

A Modular Implementation Framework for Code Mobility

Cidiane Lobato¹

¹PUC-Rio
Computer Science Department
Rio de Janeiro, Brazil
+55 21 2540-6915 ext. 136
{cidiane,lucena}@inf.puc-rio.br

Alessandro Garcia²

²Lancaster University
Computing Department, InfoLab21
Lancaster, UK
+44 (0)1524 510-317
garciaa@comp.lancs.ac.uk

Carlos Lucena¹

Alexander Romanovsky³

³University of Newcastle
Computing Science School
Newcastle upon Tyne, UK
+44 191 222-8135

alexander.romanovsky@ncl.ac.uk

ABSTRACT

With the growing popularity of open distributed applications, mobile agents have naturally emerged as the fundamental technique for tackling the complexity of the emerging applications. However, the pervasive nature of code mobility issues implies that their implementation cannot be modularized based only on object-oriented (OO) abstractions and mechanisms. In fact, programmers of complex mobile agent systems frequently evidence the presence of mobility tangling and scattering in the modules of their systems. Despite these modularity breakdowns caused by code mobility, the developers have mostly relied on OO application programming interfaces (APIs) from mobility platforms and on the Java programming language. As a consequence, there is a pressing need for empowering developers with a modular implementation framework that supports a transparent, flexible incorporation of code mobility-specific capabilities into their applications. This paper presents an aspect-oriented software framework, called AspectM, that ensures: (1) improved modularization of the code mobility issues, (2) a seamless introduction of code mobility into stationary agents, and (3) overall enhanced variability of the mobile agent systems, such as flexible integration of these systems with distinct mobility platforms. The usefulness and usability of the AspectM framework has been assessed in the context of two medium-sized case studies from different application domains, and through its composition with two mobility platforms.

Keywords

Mobile Agent, Mobility Platform, Aspect-Oriented Programming.

1. INTRODUCTION

Using mobile agents, open distributed applications can be developed to run over the Internet in a much more flexible way than before [6]. As a result, code mobility capabilities can be exploited as a means to reduce the complexity of such contemporary applications. However, with mobile agent systems growing in size and complexity, developers are still facing the challenge of achieving enhanced system modularization in the presence of code mobility. The central problem is the invasive and widely-scoped nature of mobility concerns [8, 15], hindering the system modularity, composability, and changeability. It is widely recognized nowadays that these modularity problems mainly occur because the mobility

issues cannot be explicitly captured using only object-oriented (OO) abstractions and mechanisms [7-8, 15]. This is mainly due to the fact that the implementation of many of these mobility-specific concerns tends to crosscut other system concerns, such as the basic agent functionalities and coordination activities [8, 15].

Modular design support for mobile agents has been studied from different perspectives, including design patterns (e.g. [1]) and mobility frameworks supporting the structuring of code mobility concerns in software agents, such as Aglets [13], JADE [3] and RoleEP/EpsilonJ [15]. Although these frameworks provide OO APIs that offer a number of mobility abstractions and services, they also inherently bring a number of design breakdowns. First, they impose architectural restrictions on the agent design, which are responsible for the tangling and scattering of mobility-specific code over the system. Second, in order to introduce the mobility capabilities into systems, developers must usually modify the agent design to: declare that application agent classes extend specific API classes from mobility platforms, implement the API abstract methods, declare and possibly specify the implementation of the API interfaces, and explicitly invoke the API mobility methods on the system classes which are not created to address mobility concerns. Hence such implementation strategies result in a high coupling between the underlying models of mobility platforms and the design of mobile agent systems. Third, the usage of such APIs does not allow the reuse and explicit handling of the scattered mobility code as variability points in systems implementation. Finally, the lack of proper modularization also makes it difficult the composition of the mobility framework with infrastructures addressing other concerns, such as collaboration, learning, and distribution.

The contributions of this paper are twofold. First, it provides a systematic analysis of the modularity problems caused by the crosscutting nature of code mobility concerns. Second, it presents an aspect-oriented implementation framework, called AspectM (“Aspectizing Mobility”) exploiting aspect-oriented programming [11] to support a more straightforward introduction of code mobility into software agents. Aspects are used to improve the variability of the mobility concerns, such as a flexible choice of the mobility platform and improved composability with other frameworks in use. AspectM is implemented in AspectJ [12], an aspect-oriented extension of the Java programming language. Our framework has been identified and abstracted from its instantiation in different mobile agent applications [2, 7-8] and from the study of a number of mobility frameworks [1, 3, 13, 15] and can be applied to a wide range of mobile agent systems.

The paper is organized as follows. Section 2 presents the mobile agent systems in general and recalls fundamental definitions of aspect-oriented programming. Section 3 systematically discusses the modularity problems relative to code mobility in terms of a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Conference '04, Month 1–2, 2004, City, State, Country.
Copyright 2004 ACM 1-58113-000-0/00/0004...\$5.00.

case study. Section 4 shows the AspectM design, which allows the modularization of mobility code. Section 5 overviews the related work. Finally, section 6 presents conclusion.

2. BACKGROUND

2.1 Mobile Agent Systems

Mobile Agent Systems (MAS) may follow strong or weak mobility [6]. This work focuses on weak mobility since most mobility frameworks support this form of mobility [6]. A mobile agent system consists of a mobility platform and mobile agents being executed on it. The *instantiation* of the mobile agent is made only once when it is created. Every agent receives a unique id and an initial state. *Initialization* is performed each time when the agent arrives to a new host. *Destruction* means that the agent terminates all its activities and frees all the resources it was using. *Migration* represents a transfer of an agent from one host to another. The instantiation, initialization, migration, and destruction of a mobile agent define what we will refer as the *agent mobility protocol*. The instants where the migration action must be carried out are defined by the application developers, we will refer to them as *migration points*. In general, departure and arrival procedures data of mobile agents are related to execution contexts. Such data are called *itinerary* throughout this paper.

2.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [11] is an emerging programming paradigm with the goal of improving separation of crosscutting concerns at the implementation level through new abstractions and composition mechanisms. *Aspect* is the abstraction to support improved modularity of such crosscutting concerns. An aspect can crosscut one or more classes, changing their structure or dynamics. *Join points* are well-defined points in the dynamic execution of a system which are used to specify how classes and aspects are related. A collection of join points can be specified through a *pointcut*. *Advice* is a special method-like construct attached to pointcuts, which defines a crosscutting feature to affect the dynamic behavior of classes. An advice can run before, after, or around whenever a join point is reached. An aspect can also contain internal attributes, methods, and *inter-type declarations*. These declarations specify new members (attributes or methods) for classes to which the aspect is attached, or change the inheritance relationship between classes. Like classes, aspects can be defined as abstract and extended by concrete aspects; both methods and pointcuts can be qualified as abstract. AOP has been empirically examined in the context of different crosscutting concerns, such as exception handling [5], persistence and distribution [14], and design patterns [9]. However, AOP has not been fully explored an improved modularization of code mobility.

3. MODULARITY OF CODE MOBILITY

The literature has identified that code mobility is often crosscutting [8, 15], which has been confirmed by some empirical studies [7-8, 15]. This section illustrates the design problems relative to code mobility in terms of the solution using the JADE framework [3]. Figure 1 shows the tangling and scattering of mobility concerns in a partial representation of the Expert Committee (EC), an open multi-agent system that supports the management of paper submissions and reviews for scientific conferences.

The Case Study. EC is a classical example of MAS with design based on OO abstractions and mechanisms and on the Java language. EC will be also used in Section 4 to show the applicability of our proposal. For simplification, Figure 1 only shows some important classes, the others essentially follow the same pattern; we also omit the classes related to learning, adaptation, and autonomy. Each set of classes, surrounded by a gray rectangle, has the main purpose of modularizing a specific agent concern, namely interaction, collaboration, mobility, and basic concerns. For example, the collaboration concern is represented by the roles played by the agents: (i) chair, (ii) reviewer, (iii) PC member, and (iv) paper author. Each role is associated with a set of plans which are used to implement more sophisticated collaborative activities. Figure 1 illustrates that the chair role is associated with a plan for distributing review proposals. EC agents need to move in some circumstances, including when they are playing a specific role. For example, when an agent is playing the chair role, it needs to consult the reviewer profiles in order to optimize the paper distribution in terms of the research interests of each reviewer. If a reviewer profile is not available, it collaborates with a stationary agent and requests this agent to search for information of the specified reviewer. The stationary agent controls the local database and is able to query for the profile. However, if the information is not available in the database, the chair role needs to move and try to find the missing profile in remote environments. Note that EC agents and their roles use the JADE framework.

Modularity Breakdowns in the Presence of Code Mobility. In Figure 1, note that the mobility concerns crosscut classes implementing other agent concerns; they have a huge impact on the basic agent structure, and on the collaboration and interaction designs. Although part of the mobility concerns are localized in the mobility classes, such as `JADEAgent` and `Itinerary`, mobility-specific code is replicated and spread across several class hierarchies of a software agent. Figure 1 shows each crosscutting-related problem with a number surrounded by a circle. There are classes that represent the agent types and roles, extending the abstract `JADEAgent` class to incorporate the mobility capabilities. However the use of inheritance results in code replication as well as in both code tangling and scattering; the basic functionality and collaborative activities are amalgamated to mobility methods (problem ①). The agent and role classes also need to hold an explicit reference to mobility elements (e.g. `itinerary`) as attributes (problem ②). These classes have additional methods to manage these elements (problem ③). In addition, several methods have mobility code in them in order to define the agent migration points, w.r.t. the decisions on when the agent should move to a remote environment (problem ④), and when going back to the home location (problem ⑤). As a result, this code is replicated on various methods of plan, role, and agent type classes. Moreover, there can be a spread of usual preconditions and postconditions when an agent moves to another host (problem ⑥). Such conditions can be either dependent or independent from the application agents. Finally, several classes have to implement the `Serializable` interface for allowing the objects, which are part of the agent, to be moved between hosts (problem ⑦). All these problems decrease the system reusability and maintainability, since adding or removing the mobility code from classes requires invasive changes in those classes. We cannot find a more modular solution even if we try OO refactoring or use another mobility framework; mobility is a crosscutting concern independently of the OO decomposition used [8, 15].

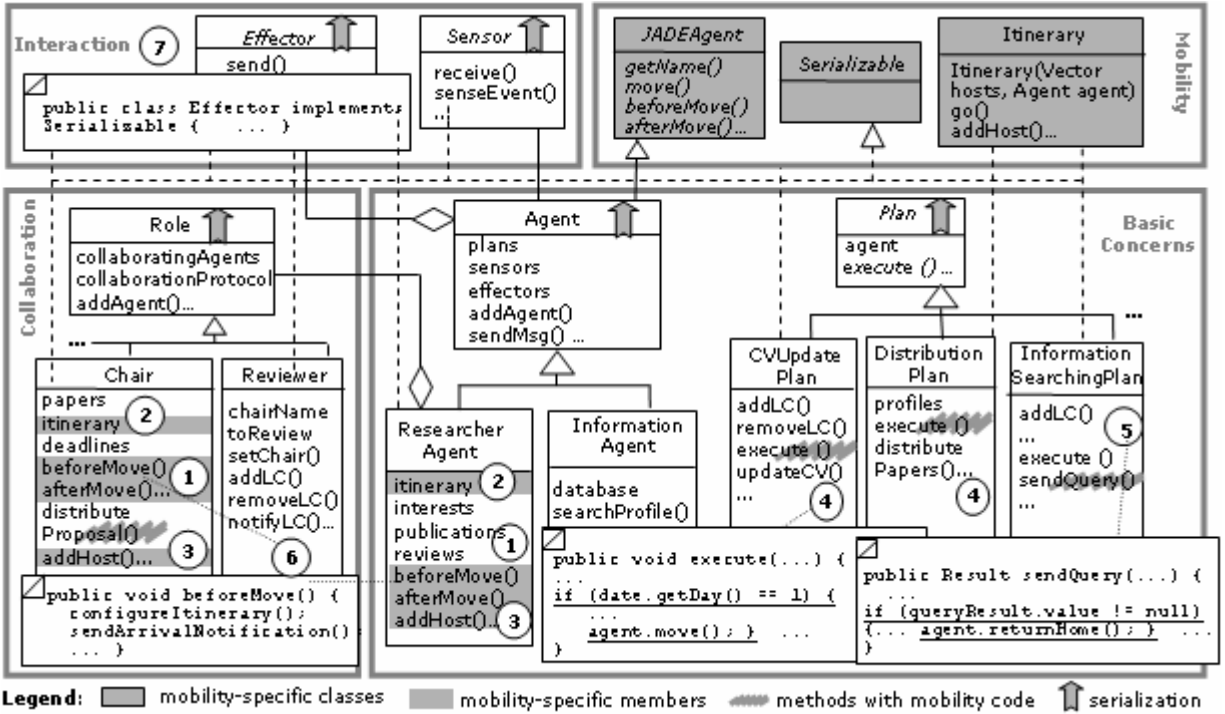


Figure 1. JADE Solution for Expert Committee and Crosscutting Mobility Concerns

Architectural Restrictions and Conceptual Mismatches. With respect to problem ②, the `Serializable` interface is just a representative example: OO APIs from platforms usually provide a number of interfaces with methods that are implemented by systems in order to make actions automatically be executed at specific moments through the mobile agent lifecycle; for instance, actions can be automatically executed immediately after the agent instantiation or just the migration. In this way, mobility-specific code spreads across several plan methods, roles, and agent types. Beyond the architectural restrictions on the agent design, the pervasiveness of existing mobility frameworks also introduces conceptual problems. The `Agent` class has to extend the `JADEAgent` class even when some agent types are stationary. For example, it is not possible to directly define the `UserAgent` class (Figure 1) as a `JADEAgent` subclass since the former already extends the `Agent` class. The same problem happens when specifying a specific role as mobile (e.g. the `Chair` class in Figure 4). Moreover, the use of OO frameworks can lead to potential conflicts. For example, the `JADEAgent` class defines an abstract method `getName()`; the `Agent` class also has a method `getName()` with a different purpose. The introduction of mobility in this system causes implementation clashes, and requires the renaming of this method and changes in the respective callers.

4. THE ASPECTM FRAMEWORK

4.1 Separation of Mobility Concerns

Figure 2 illustrates the detailed design of the AspectM framework by using the ASideML modeling language [4]. In this design language, an aspect is represented by a diamond and is composed of internal structure and crosscutting interfaces. The *internal structure* declares the aspect's internal attributes and methods. A *crosscutting interface* specifies the way the aspect affects one or more classes. Each crosscutting interface is presented using a

rectangle symbol with compartments. A crosscutting interface is composed of inter-type declarations, pointcuts, and advice (Section 2.2). The first compartment represents inter-type declarations, and the second one represents pointcuts and their attached advice. We have also enhanced the notation in Figure 2 to explicitly distinguish the hot spots (variable parts), which are marked with a star, from the frozen parts of the framework. The aspects in the AspectM framework are classified in two main categories: (i) the *Mobility* aspects, and (ii) the *Event* aspects (see Section 4.3). The purpose of the *Mobility* aspects is to decouple the mobility concerns from the basic functionalities and other system concerns. To do that, the abstract *Mobility* aspect contains pointcuts which capture the instantiation, migration, initialization, and destruction of mobile agents. It also contains the advice that are associated with these pointcuts. These advice are implemented following a general pattern: events are picked out, conditions are checked, and the appropriate mobility-specific methods are invoked. In other words, the *Mobility* advice are the AspectM frozen spots. For example, the migration advice is responsible for checking the need for the agent roaming and for calling the mobility actions in an abstract way. Thus, the *Mobility* advice correspond to the agent mobility protocol (Section 2.1).

Note that the `doAfterAgentInstantiation()` and `doAfterArrivalHost()` methods in the `ChairMobility` aspect correspond to control flow's return to the EC-specific procedures. Conversely it does not occur in migration and destruction protocols, where control flow is not returned to the EC system. Note also that the instantiation and migration pointcuts depend on application; they run after join points such as those from a `Plan` subclass (Figure 2); we need to specify on *Mobility* subspects the elements affected by the `instantiation_()` and `migration_()` advice. The initialization and destruction pointcuts are detected directly from platforms.

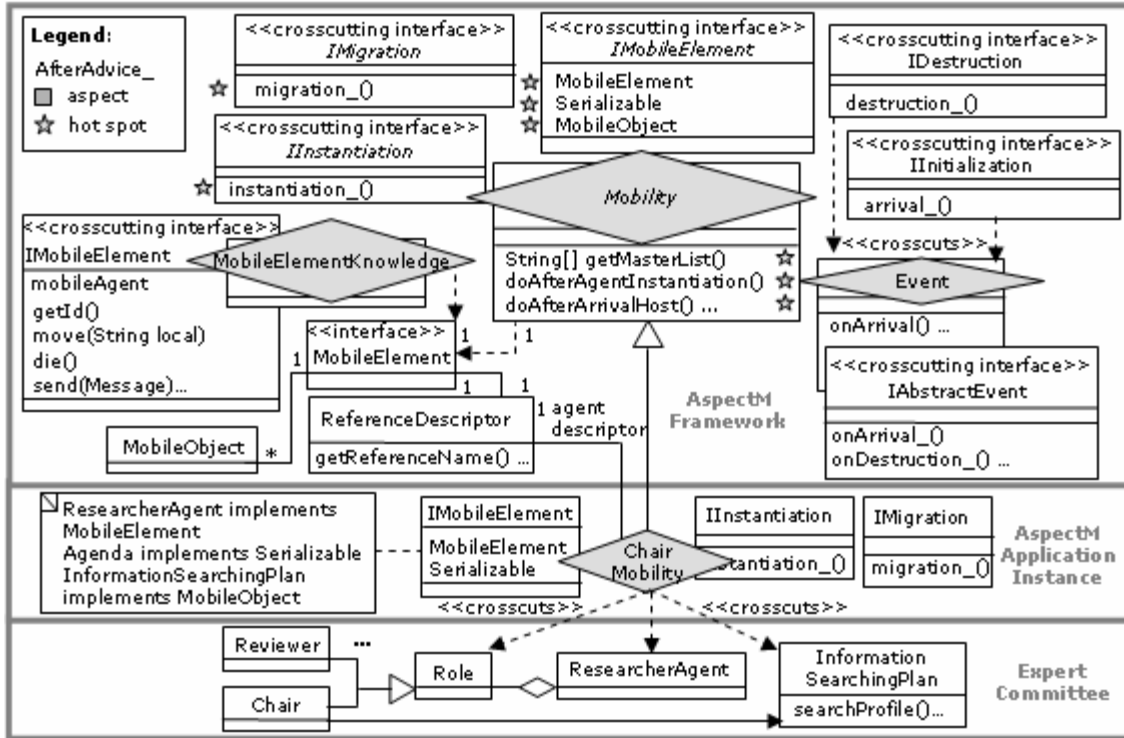


Figure 2. AspectM Solution for Expert Committee

In addition to the *Mobility* and *Event* aspects, we introduce the *MobileElement* interface, used by the *Mobility* aspects to delegate the mobility protocol actions to a specific platform. More specifically, this interface defines mobile agent-specific services abstracted from the distinct platforms, and is responsible for delegating to a specific mobility platform the services provided by its interface. These services, usually important to the MAS developers, include: (i) methods that grant access to ids (*getName()*, *getId()*, etc.); (ii) methods related to the mobile agent lifecycle (*move()*, *clone()* and *die()*, etc.); and (iii) methods for agent messaging (*send()*, *sendAsync()*, etc.). Thus, the *MobileElement* interface implements the methods that usually crosscut the OO design of software agents. An inter-type declaration is used to specify if an application class implements the *MobileElement* interface. An additional declaration is used to define which objects implement the *Serializable* interface. These declarations are represented by the specification of the *IMobileElement* crosscutting interface in Figure 2.

Note that the *Mobility* aspect holds a reference to the OO mobility framework in use and encapsulates the mobility protocol interactions with a specific platform (see Section 4.3). Figure 2 shows that all the mobility code is localized in the *Mobility* aspects. As a result, the agent and role classes are not intermingled with mobility code, therefore improving their modularity, reusability and changeability. In fact, the *Mobility* aspects modularize the crosscutting concerns presented in Section 3, thereby solving the problems ①, ②, ③, ④, ⑤, ⑥ and ⑦.

4.2 Transparent Introduction of Mobility

The AO design in Figure 2 uses AOP to make an explicit separation of mobility concerns in MAS possible. The mobility tangling and scattering problems are solved in AspectM by combining the

following design decisions. We use (i) the *Mobility* aspect for the generic code mobility, (ii) a *Mobility* subsaspect for each mobile element (role, agent, or plan) in order to prevent the explicit invocation of mobility methods in the MAS classes, and (iii) the *Mobility* pointcuts to bridge the AspectM framework with the agent-based application.

Following these ideas, we invert the way in which mobility concerns are typically implemented in MAS: OO abstractions and mechanisms, as inheritance and delegation, are replaced by AO abstractions and mechanisms. The latter ones are used to crosscut mobility join points over MAS at the same time they provide mobility modularization. For example, in order to solve the problem related to explicit inheritance-based extensions involving MAS classes and elements of the OO frameworks, we have used AO programming languages idioms [10] that allow the use of interfaces as if we were using abstract classes; mobility methods can be called by an agent or role class just by the use of interfaces while applications can maintain their own agent hierarchies.

Therefore, in order to introduce mobility-specific concerns into a stationary agent, the designer of mobile applications only have to specify the following AspectM hot spots: the elements to be defined as mobile, the instantiation pointcut and methods, the initialization methods, the migration points, the definition of which objects will be moved together with the mobile element, and the definition of the serializable elements. Figure 2 shows that the *ChairMobility* aspect extends the *Mobility* aspect in order to specify the *Chair* mobility-specific behavior. Concrete implementations of the AspectM hot spots are defined for the context of the *Chair* role: (1) the *ResearcherAgent* type implements the *MobileElement* interface, making the agent type becomes mobile (the agent type must be mobile because such a role depends on the agent knowledge wherever it is transferred); (2) the *IN-*

formationSearchingPlan type implements the MobileObject interface, making the plan becomes an object that will be moved with its respective agent; (3) the Agenda type implements the Serializable interface, which allows an Agenda object to be moved together with its respective instance; (4) we make concrete the ChairItinerary class, the getItineraryType() and getContextList() methods; (5) we make concrete the instantiation pointcut, which triggers the agent instantiation protocol (agentInstantiation_() advice); (6) we specify that the InformationSearchingPlan searchProfile() method execution is the migration pointcut; and (7) the initialization EC procedures to be executed on the doAfterArrivalHost() method.

All we have described confirms that the AspectM framework in general promotes a seamless introduction of mobility concerns into stationary agents. Introducing mobility concerns into agents corresponds to the user's task of making concrete a Mobility aspect for each stationary agent on his design. Otherwise, in order to turn a mobile agent into a stationary, we just have to remove the concrete Mobility subaspect corresponding to this agent.

4.3 Integration with Distinct Platforms

Once mobility platforms present meaningful differences in their implementations for context and messaging services, a general strategy of maintaining an agent reference table is applied in order to integrate MAS with distinct platforms. Through the use of such a table, it is possible to encapsulate contexts and message proxies through the Mobility aspects. In this way, considering the ReferenceTable class that implements the reference table concept, Figure 3 presents the AspectM design elements which are used to support the integration between MAS and distinct platforms. The hot spots corresponding to each platform are the following:

ReferenceManager Class is responsible for (i) the reference table instantiation and update on each agent instantiation, initialization or destruction, and (ii) response to common requests, such as

getting the agent list in a specific context. This class is a stationary agent that is a singleton.

ReferenceCreator Class is responsible for (i) the context creation for mobile agent execution at a specific host (createContext()), (ii) the instantiation of mobile agents on this context (createAgent()), (iii) the starting of platform services for instantiated agents (startAgent()), and (iv) the template method for the agent instantiation (createMobileAgent()). This class is implemented as a singleton for each context to be instantiated.

MobileAgent Class defines mobile agent services abstracted from distinct platforms. In other words, it is responsible for delegating to a specific mobility platform the agent services provided by its interface. These services include methods, such as: getName(), getId(), move(), clone(), die(), send(), sendAsync(), and so forth. This class communicates with the ReferenceManager in order to reply to common requests, such as getting the agent list in a specific context.

Event Aspect allows detection of relevant platform-specific join points, such as the mobile agent initialization, and destruction.

MessageParser Class executes parsing between platform-specific message formats and the AspectM message format.

Figure 3 shows that the Mobility aspect crosscuts the Event aspect when the onArrival() method is executed. This method is invoked on the Event onArrival() after advice. In turn, the onArrival() advice is executed just after the detection of the initialization pointcut related to a specific platform, which must be concrete in an Event subaspect. This same detection strategy can be used to access other agent lifecycle events, such as destruction, cloning, and message exchanges. Note that the ReferenceManager and the MobileAgent classes are interfaces with an abstract class behavior [10]. Note also that the MobileElement class bridges an AspectM application instance with an AspectM instance platform.

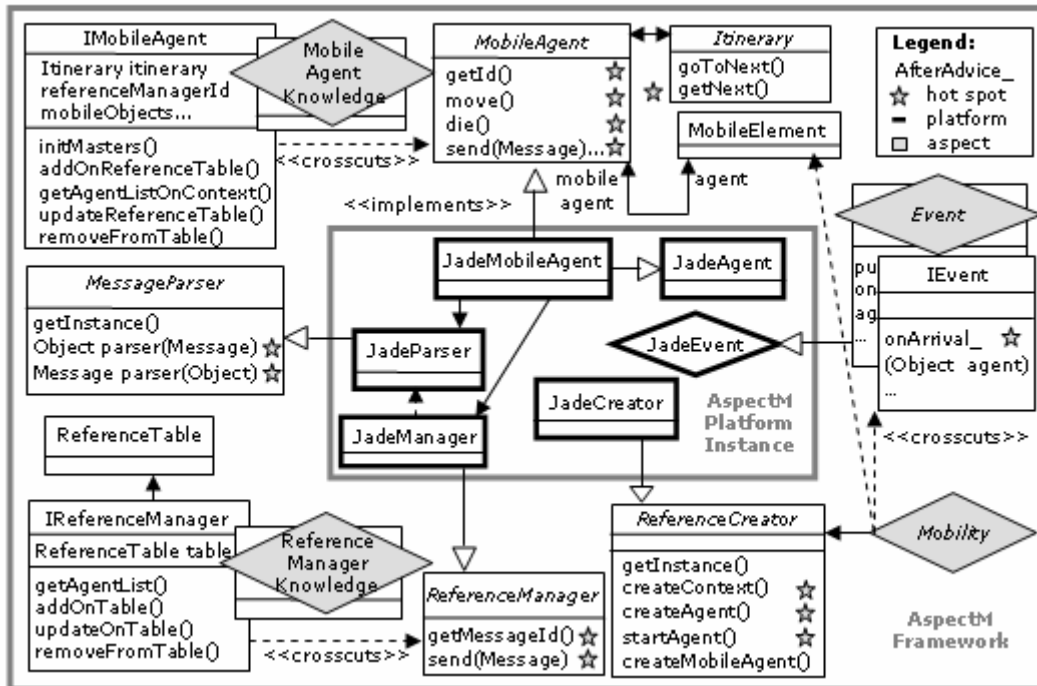


Figure 3. Integrating the JADE Mobility Platform with the AspectM Framework

5. EVALUATION AND RELATED WORK

To the best of our knowledge, the Epsilon/J reflective framework, which supports the RoleEP approach [15], is the only solution that has some explicit features for improving the separation of mobility concerns. However, this approach supports only role mobility but not agent mobility. In addition, the agent programmers have to extend several Epsilon/J interfaces and abstract classes, which decreases the mobility modularization. In RoleEP, the use of *binding-operation* [15] eliminates the necessity of AOP-style weaving. Inter-type declarations in AspectJ can be replaced by adding role methods through binding-operation. However, the advice construct does not correspond to any model constructs in RoleEP. This is a weak point of RoleEP, which often causes code duplication [15]. Finally, the Epsilon/J framework is dependent on the Aglets platform. In addition to the comparison with EpsilonJ, we have evaluated the AspectM framework to implement two case studies: the EC system (Section 3), and the MobiGrid framework [2]. In the following, we discuss some observations we have made on how AspectM was effective to address the problems discussed in Section 3 in those case studies.

Addressing Restrictions and Conceptual Mismatches. Section 3 has shown that an application agent superclass must extend OO mobility framework classes even when some agent or role types in hierarchy are stationary. Introducing mobility concerns into agents may also require design and implementation changes, such as the renaming of methods and changes in the respective callers. The solution for these problems is achieved by replacing the direct use of OO abstractions and mechanisms, as inheritance and delegation, by the AO abstractions and mechanisms (see Sections 4.1 and 4.2). Our case studies [2, 7-8] show that this replacement effectively promotes the modularization of the mobility concerns.

Promoting Superior Customizability and Variability. AspectM is generic enough and independent of specific mobility frameworks. It has some abstract intermediary classes and interfaces to bridge our framework with the chosen mobility platform (Section 4.3). Reuse of services of the mobility platform is achieved because AspectM provides a customization point to plug in a specific mobility platform. We have evaluated the integration between our case studies and frameworks provided by Aglets [13] and JADE [3]. The AspectM design has presented a good stability. In order to change from one framework to another, MAS developers only have to perform some setup. There was no impact on the design and implementation of other agent concerns.

Enhanced Composability. As discussed in Section 3, code mobility affects not only the agent basic functionalities, but also other important agent concerns (e. g. collaboration, interaction, and learning). In our case studies [2, 7-8] we have used aspects to improve the modularization of these crosscutting agent concerns. Since the mobility concern is related to these concerns, the presence of sophisticated composition mechanisms is important to modularly specify the relationships between the mobility and the other agent concerns. Although the AO composition required some refactoring to expose certain join points to other aspects, it was more straightforward than the OO composition, including the scenarios involving the composition between the AspectM framework and infrastructures [2].

Measurements. In order to provide a more concrete idea of the benefits of the AspectM implementation, we now provide the comparative percentage of structural elements (classes, methods,

etc.) in both OO and AO implementations of our case studies: the numbers related to the AspectM are smaller in the order of 20%.

6. CONCLUSION AND FUTURE WORK

The facets of a mobility strategy should be transparent to the rest of a mobile software system so that changes in the mobility concerns have no impact on the implementation of the other agent concerns. This paper presents an aspect-oriented framework that improves the separation of the mobility concerns, which typically cut across the modularity of several classes and methods of software agents. It allows not only the specification of the basic mobility behaviors, but also the specification of the agent types or roles that are mobile, the declaration of the traveling circumstances, the calls to departure, and the control of agent itinerary. Since the mobility strategies explored in MAS may vary as the system evolves, the AspectM framework is more flexible in supporting the changes in the used mobility strategy. Moreover, the use of our framework results in fewer classes, operations, and attributes, since agent classes do not need to incorporate mobility code. As a future work, we are planning to support dynamic re-configuration of mobility issues in the AspectM framework. To achieve this we will explore the use of dynamic weavers.

7. REFERENCES

- [1] Aridor, Y., Lange, D. Agent Design Patterns: Elements of Agent Application Design. *Proc. Int'l Conf. on Agents '98*, 108-115.
- [2] Barbosa R., and Goldman, A. MobiGrid: Framework for Mobile Agents on Computer Grid Environments. In *Proceedings of MATA '05*, Springer-Verlag, 2005, 147-157.
- [3] Bellifemine, F., Poggi, A., and Rimassi, G. JADE: A FIPA-Compliant Agent Framework. In *Proceedings of the Practical Applications of Intelligent Agents and Multi-Agents*, 1999, 97-108.
- [4] Chavez, C. A *Model-Driven Approach to Aspect-Oriented Design*. Ph.D. Thesis, PUC-Rio, Rio de Janeiro, Brazil, 2004.
- [5] Filho, F., Rubira, C., Ferreira, R., Garcia, A. Aspectizing Exception Handling: A Quantitative Study. In: "Advances in Exception Handling Techniques", LNCS, May 2006.
- [6] Fuggetta, A., Picco, G., and Vigna, G. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24, 5, (1998), 342-361.
- [7] Garcia, A., Lucena, C., and Cowan, D. Agents in Object-Oriented Software Engineering. *Software: Practice & Experience*, 34, 5 (May 2004), 489-521.
- [8] Garcia, A. et al. *Separation of Concerns in Multi-Agent Systems: An Empirical Study*. In: C. Lucena et al. (Eds). "Software Engineering for MAS II". Springer-Verlag, LNCS 2940, Jan. 2004.
- [9] Garcia, A. et al. Modularizing Design Patterns with Aspects: A Quantitative Study. LNCS Transactions on Aspect-Oriented Software Development, Springer, Feb 2006.
- [10] Hanenberg, S. et al. AspectJ Idioms for Aspect-Oriented Software Construction. In *Proc. EuroPlop '03*, Irsee, Germany, 2003.
- [11] Kiczales, G. et al. Aspect-Oriented Programming. In *Proceedings of the ECOOP '97*, LNCS (1241), Springer-Verlag, Finland, 1997.
- [12] Kiczales, G. et al. An Overview of AspectJ. In *Proceedings of ECOOP '2001*, Budapest, Hungary, 2001.
- [13] Lange, D., and Oshima, M. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.
- [14] Soares, S., Laureano, E., and Borba, P. Implementing Distribution and Persistence Aspects with AspectJ. In *Proc. OOPSLA '02*.
- [15] Ubayashi, N., and Tamai, T. Separation of Concerns in Mobile Agent Applications. In *Pro. of the 3rd International Conference Reflection 2001*, LNCS 2192, Springer, Kyoto, Japan, 2001, 89-109.