomos. Mientras que RPD consiste en cómo solucionar un problema a través de un número de módulos que cooperan compartiendo el conocimiento disponible sobre el problema.

Las características de los sistemas multi-agente son:

- Cada agente tiene información incompleta, y no posee todas las capacidades para resolver el problema por si mismo, así cada agente tiene un punto de vista limitado.
- No existe un sistema de control global.
- Los datos son descentralizados.
- La computación es asincrónica.

Las razones del interés por los sistemas multi-agente abarcan:

- Proveer robustez y eficiencia en sistemas distribuidos.
- La habilidad para coexistir con sistemas actuales.
- Las ventajas del enfoque para resolver problemas en los cuales los datos, la experiencia y/o el control es distribuido.

Sin embargo, los sistemas multi-agente presentan también una serie de inconvenientes:

- Cómo formular, describir, descomponer y distribuir problemas, y cómo sintetizar los resultados entre un grupo de agentes.
- Cómo permitir que los agentes se comuniquen e interactuen. Qué lenguaje y protocolo usar.
- Cómo asegurar que los agentes realicen tareas coherentemente.
- Cómo balancear la computación local y la comunicación.
- Cómo diseñar sistemas multi-agente.

2.1.3. Clases de agentes

Algunas de las clases de agentes y aplicaciones de agentes más difundidas son presentadas a continuación. Primero se describe la topología de agentes de [Nwana 96] que los clasifica de acuerdo a:

- Su capacidad para deliberación o reacción. Los primeros procesan un modelo de razonamiento interno simbólico, realizando planeamiento y negociación, coordinando acciones con otros agentes. Los segundos no tienen un modelo interno simbólico del entorno y actúan según el patrón estímulo – respuesta.
- Su capacidad de movilidad, es decir para trasladarse a través de una red.
- Las características que deberían exhibir los agentes: autonomía, aprendizaje y cooperación (Figura 2 - 2).

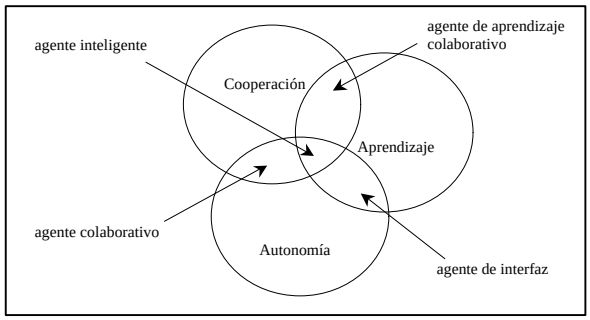


Figura 2 - 2 – Taxonomía de agentes [Nwana 96]

2.1.3.1. Agentes colaborativos

Los agentes colaborativos enfatizan la autonomía y la cooperación a fin de satisfacer las necesidades de sus usuarios. Tienen la capacidad para negociar y decidir por si mismos que hacer en el próximo instante de tiempo.

El proyecto *Pleiades* [Sycara 96] trabaja con este tipo de agentes a fin de aumentar la robustez, efectividad, mantenimiento y hacer sistemas escalables. En este proyecto se emplean agentes colaborativos para la toma de decisiones en organizaciones.

2.1.3.2. Agentes de interfaz

Agentes de interfaz trabajan sobre el aprendizaje y la autonomía. Pattie Maes [Maes 94] propuso los asistentes personales que colaboran con el usuario. Como se observa en Figura 2 - 3 existe diferencia entre colaborar con el usuario y colaborar con otros agentes (en el primer caso no es necesario un lenguaje de comunicación entre agentes explícito, mientras que en el segundo si).

El funcionamiento de un agente de interfaz (Figura 2 - 3), permite la asistencia al usuario y típicamente aprende observando la interacción entre el usuario y el sistema. El agente se desempeña como un asistente personal autónomo que coopera con el usuario en la realización de las tareas.

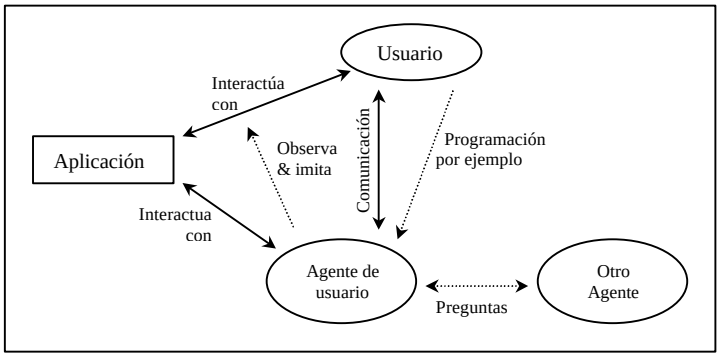


Figura 2 - 3 – Agentes de interfaz [Maes 94]

2.1.3.3. Agentes móviles

Los agentes móviles son capaces de trasladarse de un sitio (host) a otro usando una red de computadoras, como por ejemplo Internet. De esta forma pueden hacer un recorrido interactuando en cada sitio con otros agentes en nombre de su usuario a fin de satisfacer su(s) objetivo(s), para luego retornar al origen. Ejemplos de objetivos son: búsqueda de información, reservas de pasajes.

La movilidad no es una condición suficiente ni necesaria para que un sistema de computadora sea un agente. Los agentes móviles son agentes si además son autónomos y cooperan.

Entre las aplicaciones de agentes móviles se encuentra el sistema *Teleporting* desarrollado por el Laboratorio de Cambridge [Bennet 94] que soporta movilidad. Permite el movimiento de ventanas desde un sitio a otro. Esto se logra introduciendo un nivel de indirección entre las aplicaciones y su interfaz.

Así también los *Aglets* [Lange 98] son objetos móviles Java capaces de visitar diferentes sitios en una red. Los *Aglets* son autónomos, ya que cada uno contiene un *thread* activo de ejecución y es capaz de reaccionar a mensajes enviados a él.

Por otro lado esta el desarrollo de General Magic: *Telescript* [White 95]: un lenguaje orientado a objetos para aplicaciones distribuidas. Introduce un conjunto de mecanismos de protección basado en permisos sobre las distintas entidades. El sistema de runtime de *Telescript* es completamente interpretado y ejecuta *bytecodes* de alto nivel. Esta arquitectura fue recientemente re-implementada usando Java en un sistema llamado *Odyssey* [Odyssey 97].

2.1.3.4. Agentes de información / Internet

Estos agentes han surgido en respuesta a la demanda creciente de herramientas que colaboren en la búsqueda de información sobre grandes repositorios de datos como es Internet. Los agentes de información filtran y administran información desde múltiples fuentes.

Existe similitud entre estos agentes, los colaborativos y los de interfaz. Mientras que los primeros están definidos por *qué pueden hacer*, los otros están definidos por *qué son* (según sus características de cooperación, aprendizaje y autonomía - Figura 2 - 2). Los agentes de Internet deberían ser móviles, aunque no es imprescindible.

En [Etzioni 94] se describen los *Softbot* (robots de software) que permiten al usuario hacer consultas de alto nivel. El agente *Softbot* busca información en Internet y usa conocimiento sobre el usuario (perfil de usuario) para determinar cómo satisfacer las consultas.

2.1.3.5. Agentes de software reactivos

Los agentes reactivos son una clase especial de agentes, no poseen una representación interior con un modelo del entorno en el que se encuentran. Aun así pueden percibir y actuar sobre este entorno a través de acciones del tipo estímulo – respuesta.

Los trabajos en agentes reactivos datan de [Brooks 86] y [Agree 87]. Muchas teorías, arquitecturas y lenguajes para agentes reactivos han sido desarrolladas desde entonces. El punto importante en el desarrollo de este tipo de agente es que su diseño e interacción con otros agentes es relativamente simple, aun así surgen de estas interacciones complejos patrones de comportamiento cuando la unión de varios agentes es vista globalmente.

2.1.3.6. Agentes Híbridos

Las cinco clases de agentes nombrados anteriormente (colaborativos, interface, móviles, información y reactivos) poseen características propias, no existe hoy en día un reconocimiento de qué una de estas clases es la mejor. La alternativa frecuente es tomar una aproximación híbrida, es decir diseñar un agente que posea dos o más de las características de las cinco clases nombradas.

Usualmente las arquitecturas híbridas aprovechan los beneficios de poder reaccionar rápidamente a eventos percibidos del mundo exterior y emplean al menos una sección deliberativa para realizar planning a largo plazo. Ejemplos de estas arquitecturas fueron propuestos por [Muller 94] en *InteRRaP*, [Ferguson 92a] en *Touring Machines* y por [Hayes-Roth 91].

2.1.3.7. Sistemas de agentes heterogéneos

Los sistemas de agentes heterogéneos están compuestos por al menos dos tipos de agentes, donde cada tipo pertenece a una de las cinco clases nombradas. Un sistema de agentes heterogéneos puede

también contener agentes híbridos. La motivación para este tipo de sistemas de agentes es que el mundo tiene diversos sistemas de software que abarcan un amplio espectro de servicios. El objetivo es lograr que estos sistemas puedan operar entre si.

En base a estos sistemas ha surgido la "Ingeniería del Software basada en Agentes" que intenta facilitar la operación entre sistemas de agentes heterogéneos. Uno de los requerimientos principales para este tipo de sistemas es disponer de un Lenguaje de Comunicación entre Agentes (LCA, ó ACL: Agent Communication Language). ARCHON [Witting 92] es una arquitectura propuesta para este tipo de sistemas.

2.2. Frameworks

Un framework es una técnica de reuso Orientada a Objetos. Los framework intentan reducir el costo y mejorar la calidad del software. Un framework es una aplicación reusable, "semi-completa" que puede ser especializada para producir aplicaciones concretas (pertenecientes a un dominio de aplicación) [Johnson 97a].

Un framework puede ser definido a través de un conjunto de clases genéricas que establecen la estructura de comportamiento abstracta de aplicaciones pertenecientes a un determinado dominio [Johnson 88] [Johnson 91]. La estructura de comportamiento abstracta de un framework tiene la característica de poder ser utilizada como base para implementar aplicaciones del dominio del framework. La implementación es realizada mediante la subclasificación de clases genéricas del framework, seguida por la composición de clases concretas, utilizando el concepto de colaboración. De esta manera, componentes de software pueden ser reutilizados a través de herencia y composición.

La estructura de los frameworks y como ésta es utilizada de base para la construcción de sistemas pertenecientes al dominio de aplicación de cada framework es mostrada en la Figura 2 - 4. Las clases A, B, C y D componen la estructura de clases de un framework. Los rectángulos dentro del cuadrado de clases indican métodos definidos en las mismas. Un rectángulo lleno representa un método completamente implementado que no requiere ser reimplementado por subclases. Un rectángulo gris representa un método hook, el cual implementa completamente un determinado comportamiento, pero por defecto, pudiendo requerir su reimplementación en las subclases. Un rectángulo blanco representa un método abstracto, el cual se caracteriza por no presentar implementación, definiendo solamente una interfaz. Este tipo de método debe ser implementado en las subclases.

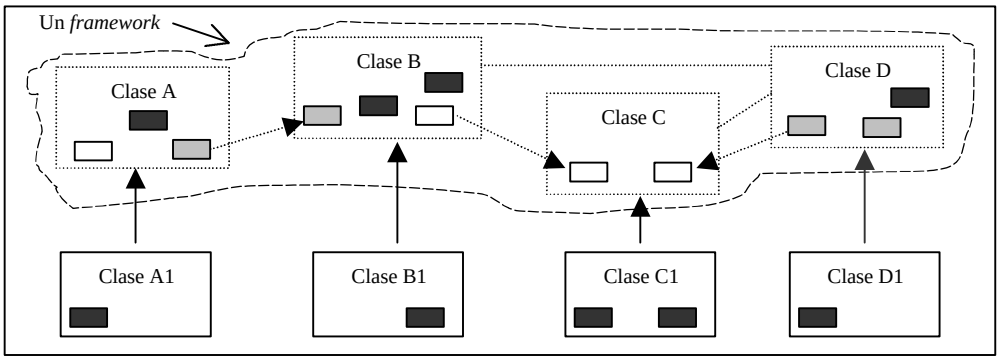


Figura 2 - 4 – Estructura de un framework y una aplicación

En concreto, hay cuatro tipos de métodos: (a) *abstractos*, definen únicamente una interfaz sin implementación y deben ser implementados en cada subclase; (b) *template*, definen un flujo de control común entre objetos del sistema; (c) *hook*, definen una implementación por defecto que puede ser re-implementada en las subclases; y (d) *base*, que representan un comportamiento completo y genérico de los sistemas pertenecientes al dominio del framework, y no puede ser re-implementado en las subclases.

Un framework, como puede ser observado en la Figura 2 - 4, esta definido por un conjunto de clases genéricas (las clases marcadas con líneas punteadas en la figura) y las relaciones establecidas a través de composición (líneas punteadas) y colaboración (flechas punteadas).

Las relaciones de colaboración son establecidas a partir de los métodos. De esta manera, la estructura de composición y de control de aplicaciones de un determinado dominio son establecidas en un framework, los detalles específicos de cada aplicación del dominio son definidos a través de métodos *hook/template* (rectángulo gris) y *abstractos* (rectángulo blanco).

Una aplicación construida a partir de un framework reutiliza tanto la estructura de composición como la estructura de control definida ahí mismo. Solo las subclases necesitan ser definidas para una aplicación particular, implementando métodos *abstractos* heredados y reimplementando métodos *hook* heredados, que requieren cambios de comportamiento. En la Figura 2 - 4 puede observarse que la clase *A1* (subclase de la clase *A*) implementa un método *abstracto* heredado, la clase *B1* (subclase de la clase *B*) implementa un método *abstracto* heredado, la clase *C1* (subclase de la clase *C*) implementa dos métodos *abstractos* heredados, y la clase *D1* (subclase de la clase *D*) reimplementa uno de los dos métodos *hook* heredados.

A continuación se presentan las diferencias entre los frameworks, patrones de diseño y componentes. Luego se presentan las características involucradas en el uso y aprendizaje de los frameworks, como así también las técnicas para el desarrollo de frameworks.

2.2.1. Patrones de diseño, componentes y frameworks

Un patrón de diseño describe un problema que ocurre repetidas veces y su respectiva solución. En general, un patrón de diseño está compuesto de un nombre, la descripción del problema, su solución y las consecuencias de aplicar esta solución. Esta técnica permite el reuso de ideas de diseño.

Un componente de software es una porción de código a ser reusada. El diseño de un componente de software implica un compromiso entre flexibilidad y poder de aplicación. Si el componente tiene una funcionalidad simple es fácil de usar, pero puede emplearse un número pequeño de veces. Por otro lado si tiene muchos parámetros y opciones puede ser usado frecuentemente, pero su uso es difícil.

En general, las técnicas de reuso van desde simples e inflexibles, a complejas y poderosas [Johnson 97b]. Los frameworks están ubicados en un punto medio de estas técnicas de reuso. Por un lado son más abstractos y flexibles que los componentes, y por el otro más concretos y fáciles de reusar que el diseño puro de los patrones. Así los frameworks permiten el reuso de diseño y código. Los patrones de diseño son ejemplificados por programas, pero los frameworks son programas (aunque incompletos).

Cuando los frameworks son usados en conjunto con patrones, componentes y librerías de clases pueden incrementar significativamente la calidad del software y reducir los esfuerzos de desarrollo [Johnson 97a]. Los frameworks permiten la *modularidad* porque encapsulan los detalles de implementación detrás de una interfaz estable. Estas interfaces permiten la *reusabilidad* al definir componentes generales que pueden ser reaplicados para crear nuevas aplicaciones. Análogamente, los frameworks permiten la *extensibilidad* a través de los métodos *hook*, que permiten a las aplicaciones cambiar el comportamiento por defecto definido en el framework.

2.2.2. Uso y aprendizaje de los frameworks

Una aplicación construida usando un framework tiene tres partes [Fayad 99a]: el framework, las subclases concretas de las clases del framework, y una documentación.

La forma más simple de usar un framework es conectando los componentes existentes (esto no cambia el framework ni implementa nuevas subclases concretas). Sin embargo, no todos los frameworks pueden ser usados de esta forma, a veces es necesario subclasificar las clases del framework, lo que resulta más complejo porque hay que conocer la estructura de las clases. Por

último, la forma que requiere más conocimiento para usar un framework es cuando es preciso extender las clases abstractas, usualmente agregando operaciones o variables.

Si los programadores de aplicación pueden usar el framework sin modificar/conocer la implementación del mismo estamos hablando de una *framework de caja negra*. Frameworks que necesitan la extensión de sus clases abstractas usualmente necesitan más conocimiento por parte de los desarrolladores y son llamados *frameworks de caja blanca*. El uso de un framework requiere un mapeo desde la estructura del problema a ser solucionado a la estructura del framework.

El aprendizaje de los frameworks es más difícil que el aprendizaje de una librería de clases, porque se debe aprender el funcionamiento del todas las clases como un todo y no solamente de las clases aisladas. Existe un número de desafíos en la aplicación exitosa de los frameworks:

- *Esfuerzo de desarrollo*. Si el desarrollo de aplicaciones extensibles, y de alta calidad es difícil, el desarrollo de frameworks lo es aún más, porque éstos representan el diseño abstracto de sistemas que pertenecen a un dominio de aplicaciones.
- *Tiempo de aprendizaje*. El empleo de frameworks construidos por terceros no es trivial e involucra un tiempo importante para dominar su aplicabilidad, relaciones y potencial.
- *Mantenimiento del framework.* Al igual que las aplicaciones, los frameworks evolucionan adaptándose a nuevos requerimientos. Por esto se debe tener una visión clara del funcionamiento del framework, además las aplicaciones ya construidas desde el framework deberían reflejar estos cambios.

La mejor forma de aprender a usar un framework es desde ejemplos. Es útil si se cuenta con la documentación del framework y no sólo con el código fuente. La documentación debería al menos explicar: el propósito del framework, cómo usarlo y cómo funciona el framework.

2.2.3. Desarrollo de frameworks

Un framework puede ser construido desde un modelo del dominio (una arquitectura) o desde ejemplos (aplicaciones), en un desarrollo *bottom-up* [Brugali 97]. El diseño de un framework es un proceso iterativo, principalmente porque los desarrolladores no conocen cómo hacerlo en una sola iteración, para lo que es necesario un buen entendimiento del dominio del problema.

Debido al proceso iterativo, el diseño de un framework no debería ser parte del camino crítico de un proyecto. Debería ser desarrollado por un grupo de desarrollo o investigación avanzado, y no por un grupo de producción. Sin embargo, el diseño de un framework debe estar asociado con los desarrolladores de aplicaciones, ya que un framework requiere de la experiencia del dominio de las aplicaciones.

Cada framework que pertenece a un dominio de aplicación, abstrae en su estructura y relaciones las características del dominio. Es decir, aquellas características que se repiten en todas las aplicaciones pertenecientes a mismo dominio. Las partes que cambian de aplicación en aplicación (de un dominio particular) son llamadas *hot spots* [Fayad 99a].

Diferentes aplicaciones de un mismo dominio difieren en al menos un *host spot*. Cada aplicación selecciona una de las diferentes alternativas para cada *hot spot*.

Para aplicaciones reales únicamente un número limitado de frameworks puede ser personalizado usando solamente los *hot spots*. En la mayoría de los casos el proceso de desarrollo de una aplicación a partir de un framework es más complejo, llegando en ocasiones a violar parte de la arquitectura del framework.

Así como evolucionan las aplicaciones comerciales, lo hacen los frameworks. Por lo tanto, no es posible concebir un framework que anticipe todas las futuras evoluciones en el dominio de

aplicación. Un framework nunca está terminado [Codenie 97]. Eventualmente, un framework está "maduro" cuando las sugerencias de mejoras son escasas.

2.3. Java

El lenguaje de programación Java® está diseñado para atender a los requerimientos de los sistemas de computadoras actuales. Fue desarrollado originalmente para artefactos electrónicos, con la funcionalidad para que éstos sean: portables, distribuidos, pequeños, sistemas embebidos de tiempo real, entre otras.

A continuación se describen las características de Java. Las mismas hacen que resulte apropiado su aplicación para la implementación de frameworks:

- *Simple*. Ya que tiene muchas similitudes con C/C++ (lenguaje ampliamente usado desde hace tiempo), omitiendo sus características confusas tales como sobrecarga de operadores, herencia múltiple, etc. Posee un recolector de basura[1] que simplifica la programación y el lenguaje en si es pequeño (puede ejecutarse en computadoras con 40K).
- *Orientado a Objetos*. El diseño OO es muy útil porque facilita la definición de interfaces claras y permite hacer software reusable. El diseño OO es una técnica que enfoca el diseño hacia los datos (objetos) y en las interfaces a ellos.
- *Distribuido*. Java tiene incorporado librerías de rutinas con los protocolos TCP/IP, tipo HTML y FTP. Las aplicaciones Java pueden acceder a objetos a través de una red usando la URL de la misma forma que los programadores acceden al sistema de archivos local.
- *Robusto*. El lenguaje fue diseñado para escribir programas que deben ser confiables de diferentes formas, como así también para detectar errores lo más temprano posible. La mayor diferencia en cuanto a la administración de memoria de Java con C/C++ es que Java tiene un modelo de punteros que elimina la posibilidad de sobreescritura de una posición de memoria.
- *Seguro*. Java pretende ser usado en entornos de red / distribuidos, por ello dispone de varias herramientas incorporadas para permitir construir sistemas seguros y libres de virus. Las técnicas de autenticación están basadas en el encriptado con clave pública.
- *Independiente de la arquitectura*. Fue diseñado para su uso en redes (compuestas por una variedad de sistemas con una variedad de sistemas operativos y hardware heterogéneo). Para permitir que una aplicación Java se ejecute en cualquier lugar de una red, el compilador genera un archivo con formato de objetos independiente de la arquitectura (*bytecodes*) que puede ser ejecutado en distintos procesadores con la presencia del sistema de ejecución Java.
- *Portable*. Java es portable porque además de ser independiente de la arquitectura es independiente de la implementación, es decir que el tamaño de los tipos de datos primitivos y el comportamiento aritmético es estándar.
- *Interpretado*. El interpretador Java puede ejecutar directamente los *bytecodes* en cualquier plataforma que disponga de la máquina virtual Java (JVM).
- *Alta Performance*. Si bien la performance de la interpretación de los bytecodes es aceptable, hay situaciones donde se requiere mayor performance. Para esto los bytecodes pueden ser traducidos en tiempo de ejecución al código particular de la plataforma donde se están ejecutando.
- *Multi-thread*. Es posible construir aplicaciones que ejecuten múltiples threads. Java tiene un conjunto de primitivas para la sincronización inter-procesos. Esta característica es buena para brindar respuestas interactivas y en tiempo real. Sin embargo, está limitado a la

[1] Garbage collector

plataforma en la que se encuentra ejecutando los *bytecodes*, es decir que el tiempo de respuesta dependerá del sistema operativo.

2.4. Ingeniería de Software basada en Agentes

La Ingeniería de Software basada en Agentes surge desde la necesidad de contar con herramientas que permitan la operación entre (distintos) sistemas compuestos de agentes. Los beneficios de dicha tecnología son:

- Permitir que aplicaciones aisladas provean un servicio adicional al poder cooperar con otros sistemas.
- Proveer una nueva forma de diseñar e implementar el software en general, y una nueva forma de inter-operabilidad en particular.
- Mejorar el mantenimiento, actualización y reescritura de rutinas de software.

La Ingeniería de Software basada en Agentes es usualmente comparada con la Programación Orientada a Objetos (POO); en el sentido de que un agente, al igual que un objeto, provee una interfaz basada en mensajes hacia sus estructuras de datos internas y algoritmos [Genesereth 94].

El concepto de objetos data de hace tiempo, incluso Platon en su Diálogo sobre las Ideas establece la noción del mundo compuesto por Objetos (que él llama Ideas):

> *"...Una violeta, un hombre o un tigre cualesquiera no tienen existencia real independiente; lo que únicamente existe es la idea de la Violeta, del Hombre, del Tigre. Esta Idea está más allá de la materia; es un modo de existencia específico que, al realizarse en materia engendra el género natural apropiado a él, la clase llamada violetas, hombres o tigres."* [Platon 370ac]

Las técnicas orientadas a agentes representan una nueva forma de analizar, diseñar y construir sistemas de software complejos [Jennings 00]. La realización de sistemas de software distribuidos y heterogéneos es el desafío que la Ingeniería de Software debe solucionar, brindando herramientas para tal fin.

Mas allá del potencial de la Ingeniería de Software basada en Agentes existen algunos problemas tecnológicos derivados:

- Cómo balancear la carga entre satisfacer los objetivos del agente y mantener una interacción con el entorno, así como entre el comportamiento reactivo y el deliberativo.
- Las interacciones entre agentes son sofisticadas, poderosas y también impredecibles en el caso general. Como los agentes son autónomos, los patrones y efectos de sus interacciones son desconocidos.
- Existe una falta de certeza relacionada con el "comportamiento emergente", aquel que surge como consecuencia de la interacción entre componentes y que no puede ser entendido en término del comportamiento de los componentes individuales.

Por último, se mencionan algunas "fallas" que surgen al pensar en proyectos de desarrollo de agentes [Wooldridge 98]:

- Se sobrestiman las soluciones de agentes, o dónde los agentes pueden ser bien aplicados. La amplia difusión de la palabra *agente* ha provocado una falta de conciencia de que los agentes tienen limitaciones también, si bien representan una forma nueva (y con mucho potencial) de analizar sistemas de software complejos.

- Dogmatismo sobre la tecnología de agentes. Más allá de que los agentes han sido aplicados a un gran número de aplicaciones, no son una solución general. Existen aplicaciones donde el desarrollo de software convencional es más apropiado.
- Se quiere usar la tecnología de agentes pero no se sabe porqué. Es algo que ocurre frecuentemente con el advenimiento de nuevas tecnologías.
- Se olvida el hecho de estar desarrollando software, lo que implica un proceso de experimentación. No existe una técnica probada, confiable y que tenga una performance garantizada para asistir en el desarrollo de sistemas de agentes.

La Ingeniería de Software basada en Agentes está en una etapa de desarrollo temprana y posee potencial para ser aplicada en las Ciencias de la Computación en general, no sólo en Inteligencia Artificial. Las claves para su evolución pasan por entender en que situaciones la solución de agentes de software puede ser aplicada y por el desarrollo de técnicas para la construcción de dichos sistemas.

2.5. Resumen

En este capítulo se presentaron los conceptos involucrados en el desarrollo de técnicas para la construcción de sistemas multi-agente. En particular, se definió el término *agente* como un sistemas de computadoras autónomo situado en un entorno, en el cual puede actuar, es decir percibir eventos y realizar acciones sobre él a fin de satisfacer sus objetivos. Se identificaron distintas clases de agentes: colaborativos, de interfaz, móviles, de información, reactivos e híbridos, según sus características de autonomía, aprendizaje y colaboración.

Los frameworks son una técnica orientada a objetos de reuso, permite el reuso de diseño y código. Su diseño involucra un estudio del dominio de aplicación y un proceso iterativo. Tienen la ventaja de ser más abstractos y flexibles que los componentes, más concretos y fáciles de reusar que el diseño de los patrones. Sin embargo, presentan dificultades al intentar que el framework sintetice las características de su dominio y para su aprendizaje. Se describieron las características del lenguaje Java para la implementación de frameworks.

La Ingeniería de Software basada en Agentes establece una nueva forma de construir sistemas complejos, cuyas características son la distribución y heterogeneidad de sus componentes, involucrando varias áreas de las Ciencias de la Computación.

3. Modelos de Sistemas Multi-agente

Hay un esfuerzo en investigación considerable para el desarrollo de técnicas de diseño de agentes, debido a la diversas áreas donde son aplicables (industria, sistemas de control, sistemas de información, etc.) En este capítulo se presentan las características de algunos de éstos proyectos, en particular los relacionados con frameworks y patrones de diseño para sistemas multi-agente.

Se introducen los conceptos de teoría de agente, arquitectura de agente y lenguajes de programación para agentes, y desarrollos realizados en cada uno de estas áreas. Por último, se describen tres ambientes para la programación de agentes (*AgentBuilder*, *KidSim* y *Simula*).

3.1. Frameworks

En esta sección se describen las características de algunos de los frameworks para agentes existentes. Si bien todos estos frameworks son para el dominio de los sistemas multi-agente no consideran los mismos aspectos, ya que algunos están centrados en movilidad (*Aglets*), otros en coordinación (*Jafmas*), o en simulación de agentes (*Bubble*), etc. A modo de comparación se presenta al final de la sección una tabla que resume sus características.

3.1.1. ...Aglets

Los Aglets[1] son una creación del equipo del laboratorio de desarrollo de IBM en Tokio [Lange 98]. Se definen a través de un conjunto de clases e interfaces Java que permiten crear agentes móviles[2] (Aglet API).

Un Aglet tiene una serie de estados que definen su funcionamiento. Los principales eventos en la vida de un Aglet son: (a) Creación o Clonación; (b) Liberación de recursos; (c) Movilidad; y (d) Persistencia (permite la desactivación y activación del Aglet).

El servidor crea un contexto para los Aglets. Allí serán insertados todos los Aglets que arriban al servidor, además tiene un monitor de red que permite monitorear la red buscando por otros Aglets, y un administrador de seguridad que protege al sitio. Los Aglets están en cada instante de tiempo en un único contexto (Figura 3 - 1)

Los Aglets pueden enviarse mensajes, aún sin "conocerse" entre ellos, a través del empleo de un framework incorporado para tal fin que permite independencia de la ubicación, soporta mensajes sincrónicos y asincrónicos, y puede ser extendido. Para lograr la independencia de ubicación emplea un mecanismo de proxy, de esta manera cada agente envía el mensaje al proxy en lugar de hacerlo al destinatario.

---

[1] código de programa que puede ser transportado junto a la información de estado

[2] una implementación de estas clases, denominada ASDK (Aglets Software Development Kit), contiene los paquetes del Aglet API, ejemplos de agentes y un servidor de Aglets llamado Tahiti y permite al usuario recibir, administrar y enviar Aglets entre computadoras que estén ejecutando el servidor.

La estructura definida en los Aglets permite que estos actúen autónomamente colaborando con otros Aglets. La colaboración es una necesidad en los agentes porque en general ninguno está aislado ni puede realizar tareas complejas solo. La movilidad y el pasaje de mensajes son características importantes para la colaboración.

Un agente de software debería poder migrar con su estado: memoria, pila de ejecución y registros; sin embargo esto no es posible con los Aglets porque están implementados en Java donde un programa no tiene acceso directo para manipular su estado (definido en el modelo de seguridad de la máquina virtual Java -JVM). Por lo tanto, cuando el Aglet va a ser transferido, debe almacenar todo lo necesario para luego retomar su estado cuando es reactivado, esto es llamado movilidad débil.

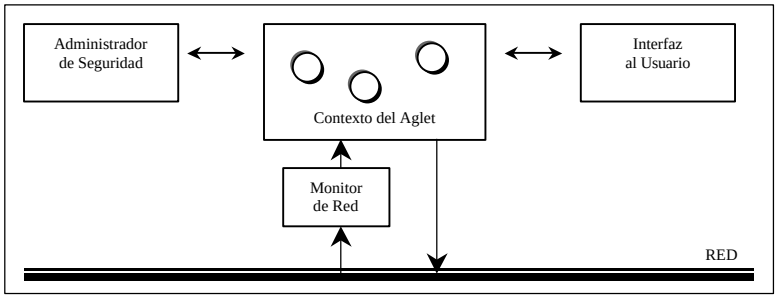


Figura 3 - 1 – Contexto de los Aglets, Interfaz al usuario y Monitor de red [Lange 98]

Existe una limitación de seguridad para que los Aglets sean usados en la práctica, ya que debería haber una estructura en los sitios que proteja de los Aglets no seguros, como así también para permitir el acceso a los recursos a los Aglets seguros. Si bien Java tiene mecanismos de seguridad, hay ataques como la sobrecarga de pedidos al sitio[3] que son posibles. Igualmente, la implementación del servidor Tahiti tiene muchas restricciones de seguridad para los Aglets que no fueron creados en el mismo servidor.

3.1.2. ... Brainstorm/J

El framework Brainstorm/J permite la construcción de agentes de software implementados en Java [Zunino 00]. Fue construido desde la arquitectura *Brainstorm* [Amandi 97a], es decir que usa en su implementación las características del dominio[4] allí especificadas.

La arquitectura *Brainstorm* define cada agente en dos partes, una parte *base* y una parte *meta*. Esto se refleja en el framework a través de un diseño simple de reflexión, dónde cada agente está compuesto por objetos de nivel *base* y un conjunto de *meta objetos* a nivel *meta*, que implementan las capacidades el agente (tales como comunicación, percepción, reacción). Al emplear reflexión es posible que los *meta-objetos* observen el flujo de mensajes entre objetos sin tener que modificar los métodos de las clases.

Esta funcionalidad de reflexión es lograda modificando el cargador de clases Java. El cargador de clases adiciona automáticamente la llamada al meta nivel en el momento que una clase es cargada. La desventaja de este sistema está asociada a la performance de los sistemas reflexivos, el mecanismo de reflexión produce la multiplicación de los mensajes entre objetos.

Cada agente tiene un administrador de situaciones, que permite detectar las situaciones de interés para el agente definidas a través de un conjunto de cláusulas Prolog. Esto es usado, por ejemplo, para implementar el comportamiento reactivo (definido en Brainstorm), disparando una tarea cada vez que la condición de activación es detectada.

El framework Brainstorm/J hace uso de dos frameworks de IBM, por un lado el llamado JKQML [AlphaWorks] para la comunicación entre agentes, lo que le permite soportar la

[3] dejándolo inutilizado para el resto de los usuarios
[4] sistemas multi-agente

comunicación KQML (lenguaje para el envío de mensajes); y por otro el framework para movilidad Aglets [Lange 98] que permite soportar movilidad débil de agentes, heredando así también las limitaciones de seguridad presentes en los Aglets cuando éstos son usados en la práctica.

Para movilidad incorpora la posibilidad de enviar sólo una parte del agente, llamado esclavo. Este agente sigue un recorrido preestablecido, realiza una tarea definida en cada uno de los sitios que visita, para luego retornar al origen con los resultados obtenidos.

La sección deliberativa del agente es tratada en el framework a través del "razonador", el mismo selecciona los objetivos que intentará satisfacer, este razonador se ejecuta en un *thread* independiente permitiendo así la realización de otras tareas simultáneas. Los objetivos dan origen a un plan y éste a las intenciones. Los planes pueden tener dos tipos de acciones: simples (pueden ser ejecutadas directamente) y deliberativas (descripción sobre cómo alcanzar un objetivo).

3.1.3. ... Bubble

Bubble es un framework para construir modelos de simulación de sistemas complejos implementados en Java [Diaz Pace 99]. Permite definir y organizar conjuntos de agentes reactivos con el objetivo de simular el comportamiento global a partir de la interacción entre estos agentes.

Los agentes reactivos constituyen las unidades básicas de la simulación, y se caracterizan por un estado interno y un conjunto de tareas a ejecutar. Además en Bubble, un agente reactivo puede estar compuesto por otros agentes, logrando así distintos niveles de abstracción utilizando estructuras jerárquicas de agentes.

La interacción entre los agentes es lograda mediante un modelo de eventos, contando así con agentes que generan y/o reciben eventos. Existe en el framework un mecanismo de invocación implícita, llamado sensores, que permite a los agentes ser notificados de los eventos que ocurren.

El comportamiento de los agentes es definido a través de un conjunto de tareas de la forma precondición – acción. La representación interna de cada agente refleja un estado y un conjunto de tareas a ejecutar, esta representación puede cambiar a lo largo de la simulación.

Bubble define en su modelo otros tipos de "agentes", como es el caso del contenedor para los agentes en si y también para la visualización, recolección de información. Como se describió en el capítulo anterior los sistemas reactivos exhiben complejos patrones de comportamiento cuando los agentes son analizados globalmente. Sin embargo entidades responsables de otras tareas (visualización, estadísticas, etc.) son considerados agentes cuando en realidad no son parte del sistema multi-agente.

El modelo definido por el framework tiene una parte de estructura, otra de comunicación y una de comportamiento. La primera define cómo los conjuntos de agentes son organizados en la simulación. La segunda se ocupa de la interacción entre los componentes a través de eventos. La tercera establece el conjunto de tareas que pueden realizar los agentes, como así también las condiciones para su activación.

Por último, el framework no soporta entidades distribuidas. Por otro lado, se podría considerar incorporar agentes con capacidades cognitivas (todos los agentes son del tipo precondición – acción), o la actividad concurrente entre las entidades.

3.1.4. ... Concordia

Concordia es un framework implementado en Java que provee un modelo para trabajar con aplicaciones móviles [Walsh 98]. Está compuesto por un conjunto de clases que permite: la ejecución del servidor, el desarrollo de la aplicación del agente y la activación del agente en si.

Tiene mecanismos de protección para los servidores y agentes. La protección de agentes consiste en evitar que se viole la integridad de la información contenida en un agente móvil durante su paso por la red o cuando es almacenado en un disco local.

Para lograr esta funcionalidad, antes de trasmitir un agente de un sitio a otro sobre una red, *Concordia* codifica los *bytecodes* correspondientes al agente, los datos del usuario y la información

de estado usando el mecanismo de *clave pública*. Luego los servidores *Concordia* se autentican uno a otro intercambiando certificados digitales. Análogamente codifica la representación del agente en disco lo que trae aparejado una caída de la performance en la ejecución de los agentes.

Principalmente el framework permite que las aplicaciones accedan a información distribuida en cualquier momento. Las aplicaciones pueden procesar información aún si el usuario está desconectado de la red, usando diversos tipos de redes (LAN, Intranet, Internet) y dispositivos de clientes múltiples (PC, PDA, teléfonos).

Las características más importantes de los agentes definidos en *Concordia* son: movilidad y acceso a datos eficiente. Los agentes tienen reflejado en su estado un itinerario a seguir y la consulta que realizarán en la base de datos. Sin embargo, es el usuario el que crea y envía el agente, indicando eventualmente su itinerario (también puede ser programado en la aplicación). Con lo cual estos "agentes" no resultan tan autónomos como para ser llamados agentes, aunque es posible extender la funcionalidad definida en el framework para lograr la autonomía deseada.

3.1.5. ... JAFIMA

JAFIMA es un framework utilizado en proyectos de sistemas basados en agentes, está implementado en Java y es el resultado de la investigación en el ámbito académico llevado a cabo por Kendall, más que una aplicación comercial (el framework es descripto en [Fayad 99b]).

Incorpora el concepto de agente como una entidad de software de propósito general y ha sido aplicado a diversos dominios. Es definido por sus autores como un framework en desarrollo. Su diseño actual está documentado a través de patrones de diseño.

La definición de los agentes es flexible como para permitir el diseño de agentes simples (reactivos) y agentes complejos (con capacidades de deliberación, movilidad, etc.). El framework define un modelo del comportamiento del agente que incluye sensores para actualizar su conocimiento del estado del mundo, y un mecanismo de selección de objetivos en función de la situación.

La arquitectura de cada agente es por niveles, en los que se encuentran definidos (ver Figura 3 - 2): Sensores, Creencias (mantiene la base de creencias en función de la información de los sensores), Razonamiento (analiza las creencias para determinar que debe ser realizado), Acción, Colaboración, Traducción (de los mensajes) y Movilidad.

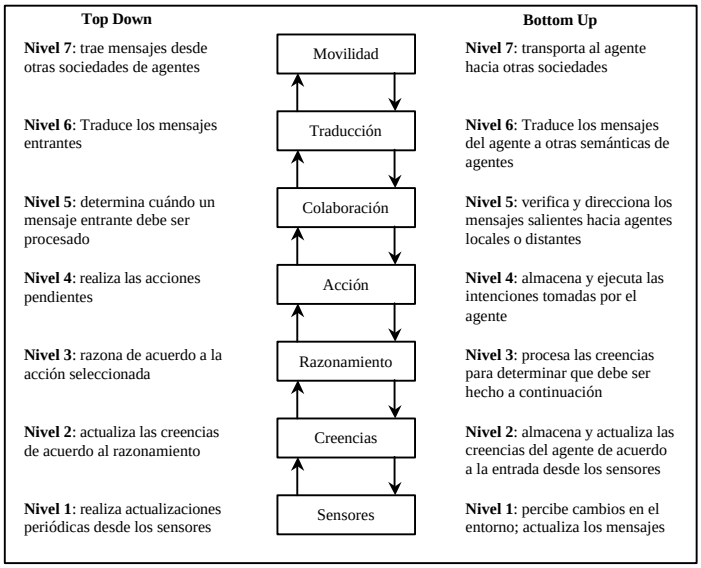


Figura 3 - 2 – Patrón de la arquitectura del agente por niveles [Fayad 99b, p.118]

JAFIMA provee una plataforma para el desarrollo de sistemas basados en agentes, sin embargo hay diversos niveles de los definidos en la arquitectura que no cuentan con comportamiento definido y sólo establecen la interfaz; como es el caso en el nivel de traducción, el nivel de razonamiento, etc. En cuanto a movilidad no tiene definidos mecanismos de seguridad.

3.1.6. ... JAFMAS

El enfoque adoptado en JAFMAS es proveer de una metodología genérica para el desarrollo de sistemas multi-agente basados en comunicación [Chauham 98]. Esta metodología tiene 5 pasos fundamentales: (1) identificación de los agentes; (2) definición de las conversaciones entre agentes; (3) determinación de las reglas de comunicación; (4) analizar la coherencia entre las conversaciones del sistema; e (5) implementación.

El framework está implementado en Java; brinda facilidades para la comunicación (directa por broadcast y usando un blackboard) y soporte lingüístico y de coordinación (KQML). Sin embargo, no incorpora en el diseño del agente capacidades como percepción y deliberación. En cuyo caso habrá que implementar la funcionalidad para estas características. Este framework es flexible, pero exige mucho trabajo de programación construir una aplicación que soporte estas otras características.

Básicamente, JAFMAS utiliza (y extiende) los conceptos definidos en el lenguaje para la comunicación entre agentes COOL [Barbuceanu 93], el cual posee un modelo social centralizado y un directorio para administrar a los agentes, permitiendo que los agentes mantengan varias conversaciones simultaneas al usar distintos threads para cada una.

Las principales diferencias con COOL, además del lenguaje de implementación (Lisp para COOL y Java para JAFMAS), son el análisis de las conversaciones, el soporte de comunicación por broadcast, la falta de una entidad centralizada (que mantiene información sobre los agentes en COOL) y la posibilidad de ejecutar cada agente en su propio thread. Como así también la posibilidad de mantener varias comunicaciones simultaneas en cada agente.

Este framework además de proveer con la metodología mencionada tiene un buen soporte para la comunicación, interacción de los agentes y coordinación del sistema multi-agente; aun así sería importante que contara también con componentes para razonamiento, aprendizaje, etc.

3.1.7. ... JATLite

El framework de la Universidad de Stanford para la construcción de sistemas multi-agente llamado JATLite [JATLite 97] está implementado en Java y provee un conjunto de paquetes Java con:

1. Herramientas para la comunicación, a través del intercambio de mensajes KQML usando TCP/IP.
2. Servidor de nombres de agentes, con la información de todos los nombres y direcciones de los agentes existentes.
3. Funcionalidad para que los Applets Java intercambien mensajes con cualquier agente registrado en Internet.

En general, provee la funcionalidad básica para construir aplicaciones de sistemas multi-agente. Tanto el servidor de nombres como la funcionalidad para el intercambio de mensajes son centralizados lo que puede producir un cuello de botella en el sistema cuando se trata de una aplicación real con varios agentes ejecutándose simultáneamente [O'Hare 96].

La arquitectura por niveles de JATLite está formada por los niveles (de mayor a menor nivel de abstracción): *Protocolo* (soporta SMTP, FTP), *Ruteo* (registro de nombres), *KQML* (almacenamiento y reconocimiento de mensajes en KQML), *Base* (comunicación simple basada en TCP/IP) y *Abstracto* (clases abstractas con los paquetes JATLite).

JATLite no facilita el uso de herramientas de razonamiento ni establece ninguna estructura para autonomía, percepción, etc.; siendo necesaria su implementación para construir sistemas multi-agente con las características mencionadas en el capítulo anterior.

En general el término “agente” en JATLite es usado para significar una comunidad de “entidades” que intercambian mensajes a fin de desarrollar una tarea (agentes reactivos). Esto significa que cada entidad coopera con las otras a fin de solucionar un problema, aún así JATLite no cuenta con estructuras definidas que le permitan representar el problema en su estado interno, aprender, o decidir que acciones llevar a cabo a fin de lograr autonomía.

3.1.8. ... MadKit

MadKit es un framework implementado en Java para el diseño de agentes agrupados por roles en un sistema multi-agente [Gutknecht 97]. Presenta un mecanismo para la construcción de agentes con capacidades de comunicación, soportando KQML y está basado en un modelo de eventos. Los agentes son distribuidos y tienen distintos atributos: percepción, reactividad y conocimiento.

Básicamente el framework define tres conceptos:

- *Agentes*, un agente es únicamente una entidad inserta en la comunidad que desempeña roles y pertenece a un grupo. Esta definición de agente es general y deja al diseñador decidir e implementar el modelo de agente particular. Si bien esto resulta flexible, exige mucho trabajo extra.
- *Grupos*, están definidos como conjuntos de agentes con funcionalidad similar. Cada agente puede pertenecer a uno o más grupos.
- *Roles*, es una representación abstracta de la función o servicio de los agentes. Cada agente puede tener varios roles, y cada rol del agente es local a un grupo.

En resumen el framework está centrado en los aspectos de cooperación y organización más que de la estructura y capacidades de cada agente. El modelo representado en MadKit está formado por agentes heterogéneos agrupados en algún conjunto, el sistema multi-agente como un todo es la unión de estos conjuntos de agentes. Así también, cada agente en los conjuntos tiene una o más funciones, además de su identificación.

3.1.9. ... RETSINA

Se define una estructura de tareas reusable, basada en agentes de red inteligente, implementada en un framework [Sycara 96]. El modelo de agentes de *RETSINA* extiende el concepto del modelo de objetos:

- *Estado del agente*, representa el conocimiento que puede cambiar a lo largo de la vida del agente a través de sus inferencias sobre hechos pasados y presentes.
- *Comportamiento del agente*, definido a través del conjunto de operaciones admisibles sobre su estado interno. Las operaciones internas están representadas por acciones, en lugar de procedimientos dentro de métodos como en objetos.
- *Interfaz del agente*, permite la administración de mensajes. Emplea el lenguaje de comunicación declarativo KQML lo que brinda independencia para que agentes implementados en diferentes lenguajes de programación puedan comunicarse.
- *Identidad del agente*, cada agente está representado por un identificador único dentro del sistema.

Sin embargo no se hace ninguna consideración en el modelo de representación del conocimiento privado del agente, con lo cual insume trabajo extra para instanciar el framework. Por

otro lado establece un modelo de conocimiento colectivo que permite a todos los agentes del sistema compartir dicho conocimiento e inter-operar.

La representación explícita de las acciones permite que el agente razone sobre su propio conocimiento (reflexión). La agrupación de estas acciones produce comportamiento de alto nivel como la realización de tareas y planes.

3.1.10. . Tabla de Clasificación

	Lenguaje de implementación	Tipo de comunicación	Mecanismos de coordinación	Soporte para movilidad	Características de seguridad
Aglets	Java	Directa	No	Si	Desde Java
Brainstorm/J	Java	Directa/KQML	No	Si	Desde Java
Bubble	Java	Por eventos	c/eventos	No	Desde Java
Concordia	Java	Directa	No	Si	Desde Java
JAFIMA	Java	TCP/UDP	No	Si	Desde Java
JAFMAS	Java	Directa/Multicast	Si/Cool	No	Desde Java
JATLite	Java	Socket	No	No	Desde Java
MadKit	Java	KQML	por roles	No	por roles
RETSINA	Java/C++/Perl	KQML	No	No	No

3.2. Teorías

Los trabajos teóricos de agentes se refieren a la especificación de los agentes. En general, se mapea a través de un conjunto de formalismos que representan las propiedades de los agentes. El sistema correspondiente a un agente es descripto de forma conveniente como un sistema de *intenciones* (cuyo comportamiento se puede predecir de acuerdo a los métodos de creencias, deseos, etc.) [Wooldridge 94].

La noción de *intención* suele llamarse también *actitud* y está definida por distintas categorías. Las categorías más usadas son las actitudes de información y las pro-actitudes:

actitudes de información { creencias, conocimiento } pro-actitudes { deseos, intenciones, obligaciones, ... }

Un agente debe ser representado por al menos una actitud de información y una pro-actitud. La lógica de primer orden clásica falla al intentar su uso para representar nociones de intención, porque sus reglas de sustitución no son aplicables a nociones de *deseo* o *creencias*. Los formalismos a ser usados para estas nociones deben cubrir una parte sintáctica[5] y una parte semántica[6].

Para la parte sintáctica se suelen usar *meta-lenguajes*, es decir aquellos lenguajes en los que es posible representar las propiedades de otro lenguaje.

Para la parte semántica existe una semántica llamada *semántica de los mundos posibles*, donde las creencias, conocimientos, etc. del agente están caracterizados por *mundos posibles* y una relación entre ellos. Esta semántica soluciona las dificultades anteriores, aunque trae asociada el problema de omnisciencia, es decir considerar que los agentes son razonadores perfectos.

[5] lenguaje de formulación
[6] modelo semántico

Como solución al problema de la omnisciencia se han propuesto diversas alternativas:

- *Levesque* propuso una solución que hace una distinción entre creencia implícita y creencia explícita [Cohen 90] aunque trae el problema de que realiza predicciones no realistas.
- *Konolige* propuso un modelo de deducción de creencias compuesto por dos componentes: la representación simbólica de las creencias en una base de datos y el mecanismo de inferencia incompleto [Wooldridge 94].

Además, una teoría de agentes debe representar los aspectos dinámicos de los agentes, indicando cómo se relacionan la información del agente y las pro-actitudes, cómo cambian los estados cognitivos del agente, y cómo la información del agente y las pro-actitudes generan las acciones. Así también, una teoría de agentes debe poseer formalismos para representar la comunicación entre agentes.

Entre las teorías de agentes se mencionan: Moore [Moore 90] (conocimiento y acción), Cohen y Levesque [Cohen 90] (intención), Rao y Georgeff [Rao 91] (arquitectura para creencias, deseos e intenciones), Singh [Singh 90], Werner [Werner 88], Wooldridge [Wooldridge 92] (modelamiento de sistemas multi-agente).

3.3. Arquitecturas

Las arquitecturas de agentes son modelos de un sistema, especificados a través de un conjunto de componentes y sus relaciones. La definición de las interfaces entre los componentes involucrados intenta satisfacer las propiedades especificadas en las teorías de agentes. La aproximación clásica analiza a los sistemas de agentes como un tipo particular de los sistemas basados en conocimiento, empleando el paradigma conocido como *Inteligencia Artificial Simbólica* produciendo arquitecturas *Deliberativas*. Otro enfoque son las arquitecturas *Reactivas* y las arquitecturas *Híbridas*.

- *Arquitecturas Deliberativas*, representan un *sistema físico simbólico*. Un sistema físico simbólico está definido como un conjunto realizable físicamente de entidades[7] que pueden ser combinados para formar estructuras, opera sobre estos símbolos de acuerdo a un conjunto de instrucciones. La hipótesis sobre este tipo de sistemas es que son capaces de realizar acciones inteligentes.

  Desde las primeras aproximaciones de agentes artificiales en los 70, que intentaban básicamente realizar sistemas de planning [Fikes 71] [Weld 94], se han desarrollado un gran número de arquitecturas para este tipo de sistemas. Por ejemplo: IRMA [Bratman 88] que cuenta con estructuras de datos para una librería de planes, representación de deseos, creencias e intenciones, un razonador, un analizador de oportunidades, procesos de filtrado y procesos de deliberación; HOMER [Vere 90] que permite la construcción de agentes inteligentes con capacidades lingüísticas y de planning; y GRATE [Jennings 93] que es una arquitectura por niveles en la cual cada agente es guiado por actitudes mentales tales como creencias, deseos e intenciones, y es compuesto por dos partes: un sistema a nivel de dominio y un nivel de control y cooperación.
- *Arquitecturas Reactivas*, han surgido como una alternativa a los problemas asociados a los sistemas simbólicos, su estructura no

[7] símbolos

incluye ningún tipo de modelo simbólico central del mundo ni razonamiento simbólico complejo.

Algunos ejemplos de estas arquitecturas son: *arquitectura subsumptions* [Brooks 86] compuesta por una jerarquía de comportamientos que compiten por tener el control del agente; PENGI [Agree 87] basada en la idea de que la mayoría de las decisiones son rutina y pueden ser codificadas usando estructuras de bajo nivel; el paradigma del *situated automata* [Rosenschein 85] donde un agente es especificado en forma declarativa; y la *arquitectura de red de agentes* [Maes 89] donde los agentes son definidos como módulos de competencia.

- *Arquitecturas Híbridas*, estas intentan sumar en su estructura las ventajas de las dos aproximaciones anteriores, en función de que ninguna de ellas en forma aislada ha resultado completamente exitosa para construir agentes.

  PRS, o Sistema de Razonamiento Procedural [Georgeff 87], es una arquitectura que representa creencias, deseos e intenciones (BDI), tiene una librería de planes y una representación simbólica explícita de las actitudes. TOURINGMACHINES [Ferguson 92a] es otra arquitectura híbrida que consiste de dos subsistemas, uno de percepción y otro de acción (los cuales actúan directamente con el entorno del agente), y tres niveles (reactivo, planning, modelamiento). La arquitectura COSI [Burmeister 92] es una arquitectura híbrida BDI que incluye elementos de PRS y de IRMA. Por último, la arquitectura InterRRaP [Müller 94b] es una arquitectura por niveles (donde cada nivel posee un nivel de abstracción diferente): modelo social, modelo mental y modelo del mundo.

3.4. Lenguajes

Un lenguaje de agentes es un sistema que permite la programación de sistemas de computadora (hardware o software) de acuerdo a las definiciones de las teorías de agentes. Estos lenguajes deben incluir estructuras correspondientes a los agentes, como así también incluir primitivas para administrar creencias, deseos e intenciones.

Además de la POA usando Lenguajes de Objetos Concurrentes (capítulo anterior) se emplean otros lenguajes de programación específicos del paradigma de agentes, entre ellos:

- Agent0, básicamente está basado en una vista social de la computación. El lenguaje permite la programación de agentes directamente en términos definidos en las teorías de agentes [Shoham 93]. Está compuesto por un sistema lógico para definir los estados mentales de los agentes, un lenguaje de programación interpretado para la programación de agentes, y un proceso de "agentificación", para compilar programas de agentes en código ejecutable de bajo nivel.
- PLACA, es un lenguaje más refinado basado en Agent0, que soluciona los problemas de su ancestro: la inhabilidad de los agentes para planear y comunicar requerimientos de acción a través de objetivos de alto nivel [Thomas 93].
- METATEM, una dificultad presente en Agent0 y PLACA es que la relación entre la lógica y el lenguaje de programación interpretado es muy débil [Fisher 94b]. El lenguaje concurrente METATEM permite la ejecución concurrente de agentes, los mismos se pueden comunicar

enviando mensajes por *broadcast*. Cada agente es regido por reglas especificadas usando lógica temporal.

- IMAGINE, es un proyecto en el que se desarrollaron dos lenguajes para programas agentes: APRIL y MAIL [Haugeneder 94]. El primero provee facilidades para la multitarea, comunicación y capacidades de procesamiento simbólico y mapeo de patrones. El segundo provee un conjunto de abstracciones predefinidas, tales como planes; y fue empleado para el desarrollo de sistemas multi-agente
- TELESCRIPT, es un lenguaje basado en el entorno para la construcción de sociedades de agentes desarrollado por General Magic [White 94]. El lenguaje define "lugares" (sitios virtuales ocupados por agentes) y "agentes" (procesos de software móviles con capacidades de comunicación).
- JAVALOG, es un lenguaje de programación implementado en Java que incorpora las características del paradigma OO y el paradigma Lógico [Amandi 99]. Permite la creación y uso de objetos Java en programas Prolog. JavaLog realiza un preprocesamiento de los métodos Java (con código Prolog) permitiendo el uso de variables en ambos paradigmas, además soporta la administración del conocimiento común a través del uso de una arquitectura *blackboard*.

3.5. Ambientes para la programación de agentes

3.5.1. AgentBuilder

AgentBuilder es un ambiente integrado para la construcción de agentes [Reticular 99]. Consiste de dos componentes principales: el *Toolkit* y el *Sistema de Ejecución*. El Toolkit incluye herramientas que permiten el desarrollo de software basado en agentes, el análisis del dominio de las operaciones de agentes, el diseño e implementación de redes de agentes que se comunican, la definición de los comportamientos individuales de los agentes, como así también la prueba y depuración de dichos sistemas. Mientras que el Sistema de Ejecución incluye un motor que provee un entorno para la ejecución de agentes.

Su modelo de agentes está basado en la arquitectura BDI y está implementado en Java. Cada agente construido desde *AgentBuilder* se comunica usando KQML. La extensión empleada por AgentBuilder de la arquitectura BDI mantiene una base de datos que representan las creencias del agente y un conjunto de reglas de comportamiento.

Las capacidades de estos agentes son una asociación entre una acción y las precondiciones necesarias para ejecutar la acción. Esto representa una limitación debido a que no cuenta con estructuras para deliberación, razonamiento y/o aprendizaje.

El ciclo de ejecución de un agente consiste de:

- Procesar los nuevos mensajes.
- Determinar las reglas que son aplicables a la situación actual.
- Ejecutar las acciones especificadas por estas reglas.
- Actualizar el modelo mental de acuerdo a estas reglas.
- Realizar el plan de ejecución de las acciones (*planning*).

Si bien el Toolkit provee una serie de herramientas muy completa para todo el ciclo de desarrollo de agentes de software no es posible realizar extensiones a fin de incorporar nuevas características.

3.5.2. KidSim

KidSim es una herramienta que permite programar agentes sin usar un lenguaje de programación [Canfield 97]. En particular, está diseñada para chicos, permitiéndoles construir simulaciones simbólicas. Es posible modificar la programación de los objetos de simulación existentes y definir otros nuevos desde cero.

Básicamente las simulaciones en KidSim están compuestas de:

- Una tabla de juegos dividida en una grilla.
- Un reloj, que administra un tiempo discreto.
- Uno (o más) objetos de simulación (agentes).
- Una caja de copiado, fuente de nuevos objetos de simulación.
- Un editor de reglas.

La "tabla" representa el entorno donde los objetos de la simulación (agentes) se pueden desplazar. El reloj inicia y detiene la ejecución, siendo posible volver hacia atrás las acciones realizadas.

Cada agente tiene tres atributos: apariencia, propiedades y reglas. Estas últimas definen las acciones de los agentes y son del tipo precondición – acción. Por lo tanto, no es posible obtener agentes con comportamiento deliberativo.

La principal característica de KidSim es que los chicos pueden especificar el comportamiento de los agentes sin usar un lenguaje de programación, a través de la escritura de reglas en un ambiente gráfico y la programación por demostración.

Por último, la extensión propuesta de este tipo de sistema para su empleo en la programación de sistemas de agentes[8] no es en absoluto trivial, al menos no desde los conceptos definidos en KidSim.

3.5.3. Simula

Simula es un prototipo de un ambiente de simulación que posibilita el desenvolvimiento de agentes de software reactivos. El ambiente SIMULA permite crear una simulación de agentes reactivos a través de la programación de los mismos usando una interfaz gráfica. Los agentes están basados en la arquitectura *subsumption* [Frozza 98].

La definición de cada simulación está compuesta por tres partes:

- La especificación de los tipos de agentes. Cada uno posee un nombre y una capacidad de percepción.
- La definición de las reglas de comportamiento, a través de acciones predefinidas en el ambiente y variables. Las reglas son del tipo precondición – acción.
- La ubicación de los agentes en el ambiente.

El comportamiento de los agentes se define como: una serie de Comportamientos Activos y Comportamientos de Estado. El Comportamiento Activo involucra el Movimiento (aleatorio, direccionado, seguir gradiente, seguir una pista, seguir un agente, escapar), las Pistas (dejar, seguir o tomar una pista) y la Vida de agente (morir, reproducirse, transformarse, matar, atacar). El Comportamiento de Estado está conformado por la Percepción de pistas, agentes, etc.

El ambiente, desarrollado en Java, permite generar el código de la aplicación programada, como así también su ejecución. Entre las desventajas de SIMULA se encuentra la limitación para definir qué acción (de las que cumplen con las precondiciones) se ejecutará (el sistema elige una y la ejecuta, debiendo esperar hasta el siguiente ciclo de reloj para ejecutar otra acción).

[8] asistentes personales, tutores inteligentes, etc.

Además, diversos aspectos de la simulación se rigen por funciones de probabilidad que no están accesibles desde la interfaz de programación de los agentes, generando esto una gran dificultad a la hora de emplear el sistema en simulaciones concretas.

3.6. Resumen

Actualmente los frameworks para sistemas multi-agente están siendo implementados en Java debido a las características que presenta el lenguaje. Además estos framework permiten, en mayor o menor medida, facilitar el proceso de desarrollo de sistemas multi-agente a través de un conjunto de estructuras y relaciones entre estas estructuras. Algunos de ellos están centrados en coordinación y comunicación [Chauham 98], otros en movilidad [Lange 98]. También varían respecto al modelo que define a los agentes y el sistema multi-agente. Sin embargo, en términos generales no existe un framework que sea aplicable a todas las aplicaciones del dominio de los sistemas multi-agente y en su mayoría tienen poco control de la privacidad de la información que administran.

Las *teorías de agentes* son formalismos que permiten describir las características de los agentes, cómo ellos realizan acciones, analizan su entorno, etc. Mientras que las *arquitecturas de agentes* llevan estos formalismos a la práctica, estableciendo estructuras y relaciones que reflejen las características definidas en las teorías. Por último, los *lenguajes de agentes* poseen primitivas directamente asociadas al comportamiento y tipo de acciones que un agente puede realizar.

Una importante tendencia actual es programar agentes usando lenguajes orientados a objetos, como lo demuestra la gran cantidad de desarrollos de este tipo (entre ellos los enumerados en la sección 3.1). Esto es debido a que los agentes son vistos como una extensión natural de los objetos (una entidad con un estado interno y un comportamiento, que envía y recibe mensajes).

Ya sea con el empleo de los frameworks, de los lenguajes orientados a agentes, etc. el diseño y programación de sistemas multi-agente representa una tarea no trivial. Es así que se intenta simplificar las construcción de este tipo de sistemas usando ambientes (gráficos) que colaboren en las distintas etapas de desarrollo. En este capítulo se describieron tres ambientes con características distintas: *AgentBuilder* para la programación de agentes a través de un Toolkit para especificar los agentes y su comportamiento; *KidSim* para la simulación de sistemas de agente, pensado para que chicos puedan programar gráficamente, es decir sin usar un lenguaje de programación; y *Simula* un ambiente para la simulación de sistemas compuestos por agentes reactivos.

4. Propósito y Evolución de FraMaS

Se describe en este capítulo el tipo de problemas al que puede ser aplicado el framework aquí presentado. Son introducidos los lineamientos generales del proceso de reuso y su relación con los frameworks. Se nombran algunos ejemplos de los tipos de agentes que pueden ser construidos usando FraMaS y sus capacidades.

A continuación se detallan los pasos seguidos durante el desarrollo del framework de la siguiente manera: primero, describiendo las formas de obtener un framework en general y, a continuación, la metodología empleada en el caso particular de FraMaS. Finalizando con una descripción de las implementaciones realizadas para abstraer los componentes comunes a los sistemas multi-agente. El presente framework ha sido denominado FraMaS: acrónimo de *Framework para Sistemas Multi-agente*.

4.1. Descripción

El desarrollo de sistemas de software que puedan compartir información y servicios unos con otros aumenta con las posibilidades de interconexión entre computadoras. Al existir mayor número de sistema conectados, los sistemas deben soportar la comunicación entre distintas plataformas. Los sistemas multi-agente facilitan la construcción de este tipo de sistemas.

Sin embargo, el desarrollo de sistemas multi-agente es un problema de ingeniería, y como tal, requiere de la correcta aplicación de las técnicas existentes. Los SMA (y agentes) tienen características que dificultan su construcción respecto a los sistemas centralizados. Por ejemplo la distribución de sus componentes hace necesario el establecimiento de protocolos de comunicación y coordinación entre ellos.

Intentar construir cada sistema multi-agente desde cero implica altos costos de producción y manutención, por esto es deseable emplear técnicas de reuso para implementar el nuevo sistema.

4.2. Reuso y frameworks

La clave del reuso es usar lo que existe para conseguir lo que se desea [Goldberg 95]. El reuso es el resultado de un buen diseño y tiene inconvenientes como localizar los componentes a ser reutilizados. Estos componentes de reuso pueden ser tanto de diseño como de código.

En general, un componente de reuso se refiere, en este contexto, a una función, una porción de código, una clase, una jerarquía de clases, un conjunto de clases, jerarquías y asociaciones entre clases. Usualmente, estos componentes están disponibles desde el código fuente y/o una especificación del componente en algún lenguaje formal o semi-formal.

La ventaja de reusar un componente, a desarrollarlo desde cero para un propósito particular, es que se espera que el componente a ser reusado sea un componente testeado con anterioridad. Es decir, que insume menos recursos, mejorando los tiempo de desarrollo, la calidad del software y un mantenimiento menos costoso.

Sin embargo, no es tan simple disponer de una librería de componentes que realice exactamente lo que se necesita. Entonces, una alternativa es “capturar” los componentes comunes

(y relaciones) de un "tipo" de problemas. Los frameworks de aplicación orientados a objetos son una tecnología de la Ingeniería de Software que intenta reducir los costos de desarrollo y mejorar la calidad del software. El framework aquí presentado está compuesto por una serie de clases y relaciones entre clases que abstrae las características de los SMA.

Como se mencionó anteriormente, los sistemas multi-agente son un ejemplo particular de sistemas distribuidos y comparten entre ellos un conjunto de componentes y relaciones, que son identificadas y representadas en el framework. Los agentes en general representan una forma de analizar, diseñar e implementar sistemas de software complejos [Jennings 98].

4.3. Dominio de aplicación

Este framework tiene por objetivo ayudar en el desarrollo de agentes, evitando tener que construir sistemas multi-agente desde cero. El dominio de aplicación de FraMaS son sistemas multi-agente tales como:

- *Asistentes Personales*. Emplean técnicas de aprendizaje para formar un modelo con las preferencias del usuario y así ayudar en la organización y realización de tareas. Estos asistentes son empleados para administración de compromisos, recomendar libros, películas, entretenimiento; por ejemplo CAP [Mitchell 94] aprende las preferencias del usuario sobre sus reuniones, M [Riecken 94] soporta el trabajo cooperativo entre personas y RINGO [Maes 94] agente que recomienda música, son algunos ejemplos de asistentes personales.
- *Agentes para filtrado de información*. Actualmente la cantidad de información disponible a diario es enorme (vía diarios electrónicos, mails, canales de noticias), la idea de estos agentes es brindar al usuario aquellas porciones de información que son significativas para ellos. Maxims [Lashkari 94], administrador de mails, y NEWT [Maes 94], agente para filtrar noticias, son algunos ejemplos.
- *Tutores Inteligentes*. Construyen una relación con el usuario a fin de educarlo. Para ello se emplean agentes con técnicas de aprendizaje y razonamiento que brindan ayuda a las necesidades de un usuario particular. Dicha ayuda es brindada pro-activamente, es decir, el sistema detecta una necesidad del usuario brindándosela antes de que éste la requiera. Por ejemplo COACH [Selker 94] aprende del usuario y se adapta para brindarle ayuda mientras escribe.
- *Comercio electrónico*, se intenta que algunas de las decisiones involucradas en una transacción sea realizada por agentes. Estos son algunos ejemplos de agentes para comercio electrónico: Kasbah [Chavez 96] define un "l*ugar de comercio*" electrónico donde los agentes pueden comprar y vender bienes, y *BargainFinder* [Krulwich 96] busca por los CDs más baratos.

Es decir, agentes con capacidad de:

- *Comunicarse y negociar*, es de una importancia considerable en este tipo de sistemas porque es necesario resolver los problemas interactuando con otros sistemas, ya que (posiblemente) no es posible hacerlo individualmente.

- Observar *al usuario y/o aplicación*, en términos generales, son capaces de formar un modelo con las preferencias del usuario.
- *Percibir su entorno*. Se plantea un modelo de sistema multi-agente donde cada entidad (agente, aplicaciones, y el sistema en sí) produce eventos. Estos eventos pueden ser percibidos por componentes llamados *sensores.*
- *Decidir qué acción llevar a cabo en el próximo instante de tiempo* (usando un mecanismo de planning, un razonador basado en casos, etc.)

Los Agentes, y los Sistemas Multi-agente son, en general, cada vez más aplicados en la industria. Cada vez se quiere incorporar sistemas que integren diversas áreas, que aprendan las preferencias del usuario, que coordinen con otros agentes a fin de llevar a cabo una tarea común, etc.

Además, este tipo de aplicaciones presenta una forma de analizar, diseñar e implementar sistemas, por ello se torna prácticamente imposible abstraer en un framework todas las características del dominio, sin embargo se puede construir un framework con la flexibilidad para la incorporación de nuevos componentes. FraMaS define un protocolo simple para que nuevos componentes sean agregados, tanto decoradores, como estrategias de decisión o servicios del Sistema Multi-agente.

4.4. Desarrollo de frameworks

El desarrollo de un framework es un proceso iterativo. En cada iteración el framework se puede enriquecer con nuevos componentes y relaciones [Schmid 97]. Así, la construcción del mismo es hecha desde las aplicaciones. Además, es preciso realizar un análisis del dominio en el cual el framework será aplicado.

Entre los beneficios de construir sistemas orientados a objetos utilizado frameworks están la modularidad, reusabilidad y extensibilidad [Fayad 99a]. Los frameworks permiten la *modularidad* porque encapsulan los detalles de implementación detrás de una interfaz estable. Estas interfaces permiten la *reusabilidad* al definir componentes generales que pueden ser aplicados para crear nuevos sistemas. Análogamente, los frameworks permiten la *extensibilidad* a través de los métodos *hook*, que brindan a las aplicaciones la posibilidad de cambiar el comportamiento por defecto definido en el framework.

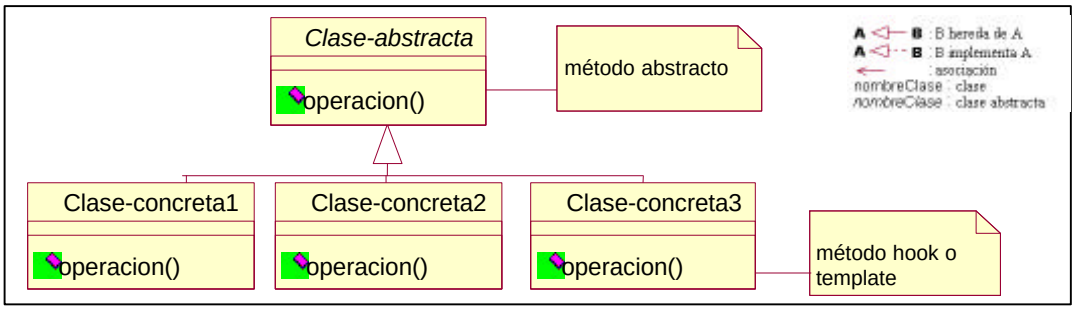


Figura 4 - 1 – Esquema de un hot spot (compuesto por clase abstracta y clases derivadas con métodos hook)

Es posible agrupar las aplicaciones en *dominios de aplicacion*, que están representados por un conjunto de características comunes. Las aplicaciones de un mismo dominio difieren unas con otras en los *hot spots*. Un *hot spot* permite introducir una clase específica a la aplicación en concreto, tanto seleccionada de un conjunto de clases como programando la clase. Una aplicación completa cada hot spot con una de las posibles alternativas.

Los hot spot contienen: una clase base (usualmente es abstracta) que define la interfaz para las responsabilidades comunes (Figura 4 - 1), clases concretas derivadas que representan las diferentes alternativas de configuración de las aplicaciones, además de clases y relaciones adicionales. Análogamente un hot spot está formado por métodos abstractos, métodos template y métodos hook.

El objetivo del desarrollo de un framework es construir una arquitectura de software que pueda ser personalizada en una aplicación concreta a través de los "hot spots" (clases abstractas, métodos template y hook) definidos [Codenie 97]. Para aplicaciones reales únicamente un número limitado de frameworks puede ser personalizado usando solamente los hot spots. En la mayoría de los casos el proceso de desarrollo de una aplicación a partir de un framework es más complejo, llegando en ocasiones a violar parte de la arquitectura del framework.

4.5. Metodología

La metodología utilizada en la construcción de FraMaS está basada en el modelo de desarrollo incremental, con una aproximación bottom-up para el desarrollo de las implementaciones particulares. Así como las necesidades impuestas a las aplicaciones cambian, también lo hacen los frameworks. Es prácticamente imposible disponer de un framework que anticipe todas las evoluciones futuras del dominio de aplicaciones que abarca. Un framework nunca está terminado.

Es decir, se diseña un sistema multi-agente particular y luego se realizan abstracciones sobre sus *componentes*, en concreto aquellos identificados como pertenecientes a los SMA en general. Así también se identifican las *relaciones* entre las distintas partes del sistema.

Este procedimiento permite obtener un primer diseño abstracto de los sistemas multi-agente. El diseño y código es empleado con los requerimientos del SMA construido originalmente para una nueva implementación. Así también, este diseño abstracto es reusado para construir otro sistema multi-agente. Construido este nuevo sistema multi-agente, se identifican los componentes generales y sus relaciones que no se encuentran presente en el framework actual. Se repite el mismo proceso según se muestra en la Figura 4 - 2.

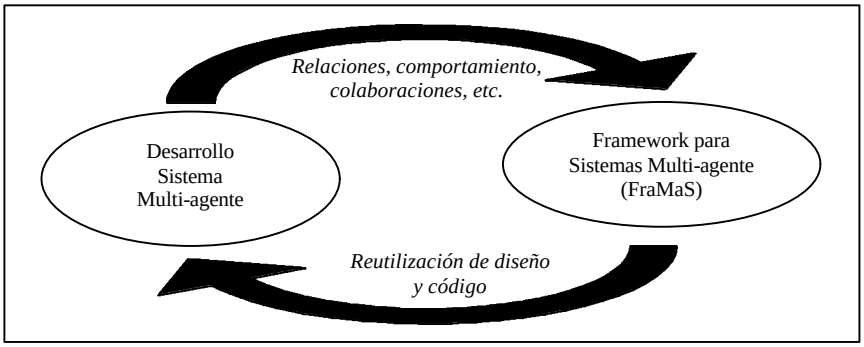


Figura 4 - 2 – Construcción del framework

Se presentan a continuación los SMA desarrollados durante la etapa de construcción del framework. El primer paso fue el planteo de los requerimientos de un sistema multi-agente particular: el Administrador de Compromisos de usuario (*Agente Agenda* – ver sección 6.2 por una descripción completa del mismo).

El Agente Agenda posee la funcionalidad para ingresar, borrar y consultar compromisos, como así también la capacidad de comunicarse con otras agendas y aprender las preferencias del usuario. Estos Agentes Agenda coexisten en el sistema multi-agente, que brinda servicios para compartir información y negociar, en particular los horarios de las reuniones.

Una vez implementado el sistema se identificaron aquellas clases y relaciones generales a la representación de cada agente, sus estados y la percepción del entorno. Además de las

correspondientes al SMA en si con sus servicios. Generando la primera versión de FraMaS. Esta fue empleada para el desarrollo del sistema multi-agente *Forklift* (ver sección 6.3 por los detalles del mismo).

Este sistema modela un conjunto de robots que tienen por objetivo llevar cajas de un sitio a otro. A fin de satisfacer el objetivo del sistema (descargar las cajas desde un camión a las estanterías), deben coordinar para evitar y/o resolver colisiones, elaborar una estrategia para decidir que acción hacer el siguiente instante de tiempo, etc.

Si bien este SMA difiere del anterior, que es un asistente personal, se comprueba que se pueden construir ambos tipos de agentes usando estructuras abstractas similares, y que estos sistemas comparten varias relaciones entre sus componentes.

Luego de realizar sucesivas iteraciones sobre el modelo de desarrollo descripto se empleó el framework en un primer diseño de agentes para comercio electrónico. Estos agentes tienen la funcionalidad necesaria para trasladarse de un host a otro, publicar listas con los servicios (o bienes) que ofrecen y comprobar la existencia de agentes con el servicio (o producto) que buscan, dentro del host en el que se encuentran inmersos.

Por último, la integración de las abstracciones anteriores fue realizada sobre el framework para permitir la interacción coherente de sus entidades (agentes), permitiendo que éstos compartan información, envíen y reciban mensajes, etc. Además, se reimplementaron los SMA presentados en esta sección a partir de la versión final del framework a fin de comprobar su viabilidad.

4.6. Resumen

FraMaS permite el desarrollo de sistemas multi-agente desde un conjunto de clases y relaciones que abstraen características comunes a este tipo de sistemas. Los SMA incluyen agentes para comercio electrónico, filtrado de información, tutores inteligentes, asistentes personales, etc.

La construcción de sistemas a partir del framework permite el reuso de componentes previamente construidos a través de composición y las relaciones entre estos componentes. Es decir, se pueden emplear en la construcción de diferentes sistemas las clases (y sus relaciones) que fueron empleadas en los SMA construidos anteriormente. Ejemplos de estos componentes a ser reusados son: estrategias de negociación, algoritmos de planning, técnicas de aprendizaje.

El diseño de un framework consiste en encontrar una arquitectura que refleje el dominio del problema y que pueda ser empleada para construir aplicaciones concretas desde la personalización de los “hot spots”. Debido a que un framework es usado para construir sistemas que pertenecen al dominio del framework, su desarrollo es mucho más complejo que el desarrollo de sistemas particulares.

El objetivo consiste en crear un framework que permita la construcción de sistemas multi-agente, permitiendo la incorporación de nuevos componentes a través de un protocolo simple. El framework está orientado a la estructura interna de cada agente a través del uso de decoradores que –a diferencia de la herencia, permite la incorporación dinámica de responsabilidades; más que a características como movilidad, coordinación, etc.

La metodología empleada en el desarrollo de FraMaS consistió en un enfoque bottom-up, es decir desde la construcción de agentes particulares se fueron abstrayendo las estructuras, y relaciones comunes a los mismos a fin de obtener un diseño general que corresponda a los SMA. Este procedimiento fue guiado de acuerdo al modelo de desarrollo incremental, donde cada iteración consiste en la implementación del sistema desde el framework actual (la primera iteración es desde cero), la identificación de componentes generales y la reformulación del framework.

5. Descripción del Framework

Este capítulo presenta un framework para sistemas multi-agente basado en composición, denominado FraMaS, implementado en Java. El objetivo del framework es la construcción de sistemas basados en agentes, donde cada una de esas entidades autónomas interactúan a fin de satisfacer sus objetivos. Cada agente está compuesto por un conjunto de *operaciones básicas, decoradas* por un conjunto de *operaciones avanzadas* que agregan funcionalidad.

Aquí se describe cómo los agentes son construidos usando este principio de decoradores, como así también la percepción del contexto de cada agente, las capacidades de comunicación y análisis de las preferencias, y el protocolo definido para incorporar las estrategias de decisión que permiten la selección de la próxima acción a ejecutar. Se presenta también cómo es modelado el sistema multi-agente en su conjunto, explicando los servicios que provee, la creación de los agentes en el sistema y cómo se implementa la interfaz al usuario.

5.1. Introducción

El framework propuesto modela a los sistemas multi-agente (SMA) como un conjunto de "entornos". Cada entorno contiene agentes que interactúan unos con otros. Estos agentes se comunican directamente a fin de satisfacer sus objetivos. Cada agente tiene la capacidad de cambiar de entorno local a fin de satisfacer sus objetivos.

Se observan en la Figura 5 - 1 dos entornos, uno llamado *x* y el otro *y*. Cada uno de ellos con tres agentes denominados: *1, 2 y 3*; y *a, b y c* respectivamente. Se simboliza con flecha de línea llena el pasaje de mensajes, ya sea entre agentes de un mismo entorno, como con agentes de distinto entorno; y con flecha de línea punteada el pasaje de un agente[1] desde un entorno a otro.

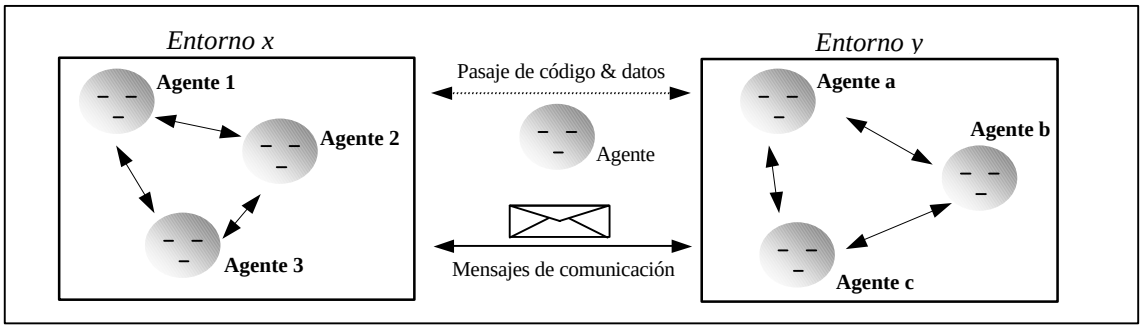


Figura 5 - 1 – Modelo de Sistema Multi-agente.

La función de los entornos es proveer servicios a los agentes, tales como las primitivas para movilidad, a fin de viabilizar la interacción entre ellos. Varios entornos pueden estar físicamente en un mismo *host*, es decir que en la figura anterior tanto el "entorno x" como el "entorno y" pueden coexistir en un mismo sitio.

[1] código + datos

En particular cada agente está formado por una serie de capas superpuestas, llamadas decoradores, lo cual corresponde al patrón de diseño orientado a objetos Decorator. Este patrón de diseño [Gamma 95] adiciona responsabilidad a los objetos dinámicamente, proveyendo una alternativa flexible para extender la funcionalidad.

El resto de esta sección presenta el concepto del patrón de diseño Decorator, la reflexión estructural en Java y las diferencias entre extender la funcionalidad usando herencia o hacerlo a través de decoradores. La sección 5.2 describe el diseño de los agentes implementados desde FraMaS, describiendo cómo perciben el entono, cómo se comunican con otros agentes, cómo está compuesto el analizador de preferencias y el protocolo para la selección de la próxima acción. La sección 5.3 describe las características de los entornos multi-agente definidos en FraMaS, así también como los agentes son creados, cómo es conducida la instanciación de la interfaz al usuario y los servicios del entorno para los agentes.

5.1.1. Decoradores

Con el objetivo de presentar el funcionamiento de los decoradores se describe a continuación un Agente Agenda que administra los compromisos del usuario y será usado como ejemplo. El diseño del agente permite diferenciar la funcionalidad de una agenda y la funcionalidad del agente que la trasforma en un *Agente Agenda*.

La agenda administra los compromisos de los usuarios, permitiendo insertar, borrar y consultar compromisos, esto es denominado “comportamiento básico”. Cada compromiso está compuesto por una fecha, hora, duración, tema, lugar y una lista de participantes. La funcionalidad requerida para esto se encuentra en la clase *Agenda* que hereda de la clase *BasicAgentActions*, perteneciente ésta última a FraMaS. En la Figura 5 - 2 se muestra la clase Agenda con el método correspondiente a ingresar un compromiso.

El Agente Agenda debe tener la capacidad de enviar la invitación a los participantes, aprender las preferencias del usuario; además, de la funcionalidad para proponer alternativas cuando se intenta ingresar un compromiso para un período de tiempo ya empleado. En ese caso de conflicto, la propuesta debe ser algún período de tiempo libre seleccionado en función del conocimiento que posee el agente sobre el usuario (ver la descripción completa del sistema del agente agenda en el capítulo 7, sección 3).

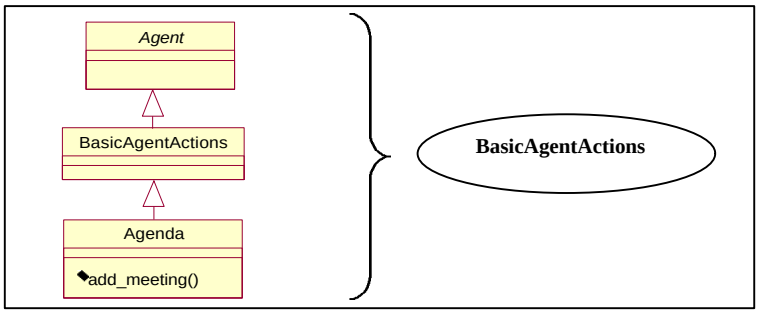


Figura 5 - 2 – Comportamiento básico del Agente Agenda

A continuación se muestra cómo es “decorado” el comportamiento básico de la agenda, en particular la inserción de un nuevo compromiso. El método `add_meeting` que inserta un compromiso en la agenda es cubierto por un decorador de comunicación (clase Communication) a fin de poder enviar las invitaciones a los participantes del compromiso (Figura 5 - 3).

El mensaje `add_meeting` de la clase Communication verifica la lista de participantes. Si no es vacía envía las invitaciones correspondientes. Luego se envía el mensaje `add_meeting` al objeto de la clase padre (`super.add_meeting`) y ésta lo reenvía al objeto de la clase Agenda (`agent.add_meeting`) para que efectivamente se inserte el compromiso en la agenda.

El mecanismo empleado por los Agentes Agenda en la recepción de invitaciones para asistir a un compromiso es a través de un método remoto invocado por el Agente Agenda que propone la reunión. Cada Agente Agenda dispone de un método publicado que escucha por la recepción de invitaciones. Si la invitación es aceptada se envía la confirmación de asistencia usando el mismo sistema.

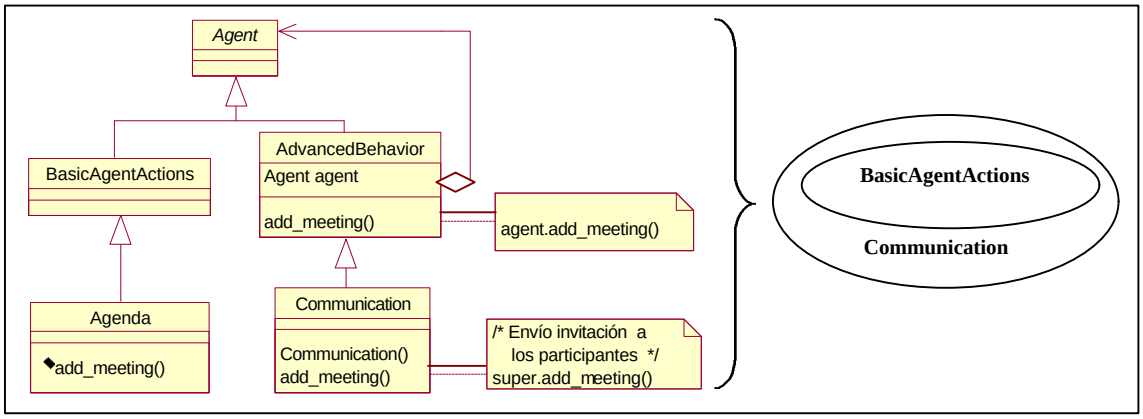


Figura 5 - 3 – Comportamiento avanzado del Agente Agenda (comunicación)

Cabe mencionar que la estrategia para la aceptación de las invitaciones puede variar desde mecanismos simples como chequear si el periodo de tiempo de la invitación está desocupado, o si es propuesta por tal usuario (no) aceptarla, o simplemente dejar la decisión al usuario; hasta políticas más complejas que toman en cuenta varios parámetros de las preferencias del usuario.

Tanto la clase *AdvancedBehavior*, con el mecanismo de reenvío de mensajes; como la clase *Communication* se encuentran implementadas en FraMaS. En la sección 5.2.2 – Comunicación se presentan los detalles de la comunicación entre agentes.

Siguiendo con el ejemplo del Agente Agenda se necesita aprender las preferencias del usuario, que servirán para proponer alternativas cuando se presente un conflicto entre compromisos, por ejemplo entre el horario de un compromiso nuevo y uno ya almacenado. Para esto se decora la capa anterior de *comunicación* con una capa que analiza las preferencias del usuario. Se muestra en la Figura 5 - 4 el diagrama de clases y el esquema correspondiente al mecanismo de aprendizaje:

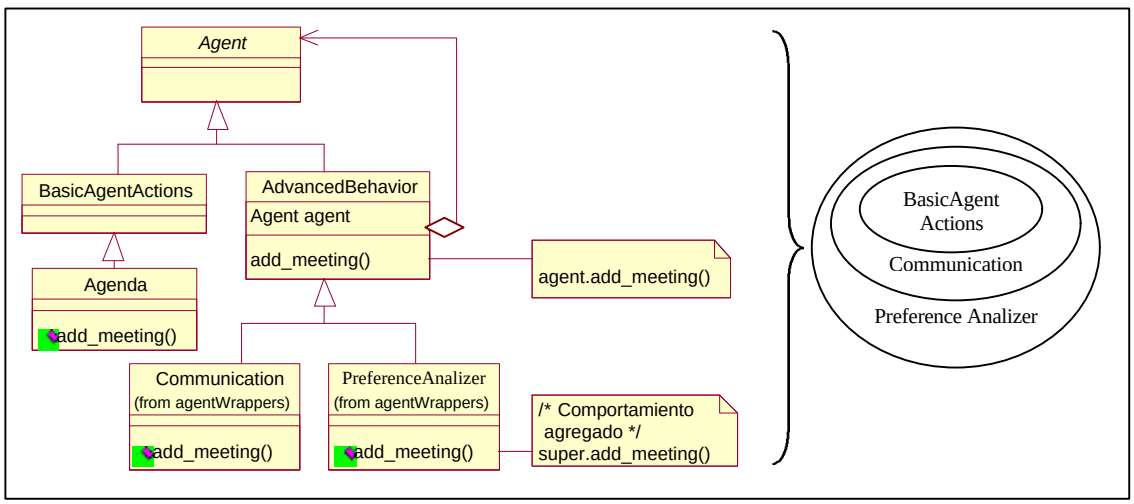


Figura 5 - 4 – Comportamiento avanzado del Agente Agenda (preferencias del usuario)

Aquí el pasaje de mensajes es similar al caso anterior. Cuando un nuevo compromiso va a ser ingresado se envía el mensaje de insertar un compromiso (`add_meeting`) al agente, es decir a la capa externa receptora del mensaje –en el ejemplo es una instancia de la clase *PreferenceAnalizer*. Se verifica que los datos del compromiso a ingresar estén completos (que no le falte fecha, tema, etc.) y que no entre en conflicto con otro previamente almacenado, éste es el "comportamiento agregado" esquematizado en la Figura 5 - 4.

Si se cumplen estas condiciones se envía el mensaje a la clase padre y ésta a la capa siguiente, es decir la capa de comunicación (continuando como fue explicado anteriormente). Si no se cumplen las condiciones anteriores, se emplea un razonador basado en casos [Leake 96] ya sea para completar el compromiso o buscar una alternativa según las preferencias del usuario. Este razonador es entrenado con ejemplos tomados de la observación de la interacción entre el usuario y su agenda.

Más adelante (sección 5.2.3 – Analizador de Preferencias) se exponen en detalle los mecanismos involucrados en FraMaS para aprender las preferencias del usuario, mientras que aquí se presenta la forma en que los decoradores son agregados y cómo intercambian mensajes.

Análogamente se pueden agregar otros decoradores sobre los presentados aquí, cada uno de los cuales adicionará responsabilidades al Agente Agenda dinámicamente. Sobre las ventajas de usar esta técnica en lugar de herencia ver sección 5.1.3 – Decoradores vs. Herencia.

La estructura de clases general al patrón de diseño Decorator es mostrada en la Figura 5 - 5, con una clase abstracta *ComponenteAbstracto*, una clase *ComponenteConcreto*, que implementa determinada funcionalidad en el método abstracto heredado *operación* (`add_meeting`) y *Decorador* que corresponden a las clases *Agent*, *Agenda* y *AdvancedBehavior* del ejemplo anterior.

Cada instancia creada de la clase *Decorator1* (*Communication* en el ejemplo anterior) tiene una referencia a una instancia de la clase *ComponenteConcreto* en su variable de instancia heredada del tipo `ComponeneteAbstracto: componente`. Como así también cada instancia de la clase *Decorador2* (*PreferenceAnalizer* en el ejemplo anterior) tiene una referencia a una instancia de la clase *Decorator1* en su variable de instancia heredada del tipo `ComponenteAbstracto: componente` (Figura 5 - 6).

Se puede "decorar" un objeto de la clase *ComponenteConcreto* con otro objeto que adicione nueva funcionalidad del tipo *Decorador1*. Así el mensaje operación es ahora recibido por este nuevo objeto que reenvía el mensaje al objeto *ComponenteConcreto* e incorpora el comportamiento agregado.

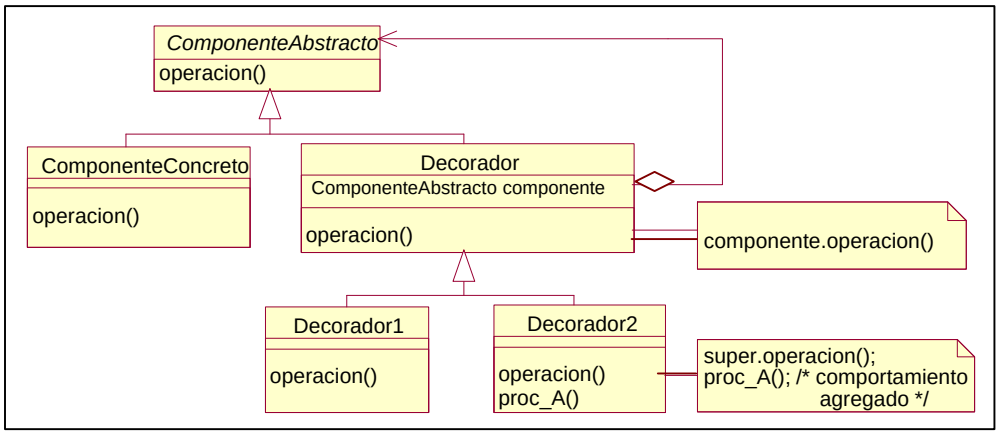


Figura 5 - 5 – Estructura de clases del patrón de diseño Decorator

Análogamente es posible decorar nuevamente estos dos objetos con otro objeto de tipo *Decorador2*. Éste objeto recibe el mensaje operación, realiza el envío del mensaje `proc_A` y lo reenvía a la capa siguiente, es decir al objeto de tipo *Decorador1*; quien lo reenvía a la capa

siguiente, es decir al objeto del tipo *ComponenteConcreto*, incorporando también el comportamiento agregado.

La capa inferior, o núcleo del agente, está formado por el comportamiento básico, es decir, aquel comportamiento que requiere de operaciones simples o que están exentas de técnicas de Inteligencia Artificial. Por ejemplo, en un agente robot, son operaciones básicas: girar, levantar un objeto y moverse en la dirección actual.

Las capas sucesivas brindan al agente del comportamiento avanzado. Como son los mecanismos de deliberación (aprendizaje de las preferencias del usuario, selección de la próxima acción, etc.), comunicación (negociación). Por ejemplo, en un agente robot son operaciones avanzadas: elaborar un plan para alcanzar el objetivo, mecanismos de deliberación y de negociación.

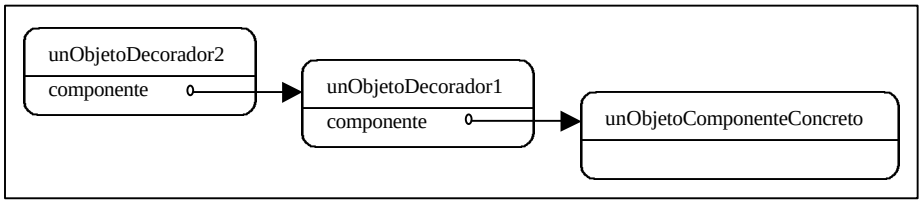


Figura 5 - 6 – Diagrama de instancia del patrón de diseño Decorator

5.1.2. Reflexión estructural

Según la estructura del patrón de diseño *Decorator* y el ejemplo mostrado anteriormente se puede observar la necesidad que la clase *AdvancedBehavior* posea cada método implementado en su subclases (decoradores: *PreferenceAnalizer* y *Communication* en el ejemplo) de forma de poder reenviar el mensaje.

Esto se torna engorroso por la gran cantidad de métodos que termina teniendo esta clase. Otra alternativa viable para lograr la misma funcionalidad es usar reflexión estructural [Ferber 89]. La reflexión estructural es la capacidad de un objeto de inspeccionarse a si mismo conociendo así que mensajes puede entender y responder (ver Apéndice III).

Por otro lado, no es posible enviar mensajes dinámicos en Java. Se dice que el mensaje es dinámico si el control de tipo se realiza en tiempo de ejecución, es decir que en tiempo de compilación no es posible saber si un mensaje enviado a un objeto será o no entendido, en tiempo de ejecución se realiza el chequeo y se determina si el mensaje es entendido por el objeto receptor.

```
Private static Object sendMessage(String message, Object args[])   {
       Object ret = null;
       try     {
         // Busca si un método “message” con argumentos “args[]” existe
         Method meth = getMethod(tclass,message,args,target == null);

         // Si existe lo invoca
         ret = meth.invoke(target,args);

        }
    catch (NoSuchMethodException e)
     {
         /* Excepción
       * invoca al mensaje “doesNotUnderstand”
      */

         Object[] newArgs = { message, args };
         ret = sendMessage(tclass,target,"doesNotUnderstand",newArgs);
        }
    finally {
         return ret;
       }
```

Ejemplo 5 - 1 – Método base sendMessage

Java permite que un objeto se inspeccione a si mismo en tiempo de ejecución, sin embargo no es posible el envío de mensajes dinámicos, porque el chequeo de tipos es realizado en tiempo de compilación. Aún así se puede simular, en Java, el envío de mensajes dinámicos usando reflexión estructural y luego invocarlo si es que lo entiende.

La clase *Agent* incorpora este mecanismo a fin de soportar el envío de mensajes dinámicos entre los objetos que conforman cada agente. Básicamente la clase *Agent* tiene implementado un método base `sendMessage` (Ejemplo 5 - 1) que recibe como argumento el mensaje a enviar.

5.1.3. Decoradores vs. Herencia

Para agregar funcionalidad a una clase se puede especializar la misma a través de herencia. Cada clase hereda las propiedades de sus ancestros, es así que la clase *Agent* puede ser especializada en *Agente_Agenda*, *Agente_Robot*, etc. Sin embargo en FraMaS se emplean decoradores para adicionar responsabilidades a los objetos.

El motivo por el que se emplea el patrón de diseño Decorator para agregar responsabilidades a los objetos, en lugar de herencia, es que ésta es inflexible; ya que la elección de las responsabilidades soportadas es realizada estáticamente. Un cliente de una clase heredada no puede controlar cómo y cuándo decorar el componente. Además, los decoradores pueden variar en tiempo de ejecución.

En este sentido, el empleo de decoradores, brinda una aproximación más flexible al cubrir un objeto con otro objeto que agrega la funcionalidad deseada. Se dice más flexible en el sentido de que cada decorador reenvía los mensajes al componente y puede realizar acciones adicionales.

Además la interfaz al componente decorado es trasparente para los clientes del mismo. Esta transparencia permite que varios decoradores sean puestos sobre el componente recursivamente.

5.2. Agentes

Un agente de software es un sistema computacional autónomo, dirigido por objetivos que está inserto en un entorno. Por autónomo se entiende que el usuario, u otra entidad computacional, no necesita iniciar explícitamente todas sus acciones. Es decir, un agente decide en cada instante de tiempo que acción ejecutar, de acuerdo al conocimiento que posee del estado del mundo, sus capacidades y objetivos.

La clase más importante, en nuestro diseño de un agente es *Agent*. Esta clase es responsable por las características comunes a todos los agentes. La clase *Agent* representa el “componente abstracto” en la estructura del patrón de diseño Decorator.

BasicAgentActions y *AdvancedBehavior* heredan de *Agent*. La primera, es responsable de contener la información de las acciones básicas que el agente es capaz de ejecutar, además de proveer los métodos base para agregar (`add`) y borrar acciones (`delete`); esta clase representa el “componente concreto” en el patrón de diseño Decorator. La segunda, *AdvancedBehavior*, es responsable del comportamiento complejo del agente, es decir los mecanismos de deliberación del agente (usando razonamiento basado en casos, *planning*, etc.) y comunicación (negociación); esta clase representa el “decorador” en el patrón de diseño Decorator.

Cada agente inserto en un entorno tiene la capacidad de percibir dicho entorno. Las entidades del entorno que pueden generar eventos son: agentes, aplicaciones, usuarios y el mismo entorno. FraMaS posee una interfaz común para percibir el entorno del agente, y una clase abstracta que especifica las funciones que el agente tendrá al interactuar con el entorno definida en *AgentContext*. Esta clase posee los siguientes métodos abstractos: `getAgentEnvironment` (obtener el entorno actual), `getAgentIdentity` (identidad del agente), `publish` (publicarse en el entorno) y `unpublish` (salir de la lista de publicados); los cuales son implementados en *AgentContextImplementation*.

El mecanismo implementado para la publicación de un agente es usado, por ejemplo, para que un agente que arriba a un sitio pueda contactar a otros en función de servicios ofrecidos.

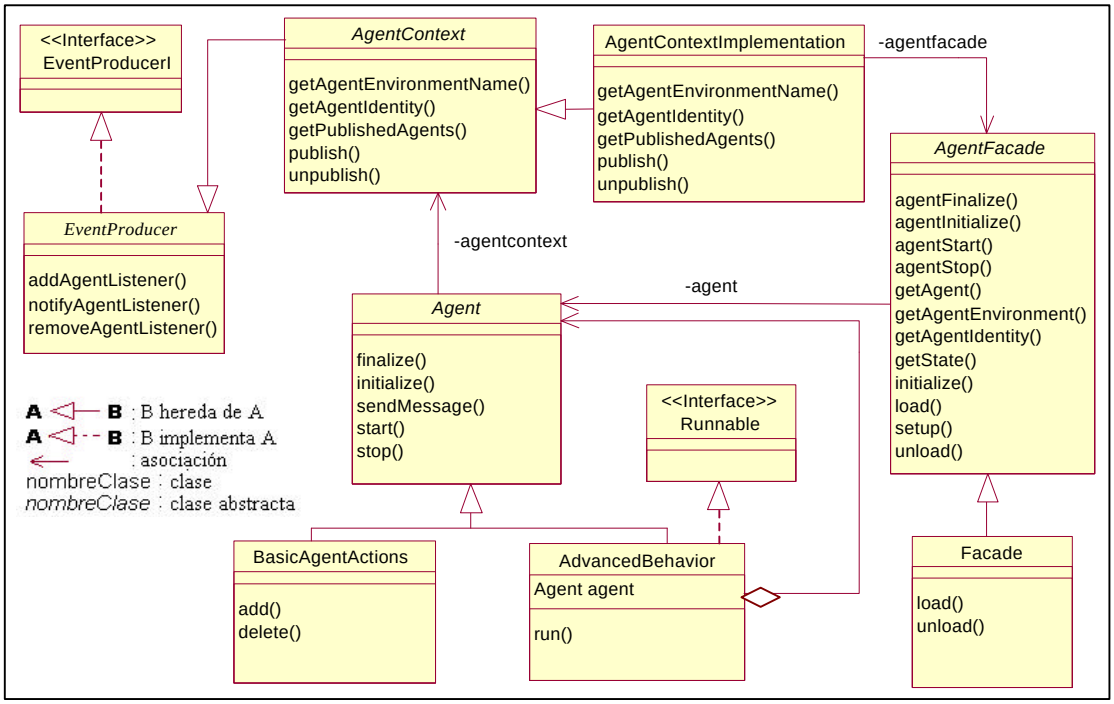


Figura 5 - 7 – Esquema de clases del diseño de un agente

Se presenta en la Figura 5 - 7 el esquema de clases de un agente correspondiente a lo descripto anteriormente, como así también la clase abstracta *AgentFacade* responsable de la interfaz al agente y la clase *Facade* responsable de la carga de cada agente en el entorno multi-agente.

FraMaS define una clase abstracta para acceder desde el entorno al agente implementada en la clase *AgentFacade*. Esta clase tiene la funcionalidad para administrar el estado del agente y la carga (descarga) de un agente al sistema. Los estados posibles de un agente (Figura 5 - 8) son:

- Inicializado, (mensaje `agentInitialize`) se envía una vez en el ciclo de vida de un agente y tiene por objetivo iniciar las estructuras del agente.
- Ejecutando, (mensaje `agentStart`) el agente es conducido por el ciclo: selección de la próxima acción–ejecución, y la observación del entorno.
- Detenido, (mensaje `agentStop`) lleva al agente a un estado seguro para que puede ser transferido, guardando los resultados intermedios de las acciones que está llevando a cabo.
- Finalizado, (mensaje `agentFinalize`) termina todas las actividades del agente y finaliza su ejecución.

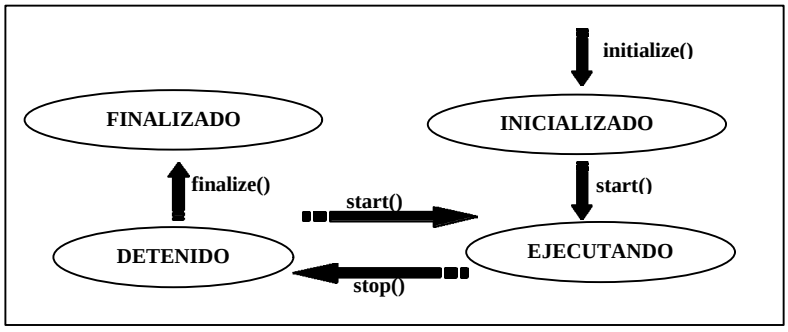


Figura 5 - 8 – Diagrama de estado de un agente.

La transición del estado *Ejecutando* al estado *Detenido* (y viceversa) implica resguardar (recuperar) los estados intermedios obtenidos por el agente según las tareas iniciadas, como por ejemplo: cantidad de agentes que está percibiendo, acciones de comunicación iniciadas, etc. El framework implementa en la clase *AgentFacade* los métodos base para las operaciones de cambio de estado, a fin de evitar que el agente llegue a un estado inconsistente si dos o más mensajes de cambio de estado son recibidos.

Usando estos métodos se simplifica la clase *Agent*, desacoplando el subsistema de clases que heredan de ésta. Proveyendo una interfaz simple se facilita la reusabilidad. Esta solución de diseño es la que describe el patrón de diseño OO Facade [Gamma 95] (ver detalles del patrón de diseño Facade en Apéndice I). Los clientes se comunican a través del envío de requerimientos a la "fachada" (clase *AgentFacade*). Como consecuencia tenemos que los clientes no acceden a los objetos del subsistema directamente, reduciendo el número de objetos que el cliente usa.

Así también, la clase *AgentFacade* posee dos métodos abstractos que definen la interfaz para la carga (descarga) de una agente al sistema: `load` y `unload` (Ejemplo 5 - 2), como así también las excepciones que pueden ocurrir: de entrada/salida, de definición en la clase del agente a cargar, por falta de alguna clase, etc.

```
abstract protected void load (ObjectInputStream objectinputstream)
       throws IOException,
              AgentDefinitionException,
              IllegalAccessException,
              InstantiationException,
              ClassNotFoundException,
              ClassCastException;

abstract protected void unload (ObjectOutputStream objectoutputstream)
        throws IOException;
```

Ejemplo 5 - 2 – Métodos abstractos para la carga (descarga) de un agente al entorno

Mientras que la clase *Facade* (que extiende *AgentFacade*) tiene una implementación por defecto, para la carga (descarga) de una agente al sistema, en los métodos *hook* que se muestran en el Ejemplo 5 - 3. La carga de un agente en el entorno consiste en iniciarlo por primera vez o re-iniciarlo desde un archivo, de recurso y datos, que contiene la información del agente (es el caso cuando un agente fue transferido desde un *host* a otro). El método `setup` empleado en este mismo ejemplo finaliza la carga del agente al entorno.

```
protected void load (ObjectInputStream objectinputstream)
            throws IOException, AgentDefinitionException, IllegalAccessException,
                  InstantiationException, ClassNotFoundException, ClassCastException {
 // Cargar un agente existente
 if (objectinputstream != null)  {
  agentidentity = (AgentIdentity) objectinputstream.readObject();
  agent = (Agent) objectinputstream.readObject();
 }
 // Crear un agente nuevo
 else {
  agentidentity = new AgentIdentity();
  agent = (Agent)Class.forName("framas.concreteAgent.Main").newInstance();
 }
 // Completa la carga de un agente al entorno
 setup(agent, agentidentity, boolNew);
}
protected void unload(ObjectOutputStream objectoutputstream)
          throws IOException  {
        objectoutputstream.writeObject(getAgentIdentity());
        objectoutputstream.writeObject(getAgent());
}
```

Ejemplo 5 - 3 – Métodos *hook* para la carga (descarga) de un agente al entorno

5.2.1. Percepción

Una característica de los agentes es la capacidad de percibir su contexto (definido a continuación). Este puede estar compuesto por agentes y aplicaciones. Cada una de estas entidades producen *eventos* que pueden ser percibidos por los componentes llamados *sensores*.

Cada agente tiene definido un contexto. Se observa en la Figura 5 - 9 el esquema de un entorno, compuesto por 6 agentes. Se toma como ejemplo el agente *a* (en la figura se visualiza en tamaño superior al resto) que percibe a los agentes en su contexto, es decir que percibe los eventos que éstos producen. Por otro lado, el entorno tiene otros agentes (como los dos mostrados en la figura) que están fuera del contexto del agente *a*. Para que un agente pueda incluir en su contexto a otro es necesario que solicite la autorización correspondiente enviando el mensaje `addAgentListener`.

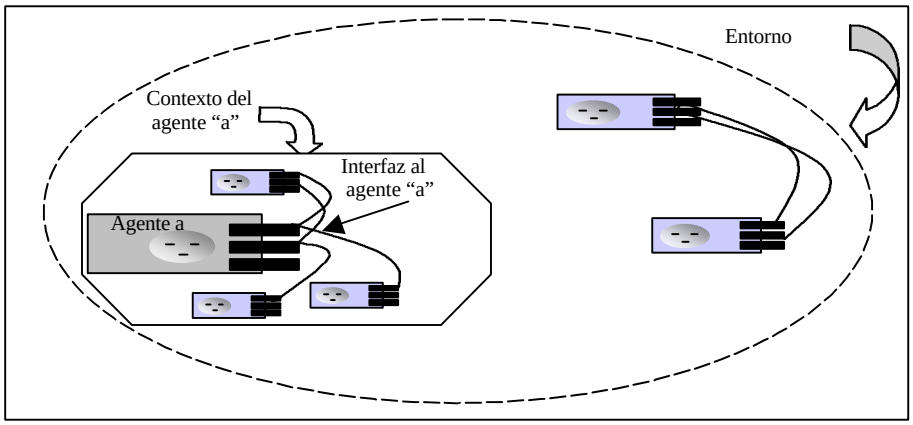


Figura 5 - 9 – Modelo de cada entorno en el SMA.

Este mensaje se define en la interfaz *EventProducerI* (Ejemplo 5 - 4) que especifica el protocolo a emplear en la comunicación agente observador ⇔ entidad observada. Para que una entidad pueda ser percibida debe mantener una lista de los observadores e implementar esta interfaz. En ella se especifica cómo los agentes observadores serán registrados (`addAgentListener`) y borrados (`removeAgentListener`) de la lista de eventos de la entidad observada; como así también establece la interfaz común para que cada entidad a ser observada notifique sobre la ocurrencia de sus eventos (`notifyAgentListener`).

```
public interface EventProducerI extends EventListener {

  public void addAgentListener(Agent agent);  //synchronized method

  public void notifyAgentListener();

  public void notifyAgentListener(EventAgent evtAgent);

  public void removeAgentListener(Agent agent);//synchronized method
}
```

Ejemplo 5 - 4 – Interfaz *EventProducerI*

Para que un agente pueda percibir a una entidad debe especificar la funcionalidad requerida en el sensor del agente, que se encuentra en el método abstracto `eventFired` (Ejemplo 5 - 5) de la clase *Agent*. Las subclases deben implementar este método con la funcionalidad para atender los eventos. Estos eventos son del tipo *EventAgent* (o subclase de éste).

```
public abstract void eventFired(EventAgent evtAgent);
```

Ejemplo 5 - 5 – Método abstracto `eventFired`: sensor de la clase *Agent* (Figura 5 – 10)

En síntesis, las entidades que se encuentran en el contexto de un agente son:

- *Agentes*. FraMaS define una implementación por defecto de la interfaz *EventProducerI*, que permite a los agentes publicar eventos que puedan ser percibidos por otros agentes. Esta implementación está ubicada en la clase abstracta *EventProducer* (Ejemplo 5 - 6).

  Básicamente, la clase abstracta define una estructura tipo *Vector* para contener la lista de agentes subscriptos a sus eventos y la implementación de los métodos para agregar (borrar) agentes: `addAgentListener` (`removeAgentListener`) respectivamente, como así también, métodos para notificar cuando ocurre un nuevo evento: `notifyAgentListener`.

- *Aplicaciones*. Para que un agente pueda observar a una aplicación, la aplicación debe ser una aplicación Java (applet o aplicación independiente) que implemente la interfaz *EventProducerI*, permitiendo así que los agentes se registren a los eventos de la misma.

Además el entono multi-agente también puede producir eventos, es por ello que se incluye al entorno como entidad a ser percibida por los agentes:

- El *entorno multi-agente* donde el agente está inserto. La clase principal del entorno, *MASServices*, tiene una implementación por defecto de la interfaz *EventProducerI*, similar a la implementación de la clase abstracta *EventProducer*, que permite a los agentes registrarse (o darse de baja) de los eventos que el entorno notifique.

Por último, los agentes pueden percibir a los usuarios:

- *Usuarios*. La percepción del usuario por parte de los agentes, se realiza a través de la observación de la interacción con una aplicación y/o con el agente del usuario en si mismo.

```
abstract class EventProducer implements EventProducerI {
  private Vector listeners = null;
  public EventProducer() {
       this.listeners = new Vector();  }
  public synchronized void addAgentListener(Agent agent) {
       listeners.addElement(agent);  }
  public void notifyAgentListener() {
       Vector aux;
       EventAgent evtAgent = new EventAgent(this);
       synchronized(this) {
              aux = (Vector) listeners.clone();
       }
       for(int i=0; i<aux.size(); i++)
        ((Agent) aux.elementAt(i)).eventFired(evtAgent);
  }
  public void notifyAgentListener(EventAgent evtAgent) {
       Vector aux;
       synchronized(this) {
              aux = (Vector) listeners.clone();
       }
       for(int i=0; i<aux.size(); i++)
        ((Agent) aux.elementAt(i)).eventFired(evtAgent);
  }
  public synchronized void removeAgentListener(Agent agent) {
       listeners.removeElement(agent);
  }}
```

Ejemplo 5 - 6 – Clase abstracta *EventProducer* (implementación por defecto de la interfaz *EventProducerI*)

Esta interfaz permite que todos los objetos derivados desde la estructura del framework, ya sean parte del agente en si o del sistema multi-agente, puedan ser observados usando el mecanismo que consiste en la suscripción a los eventos y posterior notificación cuando un evento ocurre. Resta

destacar que esta es la implementación por defecto del protocolo y puede ser extendida para soportar mecanismos más adecuados al sistema en particular.

5.2.2. Comunicación

Como se mencionó anteriormente, los decoradores cubren las tareas básicas agregando nuevo comportamiento al agente. El primer decorador que el framework puede aplicar sobre estas tareas básicas es el responsable de la comunicación. Este decorador provee las primitivas necesarias para que los agentes envíen y reciban mensajes[2]. Sin embargo, la composición de las tareas básicas y comunicación no conforman (aún) un agente; ya que por ejemplo no es autónomo, ni está dirigido por objetivos.

Si el control y los datos son centralizados, toda interacción entre agentes es a través del SMA; resultando un sistema con altas posibilidades de que se sature de mensajes a medida que estas interacciones aumentan. Por otro lado, si el control y los datos son distribuidos se puede evitar (o disminuir las posibilidades de ocurrencia de) este cuello de botella [Jennings 98], es por esto que la comunicación se hace en forma directa entre los agentes, es decir, sin la intervención del sistema multi-agente. Se emplea la invocación de métodos remota (RMI – *Remote Method Invocation*) provista en el lenguaje Java.

El diseño del framework incorpora la recepción y el envío de mensajes. Debido a que todos los agentes requieren de la comunicación, al iniciar un agente se comienza un mecanismo automático que permite la recepción de mensajes a través del método *template* remoto `receiveMessages`. El envío de mensajes se realiza a través de un *thread* de control independiente, que por cada mensaje se encarga de establecer la comunicación entre el agente origen y el agente destino.

Por esta necesidad de los sistemas de agentes de comunicarse el diseño del framework incorpora la abstracción de los mensajes, reflejado en la clase *Message*. La misma tiene información sobre el origen y el destino del mensaje, como así también un cuerpo de mensaje y una descripción de éste. Esta clase puede ser extendida por los diferentes agentes.

```
public void run() {

  String hostDestination = "//localhost";

  // Buscar dirección de destino en servidor de nombres (por defecto asume local)
  hostDestino = nameServer.addressOf(message.destination);

  try {
       ReceiveI n = (ReceiveI) Naming.lookup(hostDestination +
                  ":1099/receiveMessage"+message.getDestination().getUserName());

       n.receiveMessage(message);
  } catch (Exception e) {
       e.printStackTrace();
       // Save error in log...
       parent.save2Log("Send.run(): ERROR sending message to "+
                       message.getDestination().getUserName()+".-");

       // Save as "pending message"...
       parent.save2PendingMessages(message);
  }
}
```

Ejemplo 5 - 7 – Método *hook* para enviar mensajes

La clase responsable del envío de mensajes es denominada *Send* y hereda de la clase Java *Thread*, ejecutándose independientemente del resto del agente. Se logra así que cuando un agente

[2] estos mensajes son entre agentes y pueden ser parte de un lenguaje de comunicación entre agentes (ACL:Agent Communication Language)

envía un mensaje no espere por el resultado o que atienda explícitamente la comunicación. Esta clase posee una funcionalidad por defecto, en particular implementa el método `run` (Ejemplo 5 - 7), heredado de la clase Java *Thread*, en el que intenta establecer la comunicación con el destinatario del mensaje. Además, mantiene un registro (log) con las operaciones realizadas y actualiza un registro de los mensajes pendientes, es decir aquellos que dieron error al ser enviados.

Por otro lado se ejecuta, análogamente a lo anterior, un *thread* de control independiente al agente responsable de atender la llegada de los mensajes, clase *Receive*. Esta clase implementa la interfaz *ReceiveI* que define el protocolo a emplear para la recepción de mensajes entre agentes. Cuando el agente es iniciado se publica el método *template* remoto `receiveMessage`, que permite que otros agentes (locales o remotos) hagan referencia a él, como puede observarse en el Ejemplo 5 - 8. Este método debe ser implementado a fin de atender el mensaje recibido y así iniciar, por ejemplo, un proceso de negociación si se ha recibido un requerimiento.

```
public void run() {

  String localHostName = "//localhost";
  try {
       localHostName = InetAddress.getLocalHost().getHostName();
  } catch (Exception e) {
       e.printStackTrace;
  }
  parent.save2Log("Receive.run(): Starting "+ user+"'s server in: "+
                              localHostName+ "...");
  try {
       Naming.rebind("//darkstar.isistan.exa.unicen.edu.ar"+
                                  ":1099/receiveMessage"+ user, this);

       parent.save2Log("Receive.run(): "+ user +"'s server ready!");
  } catch (Exception e) {
       e.printStackTrace();
       parent.save2Log("Receive.run(): Exception starting "+user+"'s server");
  }
}
```

Ejemplo 5 - 8 – Método *hook* para recibir mensajes

Si bien la publicación del proceso para recibir mensajes es realizada automáticamente, al iniciar el agente, se debe detener este método cuando el agente se traslada a otro *host*. Se muestra en el Ejemplo 5.9 cómo se hace esto en general.

```
try {
       Naming.unbind("//localhost:1099/receiveMessage" + getUserName());
} catch (Exception e) {
              e.printStackTrace();
}
```

Ejemplo 5 - 9 – Ejemplo de cómo se debe detener el proceso de recepción de mensajes

Como se mencionó el diseño de la comunicación incorpora un registro de las operaciones realizadas. Para esto se almacena cada mensaje enviado y recibido. La clase *Log*, es la responsable de esta funcionalidad. Posee tres métodos base (`save, deleteAll, enumerateAll`) que permiten almacenar y borrar registros de comunicación, como así también enumerarlos.

Análogamente, el decorador de comunicación incorpora en su diseño un control sobre los mensajes que no han podido ser enviados por errores en la comunicación. Estos mensajes no enviados son almacenados para su posterior reenvío. La decisión de reenviar los mensajes puede ser tomada por el agente o por el usuario. Esta funcionalidad se hace posible a través de los métodos base: `save, enumerateAll, reSend` (incorporados a la clase *PendingMessages*); los cuales respectivamente almacenan localmente un mensaje, listan los mensajes pendientes y reenvían los mensajes que produjeron error con anterioridad.

Se observa en la Figura 5 - 10 el esquema de clases del decorador de comunicación. En particular la clase *Communication* hereda de *AdvancedBehavior* (clase base de todos los decoradores) y es responsable del envío y recepción de mensajes. Esta clase puede ser subclasificada a fin de incorporar el comportamiento particular a cada tipo de agente, como se muestra en el Capítulo siguiente, sección 6.2.4.1.

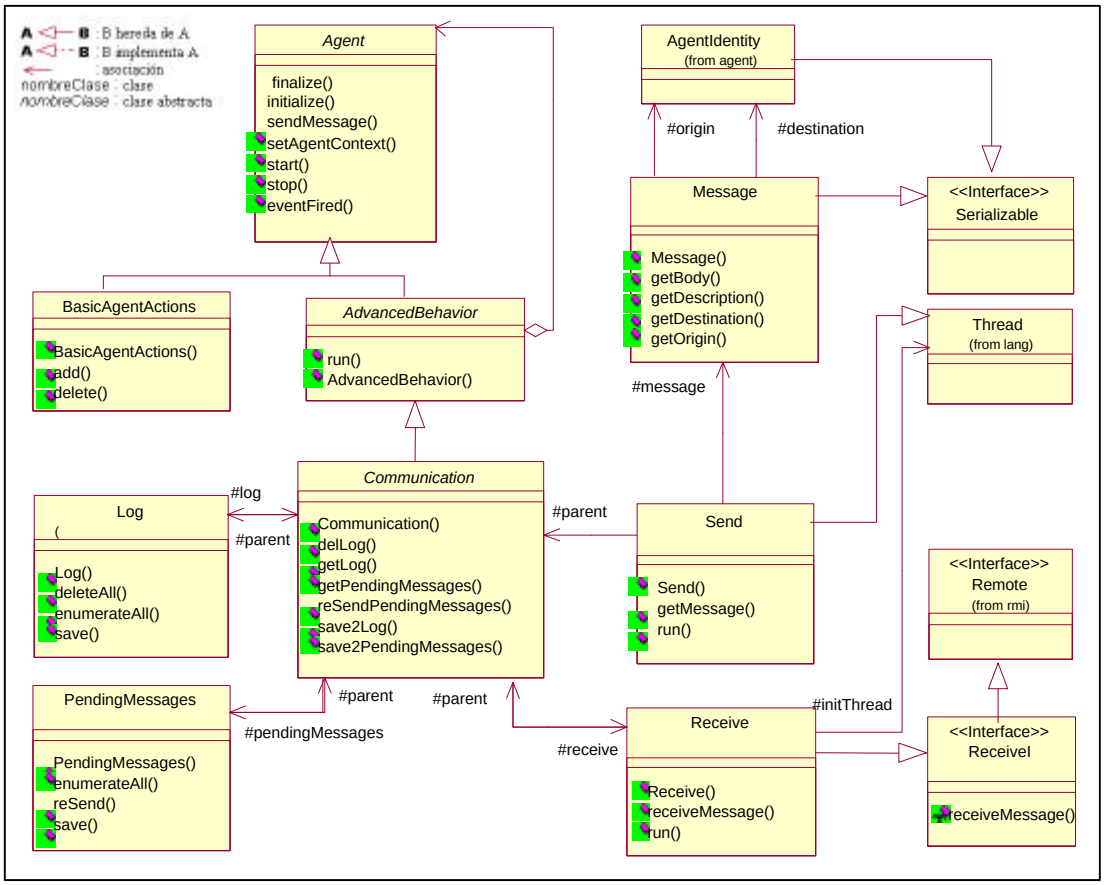


Figura 5 - 10 – Esquema de clases del decorador de comunicación

Se describe cómo se emplea el decorador de comunicación en el Agente Agenda presentado con anterioridad en este capítulo. Por un lado se implementa el método *template* `receiveMessage`, a fin de incorporar el comportamiento deseado para el Agente Agenda; y por el otro se implementa el método que detiene la recepción de mensajes, cuando el agente agenda finaliza su ejecución.

Este comportamiento permite que el Agente Agenda administre la recepción de mensajes a través de un protocolo de negociación para el envío de las invitaciones a las reuniones. Un ejemplo de protocolo es: propuesta – respuesta, es decir que se envía un pedido para asistir a una cita ("*request*") y se recibe la contestación ("*accept*" / "*deny*"). La extensión natural a este protocolo es para obtener como respuesta una contra-propuesta, en tal caso el *host* (agente que propone la cita) evalúa todas las respuestas y busca por la que permita que asistan todos los invitados. Existen varios trabajos abocados exclusivamente a los mecanismos de negociación, por ejemplo [Parsons 98].

La extensión al diseño actual de FraMaS en la comunicación esta orientada a soportar el lenguaje de comunicación KQML [Finin 97]. Cuando se habla de agentes también se hace referencia implícitamente a la interacción que ocurre entre ellos. Esta involucra compartir conocimiento, que incluye tanto el entendimiento mutuo del conocimiento, como la comunicación de dicho conocimiento. Existen autores como *Genesereth* [Singh 95] que sugieren que una entidad

es un agente de software sí y solamente sí se comunica correctamente en un lenguaje de comunicación de agentes.

Después de todo, es difícil pensar en una comunidad de agentes (entidades) que existen y se desarrollen en forma aislada solamente. KQML fue concebido como un formato para mensajes y como un protocolo para el manejo de mensajes que soporta el intercambio de conocimiento en tiempo de ejecución entre agentes.

Para que dos entidades se "entiendan", se deben solucionar al menos dos problemas: (i) traducciones entre las representaciones de los lenguajes de cada entidad; y (ii) compartir la semántica del conocimiento representado en cada lenguaje. La comunicación involucra un protocolo de interacción, un lenguaje de comunicación y un protocolo de transporte (TCP, SMTP, HTTP, etc.). Por otro lado, cada entidad posee una representación de su conocimiento (ontologías y base de conocimiento). Análogamente estas entidades pueden tener otros componentes para llevar a cabo sus acciones, como meta-conocimiento, planes, etc. que necesiten ser parte del mensaje de comunicación.

Luego los mecanismos de negociación pueden ser también definidos como componentes de la clase *Communication*. Estos deben, básicamente, establecer un protocolo de comunicación –al estilo de los soportados por KQML- para el intercambio de mensajes.

5.2.3. Analizador de preferencias

Cada agente está compuesto por una serie de decoradores que adicionan funcionalidad, cubriendo la funcionalidad básica del agente. El primer decorador –de Comunicación, es cubierto por un segundo decorador responsable de analizar las preferencias del usuario.

La idea que persigue el analizador de preferencias es que el agente se programe a si mismo con las preferencias del usuario. El agente puede obtener autónomamente el conocimiento que necesita para asistir al usuario. Comienza con un mecanismo de entrenamiento y con el comportamiento que le permite inferir desde lo que aprendió.

Este comportamiento es similar al de un asistente personal. Cuando el asistente comienza a trabajar con su jefe no le es muy útil, en el sentido de que no conoce sus preferencias, las formas en que organiza su tiempo, etc. El asistente necesita un tiempo para aprender las preferencias de su jefe, por ejemplo los horarios de reuniones y los días/horarios en los que trata determinados temas. Cada tarea que el asistente realice, y observe realizar a su jefe, o a otros asistentes más experimentados, es empleada para aprender.

La técnica de aprendizaje adoptada en el diseño del framework es la de razonamiento basado en casos (CBR: *Case-based Reasoning*). Esta técnica permite *adaptar* viejas soluciones para satisfacer nuevas *demandas* (o situaciones). Existe una relación estrecha entre la forma en que las personas resuelven problemas y el CBR [Kolodner 97], así también el CBR es ampliamente usado en el razonamiento diario. En general, cuando se resuelve un tipo de problemas que ya se trató con anterioridad resulta más simple que la primera vez, porque se recuerdan los errores que se cometieron (intentando así evitarlos) y los aciertos.

EL CBR propone un modelo de razonamiento que incorpora la solución de problemas, conocimiento y aprendizaje. Este modelo está compuesto por:

- Una *librería de casos*, con ejemplos de entrenamiento que resumen la observación de viejas soluciones, y representa el aprendizaje.
- *Conocimiento*, o interpretación, *sobre el problema*. Un CBR no puede recuperar eficientemente casos similares a menos que interprete correctamente las nuevas situaciones.
- Un *mecanismo de adaptación* entre la nueva situación y los casos almacenados es necesario porque en general ningún caso anterior será igual a la nueva situación.

- Por último, para un mejor comportamiento del razonador se emplean mecanismos de re-alimentación (*feedback*), para incorporar la información que el usuario brinde sobre las soluciones propuestas.

El fundamento del razonamiento basado en casos es que las situaciones ocurren con regularidad, es decir, lo que fue realizado en una situación anterior puede ser aplicado en una situación similar actual. Como ventaja el CBR permite la elaboración de soluciones rápidamente, respecto al tiempo para generarlas desde cero; además, el razonador puede proponer soluciones que pertenecen a un dominio de problema que no conoce completamente. Sin embargo, entre sus desventajas se encuentra que el razonador podría intentar usar los casos viejos sin validarlos correctamente en la nueva situación.

Un *caso* es la representación de un conocimiento específico sobre una situación particular (contexto), en otras palabras: indica cómo una tarea fue llevada a cabo o cómo el conocimiento fue aplicado en determinado contexto.

El razonador funciona con una librería de casos, generada desde la observación del usuario, repitiendo el ciclo mostrado en la Figura 5 - 11. Cuando se presenta una situación (o caso nuevo), el razonador emplea la librería de casos y un algoritmo de mapeo (*matching*) para recuperar los casos más similares a la situación.

Luego, se genera un orden (*ranking*) de acuerdo a la similitud entre la situación y los casos recuperados que será usado para la formación de la solución, adaptando los casos recuperados. El usuario puede evaluar la solución propuesta por el razonador y brindar una re-alimentación al razonador (*feedback*). Un nuevo caso de entrenamiento es generado, compuesto por: situación – solución propuesta –*feedback*, y almacenado en la librería de casos.

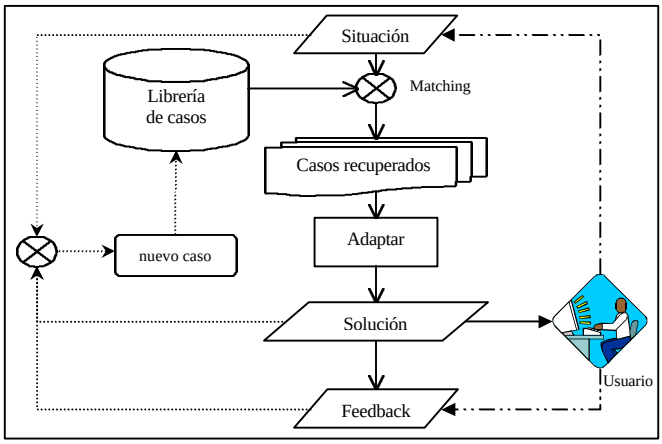


Figura 5 - 11 – Ciclo de un razonador basado en casos

FraMaS implementa un razonador basado en casos como técnica para aprender las preferencias del usuario (y/o de otros agentes). El mecanismo consiste en observar la interacción entre el usuario y el sistema, generando desde esta observación los ejemplos de entrenamiento, que son almacenados en la librería. Luego, usando esta librería y una situación actual, el razonador genera una solución. El CBR incorporado en el framework es un *wrapper* que puede ser empleado por los sistemas derivados desde el framework.

Se describe a continuación el esquema de clases que conforma el Analizador de Preferencias, como así también los métodos involucrados. Al igual que otros decoradores en FraMaS, la clase principal del analizador de preferencias (clase abstracta *PreferenceAnalizer* - Ejemplo 5 - 10) hereda de *AdvancedBehavior*; además define una asociación con un "razonador" del tipo "*CBR*".

La clase abstracta *PreferenceAnalizer* es extendida por el analizador de preferencias concreto del agente, que debe implementar los métodos abstractos heredados de *Agent*: `initialize`, `start`, `stop` y `finalize`; como así también el método abstracto heredado de *PreferenceAnalizer*: `setReazoner` (para instanciar la referencia al razonador concreto del agente que hereda de la clase abstracta *CBR* – Ejemplo 5 - 10).

```
public abstract class PreferenceAnalizer extends AdvancedBehavior {
       protected CBR reazoner;

  public PreferenceAnalizer(Agent agent, String user) {
       super(agent);
  }
  protected abstract void setReazoner(String user);
}

public abstract class CBR {
       private   PreferenceAnalizer parent;
       protected CaseLibrary        caseLibrary;
       protected Matching           matching;

  public CBR(PreferenceAnalizer parent, String user) {
       this.parent = parent;
  }
  public CaseAgent searchSolution(CaseAgent situation){
       matching.start(situation);
       return adapt(situation, matching.getRanking());
  }
  public void trainingExample(CaseAgent caseAgent){
       caseLibrary.save(caseAgent);
  }
  protected abstract void setCaseLibrary(String user);

  protected abstract void setMatching(String user);

  public abstract CaseAgent adapt(CaseAgent caseAgent, Ranking ranking);
}
```

Ejemplo 5 - 10 – Clase abstracta *PreferenceAnalizer* y clase abstracta *CBR*

La clase abstracta *CBR* está conformada por dos métodos *hook*: `searchSolution` y `trainingExample`. El primero inicia el recorrido por la librería de casos recuperando los casos *similares* a la situación actual, y con ellos genera una lista ordenada por similitud (se define a continuación). El segundo almacena un caso en la librería como ejemplo de entrenamiento. Además, están definidos tres métodos abstractos: `setCaseLibrary`, `setMatching` y `adapt` responsables respectivamente de: instanciar la librería de casos a emplear por el razonador, de instanciar la estrategia de mapeo (*matching*) a usar por el razonador para recuperar los casos más similares a una situación actual desde la librería de casos, y de generar la solución a la situación actual desde los casos recuperados.

La responsabilidad de almacenar/recuperar los ejemplos de entrenamiento a/desde la librería de casos es de la clase *CaseLibrary*. Implementa métodos *hook* para guardar un caso (`save`), y recorrer la librería (`getCase`), que retorna uno a uno los casos de la librería. Esta clase realiza el acceso a la librería a través de una implementación por defecto, sin embargo se puede extender a fin de soportar otro tipo de almacenamiento para los casos, por ejemplo una base de datos.

Se observa, en el Ejemplo 5 - 10, como el método *hook* `searchSolution` inicia el algoritmo de *matching* (método *template* `start`) y luego retorna la solución, adaptada desde la situación y el *ranking* de los casos recuperados. El algoritmo de *matching* está definido en la clase abstracta *Matching* (Ejemplo 5 - 11). Esta clase esta asociada con un caso que representa la situación (clase abstracta *CaseAgent*) y con la técnica para ordenar casos (clase *Ranking*), que es responsable de

mantener los casos más similares a la situación, generando una lista ordenada en función de la similitud (definida en cada implementación particular) entre la situación y cada caso.

```
public abstract class Matching {
       protected CaseAgent   situation   = null;
       protected Ranking     ranking     = null;
       protected CaseLibrary caseLibrary = null;

  public Matching(CaseLibrary caseLibrary) {
       this.caseLibrary= caseLibrary;
  }
  public void start(CaseAgent sit) {
       this.situation    = sit;
       CaseAgent auxCase = null;
       int       auxInt  =  0;

       while(caseLibrary.getPointer() < caseLibrary.length()) {
              auxCase = caseLibrary.getCase();
              auxInt  = similar(auxCase);
              ranking.update(auxCase, auxInt);
       }
  }
  public Ranking getRanking() {
       return ranking;
  }
  public    abstract int  similar(CaseAgent caseAgent);
  protected abstract void setRanking();
}
```

Ejemplo 5 - 11 – Clase abstracta *Matching*

Se observa en el Ejemplo 5 - 11 que el método *template* `start`, de la clase abstracta *Matching*, recupera los casos desde la librería de casos, obtiene el *peso* del caso respecto a la situación enviando un mensaje: `similar` (definido por un método abstracto de esta clase) y luego actualiza el *ranking* en función del peso.

Por último, un razonador basado en casos posee un mecanismo que le permite recibir una re-alimentación (*feedback*) sobre el éxito o fracaso de las soluciones que propuso. Si el resultado fue el esperado no es necesario un futuro análisis, pero si fue diferente a lo esperado es necesario analizar la falla a fin de explicar que pasó. Este análisis es muy importante para el razonador, ya que permite evaluar sus soluciones en el mundo real y así aprender a través del uso de casos de entrenamiento que le permitan evitar futuros errores similares. Se observa en la Figura 5 - 12 el esquema de clases del Analizador de Preferencias, como una extensión desde la clase abstracta *AdvancedBehavior*.

A continuación se describe cómo se instancia el analizador de preferencias en un agente concreto. Para ello se explica cómo este analizador es empleado en el Agente Agenda para aprender las preferencias del usuario (por detalles ver Capítulo 6, Sección 2).

Entre los requerimientos, planteados al inicio de este capítulo, para el Agente Agenda, se encuentra: aprender las preferencias del usuario respecto a tratar temas con determinado conjunto de personas, días de la semana y las horas para cada uno de ellos. De esta forma el agente puede sugerir al usuario información para completar los datos de los compromisos, evitando que deba especificar cada uno de ellos.

Por ejemplo, el usuario podría ingresar los datos correspondientes a un compromiso con la persona “x” y la persona “y” para tratar el tema “inversiones de capital”. Entonces, el razonador genera una *situación* con la lista de personas y el tema, recupera de la librería los casos *similares* y genera una *solución* (*adaptando* estos casos con la situación), que presenta al usuario con los datos del día y hora, duración del compromiso y el lugar donde se realizará. Para esto, comprobó que la solución encontrada no entre en conflicto con compromisos ya almacenados. Por último, el usuario

puede brindar una re-alimentación (*feedback*) sobre la solución propuesta, por ejemplo especificando si la reunión tuvo éxito actualizando así el caso almacenado.

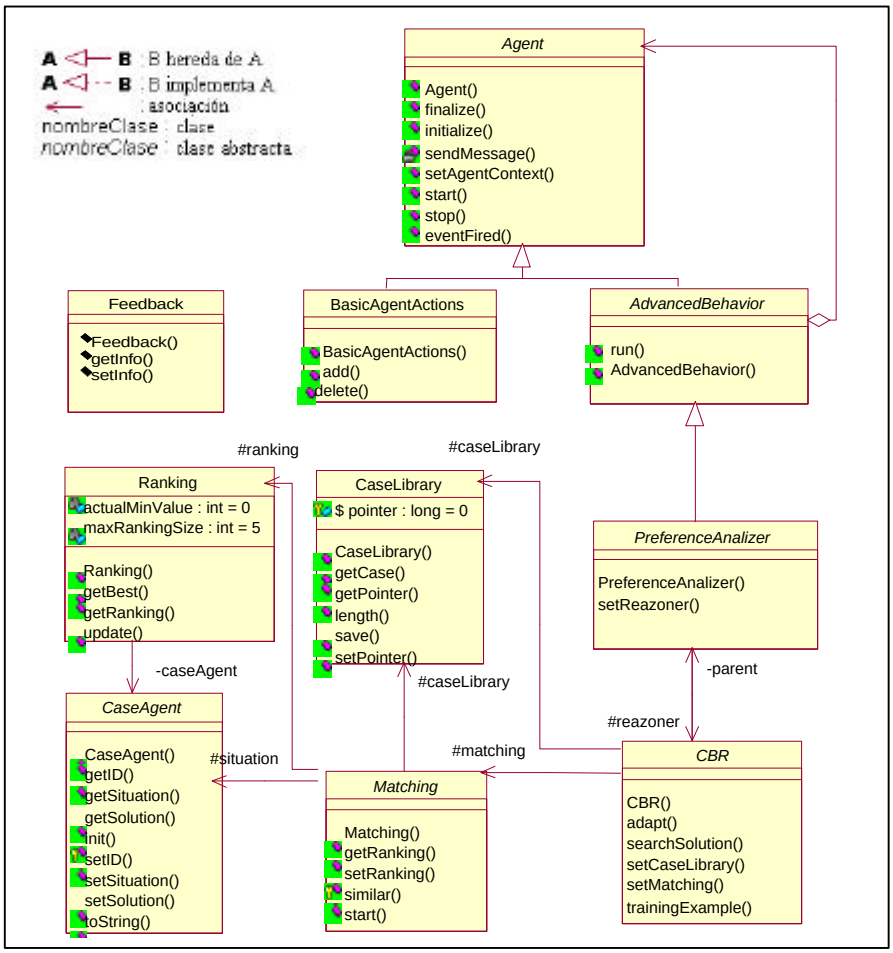


Figura 5 - 12 – Esquema de clases del decorador analizador de preferencias

Para obtener esta funcionalidad desde el analizador de preferencias, definido en el framework, es necesario extender las clases abstractas *PreferenceAnalizer*, *CBR*, *Matching* y *CaseAgent*, cada una de ellas es explicada a continuación.

Extender la clase abstracta *PreferenceAnalizer* del analizador de preferencias, en *UserPreferences*. Esta clase implementa el segundo decorador del framework y, como tal, define una capa sobre el decorador de comunicación explicado anteriormente. En particular es responsable de instanciar el razonador a emplear, y además decora al método *nueva* de la clase Agenda (que ingresa una cita a la agenda).

El Ejemplo 5 - 12 muestra cómo se hace concreta la clase abstracta *PreferenceAnalizer* del analizador de preferencias. En el constructor de la clase se instancia un razonador (clase *MS_CBR* explicada más adelante). En la decoración del método *nueva* se comprueba que la cita a ingresar esté completa (posea toda la información requerida) y no genere conflicto con una previamente almacenada, en tal caso envía el mensaje al decorador de comunicación y almacena la cita como ejemplo de entrenamiento; caso contrario usa el razonador para generar una solución a esta situación y la presenta al usuario.

```
public class UserPreferences extends PreferenceAnalizer {

  public UserPreferences(framas.agent.Agent ag, String user) {
       super(ag, user);
       reazoner       = new MS_CBR(this, user);        // Se instancia el razonador
  }
  public void nueva(Object objCita) {
       MS_CaseAgenda caseAgent = new MS_CaseAgenda();  // Nuevo caso de entrenamiento
       caseAgent.setID(new Date().toGMTString());
       caseAgent.setSituation(objCita);
       nueva(objCita, caseAgent);                   // Decorador sobre método “nueva”
  }
  public void nueva(Object objCita, MS_CaseAgenda caseAgent) {
       Cita citaN = (Cita) objCita;
       if ((citaN.completa()) && (libreSlot(citaN))) {
              Object[] args = {citaN);
              // Envía mensaje al wrapper de Comunicación
              agent.sendMessage(agent,"nueva",args);
              // Almacena ejemplo de entrenamiento en la librería de casos
              caseAgent.finalSolution = citaN;
              reazoner.trainingExample(caseAgent);
       } else {
              // Almacena ejemplo de entrenamiento en la librería de casos
              reazoner.trainingExample(caseAgent);
              // Genera un nuevo caso de entrenamiento
              caseAgent = new MS_CaseAgenda();
              caseAgent.setID(new Date().toGMTString());
              caseAgent.setSituation(citaN);
              // Busca una solución en la librería de casos
              caseAgent = (MS_CaseAgenda) reazoner.searchSolution(caseAgent);
              // Presenta la solución encontrada por el razonador y,
              // si había conflicto con el compromiso ingresado,
              // muestra además los compromisos en conflicto.-
       }
  }
}
```

Ejemplo 5 - 12 – Clase *UserPreferences* del Agente Agenda

La extensión de la clase abstracta *CaseAgent* del analizador de preferencias, en *MS_CaseAgenda*, es para abstraer la representación de que es un caso para el analizador de las preferencias del usuario del Agente Agenda. Como se muestra en el Ejemplo 5 - 13, cada caso está compuesto por una identificación, el usuario que produce el caso (*host*), una situación (compromiso que produjo el caso), una solución (si fue propuesta por el razonador) y una solución final (lo que finalmente fue almacenado en la agenda). Así también compone un caso, el día de la semana al que pertenece la cita e información del *feedback* del usuario, compuesto por el éxito o fracaso del compromiso y la diferencia entre la solución (propuesta por el razonador) y la solución final (aceptada por el usuario).

Se observa en el caso de la izquierda que el usuario ingresó todos los datos correspondientes al compromiso, con lo cual el razonador generó un ejemplo de entrenamiento. Mientras que, el caso de la derecha representa el ingreso de un compromiso “incompleto” (Situación): falta la duración del mismo y el tema de la reunión. El razonador completó los datos del compromiso en función de su librería de casos (Solución) y el usuario aceptó la misma cambiando antes la duración que le propuso el CBR (Solución final). Ambos compromisos están sin la información del *feedback* aun.

La siguiente especialización es realizada desde la clase abstracta *Matching* del analizador de preferencias, en *MS_Matching*. Esta clase implementa los métodos abstractos heredados `setRanking` y `similar` –que define el algoritmo de *matching* entre casos. Este algoritmo analiza la información de un compromiso de la librería de casos con la situación actual, asignando pesos por cada similitud

entre estos. Por ejemplo, si son del mismo día de la semana, si fueron invitadas las mismas personas, si son del mismo tema, etc.

CASO_ID 24 Feb 2000 14:06:38 GMT	CASO_ID 24 Feb 2000 14:07:26 GMT
host Martín	host Gastón
SITUACION	SITUACION
fecha Mon, 28 Feb 2000 10:00:00 GMT	fecha Fri, 25 Feb 2000 16:00:00 GMT
duracion 1	duracion
lugar ISISTAN	lugar oficina52
particip. [Hugo, José]	particip. [Mario]
tema CBR	tema
SOLUCION	SOLUCION
fecha	fecha Fri, 25 Feb 2000 16:00:00 GMT
duracion	duracion 2
lugar	lugar oficina52
particip.	particip. Mario]
tema	tema medio ambiente
SOLUCION_FINAL	SOLUCION_FINAL
fecha Mon, 28 Feb 2000 10:13:31 GMT	fecha Fri, 25 Feb 2000 16:00:00 GMT
duracion 1	duracion 1
lugar ISISTAN	lugar oficina52
particip. [Hugo, José]	particip. Mario]
tema CBR	tema medio ambiente
DIA_SEMANA 1	DIA_SEMANA 5
FEEDBACK1 null	FEEDBACK1 null
FEEDBACK2 null	FEEDBACK2 null

Ejemplo 5 - 13 – Casos pertenecientes a la librería del razonador

Por último se extiende la clase abstracta *CBR* del analizador de preferencias en la clase concreta *MS_CBR*, que implementa los métodos abstractos heredados: `setCaseLibrary` y `setMatching`, los cuales instancian la librería de casos concreta y el algoritmo de *matching* a emplear; como así también el método `adapt` responsable de generar la solución desde los casos recuperados (*ranking*) y la situación actual.

La adaptación de casos se realiza de la siguiente forma: desde un número de casos recuperados de la librería del razonador (como los del Ejemplo 5 - 13), que están ordenados según su similitud con la situación actual, el algoritmo de adaptación extrae la información del caso con mayor “peso” y lo emplea para generar la solución desde la situación actual. Por ejemplo, para completar la fecha, busca por un día libre que sea igual al día de la semana del caso recuperado, instanciando el resto de la información faltante desde el *ranking*.

Así queda implementado el razonador basado en casos desde la estructura de clases del Analizador de Preferencias. Esta estructura permite que distintos razonadores sean instanciados; por ejemplo, para aprender las preferencias de diferentes usuarios, para ello se supone un esquema donde cada usuario posee un agente que interactúa con el resto, entonces se emplea el razonador para aprender desde las interacciones de negociación entre agentes.

Si bien el framework incorpora el razonador basado en casos para analizar las preferencias, es posible usar otra técnica de aprendizaje, o una combinación de ellas. Los pasos a seguir para la incorporación de otra técnica son a través del reemplazo del decorador aquí descripto o sino agregando sobre este otro decorador con la técnica deseada. En cualquier caso no difiere de adicionar un decorador sobre los existentes como se describió al principio de este capítulo.

5.2.4. Selección de la próxima acción – Threads

Los agentes de software son autónomos, es decir, toman la decisión de realizar y realizan tareas sin la directa intervención del usuario u otros agentes. En cada instante de tiempo, los agentes tienen la capacidad para decidir que acción ejecutar en el instante siguiente.

El framework emplea un *thread* (`actionThread`) de la clase *AdvancedBehavior* (Ejemplo 5 - 14). Esta clase implementa la interfaz *Runnable* de Java, que permite una línea de ejecución propia para el *thread*.

Una estructura de control común a todos los agentes para la selección de la próxima acción se puede definir a través de un método *template* (`run`). Éste método implementado en la clase *AdvancedBehavior*, corresponde al código del *thread "actionThread"* definido en esa clase y establece la interfaz que usa cada agente para la selección de la próxima acción.

```
public abstract class AdvancedBehavior extends Agent
                                       implements Runnable {

   protected Thread   actionThread = null;
   public    Strategy strategy     = null;
   public    Agent    agent        = null;

   public AdvancedBehavior(Agent ag) {
      this.agent = ag;

      if (this.actionThread == null) {
                this.actionThread = new Thread(this, "Action");
                this.actionThread.start();  }
   }
   public void run() {
      Action action;
      while (isRunning()) {
           /* Selección de la próxima acción */
           action = strategy.select_action();
           strategy.do_action(action);
           /* Other agent actions */
      }
   }
   public void setStrategy(Strategy st) {
      this.strategy = st;
   }
}
```

Ejemplo 5 - 14 – Método *template* `run`: selección de la próxima acción a ejecutar

La estrategia de decisión que utiliza cada agente es parte de una "familia" de algoritmos encapsulados en distintas clases. Todas estas clases están relacionadas y difieren solamente en su comportamiento. Se emplea el patrón de diseño *Strategy* [Gamma 95] que tiene una estructura de clases para soportar este comportamiento (ver detalles de este patrón en Apéndice I – Patrones de Diseño).

Se define una clase abstracta *Strategy* (Figura 5 - 13) que tiene encapsulado el comportamiento de los algoritmos de decisión en dos métodos abstractos: `select_action` (del que se espera retorne una acción) y `do_action` (que ejecuta una acción). Esta clase abstracta puede ser subclasificada para así definir los algoritmos concretos, que respetarán los protocolos en ella definidos. Un algoritmo de *planning* puede ser implementado como una estrategia alternativa de decisión.

5.3. Sistema Multi-agente

El modelo de sistema multi-agente definido en FraMaS establece un conjunto de "entornos" (o *host*). Cada uno de ellos contiene agentes, los cuales pueden mudarse de un entorno a otro. Una dirección física puede contener varios "entornos". Sin embargo, dado un instante de tiempo, cada agente está registrado en uno solo y posee una identidad única.

Cada entorno brinda una serie de primitivas para el uso de los servicios disponibles, además su funcionalidad puede ser extendida para soportar nuevas responsabilidades. El entorno así definido puede tener una interfaz al usuario que representa a los agentes contenidos en él y permita la administración de recursos y la creación de agentes.

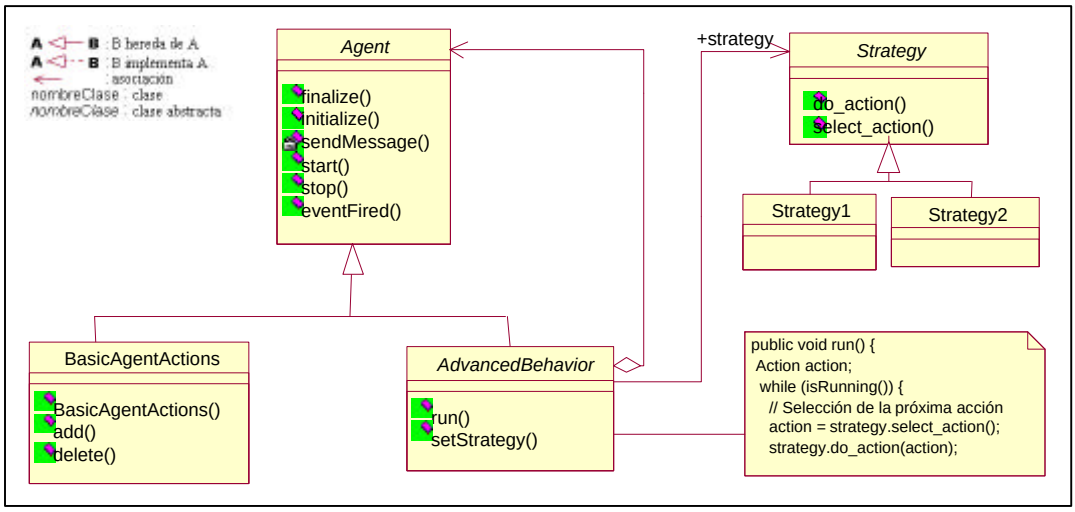


Figura 5 - 13 – Selección de la próxima acción, estructura de clases

Entre los *servicios* se mencionan:

- Primitivas para entrar y salir del entorno. Los entornos poseen una identidad y una dirección física permanente. Por otro lado los agentes tienen una identidad única pero pueden migrar de un entorno a otro, para lo cual el sistema establece un protocolo de movilidad.
- Cuando un agente ingresa a un entorno queda registrado en el mismo. Sin embargo, no esta "publicado" en él. La responsabilidad de publicarse es de cada agente, para ello dispone de los métodos *hook* `publish` y `unpublish` de la clase *AgentContextImplementation*. Por otro lado, un agente puede preguntar por los agentes publicados enviando el mensaje `getPublishedAgents` al entorno (método *hook* de *AgentContextImplementation*).
- Cada entorno puede producir eventos que serán escuchados por los agentes registrados. Análogamente al caso anterior un agente puede estar en un entorno y no percibir los eventos que ocurren en él, es decir, para que un agente en un entorno reciba los eventos debe registrarse a los mismos.

Ésta y otras características de los entornos son descriptas en esta sección. A continuación se presenta cómo es el proceso de inicio de cada entorno y en la sección 5.3.1 cómo los agentes son creados desde cero (la primera inicialización de un agente) o desde un archivo de recursos (cuando el agente detuvo su ejecución). La sección 5.3.2 describe los pasos a seguir para hacer concreta la interfaz al usuario del entorno. Por último, la sección 5.3.3 describe los protocolos para utilizar los servicios definidos en el entorno y cómo se extiende su funcionalidad.

La clase principal de cada entorno es *MASServices* (Figura 5 - 14). Esta tiene implementada la funcionalidad para crear agentes, generar eventos que son percibidos por los agentes del entorno registrados, comunicarse con otros entornos a fin de transmitir agentes, etc. Esta clase extiende de la clase Java *UnicastRemoteObject*, que provee el soporte para la referencia a objetos activos punto a punto (invocaciones, parámetros y resultados) usando TCP.

Por otro lado *MASServices* implementa dos interfaces:

- *AgentEnvironment*, define la funcionalidad básica que todo entorno de agente debe tener para transferir agentes entre *hosts*. Esta interfaz hereda de la interfaz Java *Remote*, la cual identifica los métodos que serán accedidos por una máquina virtual (JVM) no local. Cada objeto que es un objeto remoto debe implementar esta interfaz.
- *EventProducer*, establece el protocolo a emplear en la comunicación agente observador ⇔ entidad observada (en este caso el entorno multi-agente). Los eventos, al igual que lo explicado con anterioridad, deben ser del tipo *EventAgent*, o una especialización de esta clase.

Por otro lado, la estructura de clases del entorno multi-agente está compuesta por la clase *RepositoryEntry*, que está formada por dos partes: una de "recurso" (el *bytecode* correspondiente a una clase) y una de "dato" (de las variables de instancia de la clase). Así se encapsula la información correspondiente a cada clase (código + datos). Esta clase está asociada con la clase Java *File* (que representa el nombre de un archivo en el *host*).

La clase *Repository* define una colección de agentes, y es responsable de la creación de instancias de agentes. Esto es empleado, por ejemplo, cuando un agente congela su ejecución y es almacenado en disco tanto su código como sus datos.

Al igual que los agentes, cada entorno tiene una identidad asociada. Está formada por un nombre que se empleará junto con la dirección física de *host* donde se encuentra el entorno para hacer referencia a él. Varios entornos pueden existir en un mismo *host*, sin embargo no está permitido en el modelo la trasferencia de entornos entre *host* (sólo los agentes tienen movilidad).

Por último la clase *AgentFacade* (la interfaz a cada agente) está asociada con *MASServices*. Así se establece la comunicación entre el agente y el entorno. Cuando un agente quiere ser transferido a otro *host* lo solicita al entorno a través de su "fachada", luego el entorno establece la comunicación con el *host* destino y cuando la trasferencia toma lugar el entorno le indica al agente que detenga su ejecución (los detalles de la comunicación entre entornos y con el agente son indicados más adelante en esta sección)

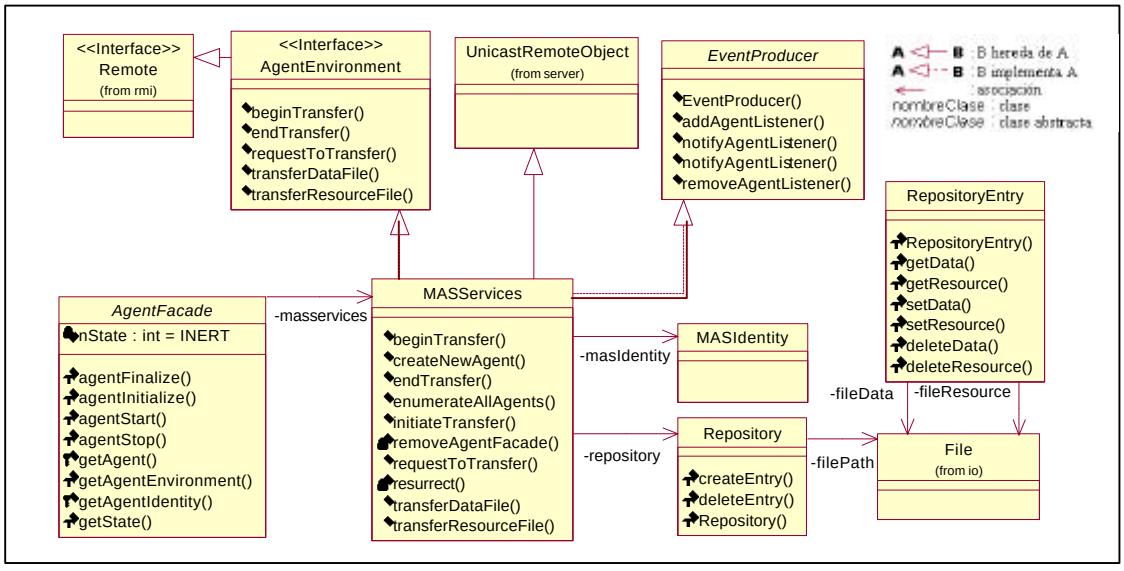


Figura 5 - 14 – Estructura de clases del entorno del SMA

5.3.1. Creación de agentes

La responsabilidad de crear cada agente es del entorno. El framework provee las primitivas requeridas para obtener esta funcionalidad. La carga de un agente al sistema puede ocurrir desde archivos de recurso y datos (es un archivo comprimido que contiene el *bytecode* correspondiente a la clase y la información con los valores de las variables de estado de la clase), o desde cero cuando el agente es iniciado por primera vez.

La creación de agentes consiste en iniciar las estructuras correspondientes al agente en si y las correspondientes al entorno. Las estructuras correspondientes al agente incluyen: la identificación del agente (clase *AgentIdentity*), el agente en si (estructura de clases desde la clase *Agent* y *AgentFacade*) donde se mantiene el estado del agente (iniciado, ejecutando, detenido o finalizado), el contexto del agente y los decoradores. Las estructuras del entorno incluyen las correspondientes a actualizar la lista de agentes en el sistema y (posiblemente) la lista de agentes publicados.

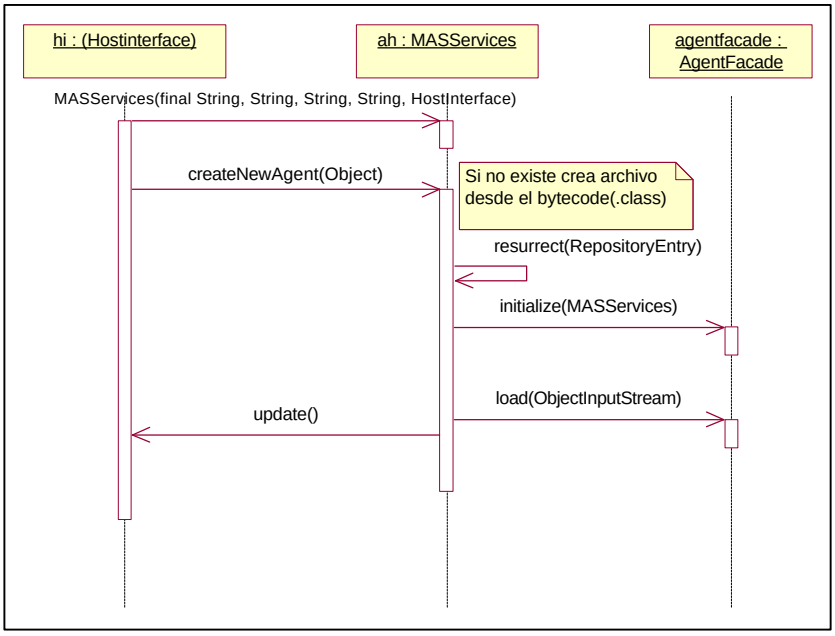


Figura 5 - 15 – Diagrama de interacción correspondiente a la creación de un agente

Se describe a continuación el proceso de creación de un agente desde una interfaz al usuario del entorno. Luego de iniciados los servicios del entorno desde la interfaz al usuario (clase *MASServices –ah* y clase *HostInterface –hi* en el la Figura 5 - 15), se puede enviar el mensaje *createNewAgent* que inicia la creación del agente en si.

El entorno verifica si existe un archivo de recurso que corresponda al agente que se quiere iniciar (caso contrario crea uno, deben existir las clases correspondientes al agente en el CLASSPATH). Una vez que el entorno posee el archivo con el *bytecode* del agente:

1. Crea un cargador para las clases del agente.
2. Crea una instancia de la fachada del agente (*AgentFacade*).
3. Inicia la fachada con la información correspondiente al entorno (método base `initialize`).
4. Lee la información del archivo de recurso, creando el agente en si (método abstracto `load`).
5. Inicia la computación del agente (`agentInitialize` y `agentStart` de *AgentFacade*).

5.3.2. Interfaz del Sistema Multi-agente

La interfaz al usuario del entorno multi-agente está incorporada en el framework para permitir realizar un seguimiento de los agentes en el entorno. Debido a que los tipos de sistema multi-agente, y los agentes en si, son muy diferentes la estructura del framework especifica sólo que cada entorno puede tener una interfaz que será una clase que especialice la clase abstracta *HostInterface*.

HostInterface define dos tipos de métodos para la actualización de la interfaz (Ejemplo 5 - 15). Entonces cuando el entorno considera que ha ocurrido un cambio que debe ser notificado a la interfaz le envía el mensaje `update`.

```
Package framas.environment;

Public abstract class HostInterface extends Frame {
        protected MASServices ah;

  public boolean handleEvent(Event event) {
        if (event.id == Event.WINDOW_DESTROY) {
            System.exit(0);
        }return super.handleEvent(event);
  }
  public abstract void update();
  public abstract void update(Object o);
}
```

Ejemplo 5 - 15 – Clase abstracta *HostInterface*

5.3.3. Servicios del Sistema Multi-agente

La clase principal de cada entorno (*MASServices*) define la funcionalidad básica con los servicios que provee. Puede ser extendida para incorporar otras responsabilidades y es responsable por la interacción entre agentes, es decir que brinda un entorno común que permite a los agentes compartir información, comunicarse, negociar y moverse de un entorno a otro.

5.3.3.1. Movilidad

Los agentes móviles han sido propuestos desde hace tiempo como una herramienta de software flexible para entornos de red. El objetivo de incorporar el paradigma de movilidad de agentes en FraMaS es permitir la ejecución autónoma de programas en *hosts* remotos heterogéneos.

El aspecto principal considerado en el diseño de la movilidad de agentes es el de sub-agentes que se trasladan de un sitio a otro. Estos sub-agentes forman parte de un agente particular. Antes de describir el protocolo de movilidad en FraMaS se compara ésta con otras arquitecturas de red, en particular con la arquitectura cliente - servidor.

Una aplicación cliente - servidor básicamente está compuesta por éstas dos partes que usualmente se ejecutan en máquinas separadas e interconectadas por una red. Cuando un cliente necesita un dato del servidor, o acceder a un servicio que éste provee, envía el pedido al servidor usando la red (Figura 5 - 16, (a)). Luego el servidor procesa el requerimiento y envía la respuesta al cliente. Cada interacción requiere de un completo uso de los recursos de la red.

Por otro lado una arquitectura de agentes móviles (Figura 5 - 16, (b)), también existe un cliente y un servidor sin embargo el cliente (o parte de él) se traslada al servidor, y realiza los requerimientos interactuando localmente, retornando cuando finaliza con las operaciones.

Las ventajas de la movilidad de código es que permite reducir el ancho de banda usado en sucesivas consultas al servidor, porque una vez que el cliente está en el entorno del servidor realiza todas las consultas localmente; y además solucionan problemas relacionados con las redes no confiables y pueden trabajar cuando la conexión deja de existir, por ejemplo una vez que el cliente está en el entorno del cliente no es necesario mantener la conexión entre el entorno del cliente y el servidor.

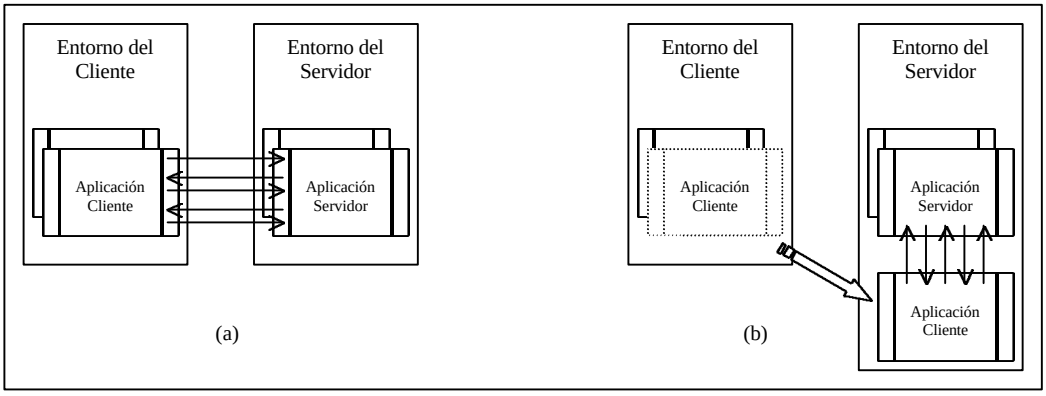


Figura 5 - 16 – Arquitectura cliente - servidor. (a) Tradicional. (b) Con movilidad de código.

En la parte inferior de la jerarquía de servicios (Figura 5 - 17) están las primitivas del sistema operativo y los servicios de la capa de red, que permiten la transmisión de código y datos sobre la red de computadoras. El siguiente nivel ejecuta el código de los programas en una plataforma independiente del sistema de computadora. A continuación sigue el framework multi-agente con la librería de clases y sus relaciones que implementan las primitivas de movilidad, tales como `requestToTrasfer(AgentEnvironment, AgentIdentity)` que indica el pedido de un agente identificado con *AgentIdentity* para ser transferido a otro entorno identificado en *AgentEnvironment*. Por último, está el nivel de los agentes móviles en si que hacen uso de estas primitivas de "alto nivel".

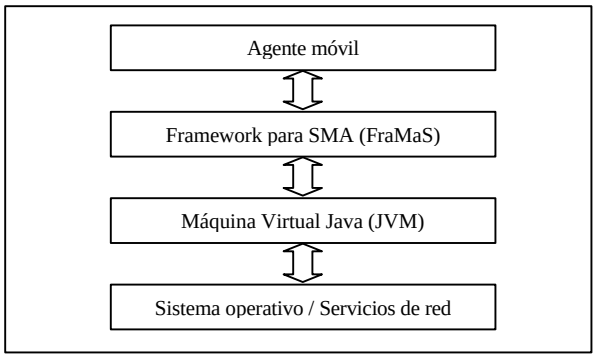


Figura 5 - 17 – Jerarquía de servicios de agentes móviles

Se describe a continuación una estructura correspondiente a las bases de un agente para comercio electrónico (ver detalles en sección 6.4). La tecnología de agentes de software puede ser usada para automatizar varios procesos, que consumen mucho tiempo, involucrados en la compra de productos. La ventaja de esta aproximación es que los agentes de software son personalizados, autónomos, etc.; estas características permiten optimizar todo el proceso de compra [Maes 99].

Se denomina este agente como: *MovingAgent*, y tiene la funcionalidad básica para trasladarse de un sitio a otro siguiendo un "recorrido" y buscando satisfacer una "lista de requerimientos". La idea es que este agente es parte de una agencia (por ejemplo un agente de interfaz) que en determinado instante de tiempo decide visitar una serie de sitios (recorrido), buscando satisfacer una serie de objetivos (lista de requerimientos). Se presenta este agente como la estructura a partir de la cual se pueden construir agentes móviles, por ejemplo para comercio electrónico.

Un agente móvil está compuesto por dos partes: el código en si mismo que define el comportamiento del agente y los datos que representan el estado de ejecución del agente (el valor de sus variables de instancia).

Cuando el agente se traslada a otro sitio ambas partes del mismo son transferidas. Para esto, se detiene la ejecución del agente y se "congela" su estado (es decir, se guardan sus variables de instancia). Luego se "empaqueta" el código, que consiste de los archivos con los *bytecodes* de las clases correspondientes al agente; y los datos con las estructuras del agente.

Este tipo de movilidad es llamada "movilidad débil" porque el programador debe explícitamente detener el agente, guardar su estado y luego reiniciar la ejecución. Esta dificultad es debido a la implementación de la máquina virtual Java (JVM) que no permite que un programa acceda a su estado interno: pila, registros, etc.

En resumen, los agentes migran de un entorno a otro. Cada entorno es responsable de brindar los servicios necesarios para que los agentes interactúen, además de empaquetar al agente para permitir su movilidad. Se muestra en la Figura 5 - 18 el diagrama de interacción de un agente que decide trasladarse desde un entorno "local" a otro entorno "remoto". Al igual que con el envío de mensajes entre agentes aquí se emplea la invocación remota a métodos provista por Java (RMI).

Al iniciar el entorno multi-agente se da comienzo el mecanismo que permite la trasferencia de agentes entre entornos. La publicación del entorno se hace a través del método *template MASServices* (constructor de la clase), esto habilita la invocación de métodos remota.

Los objetos involucrados son: un agente (`agent: MovingAgent`), la fachada del agente (`agentFacade: AgentFacade`), el contexto del agente (`agentContext: AgentContext`), el entorno local (`localHost: MASServices`), es decir el lugar dónde se encuentra el agente, y el entorno remoto (`remoteHost: MASServices`), es decir dónde el agente ha pedido ser transferido.

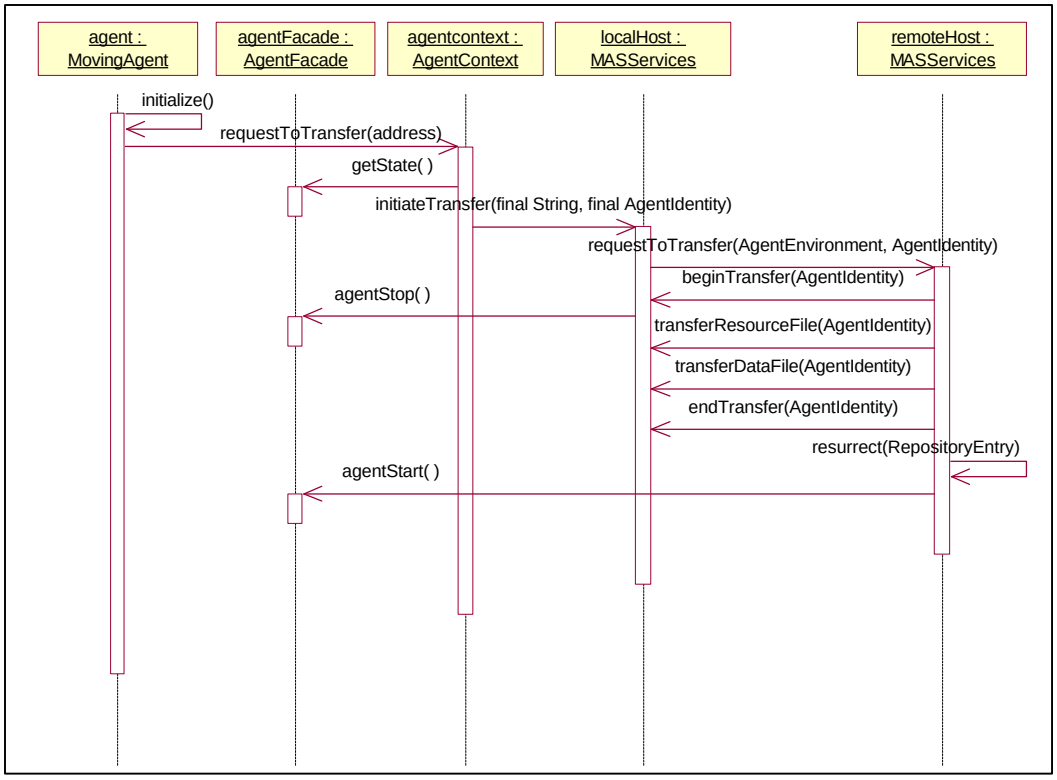


Figura 5 - 18 – Diagrama de interacción: cambio de *host* por parte de un agente

El agente envía el mensaje `requestToTransfer(address)` a su contexto (que es el responsable de la interacción con el entorno) y se verifica el estado actual del agente (`getState`), para luego enviar el mensaje `initiateTransfer` al entorno *localHost*. Este analiza la dirección destino enviando el mensaje `requestToTransfer`, dónde indica la dirección origen y la identidad del agente a transferir.

Cuando el agente es aceptado en el entorno remoto, *remoteHost*, este envía los mensajes correspondientes para la trasferencia del agente:

- Comenzar transferencia, mensaje `beginTransfer`, detiene la ejecución del agente en el entorno local para iniciar la transferencia.
- Transferir archivo de recurso, mensaje `transferResourceFile`, obtiene el código de las clases del agente y las transfiere al entorno remoto.
- Transferir archivo de datos, mensaje `transferDataFile`.
- Finalizar transferencia, mensaje `endTransfer`, reinicia la ejecución del agente en el entorno remoto.

Si bien el aspecto principal del framework es la estructura de cada agente a partir de decoradores, se proponen estas primitivas para movilidad. Existen una serie de problemas asociados a la movilidad de agentes, entre ellos de seguridad del código que se ejecutará en diferentes *host*, de robustez (es decir problemas que ocurran durante el traslado de un agente de un sitio a otro).

Estos inconvenientes pueden ser solucionados desde una especialización de la clase *MASServices* o usando un framework específico de movilidad, como por ejemplo el descripto en [Lange 98].

5.3.3.2. Publicación de agentes en el entorno

Cuando un agente es iniciado en un entorno (ya sea desde cero o porque ha sido transferido desde otro sitio) se actualizan las estructuras correspondientes del entorno que reflejan la entrada del mismo al sistema, como así también las del agente que reflejan el entorno en el que se encuentra inserto.

Sin embargo, para que un agente sea percibido por otros agentes en el entorno, éste debe "publicar" su presencia en este entorno. En la vida de cada agente en un entorno, éste se puede publicar y borrar de la lista de publicados cuantas veces crea necesario. Se establece así un protocolo, a través de métodos abstractos y *hook*, entre el contexto del agente y el entorno. El mecanismo permite que agentes sean contactados en función de los servicios ofrecidos.

Este protocolo definido en la clase abstracta *AgentContext* (Ejemplo 5 - 16) posee cuatro métodos abstractos para obtener la lista de agentes publicados (`getPublishedAgents`), obtener la identidad de un agente en particular (`getPublishedAgentIdentity`), publicar un agente (`publish`) y borrar un agente de la lista de publicados (`unpublish`). Se provee en el framework una implementación por defecto de *AgentContext* en la clase *AgentContextImplementation*. Esta clase posee los métodos *hook* que implementan estos cuatro métodos abstractos.

```
abstract public class AgentContext extends EventProducer {

  // Obtine una lista de los agentes publicados
  abstract public Enumeration getPublishedAgents();

  // Obtiene la identidad de un agente publicado.
  abstract public AgentIdentity getPublishedAgentIdentity(String strIdentifier);

  // Publica un agente en el entorno
  abstract public void publish(String strIdentifier);

  // Se elimina el agente de la lista de publicados
  abstract public void unpublish();
}
```

Ejemplo 5 - 16 – Clase abstracta *AgentContext*

5.3.3.3. Producción de eventos

Cada entorno multi-agente produce eventos que pueden ser escuchados por los agentes a través de sus sensores. El mecanismo diseñado en el framework para esto es similar al empleado cuando un agente produce un evento y es percibido por otro (según lo explicado en la sección sobre Percepción).

El entorno multi-agente implementa la interfaz *EventProducerI* (Ejemplo 5.4), lo que permite a un agente percibir los eventos que produce el entorno (siempre y cuando se suscriba al mismo a través del método `addAgentListener`). Cuando el entorno dispare un evento (*EventAgent*), lo notifica enviando el mensaje `eventFired` a las entidades registradas. Análogamente a la observación entre agentes, un agente puede dejar de recibir los eventos del entorno si así lo decide (`removeAgentListener`) y es posible establecer políticas de notificación extendiendo esta estructura.

5.4. Resumen

Se describió en este capítulo el framework para sistemas multi-agente: FraMaS. El mismo establece el sistema multi-agente como un conjunto de "entornos", cada entorno contiene agentes. Varios entornos pueden coexistir en una misma dirección física (o sitio o *host*). Los agentes pueden migrar de un entorno a otro.

Cada agente es construido a través de la superposición de "decoradores". Esto permite adicionar comportamiento a los agentes dinámicamente –en contrapartida con la herencia que establece una estructura estática. Los agentes están formados por un conjunto de acciones básicas que son decoradas con el comportamiento avanzado.

Una característica de los agentes es la percepción de su entorno, compuesto por: agentes, aplicaciones, usuarios y el propio entorno multi-agente. Para esto se proveen una serie de primitivas e interfaces que permiten la coexistencia de entidades a ser observadas y observadores. Los primeros producen eventos que son percibidos por los sensores de los agentes.

El primer decorador que establece el diseño del framework es el Analizador de Preferencias. El mismo permite, empleando un razonador basado en casos, el aprendizaje de las preferencias del usuario y, eventualmente, de otros agentes. El segundo decorador es de Comunicación y establece los protocolos a emplear para el envío de mensajes entre agentes, permitiendo construir sobre éste los mecanismos de negociación entre agentes.

Por último, se presentó en este capítulo la estructura de cada entorno multi-agente, detallando cómo los agentes son creados/iniciados (sea desde cero o desde un archivo de recurso –*bytecodes* con las clases del agente y datos –con los valores de las variables de instancia del agente), cómo se crea una interfaz al usuario del entorno, los servicios que se proveen y cómo se extienden los mismos.

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

6. Instanciaciones

FraMaS fue construido desde las aplicaciones al diseño. Se presenta a continuación la descripción de las implementaciones realizadas durante el diseño del framework: Agentes Agenda, Sistema de Agentes Forklift y la estructura de agentes para comercio electrónico.

Este capítulo describe en líneas generales los pasos a seguir en la instanciación del framework, brindando luego ejemplos de instanciación usando FraMaS de las aplicaciones mencionadas.

6.1. Introducción

El framework descripto en el capítulo anterior permite el desarrollo de sistemas multi-agente, brindando una estructura de clases y el control entre ellas que evitan tener que construir el sistema desde cero. El diseño consiste en la identificación de los agentes[1] y los servicios que el entorno multi-agente facilitará a los agentes para que colaboren entre sí.

Se describen a continuación brevemente los pasos a seguir para instanciar un sistema multi-agente usando FraMaS. Respecto a los agentes:

- Especializar la clase abstracta *AgentFacade* con la funcionalidad para crear y almacenar el estado de un agente, según los pasos descriptos en la sección anterior.
- Identificar la funcionalidad básica del agente y hacer que las clases que implementan dicha funcionalidad hereden de *BasicAgentActions*.
- Especializar si es necesario el decorador de comunicación (clase *Communication*), básicamente las clases *Receive*, *Send* y *Message*.
- Especializar el Analizador de Preferencias, en particular las clases *PreferenceAnalizer* y los métodos abstractos del razonador basado en casos.
- Los demás decoradores se pueden adicionar heredando desde *AdvancedBehavior*. Cada decorador agregado debe poseer los métodos públicos de la clase que cubre (para que sigan siendo accesibles).

Respecto al entorno multi-agente:

- Identificar los eventos que serán notificados a los agentes registrados.
- Si el entorno multi-agente brindará una interfaz al usuario, crear ésta como una especialización de la clase abstracta *HostInterface*.
- Implementar la funcionalidad de los servicios a agregar del entorno multi-agente especializando la clase *MASServices*.

El primer ejemplo de instanciación del framework presentado corresponde a un Agente Agenda. Este agente permite la administración de los compromisos del usuario, como así también comunicarse con otras agendas a fin de enviar a los invitados -a través de sus agentes agendas, las invitaciones para las reuniones. Por otro lado, el Agente Agenda, aprende las preferencias del usuario observándolo.

[1] formados por un conjunto de operaciones básicas cubiertas por un conjunto de decoradores, que adicionan funcionalidad dinámicamente a los objetos

El segundo ejemplo corresponde al sistema de Agentes Forklift. Este sistema representa a un conjunto de robots interactuando para satisfacer el objetivo del sistema, consistente en llevar cajas desde un camión a unas estanterías. Surge aquí la posibilidad de probar estrategias de *planning* y coordinación (por ejemplo para obtener un camino entre el origen y el destino, y cuando dos agentes coinciden en sus caminos respectivamente).

El tercer sistema multi-agente descripto corresponde a la estructura de agentes para comercio electrónico, en el cual se modela un conjunto de agentes interactuando en "entornos", donde los agentes pueden publicar una lista de servicios ofrecidos o consultar por los servicios ofrecidos por otros. Cada agente es guiado por un "recorrido" que sigue a fin de satisfacer sus objetivos, para luego regresar al origen.

Se describen a continuación los sistemas multi-agente correspondientes a los Agentes Agenda (sección 6.2), los Agentes Forklift (sección 6.3) y los Agentes para Comercio Electrónico (sección 6.3).

6.2. Agente Agenda

El sistema de los Agentes Agenda fue construido como parte del desarrollo del framework para sistemas multi-agente llamado FraMaS. Esta aplicación fue empleada para identificar y abstraer componentes abstractos, y sus relaciones, a fin de ser adicionadas al framework.

El Agente Agenda es la composición de una *agenda* que, básicamente, acepta el ingreso/borrado de compromisos, como así también la visualización de los compromisos almacenados; y un conjunto de *decoradores* (o *wrappers*) que cubren a la agenda, adicionando nueva funcionalidad (Figura 6 - 1).

El sistema resultante es un agente de interfaz que asiste al usuario en la administración de los compromisos, aprende sus preferencias y se comunica con otras agendas a fin de coordinar las reuniones con otros participantes.

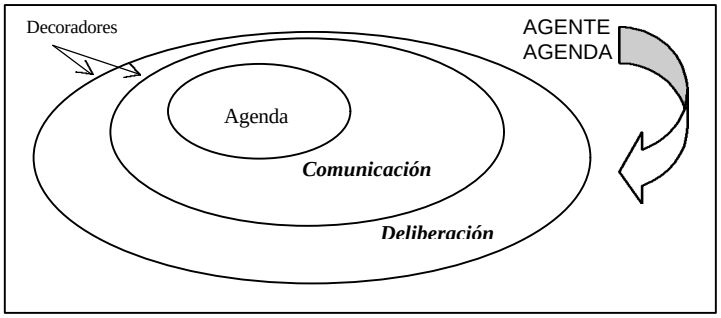


Figura 6 - 1 – Agente Agenda

6.2.1. Requerimientos

Los requerimientos planteados al Agente Agenda son:

- Brindar la funcionalidad de un sistema de agenda, es decir, ingreso/borrado de compromisos, consulta de los compromisos almacenados, etc.
- Cada *compromiso* contiene: fecha, hora, duración, tema, lugar y una, posiblemente no vacía, lista de participantes. Si la lista de participantes es vacía se refiere a una *tarea* (propia). Se emplea el término *cita* y *reunión* como sinónimos de *compromiso*.
- Dado un nuevo compromiso:
    - Si no existe conflicto con otros ya guardados se lo almacena y se envía una notificación del mismo a cada usuario en la lista de participantes.
    - Si existe conflicto, se busca una alternativa usando un razonador basado en casos, a fin de encontrar una opción según las preferencias del usuario.

- El agente administra los pedidos de invitación a los compromisos enviados, encargándose de buscar el Agente Agenda del participante invitado y reenviar el pedido si fuera necesario.
- La agenda posee una lista de lugares con los tiempos estimados para ir de un sitio a otro. Cuando se ingresa un nuevo compromiso puede ocurrir que los horarios no se superpongan pero que no sea posible llegar al sitio del compromiso porque están distantes geográficamente. En los casos así el agente sugiere alternativas usando la base de casos con las preferencias del usuario.
- El agente sugiere, por ejemplo cuando el usuario no puede llegar de un punto a otro (por cuestiones de distancia), cuando hay conflicto con otros compromisos, cuando se ingresó una cita incompleta, etc.

6.2.2. Diseño

Se describe a continuación el diagrama de clases del Agente Agenda esquematizado en la Figura 6 - 2 (las clases de la figura con fondo blanco corresponden al agente, mientras que las otras corresponden a clases de FraMaS), algunas clases son especializadas para lograr la funcionalidad deseada (como es el caso del analizador de preferencias) y análogamente son implementados los métodos abstractos a fin de instanciar el framework. Se indican con fondo gris las clases FraMaS y con fondo blanco las del Agente Agenda.

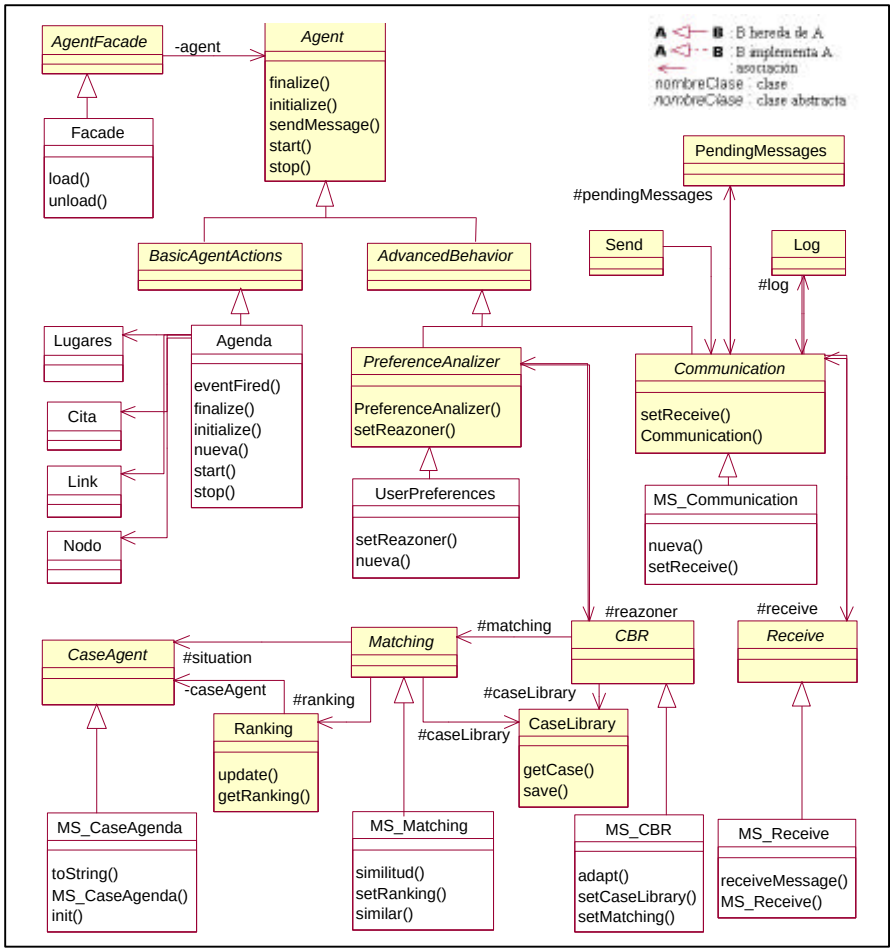


Figura 6 - 2 – Esquema de clases del Agente Agenda

Es necesario implementar los métodos abstractos de las clases del framework, en particular aquéllos para la carga del agente en el sistema. La clase abstracta *AgentFacade* tiene los métodos abstractos `load` y `unload` para la carga y descarga de un agente en el sistema, ambos descriptos en el capítulo anterior.

Estos métodos se hacen concretos en la clase *Facade*. La misma realiza los siguientes pasos para crear un Agente Agenda nuevo: (a) inicia una nueva identidad para el Agente Agenda correspondiente al usuario del entorno multi-agente, (b) instancia la funcionalidad básica del agente (en decir, los objetos correspondientes a la agenda), (c) decora las acciones básicas con la especialización del decorador de comunicación[2], (d) adiciona un nuevo decorador con un analizador de las preferencias del usuario[3] especializado desde el analizador de preferencias y, por último, (e) crea una interfaz al usuario de la agenda.

En la Figura 6 - 3 se observa una interfaz de la agenda para ingresar un nuevo compromiso. El agente está "mirando" la interacción usuario – agenda, a fin de aprender sus preferencias y brindar su ayuda cuando existen conflictos con citas anteriores, sugiriendo alternativas (basadas en el conocimiento que posee sobre las preferencias del usuario).

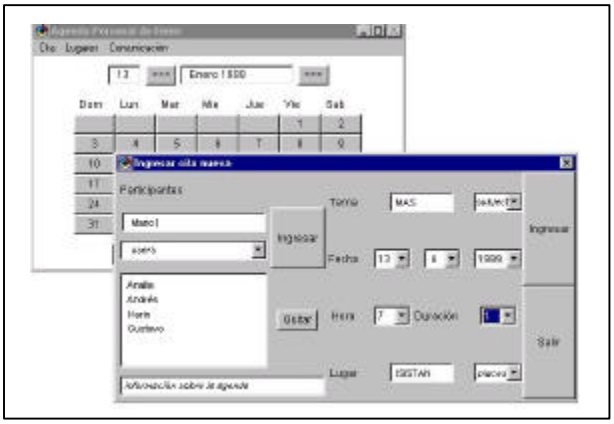


Figura 6 - 3 – Interfaz para ingresar una cita en la agenda

6.2.3. Comportamiento básico

El comportamiento básico del agente es incorporado a la estructura del framework a partir de la clase abstracta *BasicAgentActions*. Esta funcionalidad, correspondiente a la agenda es implementada en la clase *Agenda* y sus asociaciones, como especialización de la clase abstracta *BasicAgentActions* (Ejemplo 6 - 1).

Cuando la agenda se inicia recupera la información almacenada correspondiente a los compromisos del usuario, o crea las estructuras desde cero si el agente se inicia por primera vez.

Como se mencionó en el capítulo anterior, los agentes pueden tener observadores que en el agente agenda son incorporados usando el mensaje `addAgentListener`. La responsabilidad de la administración de los observadores es del contexto del agente. Análogamente, cuando ocurre un evento a ser notificado (ver Ejemplo 6 - 1 método `nueva`), se crea un evento y, se notifica a los observadores a través del contexto del agente.

El funcionamiento de los eventos y la notificación a los observadores es similar tanto en las acciones básicas, como en los decoradores. Es decir, todos los decoradores realizan el mismo procedimiento al descripto en cuanto a la notificación de eventos. Por otro lado, el método abstracto de la clase *Agent*: `eventFired` implementado por los decoradores, recibe la notificación del evento ocurrido.

[2] clase concreta *MS_Communication*
[3] clase concreta *UserPreferences*

```
public class Agenda extends framas.agentWrappers.BasicAgentActions {
       Vector  citas   = null; // Compromisos del ususrio
       Vector  part    = null; // Participantes
       Vector  temas   = null; // Temas
       Lugares lugares = null; // Grafo de con las distancias conocidas
  public Agenda(String usr) {
       try {   // Recupera información de compromisos almacenados.
              Recuperar(usr);
       } catch (java.io.IOException e) {...}
  }
  // Agrega un observador de la agenda
  public void addAgentListener(framas.agent.Agent agent) {
       getAgentContext().addAgentListener(agent);
  }
  public void nueva(Object cN) {
       /* Almacena en la agenda la cita nueva */
       framas.agent.EventAgent evt =              // Crea un evento
                           new framas.agent.EventAgent(this, citaNueva);
       getAgentContext().notifyAgentListener(evt);// Notifica el evento
  }
  public void cerrarAgenda() {
       try {
              guardar();
       } catch (java.io.IOException e) {...}
   }...}
```

Ejemplo 6 - 1 – Clase concreta *Agenda*

6.2.4. Comportamiento avanzado

Una vez desarrollado el comportamiento básico del agente se especializan los decoradores correspondientes a la comunicación y al analizador de preferencias.

6.2.4.1. Comunicación

FraMaS dispone de la funcionalidad para enviar y recibir mensajes usando la invocación de métodos remotos de Java (RMI). Sin embargo es necesario especializar la clase *Communication* para implementar el método abstracto `receiveMessage` y hacer concreta la clase abstracta *Communication* en *MS_Communication* (Ejemplo 6 - 2).

```
public class MS_Communication extends Communication {
  public MS_Communication(Agent ag, String user) {
       super(ag, user);
       setReceive(user);  }
  public void addAgentListener(framas.agent.Agent ag) {    //método decorador
       Object[] args = {ag};
       Agent.sendMessage(agent,"addAgentListener", args);
  }
  public void nueva(Object objCita) {                      //método decorador
       // Enviar mensaje a las acciones básicas del agente
       agent.sendMessage(agent,"nueva",objCita);

       for (cada_participante) {
          Message msj = new Message(this_AgentID, target_AgentID, “request", cita);
          // Guardar msj en Log
          // Enviar mensaje
          try { new Send(msj, this);
          } catch (Exception e) {...}
       }
  }
  protected void setReceive(String user) {
       try {  receive = new MS_Receive(this, user);
       } catch (Exception e) {...}
  }}
```

Ejemplo 6 - 2 – Clase *MS_Communication*

El método `addAgentListener` de la clase *Agenda* es cubierto por este decorador, reenviando el mensaje a la capa siguiente, sin adicionar comportamiento. Por otro lado el método `nueva` de la clase *Agenda* es cubierto y se adiciona el comportamiento para enviar los mensajes[4] a los participantes (igualmente, se hace el reenvío del mensaje[5] a la clase siguiente (*Agenda*) para efectivamente adicionar el compromiso a la agenda. De igual forma son "cubiertos" todos los métodos públicos de la clase Agenda (se agregue comportamiento o no).

Por último, se implementa el método abstracto heredado `setReceive` que instancia el objeto encargado de la recepción de los mensajes del tipo *MS_Receive*. Esta clase implementa otro método abstracto heredado de *Receive*: `receiveMessage` (Ejemplo 6 - 3), que atiende los mensajes recibidos. Estos mensajes son una invitación a una reunión ("*request*") o la respuesta al envío de una invitación ("*accept*" o "*deny*").

```
public void receiveMessage(Message msjRec) throws RemoteException {
       if (msj_of_request)
          // Enviar respuesta ("accept" o "deny")
      } else
          // Confirmar recepción del mensaje de respuesta
       }
}
```

Ejemplo 6 - 3 – Método `receiveMessage` (implementación del método abstracto heredado de *Receive*)

6.2.4.2. Analizador de preferencias

El analizador de preferencias del framework es especializado a fin de soportar el aprendizaje de las preferencias del usuario. Así la clase abstracta *PreferenceAnalizer* se especializa en *UserPreferences* (Ejemplo 5.3).

Continuando con el ejemplo: el método `addAgentListener`[6] es cubierto sin adicionar nuevo comportamiento. Por otro lado el método `nueva`[7] es cubierto con comportamiento genera los casos de entrenamiento, como así también brinda las suguerencias al usuario cuando el compromiso ingresado está en conflicto con otros previamente almacenados, cuando están sus datos incompletos. En general, cuando el Agente Agenda detecta la posibilidad de usar su conocimiento sobre las preferencias del usuario emplea el razonador basado en casos.

Por último, esta clase implementa el método abstracto heredado `setReazoner` que instancia el objeto razonador *MS_CBR*, que implementa el componente del razonador basado en casos concreto desde la estructura del framework.

Este asistente personal tiene la capacidad de aprender las preferencias del usuario. Estas preferencias pueden cambiar con el tiempo, para lo cual el agente debe reflejar este cambio en su conocimiento sobre el usuario. Esta funcionalidad es lograda desde *MS_CBR* a través de una librería de casos con la interacción entre el usuario y la aplicación (agenda).

La implementación del razonador consiste en la especialización de tres clases abstractas descriptas a continuación y esquematizadas en la Figura 6 - 2:

- *MS_CaseAgenda*, que define un caso de entrenamiento para el Agente Agenda (ver Ejemplo 5.4).
- MS_Matching, responsable de establecer la similitud entre dos casos[8].
- MS_CBR, hace concreta la clase abstracta CBR y tiene la funcionalidad para generar una solución desde una situación y los casos recuperados por el razonador[9].

[4] mensajes entre Agentes Agenda
[5] mensajes entre objetos, en particular del objeto decorador al objeto de las acciones básicas
[6] decorador de comunicación
[7] decorador de comunicación
[8] según la descripción presentada en el capítulo anterior
[9] según la descripción presentada en el capítulo anterior

6.2.5. Servicios del Sistema Multi-agente

El sistema multi-agente provee el entono para que los agentes interactuen. Ese entorno multi-agente donde cada asistente personal (Agente Agenda) está inserto administra las direcciones físicas de los agentes. Puede ocurrir que un agente cambie su dirección física (al mudarse de un host a otro) sin embargo su identificación seguirá siendo la misma, para ello el entorno administra un registro con las direcciones físicas.

Esta funcionalidad se logra especializando la clase *MASServices* (descripta en el capítulo anterior) a través de los métodos:

- *registrar(User, Address)*, para registrar un agente agenda en una dirección física.
- *borrar(User)*, cuando un agente sale del entorno multi-agente.
- *direcDe(User)*, solicitud de la dirección física del agente agenda de un usuario en particular.

La Figura 6 - 4 muestra el esquema de clases del entorno multi-agente correspondiente a los Agentes Agenda.

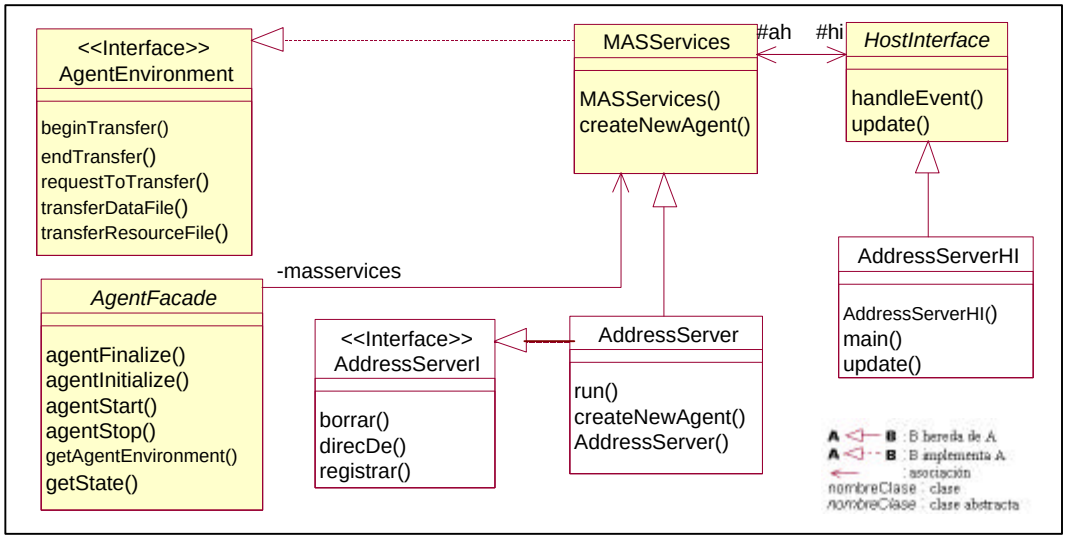


Figura 6 - 4 – Esquema de clases del entorno multi-agente para los Agentes Agenda

La especialización del entorno multi-agente que administra las direcciones físicas de los Agentes Agenda es la clase *AddressServer*, la misma implementa la interfaz *AddressServerI* que establece los protocolos de acceso al servidor de nombres. Por último, la interfaz al usuario[10] permite la creación de los asistentes personales para un usuario ingresado y posee una consola con información sobre los eventos que ocurren en el sistema (creación de agentes, llegada de pedidos).

6.3. Agente Forklift

El entorno multi-agente que contiene a los Agentes Forklift modela un conjunto de autoelevadores que tienen por objetivo llevar cajas desde un camión a las estanterías que se encuentran en un depósito. Se describe en esta sección cómo este sistema fue implementado usando FraMaS.

El objetivo de realizar este sistema usando FraMaS fue continuar la validación del framework. Estos agentes tienen características diferentes a las de los asistentes personales. Además, como este sistema fue implementado usando otras técnicas [Fischer 94a][Zunino 00] se hace viable hacer comparaciones de performance. Por último, la estructura del sistema permite la experimentación de técnicas de planning y negociación, para resolver la búsqueda del camino para

[10] clase *AddressServerHI*

descargar las cajas, y cuando dos agentes se encuentran con intenciones de moverse al mismo lugar respectivamente.

Cada agente que se desplaza dentro del depósito busca tomar una caja que encuentre en el camión para intentar llevarla a las estanterías. La decisión de que acción realizar en el próximo instante es determinada siguiendo una estrategia, por ejemplo, una función de decisión basada en prioridades para las distintas acciones. Se muestra en la Figura 6 - 5 un esquema del sistema mencionado, con los agentes Forklift, el depósito compuesto por estanterías y el camión con las cajas a descargar.

La estructura de cada agente en FraMaS está formada por un conjunto de acciones básicas, cubiertas por decoradores que agregan comportamiento dinámicamente. A continuación se describen los requerimientos y el diseño del sistema Forklift, describiendo el conjunto de acciones básicas, las estrategias de decisión y los servicios del entorno multi-agente.

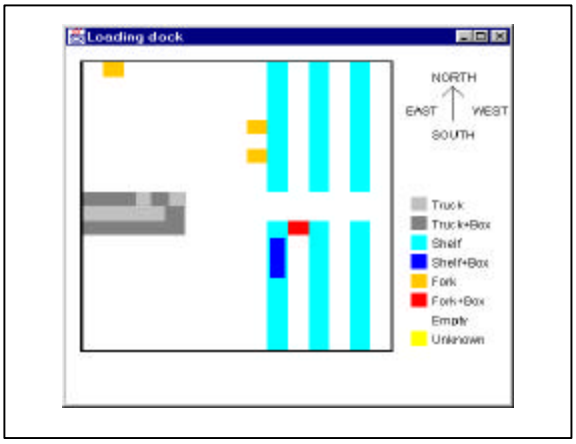


Figura 6 - 5 – Interfaz del entorno multi-agente para los Agentes Forklift

6.3.1. Requerimientos

El ambiente donde los agentes interactúan[11], es en este sistema, el modelo de un depósito dónde se mueven "autoelevadores"; está formado por una grilla de *n* por *m* casillas. Cada casilla puede contener un autoelevador o una estantería o un camión, cada uno de estos puede a su vez contener o no una caja.

Los agentes Forklift poseen sensores para percibir su entorno. Se definen cinco tipos de agentes, donde la diferencia entre uno y otro radica en el estrategia de decisión empleada para seleccionar su próxima acción y cómo resuelven sus conflictos (cuando sus caminos coinciden) :

- RWK, selecciona su próxima acción de forma aleatoria.
- BCR, utiliza una función de decisión basada en prioridades y resuelve los conflictos en forma aleatoria.
- BCH, utiliza una función de decisión basada en prioridades y resuelve conflictos de forma heurística.
- LCH, selecciona la próxima acción empleando un plan local y resuelve conflictos de forma heurística.
- LCC, selecciona la próxima acción empleando un plan local y resuelve conflictos de forma cooperativa.

[11] entorno multi-agente

Los agentes Forklift ocupan una celda de la grilla a la vez y pueden tener un rango de percepción (por ejemplo una celda por delante), se pueden comunicar con otros agentes Forklift y realizar las siguientes acciones:

- Moverse una celda en la dirección actual.
- Girar en una dirección (Este, Sur, Oeste y Norte)
- Tomar una caja (si hay delante de ellos una estantería o camión que la contenga)
- Dejar una caja (si hay una estantería o camión con lugar vacío)

Mientras los agentes realizan alguna de estas acciones pueden entrar en conflicto. Por ejemplo bloquearse uno a otro, esto ocurre cuando un agente tiene por objetivo moverse a una celda ocupada por otro agente.

6.3.2. Diseño

Análogamente al ejemplo anterior, el diseño comienza estableciendo el conjunto de acciones básicas y los decoradores que cubrirán esas acciones, además de la implementación de los métodos abstractos del framework. Para los agentes Forklift es preciso también definir las estrategias de decisión a fin de obtener la funcionalidad de cada tipo de agente (RWK, BCH, etc.). En la Figura 6 - 6 se muestra la estructura de clases de los Agentes Forklift. Se indican las clases con fondo gris cuando corresponden al framework y con fondo blanco cuando corresponden al Sistemas Forklift.

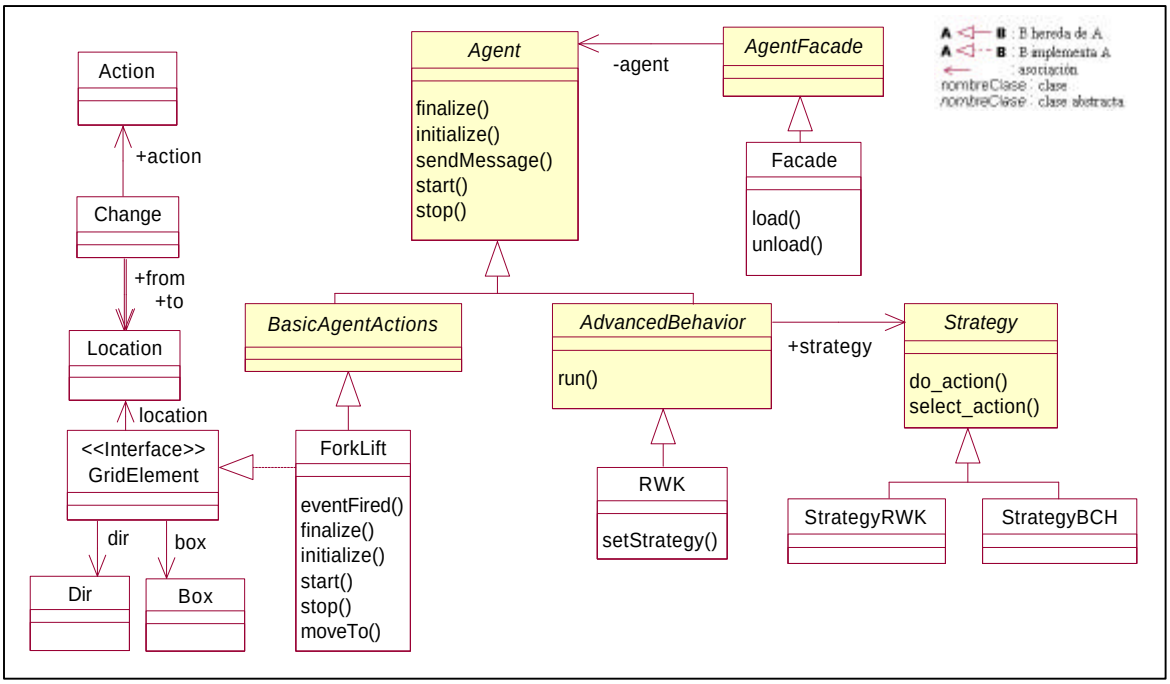


Figura 6 - 6 – Esquema de clases del Agente Forklift

Como en el Agente Agenda descripto en la sección anterior es necesario especializar la clase abstracta *AgentFacade* para implementar sus métodos abstractos `load` y `unload`. Estos, realizan la carga (descarga) de los agentes en el entorno multi-agente.

La clase concreta *Facade* es la responsable de esta funcionalidad, los pasos que sigue para cargar un agente Forklift al entorno son: (a) iniciar una nueva identidad para el agente Forklift, (b) instanciar la funcionalidad básica del agente (en decir, los objetos correspondientes al autoelevador –Forklift), (c) cubrir las acciones básicas con un decorador responsable de instanciar la estrategia que empleará el agente para decidir qué acciones llevar a cabo (Figura 6 - 6: clase concreta *RWK*).

6.3.3. Comportamiento básico

El comportamiento básico consiste en el conjunto de acciones que los autoelevadores pueden realizar, es decir moverse en la dirección actual, girar, tomar y dejar una caja. Estas acciones son realizadas enviando un mensaje al entorno multi-agente responsable de mantener una estructura con la posición de cada agente Forklift.

Si la acción solicitada es válida[12] el cambio en el estado del mundo se produce. La única acción que siempre tiene éxito (es decir, que nunca produce un estado del mundo inconsistente) es girar en una dirección. Porque el agente gira sobre si mismo, sin cambiar su ubicación. Las otras tres acciones pueden no tener éxito por más que el autoelevador intente realizarla (allí es donde puede intervenir un decorador para decidir tomar otra acción).

Esta funcionalidad es implementada en la clase llamada *Forklift* (Ejemplo 6 - 4) que extiende de la clase abstracta *BasicAgentActions* del framework.

Forklift implementa la interfaz *GridElement* que define el protocolo que siguen todos los elementos que están en el depósito (autoelevadores, camión y estanterías). Cada "grid element" representa una posición (*x, y: Location*) dentro del depósito de *n* x *m* celdas. Además establece que cada una de estas posiciones puede contener una caja (*Box*) y una dirección (*Dir*).

```
public class ForkLift extends    BasicAgentActions
                      implements GridElement, Constants {
  public ForkLift(Location l, Box b, MASServices ld, Dir d) {
  }
  public Boolean moveTo() {
       if (loadingDock.moveTo(dir, this)) {
              return new Boolean(true);
       } else return new Boolean(false);
  }
  public void turnTo(UtilForks.Dir d) {
       this.dir = d;
  }
  public Boolean graspBox() {
       if (box == null)
              return new Boolean(loadingDock.graspBox(this));
       else return new Boolean(false);
  }
  public Boolean putBox() {
       if (box != null)
              return new Boolean(loadingDock.putBox(this));
       else return new Boolean(false);
  }
  public void finalize()  {...}
  public void initialize(){...}
  public void start()     {...}
  public void stop()      {...}
}
```

Ejemplo 6 - 4 – Clase *Forklift* (acciones básicas del agente Forklift)

6.3.4. Comportamiento avanzado

El Agente Forklift tiene definido un decorador sobre las acciones básicas que implementa el método abstracto heredado de *AdvancedBehavior*: `setReazoner`. Este método es responsable de instanciar la estrategia de decisión concreta que empleará el agente, según la estructura definida en el método template de *AdvancedBehavior*: `run`[13].

[12] no produce un estado inconsistente, por ejemplo dos agentes en un mismo sitio
[13] explicado en el capítulo anterior

6.3.4.1. Selección de la próxima acción

La relación entre un agente y una estrategia de decisión responsable de determinar qué hacer en el tiempo siguiente, es registrada a través del método template `run`, que invoca a un método abstracto `selectAction` (Ejemplo 6 - 5) del que se espera que retorne una acción (o un conjunto de acciones), que serán ejecutadas en el tiempo siguiente.

```
public Action selectAction() {
  int rint = (int) (Math.random() * 10);

  if ((rint >= 0) && (rint <3))
        return new Action(Action.MOVETO);
  else if ((rint >= 3) && (rint <5))
        return new Action(Action.TURNTO);
  else if ((rint >= 5) && (rint <8))
        return new Action(Action.GRASP_BOX);
  else if ((rint >= 8) && (rint <=10))
        return new Action(Action.PUT_BOX);
}
```

```
public void doAction(Action a) {
  int action = a.getAction();
  switch (action) {
   case 0 : agent.sendMessage(
               agent,"moveTo", null); break;
   case 1 : agent.sendMessage(
               agent, "turnTo", dir); break;
   case 2 : agent.sendMessage(
               agent, "graspBox", null); break;
   case 3 : agent.sendMessage(
               agent, "putBox", null); break;
}}
```

Ejemplo 6 - 5 – Métodos `selectAction` y `doAction` para el agente Forklift (RWK)

Los algoritmos de decisión se abstraen por subclasificación de la clase abstracta *Strategy*. Estas subclases implementan los algoritmos concretos que respetan los protocolos definidos en la clase padre. Por ejemplo, un algoritmo de planning es implementado como una estrategia alternativa de decisión.

En la Figura 6 - 6 se muestra el esquema de clases del Agente Forklift con dos subclases de *Strategy*: *StrategyRWK* y *StrategyBCH* que implementan respectivamente las estrategias de los Agentes Forklift RWK[14] y Forklift BCH[15].

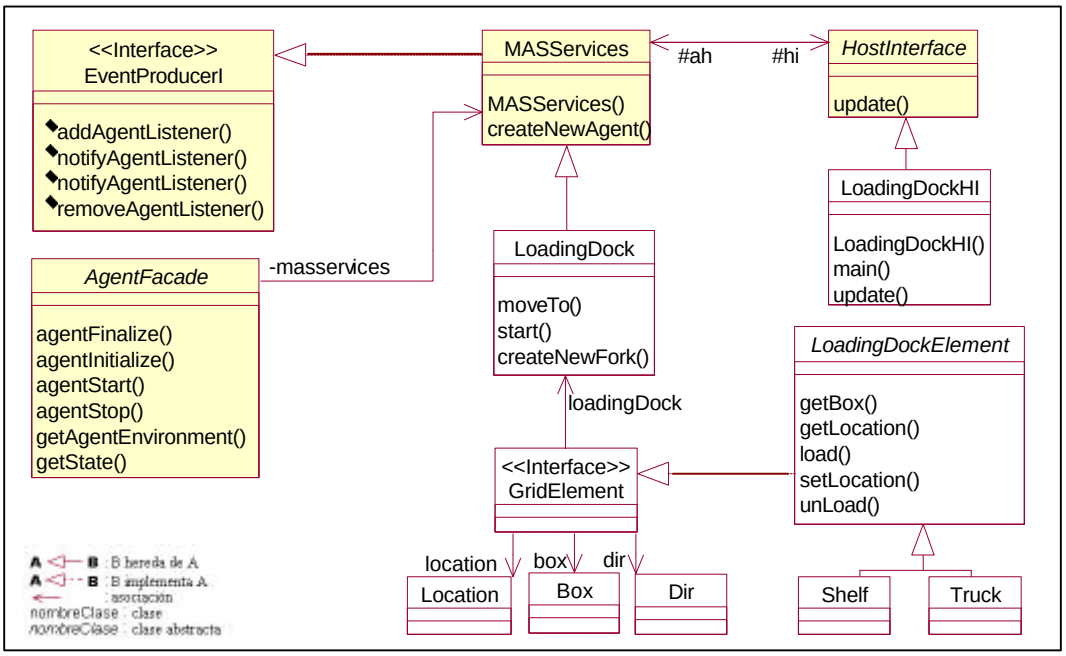


Figura 6 - 7 – Esquema de clases del entorno multi-agente de los Agentes Forklift

[14] selecciona su próxima acción de forma aleatoria

[15] que utiliza una función de decisión basada en prioridades y resuelve los conflictos en forma aleatoria

6.3.5. Servicios del Sistema Multi-agente

El entorno multi-agente en FraMaS es el responsable de proveer los servicios necesarios para que los agentes realicen sus interacciones, en particular para los Agentes Forklift son los servicios que el modelo del depósito (*LoadingDock*) brinda a los autoelevadores (*Forklift*).

La clase *MASServices* es responsable por estos servicios (por ejemplo, la notificación de los eventos que ocurren el entorno a través del protocolo definido en la interfaz *EventProducerI* explicado en el capítulo anterior), además en este sistema se especializa dicha clase a fin de soportar las estructuras de datos con la información de agentes, camión, estanterías y cajas; como así también el movimiento de los agentes y cajas (clase *LoadingDock*: Figura 6 - 7).

La interacción entre el entorno multi-agente y los agentes tiene lugar a partir de la "fachada" de los agentes (clase abstracta *AgentFacade*). Entre los servicios agregados en esta subclasificación (clase concreta *LoadingDock*) se enumeran:

- Ingresar un Agente Forklift al modelo del depósito.
- Mover, cargar y descargar un Agente Forklift.
- Lista de objetivos por cumplir, en particular las cajas que quedan por descargar desde el camión a las estanterías.

La funcionalidad particular de la interfaz al usuario del sistema Forklift es subclasificada desde la clase abstracta FraMaS: *HostInterface* en la clase concreta *LoadingDockHI*.

Así también, es necesario contar con una estructura que represente los cambios que ocurren en los Agentes Forklift para notificar a la interfaz, como por ejemplo movimientos, carga de una caja, etc. En la Figura 6 - 6 se muestra la clase *Change* que posee dicha funcionalidad; la misma contiene la acción que produjo el cambio (por ejemplo descargar una caja), el origen (determinado autoelevador), el destino (un lugar en las estanterías) y la ubicación que corresponde en la grilla al cambio producido. Cuando un cambio ocurre en el entorno se notifica a la interfaz del usuario a través del envío de un mensaje.

Por último, el diseño se completa con un tipo abstracto que define la estructura de comportamiento de todos los elementos (no agentes) que están en la grilla del depósito, clase abstracta *LoadingDockElement*. Esta clase implementa la interfaz *GridElement* y es subclasificada por las clases que contienen la funcionalidad del camión (*Truck*) y las estanterías (*Shelf*).

6.4. Estructura de un Agente para Comercio Electrónico

La evolución actual de la industria dirige sus líneas hacia el comercio electrónico. Sistemas con diversas aplicaciones para la venta electrónica están actualmente en línea, es decir disponibles en Internet. Por ejemplo, la librería electrónica Amazon.com, rei.com son algunos ejemplos de estas aplicaciones cuya cantidad y diversidad crece continuamente.

Dadas las características de las aplicaciones para comercio electrónico la Ingeniería del Software basada en Agentes posee una forma viable de analizar estos sistemas, que pueden ser vistos como un conjunto de agentes (compradores, vendedores) que interactuan a fin de satisfacer sus objetivos. Estos agentes, insertos en un entorno, están distribuidos en distintas computadoras conectadas en red.

Si bien los agentes de software fueron usados originalmente como filtros de información, para contactar a personas con intereses similares y automatizar el comportamiento repetitivo [Maes 99], hoy en día el espectro de aplicación de éstos se amplió.

Sin embargo, el potencial de Internet para la transformación de la forma de hacer negocios aún no ocurrió, ya que las compras electrónicas no son totalmente automáticas. Así la tecnología de agentes puede ser usada para automatizar muchos de los procesos involucrados en una transacción comercial. Más allá de esta realidad, la información sobre productos y vendedores es más accesible.

Se presenta un sistema multi-agente para comercio electrónico. En particular, se describe el mismo a través de un ejemplo, que consiste en la compra de artículos a través de una red de computadoras interconectadas. Se cuenta con un número de Agentes Compradores que recorren los

negocios (sitios interconectados) en busca de satisfacer requerimientos tales como: la compra de artículos (dado un límite de recursos –dinero, a emplear), buscar ofertas de los artículos usualmente comprados por sus usuarios, etc.

Se plantea en esta sección la estructura de este sistema, describiendo cómo el framework es instanciado para obtener la funcionalidad mencionada.

6.4.1. Requerimientos

El sistema está compuesto por:

- *Aplicación de administración de menúes*, que elabora las comidas de los próximos días. Para esto utiliza una base de datos con recetas, en las que se dispone de los ingredientes que insume cada preparación, las calorías, forma de preparación, características generales (entrada, plato principal, postre), etc. Dispone de una base de datos con recetas e ingredientes en existencia. Además, esta aplicación genera una lista de compras en función de las comidas planificadas al momento.
- *Agente de Compra*, que observa la interacción del usuario con el sistema de administración de menúes. Éste mantiene una Librería de Casos con las compras anteriores y lista de Preferencias ingresadas directamente por el usuario. Se emplea un razonador basado en casos para aprender las preferencias de comidas, desde la interacción del usuario con la aplicación de menúes, para la compra de artículos en determinados supermercados, o la compra de artículos de determinada marca, etc.
- *Agente Buscador*, sección móvil de los Agentes de Compra, que se desplaza entre los sitios según un recorrido en busca satisfacer una lista de artículos.

El Agente de Compra es el encargado de recibir la retroalimentación del usuario respecto a las compras realizadas, las elecciones hechas de los negocios y el agrado o no de los artículos comprados; de esta forma se completa el ciclo del razonador para la Librería de Casos de las compras realizadas.

Los negocios que disponen de artículos para ser vendidos en el ámbito electrónico se registran en un catálogo, así los Agentes Buscadores pueden reconocer las opciones que tienen en determinado momento. Cada uno de estos entornos multi-agente (supermercado virtual) dispone del ambiente para brindar servicios a los Agentes Buscadores; tales como recibir y enviar el agente de/a otro sitio, la posibilidad de contactar a otros agentes en función de los servicios que ofrece.

6.4.2. Diseño

Siguiendo el esquema de diseño de los agentes FraMaS se divide el mismo en acciones básicas (correspondientes a la aplicación que administra los menúes) y comportamiento avanzado (Agente de Compra).

Existen varias teorías que describen el proceso de compra, en las que se identifican[16] los siguientes pasos:

1) Identificación, del comprador, vendedor y productos.
2) Catálogos de productos ofrecidos, con descripciones, comparaciones, etc.
3) Catálogos de negocios, para determinar dónde se puede comprar.
4) Estrategias de negociación, para definir cómo se realiza una transacción comercial.
5) Formas de pago y distribución de la mercadería comprada.
6) Servicio y evaluación sobre los productos comprados, tales como garantía de calidad y satisfacción del comprador.

[16] creando una versión simplificada del proceso de compra

6.4.3. Comportamiento básico

La funcionalidad de la aplicación de menúes hereda de la clase FraMaS *BasicAgentActions*, y contiene las acciones básicas del Agente de Compra. Esta aplicación mantiene la estructura con las comidas planificadas para los próximos días, una base de datos con recetas (donde se especifican los ingredientes necesarios para elaborar las comidas) y un listado con la existencia de ingredientes.

6.4.4. Comportamiento avanzado

El Agente de Compra emplea el razonador basado en casos definido en el framework para aprender las preferencias de comidas, lugares de compra, marcas, etc. Para esto se especializa el Analizador de Preferencias explicado en el capítulo anterior (similar a la instanciación del Agente Agenda).

En el método template `run` (clase *AdvancedBehavior*), se selecciona la acción a realizar en el próximo instante de tiempo a través de una estrategia de decisión subclasificada desde *Strategy*. Periódicamente se controla si faltan ingredientes para los menúes, si es así se envía un Agente Buscador con una lista de negocios a recorrer (usando la base de casos con los lugares más frecuentes de compra) y la lista de ingredientes a comprar.

El Agente de Compra cubre la aplicación de menúes proveyendo a ésta con la capacidad de utilizar el conocimiento de compras pasadas. También se puede utilizar la Librería de Casos de compras anteriores para predecir futuras compras, "adelantadas" en función de lo aprendido.

El Agente Buscador es construido desde el framework a fin de emplear la estructura de clases y control definida para movilidad. Este agente tiene la capacidad de consultar por los negocios que están disponibles para el comercio electrónico, seguir un recorrido en función de estos y la lista de compras. Además mantiene una lista de supermercados propia, es decir aquellos en los que ha comprado (para tener consideraciones de costos por el uso del servicio, "calidad en la atención", confianza en ellos).

6.4.5. Servicios del Sistema Multi-agente

Los Agentes Buscadores realizan las consultas de compras y ofertas sobre la base de datos que provee cada negocio, además de las primitivas heredadas[17] con la funcionalidad para que un agente se ejecute en su entorno y para que pueda ser enviado a otro sitio.

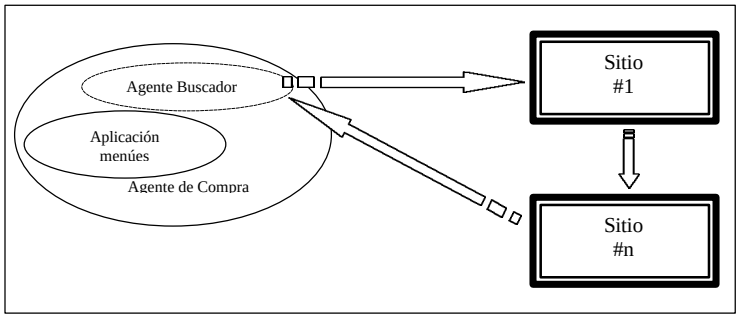


Figura 6 - 8 – Esquema general del Agente de Compra

Estas características generales del sistema de comercio electrónico[18], para la compra de artículos en negocios representados por computadoras conectadas en red y el empleo de razonamiento basado en casos para aprender las preferencias del usuario, se planteó con el objetivo de observar la viabilidad del uso del framework para este sistema.

En un sistema de comercio electrónico, como en cualquier otro en el que se intercambien bienes entre compradores y vendedores, surgen problemas de confianza entre los participantes, es

[17] de la clase *MASServices*
[18] entorno multi-agente

decir cómo saber si el otro es quien dice ser, cómo comprobar que el producto comprado es realmente lo esperado, cómo asegurar la cadena de pagos, etc. Existen métodos que permiten la realización de transacciones seguras entre participantes no identificados (no es necesario que el comprador y/o el vendedor se aseguren con quien realizan la transacción).

6.5. Resumen

El objetivo de este capítulo es presentar a través de ejemplos los pasos requeridos para emplear el framework en la construcción de un sistema multi-agente. Éstos consisten en la subclasificación de las clases abstractas (como por ejemplo *Receive*, *AgentFacade*), la implementación de los métodos abstractos heredados (como por ejemplo `setStrategy` de la clase *AdvancedBehavior*), así también como también identificación del conjunto de acciones básicas y el comportamiento avanzado de los agentes.

Por el lado del entorno multi-agente es necesario establecer los eventos que producirá el sistema y serán notificados a los observadores, implementar (si existe) la interfaz al usuario del sistema y, eventualmente, extender la funcionalidad del entorno para soportar otros servicios a ser ofrecidos a los agentes.

La primer aplicación descripta corresponde al Agente Agenda. Un asistente personal que ayuda al usuario en la administración de sus compromisos, para ello se comunica con otras agendas para enviar las invitaciones de participación a las reuniones y aprende las preferencias del usuario usando un razonador basado en casos.

La segunda aplicación es el sistema multi-agente compuesto por los Agentes Forklift. Se trata del modelo de un conjunto de robots que colaboran a fin de satisfacer el objetivo global que consiste en llevar cajas de un sitio (camión) a otro (estanterías) dentro de un depósito. Este sistema permite la experimentación de estrategias de negociación (cuando dos o más agentes coinciden en sus caminos) y planning (para decidir que acción ejecutar en el próximo instante de tiempo).

Por último, se describe brevemente las características de la estructura para comercio electrónico desde el framework. Usando Agentes de Compra (que aprenden las preferencias del usuario), Agentes Buscadores (que recorren un número de sitios en busca de satisfacer sus objetivos, que consisten en una lista de compra y –seguramente restricciones sobre esos elementos) y los entornos en si (que representan los negocios virtuales).

Este framework fue construido desde aplicaciones al diseño, siguiendo un proceso iterativo. Las aplicaciones aquí descriptas participaron en el diseño del framework y fueron empleadas a fin de identificar los componentes y las relaciones que se incluyeron en él. La descripción de las aplicaciones corresponden al último diseño de las mismas, ya que éste fue cambiando en cada iteración del desarrollo del framework.

7. Especificación formal en Object-Z

Este capítulo presenta la especificación de los aspectos más importantes del framework para Sistemas Multi-agente FraMaS usando el lenguaje *Object-Z*. Este lenguaje es una extensión del lenguaje formal *Z* diseñado para facilitar la especificación de sistemas Orientados a Objetos.

Se introducen las características principales de *Object-Z* en relación con el lenguaje formal Z; y se describe el framework, primero, en relación a la estructura de los agentes y, a continuación, al entorno multi-agente.

7.1. Introducción

El lenguaje de especificación formal *Object-Z* [Duke 91] es una extensión del lenguaje formal *Z* [Wordsworth 92]. Ha sido diseñado para especificar conceptos involucrados en los sistemas Orientados a Objetos, tales como las *clases*, *objetos*, *herencia*, etc.

Una especificación Z define un número de esquemas de estado y de operación. Determinar qué esquemas de operación pueden afectar a qué esquemas de estado implica examinar cada operación.

Un programa OO es modelado como una colección de objetos, tales como expresiones, métodos, declaraciones, clases y objetos dinámicos (instancias de clases) [Dong 96]. Una especificación Orientada a Objetos describe un sistema como una colección de objetos que interactúan, cada uno de los cuales tiene una estructura interna y un comportamiento.

Object-Z asocia cada operación con un esquema de estado particular, entonces la definición de los esquemas de estado con sus operaciones asociadas representa una *clase*. Así una especificación de un sistema en *Object-Z* está formada por un número de definiciones de clases posiblemente relacionadas por herencia (en el Apéndice II se presenta la sintaxis de *Object-Z*).

7.2. Especificación

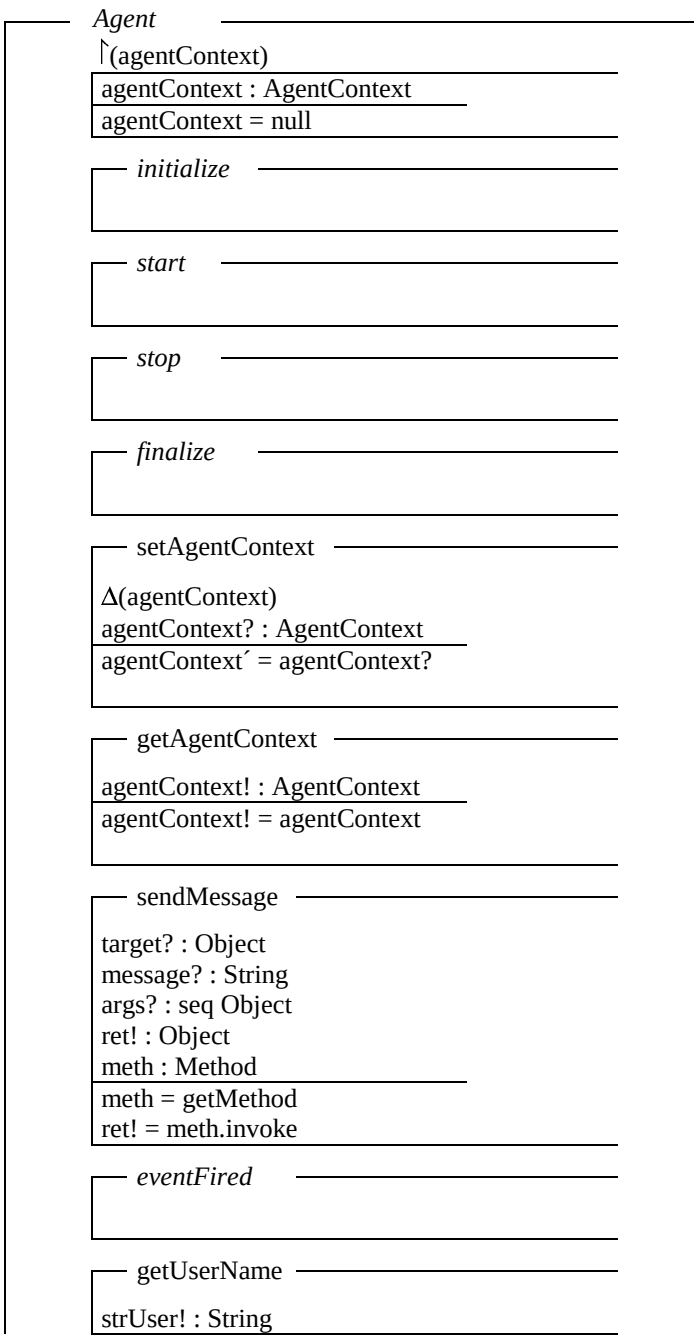
Un sistema multi-agente construido usando FraMaS está compuesto por *agentes*, diseñados desde un conjunto de acciones básicas y una serie de decoradores que adicionan responsabilidad dinámicamente; y el *entorno* multi-agente, que provee los servicios para la interacción entre agentes.

El framework para sistemas multi-agente FraMaS ha sido implementado en el lenguaje OO Java. En Java la clase *Object* es la raíz de toda la jerarquía de clases. Toda clase tiene a *Object* como una superclase. De esta forma los métodos definidos en la clase *Object* son accesibles por todas las clases.

7.2.1. Agentes

La clase abstracta[1] *Agent* define la funcionalidad básica del agente, es decir el contexto del agente en si, como así también la funcionalidad para iniciar el agente por vez primera (`initialize`), comenzar el ciclo de ejecución de acciones (`start`), detener el agente (`stop`) y finalizar su ejecución (`finalize`). El método abstracto `eventFired` es responsable de las acciones que el agente realizará cuando percibe un evento.

Por último esta clase implementa el mecanismo de reflexión estructural provisto en Java para permitir el envío de mensajes dinámicos (`sendMessage`).



[1] Las definiciones de clases abstractas y métodos abstractos son indicados con letra *itálica*.

```
strUser! =
agentContext.getAgentIdentity.getUserName
```

El contexto del agente (*AgentContext*) provee una interfaz entre el agente y el entorno en el que se encuentra inserto. Esta clase abstracta hereda de la clase abstracta *EventProducer*. La interfaz establece un protocolo con el entorno que permite conocer en qué entorno se encuentra el agente, requerir servicios al entorno tales como movilidad, publicación del agente y obtención de los agentes publicados para –por ejemplo, localizar otros agentes en función de los servicios que proveen.

AgentContext

EventProducer

getAgentEnvironmentName

```
ret! : String
```

getAgentIdentity

```
ret! = AgentIdentity
```

requestToTransfer

```
strAgentHostName? : String
ret! : TransferResponse
```

publish

```
strIdentity? : String
```

unpublish

```
strIdentity? : String
```

getPublishedAgents

```
ret! : Enumeration
```

getPublishedAgentIdentity

```
strIdentity? : String
ret! : AgentIdentity
```

AgentIdentity identifica a cada agente en el sistema. Los agentes están en un entorno en cada instante de tiempo (`strAgentHostName`), tienen un usuario al que representan (`strUser`) y una identificación única (`id`). Esta clase implementa la interfaz Java *Serializable* que permite serializar los objetos de una clase.

AgentIdentity

```
Serializable      //Interfaz Java
(strAgentHostName, strUser, id)
```

```
strAgentHostName : String
strUser : String
id : SecureRandom
```

```
strAgentHostName = null
strUser = null
id = null
```

INIT

Δ(strAgentHostName, strUser, id)
strAgentHostName? : String
strUser? : String
---
strAgentHostName´ = strAgentHostName?
strUser´ = strUser?
id´ = SecureRandom.Create

getAgentHostName

strAgentHostName! : String
---
strAgentHostName! = strAgentHostName

getUserName

strUserName! : String
---
strUserName! = strUserName

La clase concreta *EventAgent* extiende de la clase Java *EventObject*, que representa un evento ocurrido en el sistema. La especialización de esta clase para los sistemas de agentes incorpora un argumento del tipo "Object" que es empleado en la notificación de los eventos.

EventAgent

EventObject
↾(argObject)
---
argObject : Object
---
argObject = null

INIT

Δ(argObject)
argObject? : Object
source? : Object
---
super(source?)
argObject´ = argObject?

getArgObject

argObject! : Object
---
argObject! = argObject

La clase abstracta *EventProducer* implementa la interfaz *EventProducerI* (explicada más adelante en este capítulo). De esta forma es posible que los agentes perciban los eventos que ocurren en su entorno y notifiquen de los eventos que producen.

EventProducer

EventProducerI // Interfaz
↾(listeners)
---
listeners : Vector
---
listeners = null

INIT

Δ(listeners)
---
listeners = Vector.Create

addListener

Δ(listeners)
agent? : Agent
---
listeners´.addElement

notifyAgentListener

evtAgent? : EventAgent

∀ i: 1..#listeners • listeners.elementAt(i).
eventFired(evtAgent)

removeAgentListener

Δ(listeners)
agent? : Agent
---
listeners´.removeElement

La clase concreta *AgentException* (que hereda de la clase Java *Exception*), y *AgentDefinitionException* y *AgentNotFoundException* (que heredan de *AgentException*) establecen errores que pueden ocurrir con los agentes

AgentException

Exeption
↾(agentIdentity)
---
agentIdentity : AgentIdentity
---
agentIdentity = null

INIT

Δ(agentIdentity)
agentIdentity? : AgentIdentity
---
agentIdentity´ = agentIdentity?

AgentDefinitionException

AgentExeption

AgentNotFoundException

AgentExeption

A continuación se especifican las clases que corresponden a las acciones básicas del agente y sus decoradores. Primero la clase abstracta *BasicAgentAction* que hereda de *Agent* y que representa la clase padre de la funcionalidad básica que tiene cada agente. Se define una colección con éstas acciones (`actions`).

BasicAgentActions

Agent
↾(actions)
---
actions : Vector
---
actions = null

INIT

Δ(actions)

actions = Vector.Create

add

Δ(actions)
actions? : Action

actions´.addElement

delete

Δ(actions)
action? : Action

actions.removeElement

Análogamente a ésta clase se encuentra la clase abstracta *AdvancedBehavior* que hereda de *Agent* y que representa la clase padre de los decoradores del agente. Tiene implementada la funcionalidad para configurar una estrategia de decisión y define –a través del método template `run`, el mecanismo empleado por los agentes para la selección de la próxima acción a ejecutar.

AdvancedBehavior

Agent
Runnable // Interfaz Java

agent : Agent
strategy : Strategy
actionThread : Thread

agent = null
strategy = null
actionThread = null

INIT

Δ(agent, actionThread)
agent? : Agent

agent' = agent?
actionThread' = Thread.Create

setStrategy

Δ(strategy)
strategy? : Strategy

strategy´ = strategy?

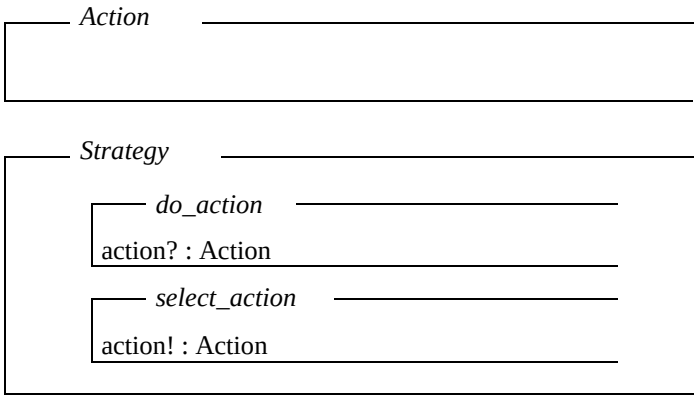
run

action? : Action

action = strategy.select_action
strategy.do_action

Execute ≙ this.isRunning ⨾
body.Execure
▯
¬this.isRunning

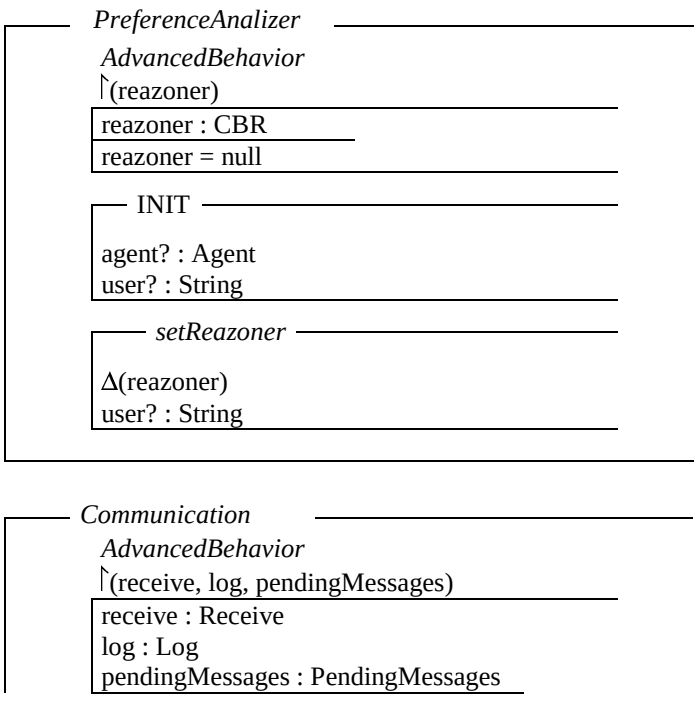
En la clase abstracta *Action* se implementa la funcionalidad de las acciones que los agentes particulares pueden realizar. Entre las que se encuentran las acciones básicas definidas en *BasicAgentActions* (colección `actions`). Mientras que la clase abstracta *Strategy* define la interfaz a emplear en la selección y ejecución de las acciones, a través de dos métodos abstractos: `select_action` y `do_action`.

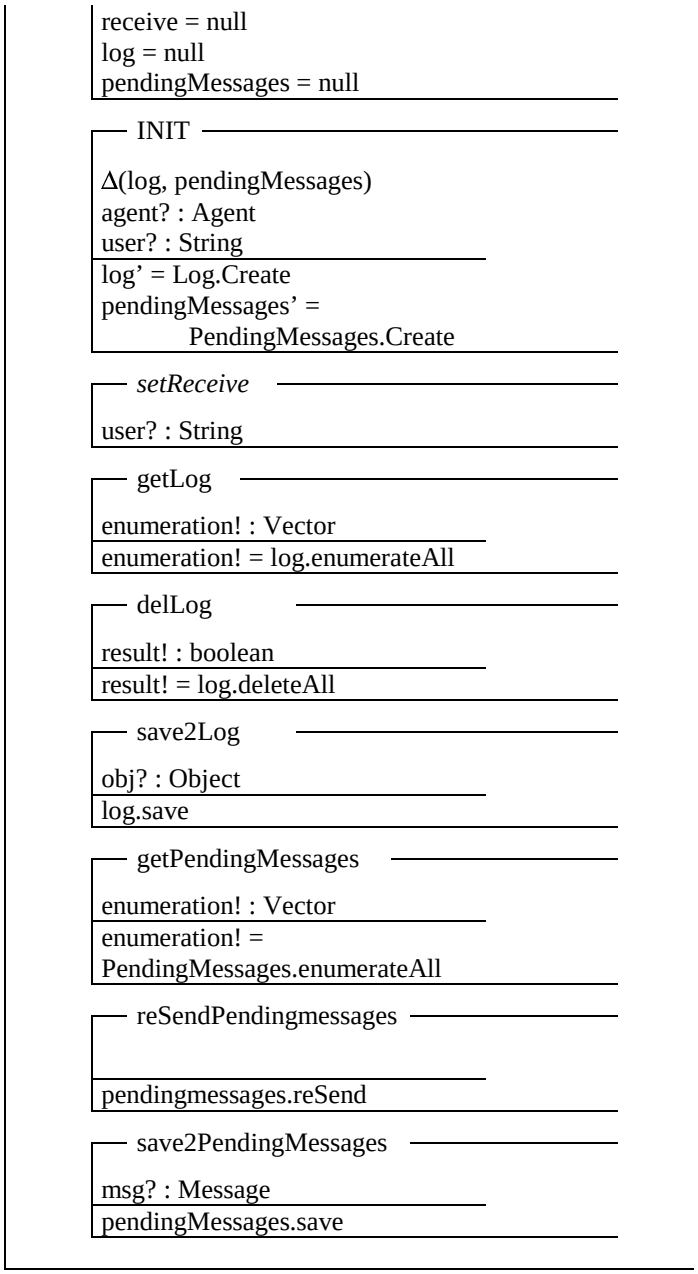


PreferenceAnalizer y *Communication* son dos clases abstractas que tienen la responsabilidad de los decoradores correspondientes al analizador de preferencias del usuario y a la comunicación entre agentes respectivamente. Ambas clases heredan de *AdvancedBehavior*.

La primera emplea la estructura de un razonador basado en casos para aprender las preferencias del usuario o de otros agentes. Se describe más adelante en éste capítulo la estructura correspondiente al razonador.

La segunda administra la comunicación entre agentes. Básicamente establece el mecanismo para enviar y recibir mensajes, como así también para mantener un registro de los mensajes enviados y recibidos (`log`) y de los mensajes que no pudieron ser enviados (`pendingMessages`)





Las seis definiciones siguientes (junto a la clase abstracta *PreferenceAnalizer*) corresponden a la estructura del decorador que analiza las preferencias. Como se mencionó emplea un razonador basado en casos, definido a través de las clases *CBR*, *Matching*, *Ranking*, *Feedback*, *CaseLibrary* y *CaseAgent*.

La clase abstracta CBR está asociada con una librería de casos donde almacena los casos de entrenamiento y con un algoritmo de *matching* entre casos. Así también tiene definido un método abstracto `adapt` para generar una solución a una situación dada desde un conjunto de casos extraídos de la librería (pertenecientes al *ranking* elaborado en función de la similitud entre los casos de la librería de casos y la situación). Además esta clase dispone de dos métodos implementados para el almacenamiento de un ejemplo de entrenamiento en la librería de casos (`trainingExample`) y para buscar una solución a una situación nueva (`searchSolution`).

CBR

(parent, caseLibrary, matching)

parent : PreferenceAnalizer
caseLibrary : CaseLibrary
matching : Matching

parent = null
caseLibrary = null
matching = null

INIT

Δ(parent)
parent? : PreferenceAnalizer
user? : String

parent' = parent?

adapt

caseAgent? : CaseAgent
ranking? : Ranking
result! : CaseAgent

searchSolution

situation? : CaseAgent
result! : CaseAgent

matching.start
result! = adapt

setCaseLibrary

user? : String

setMatching

user? : String

trainingExample

caseAgent? : CaseAgent

caseLibrary.save

El siguiente esquema de clase especifica la clase abstracta *Matching*, responsable de recorrer la librería de casos en busca de los casos más similares a la situación (método concreto `start`). El método abstracto `similar` establece el protocolo para implementar la similitud entre casos, requerida para conformar el *ranking*.

Matching

(caseLibrary, situation, ranking)

caselibrary : Caselibrary
ranking : Ranking
situation : caseAgent

caseLibrary = null
ranking = null
situation = null

INIT

Δ(caseLibrary)
caseLibrary? : CaseLibrary
caseLibrary' = caseLibrary?

setRanking

ranking? : Ranking

getRanking

result! : Ranking
result! = ranking

setCaseLibrary

user? : String

similar

caseAgent? : CaseAgent
result! : int

start

Δ(situation)
sit? : CaseAgent
auxCase : CaseAgent
auxInt : int
situation' = sit?
caseLibrary.getPointer < caseLibrary.length
⨟ [auxCase = caseLibrary.getCase
auxInt = similar
ranking.update]

El esquema *Ranking* especifica la clase concreta responsable de mantener una colección ordenada de los casos más similares. Esta clase ordena los `maxRankingSize` casos más similares de acuerdo al valor de similitud con la situación actual. Cada vez que el método `update` es invocado controla si el caso que se quiere ingresar en el *ranking* tiene un valor de similitud por encima del mínimo actual (`actualMinValue`)

Ranking

(ranking, caseAgent, rankingValues,
actualMinValue, maxRankingSize)

ranking : Vector
caseAgent : caseAgent
rankingValues : Vector
actualMinValue : int
maxRankingSize : int
ranking = null
caseAgent = null
rankingValues = null
actualMinValue = 0
maxRankingSize = 10

INIT

Δ(ranking, rankingValues)

ranking' = Vector.Create
rankingValues' = Vector.Create

getBest

result! : CaseAgent
result! = ranking.elementAt(0)

getRanking

result! : Vector
result! = ranking

update

Δ(ranking, rankingValues)
caseAgent? : CaseAgent
value? : int
i : int
aucValor : Integer

([[value? > actualMinValue] ^
[ranking.size ≤ maxRankingSize]]
ranking.insertElement •
rankingValues.inserElement)
▯
(¬[[value? > actualMinValue] ^
[ranking.size ≤ maxRankingSize]]
ranking.removeElement •
rankingValues.removeElement •
ranking.insertElement •
rankingValues.inserElement)

La clase *Feedback* es responsable de la información que brinda el usuario (o es adquirida automáticamente por al agente) sobre el resultado de la solución propuesta desde la librería de casos ante una situación. Esta depende del razonador en concreto desarrollado.

Feedback

↾(info)

info : Object
info = null

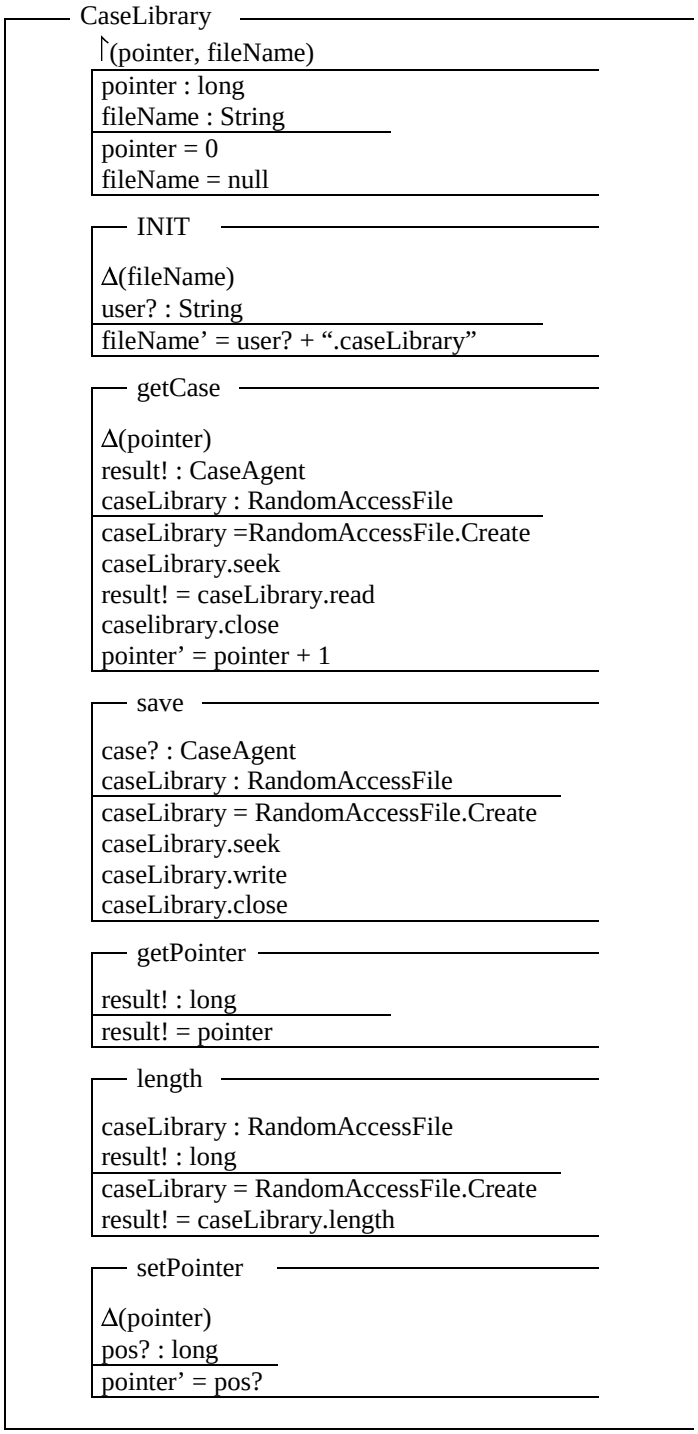
getInfo

result! : Object
result! = info

setInfo

Δ(info)
info? : Object
info' = info?

El siguiente esquema define la estructura de la librería de casos. La misma permite almacenar casos (`save`) y recuperarlos, en particular el método `getCase` recorre toda la librería devolviendo uno a uno los ejemplos de entrenamiento previamente guardados.



CaseAgent es el último esquema correspondiente al razonador basado en casos y especifica la estructura y el comportamiento general de cada caso en un sistema de agente. Éstos están compuestos por una identificación (`ID`), la situación que dio origen a la generación del caso (`situation`), la posible solución propuesta por el razonador (`solution`) y –eventualmente el `feedback` del caso.

CaseAgent

(ID, situation, solution, feedback)

ID : Object
situation : Object
solution : Object
feedback : Feedback

ID = null
situation = null
solution = null
feedback = null

getID

result! : Object

result! = ID

getSituation

result! : Object

result! = situation

getSolution

result! : Object

result! = solution

getFeedback

result! : Object

result! = feedback

setID

Δ(ID)
value? : Object

ID’ = value

setSituation

Δ(situation)
value? : Object

situation’ = value

setSolution

Δ(solution)
value? : Object

solution’ = value

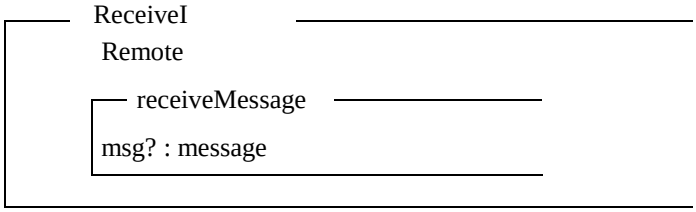
setFeedback

Δ(feedback)
value? : Object

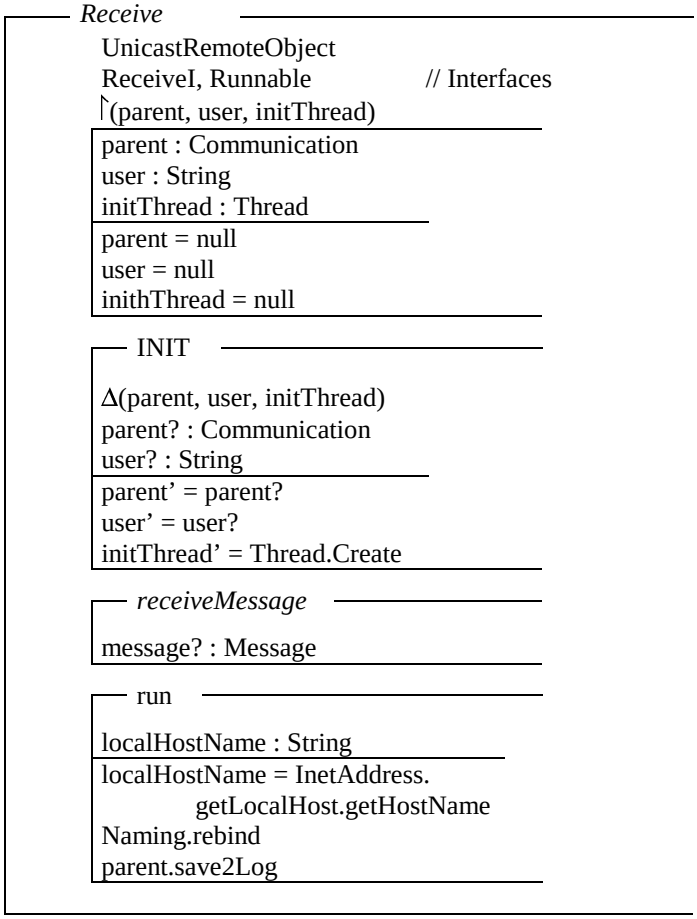
feedback’ = value

Los siguientes esquemas de clase: *Receive*, *Send*, *Message*, *Log* y *PendingMessages*; y de interfaz: *ReceiveI* especifican la estructura correspondiente al decorador de comunicación.

La interfaz *ReceiveI* extiende la interfaz Java *Remote* que debe ser implementada por todos los objetos que quieran disponer de métodos disponibles de ser accedidos en forma remota (`receiveMessage`).



La clase abstracta *Receive* hereda de *UnicastRemoteObject* (clase Java) e implementa la interfaz *ReceiveI* y la interfaz Java *Runnable* que define los *Threads*. Éste *Thread* (`initThread`) es el responsable de recibir los mensajes dirigidos al agente, su funcionalidad es especificada en `run`. Mientras que el método abstracto `receiveMessage` establece el protocolo que usarán los agentes para recibir mensajes y en este método se implementa la funcionalidad para dicha acción.



Análogamente, el esquema *Send* define el protocolo básico para enviar mensajes a otros agentes. Ésta clase hereda de *Thread*, con lo que cada mensaje enviado se ejecuta independientemente del resto del agente y de los otros mensajes enviados.

Send

Thread
↾(parent, message)

parent : Communication
message : Message

parent = null
message = null

INIT

Δ(parent, msg)
parent? : Communication
msg? : Message

parent' = parent?
msg' = msg?

getMessage

msg! : Message

msg! = message

run

address? : String
n : ReceiveI

n = Naming.lookup
n.receiveMessage
parent.save2Log

La clase *Message* implementa la interfaz Java *Serializable* y está compuesta por las identidades del agente originario y de agente destinatario del mensaje. Como así también por una descripción del mensaje y el cuerpo del mismo.

Message

Serializable // Interfaz
↾(origin, destination, description, body)

origin : AgentIdentity
destination : AgentIdentity
description : Object
body : Object

origin = null
destination = null
description = null
body = null

INIT

Δ(origin, destination,
description, body)
origin? : AgentIdentity
destination? : AgentIdentity
description? : Object
body? : Object

origin' = origin?
destination' = destination?
description' = destination?
body' = body?

getOrigin

result! : AgentIdentity

result! = origin

getDestination

result! : AgentIdentity

result! = destination

getDescription

result! : Object

result! = description

getBody

result! : Object

result! = body

Los dos esquemas siguientes (*Log* y *PendingMessages*) terminan con la especificación del decorador de comunicación. El primero brinda la funcionalidad básica para almacenar información sobre los mensajes enviados, enumerar los registros actuales y borrarlos. El segundo permite guardar los mensajes que no pudieron ser enviados satisfactoriamente, enumerar éstos mensajes y reenviarlos.

Log

(parent, user)

parent : Communication
user : String

parent = null
user = null

INIT

Δ(parent, user)
parent? : Communication
user? : String

parent' = parent?
user' = user?

deleteAll

logFile : File
result! : boolean

logFile =
File.Create(user+".log")
result! = logFile.delete

enumerateAll

vecLog! : Vector
logFile : RandomAccessFile

vecLog = Vector.Create
logFile = RandomAccessFile.Create
([logFile.getFilePointer < logFile.length] •
    ⨾ logFile.read ‖ vecLog.addElement)

save

info? : Object
logFile : RandomAccessFile
---
logFile = RandomAccessFile.Create
logFile.seek
logFile.write
logFile.close

PendingMessages

↾(parent, user)

parent : Communication
user : String
---
parent = null
user = null

INIT

Δ(parent, user)
parent? : Communication
user? : String
---
parent' = parent?
user' = user?

reSend

vecMsgP : Vector
fileMsgP : RandomAccessFile
msg : Message
---
vecMsgP = Vector.Create
fileMsgP = RandomAccessFile.Create
([fileMsgP.getFilePointer < fileMsgP.length]
• ⨾ fileMsgPFile.read ‖ vecMsg.addElement)
([vecMsgp.length > 0]
• ⨾ i : 1..#vecMsgP •
msg=vecMsgP.elementAt(i) ‖
Send.Create(msg, parent)

enumerateAll

vecMsgP! : Vector
msgPFile : RandomAccessFile
---
vecMsgP = Vector.Create
msgPFile = RandomAccessFile.Create
([msgPFile.getFilePointer < msgPFile.length]
• ⨾ msgPFile.read ‖ vecMsgP.addElement)

save

msg? : Object
msgPFile : RandomAccessFile
---
msgPFile = RandomAccessFile.Create
msgPFile.seek
msgPFile.write
msgPFile.close

7.2.2. Entorno Multi-agente

Los dos esquemas de clases siguientes (*AgentFacade*, *AgentContextImplementation*) son parte de la estructura del agente, en particular aquellas partes del agente que se relacionan con el entorno multi-agente.

La clase abstracta *AgentFacade* implementa la funcionalidad para que el entorno multi-agente pueda cambiar los estados del agente a través de una interfaz bien definida. Esta clase es extendida por los agentes concretos implementado los métodos abstractos `load` y `unload` para la carga y descarga del agente respectivamente.

AgentFacade

(agent, agentIdentity, masservices, nState)

INERT = -1
INITIALIZING = 1
INITIALIZED = 2
STARTING = 3
STARTED = 4
STOPPING = 5
STOPPED = 6
FINALIZING = 7
FINALIZED = 8

---

agent : Agent
agentIdentity : AgentIdentity
masservices : MASServices
nState : int

---

agent = null
agentIdentity = null
masservices = null
nState = INERT

agentFinalize

Δ(nState)

---

nState’ = AgentFacade.FINALIZING
agent.finalize
nState’ = AgentFacade.FINALIZED

agentInitialize

Δ(nState)

---

nState’ = AgentFacade.INITIALIZING
agent.initialize
nState’ = AgentFacade.INITIALIZED

agentStart

Δ(nState)

---

nState’ = AgentFacade.STARTING
agent.start
nState’ = AgentFacade.STARTED

agentStop

Δ(nState)

nState' = AgentFacade.STOPPING
agent.stop
nState' = AgentFacade.STOPPED

getAgent

result! : Agent
---
result! = agent

getAgentEnvironment

result! : MASServices
---
result! = masservices

getAgentIdentity

result! : AgentIdentity
---
result! = agentIdentity

getState

result! : int
---
result! = nState

initialize

Δ(masservices)
MAS? : MASServives
---
masservices' = MAS?

load

obj? : ObjectInputStream

unload

obj? : ObjectOutputStream

setup

Δ(agent, agentIdentity, nState)
agent? : Agent
agentID? : AgentIdentity
boolNew? : boolean
---
agenr' = agent?
agentIdentity' = agentID?
agent.setAgentContext
    (AgentContextImplementation.Create)
([boolNew] • nState' = INERT)
  □
(¬[boolNew] • nState' = STOPPED)

AgentContextImplementation extiende la clase *AgentContext* implementado los métodos abstractos allí definidos para la publicación de agentes en el entorno y para iniciar la transferencia del agente a otro entorno.

AgentContextImplementation

AgentContext
↾(agentFacade, strNicName)

agentFacade : AgentFacade
strNicName : String

agentFacade = null
strNicname = null

INIT

Δ(agentFacade)
agF? : AgentFacade

agentFacade' = agF?

getAgentEnvironmentName

result! : String

result! =- agentFacade.getAgentEnvironment.
getAgentEnvironmentName

getAgentIdentity

result! : AgentIdentity

result! = agentFacade.getAgentIdentity

getPublishedAgents

result! : Enumeration

([agentFacade.getState = agentFacade.STARTING]
∨
[agentFacade.getState = agentFacade.STARTED] •
result! = agentFacade.getAgentEnvironment.
htPublishedAgents.keys

getPublishedAgentIdentity

agNicID? : String
agentID! : AgentIdentity

([agentFacade.getState = agentFacade.STARTING]
∨
[agentFacade.getState = agentFacade.STARTED] •
agentID! = agentFacade.getAgentEnvironment.
htPublishedAgent.get

publish

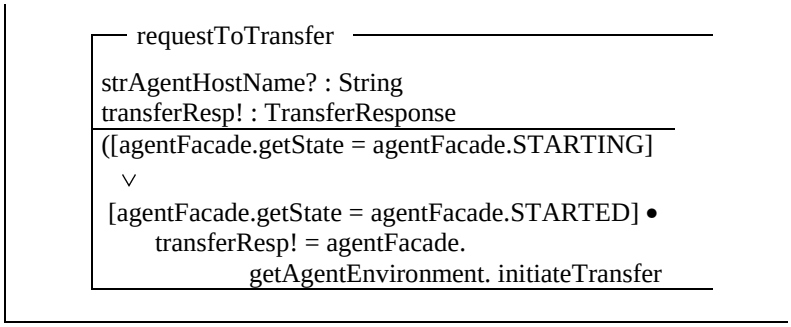
Δ(strNicName)
agNicID? : String

([agentFacade.getState = agentFacade.STARTING]
∨
[agentFacade.getState = agentFacade.STARTED] •
agentFacade.getAgentEnvironment.
htPublishedAgent.put •
strNicName' = agNicID?

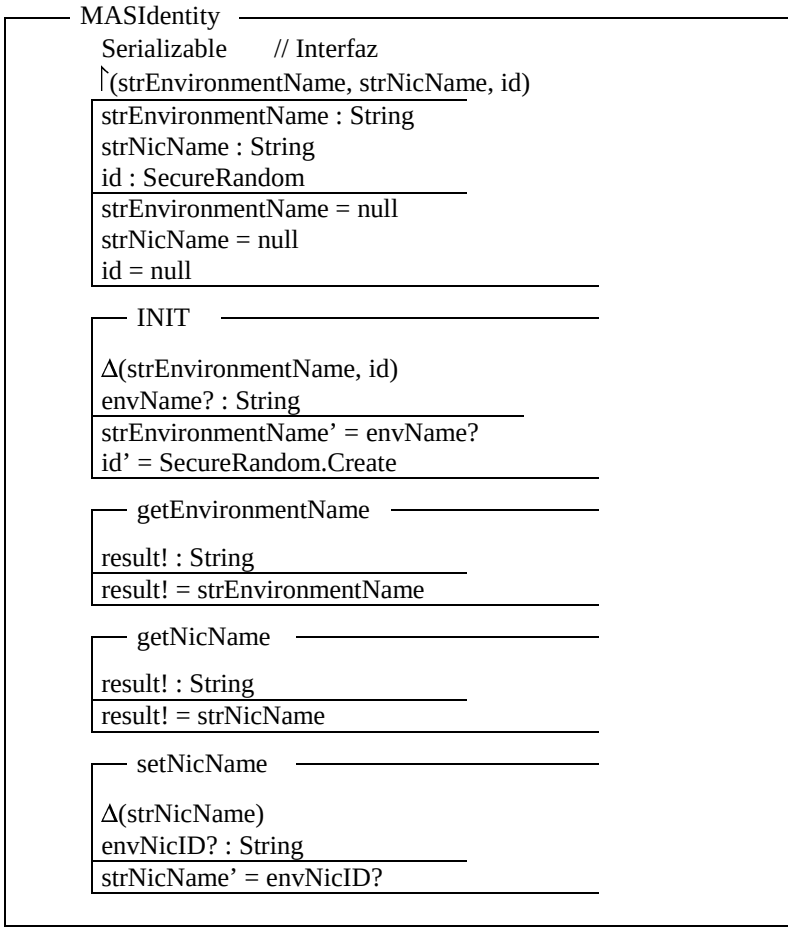
unpublish

Δ(strNicName)

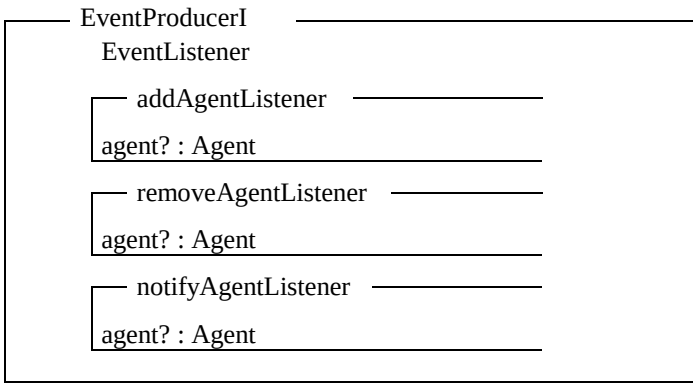
([agentFacade.getState = agentFacade.STARTING]
∨
[agentFacade.getState = agentFacade.STARTED] •
agentFacade.getAgentEnvironment.
htPublishedAgent.remove •
strNicName' = null



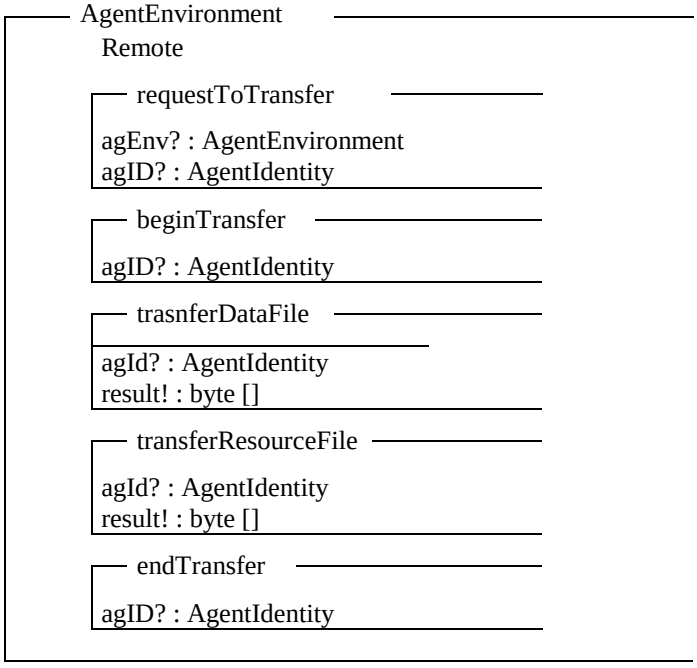
El sistema multi-agente está compuesto en FraMaS por *entornos.* En un mismo sitio pueden coexistir varios entornos, cada uno de los cuales provee a los agentes con determinados servicios. El esquema *MASIdentity* define la identidad de éstos entornos y esta compuesto por un nombre usualmente la dirección del *host* –por ejemplo: *rmi://darkstar.isistan.exa.unicen.edu.ar/biblioteca* (`strEnvironmentName`), un alias (`strNicName`) y una identificación única (`id`).



La siguiente especificación corresponde a una interfaz que debe ser implementada por los agentes y entornos para escuchar y transmitir eventos. La interfaz *EventProducerI* hereda de la interfaz Java *EventListener* y define el protocolo a emplear para mantener la lista de agentes registrados (es decir los que escucharan los eventos que se quieran notificar), como así también el mecanismo para notificar dichos eventos.



El siguiente esquema especifica otra interfaz FraMaS, *AgentEnvironment*, que hereda de *Remote* y define el protocolo que emplean los entornos multi-agente para la trasferencia de los agentes. La interfaz Java *Remote* debe ser implementada por todos los objetos que quieran disponer de métodos a ser accedidos en forma remota (`requestToTransfer`, `beginTransfer`, `transferDataFile`, `transferResourceFile`, `endTransfer`).



El esquema de clase *MASServices* especifica la clase principal de los entornos multi-agente, desde la cual se especializan los servicios particulares al SMA desarrollado. Esta clase hereda de la clase Java *UnicastRemoteObject*, necesaria para la comunicación remota usando RMI; e implementa las interfaces *AgentEnvironment* y *EventProducerI*, para la transferencia de agentes entre entornos y la notificación de los eventos ocurridos en el entorno respectivamente.

Además, define la estructura para que los agentes puedan ser insertados en el entorno (`createNewAgent`) y mantiene las colecciones de los agentes que se encuentran el entorno (`htFacade`) y de éstos cuáles están publicados (`htPublishedAgents`).

MASServices

UnicastRemoteObject
AgentEnvironment, EventProducerI // Interfaces
(hi, masIdentity, listeners, htfacade, htPublishedAgents)

hi : HostInterface
masIdentity : MASIdentity
listeners : Vector
htFacade : Hashtable
htPublishedAgents : Hashtable

---

hi = null
masIdentity = null
listeners = null
htFacade = null
htPublishedAgents = null

INIT

Δ(hi, masIdentity, listeners, htfacade,
htPublishedAgents)
strAgEnv? : String
hi? : HostInterface

---

hi' = hi?
masIdentity' = MASIdentity.Create
listeners' = Vector.Create
htFacade = Hashtable.Create
htPublishedAgents = Hashtable.Create
Naming.rebind(strAgEnv?, this)

createNewAgent

obj? : Object
agID! : AgentIdentity

---

agID! = resurrect
hi.update

resurrect

Δ(hfFacade)
agID! : AgentIdentity
agFacade : AgentFacade

---

agFacade = AgentFacade.Create
agFacade.initialize
agFacade.load
agID! = agentFacade.getIdentity
agID!.setAgentHostName(
masIdentity.getEnvironmentName)
htFacade'.put(agID!, agentFacade)
agFacade.agentInitialize
agFacade.agentStart

enumerateAllAgents

result! : Enumeration

---

result! = htFacade.keys

getAgentEnvironmentName

result! : String

---

result! = masIdentity.getEnvironmentName
getAgentEnvironmentNicName

result! : String
result! = masIdentity.getNicName

getAgentFacade

agID? : AgentIdentity
result! : AgentFacade
result! = htFacade.get(agID?)

removeAgentFacade

Δ(hfFacade)
agID? : AgentIdentity
result! : AgentFacade
result! = htFacade.remove(agID?)

initiateTransfer

strAgentHostName? : String
agentID? : AgentIdentity
result! : TransferResponse
agEnv : AgentEnvironment
result! = TransferResponse.Create
([agEnv = Naming.lookup(strAgentHostName?] •
  agEnv.requestToTransfer(this, agentID?) •
  result! = TransferResponse.SUCCEEDED)
  □
(¬[agEnv = Naming.lookup(strAgentHostName?] •
  result! = TransferResponse.FAILED)
hi.update

requestToTransfer

agEnv? : AgentEnvironment
agID? : AgentIdentity
agEnv?.beginTransfer(agID?)
agEnv?.transferResourceFile(agID?)
agEnv?.transferDataFile(agID?)
agEnv?.endTransfer(agID?)
agID = resurrect
hi.update

beginTransfer

Δ(htFacade)
agID? : AgentIdentity
agFacade : AgentFacade
afFacade = htFacade.remove(agID?)
agFacade.agentStop
agFacade.unload

transferResourceFile

result! = byte []
result! = getResourceAsBytes

transferDataFile

result! = byte []
result! = getDataAsBytes

endTransfer

Δ(listeners)
ag? : Agent

addAgentListener

Δ(listeners)
ag? : Agent
---
listeners' = addElement(ag?)

removeAgentListener

Δ(listeners)
ag? : Agent
---
listeners' = removeElement(ag?)

notifyAgentListener

evtAgent? : EventAgent
---
i : 1..#listeners • listeners.elementAt(i).
eventFired(evtAgent?)

La clase abstracta *HostInterface* hereda de la clase Java *Frame* y es responsable de la interfaz al usuario de los eventos ocurridos en el entorno multi-agente. Esta clase es especializada por los SMA construidos desde FraMaS.

HostInterface

Frame
↾(ah)
---
ah : MASServices
---
ah = null

update

obj? : Object

Los dos esquemas siguientes corresponden a errores relacionados con la falla de acceso a disco (`RepositoryException`) y con la falla en la trasferencia de agentes entre los entornos (`TransferFailedException`).

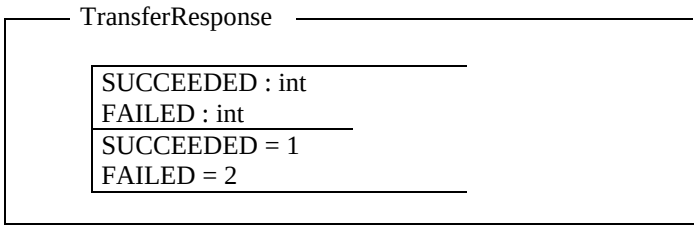
RepositoryException

IOException

TransferFailedException

Exception

Por último el esquema denominado *TransferResponse* corresponde a la clase que define las posibles respuestas en la transferencia de agentes entre dos entornos. El método `requestToTransfer` definido en la clase abstracta *AgentContext* e implementado en *AgentContextImplementation* retorna un objeto de este tipo en respuesta a la transferencia de un agente.



7.3. Resumen

Se describió en este capítulo la estructura de clases correspondiente al diseño del framework para sistemas multi-agente FraMaS usando el lenguaje de especificación formal *Object-Z*. Primero se especificaron los aspectos más importantes de las clases correspondientes a los agentes y luego los de las clases del entorno multi-agente.

La especificación de los agentes está compuesta por la especificación del agente en si (*Agent*, *AgentContext*, *EventAgent*, etc.); las clases y métodos para la funcionalidad básica, la incorporación de decoradores y las estrategias de decisión (*BasicAgentActions*, *AdvancedBehavior*, *Strategy*, etc.); la estructura correspondiente al Analizador de Preferencias y a la Comunicación .

A continuación se presenta la organización del diseño de las clases de FraMaS, implementado con clases e interfaces Java agrupadas en paquetes[1] y la lista de los métodos *template*:

Paquete	**Nombre de Clase o Nombre de Interfaz**	**Métodos**				
		base	**hook**	**template**	**abstracto**	**Total**
framas.agent	*Agent* [A]	5	2		5	**12**
	AgentContext [A]				7	**7**
	AgentIdentity	6				**6**
	EventAgent		1			**1**
	EventProducer [A]		2	2		**4**
	AgentException [E]					
	AgentDefinitionException [E]					
	AgentNotFoundException [E]					
framas.agentWrappers	*AdvancedBehavior* [A]		1	1		**2**
	BasicAgentActions [A]		2			**2**
	PreferenceAnalizer [A]				1	**1**
	Communication [A]		6		1	**7**
	Strategy [A]				2	**2**
	Action [A]					
framas.agentCBR	*CBR* [A]			2	3	**5**
	Matching [A]		1	1	2	**4**
	CaseLibrary		1	3		**4**
	Ranking		2	1		**3**
	CaseAgent [A]		6		2	**8**
	Feedback		2			**2**

[1] *Se especifica en **negrita** los nombres de los **paquetes**.* [A] *: clase abstracta,* [I] *: interfaz,* [E] *: clase de manejo de errores. Las clases sin superíndices corresponden a clases concretas.*

framas. agentCommuni cation	*ReceiveI* [I]				1	**1**
	Receive [A]			1	1	**2**
	Send		1	1		**2**
	Message		4			**4**
	PendingMessages		3			**3**
	Log		3			**3**
framas. environment	*AgentEnvironment* [I]				5	**5**
	MASServices	4	18	3		**25**
	MASIdentity	6				**6**
	HostInterface [A]		1		2	**3**
	AgentFacade [A]	4	6		2	**12**
	AgentContextImplementation	8			1	**9**
	AgentClassLoader	7	2			**9**
	EventProducerI [I]				4	**4**
	Repository	6				**6**
	RepositoryEntry	16				**16**
	TransferFailedException [E]					
	TransferResponse [E]					
	RepositoryException [E]					
Total	**39**	**62**	**64**	**15**	**39**	**180**

Resumen General		**Resumen de clases**		**Resumen de métodos**	
Packages	5	Abstract classes	15	Base methods	62
Classes	36	Concrete classes	15	Hook methods	64
Interfaces	3	Error handling classes	6	Template methods	15
Methods	180			Abstract methods	39

Paquete	**Clase / Interfaz**	**Nombre del método template**
framas.agent	EventProducer[A]	public void notifyAgentListener()
		public void notifyAgentListener(EventAgent evtAgent)
framas.agentWrappers	AdvancedBehavior[A]	public void run()
framas.agentCBR	CBR[A]	public CaseAgent searchSolution(CaseAgent situation)
		public void trainingExample(CaseAgent caseAgent)
	Matching[A]	public void start(CaseAgent sit)
	CaseLibrary[A]	public CaseAgent getCase(Object caseObj)
		public long length()
		public void save(CaseAgent caseAgent)
	Ranking	public void update(CaseAgent caseAgent, int value)
framas.agentCommunication	Receive[A]	public void run()
	Send	public void run()
framas.environment	MASServices	public void requestToTransfer (AgentEnvironment agentenvironment, AgentIdentity agentidentity)
		public void notifyAgentListener()
		public void notifyAgentListener (Event Agent evtAgent)

8. Conclusión y Trabajo futuro

Las características más importantes del framework, nombrando las aplicaciones construidas desde éste, son descriptas en este capítulo. Luego, se presentan las contribuciones y limitaciones del presente trabajo. Finalmente, se describen posibles líneas de trabajo futuro.

8.1. Características de FraMaS

Los sistemas multi-agente tienen particularidades que los hacen apropiados para los sistemas multi-plataforma, interconectados, etc. Estos sistemas tienen un gran campo de aplicación en diversas áreas de la industria y del comercio en general.

Básicamente se habla más de una Ingeniería del Software basada en Agentes, que establece una forma de analizar, diseñar e implementar este tipo de sistemas. La idea es ver a los sistemas de computadoras (hardware o software) como un conjunto de entidades (agentes) que interactúan, enviándose mensajes y coordinando tareas, con el objetivo de satisfacer sus objetivos y los del sistema.

El framework presentado es viable (en función de las pruebas empíricas) para la construcción de Sistemas Multi-agente. Las instanciaciones del framework consisten en:

- Asistentes personales, en particular el sistema de los Agentes Agenda que asisten al usuario en la administración de sus compromisos.
- Sistema de Agentes Forklift, compuesto por un conjunto de robots que interactúan en un entorno común a fin de satisfacer el objetivo del sistema, necesitando resolver conflictos y –eventualmente negociar.
- La estructura de agentes para comercio electrónico, que realizan un recorrido buscando satisfacer una lista de requerimientos, contactando en cada sitio los agentes según los servicios y/o productos ofrecidos.

FraMaS fue construido siguiendo un proceso iterativo desde las aplicaciones al diseño e implementado en Java. En cada paso de iteración se identifican los componentes comunes al dominio del problema, como así también sus relaciones, adicionando éstos al diseño del framework. Luego, se reimplementan las aplicaciones para reflejar el cambio.

El objetivo de este proceso es capturar en el framework las características del dominio, las cuales son desconocidas a priori. Las aplicaciones que pertenecen a un dominio evolucionan, por lo tanto también deben evolucionar los frameworks construidos para ese dominio. Es prácticamente imposible concebir un framework que anticipe todas las futuras evoluciones en el dominio de aplicación.

Aun así es ventajoso construir este tipo de modelo porque permite el reuso de diseño y código. Otra ventaja se pone de manifiesto si se considera la dificultad inherente a los SMA en si, tales como la descripción, descomposición y distribución de las entidades, como así también el problema de coordinar y sintetizar los resultados de un grupo de agentes sin un control central.

La estructura de cada agente fue diseñada para soportar la adición de nuevas responsabilidades dinámicamente (como en el caso del Agente Agenda). Esto se logró empleando el protocolo de decoradores definido en el patrón de diseño Decorator [Gamma 95] y reflexión estructural. A través de este protocolo simple se permite la composición del agente a partir de un conjunto de acciones básicas que son cubiertas por decoradores, los mismos incorporan el comportamiento avanzado del agente.

Usar este sistema permite una mayor flexibilidad que la incorporación de funcionalidad por medio de herencia. La subclasificación de clases es inflexible. La elección de las responsabilidades soportadas es realizada estáticamente y los clientes de una clase heredada no pueden controlar cómo y cuándo decorar el componente.

8.2. Contribuciones

El framework descripto permite la construcción de sistemas multi-agente, siendo una metodología general para el desarrollo de este tipo de sistemas. Los agentes construidos usando el framework tienen la capacidad de:

- *Percepción*, se define un protocolo que permite que todos los objetos puedan ser observados, dejando la responsabilidad de la notificación a cada agente.
- *Comunicación*, diseñada a través de un "decorador" de comunicación que provee las primitivas para enviar y recibir mensajes. Este decorador debe ser especializado a través de los métodos abstractos definidos en él.
- *Analizar las preferencias*, este "decorador" está asociado a un componente genérico para razonamiento basado en casos, que permite aprender las preferencias del usuario o de otros agentes.
- *Incorporar decoradores* a través de un protocolo simple. Logrando así adicionar componentes al framework que pueden ser reutilizados por los agentes construidos, componiendo los mismos para lograr la funcionalidad deseada.
- *Publicar agentes en el entorno*, a través de una interfaz implementada. Con este mecanismo es posible localizar agentes en función de los servicios ofrecidos.
- *Agregar otros servicios* al sistema multi-agente.

Los frameworks para SMA descriptos en el capítulo 3 proveen la estructura para el desarrollo de agentes con características particulares (deliberativos, reactivos, etc.), sin embargo ninguno de ellos (incluyendo a FraMaS) captura por completo el conocimiento del dominio de los sistemas multi-agente. Un aporte de FraMaS en este sentido es la simplicidad para adicionar nuevos "decoradores" incorporando así nueva funcionalidad a los agentes ya construidos.

Esta característica es importante porque al implementar un mecanismo nuevo de deliberación, por ejemplo, un algoritmo de planning, puede ser incorporado al framework como una estrategia alternativa de decisión simplemente respetando los protocolos definidos. Además, estas abstracciones pueden ser incorporadas desde los ejemplos en un proceso similar al empleado en la construcción del framework.

8.3. Limitaciones

Los servicios de cada entorno FraMaS permiten la "movilidad débil" de sus agentes, es decir la misma no es transparente. Esto se debe a que el framework está implementado en Java, y la máquina virtual de dicho lenguaje (JVM) no permite el acceso directo desde un programa a la memoria para manipular su estado interno (pila de ejecución, registros, etc.). Por esto, el

programador es responsable de resguardar el estado de ejecución del agente cuando va a ser transferido, como así también restaurar el estado cuando se reinicia.

FraMaS no incorpora ningún mecanismo de seguridad extra, a los provistos por la máquina virtual Java, esto podría ocasionar inconvenientes cuando un agente no confiable entra a un sitio, o cuando un agente es transferido de un sitio a otro por una red.

Si bien el framework provee una interfaz para la incorporación de estrategias de decisión, ningún componente concreto ha sido incorporado aún.

8.4. Extensiones

Componentes concretos del framework que pueden formar parte de líneas de trabajo futuro:

- *Lenguaje de comunicación de agentes*

La interacción inteligente implica compartir conocimiento. Esto incluye el entendimiento mutuo del conocimiento y el mecanismo de comunicación de dicho conocimiento. KQML (Knowledge Query Manipulation Language) es un lenguaje que facilita la cooperación e interacción inteligente entre agentes [Finin 97]. La incorporación de este tipo de lenguaje de comunicación de agentes puede ser realizada extendiendo el decorador de comunicación actual, o simplemente incorporando a FraMaS algún framework de KQML existente.

Continuando con el componente de comunicación de los agentes, la incorporación del envío de mensajes por Broadcast (es decir a muchos usuarios simultáneamente) permitiría, por ejemplo, descomponer el sistema multi-agente en “comunidades” o “grupos”. El lenguaje Java provee (clase *MulticastSocket*) las primitivas para implementar un socket UDP para enviar y recibir paquetes multicast IP que podrían ser empleadas para este tipo de organización.

- *Mecanismos de deliberación*

Subclasificar desde la clase definida para la selección de la próxima acción (*Strategy*), para incorporar componentes concretos de deliberación, tales como algoritmos de planning y scheduling. Esto permite desarrollar agentes con estas capacidades a través de composición desde el framework, realizando la implementación de los mismos como en el caso del razonador basado en casos descripto.

- *Mecanismos de coordinación*

En un sistema multi-agente el comportamiento social de cada agente está determinado por la forma en que se comunica, interactúa y coordina con otros agentes. La colaboración es una necesidad en los agentes porque ninguno está aislado ni puede realizar tareas complejas solo.

Para lograr tal comportamiento existen diversos formalismos, por ejemplo COOL [Barbuceanu 93], un lenguaje para representar explícitamente y aplicar el conocimiento de cooperación en un sistema multi-agente.

- *Analizador de preferencias*

La extensión consiste en emplear la librería de casos para extraer de ella un conjunto de reglas (del tipo precondición – acción) que representen las preferencias del usuario y sean comprensibles a éste. El objetivo es que el usuario tenga disponible de forma inteligible las preferencias que el agente mantiene, con la posibilidad de modificarlas directamente.

Como así también emplear los mensajes de coordinación enviados entre los agentes para aprender las preferencias del otro generando una librería de casos con estas interacciones. Para esto, se puede emplear un modelo de clasificación construido inductivamente desde la base de datos, generalizando desde ejemplos específicos (interacciones). Este modelo genera un *clasificador* llamado árbol de decisión [Quinlan 93]. Desde el árbol de decisión se pueden generar "reglas" que tienen por objetivo hacer que el modelo sea comprensible para las personas que lo utilizan.

Bibliografía

[Agree 87] AGREE, P.; CHAPMAN, D. PENGI: An Implementation of a Theory of Activity. Proceeding of the 6th National Conference on Artificial Intelligence. Morgan Kaufmann. 1987.

[Aiken 94] AITKEN, J. et al. A Knowledge Level Characterization of Multi-Agent Systems. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents. Berlin: Springer-Verlag, 1994.

[Allen 94] ALLEN, J. et al. The TRAINS Project: A case study in building a conversational planning agent. Technical Note 94-3. Rochester: Rochester University, Sept. 1994.

[AlphaWorks] http://www.alphaworks.ibm.com/

[Amandi 97a] AMANDI, A. Programação de Agentes Orientada a Objetos. Tese de Doutorado. Porto Alegre: CPGCC da UFRGS, 1997.

[Amandi 97b] AMANDI, A.; PRICE, A. Towards Object-Oriented Agent Programming: The Brainstorm Meta-Level Architecture. Proceedings of the Autonomous Agents Conference, 1., 1997, Los Angeles. [S.e.]: ACM Press, 1997.

[Amandi 97c] 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.; ZUNINO, A.; ITURREGUI, R.. Multi-paradigm languages supporting multi-agent development. Francisco J. Garijo and Magnus Bornan, editors, Multi-Agent System Engineering, 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World MAAMAW'99, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1999.

[Arcand 95] ARCAND, J.; Pelletier, S. Cognition Based Multi-Agent Architecture. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents II, p.267-282. Berlin: Springer-Verlag, 1995.

[Aridor 98] ARIDOR, Y.; LANGE, D. Agent Design Patterns: Elements of Agent Application Design. Proceedings of the Second International Conference on Autonomous Agents. 1998.

[Atherton 98] ATHERTON, R. Moving Java to the Factory. IEEE Spectrum. December 1998.

[Avancini 99a] AVANCINI, Henri. Experiencias en Reuso de Sistemas Multi-agente. Proceeding del Workshop de Investigadores en Ciencias de la Computación. San Juan, Argentina, 27-28 de Mayo de 1999.

[Avancini 99b] AVANCINI, Henri; AMANDI, Analía. Towards a Framework for Multi-agent Systems. Proceeding of the Third Argentine Symposium on Object Orientation. September 1999.

[Avancini 99c] AVANCINI, Henri; AMANDI, Analía. A Java Framework for Multi-agent Systems. Electronic Journal of SADIO. 1999.

[Balsubramanian 95] BALSUBRAMANIAN, S.; NORRIE, D. A Multi-Agent Intelligent Design System Integrating Manufacturing and Shop-Floor Control. Proceeding of the International Conference on Multi-Agent Systems, 1., 1995, San Francisco. Menlo Park: AAAI Press/ MIT Press, 1995.

[Barbuceanu 93] BARBUCEANU, M.; FOX, M. The Design of COOL: A Language for Representing Cooperation-Knowledge in Multi-agent Systems. http://www.ie.utoronto.ca/EIL/ABS-pages/ABS-intro.html. 1993.

[Barbuceanu 95] BARBUCEANU, M. ; FOX, M. The Architecture of an Agent Building Shell. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents II. Berlin: Springer-Verlag, p.235-250. 1995.

[Bennet 94] BENNET, F. et al. Teleporting – Making Applications Mobile. Proceedings of the Workshop on Mobile Computing Systems and Applications. 1994.

[Blum 97] BLUM, A.; FURST, M. Fast Planning Through Planning Graph Analysis. Artificial Intelligence, 90:281-300. 1997.

[Bonasso 95] BONASSO, R. et. al. Experiences with an Architecture for Intelligent, Reactive Agents. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents II. Berlin: Springer-Verlag, p.187-202. 1995.

[Bradshaw 97] BRADSHAW, J. An Introduction to Software Agents. Software Agents. J.M. Bradshaw (Ed.). AAAI Press / The MIT Press. 1997.

[Bratman 88] BRATMAN, M.E.; ISRAEL, D.J.; POLLACK, M.E. Plans and resource-bounded practical reasoning. Computational Intelligence, 4:349-355. 1988.

[Broce 97a] BROCE, Gerald; LÖHR, Klaus-Peter; SPIEGEL, André. Java Resists Transparent Distribution. Object Magazine, Dec. 1997.

[Broce 97b] BROCE, Gerald; LÖHR, Klaus-Peter; SPIEGEL, André. Java Does not Distribute. Proceedings of TOOLS Pacific´97. November 1997.

[Brooks 86] BROOKS, R. A Robust Layered Control Systems for a Mobile Robot. IEEE Journal of Robotics and Automation 2 (1). 1986.

[Brugali 97] BRUGALI, D. et al. The Framework Life Span. Communications of the ACM, 40 (10). October, 1997.

[Burmeister 92] BURMEISTER, B.; SUNDERMEYER, K. Cooperative problem solving guided by intentions and perception. E.Werner and Y.Demazeau, editors, Decentralized AI 3 – Proceedings of the Third European Workshop on Modeling Autonomous Agents and Multi-agent Worlds (MAAMAW-91). Elsevier Science Publishers B.V. 1992.

[Bussmann 92] BUSSMANN, S.; MÚLLER, H. A negotiation framework for cooperating agents. Proceedings of the CKBS-SIG Conference, p. 1-17. 1992.

[Canfield 97] CANFIELD SMITH, D. et al. KidSim: Programming Agents without a Programming Language. Software Agents. J.M. Bradshaw (Ed.). AAAI Press / The MIT Press. 1997.

[Cardelli 85] CARDELLI, Luca; WEGNER, Peter. On Understanding Types, Data Abstraction, and Polymorphism. Computing Surveys, Vol. 17 n.4, pp.471-522, December 1985.

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

[Chavez 96] CHAVEZ, A.; MAES, P. Kasbah: An agent marketplace for buying and selling goods. Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM-96), London, UK. 1996.

[Codenie 97] CODENIE, W. et al. From Custom Applications to Domain-Specific Frameworks. Communication of the ACM 40 (10). October, 1997.

[Cohen 90] COHEN, P.R.; LEVESQUE, H.J. Intention is choice with commitment. Artificial Intelligence, 42:213-261. 1990.

[Cointe 97] COINTE, P. Metaclasses are First Class: the ObjVlisp Model. N.K. Meyrowitz, pp. 44-46. 1997.

[Diaz Pace 99] DIAZ PACE, A.; TRILNIK, F.; CLAUSSE, A.; CAMPO, M. Un framework para simulación de procesos colectivos utilizando agentes reactivos. Proceedings of the SyM'99 (Simulación y Modelística) Pp 50-65, 28 JAIIO (Jornadas Argentinas de Informática e Investigación Operativa), Buenos Aires, Argentina. Septiembre de 1999.

[Diller 94] DILLER, Antoni. Z, An Introduction to Formal Methods. John Willey & Sons. 2nd Edition. 1994.

[Dong 96] DONG, J.S.; DUKE, R. Using Object-Z to Specify Object-Oriented Programming Languages. Technical Report No. 96-02. Department of Computer Science. University of Queensland. 1996.

[Duke 91] DUKE, R. et al. The Object-Z Specification Language. Version 1. Technical Report, No. 91-1. Department of Computer Science. University of Queensland. 1991.

[Duke 94] DUKE, R. et al. Object-Z: a Specification Language Advocated for the Description of Standards. Technical Report, No. 94-45. Department of Computer Science. University of Queensland. 1994.

[Eckel 98] ECKEL, B. Thinking in Java. Prentice Hall, 1998.

[Ekdahl 94] EKDAHL, B., et al. Towards Anticipatory Agents. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents. Berlin: Springer-Verlag, p.191-202. 1994.

[Eriksson 98] ERIKSSON, Hans-Erik; PENKER, Magnus. UML TOOLKIT. Wiley Computer Publishing, 1998.

[Etzioni 94] ETZIONI, O.; WELD, D. A Softbot-Based Interface to the Internet. Communications of the ACM 37 (7). 1994.

[Fayad 99a] FAYAD, M.; SCHMIDT, D.; JOHNSON, R. Building Application Frameworks. Willey Computer Publishing. 1999.

[Fayad 99b] FAYAD, M.; SCHMIDT, D.; JOHNSON, R. Implementing Application Frameworks. Willey Computer Publishing. 1999.

[Ferber 89] FERBER, Jacques. Computational Reflection in Class based Object Oriented Languages. SIGPLAN Notices, New York, v.24, n.10, p.317-326, Oct. 1989.

[Ferguson 92a] FERGUSON, I. Touring Machines; An Architecture for Dynamic, Rationale, Mobile, Agents. Ph.D. Thesis, Computer Laboratory, University of Cambridge, UK. 1992.

[Ferguson 92b] FERGUSON, I. Integrated Control and Coordinated Behavior: a Case for Agent Models. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents. Berlin: Springer-Verlag, p.203-218. 1994.

[Ferguson 92c] FERGUSON, I.; GEORGEFF, M.; RAO, A. An architecture for real-time reasoning and system control. IEEE Expert, Los Alamitos, v. 7, n.6, p.34-44, Dec. 1992.

[Fikes 71] FIKES, R.E. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, 5(2): 189-208, 1971.

[Finin 97] FININ, T. et al. KQML as an Agent Communication Language. Software Agents. J.M. Bradshaw (Ed.). AAAI Press / The MIT Press. 1997.

[FIPA 97] FIPA. http://drogo.cselt.it/fipa/

[Fisher 94a] FISHER, K.; MÜLLER, J.; PISCHEL, M. Unifying Control in a Layered Agent Architecture. Technical Report TM-94-05 from DFKI GmbH. German Research Center for Artificial Intelligence. 1994.

[Fisher 94b] FISHER, K. A survey of Concurrent METATEM – the language and its applications. D.M. Gabbay and H.J.Ohlbach, editors, Temporal Logic – Proceedings of the First International Conference (LNAI Volume 827). Springer-Verlag. 1994.

[Fisher 95] FISHER, K.; MÜLLER, J.; PISCHEL, M. A pragmatic BDI Architecture. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents II. Berlin: Springer-Verlag, p.203-218.(LNAI, v.1037). 1995.

[Foner 93] FONER L. What's an agent, anyway? A Sociological Case Study. Agents Memo 93-01. Massachusetts: MIT Media Laboratory, May 1993.

[Fowler 97] FOWLER, M. with SCOTT, K. UML Distilled. Addison-Wesley. 1997.

[Franklin 96] FRANKLIN, STAN; GRAESSER, ART. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. Proceedings of the Third international Workshop on Agent Theories, Architectures, and Languages. Springer-Verlag, 1996.

[Frozza 98] FROZZA, R. SIMULA – Ambiente de Simulacão em Sistemas Multiagentes Reactivos. Proceeding of the V Workshop sobre Aspectos Teóricos de la Inteligencia Artificial. Neuquén. 1998.

[Gabriel 91] GABRIEL, P.; WHITE, J.; BOBROW, P. CLOS: Integrating Object-Oriented and Functional Programming. Communications of the ACM, v.34, n.9, Set. 1991.

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

[Garlan 93] GARLAN, D.; SHAW, M. An Introduction to Software Architecture. AMBRIOLA, V.; TORTORA, G. (Eds.). Advances in Software Engineering and Knowledge Engineering, v.2. Singapore: World Scientific, 1993.

[Genesereth 94] GENESERETH, M.; KETCHPEL, S. Software Agents. Communications of the ACM, 37 (7). July 1994.

[Georgeff 87] GEORGEFF, M.P.; LANSKY, A.L. Reactive reasoning and planning. Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87). 1987.

[Goldberg 95] GOLDBERG, A.; RUBIN, K. Succeeding with Objects: Decision Frameworks for Project Management. 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.

[Haugeneder 94] HAUGENEDER, H. IMAGINE final project report. 1994.

[Hayes-Roth 91] HAYES-ROTH, B. An Integrated Architecture for Intelligent Agents. SIGART Bulletin 2, (4). 1991.

[Herbert 97] HERBERT, J.; PRICE, A. Una Proposta de Framework Orientado para a Avaliação de Software. Argentine Symposium on Object Orientation, Proceedings...Buenos Aires, August 11-12, 1997.

[Hoque 98] HOQUE, R. CORBA 3 Developer's Guide. IDG Books Worldwide, Inc. 1998.

[Horstmann 97] HORSTMANN, C. Practical Object-Oriented Development in C++ and Java. Willey Computer Publishing. 1997.

[Hoyle 97] HOYLE, M.; LUEG, C. Open Sesame!: A Look at Personal Assistants. Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, 1997.

[Huang 94] HUANG, J.; JENNINGS, N.; FOX, J. An Agent Architecture for Distributed Medical Care. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents. Berlin: Springer-Verlag, p.219-232. 1994.

[Huhns 97] HUHNS, M.; SINGH, M. Agents and Multiagent Systems: Themes, Approaches, and Challenges. Reading in Agents. Huhns & Singh Eds. Morgan Kaufmann Publishers. 1997.

[IBM] http://activist.gpl.ibm.com/WhitePaper/ptc2.htm

[Iglesias 95] IGLESIAS, C. et. al. MIX: A General Purpose Multiagent Architecture. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents II. Berlin: Springer-Verlag, p.251-266. 1995.

[JATLite 97] JATLite, JATLite overview. http://java.stanford.edu/java_agent/html/JATLiteOverview.htm. 1997.

[Jean-Piaget 92] JEAN-PIAGET. Psicología de la Inteligencia. Siglo Veinte. 1992.

[Jennings 93] JENNINGS, N. Specification and implementation of a belief desire join-intention architecture for collaborative problem solving. Journal of Intelligent and Cooperative Information Systems, 2(3): 298-318. 1993.

[Jennings 94] JENNINGS, N. Cooperation in Industrial Multiagent Systems. World Scientific Series in Computer Science, v. 43. Singapore: World Scientific, 1994.

[Jennings 98] JENNINGS, N. et. al. A Roadmap of Agent Research and Development. Autonomous Agents and Multi-Agent Systems. Volume 1, No. 1, 1998.

[Jennings 00] JENNINGS, N.; WOOLDRIDGE, M. Agent-Oriented Software Engineering. Handbook of Agent Technology (ed. J. Bradshaw) AAAI/MIT Press. (to appear). 2000

[Johnson 92] JOHNSON, Paul; REES, Ceri. Reusability Through Fine-grain Inheritance. Software-Practice and Experience, Vol. 22(12), 1049-1068. December, 1992.

[Johnson 88] JOHNSON, R.; FOOTE, B. Designing reusable classes. Journal of Object-Oriented Programming, New York, v.1, n.12, June/July 1988.

[Johnson 91] JOHNSON, R.; RUSSO, V. Reusing Object-Oriented Design. Technical Report UIUCDCS 91-1696. Illinois: Illinois University, 1991.

[Johnson 92] JOHNSON, R. Documenting Frameworks using Patterns. Proceeding of the O.O.P.S.L.A., 1992.

[Johnson 97a] JOHNSON, R. Components, Frameworks, Patterns. Proceedings of the Symposium on Software Reusability, pp.10-17, 1997.

[Johnson 97b] JOHNSON, R. Frameworks = Components + Patterns. Communications of the ACM, 40 (10). October, 1997.

[Kafura 98] KAFURA, D.; BRIOT, J. Actors & Agents. IEEE Concurrency. April-June, 1998.

[Kantorowitz] KANTOROWITZ, Eliezer. Algorithm Simplification through Object Orientation. Software-Practice and Experience, Vol. 27(2), 173-183. February, 1997.

[Karnik 98] KARNIK, N.; TRIPATHI, A. Design Issues in Mobile-Agent Programming Systems. IEEE Concurrency, July/September 1998.

[Kendal 98] KENDAL, E. et al. Patterns of Intelligent and Mobile Agents. Proceeding of the Second International Conference on Autonomous Agents. 1998.

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

[Kristensen 96] KRISTENSEN, B.; MAY, D. Activities: Abstractions for Collective Behavior. European Conference on Object-Oriented Programming, ECOOP96, July 8-12 1996.

[Krulwich 96] KRULWICH, B. The BargainFinder agent: Comparison price shopping on the internet. J. Williams, editor, Bots, and other Internet Beasts. Macmillan Computer Publishing: Indianapolis. 1996.

[Lange 98] LANGE, Danny B.; OSHIMA, Mitsuru. Programming and Deploying Java Mobile Agents With Aglets. Addison-Wesley Pub Co. 1998.

[Larman 97] LARMAN, Craig. Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design. Prentice Hall, 1997.

[Lashkari 94] LASHKARI, Y. et al. Collaborative interface agents. Proceedings of the National Conference on Artificial Intelligence. MIT Press, Cambridge, Mass. 1994.

[Leake 96] LEAKE, David. Case-Based Reasoning: Experiences, Lessons, & Future Directions. Edited by David B. Leake. American Association for Artificial Intelligence, 1996.

[Lewis 95] LEWIS, T. et al. Object Oriented Application Frameworks. Prentice Hall, Manning. 1995.

[Ljungberg 92] LJUNGBERG, M.; LUCAS, A. The OASIS air-traffic management system. Proceedings of the Pacific Rim International Conference on Artificial Intelligence, 2., 1992. [S.l.: s.n.], 1992.

[Li 97] LI, G.; WELLER, J.; HOPGOOD, A. Shifting Matrix Management – A Framework for Multi-Agent Cooperation. Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, 1997.

[Lieberman 98] LIEBERMAN, H. Integrating User Interface Agents with Conventional Cooperation Applications. Proceedings of the International Conference on Intelligent User Interfaces, 1998.

[Maes 87] MAES, P. Concepts and Experiments in Computational Reflection. SIGPLAN Notices, New York, v.22 , n.12, p. 147-169. 1987.

[Maes 89] MAES, P. The dynamics of action selection. Proceedings of the Eleventh International Join Conference on Artificial Intelligence (IJCAI-89), Detroit, MI. 1989.

[Maes 91] MAES, P. Situated Agents Can Have Goals. MAES, P. (Eds.). Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back. Cambridge: MIT Press, p.49-70. 1991.

[Maes 94] MAES, P. Agents that reduce work and Information Overload. Communications of the ACM. 37 (7). Julio, 1994.

[Maes 95] MAES, P. Artificial Life Meets Entertainment: Life like Autonomous Agents. Communications of the ACM, 38 (11). 1995.

[Maes 99] MAES, P. et al. Agents That Buy and Sell. Communications of the ACM, 42 (3). March, 1999.

[Malec 94] MALEC, J. A Unified Approach to Intelligent Agency. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents. Berlin: Springer-Verlag, p.233-244. 1994.

[McGuire 93] MC GUIRE, J. et al. DRAFT. Specification on the KQML Agent-Communication Language. June 15, 1993.

[Meyer 97] MEYER B. Object Oriented Software Construction. 2/E. Prentice Hall PTR. 1997

[Minsky 85] MINSKY, M. The Society of Mind. New York: Simond & Schuster. 1985.

[Mitchell 94] MITCHELL, R. et al. Experience with a Learning Personal Assistant. Communications of the ACM. 37 (7). Julio, 1994.

[Moffat 94] MOFFAT, D.; FRIJDA, N. Where There's Will There's an Agent. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents. Berlin: Springer-Verlag, p.245-260. 1994.

[Moore 90] MOORE, R.C. A formal theory of knowledge and action. J.F.Allen, J. Hendler, and A. Tate, editors, Readings in Planning. Morgan Kaufmann. 1990.

[Mowbray 97] MOWBRAY, T,; MALVEAU, R. CORBA Design Patterns. Wiley Computer Publishing. 1997.

[Mullen 95] MULLEN, T.; WELLMAN, M. Some Issues in the Design of Market-Oriented Agents. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents II. Berlin: Springer-Verlag, p.283-298. 1995.

[Müller 94a] MÜLLER, J. et al. Modeling Reactive Behavior in Vertically Layered Agent Architectures. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents. Berlin: Springer-Verlag, p.261-276. 1994.

[Müller 94b] MÜLLER, J.; PISCHEL, M. Modeling interacting agents in dynamics environments. Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI-94). 1994

[Norman 94] NORMAN, T.; LONG, D. Goal Creation in Motivated Agents. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents. Berlin: Springer-Verlag, p.277-290. 1994.

[Norman 95] NORMAN, T.; LONG, D. Alarms: An Implementation of Motivated Agency. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents II. Berlin: Springer-Verlag, p.219-234. 1995.

[Nwana 96] NWANA, H. Software Agents: An Overview. Knowledge Engineering Review. 1996

[Odyssey 97] Odyssey, Odyssey Frequently Asked Questions, http://www.genmagic.com/agents/odyssey-faq.html, General Magic Inc., 1997.

[O'Hare 96] O´HARE, G.; JENNINGS, N. (Eds.) Foundations of Distributed Artificial Intelligence. John Wiley & Sons, Inc. ISBN 0-471-00675-0, 1996.

[Orfali 98] ORFALI, Robert; HARKEY, Dan. Client/Server Programming with Java and Corba. 2/E John Willey & Sons, INC. 1998.

[Parsons 98] PARSONS, S.; SIERRA, C.; JENNINGS, N. Agents that reason and negotiate by arguing. University of London, London. April 8, 1998.

[Platon 370ac] PLATON. 5-Diálogos (Parmenides). Editorial Gredos (ISBN 8424912799). 370 a.c.

[Pool 98] POOL, David; MACKWORTH, Alan; GOEBEL, Randy. Computacional Intelligence, A Logical Approach. Oxford University Press, 1998.

[Pree 95] PREE, W. Design Patterns for Object-Oriented Software Development. Addison-Wesley. 1995.

[Quinlan 93] QUINLAN, J.R. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers. San Mateo California, 1993.

[Rao 91] RAO, A.; GEORGEFF, M. Modeling Rational Agents within a BDI-Architecture. Proceeding of the Knowledge Representation and Reasoning, 1991. California: Morgan Kaufmann, p.473-484. 1991.

[Reticular 99] Reticular Systems Inc. AgentBuilder: An integrated toolkit for constructing intelligent software agents. White paper. http://www.agentbuilder.com/index.html. 1999.

[Riecken 94] RIECKEN, D. M: An Architecture of integrated agents. Communications of the ACM, 37 (7). Lulio 1994.

[Rosenschein 85] ROSENSCHEIN, S. Formal theories of knowledge in AI and robotics. New Generation Computing. 1985.

[Russell 95] RUSSELL, Stuart; NORVIG, Peter. Artificial Intelligence, A Modern Approach. Prentice Hall, 1995.

[Schmid 97] SCHMID, H. Systematic Framework Design by Generalization. Communication ot the ACM 40 (10). October, 1997.

[Selker 94] SELKER, T. Coach: A teaching agent that learns. Communications of the ACM 37 (7). Julio 1994.

[Sen 98] SEN, S.; DURFEE, E. A Formal Study of Distributed Meeting Scheduling. Group Decision and Negotiation, 1998.

[Shaw 96] SHAW, M.; GARLAN, D. Software Architecture. Perspectives on an Emerging Discipline. New Jersey: Prentice Hall, 1996.

[Shoham 93] SHOHAM, Y. Agent-oriented programming. Artificial Intelligence, Amsterdam, v.60, n.1, p.51-92, Mar. 1993.

[Shoham 94] SHOHAM, Y.; COUSINS, S. Logic of Mental Attitudes in AI. Lakemeyer, G.; Nebel, B. (Eds.). Foundations of Knowledge Representation and Reasoning. Berlin: Springer-Verlag, p.296-309. (LNAI, v. 810). 1994.

[Sian 97] SIAN, K.; WEAVER, J.; MATHIS, J.; CASSADY-DORION, L. JAVA Inside. New Riders, Indianapolis, Indiana. 1997.

[Singh 90] SINGH, M. P. Towards a theory of situated know-how. Proceedings of the Ninth European Conference on Artificial Intelligence (ECAI-90). Stockholm, Sweden. 1990.

[Singh 95] SINGH, M. P., GENESERETH, M. R., Syed, M.: A Distributed and Anonymous Knowledge Sharing Approach to Software Interoperation. International Journal of Cooperative Information Systems, Volume 4, number 4, pp. 339-367, 1995.

[Sterling 94] STERLING L.; SHAPIRO E. The Art of Prolog. 2/E. The MIT Press, Cambridge, Massachusetts. London, England. 1994

[Stuart 94] STUART, J., et al. A Knowledge Level Characterization of Multi-Agent Systems. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents. Berlin: Springer-Verlag, p.179-190. 1994.

[Sycara 94] SYCARA, K.; ZENG, D. Visitor-Hoster: Towards an Intelligent Electronic Secretary. Proceeding of the International Conference on Information and Knowledge Management, Workshop on Intelligent Information Agents. 1994.

[Sycara 96] SYCARA, K.; PANNU, M.; ZENG, D.; WILLIAMSON, M. Distributed Intelligent agents. IEEE Expert, Special Issue in Intelligent Systems and their Applications, 11(6). 1996.

[Tarau 97] TARAU, P.; DAHL, V.; DE BOSSCHERE, K.. A Logic Programming Infrastructure for Remote Execution, Mobile Code and Agents. Proceedings of IEEE WETICE'97, Boston, MA, June 1997.

[Tarau 97] TARAU, P.; DAHL, V.; ROCHEFORT, S.; DE BOSSCHERE, K. LogiMOO: a Multi-User Virtual World with Agents and Natural Language Programming. S. Pemberton, editor, Proceedings of CHI'97, p. 323-324, ACM ISBN 0-8979-926-2. March 1997.

[Thomas 93] THOMAS, S.R. PLACA: an Agent Oriented Programming Language. Ph.D. Thesis, Computer Science Department, Stanford University. 1993.

[Vere 90] VERE, S.; BICKMORE, T. A basic agent. Computational Intelligence, Canada, v.6, n.1, p.41-60, Feb. 1990.

[Vives 97] VIVES, F. et al. Un Framework para soportar objetos + hipermedia. Proceeding of the Argentine Symposium on Object Orientation. Buenos Aires, August 11-12, 1997.

[Walrath 97] WALRATH, K.; CAMPIONE, M. The Java Tutorial. 1997.

[Walsh 98] WALSH, T.; PACIOREK, N.; WONG, D. Security and Reliability in Concordia. 31st Annual Hawai International Conference on System Sciences (HICSS31). Kona, Hawaii. January 6-9, 1998.

[Weld 94] WELD, D. An Introduction to Least Commitment Planning. AI Magazine, Summer 1994.

[Wels 98] WELD, D. Recent Advances in AI Planning. Technical Report UW-CSE-98-10-01. Department of Computer Science & Engineering. University of Washington. 1998.

[Werner 88] WERNER, E. Toward a theory of communication and cooperation for multiagent planning. M.Y.Vardi, editor, Proceedings of the Second Conference on Theoretical Aspects of Reasoning About Knowledge. Morgan Kaufman. 1988.

[White 94] WHITE, J.E. Telescrpt technology: The foundation for the electronic marketplace. White paper, General Magic, Inc. 1994.

[White 95] WHITE, J.E. Mobile Agents. White paper de General Magic Inc. http://www.genmagic.com. 1995.

[Witting 92] WITTING, T. ARCHON: An Architecture for Multi-agent Systems. London: Ellis Horwood. 1992.

[Wirfs-Brok 90] WIRFS-BROCK, R. et al. Designing Object-Oriented Software. Prentice Hall, 1990.

[Wooldridge 92] WOOLDRIDGE, M. The Logical Modeling of Computational Multi-agent Systems. Ph.D. Thesis, Department of Computation, UMIST, Manchester, UK. 1992.

[Wooldridge 94] WOOLDRIDGE, M.; JENNINGS, N. Agent Theories, Architectures, and Languages: A Survey. Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents. Berlin: Springer-Verlag, p.1-39. (LNAI, v. 890). 1994.

[Wooldridge 95] WOOLDRIDGE, M.; JENNINGS, N. Intelligent Agents: Theory and Practice. Knowledge Engineering Review. October, 1995.

[Wooldridge 97] WOOLDRIDGE, M. Agent-Based Software Engineering. IEE Proceedings of Software Engineering, 144(1), pp.26-37, February 1997.

[Wooldridge 98] WOOLDRIDGE, M.; JENNINGS, N. Pitfalls of Agent-Oriented Development. Proceeding of the Second International Conference on Autonomous Agents. 1998.

[Wordsworth 92] WORDSWORTH, J. Software Development with Z. New York: Addison-Wesley, 1992.

[Zunino 00] ZUNINO, A.. Brainstorm/J: un framework para agentes inteligentes, Disertación de Maestría. Universidad Nacional del Centro, Facultad de Ciencias Exactas, Instituto de Sistemas (ISISTAN). Marzo 2000.

Apéndice I. Patrones de Diseño

Los patrones de diseño son descripciones de un problema que ocurre repetidas veces en un diseño, junto a una descripción de la solución al problema. Un patrón de diseño OO especifica la comunicación entre objetos y clases que son personalizadas para solucionar un problema de diseño general en un contexto particular [Gamma 95].

Los patrones de diseño que se consideran en este apéndice (*Decorator*, *Facade* y *Strategy*) han sido descriptos en su totalidad en [Gamma 95]. Aquí se introducen estos patrones a los efectos de su empleo en el diseño del framework[1].

Sin embargo hay otros patrones de diseño que son relevantes para su aplicación al diseño de agentes, como por ejemplo *Abstrac Factory*, que provee una interface para crear familias de objetos relacionados sin especificar su clase concreta.

I.A. Decorator

I.A.1. Propósito

El propósito de este patrón de diseño es adicionar responsabilidades a los objetos dinámicamente. Esta alternativa es más flexible que la extensión de responsabilidades por herencia.

I.A.2. Motivación

Otra forma de extender la funcionalidad es a través de subclasificación, es decir extendiendo las clases y creando una jerarquía de clases. Por ejemplo, la clase *Agent*[2] podría ser extendida en *AgentCBR* (adicionando un razonador basado en casos al Agente original), o en *AgentMovil* (incorporando la capacidad de movilidad), etc. Sin embargo la jerarquía creada es compleja[3] y principalmente estática.

Por otro lado es posible lograr esta funcionalidad usando “decoradores”, los cuales tienen una estructura definida en el patrón *Decorator*. Esta alternativa es dinámica porque la nueva funcionalidad es adicionada al objeto (en lugar de a toda la clase), y además no se cambia la interface de éste. La idea principal es cubrir al objeto con un decorador que implementa la funcionalidad agregada. Este decorador recibe los mensajes que iban al objeto decorado, realiza la operación “agregada” reenviando el mensaje al objeto decorado.

Además, existen ocasiones dónde se desea agregar funcionalidad a objetos individuales, y no a toda la clase. El decorador es transparente para los clientes porque tiene la misma interface que el objeto decorado. Esta transparencia permite anidar decoradores recursivamente y el paso del mensaje, del decorador a la capa inferior, puede ser realizada antes o después de ejecutar el comportamiento agregado.

---

[1] La presentación realizada ha sido extraída de [Gamma 95], en particular los diagramas de estructura son reproducidos sin alteración.

[2] Con la funcionalidad básica de los agentes de software.

[3] Cuando se desea incorporar varias de estas características en un mismo Agente

I.A.3. Aplicación

(i) Adicionar responsabilidades a objetos individuales dinámicamente y de forma trasparente, es decir, sin modificar a los clientes de éstos objetos.
(ii) Para adicionar responsabilidades que puedan ser retiradas.
(iii) Cuando la extensión por subclasificación es impráctica. Esto ocurre cuando las posibles extensiones crean una jerarquía de clases compleja, por la diversidad de combinaciones (como en el caso descripto de la clase *Agent*).

I.A.4. Estructura

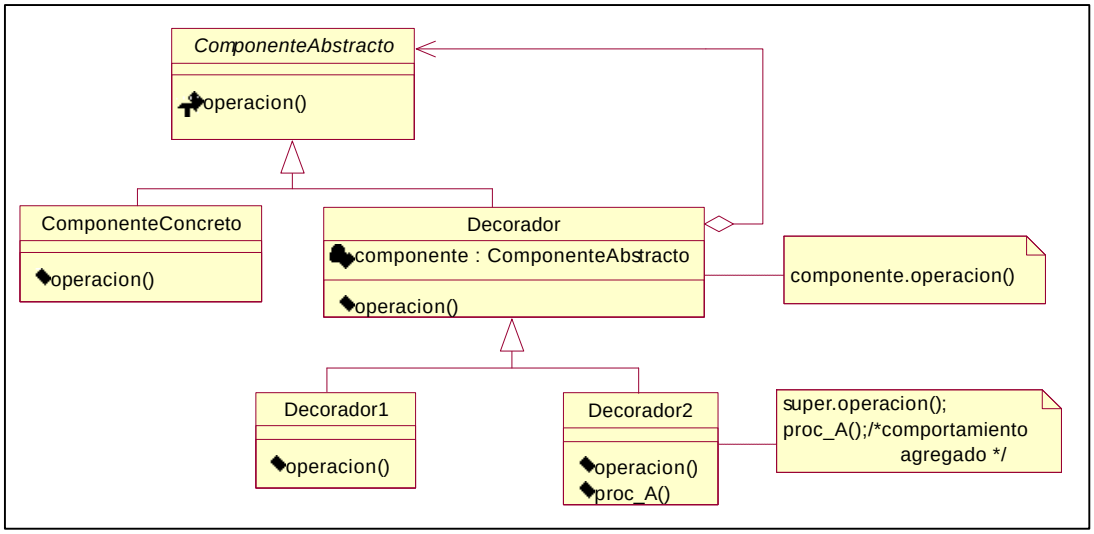


Figura I - 1 – Patrón de diseño *Decorator* [Gamma 95, p.177]

I.A.5. Ejemplo

FraMaS emplea este patrón de diseño para la construcción de cada agente. Así la funcionalidad de un agente está formada por la unión entre un conjunto de operaciones básicas y una serie de "decoradores", que van cubriendo al agente con *funcionalidad avanzada* (estrategias de negociación, capacidades de comunicación, etc.).

Esto brinda libertad en el diseño de los agentes particulares. El framework tiene implementados "decoradores", como así también una serie de métodos *base* y *abstractos* que simplifican la incorporación de nuevos decoradores.

I.B. Facade

I.B.1. Propósito

Provee una interface unificada para un conjunto de interfaces en un subsistema. El patrón de diseño *Facade* define una interface de alto nivel que hace al subsistema más fácil de usar.

I.B.2. Motivación

Se puede reducir la complejidad de un sistema estructurándolo en subsistemas, e intentando minimizar la dependencia y comunicación entre ellos. Para esto se emplea el *objeto facade* que

provee una interface única y simplificada para acceder a las propiedades de los subsistemas (ver siguiente figura).

I.B.3. Aplicación

(i) Cuando se necesita disponer de una interface común para el acceso a un subsistema (complejo). Considerando el caso particular del Agente, compuesto por decoradores que pueden cambiar dinámicamente, es deseable contar con una interfaz simple para los clientes[4]. El *objeto facade* provee una vista por defecto del Agente, lo que es suficiente para la mayoría de los clientes.

(ii) También este patrón de diseño es aplicado cuando existen muchas dependencias entre los clientes y las clases del sistema (Agente), logrando así independencia, es decir portabilidad entre las partes.

(iii) Por otro lado puede ser aplicado el *Facade* cuando se desea hacer una estructura por niveles de los subsistemas. Se define un objeto *facade* para cada entrada de nivel (métodos del protocolo que usaran los clientes).

I.B.4. Estructura

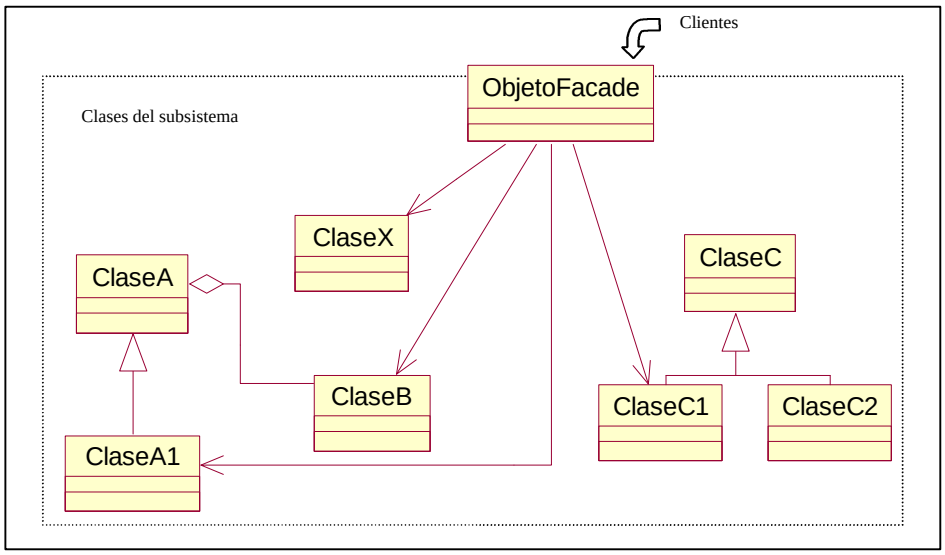


Figura I - 2 – Patrón de diseño *Facade* [Gamma 95, p.187]

I.B.5. Ejemplo

El patrón de diseño *Facade* es incorporado al framework para hacer trasparente desde el entorno multi-agente la carga de un agente y el cambio de estado[5].

Esto debido a que las operaciones en el agente dependen de los decoradores por los que está formado, se logra de esta forma que el entorno multi-agente (cliente) acceda por igual a todos los agentes usando una interface simple (*ObjetoFacade*).

[4] Por ejemplo el entorno multi-agente

[5] Los estados posibles de un Agente son: `initialize, start, stop, finalize`

I.C. Strategy

I.C.1. Propósito

Define una familia de algoritmos encapsulados e intercambiables. Los algoritmos pueden ser usados para definir distintas estrategias de decisión de acuerdo a las necesidades del agente.

I.C.2. Motivación

Existen muchos algoritmos que implementan estrategias de decisión[6] que pueden usar los agentes para decidir que acción ejecutar en el próximo instante de tiempo. Además nuevos algoritmos son desarrollados. Es deseable que cada agente pueda hacer uso de ellos[7], como así también cambiar la estrategia de decisión dinámicamente según la tarea a realizar.

Incorporar esta funcionalidad por completo en las clases del agente hace el empleo de dichas clases más complejo y difícil de mantener. Se pueden evitar estos problemas definiendo clases que encapsulan diferentes algoritmos de decisión. Un algoritmo que es encapsulado de esta forma es llamado *strategy*.

I.C.3. Aplicación

Este patrón de diseño es aplicado cuando:

(i) Existen muchas clases relacionadas que difieren únicamente en su comportamiento. Las estrategias permiten configurar una clase con uno de muchos comportamientos.

(ii) Se necesitan diferentes variantes de un algoritmo. Por ejemplo según el tipo de comportamiento que deba tener el agente en la selección de la próxima acción a ejecutar.

(iii) Un algoritmo usa datos que no son necesarios en los clientes. Se usa este patrón para evitar la exposición de estructuras de datos complejas y específicas al algoritmo.

(iv) Una clase que exhibe muchos comportamientos, que aparecen como muchas sentencias condicionales en sus operaciones. En lugar de las sentencias condicionales se mueven éstas a sus propias clases.

I.C.4. Estructura

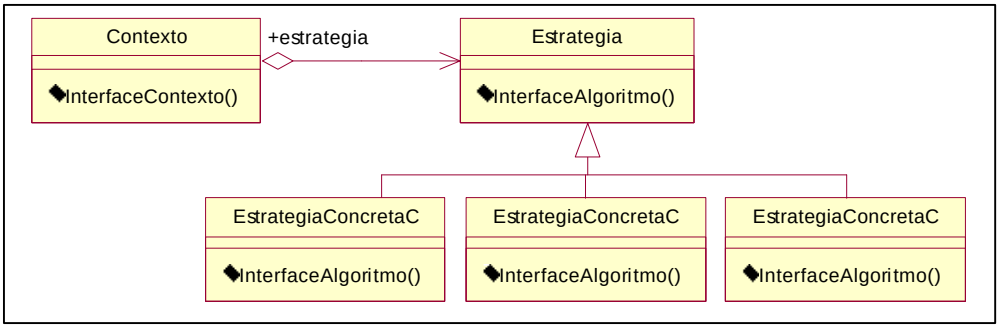


Figura I - 3 – Patrón de diseño *Strategy* [Gamma 95, p.316]

[6] *Planning, scheduling, etc.*
[7] Incluso varios

I.C.5. Ejemplo

En el framework el patrón de diseño *Strategy* es usado para crear una interface común de acceso a diferentes estrategias de decisión. Cada estrategia establece cómo una acción[8] es seleccionada de las posibles para que sea ejecutada en el próximo instante de tiempo.

La clase abstracta *Strategy* de FraMaS establece la interface común que deben respetar todas sus subclases. Básicamente esta interface define un protocolo a través de dos métodos abstractos: `do_action` y `select_action`, los cuales son implementados en cada estrategia de decisión particular.

[8] o un conjunto de acciones

Apéndice II. Lenguaje de especificación Object-Z

II.A. Introducción

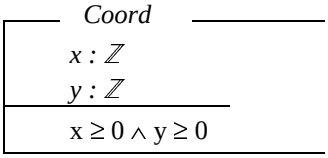
Object-Z es una extensión del lenguaje de especificación formal *Z* [Wordsworth 92] [Diller 94] que incorpora características del paradigma Orientado a Objetos tales como: *clase*, *herencia*, *objeto*, etc. [Duke 91]

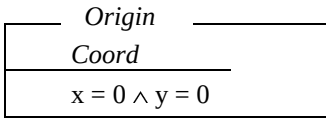
Una especificación *Z* describe un número de esquemas de operación y de esquemas de estado. Un *esquema de estado* agrupa variables y define relaciones que son verdaderas entre los valores de las variables. Un *esquema de operación* define la relación entre uno o más esquemas de estado. Para determinar qué operaciones pueden afectar a un estado particular hay que examinar las descripciones de todas las operaciones.

Los tipos de datos en *Z* están basados en conjuntos. El lenguaje *Z* tiene algunos tipos de datos definidos: números enteros ($\mathbb{Z}$) y números naturales ($\mathbb{N}$). Además permite la incorporación de tipos de datos nuevos (conjuntos). Por ejemplo, se introducen los siguientes conjuntos: [*Agent*, *Service*], que representan el conjunto de los agentes y de los servicios.

Los tipos de datos en *Z* son definidos usando constructores tales como $\mathbb{P}$ (conjunto potencia), $\times$ (producto cartesiano) y esquemas (explicados a continuación). Además se emplean los siguientes símbolos para definir funciones parciales ($\nrightarrow$), funciones parciales finitas ($\pfun$), funciones totales ($\rightarrow$), etc. Así también, el lenguaje Z, soporta las operaciones de conjunto: pertenece ($\in$), unión ($\cup$), subconjunto ($\subseteq$), etc.

Los *esquemas de estado* son declaraciones de variables y predicados (expresados en lógica de primer orden) que establecen restricciones sobre las variables. Por ejemplo:





El esquema *Coord* representa el primer cuadrante de coordenadas con un par de enteros *x*, *y*. El esquema *Origin* incluye la definición de *Coord*, y la restringe estableciendo *x*, *y* con valores iguales a cero. Los esquemas se pueden escribir también de una forma más compacta:

$$Origin \triangleq [Coord \mid x = 0 \land y = 0]$$

Los *esquemas de operación* modelan la transición de estados, relacionando los valores de las variables antes y después de la operación. Las operaciones pueden tener entradas –identificadas con "?"[1], y salidas –identificadas con "!"[2].

[1] Por ejemplo la variable de entrada *input*?

[2] Por ejemplo la variable de salida *output*!

Las variables que pueden cambiar en el esquema de operación son precedidas por el símbolo "Δ". Una descripción completa del lenguaje Z se presenta en [Diller 94].

Object-Z define una estructura de clases a través de un *esquema de clase* que agrupa cada esquema de estado con las operaciones que pueden afectar dicho estado. Las instancias de una clase son llamadas *objetos*. La diferencia con *Z* es que, *Object-Z*, asocia cada operación con un esquema de estado, entonces es simple determinar qué operaciones afectan a un estado particular.

II.B. Sintaxis

Una *clase* es representada por un cuadro vacío o con: (i) la especificación de herencia, (ii) definiciones de tipo y (iii) de constantes, (iv) esquemas de estado, (v) a lo sumo un esquema de estado inicial, (vi) cero o más esquemas de operación, y (vii) un invariable[3]. A la sintaxis del lenguaje *Z* se le agrega este esquema de clase y otros componentes detallados más adelante.

Nombre_de_Clase [parámetros_ generales]

Clases heredadas
Definiciones de tipo
Definiciones de constantes
Esquemas de estado
Esquema de estado inicial
Esquemas de operación

Invariable

Se emplea en este trabajo *Object-Z* para especificar el lenguaje OO Java, utilizando en la implementación del framework. En [Dong 96] se describe cómo usar *Object-Z* para especificar lenguajes de programación OO. A continuación se presentan algunos puntos de dicha descripción.

La *herencia* es un mecanismo empleado para realizar una especificación de forma incremental, permitiendo la derivación de nuevas clases a partir de las que ya han sido definidas. Cuando una clase hereda de otra se adicionan en la nueva clase los tipos, constantes, variables y operaciones de sus ancestros, sobreescribiendo aquellos con igual nombre. A los efectos del presente trabajo se especifican las implementaciones de las *interfaces* en esta misma sección del esquema de clase[4] (cuando se implementa una interfaz se hace explícito en la declaración).

Las definiciones de constantes y las variables son llamadas "atributos". El *esquema de estado* no tiene nombre y contiene las variables de estado. El *esquema de estado inicial* tiene el nombre INIT. Los *esquemas de operación* contienen una lista "Δ"con las variables en la declaración que pueden cambiar cuando la operación es aplicada a un objeto de la clase.

Si *Agent* en una clase, entonces `myAgent : Agent` es un objeto de la clase *Agent*. Análogamente, si `start` es una operación definida en *Agent* y `agentContext` es una variable de instancia, entonces `myAgent.start` representa la aplicación de la operación `start` sobre *myAgent* y `myAgent.agentContext` representa el valor del atributo `agentContext` del objeto *myAgent*.

La declaración `c, d : C`, denota que `c` y `d` son objetos del tipo `C`, aunque podrían ser el mismo objeto; si se desea que los objetos sean diferentes se debe especificar $c \neq d$). El subíndice: ©, representa que un objeto no se contiene a si mismo (directa o indirectamente) y que ningún objeto es directamente contenido en dos objetos diferentes.

[3] opcional
[4] Debido a que ninguna de las referencias citadas hace uso de interfaces se adopta esta solución.

El *símbolo de proyección*, ↾, indica la visibilidad, es decir aquellas constantes, variables, etc. que serán accesibles a través de la notación de punto (`objeto.operación`). Por defecto todas las características son visibles.

La declaración `sc : ℙ c` modela una *agregación* de objetos de la clase `c`.

El *invariable* es un predicado opcional que puede restringir el comportamiento de los objetos, agregando restricciones expresadas en lógica temporal. En esta sección se pueden definir operaciones compuestas por operaciones definidas en los esquemas de operaciones. Por ejemplo:

```
Operación_compuesta ≙ operacion1 • operacion2
```

permite que las variables declaradas en `operacion1` sean interpretadas en `operacion2`.

Entre los operadores de *Object-Z* se encuentran:

- El *operador paralelo*, ‖, es un operador binario usado para la comunicación entre objetos, es decir que iguala las salidas de un esquema con las entradas del siguiente esquema.
- *El operador secuencial*, ⨾, no es conmutativo, ni asociativo, y es usado también para la comunicación entre objetos (de izquierda a derecha estrictamente).
- El *operador de selección*, [], indica la selección no determinística de una operación que satisface las precondiciones.

Todas las operaciones en *Object-Z* son relaciones o combinaciones de relaciones, por ejemplo intersección y composición secuencial.

Un *método* de una clase *Object-Z* es una operación que está definida en la clase. El estado interno de un objeto de una clase puede cambiar por la ejecución de un método de dicha clase. Un método consiste de un nombre (*id*), una lista de parámetros formales (*lpf*), un conjunto de variables locales (*locVar*) y un cuerpo con una secuencia de sentencias (*body*):

nombre_del_método

$$
\begin{array}{|l}
\hline
id : Id \\
lpf : seq\ VarRef \\
locVar : \mathbb{P}\ VarRef \\
body : SeqStmt \\
\hline
\#\{r : (VarRef \cap scontain) \bullet r.id\} = (VarRef \cap scontain) \\
\hline
\end{array}
$$

$$Execute \triangleq body.Execute$$

El *invariable* especifica que, en un método, cualquier par de referencias de variables que tengan el mismo identificador deben tener la misma referencia. El significado de un método es la ejecución de las sentencias del cuerpo del método. (`body.Execute`).

Un bucle `for` es representado por el siguiente esquema compacto:

```
BucleFor ≙ (i:1..10 • operacion1)
```

O por la siguiente sentencia:

```
⨾ i: 1..10 • operacion1
```

La estructura repetitiva `while` es representada por el siguiente esquema:

while

test : BoolExpr
body : SeqStmt

$Execute \triangleq ([\ test.obj.val\ ] \wedge body.Execute\ ) \fatsemi Execute$
$\Box$
$[\neg test.obj.val]$

De forma análoga se extiende el modelo presentado para contemplar otros constructores, tales como: clases predefinidas (*Boolean*, *String*), expresiones booleanas, etc.

En el caso particular del constructor de una clase se especifica según el modelo de lenguaje OO definido en [Dong 96] especificando el nombre de la clase y la invocación al método *Create*:

NombreClase.Create

La operación *Create* especifica que un nuevo objeto de esa clase es creado y la dirección (identidad) del objeto puede ser usada en el entorno. Junto a la creación del objeto en si, la invocación de este método envía el mensaje "`INIT`" al objeto creado. En el modelo de *Object-Z* usado en el presente trabajo es posible definir en un esquema de clase más de un método "`INIT`", aunque éstos deben tener parámetros de entrada diferentes.

Apéndice III. Reflexión estructural

III.A. Introducción

La *reflexión computacional*[1] es la capacidad de un sistema de software para razonar y actuar sobre si mismo, como así también para autoajustarse a las condiciones de cambio [Maes 87]. Por otro lado, la *reflexión*, es la capacidad de un programa de manipular su estado como si fueran datos [Gabriel 91].

Surgen dos conceptos con el de reflexión: uno relacionado con la capacidad del programa de observar y razonar sobre su estado interno[2] (*introspection*); y el otro relacionado con la capacidad del programa para modificar su propio comportamiento sin necesidad de hacer cambios en el código de la aplicación (*effectuation*).

La *reflexión* surge como una solución al problema de crear aplicaciones simples de mantener, usar y capaces de ser extendidas y/o modificadas.

Una *arquitectura reflexiva* es vista como la composición de dos partes: la parte de la aplicación y la parte reflexiva. La tarea de la aplicación es resolver problemas y retornar los resultados al "mundo exterior", mientras que la tarea de la parte reflexiva es solucionar problemas y retornar información sobre la aplicación.

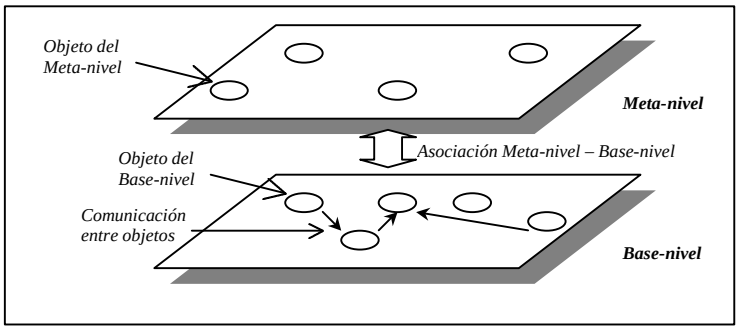


Figura III - 1 – Arquitectura reflexiva

Los lenguajes de programación que soportan arquitecturas reflexivas permiten acceder y manipular sus propias representaciones dinámicamente. Entre sus características se encuentran [Maes 87]: la representación del modelo del sistema (proveyendo las primitivas para el acceso a las representaciones internas del programa), algún mecanismo de reflexión (que haga posible interrumpir la ejecución del programa –aplicación para seguir con la ejecución de la parte reflexiva y viceversa; esto puede ser hecho implícitamente o explícitamente) y una conexión causal (entre los datos y el sistema en si).

En el paradigma orientado a objetos (OO) se representan las arquitecturas reflexivas como un conjunto de objetos de *nivel base*[3] y un conjunto de objetos de *nivel meta*, es decir la parte

[1] o simplemente *reflexión*
[2] es decir conocer que métodos, variables, etc. tienen las clases
[3] la aplicación

reflexiva. Ambos niveles están asociados de forma que los cambios en el nivel base son reflejados en el nivel meta. El nivel meta tiene acceso a la información del nivel base, sin embargo éste último no tiene conocimiento del nivel meta.

Esta jerarquía de dos niveles puede ser generalizada a un número más grande de niveles reflexivos, trasformándose en una arquitectura de *n* niveles, donde cada nivel meta puede hacer diferentes computaciones.

En el paradigma OO la reflexión está asociada con obtener datos sobre las propiedades y el comportamiento de las clases y objetos. Así como también la posibilidad de cambiar estos datos, es decir las propiedades de las clases y objetos.

III.B. Modelos y taxonomía de reflexión

Ferber [Ferber 89] identifica dos *modelos* de reflexión en el paradigma OO: uno llamado *reflexión estructural* y otro llamado de *reflexión comportamental*.

El primero, la *reflexión estructural*, representa capacidad del lenguaje para permitir en el nivel meta una representación completa del programa ejecutado y de los tipos de datos abstractos (TDA) [Cointe 97].

El segundo, la *reflexión comportamental*, representa la capacidad del lenguaje para permitir en el nivel meta una representación de su semántica e implementación, como así también de los datos y la implementación del sistema de run-time [Maes 87].

Una *taxonomía de reflexión* clasifica los tipos de reflexión posibles. El mecanismo de reflexión puede ser desarrollado usando estrategias diferentes, éstas tienen relación con el momento en que el thread de control es re-dirigido al nivel meta. Básicamente, cuando un objeto recibe un mensaje, el mecanismo de reflexión re-dirige el thread de control al meta-objeto asociado en el nivel meta, éste realiza su ejecución y el thread de control es retornado al objeto de nivel base.

Cuando todos los objetos y métodos de una clase tienen un comportamiento reflexivo es una *reflexión de clase*. Por otro lado, si el comportamiento reflexivo se aplica a algunos métodos de la clase es llamada *reflexión de métodos*. Mientras que si las aplicaciones tienen algunos objetos de una clase con comportamiento reflexivo se denomina *reflexión de objetos*. Por último, el comportamiento reflexivo puede ser aplicado a algunos atributos de una clase, así esta estrategia es llamada *reflexión de atributo*.

La principal desventaja de las arquitecturas reflexivas en general es la performance obtenida, esto debido a la multiplicación en el paso de mensajes entre el nivel base y el (los) meta niveles. En general es una solución entre flexibilidad (de las arquitecturas reflexivas) y performance (de un sistema implementado de forma estándar).

III.C. Reflexión en Java

Los lenguajes de programación actuales no soportan reflexión computacional, sin embargo se puede incorporar en algunos un soporte básico. Entre los lenguajes que han sido extendidos para soportar los conceptos de reflexión se encuentran: Java, Smalltalk, y C++.

El lenguaje de programación Java[4] permite la reflexión estructural, haciendo posible que un objeto conozca en tiempo de ejecución qué métodos y variables posee. Con estas características es posible en Java la invocación de métodos dinámicos, es decir que en tiempo de ejecución cada objeto puede preguntarse si dispone de un método y si es así invocarlo.

Esta característica de invocación de mensajes dinámicos, disponible en lenguajes dinámicos como Smalltalk y Objective C, permite que un objeto envíe un mensaje a otro objeto y la comprobación de tipos se haga en tiempo de ejecución.

[4] *Java versión 1.1 o superior*

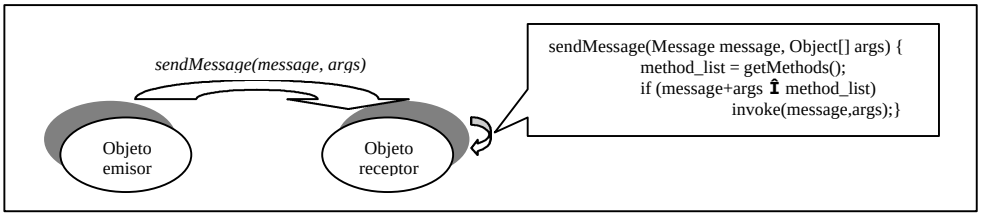


Figura III - 2 – Mensajes dinámicos en Java

Sin embargo, la invocación de mensajes dinámicos no puede ser realizada directamente en Java, porque éste tiene un chequeo de tipos estático[5]. El sistema que permite enviar mensajes dinámicos en Java es esquematizado en la Figura III - 2. Se observan dos objetos: uno emisor y otro receptor del mensaje "sendMessage" con argumentos "message" y "args".

El mensaje "sendMessage" forma parte de la estructura de las clases a las que pertenecen ambos objetos (el emisor y el receptor) y tiene implementada la funcionalidad para que el objeto conozca (en tiempo de ejecución) que mensajes entiende. Así si el objeto entiende el mensaje "*message*" con argumento "*args*" lo invoca.

Usando mensajes dinámicos de acuerdo a la descripción aquí presentada es posible simplificar considerablemente las clases del framework que conforman los decoradores, haciendo más simple la incorporación de nuevos decoradores al framework, como así también la comprensión de su funcionamiento.

[5] *en tiempo de compilación*