Patrones de implementación para incluir comportamientos proactivos

Mailyn Moreno, Alternán Carrasco, Alejandro Rosete y Martha D. Delgado

***Resumen*—La programación orientada a objeto enfrenta retos como es el desarrollo de software en ambientes distribuidos. En esta línea ha surgido el paradigma de agentes. Un agente exhibe comportamientos que lo diferencia de un objeto, como la autonomía y la proactividad. La proactividad permite desarrollar sistemas dirigidos por metas, en los que no es necesaria una petición para que se inicie un trabajo. Incorporar proactividad a un software es hoy una necesidad, existe una gran dependencia de los sistemas computarizados y es mayor la delegación de tareas en ellos. Los patrones se han utilizado con éxito en la reducción de tiempo de desarrollo y el número de errores en el desarrollo de software, además de ser una guía para resolver un problema típico. En este trabajo se presentan dos patrones de implementación para incorporar proactividad en un software y facilitar el trabajo con los agentes. Se incluye un caso de estudio del uso de los patrones propuestos en un observatorio tecnológico.**

***Palabras claves*—Agente, patrones, patrón de implementación, proactividad.**

Implementation Patterns to include Proactive Behaviors

***Abstract*—Object oriented programming is facing challenges such as the development of software in distributed environments. Along this line has emerged the paradigm of agents. An agent shows behaviors, such as autonomy and proactivity, that differentiates it from an object. Proactivity allows developing goal-directed systems, in which a request is not necessary to start a task. Adding proactivity to a software is nowadays essential, there is a big dependence on computer systems and it is greater the delegation of tasks to them. The patterns have been used successfully in reducing development time and the number of errors in software, besides of being a guide to solve a typical problem. In this paper, we present two implementation patterns to add proactivity to software and to make it easier to work with agents. A case study about the development of a technology observatory using both patterns is also included.**

***Index Terms*—Agent, patterns, implementation pattern, proactivity.**

Manuscrito recibido el 18 de marzo de 2013; aceptado para la publicación el 23 de mayo de 2013.

Mailyn Moreno, Alejandro Rosete y Martha D. Delgado pertenecen a la Facultad de Ingeniería Informática, Instituto Superior Politécnico "José Antonio Echeverría", La Habana, Cuba (e-mail: {my, rosete, marta}@ceis.cujae.edu.cu).

Alternán Carrasco pertenece al Complejo de Investigaciones Tecnológicas Integradas, La Habana, Cuba (e-mail: acarrasco@udio.cujae.edu.cu).

I. Introducción

El paradigma de agentes constituye una tecnología prominente y atractiva en la informática actual. Los agentes y los sistemas multiagente están contribuyendo actualmente en dominios diversos, tales como recuperación de datos, interfaces de usuario, comercio electrónico, robótica, colaboración por computadora, juegos de computadora, educación y entrenamiento, entre otras [1]. Además los agentes están emergiendo como una nueva manera de pensamiento, como un paradigma conceptual para analizar problemas y diseñar sistemas, y ocuparse de la complejidad, distribución e interactividad; mientras que proporcionan una nueva perspectiva en la computación y la inteligencia [1]. Los agentes son entidades que poseen propiedades como la autonomía, la proactividad, la habilidad social, entre otras [2]. Existen agentes orientados a metas, que le dan solución a diferentes problemas. Entre las propiedades más significativas que diferencian a los agentes de los objetos está la proactividad, es decir, los agentes no sólo actúan en respuesta a su ambiente sino que son capaces de tener comportamiento orientado a metas [3].

En la actualidad, los software mayormente se construyen bajo el paradigma de orientación a objetos, ya que este paradigma ha alcanzado un gran auge [4]. En nuestros días hay una tendencia elevada de utilizar enfoques orientados a objetos en todos los sistemas que se construyen, debido a las facilidades que brinda para la reutilización del código [4]. Los patrones de diseño son una muestra indiscutible de la fortaleza que tiene la orientación a objetos [5].

Los patrones son una solución a problemas típicos y con su empleo se puede hacer un desarrollo de software más rápido y con mayor calidad [6].

En el desarrollo de un software orientado a objetos no se tiene en cuenta los comportamientos proactivos que se puedan incluir, los que pueden ser beneficiosos a los usuarios finales. Esto se debe a la naturaleza propia del objeto que exhibe un comportamiento mediante la invocación de un método [7].

La proactividad es una característica muy beneficiosa para el software actualmente, se desea que los programas trabajen por las personas con sólo saber sus intereses. Con la proactividad se pueden obtener asistentes personales en las computadoras que ayuden en la búsqueda de información [8] y a la hora de la toma de decisiones [9]. En la vigilancia tecnológica la proactividad es muy provechosa [10]. Según Henderson-Seller es posible obtener un sistema híbrido

agente + objetos, donde se tenga en un software orientado a objeto con características de los agentes [3].

Se han desarrollado trabajos con patrones para la orientación a agentes con son [11], [12] pero estos no se enfocan en la implementación, o en problemas medulares como la proactividad.

En este trabajo se hace una propuesta de patrones de implementación que utilizan como base los patrones de diseño de la orientación a objetos y las recomendaciones de trabajos de patrones para la orientación a agentes para incorporar proactividad a un software. Se aprovecha las ventajas que provee la filosofía y las plataforma de desarrollo de agentes para incluir este comportamiento. Se desarrolla un caso de estudio sobre un observatorio tecnológico para aplicar los patrones propuestos.

II. Ingeniería de software

La ingeniería de software es el uso de los principios de ingeniería robustos, dirigidos a obtener software económico de gran fiabilidad, además de ser capaces de trabajar sobre las máquinas reales con las que se cuentan. En el desarrollo de un software es fundamental o casi imprescindible utilizar los principios de la ingeniería de software. La misma comprende todos los flujos de trabajo dentro el desarrollo de un software, el análisis, el diseño del sistema, la implementación, las pruebas, el control de versiones entre otros [13].

La ingeniería de software ha evolucionado por diferentes etapas para llegar a lo que existe hoy en día. Por ejemplo pasó por el enfoque estructurado, luego llegó el enfoque orientado a objetos. El paradigma orientado a objetos fue un cambio en la forma de pensar acerca del proceso de descomposición de problemas [7]. Un objeto encapsula estados (valores de datos) y comportamientos (operaciones). En la programación orientada a objetos la acción se inicia mediante la trasmisión de un mensaje al objeto. Un objeto exhibirá su comportamiento mediante la invocación de un método como respuesta a un mensaje [7].

El enfoque orientado a objetos está lejos de ser perfecto para el desarrollo de un software, pero para la mayoría de los desarrolladores es lo mejor que existe para el desarrollo de los mismo [4]. En nuestros días hay una tendencia elevada de utilizar enfoques orientados a objetos en todos los sistemas que se construyen, debido a las facilidades que brinda para la reutilización del código [14].

En el desarrollo del software orientado objeto ha tomado auge la utilización del Proceso Unificado de Desarrollo (*RUP*, *Rational Unified Process*) [4], que es una propuesta de proceso para el desarrollo de software orientado a objetos que utiliza *UML* (*Unified Modelling Language*) [15], [16] como lenguaje que permite el modelado de sistemas con tecnología orientada a objetos.

RUP es un proceso de desarrollo dirigido por casos de uso. Según [4] "Un caso de uso especifica una secuencia de acciones, incluyendo variantes, que el sistema puede llevar a cabo, y que producen un resultado observable de valor para un actor concreto". La comunicación para iniciar un caso de uso es a través de un mensaje o petición de un actor. Esto implica que los mismos no son autoiniciables, no tienen la iniciativa de hacer algo sin una petición u orden. Este comportamiento es intrínseco en la orientación a objetos porque los objetos trabajan para dar respuesta a un mensaje.

A. Tendencias de la Computación

La historia de la computación en la actualidad se ha caracterizado por cinco importantes y continuas tendencias [1]:

1. Ubicuidad

La ubicuidad es una consecuencia de la reducción constante en el costo de la computación, posibilitando introducir el poder de procesamiento en los lugares y con dispositivos que no han sido rentables hasta ahora.

2. Interconexión

Hace dos décadas los sistemas de computadoras eran entidades generalmente aisladas, solo se comunicaban con los operadores humanos. Los sistemas de computadora hoy en día se conectan a una red en grandes sistemas distribuidos. Internet es el claro ejemplo en que se evidencia la dificultad de encontrar computadoras que no tengan la capacidad y necesidad de acceder a Internet.

3. Inteligencia

Esta tendencia está dirigida hacia sistemas cada vez más complejos y sofisticados. Es por ello que la complejidad de las tareas que es capaz el ser humano de automatizar y la delegación en las computadoras ha crecido regularmente.

4. Delegación

La delegación implica que se le dé el control a sistemas informáticos de tareas cada vez más numerosas e importantes. Se observa con regularidad que se delegan tareas a los sistemas de computadoras como pilotear aeronaves.

5. Orientación a humano

Esta última tendencia trata acerca del trabajo constante de aumentar el grado de abstracción de las metáforas que se usan para entender y usar las computadoras. Estas se acercan cada vez más a la forma humana de actuar, que reflejen la forma en que el humano entiende el mundo. Esta tendencia es evidente en todas las formas en que se interactúa con las computadoras.

B. Retos

Ante las nuevas tendencias, la orientación a objetos y *RUP* tratan de adaptarse a los requisitos de los sistemas distribuidos abiertos. Uno de los autores del conocido *RUP*, Grady Booch, ha planteado la necesidad de nuevas técnicas para descomponer (dividir en pedazos más pequeños que puedan tratarse independientemente), abstraer (posibilidad de modelar concentrándose en determinados aspectos y obviando otros detalles de menor importancia), organizar jerárquicamente (posibilidad de identificar organizaciones, gestionar la relaciones entre los componentes de la misma solución que

incluso permitan su tratamiento de grupo como un todo según convenga y ver cómo lograr que entre todos se haga la tarea) [17].

En esta misma línea de desarrollo de software para ambientes distribuidos cada día toma más fuerza un paradigma que muchos consideran como el próximo paso de avance en la tecnología de desarrollo de software: los agentes [17].

Construir software que resuelvan problemas de negocios actuales no es una tarea fácil. Al incrementarse las aplicaciones sofisticadas demandadas por diversos tipos de negocios y competir con un ventaja en el mercado las tecnologías orientadas a objetos pueden ser complementada por las tecnologías orientas a agentes [3].

III. AGENTES

Aunque no hay total unificación en cuanto a qué es un agente, un intento de unificar los esfuerzos para el desarrollo de esta tecnología puede encontrarse en FIPA (*Foundation for Intelligent Phisical Agents*) [18] que los define como una entidad de software con un grupo de propiedades entre las que se destacan ser capaz de actuar en un ambiente, comunicarse directamente con otros agentes, estar condicionado por un conjunto de tendencias u objetivos, manejar recursos propios, ser capaz de percibir su ambiente y tomar de él una representación parcial, ser una entidad que posee habilidades y ofrece servicios, que puede reproducirse, etc. [1].

De forma general, varios autores reconocen en los agentes diversas propiedades, entre las que se destacan el ser autónomos, reactivos, proactivos y tener habilidad social [17], [19], [20].

Los agentes brindan una vía efectiva para descomponer los sistemas complejos, son una vía natural de modelarlos y su abstracción para tratar las relaciones organizacionales es apropiada para estos sistemas [2].

Franklin, después de estudiar doce definiciones, llegó a la conclusión que los agentes tienen entre sus propiedades principales la autonomía, estar orientados a metas, ser colaborativos, flexibles, autoiniciables, con continuidad temporal, comunicativos, adaptativos y móviles. Agrega que un agente autónomo es un sistema situado en un ambiente que percibe el ambiente y actúa sobre él, en el tiempo, según su agenda propia y de esta manera produce efectos en lo que él mismo podrá sentir en el futuro [21].

Russel y Norvig, tienen una visión más flexible de los agentes, como una herramienta para analizar sistemas y no como una característica abstracta que divida al mundo en agentes y no-agentes [22].

De forma general las anteriores definiciones son válidas, con distintos grados de amplitud y reflejando aspectos diferentes, aunque ninguna entra en contradicción con las otras. Se puede decir que las propiedades fundamentales de los agentes son: autonomía, reactividad, proactividad y habilidad social. Las mismas se pueden resumir como sigue [1].

1. Autonomía

Actúan totalmente independientes y pueden decidir su propio comportamiento, particularmente como responder a un mensaje enviado por otro agente.

2. Reactividad

Perciben del entorno y responden a los cambios de éste.

3. Proactividad

No sólo actúan en respuesta a su ambiente, sino que son capaces de tener comportamiento orientado a metas y objetivos. Pueden actuar sin que exista una orden externa, tomando la iniciativa.

4. Habilidad *Social*

Tienen la capacidad de interactuar con otros agentes mediante algún mecanismo de comunicación. Esto le permite lograr metas que por sí solos no puede lograr.

Lo novedoso de los agentes es que pueden ser proactivos, tienen un alto grado de autonomía y están situados en un entorno con el que interactúan. Esto se hace especialmente cierto en áreas como ambientes inteligentes, negocio electrónico, servicios Web, bioinformática, entre otras. Estas áreas demandan software que sean robustos, que puedan operar con diferentes tipos de ambientes, que puedan evolucionar en el tiempo para responder a los cambios de los requisitos, entre otras características.

La mayor diferencia del enfoque orientado a agentes con el enfoque orientado a objetos, es que los agentes pueden tener autonomía, mostrar comportamientos proactivos que no se puedan predecir completamente desde el inicio [23].

Uno de los retos que enfrenta la orientación a objetos además, es sencillamente que no permite capturar varios aspectos de los sistemas de agentes. Es difícil capturar en un modelo de objetos nociones de los agentes como acciones que se hacen proactivamente o reacciones dinámicas a un cambio de su entorno.

Es la proactividad una de las características más distintivas de los agentes [1], [24]. La proactividad es un comportamiento dirigido por metas. El agente trabaja para alcanzar una meta. El comportamiento proactivo permite que se le pase las metas al software y este trabaje para cumplirlas cuando tenga las condiciones para hacerlo.

Los agentes al tener habilidad social normalmente no se les encuentra solos en un sistemas, sino que un sistema puede estar compuesto por más de un agente. Los Sistemas Multiagente (SMA) están compuestos por agentes que tienen conocimiento sobre su entorno, que cumplen con objetivos y metas determinadas por sus responsabilidades. Estos agentes no son independientes aunque sí autónomos en mayor o menor medida. Son entidades de un todo, donde pueden interactuar entre ellos informando y consultando a otros agentes teniendo en cuenta lo que realiza cada uno de ellos, llegando a ser capaces de conocer el papel que tienen todos dentro del sistema según la capacidad que cada uno tenga de actuar y percibir [1].

Un aspecto clave para el desarrollo de los SMA ha sido la especificación de los lenguajes de comunicación de agentes (ACL por sus siglas en inglés) [25]. Un ejemplo de ACL es FIPA-ACL [26], [27],donde se define una biblioteca con una lista de actos comunicativos estándares, cada uno descrito con los parámetros y sus significados, junto a especificaciones en una lógica de precondiciones y efectos racionales. Además, los mensajes pueden ser descritos según un acto comunicativo, una acción ejecutada en un contexto determinado que implica un grupo de consecuencias, que permite a los agentes entender la intención del mensaje recibido, en la forma de compromisos, derechos y comportamientos.

Para el desarrollo de sistemas multiagente existen varias plataformas de desarrollo. JADE[1] está entre las más conocida y utilizada de las plataformas por las facilidades que brinda, entre las que están permitir el desarrollo de aplicaciones de agentes en el cumplimiento con las especificaciones FIPA para sistemas multiagente [28].

La plataforma de agentes JADE trata de mantener en alto el funcionamiento de un sistema de agentes distribuidos con el lenguaje Java. De acuerdo con el enfoque de sistemas multi-agente, una aplicación sobre la base de la plataforma JADE se compone de un conjunto de agentes cooperantes que se pueden comunicar entre sí a través del intercambio de mensajes. Cada agente está inmerso en un ambiente sobre el que puede actuar y en los cuales los acontecimientos pueden ser percibidos. El ambiente puede evolucionar de forma dinámica y los agentes aparecen y desaparecen en el sistema de acuerdo a las necesidades y los requisitos de las aplicaciones. JADE proporciona los servicios básicos necesarios para la distribución de aplicaciones en el ambiente permitiendo a cada agente descubrir a otros dinámicamente y comunicarse con ellos [29].

Para desarrollar un sistema multiagente los desarrolladores necesitan implementar algunas funcionalidades que son de vital importancia para la ejecución del sistema. Entre las características más comunes están las siguientes:

1. Ejecución y control de la plataforma que contiene a los agentes.
2. Gestionar el ciclo de vida de los agentes.
3. Comunicar los agentes que viven dentro del sistema.
   - 3.1. Enviar y recibir mensajes desde y hacia otros agentes.
   - 3.2. Procesar el mensaje y tomar acciones dependiendo de su contenido.
4. Comunicarse con los agentes coordinadores de la plataforma.
   - 4.1. Ver el estado de algún módulo del sistema.
   - 4.2. Buscar un agente para ver su estado.

[1] Java Agent DEvelopment Framework, http://jade.tilab.com

IV. PATRONES

El desarrollo de software basado patrones y modelos está rehaciendo el mundo de los desarrolladores de software [6].

De acuerdo con el diccionario de inglés Oxford[2] un patrón "es una forma lógica o regular", "un modelo", "el diseño o las instrucciones para hacer algo" o "un ejemplo excelente". Todos estos significados se aplican en el desarrollo de software, según [30] la tercera definición es la más acertada.

Christopher Alexander dice que, "Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, entonces describen el núcleo de la solución para ese problema, de manera tal que usted pueda utilizar esta solución un millón de veces, sin tener que hacerlo dos veces de la misma forma" [31]. Aunque es una definición para la arquitectura y la construcción, se puede utilizar para el desarrollo de software. Los patrones, según la disciplina de la Ingeniería de Software en que se manifiesta el problema que resuelven, pueden ser de diseño, de implementación, etc. [5], [14].

"Los patrones de diseños son descripciones de las comunicaciones entre objetos y clases que son personalizables para resolver un problema general de diseño en un contexto particular" [5].

Los patrones de implementación son un módulo de software único en un lenguaje de programación en particular. Una característica crucial es que son fácilmente reconocibles por el software, lo que facilita la automatización [14], [32].

Los patrones de implementación comparten muchos beneficios con los patrones de diseño por ejemplo establecen un vocabulario. Como los patrones de diseño, los patrones de implementación proveen un medio para organizar y trasmitir una base de conocimiento.

Según Beck "Los patrones de implementación proveen un catálogo de problemas comunes en programación y la forma en que […] se pueden resolver estos problemas" [14].

Los patrones de implementación permiten que el trabajo de los programadores sea más efectivo a medida que gastan menos tiempo en partes mundanas y repetitivas de su trabajo y le dedican más tiempo a resolver problemas verdaderamente únicos [14].

En general, un patrón tiene cuatro elementos esenciales [5]:

1. Nombre del patrón

El nombre del patrón es un indicador que se usa para describir un problema, sus soluciones y consecuencias en pocas palabras.

2. El problema

El problema describe cuándo aplicar el patrón, explicando el problema y su contexto.

3. La solución

La solución se compone de los elementos que resulten el problema, sus relaciones, responsabilidades y las colaboraciones.

[2] Oxford: Oxford English Dictionary, http://www.oed.com

4. Las consecuencias

Las consecuencias son los resultados y los cambios resultantes de aplicar el patrón lo que incluye su impacto sobre la flexibilidad de un sistema, la extensibilidad o la portabilidad.

Los patrones como idea y principio se pueden utilizar tanto en la orientación a objetos [5] y en la orientación a agentes [11], [12], [33]. En la orientación a objetos los patrones más conocidos y utilizados son los patrones de diseño, ya que los mismos se pueden llevar hasta la implementación.

A. Patrones de Diseño en la orientación a objetos

Los patrones de diseño pueden ser de diferentes tipos dentro de los cuales se encuentran los creacionales, los estructurales y los de comportamiento [5], [6].

1. Patrones de creación:

Ayudan a hacer un sistema independientemente de cómo son creados, compuestos y representados sus objetos. Un patrón creacional de clase utiliza la herencia para cambiar la clase que es instanciada, mientras que un patrón creacional de objeto delegará la particularización a otro objeto. Dentro de estos se encuentran: *abstract factory, builder, factory method, prototype y singleton*.

2. Patrones estructurales

Los patrones estructurales se relacionan con el modo en que las clases y objetos son compuestas para formar estructuras más grandes. Las clases de patrones estructurales usan, por ejemplo, la herencia para componer o implementar interfaces. Por ejemplo considere como la herencia múltiple mezcla dos o más clases en una; el resultado es una clase que combine las propiedades de su clase padre. Este patrón es particularmente útil para hacer bibliotecas de clases independientes desarrolladas para trabajar juntas. Algunos de estos patrones estructurales son: *adapter, bridge, composite, decorator, facade, flyweight y proxy*.

3. Patrones de comportamiento

Los patrones de comportamiento tienen relación con los algoritmos y la asignación de responsabilidades entre objetos. Este no solo describe patrones de objetos o clases, sino también la comunicación entre ellos. Estos patrones caracterizan flujos de control que son difíciles de seguir en tiempo de ejecución. Cambian su enfoque de flujo de control para dejar que se concentre en la manera en que los objetos son interconectados. Dentro de estos se localizan los siguientes patrones: *chain of Responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method y visitor*.

En este punto se quiere hacer énfasis en el patrón *Observer*, conocido también por Dependencia o Publicación-Suscripción. Este patrón define una dependencia de uno a muchos entre objetos, de modo que cuando un objeto cambia de estado, todas sus dependencias son notificadas y actualizadas automáticamente.

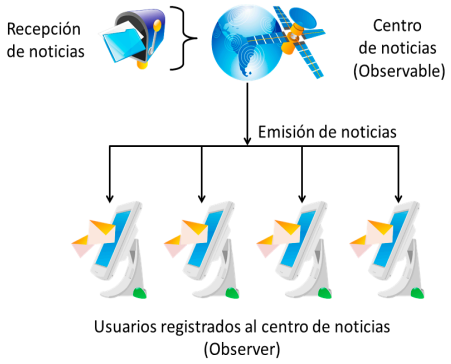

Fig. 1. Ejemplo del patrón *Observer*.

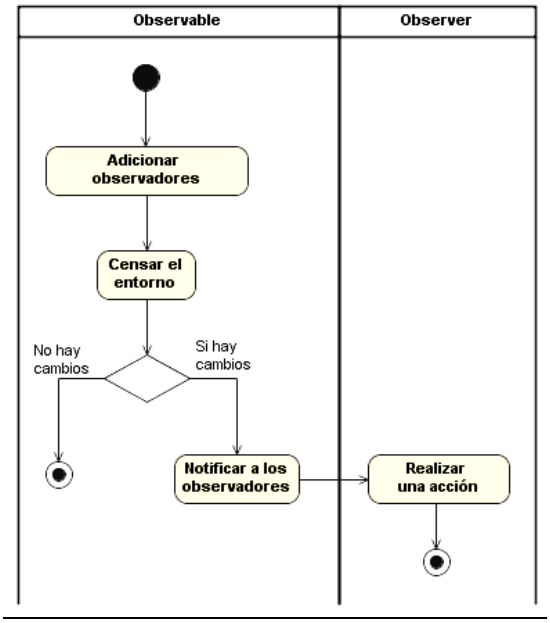

Fig. 2. Diagrama de actividad del patrón *Observer*.

La figura 1 muestra un caso típico del patrón *Observer*. Se trata de un centro de noticias al que están inscritos usuarios con sus preferencias. Al recibir noticias nuevas, el centro distribuye las mismas según la preferencia de los usuarios inscritos.

En la figura 2 se muestra el flujo de trabajo de los componentes en el sistema. Esta figura 2 muestra el ciclo de comportamiento del patrón *Observer*. Se inicia con la adición de observadores, luego la instancia *Observable* monitorea el entorno para ver si han ocurrido cambios. Si existe algo que deba ser notificado a los observadores, envía un mensaje con los datos necesarios para que las instancias de observadores actúen en consecuencia. Este proceso se repite periódicamente en el tiempo.

B. Patrones en la orientación a agentes

En la orientación a agentes se han propuesto patrones de diseño para resolver varios problemas propios de los sistemas multiagente.

Uno de los primeros trabajo es propuesto en [34] y está enfocado en agentes móviles con Aglets[3]. Ese trabajo incluye tres clasificaciones muy orientadas a agentes móviles. Están los patrones de viaje (*traveling*) que están relacionados con el reenvio y enrutamientos de los agentes móviles, patrones de tareas (*task*) para estructurar el trabajo con los agentes y patrones de interacción (*interaction*) para localizar y facilitar las interacciones entre los agentes. Los patrones están desarrollados en Java y enfocados en el diseño. Utilizan diagramas de clases y de interacción para exponerlos y explicarlos.

En ese trabajo se presentan dos aplicaciones basadas en agentes (*File Searcher* y *Enhanced File Searcher*), donde se emplean combinación de patrones. Esas aplicaciones se utilizan para la implementación de agentes móviles que buscan ficheros con cierta cadena en el nombre y que pueden viajar por varios servidores para hacer la búsqueda. En ambos casos se basan en una filosofía reactiva donde los agentes buscan lo que le pide un "*master*" y lo devuelven, pero no conservan ninguna memoria de esa búsqueda. Tampoco se modela una solución a la gestión de cambios en los ficheros almacenados en los servidores sin necesidad de volver a enviar la búsqueda [34].

En [11] y [35] se enfatiza en la necesidad de los patrones de diseño en orientación a agentes, como forma de recolectar y formalizar experiencias para soluciones basadas en este paradigma. En ese trabajo se definen 4 clases de patrones: metapatrones, patrones metafóricos, patrones arquitecturales y antipatrones. Siguiendo esta clasificación se desarrolla una propuesta de 11 patrones. Según Sauvage muchos patrones orientados a agentes son realmente patrones orientados a objetos, ya que no van a aspectos singulares de la orientación a agentes, como la autonomía, las interacciones, entre otras [11]. Además expresa que muchos patrones en la orientación a agentes se enfocan en el diseño obviando la importancia de tener patrones orientados a agentes en varías dimensiones como el análisis o la implementación.

En [36] se presenta un esquema de clasificación bidimensional de los patrones. En su clasificación según el aspecto de diseño (clasificación horizontal), están los estructurales, de comportamiento, sociales y de movilidad. Según el nivel de diseño (clasificación vertical), están los patrones de análisis de roles, patrones de diseño de agentes, patrones de diseño de sistema, patrones de la arquitectura del agente y los patrones de implementación del agente. Un mérito importante de ese trabajo es que su clasificación es amplia y cubre varios niveles de abstracción. Aunque los patrones se presentan en términos de los conceptos de la metodología ROADMAP [37] lo hace de una forma abarcadora y general. Se exponen algunos ejemplos de patrones de agentes y sus clasificaciones. Se enfatiza en que esta clasificación se enfoca más en las nociones del paradigma de agentes, no usando los de orientación a objetos. Entre los campos que sugieren para describir los patrones están: el aspecto de diseño (clasificación horizontal) y el nivel de diseño (clasificación vertical) [36].

Existen otros trabajos, tal es el caso del repositorio de patrones propuesto por el grupo de desarrollo de la metodología PASSI que propone un conjunto de patrones, algunos ejemplos son [33], [38]:

- Patrones multiagente que están relacionados con la colaboración entre dos o más agentes
- Patrones para un solo agente donde se propone una solución para la estructura interna de un agente junto con sus planes de realización de un servicio específico
- Patrones de comportamiento que proponen una solución para agregar una capacidad específica al agente
- Patrones de especificación de acciones que agregan una funcionalidad simple al agente.

Todos estos patrones son para desarrollar un sistema multi-agente más robusto.

Sabatucci en el trabajo [12] se enfoca en patrones de diseño y defiende que un aspecto importante de los patrones no es usarlo solos, sino hacer una combinación de varios. En ese trabajo se hace la formalización de los patrones con un lenguaje que favorece la combinación. Los patrones que propone están integrados a PASSI.

En ninguno de los patrones que se describen en los trabajos anteriormente mencionados sobre patrones para la orientación a agentes se hace énfasis en la proactividad o ambientes a observar, sino en otras propiedades como la cooperación, la comunicación, la estructura organizacional de los agentes u otras. La mayoría de estos patrones se enfocan en el diseño y no en la implementación. En esta dirección, no se conoce de ningún trabajo enfocado en simplificar el trabajo con JADE [39], encapsulando la solución de problemas comunes en la construcción de un SMA.

Particularmente estos dos aspectos (la proactividad, y la simplificación la configuración de JADE) son dos problemas comunes en muchas soluciones basadas en SMA, para las cuales no se conocen que hayan patrones definidos.

V. Patrones de implementación para incluir proactividad

En esta sección se proponen dos patrones de implementación. El patrón *Implementation_JADE* se enfoca en simplificar la configuración de JADE (para crear y manejar agente) y el patrón *Proactive Observer_JADE* se enfoca en la incorporación de proactividad. Estos patrones siguen las recomendaciones de [11] y [36] de que los patrones en la orientación a agentes se enfoquen a las propiedades singulares de los agentes.

Como un patrón de implementación debe estar hecho en un lenguaje de programación específico, se utiliza el lenguaje Java, que es el utilizado por la plataforma JADE. Teniendo en cuenta la consolidación alcanzada por JADE como plataforma

[3] Aglets, http://www.research.ibm.com/trl/aglets

de software libre para el despliegue de un sistema multi-agente y que está basada en el estándar FIPA, se decidió utilizar dicha plataforma para la propuesta de los patrones.

El patrón *Implementation_JADE* respeta la idea de Beck [14] de que los patrones de implementación traten de que los programadores se enfoquen en lo que es realmente singular de cada problema, ya que encapsula parte de la complejidad del trabajo con JADE. Esto se muestra en el ejemplo que se presenta en sección siguiente. El patrón *Implementation_JADE* desarrollado se usa luego en varios lugares y simplificó el trabajo evitando "trabajo mundano y repetitivo" [14] y enfocando el esfuerzo en "problemas realmente únicos" [14].

El patrón *Proactive Observer_JADE* respeta la sugerencia de Sabatucci [12] de enfatizar en la composición de patrones para crear nuevos patrones. Este patrón tiene relación con el patrón *Ecological Recogniser* mencionado en [36]. Ese patrón trata de inferir las intenciones de los agentes y se enfoca en el descubrimiento. En el caso del *Proactive Observer_JADE* las intenciones se conocen y se relacionan con un ambiente que se observa. Esto no está concebido en *Ecological Recogniser* en la forma de estar enfocado en la observación y la decisión cuando la intención es conocida

En la presentación que sigue de ambos patrones se incluyen los campos clasificación horizontal y clasificación vertical sugerida en [36].

A. Descripción de los patrones de implementación en JADE

1. Patrón Implementation_JADE

Este patrón simplifica el uso y configuración de los aspectos principales para el trabajo en la plataforma JADE. A continuación se detallan los elementos esenciales del patrón:

Nombre del patrón: *Implementation_JADE*.

Problema: Específicamente, el patrón que aquí se describe se debe utilizar cuando se quiera implementar las características más comunes y que son de vital importancia para la ejecución del sistema de un sistema multi-agentes mencionadas en la sección III.

Solución: Este patrón utiliza la plataforma JADE para el trabajo con los agentes, sirviendo como intermediario a las funcionalidades que implementa JADE.

El patrón *Implementation_JADE* contiene 7 grupos de operaciones:

1. Inicialización de la plataforma de agentes JADE.
   - 1.1. Configurar algunos parámetros de funcionamiento de la plataforma.
   - 1.2. Crear los contenedores (son necesarios para colocar los agentes dentro).
   - 1.3. Ejecutar los agentes en los contenedores correspondientes.
2. Unión a una plataforma ya existente.
   - 2.1. En ocasiones es necesario tener a los agentes en localidades físicas diferentes, por lo que hay que ejecutar contenedores y agentes en una plataforma que ya existía con anterioridad.
3. Volver a conectar a un agente que ha perdido a la plataforma que lo maneja.
   - 3.1. En el escenario en que un agente esté de forma física en una localidad diferente, puede ser posible que la plataforma colapse por alguna razón y sin embargo, que sea necesario que los agentes sigan trabajando de forma independiente.
   - 3.2. Luego cuando la plataforma vuelva a funcionar, los agentes pueden reincorporarse a su plataforma correspondiente y socializar los resultados que obtuvieron mientras trabajaban solos.
4. Implementar un comportamiento cíclico para poder procesar los mensajes que recibe un agente determinado.
   - 4.1. El desarrollador puede decidir qué tipo de mensaje son aceptados.
   - 4.2. Permite que el desarrollador procese el mensaje como desee.
5. Todo el trabajo relacionado con enviar mensajes hacia los agentes.
   - 5.1. El desarrollador puede enviar mensajes a un agente con una gran variedad de parámetros, que van desde los más básicos a los más complejos.
   - 5.2. La mayoría de las veces solo se necesita enviar un mensaje sencillo a un agente conocido, pero en otras ocasiones, el procesamiento es mayor.
6. Trabajo con el agente AMS (*Agent Management Service*)
   - 6.1. El AMS tiene un registro de los agentes y sus atributos.
   - 6.2. Controla el buen funcionamiento de la plataforma.
   - 6.3. Incluir maneras de interactuar con el AMS para conocer datos sobre los agentes de la plataforma. Ejemplo: saber dónde están los agentes para comunicarse con ellos o el estado de un contenedor determinado.
7. Trabajo con el agente DF (*Directory Facilitator*)
   - 7.1. El control que tiene el DF sobre los agentes es comparado con el de las páginas amarillas.
   - 7.2. Con el DF se puede encontrar un agente que cumpla con un atributo determinado siempre y cuando ese agente se haya registrado con el DF.
   - 7.3. Del mismo modo que con el AMS, aquí están las posibles maneras de comunicarse con el DF para obtener los datos que el desarrollador necesita.

Para lograr estas operaciones mencionadas anteriormente se definieron un grupo de clases con funcionalidades generales.

Init_Platform inicializa la plataforma JADE con las configuraciones que esta permite, como por ejemplo, el puerto de conexión. Se encarga de crear los contenedores en los que se almacenarán los agentes e inicializa los agentes.

Join_Platform crea los contenedores y agentes externos, que son subscritos a una plataforma que ha sido inicializada.

Work_DF contiene todo el trabajo que se realiza en coordinación con el DF (*Directory Facilitator*), que es similar a las páginas amarillas de la guía telefónica. Registra en el directorio del DF el servicio que un agente ofrece, para ofrecer la posibilidad de buscar agentes en esos registros.

Work_ACL engloba el trabajo que se realiza con el uso de los mensajes *ACL* (*Agent Communication Language*). Configura los mensajes con los parámetros que son introducidos como: tipo de mensaje, el contenido del mensaje, la identificación de los agentes involucrados en la comunicación, el objeto que se quiere enviar, etc.

Work_AMS contiene todo el trabajo que se realiza en coordinación con el AMS (Agent Management System). Devuelve la descripción del agente que cumple con la condición que el usuario desee.

Behaviour_Receive_Msg implementa un *Cyclic Behaviour*[4], para obtener y procesar los mensajes ACL que le llegan al agente al que pertenece. Brinda la posibilidad de extender el método "*processMessage*", que se ejecuta cuando se obtiene un mensaje.

El diagrama de clase donde se exponen las principales clases del patrón Implementation_JADE están en la figura 3 junto con las clases del otro patrón de implementación que aquí se propone.

Consecuencias: Este patrón encapsula la capa de abstracción que facilita la configuración del patrón *Proactive Observer_JADE*.

Su objetivo es simplificar el desarrollo de un conjunto de agentes, al tiempo que garantiza el cumplimiento de los estándares a través de un amplio conjunto de servicios del sistema y los agentes con JADE. Permite que los desarrolladores puedan hacer uso de la tecnología de agentes de una forma más sencilla

Clasificación horizontal: Estructural y social.

Clasificación vertical: Implementación del agente.

2. Patrón Proactive Observer_JADE

Este patrón utiliza los principios de patrón de comportamiento *Observer* [5] y los combina con el patrón *Implementation_JADE*.

Nombre del patrón: *Proactive Observer_JADE.*

Problema: Se utiliza el patrón en cualquiera de las siguientes situaciones:

Cuando una abstracción tiene dos aspectos, uno en función de las demás.

Cuando hay un cambio en una entidad, y es necesario cambiar a los demás y no se sabe cuántas entidades más habría que cambiar.

Cuando un entidad debe ser capaz de notificar a otras entidades sin hacer suposiciones acerca de quiénes son estas entidades.

[4] Comportamiento implementado en JADE para efectuar una acción indefinidamente.

Solución: Para implementar el patrón se necesitan de dos entidades: "*Observable*" y "*Observer*".

Entre los métodos más relevantes de la entidad "*Observable*" están los siguientes:

1. Censar el ambiente cada determinado tiempo para detectar algún cambio.
2. Gestionar una lista con los observadores que se subscriban.
3. Notificar a los observadores con los datos encontrados en el cambio ocurrido.

La entidad "*Observer*" tiene que tener métodos como:

1. Implementar el proceso de subscripción a la lista del "*Observable*".
2. Actualizar su estado interno.

Esta permite realizar una acción cuando se le notifique del cambio.

Para lograr una implementación genérica del patrón con agentes, se deben crear fundamentalmente dos entidades: *Observer* y *Observable*. Los agentes que cumplirán con estos roles tendrán la capacidad de comunicarse y actuar autónomamente, pudiendo hasta cambiar su comportamiento dependiendo de las situaciones a las que se enfrenten.

El Agente Observable debe tener una lista interna de los agentes Observer para poder alertarlos de los cambios que detecta. Además debe esperar los mensajes de subscripción de los agentes Observer para poder sumarlos a la lista mencionada y enviar la respuesta de la subscripción. Por último debe tener la capacidad de enviar un mensaje a los Observers con los datos necesarios del cambio encontrado.

El agente Observer tiene que conocer los Observables existentes en el entorno para luego decidir a cual subscribirse. Además debe actualizar su estado cuando le llegue un mensaje con los datos que describen el cambio ocurrido y actuar en consecuencia.

De forma general se necesita que estos agentes se ejecuten en una infraestructura que gestione los mensajes y el ciclo de vida de los agentes.

Como se describió anteriormente en el patrón *Implementation_JADE* se implementaron un grupo de clases para facilitar el trabajo con la plataforma de agentes JADE. Sobre la base de estas clases se realizaron otras que ejecutan las funcionalidades del patrón *Proactive Observer_JADE* para incluir proactividad. De esta forma, los siguientes pasos deben ejecutarse en el momento de comenzar a utilizar el patrón *Proactive Observer_JADE* creado:

1. Iniciar la plataforma a través de la clase de apoyo *Init_Paltform*.
2. Crear los contenedores que contienen a los agentes.
3. Añadir los agentes necesarios a los contenedores.
4. El usuario programador debe implementar las acciones del *Observable* y los *Observers*. En este caso la clase creada para este fin es *Agent_Actions*.

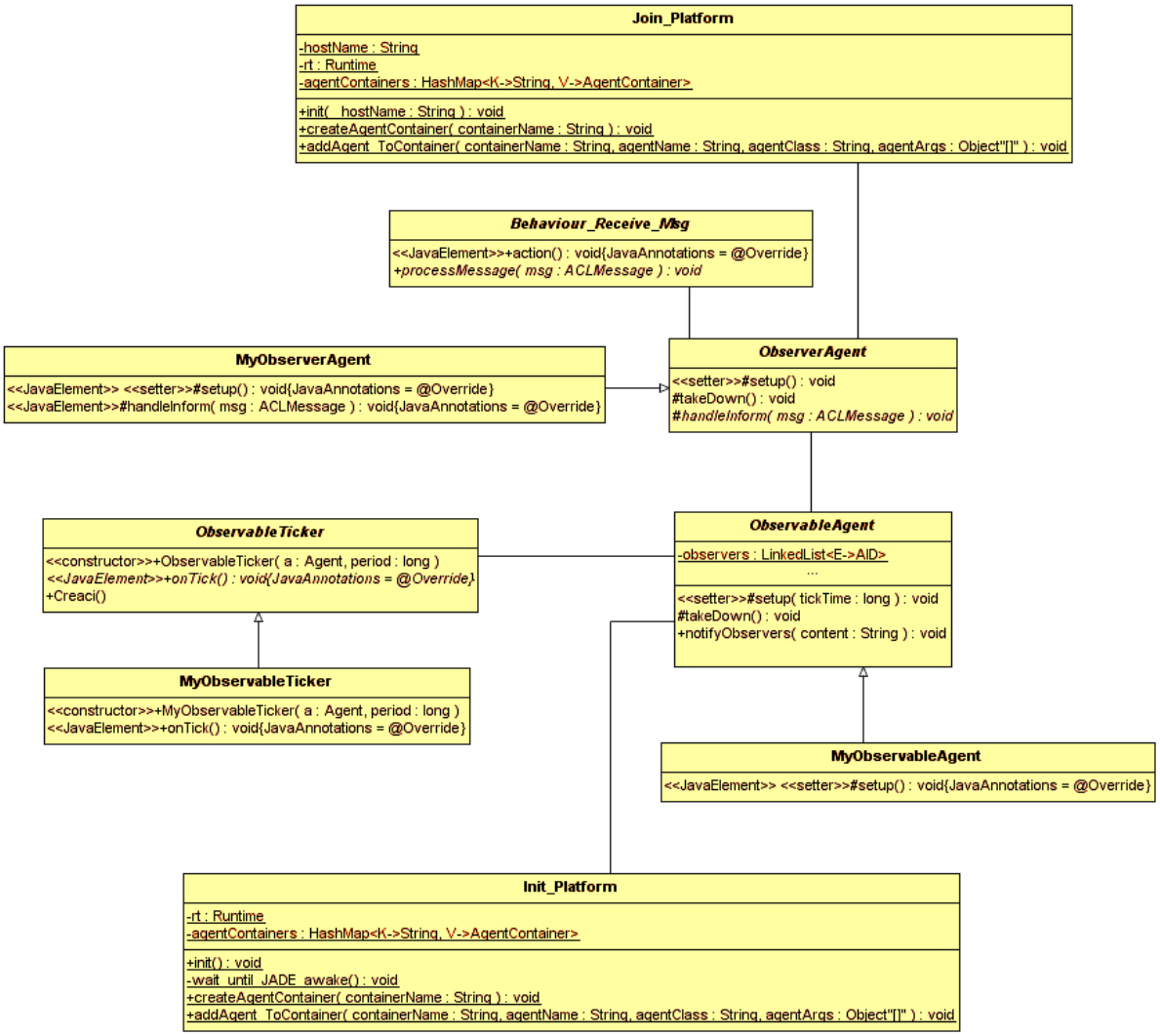

Fig. 3. Diagrama de clases de los patrones *Implementation_JADE* y *Proactive_Observer_JADE*.

Se implementó el patrón *Proactive_Observer_JADE* utilizando agentes, con las clases que proporciona JADE, logrando que el patrón funcione en un ambiente distribuido. A continuación se explican las clases que le dan las funcionalidades al patrón.

Agent_Actions tiene un método *ObserverAction* que se ejecuta cuando al agente *Observer* le llega un mensaje y un método *ObservableAction* que censa el ambiente cada determinado tiempo. La clase está diseñada para que el usuario coloque el código de las acciones que desea realizar en cada caso.

ObservableAgent extiende de *Agent*, escucha los mensajes del tipo *Subscribe* [27] que le llegan de un *Observer*. Cuando un mensaje de este tipo es recibido, se decide si adicionarlo a la lista de *Observers* o no. Si hay algún cambio se notifica a los *Observers* que se encuentren en la lista. La observación del ambiente se realiza usando la implementación del *ObservableTicker*.

ObserverAgent extiende *Agent*, al ejecutarse se subscribe a un Observable y espera el mensaje de respuesta de si fue subscrito o no. Espera por la notificación del *Observable* y realiza alguna acción.

ObservableTicker extiende el funcionamiento de *"TickerBehaviour"* de JADE. Brinda la posibilidad de extender el método *"onTick"*. Este método se encarga de cada cierto tiempo verificar si hubo algún cambio en el entorno.

Las clases que heredan de estas tres últimas, llamadas con el sufijo *My*, son aquellas que el programador debe implementar para que realicen las operaciones que se necesiten. Por ejemplo, como revisar el ambiente y qué hacer cuando al *Observer* le llega la notificación de que *Observable* encontró algo monitoreando el ambiente.

La figura 3 muestra el diagrama de las clases principales del patrón *Implementation_JADE* y del patrón *Proactive_Observer_JADE*. La figura 4 describe el modelo en capas que muestra cómo se relacionan las clases.

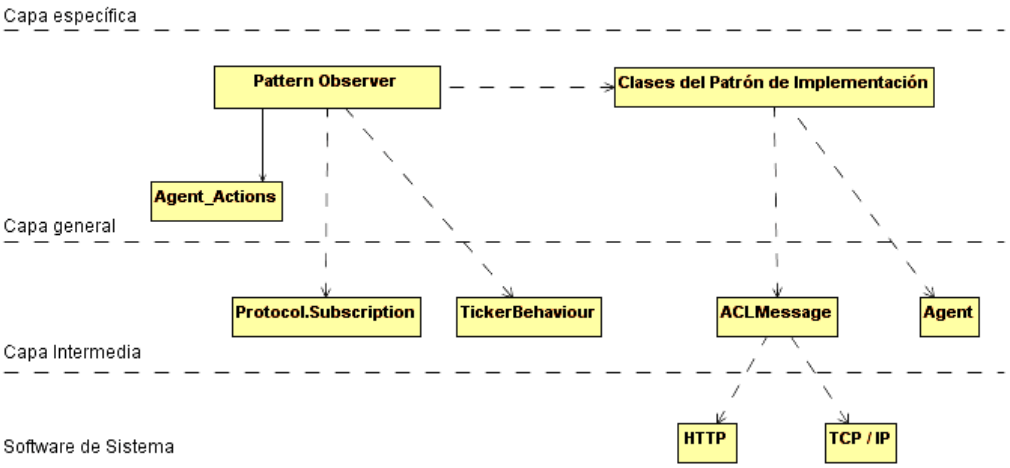

Fig. 4. Diagrama en capas de la relación entre las clases.

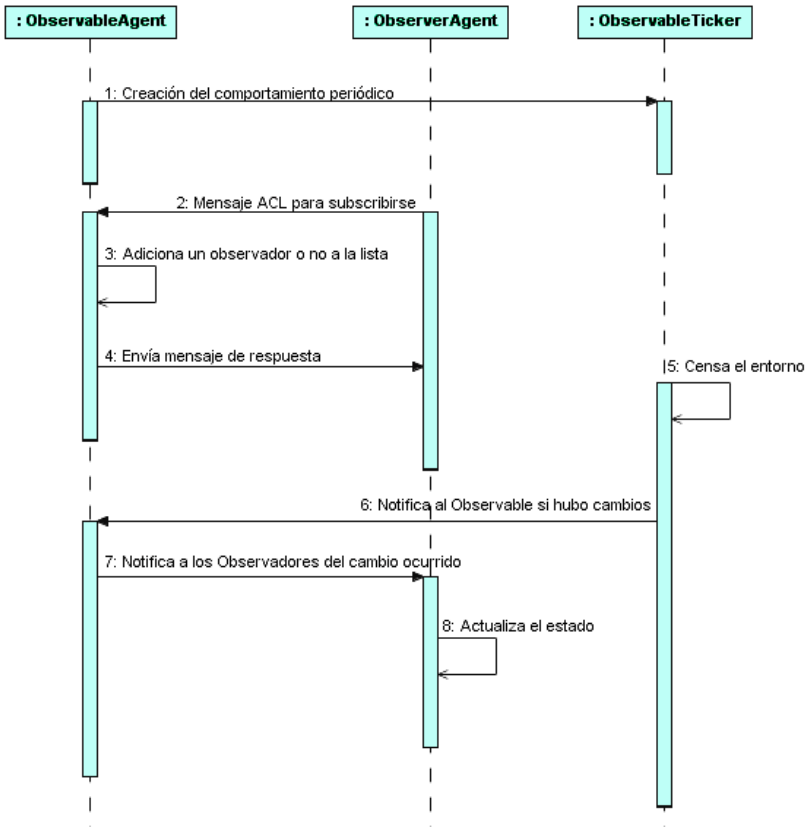

Fig. 5. Diagrama de secuencia del funcionamiento de *Proactive Observer_JADE*

El funcionamiento de las entidades que se ejecutan en el patrón *Proactive_Observer_JADE* puede entenderse mejor en el diagrama de secuencia de la figura 5. Se indican las diferentes llamadas a las funciones de los agentes para que se vea la interacción entre ellos.

Las clases representadas en este diagrama interactúan de manera que *ObservableAgent* llama al constructor de *ObservableTicker*, para que así se pueda censar el ambiente cada cierto tiempo. *ObserverAgent* envía un mensaje a *ObservableAgent* para subscribirse a él y espera la respuesta del mismo.

ObservableTicker llama al método "*onTick*" para censar el entorno y ver si ha ocurrido algún cambio. Si hay cambios entonces notifica a *ObservableAgent* para que envíe un mensaje a los observadores de su lista, los que a su vez actualizan su estado.

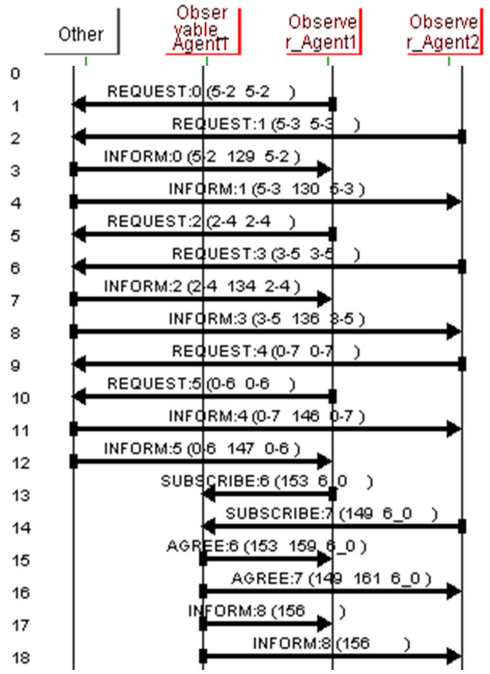

Fig. 6. Vista del despliegue del Proactive_Observer_JADE

Entre las funcionalidades más importantes del *Proactive_Observer_JADE*, que utiliza como base el Observable, es la gestión de los *Observers*, avisándoles de los cambios encontrados. A su vez, los *Observers* deben subscribirse a un *Observable* cuando se inician.

De acuerdo a estas características se implementó un agente *Observable* que al iniciar espera un mensaje de subscripción de los sucesores y ejecuta un comportamiento "*Ticker*" que censa el ambiente para informar de algún cambio ocurrido. Cuando inicia el *Observer*, él conoce quien debe ser su predecesor, y le envía un mensaje de subscripción. El ciclo de ejecución del patrón sigue los pasos explicados anteriormente y se muestran en la figura 6.

Los cuadrados de color rojo representan los agentes que se están ejecutando en la plataforma JADE. Las flechas son los mensajes que son enviados entre ellos en el transcurso del tiempo. Además los mensajes de un mismo color significan que uno es respuesta del otro. Los primeros dos mensajes son parte del funcionamiento de JADE, y como se puede observar las flechas van desde el **ams** al **df** y a otro agente**.** Esto significa que el primero está reportando alguna información interna de JADE. Para un mejor entendimiento, se detallarán solo los mensajes relacionados con el agente **Observer 1**. Pero cabe destacar que los demás *Observer* también realizan las mismas actividades.

En la figura 6 se muestran los mensajes numerados, a continuación se explican algunos de ellos según esa numeración:

3: primer mensaje relacionado con el patrón, donde el **Observer 1** se registra al directorio del **df**. Luego el **df** informa de que el registro fue realizado.

7: respuesta recibida del mensaje 3.

10: enviado desde el **Observer 1** hacia el **ams** para pedir la identificación del agente **Observable** al cuál quiere subscribirse.

14: respuesta recibida del mensaje 10.

19: el **Observer 1** se subscribe a **Observable.**

21: el **Observer 1** recibe la respuesta de la subscripción.

27: **Observable** le informa a **Observer 1** del cambio encontrado.

Consecuencias: A partir del uso del patrón se garantiza que las entidades *Observer* reciban una notificación encontrada por la entidad *Observable* y puedan tomar decisiones o realizar una acción. Con el uso del patrón se logra además el funcionamiento de los agentes con un despliegue distribuido.

Clasificación horizontal: Estructural, de comportamiento y social.

Clasificación vertical: Implementación del agente.

VI. Aplicación de los patrones en un caso de estudio

La proactividad se puede utilizar en cualquier sistema para aumentar sus prestaciones de cara al usuario. Algunos agentes pueden verse como un "objeto astuto" o un "objeto que puede decir no". Viéndolo así un sistema híbrido agente+objetos es completamente viable [3].

Para lograr incorporar comportamientos proactivos en un software orientado a objetos, lograr una hibridación y manejar la proactividad en software orientados a objetos se pueden utilizar los dos patrones de implementación propuestos.

Un caso a tener en cuenta la proactividad es en la construcción de un observatorio tecnológico. Un observatorio es una herramienta para realizar vigilancia tecnológica, que reconoce cambios en el dominio de información que procesa, gestiona y observa, por lo tanto, teniendo en cuenta comportamientos previos, puede avisar con antelación de ciertas variaciones o diferencias en parámetros que evalúa, generando un conocimiento con un alto nivel de importancia al ser actual y novedoso, que puede ser utilizado por los receptores que tengan interés en esa información [40].

Una situación que aún no tiene una respuesta acertada, es que muchos OT operan gracias a las personas que trabajan dándole soporte, buscando, procesando, resumiendo, colocando noticias en los sitios web e informando a los clientes de sus descubrimientos. El desarrollo y buen funcionamiento de un OT enfrenta no sólo el problema relacionado con el número y nivel académico del personal que lo integra [40]. También es necesario que los OT tengan la capacidad de ser proactivos en cuanto a la búsqueda de información, de estar orientados a metas a partir de las necesidades de sus usuarios. Los observatorios deben utilizar un método claro, riguroso y neutro de alerta temprana para sus usuarios [10].

Descripción general del Observatorio Tecnológico: El sistema se divide en una capa cliente y una capa que representa al observatorio como se ve en la figura 7. El sistema tiene agentes personales, estos son una frontera entre

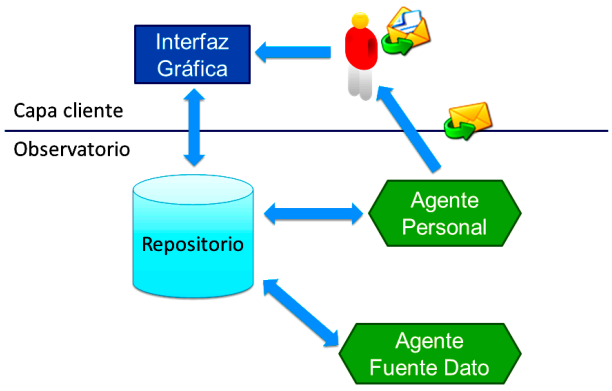

Fig. 7. Vista esquemática del observatorio.

el usuario y el observatorio, son los encargados de representar al usuario en todo momento. El Agente Personal (AP) se dedica a gestionar la información que el usuario necesita y lo hace a partir de sus intereses. También tiene un repositorio, al cual se subscriben los AP, de esta manera, publicando la documentación perteneciente a las línea de investigación de sus usuarios.

Otros pueden acceder a ella cuando sea necesario. Además tiene agentes Fuentes de Datos que están alerta a los pedidos de descarga y búsqueda de los AP. Si alguien necesita una información en específico ésta es pedida a su Agente Personal, que busca en las fuentes de datos disponibles y envía la respuesta al usuario.

Problemas: En este sistema se presentan dos problemas fundamentales que son tratados por los patrones descritos en este documento.

1. El primero está relacionado con la dificultad del trabajo con JADE de forma general, ya sea la gestión de los agentes, los contenedores, la plataforma, envío y recepción de mensajes ACL, etc.

2. Otro problema se encuentra cuando un especialista necesita una información que es relevante para su trabajo y debe esperar a que en la planificación de su Agente Personal, se realice la búsqueda de recursos. Esto conlleva a que el especialista debe esperar un tiempo para recibir resultados de la búsqueda.

Se quiere que de forma periódica el AP haga un reconocimiento del entorno para encontrar información nueva y relevante para su usuario. Primero, debe comunicar con otros AP y si estos no dan una respuesta satisfactoria debe pasar a tramitar sus pedidos con el Agente Fuente de Datos. Cuando se encuentra algo nuevo el AP debe enviar un correo electrónico a su usuario con los resultados.

Soluciones: En la implementación del sistema se utilizó el patrón *Implementation_JADE* y el patrón *Proactive_Observer_JADE*. A continuación se explica cómo fueron usados cada uno.

1. El sistema del Observatorio utiliza las clases de apoyo para ejecutar la Plataforma JADE, brindadas por el patrón Implementation_JADE, para inicializar los contenedores y los agentes, para comunicarse con los agentes del servidor JADE (ams y df), además envía y recibe los mensajes ACL entre agentes. El programador realiza todas estas funciones de forma sencilla y flexible con el mínimo esfuerzo posible.

2. Se implementó un agente **FuentedeDatos_Agent** que extiende las funcionalidades de la clase ObservableAgent, que por las características de este sistema la función que efectúa es gestionar un sitio (repositorio). También se implementaron tres agentes **Personal_Agent** que extienden la clase ObserverAgent, que en este sistema atiende a los especialistas. Ambas clases se vinculan por medio del patrón Proactive_Observer_JADE.

De forma natural el agente personal solicita la búsqueda de documentos a los agentes fuentes de datos del sistema, de forma que los últimos realizan búsquedas en sus correspondientes sitios y devuelven una lista de aquellos documentos que cumplan con las palabras pedidas por los usuarios.

Al utilizar el patrón *Proactive_Observer_JADE* una persona puede desear subscribirse a un sitio, consiguiendo que el agente Fuente de Dato que gestiona ese sitio pueda enviar resultados en el momento en que los encuentra al Agente Personal que atiende a ese usuario. Este último entonces envía un correo con los resultados encontrados antes de tener que realizar una búsqueda que viene condicionada por la planificación en su ciclo.

Los tres Agentes Personales asumen el rol de *Observer* y el Agente Fuente de Datos el de *Observable*. En la figura 8 se muestra como los Agentes Personales se subscriben al Agente Fuente de Datos de la misma forma en que fue explicado el patrón *Proactive_Observer_JADE* anteriormente. En los mensajes del 31–33 están los envíos de informaciones encontradas para estos tres especialistas, en el momento en que se encuentran los documentos para ellos.

En la figura 9 se muestra un ejemplo del correo enviado por el Agente Personal al especialista.

Todo la información relevante a un usurario se obtiene de forma proactiva, con solo decir sus intereses. El agente personal que representa al usuario con sus intereses, utilizando el patrón *Proactive_Observer_JADE*, se mantiene buscando cada cambio, en los sitios que se escogen. Cuando hay un cambio, el usuario recibe un correo con lo nuevo encontrado en los sitios o lo que ha socializado otro agente personal.

VII. Conclusiones

En este trabajo se propuso dos patrones de implementación utilizando como base el patrón de diseño de la orientación a objeto *Observer* y siguiendo la filosofía de agentes. Para desarrollar el patrón *Implemetation_JADE* se tomó como base la plataforma de desarrollo de agentes JADE y en el mismo se

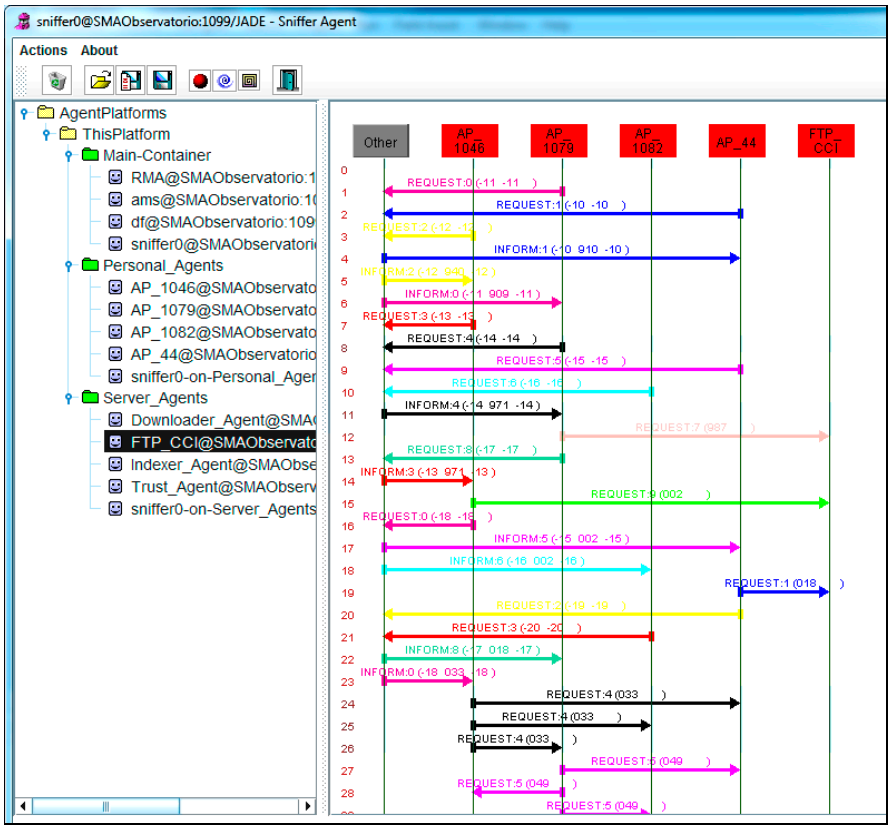

Fig. 8. La pantalla de ciclo de vida de los patrones Implementation_JADE y el patrón Proactive_Observer_JADE en el Observatorio

Las palabras buscadas fueron: web service, team foundation server
El Observatorio ha encontrado 52 recursos y 0 URL que pueden ser de su interés.

A continuación se muestra la cantidad de recursos encontrados de cada fuente.

FTP_CCI	49
FTP_TELECO	3

Recursos organizados por orden de relevancia.

Lista de recursos encontrados:
Nombre: Professional Team Foundation Server 2010.pdf
Relevance: 93.04
Term - Frequency:
web service - 19
team foundation server - 2120
Page count: 722
Author: Ed Blankenship, Martin Woodward, Grant Holliday & Brian Keller
Creation date: 2011/03/15 08:52:55
Modification date: 2011/05/18 22:57:22
Title: Professional Team Foundation Server 2010

Fig. 9.Correo electrónico de resultado de la ejecución de los patrones *Implementation_JADE* y el patrón *Proactive_Observer_JADE* en el Observatorio

da una capa de abstracción para el trabajo con las funcionalidades de agente de una forma sencilla. Este patrón sirvió como núcleo para el patrón *Proactive_Observer_JADE* permite incluir entidades que a partir de una meta y cambios en el ambiente que revisan se realice una acción proactiva.

Ambos patrones se utilizaron en un caso de estudio relacionado con problemas en un observatorio tecnológico. Al aplicar los patrones en el caso de estudio se pudo comprobar que se pudo agregar de forma satisfactoria un comportamiento proactivo beneficioso para el usuario. La inclusión de características proactivas en un Observatorio Tecnológico mejora el rendimiento del mismo, ya que el sistema es capaz de adelantarse a las solicitudes de información de los usuarios. Los patrones propuestos presentan una alta reutilización para los programadores que deseen utilizarlos, debido a la facilidad del lenguaje Java con el que fueron desarrollados. Con los mismos se puede incorporar proactividad en un sistema y manejar de una forma sencilla los agentes.

Referencias

[1] M. Wooldridge, "An Introduction to MultiAgent Systems," 2nd ed. John Wiley & Sons, 2009.

[2] N.R. Jennings. (2000). "On agent-based software engineering," *Artificial Intelligence*, 117(2), pp. 277-296.

[3] B. Henderson-Sellers and P. Giorgini, "Agent-Oriented Methodologies," 1st ed. Hershey: Idea Group Inc, 2005.

[4] I. Jacobson, G. Booch and J. Rumbaugh, "The Unified Software Development Process," reprint ed. Prentice Hall, 2012.

[5] E. Gamma, Design Patterns: "Elements of Reusable Object-oriented Software," ed. Pearson Education, 2004.

[6] A. Shalloway and J.J. Trott, "Design Patterns Explained: A New Perspective on Object-Oriented Design," ed. Addison-Wesley, 2002.

[7] T. Budd, "An introduction to object-oriented programming," 3rd ed. Addison-Wesley, 2002.

[8] C. Ruey Shun and C. Duen Kai. (2008). "Apply ontology and agent technology to construct virtual observatory," *Expert Systems with Applications*, 34(3), pp. 2019–2028.

[9] A. Adla. (2006). "A Cooperative Intelligent Decision Support System for Contingency Management," *Journal of Computer Science*, 2(10).

[10] L. Rey Vázquez. (2009). "Informe APEI sobre vigilancia tecnológica", Asociación Profesional de Especialistas en Información. [Online]. Available: http://eprints.rclis.org/17578.

[11] S. Sauvage, "Agent Oriented Design Patterns: A Case Study," in *Proc. of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, Vol. 3, 2004, pp. 1496–1497.

[12] L. Sabatucci, M. Cossentino and S. Gaglio, "A Semantic Description For Agent Design Patterns," in *Proceedings of the Sixth International Workshop "From Agent Theory to Agent Implementation" (AT2AI-6) at The Seventh International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2008)*, 2008, pp. May 13.

[13] R.S. Pressman, "Software engineering: a practitioner's approach", 7th ed. McGraw-Hill Higher Education, 2010.

[14] K. Beck, "Implementation patterns," 1st ed. Addison-Wesley, 2008.

[15] M. Fowler, "UML distilled", 3rd ed. Addison-Wesley, 2004.

[16] J. Rumbaugh, I. Jacobson and G. Booch, "The Unified Modeling Language Reference Manual," 2nd reprint ed. Addison-Wesley, 2010.

[17] N.R. Jennings, "An agent-based approach for building complex software systems," *Comm. of the ACM*, 44(4), 2001, pp. 35–41.

[18] FIPA, "FIPA Agent Management Specification," Foundation for Intelligent Physical Agents, 2003. [Online]. Available: http://www.fipa.org/specs/fipa00023/XC00023H.html.

[19] F. Zambonelli and A. Omicini, "Challenges and Research Directions in Agent-Oriented Software Engineering," *Autonomous Agents and Multi-Agent Systems*, 9(3), 2004, pp. 253–283.

[20] F. Dignum *et al*., "Open Agent Systems," in *Agent-Oriented Software Engineering VIII*, Springer Berlin Heidelberg, 2008, pp. 73–87.

[21] S. Franklin and A. Graesser, "Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents," in *Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, 1997, pp. 21–35.

[22] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," 3rd, illustrated ed. Prentice Hall, 2010.

[23] B. Henderson-Sellers, "From Object-Oriented to Agent-Oriented Software Engineering Methodologies," in *Software Engineering for Multi-Agent Systems III*, Springer Berlin Heidelberg, 2005, pp. 1–18.

[24] S.A. O'Malley and S.A. DeLoach, "Determining When to Use an Agent-Oriented Software Engineering Paradigm," in *Agent-Oriented Software Engineering II*, Springer, 2002, pp. 188–205.

[25] E. German and L. Sheremetov, "An Agent Framework for Processing FIPA-ACL Messages Based on Interaction Models," in *Agent-Oriented Software Engineering VII*, Springer, 2008, pp. 88–102.

[26] FIPA, "FIPA Communicative Act Library Specification," Foundation for Intelligent Physical Agents, 2003. [Online]. Available: http://www.fipa.org/specs/fipa00037/SC00037J.html.

[27] FIPA, FIPA ACL Message Structure Specification. Foundation for Intelligent Physical Agents, 2003. [Online]. Available: http://www.fipa.org/specs/fipa00061/SC00061G.html.

[28] F. Bellifemine *et al*., "Jade—A Java Agent Development Framework," in *Multi-Agent Programming*, Springer US, 2005, pp. 125–147.

[29] F.L. Bellifemine, G. Caire and D. Greenwood, "Developing Multi-Agent Systems with JADE," ed. Wiley, 2007.

[30] P. Evitts, "A UML pattern language," ed. Macmillan Technical Publishing, 2000.

[31] C. Alexander, S. Ishikawa and M. Silverstein, "A Pattern Language: Towns, Buildings, Construction," 21th ed. New York: Oxford University Press, 1977.

[32] J. Gil and I. Maman, "Implementation Patterns. Department of Computer Science Technion-Israel Institute of Technology", 2004. [Online]. Available: http://www.cs.technion.ac.il/~imaman/stuff/ip-ecoop05.pdf.

[33] M. Cossentino, L. Sabatucci and A. Chella, "Patterns Reuse in the PASSI Methodology," in *Engineering Societies in the Agents World IV*, Springer Berlin Heidelberg, 2004, pp. 294–310.

[34] Y. Aridor and D.B. Lange, "Agent design patterns: elements of agent application design," in *Proceedings of the Second international conference on Autonomous agents*, 1998, pp. 108–115.

[35] S. Sauvage, "Design Patterns for Multiagent Systems Design," in *MICAI 2004: Advances in Artificial Intelligence*, Springer Berlin Heidelberg, 2004, pp. 352–361.

[36] A. Oluyomi, S. Karunasekera and L. Sterling, "An Agent Design Pattern Classification Scheme: Capturing the Notions of Agency in Agent Design Patterns," in *Proceedings of the 1th Asia-Pacific Software Engineering Conference*, 2004, pp. 456–463.

[37] F. Bergenti, M.-P. Gleizes and F. Zambonelli, Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook, ed. Springer, 2004.

[38] A. Chella, M. Cossentino and L. Sabatucci. "Tools and patterns in designing multi-agent systems with PASSI," *WSEAS Transactions on Communications*, 3(1), 2004, pp. 352-358.

[39] F. Bellifemine *et al*., "JADE-A Java Agent Development Framework," in *Multi-Agent Programming Languages, Platforms and Applications*, Springer, 2005, pp. 125–147.

[40] I. de la Vega, "Tipología de Observatorios de Ciencia y Tecnología," Los casos de América Latina y Europa. *Revista Española De Documentación Científica*, 2007, 30(4), pp. 545–552.