An Approach to Modelling and Applying Mobile Agent Design Patterns

Emerson F. A. Lima, Patrícia D. L. Machado, Flávio R. Sampaio and Jorge C. A. Figueiredo
Departamento de Sistemas e Computação, Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brazil
e-mail: {emerson,patricia,ronison,abrantes}@dsc.ufcg.edu.br

Abstract

Mobile agent design patterns represent solutions to specific problems of implementing mobile agent-based applications that have evolved over time. The use of design patterns can increase productivity, promote reuse and reduce complexity when developing applications. However, most of the mobile agent design patterns presented in the literature are difficult to apply in practice due to the lack of a suitable approach to identify, document and apply them. Also, they are usually related to a specific mobile agent platform. We present an approach for modelling and applying patterns independently of specific platforms along with their counterparts in specific platforms. Also, we show that an adequate platform independent view can be constructed to be used as a guide to implement the pattern in different platforms.

Introduction

The widespread use of the Web along with the continued development of mobile computing technology give rise to a new scenario where services are offered by applications that can be dynamically configured according to different user profiles in a global and heterogeneous environment.

Mobile computing extends the conventional notion of distributed computing. Devices can be moved from a network to the other dynamically (physical mobility) and programs can interrupt their execution in a host and migrate to others to fulfil a task (virtual mobility). The goal is to give more flexibility to mobile users by reducing the need for keeping active expensive and/or instable connections.

However, mobility introduces further problems: adverse communication circumstances and resource management constraints. Mobile computing can be seen as the worst scenario in a distributed system where the environment topology is dynamic and communication and disconnection represent a definite challenge [6, 4].

Mobile agents is a promising approach to developing distributed applications that addresses these problems in a mobile computing environment. Particularly, the growing interest in agent technology has been motivated by its potential use in a number of application areas including electronic commerce, information gathering and dissemination, telecommunication systems, customisable services and monitoring systems [17, 10].

Nevertheless, mobile agents technology has limitations and challenges that need to be overcome: the lack of agents and execution environment security and the need for established methodologies of developing mobile agent-based applications.

Some approaches toward a methodology of developing multi-agent systems focusing on patterns of interaction and cooperation have already been proposed [18]. But, mobility has not yet been appropriately considered, although some important attempts have already been made [20, 4, 3, 9, 16]. The approach presented in [13] focus on modelling mobile agent systems using an extension of UML. In [9], a methodology of developing mobile agent-based applications is introduced. This is based on the ideas presented by Tahara et al [20] on the designing of mobile agent-based applications by applying design patterns according to specific architectural levels, where, to maximise reuse, higher levels are independent of specific agent platforms.

Design patterns have been widely applied in object-oriented design since they can improve productivity, flexibility and reuse. Without meaningful technology patterns, applications development entails high costs. This is more critical in the development of mobile agent-based applications due to the inherited complexity of constructing such applications. However, most of the mobile agent design patterns presented in the literature are difficult to apply due to the lack of a suitable approach to identify, document and apply them. Also, they are mainly related to a specific mobile agent platform and there is a lack of substantial case studies.

Motivated by the approaches presented in [20, 9], we introduce an approach of modelling and applying mobile agents design patterns. We present design pattern models, using the UML notation, at two different abstract levels: platform independent and platform dependent. For the lack of space, only one design pattern is considered as example.

We show that modelling patterns independently of specific platforms promotes their use, since platform dependent models can be refined from the independent ones to handle specific details of a given platform. Current platforms present significant differences that justify a platform dependent design effort. Furthermore, we hope the ideas presented in this paper can contribute to future development in the theory and practice of documenting and applying mobile agent design patterns to develop applications.

In the following sections, we present a methodology of designing mobile agent-based applications that is based on the use of mobile agent design patterns. Then, an approach for modelling design patterns is introduced. Also, we present an approach for applying the patterns that is illustrated by a case study. Finally, concluding remarks and pointers for further work are given. We assume the reader to be familiar with basic mobile agent concepts as presented in [8, 14, 6].

A Design Methodology

In this section, we introduce a methodology of designing mobile agent-based applications that is based on the development methodology presented in [9]. As usual, the main goal of the design discipline is to extend and refine analysis models to provide a feasible solution to the system, considering functional and non-functional requirements. This discipline is divide into three sub-disciplines: platform independent architectural design, platform independent detailed design and platform dependent detailed design (Figure 1).

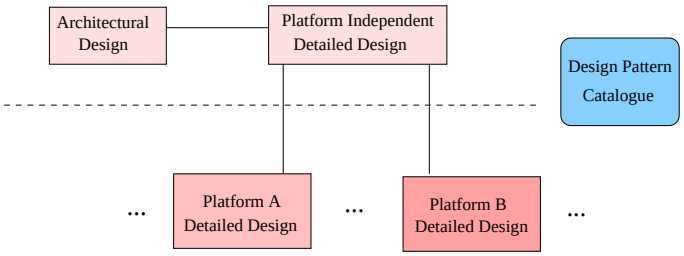

Figure 1: Mobile Agent-based Design Model.

Architectural design represents an outline of the system. The system can be decomposed into layers and/or partitions. Agent behaviour is modelled for the main scenarios according to specific architectural design patterns. Architectural mobile agent patterns are usually classified as: mobility and task patterns. Mobility patterns are applied when agents perform a task while moving in a chain of agencies. The intention is to allow for an efficient use of network resources. Task patterns are used to specify how agents should interact in order to perform a task. In general, tasks can be dynamically assigned to agents. Also, a given task can be performed either by a single agent or by a group of agents working in parallel and cooperating.

In the platform independent detailed design, a specification of a logical design independently of any specific mobile agent platform is produced. Artifacts must reflect how agent behaviour is implemented according to the architectural patterns chosen and the functional and non-functional requirements. Class diagrams are refined to include design concepts (mainly from patterns chosen) and interaction diagrams are created to illustrate sequences of interactions of the system and show how operation contracts are implemented and fulfil postconditions. Agents are usually modelled as classes. An extension of UML proposed in [13] for modelling mobile agent-based applications is used. This extension introduces new elements to represent agencies, regions, agents, migration and cloning.

In the platform dependent detailed design, the logical specification produced in the platform independent design is extended to address design and implementation issues related to a specific agent system platform. Platform dependent design patterns are applied and interaction and class diagrams are refined to reflect them. Platform dependent patterns are usually categorised as: (1) refinement of patterns to include details related to implementing them in the platform and (2) patterns related to issues such as communication, security and safety.

To support these disciplines, it is important that patterns are documented and added to a catalogue whenever they are successfully modelled and implemented in a given application. Modifications and experience on the use of the patterns must be continuously updated in the catalogue.

The division between platform dependent and independent design is motivated to promote design reuse and also because mobile agent-based applications may be required to run on different platforms. Moreover, due to the complexity of implementing mobile agent-based solutions, a platform independent view of the logical solution being considered can be staple to make it possible for a global understanding and analysis of its quality, even if only a single platform is being considered [16]. Also, this can help analysing different logical solutions for implementing the application.

Mobile Agent Design Patterns

Mobile agent design patterns represent solutions that have evolved over the time to problems of designing applications based on the mobile agent technology. Such patterns have been identified, classified and documented in the literature [1, 19, 20, 5, 21, 12, 11]. However, in general, their documentation and classification do not follow a standard. Some patterns are described using natural language, whereas others are presented by UML diagrams. In [16], Petri Nets are used to model mobile agent design patterns for performance comparison. In addition, it is usually difficult to apply them since their purposes are not clearly stated or unnecessarily related to a specific platform, even though the main ideas could be applied when implementing applications in different platforms. In [15], a preliminary catalogue of patterns, classified according to their applicability, is presented. Patterns documentation includes intent, motivation, applicability, structure (platform independent view), consequences, implementation (platform dependent view), known uses and related patterns. The main categories of patterns are: migration, task, resources, security and communication. Patterns have been investigated and implemented in three different agent platforms. For the lack of space, in this paper, we focus on the meeting pattern [1]. The master-slave pattern is also briefly introduced.

Meeting Pattern. Suggests a way of promoting local interactions between agents that are distributed in a network. Two or more agents need to migrate to an agency where the meeting takes place. In the meeting place, there is a meeting manager responsible for controlling the process of joining and leaving the meeting. The manager is a shared resource that keeps a list of participants. Interactions between agents can make execution of complex tasks possible as well as results optimisation in a distributed effort to undertake a task.

Master-Slave Pattern. A master agent can create slaves that migrate to remote hosts in order to perform a task.

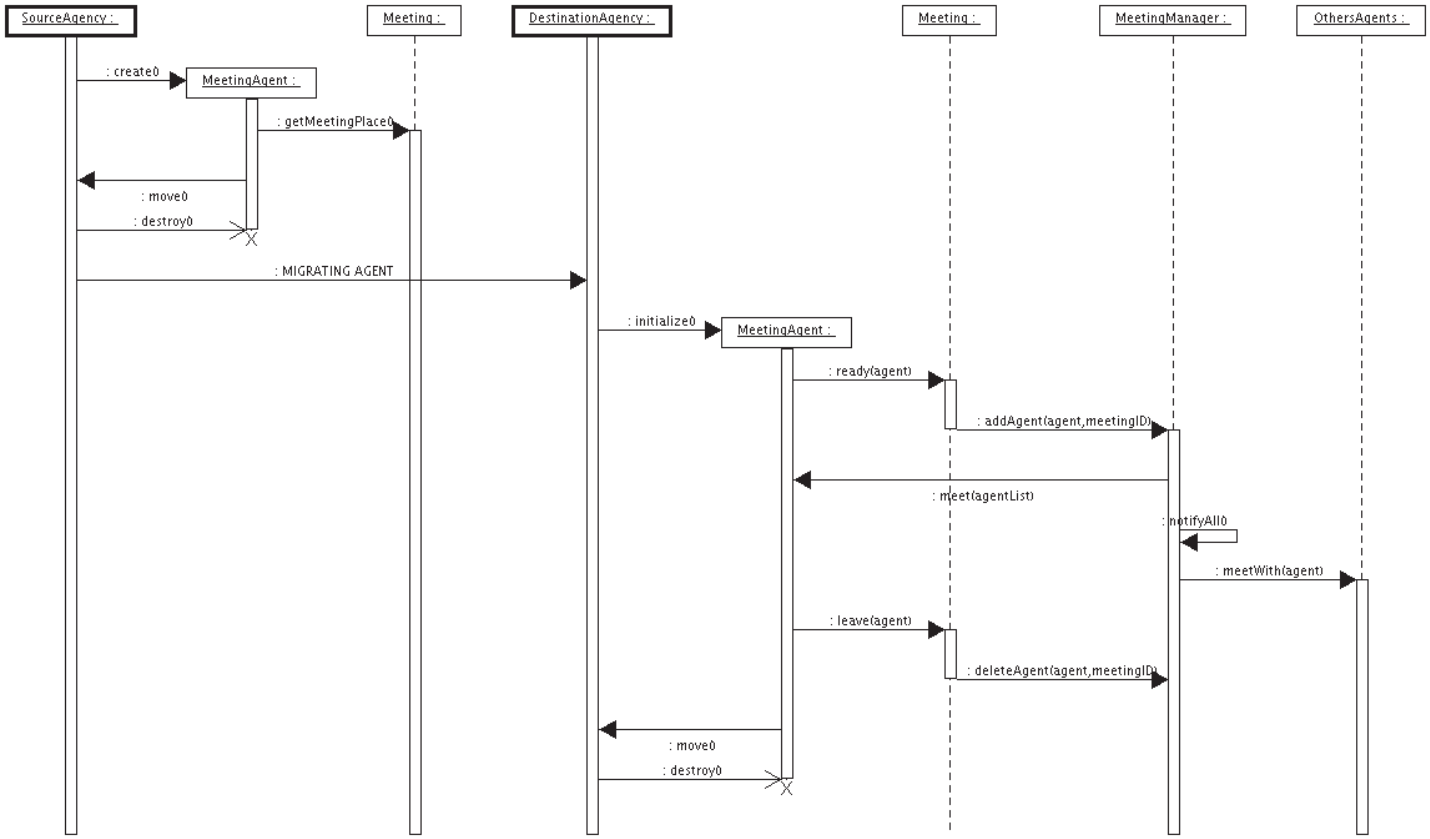

Figure 2: Meeting Pattern – Platform Independent Sequence Diagram.

When slaves come back with the results produced, the master agent collect them to conclude the task and may decide to destroy the slaves. This pattern can be useful, for instance, when the master agent has to perform other tasks and need to be freed from the task of monitoring the agents.

Modelling Mobile Agent Patterns

We propose that the documentation of mobile agent design patterns includes both platform independent and dependent models. According to the standard presented in [7], the structure section (graphical representation of the main classes in the pattern and possible interactions) can be composed of platform independent models, both architectural and detailed, whereas the implementation section can be composed of platform dependent models that reflect implementations issues in a given platform. These models are described in the sequel.

Platform Independent Models

The platform independent model presents a general structure of the pattern and its abstract functionality. The goal is to precisely state the proposed solution without considering specific details of any agent platform. For this, only features of mobile agents usually supported by any platform is considered such as agent cloning and migration [8, 6]. Main classes, including agents and domain concepts, and interactions are identified and documented. We construct UML class and sequence diagrams, since they can represent, in a simple and abstract way, the entities involved in the pattern and their relationships.

To follow the description of an independent model of the Meeting Pattern, consider the sequence diagram presented in Figure 2. Throughout this paper, we use a notation that is equivalent to the one presented in [13][1]. In this notation, an object is used to represent an entity that controls agents execution in a given platform[2] (creation, migration) and indicates their location (*SourceAgency* and *DestinationAgency*). Migrations are represented by message passing, labelled as *MIGRATING AGENT*, from one agency entity to the other. This is requested by an agent sending the *move()* message. Before migrating, agent execution is interrupted (arrow labelled as *destroy()*). Execution is continued in the target agency (arrow labelled as *initialize()*).

From Figure 2, *MeetingAgent* is a mobile agent, in a source agency, that has a reference to a *Meeting* object indicating the place and the identification of a meeting. A reference to a meeting object can be obtained, for instance, in agents initialisation or recovered from a repository. From this point,

[1]Some graphical features differ due to limitations of the edition tool.
[2]This entity represents the agency, but it is not the agency itself.

the agent can request the target of the meeting to the *Meeting* object, using the *getMeetingPlace()* method. Then, it can request to be moved (*move()* method) to the meeting place (*DestinationAgency*).

After migrating, the agent is initialised by the target agency (*initialize()* method). Then, the agent calls the *ready* method of its *Meeting* object. This object then calls the *addAgent* method of the (*MeetingManager*) object. This method is responsible for registering an agent in the specified meeting controlled by the meeting manager. As a consequence, the manager sends to the agent a list of agents attending in the meeting (*meet* method). It also notifies the other agents of the arrival of the agent (*notifyAll* and *meetWith* methods). From here, the agent can interact with the other agents by message passing. When the agent decides to leave the meeting, it calls the *leave* method of the *Meeting* object. The agent is then removed from the manager list (*deleteAgent*). In the scenario presented in Figure 2, the agent migrates to another agency.

Figure 3 presents a class diagram illustrating the pattern structure. As mentioned, *MeetingAgent* is a mobile agent (*MobileAgent* class). This agent has a *Meeting* object that keeps a meeting location (*meetingPlace*) and identification (*meetingID*). The *MeetingManager* object is responsible for controlling the meeting by executing operations on a list of agents kept by the *arrivedAgents* attribute.

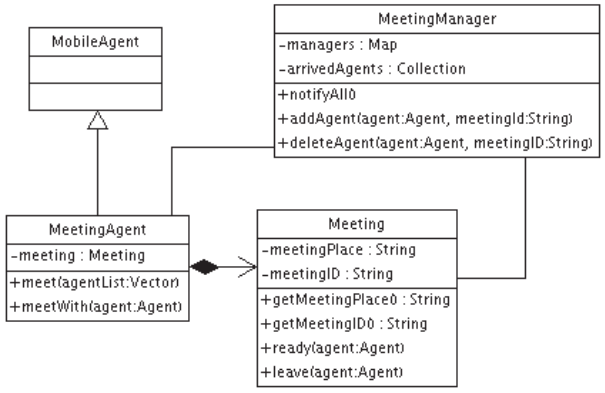

Figure 3: Meeting Pattern – Class Diagram.

Platform Dependent Models

Platform dependent models refine platform independent models to consider specific details of implementing a pattern in a given agent platform. The UML extended notation is the same used on the previous section.

In order to conduct our investigation, dependent models have been developed based on three mobile agent based platforms, chosen according to the following criteria: (1) complete and free of charge distribution, (2) portability – run in different operating systems, including versions for portable devices, (3) have JAVA as the main programming language and (4) have complete documentation available and support for community discussion.

The platforms chosen are the following. Aglets [14] is based on the idea of creating special Java applets, named aglets, that are able to move from one machine to the other in a network. This is supported by a library named Aglets/API. A protocol named Agent Transfer Protocol (ATP) is used to support communication and mobility of agents from one context to the other. Contexts are Java Virtual Machine (JVM) abstractions.

The Grasshopper platform [11] is an agent development platform, launched by IKV++, that is compliant to the first mobile agent standard of the Object Management Group (OMG), the so-called, Mobile Agent System Interoperability Facility (MASIF) [8]. The platform is based on the Java Language and is built on top of a distributed processing environment to achieve interoperability between mobile agent platforms of different manufacturers and also traditional client/server paradigm technology.

JADE (Java Agent DEvelopment Framework) [2] is a platform completely implemented in Java that simplifies the implementation of multi-agent systems by providing a middleware based in the FIPA specifications and a number of tools to support development of applications. Recently, the virtual machine has been extended to support execution on portable devices allowing the use of Java Micro Edition.

The dependent models created for each platform are briefly described in the sequel. Finally, the models are compared.

Aglets Model. The Aglets solution, shown partially in the sequence diagram of Figure 4, is the simplest one due to the existence of a package named *com.ibm.agletx.patterns* that supports patterns, including the meeting pattern. This package has a class named *Meeting* that is implemented according to the independent model presented in the previous section. An agent has a *Meeting* object as attribute that keeps the meeting place and identification. From Figures 2 and 4, it can be noticed that when an agent arrives at the meeting place, it calls the *ready* method as usual. This method returns an enumeration of the agents in the meeting. Then, the *Meeting* object sends a notification to the other agents in the meeting, so that they can interact with the agent by calling the corresponding *Agent Proxy*.

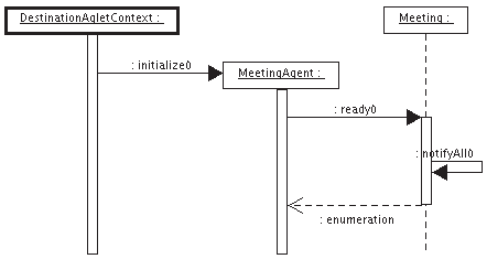

Figure 4: Sequence Diagram of the Meeting Pattern in the Aglets platform (after migrating).

Note that the *MeetingManager* class and the *leave* method in Figure 2 are not used in the Aglets model. *MeetingMan-*

ager is an auxiliary class responsible for maintaining and sharing references to participants of a meeting. This functionality is directly supported in Aglets by the *AgletContext* class that it is also the entity responsible for controlling agents execution in a given instance of the platform. Using the facilities provided by the *AgletContext* class, the *Meeting* object do the following (*ready* method): (1) broadcast a message to the agents in a context whose subject is the meeting identifier and the content is the identifier of the recently arrived agent; and (2) collect the identifiers of the agents that acknowledge the message and send an enumeration to the recently arrived agent. The *leave* method is not necessary since when leaving a context, the agent automatically leave the meeting. Finally, it is also possible to specify which participants must receive messages broadcasted to a meeting. An exception is raised if a participant is not in the meeting.

Grasshopper Model. In Grasshopper, special places can be created to give access to an object that provides resources and services to agents. For this, it is necessary to create an interface, a class that implements this interface and a configuration file named *<special_place_name>.properties*. In our solution, a special place named *MeetingPlace* has an interface to provide access to the *MeetingManager* object. This object is a *singleton*, since it should only be created once, when the first agent arrives at a meeting. In the sequence diagram partially shown in Figure 5, it can be noticed that when an agent arrives at the meeting agency, the *Meeting* object accesses the special place interface and gets a reference to the *Meeting Manager* object. At this point, the *Meeting* object can interact with the *MeetingManager* to register the agent. Note that the refinement of the independent model is direct and intuitive. The main difference is in the way the *Meeting* object interacts with the *MeetingManager* object.

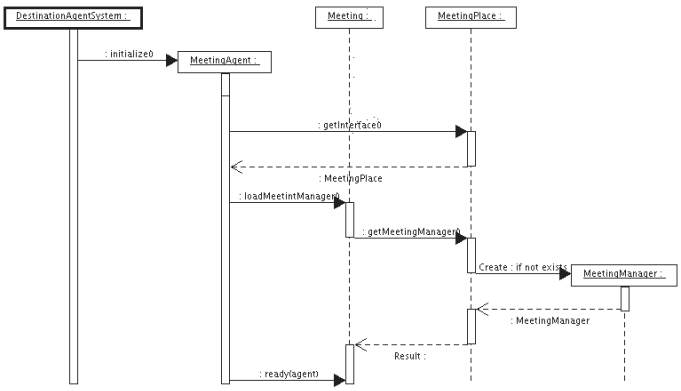

Figure 5: Sequence Diagram of the Meeting Pattern in the Grasshopper platform (after migrating)

JADE Model. Figure 6, presents a partial sequence diagram of the meeting pattern in JADE. In this platform, the *meet* and *meetWith* methods of the *MeetingAgent* receive as parameters a list of agent identifiers (AID) and the source agent identifier respectively. This parameters differ from the ones in the independent model since there is no direct interaction between agents in JADE. Interactions are done by message passing where only the identifier is necessary. The *MeetingAgent* has a *Meeting* object as in the independent model. But the agent has a cyclic behaviour (*CyclicBehaviour*) named *ReceiveMessageBehaviour* that is responsible for receiving messages. Also *MeetingManager* is defined as an agent and has a behaviour named *UpdateMeetingBehaviour* that receives messages and calls the methods responsible for updating the meeting.

Note that JADE models have an additional feature related to the way agents interact with the *MeetingManager* object. As explained before, interactions are made by messages that are received by behaviour objects. Behaviours are a special feature of JADE that can be used to define what an agent has to do in different situations such as receiving messages and executing specific tasks. The use of behaviours can make models look more complex. However, there is a considerable gain in modularity since agents can be modified by locally altering specific behaviours rather than the agent class. It is also important to remark that a message target can be a vector of agent identifiers, making broadcast easier.

Comparing Models

The platform independent model of the Meeting Pattern gives an abstract view of its functionality, showing entities and their relationships and operations that must be implemented in general. Note that, even though dependent models are refinements of the independent model, there are considerable differences in the details presented in the dependent models of the platforms considered. Dependent models include details that can make them unsuitable if a more abstract understanding is required. On the other hand, a number of issues need to be considered when implementing an independent model in a given platform and it can be very difficult to handle them if a design approach is not followed at this level. This clearly justifies the need for independent and dependent models.

Note that *MeetingAgent* can be mapped directly in each platform. On the other hand, the *Meeting* and *MeetingManager* objects require a better understanding of platforms specific features. In Grasshopper and JADE, the *Meeting* object can be implemented using the interface suggested in the independent model. However, in Aglets, there is a predefined implementation that differs from the independent model interface that can be easily mapped to it. The major change is in the *MeetingManager* object implementation. In Aglets, this is the *AgletContext*. In Grasshopper, this is a singleton accessed through a special place. And in JADE, this is modelled as an agent.

When evaluating design models, it is important to consider measures such as scalability, reuse and modifiability. In the Meeting pattern, scalability depends on the way the developer implements the *MeetingManager* object. Once this object is concurrently accessed, care must be taken to avoid failures and inconsistencies in managing the meeting. In Aglets,

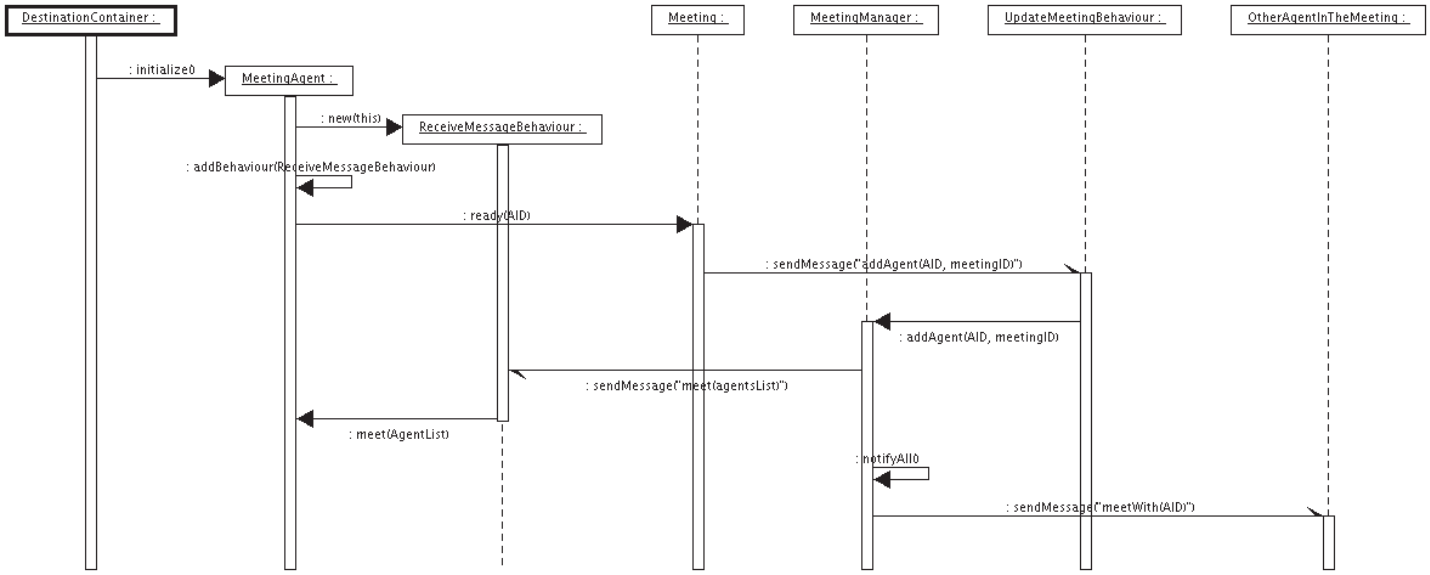

Figure 6: Sequence Diagram of the Meeting Pattern in the Jade platform (after migrating)

this matter is dealt with by the platform. On the other hand, in JADE and Grasshopper, the *MeetingManager* is fully implemented by the programmer. Regarding reuse, except from the *MeetingAgent* that is implemented according to specific application requirements, all other entities can be reused. As for modifiability, the design approach adopted by the developer is decisive. In other words, developers must make a good use of resources. However, the JADE behaviours can lead to modular designs that make changes easier and safer than in the other platforms.

Applying Mobile Agent Patterns

According to the design methodology presented in this paper and the modelling approach proposed, design patterns can be applied at different abstract levels as follows.

In the platform independent design, independent pattern models can be combined to compose a possible solution to implement a given application. This basically consists in mounting class diagrams by combining class templates provided in the pattern documentation with classes in the application domain. Sequence diagrams are created to illustrate the scenarios based on the combination of sequences of each pattern used. However, additional classes and interactions can be identified. For instance, the solution may be also based on other paradigms such as client-server. Moreover, when facing new problems that can give rise to new patterns or variants of catalogued patterns. It is important to continuously document new patterns and variants in the catalogue.

At platform independent level, different combinations of patterns can be tried, so that the most appropriate one can be chosen according to specific project goals such as performance. Due to the simplicity of platform independent models when compared to dependent ones, it is usually more feasible to simulate and check the behaviour of the system and properties of interest when considering the former [16].

In the platform dependent design, the platform independent design is refined according to the corresponding dependent pattern models provided in the catalogue for the specific platform chosen to implement the application. In practice, this basically consists in adding concrete classes and implementation issues related to the platform.

To illustrate these ideas and a potential use of the Meeting pattern, consider an electronic-commerce application started from portable devices (PDAs). Features of a given product a client wishes to acquire are specified. The specification is transformed into a number of mobile agents that can migrate in the Web in the search of products that meets it. While the search is pursued, the client can disconnect his device and/or start other applications. Such devices usually allow only one operation at a time (mobile phones).

The use of multiple agents can maximise search results and reduce the time to perform it. Given the possibility of different agents visit the same places, collect redundant information and collect information that meets only partially the client specification, it would be interesting if they could meet in order to exchange information and also synthesising results. This can avoid unnecessary traffic in the network as well as reduce the amount of data to be delivered to the client. These can be critical in a mobile computing environment where connections are not very trusty and resources in portable mobile devices are constrained.

In the sequel, the design of the application is described. For the lack of space, only class diagrams are shown.

Architectural Design. Figure 7 illustrates an architecture of the proposed electronic-commerce application. Clients call for services from the domain provider. Services are offered based on a net of ¨market places¨ in the Internet.

To search for a product, a client, possibly disconnect, spec-

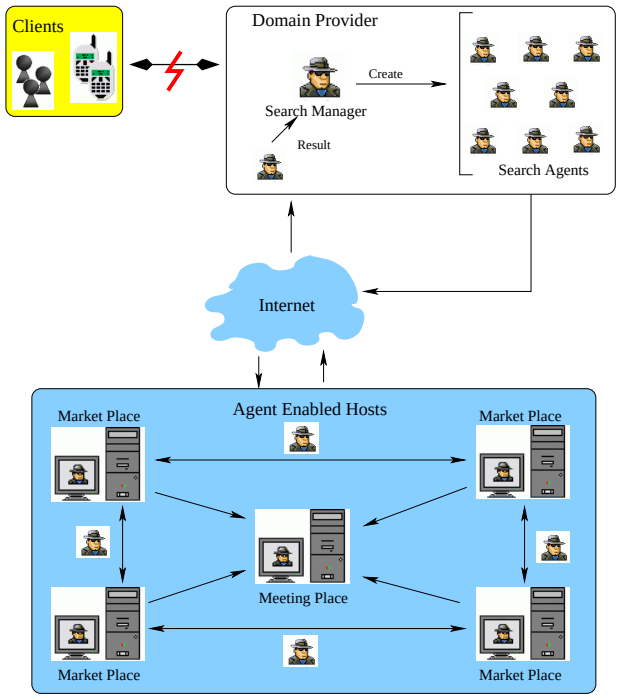

Figure 7: Electronic-Commerce Application Architecture.

ify requirements such as weight, category, brand and price. Then, the resulting query is the sent to the domain provider where a manager agent delegates the search task to mobile agents. During the search, the client may remain disconnect or performing other tasks. To perform the search task, the mobile agents move from one network host to the other, according to a list of possible market places. Since each agent is independent from the other, it is possible that two agents visit the same place. This redundancy is important to cope with network instability and guarantee that all places will be visit. Therefore, at the end of the search, mobile agents meet in a meeting host to synthesise results by applying, for instance, predefined heuristics.

Platform Independent Design. A class diagram with the main entities in the platform independent design is shown in Figure 8. Note that two patterns are combined: the meeting and the master-slave pattern [14]. This diagram is basically a combination of the class diagram of the meeting pattern presented in Figure 3, the class diagram of the master-slave pattern and conceptual model of the application [15]. Note that *Search Agent* is a *MeetingAgent* as well as a *SlaveAgent*. Also, *SearchManagerAgent* is the master agent running on the *Domain Provider*. Finally, *CollectorAgent* is the agent running on the client device that is responsible for collecting and sending the search requirements.

Platform Dependent Design – Aglets. From Figure 9, *CollectorAgent* and *SearchManagerAgent* agents must implement the *MobilityListener* interface, since they are stationary. All agents extend the *Aglets* class. Note that the *Meeting* and *MeetingManager* classes are implemented in the platform as explained in the previous section.

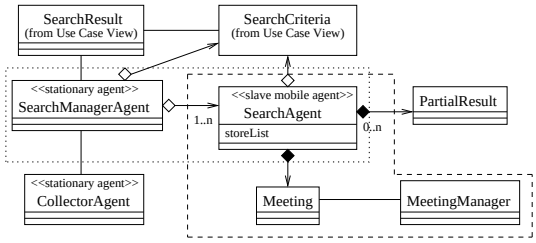

Figure 8: Platform Independent Class Diagram.

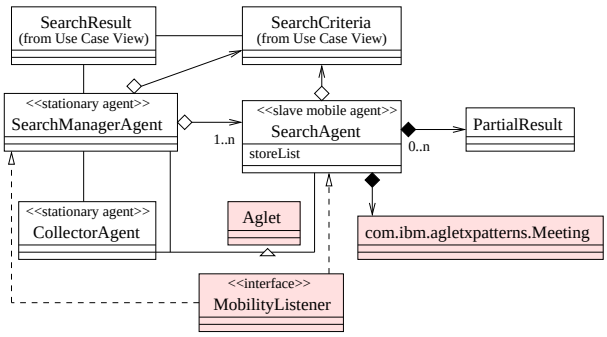

Figure 9: Class Diagram in the Aglets Platform.

Platform Dependent Design - JADE. The class diagram is shown in Figure 10. In this platform, agent functionality is implemented in behaviour classes. *ReceiveResultbehaviour* and *ReceiveMessageBehaviour* are responsible for message receiving in the *SearchManagerAgent* and *SearchAgent* respectively. The *JobBehaviour* is responsible for the search task in the *SearchAgent* agent. Finally, the *UpdateMeetingBehaviour* is responsible for receiving messages from agents that update a meeting (see Figure 6).

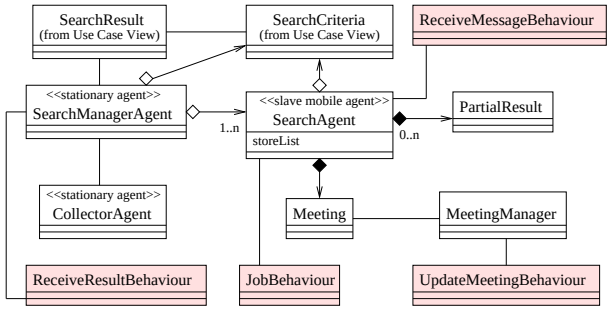

Figure 10: Class Diagram in the JADE Platform.

From this application, it can be noticed that the use of design patterns can improve productivity and lead to a platform independent solution that can be implemented in different platforms. Design patterns are key to construct mobile agent-based applications due to the complexity of implementing such applications. Moreover, its likely that any solution relies on patterns of agent mobility, communication and task delegation. On one hand, the platform independent design gives an abstract view of the solution to be implemented. On the other hand, platform dependent design

considers detailed issues related to Aglets and JADE. At this stage, decisions are taken before implementation can effectively be started. Finally, the approach to modelling patterns presented in this paper clearly promotes their use. In our example, both independent and dependent pattern models are essential and useful guides.

Concluding remarks

The use of mobile agent technology in the development of applications is highly dependent on the existence of meaningful design patterns. This is even more critical in the mobile computing setting. We presented an approach to modelling and applying mobile agent design patterns at different abstract levels using the UML notation. Patterns have been considered in three different mobile agent platforms. We showed that it is important to construct a platform independent model focusing on abstract features of mobile agents and platforms. This can promote reuse of the corresponding design solution when implementing in different platforms. Also, logical design can proceed independently of specific platforms. On one hand, it can be faster for an expert user in a particular platform to construct the dependent models directly. However, by developing the independent model we get advantages such as reusability and comprehensibility of the design. This is a realistic approach to deal with the constantly evolving technology. Once the features of a given platform are known, independent models can be easily refined towards an implementation. It is important to remark that to make maintenance easier, dependent models must be refinements of the correspondent independent model.

As further work, we aim at providing tools to support documentation and refinement of patterns in UML. Also, extensively apply the approach to develop case studies. From them, new patterns and variants can be identified. A catalogue of patterns is under construction, presenting, together with usual requirements of documenting object-oriented patterns, a platform independent and dependent models.

Acknowledgements This work is supported by CNPq – Brazilian Research Council, MÓBILE Project (Process 552190/2002-0). First author is supported by CAPES.

References

[1] Y. Aridor and D. B. Lange. Agent design patterns: Elements of agent application design. In *Proceedings of the Second International Conference on Autonomous Agents*, pages 108–115. ACM Press, May 1998.

[2] F. Bellifemine, G. Caire, T. Trucco, and G. Rimassa. *JADE PROGRAMMER'S GUIDE*. http://jade.cselt.it/docs, February 2003.

[3] G. Cabri, L. Leonardi, and F. Zambonelli. Engineering mobile-agent applications via context-dependent coordination. In *Proceedings of International Conference on Software Engineering - ICSE'2001*, 2001.

[4] L. Cardelli. Abstractions for mobile computation. In *Proceedings of Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, 1999.

[5] D. Deugo, F. Oppacher, J. Kuester, and I. Von Otte. Patterns as a means for intelligent software engineering. In *ICAI-99*, School of Computer Science, Carleton University, Ottawa, Ontario, Canada,, 1999.

[6] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Trans. on Software Engineering*, 24, 1998.

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

[8] The Object Management Group. *The Mobile Agent System Interoperability Facility*. The Object Management Group, Framingham, MA, 1997.

[9] F. P. Guedes, P. D. L. Machado, and V. N. Medeiros. Developing mobile agent-based applications. In *XXIX Conferencia Latino Americana de Informática - CLEI 2003*, La Paz, 2003.

[10] Hayzelden and Bigham, editors. *Agents for Future Communication Systems*. Springer, 1999.

[11] IKV++, Germany. *Grasshopper Basics and Concepts*. http://www.grasshopper.de.

[12] W. Jansen and T. Karygiannis. Mobile agent security. In *NIST Special Publication 800-19*, 1999.

[13] C. Klein, A. Rausch, M. Sihling, and Z. Wen. Extension of the Unified Modeling Language for mobile agents. In Keng Siau and Terry Halpin, editors, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, chapter 8, pages 116–128. Idea Publishing Group, 2001.

[14] B. D. Lang and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.

[15] E. F. A. Lima. Formalização e análise de padrões de projeto para agentes móveis. Master's thesis, COPIN - Universidade Federal de Campina Grande, 2004. To be concluded in February 2004.

[16] E. F. A. Lima, J. C. A. de Figueiredo, and D. D. S Guerrero. Using coloured petri nets to compare mobile agent design patterns. *Electronic Notes in Theoretical Computer Science*, 2004. Selected Papers from VI Workshop on Formal Methods - WMF'2003. To appear.

[17] D. Milojicic, F. Douglis, and R. Wheeler, editors. *Mobility: process, computers and agents*. ACM, 1999.

[18] J. Mylopoulos, M. Kolp, and J. Castro. Uml for agent-oriented software development: The tropos proposal. In M. Gogolla and C. Kobryn, editors, *Proceedings of UML 2001*, volume 2185 of *Lecture Notes in Computer Science*, pages 422–441, 2001.

[19] A. Silva and J. Delgado. The agent pattern for mobile agent systems. In *European Conference on Pattern Languages of Programming and Computig, EuroPLoP'98*, 1998.

[20] Y. Tahara, A. Ohsuga, and S. Honiden. Agent system development method based on agent patterns. In *Proceedings of the 21st international conference on Software engineering*, pages 356–367. IEEE Computer Society Press, 1999.

[21] Yasuyuki Tahara, Nobukazu Toshiba, Akihiko Ohsuga, and Shinichi Honiden. Secure and efficient mobile agent application reuse using patterns. In *Proceedings of the 2001 symposium on Software reusability*, pages 78–85. ACM Press, 2001.