# Developing Multi-Agent Software: an Aspect-Based Approach and a Pattern-Based Approach

Alessandro F. Garcia
e-mail: afgarcia@inf.puc-rio.br

Christina Chavez
e-mail: flach@inf.puc-rio.br

Viviane T. da Silva
e-mail: viviane@inf.puc-rio.br

Carlos J. P. Lucena
e-mail: lucena@inf.puc-rio.br

PUC-RioInf.MCC40/01 November, 2001

## Abstract

This paper proposes and compares two software engineering approaches for multi-agent systems. The first proposal explores the benefits of aspect-based design and programming and the second one is based on design patterns. Both approaches have the following goals: (i) describe structured integration of agents into the object model, (ii) incorporate flexible facilities to build different types of software agents, (iii) encourage the separate handling of each property and capability of an agent, (iv) provide explicit support for disciplined and transparent composition of agency properties and capabilities in complex software agents, and (v) allow the production of agent-based software so that it is easy to understand, maintain and reuse. We also demonstrate our proposals through the Portalware system, a web-based environment for the development of e-commerce portals.

**Keywords:** Multi-agent systems, aspect-oriented design and programming, design patterns, comparative study.

## Resumo

Este artigo propõe e compara duas abordagens para o desenvolvimento de sistemas de software multi-agente. A primeira proposta explora os benefícios de projeto e programação orientada a aspectos, enquanto a segunda é baseada em padrões de projeto. Ambas abordagens tem os seguintes objetivos: (i) descrever a integração estruturada de agentes no modelo de objetos, (ii) possibilitar a construção de diferentes tipos de agentes de software, (iii) encorajar o tratamento separado de cada propriedade e capacidade de um agente, (iv) fornecer suporte explícito para a composição disciplinada e transparente de propriedades e capacidades de agentes de software complexos, e (v) permitir a produção de software baseado em agentes que seja fácil de entender, manter e reutilizar. Nós também demonstramos nossas propostas através do sistema Portalware, um ambiente web para o desenvolvimento de portais.

**Palavras-chave:** Sistemas multi-agentes, projeto e programação orientada a aspectos, padrões de projeto, estudo comparativo.

## 1 Introduction

The effort and cost of designing and implementing multi-agent software while satisfying quality requirements, such as maintainability and reusability, are still deep concerns to object-oriented software engineers. The design and implementation of a single agent is very complex. Like objects, software agents include a specific set of services (capabilities) for their users. Even though objects and agents have many common concerns [5, 23], agents are more complex software entities since they encompass additional concerns. The state of agents is driven by beliefs, goals, capabilities, plans, and their behavior is composed of a number of agency properties such as autonomy, adaptation, interaction, learning, mobility, and collaboration. Moreover, collaborative software agents play different roles to cooperate with other agents in heterogeneous contexts. In practice, a complex application is composed of multiple types of agents, each of them having distinct agency concerns, i.e. different states, agency properties and roles. Agents pose other design and implementation challenges because many agency concerns overlap and interact with each other, and a disciplined approach is required for composition. As a consequence, there is a need for a software engineering approach from an early stage of design that encourages the separate handling of each agent state component and behavioral property as well as provides explicit support for disciplined composition of complex software agents. Ideally, this approach should incorporate flexible facilities to build different types of software agents, and allow the production of agent-based software that is easy to understand, maintain and reuse.

However, existing object-oriented proposals often focus on the implementation phase, and do not provide direct support for handling and reusing agency concerns separately [3, 7, 20]. Moreover, the current proposals generally support a limited number of agent types, and the state and behavior of an agent are often encapsulated as an object. Even though it is desirable for an agent to appear as a single object, this scheme results in agent design and implementation being quite poor, complex and difficult to understand, maintain and reuse in practice. In fact, it is not often easy to design software agents properly, as the developers of multi-agent systems have to take into account many agency concerns at the same time. In addition, the lack of support for dealing with the interactive and overlapping nature of agency concerns limits the understanding, maintainability and reusability of multi-agent applications. Ideally, agent system developers should apply special structuring techniques and disciplined ways of associating the different properties and roles of an agent with its core state and behavior.

In this context, the goals of this paper are: (i) proposing two approaches for designing and implementing agent-based object-oriented systems, and (ii) comparing these approaches in terms of understandability, reusability, and maintainability. The first approach [11] explores the benefits of aspect-based design and implementation for mastering the increasing complexity of integrating software agents into the object model. Aspect-oriented design encourages modular descriptions of software systems by providing support for cleanly separating the object's core functionality from its concerns. Aspect is the abstraction that modularizes a concern and is associated with one or more objects. The aspect-based approach explores this abstraction to support the construction of multi-agent object-oriented software with improved structuring for design reuse and evolution. Each agent is represented by a single object and a set of aspects that modularize its overlapping and interactive concerns. We also present some results gathered when applying our approach to introduce multiple software agents in Portalware [12], a web-based environment for the development of e-commerce portals. The second approach uses design patterns [10] in order to deal with the complexity associated with multi-agent object-oriented software development. Patterns provide good design solutions organized in terms of a set of interrelated objects, aiming to produce object-oriented software with improved reusability and maintainability. Our pattern-based approach applies different patterns to address some design problems in the agent domain.

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 an example which is used throughout this paper to illustrate our approach. Section 3 overviews two software engineering approaches: pattern-based object-oriented design and programming and aspect-based object-oriented design and programming. Section 4 presents our aspect-based approach for designing agent-based applications, and applies it to the Portalware system. Section 5 presents the pattern-based approach and compares it with the aspect-based approach, assessing the relative advantages and disadvantages of applying our proposal. Section 6 discusses related work. Finally, Section 7 presents some concluding remarks and directions for future work.

## 2. Multi-Agent Systems: Definitions and Case Study

### 2.1. Software Agents and Agency Aspects

*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 in an application, but often forming an organization with other agents; this organization is called a *multi-agent application*. A multi-agent application generally has several types of software agents [22], such as *information agents*, *user agents*, and *interface agents*. Each agent type typically has different agency concerns. We can classify the agency concerns into three types: (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* [23, 26]. 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 agent's beliefs. In this way, the behavior of agents is driven by the execution of their plans that select appropriate capabilities in order to achieve the stated goals. There are different kinds of plans, and they are application-specific [16]. Plans are divided into three categories: (i) *reaction plans*, (ii) *decision plans*, and (iii) *collaborative plans*. Each of them 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 [16, 22, 23] and our experience in developing multi-agent applications [12, 25, 27, 29]. In general, autonomy, interaction and adaptation are considered as fundamental properties of software agents, while learning, mobility and collaboration are neither a necessary nor sufficient condition for *agenthood* [23] (Figure 1). 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 [16]. Since agents are autonomous software entities, the agent itself starts its control thread and decides whether to accept or reject incoming messages. If a message is accepted, the agent may have to adapt its state. The adaptation 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. During the execution of plans, software agents alternatively: (i) extend or refine their knowledge when interacting with their environment (learning), (ii) move themselves from one environment in a network to another (mobility), and (iii) join a conversation channel with other agents (collaboration).

| 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 components 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

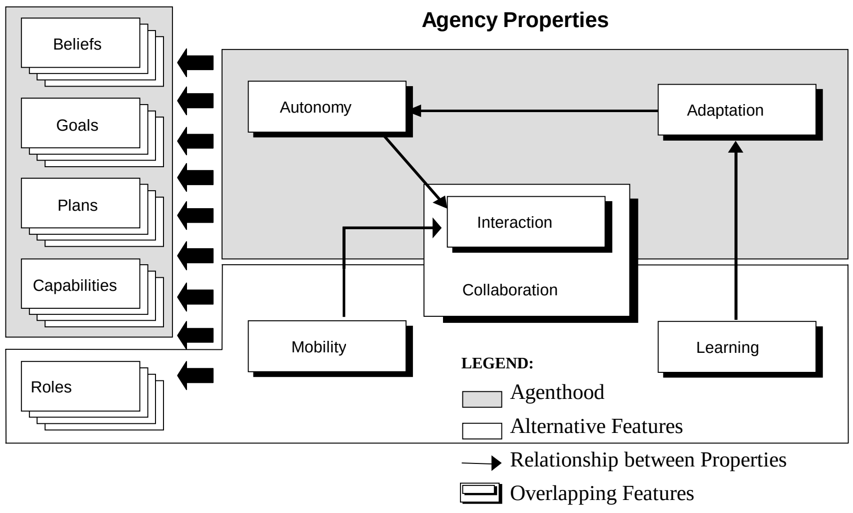


**Figure 1:** A Definition for Agenthood

**Roles.** A collaborative agent plays a role to cooperate with another agent. Roles are application-dependent and are specific for each context. So, since software agents can cooperate while pursuing their goals in different situations, a cooperating agent may include different roles in order to work together in multiple contexts. In order to perform a cooperation, a collaborative plan is instantiated, and it chooses the eligible roles.

**Interacting and Overlapping Properties.** By the very nature of agency properties, these properties are not ortoghonal – they interact with each other (Figure 1). For instance, adaptation depends on autonomy since it is necessary to adapt the agent's state (beliefs and goals) and behavior (plans) when the autonomy property decides to accept an incoming message. In addition, two agency properties are overlapping: interaction and collaboration. Collaboration is viewed as a more sophisticated kind of interaction, since the former comprises communication and coordination. Interaction is only concerned with communication, i.e. sending and receiving messages. During a collaboration, messages are also received from and sent to the participating agents. However, the collaboration property additionally defines how to collaborate, i.e. it addresses the coordination protocol. A simple coordination protocol consists of synchronizing the agent that is waiting for a response to the agent that will send a response.

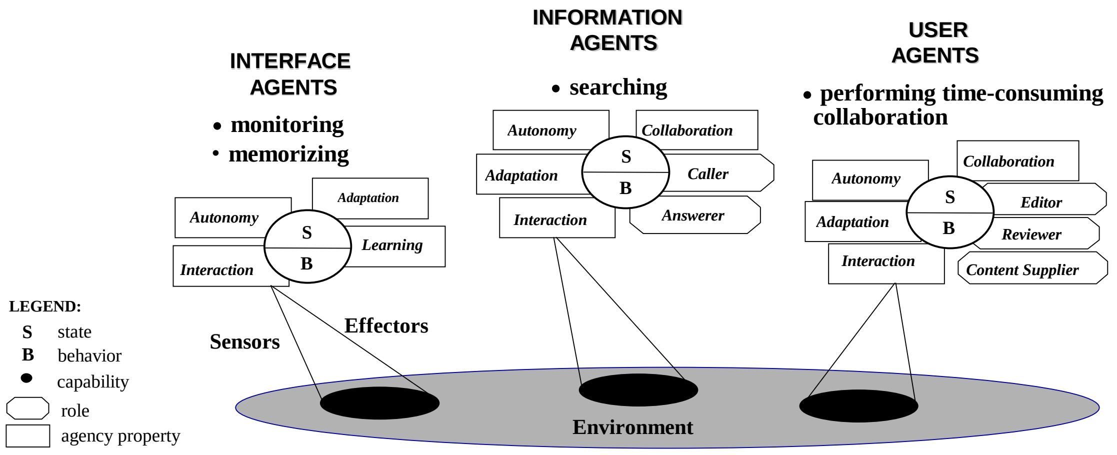


**Figure 2:** Portalware Agents and their Agency Concerns

### 2.2. Software Agents in Portalware: A Case Study

Figure 2 illustrates the software agents in Portalware [12], 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 2 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 [11].

Portalware users often need to search for information, which is stored into 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 other 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 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 caller and answerer roles since they can perform these different roles in distinct situations.

## 3 Software Engineering for Agent Systems

The inherent complexity in the organization and introduction of software agents into object-oriented applications requires the use of appropriate software engineering principles. Modularity and separation of concerns are two complementary well-established principles in software engineering, which use high-level abstractions to hide complexity by decomposing a software system into *modules* and *concerns*, respectively [28]. The importance of these principles_increases as new technologies are introduced and software applications (such as multi-agent applications) become more complex. From the viewpoint of modular decompositions, complex problems can be divided into smaller parts (abstractions), such as: (i) data, (ii) functions, (iii) objects, and (iv) agents. The common feature of these abstractions is that the decomposed parts are disjoint [21]. From the viewpoint of concern decompositions, complex problems can be divided into different abstractions, such as (i) aspects [18] and (ii) subjects [14]. What distinguishes this concern decomposition from the module decomposition is the fact that the decomposed parts are not disjoint. 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 [21]. In other words, concerns naturally cut across application modules.

Object-oriented software engineering approaches focus on modular decomposition. In this paper, we present an approach that explores the benefits of both modular and concern decompositions to deal with the complexity of integrating software agents in the object model. Our proposal incorporates the recent advances in separation of concerns techniques [18, 28], in particular, those provided by *aspect-oriented design and programming* [8,18]. To be able to evaluate our proposal against others, we propose a *pattern-based object-oriented software engineering* approach for agent systems. In the remainder of this section, we describe some characteristics of two development techniques, that is, pattern-based object-oriented design and programming (design patterns) and aspect-oriented design and programming (AOP).

### 3.1. Design Patterns and Module Decomposition

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 object-oriented paradigm are extremely useful and patterns offer a wide range of flexible ways to combine classes and objects, yet they are inherently unable to modularize all concerns of interest in software engineering, mainly because some of them naturally cut across modules. Furthermore, *design for change*, as prescribed by design patterns, sometimes imposes a not so simple structure to provide the required flexibility. Last but not least, combining several design patterns in the same project is not a trivial task.

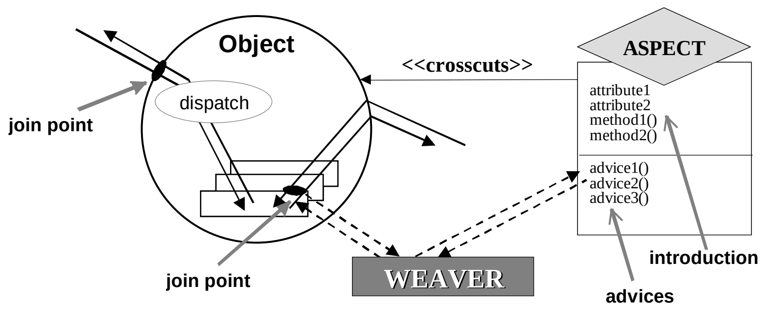


**Figure 3:** AspectJ Mechanisms for Dealing with Crosscutting Aspects.

## 3.2. Aspect-Oriented Design and Programming

Aspect-oriented design and programming has been proposed as a technique for improving separation of concerns in software design and implementation. The central idea is that while hierarchical modularity mechanisms of object-oriented design and implementation languages are extremely useful, they are inherently unable to modularize all concerns of interest in complex systems. Thus, the goal of aspect-oriented design and programming [8, 18] 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 properties that *crosscut* (i.e., cut across) components in system's design and implementation. Separating aspects 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 3). Examples of join points are method calls, method executions, and field sets and reads.

AspectJ [19] is a practical aspect-oriented extension to the Java programming language [13]. *Aspects* are modular units of crosscutting implementation 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 advice: (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 ordinary Java member declarations to the object to which the aspect is attached (such as, attributes and methods). *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 3). Up to the current version of AspectJ, almost all of the weaving process is realized as a pre-processing step at compile-time [19].

# 4 An Aspect-Based Approach for Multi-Agent OO Systems

In this section, our multi-agent approach is presented as an aspect-oriented extension of the traditional object model. In particular, our proposal is discussed in terms of: (i) *agent's core state and behavior*, (ii) *agent types*, (iii) *agency aspects for agenthood*, (iv) *particular agency aspects*, (v) *role aspects*, (vi) *aspect composition*, and (vii) *agent evolution*. We adopt UML diagrams [4] as the modeling language throughout this paper. The design notation for aspects is based on [17]: 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 one of *before*, *after* or *around*.

## 4.1. Agent State

In our approach, classes represent agents as well as their beliefs, goals and plans. The Agent class specifies the core state and behavior of an agent (Figure 4), and should be instantiated in order to create application's agents.

Methods defined in the interface of the Agent class are used to query and update its state and to implement 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 different plans, and hence a Goal object may have more than one associated Plan object.

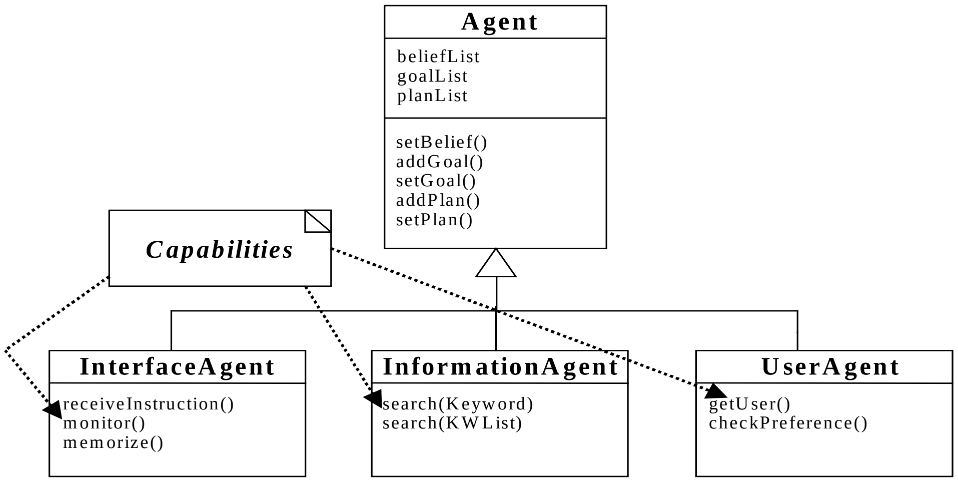


**Figure 4:** Agent State and Agent Types.

## 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 4 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 Aspects for 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 5 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 5), 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.

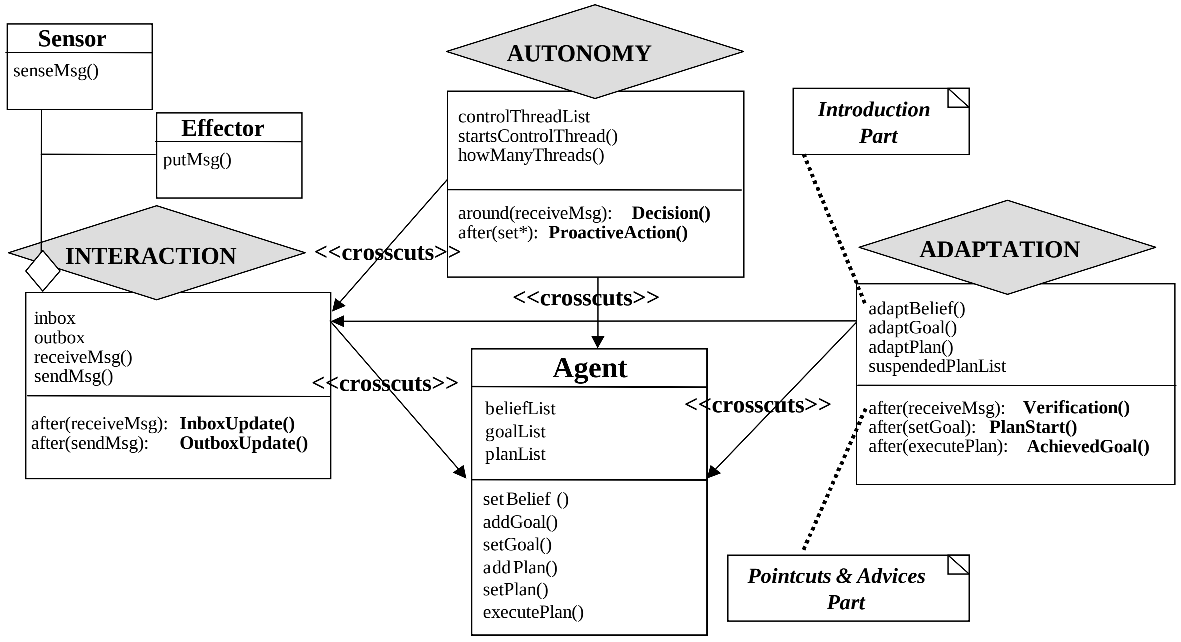


**Figure 5:** Agency Aspects and the Design for Agenthood.

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, it adapts 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. Particular Agency Aspects

The agency aspects that are specific to each agent type are associated with the corresponding subclasses (Figure 6). 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. Role Aspects

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.

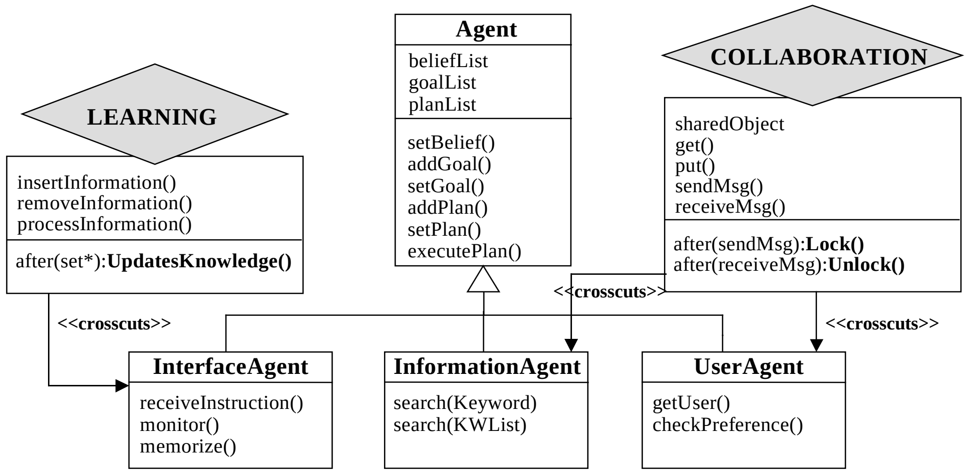


**Figure 6:** Particular Agency Aspects.

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 7 illustrates this situation for the information agents of Portalware (Section 2.2). An information agent needs to support the calling and answering 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.

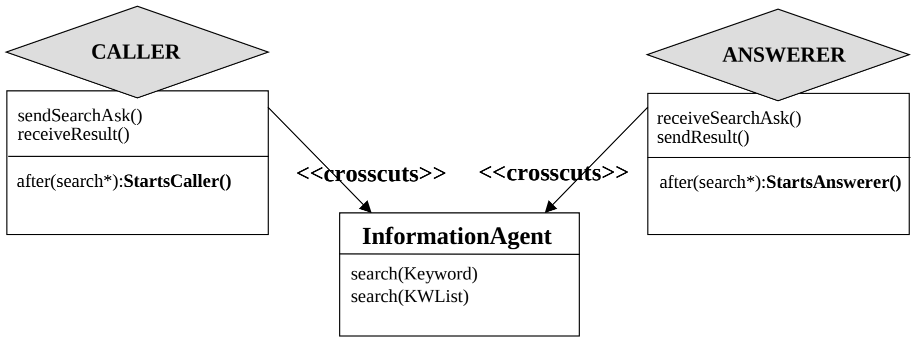


**Figure 7:** Agent Roles.

### 4.6. Aspect Composition

Our approach establishes relationship patterns which provide 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 kept . 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 other host in order to start a new search for the required information.

## 5 Comparison with a Pattern-Based Approach for Multi-Agent OO Systems

The benefits of the proposed aspect-based model seem to be very appealing regarding the easy 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]. Furthermore, our aspect-based approach must be compared to other well-known approaches used for constructing high-quality software, such as object-oriented design with design patterns [10], under the requirements and constraints of the agent domain. Design patterns may offer solutions that structure and discipline the composition of separated agency concerns, ensuring that the system can only change or evolve in specific, predictable ways.

Hence, we have developed a comparative case study to assess and evaluate the potential benefits and possible problems of applying aspect-oriented techniques and advanced object-oriented techniques (based on design patterns) to the design and implementation of Portalware. The main purpose of our case study is then to characterize, evaluate and compare two design models built from the perspective of multi-agent system developers in a single project scope such as Portalware. The case study was structured into three phases, performed by two different teams in parallel (each team using one technique), 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). On phase 1, the two teams designed and implemented object-oriented and aspect-based solutions for Portalware. On 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.

## 5.1. The Pattern-Based Approach

In this section, we focus on the description of agency aspects and role aspects using design patterns, as well as on issues regarding aspect composition and agent evolution. Agent state and agent types follow the same design decisions presented in Sections 4.1 and 4.2.

**Agency Aspects for 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 8). 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.

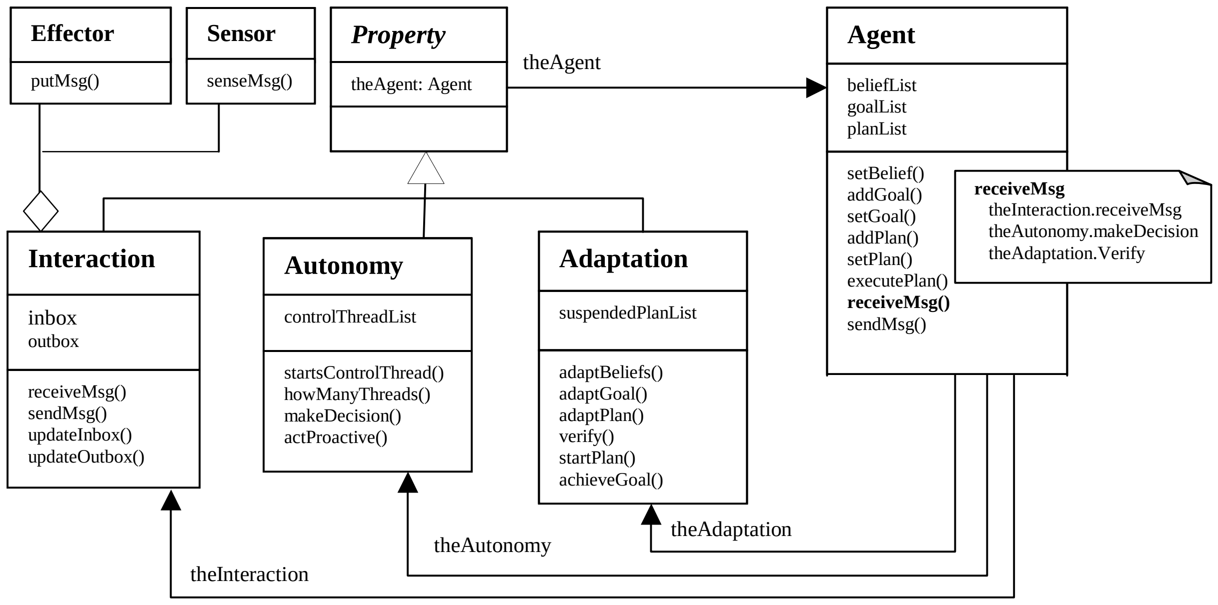


**Figure 8**: Using the Mediator design pattern to model Agency Properties

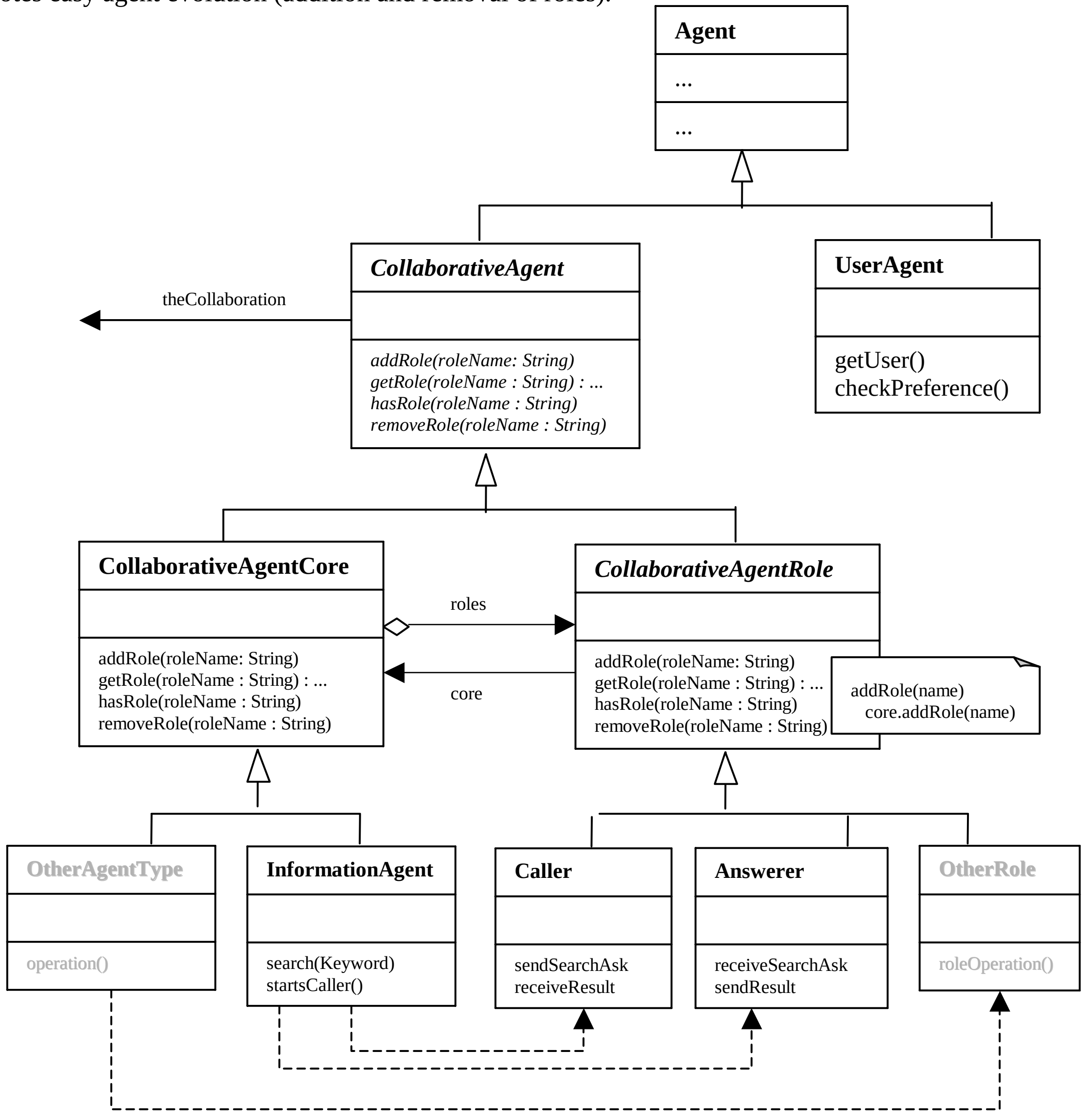
In our solution, the mediator interface subsumes agent's core state and behavior as well as the interaction protocol among agency properties. Agency properties are encapsulated as classes, and play the role of colleagues in the pattern, i.e., they interact but do not refer to each other directly, only 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 5 (Section 4.3), are used for methods defined in the interface of each property subclass, properly rewritten to express actions (for example, makeDecision() instead of Decision()). These methods are explicitly called from methods defined in the Agent class interface (more precisely, from methods with the same name of the pointcut, e.g., receiveMsg() in Figure 8).

**Particular Agency Aspects.** Specific agency properties such as the ones described in Section 4.4 are also 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.

An alternative solution is the use of inheritance to implement a new kind of mediator that incorporates properties (by specializing Agent to create CollaborativeAgent, and defining a new association from it to the Collaboration property, for example). Unfortunately, in our case study, subclassing the mediator is not enough, since the Agent class has already subclasses (the agent types) and the inheritance hierarchy must be adapted to deal with the necessary changes.

**Role Aspects.** As stated before, we want to isolate and encapsulate each role an agent may play and to be able to compose multiple roles with agent's state and behavior under the context of specific collaborations, in such a way that promotes easy agent evolution (addition and removal of roles).



Agent
...
...
CollaborativeAgent
theCollaboration
addRole(roleName: String)
getRole(roleName : String) : ...
hasRole(roleName : String)
removeRole(roleName : String)
UserAgent
getUser()
checkPreference()
CollaborativeAgentCore
addRole(roleName: String)
getRole(roleName : String) : ...
hasRole(roleName : String)
removeRole(roleName : String)
roles
core
CollaborativeAgentRole
addRole(roleName: String)
getRole(roleName : String) : ...
hasRole(roleName : String)
removeRole(roleName : String)
addRole(name)
core.addRole(name)
OtherAgentType
operation()
InformationAgent
search(Keyword)
startsCaller()
Caller
sendSearchAsk
receiveResult
Answerer
receiveSearchAsk
sendResult
OtherRole
roleOperation()

**Figure 9**: Using Role Object pattern to model Agent 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**.** Following the pattern, the CollaborativeAgent class defines a protocol for managing roles (Figure 9). The CollaborativeAgentCore subclass implements the CollaborativeAgent interface and manages its role objects. The InformationAgent class is now defined as a specialization of the CollaborativeAgentCore class. The CollaborativeAgentRole class implements the CollaborativeAgent interface by forwarding requests to its core attribute. CollaborativeAgentRole is the common superclass for the concrete roles, the Caller and Answerer classes. When an information agent needs to perform a role, the corresponding role class is instantiated and added to the roles dictionary. The resulting object aggregate represents one logical object, even though it consists of several physically distinct objects. Other agent types (e.g. user agents) may become collaborative by (i) specializing the CollaborativeAgentCore class and (ii) introducing their respective roles as subclasses of the CollaborativeAgentRole class.

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 get more complex, since working with an object through one of its role interfaces implies 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.

**Aspect Composition.** The need for capturing the interactive and overlapping characteristics of the multiple agency aspects is also an important issue in our pattern-based approach. 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.

**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, (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.

## 5.2. Results and Discussion

The comparative case study led to interesting results and insights concerning the overall benefits and usefulness of our proposed aspect-based approach. We have proposed a good pattern-based approach to address the most compelling problems that increase the complexity of agent systems: the suitable structuring of agency properties and roles, and their disciplined composition to agent's state and behavior. Nevertheless, as a preliminary qualitative evaluation from the collected results, we have noticed that:

**The aspect-based approach supports better writability.** The use of design patterns with its demand on preplan for change, requires the definition of several classes and methods with only trivial structure and behaviour, 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-based approach supports better reuse.** Design patterns have no first-class representation at the implementation level. The implementation of a design pattern can therefore not be reused and, although its design is reused, the software developer is forced to implement the pattern many times. Unlike patterns, recent AOP 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-based approach supports better evolution.** During the evolution phase, the introduction of the Mobility property was much more simpler in the aspect-based design than in the pattern-based design with the use of Mediator (Section 5.1).

**The aspect-based approach 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-based approach 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 which 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.

**The aspect-based approach is less usable.** Since aspect-oriented programming is a recent technique, little experience in employing it is currently available. The usability of AOP is increasing at the implementation level, through the development of languages, tools and programming environments. Nevertheless, suitable support for aspect-oriented software development is still missing.

## 6 Comparison with Related Work

Some attempts to deal with agent complexity by using the object model have been proposed in the literature [15, 16]. Kendall et al. [16] proposes the Layered Agent architectural pattern, which decomposes agents into seven different layers, 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 it is not trivial removing any of these layers; it requires the reconfiguration of the adjacent layers. The afore mentioned 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 expressive 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 shizophrenia.

To implement agent's role aspects, we have followed Kendall et al. [15] 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 an 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 have 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.

## 7 Conclusions and Future Work

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, the development of software agents is not a trivial task. This work discussed the problems in dealing with agency concerns and overviewed software engineering approaches to address these problems. So, we presented an aspect-based approach to make development of sophisticated agents simple enough to be practical. In fact, the main contribution of this work is a design proposal (and corresponding implementation) that provides a unified framework for introducing complex software agents into the object model. Our proposal explores the benefits of aspect-based software engineering for the incorporation of agency aspects in object-oriented systems. Since aspect-oriented programming is still in its infancy, little experience with employing this paradigm is currently available. In this sense, we have presented a substantial case study (Section 2.2) that we have used to validate our aspect-based approach.

The achievement of good separation of concerns through the use of aspects is not a simple task. So, we are currently investigating a set of design principles and aspect-based design patterns that provide good design solutions for AOP [8]. The design principles should be proposed in order to specify a high-level description of the agent's organization in terms of its aspects and their interrelationships. Aspect-based design patterns can be used to provide solutions for each of the agency aspects of agents while following the overall guidelines of the proposed design principles.

## References

1. V. Basili et al. Experimentation in Software Engineering. *IEEE Transactions on Software Engineering,* SE-12(7), July 1986.

2. D. Bäumer, D. Riehle, W. Siberski, and M. Wulf, Role Object. In *Proceedings of the 1997 Conference on Pattern Languages of Programs* (PLoP '97), 1997.

3. J. Bigus and J. Bigus. Constructing Intelligent Agents with Java – A Programmers's Guide to Smarter Applications. Wiley, 1998.

4. G. Booch, and J. Rumbaugh. Unified Modeling Language – User Guide. Addison-Wesley, 1999.

5. J. Bradshaw, S. Dutfield, P. Benoit, and J. Woolley. KaoS: Toward an Industrial-Strength Generic Agent Architecture. In Software Agents, J. M. 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 and K. Sycara. A Model for Reusable Agent Systems. *Implementing Application Frameworks – Object-Oriented Frameworks at Work*, M. Fayad et al. (editors), John Wiley & Sons, 1999.

8. C. Chavez and C. Lucena. Design-level Support for Aspect-oriented Software Development. Position paper, *Workshop on Advanced Separation of Concerns in Object-oriented Systems at OOPSLA'2001*, Tampa, USA, 2001.

9. M. Elammari and W. Lalonde. An Agent-Oriented Methodology: High-Level and Intermediate Models. In *Proceedings of AOIS 1999 (Agent-Oriented Information Systems),* Heidelberg (Germany), June 1999.

10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

11. A. Garcia, C Lucena and Donald D. Cowan. Agents in Object-Oriented Software Engineering. To appear in Software: Practice & Experience, Elsevier, 2002.

12. A. Garcia, M. Cortés, C. Lucena. A Web Environment for the Development and Maintenance of E-Commerce Portals Based on Groupware Approach. In *Proceedings of the 2001 Information Resources Management Association International Conference (IRMA'2001)*, Toronto, May 2001.

13. J. Gosling, B. Joy, G. Steele. The Java Language Specification. Addison-Wesley, 1996.

14. W. Harrison and J. Ossher. Subject-Oriented Programming: A Critique of Pure Objects. In *Proceedings of OOPSLA'93*, ACM, pages 411-428, 1993.

15. E. Kendall. Agent Roles and Aspects. Position paper, *Workshop on Aspect-Oriented Programming, ECOOP'98* , July 1998.

16. 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. (editors), John Wiley & Sons, 1999.

17. M. Kersten and G. Murphy. Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-Oriented Programming. In *Proceedings of the OOPSLA'99*, Denver, USA, ACM Press, pages 340-352, 1999.

18. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin. Aspect-Oriented Programming. *European Conference on Object-Oriented Programming (ECOOP), LNCS,* (1241), Springer-Verlag, 1997.

19. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An Overview of AspectJ. In *Proceedings of European Conference on Object-Oriented Programming* (*ECOOP)*, Budapest, Hungary, 2001.

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

21. T. Nelson, D. Cowan, P. Alencar. A Model for Describing Object-Oriented Systems from Multiple Perspectives. *LNCS*, (1783):237-248, Springer-Verlag, 2000.

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

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

24. C.Petrie. Agent-Based Software Engineering. *Lecture Notes in IA*, Springer-Verlag, 2000.

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

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

27. O. Silva, D. Orlean, F. Ferreira, C. Lucena. A Shared-Memory Agent-Based Framework for Business-to-Business Applications. In *Proceedings of the 2001 Information Resources Management Association International Conference (IRMA'2001),* Toronto, May 2001.

28. P. Tarr, H. Ossher, W. Harrison and S. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns *In Proceedings 21st International Conference on Software Engineering (ICSE'99)*, May 1999.

29. TecComm Group. Frameworks and New Technologies to E-Commerce. URL: www.teccomm.les.inf.puc-rio.br, 2001