An OO-based Design Model of Software Agent

Jianxing LI
School of Computer, National University of Defense Technology, ChangSha, 410073 P. R. China
jxli@nudt.edu.cn

Xinjun MAO
School of Computer, National University of Defense Technology, ChangSha, 410073 P. R. China
xjmao21@21cn.com

Yao SHU
School of Computer, National University of Defense Technology, ChangSha, 410073 P. R. China
yaoshu_76@163.com

Abstract

How to implement software agent is a key problem for developing agent-oriented programming languages and tools. Aiming to solve this problem based on the OO technology, this paper first discusses differences between agent and object, and then puts forward an implemental architecture of software agent based on the OO technology and a simplified improvement of the BDI agent model. An OO-based design framework of software agent is also proposed by using the POAD method finally. These research results are helpful to clarify how to suitably extend the OO method for solving the key problem

1. Introduction

Along with rapid evolution of the Internet and distributed computing technologies, the requirements of many current application software systems have presented complicated intentional features such as autonomy, flexible scalability, proactive adaptability, open and growing etc. These features have challenged the mainstream OO method; thereby new software development methods and tools are demanded. The Agent-Oriented (viz. AO) method emerges, as the times require**.** Because it provides the agent concept which is more abstract than the object concept and can naturally be used to model the complicated intentional behavior entities, and the mode of coordination for modeling complicated interaction in the applications, the academic community and the industrial circles have paid a lot of attention to its researches and applications.

In recent years, some related researchers have certainly got some notable achievements, but the status of related applications is not as well as people expect since most current applications have been developed still by the OO method. The main reason for this is that the AO programming languages and tools that support the implementation of the related applications are not mature as the OO technologies although some representatives [see: 1,2,3,4] of the AO software development methodologies that can preferably support the analysis and design of the related applications have been put forward.

The maturation of the AO programming languages and tools should still take a longer time by analogy to the evolution history of the mature OO technology. Even at present, people have not reached an agreement about the solution of the problem "***How should the software agent be implemented?***" As this problem is a fundamental to the main reason above, we will take it as a key problem of this paper and discuss how to solve it based on the OO technologies**.**

2. Related works

At present, the researches of the AO method focus on the multi-agents system. Most of works related to the key problem are mainly in the earlier researches on the agent architecture, for example: the Bratman [5] and Rao's [6] BDI architecture agent, Brooks's [7] Subsumption architecture reactive agent and InteRRaP [8] architecture agent. Because the focus of these works is on the agent architecture, they have not made further research on AO programming languages and tools although some of them have done some implementations of the agent architecture. The Shoham's Agent0 [9] is a representative that has broad influence in aspect of the AO programming language, but the Agent0 is still a language prototype.

On the other hand, people now have some commercial products of AO software development tools (such as AgentBuilder, Aglet and JACK etc.) that have broad goodish influence in many applications. Considering the commerciality, the owners of these products have not published their method about how to solve the key problem. There are few reviews on their method, so people do not know how well they can solve the key problem.

Therefore, the key problem is still worth conducting researches. Based on the above works from the agent

Proceedings of the Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'05)
0-7695-2405-2/05 $20.00 © 2005 IEEE

architectures to the commercial products, we put forward a feasible solution to the key problem. The key of this solution is that what about extension of the OO method is suitable to implement the agent, so the notion of the agent and the differences between object and agent should be made clear firstly.

3. Agent vs. Object

Because researchers and users of agent come from different domains, they may have various emphases on the various features of agent concept. There is still no universally accepted definition of agent concept heretofore. The so-called weak definition of agent concept presented here is adapted from [10]: an agent is a ***computer system*** that is situated in some environment, and that is capable of ***flexible*** (***reactive, proactive and social*** [10]) ***autonomy*** action in the environment in order to meet its design objectives. As for the so-called strong definition of agent concept, it defines the agent is a behavior entity that is capable of learning and adapting etc. besides the flexible autonomy capability in the weak definition. In this paper, we will adopt the weak definition, furthermore we designate that the computer system is a software system, and call it ***software agent***. It should be noticed that this is only a descriptive definition of software agent, and that how to implement it based on the OO method is just the key problem of this paper.

The relationship between agent and object has not been widely discussed in literature [10]. First, we notice that agent and object have different origins although both of them can be used as the basic model construct that is an abstraction of the behavior entities in problem domains. Agent concept originates from the problem domain, but object concept originates from the computational solution domain. Therefore, it is very clear that object in implemental lateral is an encapsulation of the data and method in spite of what abstraction of object in modeling. For agent, the same thing is not very clear and needs further researches. Next, we can compare agent with object in three aspects about their concept, model standpoints and basic techniques as follows:

(1) Object can really exhibit some dynamic behavior features by the combination of its attributes and services, but the exhibitions of these dynamic features are passive and must be invoked by other objects. Therefore, object cannot exhibit flexible autonomy features as agent does. This is the fundamental difference between them.

(2) The OO method has not considered the cause of the dynamic behavior of an entity in its basic model construct: object. It considers that is an affair in higher level than the object. On the other hand, the AO method has considered not only the cause but also the interactive mechanism among the entities and the environment in its basic model construct: agent. Therefore, the AO method can be closer to the new cases in the distributed computing environment than the OO method does, and it can deal with intentional feature of requirement of the new cases.

(3) The basic techniques of OO method (such as encapsulation, class and the inheritance) are not only reserved but also evolving in the AO method. A kind of suitable encapsulation of the mental state and activity is the base of modeling the flexible autonomy behavior. It is more complex to encapsulate the mental attributes and the mental activities in agent than to encapsulate the attributes and methods in object.

Therefore, agent and object really have the essential difference: object cannot present flexible autonomy features by itself as agent does. However, this difference cannot stop us implementing software agent based on OO method. Furthermore, according to the above comparisons, we can find such a possibility of to do this. In following section 4 and section 5, we will discuss the details of the kernel techniques about how to realize this possibility.

4. The implemental architecture of software agent

After the comparisons in section 3, we have almost completed the requirement analysis of software agent as an OO software system. The next problem we will discuss is how to design the ***implemental architecture*** of software agent (Here, we mean that the implemental architecture should satisfy the specific requirement that is to develop such programming languages and tools that can be used to ***implement*** software agent). Because an intensive understanding to the agent concept is directly correlative with general agent architectures, people have developed many agent architectures in the early research period, for example those agent architectures mentioned in section 2. We will start off with the notable BDI architecture.

4.1 The BDI agent model

The BDI agent model originated from Bratman [5] is composed of seven main components:

(1) A set of beliefs, representing information the agent has;

(2) A belief revision function (brf), perceiving input and determining a new set of beliefs;

(3) A desire generation function (options), determining the options available to the agent;

(4) A set of desires, representing possible courses of actions available to the agent;

(5) A filter function (filter), representing deliberation process and determining intentions;

(6) A set of intentions, representing the agent's focus;

(7) An action selection function (execute), determining an action to perform based on intentions.

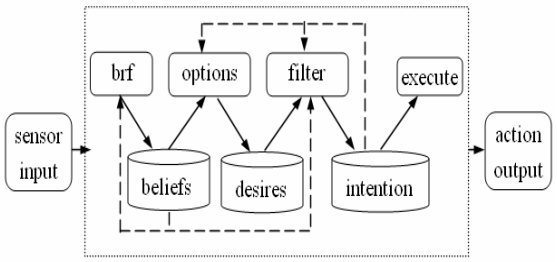

Figure 1: The BDI agent model

The relations of these components are very complicated, but they can roundly reflect the practical reasoning process that stimulates human reasoning (see Figure 1). In [6], Rao and Georgeff further gave the BDI logic for the formalization of the BDI agent model and discussed the importance of the intentions in logic. Especially, they suggested that: in concrete application, some parts of the describing capability of the BDI agent model should be abandoned and other parts of it should be simply improved based on the concrete requirements. This simplified improvement approach will be adopted for the implemental architecture of software agent in following section 4.2 and section 4.3.

4.2 The implemental architecture of software agent

In (2) and (3) of section 3, we have implied that the key for implementing software agent based on the OO method is a kind of suitable encapsulation of the mental states and activities in OO software system level not in object level so that the software agent implemented as an OO software system can really exhibit flexible autonomy features. According to this concrete requirement of software agent and the suggestions of Rao and Georgeff [6], we improve the BDI agent model for getting the implemental architecture of software agent (see Figure 2).

Contrasting Figure 2 with Figure 1, we can summarize the main improvements in Figure 2 as follows:

(1) About the cause of the mental activities, the implemental architecture of software agent in Figure 2 is simply driven by the event that is related with the agent's goal. But in Figure 1, it is more complicated: The perception to the agent's environment and the comparison with the agent's beliefs are the causes of the agent's mental activities that will drive the agent determining a new set of beliefs and the options available to the agent etc; especially, maybe it still arouses higher-level mental activities such as maintenance and modification of the mental state. Therefore, Figure 1 is an imitation of the human mental activity, but it also brings the complexity. In the case of software agent, we can assume that the computational resources and capabilities of software agent are limited, and the agent's assigned goals should not be very complex. Therefore, software agent need only perceive the events related with the agent's goal, and it need not have higher-level mental activities.

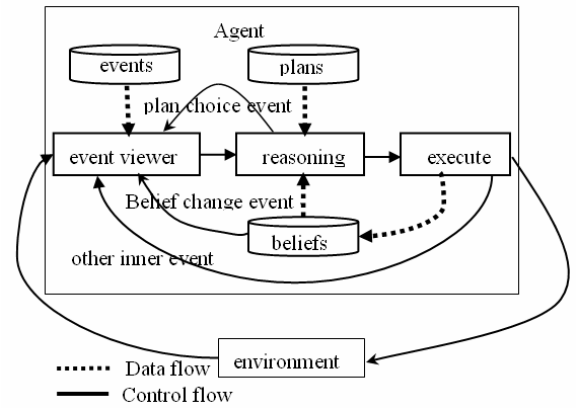

Figure 2: The implemental architecture of software agent

(2) Based on the simplification of the cause of the mental activities in (1), we simplify three mental attribute sets in Figure 1 (see Table 1). In the second row of Table 1, we have replaced the two mental attribute sets (desires and Intentions) with the two concrete attribute sets (events and plans). In addition, the structure of beliefs in the second row is unchangeable, and the case in the first row can be different.

Table 1 Representations of BDI concept in 2 levels

Philosophy level	beliefs	desires	intentions
Implement level	beliefs	events	Plans

Besides the simplification of the cause and the mental attribute sets, in Figure 2 the mental activity has been simplified as a practical reasoning process based on the event driven. The details of this process will be discussed in section 4.3. But this simplification will make software agent unable to learn and adapt.

Note that the main advantage of the simplification above is the computability of the mental activity in Figure 2 so that the implemental architecture of software agent can be implemented based on the OO method. But in Figure 1, the BDI agent model without further implemental simplified improvement has not the computability of the practical reasoning process

although it can be completeness in the practical reasoning logic [6].

4.3 The decision algorithm for the autonomous behavior of software agent

The practical reasoning process based on the event driven also called the decision algorithm for the autonomous behavior of software agent is illustrated in Figure 3.

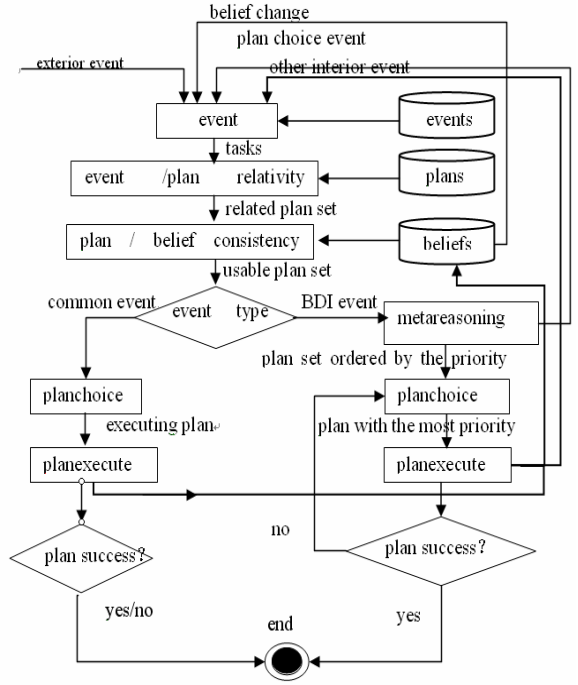

Figure 3: The reasoning process of software agent

In Figure 3, when the software agent perceives some event by its event viewer, it will first look into its events, only those events registered would be handled. Its events includes the exterior events related to the agent's goals and some interior events such as belief change event, plan choice event which is likely to happen in the practical reasoning process and the clock event. When the software agent confirms to handle some event, it will initialize a task to process the event. The software agent can initialize many such tasks. In each task, the software agent will carry out the "event / plan relativity examination" for getting a related plan set based on its plans, and then carry out the "plan / belief consistency checking" for getting a usable plan set from the related plan set based on its beliefs.

The software agent will adopt different handling strategies for different types of events. For the common event, it will randomly choose one from the usable plan set to handle of the event. Even if the executing of the plan is failure, the software agent will no longer choose any other plans and conclude the handling of the common event. Because the common event is usually transient, the precondition that arouses the software agent to handle the event may not exist when the chosen plan fails to handle the events, and the software agent need not handle the event continuously. This kind of handling strategy for the common event makes the software agent exhibiting ***reactive*** behavior feature.

The handling strategy for the BDI event is different from that for the common event. Because every BDI event corresponds to some agent's committed goal or agent's intention that the agent should insist on, the handling strategy for the BDI event should be goal-directed process. That means needing more rational mechanism for the plan choice and the plan failure disposal. First, we adopt meta-level reasoning technique that is a kind of technique of "reasoning about reasoning". It means that the software agent can have different plan choice mechanisms for different BDI events and can also bring about the further plan choice events so that the software agent can do further meta-level reasoning when the usable plan set has many possible entries. After the meta-level reasoning, the software agent can gain a usable plan set ordered by the priority level, and then it will choose the plan with the most priority level to handle the event. Second, we adopt such mechanism for the plan failure disposal. If the execution of some plan fails to handle the BDI event, the software agent will choose other plans which have lower priority level to continuously handle the BDI event until it runs out of all plan in the usable plan set and then just abandons the handling of the BDI event. This mechanism for the plan failure disposal embodies the agent's insisting on the committed goal and can remedy the fault of some simple reasoning mode. As a result, the both handling strategies above for the BDI event make the software agent exhibiting ***proactive*** behavior feature. It should be noticed that: the BDI event is mainly aroused by the reasoning modes in the plan related to agent's committed goal, but the modeling of these reasoning modes can not be completed only by the OO method and need to do some AO extension of the OO method.

In addition, whether in the handling of the common event or the BDI event, the plan executing can generate some interior event or sent some message to other agents or change some belief (not change the structure of the beliefs). The change of the beliefs will generate the belief change event.

5. Design framework of software agent

After having designed the implemental architecture of software agent, we will complete the detailed design (that is a kind of ***design framework*** [11] composed of

some design classes) of software agent by using the ***POAD*** (Pattern-Oriented Analysis and Design) ***method*** [11] in this section.

The POAD method is a kind of OO software system development method based on the reusable design pattern. The design framework developed by the POAD method is a set of some design classes that are common in some application domain, and it provides the basic mechanisms for customizing and extending the design framework for the concrete application design. Because the key of this paper is not to implement some concrete software agent but how to suitably extend the OO method so that we can develop AO programming languages and tools based on the method mentioned, it is suitable to use the POAD method for developing the design framework of software agent since the design framework will clarify the key of this paper.

The POAD method includes three phases that are the system analysis and design pattern choice, the high-level design and the refinement design.

5.1 Design pattern choice of software agent

Substantially, the discussions in section 4 are also the system analysis of the POAD method. Figures 2 and 3 have completed the architecture design of software agent. For the detailed design, the next thing the POAD method will do is the design pattern choice of software agent.

Based on searching in the pattern library [11] and matching the functional modules in Figures 2 and 3, we make out following choices: First, the Singleton pattern [11] is chosen to design the governor of the software agent because it has only a single instance class and it is suitable to satisfy the requirement (that is the software agent only has one governor itself and its instance must be created by itself and be known to other objects) of the governor. Second, the Mediator pattern [11] is chosen to design the event viewer in Figure 3. Third, we choose the Abstract Factory pattern [11] for the design of the functional module “usable plan set" in Figure 3. Finally, we choose the Strategy pattern [11] for the design of the last functional module "the software agent adopts different handling strategy to the different event type" in Figure 3.

5.2 Pattern-level design model of software agent

In the high-level design phase, the POAD method will combine all of the design patterns chosen in section 5.1 and present the pattern-level design model of software agent. That mainly includes the instantiation of the all design patterns and to ascertain the relationship among the all design pattern instances by using interface.

The so-called instantiation of a design pattern is to name each common design pattern chosen in section 5.1 an instance-name after the concrete need of the application. For the software agent, we make out the instantiation as follows: AgentManager for the Singleton pattern; EventObserver for the Mediator pattern; ApplicabePlansProducer for the Abstract Factory pattern and EventHandlingStrategies for the Strategy pattern.

As for the relationship among the all design pattern instances, Figures 2 and 3 have demonstrated it clearly in concept, but the POAD method still need to ascertain it by using interface so that we can combine the all design pattern instances by using interfaces to get the pattern-level design model of software agent. In the POAD method, an interface of a design pattern may be an interface method or an interface class. In [11], we can find the interfaces of the common design patterns: The Singleton pattern has a interface method getInstance() that can be called by any other object to know who is the "governor"; the Mediator pattern has two interfaces, and one is the interface class Colleague that can be used by event source to report the even happening to the EventObserver and the another one is the interface class Mediator that can be used by EventObserver to inform other object to handle the event; the AbstractFactory pattern has two interfaces, and one is the interface class AbstractFactory that can be used to produce different type products and the another one is the interface class AbstractProduct that can provide interface for the above products; the Strategy pattern also has two interfaces, and one is the interface class Strategy and the another one is the interface class Context. At last, we can conclude above discussions to get the pattern-level design model of software agent as Figure 4 shows.

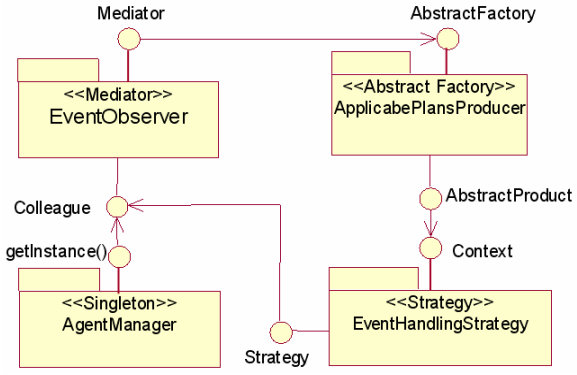

Figure 4 Pattern-level design model of software agent

5.3 Design framework of software agent

In the refinement design phase, the POAD method will complete the instantiation of the inner of the all design pattern instances and the design optimization such as deleting and combining etc. so that we can get the design framework of software agent at last.

First, for the instantiation of the inner of the design pattern instance AgentManager, we have done as follows. Name its unique class as Agent. According to the concrete requirements of the governor in sections 5.1 and 5.2, the constructor Agent() of the class Agent should be private and we still need to declare a private static variable AgentXXX: Agent = the new Agent(). And declare a public method getAgentName() for the instantiation of the interface method getInstance() of the AgentManager. Furthermore, declare private variable myBelief, myEvents[m], myPlans[m][n] to express every software agent's beliefs and events and plans respectively.

Second, for the EventObserver, we have done as follows. Because every software agent has only event viewer that can be played by the agent, we merge the EventObserver and the AgentManager, and the concrete mediator class ConcreMediator in the EventObserver will be merged with the class Agent in the AgentManager still named Agent. Thus, the Agent is not only the governor itself but also the event viewer and the concrete mediator of the software agent. The public method ColleagueChanged() in the class ConcreMediator will be a method of the Agent after the merging but named pleaseHandle(EventType, EventName) that will be used to send other objects the event handling inform. As for the class Colleague in the EventObserver, because some functions of the Colleague can be performed by the getAgentName() method and the software agent also no longer relate with the event source directly, the class Colleague will be deleted.

Third, for the ApplicabePlansProducer, the details are as follows. Because the software agent only needs a kind of product that is "usable plan set", we name the only abstract product class Plan and only remain one of the producing methods that is named productApplicablePlans (EventType, EventName) method, and it will be the interface of the ApplicabePlansProducer. When the software agent confirms to handle some event and informs some object to do that by its pleaseHandle() method, it will initialize a new thread and generate a concrete Factory class that will be named EventName+Factoty and then call its interface method productApplicablePlans() in the thread to produce the "usable plan set". In the executing of the method productApplicablePlans(), it will call the both static methods isRelevant (EventName) and isApplicable(EventName) of the plans, only those plans that its return value are all true are just the usable plans. Thus, the "usable plan set" can be produced and presented in the AppPlans[]. In addition, there is a common method body() in the abstract product class Plan so that programmer can define concrete plan executing steps in it.

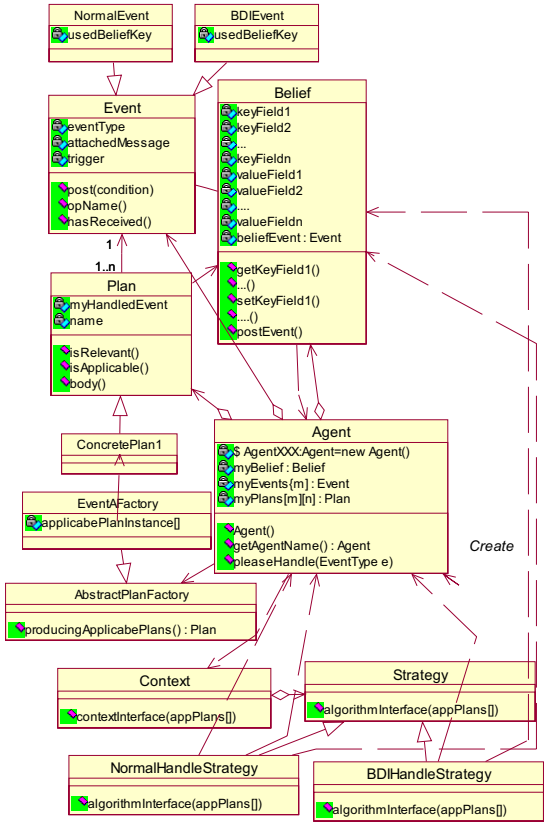

Fig 5: Design framework of software agent

Finally, for the EventHandlingStrategy, its interface class Context holds a reference to another interface class Strategy and has a public method contextInterface(). When the method productApplicablePlans() has chosen the "usable plan set" AppPlans[], it will call the public method contextInterface() and pass in the AppPlans[]. According the illustration in Figure 3, the interface class Strategy has two concrete strategy interface classes that are the NormalHandleStrategy and the BDIHandleStrategy. But it should be noticed that: Which one of two concrete strategies will be used is determined not by the EventHandlingStrategy but by the productApplicablePlans() when it indirectly refers

interface class Strategy through interface class Context according the event type.

After discussing in above four paragraphs, we have completed the reuse of the outstanding common design patterns of the POAD method for software agent and the combination of the all reuse design pattern instances by using interfaces of design classes. For getting the ultimate design framework of software agent, we still need take of the crust of the pattern and do some remedies and optimizations and simplifications as follows. In above designs, the class Agent refers beliefs and events in Figure 2 through myBelief and myEvents[m], so we need add the definitions of the class Belief and the class Event. There are some keyword fields and correspondent belief-value fields in the class Belief. These belief-value fields can be read and written and the changes of the belief-value fields can generate belief change event. As for the class Event, there are the event type and the trigger condition of the event etc. in it. In addition, we only choose the EventAFactory and ConcretePlan1 as the representatives of the concrete Factory class and the concrete Plan class for simplification. At last, we conclude above discussions and get the ultimate design framework of software agent as Figure 5 shows.

From Figure 5, we can see that: The design framework of software agent is still more complicated. The relationship among the design classes in Figure 5, especially among the Belief and the Event and the Plan, is a very coherently complicated relation by class Agent. Furthermore, the reasoning process in Figure 5 will deepen this relationship.

6. Conclusion

Having discussed the implemental architecture and the design framework of software agent, we conclude that software agent cannot be implemented as an instance of some standard OO class, but it can be implemented as such a software system that is composed of some instances of extended classes associated with very coherently complicated relationship. If we want to develop AO languages and programming tools based on the OO method for implementing software agent, we have to suitably extend the attributes and methods of OO class and OO class oneself so that these extensions can express the above **relationship that is presented by the design framework of software agent**, and then we can use this extended languages and programming tools to implement concrete software agent so that it can really exhibit flexible autonomy features.

We have developed a prototype of the above language and programming tools, and that validates the works of this paper in some sense. But it should be noticed that: Because the focus of this paper is on the problem "How should the single agent be implemented as an OO software system?" the social feature of the agent has not much been discussed in this paper, and then it cannot satisfy the requirement of the Multi-Agents System. In addition, the learning and adapting abilities of agent have not been considered yet. These problems need to be researched into further.

References

[1] M. Wooldridge, N.R. Jennings, and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design", *International Journal of Autonomous Agents and Multi-Agent System,* 3(3), 2000, pp. 285-312.

[2] Bauer, J. Muller, and J. Odell, "Agent UML: A Formalism for Specifying Multi-Agent Software Systems", *International Journal of Software Engineering and Knowledge Engineering,* 11(3), 2001.

[3] M.J. Wooldridge, G. Weifl and P. Ciancarini, "Agent-Oriented Software Engineering II ", *In Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*.

[4] F. Giunchiglia, J. Mylopoulos and A. Perini, "The Tropos Software Development Methodology: Processes, Models and Diagrams ", *In Proceedings of the Third International Workshop on Agent-Oriented Software Engineering (AOSE-2002)*.

[5] M. E. Bratman, D. J. Israel and M. E. Pollack, "Plans and resource- Bounded practical reasoning", *Computational Intelligence,* 4, 1988, pp. 349–355.

[6] S. Rao and M. Georgeff, "BDI Agents: From Theory to Practice", *In Proc. of the 1st Int. Conf. On Multi- Agent Systems (ICMAS-95)*.

[7] R.A. Brooks, "Intelligence Without Representation", *Artificial intelligence,* 47,1991, pp. 139-159.

[8] J. Muller, "A cooperation model for autonomous agents", *LNAI 1193,* Springer, Berlin, 1997, pp. 245-260.

[9] Y. Shoham, "Agent-Oriented Programming", *Artificial intelligence*, 60(1), 1993, pp. 51-92.

[10] M.J. Wooldridge, *An Introduction to MultiAgent Systems*, John Wiley & Sons, Inc. 2002.

[11] Gamma, R.Helm, R.Johnson and J.Vlissides, *Design Patterns-Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.