Engineering Multi-Agent Systems with Aspects and Patterns

Alessandro Garcia[1] **Viviane Silva**[1] **Christina Chavez**[1,2] **Carlos Lucena**[1]

[1]Depto. de Informática, Grupo SoC+Agents,
TecComm/LES, PUC-Rio
Rua Marquês de São Vicente, 225 – 22451-900
Rio de Janeiro, RJ, Brazil

[2]Depto. de Ciência da Computação,
UFBA
Av. Ademar de Barros, s/n – 40170-110
Salvador, BA, Brazil

e-mail: {afgarcia, viviane, flach, lucena}@inf.puc-rio.br

Abstract *Objects and agents are software engineering abstractions that have many common concerns. However, agents are more complex entities since they encompass additional concerns: their state is driven by beliefs, goals, capabilities and plans, and their behavior is composed of a number of agency properties such as autonomy, adaptation, interaction, learning, mobility, and collaboration. A multi-agent system usually incorporates multiple objects and types of agents, with each agent type addressing distinct agency concerns. These agency concerns typically overlap and interact with each other, and so a disciplined scheme for composition is required. In this context, this paper presents and compares an aspect-based proposal with a new pattern-based proposal for building multi-agent software. Both proposals have the following goals: (i) minimize the misalignments between high-level agent models and object-oriented designs, (ii) promote the separation of agency concerns, (iii) provide explicit support for disciplined composition of agency concerns in complex software agents, (iv) incorporate flexible facilities to build different types of software agents, and (v) allow the production of multi-agent software systems that are easy to understand, maintain and reuse. We demonstrate the applicability of the two proposals through the Portalware system, a Web-based environment for the development of e-commerce portals.*

Keywords: *Multi-agent systems, object-oriented systems, software engineering, design patterns, aspect-oriented development.*

1 Introduction

The advances in networking technology have revitalized the investigation of the agent notion as an additional abstraction to engineer complex software systems. A multi-agent system (MAS) comprises multiple agents and objects, which are distinct abstractions used to model different entities from the problem domain. Agents are tailored to represent active entities that manipulate objects, which are commonly applied to reflect passive entities in the software system. In this context, MAS development is composed of two major phases: (i) the specification phase; and (ii) the development phase (i.e., design and implementation). The specification phase is concerned with finding out and documenting the agents and objects making up the system and their relationships. The objective is to produce an abstract description of the software system, which is a basis for detailed design and implementation. Software engineers apply high-level models and methodologies to identify objects, agents and their relationships. Nowadays, there are a number of modeling languages and methodologies for MAS development [9, 18, 31]. The result of this phase is expressed in a set of high-level agent and object models (Figure 1).

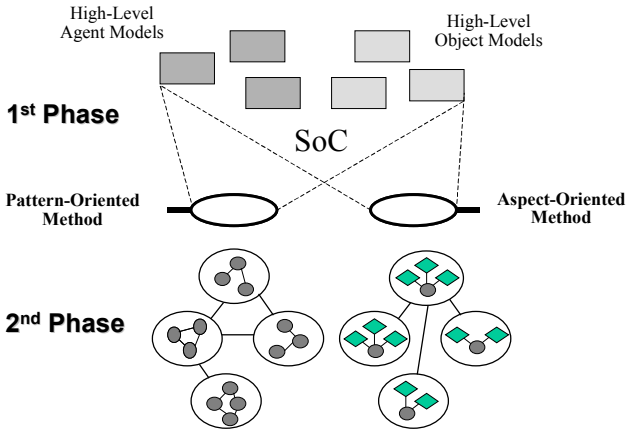

Figure 1: The Phases of MAS Development

The development phase focuses on transforming those high-level models into detailed design and implementation. Since object-orientation is the mainstream development paradigm, it is necessary to transform the agent models produced in the specification phase into object-orientated design models (development phase) for further implementation. However, it is not a trivial task due to the conceptual differences between agents and objects. Although objects and agents have common concerns [5, 28], agents are inherently more complex and consequently encompass additional concerns: their state is driven by beliefs, goals, capabilities and plans and their behavior is composed of a number of properties such as autonomy, adaptation, interaction, learning, mobility and collaboration. These agency concerns are not orthogonal. They present complex relationships since naturally overlap and interact with each other. Given such conceptual gaps between agents and objects, the major challenge in this phase is the mapping of the agent models into object-oriented design models. Software engineers should use principled methods to minimize misalignments between the high-level agent models and the detailed designs. Ideally such methods should explore separation of concerns techniques to encapsulate the agency concerns identified in the specification phase, leading to the production of MASs that are easier to understand, maintain and reuse.

In this context, the goal of this paper is threefold: (i) to present a method for MAS development that explores the benefits of aspect-oriented development (AOD) [12, 14, 15]; (ii) to propose a pattern-based method for MAS development; and (iii) to compare these methods by assessing their relative advantages and disadvantages in terms of MAS understandability, reusability, and maintainability. The central aim of both methods is to achieve improved separation of agency concerns in order to minimize misalignments with the specification phase. Both methods are independent from specific programming environments and languages. However, the first method [22, 30] uses aspects to capture the agency concerns identified in the high-level agent models and master the complexity of integrating agents into the object-oriented designs. Thus, each agent is represented by a single object and a set of aspects that modularize its overlapping and interactive concerns. The composition of the agency concerns is performed by means of a weaving process. On the other hand, the pattern-based method achieves improved separation of agency concerns using well-known design patterns that provide guidelines for the appropriate use of object-orientation mechanisms in the context of MAS development. In our comparative study, we present some results gathered when applying the two proposed methods to the MAS Portalware [11], a Web-based environment for the development of e-commerce portals. We have concluded that the composition mechanisms of AOD provides improved support for dealing with the complexity of agency concerns and producing MAS, which is easier to understand, maintain and reuse.

The remainder of this paper is organized as follows. Section 2 gives a brief description of definitions of multi-

agent systems. This section also introduces the example that is used throughout this paper to illustrate our approach. Section 3 offers an overview of software engineering approaches for agent systems, and introduces pattern-oriented development and aspect-oriented development. Section 4 presents the aspect-based method for designing MAS, and applies it to the Portalware system. Section 5 presents the pattern-oriented method and compares it with the aspect-based approach. Section 6 assesses the relative advantages and disadvantages of applying both proposals. Section 7 discusses related work. Finally, Section 8 presents some concluding remarks.

2. Multi-Agent Systems

2.1. Software Agents and Agency Concerns

Software agents are often viewed as complex objects with an attitude [6], in the sense of being objects with additional *agency concerns*. A software agent is not usually found completely alone, but often forming organizations with other agents within a *multi-agent system* (MAS). A MAS generally has several *agent types* [25], such as *information agents*, *user agents*, and *interface agents*. Each agent type typically includes different agent concerns. We can classify the agency concerns into three categories: (i) the agent *state*, (ii) the *agency properties*, and (iii) the agent *roles*.

Agent State. In general, the state of an agent is formalized by knowledge, and is expressed by mental components such as *beliefs*, *goals*, *plans* and *capabilities* [26, 28]. Beliefs model the external environment with which an agent interacts. A goal may be realized through different plans. A plan describes a strategy to achieve an internal goal of the agent, and the selection of plans is based on an the agent's beliefs. In this way, the behavior of agents is driven by the execution of their plans, which select appropriate capabilities in order to achieve the stated goals. Each plan is associated with pre-conditions and post-conditions [9]. Pre-conditions list the beliefs that should be held in order for the plan to be executed, while post-conditions describe the effects of executing a successful plan using an agent's beliefs.

Agency Properties and Agenthood. The behavior of an agent is composed of agency properties. Agency properties are behavioral features that an agent can have to achieve its goals. Table 1 summarizes the definitions for the main agency properties. These definitions are based on previous studies [20, 25, 26] and our experience in developing multi-agent applications [11, 13, 27]. In general, autonomy, interaction and adaptation are considered to be fundamental properties of software agents, while learning, mobility and collaboration are neither a necessary or sufficient condition for *agenthood* [26] (Figure 2). *Interaction* is the agency property that implements the communication with the external environment, i.e. the message reception and sending. An agent has *sensors* to receive messages, and *effectors* to send messages to the environment [20]. Since agents are autonomous software entities, the agent itself starts its control thread and decides whether to accept or reject incoming messages (the *autonomy* property). If a message is accepted, the agent may have to adapt its state. The *adaptation* property consists of processing an incoming message and defining which mental component is to be modified: beliefs can be updated, new goals can be set, and consequently plans can be selected. Software agents may have alternative properties, such as: (i) the *learning* property, i.e. extend or refine their knowledge when interacting with their environment, (ii) the *mobility* property, i.e. move themselves from one environment in a network to another, and (iii) the *collaboration* property, i.e. join a conversation channel with other agents.

AGENCY PROPERTY	DEFINITION
Interaction	An agent communicates with the environment and other agents by means of sensors and effectors
Adaptation	An agent adapts/modifies its mental state according to messages received from the environment
Autonomy	An agent is capable of acting without direct external intervention; it has its own control thread and can accept or refuse a request message
Learning	An agent can learn based on previous experience while reacting and interacting with its environment
Mobility	An agent is able to transport itself from one environment in a network to another
Collaboration	An agent can cooperate with other agents in order to achieve its goals and the system's goals

Table 1: An Overview of Agency Properties

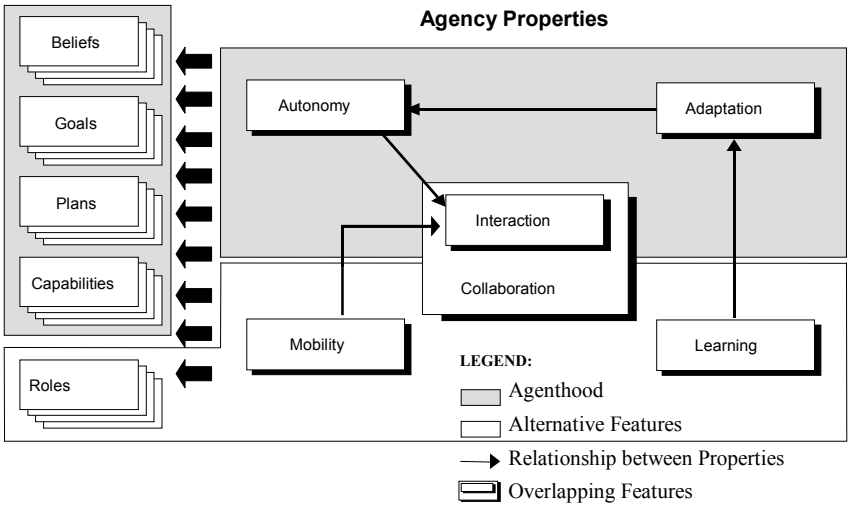

Figure 2: A Definition for Agenthood

Roles. A collaborative agent plays roles to collaborate with other agents. Roles are application-dependent and specific for each context. So, since software agents can collaborate while pursuing their goals in different situations, a collaborative agent includes different roles in order to work together in multiple contexts. In order to perform a collaboration, a plan is instantiated, and it chooses the eligible roles.

Interacting and Overlapping Properties. By the very nature of agency properties, these properties are not orthogonal – they interact with each other (Figure 2). For instance, adaptation depends on autonomy since it is necessary to adapt the agent's state (beliefs and goals) and behavior when the autonomy property decides to accept an incoming message. In addition, two agency properties overlap: interaction and collaboration. Collaboration is viewed as a more sophisticated kind of interaction, since the former comprises communication and coordination. Interaction only is concerned with communication, i.e. sending and receiving messages. During a collaboration, messages also are received from and sent to the participating agents. However, the collaboration property additionally defines how to collaborate, i.e. it addresses the coordination protocols. A simple coordination protocol consists of synchronizing the agent that is waiting for a response.

2.2. Portalware: A Case Study

Figure 3 illustrates the software agents in Portalware [11], a Web-based environment for the construction and management of e-commerce portals. Portalware encompasses three agent types: (i) interface agents, (ii) information agents, and (iii) user agents. Each of them implements the fundamental aspects defined by agenthood, but additionally includes specific agency concerns. Figure 3 summarizes capabilities and agency properties for Portalware agents. For the sake of brevity, we discuss in detail only Portalware's information agents. For a more detailed discussion about this example the reader can refer to [15]. Portalware users often need to search for information, which is stored in two different databases. Each information agent is attached to a database, and contains plans for searching for information. The search plan determines the agent's searching capability. An information agent can collaborate with another information agent when it is not able to find the information in the attached database. The agent plays the *caller* role in order to call the other information agent and ask for this information. Similarly, the latter performs the *answerer* role so that it can receive the request and send the search result back. Notice that both of them may include the caller and answerer roles since they can perform these different roles in distinct situations.

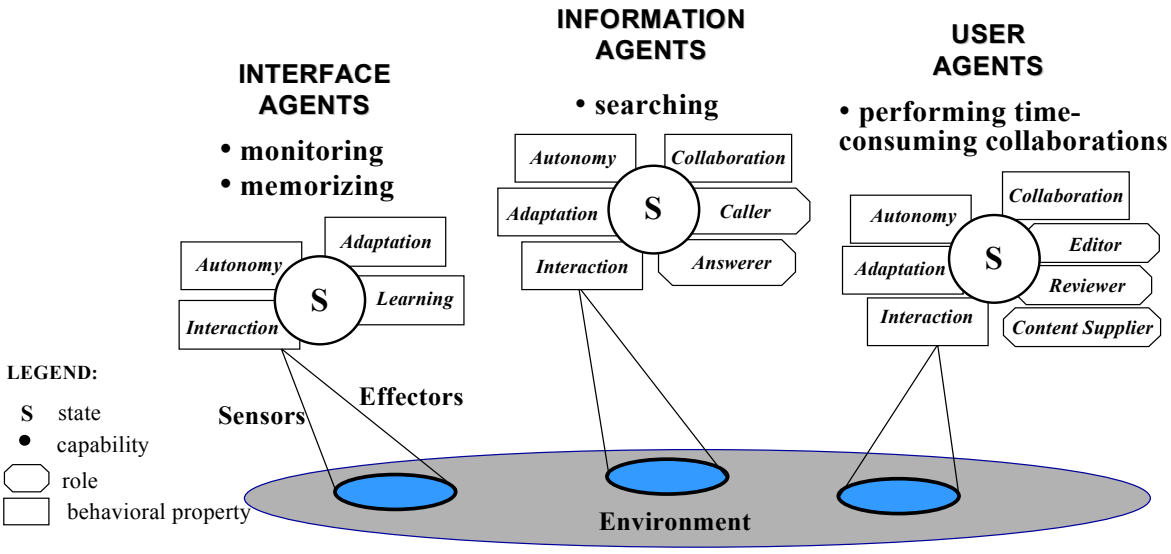

Figure 3: Portalware Agents

3 Software Engineering for MAS

The inherent complexity in the organization and introduction of software agents into object-oriented designs requires the use of principled software engineering methods. Modularity and separation of concerns are two well-established principles in software engineering, which use high-level abstractions to hide complexity by decomposing a software system into *modules* and *concerns*, respectively [30]. The importance of these principles increases as new technologies are introduced and software applications such as multi-agent applications, become more complex.

Types of Software Decomposition. From the viewpoint of *modular decomposition*, complex problems can be divided into smaller abstractions (modules), such as data, functions, objects, and agents. The common feature of these abstractions is that the decomposed parts are disjoint [24]. From the viewpoint of *concern decomposition*, complex problems can be divided into different abstractions, such as aspects [21] and subjects [18]. What distinguishes this concern decomposition from the module decomposition is the fact that the decomposed parts are not disjointed. In modular decomposition, any entity from the problem domain appears in only one of the pieces after decomposition – no entity appears in more than one piece. By contrast, an entity may appear in any number of concerns [24]. While modules are used to encapsulate concerns into the software system, nevertheless some concerns naturally cut across different application modules; such concerns are termed *crosscutting concerns*. As a consequence, modular decomposition and concern decomposition are complementary software engineering principles since module decomposition is not enough to encapsulate every relevant concern in a given software system.

Decomposition Approaches for MAS. We can classify current software engineering approaches for MAS development into two categories: (i) *agent-based software engineering*; and (ii) *object-oriented software engineering for agent systems* (see [15] for details). Both approaches concentrate on modular decomposition; however, the first approach proposes agents as the central unit of modular decomposition, and the second one realizes objects as the decomposition unit. Our proposal follows (ii) and additionally extends it with the application of recent advances in separation of concerns techniques (in particular, those provided by *aspect-oriented software development* [8, 21, 30]), exploring the benefits of both modular and concern decompositions to deal with the complexity of integrating software agents in the object model. To be able to evaluate our proposal against others, we propose a *pattern-based object-oriented software engineering* approach for agent systems, that also follows (ii). In the remainder of this section, we describe some characteristics of the selected development techniques — that is, pattern-based development and aspect-oriented development.

3.1. Pattern-Oriented Development

Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context [10]. Each design pattern describes a flexible and elegant solution to a recurring object-oriented design problem. Design patterns advocates reusability, flexibility and understandability in object-oriented software development. Object-oriented design patterns use the hierarchical modularity mechanisms of the object paradigm to provide good module decomposition, i.e. good object decomposition. Nevertheless, while hierarchical modularity mechanisms of an object-oriented paradigm are extremely useful and patterns offer a wide range of flexible ways to combine classes and objects, nevertheless they inherently are unable to modularize all concerns of interest in software engineering, mainly because some of them naturally cut across modules. Our pattern-oriented method (Section 5) shows how well known patterns can be used to design MAS.

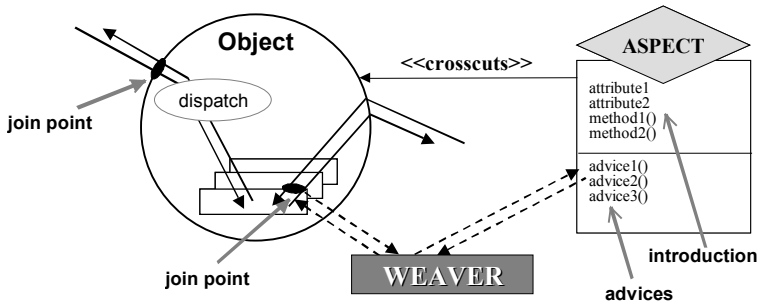

Figure 4: Mechanisms for Dealing with Crosscutting Aspects.

3.2. Aspect-Oriented Development

Aspect-oriented development (AOD) [8, 21] has been proposed as a foundation for improving separation of concerns in software construction. The central idea is that while hierarchical modularity mechanisms of object-oriented design and implementation languages are extremely useful, they inherently are unable to modularize all concerns of interest in complex systems. Thus, the goal of AOD is to support the developer in cleanly separating components (objects) and *aspects* (concerns) from each other, by providing mechanisms that make it possible to abstract and compose them to produce the overall system. Aspects are defined as system concerns that *crosscut* (i.e., cut across) components in the design and implementation of a system. Separating aspects (i.e. *crosscutting concerns*) from components requires a mechanism for composing – or *weaving* – them later. Central to the process of composing aspects and components is the concept of *join points*, the elements of the component language semantics with which the aspect programs coordinate. Join points are well-defined points in the dynamic execution of the program (Figure 4). Examples of join points are method calls, method executions, and field sets and reads.

Aspects are modular units of crosscutting concerns that are associated with one or more objects, comprised of pointcuts, advices, and introduction. *Pointcuts* are collections of join points. *Advice* is a special method-like construct that can be attached to pointcuts. In this way, pointcuts are used in the definition of advices. There are different kinds of advices: (i) *before advice* runs whenever a join point is reached and before the actual computation proceeds; (ii) *after advice* runs after the computation "under the join point" finishes, i.e. after the method body has run, and just before control is returned to the caller; (iii) *around advice* runs whenever a join point is reached, and has explicit control whether the computation under the join point is allowed to run at all. *Introduction* is a construct that defines new declarations of attributes and methods to the object to which the aspect is attached. *Weaver* is the mechanism responsible for composing the original base computation under a join point to the computation defined by one or more advices (Figure 4). AspectJ [22] is a practical aspect-oriented extension to the Java programming language. Up to the current version of AspectJ, almost all of the weaving process is realized as a pre-processing step at compile-time [22]. The next section presents our aspect-oriented method for MAS development.

4 The Aspect-Oriented Method

In this section, our aspect-oriented method is discussed in terms of: (i) *agent state*, (ii) *agent types*, (iii) *agency properties and agenthood*, (iv) *alternative agency*

properties, (v) *roles*, (vi) *composition of agency concerns*, and (vii) *agent evolution*. We adopt UML diagrams [4] as the modeling language throughout this paper. The design notation for aspects is based on [8]: aspects are represented as diamonds, the first part of an aspect represents introductions, and the second one represents pointcuts and their attached advices. Each advice is declared as: adviceKind (pointcut): adviceName, where adviceKind may be a before advice, an after advice, or an around advice.

4.1. Agent State

In our method, classes represent agents as well as their beliefs, goals and plans. The Agent class specifies the core state and behavior of an agent (Figure 5), and should be instantiated in order to create application's agents.

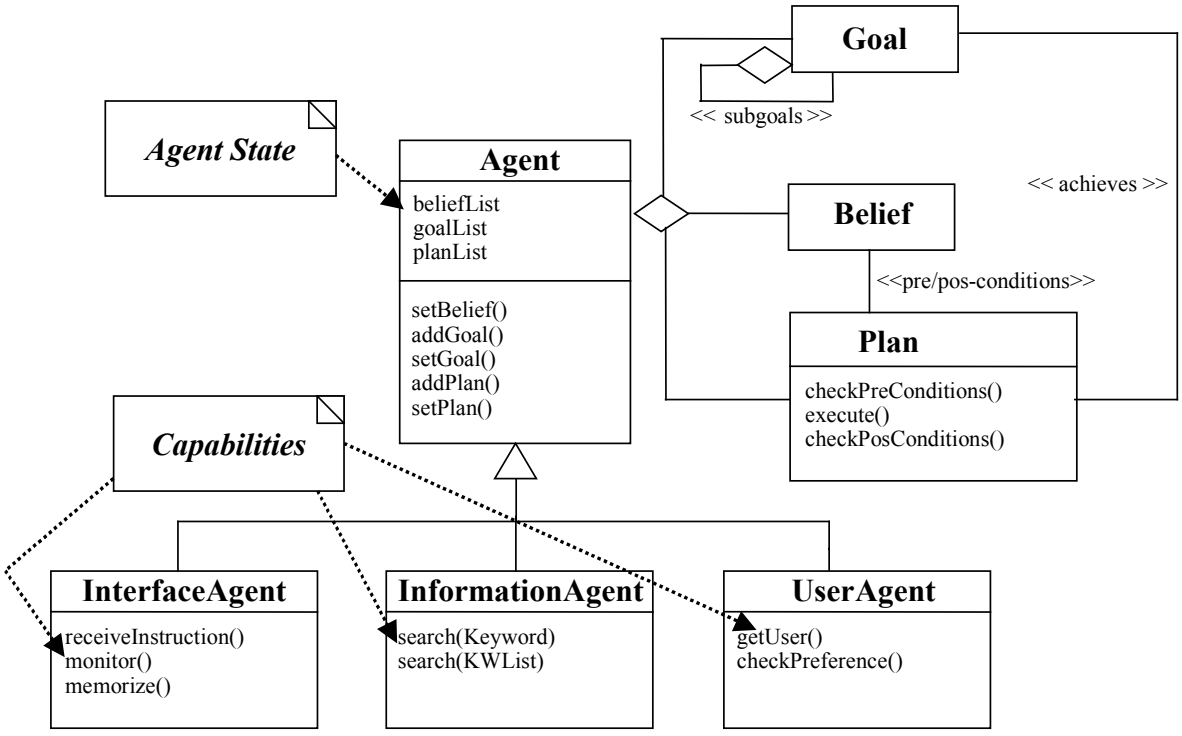

Figure 5: Agent State and Agent Types

Methods defined in the interface of the Agent class are used to query and update its state and to implement an agent's capabilities. Application designers must subclass the Belief, Goal and Plan classes to define beliefs, goals and the kinds of plans of their agents according the application requirements. The Plan class and its subclasses also define methods to check pre-conditions and set post-conditions (Section 2.1). A Goal object can be decomposed in subgoals. A Goal may have more than one associated Plan object.

4.2. Agent Types

Different types of agents are organized hierarchically as subclasses that derive from the root Agent class. The methods of these subclasses implement the capabilities of each agent type. Figure 5 illustrates the subclasses representing the different kinds of agents of our case study (Section 2.2): the InterfaceAgent class, the InformationAgent class, and the UserAgent class. The InformationAgent class, for example, defines the method search(Keyword), that provides the information agent's capability to search for information according to a specified keyword.

4.3. Agency Properties and Agenthood

Aspects should be used to implement the agency properties an agent incorporates. These aspects are termed *agency aspects*. Each agency aspect is responsible for providing the appropriate behavior for an agent's agency property. Figure 6 depicts the aspects, which

define essential agency properties for agenthood: (i) interaction, (ii) adaptation, and (iii) autonomy. These agency aspects affect both core states and behaviors of agents (Section 2.1).

For example, when the Interaction aspect is associated with the Agent class, it makes any Agent instance interactive. In other words, the Interaction aspect extends the Agent class's behavior to send and receive messages. This aspect updates messages and senses changes in the environment by means of sensors and effectors. The introduction part is used to add the new functionality related to the interaction property. The Sensor and Effector classes represent sensors and effectors respectively, and cooperate with domain-specific environment classes. When a message is received by means of a sensor, the Interaction aspect needs to update its inbox. So, the executions of the receiveMsg() method are defined as a pointcut (Figure 6), and the InboxUpdate() after advice is associated with this pointcut. Similarly, the OutboxUpdate() after advice is attached to the sendMsg() method in order to update the agent outbox. Since the process of sending and receiving messages is quite pervasive in multi-agent systems and cuts across the agent's basic capabilities, the implementation of this process as an aspect is a design decision that avoids code duplication and improves reuse.

The Autonomy aspect makes an Agent object autonomous, it encapsulates and manages one or more independent threads of control, implements the acceptance or refusal of a capability request for acting without direct external intervention (Section 2.1). For example, the Decision() around advice implements the decision-making process by invoking specified decision plans when a message is received. This advice is attached to a pointcut that represents a collection of executions of the receiveMsg() method.

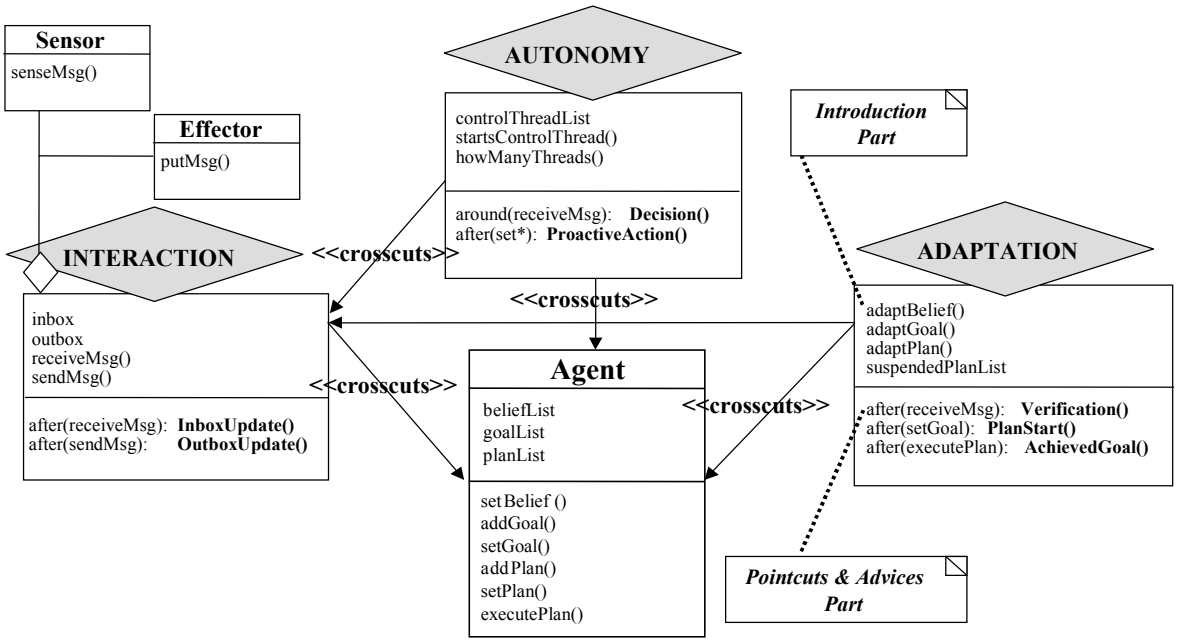

Figure 6: Agency Aspects and the Design for Agenthood.

The ProactiveAction() after advice implements the agent ability to act without direct external intervention (proactive behavior); for each method invocation where the method name matches the expression *set** (i.e., to each state change), this advice checks if a new plan must be started.

The Adaptation aspect makes an Agent object adaptive, adapting an agent's state (beliefs and goals) and behavior (plans) according to message receptions. As a consequence, this aspect crosscuts the Agent class and the Interaction aspect so that it is possible to perform state and behavior adaptations based on messages received from the environment by means of the receiveMsg() method. The Verification after advice

verifies if state change is needed and which state component must be adapted. The AdaptBelief(), AdaptGoal() and AdaptPlan() methods, defined in the introduction part, are responsible for updating beliefs, goals and plans, respectively. The Adaptation aspect also implements the following behaviors: (i) adapts the agent behavior by starting appropriate plans whenever new goals are set (PlanStart() after advice); and (ii) adapts the agent's goal list by removing a goal when this goal is achieved, i.e. when the execution of the corresponding plan is finished successfully (AchievedGoal() after advice).

4.4. Alternative Agency Properties

The agency concerns that are specific to each agent type are associated with the corresponding subclasses (Figure 7). Note that the different types of software agents inherit the agency aspects attached to the Agent superclass. As a consequence, the three agent types reuse the agenthood features and only define their specific capabilities and aspects. For example, the InformationAgent and UserAgent classes are associated with the Collaboration aspect, while the InterfaceAgent class is attached to the Learning aspect. The Collaboration aspect extends the Interaction aspect by implementing the synchronization of the agents participating in a collaboration (coordination protocol). It locks the agent sending a message as well as unlocks it when receiving the response. The Learning aspect introduces the behavior responsible for processing a new information when the agent state is updated.

4.5. Roles

Aspects are also used to implement the roles an agent may eventually play whenever they need to collaborate. These aspects are termed *role aspects*. Each role aspect defines the agent's activity within a particular collaboration.

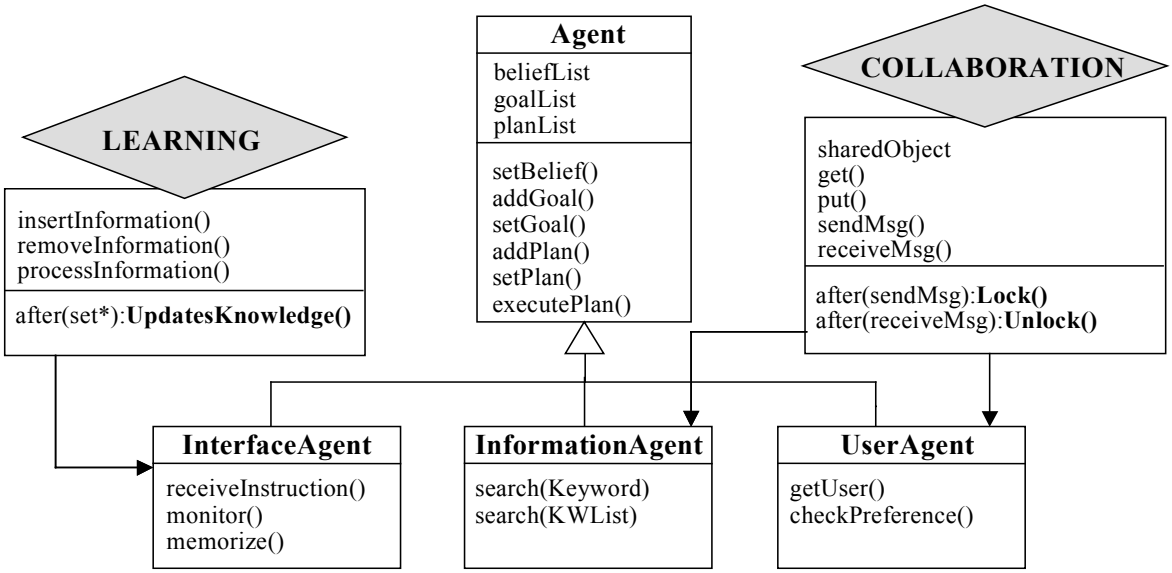

Figure 7: Alternative Agency Aspects.

Since an Agent object often needs to perform multiple roles, different role aspects can be used and easily associated with each object. As a result, role aspects decouple multiple roles from the agent's basic capabilities, which in turn improves understandability, evolution and reuse.

Figure 8 illustrates this situation for the information agents of Portalware (Section 2.2). An information agent needs to support the caller and answerer roles in order to cooperate with other information agents in different contexts. It must be able to receive or make calls. Thus, the Caller and Answerer role aspects are attached to the InformationAgent class. The Caller aspect introduces the ability to send the search request to the answering agent as well as the ability to receive the search result. Similarly, the Answerer aspect introduces the ability to receive the search request and to send the search result to the caller agent. The startsCaller() after advice is associated with executions of searching methods (search(*)) and is responsible for sending the search request when the agent itself is not able to find the required information. This advice checks results of

searching methods so that the caller is activated whenever the information is not found. Notice that these roles are introduced in a way that is transparent and non-intrusive.

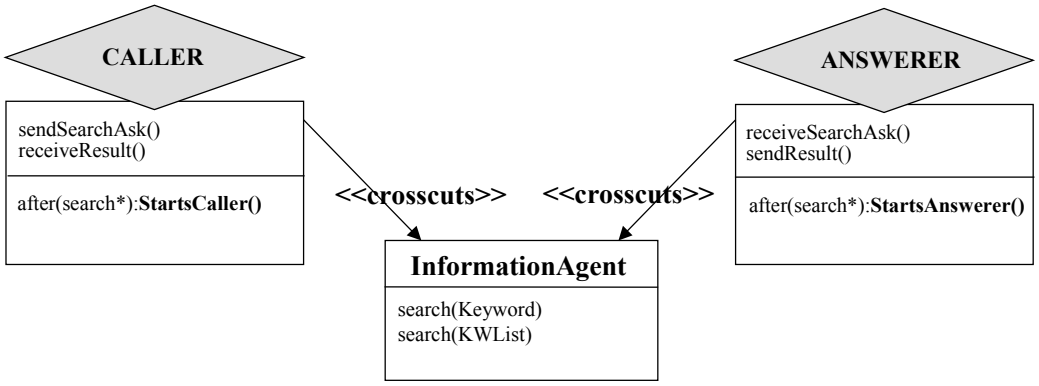

Figure 8: Agent Roles

4.6. Composition of Agency Concerns

Our approach establishes design rules that encompass the non-orthogonality of agency properties (Section 2.1). To capture the interaction among agency aspects, we define an advice to each agency aspect at the same pointcut. For example, the Autonomy aspect interacts with the Interaction aspect in order to receive the incoming message and decide if the message should be accepted. The Adaptation aspect interacts with the Autonomy aspect in order to adapt the agent state and behavior when an incoming message is accepted. As a consequence, these aspects implement different advices for the same pointcut that comprises executions to the receiveMsg() method.

We use inheritance to capture the overlapping nature between the Interaction and the Collaboration aspects. Collaboration includes the interaction behavior and refines it to add the coordination protocol. So, the Collaboration aspect is a subaspect of the Interaction aspect.

4.7. Agent Evolution

The behavior of software agents can evolve frequently to meet new application requirements. Suppose information agents do not need to cooperate with each other to find information. Instead, information agents are required to transport themselves from one environment in the network to another in order to achieve the searching goal. As a consequence, they do not need to play the caller and answerer roles, but are expected to be mobile. In our model, this modification is performed transparently, since agency aspects can be added to or removed from classes in a plug-and-play way. As a first step, the Caller and Answerer aspects are detached from the InformationAgent class without requiring any invasive adaptation for the other agent's components. The original behavior of the agent is maintained. As a second step, the Mobility aspect is associated to the InformationAgent class, introducing the ability to roam the network and gather information on the behalf of its owner. This association process uses the executions of searching methods (search(*)) as a pointcut.

At runtime, when the execution of a search() method is finished, the weaver deviates the program control flow to the Mobility aspect. The aspect evaluates the search result and if the information has not be found, this aspect is responsible for migrating the information agent to another host in order to start a new search for the required information.

5 The Pattern-Based Method

This Section presents our pattern-oriented method for developing MAS. Design patterns offer solutions that structure and discipline the representation of separated agency concerns in terms of objects, ensuring that the system can only change or evolve in specific, predictable ways. In this section, we focus on the description of agency properties and roles using design patterns, as well as on issues regarding aspect composition and agent evolution. Agent state as well as agent types follow the same design decisions already presented in Sections 4.1 and 4.2.

5.1. Agency Properties and Agenthood

The **Mediator** design pattern [10] is used to model the basic agency properties an agent incorporates and the

way they interact with Agent objects (Figure 9). The intent of the Mediator design pattern is to define an object (the mediator) that encapsulates how a set of objects (the colleagues) interact. The Mediator pattern lets us vary *how and which objects interact with each other, in a disciplined fashion*. In our solution, the mediator interface subsumes an agent's core state and behavior as well as the interaction protocol among agency properties. Agency properties are encapsulated as classes and play the role of colleagues in the pattern — i.e., they interact but do not refer to each other directly, only through the mediator. This pattern facilitates the addition of new kinds of properties (by subclassing Property), provides good separation of concerns (each property is properly modularized and encapsulated in a class) and disciplines property composition, since the interaction among properties is localized in the mediator. Nevertheless the pattern introduces object schizophrenia: a Portalware agent is explicitly broken into four objects (one object for the agent's core and three objects for each of the basic agency properties), each of which has its own object identity. To create an agent, four objects must be explicitly created and initialized, according to the pattern. Furthermore, the interaction relationships among properties are not explicit at the design level, only inside method definitions. Notice that the names of advices used in Figure 6, Section 4.3, are used for methods defined in the interface of each property subclass. These methods are explicitly called from methods defined in the Agent class interface (more precisely, from methods with the same name of the pointcut).

5.2. Alternative Agency Properties

Specific agency properties such as the ones described in Section 4.4 also are represented as colleagues in the Mediator pattern. New properties are added by subclassing Property, although additional expressive means are required at the structural view (constraints in UML, for example) to assert that the reference to the mediator (Agent class) will contain only the specific type of agent (InformationAgent, UserAgent or InterfaceAgent class) to which the new property should be associated. Moreover, the addition or removal of a new property requires invasive modification of the corresponding type of agent, to add or remove a link to the new property.

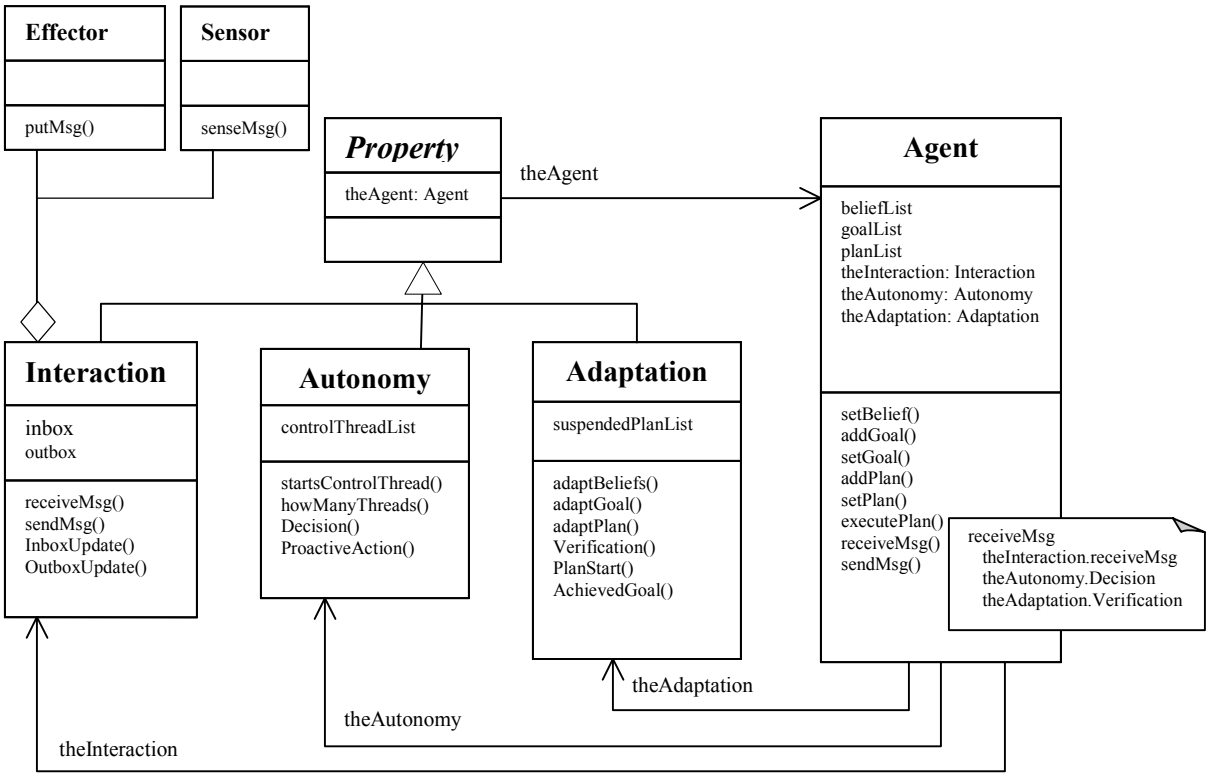

Figure 9: Using the Mediator design pattern to model Agency Properties

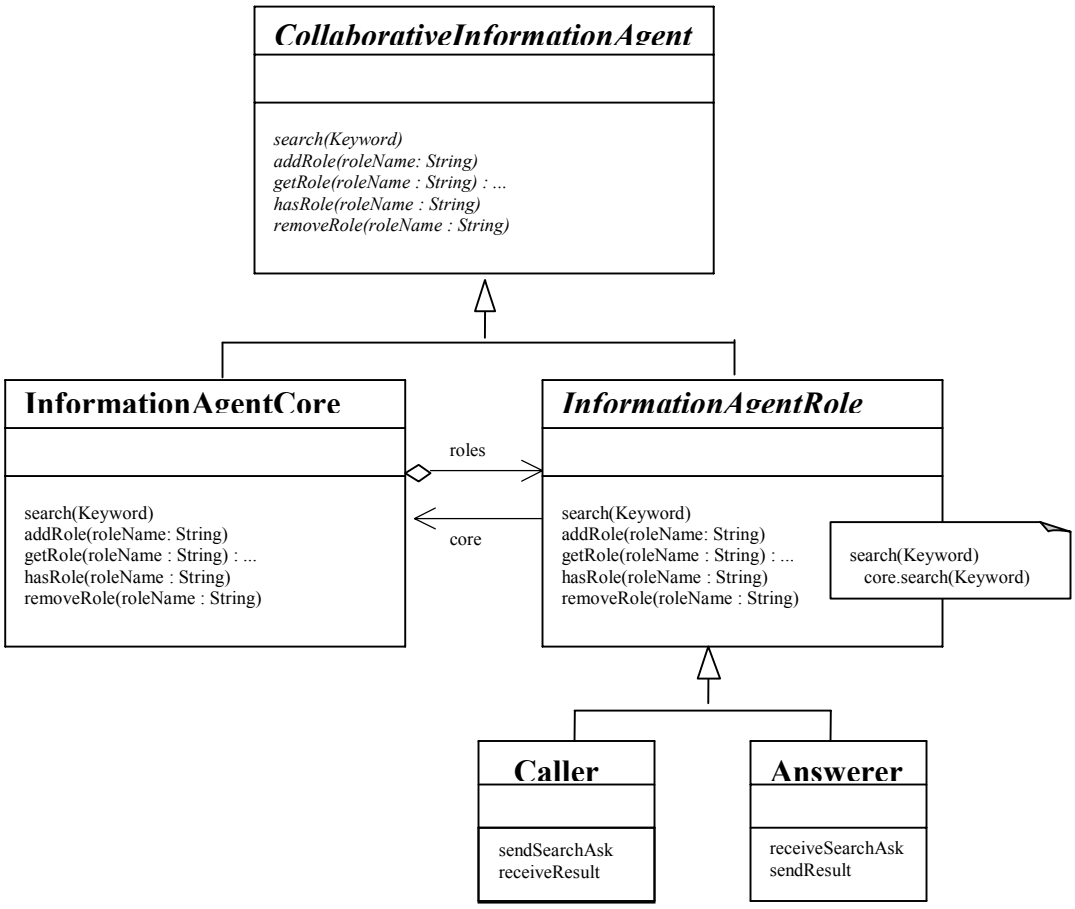

Figure 10: Using the Role Object pattern to model Agent Roles

An alternative solution is the use of inheritance to implement a new kind of mediator that incorporates the new property (subclassing Agent, the mediator, to create CollaborativeAgent, for example). Unfortunately, in our case study, subclassing the mediator does not work properly, since the Agent class already has subclasses (the agent types).

5.3. Roles

As stated before, we want to isolate and encapsulate each role an agent may play and to be able to compose multiple roles with an agent's core state and behavior under the context of specific collaborations in such a way that promotes easy and independent agent evolution (addition and removal of roles). The **Role Object** design pattern [2] is a suitable design choice since it lets us vary how objects *behave in a certain context*, by allowing the dynamic attachment and detachment of role objects from the agent's core state and behavior**.** The resulting object aggregate represents one logical object, even though it consists of several physically distinct objects. The Role Object pattern avoids the combinatorial explosion of classes, as it would result from using multiple inheritance to compose the different roles in a single class [2]. Nevertheless, clients of the Agent class are likely to become more complex, since working with an object through one of its role interfaces implies slight coding overhead compared to using the interface provided by the Agent class interface itself. For example, caller and answerer roles must be explicitly created and added to information agent objects, and the client has to check whether the object plays the desired role before explicitly activating some capability introduced by it (Figure 10).

5.4. Composition of Agency Concerns

The need for capturing the interactive and overlapping characteristics of the multiple agency aspects also is an important issue in our pattern-based solution. The interaction among agency properties is captured in the methods that comprise the interface of the mediator (Agent class). For example, the method receiveMsg() defined in the Agent class serves as a means to describe the protocol of agent's message reception, involving the Interaction, Autonomy and Adaptation properties. The modification of this protocol, including the adaptation of the precedence relationship among properties or the inclusion of new properties, may require invasive change to the original class, or some spurious use of inheritance to refine the protocol behavior. We also use inheritance to capture the overlapping nature between the Interaction and the Collaboration properties. The Collaboration class is defined as a subclass of Interaction.

5.5. Agent Evolution

In general, the use of design patterns requires preplanning for suitable support for evolution without invasive changes. As a consequence, many classes may be created just to deal with this demand (the class explosion problem). Nevertheless, some invasive changes may still be necessary. For example, consider again the scenario described in Section 4.7, where a collaborative information agent is required to move across a network to find some piece of information, instead of collaborating with other agents. In our pattern-based solution, this change requires at least the following actions: (i) the removal of a link from the InformationAgent class to the Collaboration property; (ii) the inclusion of the Mobility property in the design (subclassing Property); (iii) the definition of an association link from the InformationAgent class to this new property; and (iv) the modification of the code excerpt where an explicit call is made to the method startsCaller, replacing it with another explicit call to a method that starts the process of mobility. Unfortunately, except for (ii), all these changes are invasive.

6. Results and Discussion

The benefits of the proposed aspect-based method seem to be very appealing regarding ease of construction, evolution and reuse in multi-agent system development. Nevertheless, a realistic and systematic assessment should be conducted in order to validate the proposed ideas and demonstrate their usefulness and benefits in terms of some qualitative and quantitative criteria [1].

Hence, we have developed a comparative case study to assess and evaluate the potential benefits and possible problems of applying our aspect-oriented method and our pattern-oriented method to the design and implementation of Portalware. The main purpose of our case study then is to characterize, evaluate and compare our proposed methods built from the perspective of MAS developers in a single project scope. The case study was structured into three phases, performed by two different teams in parallel (each team using one method), both regarding: (1) initial system construction, (2) subsequent modification due to new requirements (e.g., information agents become mobile), and (3) reuse of existing features in new contexts (e.g., the Collaboration aspect may be reused for user agents). In phase 1, the two teams designed and implemented object-oriented and aspect-based solutions for Portalware. In phases 2 and 3, both teams evolved the Portalware design, by modifying and reusing agency properties and roles, according to the same requirements. The measurement process considered qualitative and quantitative criteria, regarding writability, readability, maintainability and reusability as the main key qualities for the designs at hand. The comparative case study led to interesting results and insights concerning the overall benefits and usefulness of our proposed aspect-based method. Nevertheless, as a preliminary evaluation from the collected results, we have noticed that:

The aspect-oriented method supports better writability. The use of design patterns with its demand on preplanning for change, requires the definition of several classes and methods with only trivial structure and behavior, e.g., abstract classes and explicit forwarding of messages to other objects or methods. This may lead to significant overhead for the software developer in terms of writability and also decreased understandability of the resulting code. With our aspect-based approach we write less code and furthermore, we are able (i) to isolate and encapsulate concerns more appropriately, and (ii) to compose them with little effort.

The aspect-oriented method supports better reuse. Design patterns have no first-class representation at the implementation level. The implementation of a design pattern, therefore, cannot be reused and, although its design is reused, the software developer is forced to implement the pattern many times. Unlike patterns, recent AOD approaches provide first-class representation at the implementation-level for design-level aspects and crosscutting composition mechanisms, supporting reuse both at the design and implementation levels. Moreover, our aspect-based approach supports better reuse as a side effect of AOP crosscutting mechanism, that defines implicit behavior composition at well-defined join points. For example, reusing the Collaboration property in the

context of user agents requires the association of the Collaboration aspect to UserAgent class, depicting the join points of interest, while in our pattern-based approach, some additional modifications are required to introduce the association as well as the explicit calls to methods defined in the interface of the Collaboration class.

The aspect-oriented method supports better evolution. During the evolution phase, the introduction of the Mobility property was much simpler in the aspect-based design than in the pattern-based design with the use of Mediator (Section 5.1). Furthermore, *design for change*, as prescribed by design patterns, sometimes imposes a not so simple structure to provide the required flexibility for evolution. Last but not least, combining several design patterns in the same project is not a trivial task.

The aspect-oriented method supports better expressiveness. The aspect-based approach can be extremely useful, especially on larger projects, to express requirements, relationships or contracts involving agency properties that need to be maintained, or at least kept in mind. For example, the specification of the agent's expected behavior after message reception (receiveMsg) remains explicitly documented at the structural view of the design.

The aspect-oriented method supports better flexibility to accommodate distinct definitions for agenthood. Although we have presented a definition for agenthood (Section 2.1) that tries to identify the common features of software agents, this definition is not widely accepted and varies from researcher to researcher. This variation requires an agent model that is flexible enough to encompass disciplined composition of aspects of agents. Fortunately, our aspect-based approach can accommodate distinct definitions since agency aspects can be easily attached to and removed from the Agent class.

7 Comparison with Related Work

Some attempts to deal with agent complexity by using the object model have been proposed in the literature [19, 20]. Kendall et al [20] proposes the layered agent architectural pattern, which separates different layers of an agent, such as sensory layer, action layer and so on. However, some aspects of agents, such as autonomy, cut across the different layers of this approach. We also believe that the evolution of this kind of design is cumbersome since removing any of these layers is not a trivial matter; it requires the reconfiguration of the adjacent layers. The aforementioned work does not present guidelines for evolving agent behavior in order to accommodate new aspects of agents or remove existing ones. In fact, modeling the agency properties of an agent within the traditional object model is hard to do and introduces substantial limitations. In contrast, our model allows the addition or removal of aspects of agents transparently (Section 4.7).

Moreover, in our experience on using design patterns for the agent domain, we have detected a number of problems: (i) class explosion, (ii) need for preplanning, (iii) difficulty in the application and combination of suitable design patterns, (iv) lack of expressive power, and (v) object schizophrenia.

To implement an agent's role aspects, we have followed Kendall et al's [19] guidelines for the application of aspect-oriented programming to implement role models. However, their work does not deal with agents' agency properties, which we believe are the main source of agent complexity. Our proposal builds on (enriches) their approach and presents a unified framework for dealing with roles as well as agency properties and their interrelationships.

Research in aspect-oriented software engineering has concentrated on the implementation phase, although some work has presented aspect-oriented design solutions. To date, aspect-oriented programming has been used mainly to implement generic aspects such as persistence, error detection/handling, logging, tracing, caching, and synchronization. However, these approaches are generally concerned with only one of these generic aspects. In this work, we provide an aspect-based design model which: (i) handles both agency-specific aspects as well as generic aspects (e.g. synchronization and persistence); and (ii) encompasses a number of different aspects and their relationships.

8 Conclusions

As the world moves rapidly toward the deployment of geographically and organizationally diverse computing systems, the technical difficulties associated with distributed, heterogeneous computing applications are becoming more apparent and placing new demands on software structuring techniques. The notion of agents is becoming increasingly popular in addressing these difficulties. However, MAS development is not a trivial task. This work discussed the problems in dealing with agency concerns as well as presented two software engineering methods to address these problems. The success or otherwise of a structured method for MAS development depends on its ability for (i) separating agency concerns, and (ii) minimizing the conceptual gaps between high-level agent models and object-oriented designs.

First, we presented an aspect-oriented method for MAS development. This method uses aspects to encapsulate the agency concerns in the design phase and minimize misalignments with the specification phase. Second, we proposed a pattern-oriented method that realizes a MAS as a set of design patterns. The use of this method promotes objects as potentially reusable components because they are independent encapsulations of agency concerns. Since objects are associated with agency concerns, there often is a clear mapping between high-level agent models and their respective objects identified in the subsequent phase, thus reducing potential misalignments and producing likely reusable and maintainable software systems. So we presented some findings by comparing these methods. The goal of this comparison was twofold: (i) to verify which method is likely to produce MAS with a lower degree of misalignments; and (ii) assess the quality of the produced designs in terms of reusability, maintainability and understandability. We have concluded that the composition mechanisms of AOD provide improved support for dealing with the complexity of agency concerns and producing MAS which is easier to understand, maintain and reuse.

Acknowledgments. This work has been partially supported by CNPq under grant No. 141457/2000-7 for Alessandro and grant No. 140646/2000-0 for Viviane, and by FAPERJ under grant No. E-26/150.699/2002 for Alessandro. Alessandro, Viviane, Christina and Carlos also are supported by the PRONEX Project under grant 7697102900. We would also like to thank Arndt von Staa for the good suggestions during this work.

References

1. V. Basili et al. Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, SE-12(7), July 1986.
2. D. Bäumer, et al. The Role Object. In *Proc. of the 1997 Conference on Pattern Languages of Programs* (PLoP '97), 1997.
3. J. Bigus, J. Bigus. Constructing Intelligent Agents with Java – A Programmer's Guide to Smarter Applications. Wiley, 1998.
4. G. Booch, J. Rumbaugh. Unified Modeling Language – User Guide. Addison-Wesley, 1999.
5. J. Bradshaw et al. KaoS: Toward an Industrial-Strength Generic Agent Architecture. In *Software Agents*, J. Bradshaw (ed.), Cambridge, MA: AAAI/MIT Press, 1996.
6. J. Bradshaw. An Introduction to Software Agents. In *Software Agents*, J. Bradshaw (ed.), American Association for Artificial Intelligence/MIT Press, 1997.
7. D. Brugali, K. Sycara. A Model for Reusable Agent Systems. In *Implementing Application Frameworks – Object-Oriented Frameworks at Work*, M. Fayad et al. (editors), John Wiley & Sons, 1999.
8. C. Chavez, C. Lucena. Design-level Support for Aspect-oriented Software Development. In *Proc. of the Workshop on Advanced Separation of Concerns in Object-oriented Systems at OOPSLA'2001*, USA, 2001.
9. M. Elammari, W. Lalonde. An Agent-Oriented Methodology: High-Level and Intermediate Models. In *Proc. of AOIS 1999*, Heidelberg (Germany), June 1999.
10. E. Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.
11. A. Garcia, M. Cortés, C. Lucena. A Web Environment for the Development and Maintenance of E-Commerce Portals Based on a Groupware Approach. In *Proc. of the 2001 Information Resources Management Association International Conference (IRMA 2001)*, Toronto, May 2001.
12. A. Garcia et al. An Aspect-Based Approach for Developing Multi-Agent Object-Oriented Systems. In *Proc. of the XXI Brazilian Symposium on Software Engineering*, Rio de Janeiro, Brazil, October 2001, pp. 177-192.
13. A. Garcia et al. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. *Journal of Systems and Software*, Elsevier, 2(59): 197-222, November 2001.
14. A. Garcia, C. Lucena. An Aspect-Based Object-Oriented Model for Multi-Agent Systems. In *Proc. of the 2th Advanced Separation of Concerns Workshop at ICSE'01*, Toronto, Canada, May 2001.
15. A. Garcia, C. Lucena, D. Cowan. Agents in Object-Oriented Software Engineering. *Software: Practice and Experience*, Elsevier, 2003. (Accepted to Appear).
16. A. Garcia, C. Rubira. A Unified Meta-Level Software Architecture for Sequential and Concurrent Exception Handling. *The Computer Journal,* 6(44):569-587 January 2002.
17. W. Harrison and J. Ossher. Subject-Oriented Programming: A Critique of Pure Objects. In *Proc. of OOPSLA'93*, ACM, pp. 411-428, 1993.

18. N. Jennings, K. Sycara, M. Wooldridge. A Roadmap of Agent Research and Development. *International Journal of Autonomous Agents and Multi-Agent Systems* 1(1) 7-38, 1998.

19. E. Kendall. Agent Roles and Aspects. In *Proc. of Workshop on Aspect-Oriented Programming, ECOOP'98* , July 1998.

20. E. Kendall, P. Krishna, C. Pathak and C. Suresh. A Framework for Agent Systems. In *Implementing Application Frameworks – Object-Oriented Frameworks at Work*, M. Fayad et al. (eds.), John Wiley & Sons, 1999.

21. G. Kiczales et al. Aspect-Oriented Programming. In *Proc. of ECOOP'97 Conference on Object-Oriented Programming, LNCS*, (1241), Springer-Verlag, Finland., June 1997.

22. G. Kiczales et al. An Overview of AspectJ. In *Proc. of ECOOP'01 Conference on Object-Oriented Programming*, Budapest, Hungary, 2001.

23. D. Lange, M. Oshima. Programming and Developing Java Mobile Agents with Aglets. Addison-Wesley, 1998.

24. T. Nelson, D. Cowan, P. Alencar. A Model for Describing Object-Oriented Systems from Multiple Perspectives. *Lecture Notes on Computer Science,* (1783):237-248, Springer-Verlag, 2000.

25. H. Nwana. Software Agents: An Overview. *Knowledge Engineering Review*, 11(3):1-40, 1996.

26. Object Management Group – Agent Platform Special Interest Group. Agent Technology – Green Paper. Version 1.0, September 2000.

27. P. Ripper, M. Fontoura, C. Lucena. V-Market: A Framework for e-Commerce Agent Systems. *World Wide Web*, Baltzer Science Publishers, 3(1), 2000.

28. Y. Shoham. Agent-Oriented Programming. *Artificial Intelligence*, (60): 24-29, 1993.

29. SoC+Agents Group. Separation of Concerns and Multi-Agent Systems. URL: www.teccomm.les.inf.puc-rio.br/SoCAgents, 2001.

30. P. Tarr et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. of the 21st International Conference on Software Engineering (ICSE'99)*, May 1999.

31. M. Wooldridge, N. Jennings, D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *International Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), 2000, pp. 285 – 312.