Integrating Agents in Software Applications

A.J.N. van Breemen

Software Architecture Group (SwA), Philips Research
Prof. Holstlaan 4 (WDC 1-034),
5656 AA Eindhoven, The Netherlands
+31 40 27 43440
albert.van.breemen@philips.com

Abstract. Once the decision is made of using agents in your software application you should decide how these agents are going to be implemented and integrated in your application. There are basically two options to implement agents. Either you use a dedicated agent platform or you implement the agents from scratch in some standard programming language. The choice between these options impacts the architecture of your application. This article will briefly discuss both options and will present object-oriented models to integrate agents in your application in case of the latter option being used. The presented models can be used as guidelines to integrate software agents in object-oriented software applications.

1 Introduction

Applying agents in your software application can give your application different potential values. In some cases, the value of using agents in your application can come from their special properties such as intelligence, adaptability, mobility or social behavior (see [1]). In other cases, the value comes from the fact that agents are intuitive abstractions and they can give your software solution a clear and effective structuring. In both cases, however, using agents has an impact on your software solution and therefore using agents becomes an architectural decision. In this paper we discuss some consequences of this decision and issues related to the integration problem of agents with other components of your software application.

Once the decision is made of using agents in your application you must address the problem of how to implement and integrate the agents into your application. There are basically two methods to implement agents. The first method is to use a dedicated open source or commercial agent platform. The second method is to directly implement your agents from scratch by using some standard programming language. Which method to use depends on the functional as well as nonfunctional requirements of the application and it brings you back to the question why you want to use agents.

When using agents in your software application one must implement both the internal architecture of the agents as well as ways to let the agents interact

with the remaining components of your software application. In this article we do not discuss the implementation of the internal architecture of agents. The internal architecture depends on the purpose of the agent. Sometimes an agent is used to provide an interface to existing software components and it wraps new or already existing software components (e.g. see [2]). Other times, an agent is used to realize software components with intelligent behavior and its architecture relies heavily on artificial intelligence techniques, such as neural networks, fuzzy logic or inference mechanisms. In this case the internal architecture can be efficiently implemented by using off-the-shelf artificial intelligence software libraries. Bigus & Bigus [3] describe a Java-based agent framework which contains several packages for implementing the internal architecture of agents based on artificial intelligence techniques. In other cases you just want to use dedicated agent architectures, such as a reactive or BDI architecture [1]. Vidal, Buhler & Huhns [4] present UML [5] models for such agent architectures.

The aim of this paper is to discuss the pragmatic problem of how to integrate agents in software applications and to present some object-oriented models to tackle this problem. We will show how these models relate to well known software design patterns described by Gamma et al. [6]. By this we hope that we have taken into account the remark of Odell, Van Dyke Parunak & Bauer [7] who state that two ways to reduce risk of deploying agent technology by potential adapters are to present the new technology as an incremental extension of some thrusted methods and to provide explicit engineering tools.

The remainder of this article is organized as follows. In the next section we discuss issues related to the choice of using a dedicated agent platform or starting from scratch using a standard programming language to implement agents. In section 3 some object-oriented models are discussed to integrate agents with other components of your software application. These models are applied to an example application in section 4. Finally, conclusions are presented in section 5.

2 Implementing agents

There are basically two ways to implement software agents. The first way is to use a dedicated open source or commercial agent platform. Many platforms are available nowadays, such as the *FIPA Open Source* (FIPA-OS) that was originally developed by Nortel Networks, the *Agent Development Kit* (ADK) developed by Tryllian, and the *Java Development framework* (JADE) developed at CSELT S.p.A. in Torino, Italy. The second way is to implement your agents from scratch using a standard programming language such as Java, C or C#. Several libraries are available that could help you to implement agents. For instance, Bigus & Bigus [3] provide with their book the *CAIgent* framework which contains a basic set of libraries to build agents in Java. Other, not particular agent libraries are *Jess* (Java Expert System Shell) and *CLIPS* (C Language Integrated Production System (CLIPS).[1]

[1] More information about these software packages and libraries can be easily found on the Internet by using the name as search key at your favorite search engine.

How should one decide on what implementation method to use? To answer this questions one must know some of the advantages and disadvantages of both methods. First, we start with discussing agent platforms. An agent platform is a software environment which provides resources and functionalities for software agents to operate. Some of these resources and functionalities are: a sophisticated communication infrastructure (e.g. JADE, FIPA-OS), support for agent mobility (e.g. ADK, Voyager), and reasoning engines (e.g. Jack, ABLE). Also, platforms provide you good monitoring and debugging tools.

The second implementation method — implementing agents from scratch in some programming language — requires to implement all agent functionalities yourself. It allows you to easily use non-agent and agent software components together. Developing an agent library is costly, but the copyrights are yours.

Each of the described implemented methods has a different impact on the software architecture of your application. When using an agent platform, often all components of your architecture are implemented as agents. Well meant structuring support (e.g. JADE behaviors) are often felt by developers as a restriction rather than a help. Therefore, platforms are good when your architecture must become an "agent-only" architecture. Implementing agents from scratch has less impact on the architecture of your software application as the developer can now decide which components should be modelled as agents and which not.

3 Object-Oriented Integration Models

In the previous section we discussed that when using a dedicated agent platform often you get an "agent-only" architecture instead of "an architecture with agents". In the remainder of this paper we are interested in the latter case and therefore we assume that you will implement your agents from scratch in your favorite programming language, possibly with the support of some special software library. This sections explains how to integrate your agents with other parts of your software application.

3.1 The Basic Agent Concepts

A good understanding of how agents interact with their environment helps in understanding how agents can be integrated within your software application. Therefore we start in this section with briefly recapturing the basic agent concepts.

An agent is an entity that is situated in some environment and it has the possibility to interact with this environment [1] (see figure 1). Interacting with its environment is a means for an agent to achieve the goals or desires it has.

A more detailed look at the different types of interactions reveals that we can distinguish two kinds of interactions. The first type of interaction is between the agent and some subject. The subject could be an object or an other agent. A subject is seen as an entity with attributes and a state, but without having goals and autonomous behavior. The interaction between the agent and the subject

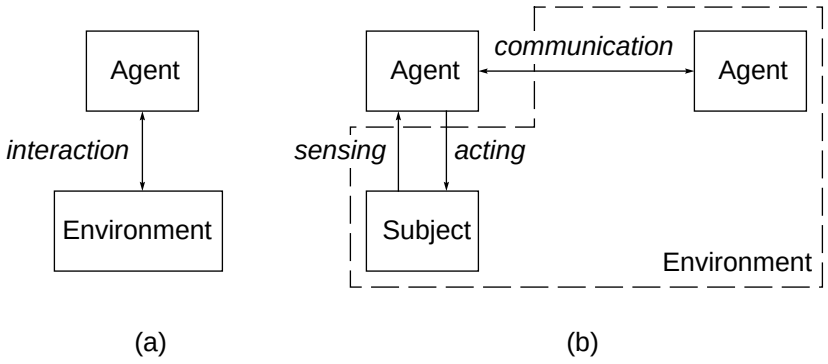


Fig. 1. a) An agent is situated in some environment. b) The environment consists of subjects which the agent can sense and on which it can act, and the environment contains other agents with which the agent can communicate.

consists of *sensing and acting*. Sensing is an activity of the agent to receive information from the subject, e.g. receiving values of the attributes and state of the subject. Acting is an activity of the agent to change the attributes or state of the subject.

The second type of interaction is between the agent and another agent. This type of interaction is called *communication*. Communicating differs from sensing and acting in the sense that it requires a particular commitment from all parties that are involved with respect to the communication protocols being used, the language being spoken, as well as the vocabulary or ontology being used [8]. Sensing and acting does not depend on such a commitment. When an agent senses a subject it has to translate the sensed data into some meaningful data for the agent. Also, when an agent acts on a subject, it has to translate the meaningful concepts of the agents into a proper change of attribute or state of the subject.

These two basic types of interactions between an agent and its environment are the basic mechanisms with which an agent can interact with other components of your software application. The next section explains how these two interaction types can be implemented.

3.2 Implementing Sensing

There are two different types of sensing. The first type is *active sensing*. In this case the agent takes the initiative when to observe the subject. The second type is *passive sensing*. In this case the subjects takes the initiative to send data to the agent. Both types of sensing requires a different implementation.

Figure 2 illustrates a class diagram of an implementation of active sensing. These are three classes: the `Agent`, `ActiveSensor` and `Subject` class. An `Agent`

object has zero or more `ActiveSensor` objects. Each `ActiveSensor` object has a reference to a `Subject` object.

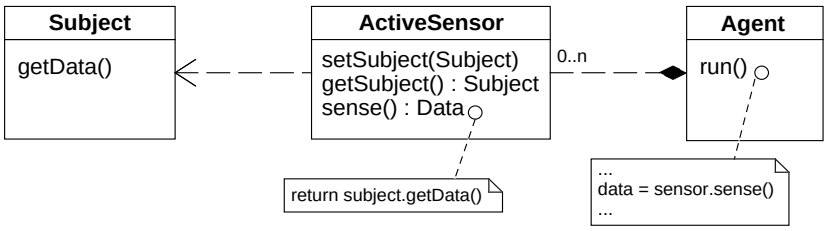


Fig. 2. Class diagram of active sensing implementation. This implementation resembles the Adapter software pattern described by [6].

The `Agent` class has a `run()` method which implements the main behavior of the agent. At some point during the execution of the `run()` method the agent will use a sensor so get data from a subject. It then calls the `sense()` method of the appropriate sensor. The `sense()` method will get the data from the subject and returns it to the agent.

This passive sensing implementation is similar to the *Adapter software pattern* described by [6]. An adapter is used when one wants to use an existing class, but its interface does not match with the one you need, or when you want to build a reusable class that cooperates with unrelated or unforseen classes. So, in some situations getting data from a subject may require more operations then just invoking a `getData()`-like function. These operations are implemented by the `sense()` method of `ActiveSensor`.

Figure 3 illustrates a class diagram of an implementation of passive sensing. This class diagram consists of three class: the `Subject`, `PassiveSensor` and `Agent` classes. The sensing interaction starts with a `Subject` object. At some point in time this object has new data that it sends to all `PassiveSensor` objects that are attached to it. A special method, called the `handleNewData()` method is used by the `PassiveSensor` object to handle the new data. There are basically two options to handle the new data. The `PassiveSensor` could store all data it receives in some buffer. Then, when the `Agent` object needs the new data it must call some method of the `PassiveSensor` object to get the data. The second option is that the `PassiveSensor` calls a method of the `Agent` class once it has received new data from a `Subject` object. The first option is somewhat similar to active sensing in that the `Agent` class takes the initiative to get data. The main difference is that the `PassiveSensor` class buffers the data received.

This passive sensing implementation is similar to the *Observer software pattern* described by [6]. The Observer software pattern is used in situations when changes in one object requires changes in an other object, or when an object

needs to notify other objects without making assumptions about who these objects are.

Of course, passive and active sensing could be combined in a more complex hybrid sensor.

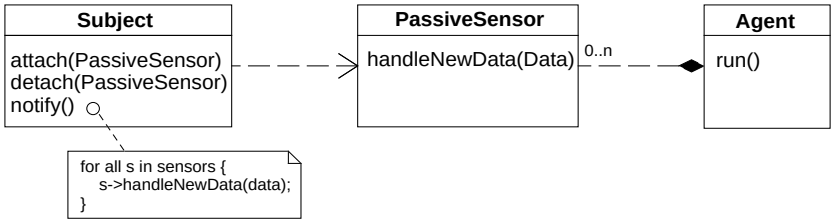


Fig. 3. Class diagram of passive sensing implementation. This implementation resembles the Observer software pattern described by [6].

Remark 1. In case a sensor is used to read data from a real physical sensor, the `Subject` class becomes a class that represents the memory of the computer system or some data acquisition card. The sensor will read data from the memory.

Remark 2. Active sensing can also be implemented by direct method invocation. If the subject is implemented at the same time as when the agent is implemented, this could be an efficient design choice. However, if the subject is an existing software component and its interface doesn't match with the interface the `Agent` class expects, using an `ActiveSensor` is an effective solution.

Also passive sensing could be implemented, for efficiency reasons, by direct method invocations between the `Subject` and `Agent` class. Just let the `Agent` class implement the `PassiveSensor` interface, such that it handles the `handleNewData()` method. A drawback of this design decision is that the agent becomes less structured and more complex.

3.3 Implementing Acting

Agents have effectors in order to act upon subjects in their environment. Acting modifies the attributes or state of the subject. Unlike sensing, acting is always initiated by the agent.

Figure 4 shows a class diagram of an implementation of acting. In this class diagram three classes are pictured: the `Agent`, `Effector` and `Subject` class. An `Agent` object has zero or more `Effector` objects. Each `Effector` object has a reference to a `Subject` object.

The main behavior of the agent is once again implemented by the `run()` method of the `Agent` class. At some point in time, the `Agent` object needs to

act upon some subject. It therefore calls the `act()` methods of the appropriate `Effector` object. The `Effector` object will then call the `someMethod()` method of the `Subject` object.

Just like the implementation of active sensing, this implementation of acting is similar to the *Adapter software pattern* [6]. Sometimes, the `Effector` object has to call several methods of the `Subject` object in order to act. This should all be implemented by the `act()` method.

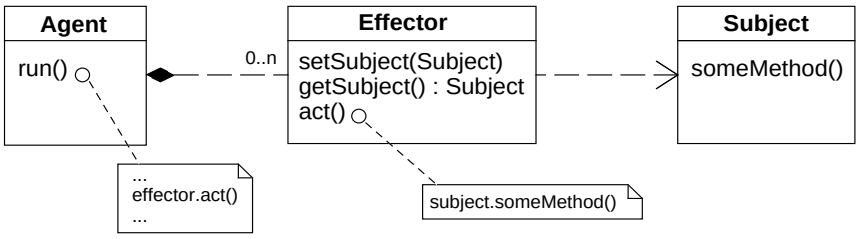


Fig. 4. Class diagram of acting implementation. This implementation resembles the Adapter software pattern described by [6].

Remark 1. In case the effector is used to control a real physical effector, the `Subject` class becomes a class that represents the memory of the computer system or some data acquisition card. The effector will write data to memory.

Remark 2. Acting can also be implemented, for efficiency reasons, by letting the `Agent` class directly invoke methods on the `Subject` class. However, if the subject is an existing software component and its interface doesn't match with the interface the `Agent` class expects, using an `Effector` is an effective and better structured solution.

3.4 Implementing Communicating

The previous two sections described the sensing and acting interaction type of agents. The second type of interaction is communicating. Communicating differs from sensing and acting in the sense that all communicating parties (agents) commit themselves to some communication protocol, communication language and ontology [8]. This has consequences for the architecture of the software application as these commitments cause dependencies between the software agent components.

In order to communicate agents must be able to send and receive messages. We realize this by using a message dispatcher. Figure 5 shows a class diagram to implement agent communication. This class diagram contains four different classes: the `Agent`, `Communicator`, `Dispatcher` and `Message` class.

Each `Agent` object has a unique `AgentID`. In order to receive messages an `Agent` object must register itself at the `Dispatcher` object. The `Dispatcher` object stores all the `AgentId`'s.

The `Communicator` is used by the `Agent` object to send and receive `Message` objects. The `Communicator` object has a queue which stores all received `Message` objects. When the `send()` method of the `Communicator` object is invoked, it will call the `dispatch()` method of the `Dispatcher` object. The `dispatch()` method looks up the address of the receiving `Agent` object by using the `AgentID` as search key. If the address is found, the `Dispatcher` will put the `Message` object in the message queue of the `Communicator` object of the receiving `Agent` object.

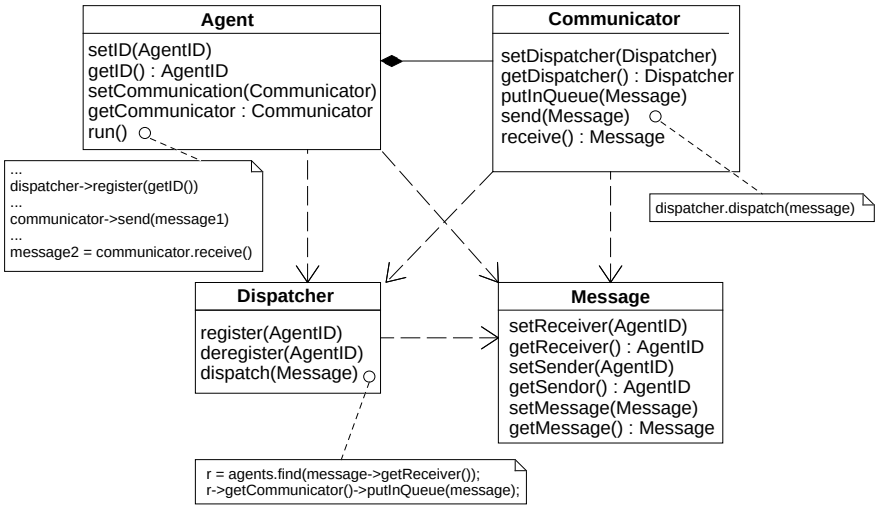


Fig. 5. Class diagram of communicating implementation.

Remark 1. When starting to implement an agent communication infrastructure one must start to become suspicious. One can spend a lot of time in building communication infrastructures. If your agents relies heavily on a good communication infrastructure, using an off-the-shelf platform or library might be an effective option. Many platforms and libraries provide you a basic communication infrastructure. This remark, in fact, is pitfall 7.4 "you spend all your time in implementing infrastructure" described by [9].

Remark 2. Communicating can also be implemented for efficiency reasons by direct method invocations. However, by using a dispatcher and agent id's, the software agent components become more loosely coupled. Furthermore, by using a communicator you can easily modify your application in case you want to communicate over some network: just modify the `Dispatcher` object.

4 An Application Example

This section demonstrates how to use the previously presented object-oriented models to integrate agents in a software application. First we describe the application example and subsequently a class diagram of the proposed solution is presented.

4.1 Application Description

The application example we consider is that of a mp3-music application. A user can use this application to play mp3-files and the application has an option to generate a playlist that fits to the user's music playing profile.

The application is built by two software components. The first component is a mp3-player. The second component is a playlist generator. Figure 6 illustrates these components and the user together with their interactions.

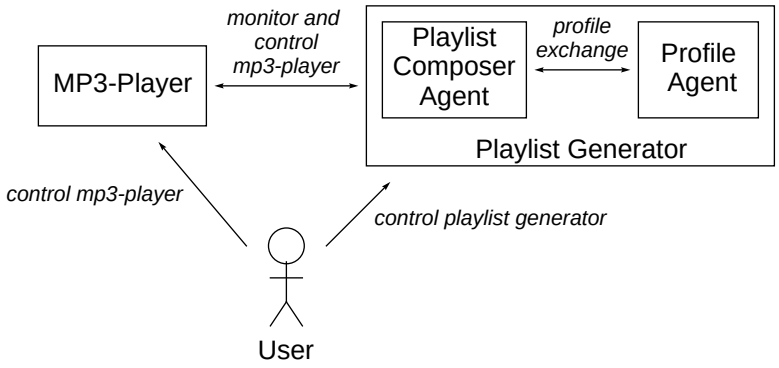


Fig. 6. Basic components of mp3-music application.

The two components are structured by objects. The mp3-player becomes a single object. The playlist generator, however, will be composed of two agents that are implemented by objects. The first agent has a responsibility to monitor the mp3-player in order to watch the mp3-files the user is playing and to translate this information into a "music-playing-profile". The second agent is responsible to compose a playlist on request and to play the mp3-files on this list.

4.2 Implementation

We provide the mp3-player and playlist generator components with user interfaces with which the user interacts. Therefore, we only consider the interactions

and implementation of the mp3-player and the profile and playlist composer agents.

Figure 7 shows a class diagram of our application example. The `ProfileAgent` object needs to monitor the `MP3Player` object. Each time the user starts playing a mp3-file by the `MP3Player` object the `ProfileAgent` object must register this. Mp3-files contain meta-data (ID3 tag, see [10]), such as artist and genre information. It is most natural to model the `MP3Player` object as taking the initiative to send information to the `ProfileAgent` object when a mp3-file is being played by a user. Because we do not consider the `MP3Player` object as an agent, we do not use communication. Rather, we use a sensor. In particular, passive sensing is used as the `MP3Player` object takes the initiative to send information. Also, the `PlaylistComposerAgent` object needs to control the `MP3Player` object in order to play files. For the same reasons as described above we do not use communication. Rather, we use an effector.

The `ProfileComposerAgent` object needs profile information about the user's playing behavior from the `ProfileAgent` object. This interaction is modelled as communication. The profile information needs to be presented in some agent communication language and based on an ontology to which both agents are committed.

5 Conclusions

In this paper we discussed the pragmatic problem of how to integrate agents in software applications. We discussed the impact on the software architecture and presented some object-oriented models to integrate agents in software applications. These models were related to known software design patterns.

We discussed the problem of whether to use a dedicated agent platform or not. The decision depends on the requirements of your application and the reason why you want to use agents. Using a platform often results in "agent-only" architectures whereas building agents from the scratch gives you the possibility to combine agent and non-agent software components into one architecture more easily.

Applications that contain agents need to address the problem of how these agents interact with the remaining components of that application. Two approaches were described. The first approach consists of sensing and acting. This approach uses sensor and effector objects as a means to interact with other software components. The second approach consists of communicating. Agents that communicate must be committed to the same communication protocol, agent communication language and ontology. For both types of interaction object-oriented models were presented to implement them. It was discussed that these models resembles the well known Adapter and Observer software patterns.

The usages of these object oriented models have been demonstrated on the application example of a mp3-music application.

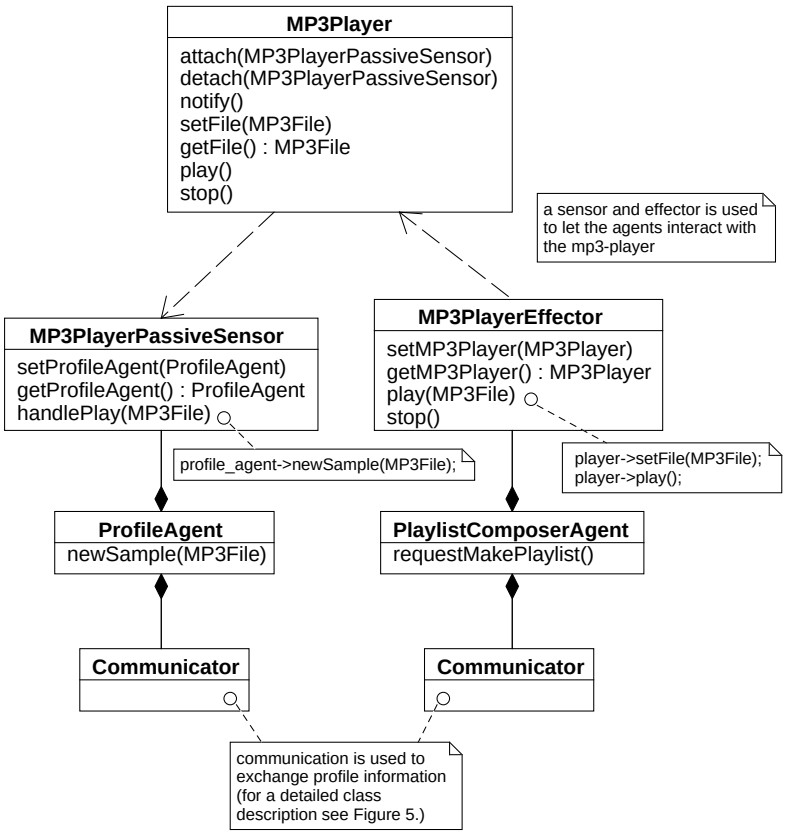


Fig. 7. Class diagram of mp3-music application using the object-oriented models for agent interaction that were presented in the previous section.

References

1. Wooldridge, M.: An Introduction to Multiagent Systems, John Wiley & Sons, Chichester, England (2002)
2. Jennings, N.R., Corera, J., Laresgoiti, I. Mamdani, E.H., Perriolat, F., Skarek, P., Varga, Z.: Using ARCHON to develop real-world DAI applications for electricity transportation management and particle accelerator control, IEEE Expert, December (1996)
3. Bigus, J.P., Bigus, J.: Constructing Intelligent Agents using Java, second edition, John Wiley & Sons (2001)
4. Vidal, J.M., Buhler, P.A., Huhns, M.N.: Inside an Agent. IEEE Internet Computing, January-February (2001)
5. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1999)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison Welsey (1995)
7. Odell, J., Van Dyke Parunak, H., Bauer, B., Extending UML for Agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence, Austin, Texas, july, 30 2000. ICue Publishing.
8. Huhns, M.N., Stephens, L.M.: Multiagent Systems and Societies of Agents. In G. Weiss, editor, Multiagent Systems: a modern approach to distributed artificial intelligence, MIT Press (1999)
9. Wooldridge, M., Jennings, N.R.: Pitfalls of Agent-Oriented Development. In Proceedings of the Second International Conference on Autonomous Agents (Agents 98), pages 385–391, Minneapolis/St Paul, MN, May (1998)
10. Hacker, S.: MP3: The Definitive Guide. O'Reilly (2000)