Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
1999

MOBILE AGENTS: PATTERNS AND REFLECTION

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange
project funded by the European Community)

By: James Roberto Windmüller

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Thomas Ledoux (Ecole des Mines de Nantes)

Acknowledgments

Each conquest in our lives has a special meaning and also a lot of hard work behind it. But the success is not due only by our efforts, if we look around us there are many factors and people that helped us to reach the success.

I would like to thank my wife Cintia, that has been walking by my side for several years and also during the last year when I took my course, for supporting me and giving me the courage to finish my work.

I want to leave a special thanks to my advisor Thomas Ledoux, that was always ready to help me when I needed, showing me the way that I should take and spending many hours discussing with me about my subject and related papers, books and other documents. I always had a feedback from him about my doubts, my questions and my work even when he was busy with other important things not related to my research.

I want to thank my father Válter and my mother Glacy, for following my studies since my first day in school. Now I can understand why they always gave a special stimulus to me.

I also thank Alberto and Vânia, my wife’s parents, for helping and advising us during all those years that we have been married.

Thanks Ricardo José Peres, my friend, for taking care of my business in Brazil during this time, and the company POLO de Software S.A., where I have been working the last eight years, for allowing me to stay out in order to do my course.

I special thanks to Prof. Dr. Edson Scalabrin and Pontifícia Universidade Católica do Paraná – PUC-PR for giving me the opportunity to participate in this project.

Finally I would like to dedicate my work to all those mentioned above.

Table of Contents

List of Figures

List of Tables

Abstract

A great effort has been done over several years in order to propose new software development technologies that can turn the software development process easier, improving the productivity and making the costs lower. “Agent Technology” is an example of such technology that makes it easier to design, implement, and maintain distributed systems.

“Agent Technology”, especially mobile agents, offers us a lot of advantages, but also offers many challenges that we have to deal with. One of the challenges is to find and give a solution to the recurrent problems met during the use of the “Agent Technology”. Making agents interact with other agents without knowing their existence, identifying agents by their capabilities and allowing an agent to try alternative destinations when some environment is temporally out of order are some examples of those recurrent problems. Having those problems in our hand, we can after apply already proven software engineering techniques such as *“Software Design Patterns”* and *“Reflection”* in order to give to each one a possible solution in an abstract format that will help programmers develop their own mobile agent applications.

In this document we give a general overview of the main elements and properties of mobile agents like definitions, advantages and their applicability followed by a small example of using mobile agents. Then, we present five design patterns that solve particular problems that we have met during our research and when we applied the mobile agent technology in our example.

Introduction

The computer science area have been done a great effort in order to propose new software development technologies that can turn the software development process easier, improving the productivity and lowing the costs. Each of those technologies when announced changes the way of thinking and acting of the software engineering community.

The Object Oriented technology is a good example of that. It was one of the responsible for the changes that occurred in the software development process. Many other technologies, based on Object Oriented technology, were proposed and adopted as the main solution for several software applications after. The Distributed Object technology is one of those approaches that has made a great difference, allowing objects to reside in different computers and also permitting them to communicate each other in order to exchange data.

Following that comes a new era were the objects in a distributed environment got a new structure and gained one basic feature: *mobility*. This new technology is called “Agent Technology”, and it allows software developers to create a new kind of application where the objects now can move themselves from one host to another carrying their code and state. The object that possesses those characteristics is called “Mobile Agent”.

Every new technology demands, before using it, a deep study of all its principles, fundamental concepts and techniques. That study will provide us the basic knowledge needed in order to put all the characteristics and features offered by the studied technology in practice. The practical part after the study of the new technology is also very important. It will organize all pieces loosed in our mind, identifying the role that each one of them plays, showing the level of importance that each one have in the whole, the dependency existing among them and how they interact each other. When we reach a reasonable level of knowledge about a new technology, in theory and practice, we are able to identify the spots of complexity and the main problems that we can face when using the related technology. This process of analyzing new technologies is necessary for all new proposed technologies, including the “Agent Technology”.

The “Agent Technology” has its properties, concepts and elements like autonomy, social ability, reactivity, pro-activity and mobility. Based on that, we can identify the main required characteristics in a mobile agent based application and also the problems that we met when dealing with the implementation of those required characteristics, for example:

- How can mobile agents interact with other agents ?
- How can we control the access to the mobile agent ?
- How to provide a transparent access to a mobile agent when it is in a remote environment ?
- How can we identify mobile agents by their capabilities ?
- How can a mobile agent in a local environment find other agents without knowing their existence ?
- How can a mobile agent change the destinations where it should go dynamically ?
- How can a mobile agent try alternative destinations when some host is temporally shut down ?

Those questions provide us good reasons to focus our efforts on the identification of possible solutions to them. Usually, to help people solve the recurrent problems we can use different techniques. One of the most used techniques in software engineering and that is very well known and accepted in the object oriented community is "*Software Design Patterns*" [18], [19]. In our work we show a preliminary study about "Agent Technology", mobile agents specially; followed by a practical example giving an approach of the main characteristics of mobile agents.

So, the main recurrent problems of mobile agents are identified and, as the main part of our work, we propose a specific "*Design Pattern*" for each one of them. The following design patterns are proposed in order to help programmers to solve the recurrent problems found when dealing with mobile agent technology:

- Abstract Agent Pattern
  Permits a programmer define the main structure for a mobile agent based application

- Agent Proxy Pattern
  Defines the main mechanism of a mobile agent.

- Agent Coordinator Pattern
  Allows an environment to control all agents that are running.

- Agent Interaction Pattern
  Allows the agent localization and interaction based on their capabilities

- Travel Plan Pattern
  Controls the destinations where the agent must go and allows dynamic changes to the itinerary.

Finally we finish our research showing the Java Documentation of a small application implemented in Java [31] using RMI where we can see the proposed design patterns in use. We want to remember that using the proposed patterns we don't need to use a special tool or agent programming language like Aglets [27], Voyager [29], etc.

Chapter I: Mobile Agents

1 State of the Art

1.1 Introduction

The Object Oriented technology is one of the most important software development technologies that appeared in the computer science area. It really changed the way of developing software, improving the productivity and making the software costs lower. Moreover, the software developed based on Object Oriented technology is much more reusable, reliable and adaptable to the natural changes during its life cycle. Another important technology in use nowadays is the Distributed Object technology. It allows objects residing in different computers to communicate each other in order to exchange data. Those facts created a revolution in the software industry and the prove is that many applications running today are based on those technologies.

After that comes a new era were the objects in a distributed environment got a new structure and gained one basic feature: *mobility*. This means that the objects now can travel to different hosts over the network, carrying their properties and behavior. This new technology is called “Agent Technology” and it provides new properties and characteristics in addition to the known object oriented properties.

The benefits of the “Agent Technology” are everywhere. It allows software developers to create a new kind of application where the local host is not the limit anymore. The small objects now can move themselves from one host to another carrying their code and state. They also have the ability to start executing their tasks in somewhere and resuming the same task anywhere else. Applications developed using that kind of characteristics are called “Agent-Based” applications and the objects that are capable of moving around different environments and hosts are called “Mobile Agents”.

In this section we will present an overview of the “Agent Technology”.

1.2 What is the best definition for Agent?

"An agent is a program that assists people and acts on their behalf. Agents function by allowing people to delegate work to them". (Danny B. Lange) [1].

The best way to start talking about mobile agents is giving a definition about what an agent is and what the term "agent" means for different authors that use or research about agent-based applications. Regarding to the enormous number of people working and researching about agent-based systems, we can suppose that it is easy to find a definition. However, there is not a unique definition or a consensus on it. Many papers [9], [10], [11] try to give a definition for agents but what we can see is that each author focus his/her definition in the research or the application where the agent technology was used.

1.2.1 The Weak and the Strong notion

One of the classical documents available about agents and very adopted by researchers is "Intelligent Agents: Theory and Practice" written by Wooldridge and Jennings [11]. In this paper the authors consider two different ways of use for the term "agent":

1) The weak notion of an agent;
2) The strong notion of an agent.

The weak notion of agent is related to the autonomy, social ability, reactivity and pro-activity properties of a hardware or software-based computer system.

In addition to the weak notion, the strong notion of agent regards also to a set of properties that resemble the human-like qualities like knowledge, belief, intention and obligation. The researchers in the AI area call those characteristics as mentalistic notions.

1.2.2 Definitions of Mobile Agents and Their Properties

- Definition of an Agent

For Smith, Cypher and Spohrer [12], agents are *"persistent software entities dedicated to a specific purpose. 'Persistent' distinguishes agents from subroutines; agents have their own ideas about how to accomplish tasks, their own agendas. 'Special Purpose' distinguishes them from entire multifunction applications; agents are typically much smaller"*.

"Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed". This is the agent definition for Maes [13].

- Definition of a Mobile Agent

"A mobile agent is not bound to the system where it begins execution. It has the unique ability to transport itself from one system in a network to another. The ability to travel, allows a mobile agent to move to a system that contains an object with which the agent wants to interact, and then to take advantage of being in the same host or network as the object". This is the definition of mobile agent for Danny B. Lange [1]

We could list here several definitions that in some way can be very similar each other and in the other hand they can be completely different. All of them will talk about agents in a particular way, describing agents according where it was applied. Taking in account various papers [9], [10], [11], [14] we can say that a mobile agent is a software object that is located in a certain logical environment where it can perform its tasks and has the following properties:

- **Autonomy:** Agents act in the environment according to their tasks and have decisions over their own actions.

- **Social Ability:** Agents interact with other agents using a common language.

- **Reactivity:** Agents perceive their environment and respond in time for changes that occurred.

- **Pro-Activity:** an agent is able to act not only by a stimulus, but also taking the initiative by itself.

- **Mobility:** an agent is able to move among the different environments; stoping the current processes and keeping its state. After moving to another host it can resumes its task.

1.3 The Mobile Agents conceptual model

Communication features are essential when we talk about distributed systems, remote computing and of course mobile agents. Without the basic network infrastructure and communications protocols and other set of hardware devices it would be impossible to have the large number of application and solutions working in local and wide area networks as we have today.

1.3.1 The RPC Model

In the beginning, applications usually worked in a stand alone computers without sharing data, CPU and other devices such as printers, hard disks, etc. The old mainframes offered the communication between the main CPU and the terminals, but no computing process were done in the terminals. Thus, the local networks came to make a revolution in the usually known computing environment, providing the opportunity to applications to share data, hardware and others devices. Applications were built following the client/server architecture where client computers perform remote calls to procedures existing in the server machine. The server, in its role, answers to those calls sending the results of the procedure invoked. This process is called RPC – Remote Procedure Call and it is represented in Figure 1.

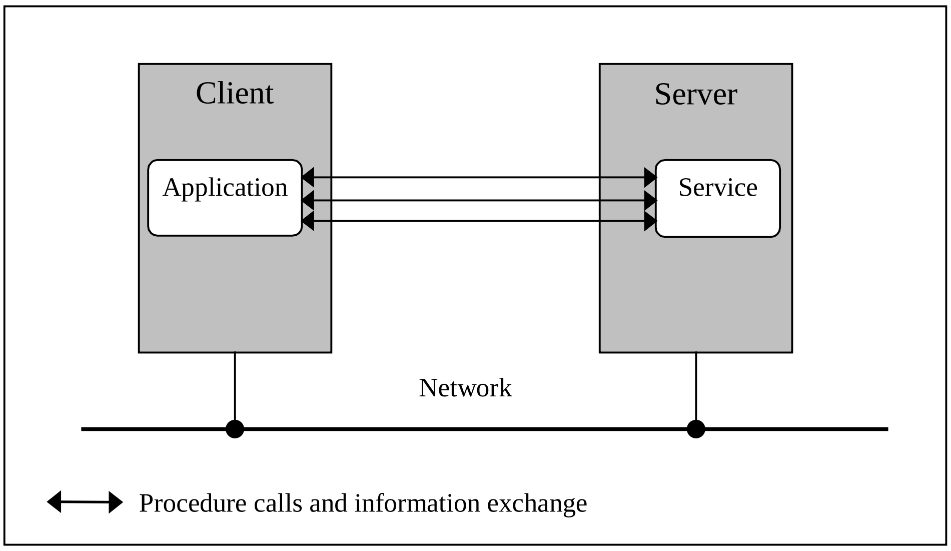

Figure 1: RPC conceptual model.

We can observe in this model that the applications in the client side are dependent of the procedures existing on the server. This centralization has some disadvantages, for example the overloading of the resources (processor, memory, etc.) on the server side.

1.3.2 The Code-on-Demand Model

The improvements done on the RPC model and on the traditional client/server applications associated with the success of the internet, helped the computer science community to propose a new kind of computing, known as "Code-on-demand Paradigm". In the RPC model the server side held the procedures (know-how), the data (knowledge) and also was the responsible to execute those procedures in order to answer to the client requests. The new code-on-demand model differs from that by sending the procedures (know-how) to the client side once detected that it does not have the required know-how to deal with the knowledge (see Figure 2). Then, the client side can execute those procedures by itself using its own resources (processors, memory, etc.), decreasing the load over the server resources. The most famous example of this can be viewed when an applet is downloaded to the client web browser and executed locally accessing servlets the are loaded in the server side.

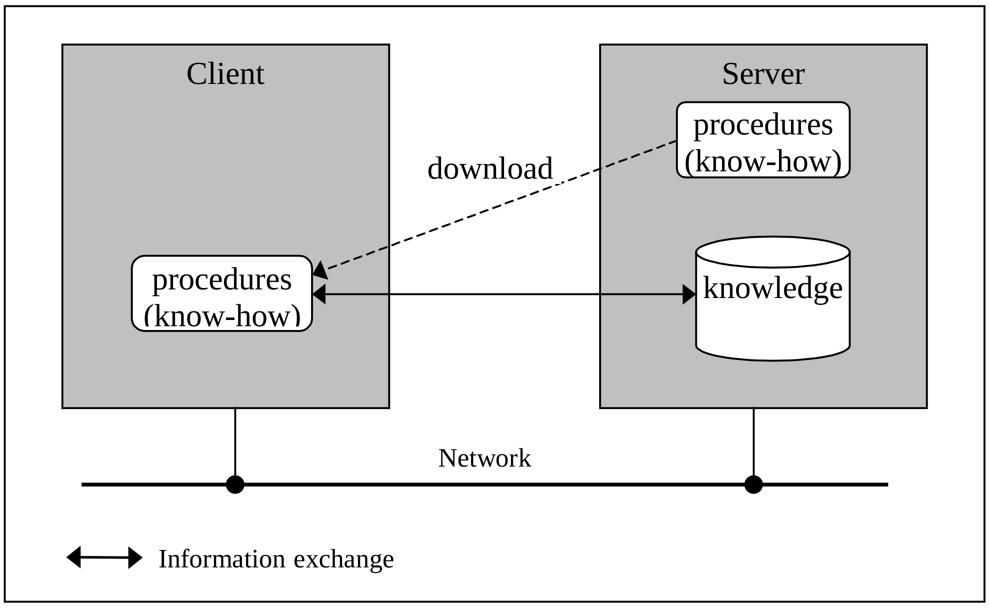

Figure 2: Code-on-demand Model

1.3.3 The Mobile Agent Model

Over the last years applications are becoming bigger and bigger and the number of remote calls performed in the server and the amount of data transported are growing every day bringing up the problem about the network bandwidth, degrading the performance on it and also in the applications.

Mobile agents appeared as a solution to this problem because mobile agents are based on the principle of take the processes to the data and not bringing the data to the processes. In this way we can see that using mobile agents is possible to reduce common problems faced by RPC such as network degradation. The mobile agent concept can be viewed in the Figure 3.

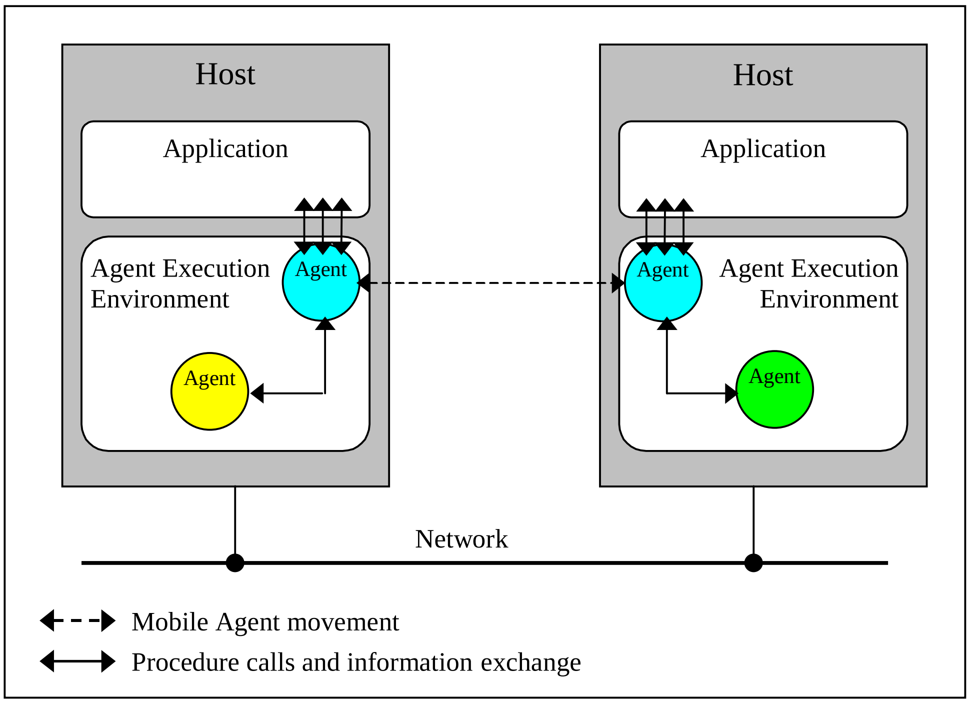
Host
Application
Agent Execution Environment
Agent
Agent
Host
Application
Agent
Agent Execution Environment
Agent
Network
Mobile Agent movement
Procedure calls and information exchange

Figure 3: Mobile Agents conceptual model.

Analyzing the figure above we can see the basic structure of an environment that is able to deal with mobile agents. There are not many differences between the hosts shown in Figure 3, both have an *Application* that interacts with an *Agent Execution Environment* that can use the features available today in the programming languages and the physic structure of a network in order to make the agents move. The environments can offer a set of services for the agents and depending on the interest of the agent they can profit from that services.

The *Application* box can be an agent-based system, an applet or any other application that is capable to connect to the environment and start an agent that has a specific task; like for example, to visit several computer stores in order to buy a computer. The agent receives the characteristics and computer configuration desired from the user and travels through the known environments in order to find the best option of computer.

The *Execution Environment* handles all calls that come from the agent. When the agent wants to move from one host to another host in order to visit another computer store it calls the environment and the environment will arrange to move the agent. The *Execution Environment* will be responsible to stop the agent thread, keep the values of attributes and other properties of the agent and move the agent object to another host in order to resume its operation. The *Execution Environment* is also able to provide to the agent basic and important information such as the host name where the agent is located, other agents that are working in the same host, other hosts address available, and a set of other useful information for the agent control.

Besides communicating with the environment and other applications, Mobile Agents are also able to communicate with other agents, thus the *Execution Environment* will act again dealing with operating system, protocols and networks in order to put agents talking to each other. Of course communication among agents is not a kind of easy task to do and we can say here that this part of the Agent Paradigm has been treated as special topic, providing enough complexity and also encouraging the researchers to study about communication languages [16], [17].

1.4 Advantages of using Mobile Agents

Mobile agents can be useful when implementing several different types of distributed applications and for each application agents can provide several advantages [1], [3] that we could not have by using a common object distributed system.

1.4.1 Reducing the network load

On of the main advantages provided by agents comparing to other technologies is that using agents you can have any computing process being executed in different host servers, in parallel or not. This fact allows the agent owner to turn off his/her computer while the agent is working and check the results obtained later. Once the agent will work locally in different hosts we can detect a reduction on the network load due the messages exchange between the agent and server be performed in the same CPU. Contrasting with the known object distributed applications, where there is a role of a client and a role of a server changing messages and data over the network, the mobile agent model move the process to the data.

1.4.2 Overcoming the network latency

Network latencies are not acceptable in networks when we are working with critical real-time applications like robots in manufacturing processes that need to respond in real time to changes in their environments. In this case mobile agents can be sent from the central controller to work locally acting direct on the robot controller and avoid the network latency.

1.4.3 Executing asynchronously and autonomously

Another situation where mobile agents can be applied is in applications that are based on fragile network connections or when continuously open connections become economically or technically feasible. The process can be implemented using an agent that will work autonomously after dispatched and will be caught later. Thus, the use of a network connection is necessary only when dispatching and retrieving the mobile agent after finishing its work.

1.4.4 Integration of Seamless Systems

Network computing is heterogeneous often from both hardware and software perspectives. Mobile agents are, most of the times, computer and transport layer independent, and they depend only on the environment where they will act. These characteristics provide us really good reasons to use mobile agents to solve problems such as the integration of seamless system.

1.4.5 Robust and Fault-Tolerant Distributed System

We can build, in an easier way, distributed systems that are more robust and fault-tolerant than non mobile agents based systems because mobile agents have the ability to react dynamically to unfavorable events and situations that are common in distributed computing environments. For example, if a host is being shut down, all agents executing on that host will be warned about this fact and will get some time in order dispatch to another host where they will resume their tasks.

1.5 Applications that can benefit from mobile agents

There are many areas in computer science where applications can be developed using mobile agent technology, especially those areas that demand for repetitive tasks and usually are done by humans. Other kind of applications where agents could be applied are: applications that need to run in a distributed environment, applications that can not be interfered by the latency of networks, applications that need to use full-time connections but technically or economically those connections are not feasible.

1.5.1 E-Commerce

As example of applications we can take the electronic commerce offered over the internet. Using mobile agents to implement *Electronic Commerce* applications we can have real-time access to remote resources like stock information and even agent-to-agent negotiation in behalf of ourselves. The agents can go over several different hosts and obtain the best option for what we are looking for.

1.5.2 Personal Assistance

Agents are providing *Personal Assistance* in applications used as personal agendas. In this kind of applications the agents can go visit other agents that represent other people, they interact with their schedules in order to make appointment for a meeting. The agent has also the ability to negotiate with the other agents in order to rearrange the appointment in case of any participant of the meeting has something else already scheduled for that time.

1.5.3 Distributed Information Retrieval

Agents are a good option also for *Distributed Information Retrieval*. Instead of moving large amount of data to the process, an agent can be sent to the host where the data is and work locally, avoiding the high traffic of information over the network.

1.5.4 Workflow and Groupware

In *Workflow and Groupware* applications agents can facilitate the flow of information among the coworkers using its mobility property and provide a certain degree of autonomy to the workflow items.

1.5.5 Monitoring and Notification

Agents can show their ability to work asynchronously in applications where the main goal is *Monitoring and Notification*. In this case an agent can monitor a source of information without being dependent on the application that originated it, and after notify the environment or other agents about what it found.

1.5.6 Information Dissemination

The use of agents for *Information Dissemination* is the facility that gives to agents the characteristic of “Internet push model”. In this case agents are used to disseminate information such as news and to make automatic software update. The agents bring the new components accompanied by the installation process to the client’s computers, and autonomously, install the new components and manage the software.

1.5.7 Parallel Processing

In *Parallel Processing* applications or process that require so much processor power, agents can be used to compose an distributed infrastructure of mobile agent hosts that can allocate that process and run them in parallel.

1.5.8 Data Source Mediation

Let’s suppose we have to work with several different data sources and those data sources do not talk to each other. In this case, mobile agents can be used as mediators among the data sources providing mechanisms that allow the incompatible data sources exchange information.

1.5.9 Information Filtering

Agents can be used to filter and sort incoming information in order to avoid overloads. The agent receives enough knowledge about its user’s needs and acts as a gatekeeper selecting only the really needed information and preventing its users from being overwhelmed by a flood of information. Filtering agents can also work together with searching agents in order to keep the searches results in an acceptable level of amount but in a high level of contents.

The following table can give us a general idea where agents can be used in practice.

Application of Agents	
Category	**Domain**
Enterprise Applications	▪ Smart documents (e.g., documents that 'know'that they are supposed to be processed). ▪ Goal-oriented enterprise (e.g., workflow) ▪ Role and personnel management (e.g., dynamically attaching roles and capabilities to people)
Inter-Enterprise Applications	▪ Market making for goods and services ▪ Brokering goods and services ▪ Team management
Process Control	▪ Intelligent buildings (e.g., smart heating/cooling, smart security) ▪ Plant management (e.g., refinary) ▪ Robots
Personal Agents	▪ Email and news filters ▪ Personnel schedule management ▪ Personnel automatic secretary

Table 1: Overview of practical application of mobile agents

As we can see mobile agents can be used for several purposes, all of them trying to benefit from the main properties of the agents: autonomy, social ability, reactivity, pro-activity and mobility.

1.6 Agent Communication Languages

Communication and interaction among agents have a fundamental role in the concepts of the "Agent Paradigm". In order to allow the interaction among agents, researchers are studying and proposing what they call of ACL - Agent Communication Languages [16], [17]. An ACL is composed of a common agent communication language added of protocols and a format for the content of the communication that are the main elements to achieve the desired agent interaction.

The agent communication languages can be divided in two different groups:

- Procedural Languages
- Declarative Languages

The procedural approach is based on communication language within procedural directives (e.g, TCL, AppleEvens, Telescript). It is also easy to be used with other very known and used programming languages where the communication is based on executable content like Java [31].

The declarative approach is based on idea that the communication language can be best modeled by the exchange of the declarative statements (definition, assumptions). Declarative languages basically rely on actions such as requesting and commanding, and they have an advantage over the procedural languages because procedural languages are limited due the difficulty to control, coordinate and merge executable content. Most of the authors agree with that and to formalize their opinion they suggest seven categories of requirements that an ACL should accomplish, see the following table.

Requirement	Description
Form	it should be declarative, syntactically simple, and easily readable by people and programs
Content	it should provides a distinction between the language that express communicative acts, called *"perfomatives"* and the language that transports the content of the message.
Semantics	it should exhibit those desirable properties expected of the semantics as any other language does.
Implementation	it should be efficient, adaptable to existing software, work transparently with lower layers, and allow agents to implement sub-sets of it.
Networking	it should be able to adapt to the most important aspects of modern networking and must be independent of the transport mechanism.
Environment	it must be able to handle heterogeneity and dynamism.
Reliability	it must support reliable and secure agent communication.

Table 2: Agent Communication Language Requirements

Many different agent communication languages have been developed and used. We can enumerate some of them:
- KQML
- Arcol and FIPA
- KIF
- XML-based

Among that ACL's, KQML is the one that has accomplished most of the requirements shown in the Table 2, and a good description and evaluation of it can be found in some documents, for example in "Evaluating KQML as an Agent Communication Language" (J.Mayfield, Y.Labrou and T. Finin.) [16].

1.6.1 KQML - Knowledge Query and Manipulation Language

KQML has been used to transport oriented-oriented data due its capability to accumulate a wide range of data, and especially in agent-based systems, KQML can be used to help the agent communication in the sense that agents work autonomously and asynchronously.

KQML Characteristics:

- Structures the messages with no concern about the content of the messages.
- Specifies the syntax but not the semantics.
- Supports basic protocols.
- Assumes that the message transport is reliable, so does not guarantee the delivery of the message.

An example of a basic *performative* in KQML is *tell* and its structure is denoted by:

```
tell
:content <expression>
:language <word>
:ontology <word>
:in-reply-to <expression>
:force <word>
:sender <word>
:receiver <word>
```

KQML offers many reserved *performatives* names that can be categorized as shown in the following table:

Category	**Name**
Generic informational	`tell, achieve, cancel, untell, unachieve`
Response	`reply, sorry`
Basic Query	`evaluate, ask-if, ask-about, ask-one, ask-all`
Multi-response query	`stream-about, stream-all , eos`
Generator	`advertise, ready, next, rest, discard, generator`
Networking	`register, unregister, forward, broadcast, route`
Capability-definition	`Advertise, subscribe, monitor, import, export`

Table 3: KQML Performative Names

To have a better idea of KQML language, let's take a look at an example of a message written in KQML.
For instance, let's say that an agent "Agent-A" wants to demand to an application the price of a computer. Using KQML we could have the following statements or declarations:

```
(ask-one
     :content "computerPrice()"
     :receiver AnApplication
     :language Java
     :ontology computers)
```

As we can see, using KQML we can send and receive messages containing or not parameters (data objects) in a simple format, composed in a language of our own choice where everything is wrapped inside of a KQML message. Considering that, KQML allows us to work in distributed systems and agent-based systems with a level of abstraction that turns the message exchanging among objects very flexible and robust.

1.7 Security and Agents

Another relevant and very important topic emphasized by the authors [15] and researchers working with mobile agents is about the security. There are several security issues in the mobile agent model that must be considered in case of having agents working in an environment that demands high level of security. Basically we should think about the following topics when secure agents environments are required:

a) Authentication of the sender, the owner and the creator of the agent: with this information we can have information such as the responsible for the agent and the responsible for the code of the agent.
b) Authentication of the agent: we can check if the agent is allowed to use some services, access files, etc.
c) Secure communication between agents: the agent should protect its privacy.
d) Auditing the activities: agent activities that demand security must be recorded in order to be audited by an administrator later.

Like in any other secure environment, the “Agent Paradigm” also talks about this subject. Here we can see that the security on the authentication and verification of the agent information and activities are the main focus.

1.7.1 User Authentication

Checking the information about the user that wants to start an agent in any environment. This user must be authenticated by the server where the agent will be started and by the agent execution environment where the agent will work. Sometimes environments may also perform the user’s authentication only by checking the rights of the group that the user belongs to. No user authentication needs to be done when the environment where the agent will work has no protected functions or information. With the user authentication process we can say that the user/agent knows the server/agent execution environment and vice-versa.

1.7.2 Malign Agents

Besides having an authentication for the user the server where the agent will run can also analyze the intention of the agent by looking in its functions and the resources it wants to access in order to assure the security of the system. This kind of procedure will work on the identification and detection of malign agents. Depending on the language that the agent was written, this task becomes more difficult because some agent languages allow self-modification, as does Telescript [26]. Even after being accepted by the execution environment, an agent is able to change itself from a benign agent to a malign agent, so in this case the execution environment should observe the agent during its execution to guarantee the security and avoid the viruses.

1.7.3 Virus Detection

Mobile agents are not the only way by which viruses can be propagated in a network. However, agents can really facilitate the propagation. Virus detection is one of the most difficult problems to solve when we are talking about security and agents. The only way to detect whether the agent intentions are good or not is applying its code in several tests. However, this is not enough to say that an agent will not try to corrupt the host system.

1.7.4 Proxies

In order to protect the agent information against inappropriate access we can define the use of a proxy to work as a shield that will protect all information. All applications, agents, environment, etc. will see only a proxy of an agent, and never will access the agent information directly. If we want to have a higher level of security on accessing the information about the agent, security algorithms such as authentication can be implemented in the proxy in order to filter and select the calls that are made to the agent methods.

2 Experimentation

Studying mobile agents, their properties, concepts and principles are essential before starting with a practical experimentation. The experience acquired when we put in practice the theoretical part, by using tools for developing agent-based applications, is also important in order to better understand the functionality of the whole mobile agents mechanism, where the mobility, the interaction and autonomy play the main roles.

2.1 Available Tools

There are available today several tools that can help us develop mobile agent based applications. Each of them has its particularities, benefits and drawbacks. Depending on the area where our agent based application will work, we can choose either one or other tool. An evaluation of each one using all its features in practical tests could help a lot when a decision must be done.

The main Mobile Agent Developing Applications were developed by the famous names in software industry, as show in the following table:

Application	**Producer**	**Language**
Aglets [27]	IBM	Java
Concordia [28]	Mitsubishi	Java

Table 4: Mobile agent tools developed by famous names in software industry

and these tools are also offered by software companies specialized in Object Oriented technologies like:

Application	**Producer**	**Language**
Voyager [29]	Object Space	Java
Odyssey [30]	General Magic Inc.	Java

Table 5: Mobile agent tools developed by OO specialized companies

As we can see in the Table 4 and Table 5 all of the four examples of Mobile Agent Applications are based on Java language [31]. This fact did not happen by chance, but this is a demonstration that Java is powerful enough in order to attend all Mobile Agent requirements. Java allows programmers to deal with objects, security, object serialization [21], network and remote computing facilities besides being platform independent. It has been used to develop applications for different kind of business and also is really used in the academic field. Java is also responsible for the most part of the web applications running nowadays in the internet. All those characteristics were fundamental to make Java the preferred language for developing Mobile Agent Applications and Tools.

In the practical part of our study we used Voyager [29] in order to implement a small example of mobile agent based application. Besides allowing mobile agents, Voyager has many other features that are not related with our study, so we would like to emphasize here that only the mobile agent feature of Voyager was used.

2.2 The Example Created

The purpose of this example was to introduce us to the mobile agent programming and try to understand the functionality of an agent-based application by observing the mobility, communication and behavior of a mobile agent.

Using Java 1.1.7 [31] as the programming language and Voyager 3.0 [29] as our mobile agent software application we simulated a really simple e-commerce like application that has three computer shops running in different hosts and one mobile agent, called "Shopping Agent", that will work in this environment. The task of our mobile agent here in this application is to travel around those three hosts and get the computer price from each one of the computer shops. Ending the travel the agent returns to the host were it was started and shows the smallest price found. Figure 4 represents the scenario of our application, let's take a look on it in order to understand better our example:

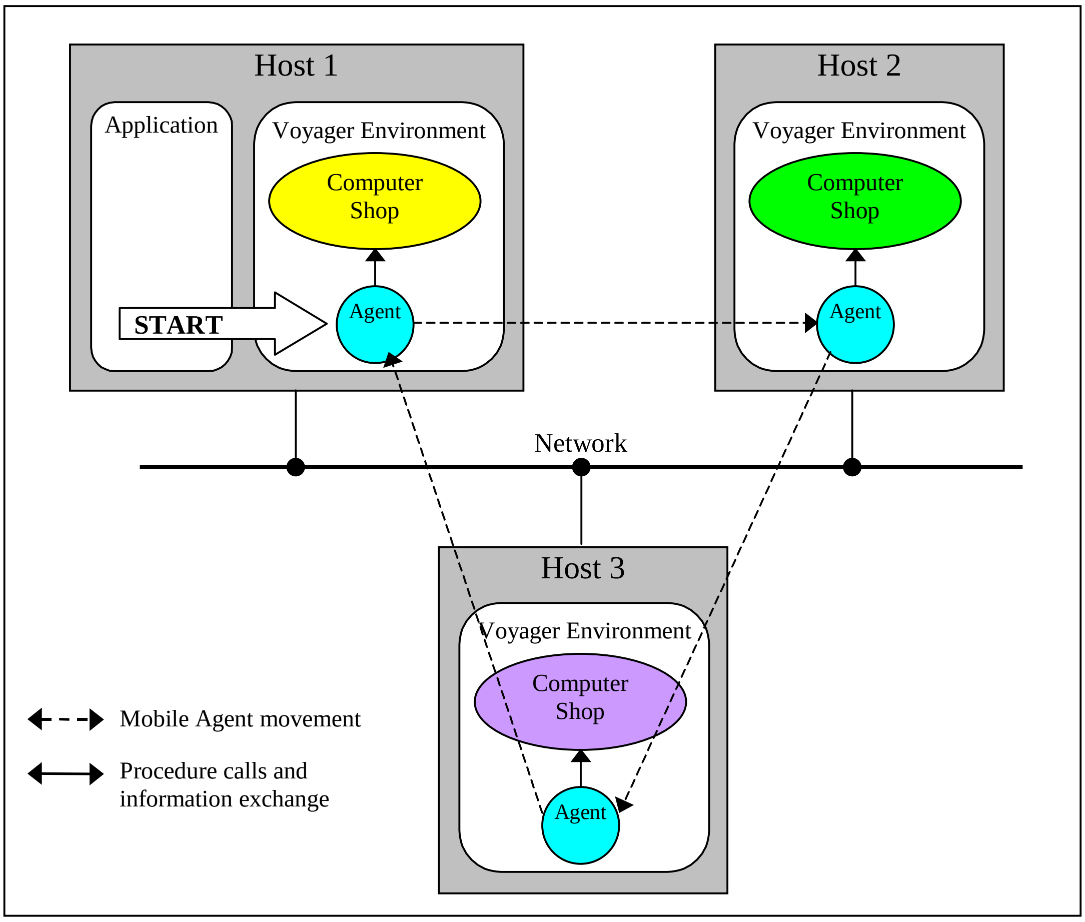

Figure 4: The simple e-commerce example

Looking at Figure 4 we can see our application starting an agent in a Voyager [29] environment. The agent has a previous knowledge of the address of the other environments and the computer shop applications running on them.

At each host the agent will interact with the computer shop accessing it through a pre-defined interface and will invoke a method in order to get the price of the computer. For reason of practicing mobile agent programming only, let's say here that each of the computer shops sells only one configuration of computer and this configuration is suitable for us.

2.3 Description of the Participants

Continuing with the description of our example let's talk a little bit deeper in a detailed way about the participants we can find and their roles.

The following UML [32] class diagram shows us the main attributes and methods of each class existing in the example.

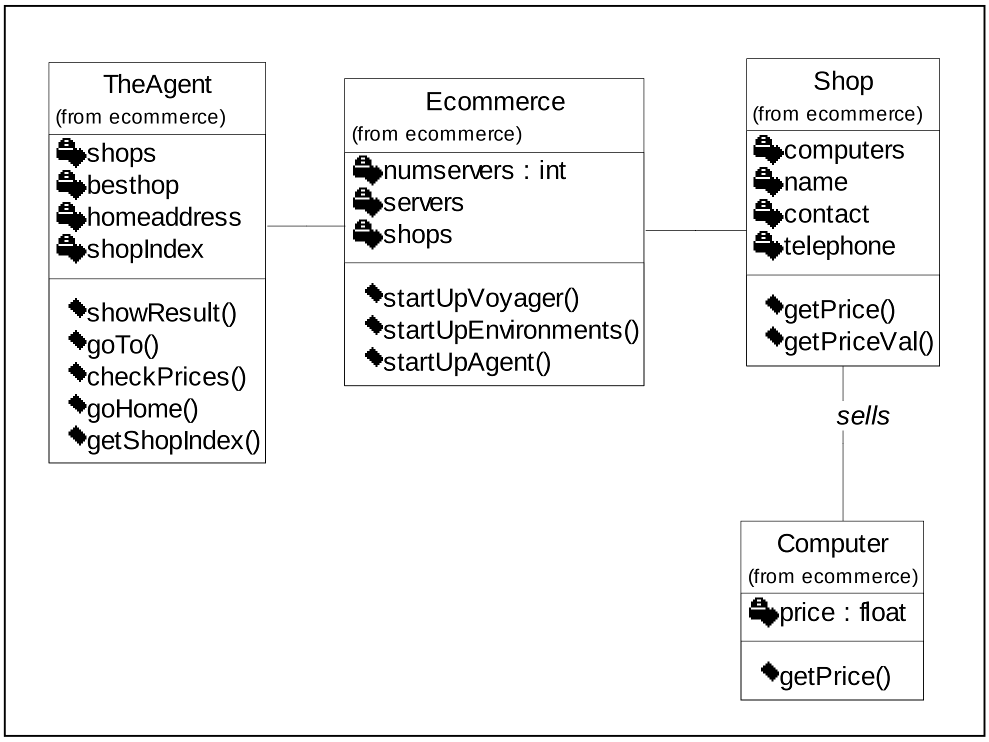

Figure 5: Class Diagram of the e-commerce example

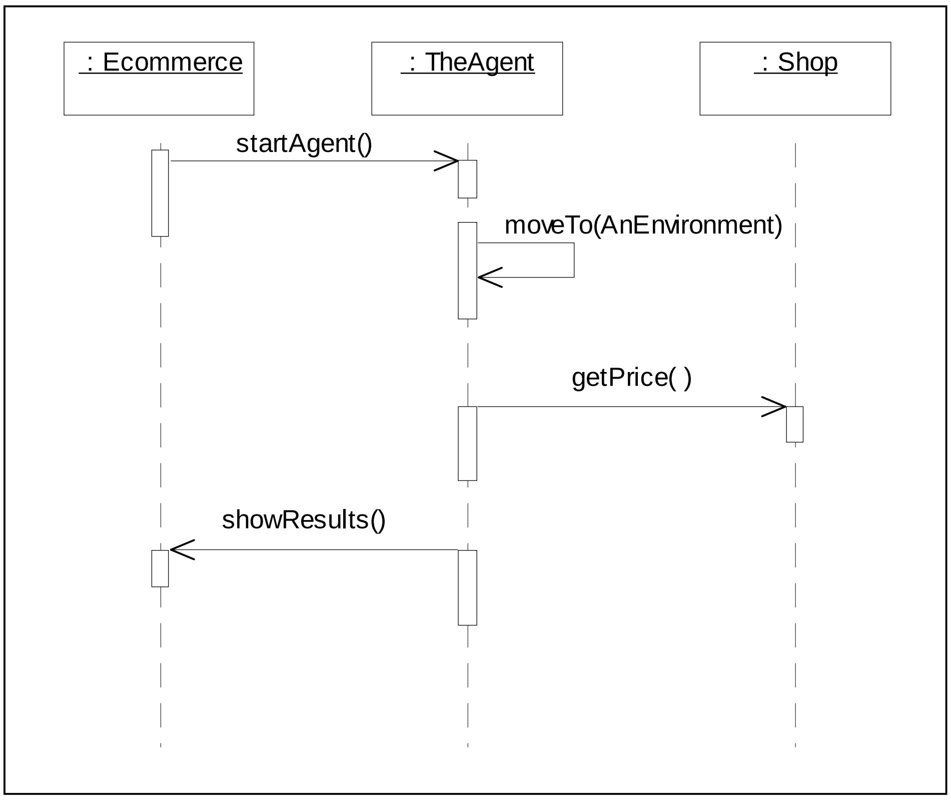

Figure 6: Collaboration diagram of the e-commerce example

2.3.1 TheAgent

The `TheAgent` class is our mobile agent in this example. It implements an interface with the method `goTo()` that will be called once by the application that creates an instance of the agent. Mobile agents can work autonomously [1], [14], so the method `goTo()` will be invoked by the agent itself when it wants to move to another location.

Method	Functionality
`goTo()`	Makes the agent move to another environment
`checkPrices()`	Interact with the `Shop` class in order to obtain the price of a computer.
`goHome()`	Makes the agent go back to the environment where it was started
`showResult()`	Show the result obtained after finishing its travel

Implementation of those methods:

```
import com.objectspace.voyager.*;
import com.objectspace.voyager.agent.*;
import com.objectspace.lib.util.*;
import java.io.*;

public class TheAgent implements ITheAgent, Serializable {
   private Vector shops;
   private int shopindex;
   private IShop bestshop = null;
   private String homeaddress = "//us10:10000";
```

```
/******************************/
public TheAgent(Vector s) {
/******************************/
   shopindex = 0;
   shops = new Vector();
   shops = s;
}

/******************************/
public void checkPrices(IShop shop)  {
/******************************/
   Date date = new Date();
   System.out.println("------------------------"+
                    date.toString());
   System.out.println("-> agent arrived");
   System.out.println("-> checking prices");

   if(bestshop == null) {
      bestshop = shop;
   }else if (shop.getPriceVal() < bestshop.getPriceVal()) {
      bestshop = shop;
   }
```

Compares the prices

```
   try {Thread.sleep(5000);}catch(Exception e){}
   System.out.println("-> price: "+ shop.getPrice());
   shopindex++;

   if (shops.size() > shopindex) {
      goTo(shopindex);
   }else {
      goHome();
   }
}
```

Goes to the next shop or get back home

```
/******************************/
public void goTo(int ind) {
/******************************/
   Date date = new Date();
   try {
      System.out.println("-> moving to next shop");
      System.out.println("------------------------"+
                       date.toString());
      Agent.of(this).moveTo((IShop)shops.elementAt(ind),
                         "checkPrices");
   } catch (Exception e) {
     System.err.println(e);
   }
}
```

Invokes the Voyager's environment method in order to move the agent to another location

Defines which method should be executed when arriving in a new host.

```
/******************************/
public void goHome() {
/******************************/
   boolean moved = false;
   try {
      System.out.print("-> going home ("+
                    Agent.of(this).getHome()+")");
      Agent.of(this).moveTo(Agent.of(this).getHome(),
                    "showResult");
   } catch (Exception e) {
      System.err.println(e);
   }
}
```

Again, invokes the Voyager's environment method `moveTo` . Now to go home.

```
/******************************/
public void showResult() {
/******************************/
   Date date = new Date();
```

```
      System.out.println("****************************"+
                         date.toString());
      System.out.println("Best price found: " +bestshop.getPrice());
      System.out.println("****************************");
   }

}
```

Uses the shop's interface to get the price and show it.

2.3.2 Shop

The class `Shop` will have an instance of the class Computer, where the price will be stored. This class keeps a list of all computers available in the computer shop. In our example, we defined that each shop will sell only one kind of computer. To check the price of the computer we invoke the method `getPrice()` and `getPriceVal()`. The first returns a currency formatted string and the second a float value.

Method	Functionality
`getName()`	Returns the name of the shop
`getContact()`	Returns the personnel contact in the shop
`getTelephone()`	Returns the shop's telephone number
`getPriceVal()`	Returns the price of a computer in a value (float) format
`getPrice()`	Returns the price of a computer in a currency formatted string

Implementation of those methods:

```
package ecommerce;

import java.io.*;
import java.util.*;
import java.text.*;
import com.objectspace.voyager.space.*;
import com.objectspace.voyager.*;


public class Shop implements IShop,Serializable {

   private String name;
   private String contact;
   private String telephone;
   private Vector computers;
   private Date date;

   /******************************/
   public Shop(int i) {
   /******************************/
      date = new Date();
      computers = new Vector();
      computers.addElement(new Computer());
      System.out.println("======================="+date.toString());
      System.out.println("-> object SHOP"+i+" created");
   }

   /******************************/
   public String getName() {
   /******************************/

      return name;
   }
   /*******************************/
   public String getContact() {
   /*******************************/
      return contact;
   }

   /*******************************/
   public String getTelephone() {
   /*******************************/
      return telephone;
   }

   /*******************************/
   public float getPriceVal() {
   /*******************************/
      return ((Computer) computers.elementAt(0)).getPrice();
   }

   /*******************************/
   public String getPrice() {
   /*******************************/
      float p = ((Computer) computers.elementAt(0)).getPrice();
      String ret = NumberFormat.getCurrencyInstance(
                                          Locale.FRENCH).format(p);
      return ret;
   }
```

2.3.3 Computer

The `Computer` class will contain and provide the information about the computers sold in a computer shop. It will interact directly with the Shop class when a price of a computer is demanded. The method `getPrice()` is the responsible to return that information. In our example the price and the description of a computer is generated randomly at the time of creation of an instance.

Method	**Functionality**
`getDescription()`	Returns the description of the computer
`getPrice()`	Returns the price of the computer

Implementation of those methods:

```
package ecommerce;

import java.io.*;
import java.util.*;

public class Computer {

   private String description;
   private float price;
   private Random num;


   /*******************************/
   public Computer() {
   /*******************************/
      num = new Random();
      price = (num.nextFloat())*10000;
      description = "COMPUTER - " + price;
   }
   /******************************/
   public String getDescription() {
   /******************************/
      return description;
   }

   /******************************/
   public float getPrice() {
   /******************************/
      return price;
   }

}
```

2.3.4 Ecommerce

The `Ecommerce` class has also a certain importance for our example because it plays to distinct roles. In the first role it is the responsible for starting up our environment, creating the shops in the different Voyager environments, and creates the mobile agent that will travel through the servers searching for computer prices. In the second role it plays the "Mobile agent Based Application", that will start our agent and get its results in the end. When developing agent-based applications in the real world we should have different implementations for each of those roles. Is always good to have a small application that will manage the environments and its members (agents and other objects), the final applications that will benefit from the use of mobile agents should only take care of their own started agents.

Implementation of this class:

```
package ecommerce;

import com.objectspace.voyager.*;
import com.objectspace.voyager.agent.*;
import com.objectspace.voyager.space.*;
import java.util.*;

public class Ecommerce {

   private int numservers;
   private Vector shops;
   private Vector servers;
   private Date date;

   /******************************/
   private Ecommerce() {
   /******************************/
      date = new Date();
      shops = new Vector();
      servers = new Vector();
      servers.addElement("//us10:8000");
      servers.addElement("//us20:8000");
      servers.addElement("//us30:8000");
      numservers = servers.size();
   }
```

Defines the hosts where the agent will travel

Starts up a local Voyager

```
/*******************************/
private void startupVoyager() {
/*******************************/
   System.out.println("************************"+date.toString());
   System.out.println("-> starting up Voyager ");
   try {
      Voyager.startup("9000");
      System.out.println(Agent.of(this).getHome());
   } catch (Exception e) {
    System.out.println(e);
   }

}


/*******************************/
private void startupEnvironment(){
/*******************************/
   System.out.println("-> inicializing environment");

   for(int i=0; i < numservers; i++) {

    try {

       // creates the object SHOPS and bind them to a name and
       // store them in a Vector
         System.out.println("-> creating object SHOP at "+
                                 (String) servers.elementAt(i));
       Object[] args = new Object[] {new Integer(i)};
       IShop shop = (IShop) Factory.create("ecommerce.Shop",
                            args, (String) servers.elementAt(i));
       shops.addElement(shop);

    } catch (Exception e) {
       System.out.println(e);
    }
   }
}
```

Creates the Shops objects in each one of the servers and put them in a vector

Creates the Agent

Sends the agent to the first Voyager Environment

```
/*******************************/
private void startupAgent() {
/*******************************/
   Date date = new Date();

   System.out.println("-> creating object AGENT");
   TheAgent anAgent = new TheAgent(getShops());
   System.out.println("-> putting AGENT to work");

   anAgent.goTo(anAgent.getShopIndex());
}


/*******************************/
private void shutdown() {
/*******************************/
   Date date = new Date();
   System.out.println("-> shutting down Voyager");
   System.out.println("************************"+date.toString());
   Voyager.shutdown();
}


/*******************************/
private Vector getShops() {
/*******************************/
   return shops;
}
```

```
   /******************************/
   public static void main( String[] args ) {
   /******************************/
      Ecommerce ecom = new Ecommerce();
      ecom.startupVoyager();
      ecom.startupEnvironment();
      ecom.startupAgent();
      ecom.shutdown();

   }
}
```

The method `Main()`, where everything starts

2.4 Conclusion

After this experimentation with a small example of an agent-based system, using Voyager 3.0 [29] and Java [31], we could feel how the main mechanisms and concepts in the "Agent Paradigm" works. From this example we could identify some basic requirements in agent-based systems, such as the environments, the agents themselves, the interaction that exists between agents and applications or even between two agents, their mobility, etc. All of this can prove that researches and studies done by the software industry, universities research groups and other laboratories are absolutely right when they say that the "Agent Paradigm" is a new and modern way to design and implement distributed systems.

From now, we can also say that a top-down study of the whole structure of an agent-based system beginning from the macro and going to each specific component is necessary. With such a kind of study, we would be able to identify the points of complexity where we could use advanced software engineering techniques in order to propose a solution. That solution would be showed in an abstract format, with a description of the generic problem in such way that it could help programmers to identify and solve similar problems of implementation involving the same characteristics.

A good way to do that is proposing Mobile Agent Patterns that could be applied when developing mobile agent-based applications in order to facilitate the comprehension, implementation and maintenance of those systems and also allowing them to be more reusable and robust.

Chapter II: Design Patterns for Mobile Agents

3 Introduction

"In most areas of research there comes a time when the researchers begin to understand the principles, facts, fundamental concepts, techniques, architectures, and other research elements in their fields of study". (Dwight Deugo) [4].

After studying and practicing the main "Mobile Agent" elements like autonomy, social ability, reactivity, pro-activity and mobility, we could reach a certain level of knowledge related to those properties and also identify where were their main recurrent problems. In the next step of our work, based on what we have learned about mobile agents, we propose a possible solution for each one of the identified problems and also we document them using a well known and accepted technique in the object oriented community: *software design patterns* [18],[19].

3.1 Design Patterns

The term "Design Pattern" was created by Christopher Alexander [20] a long time ago when he explained a pattern with the following statement:
"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". [20]

As an architect, Alexander talked about patterns related to buildings and towns, but what he said is completely true when we apply that to object oriented design in computer science. The terms differ from house architecture and software development, but the main idea of using design patterns remains the same: pattern is a solution to a problem in a certain context. Instead of talking about doors and walls we focus on objects and interfaces.

3.2 Scope of Our Solution

The mobile agent technology offers distinct levels of granularity that are not explicit when we look at it from the outside. Going a little bit deeper, we can identify many different recurrent problems that need to be solved at different levels, like for example:

- The mobility of the agent that involves sending its state and code to another environment in order to resume its execution.

- Allowing the interaction between agents its not an easy task for several reasons. Agents might not now other agents so someone should help agents to find other agents. After an agent found other agents how can they interact and exchange information ?

- The coordination of agents in an environment must be done carefully. The environment must manage the arriving and dispatching of agents, besides providing good ways of finding an agent once it knows all current agents.

- An agent based application must be secure. Not every agent can access other agent's specific methods. An identification of all agents and their intentions when trying to access other agent could be a good idea.

- Sending an agent object from one host to another requires some steps before. One of those steps is the serialization of the agent object. There are many different ways to do that and we can not forget that an agent based application must be heterogeneous and platform independent.

- When we talk about mobility we have to pay a special attention to the agent thread. When an agent is dispatched from one environment to another we must resume the agent operation from the same point where it was in the previous environment.

Some of those problems won't be covered here because they are really complex and demand specific and dedicated studies or because other authors have already solved them. This is the case of *threads* (concurrent programming) for example that is covered by the Active Object Pattern [8], [6] where the authors describe how we can manage different threads of control for agents.

The scope of our work is headed to provide solutions to high level recurrent problems, leaving the low level problems to be solved by using design patterns already proposed by other authors. Our solution will help other people to solve the following recurrent problems:

- How can we have mobile agents interacting with each other ?
- How can we control the access to the mobile agent ?
- How can we have a transparent access to a mobile agent when it is in a remote environment ?
- How can we identify mobile agents by their capabilities ?
- How can a mobile agent in a local environment find another agents without knowing their existence ?
- How can a mobile agent change the destinations where it should go dynamically ?

As solution to those commons problems found in mobile agent technology, we propose the following agent design patterns:

- Abstract Agent Pattern
- Agent Proxy Pattern
- Agent Coordinator Pattern
- Agent Interaction Pattern
- Travel Plan Pattern

In order to organize the patterns and give a general overview of the capabilities of each one we categorized them as shown in the following table:

Category	**Patterns**	**Deal with**
Agent	Abstract Agent	Environment / Mobility
	Agent Proxy	Mobility / Social Ability
Collaboration	Agent Coordinator	Social Ability
	Agent Interaction	Social Ability/ Goal Driven/ Reactive
Traveling	Travel Plan	Mobility

Table 6: Proposed agent design patterns

Each of those patterns will be discussed in the next sections of our work giving a complete explanation about their structures, participants, applicability, etc.

4 Abstract Agent Pattern

4.1 Intent

The intent of the Abstract Agent Pattern is to define a basic infrastructure in order to implement a mobile agent based system.

4.2 Scope and Motivation

We want to benefit from the various advantages offered by the mobile agent technology. The ease to design, implement and maintain an application developed using mobile agent technology led us to adopt it. In our mobile agent based application, our agents will be able to be started and after move around different hosts where an environment will be ready to receive them. In that environment our agent will be able to interact with other agents and exchange information by sending messages each other.

4.3 Applicability

The Abstract Agent Pattern is applicable when:

- The advantages provided by mobile agent technology [1] are need in a distributed application.
- Defining a basic and abstract infrastructure of a mobile agent based application.
- Developing a mobile agent based application just using Java [31].

4.4 Structure and Participants

Figure 7 shows the Abstract Agent Pattern represented by a generic UML [32] class diagram containing its main participants.

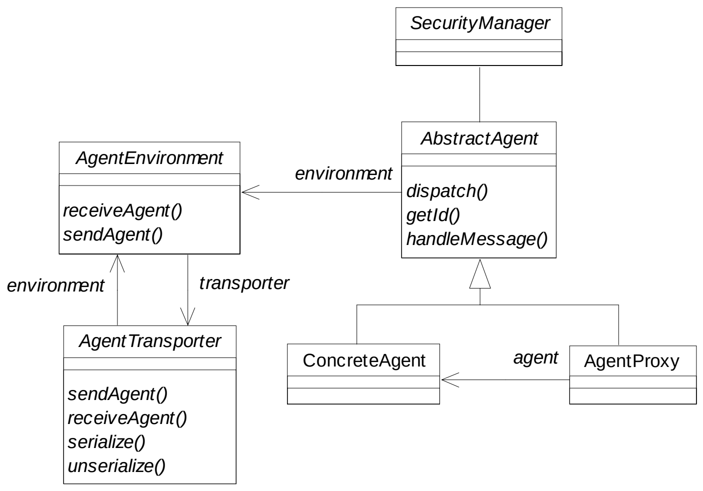

Figure 7: The abstract agent pattern structure

The participants are:

- **AgentEnvironment**

The `AgentEnvironment` class has a very important role in this pattern. It will be the responsible to provide to the mobile agent basic services that allow an agent to move to other environments, to find other agents, to interact with other agents, etc. The environment will also contains a reference to the `AgentTransporter` class, leaving the transport implementation separated of our environment behavior makes it easier to deal with different types of protocols and communication architectures.

- **AbstractAgent**

The main role of the `AbstractAgent` class is to define a common interface for the `AgentProxy` class and the `ConcreteAgent` class, so the proxy can be used anywhere as the real agent object.

- **ConcreteAgent**

The `ConcreteAgent` class will contain the implementation of the desired behavior of our agent and always will be represent by an `AgentProxy`.

- **AgentProxy**

The `AgentProxy` class will implement different behaviors in order to represent the `ConcreteAgent` object. Depending on the location of the agent, the `AgentProxy` can have two distinct states, the local state and the remote state. The local state will have a reference to a `ConcreteAgent` object in the local environment while the remote state will have a reference to an `AgentProxy` in a remote environment. This will allow us to have a transparent access to the agent independently where it is located. The agent proxy is also responsible to provide an interface that defines the methods that can be invoked in the concrete agent object. It always represents the `ConcreteAgent`.

- **AgentTransporter**

The `AgentTransporter` class will be responsible for take care of the mobile agent object transportation. It will implement methods to serialize and unserialize mobile agent objects [21] and also to send them over the network to another host.

- **SecurityManager**

This class will specify the agent access security policies and controls all messages that will be sent to the `ConcreteAgent` class through its respective `AgentProxy`.

4.5 Consequences

The Abstract Agent Pattern shows us the main components necessary in a mobile agent based application.

- This pattern can be implemented with no special tool or agent programming language, so this is one reason that will make its implementation platform independent.

- New features to the environment can be added without changing its main behavior. For example, the environment can provide a service of security checking that verifies whether the mobile agent is allowed or not to enter in the environment.

- The fact of having a separated object that is responsible only for the transportation of objects permits us to extend this class if needed in order to allow our environment to send and receive mobile agents using different ways of communications like RMI, CORBA, DCOM, sockets, etc.

4.6 Implementation

To implement the Abstract Agent pattern, carry out the following steps:

1. Identify the basic methods that a mobile agent must have and define those methods in an abstract class. The abstract class can be used as an interface defining all methods that should exist in the concrete agent and in the agent proxy.

2. Extend that abstract class and create a concrete agent and an agent proxy.

3. In the concrete agent implement the methods defined in the abstract class and the other desired methods.

4. In the agent proxy implement the methods only calling the same methods in the concrete agent. For example:

```
Public class AgentProxy
   Concrete agent agent;

   public getId() {
         agent.getId();
   }
}
```

   Before invoking the methods in the concrete agent an access control can be done. A security manager is helpful to do that.

5. Create an abstract class for your environment. The environment will be responsible to implement the methods that will deal with the agents transportation (dispatch and receive), so we can use also a agent transport that will carry out the serialization of the agent object and its sending.

4.7 Related Patterns

Agent Pattern [7]

5 Agent Proxy Pattern

5.1 Intent

The intent of this pattern is to define a mobile agent capable to run in any environment allowing other agents to have transparent access to it, through a proxy, even when the agent is remote.

5.2 Scope and Motivation

Mobile agents can work autonomously and asynchronously in places called environments. The environments can be located in a same host or they can be spread over different hosts on the network. When traveling, the agent must allow other agents or applications to access them without having to announce its location to all its collaborators. Other agents can not invoke the methods on the mobile agent object directly, only through its proxy. The agent's proxy provides an interface with the methods that can be invoked by others mobile agents or applications.

5.3 Applicability

The Agent Proxy Pattern is applicable when:

- Developing mobile agent applications.
- Access to agents need to be transparent independently where the agent is located (local or remote).
- The real agent object cannot be accessed directly by its collaborators.
- An agent surrogate is necessary in order to provide an interface to the concrete agent object.

5.4 Structure and Participants

Figure 8 shows the Mobile Agent Pattern represented by a generic class diagram containing its main participants.

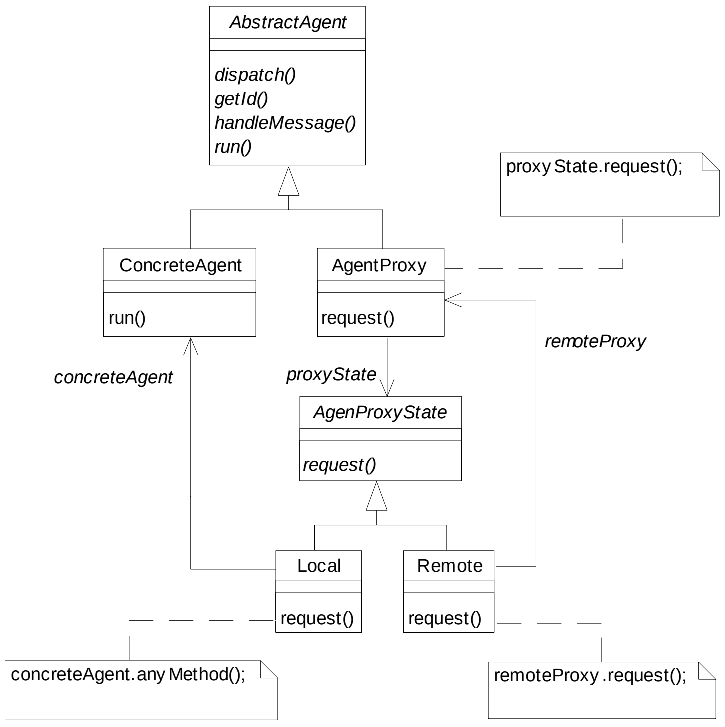

Figure 8: The agent proxy pattern structure

The participants in this pattern are:

- **AbstractAgent**

The main role of the `AbstractAgent` class is to define a common interface for the `AgentProxy` class and the `ConcreteAgent` class, so the proxy can be used anywhere as the real agent object.

- **AgentProxy**

The `AgentProxy` class will maintain a reference to its state class, `Local` or `Remote` depending on the location of the agent. All incoming requests done by any other object will be forwarded to the respective state class. The agent proxy is also responsible to provide an interface that defines the methods that can be invoked in the concrete agent object.

- **AgentProxyState**

This class defines the common interface that will be used by the `Local` and `Remote` classes so that the AgentProxy will be able to reach the real agent object wherever is the agent location.

- **Local**

The `Local` class will be responsible to keep a reference to the real agent object represented by the `ConcreteAgent` class when the agent object is located in the same environment as the proxy. This is the only class that can invoke the `ConcreteAgent` methods directly.

- **Remote**

The `Remote` class represents the state of the proxy when the real agent object is not in the same environment of its proxy. It will maintain a reference to another `AgentProxy` that is remote. Doing that we can guarantee that other objects will have transparent access to our agent object.

- **ConcreteAgent**

The `ConcreteAgent` is the real agent object. It will contain the implementation of the desired behavior and always will be represented by an `AgentProxy`.

5.5 Collaborations

Any collaborator can call agent methods through an `AgentProxy` by sending a message to it. The only class that is able to call agent methods directly is the `Local` class, it represents the `AgentProxy` state when the `ConcreteAgent` object and its respective `AgentProxy` object are in the same environment. Otherwise the `AgentProxy` state will be represented by the class Remote that will make a reference to another `AgentProxy` remotely.

Figure 9 shows the collaboration diagram for a local mobile agent.

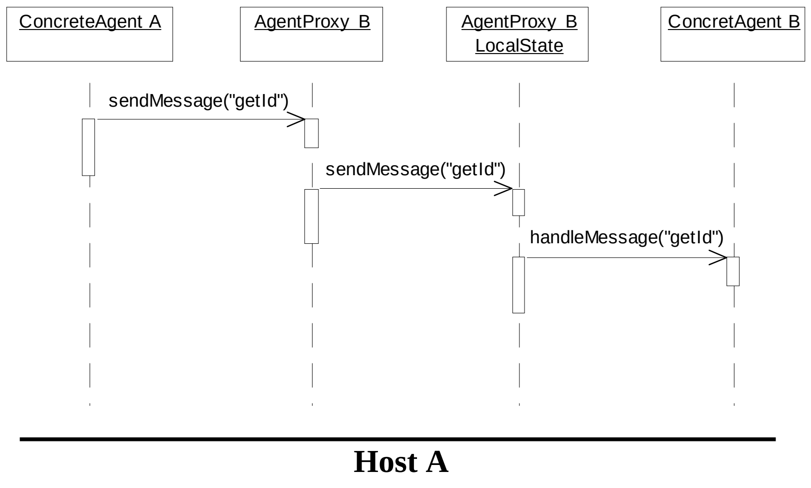

Figure 9: Collaboration diagram of a local mobile agent

Figure 10 shows the collaboration diagram for a remote mobile agent.

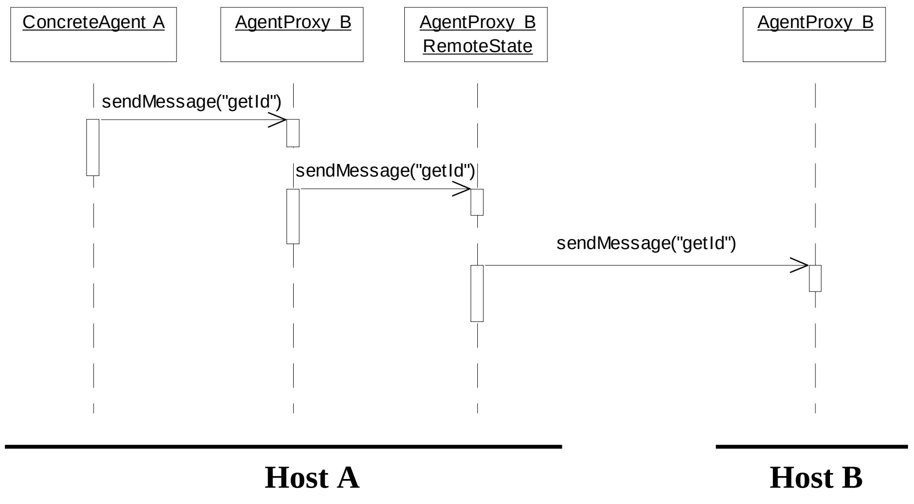

Figure 10: Collaboration diagram of a remote mobile agent

5.6 Consequences

The Agent Proxy Pattern introduces a simple and robust structure to implement a mobile agent.

- This pattern can be implemented with no special tool or agent programming language, so this is one reason that will make its implementation platform independent.

- It offers a good level of flexibility and can be easily improved if needed because it is based on the already proven design patterns – State Pattern [18] and Proxy Pattern [18], [19].

- New features based on the current structure can be added depending on the needs of who will implement it. For example, just implementing a Security Manager that first checks the rights of who wants to access the agent can provide us a more rigorous access control to the agent object.

5.7 Implementation

To implement the Agent Proxy pattern, carry out the following steps:

1. Reuse the already implemented `AbstractAgent` and `ConcreteAgent` classes defined in the Abstract Agent Pattern.

2. Create the `AgentProxy` class with a variable that will contain an instance of the `AgentProxyState` class. When we are working with mobile agents, agents and applications can interact with other agents that may move to a remote environment.

When an agent has moved to a remote environment the other agents and applications must be able to interact with it transparently even when the agent is located in a remote environment. So, the `AgentProxy` class must change its states accordingly to the location of the agent. Implement a method that updates the agent proxy state every time that the agent moves from environment to another.

3. Create an abstract class called `AgentProxyState` and define, as abstract, the same methods existing in the `AgentProxy` class.

4. Extend the `AgentProxyState` creating the `AgentProxyStateLocal` and `AgentProxyStateRemote` classes.

5. In the `AgentProxyStateLocal` class define a variable that references a concrete agent object and implement all methods defined in its super class by invoking the same methods in the concrete agent object that this states represents. For example:

```
public class AgentProxyStateLocal
   ConcreteAgent agent;

   public getId() {
         agent.getId();
   }
}
```

   Before invoking the methods in the concrete agent object an access control can be done. A security manager is helpful to do that.

6. In the `AgentProxyStateRemote` class define a variable that references an agent proxy in a remote environment. Implement all methods defined in the `AgentProxyState` class by invoking the same methods in the remote agent proxy. For example:

```
   public class AgentProxyStateRemote
      AgentProxy remoteProxy;

      public getId() {
            remoteProxy.getId();
      }
}
```

5.8 Related Patterns

Proxy [18], [19]
State [18]

6 Agent Coordinator Pattern

6.1 Intent

The intent of the Agent Coordinator is to allow the agent environment to provide a service to register incoming and outgoing mobile agents.

6.2 Scope and Motivation

When an agent arrives at an environment it may not know the other agents that are working in the same environment. At the same time, other agents want to know the all agents that are current working on that environment and interact with them. This would be possible if we force our mobile agents to register and unregister with a coordinator at the moment of their arrival or departure respectively, facilitating the control over all agents that are located at an environment. Once the agent coordinator knows all agents running in its environment it can provide different methods of agent lookup.

6.3 Applicability

The Agent Coordinator pattern is applicable when:

- The agent environment must have control over all agents that are running on it.
- The interaction among mobile agents is necessary even when the agents do not know each other.
- Different agent lookup methods are necessary.

6.4 Structure and Participants

Figure 11 shows the Agent Coordinator Pattern represented by a generic class diagram containing its main participants.

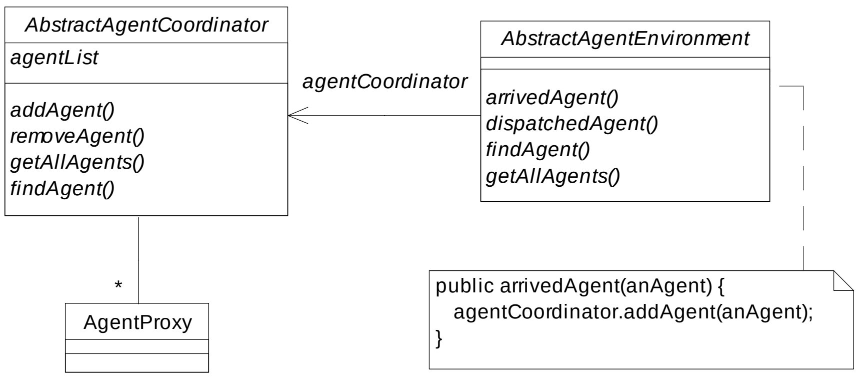

Figure 11: The agent coordinator pattern structure

The participants are:

- **AbstractAgentCoordinator**

The `AbstractAgentCoordinator` class will maintains a list of all current agents (proxies) running in a certain agent environment. It provides the basic methods of adding, removing and searching mobile agents. It can be easily extended in order to provide new methods for agent lookup.

- **AbstractAgentEnvironment**

The `AbstractAgentEnvironment` class keeps a reference to the `AgentCoordinator` class. After a mobile agent has arrived or departed, the environment will call the agent coordinator methods in order to update the list of current running agents. This class is also responsible to provide an interface that allows agents to perform searches on the agent coordinator.

- **AgentProxy**

The `AgentProxy` class is the surrogate of the real agent object (see item 5 – Agent Proxy Pattern) and will be referenced by the `AgentCoordinator` list.

6.5 Collaborations

Figure 12 shows the collaboration diagram of the Agent Coordinator Pattern.

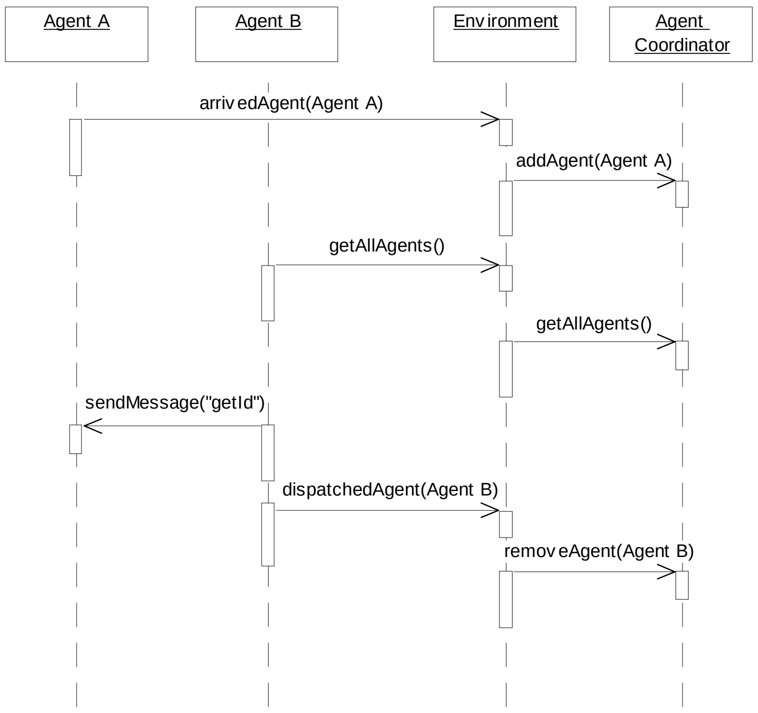

Figure 12: Collaboration diagram of the Agent Coordinator Pattern

6.6 Consequences

The Agent Coordinator Pattern is a good solution when mobile agents have limited knowledge of other agents or when they really do not know other agents. All agents are forced to register when arriving and leaving of an environment allowing the environment to have a whole control of the current running agents.

- The agents do not need to have previous knowledge of other agents in order to interact with them.
- We decrease the complexity of the mobile agent and avoid high maintenance when the agent coordinator implements the lookup methods.
- Not only mobile agents can benefit from the agent coordinator lookup methods; stationary agents, other applications like web applications, viewers and environment managing tools are able to use them.

6.7 Implementation

To implement the Agent Coordinator Pattern carry out the following steps:

1. Reuse the already defined `AbstractAgentEnvironment` class in the Abstract Agent Pattern. Add new methods to this class that will allow controlling all agents that arrive and depart from the environment. Add also a variable that will contain an instance of the agent coordinator.

2. Create an abstract class `AbstractAgentCoordinator` with a variable that can keep a list of the current agents in the environment.

3. Implement basic methods to manipulate that list. For example: `addAgent()`, `removeAgent()`, `findAgent()`.

4. Every time an agent arrives or departs from the environment invoke the correspondent methods in order to keep the list always updated.

6.8 Related Patterns

AgentProxy (see item 5 – Agent Proxy Pattern)
Mediator [18]
Iterator [18]

7 Agent Interaction Pattern

7.1 Intent

The Agent Interaction Pattern intents to facilitate the interaction between agents based on their *capabilities*.

7.2 Scope and Motivation

A mobile agent can interact with other different agents even when those agents are completely different on their behaviors. For example, an agent can interact with an agent that represents a bookstore and after it interacts with another agent that represents a shipping company. Our mobile agent will by a book from the bookstore agent and asks to the shipping company agent to delivery it to a certain address. Our agent knows neither the bookstore agent nor the shipping company agent. If we associate *capabilities* to each one of those agents it would be much simpler to find them. Moreover, if we define in the *capabilities* what our agent is looking for, the environment can notify our mobile agent when an agent that matches our needs arrives.

7.3 Applicability

The Agent Interaction Pattern can be applied when:

- A mobile agent must interact with other agents that it doesn't know, but it knows the capability that the other agents must have.
- The environment should notify all agents about the capabilities of a mobile agent that arrived.
- When an agent wants to be announced to other agents about its capabilities when arriving in any environment.

7.4 Structure and Participants

Figure 13 shows the Agent Interaction Pattern represented by a generic class diagram containing its main participants.

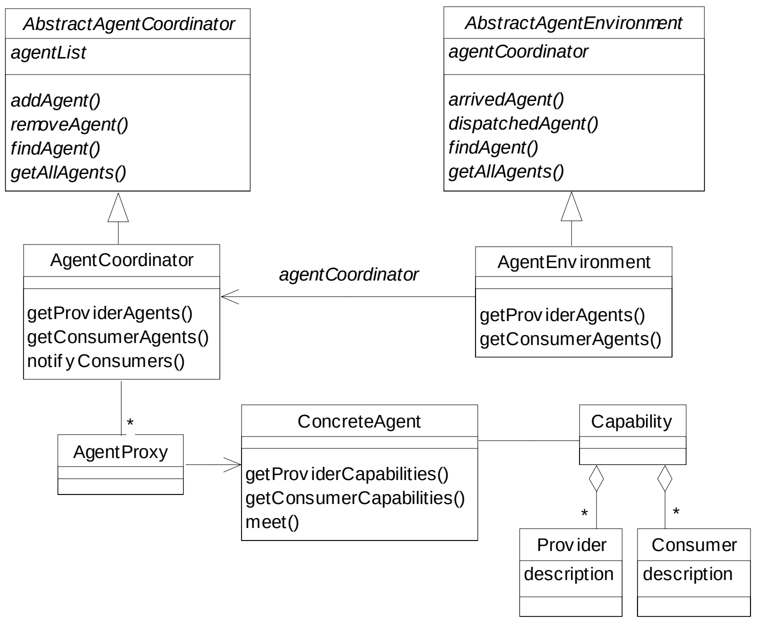

Figure 13: The agent interaction pattern structure

The participants are:

- **AbstractAgentCoordinator**

The `AbstractAgentCoordinator` class implements the basic functionality of the agent coordinator, managing a list of all agents running in the current environment (see item 6 – Agent Coordinator Pattern for).

- **AbstractAgentEnvironment**

The `AbstractAgentEnvironment` class keeps a reference to the `AgentCoordinator` class. After a mobile agent has arrived or departed, the environment will call the agent coordinator methods in order to update the list of current running agents. This class is also responsible to provide an interface that allows agents to perform searches on the agent coordinator.

- **AgentCoordinator**

This class extends the `AbstractAgentCoordinator` class. It implements new lookup methods that can find mobile agents based on their capabilities. It takes in account the provider and consumer capabilities of each mobile agent during the searches. It also can notify agents about the arriving of other agents. The notification is based on the provider capabilities of the arrived agent and the consumer capabilities of the current agents in the environment.

- **AgentEnvironment**

The `AgentEnvironment` class extends the `AbstractAgentEnvironment` and implements new methods for agent lookup. For example, the method `getProviderAgents()` returns only the mobile agents that has capabilities of providers. A parameter could be defined in this method in order to specify the provider capability we are searching, for example `getProviderAgents("shipping company")`could return only the agents that represent companies which provide services for shipping merchandises.

- **AgentProxy**

The `AgentProxy` class is the surrogate of the real agent object (see item 5 – Agent Proxy Pattern) and will be referenced by the `AgentCoordinator` list.

- **ConcreteAgent**

The `ConcreteAgent` is the real agent object. It will contain the implementation of the desired behavior and always will be represented by an `AgentProxy`. The `ConcreteAgent` in this pattern implements two methods that return its capabilities as provider and as consumer respectively.

- **Capability**

The `Capability` class is where we define the agent's capabilities. It is composed of instances of classes `Provider` and `Consumer`. A mobile agent can have several capabilities as provider or as consumer. The mobile agent might also have only provider capabilities or even only consumer capabilities depending on how it was implemented. Depending on the agent policy it might also define no capability at all, then our agent won't be notified by the environment nor be found by other agents. This could be useful when implementing a security agent that should work anonymously to investigate and monitor other agents.

- **Provider**

This class has a description of one provider capability of a mobile agent. The `Provider` class can describes the capability in different ways, depending on the needs of who will use it. For example it can only have a string variable containing: "books" in order to identify an agent that sells books. In our implementation of this pattern (see Appendix A – Java Documentation) we used the string "computer" to identify a computer shop that sells computers.

- **Consumer**

This class has a description of one consumer capability of a mobile agent. The `Consumer` class can also describes the capability in different ways. In our example we defined a standard that a consumer agent contains in its capability the description of the provider it is looking for. So, if we define "books" as the capability of a consumer, it means that the agent is a consumer of "books" and will look for other agents that have a provider capability "books".

7.5 Collaborations

Figure 14 shows the collaboration diagram of the Agent Interaction Pattern.

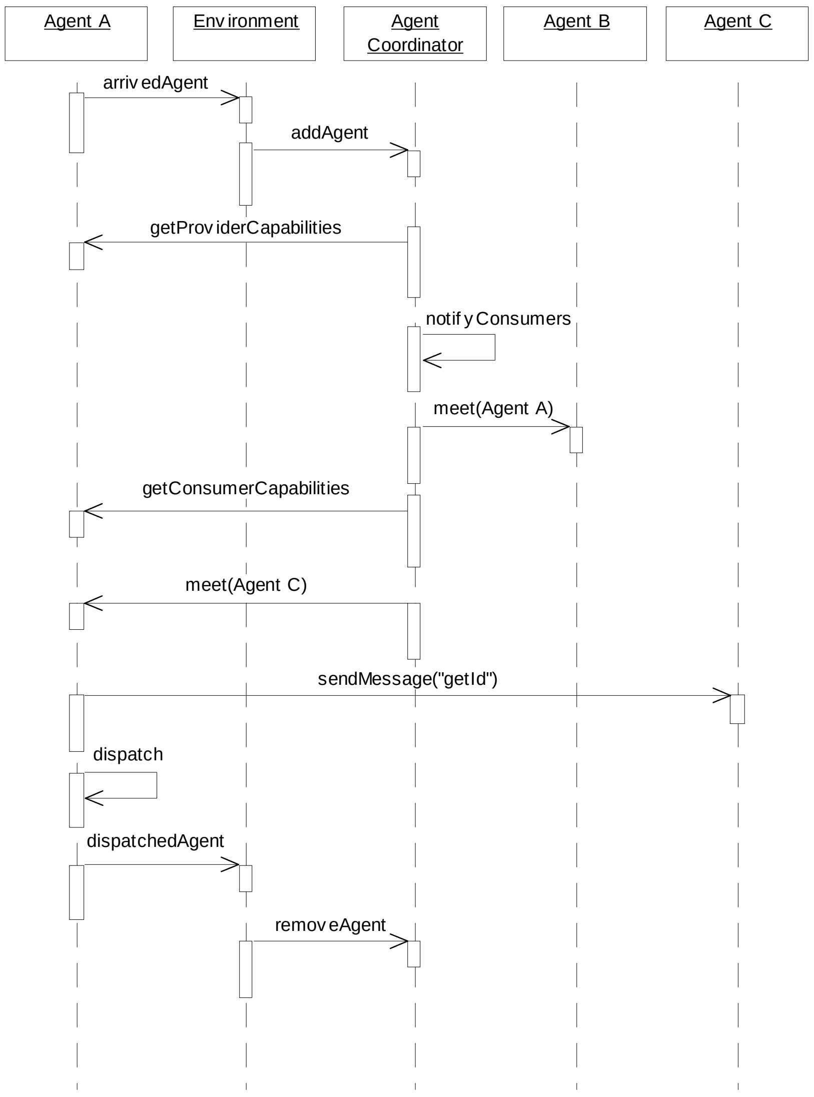

Figure 14: Collaboration diagram of the Agent Interaction Pattern

7.6 Consequences

The Agent Interaction Pattern is helpful when mobile agents have limited knowledge of other agents or when they really do not know other agents and they must find other agents based on the their capabilities.

- The agents do not need to have previous knowledge of other agents in order to interact with them.
- The Agent Interaction Pattern benefit from the Agent Coordinator Pattern (see item 6 – Agent Coordinator Pattern) in order to have the basic registration facility in the environment. In this case, it is only necessary to implement the specific lookup methods, which use the agent's capabilities as the parameters, in the `AgentCoordinator` class.
- Agents can be notified when other agents of interest have arrived.

- Agents that do not want to advertise their capabilities can work anonymously without being reached by other agents. However, they still able to find other agents in accordance with their needs.

7.7 Implementation

To implement the Agent Interaction Pattern carry out the following steps:

1. Reuse the `AbstractAgentCoordinator`, `AbstractAgentEnvironment` and `ConcreteAgent` classes defined in the other patterns.

2. Create a class `Capability` where the agent's capabilities will be described. Implement basic methods for adding, removing and obtaining the agent's capabilities.

3. Create a class `Provider` and a class `Consumer` that describes the provider and consumer capabilities respectively. Implement basic methods for obtaining the description of the capabilities.

4. Create a variable in the `ConcreteAgent` class that will make a reference to an object capability.

5. Extend the class `AbstractAgentCoordinator` creating a class `AgentCoordinator` and implementing specific methods to deal with the agent's capabilities.

6. Do the same of step 5 to the `AbstractAngentEnvironment`.

7. In the `AgentCoordinator` class implement wanted and personalized methods. For example: `notifyConsumers()`, that notifies all agents that has a specific consumer capability that an agent having a specific provider capability has arrived.

7.8 Related Patterns

Agent Coordinator (see item 6 – Agent Coordinator Pattern)
Agent Proxy (see item 5 – Agent Proxy Pattern)
Observer [18]
Meeting [2]

8 Travel Pattern

8.1 Intent

This pattern intents to provide to mobile agents the route for traveling among the different environments.

8.2 Scope and Motivation

Mobile agents are objects capable of moving autonomously through different hosts in order to perform their tasks in the existing environments. To do so, an agent should know the addresses of environments where it can go. Besides knowing the address where they are supposed to go, the agents would also take some actions related to each one of those address. For example, trying alternative hosts in case of some of the hosts were shut down, control the already visited hosts, etc. An agent can also add some new destinations in its travel plan during its travel by interacting with the visited environments. All those characteristics do not belong to the main goal of our agent, so they can be separated of its main behavior.

8.3 Applicability

The Travel Plan Pattern can be applied when:

- You want to define a travel plan for your mobile agent.
- The traveling procedures are going to be implemented separately of the main agent behavior, facilitating the maintenance and future improvements.
- The travel plan could be updated or changed during traveling by interacting with the visited environments.
- When controls over already visited destinations, alternative destinations, time out connections and maximum number of visits are necessary.

8.4 Structure and Participants

Figure 15 shows the Travel Pattern represented by a generic class diagram containing its main participants.

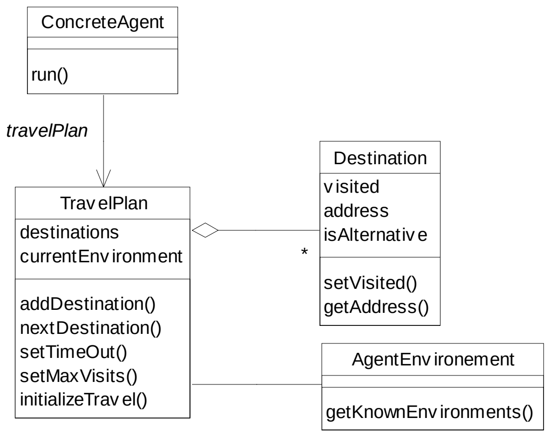

Figure 15: The travel plan pattern structure

The participants are:

- **TravelPlan**

The `TravelPlan` class will keep a list containing instances of the class `Destination`. Besides providing the necessary methods to manage the list of destination, the `TravelPlan` class will interact with the environment where its agent located. This interaction objectives to find out new destinations whose the agent does not knows and set those destinations as alternatives to the agents travel plan.

- **Destination**

This class represents the each one of the destinations that the mobile agent will visit. It contains the IP address of the hosts, and information whether it was already visited and whether the destination is an alternative one.

- **AgentEnvironment**

The `AgentEnvironment` class provides the basic functionality like transportation, searching, etc., in order to mobile agents work. (see items 4 – Abstract Agent Pattern, 6 – Agent Coordinator Pattern and 7 – Agent Interaction Pattern)

- **ConcreteAgent**

The `ConcreteAgent` is the real agent object. In this pattern it has a reference to the `TravelPlan` and contains the implementation of the desired behavior and always will be represented by an `AgentProxy`. (see item 5 – Agent Proxy Pattern).

8.5 Collaborations

Figure 16 shows the collaboration diagram of the Travel Plan Pattern.

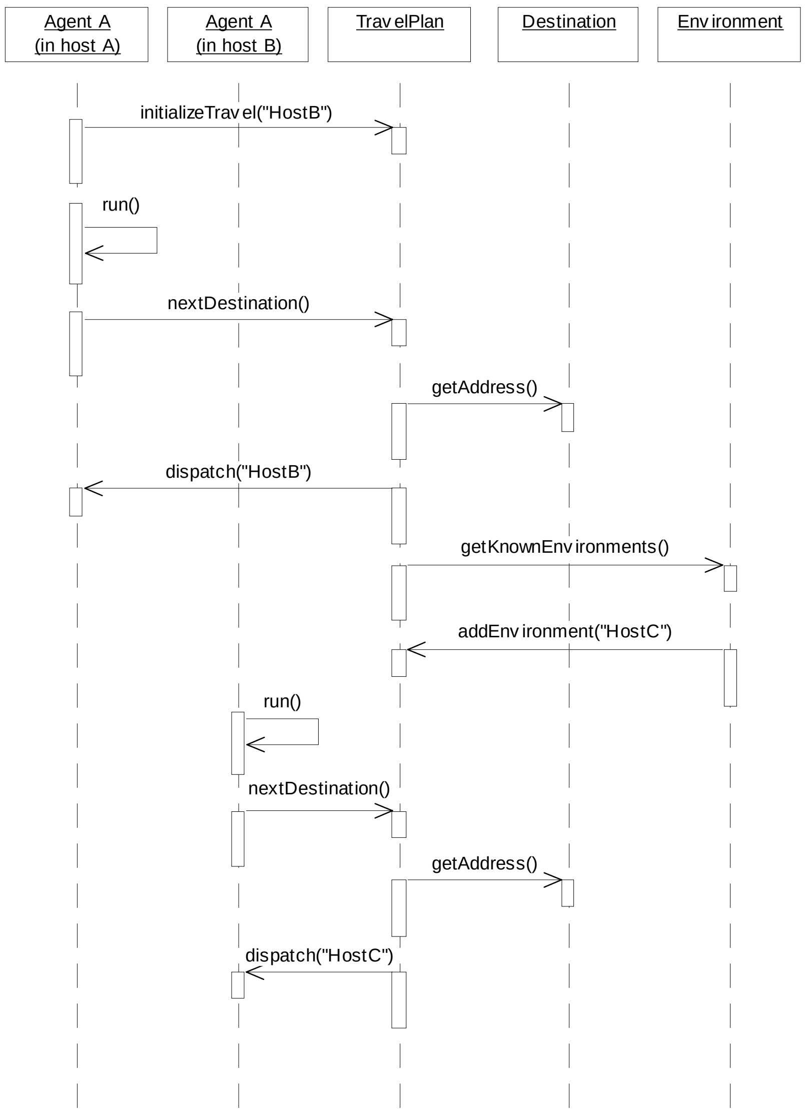

Figure 16: Collaboration diagram of the Travel Plan Pattern

8.6 Consequences

With the Travel Plan Pattern we allow our mobile agent to travel through several different destinations, even to destinations that our agent does not know.

- The mobile agents don't need to know many destinations, they will know new ones during their travel.
- The travel plan can be easily set up, defining time out value for communication and determining the maximum number of destinations to visit.
- When some destination fails to answer to our mobile agent, an alternative destination can be addressed to the agent automatically.
- All functionality about the travel itself is separated of the main behavior of the agent, facilitating in the maintenance and improvement.
- We can define a travel plan once and use it in several mobile agents.

8.7 Implementation

To implement the Travel Plan Pattern carry out the following steps:

1. Reuse the already defined `ConcreteAgent` class and create on it a new variable that will reference the travel plan object.

2. Create a `Destination` class containing basic variables to control the status of the destination object. For example: visited, isAlternative, etc. Implement also methods to deal with that variables (sets and gets).

3. Create a `TravelPlan` class having a list with destination objects that represents the places where the agent should go. Implement basic methods to manage that list and to deal with other destination variables. For example:

   `addDestination()`: to add new destinations to the travel plan.

   `removeDestination()`: to remove a destination from the travel plan.

   `nextDestination()`: to obtain the next destination where de agent must go.

   `getAddress()`: to get the address o a destination.

   `setTimeOut()`: to set how many seconds should wait before try another destination in case of not connecting.

   `setMaxVisits()`: to set the maximum number of visits during a travel. Once the travel plan can be changed dynamically the agent can have a travel that never ends, so its is important to define a number to limit the agent visits.

4. Implement in the `AgentEnvironment` class a new method that returns other environments addresses, so the agent can add new destinations to its travel plan during its travel. Also the environment can ask other environment address known by the agent.

8.8 Related Patterns

Itinerary [2]
Iterator [18]

9 Conclusions

“Agent Technology” offers a much easier way of designing, implementing and maintaining distributed systems. In our research we concluded that even providing that advantages, agent technology demands for an initial study and a practical experimentation of all its internal and not explicit features like autonomy, social ability, reactivity, pro-activity and mobility. That study can follow a top-down hierarchy beginning from the macro and going to each specific component in order to identify possible recurrent problems.

Using software engineering techniques such as “*Software Design Patterns*” [18], [19] we can design a solution to that problems, facilitating the comprehension, implementation and helping in future maintenance of the systems that will use such a solution. Complementary to that, other people can benefit from the same design patterns to solve their own similar problems about mobile agents without having to reinvent the wheel. The main consequence of all this is that our applications become more reusable and robust.

- **Contributions**

With “Agent Technology”, especially mobile agents, being the focus of our work, we showed one specific chapter where the main definitions, advantages, applicability, properties and elements of mobile agents are explained in order to give to the readers a general overview about it. The first experimentation with mobile agents, described by the example of an electronic commerce in the same chapter, allowed us to better understand the agents mechanisms and architecture.

Based on that we introduced different solutions for the recurrent problems that we believe anyone will face when implementing a mobile agent based application. The five design patterns proposed: **Abstract Agent Pattern**, **Agent Proxy Pattern**, **Agent Coordinator Pattern**, **Agent Interaction Pattern** and **Travel Plan Pattern**, allow the mobile agent programmer to implement basic concepts of agent technology by just using a common programming language like Java [31]. Common properties like mobility, autonomy, social ability, etc. are covered by our patterns, that present a abstract view of each related problem in order to make them as much reusable as possible.

After proposing the five design patterns we used them to implement the same example described in section 2 were we experimented to create a small application using Java [31] and Voyager [29]. The documentation of the implementation can be found in the Appendix A – Java Documentation. Besides having a running example constructed with our own proposed design patterns using only Java RMI and proving that they really work, another interesting result was the productivity when using our design patterns. In our first experimentation we spent almost four times more the time we spent implementing the same application using our design patterns. Even considering that in the first experimentation with agents we did not have the same experience that we have now, we believe that our design patterns really reached their goals.

- **Future Work**

Of course there are a lot of work to do yet. For future researches we can see that new patterns can be created involving the high complexity mechanisms in mobile agents, like threads for example. Other patterns describing the *Life Cicle* of the mobile agent, *Organized Groups* of mobile agents where they can share different tasks in order to accomplish with a final goal are also possible recurrent problems to be solved using design patterns.

Other research can be investigated by joining reflection and agent technology. Reflection is the process of reasoning about and acting upon itself [22], [23]. According to J. Des Rivieres, "*a reflective language is a language allowing you to deal with explicit representations of implicit aspects of the language itself*" [24].

A reflective language may have a representation of its own behavior dealing with message sending, encapsulation, execution and so on. Then, reflection allows deriving new behaviors from initial ones by introducing some variations of the computational model, for example:

- asynchronous message vs. synchronous message
- remote invocation vs. local invocation
- multiple inheritance vs. simple inheritance
- etc.

Moreover, reflection allows us to separate what an object does (the base level) from how it does it (its meta level) [25]. A reflective language encourages a clean separation between the basic functionality of the application from its representations and controls. Reusability, modularity, readability and quality of code are some of the main advantages that someone can expect from reflection.

By studying concurrent and distributed architectures, we can say that they deal with well-known mechanisms and policies, which are independent from the system's basic functionality (i.e. business objects) and could be implemented at the meta-level. Then, we are strongly convinced that mobile agent technology can benefit from the features of reflection when developing agent-based system.

At a low-level design, reflective techniques applied in mobile agents technology can help us to deal with:

- several policies of communication (e.g. synchronous, no reply, future, multicast)
- thread management (e.g. single-threaded, thread-per-message, thread-pool)
- data transport (e.g. sockets, pipes, shared memory)

At a high-level design, we can use introspection to develop tools dealing with the explicit representation of the agent patterns in the language itself (cf. meta-model in case tools).

Other interesting subject raised by authors [1], is the possible standardization of the "Agent Technology". Many people are working, or at least they say they are working, with mobile agents. However, each one of them proposes different solutions and different standards, differing widely in their architectures and implementations. Now we are in a certain phase that a standardization of all we have learned so far is really necessary. In order to have mobile agents interoperating in a standardized way a group of companies (Crystaliz, General Magic, Inc., GMD Fokus, IBM Corporation and the Open Group) were created. They proposed what they call of MASIF – *Mobile Agent System Interoperability Facility* and brought it to the attention of the OMG – Object Management Group.

With this work we hope we could introduce a new approach of how to solve particular problems in the “Agent Technology” and not to give to the reader another mobile agent framework. We hope also this work can encourage other people to work on mobile agent technology and propose new design patterns for it.

References

[1] D. Lange. “Mobile Objects and Mobile Agents: The Future of Distributed computing? ”. ECOOP’98 – Brussels, Belgium, July1998.

[2] D. Lange. “Agent Design Patterns: Elements of Agent Application Design”. Autonomous Agents’98, Minneapolis, May 1998.

[3] D. Lange and M. Oshima. “Dispatch your agents; shut off your machine”. In *Communications of the ACM*, 42, 3, 88-89, 1999.

[4] D. Deugo, M. Weiss. “A Case for Mobile Agent Patterns”. School of Computer Science – Carleton University, 1999.

[5] D. Deugo, F. Oppacher, B. Ashfield, M. Weiss. “Communication as a Means to Differentiate Objects, Components and Agents”. Submitted to TOOLS USA 99, 1999.

[6] E. Kendall, M. Malkoun, C.Jiang . “The Layered Agent Patterns”. The Pattern Languages of Programs, PLOP’96, Illinois, USA, September 1996.

[7] A. Silva, J. Delgado. “Agent Pattern for Mobile Agent Systems”. In European Conference on Pattern Languages of Programming and Computing, EuroPLoP’98, 1998.

[8] R. Lavender, D. Schmidt, “Active Object: An Object Behavioral Patter for Concurrent Programming”. In J. Vlissides, J. Coplien and N. Kerth, *“Pattern Languages of Program Design 2”*, Addisson-Wesley, 1996.

[9] N. Jennings, K. Sycara and M. Wooldridge. “A Roadmap of Agent Research and Development”. In *Autonomus Agents and Multi-Agent Systems*, 1, 7-38, 1998.

[10] S. Franklin and A. Graesser. “Is it an agent, or just a program ?”. In Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[11] M. Wooldridge and N. Jennings. “Intelligent Agents: Theory and Practice”. *Knowledge Engineering Review*, 1994.

[12] Smith, A. Cypher and J. Sphorer. KidSim: “Programming Agents without a Programming Language”. *Communications of the ACM*, 37, 7, 55-67.

[13] P. Maes. Artificial Life Meets Entertainment: “Life like Autonomous Agents”. In *Communications of the ACM*, 38, 11, 108-114, 1995.

[14] OMG – Object Management Group. “Agent Green Paper”. OMG Document ec/99-03-11, March 1999.

[15] C. Harrison, D. Chess and A. Kershenbaum. Mobile Agents: “Are they a good idea ?”. IBM Research Division, 1995.

[16] J.Mayfield, Y.Labrou and T. Finin. “Evaluating KQML as an Agent Communication Language”. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents Volume II*. Springer-Verlag, 1996.

[17] T. Finin, R. Fritzson, D. McKay, R. McEntire. “KQML as an Agent Communication Language”. *In Proceedings of the Third International Conference on Information and Knowledge Management (CIKM’94)*, ACM Press, November 1994.

[18] E. Gamma, R. Helm, R. Johnson, J. Vilissides. “Design Patterns: Elements of Reusable Object Oriented Software”. Addison-Wesley, 1998.

[19] F. Buchmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. “Pattern Oriented Software Arquitecture: A System of Patterns”.Wiley and Sons, 1996.

[20] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King and S. Angel. “A Pattern Language”. Oxford University Press, New York, 1977.

[21] J. Nelson. “Programming Mobile Objects with Java”. Wiley, 1999.

[22] B. Smith. “Reflection and Semantics in a Procedural Programming Language”. MIT, January 1982.

[23] P. Maes. “Concepts and Experiments in Computational Reflection”. Vrije Universiteit Brussel, 1987.

[24] G. Kiczales, J. des Rivieres, D. Bobrow. “The Art of the Metaobject Protocol”. Cambridge, MIT Press, 1991.

[25] J. McAffer. “Meta-level Programming with CodA”. In Proceedings of ECOOP'95 - Object-Oriented Programming - 9th European Conference. p 190-214.

[26] General Magic Inc. Telescript. http://www.genmagic.com

[27] IBM Japan. Aglets http:/www.trl.ibm.co.jp/aglets

[28] Mitsubishi Electric. Concordia. http://www.meitca.com/HSL/Projects/Concordia

[29] Object Space. Voyager. http://www.objectspace.com/voyager

[30] General Magic Inc. Odyssey. http://www.genmagic.com/

[31] Sun Systems. Java Language. http://www.javasoft.com - http://www.sun.com

[32] Rational Software Corp. UML – Unified Modeling Language. http://www.rational.com

The following Java Documentation represents an implementation of the five patterns that we have showed in this document.

API User's Guide Class Hierarchy Index

Package Index

Other Packages

- package agent
- package capability
- package coordinator
- package environment
- package examples
- package exceptions
- package interaction
- package message
- package transporter
- package travel

All Packages Class Hierarchy Index

package agent

Class Index

- AbstractAgent
- Agent
- AgentID
- AgentProxy
- AgentProxyState
- AgentProxyStateLocal
- AgentProxyStateRemote

All Packages Class Hierarchy This Package Previous Next Index

Class agent.AbstractAgent

```
java.lang.Object
   |
   +----java.lang.Thread
           |
           +----agent.AbstractAgent
```

public abstract class **AbstractAgent**
extends Thread
implements Serializable

The AbstractAgent class defines the main methods for the Agent.

Field Index

- **agentId**
- **currentAddress**
- **currentEnvironment**
- **homeAddress**

Constructor Index

- **AbstractAgent**()
  Constructs an abstract agent.
- **AbstractAgent**(String)
  Constructs an abstract agent defining the home address of the Agent.

Method Index

- **dispatch**(String)
  Dispatch an agent to another Environement.
- **getHomeAddress**()
  Returns the Agent home address.
- **getId**()
  Returns the Id of the Agent.
- **handleMessage**(Message)
  Handles all incoming messages for the Agent.
- **setCurrentAddress**(String)
  Sets the Agent current address.

Fields

agentId

```
protected String agentId
```

homeAddress

```
protected String homeAddress
```

currentAddress

```
protected String currentAddress
```

currentEnvironment

```
protected IAgentEnvironment currentEnvironment
```

Constructors

AbstractAgent

```
public AbstractAgent()
```

Constructs an abstract agent.

AbstractAgent

```
public AbstractAgent(String homeAddress)
```

Constructs an abstract agent defining the home address of the Agent.

Parameters:
homeAddress - java.lang.String

Methods

dispatch

```
protected final void dispatch(String address)
```

Dispatch an agent to another Environement.

Parameters:
address - java.lang.String

getHomeAddress

```
protected String getHomeAddress()
```

Returns the Agent home address.

Returns:
java.lang.String

●getId

```
protected final String getId()
```

Returns the Id of the Agent.

Returns:
java.lang.String

●handleMessage

```
public abstract Object handleMessage(Message msg) throws
                                        MessageNotUnderstood
```

Handles all incoming messages for the Agent. This method must be implemented by all classes that extends AbstractAgent class.

Parameters:
msg - message.Message
Returns:
java.lang.Object
Throws: MessageNotUnderstood
if the Message was not handled by the Agent.

●setCurrentAddress

```
protected void setCurrentAddress(String currentAddress)
```

Sets the Agent current address.

Parameters:
currentAddress - java.lang.String

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy This Package Previous Next Index

Class agent.Agent

```
java.lang.Object
   |
   +----java.lang.Thread
           |
           +----agent.AbstractAgent
                   |
                   +----agent.Agent
```

public class **Agent**
extends AbstractAgent

The Agent class is the abstract base class to create your own personalized Agents.

Field Index

- **capability**
- **travelPlan**

Constructor Index

- **Agent**()
  Constructs an agent.
- **Agent**(String)
  Contructs an agent defining its home address.

Method Index

- **getConsumerCapabilities**()
  Returns the agent's consumer capabilities.
- **getProviderCapabilities**()
  Returns the agent's provider capabilities.
- **handleMessage**(Message)
  Handles the incoming messages to the agent.
- **meet**(AgentProxy)
  Gives to an agent a proxy of another agent in order to allow them to interact.

Fields

capability

```
protected AgentCapability capability
```

travelPlan

```
protected TravelPlan travelPlan
```

Constructors

Agent

```
public Agent()
```

Constructs an agent.

Agent

```
public Agent(String homeAddress)
```

Contructs an agent defining its home address.

Parameters:
homeAddress - java.lang.String

Methods

getConsumerCapabilities

```
public Vector getConsumerCapabilities()
```

Returns the agent's consumer capabilities.

Returns:
java.util.Vector

getProviderCapabilities

```
public Vector getProviderCapabilities()
```

Returns the agent's provider capabilities.

Returns:
java.util.Vector

handleMessage

```
public Object handleMessage(Message msg) throws MessageNotUnderstood
```

Handles the incoming messages to the agent. This method must be overrided in the user agent class in order to handle the desired messages.

Parameters:
msg - message.Message

Returns:
java.lang.Object
Throws: MessageNotUnderstood
If the Message has not handled by the agent.
Overrides:
handleMessage in class AbstractAgent

●**meet**

```
public void meet(AgentProxy agent)
```

Gives to an agent a proxy of another agent in order to allow them to interact.

Parameters:
agent - agent.AgentProxy

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy This Package Previous Next Index

Class agent.AgentID

```
java.lang.Object
   |
   +----agent.AgentID
```

public class **AgentID**
extends Object

The AgentID class generates Id numbers for the agents. It uses the Singleton Design Pattern.

Field Index

- **agentId**
- **id**

Constructor Index

- **AgentID**()
  Constructs an AgentID.

Method Index

- **instance**()
  Returns always the same instance.
- **newAgentId**()
  Generates a new sequential number to be used as Id.

Fields

agentId

```
private static AgentID agentId
```

id

```
private int id
```

Constructors

AgentID

```
public AgentID()
```

Constructs an AgentID.

Methods

instance

```
public static AgentID instance()
```

Returns always the same instance.

Returns:
agent.AgentID

newAgentId

```
public int newAgentId()
```

Generates a new sequential number to be used as Id.

Returns:
int

All Packages Class Hierarchy This Package Previous Next Index

Class agent.AgentProxy

```
java.lang.Object
   |
   +----agent.AgentProxy
```

public class **AgentProxy**
extends Object
implements Serializable

The AgentProxy class is a placeholder for Agent objects. The purpose of it is to provide a transparent access to the real Agent object.

Field Index

- **proxyState**

Constructor Index

- **AgentProxy**(Agent)
  Constructs a new AgentProxy class defining the real Agent object that it will represent.
- **AgentProxy**(String)
  Constructs a new AgentProxy class defining the address where the agent will be created.

Method Index

- **dispatch**(String)
  Dispatchs the agent to the especified address.
- **getAgent**()
  Returns the real Agent object that the proxy represent.
- **getConsumerCapabilities**()
  Returns the agent's consumer capabilities.
- **getHomeAddress**()
  Returns the Agent home address.
- **getId**()
  Returns the Agent identification.
- **getProviderCapabilities**()
  Returns the agent's provider capabilities.
- **sendMessage**(Message)
  Sends a synchronous message to an agent.
- **setCurrentAddress**(String)
  Sets the agent's current address.

- **updateAgentProxyState**(Agent)
  Updates the state of the agent proxy changing to Local state.
- **updateAgentProxyState**(AgentProxy)
  Updates the state of the agent proxy changing to Remote state.

Fields

proxyState

```
private AgentProxyState proxyState
```

Constructors

AgentProxy

```
public AgentProxy(Agent agent)
```

Constructs a new AgentProxy class defining the real Agent object that it will represent.

Parameters:
agent - Agent

AgentProxy

```
public AgentProxy(String homeAddress)
```

Constructs a new AgentProxy class defining the address where the agent will be created.

Parameters:
homeAddress - java.lang.String

Methods

dispatch

```
public void dispatch(String address)
```

Dispatchs the agent to the especified address.

Parameters:
address - java.lang.String

getAgent

```
public Agent getAgent()
```

Returns the real Agent object that the proxy represent.

Returns:
agent.Agent

getConsumerCapabilities

```
public Vector getConsumerCapabilities()
```

Returns the agent's consumer capabilities.

Returns:
java.util.Vector

getHomeAddress

```
public String getHomeAddress()
```

Returns the Agent home address.

Returns:
java.lang.String

getId

```
public String getId()
```

Returns the Agent identification.

Returns:
java.lang.String

getProviderCapabilities

```
public Vector getProviderCapabilities()
```

Returns the agent's provider capabilities.

Returns:
java.util.Vector

sendMessage

```
public Object sendMessage(Message msg) throws MessageNotUnderstood
```

Sends a synchronous message to an agent.

Parameters:
msg - message.Message
Returns:
java.lang.Object
Throws: AgentException
If the Message was not handled by the agent.

setCurrentAddress

```
public void setCurrentAddress(String currentAddress)
```

Sets the agent's current address.

Parameters:
currentAddress - java.lang.String

updateAgentProxyState

```
private void updateAgentProxyState(Agent agent)
```

Updates the state of the agent proxy changing to Local state.

Parameters:
agent - agent.Agent

updateAgentProxyState

```
private void updateAgentProxyState(AgentProxy remoteProxy)
```

Updates the state of the agent proxy changing to Remote state.

Parameters:
remoteProxy - agent.AgentProxy

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy This Package Previous Next Index

Class agent.AgentProxyState

```
java.lang.Object
   |
   +----agent.AgentProxyState
```

public abstract class **AgentProxyState**
extends Object
implements Serializable

This class represent the state of the agent's proxy.

Constructor Index

- **AgentProxyState**()
  AgentProxyState constructor.

Method Index

- **getAgent**()
  Returns the real agent object that the proxy represents.
- **getConsumerCapabilities**()
  Returns the agent's consumer capabilities.
- **getHomeAddress**()
  Returns the host address were the agent was created.
- **getId**()
  Returns the agent identification.
- **getProviderCapabilities**()
  Returns the agent's provider capabilities.
- **handleMessage**(Message)
  Handles the incoming messages to the agent.
- **setCurrentAddress**(String)
  Sets the agent current address.

Constructors

AgentProxyState

```
public AgentProxyState()
```

AgentProxyState constructor.

Methods

getAgent

```
public abstract Agent getAgent()
```

Returns the real agent object that the proxy represents. This method must be implemented by classes that extends AgentProxyState class.

Returns:
agent.Agent

getConsumerCapabilities

```
public abstract Vector getConsumerCapabilities()
```

Returns the agent's consumer capabilities. This method must be implemented by classes that extends AgentProxyState class.

Returns:
java.util.Vector

getHomeAddress

```
public abstract String getHomeAddress()
```

Returns the host address were the agent was created. This method must be implemented by classes that extends AgentProxyState class.

Returns:
java.lang.String

getId

```
public abstract String getId()
```

Returns the agent identification. This method must be implemented by classes that extends AgentProxyState class.

Returns:
java.lang.String

getProviderCapabilities

```
public abstract Vector getProviderCapabilities()
```

Returns the agent's provider capabilities. This method must be implemented by classes that extends AgentProxyState class.

Returns:
java.util.Vector

handleMessage

```
public abstract Object handleMessage(Message msg) throws
                                                MessageNotUnderstood
```

Handles the incoming messages to the agent. This method must be implemented by classes that extends AgentProxyState class.

Parameters:
msg - message.Message
Returns:
java.lang.Object
Throws: MessageNotUnderstood
If the Message has not handled by the agent.

●setCurrentAddress

```
public abstract void setCurrentAddress(String currentAddress)
```

Sets the agent current address. This method must be implemented by classes that extends AgentProxyState class.

Parameters:
currentAddress - java.lang.String

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy This Package Previous Next Index

Class agent.AgentProxyStateLocal

```
java.lang.Object
   |
   +----agent.AgentProxyState
           |
           +----agent.AgentProxyStateLocal
```

public class **AgentProxyStateLocal**
extends AgentProxyState

This class defines the agent proxy state when the agent object is in the same environement as its proxy.

Field Index

- **agent**

Constructor Index

- **AgentProxyStateLocal**()
  Creates a new proxy for a local agent.
- **AgentProxyStateLocal**(Agent)
  Creates a new proxy for a local agent defining the agent object.
- **AgentProxyStateLocal**(String)
  Creates a new state proxy for a local agent defining the address where the agent will be created.

Method Index

- **getAgent**()
  Returns the real agent object.
- **getConsumerCapabilities**()
  Returns the agent's consumer capabilities.
- **getHomeAddress**()
  Returns the host address were the agent was created.
- **getId**()
  Returns the Id of the Agent.
- **getProviderCapabilities**()
  Returns the agent's provider capabilities.
- **handleMessage**(Message)
  Handles all incoming messages for the local agent.
- **setCurrentAddress**(String)
  Sets the Agent current address.

Fields

agent

```
Agent agent
```

Constructors

AgentProxyStateLocal

```
public AgentProxyStateLocal()
```

Creates a new proxy for a local agent.

AgentProxyStateLocal

```
public AgentProxyStateLocal(Agent agent)
```

Creates a new proxy for a local agent defining the agent object.

Parameters:
agent - agent.Agent

AgentProxyStateLocal

```
public AgentProxyStateLocal(String homeAddress)
```

Creates a new state proxy for a local agent defining the address where the agent will be created.

Parameters:
homeAddress - java.lang.String

Methods

getAgent

```
public Agent getAgent()
```

Returns the real agent object.

Returns:
agent.Agent
Overrides:
getAgent in class AgentProxyState

getConsumerCapabilities

```
public Vector getConsumerCapabilities()
```

Returns the agent's consumer capabilities.

Returns:
java.util.Vector
Overrides:
getConsumerCapabilities in class AgentProxyState

●getHomeAddress

```
public String getHomeAddress()
```

Returns the host address were the agent was created.

Returns:
java.lang.String
Overrides:
getHomeAddress in class AgentProxyState

●getId

```
public String getId()
```

Returns the Id of the Agent.

Returns:
java.lang.String
Overrides:
getId in class AgentProxyState

●getProviderCapabilities

```
public Vector getProviderCapabilities()
```

Returns the agent's provider capabilities.

Returns:
java.util.Vector
Overrides:
getProviderCapabilities in class AgentProxyState

●handleMessage

```
public Object handleMessage(Message msg) throws MessageNotUnderstood
```

Handles all incoming messages for the local agent.

Parameters:
msg - message.Message
Returns:
java.lang.Object

Throws: MessageNotUnderstood
if the Message was not handled by the Agent.
Overrides:
handleMessage in class AgentProxyState

●setCurrentAddress

```
public void setCurrentAddress(String currentAddress)
```

Sets the Agent current address.

Parameters:
currentAddress - java.lang.String
Overrides:
setCurrentAddress in class AgentProxyState

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy This Package Previous Next Index

Class agent.AgentProxyStateRemote

```
java.lang.Object
   |
   +----agent.AgentProxyState
           |
           +----agent.AgentProxyStateRemote
```

public class **AgentProxyStateRemote**
extends AgentProxyState

This class defines the agent proxy state when the agent object is not in the same environement. It is remote.

Field Index

- **remoteProxy**

Constructor Index

- **AgentProxyStateRemote**()
  Creates a new proxy for a remote agent.
- **AgentProxyStateRemote**(AgentProxy)
  Creates a new proxy for a remote agent defining the remote proxy that represent an agent.

Method Index

- **getAgent**()
  Returns the real agent object of a remote proxy.
- **getConsumerCapabilities**()
  Gets the remote agent's consumer capabilities.
- **getHomeAddress**()
  Returns the host address were the remote agent was created.
- **getId**()
  Returns the Id of the remote agent.
- **getProviderCapabilities**()
  Gets the remote agent's provider capabilities.
- **handleMessage**(Message)
  Handles all incoming messages for the agent and forward them to the remote agent.
- **setCurrentAddress**(String)
  Sets the remote agent current address.
- **setRemoteProxy**(AgentProxy)

Sets the remote proxy.

Fields

remoteProxy

```
private AgentProxy remoteProxy
```

Constructors

AgentProxyStateRemote

```
public AgentProxyStateRemote()
```

Creates a new proxy for a remote agent.

AgentProxyStateRemote

```
public AgentProxyStateRemote(AgentProxy remoteProxy)
```

Creates a new proxy for a remote agent defining the remote proxy that represent an agent.

Parameters:
agentCurrentAddress - java.lang.String

Methods

getAgent

```
public Agent getAgent()
```

Returns the real agent object of a remote proxy.

Returns:
agent.Agent
Overrides:
getAgent in class AgentProxyState

getConsumerCapabilities

```
public Vector getConsumerCapabilities()
```

Gets the remote agent's consumer capabilities.

Returns:
java.util.Vector
Overrides:
getConsumerCapabilities in class AgentProxyState

getHomeAddress

```
public String getHomeAddress()
```

Returns the host address were the remote agent was created.

Returns:
java.lang.String
Overrides:
getHomeAddress in class AgentProxyState

getId

```
public String getId()
```

Returns the Id of the remote agent.

Returns:
java.lang.String
Overrides:
getId in class AgentProxyState

getProviderCapabilities

```
public Vector getProviderCapabilities()
```

Gets the remote agent's provider capabilities.

Returns:
java.util.Vector
Overrides:
getProviderCapabilities in class AgentProxyState

handleMessage

```
public Object handleMessage(Message msg) throws MessageNotUnderstood
```

Handles all incoming messages for the agent and forward them to the remote agent.

Parameters:
msg - message.Message
Returns:
java.lang.Object
Throws: MessageNotUnderstood
if the Message was not handled by the remote Agent.
Overrides:
handleMessage in class AgentProxyState

setCurrentAddress

```
public void setCurrentAddress(String currentAddress)
```

Sets the remote agent current address.

Parameters:
currentAddress - java.lang.String
Overrides:
setCurrentAddress in class AgentProxyState

setRemoteProxy

```
public void setRemoteProxy(AgentProxy remoteProxy)
```

Sets the remote proxy.

Parameters:
remoteProxy - agent.AgentProxy

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy Index

package capability

Class Index

- AgentCapability
- Consumer
- Provider

All Packages Class Hierarchy This Package Previous Next Index

Class capability.AgentCapability

```
java.lang.Object
   |
   +----capability.AgentCapability
```

public class **AgentCapability**
extends Object
implements Serializable

This class represent the consumer and provider capabilities of an agent.

Field Index

- **consumerCapabilities**
- **providerCapabilities**

Constructor Index

- **AgentCapability**()
  Creates a new agent capability.

Method Index

- **addConsumerCapability**(String)
  Adds a new consumer capability.
- **addProviderCapability**(String)
  Adds a new provider capability.
- **getConsumerCapabilities**()
  Returns the current consumer capabilities of an agent.
- **getProviderCapabilities**()
  Returns the current provider capabilities of an agent.

Fields

providerCapabilities

```
private Vector providerCapabilities
```

consumerCapabilities

```
private Vector consumerCapabilities
```

Constructors

AgentCapability

```
public AgentCapability()
```

Creates a new agent capability.

Methods

addConsumerCapability

```
public void addConsumerCapability(String description)
```

Adds a new consumer capability.

Parameters:
description - java.lang.String

addProviderCapability

```
public void addProviderCapability(String description)
```

Adds a new provider capability.

Parameters:
description - java.lang.String

getConsumerCapabilities

```
public Vector getConsumerCapabilities()
```

Returns the current consumer capabilities of an agent.

Returns:
java.util.Vector

getProviderCapabilities

```
public Vector getProviderCapabilities()
```

Returns the current provider capabilities of an agent.

Returns:
java.util.Vector

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy This Package Previous Next Index

Class capability.Consumer

```
java.lang.Object
   |
   +----capability.Consumer
```

public class **Consumer**
extends Object
implements Serializable

This class describes a consumer cabability.

Field Index

● **description**

Constructor Index

● **Consumer**()
Creates a new consumer capability.
● **Consumer**(String)
Creates a new consumer capability defining its description.

Method Index

● **getDescription**()
Returns the description of a consumer capability.

Fields

● **description**

```
private String description
```

Constructors

● **Consumer**

```
public Consumer()
```

Creates a new consumer capability.

● **Consumer**

```
public Consumer(String description)
```

Creates a new consumer capability defining its description.

Parameters:
description - java.lang.String

Methods

● **getDescription**

```
public String getDescription()
```

Returns the description of a consumer capability.

Returns:
java.lang.String

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy This Package Previous Next Index

Class capability.Provider

```
java.lang.Object
   |
   +----capability.Provider
```

public class **Provider**
extends Object
implements Serializable

This type was created in VisualAge.

Field Index

description

Constructor Index

Provider()
Creates a new provider capability.
Provider(String)
Creates a new provider capability defining its description.

Method Index

getDescription()
Returns the description of a provider capability.

Fields

description

```
private String description
```

Constructors

Provider

```
public Provider()
```

Creates a new provider capability.

Provider

```
public Provider(String description)
```

Creates a new provider capability defining its description.

Parameters:
description - java.lang.String

Methods

getDescription

```
public String getDescription()
```

Returns the description of a provider capability.

Returns:
java.lang.String

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy Index

package coordinator

Class Index

- AbstractAgentCoordinator

All Packages Class Hierarchy This Package Previous Next Index

Class coordinator.AbstractAgentCoordinator

```
java.lang.Object
   |
   +----coordinator.AbstractAgentCoordinator
```

public abstract class **AbstractAgentCoordinator**
extends Object

This class is the environment's agent coordinator. It holds all agents that are in the environment.

Field Index

- **agentList**

Constructor Index

- **AbstractAgentCoordinator**()
  Creates a new agent coordinator.

Method Index

- **addAgent**(AgentProxy)
  Adds a new agent in the agent coordinator.
- **findAgent**(String)
  Looks for an agent in the agent coordinator.
- **getAllAgents**()
  Returns all agents in the agent coordinator.
- **removeAgent**(AgentProxy)
  Removes an agent from the agent coordinator.
- **startAgent**(String)
  Starts an agent thread.

Fields

agentList

```
protected Vector agentList
```

Constructors

AbstractAgentCoordinator

```
public AbstractAgentCoordinator()
```

Creates a new agent coordinator.

Methods

addAgent

```
public AgentProxy addAgent(AgentProxy agentProxy)
```

Adds a new agent in the agent coordinator.

Parameters:
agent - agent.AgentProxy
Returns:
agent.AgentProxy

findAgent

```
public AgentProxy findAgent(String id) throws AgentNotFound
```

Looks for an agent in the agent coordinator. The search key is the agent identification. This method can be overrided in order to create different searchs with different search keys.

Parameters:
id - java.lang.String
Returns:
agent.AgentProxy

getAllAgents

```
public final Vector getAllAgents()
```

Returns all agents in the agent coordinator.

Returns:
java.util.Vector

removeAgent

```
public void removeAgent(AgentProxy agentProxy)
```

Removes an agent from the agent coordinator.

Parameters:

agentProxy - agent.AgentProxy

● **startAgent**

```
public void startAgent(String agentId)
```

Starts an agent thread.

Parameters:
agentId - java.lang.String

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy Index

package environment

Interface Index

- IAbstractAgentEnvironment
- IAgentEnvironment
-

Class Index

- AbstractAgentEnvironment
- AgentEnvironment

All Packages Class Hierarchy This Package Previous Next Index

Class environment.AbstractAgentEnvironment

```
java.lang.Object
   |
   +----java.rmi.server.RemoteObject
           |
           +----java.rmi.server.RemoteServer
                   |
                   +----java.rmi.server.UnicastRemoteObject
                           |
                           +----environment.AbstractAgentEnvironment
```

public abstract class **AbstractAgentEnvironment**
extends UnicastRemoteObject
implements IAbstractAgentEnvironment, Serializable

This class defines the basic structure of an agent environment.

Field Index

- **address**
- **coordinator**
- **otherEnvironments**
- **transporter**

Constructor Index

- **AbstractAgentEnvironment**()
  Inicialize the agent environment.
- **AbstractAgentEnvironment**(int)
  Inicialize the agent environment in a specified port.

Method Index

- **addOtherEnvironment**(String)
  Adds other environment address.
- **arrivedAgent**(AgentProxy)
  Adds a new agent to the environment when the agent arrives.
- **createAgent**(String)
  Creates a new agent in the environment and returns the respective agent proxy.
- **dispatch**(String, String)
  Dispatch an agent to a specific address.

- **dispatchedAgent**(AgentProxy, String)
  Is perfomed when the agent has been dispatched to another environment.
- **findAgent**(String)
  Looks for an agent in the agent coordinator.
- **getAddress**()
  Returns the environment address.
- **getAllAgents**()
  Returns all agents in the agent coordinator.
- **getKnownEnvironments**()
  Returns a list of other environment addresses.
- **handleRemoteMessage**(Message)
  Handles the incoming messages from other environments.
- **receiveAgent**(byte[])
  Receive an agent that has been dispatched to this environment.
- **sendRemoteMessage**(Message)
  Sends a message to an other environment.
- **startAgent**(String)
  Starts an agent thread.

Fields

address

```
protected String address
```

transporter

```
protected RMITransporter transporter
```

coordinator

```
protected AgentCoordinator coordinator
```

otherEnvironments

```
protected Vector otherEnvironments
```

Constructors

AbstractAgentEnvironment

```
public AbstractAgentEnvironment() throws RemoteException
```

Inicialize the agent environment.

AbstractAgentEnvironment

```
public AbstractAgentEnvironment(int port) throws RemoteException
```

Inicialize the agent environment in a specified port.

Parameters:
port - int

Methods

addOtherEnvironment

```
public void addOtherEnvironment(String address)
```

Adds other environment address. The agents can ask to the local environment other environment addresses where they can go.

Parameters:
address - java.lang.String

arrivedAgent

```
protected void arrivedAgent(AgentProxy agentProxy)
```

Adds a new agent to the environment when the agent arrives.

Parameters:
agentProxy - agent.AgentProxy

createAgent

```
public AgentProxy createAgent(String className)
```

Creates a new agent in the environment and returns the respective agent proxy.

Parameters:
className - java.lang.String
Returns:
agent.AgentProxy

dispatch

```
public void dispatch(String agentId, String address)
```

Dispatch an agent to a specific address.

Parameters:
agentId - java.lang.String
address - java.lang.String

dispatchedAgent

```
protected final void dispatchedAgent(AgentProxy agentProxy,
                                     String homeAddress)
```

Is perfomed when the agent has been dispatched to another environment.

Parameters:
agentProxy - agent.AgentProxy
homeAddress - String

findAgent

```
public AgentProxy findAgent(String id) throws RemoteException,
                                              AgentNotFound
```

Looks for an agent in the agent coordinator.

Parameters:
id - String

getAddress

```
public final String getAddress()
```

Returns the environment address.

Returns:
java.lang.String

getAllAgents

```
public final Vector getAllAgents()
```

Returns all agents in the agent coordinator.

Returns:
java.util.Vector

getKnownEnvironments

```
public Vector getKnownEnvironments()
```

Returns a list of other environment addresses.

Returns:
java.util.Vector

handleRemoteMessage

```
public Object handleRemoteMessage(Message msg) throws
                       RemoteException, RemoteMessageNotUnderstood
```

Handles the incoming messages from other environments.

Parameters:
msg - message.Message

Returns:
java.lang.Object
Throws: RemoteMessageNotUnderstood
If the message was not handled by the agent.

●**receiveAgent**

```
public void receiveAgent(byte byteArray[])
```

Receive an agent that has been dispatched to this environment.

Parameters:
byteArray - byte[]

●**sendRemoteMessage**

```
public Object sendRemoteMessage(Message msg) throws
                                      RemoteMessageNotUnderstood
```

Sends a message to an other environment.

Parameters:
msg - message.Message
Returns:
java.lang.Object
Throws: RemoteMessageNotUnderstood
If the message was not handled by the agent.

●**startAgent**

```
public void startAgent(String agentId)
```

Starts an agent thread.

Parameters:
agentId - java.lang.String

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy This Package Previous Next Index

Class environment.AgentEnvironment

```
java.lang.Object
   |
   +----java.rmi.server.RemoteObject
           |
           +----java.rmi.server.RemoteServer
                   |
                   +----java.rmi.server.UnicastRemoteObject
                           |
                           +----environment.AbstractAgentEnvironment
                                   |
                                   +----environment.AgentEnvironment
```

public class **AgentEnvironment**
extends AbstractAgentEnvironment
implements IAgentEnvironment

This class extends the AbstractAgentEnvironment and defines other specific attributes.

Field Index

- **envUI**

Constructor Index

- **AgentEnvironment**(int)
  Creates an environment that will listen in a specific port.

Method Index

- **arrivedAgent**(AgentProxy)
  Receives an agent that has arrived to this environment.
- **createAgent**(String)
  Creates a new agent of the specified class and returns the respective agent proxy.
- **dispatch**(String, String)
  Dispatch an agent to a specific address.
- **getConsumerAgents**(String)
  Returns only the agents that has a specific consumer capability.
- **getProviderAgents**(String)
  Returns only the agents that has a specific provider capability.
- **print**(String)
  Sends a string to the user interface.
- **setUI**(EnvironmentGUI)

  Sets an user interface to the environment.
- **startAgent**(String)
  Starts an agent thread.

Fields

envUI

```
protected EnvironmentGUI envUI
```

Constructors

AgentEnvironment

```
public AgentEnvironment(int port) throws RemoteException
```

Creates an environment that will listen in a specific port.

Parameters:
port - int

Methods

arrivedAgent

```
public void arrivedAgent(AgentProxy agentProxy)
```

Receives an agent that has arrived to this environment.

Parameters:
agentProxy - agent.AgentProxy
Overrides:
arrivedAgent in class AbstractAgentEnvironment

createAgent

```
public AgentProxy createAgent(String className)
```

Creates a new agent of the specified class and returns the respective agent proxy.

Parameters:
className - java.lang.String
Returns:
agent.AgentProxy
Overrides:
createAgent in class AbstractAgentEnvironment

●**dispatch**

```
public void dispatch(String agentId, String address)
```

Dispatch an agent to a specific address.

Parameters:
agentId - java.lang.String
address - java.lang.String
Overrides:
dispatch in class AbstractAgentEnvironment

●**getConsumerAgents**

```
public Vector getConsumerAgents(String description)
```

Returns only the agents that has a specific consumer capability.

Return:
java.util.Vector
Parameters:
java.lang.String description

●**getProviderAgents**

```
public Vector getProviderAgents(String description)
```

Returns only the agents that has a specific provider capability.

Return:
java.util.Vector
Parameters:
java.lang.String description

●**print**

```
public void print(String line)
```

Sends a string to the user interface.

Parameters:
line - java.lang.String

●**setUI**

```
public void setUI(EnvironmentGUI envUI)
```

Sets an user interface to the environment.

Parameters:
envUI - userinterface.EnvironmentGUI

●**startAgent**

```
public void startAgent(String agentId)
```

Starts an agent thread.

Parameters:
agentId - java.lang.String
Overrides:
startAgent in class AbstractAgentEnvironment

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy Index

package examples

Class Index

- Computer
- ComputerBuyer
- ComputerShop

All Packages Class Hierarchy This Package Previous Next Index

Class examples.Computer

```
java.lang.Object
   |
   +----examples.Computer
```

public class **Computer**
extends Object
implements Serializable

This class defines the computer structure.

Field Index

- **description**
- **num**
- **price**

Constructor Index

- **Computer**()
  Creates a new computer instance.

Method Index

- **getDescription**()
  Gets the computer description.
- **getPrice**()
  Gets the computer price.

Fields

price

```
private float price
```

num

```
private Random num
```

description

```
private String description
```

Constructors

Computer

```
public Computer()
```

Creates a new computer instance.

Methods

getDescription

```
public String getDescription()
```

Gets the computer description.

Returns:
java.lang.String

getPrice

```
public Float getPrice()
```

Gets the computer price.

Returns:
float

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy This Package Previous Next Index

Class examples.ComputerBuyer

```
java.lang.Object
   |
   +----java.lang.Thread
           |
           +----agent.AbstractAgent
                   |
                   +----agent.Agent
                           |
                           +----examples.ComputerBuyer
```

public class **ComputerBuyer**
extends Agent

The ComputerBuyer class specifies an agent that buys computers.

Field Index

- **bestPrice**
- **bestPriceId**

Constructor Index

- **ComputerBuyer**()
  Creates a new ComputerBuyer instance.
- **ComputerBuyer**(String)
  Creates a new ComputerBuyer instance defining its home address.

Method Index

- **handleMessage**(Message)
  Handles all incoming messages to the agent.
- **run**()
  Defines the agent behavior that will be performed in each one of the environments where the agent will go.

Fields

bestPrice

```
float bestPrice
```

bestPriceId

```
String bestPriceId
```

Constructors

ComputerBuyer

```
public ComputerBuyer()
```

Creates a new ComputerBuyer instance.

ComputerBuyer

```
public ComputerBuyer(String homeAddress)
```

Creates a new ComputerBuyer instance defining its home address.

Parameters:
homeAddress - java.lang.String

Methods

handleMessage

```
public Object handleMessage(Message msg) throws MessageNotUnderstood
```

Handles all incoming messages to the agent.

Parameters:
msg - message.Message
Returns:
java.lang.Object
Overrides:
handleMessage in class Agent

run

```
public void run()
```

Defines the agent behavior that will be performed in each one of the environments where the agent will pass.

Overrides:
run in class Thread

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy This Package Previous Next Index

Class examples.ComputerShop

```
java.lang.Object
   |
   +----java.lang.Thread
           |
           +----agent.AbstractAgent
                   |
                   +----agent.Agent
                           |
                           +----examples.ComputerShop
```

public class **ComputerShop**
extends Agent

This class defines the structure of the computer shop. The computer shop is a stationary agent.

Field Index

- **computer**

Constructor Index

- **ComputerShop**()
  Creates a new ComputerShop instance.
- **ComputerShop**(String)
  Creates a new ComputerShop instance defining its home address.

Method Index

- **handleMessage**(Message)
  Handle the messages from the ComputerBuyer agent when it asks for the computer price.
- **run**()
  Defines the main behavior of our computer shop.

Fields

computer

```
private Computer computer
```

Constructors

ComputerShop

```
public ComputerShop()
```

Creates a new ComputerShop instance.

ComputerShop

```
public ComputerShop(String homeAddress)
```

Creates a new ComputerShop instance defining its home address.

Parameters:
homeAddress - java.lang.String

Methods

handleMessage

```
public Object handleMessage(Message msg) throws MessageNotUnderstood
```

Handle the messages from the ComputerBuyer agent when it asks for the computer price.

Parameters:
msg - message.Message
Returns:
java.lang.Object
Overrides:
handleMessage in class Agent

run

```
public void run()
```

Defines the main behavior of our computer shop.

Overrides:
run in class Thread

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy Index

package interaction

Class Index

- AgentCoordinator

All Packages Class Hierarchy This Package Previous Next Index

Class interaction.AgentCoordinator

```
java.lang.Object
   |
   +----coordinator.AbstractAgentCoordinator
           |
           +----interaction.AgentCoordinator
```

public class **AgentCoordinator**
extends AbstractAgentCoordinator

This class extends the AbstractAgentCoordinator class and enables agent to interact.

Constructor Index

● **AgentCoordinator**()
Creates a new AgentCoordinator instance.

Method Index

● **getConsumerAgents**()
Returns all agents that has at leat one consumer capability.
● **getConsumerAgents**(String)
Returns all agents that has one specific consumer capability.
● **getProviderAgents**()
Returns all agents that has at leat one provider capability.
● **getProviderAgents**(String)
Returns all agents that has one specific provider capability.
● **notifyConsumers**(AgentProxy)
Notifies all consumer agents about a provider agent.

Constructors

● AgentCoordinator

```
public AgentCoordinator()
```

Creates a new AgentCoordinator instance.

Methods

● getConsumerAgents

```
public Vector getConsumerAgents()
```

Returns all agents that has at leat one consumer capability.

Returns:
java.util.Vector

● getConsumerAgents

```
public Vector getConsumerAgents(String description)
```

Returns all agents that has one specific consumer capability.

Parameters:
java.lang.String - description
Returns:
java.util.Vector

● getProviderAgents

```
public Vector getProviderAgents()
```

Returns all agents that has at leat one provider capability.

Returns:
java.util.Vector

● getProviderAgents

```
public Vector getProviderAgents(String description)
```

Returns all agents that has one specific provider capability.

Parameters:
java.lang.String - description
Returns:
java.util.Vector

● notifyConsumers

```
public void notifyConsumers(AgentProxy agent)
```

Notifies all consumer agents about a provider agent. param agent.AgentProxy agent

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy Index

package message

Class Index

- Message

All Packages Class Hierarchy This Package Previous Next Index

Class message.Message

```
java.lang.Object
   |
   +----message.Message
```

public class **Message**
extends Object
implements Serializable

This class defines the structure of the messages that will be sent to agents and to environments.

Field Index

- **content**
- **param**
- **receiverId**
- **remoteAddress**

Constructor Index

- **Message**()
  Creates a new Message instance.
- **Message**(String)
  Creates a new message instance defining its content.
- **Message**(String, Object[])
  Creates a new message instance defining its content and parameters.

Method Index

- **getContent**()
  Returns the content of a message.
- **getParameters**()
  Returns the parameters of a message.
- **getReceiver**()
  Returns the receiver of a message.
- **getRemoteAddress**()
  Returns the remote address of the environment where the message will be sent.
- **setContent**(String)
  Sets the content of a message.
- **setParameters**(Object[])
  Sets the parameters of a message.

- **setReceiver**(String)
  Sets the receiver of a message.
- **setRemoteAddress**(String)
  Sets the remote address of the environment where the message will be sent.
- **toString**()
  Returns a string of this class.

Fields

content

```
private String content
```

param

```
private Object param[]
```

receiverId

```
private String receiverId
```

remoteAddress

```
private String remoteAddress
```

Constructors

Message

```
public Message()
```

Creates a new Message instance.

Message

```
public Message(String content)
```

Creates a new message instance defining its content.

Parameters:
content - java.lang.String

Message

```
public Message(String content, Object param[])
```

Creates a new message instance defining its content and parameters.

Parameters:
content - java.lang.String
param - java.lang.Object[]

Methods

getContent

```
public String getContent()
```

Returns the content of a message.

Returns:
java.lang.String

getParameters

```
public Object[] getParameters()
```

Returns the parameters of a message.

Returns:
java.lang.Object[]

getReceiver

```
public String getReceiver()
```

Returns the receiver of a message.

Returns:
java.lang.String

getRemoteAddress

```
public String getRemoteAddress()
```

Returns the remote address of the environment where the message will be sent.

Returns:
java.lang.String

setContent

```
public void setContent(String content)
```

Sets the content of a message.

Parameters:
content - java.lang.String

setParameters

```
public void setParameters(Object param[])
```

Sets the parameters of a message.

Parameters:
param - java.lang.Object[]

setReceiver

```
public void setReceiver(String receiverId)
```

Sets the receiver of a message.

Parameters:
receiver - java.lang.String

setRemoteAddress

```
public void setRemoteAddress(String remoteAddress)
```

Sets the remote address of the environment where the message will be sent.

Parameters:
remoteAddress - java.lang.String

toString

```
public String toString()
```

Returns a string of this class.

Returns:
java.lang.String
Overrides:
toString in class Object

All Packages Class Hierarchy Index

package transporter

Class Index

- AbstractTransporter
- RMITransporter

All Packages Class Hierarchy This Package Previous Next Index

Class transporter.AbstractTransporter

```
java.lang.Object
   |
   +----transporter.AbstractTransporter
```

public abstract class **AbstractTransporter**
extends Object

Defines a basic structure of the object transporter.

Field Index

- **environment**

Constructor Index

- **AbstractTransporter**()
  Creates a new instance of AbstractTransporter class.

Method Index

- **receiveAgent**(byte[])
  Receives a serialized agent and returns a agent object.
- **sendAgent**(Agent, String)
  serializes and send an agent object to a specific address.

Fields

environment

```
protected AgentEnvironment environment
```

Constructors

AbstractTransporter

```
public AbstractTransporter()
```

Creats a new instance of AbstractTransporter class.

Methods

receiveAgent

```
public abstract Agent receiveAgent(byte byteArray[])
```

Receives a serialized agent and returns a agent object.

Parameters:
byteArray - byte[]
Returns:
agent.Agent

sendAgent

```
public abstract void sendAgent(Agent agent, String address)
```

Serializes and send an agent object to a specific address.

Parameters:
agent - agent.Agent
address - java.lang.String

All Packages Class Hierarchy This Package Previous Next Index

Class transporter.RMITransporter

```
java.lang.Object
   |
   +----transporter.AbstractTransporter
           |
           +----transporter.RMITransporter
```

public class **RMITransporter**
extends AbstractTransporter
implements Remote

This class defines a object transport using Java RMI.

Constructor Index

- **RMITransporter**()
  Creates a new RMITransporter instance.
- **RMITransporter**(AgentEnvironment)
  Creates a new RMITransporter defining the environment that it will work for.

Method Index

- **receiveAgent**(byte[])
  Receive a serialized agent object and returns the agent object.
- **sendAgent**(Agent, String)
  Serializes an agent object and sends it to a specific address.

Constructors

RMITransporter

```
public RMITransporter()
```

Creates a new RMITransporter instance.

RMITransporter

```
public RMITransporter(AgentEnvironment environment)
```

Creates a new RMITransporter defining the environment that it will work for.

Parameters:
environment - environment.AgentEnvironment

Methods

receiveAgent

```
public Agent receiveAgent(byte byteArray[])
```

Receive a serialized agent object and returns the agent object.

Parameters:
byteArray - byte[]
Returns:
agent.Agent
Overrides:
receiveAgent in class AbstractTransporter

sendAgent

```
public void sendAgent(Agent agent, String address)
```

Serializes an agent object and sends it to a specific address.

Parameters:
agent - agent.Agent
address - java.lang.String
Overrides:
sendAgent in class AbstractTransporter

All Packages Class Hierarchy Index

package travel

Class Index

- Destination
- TravelPlan

All Packages Class Hierarchy This Package Previous Next Index

Class travel.Destination

```
java.lang.Object
   |
   +----travel.Destination
```

public class **Destination**
extends Object
implements Serializable

This class defines a destination where an agent can go.

Field Index

address
isAlternative
visited

Constructor Index

Destination()
    Creates a new Destination instance.
Destination(String)
    Creates a new Destination instance defining its address.

Method Index

getAddress()
    Returns the address of a destination.
isAlternative()
    Check whether the destination is alternative or not
setAlternative(boolean)
    Sets a destination as alternative or not.
setVisited(boolean)
    Sets a destination as visited or not.
wasVisited()
    Check whether the destination was visited already or not.

Fields

visited

```
private boolean visited
```

address

```
private String address
```

isAlternative

```
private boolean isAlternative
```

Constructors

Destination

```
public Destination()
```

Creates a new Destination instance.

Destination

```
public Destination(String address)
```

Creates a new Destination instance defining its address.

Parameters:
address - java.lang.String

Methods

getAddress

```
public String getAddress()
```

Returns the address of a destination.

Returns:
java.lang.String

isAlternative

```
public boolean isAlternative()
```

Check whether the destination is alternative or not

Returns:
boolean

setAlternative

```
public void setAlternative(boolean isAlternative)
```

Sets a destination as alternative or not.

Parameters:
isAlternative - boolean

●**setVisited**

```
public void setVisited(boolean visited)
```

Sets a destination as visited or not.

Parameters:
visited - boolean

●**wasVisited**

```
public void wasVisited()
```

Check whether the destination was already visited or not.

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy This Package Previous Next Index

Class travel.TravelPlan

```
java.lang.Object
   |
   +----travel.TravelPlan
```

public class **TravelPlan**
extends Object
implements Serializable

This class defines the travel plan on an agent.

Field Index

- **agentHomeAddress**
- **destinations**
- **goAlternative**
- **index**
- **maxVisits**
- **timeOut**

Constructor Index

- **TravelPlan**()
  Creates a new TravelPlan instance.

Method Index

- **addDestination**(String)
  Adds a destination in the travel plan.
- **addOtherDestinations**(Vector)
  Adds other destinations in the travel plan that were obtained from the visited environment.
- **hasFinished**()
  Checks whether the travel has finished or not.
- **initializeTravel**(Vector)
  Initialize a travel with a list of address.
- **nextDestination**()
  Gets the next destination where the agent should go.
- **removeDestination**(String)
  Removes a destination from the travel plan.
- **resetIndex**()
  Sets the travel plan index to zero.
- **resetVisited**()

  Sets all destinations as not visited yet.
- **setAgentHomeAddress**(String)
  Sets in the travel plan the current agent address.
- **setGoAlternatives**(boolean)
  Sets the goAlternative parameter whether the agent should visit or not the alternative addresses.
- **setMaxVisits**(int)
  Sets the maximum number of visits per travel.
- **setTimeOut**(int)
  Sets the time out for a connection.

Fields

index

```
private int index
```

destinations

```
private Vector destinations
```

timeOut

```
private int timeOut
```

maxVisits

```
private int maxVisits
```

goAlternative

```
private boolean goAlternative
```

agentHomeAddress

```
private String agentHomeAddress
```

Constructors

TravelPlan

```
public TravelPlan()
```

Creates a new TravelPlan instance.

Methods

addDestination

```
public void addDestination(String address)
```

Adds a destination in the travel plan.

Parameters:
address - java.lang.String

●addOtherDestinations

```
public void addOtherDestinations(Vector otherDestinations)
```

Adds other destinations in the travel plan that were obtained from the visited environment.

Parameters:
otherDestinations - java.util.Vector

●hasFinished

```
public boolean hasFinished()
```

Checks whether the travel has finished or not.

Returns:
boolean

●initializeTravel

```
public void initializeTravel(Vector dest)
```

Initialize a travel with a list of address.

Parameters:
dest - java.util.Vector

●nextDestination

```
public String nextDestination()
```

Gets the next destination where the agent should go.

Returns:
java.lang.String

●removeDestination

```
public void removeDestination(String address)
```

Removes a destination from the travel plan.

Parameters:
address - java.lang.String

●resetIndex

```
public void resetIndex()
```

Sets the travel plan index to zero.

●resetVisited

```
public void resetVisited()
```

Sets all destinations as not visited yet.

●setAgentHomeAddress

```
public void setAgentHomeAddress(String AgentHomeAddress)
```

Sets in the travel plan the current agent address.

Parameters:
AgentHomeAddress - java.lang.String

●setGoAlternatives

```
public void setGoAlternatives(boolean goAlternative)
```

Sets the goAlternative parameter whether the agent should visit or not the alternative addresses.

Parameters:
goAlternative - boolean

●setMaxVisits

```
public void setMaxVisits(int maxVisits)
```

Sets the maximum number of visits per travel.

Parameters:
maxVisits - int

●setTimeOut

```
public void setTimeOut(int timeOut)
```

Sets the time out for a connection.

Parameters:
timeOut - int

All Packages Class Hierarchy This Package Previous Next Index