Note

A similar version of the following paper has been tentatively scheduled for publication in the May 2006 issue of the International Journal of Computer Systems Science & Engineering special issue on "Software Engineering for Multi-Agent Systems". The methodology has been developed in conjunction with Magid Nikraz of Murdoch University in Western Australia as part of his PhD project. Comments and suggestions are welcome – please forward them to Magid at m.nikraz@murdoch.edu.au.

A Methodology for the Analysis and Design of Multi-Agent Systems using JADE

Magid Nikraz[1][a], Giovanni Caire[b], and Parisa A. Bahri[a]

[a]School of Engineering Science and Parker Center, Murdoch University, Dixon Road, Rockingham, Western Australia 6168

[b]Telecom Italia Lab, Via Reiss Romoli, Turin, Italy 10148

Abstract

A methodology is proposed for the development of multi-agent systems using the JADE platform. The proposed methodology focuses on the key issues in the analysis and design of multi-agent systems. The analysis phase is generic in nature, while the design phase specifically focuses on the constructs provided by the popular FIPA-compliant JADE platform. The methodology essentially serves as a guide, providing a direction for the multi-agent system designer, while also giving them the opportunity to add or remove components as desired, based on the specific problem domain. In contrast to current methodologies, support is also provided for existing systems and people to be included and accounted for in the multi-agent system.

The methodology is illustrated by applying it to a hypothetical but illustrative scenario. Moreover, it is presented in such a way that designers new to the field of agent-based computing and the JADE platform can quickly grasp the most important concepts in the development of a multi-agent system. Similarly, people currently familiar with the JADE platform can also benefit from the methodology, which, when applied, should lead to a significant reduction in system development time.

Keywords

Agent, methodology, analysis, design.

---

[1] To whom all correspondences should be addressed. Tel.: +61 8 9360 7135, fax: +61 8 9360 7104. Email address: m.nikraz@murdoch.edu.au.

1.   Introduction

There are four fundamental phases to the software development lifecycle: planning, analysis, design, and implementation. When implementing the software development lifecycle, it is often useful to have some formal guidelines (i.e. a list of steps and deliverables) on how to progress through these phases. This is the task of a methodology [9]. A methodology saves time and effort by crystallizing the important steps that the designer should follow, and as a result, providing them with the right "direction". A methodology, thus, essentially acts like a "recipe," which helps the designer to implement the solution by specifying some of the steps of the process, while leaving others to the creativity of the designer [3]. The importance of a methodology in the software development cycle can, therefore, not be overstated.

Agent-based software engineering is a relatively new field and can be thought of as a evolution of object-oriented programming [24]. Though agent technology provides a means to effectively solve problems in certain application areas, where other techniques may be deemed lacking or cumbersome, there is a current lack of mature agent-based software development methodologies. This deficiency has been pointed out as one of the main barriers to the large-scale uptake of agent technology [20]. Thus, the continued development and refinement of methodologies for the development of multi-agent systems is imperative, and consequently, an area of agent technology deserving significant attention.

Current methodologies exist for the development of multi-agent systems including Gaia [35], MESSAGE [5], and Cassiopeia [7]. Some good reviews are provided in [3, 19, 34]. Most current methodologies attempt to adapt object-oriented analysis and design methodologies to agent-based design [34], and in addition, follow a top-down approach. It has been pointed out that adapting object-oriented analysis and design methodologies to multi-agent system development has several disadvantages [34], mainly arising from the fact that objects and agents provide different abstractions, and as a result, should be thought at different levels [24]. In addition, the wholly top-down approach assumed by many of the current methodologies is not sufficient for systems containing existing resources which need to be utilized within the multi-agent system.

The proposed methodology does not attempt to extend object-oriented techniques, instead focusing on agents specifically and the abstractions provided by the agent paradigm. Furthermore, it combines a top-down and bottom-up approach so that both existing system capabilities (including those provided by legacy software and people) and the applications overall needs (based on the requirements) can be accounted for. As mentioned above, not explicitly accounting for existing systems is a point lacking in many of the currently available methodologies for multi-agent system development. The proposed methodology attempts to formalize the analysis and design phases of the agent-based software development life cycle. The formalization of the planning and implementation phases of the software development life cycle are currently outside the scope of the methodology, though some brief pointers are given.

The design phase specifically focuses on the JADE platform, and the concepts provided by it. JADE[2] is the abbreviation for the Java Agent DEvelopment Framework and has been developed by the Telecom Italia Lab (TILAB) in Italy, in compliance with the FIPA (Foundation for Intelligent Physical Agents) specifications [10]. FIPA is a non-profit organization geared at producing standards for the interoperation of heterogeneous agents. Essentially, JADE is a middle-ware (written entirely in the Java language, using several Java technologies), which simplifies the implementation of multi-agent systems by providing a set of graphical tools that support the debugging and deployment phases. The agent platform can be distributed across multiple machines, regardless of the underlying operating system, and the configuration controlled via a remote graphical user interface. More information on JADE can be found at [15]. By specifically focusing on the JADE platform in the design phase, the designer can move straight to implementation afterwards, without having to tediously adapt the results of the design phase to an agent platform of their choice. This will obviously result in significant time gains for the designer, in addition to providing them with a much clearer picture on how to progress in implementation.

It is in the early stages of the software development cycle (i.e. planning), where it is decided which tool (e.g. which programming paradigm) to use, and an assessment is made on whether an agent-based option (among other options) is the most appropriate solution tool. As mentioned, the proposed methodology does not formally cover planning, and assumes that a decision has been made to use an agent-based solution. However, it should be pointed out that the decision to use an agent-based solution should be well thought out, since it may not always be the best option. To help in making this decision, one should consult the literature, and in particular, applications where agent technology has been successfully applied. Some good sources which provide tips and guidelines on when an agent-based solution should (or should not) be used are [2, 18], and [21], in particular. Furthermore, some potential pitfalls associated with agent-based development, which should be taken into consideration when developing or contemplating such a system, are outlined and discussed in [17, 36]. These valuable references will save the designer a lot of time and effort, by preventing the unnecessary implementation of an agent-based solution, or, on the other hand, confirming the appropriateness of an agent-based solution, for a particular case.

The proposed methodology is presented as follows: Section 2 gives an overview of the methodology, while also outlining the assumed definition of an agent (Section 2.1) and the hypothetical example used to illustrate the methodology (Section 2.2). Section 3 outlines the steps in the analysis, while Section 4 outlines the steps in the design. Section 5 gives some brief indicators on the post design stage. Section 6 gives details on how the methodology could be adapted to other agent development platforms. Finally, in Section 7, some conclusions are presented, and further work discussed.

---

[2] JADE is a software distributed by TILAB, the copyright holder, in open source under the terms of the LPGL (Lesser General Public License Version 2).

2.   Methodology Overview

As described in the Introduction, a methodology serves as a guide for the system designer when developing a system. In general, a software development methodology may comprise of:

- A process, i.e. a sequence of phases and steps that guide the developer in building the system.
- A set of heuristic rules that support the developer in making relevant choices.
- A number of artifacts, i.e. diagrams, schemas or documents representing in graphical or textual form one or more models of the system.
- A suitable notation to be used in the artifacts.
- A set of patterns that can be applied to solve common situations.
- One or more tools that: automate, as much as possible, the phases and steps specified in the process; force consistency between the models produced; highlight problems arising from incorrect design choices, when possible; generate code and documentation, etc.

The current focus of the proposed methodology is on the process and the artifacts that are produced, illustrating them through an example (detailed in Section 2.2). A draft notation is also introduced to be used in constructing these artifacts and, where relevant, some heuristic rules and design patterns are presented. The described process covers the analysis phase and the design phase and is shown in Figure 1. The analysis phase is general in nature and independent of the adopted platform. Conversely, the design phase specifically assumes JADE as the implementation platform and focuses directly on the classes and concepts provided by JADE (for other platforms, see Section 6). Observing Figure 1, it can be seen that there is no strict boundary between the analysis and design phases. Moreover, the methodology is of an iterative nature, thus allowing the designer to move back and forth between the analysis and design phases and the steps therein.

At the end of the design phase, the developer should be able to progress straight to the implementation, which is where the actual coding occurs. In addition, most of this phase can probably be carried out by means of a proper tool which automates the implementation process. However, this point, and likewise, testing and deployment, are not addressed formally in the current version of the proposed methodology (as shown in Figure 1), leaving them as a topic for further work[3] (see Further Work, Section 7).

The planning stage, like implementation and testing, is not formally addressed in the proposed methodology. However, for the sake of the methodology, a question is included (see Figure 1), which initially asks if the designer has made a rational decision on whether to use an agent-based solution. If the answer is yes, the designer moves on the analysis, while if the answer is no, the designer should seek an alternative solution[4]. As

---

[3] Some informal pointers are provided in Post Design (Section 5).

[4] Obviously planning, which is the first phase in the software development life cycle, will entail many other considerations. For the sake of analysis and design though, the only assumption required in the proposed methodology is that an agent-based solution has been chosen as the best alternative.
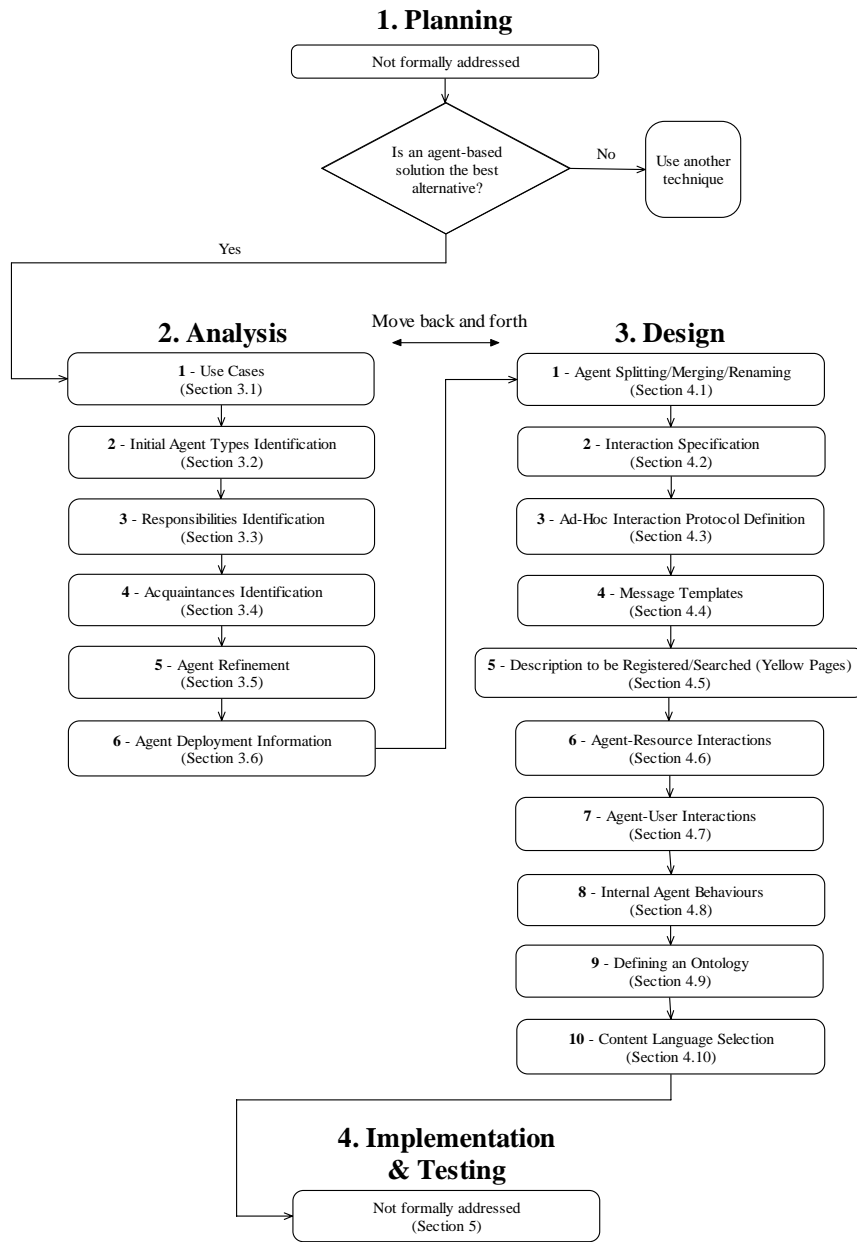
Figure 1. Overview of the methodology.

mentioned in the Introduction, the decision on whether to adopt an agent-based solution is one which should be made by the designer only after consulting the literature, and, if possible, analyzing their problem with respect to some guidelines (such as in [21]). Furthermore, observing problem domains and cases where an agent-based solution has been implemented effectively, is also another good indicator of the appropriateness of an agent-based solution.

As with any methodology, some assumptions must be made. These include:
- The definition of an agent defined in Section 2.1 is assumed.

- The JADE platform is the platform of choice for implementation.
- There are a relatively small number of agents (less than 100).
- The organizational structure of system is static, meaning that non-emergent behaviour at runtime is not expected, and thus, not considered.
- No specific target domain is assumed.
- Security is not a concern.

2.1   What is an Agent?

The term agent is very broad and has different meanings to different people. However, on close observation of the literature, it is sufficient to say that two usages of the term agent can be identified: the *weak* notion of agency and the *strong* notion of agency [37]. The weak notion of agency constitutes the bare minimum that most researches agree on, while the stronger notion of agency is more controversial and a subject of active research.

The weak notion of agency denotes a software-based computer system with the following properties [37]:
- *Autonomy*: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state.
- *Social ability*: agents interact with other agents (and possibly humans) via some kind of agent communication language.
- *Reactivity*: agents perceive their environment and respond in a timely fashion to changes occurring therein.
- *Pro-activeness*: in addition to acting in response to their environment, agents are able to exhibit goal-directed behaviour by taking the initiative.

The strong notion of agency is an extension of the weaker notion, and advocates additional humanistic, mental properties such as belief, desire, and intention [29].

Consistent with the weak notion of agency, one author [13] has gone so far as to say that software agents are application programs that communicate with each other in an expressive agent communication language. Though at first this definition may seem a little simplistic, it allows one to clearly identify what constitutes a multi-agent system, i.e. agents are just pieces of autonomous code, able to communicate with each other using an agent communication language. The view of agents assumed in the proposed methodology is based on this definition. Specifically, the methodology assumes the following definition for an agent:

> agents reside on a platform that, consistent with the presented vision,
> provides the agents with a proper mechanism to communicate by names,
> regardless of the complexity and nature of the underlying environment
> (i.e. operating systems, networks, etc).

Thus, the assumed view is exactly the same as that presented in [13], but in addition, the agents have unique names as a means of identification.

This particular view of agents is the only assumption for analysis, while the design is specific to the JADE platform, which is a FIPA-compliant realization of the above vision, i.e. in the design phase, the constructs provided by the JADE platform are assumed.

2.2. The Cinema Organizer Scenario

The proposed methodology is illustrated by applying it to a simple scenario referred to henceforth as the cinema organizer case study.

In the cinema organizer case study, it is assumed that a mobile telecom operator wants to provide a service to its subscribers (cinema organizer users), which will support a group of friends in organizing an evening to the cinema. The service should allow a subscriber to: invite some friends to see a movie, collect preferences from both the inviter and the invitees, and suggest the option that best matches the average preferences of the group.

The service must be accessible by users through their mobile phones. It is also assumed that the mobile operator has proper agreements with one information provider for each city the cinema organizer service is going to be offered in. These providers will make available all the information related to the cinemas in the local city, and the schedules of movies in these cinemas.

Users can move freely between cities and the system must account for this. User localization (i.e. where the user is situated) must be based on the mobile operator localization system. The system allows for the retrieval of the position of a mobile phone given its telephone number and notification of relevant changes in the mobile's position.

3. Analysis

The analysis phase aims to clarify the problem without any (or minimal) concerns about the solution. In the proposed methodology, the analysis phase is carried out through a number of steps, described in Sections 3.1-3.6.

3.1 Step 1: Use Cases

Use cases are an effective way to capture the potential functional requirements of a new system. Each use case presents one or more scenarios that demonstrate how the system should interact with the end user or another system to achieve a specific goal. There are a number of standards for representing use cases. The most popular is the Unified Modeling Language (UML) specification [33], which defines a graphical notation (as an alternative, it is also possible to produce written use cases). Though use cases are used extensively by object-oriented practitioners, their applicability is *not* restricted to object oriented systems, because they are not object orientated in nature [14]. Hence, it is also possible to apply use cases (without modification) to capture the functional requirements of multi-agent systems.

Based on the description of the cinema organizer case study given in Section 2.2, and after interviewing the potential system users, it is possible to build up a preliminary list of possible scenarios. Accordingly, the use cases can be defined, and a *use case diagram* produced as shown in Figure 2.
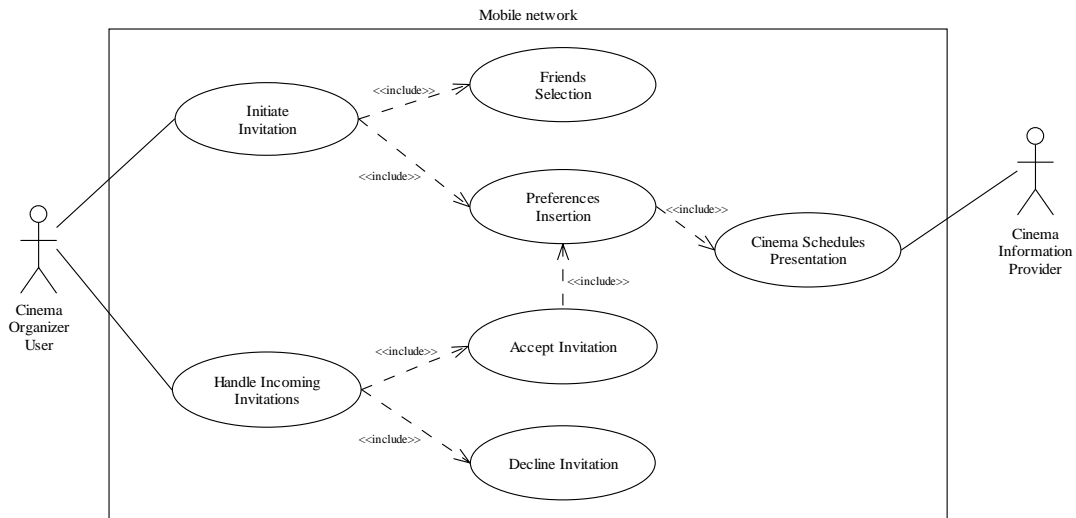
Figure 2. Use case diagram for cinema organizer case study.

3.2   Step 2: Initial Agent Types Identification

This step involves identification of the main agent types and subsequent formation of a first draft of the *agent diagram*. The following rules should be applied in this step:
- Add one type of agent per user/device.
- Add one type of agent per resource (which includes legacy software).

By applying the above rules to the cinema organizer case study, the initial diagram shown in Figure 3 is obtained.

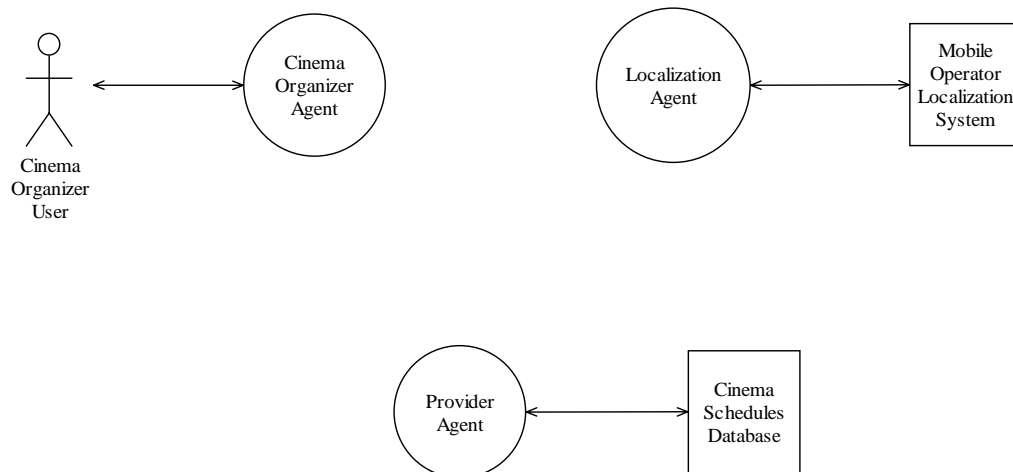The agent diagram is one of the main artifacts produced in the analysis phase and is

Figure 3. Agent diagram for cinema organizer case study after Step 2.

progressively refined from Steps 2 to 5. With reference to Figure 3, the agent diagram includes four types of elements:

1. *Agent types*: the actual agent types, represented by circles.
2. *Humans*: people that must interact with the system under development, represented by the UML actor symbol.
3. *Resources:* external systems that must interact with the system under development, represented by rectangles.
4. *Acquaintances:* represented by an arrow linking instances of the above elements, specifying that the linked elements will have to interact in some way while the system is in operation. Note that, at this stage, only acquaintances between agents and resources/humans are shown in the agent diagram (i.e. agent-agent interactions are deferred to a later step).

It should be noted that in the agent diagram, unlike UML use case diagrams, a distinction is made explicitly between humans and external systems. Interacting with a human through a user interface presents additional problems with respect to interacting with an external system as is highlighted in Steps 6 and 7 of the design phase (Sections 4.6 and 4.7, respectively).

The way external/legacy systems and people that interact with the agents are accounted for in a multi-agent system, is an important consideration (and one that is lacking in many currently available methodologies, as described in the Introduction). One author has defined three techniques (see Figure 4) to account for such entities [13]:

- The use of a *transducer agent*. The transducer agent serves an interface between a legacy system and the other agents in the system. The transducer agent accepts messages from the agents in the system (in agent communication language), translates them into the legacy systems native language, and forwards these equivalent messages to the legacy system. Similarly, in the reverse direction, the transducer agent receives the legacy systems responses and makes them available to the other agents in the system. In addition to acting as an interface between agents and legacy software, the transducer approach also works for other resources such as files and people (more details given in Section 4.6 and 4.7, respectively).
- The insertion of a *wrapper*. A code is injected into the legacy resource (i.e. software in this case), provided the legacy resource's code is available. This inserted code will allow the resource to communicate in agent communication language, thus, converting it into an agent.
- *Rewriting* of the code. This is the most extreme approach, which involves rewriting the code to mimic (and possibly extend) the operation and capabilities of the legacy resource (i.e. software in this case), but with the added ability to communicate in agent communication language. Note that this approach is usually the last resort, when no other options (i.e. the use of a transducer or a wrapper) are deemed practical.
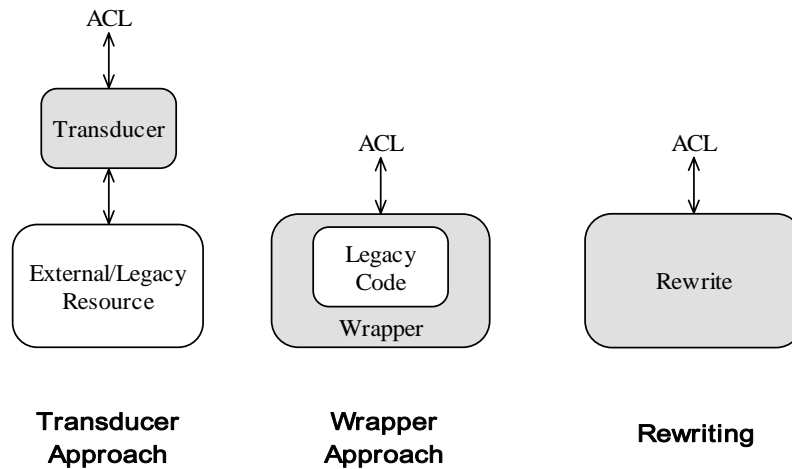
Figure 4. Different approaches to account for external/legacy systems.

In the agent diagram produced in this step (see Figure 3), the agents are acting as transducers, i.e. as an interface between the external/legacy systems/people, and the other agents in the system. Transducers are seen, in general, as the most practical and efficient method for accounting for legacy systems and are advocated in the proposed methodology. The reason is that by treating the legacy systems as a black-box, there is no need to tamper with or rewrite code, thus providing a quick means to get that resource functioning as part of the multi-agent system (though translating between agent communication language and the resources language is not always trivial). However, in some cases a wrapper may be more relevant, and to an extreme rewriting, but such considerations should be deferred to the design stage (i.e. in the analysis it suffices to assume the transducer approach).

3.3   Step 3: Responsibilities Identification

In this step, for each identified agent type, an initial list is made of its main responsibilities in an informal and intuitive way. The artifact resulting from this process is the *responsibility table*.

The following rules should be applied in this step:
- Derive the initial set of responsibilities from the use cases identified in Step 1.
- Consider the agents where these responsibilities are clearer first and delay the identification of responsibilities for other agents to later steps.

By applying the above rules to the cinema organizer case study, the consideration of the cinema organizer agent is initiated and Table 1 is produced.

Many existing methodologies such as Gaia [35] and MESSAGE [5] propose a different approach where atomic roles (roughly equivalent to responsibilities defined in this step) are initially identified and then possibly merged into agent types. However, this approach is considered less intuitive because in some cases it may become difficult to determine how the atomic roles should be aggregated into agent types, i.e. how many agent types

Table 1. Responsibility table for cinema organizer case study after Step 3.

Agent type	Responsibilities
Cinema organizer agent	Serve requests to initiate invitations from the cinema organizer user. Let the cinema organizer user select friends to invite. Let the user insert preferences about movies and cinemas. Present cinema schedules. Respond to invitations from other cinema organizer agents.

there should be and which type should cover which atomic role(s). The definition of agent types then responsibilities, as in the proposed methodology, removes this ambiguity.

3.4 Step 4: Acquaintances Identification

In this step, the focus is on who needs to interact with whom and the agent diagram (Figure 3) is updated by adding proper acquaintance relations connecting agents that need to have one or more interactions. The term acquaintance comes from Gaia [35], and is used in the same sense in the proposed methodology.

An obvious acquaintance relation in the cinema organizer case study is required between different cinema organizer agents: the inviter and the invitees. Then, since a cinema organizer agent must present cinema schedules to its user and this information is stored in the provider's database and made available by the relevant provider agents, there will certainly be an acquaintance relation between the cinema organizer agent and the provider agent. Thus, going one step backward (to Step 3, Section 3.3), some new responsibilities can be added to the cinema organizer agent and the provider agent. For the cinema organizer agent, these are:
- Present (to the user) incoming invitations from other cinema organizer agents.
- Let the user accept an incoming invitation.
- Let the user reject an incoming invitation.
- Retrieve cinema schedules from the relevant provider agent.

For the provider agent, the new responsibility is:

Respond to cinema schedules retrieval requests from cinema organizer agent.

Therefore, the agent diagram is updated as in Figure 5, and the responsibility table updated as in Table 2. Note that no distinction is made between acquaintances and responsibilities, so that all acquaintances are placed in the responsibility table and not separated.
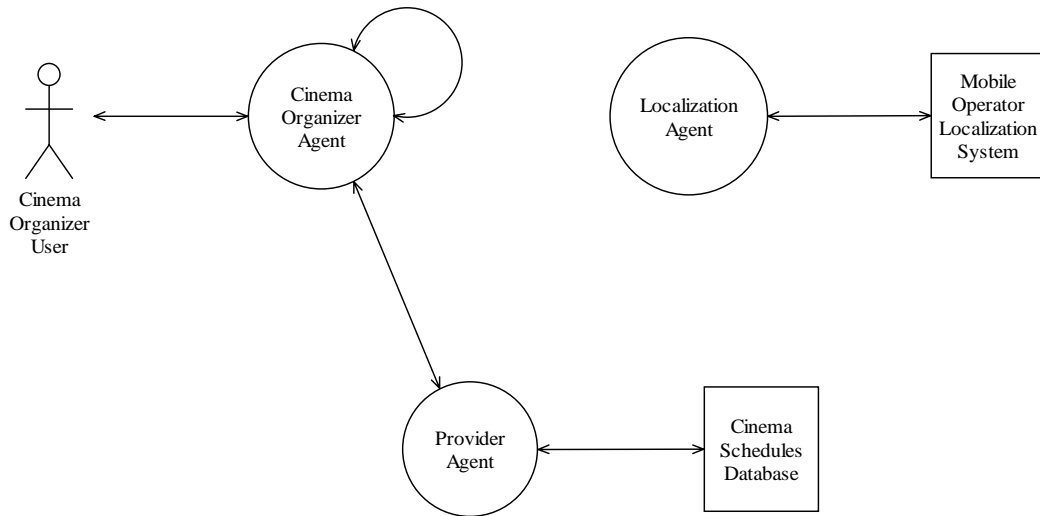
Figure 5. Agent diagram for cinema organizer case study refined after Step 4.

Table 2. Responsibility table for cinema organizer case study updated after Step 4.

Agent type	Responsibilities
Cinema organizer agent	Serve requests to initiate invitations from the cinema organizer user. Let the cinema organizer user select friends to invite. Let the user insert preferences about movies and cinemas. Present cinema schedules. Present incoming invitations from other cinema organizer agents. Let the user accept an incoming invitation. Let the user reject an incoming invitation. Retrieve cinema schedules from the relevant provider agent.
Provider agent	Respond to cinema schedules retrieval requests from cinema organizer agents.

3.5   Step 5: Agent Refinement

In this step, the set of agent types initially identified in Step 2 (see Section 3.2) are refined by applying a number of considerations. These are related to:

- *Support:* what supporting information agents need to accomplish their responsibilities, and how, when and where is this information generated/stored.
- *Discovery:* how agents linked by an acquaintance relation discover each other.
- *Management and monitoring:* is the system required to keep track of existing agents, or the starting and stopping of agents on demand.

These above considerations are discussed in more detail in Sections 3.5.1, 3.5.2, and 3.5.3, respectively.

3.5.1 Support

These considerations are highly dependent on the domain, and hence, it is quite difficult to provide generic indications. In the cinema organizer case study, the only additional information with respect to what has already been taken into account is that the cinema organizer agent requires the city it is currently in. Thus, the cinema organizer agent is required to question the provider agent for details of the current city when retrieving the cinema schedules information. This information, however, can be obtained by accessing the localization system through the localization agent and therefore there is no need for any new agent type in this case. Therefore, going back to Step 4 (Section 3.4), an acquaintance relation is added between the cinema organizer agent and the localization agent (see Figure 6), and (back again to Step 3, Section 3.3) new responsibilities are added to the cinema organizer agent and the localization agent: "retrieve current city from the localization agent" and "respond to city retrieval requests from cinema organizer agents," respectively (see Table 3[5]).

3.5.2 Discovery

In the simplest case, agent discovery can be accomplished by means of proper *naming conventions*. For example, in the cinema organizer case study, it can be assumed that the cinema organizer agent acting on behalf of a user and running on his or her mobile phone is named using the user's telephone number. This immediately allows a cinema organizer agent to identify the other cinema organizer agents to send an invitation to when its user selects some friends to organize an evening at the cinema. Adopting naming conventions is very simple and efficient, but has some limitations:

- Agent names must be globally unique. Thus, while a convention based on telephone numbers is typically acceptable, a convention based on people's names may often lead to duplication problems.
- Agents which are going to be involved in an interaction must typically be known in advance. For example, it can also be assumed that the localization agent has a predefined name "localizer". This works well provided that it is known in advance that there is one, and only one, such agent.
- Assuming naming conventions is typically not very extensible. Referring to the cinema organizer case study, assume that the provider agent for a given city is named after that city, e.g. the provider agent for CityX is called CityXProvider agent. In this way, given a city, the provider agent to retrieve cinema schedules for that city is immediately known. If it was desired to extend the system so that there can be more providers for each city, or a given provider can cover more that one city, clearly the proposed naming system would no longer work.
- Naming conventions may lead to additional work when applied to an agent that can appear and disappear dynamically. The reason being that a naming convention does not provide any presence information, and therefore, addressed agents may not be available when an attempt is made to contact them.

---

[5] Responsibility 9 for cinema organizer agent and responsibility 1 for localization agent.

- Naming conventions cannot be adopted when different users may start their own agents and choose names themselves. In such cases, there is no guarantee that name uniqueness is preserved.

A more sophisticated way to solve the agent discovery problem is the adoption of a *yellow pages mechanism*. This allows discovery of agents on the basis of their characteristics, e.g. the services they provide. A yellow pages mechanism can be fully distributed across all agents in the system or centralized with a single agent (with a well-known name) responsible for it. Even if this choice, at this point, is a high level design choice, considering that the proposed methodology targets the JADE platform, it is strongly suggested to adopt a centralized approach. This approach completely maps to the directory facilitator agent provided by JADE and thus saves a lot of work in successive phases of the development process.

In the cinema organizer case study, it is assumed that there are some extensibility requirements so that the "one provider per city" relation may no longer be true in the future. A yellow pages mechanism is thus adopted for the discovery of provider agents and a yellow pages agent is added to the agent diagram (see Figure 6).

For more information about service discovery mechanisms (including centralized and decentralized solutions) in multi-agent systems, refer to [6, 30].

3.5.3   Management and Monitoring

Other agent types can be added to address issues such as monitoring agent faults and restoring them, creation of supporting agents that are needed only under certain conditions, or providing presence information. No new agent types need to be added in the cinema organizer case study for management and monitoring purposes.

Having refined the set of agent types, the process is to go back to Steps 2, 3, and 4 (Sections 3.2, 3.3, and 3.4, respectively), and iterate until sufficiently detailed descriptions of the agent types, their responsibilities, and acquaintance relations, respectively, are reached[6]. On doing this, with respect to the cinema organizer case study, the artifacts shown in Figure 6 and Table 3 are obtained.

3.6   Step 6: Agent Deployment Information

Another artifact that can be useful to produce is the *agent deployment diagram*, where the physical hosts/devices agents are going to be deployed (referred to as domains in some methodologies) are indicated. The agent deployment diagram for the cinema organizer scenario is shown in Figure 7.

It should be noted that this diagram is not intended to give any detailed information about deployment (in contrast to the UML deployment diagram, where details such as the

---

[6] Note that these iterations mentioned here have been (and may be) carried out partially in Sections 3.5.1-3.5.3.
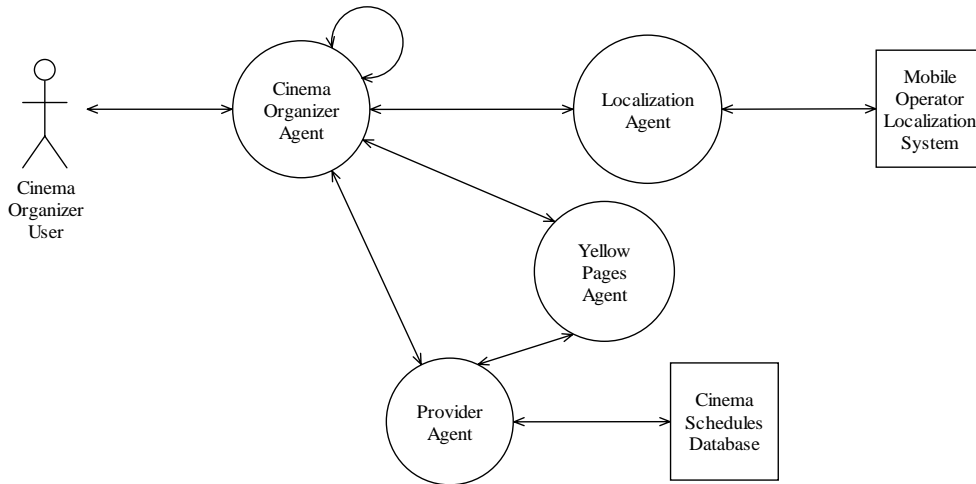
Figure 6. Agent diagram for cinema organizer case study refined after Step 5.

Table 3. Responsibility table for cinema organizer case study updated after Step 5.

Agent type	Responsibilities
Cinema organizer agent	1. Serve requests to initiate invitations from the cinema organizer user.
	2. Let the cinema organizer user select friends to invite.
	3. Let the user insert preferences about movies and cinemas.
	4. Present cinema schedules.
	5. Present incoming invitations from other cinema organizer agents.
	6. Let the user accept an incoming invitation.
	7. Let the user reject an incoming invitation.
	8. Retrieve cinema schedules from the relevant provider agent.
	9. Retrieve the current city from the localization agent.
	10. Retrieve the relevant provider agent from the yellow pages agent.
Provider agent	1. Respond to cinema schedules retrieval requests from cinema organizer agents.
	2. Register with the yellow pages agent.
Localization agent	1. Respond to city retrieval requests from cinema organizer agents.
	2. Notify city changed to cinema organizer agents.

communication modes between nodes are given). The sole purpose of the agent deployment diagram is to highlight basic deployment requirements that are referred to during design when applying considerations such as agent splitting and merging (Section 4.1) or when considering communication efficiency.
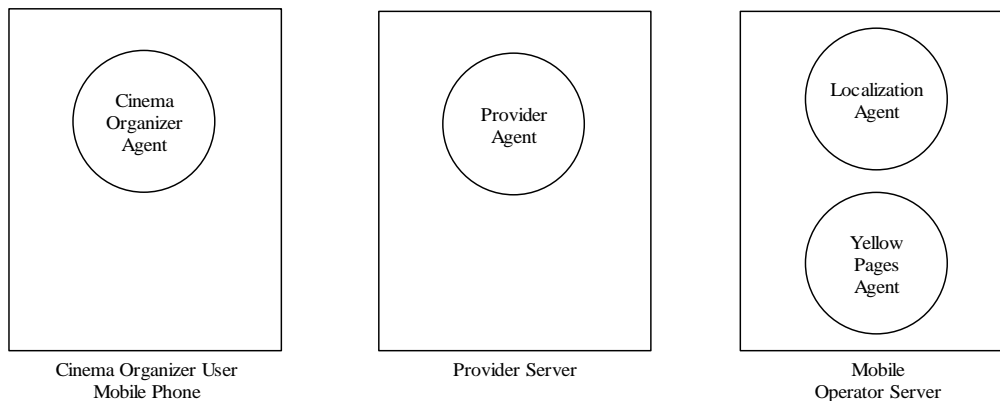
Figure 7. Agent deployment diagram for cinema organizer case study after Step 6.

3.7   Analysis Summary

The analysis aims to clarify the problem to a sufficient level of detail, with minimal concern about the solution. The steps in the analysis phase can now be summarized below:

- *Step 1: Use Cases*. The system requirements are analyzed and a *use case diagram* created based on these requirements.
- *Step 2: Initial Agent Types Identification*. By applying a set of rules, an initial diagram of the multi-agent system called the *agent diagram* is produced.
- *Step 3: Responsibilities Identification*. By observing the agent types produced in the agent diagram and applying a set of rules, an initial table of responsibilities is produced, called the *responsibility table*, for those agents whose responsibilities are clear initially.
- *Step 4: Acquaintances Identification*. The obvious acquaintances between agents are identified, and subsequently the agent diagram and responsibility table are updated.
- *Step 5: Agent Refinement*. The agent diagram and responsibility table are updated by applying a number of considerations related to support, discovery, and management and monitoring.
- *Step 6: Agent Deployment Information*. The agent deployment diagram is produced, where the agents and the physical hosts/devices the agents are going to be deployed are indicated.
- Iterate Steps 1-6.

The important elements gained from carrying out the above steps are the artifacts. These artifacts form the basis for the design phase. The artifacts produced in each step and their relationships are summarized in Figure 8.
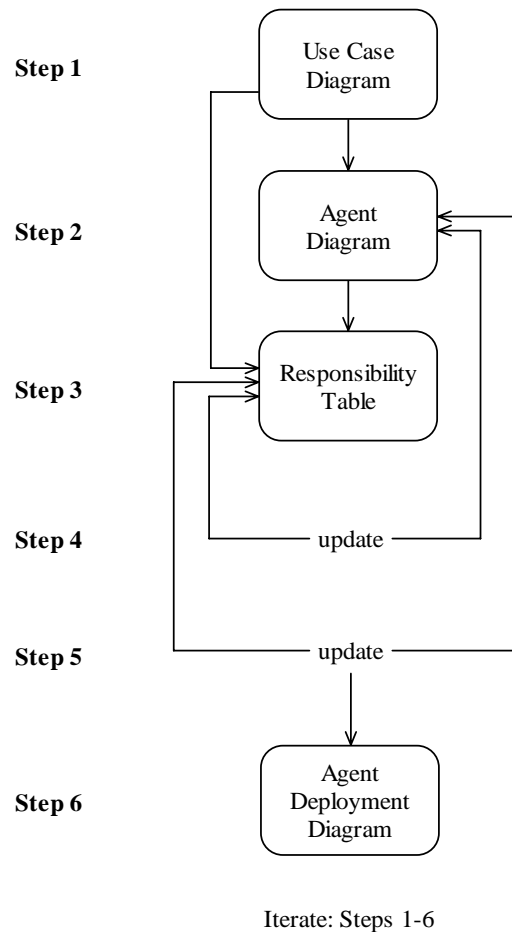
```
Step 1        ┌──────────────┐
              │   Use Case   │
              │   Diagram    │
              └──────────────┘

Step 2        ┌──────────────┐
              │    Agent     │
              │   Diagram    │
              └──────────────┘

Step 3        ┌──────────────┐
              │Responsibility│
              │    Table     │
              └──────────────┘

Step 4              ─── update ───

Step 5              ─── update ───

Step 6        ┌──────────────┐
              │    Agent     │
              │  Deployment  │
              │   Diagram    │
              └──────────────┘
```

Iterate: Steps 1-6

Figure 8. Summary of the analysis phase.

4.   Design

Once the problem has been clarified to a sufficient level of detail, a move is made from the analysis to the design phase, which aims to specify the solution. There is not a strong boundary between these two phases, and while iterating on the analysis or design, one can move between the two (see Figure 1). Since from this point on, the proposed methodology focuses on the JADE platform (and hence, the constructs provided by it), it may not be suitable for *direct* application to other multi-agent development platforms (see Section 6 for further information). Carrying out the design phase will allow one to reach a level of detail that is sufficient enough to have a relatively straightforward transition to the implementation, with the possibility of a significant amount of code being generated by an automatic tool (see Further Work, Section 7).

Since the design phase is specific to the JADE platform, there may be some parts that are unfamiliar to new JADE users. If this is the case, the user is referred to the rich set of tutorials, documents and programming APIs provided by the JADE team. More details are given in Post Design (Section 5).

Similar to the analysis, the design phase is carried out by following a number of logical steps, with a certain degree of overlap. As has been mentioned, the designer is not required to strictly follow these steps, and can add or remove steps and other relevant specifications as they desire. The steps in the design phase are discussed in detail in Sections 4.1-4.10.

4.1   Step 1: Agent Splitting/Merging/Renaming

This step involves observing the artifacts produced in the design phase and determining whether the agent types produced in the agent diagram should be split or merged. This step is considered important, since it has a direct effect on overall system efficiency and complexity. Based on this, the following rules should be applied in this step:

- Data duplication should be avoided. If there are two or more agents that share a large majority of the information required to carry out their tasks, these agents can possibly be merged into a single one.
- Duplication of code to access resources should be avoided. If there are two or more agents that need to access the same resource, these agents can possibly be merged.
- Avoid splitting agents unless there are good reasons for doing this (see below). Dealing with too many agents increases the overall system complexity and decreases system efficiency since unnecessary communication between agents will possibly take place.
- Each agent is situated on a single machine. A major factor which leads the splitting of an agent is deployment issues (based on the deployment diagram – see Section 3.6). If two pieces of functionality must be provided on different machines, these pieces of functionality must be provided by different agents.
- Avoid having agents that are too big and complex. This makes them difficult to design and to maintain.
- In some cases where the wrapper approach is used (see Section 3.2), the agent is assumed to cover and take the size of the java code it is wrapping. Thus, in such a case, it may be difficult to merge or split such an agent. Furthermore, this is another good reason to adopt the transducer approach discussed in Section 3.2.

Looking at the third and fifth rules, there may seem to be a contrast. However, what is being advocated here is a rational *balance* between an excessive number of simple agents and a small number of large complex agents.

In the case of the cinema organizer case study, there are a relatively small number of agent types identified in the analysis phase. Hence, splitting or merging of agent types is not considered to be a major issue.

4.2   Step 2: Interactions Specification

In this step, for each agent type, all responsibilities that are related to an acquaintance relation with another agent (based on the responsibility table produced in analysis) are

taken into account and an *interaction table* is produced for each agent type. Each row in the table will represent an interaction and will include:

- A descriptive name for the interaction.
- The responsibility (identified in the responsibly table produced in the analysis phase) that originates this interaction. This links design artifacts to analysis artifacts and can be used later to check consistency.
- A suitable *interaction protocol* (IP) chosen to implement the interaction. The standard FIPA interaction protocols should be considered as a candidate first [11]. If none of these protocols are deemed suitable, an ad-hoc interaction protocol should be defined, as described in Step 3 of the design phase (Section 4.3).
- The role played by the considered agent in the interaction protocol. This can be *I* for initiator or *R* for responder. Other roles can possibly be added if required.
- The agent type and name (if relevant) of the complementary role.
- The trigger condition, i.e. when this interaction takes place. This condition should be expressed in an informal but descriptive way.

Table 4 shows how the interaction table might look for the cinema organizer agent.

Similar tables can be filled for the other agent types identified in the analysis phase. It should be noted that since the focus is now on JADE, it is necessary to consider the features provided by the JADE platform. Hence, it is possible to start mapping the yellow pages agent with the ready-made JADE directory facilitator. More information on this is provided in Step 4 of the design phase (Section 4.4).

Table 4. Interaction table for cinema organizer agent after Step 2 of design.

Interaction	Resp.	IP	Role	With	When
Invite other users.	1	Contract Net	I	Cinema organizer agent	The user initiates an invitation.
Respond to an invitation.	5	Contract Net	R	Cinema organizer agent	An invitation is received.
Retrieve cinema schedules.	8	FIPA Request	I	Provider agent of the current city	- Startup. - A city change is detected and the provider for the new city has been retrieved.
Retrieve the current city.	9	FIPA Subscribe	I	Localization agent	Always.
Retrieve the provider agent for a given city.	10	FIPA Request	I	Yellow pages agent	- Startup. - A city change is detected.

4.3   Step 3: Ad-Hoc Interaction Protocol Definition

Whenever possible, existing interaction protocols defined by FIPA should be adopted [11]. However, it is often the case that an interaction requires an *ad-hoc interaction protocol* to be defined (i.e. when none of the FIPA defined interaction protocols are deemed adequate). In such cases, the interaction protocol should be defined by means of a proper formalism. Two options are suggested for this:
* The interaction protocol formalism defined in AUML [12].
* Other (possibly user-defined) interaction protocols such as the FSM[7]-based formalism (see Further Work, Section 7), Petri-nets [8], or enhanced Dooley graphs [25]. Some good reviews on the modeling of agent conversation are provided in [23, 27].

The reason for including alternative interaction protocols in addition to AUML is that although AUML is suitable for simple conversations, it is not practical for expressing complex interaction sequences.

In the case where an ad-hoc interaction protocol is required, a schema should be provided with its definition. This schema should be compliant with the AUML formalism, or, in the future, with the proposed FSM-based formalism.

4.4   Step 4: Message Templates

All the interaction protocol roles identified in the previous step are implemented as JADE behaviours (see Section 4.8). In this step, suitable `MessageTemplate` objects are specified to be used in these behaviours to receive incoming messages, and these templates are added to the rows of the interaction table. The following rules should be applied in this step:
* Use MessageTemplates based on the conversation ID in behaviours implementing initiator roles (ensure conversation IDs generated within an agent are unique).
* Merge in a single behaviour responder roles dealing with the same combination of initiation message performative, ontology, and language.
* Use MessageTemplates based on the above combinations of performative, ontology, and language, in all always-active behaviours implementing responder roles.
* Analyze conflicts (this can be carried out by means of a proper automatic tool as discussed in Further Work, Section 7) and modify MessageTemplate used in responder behaviours.
* If conflicts cannot be solved working on performative, ontology, and languages, consider applying the dynamic template pattern described in Section 4.4.1.

It should be noted that at this stage some assumptions are made about the ontology and language used in the system when specifying the templates. This will, therefore, necessitate refinements in later steps. Based on the above rules, Table 4 can be updated to Table 5 which is partially shown below.

---

[7] FSM is the abbreviation for Finite State Machine.

Table 5. Interaction table for cinema organizer agent after Step 4 of design.

Interaction	Resp.	IP	Role	With	When	Template
Invite other cinema organizer agent users.	1	Contract Net	I	Cinema organizer agent	The user initiates an invitation.	Conv-id
Respond to an invitation.	5	Contract Net	R	Cinema organizer agent	An invitation is received.	Perf = CFP
……						

4.4.1 The Dynamic Template Pattern

The dynamic template pattern shown in Figure 9 is based on the usage of a `jade.lang.acl.ConversationList` object inside the agent. All initiator behaviours register to the `ConversationList` in their `onStart()` method and deregister in their `onEnd()` method. The ConversationList therefore can keep track of all interactions initiated by the agent and is able to provide a MessageTemplate that matches all messages *not* belonging to any of these conversations. Responder behaviours with conflicting templates can then use the template provided by the ConversationList (properly refined with combinations of performative, ontology, and language) to avoid conflicts with all initiators.

4.5 Step 5: Description to be Registered/Searched (Yellow Pages)

In this step, the naming conventions and the services registered/searched by agents in the yellow pages catalogue maintained by the JADE directory facilitator are formalized (when relevant).

Naming conventions are mainly domain dependent and it is suggested to use natural language to specify them. Furthermore, a *class diagram form* is proposed to describe service registrations/searches as shown in Figure 10 with reference to the cinema organizer case study.

4.6 Step 6: Agent-Resource Interactions

It is often the case that one or more agents in the system must interact with external resources such as databases, files storing information, or legacy software. In some cases some hardware appliances must be controlled or monitored, but this always happens through some dedicated software that actually hides the hardware behind it. Agents interacting with external resources have been identified in Step 2 of the analysis phase (Section 3.2), and are expressed in the agent diagram by an acquaintance relation with a resource element. Such resources can be classified into two main categories:
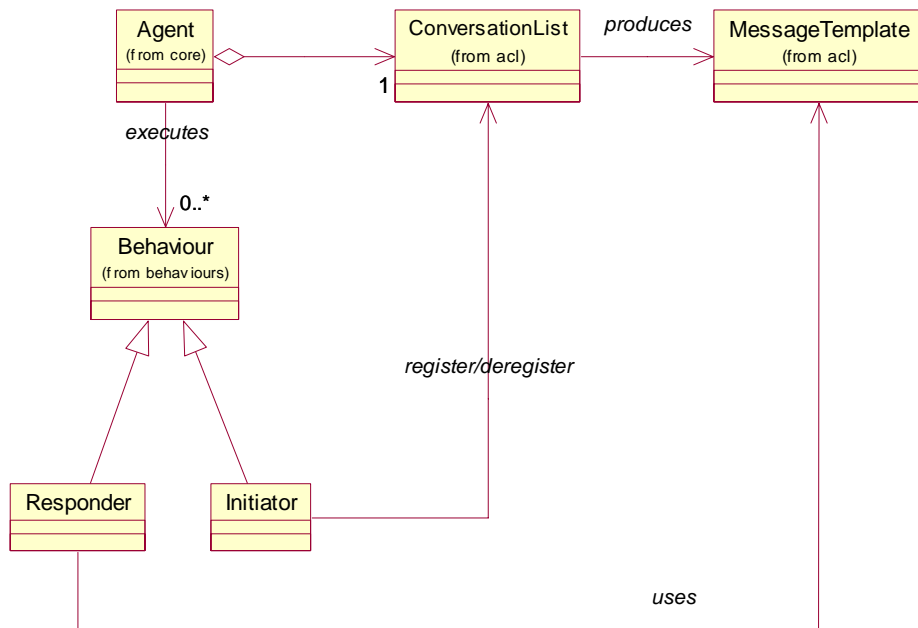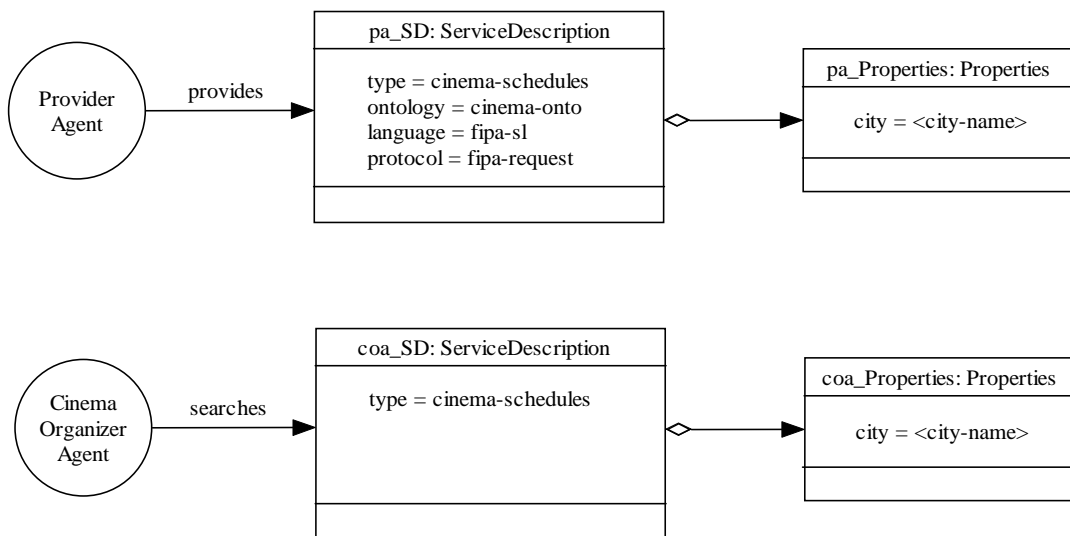
Figure 9. The dynamic template pattern.

Figure 10. Service registrations and searches in the cinema organizer case study.

- *Passive resources:* resources that change their status only as a consequence of some stimulus issued by the agent controlling the resource itself.
- *Active resources:* resources that may change their status independently from the controlling agent.

Agent interactions with passive and active resources as defined above are discussed in Sections 4.6.1 and 4.6.2, respectively.

4.6.1 Passive Resources

Examples of passive resources are a database fully controlled by the interacting agent, a data file in the local file system or a C library providing computational functions.

Interacting with passive resources is out of the scope of this methodology. Moreover, a JADE agent is, in effect, a piece of Java code and standard Java techniques can be used to handle these cases. For example, in the case of a database, JDBC should be used, in the case of a data file `java.io` should be used, and in the case of a C library, the JNI should be used. These are standard Java techniques and their explanation is out of the scope of the proposed methodology.

4.6.2 Active Resources

Examples of active resources are a database where a human operator (or an external program) can insert or modify data, a log file continuously filled (updated) by an external program, an appliance that can raise alarms and software controlling a sensor detecting changes in the local environment. Active resources may provide a listener-based interface so that the controlling agent can immediately detect changes inside the resource. In other cases, the resource may provide an interface with methods that block until a change is detected, e.g. a network socket where some data is expected to be received. Finally, in certain cases the only way to detect relevant changes in an active resource is to periodically poll the resource itself.

Though several approaches are possible to deal with active resources, a single approach is proposed, which attempts to homogenize all the possible combinations of cases described above. This approach is based on the following rules:
- If no listener-based interface is available, use a dedicated Java thread, or pool of threads, to emulate it, i.e. to detect relevant changes inside the resource and act as listener notifier.
- Provide the notifier with a listener implemented so that each call from the notifier results in adding a proper Behaviour to the agent according to the Listener adding behaviours pattern depicted in Figure 11.
- Use a `jade.util.Event` object and its `waitUntilProcessed()` and `notifyProcessed()` methods to synchronize the listener and the added Behaviour when a result (produced by the behaviour) must be passed back to the notifier as the return value of a method of the listener interface.

The proposed approach is quite flexible and avoids synchronization problems between the notifier threads and the agent thread since all relevant operations are carried out by the agent thread within the added behaviours. Moreover, using different behaviours to serve the events generated by the notifier transparently deals with the case where the notifier holds several threads that may notify events concurrently.
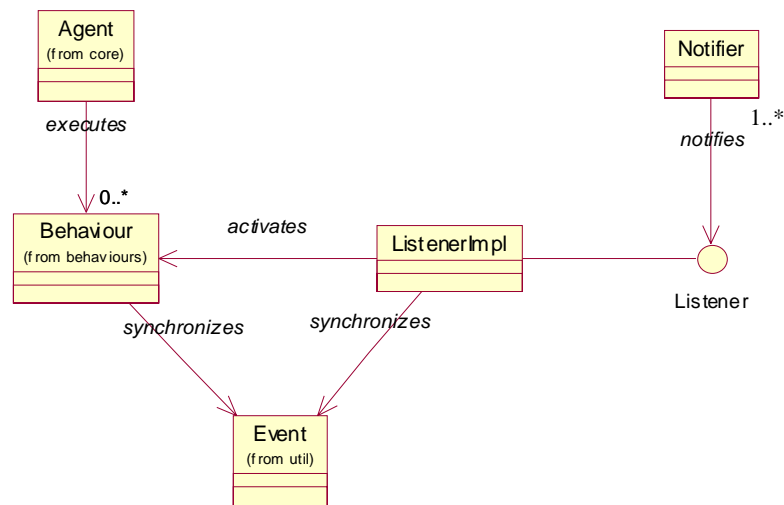
Figure 11. The listener adding behaviour pattern.

4.6.2.1 Querying a Relational Database

A particular case that deserves some more consideration is that of a relational database. Since querying a database is typically performed using a very flexible language such as SQL, just applying the transducer approach discussed in Step 2 of analysis phase (Section 3.2) to a database, may not be the right choice, since only the controlling agent (i.e. the transducer) can actually exploit the power of SQL. Another agent wishing to retrieve information from the database should send a query expressed in, for example, SL (see Section 4.10 for more information on Content Languages including SL) to the transducer, that, on its turn, should translate it into SQL, get the result from the database, and send them back to the initiator. However, expressing SQL queries in SL is definitely not a trivial task and typically one ends up with mapping all possible queries that other agents may wish to perform on the database to dedicated actions, thus making the domain ontology much more complex than it should be.

The approach proposed is to stick to the transducer approach and avoid embedding JDBC code inside all agents that may need to query the database, as depicted in Figure 12, and is based on the Iterated version of FIPA-Request-like protocols and on the JDBC ontology that is currently under development. The former is just an extension of the normal FIPA-Request protocol, but allows one to obtain the results divided in chunks, rather than all together. The latter is an ontology that provides a single concept basically mapping the JDBC ResultSet.

The proposed solution preserves the full power of SQL/JDBC without the need for embedding JDBC code (and the related complexity) inside all agents that need to perform queries on the database. Furthermore, this shows the effectiveness of the transducer approach over other approaches (see Section 3.2).
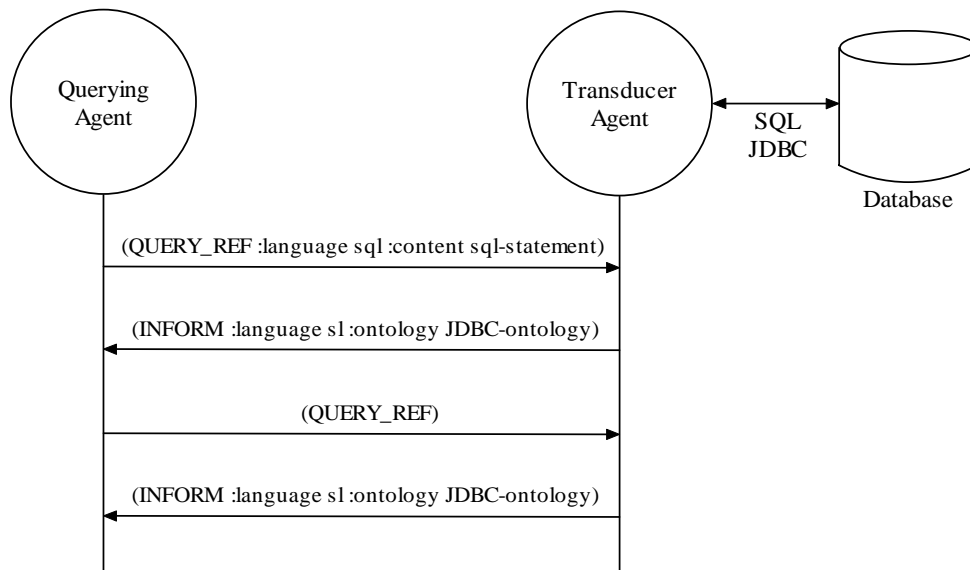
Figure 12. Performing SQL queries on a "transduced" database.

4.7   Step 7: Agent-User Interactions

In many cases, an agent needs to interact with a user. Agents interacting with users have been identified in Step 2 of the analysis phase (Section 3.2) and are expressed in the agent diagram by an acquaintance relation with an actor element.

There are several ways a human user can interact with a piece of software such as a JADE agent. Here, the focus is on the graphical user interface (GUI), which is by far the most commonly used type of user interface. Two cases are distinguished:
- A local GUI, typically implemented using Swing, the Abstract Windowing Toolkit (AWT) or some other graphical toolkit.
- A web GUI implemented using JavaServer Pages (JSP) technology.

In both cases, one could argue that a GUI can be viewed as an active resource and the rules listed in Step 6 of the design phase (Section 4.6) should apply to the agent-user interactions too. On the contrary, however, it can be argued that there are elements which make the GUI case more complex with respect to a generic external resource. Thus, it is necessary that the local GUI and the web GUI cases are discussed in more detail in Sections 4.7.1 and 4.7.2, respectively.

4.7.1   Local GUI

Here, the main issue is that the agent and the GUI must typically work on the same data (the agent to perform its domain specific tasks, and the GUI to present the data to the user), but must organize this data in different ways. As an example, the developer may want to organize a list of elements as a tree to support fast searches based on some criteria, while a `java.awt.List` would be the best structure to keep the same data from

an AWT GUI point of view. The risk, therefore, is to end up with duplication of data and consequent problems in keeping this data consistent.

Using a toolkit such as Swing, which implements the Model-View-Controller architecture [22], it is possible to overcome this problem by applying the following rules:
- Store data inside the agent in structures that are suitable to accomplish agent tasks.
- Make these structures fully synchronized.
- Make these structures implement the proper swing model interfaces and link them to the related graphic classes inside the GUI module.
- Define proper renderer classes when necessary.

With reference to the example above, by applying these rules it is possible to end up with a tree structure optimized to perform fast searches and it would be made to implement the `javax.swing.ListModel` interface. Inside the GUI module, a `javax.swing.JList` object would be kept linked to the `ListModel` implemented by the tree structure.

Unfortunately, the swing model interfaces are often quite complex, and as a consequence, following the presented approach may not be straight forward. Therefore, the decision on whether or not to adopt it is left to the developers on a case by case basis.

Additional indicators are left for possible future refinements of the methodology.

4.7.2 Web GUI based on JSP

This section provides guidance about the development of a web interface to a JADE multi-agent system implemented using Java Server Pages (JSP) technology [16]. The proposed approach is based on the creation of a dedicated agent inside the JSP, acting as gateway between the JSP and the JADE world (i.e. the other agents in the system). In particular, it is suggested to use the `JADEGateway` and `GatewayAgent` classes included in the `jade.wrapper` package in JADE 3.3[8], which hides from JSP developer's point of view the volatile nature of the agent acting as gateway. This is to say that an agent created inside a JSP is not guaranteed to stay alive forever, since the JSP container may suddenly shut it down. Therefore, managing a direct pointer to the agent may become complex since one always needs to take into account that the agent may have died in the meantime. The JADEGateway is essentially a proxy that hides to JSP developers the above complexity and automatically manages agent re-creations when needed.

There is a singleton `JADEGateway` object that embeds a container and a GatewayAgent on it. The JADEGateway provides the `execute()` method by means of which generic objects can be submitted to the GatewayAgent inside the JADEGateway. The `processCommand()` method of the GatewayAgent must be redefined to process objects passed to the JADEGateway `execute()` method. The `execute()` method returns as

---

[8] The current version of JADE (9th of November 2005).

soon as the GatewayAgent processing the Object passed to the `execute()` method calls the `releaseCommand()` method specifying that Object as argument.

An efficient way to exploit these classes can be achieved by applying the following rules:

1. Do not embed any status information inside the GatewayAgent. Status information is intrinsically embedded into the multi-agent system and there is no need to duplicate it. Caching status information inside the GatewayAgent to improve performances is typically not relevant since the performance bottleneck is always the web stuff.

2. Use domain specific "bean-behaviours" as arguments when invoking the `JADEGateway.execute()` method. The term "bean-behaviours" refers to behaviours with proper `get` and `set` methods for all domain specific parameters to be passed by the JSP to the JADE system and from the JADE system back to the JSP.

3. Extend the GatewayAgent redefining the `processCommand()` method as shown in Figure 13 below:

```
protected void processCommand(Object obj) {

      addBehaviour(Behaviour) obj);

}
```

Figure 13. Code for rule 3.

4. Call the `releaseCommand()` method of the GatewayAgent inside the `onEnd()` method of each bean-behaviour as shown in Figure 14 below:

```
public int onEnd() {

      ((GatewayAgent) myAgent).releaseCommand(this);

      return super.onEnd();

}
```

Figure 14. Code for rule 4.

4.8.   Step 8: Internal Agent Behaviours

The actual job an agent has to do is typically carried out within the agent's "behaviour(s)". Hence, in this step, the system designer should look at the agent responsibilities (via the responsibility table) identified in the analysis phase and map them to agent behaviours.

First of all, the following rule should be applied:

For a responsibility related to an interaction in the interaction table described in Step 2 of the design phase (Section 4.2), obtain the JADE class implementing the

interaction protocol and role selected for that interaction and provide a suitable extension.

In the cinema organizer case study, for instance, the "serve requests to initiate invitations from the cinema organizer user" responsibility corresponds to an interaction in the interaction table where the cinema organizer agent plays the initiator role in a Contract-Net protocol. Therefore the behaviour implementing that responsibility should be a proper subclass of the `jade.proto.ContractNetInitiator` class.

Other responsibilities must be implemented using completely application-specific behaviours and therefore it is quite difficult to provide a formal guidance for this process. The suggestion, in any case, is not to extend the `jade.core.Behaviour` class directly, but to start from the JADE classes that implement the skeletons for commonly required types of task. These classes include:

- `OneShotBehaviour`: implementing an atomic task that runs once and terminates immediately.
- `CyclicBehaviour`: implementing a task that is always active, and performs the same operations each time it is scheduled.
- `TickerBehaviour`: implementing a task that periodically executes the same operations.
- `WakerBehaviour`: implementing an atomic task that runs once after a certain amount of time, and then terminates.

When dealing with complex responsibilities, it is suggested to attempt splitting them into a number of simpler tasks combined together and adopt one of the composite behaviour classes provide by JADE. These composite behaviour classes include:

- `SequentialBehaviour`: implementing a composite task that schedules its sub-tasks sequentially.
- `FSMBehaviour`: implementing a composite task that schedules its sub-tasks according to a Finite State Machine[9].

Composite behaviour can be nested and therefore there can be, for instance, a subtask of a SequentialBehaviour that is on its turn a FSMBehaviour and so on. In particular, all complex responsibilities that can be modeled as Finite State Machines can be effectively implemented as FSMBehaviour instances.

To demonstrate an example of a complex responsibility, take for instance the "let the cinema organizer user select friends to invite" responsibility (see Table 3). This responsibility can be modeled as a Finite State Machine, and is shown by the State Transition Diagram in Figure 15 (note that this is a hypothetical example to demonstrate the point and doesn't necessarily represent efficient operation).

The State Transition Diagram in Figure 15 demonstrates that for the responsibilities defined in the analysis phase, there may be many "sub-responsibilities" (unanticipated

---

[9] A Finites State Machine (FSM) is a software (or hardware) entity that, when working, can be modeled as a sequence of states, such that the entity is only in one state at one time. Finite State Machines are typically represented using State Transition Diagrams.

responsibilities arising from the main responsibilities in the responsibility table) when mapping to agent behaviours (leading to an update of the responsibility table). Moreover, during the process of defining agent behaviours, an interaction may arise that was unanticipated in the earlier stages of the design phase. For example, the "update friends list" state (sub-responsibility) shown in Figure 15, requires an interaction with the other cinema organizer agents in order to update the friends list. This will lead to the update of the interaction table and subsequent definition of a behaviour that is a proper subclass of the `jade.proto.ContractNetInitiator` class (in this case the cinema organizer agent will play the Initiator role in a Query protocol).

4.9   Step 9: Defining an Ontology

When agents in the system interact, they exchange information that refers to entities, abstract or concrete, that exist in the environment agents reside in. These entities may be primitive, such as a String or a number, or may have complex structures defined by templates specified in terms of a name and a set of slots whose values must be of a given type. These complex entity templates are referred to as *Concepts*. For example, in the cinema organizer case study, there may be concepts such as those shown in Figure 16.
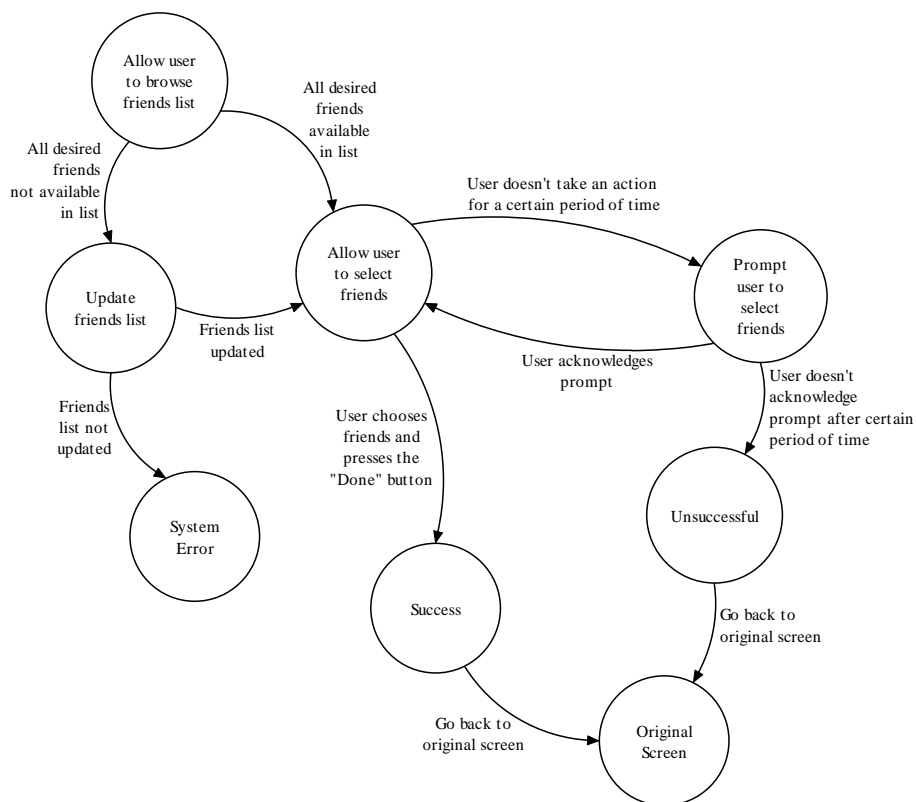

Figure 15. State Transition Diagram for "let the cinema organizer user select friends to invite" responsibility.

```
┌─────────────────────────────┐
│ Movie:                      │
│                             │
│        - title (String)     │
│                             │
│        - duration (long)    │
│                             │
│                             │
│                             │
│ Cinema:                     │
│                             │
│        - name (String)      │
│                             │
│        - address (Address)  │
│                             │
│                             │
│                             │
│ Address:                    │
│                             │
│        - city (String)      │
│                             │
│        - street (String)    │
│                             │
│        - number (int)       │
└─────────────────────────────┘
```
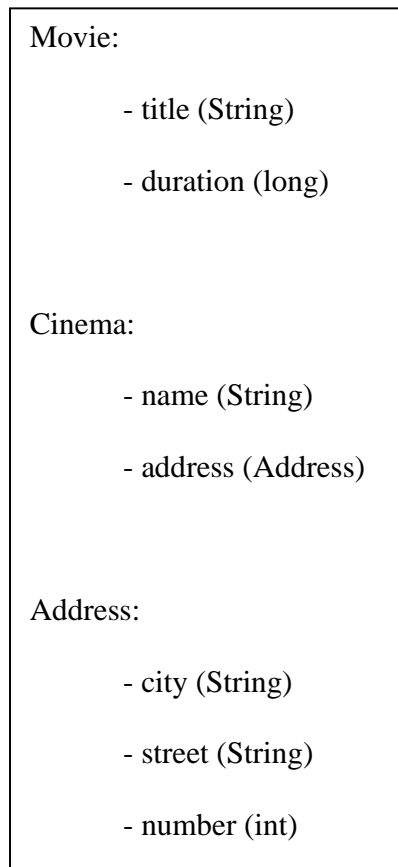
Figure 16. Example concepts for cinema organizer case study.

Moreover, entities are typically related by means of relations that can be either `true` or `false`. Similar to complex entities, relations also have structures defined by templates and again, these templates are specified in terms of a name and a set of slots whose values must be of a given type. These relation templates are referred to as *Predicates* and, considering in the cinema organizer case study, predicates such as those shown in Figure 17 can be allocated.

Finally, a particular kind of complex entity is represented by descriptors of actions that agents can perform. The templates of these action descriptors are referred to as *AgentActions*. Figure 18 shows a possible agent action (to be performed by the localization agent) relevant to the cinema organizer case study.

Actions, when executed, may produce an effect and/or generate a result to be sent back to the requester. For instance, the LocatePhone action, shown in Figure 18, will return to the requester the city where the mobile phone, whose number is specified as parameter of the action, is currently in.

An ontology is a set of concepts, predicates and agent actions referring to a given domain. Some more details on expressing ontologies are provided in the following section.
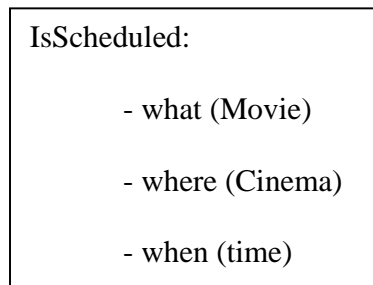
```
┌─────────────────────────┐
│ IsScheduled:            │
│                         │
│     - what (Movie)      │
│                         │
│     - where (Cinema)    │
│                         │
│     - when (time)       │
└─────────────────────────┘
```

Figure 17. Example predicates for cinema organizer case study.

```
┌─────────────────────────┐
│ LocatePhone:            │
│                         │
│     - number (int)      │
└─────────────────────────┘
```

Figure 18. A possible agent action performed by localization agent.

4.9.1   Formalisms for Expressing Ontologies

Different formalisms can be adopted for expressing an ontology. In the proposed methodology, a graphical formalism is advocated, based on UML class diagrams as shown in Figure 19.

The following points should be noted in relation to Figure 19:
- Each ontological template is expressed as a class.
- The stereotype is used to differentiate between concepts, predicates and agent actions.
- A slot of an ontological template whose type is primitive is expressed as an attribute of the corresponding class.
- A slot of an ontological template whose type is itself a concept in the ontology is expressed as a role of an association linking the ontological element that owns the slot with the concept representing the type of the slot.
- Effects and results produced by the execution of an action are documented as comments attached to the agent action.
- The inheritance relation is used (not shown in Figure 19) as usual to indicate that an ontological template is a specialization of another ontological template.

4.9.2   Heuristic rules

Defining an ontology is typically not an easy task since the same domain can typically be described by means of several different sets of concepts, predicates, and agent actions, i.e. by means of several different ontologies. For example, an alternative ontology for the cinema organizer domain is represented in Figure 20.

Generally, there are a number of choices that must be made. The most important ones are indicated in the following three sections (Sections 4.9.2.1-4.9.2.3). It should be noted that
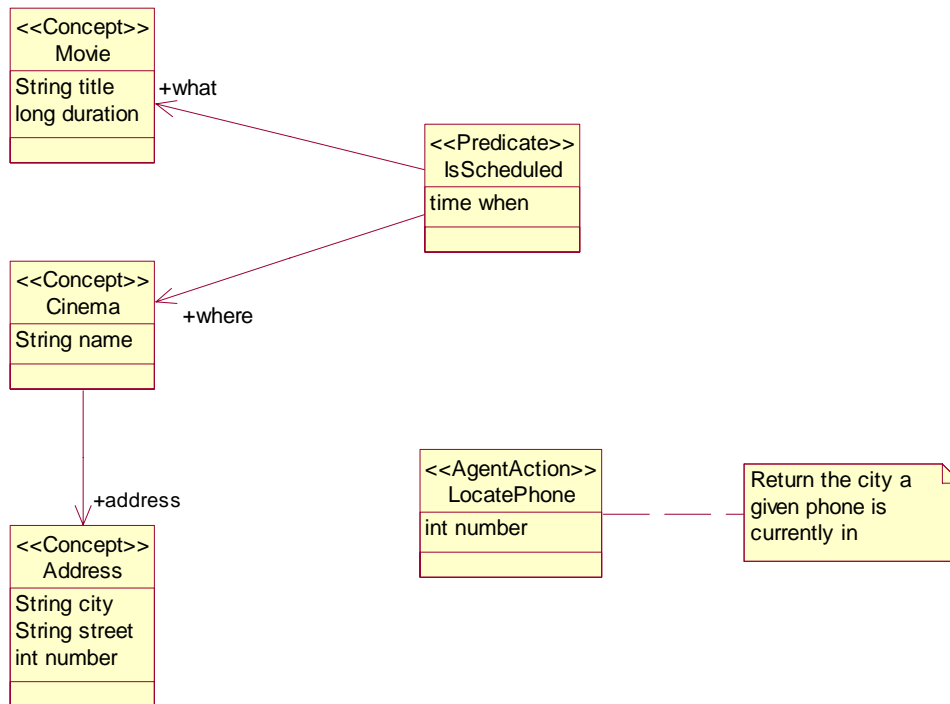
Figure 19. The cinema organizer ontology.

the details that motivate the outcome of each choice become clearer as the development process progresses. For this reason, the ontology is typically refined while iterating through the steps in the design phase.

4.9.2.1 Ontology Boundaries

An ontology is essentially a model of the application domain addressed by the system. Moreover, it is not always trivial to decide which types of entities and relations to model inside the ontology. On the one hand, it is desirable to keep the ontology as simple as possible, while on the other hand, it must be ensured that the ontology is complete enough to allow agents to perform their jobs. The guideline provided to drive this choice is the following:

> Include in the ontology only concepts and predicates that agents need to talk about, i.e. whose instances must be encoded inside the content of ACL[10] messages exchanged by two or more agents in the system.

4.9.2.2 Predicates versus Concept Slots

Each time entities of a complex type X are related to entities of type Y, this can be expressed either by adding a slot of type Y in concept X or by adding a predicate that

---

[10] ACL is the abbreviation for agent communication language.
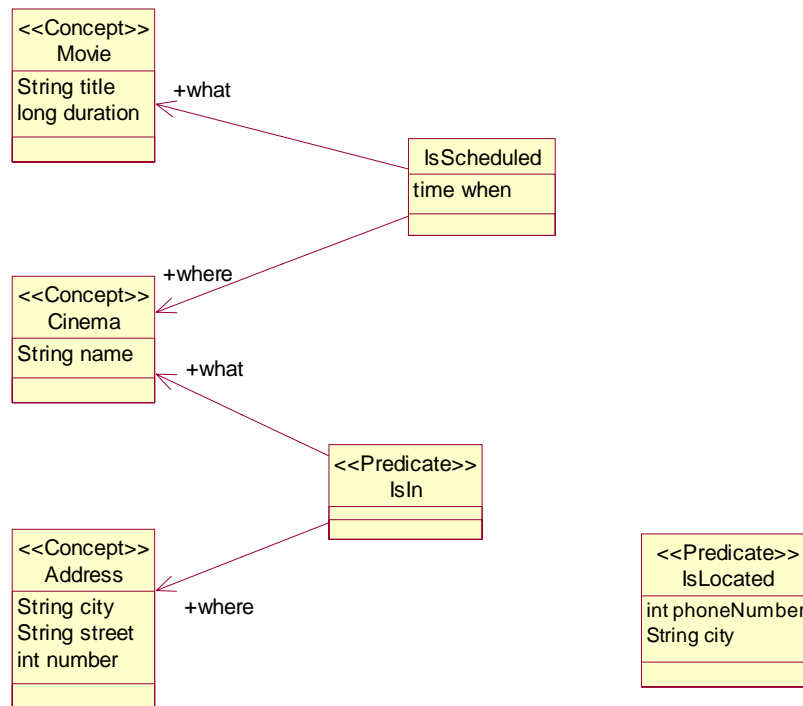
Figure 20. An alternative ontology to describe the cinema organizer domain.

relates X and Y. For example, the fact that a cinema has a given address, can be expressed either by a slot of type Address in the Cinema concept (as in Figure 19) or by a predicate `IsIn` that relates a cinema and an address (as in Figure 20).

To guide this choice, the following heuristic rule is provided:

> When, given an entity of type X, the related entity of type Y is fixed and will never change, use a slot. Conversely, when the latter can change during the lifetime of the system that is being developed, use a predicate.

In the example above, applying the rule would lead to the first choice, since it can be confidently assumed that the address of a given cinema will never change.

4.9.2.3 Information Retrieval

It is often the case that an agent in the system must retrieve some information from another agent. Using the ACL language, as JADE agents do, this may be achieved through either a QUERY_REF message including a proper Identifying Referential Expression (e.g. `(iota ?x (p ?x))`) as content, or a REQUEST message specifying an action whose result is the information that must be retrieved. This choice is clearly related to the ontology. If the cinema organizer case study is considered, a cinema organizer agent needs to retrieve from the localization agent the current city. In the ontology depicted in Figure 19, the cinema organizer agent would have to request the localization agent to perform the LocatePhone action, i.e.:

```
(REQUEST (action …. (LocatePhone :number <phone number>)))
```

In the ontology depicted in Figure 20, on the contrary, it would have to query the localization agent for the city that makes the IsLocated predicate hold for its phone number, i.e.:

```
(QUERY_REF (iota ?c (IsLocated <phone number> ?c)))
```

There are advantages and disadvantages in both alternatives. Defining predicates in the ontology is the tidiest and most flexible approach and typically allows one to keep the ontology smaller. In contrast, dealing with identifying referential expression is more complex. As a heuristic rule, the following is provided:
- If the agent providing the information to be retrieved includes a knowledge base that is able to handle Identifying Referential Expression directly, define predicates in the ontology. Otherwise, define actions.

4.9.3 Tools for Defining an Ontology[11]

JADE provides a sophisticated mechanism, described in [4], to handle ontological elements as instances of Java classes that are basically beans with proper `get` and `set` methods for all the slots in the template and to automatically convert them back and forth strings to be used as message contents. When moving from the design to the implementation, the creation of these ontological Java classes is very straightforward, but, especially when dealing with large ontologies with a lot of templates, it may be quite time consuming. Thanks to a proper plug-in, (called *beangenerator*) implemented by C.J. van Aart from the Department of Social Science Informatics at the University of Amsterdam [1], it is possible to define the ontology using Protégé [26], and then let the beangenerator automatically create the ontology definition class and the predicates, agent actions, and concepts classes. Use of this approach is particularly convenient when:
- There are several templates in the ontology.
- The ontological classes do not require any other method than the `get` and `set` methods corresponding to the ontological template slots.

Furthermore, if other methods or fields are added manually after the automatic generation, and if the ontology must subsequently be modified and therefore re-generated, all manual modifications are not preserved by the beangenerator.

4.10   Step 10: Content Language Selection

JADE provides codecs for two content languages: the SL language and the LEAP language (through the `jade.content` package). Furthermore, a codec can be defined by a programmer if they desire for the agents to "speak" a different content language [4].

The SL language is a human-readable string-encoded content language, while LEAP is a non-human readable byte-encoded content language. Based on this, some heuristics on choosing the appropriate content language are as follows:

---

[11] This section is tied to the implementation phase.

- SL is suitable for agent based applications that are (or can become) open (i.e. where agents from different developers, running on different platforms must communicate).
- The `LEAPCodec` class is lighter than the `SLCodec` class. Thus, when there are strong memory limitations the LEAP language is preferable. For example, for applications which rely on portable devices such as mobile phones, the LEAP language is much more preferable. Conversely, for applications where high capacity computers are involved, the SL language is more suitable.
- When it is required that a content language be used that should be readable by humans, SL should be chosen; otherwise, if there are no such requirements, it is advisable to use LEAP.
- If the developer wishes to define their own codec, they should do so in such a way that it is consistent with the languages handled by the resources, e.g. SQL, XML, RDF, etc. For more information, refer to [4, 28, 32].

In the case of the cinema organizer case study, since mobile phones are involved, which have a low capacity compared to computers, the LEAP language is undoubtedly the best option.

4.12 Design Summary

Once the analysis has been carried out, a move is made to the design phase, which aims to specify the solution. The solution focuses on the JADE platform. It is possible to move back and forth between the analysis and design whenever necessary. The steps in the design can now be summarized below:
- *Step 1: Agent Splitting/Merging/Renaming*. By considering system performance and complexity in relation to the agent deployment diagram produced in analysis, it is determined whether agents should be split, merged or left as is.
- *Step 2: Interaction Specification*. All responsibilities in the responsibility table related to an acquaintance relation with another agent are considered, and the interaction table produced for each agent type.
- *Step 3: Ad-Hoc Interaction Protocol Definition*. In the case that an existing interaction protocol can not be used for an interaction, an *ad-hoc interaction protocol* is defined using a suitable formalism.
- *Step 4: Message Templates*. The interaction table is updated to specify suitable `MessageTemplate` objects in behaviours to receive incoming messages.
- *Step 5: Description to be Registered/Searched (Yellow Pages)*. The naming conventions and the services registered/searched by agents in the yellow pages catalogue maintained by the JADE directory facilitator are formalized. A *class diagram form* is used as a representation.
- *Step 6: Agent-Resource Interactions*. Based on the agent diagram produced in analysis, passive and active resources in the system are identified, and it is determined how agents will interact with these resources.
- *Step 7: Agent-User Interactions*. Based on the agent diagram produced in analysis, agent-user interactions are identified and detailed.

- *Step 8: Internal Agent Behaviours.* Based on the responsibility table produced in analysis, the agent responsibilities are mapped to agent behaviours. Different types of responsibilities (including interactions) will require different types of agent behaviours to be specified.
- *Step 9: Defining an Ontology.* An appropriate ontology for the domain is specified, by making a number of considerations.
- *Step 10: Content Language Selection.* By following some rules, a suitable content language is selected.
- Iterate Steps 1-10. Move back and forth between analysis and design whenever necessary.

The artifacts produced in each step of the design and their relationships to those produced in the analysis are summarized in Figure 21 (design relationships are in bold).
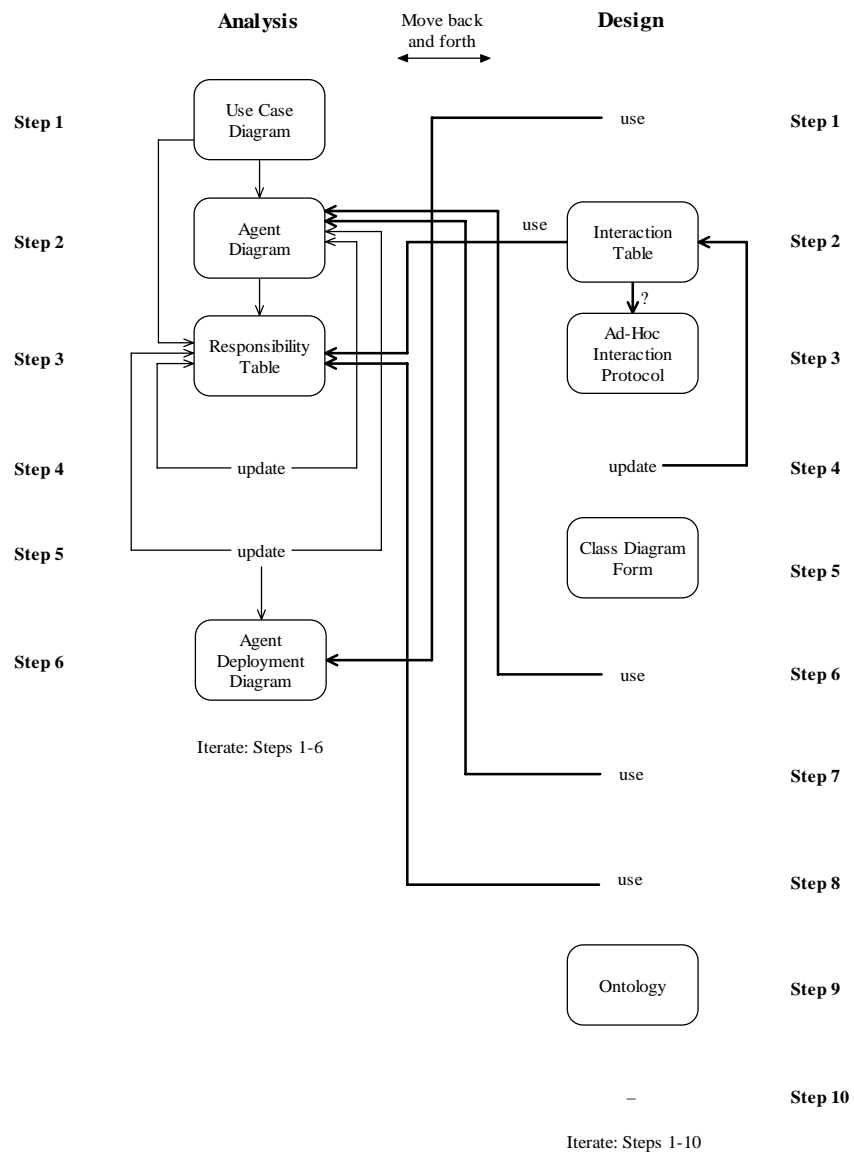

Figure 21. Summary of the design phase.

5.   Post Design

As stated in the Methodology Overview (Section 2), the proposed methodology does not attempt to formalize implementation, and related issues such as deployment and testing. However, by completing the steps of analysis and design, the multi-agent system designer should have a good idea of how the system should be implemented using the JADE platform. Thus, the systems designer can proceed to implementation by consulting the rich set of tutorials, programming APIs and other documentation on the JADE website [15], which have been provided by both the JADE team and third party contributors. Furthermore, the development of an automatic code generation tool to produce code based on the results of the design phase would be very useful, and hence, is left as possible candidate for further work (see Further Work, Section 7). The user should always remember that in the end, the development of a multi-agent system using JADE is done through the Java language, and hence, standard Java coding practices should be adhered to.

6.   Applicability to other Agent Development Platforms

The ideas presented in the analysis and design steps can be adapted to other platforms if the designer wishes. Specifically, the concepts of agent behaviours, ontologies, interactions between agents, content language selection, etc, are all abstract concepts of the agent paradigm, and hence, can be adapted to other platforms which have the means to support such concepts. It should be noted that if the desired platform is FIPA-compliant (as JADE is) the adaptation of this methodology should be easier. Furthermore, as mentioned in several parts of the methodology, the steps provided in analysis and design serve as a guide and can be removed or other steps added if deemed necessary.

7.   Conclusions and Further Work

The JADE platform is a popular, FIPA-compliant platform for the development of multi-agent systems. However, prior to this, no formal methodology had been proposed for the analysis and design of multi-agent systems using the JADE platform. The proposed methodology serves to fill this gap and, in conjunction with the tutorials and other resources provided by the JADE team and contributors, allows the system designer to fully build a multi-agent system, from requirements to implementation.

The proposed methodology assumes that an agent-based solution has been chosen as the best alterative, from a range of options. The proposed methodology does not explicitly provide pointers on when an agent-based solution should be used, but provides references, which can be consulted when a designer is contemplating an agent-based solution. Furthermore, the literature has many examples of multi-agents systems being developed for a wide range of applications. Paying attention to such cases will thus help to determine if an agent-based solution is the best choice.

The proposed methodology has been presented as a series of steps, therein containing (where appropriate) guidelines for creating artifacts, heuristic rules, notations and design patterns for the designer to use and adapt (when necessary) to their own situation. It is not essential that the proposed methodology be strictly followed, and it should only serve as a means of guidance in the development process. In addition, the designer is encouraged to use their own imagination (in a rational manner), and add or remove steps as they desire, in order to adapt the proposed methodology to their own needs. Following the steps in the methodology should be an active process and the designer should continuously iterate the steps and move back and forth between analysis and design whenever necessary.

Though the methodology fills a gap (as outline above), there are several issues remaining for future work, which include:
- The definition of a meta-model.
- More emphasis on agent internal structures and mechanisms.
- Addressing of additional requirements such as security, persistency/transaction, non functional requirements such as performances and scalability.
- Development of a FSM-based formalism in compliance with the JADE platform, which will provide a means for complex interactions to be modeled.
- Development of an automatic tool to automatically generate code based on the constructs developed in the design phase.
- Comparison of the proposed methodology with other currently existing methodologies for multi-agent system development (using guidelines such as those provided in [31]).
- Formal adaptation and extension of the methodology to other multi-agent platforms including those platforms which have capabilities for defining agent mental attitudes.

The final point is quite important and will be very desirable and valuable for developers using or having had experience with other platforms.

Acknowledgements

References

1. Beangenerator, see: http://acklin.nl/beangenerator/.
2. A. H. Bond and L. Gasser (Eds.), *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann Publishers: San Mateo, CA, 1988.
3. S. Bussmann, N. R. Jennings, and M. Wooldridge, *Multiagent Systems for Manufacturing Control*, Springer-Verlag, 2004.
4. G. Caire and D. Cabanillas, "JADE tutorial: creating and using application specific ontologies", 2004, see: http://jade.tilab.com/doc/CLOntoSupport.pdf.
5. G. Caire, W. Coulier, F. Garijo, J. Gomez, J. Pavon, F. Leal, P. Chainho, P. Kearney, J. Stark, R. Evans, and P. Massonet, "Agent Oriented Analysis Using Message/UML," *Lecture Notes in Computer Science*, M. Wooldridge, G. Weiss, and P. Ciancarini (Eds.), Springer-Verlag, vol. 2222, 2002, pp. 119-135.
6. C. Campo, Directory Facilitator and Service Discovery Agent, FIPA Document Repository, 2002, see: http://www.fipa.org/docs/input/f-in-00070/f-in-00070.pdf.
7. A. Collinot, A. Drogoul, and P. Benhamou, "Agent oriented design of a soccer robot team," in *Proceedings of the 2nd International Conference on Multi-Agent Systems (ICMAS-96)*, Kyoto, Japan, 1996, pp. 41–47.
8. R. Cost, Y. Chen, T. Finin, Y. Labrou, and Y. Peng, "Modeling agent conversations with colored petri nets," in *Workshop on Specifying and Implementing Conversation Policies*, 1999, pp. 59-66.
9. A. Dennis and B. H. Wixom, *Systems Analysis and Design: An Applied Approach*, John Wiley and Sons, 2000.
10. Foundation for Intelligent Physical Agents (FIPA), see: http://www.fipa.org/.
11. FIPA Interaction Protocol Specifications, see: http://www.fipa.org/repository/ips.php3.
12. The FIPA AUML Web Site, see: http://www.auml.org/.
13. M. R. Genesereth and S. P. and Ketchpel, "Software Agents," *Communication of the ACM*, vol. 37(7), 1994.
14. L. Hampton, R. C. Martin, F. G. Pitt, and T. Ottinger, A Critique of Use Cases, 9 July 1997, see: http://ootips.org/use-cases-critique.html.
15. JADE – Java Agent DEvelopment Framework, see: http://jade.tilab.com/.
16. JavaServer Pages (JSP) Technology, see: http://java.sun.com/products/jsp/.
17. N. R. Jennings, "Agent-based Computing: Promise and Perils," *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, Stockholm, Sweden, 1999, pp. 1429-1436.
18. N. R. Jennings and M. Wooldridge, "Applications of intelligent agents," in *Agent Technology: Foundations, Applications and Markets* (Eds. N. R. Jennings and M. Wooldridge), pp. 3–28. Springer, Berlin, 1998.
19. M. Luck, R. Ashri, and M. D'Inverno, *Agent-Based Software Development*, Artech House Publishers, 2004.
20. M. Luck, P. McBurney, and C. Preist. "Agent Technology: Enabling Next Generation Computing," AgentLink, 2003, see: http://www.agentlink.org/admin/docs/2003/2003-48.pdf.
21. E. Milgrom (Ed.), "Final guidelines for the identification of relevant problem areas where agent technology is appropriate" (from the MESSAGE methodology), EURESCOM, 2001, see:

http://www.eurescom.de/~pub-deliverables/P900-series/P907/D2/p907d2.pdf.

22. The Model-View-Controller, see:
http://java.sun.com/blueprints/patterns/MVC-detailed.html.

23. M. Nowostawski, M. Purvis, and S. Cranefield, "A layered approach for modelling agent conversations," *Second International Workshop on Infrastructure for Agents, MAS and scalable MAS*, Montreal, Canada, 2001.

24. J. Odell, "Objects and agents: how do they differ?," *Journal of Object-Oriented Programming*, October 2000.

25. H. V. D. Parunak, "Visualizing agent conversations: using enhanced Dooley graphs for agents design and analysis," *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS'96)*, 1996.

26. Protégé, see: http://protege.stanford.edu/.

27. S. Paurobally, J. Cunningham, and N. R. Jennings, "Developing agent interaction protocols graphically and logically," *Proceedings of the First International Workshop on Programming Multi-Agent Systems*, Melbourne, Australia, 2003, pp. 45-54.

28. F. Quarta, How to use the XML support with JADE, 2003, see:
http://jade.tilab.com/doc/tutorials/XMLCodec.html.

29. Y. Shoham, "Agent Oriented Programming," *Artificial Intelligence*, vol. 60(1), pp. 51-92, 1993.

30. T. Skylogiannis, "Automated Negotiation and Semantic Brokering with Intelligent Agents Using Defeasable Logic," MSc Thesis, 2005, see:
http://www.ics.forth.gr/isl/publications/paperlink/Skylogiannhs.pdf.

31. A. Sturm and O. Shehory, "A framework for evaluating agent-oriented methodologies," *Agent-Oriented Information Systems, 5th International Bi-Conference Workshop (AOIS 2003)*, Melbourne, Australia, July 14, 2003.

32. P. Turci, How to use the RDF support with JADE, 2001, see:
http://jade.tilab.com/doc/tutorials/RDFCodec.html.

33. Unified Modeling Language (UML), see: http://www.uml.org/.

34. M. Wooldridge, *An Introduction to Multiagent Systems*, John Wiley and Sons, 2002.

35. M. Wooldridge, N. R. Jennings, and D. Kinny, "The gaia methodology for agent-oriented analysis and design," *Autonomous Agents and Multi-Agent Systems*, vol. 3(3), pp. 285-312, 2000.

36. M. J. Wooldridge and N. R. Jennings "Pitfalls of agent-oriented development," *Proceedings of the 2nd International Conference on Autonomous Agents (Agents-98)*, Minneapolis, USA, pp. 385-391, 1998.

37. M. Wooldridge and N. R. Jennings, "Intelligent agents: theory and practice," *The Knowledge Engineering Review*, vol. 10(2), pp. 115-152, 1995.