A UML Extension for Agent-Oriented Analysis and Design

Amir Zeid

Email:azeid@aucegypt.edu,
Department of Computer Science,
The American University in Cairo

Abstract. In recent years, agent-based systems have received considerable attention in both academia and industry. The agent-oriented paradigm can be considered a natural extension to the object-oriented (OO) paradigm. Agents differ from objects in many issues which requires special modeling elements. Although there is a well-defined OO analysis and design modeling language (UML), agent-oriented development has neither a standard analysis and design modeling language nor a standard development process. In this paper, we propose a software engineering process for agent-oriented analysis and design. The process is based on existing object-oriented software engineering concepts. We also extend the Unified Modeling Language (UML) to model agents.

Keywords: Software Engineering methodologies, Intelligent agents, UML extensions, Mobile agents.

1.0 Introduction

Agent-Oriented (AO) technology builds upon and extends the now widely accepted Object-oriented (OO) approach to system design. The OO paradigm views an application as being composed of entities called objects that interact with each other by sending and receiving messages. Objects are grouped into classes. Then, class relationships can be captured in separate hierarchies. Objects are reactive entities as they receive messages and react to specific external events.

The AO paradigm goes one step further by viewing an application as being made up of one or more autonomous agents. Each agent continuously receives input. Agents respond to the environment where they reside by taking actions that affect the environment. Agents are pro-active entities, evaluating situations and then acting accordingly. Agents do not require specific instruction messages. Current applications being developed based on AO technology include Spacecraft Fault Diagnosis, Management and Business Process Management, Air Traffic Management, Air Mission Simulation and Telecommunication Network Management.

Objects in Object-Oriented Programming (OOP) represent real-world counterparts such as inventory items, documents, financial transactions and business processes. Objects are an intuitive mapping of the problem space to a computational model. They are characterized by state and defined behavior. They interact by passing messages since object internals are obscure.

Agent-oriented programming deals with both object and agent primitives. In some ways, agents may be thought of as objects, because they are also characterized by state and behavior. Unlike objects, agents have goals and proactively work to achieve them. On one hand, object methods are simply procedures coded by a programmer to deal with different messages. For example, a dog object can bark when pinched. On the other hand, an agent can synthesize plans to deal with contingencies not foreseen by the programmer and can learn from experience. For example, a dog agent can discover that insistent barking gets him fed a lot quicker.

The rest of this paper is organized as follows:

- Section 2 gives a background about agent-based software engineering and the UML (Unified Modeling Language)
- Section 3 describes our proposed process and extensions to the UML
- Section 4 shows the Rational environment after adding the new notations and diagrams
- Section 5 concludes our paper and suggests research directions

2.0 Background

In this section, we will discuss the differences between agents and objects. A brief description of the UML and one of its supporting tools (Rational) will also be introduced.

2.1 Agents Vs. Objects

After a thorough investigation of OO analysis and design techniques it was found that they are not directly applicable to the development of multi-agent systems. This is basically due to the conceptual differences between objects and agents:

- Agents have a more complex behavior and structure than objects, and in this respect they are more comparable to subsystems in some OO methodologies, (e.g., [WWW-90]). Their internal structure differs from objects since agents have a more complex underlying functional architecture such as the belief-desire-intention (BDI) architecture [RG-92]. In this respect they are on a higher level of abstraction than objects.
- In contrast to objects that are rather passive entities, agents are active. On one hand, objects immediately become active through messages and in this respect they are benevolent. On the other hand, agents act on their own behalf by following their goals, and can decide whether they act and respond to the events and messages received from other agents.
- Mobility modeling is not handled by existing OO methods.

2.2 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of engineering practices that have proven successful in the modeling of large and complex systems. The UML is a language, it is not just a notation for drawing diagrams. The UML is not a process, it provides a modeling language without enforcing a lifecycle.

The UML can be extended to model new elements using the concepts of stereotyping and tagged values.

Stereotypes

Stereotypes are a mechanism for introducing new types of modeling elements. Stereotypes are strings enclosed in (<< >>), preceding the name of an element. They may have an associated graphic icon or graphic marker. The new modeling elements that can be introduced using stereotypes must be subclasses of existing modeling elements [Alhi-98].

2.3 Rational© Suite

The Rational Suite package is an integrated, full-lifecycle solution for software architects, designers, analysts, and developers. Available in Windows, UNIX, and RealTime editions, Rational Suite gives the power to create more resilient, component-based architectures; automatically generate test harnesses for thorough model- and scenario-based testing; and utilize run-time tools that automatically pinpoint hard-to-find bugs, highlight performance bottlenecks, and find untested code [Rational-2002].

2.4 Related work

Agent-based analysis and design methods can be categorized into two main groups:

- Methodologies based on object-oriented methodologies: The Gaia methodology by Wooldridge et al and AUML by Odell et all are the well-known examples of such methodologies. Other methodologies include: [Bur-96], [Bau-2001] and [Ken-2001]
- Methodologies that adapt knowledge engineering: [Igl-98] and [Luc-97].

Our proposed extensions belongs to the first category since it is based on UML concepts.

3.0 Proposed UML extensions

As mentioned before, we propose a process based on UML to model agents (static and mobile). The process covers both analysis and design phases.

3.1 Analysis

Analysis aims at answering "what" the system does rather than "how" to do it. The proposed analysis process is illustrated in Figure (1). The first step is to define the expected tasks of the system. The second step is to identify "places", agents and classes. Then, a static model of the identified agents, classes and places is constructed. The analysis process is concluded by filling a data dictionary.

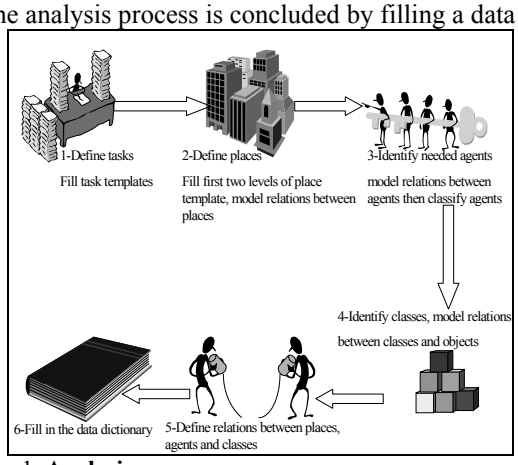


Fig. 1. **Analysis process**

3.1.1 Define tasks

Agent-oriented paradigm is goal-oriented. Consequently, the first step should be defining expected tasks because agents will be identified based on these tasks. Figure (2) gives the syntax of the task template. A task template will be filled for each task. The template includes the task name, its super-task (if it has any), sub-tasks (if it can be sub-divided), alternatives to achieve the task and exceptions.

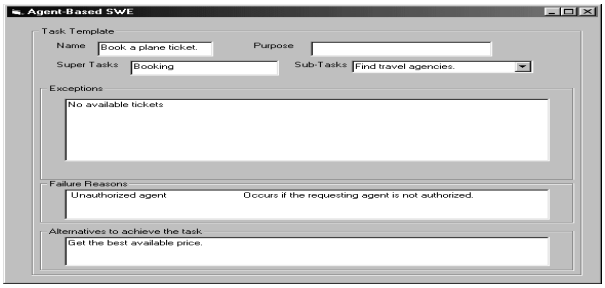


Fig. 2. **Task template**

3.1.2 Identify Places

The second step is to define "places". The concept of places is associated with mobile agents. When a mobile agent transfers itself, it travels between execution

environments called *places*. A *place* is a context within an agent system in which an agent can execute. This context can provide functions such as access control. The source place and the destination place can reside on the same agent system, or on different agent systems that support the same agent profile. A place is associated with a location, which consists of the place name and the address of the agent system within which the place resides. When a client requests the location of an agent, it receives the address of the place where the agent is executing.

A place template will be filled for each identified place. The template contains place name, purpose and super-place (if it has any). Figure (3) shows the syntax of a place template. The following level describes the communications among places, which places are safe to accept agents from, which places are authorized to send agents to. The last level of the template contains the attributes and methods of the place.

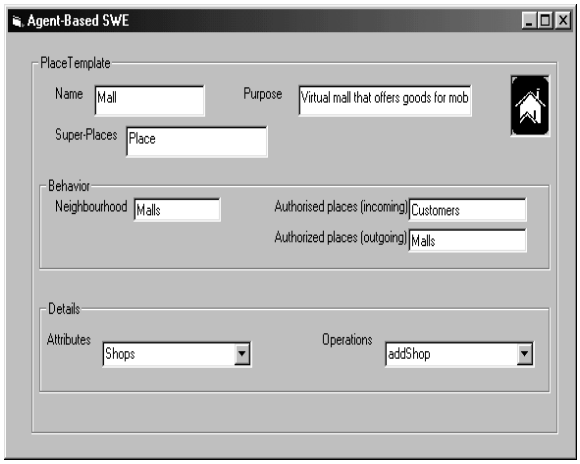


Fig. 3. **Place template**

3.1.4 Identify Agents

Agents can be considered similar to active objects in UML. In order to be able to have secure environments for mobile agents, the concept of agent's owner is utilized. The owner of an agent can be defined as the place where it was created or originated from. The owner can later be used as a security identifier for the mobile agent. As for mobile agents there are three main actions that they can do. Mobile agents can *go to a place*, *meet another agent who resides in the same place* and *connect to another agent on another place*.

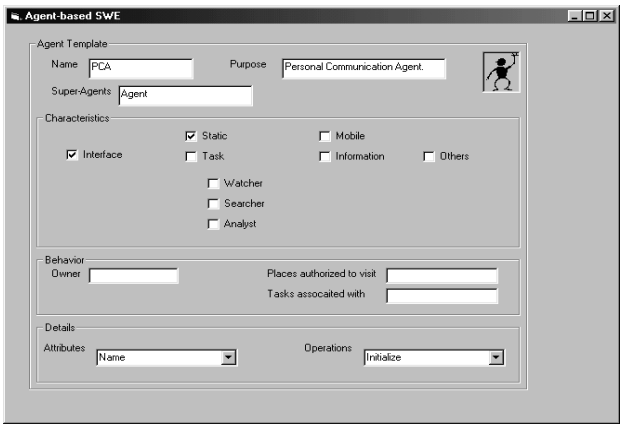


Fig. 4. **Agent template**

A good starting point to identify candidate agents is to assign an agent for each task identified earlier in the first step. Moreover, agents can be identified for other purposes such as representing users, devices and databases. Figure 4 shows an example of agent template.

The next step is to classify the identified agents. First, classify them in terms of mobility, whether they are static or mobile. Then agents will be classified according to their functionality:

- *Interface agents* are agents that interact with the user receiving user specifications and delivering results. They acquire, model and utilize user preferences to guide system coordination in support of the user's tasks. The main functions of an interface agent include: collecting relevant information from the user to initiate a task, and asking the user for additional information during problem solving.
- *Task agents* are agents that support decision making by formulating problem solving plans and carrying out these plans through querying and exchanging information with other software agents. A task agent receives user delegated task specifications from an interface agent and forms plans to satisfy these goals. Task agents are then classified into three categories:
    - *Watcher agents* are agents that continuously monitor a stream of information and notifies its customer when it finds out ,
    - *Searcher agents* are agents that actively search,
    - *Analyst agents* are agents that make conclusions out of given data.
- *Information agents* are agents that answer a one-shot query about associated information sources.

3.1.5 Identify classes and objects

This step can be done using any of the existing OO methods. We chose Unified Modeling Language (UML) notations since UML is the most commonly used set of notations.

3.1.6 Model relationships between agents, classes and places

In this step, a static view of the whole system is constructed. The aim is to describe the relationships between agents, classes and places in one global view. Figure (5) illustrates the notations that will be used. It includes the notations for static and mobile agents, classes and places.

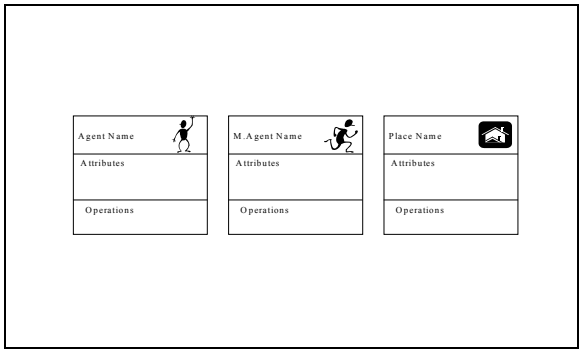


Fig. 5. **Static and mobile agents icons**

3.1.7 Create data dictionary

The concept of a data dictionary has been used in traditional software engineering and database tools for many years. A data dictionary is a central repository of definitions of terms and concepts. Without it, the process has little semantic content. A data dictionary is a vital tool by virtue of its role as a single place to look up definitions. It provides clarity of thought and transition from phase to phase. A further important reason for having a data dictionary is that it helps someone unfamiliar with the development to understand it. Without a data dictionary the unfamiliar reader is often forced to rely on decoding meaningful identifiers. A set of shared vocabulary and meaning for concepts (often called an ontology) is then defined to conclude the analysis process.

3.2 Design

Design deals mainly with how to implement the system to produce the expected outcomes as stated in the analysis phase. The proposed design process is illustrated in Figure (6). Design starts by defining authorization rights for places. After that, state machines for agents, places and classes are constructed. Then, a life-cycle model (section 3.2.3) is constructed for each major task. Related places are then grouped in neighborhoods. The next step is to describe the dynamic behavior of the system by defining mobile agents itineraries and modified interaction diagrams (based on Jacobson's software engineering process[Jac-92]). The design phase is concluded by detailed design for methods (operations) and attributes of classes, agents, and places. Finally, the data dictionary is updated.

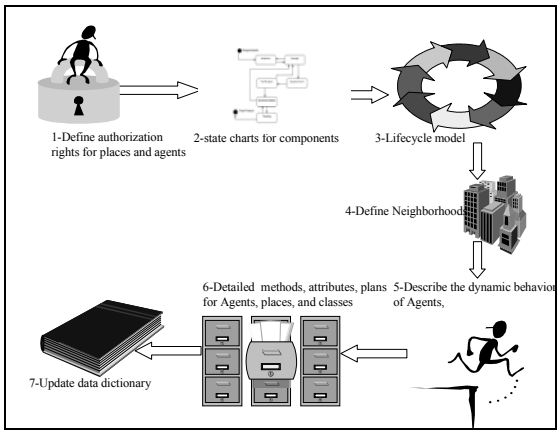


Fig. 6. **Design process**

3.2.1 Define authorization rights

The concept of authorization rights for each defined place is introduced because security issues have to be taken into consideration when dealing with mobile agents. Authorization rights indicate which agents are allowed to visit a certain place (their owner must be trustworthy). Places templates will be updated accordingly.

3.2.2 Define State charts for agents, classes and places

State diagrams render the states and responses of a component. These diagrams describe the behavior of a class in response to external actors. These diagrams contain states and transitions. We use the UML notations for state diagrams.

3.2.3 Life-cycle model

The life-cycle model describes the behavior of agents and how they communicate with the environment from creation to death. A life-cycle expression defines the allowable sequences of interactions that an agent may participate in, over its lifetime. Figure (7) shows syntax and semantics of life-cycle expressions. The idea is applied in the Fusion method [Col-94]. Life-cycle expressions define patterns of communication. As an example consider a simple banking system. A very simple pattern occurs in response to balance inquiries. A check-balance event or task is always followed by the output of a current balance event. An example would be Inquiry= check_balance. #current_balance.

Alphabet: Any input or output event maybe used in an expression. Output events are prefixed with #.
Operators: Let x and y be life-cycle expressions, then:
.x.y denotes x is followed by y.

.x|y denotes either x or y occurs.
.X* denotes zero or more occurrences of x.
.X+ denotes one or more occurrences of x.
.[x] denotes that x is optional.
.X||y means arbitrarily interleaving the elements of x and y.

Fig. 7. **Lifecycle notation**

3.2.4 Define subsystems and neighborhoods

The next step is to start defining subsystems and neighborhoods. Subsystems are groups of agents, classes and places that collaborate together to perform a set of related tasks. First, we have to introduce the concept of a "Neighborhood". Two places are in the same "neighborhood" if they are physically on the same host; i.e. agents do not have to travel from one place to another to perform a task if the two places are in the same neighborhood. Neighborhoods are useful to keep track of mobile agents. The main goal is to minimize agents roaming over the network.

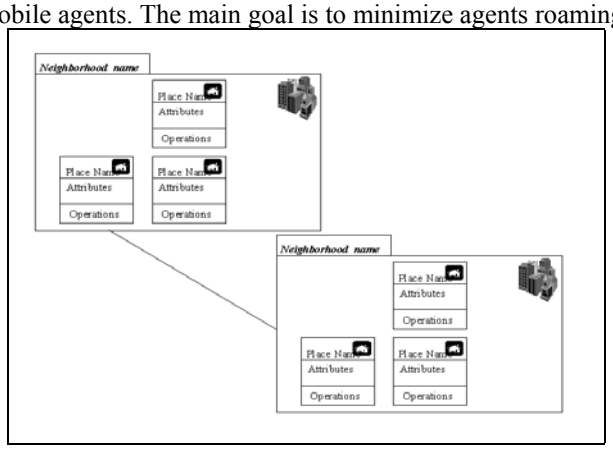


Fig. 8. **Neighborhood diagram**

3.2.5 Describe the dynamic behavior of the system

An itinerary graph is constructed for each scenario a mobile agent participates in. The itinerary graph shows places where the mobile agent passes by to achieve a certain task. Figure (8) shows an example of the itinerary graph of a personal travel agent. The example shows a mobile agent (Personal Travel Agent) that travels between 3 places (Personal PC, Flight Status Place and Paging Place) in order to book a ticket for a customer. The character indicates where the mobile agent started its itinerary. The number on each place indicates the number of the detailed interaction diagram that describes the interactions between the mobile agent and agents/classes in that place. For each identified task, an interaction diagram will be constructed.

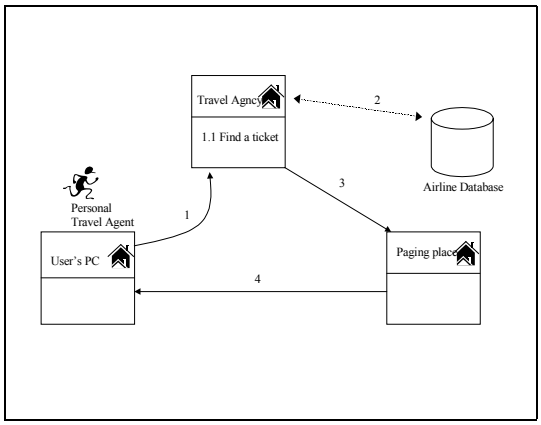


Fig. 9. **Itinerary graph**

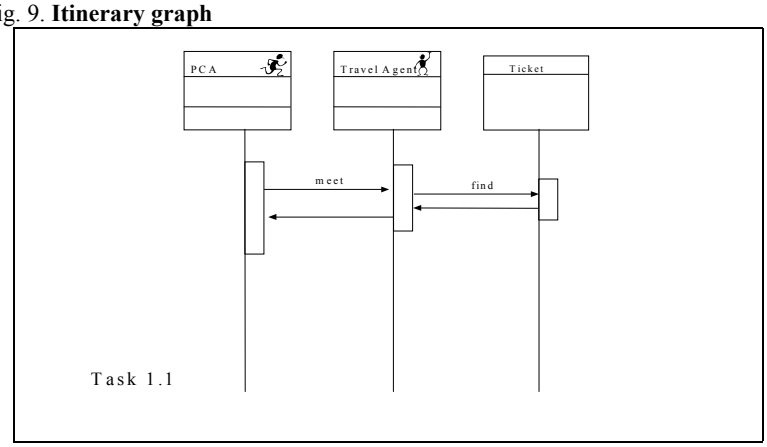


Fig. 10. **Interaction diagram**

3.3 Stereotypes (extensions to UML)

The following table includes a list of the stereotypes we added to the UML

Name	Application	Name	Application
<<Agent>>	Class	<<Life Cycle>>	Package
<<Mobile Agent>>	Agent	<<Task Template>>	Package
<<Place>>	Class	<<Interface Agent>>	Agent
<<Neighborhood>>	Package	<<Task Agent>>	Agent
<<Agent Template>>	Package	<<Info. Agent>>	Agent
<<Itinerary graphs>>	Package	<<go>>	Operation
<<Place Template>>	Package	<<meet>>	Operation

4.0 Extending Rational tools

We used the proposed extensions to extend the Rational tool. Extensions included new diagrams, new notations to represent agents, places and neighborhood and new operations and connection types.

4.1 Upgrading Rational Configuration files

To upgrade the Rational tool, The *Stereotype Configuration File* has to be edited. A stereotype configuration file defines a set of stereotypes, including the location of icon files as well as settings that affect the usage and display of the defined stereotypes. Typically, the extension of a stereotype configuration file is .ini, but any extension is allowed. Each stereotype configuration file needs to be located in the Rational Rose installation folder, defined by the InstallDir registry value, and listed in the StereotypeCfgFiles registry section.

Below is a sample of one of the stereotypes we added to the Rational tool.

```
[Stereotyped Items]
Class:Mobile Agent
 [Class:Mobile Agent]
Item=Class
Stereotype=Mobile Agent
Metafile=E:\AUC\MyResearch\mag.emf
SmallPaletteImages=E:\AUC\MyResearch\mag.bmp
SmallPaletteIndex=1
MediumPaletteImages=E:\AUC\MyResearch\mag.bmp
ListImages=E:\AUC\MyResearch\mag.bmp

ToolTip=Creates a Mobile Agent
```

4.2 The new Class diagram menu

Figure (11) is a snapshot of the new class diagram menu after adding our new features: Agent, Mobile Agent, Place, Neighborhood, Authorized relation. The figure also shows that the meta-model is updated.

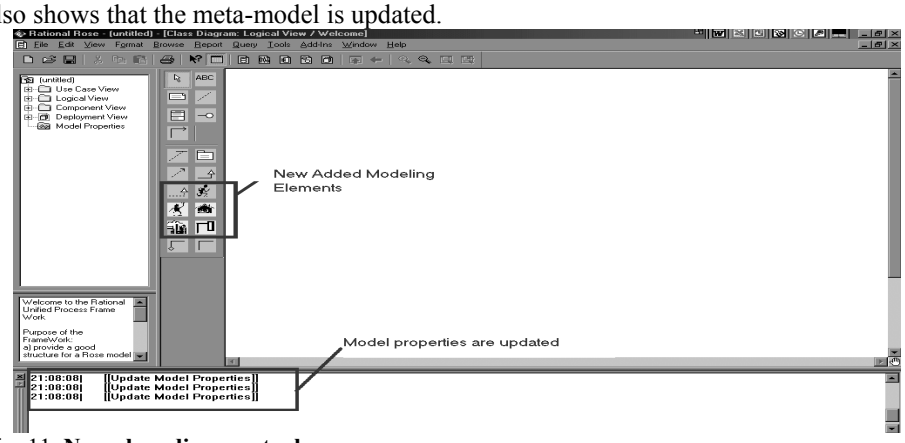


Fig. 11. **New class diagram tools**

4.3 Example

Figure (12) shows an example of a static relation diagram. It shows static agents, mobile agents, places, neighborhood, and classes. The model describes a virtual mall

that contains shops (places), each shop has an inventory that may contain items. The PCA travels from the user's host (given it has authorization rights) to the virtual mall. The PCA can communicate with buyers and sellers to get the best price of the item he is looking for.

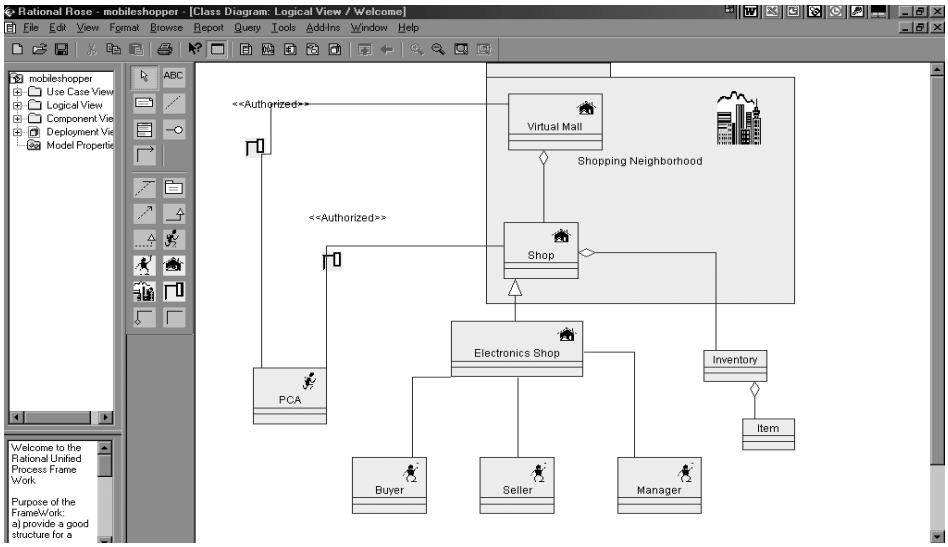


Fig. 12. **Example using the new notations**

5.0 Conclusion

At this stage in the life of the agent-oriented paradigm, it is in the best interest of the software development community to have a standard agent-based modeling language. In this paper we presented some extensions to the UML to model agents (static and mobile). The new extensions include notations for some new elements like agents, places and neighborhoods. The extensions also include some new diagrams like the itinerary and authorization graphs. The extensions also include some modifications to existing diagrams like the interaction diagram. We integrated our proposed extensions with the Rational suite. Future work will include code generation for agent-based languages.

References

[Alhi-98] Alhir, Siman. UML in a Nutshell. USA: O'Rielly. 1998.

[Bau-2001] Bauer, B, et al. Agent UML: a formalism on specifying multi-agent software systems. Proceedings of the first international workshop (AOSE-2000).

[Bur-96] Burmeister, B. Models and methodologies for agent-oriented analysis and design, DFKI document D-96-06 DFKI.

[Col- 94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes: Object-Oriented Development. The Fusion Method, Prentice Hall, Englewood Cliffs, 1994.

[Igl-98] Iglesias, C. et al. Analysis and Design of multi-agent systems using MAS-commonKADS. In Intelligent Agents, IV, LNAI V 1365, Springer Berlin, 98.

[Jac-92] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard: Object-Oriented Software Engineering. A Use Case Driven Approach, ACM Press/Addison-Wesley, 1992.

[Kin-93] D. Kinny: ``The Distributed Multi-Agent Reasoning System Architecture and Language Specification", Australian Artificial Intelligence Institute, Melbourne, 1993.

[Luc-97] Luck, M et al. From Agent theory to agent construction: a case study. LNAIV 1193, Springer, Berlin, 97.

[Rational-2002] http://www.rational.com/uml/

[RG-92] A. S. Rao, M. P. Georgeff: "An Abstract Architecture for RationalAgents" in B. Nebel, C. Rich, W. Swartout (eds.): Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR-92), Morgan Kaufmann, San Mateo, 1992.

[WWW-90] R. Wirfs-Brock, B. Wilkerson, L. Wiener: Designing Object-Oriented Software, Prentice-Hall, Englewood Cliffs, 1990