An Aspect-Based Approach for Developing Multi-Agent Object-Oriented Systems

Alessandro F. Garcia Viviane T. da Silva Carlos J. P. de Lucena Ruy L. Milidiú

Computer Science Department – TecComm Group
Pontifical Catholic University of Rio de Janeiro – PUC-Rio
Rua Marquês de São Vicente, 225 – 22453-900 Rio de Janeiro – Brazil
e-mail:{afgarcia, viviane, lucena, ruy}@inf.puc-rio.br

Abstract

Agent technology has been revisited as a complementary approach to the object paradigm in order to design and implement complex distributed software. Objects and agents have many similarities, but agents are also driven by beliefs, goals, capabilities, plans, and a number of agency properties such as autonomy, adaptation, interaction, learning and mobility. Moreover, cooperating software agents must incorporate different collaborative capabilities in order to work together in heterogeneous contexts. In practice, a complex application is composed of objects and multiple types of agents, each of them having distinct agency properties and capabilities. An additional difficulty is that these capabilities and properties typically overlap and interact with each other, and a disciplined scheme to composition is required. This paper discusses software engineering approaches for multi-agent systems, and presents a new approach for building multi-agent object-oriented software from early stage of design. This approach (i) describes structured integration of agents into the object model, (ii) incorporates flexible facilities to build different types of software agents, (iii) encourages the separate handling of each property and capability of an agent, (iv) provides explicit support for disciplined and transparent composition of agency properties and capabilities in complex software agents, and (v) allows the production of agent-based software so that it is easy to understand, maintain and reuse. The proposed approach explores the benefits of aspect-based design and programming for the incorporation of agents in object-oriented systems. We also demonstrate our multi-agent approach through the Portalware system, a web-based environment for the development of e-commerce portals.

1. Introduction

With agent-based software systems growing in size and complexity, the effort and cost of designing and implementing them while satisfying quality requirements, such as maintainability and reusability, are still deep concerns to software engineers. The design and implementation of a single agent is very complex. Software agents, like objects, include a specific set of capabilities (services) for their users. In fact, objects and agents have many similarities [4, 30], but the agents are driven by beliefs, goals, plans, and a number of agency properties such as autonomy, adaptation, interaction, learning and mobility. Moreover, software agents generally must incorporate different collaborative capabilities to cooperate with other agents in heterogeneous contexts. In practice, a complex application is composed of objects and multiple types of agents, each of them having distinct capabilities and agency properties. Agents pose other design and implementation problems because many properties and capabilities of agents 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 property and capability of an agent 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 so that it is easy to understand, maintain and reuse. In this context, research in software engineering of multi-agent systems has been carried out according two different approaches. Researchers in the first approach [18, 32, 33] argue persuasively that adopting a multi-agent approach to system development affords software engineers a number of significant advantages over contemporary methods and, therefore, they view multi-agent systems as a “new software engineering”. In contrast, researchers in the second approach [13, 20, 21, 26, 30] propose the integration of agents into the object-orientation world and, thus, they think of objects and agents as complementary abstractions; as a result, they have centered on extending existing techniques from object-oriented software engineering to agent-based systems. Although there are several motivations to introduce software agents in the object model [17, 30], it is not a trivial task due to the differences between objects and agents [18].

Traditionally, existing object-oriented proposals often focus on the implementation phase, and do not provide direct support for handling and reusing properties and capabilities separately (e.g. [2], [6] and [26]). Moreover, these current proposals generally support a limited number of agent types, and the state and behavior of an agent generally are encapsulated as an object. Even thought 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 properties at the same time. In addition, the lack of support for dealing with the interactive and overlapping nature of agency properties 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 collaborative capabilities of an agent with its core state and behavior.

In this paper, we briefly discuss the current research in software engineering of agent systems, and present an innovative aspect-based approach for designing and implementing agent-based object-oriented systems. Our proposal 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 crosscutting concerns. Aspect is the abstraction that modularizes a crosscutting concern and is associated with one or more objects. Our approach explores this abstraction to support the construction of multi-agent object-oriented software with improved structuring for design reuse and evolution. We will also present results applying our approach to introduce multiple software agents in Portalware [14], a web-based environment for the development of e-commerce portals. To implement this system, we have used AspectJ [24] which is practical aspect-oriented extension to the Java programming language [15].

The remainder of this paper is organized as follows. Section 2 gives a brief description of definitions and applications of multi-agent systems. This section also introduces an example which is used throughout this paper to illustrate our approach. Section 3 overviews software engineering approaches for agent systems, and introduces aspect-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 assesses the relative advantages and disadvantages of applying our approach, and describes some implementation issues. 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 often are viewed as complex objects with an attitude [5], in the sense of being objects with some additional agency properties. In general, the state of an agent is formalized by knowledge, and is expressed by mental components such as *beliefs*, *goals*, *plans* and *capabilities* [30, 35]. 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 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 [21]. 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 [8]. 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 succesful plan using an agent's beliefs. 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 [29], such as *information agents*, *user agents*, and *interface 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

Agency Properties and Agenthood. The state and behavior of an agent is respectively affected by and 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 [21, 29, 30] and our experience in developing multi-agent systems [14, 34, 36, 38]. 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* [30] (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 [21]. Since agents are autonomous software entities, the agent itself starts its control thread and decides between accepting and rejecting incoming messages. Since the message is accepted, the agent can have to adapt its mental 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 its knowledge when interacting

with its environment (learning), (ii) move itself from one environment in a network to another (mobility), and (iii) join a conversation channel with other agents (collaboration). Each agent type typically has different application-specific capabilities and agency properties.

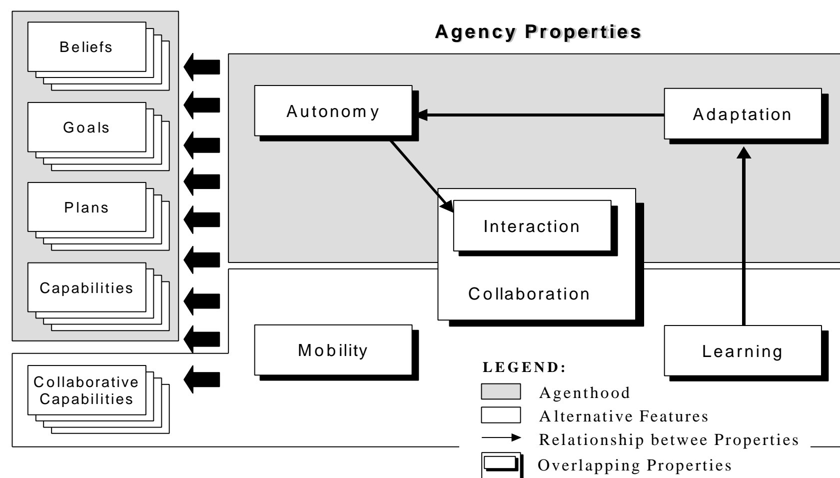

Figure 1. A Definition for Agenthood

Collaborative Capabilities. An agent can use the capabilities provided by other agents via some communication language. Collaborative capabilites 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 generally includes different collaborative capabilities in order to work together in multiple contexts. In order to perform a cooperation, a collaborative plan is instantiated, and it chooses the eligible collaborative capabilities.

Interacting and Overlapping Properties. By the very nature of agency properties, these properties are not ortoghonal – in general, they interact with each other (Figure 1). For instance, the 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 accepting an incoming message. In addition, two agency properties are overlapping: interaction and collaboration. Collaboration is viewed as a more sophisticated interaction form, 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 additionaly defines how to collaborate, it addresses the coordination protocol. A simple coordination protocol consists of synchronizing the agent waiting for a response.

2.2. Software Agents in Portalware: A Case Study

Figure 2 illustrates the software agents in Portalware [9], 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 has different capabilities and properties, but everyone implements the fundamental aspects definided by agenthood. Figure 2 summarizes capabilities and agency properties for the Portalware agents. For purpose of brevity, we discuss in detail only the Portalware's information agents. For a more complete discussion about this example the reader can refer to [8]. 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 alternative collaborative capability is used, when an information agent is not able to find the information

in the attached database. The agent uses its calling collaborative capabilities in order to call the other information agent and ask for this information. Similarly, the latter uses its answering capabilities so that it can receive the request and send the search result. Note that both of them need to include calling and answering capabilities.

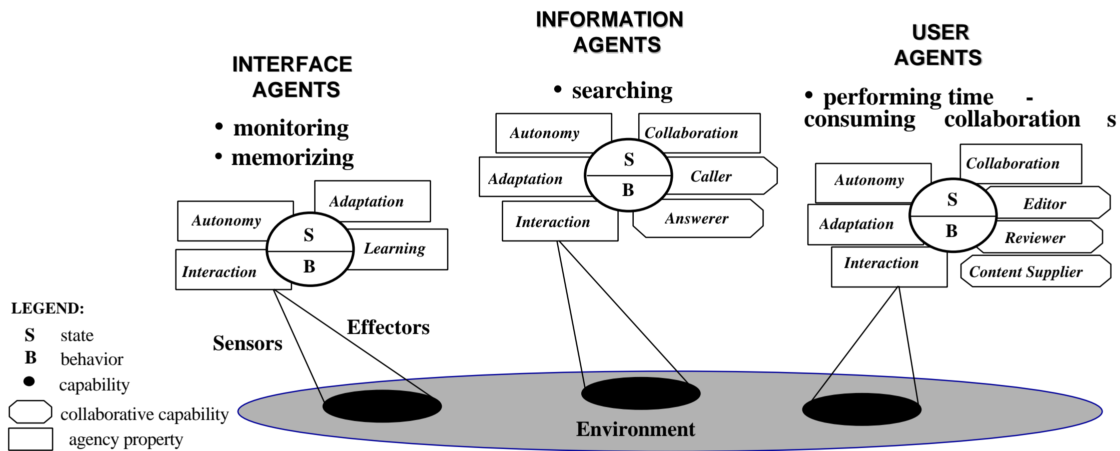

Figure 2. Portalware Agents.

3. Software Engineering for Agent Systems

The inherent complexity in the organization and design of software agents makes it necessary for developers to apply appropriate software engineering approaches. Modularity and separation of concerns are two complementary well-established principles in software engineering that use high-level abstractions to hide complexity [37]. In addition, software engineers often need to capture multiple architectural descriptions, each representing a certain perspective of the system's architecture and dealing with the multiple system's modules and concerns. For example, complex problems can be decomposed into different architectural perspectives, including: (i) a structural perspective, (ii) a behavioral perspective, and (iii) an organizational perspective. The importance of these principles and distinct descriptions increases as new technologies are introduced and software applications such as agent-based applications, become more complex.

Figure 3 provides a framework that assists in identifying relationships between architectural decomposition, modular decomposition, and concern decomposition. 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 [28]. From the viewpoint of concern decompositions, complex problems can be divided into different abstractions, such as: (i) roles [25], (ii) views [11], (iii) features [1], (iv) aspects [23], and (v) subjects [16]. 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 [28]. In other words, concerns naturally cut across application modules.

According to the framework pictured in Figure 3, we can classify current approaches to multi-agent software engineering into two categories: (i) *agent-based software engineering*, and (ii) *object-oriented software engineering for agent systems*. Both these current approaches have concentrated on modular decomposition. However, our proposal follows the second approach and additionally extends it with the application of recent advances in separation-of-concerns techniques [23, 37]. So, we present an engineering software approach which explores the benefits of both modular and concern decompositions. In particular, our proposal

uses the advantages of aspect-oriented design and programming in order to deal with the complexity of integrating software agents in the object model.

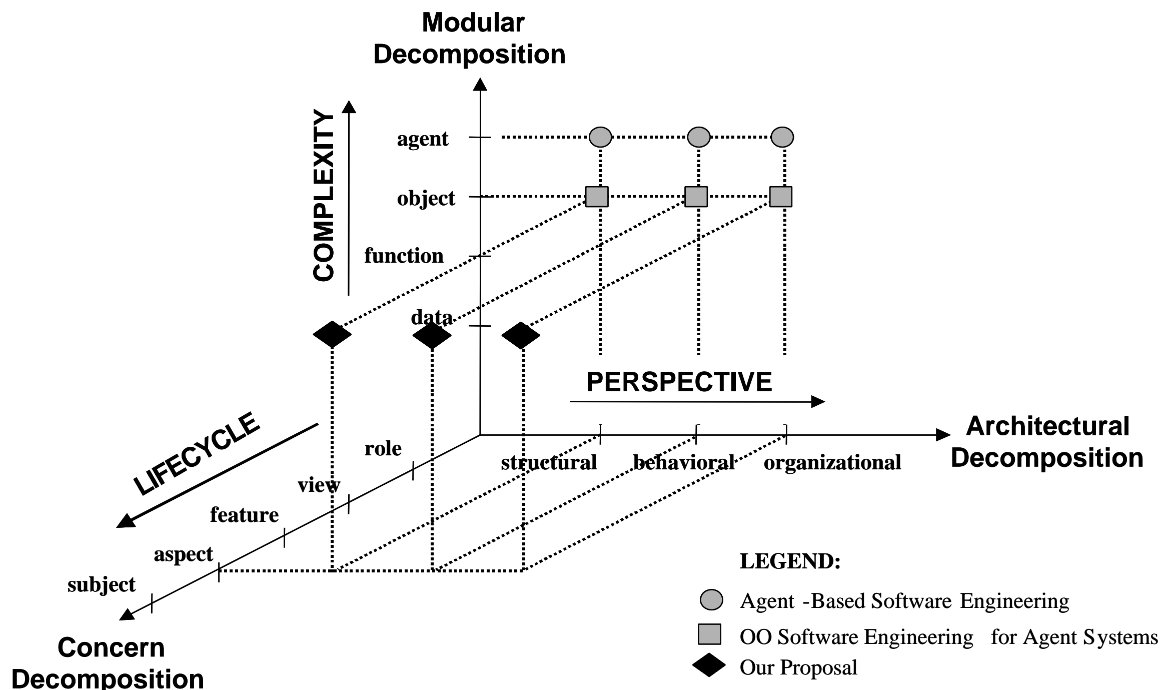

Figure 3. Software Engineering Approaches for Agent Systems (Based on [28]).

3.1. Agent-Based Software Engineering

Researchers in agent-based software engineering such as those in [19, 32] argue persuasively that adopting a multi-agent approach to system development affords software engineers a number of significant advantages over contemporary methods and, therefore, they describe multi-agent systems as "new software engineering". According to these researchers, agent systems are often more complex than object-oriented systems and hence the traditional object model generally fails to capture the complexity of agent systems. In this approach, agents are a new abstraction that substitutes for the object abstraction and realizes the agent abstraction as a software engineering paradigm. As a result, proponents of this approach claim it is necessary to develop new software engineering techniques, methods and methodologies that are specifically tailored to agents, as well as software architectures, programming languages and tools supporting these techniques, methods and methodologies. We refer to [17, 39] for a more complete survey.

3.2. Object-Oriented Software Engineering for Agent Systems

In contrast to the previous approach, other researchers [7, 13, 21, 26, 30] propose the integration of agents into the object-oriented world and, thus, they think of objects and agents as complementary abstractions. As a result, they have concentrated on extending existing techniques from object-oriented software engineering to agent-based systems. The central idea of this approach is the addition of additional features to objects so they become agents. In fact, object-oriented software engineering has proved to be extremely powerful for building complex systems, which promotes modularity, maintainability, and reusability. Moreover, object-oriented software engineering has evolved by introducing successful techniques, such as object-oriented frameworks [9] and design patterns [12]. For example, Kendall *et al.* [21] propose an object-oriented framework for developing multi-agent applications. This

framework is based on the layered agent architectural pattern, which separates different layers of an agent, such as sensory layer, collaboration layer, and so on. Section 7 compares Kendall's approach with our proposal (Section 4).

3.3. 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 (properties) of interest in complex systems. Aspect-oriented software engineering does for concerns that are naturally cut across each other what object-oriented software engineering does for concerns that are naturally hierarchical – it provides mechanisms that explicitly capture the crosscutting structure. Thus, the goal of aspect-oriented design and programming [10, 23] 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 4). Examples of join points are method calls and receptions, method executions, and field sets and reads. Pointcuts are collections of join points. AspectJ [24] is a practical aspect-oriented extension to the Java programming language [15].

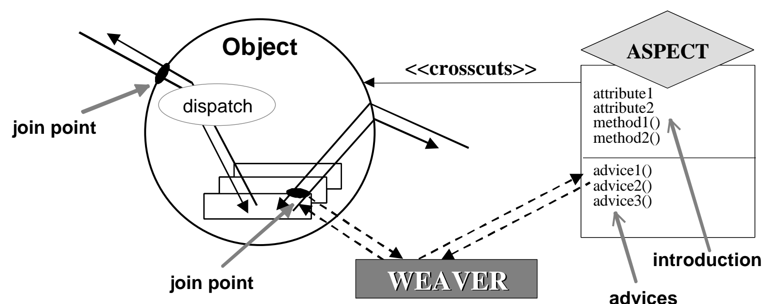

Figure 4. AspectJ Mechanisms for Dealing with Crosscutting Aspects.

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 when a join point is reached and before the computation proceeds, i.e. that runs when computation reaches the method call and before the actual method starts running; (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 when the join point is reached, and has explicit control whether the computation under the join point is allowed to run at all. Aspects are modular units of crosscutting implementation that are associated with one or more objects, comprised of pointcuts, advices, and *introduction*. 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 deviating the normal control flow to an advice, when program execution point is at a join point (Figure 4). Up to the current version of AspectJ, almost all of the weaving process is realized as a pre-processing step at compile-time [10].

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

In the following, our agent-multi 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) *collaborative aspects*, (vi) *aspect composition*, and (vii) *agent evolution*. We adopt UML diagrams [3] as the modeling language throughout this paper. The design notation for aspects is based on [22]: 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 follows: adviceKind(pointcut):adviceName.

4.1. Agent's Core State and Behavior

In our approach, classes are used to represent agents and their constituent components. 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 the application's agents. Since agent state is described in terms of its goals, beliefs, and plans, the attributes of an Agent object should hold references to objects that represent these elements, namely Belief, Goal and Plan objects.

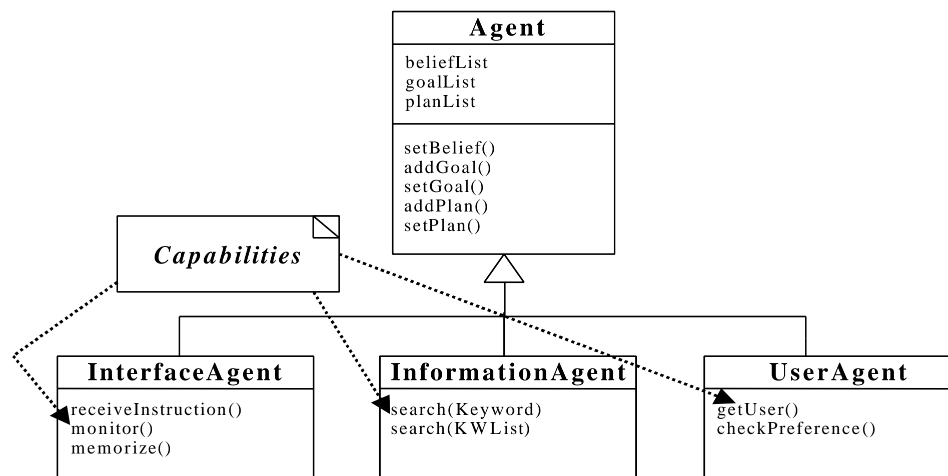

Figure 5. Agent Types.

Methods of the Agent class are used to update its basic state and 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. Plan classes 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.

4.2. Agent Types

Our approach proposes the use of inheritance in order to create different 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): (i) the InterfaceAgent class, (ii) the InformationAgent class, and (iii) the UserAgent class. For example, the method search(Keyword) of the InformationAgent class implements the capability of information agents searching 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 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).

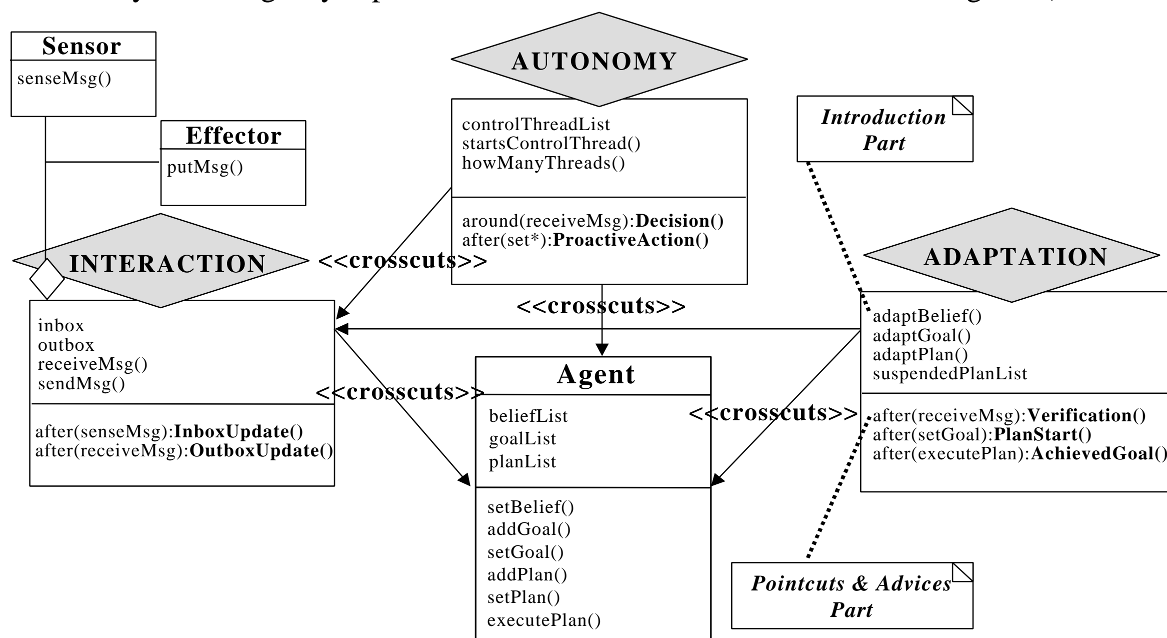

Figure 6. Agency Aspects and the Design for Agenthood.

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 receptions of calls to the senseMsg() 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 putMsg() method in order to update the agent outbox. Since the process of sending and receiving messages occurs quite often 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 and 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. Then, this advice is attached to the pointcut that is a collection of receptions of calls to the receiveMsg() method. The ProactiveAction() after advice implements the agent ability to act without direct external intervention (proactive behavior); to each invocation of methods with prefix 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 themselves, 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 when 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 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.

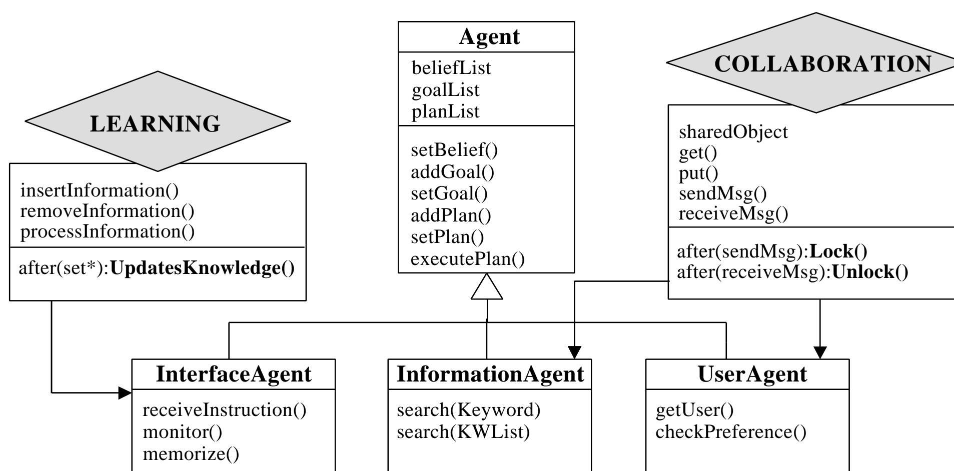

Figure 7. Particular Agency Aspects.

4.5. Collaborative Aspects

Aspects should be used to implement the collaborative capabilities of an agent. These aspects are termed *collaborative agency aspects*. A collaborative agency aspect is a part of an agent which defines the activity of the agent within a set of particular collaborations. As a result, it decouples the agent's basic capabilities from the collaborative capabilities, which in turn improves understanding, reusability and evolution. Since an Agent object needs to include multiple collaborative capabilities, different collaborative agency aspects are associated with this object.

Figure 8 illustrates this situation for the information agents of Portalware (Section 2.2). An information agent needs to support calling and answering capabilities in order to cooperate with the other information agent in different contexts. It must be able to receive or make calls. Thus, the Caller and Answerer collaborative aspects are attached to the InformationAgent class. The Caller aspect introduces to an agent the ability to send the search request to the answering agent as well receive the search result. Similarly, the Answerer aspect introduces the ability to receive the search request and to send the search result. The startsCaller() after advice is associated with receptions 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 method result is null. Note that these collaborative capabilities are introduced in a way that is transparent and non-intrusive.

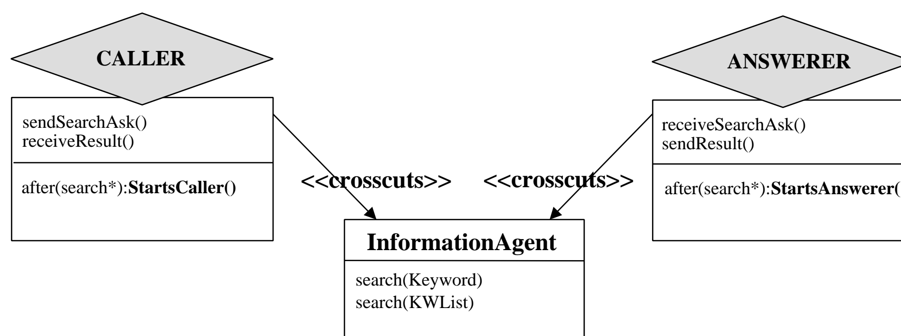

Figure 8. Collaborative Aspects of Agents.

4.6. Aspect Composition

As we have stated previously (Section 2.1) agency properties are not ortoghonal, they can overlap and interact with each other. As a result, there is a need for capturing the interactive and overlapping characteristics of the multiple agency aspects. Our model establishes relationship patterns which provide design rules that encompass the non-orthogonality of agency properties. 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 to be acepted. The Adaptation aspect interacts with the Autonomy aspect in order to adapt the agent state and behavior when a incoming message is acepted. As a consequence, these aspects implement different advice for the same pointcut that comprises receptions to calls 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.

Figure 9 shows an interaction diagram for a basic call scenario between two information agents in Portalware. Weaver is the mechanism responsible for composing the multiple agency aspects. It directs the normal control flow to an advice when program execution is at a join point (Section 3.3). An information agent receives a message to search for an information according to a specified keyword. The Interaction aspect receives this message through a sensor and updates the Inbox with the new message. The Autonomy aspect performs the decision-making process by invoking the decision plan. Thus, for every message that the agent receives, it may determine, based on its own goals and state, and the state of the ongoing conversation, whether to process the message and how to respond if it does. After the Autonomy aspect decides on processing the message, the Adaptation aspect adapts the agent's goals, since the agent must use a new search goal. When the goal is set, the Adaptation aspect

seeks for an appropriate agent plan for achieving this goal and adapts the agent's behavior for carrying out this search plan. The agent selects a plan on the basis of the current situation. During the plan execution, the agent's searching capability, the search() method is invoked. Since the information agent is not able to find the specified keyword, the Caller aspect is activated in order to perform the caller collaborative capability. It calls the other information agent and asks for the required information. The Interaction aspect sends the message to the answerer information agent. After the message is sent, the Collaboration aspect carries out the coordination protocol so that the caller agent waits for the search result. Similarly, the agent receiveing the message uses its answerer capabilities to receive the request and send the search result. It is worthwhile to highlight that advices and methods of dependent aspects are performed everytime after aspects on which they depend.

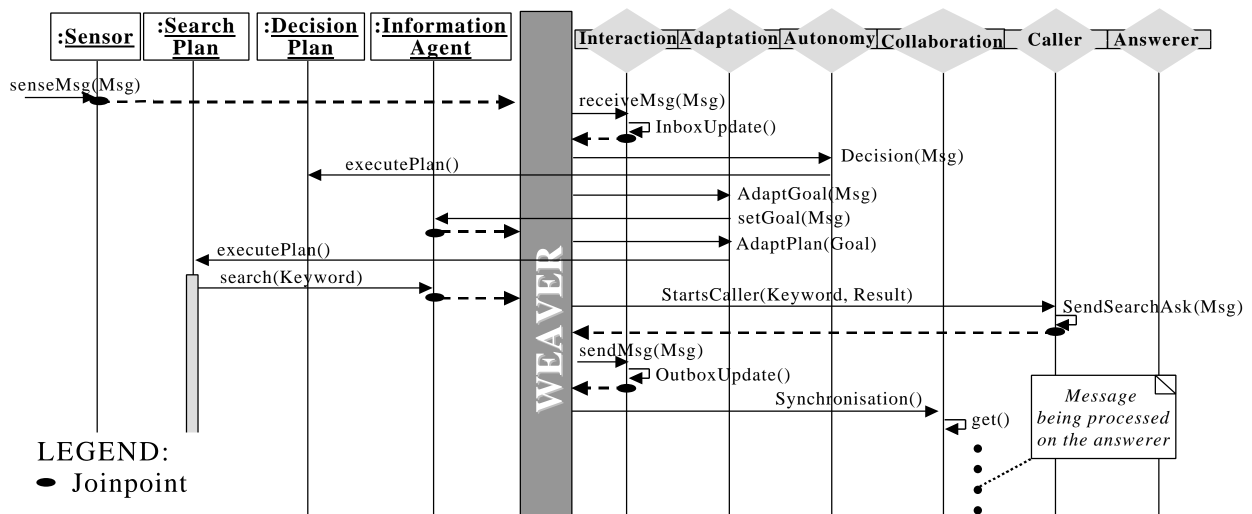

Figure 9. An Interaction Diagram for the Portwalware's Information Agents.

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 in order 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 have the calling and answering capabilities, but must to be mobile. In our model, this modification is done transparently, since agency aspects can be added to or removed from classes in a plug-and-play way. The Caller and Answerer aspects are disattached from the InformationAgent class without requiring any invasive adaptation for the other agent's components. The remaining behavior of the agent is kept. However, it is necessary to associate the Mobility aspect, which introduces to the Agent class the ability to roam the network and gather information on behalf of its owner. This association process includes defining the end of 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. It takes results of searching methods to check if the information agent was not able to find the information. Thus, if the method result is null, this aspect is responsible for migrating the agent to the other host in order to start a new search for the information.

5. Discussion and Implementation Issues

Althougth 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 every distinct definition since agency aspects can be attached to and removed from the Agent class.

We are currently developing a case study comparing two approaches for developing multi-agent systems. In this case study, we have designed and implemented Portalware using object-oriented programming (OOP) and aspect-oriented programming (AOP). Our goal is to find out which of these techniques allows building agent-based applications easier to write, read, maintain and reuse. We have found our aspect-oriented approach promotes better readability and reusability since the code for agent's basic capabilities is not amalgamated with the code devoted to the different agency properties and collaborative capabilities. For this reason, basic and collaborative capabilities as well as agency properties are easier to read and maintain.

Our aspect-based approach was implemented for the case study using AspectJ [34]. The implementation consists of 91 classes, 5 agency aspects, and 5 collaborative aspects. In order to implement the dependency relationship between these different aspects, we used the "dominates" construct of AspectJ. In addition, we have used two aspects which implement application's general aspects: exception handling and persistence. For instance, the Persistence aspect is applied for dealing with the persistence of an agent's state. In this sense, we have also used TSpaces/IBM [27], a blackboard architecture for network communication with database capabilities. TSpaces provides group communication services, database services and event notification services. It is implemented in the Java programming language and thus it automatically possesses network ubiquity through platform independence, as well as standard type representation for all datatypes. In our prototype, messages captured by different collaborations were implemented as tuples and finally, agents communicated by posting such tuples to the blackboard.

In our case study (Section 2.2), different instances of information agents should have varying characteristics. They may be collaborative or non-collaborative, they may be static or mobile, they may or may not learn. So, it is desirable to build personalized information agents to attach different aspects to distinct instances of information agents. Our proposed model supports this feature. However, the current version of AspectJ does not provide direct support for associating different aspects with different class instances. This feature is currently supported by some meta-object protocols such as Guaraná [31].

6. Comparison with Related Work

Some attempts to deal with agent complexity by using the object model have been proposed in the literature [20, 21]. Kendall et al [21] 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 evolution of this design is cumbersome since it is not trivial removing any of these layers; it requires the reconfiguration of the adjacent layers. This 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, we have found that the use of design patterns for the agent domain introduces a number of problems: (i) class explosion, (ii) need for preplanning, (iii) the application of the suitable design patterns is not trivial, and (iv) lack of expressive power. Finally, this proposal causes object shizophrenia – the agent state and behavior, which are intended to appear as a single object, are actually distributed over multiple objects.

We have followed Kendall et al [20] guidelines that describes the application of aspect-oriented programming to implement role models, to implement agent's collaborative aspects. However, their work does not deal with agents' agency properties, which we believe are the main source of agent complexity. In this sense, our paper presents a unified framework for dealing with collaborative capabilities and agency properties, and their interrelationships.

Research in aspect-oriented software engineering has concentrated on the implementation phase. A few works have presented aspect-based design solutions. In addition, since aspect-oriented programming is still in its infancy, little experience with employing this paradigm is currently available. To date, aspect-oriented programming has been used to implement generic aspects such as persistence, error detection/handling, logging, tracing, caching, and synchronization. However, each of these papers is generally dedicated to 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 (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 properties and capabilities as well as overviewed software engineering approaches to addressing these problems. So, we presented an aspect-based approach to make development of sophisticated agents simple enough to be practical. In fact, the main contribuition of this work is an implementation and design proposal that provides a unified framework for introducing complex software agents to 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), which we have used to apply our aspect-based approach.

Design patterns [12] are important vehicles for constructing high-quality software. Architectural patterns define the basic structure of an architecture and of systems which implement that architecture; design patterns are more problem-oriented than architectural patterns, and are applied in later design stages. As presented in this paper, aspect-based design can be used to address the problems of dealing with agency properties. We are currently investigating a language of aspect-based design patterns for agent systems, which provide good design solutions for dealing with each of the agency aspects of agents. An architectural pattern 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 structure of the proposed architectural pattern.

References

1. M. Aksit, L. Bergmans, and S. Vural. "**An Object-Oriented Language-Database Integration Model: The Composition Filters Approach**". Proceedings of ECOOP'92, Lecture Notes on Computer Science, vol. 615, 1992.
2. J. Bigus and J. Bigus. "**Constructing Intelligent Agents with Java – A Programmers's Guide to Smarter Applications**". Wiley, 1998.
3. G. Booch, and J. Rumbaugh. "**Unified Modeling Language – User Guide**". Addison-Wesley, 1999.
4. J. Bradshaw, S. Dutfield, P. Benoit, and J. Woolley. "**KaoS: Toward an Industrial-Strength Generic Agent Architecture**". In: J. M. Bradshaw (editor), Software Agents. Cambridge, MA: AAAI/MIT Press, 1996.
5. J. Bradshaw. "**An Introduction to Software Agents**". In: *Software Agents*, J. Bradshaw (editor), American Association for Artificial Intelligence/MIT Press, 1997.
6. D. Brugali and 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.
7. A. Chavez, D. Dreilinger, R. Guttman, and P. Maes. "**A Real-Life Experiment in Creating an Agent Marketplace**". Proceedings of the 2nd Int. Conference on the Practical Application of Multi-Agent Technology (PAAM'97), London, UK, April 1997.
8. M. Elammari and W. Lalonde. "**An Agent-Oriented Methodology: High-Level and Intermediate Models**". Proceedings of AOIS 1999 (Agent-Oriented Information Systems), Heidelberg (Germany), June 1999.
9. M. Fayad et al. "**Implementing Application Frameworks**". First Edition, John Wiley & Sons, 1999.
10. C. von Flach, C. Lucena, M. Fontoura. "**A Design Model for Aspect-Oriented Software Development**". Technical Report, PUC-Rio, 2001.
11. A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. "**Viewpoints: a Framework for Integrating Multiple Perspectives in System Development**". International Journal of Software and Knowledge Engineering, 2(1):31-37, 1992.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "**Design Patterns: Elements of Reusable Object-Oriented Software**". Addison-Wesley, Reading, MA, 1995.
13. A. Garcia, C Lucena and Donald D. Cowan. "**Agents in Object-Oriented Software Enginnering**". Technical Report CS-2001-07, Computer Science Department, University of Waterloo, Waterloo, Canada, March 2001.
14. A. Garcia, M. Cortés, C. Lucena. "**A Web Environment for the Development and Maintenance of E-Commerce Portals Based on Groupware Approach**". Proceedings of the 2001 Information Resources Management Association International Conference (IRMA'2001), Toronto, May 2001.
15. J. Gosling, B. Joy, G. Steele."**The Java Language Specification**".Addison-Wesley, 1996.
16. W. Harrison and J. Ossher. "**Subject-Oriented Programming: A Critique of Pure Objects**". Proceedings of OOPSLA'93, ACM, pages 411-428, 1993.
17. C. Iglesias, M. Garrijo, and J. Gonzalez. "**A Survey of Agent-Oriented Methodologies**", Proceedings of the 5th International Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages (ATAL-98), Paris, France, July 1998, pp. 317-330.
18. N. Jennings. "**On Agent-Based Software Engineering**". Artificial Intelligence 117(2000) 277-296, Elsevier.
19. N. Jennings and M. Wooldridge. **"Agent-Oriented Software Engineering"**. In: J. Bradshaw (editor), Handbook of Agent Technology, AAAI/MIT Press, 2000.
20. E. Kendall."**Agent Roles and Aspects**". ECOOP Workshop on Aspect Oriented Programming, July, 1998.

21. 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.
22. M. Kersten and G. Murphy. "**Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-Oriented Programming**". Proceedings of the OOPSLA'99, Denver, USA, ACM Press, pp. 340-352, 1999.
23. 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'97), Finland. Springer-Verlag LNCS 1241. June 1997.
24. G. Kiczales et al. "**An Overview of AspectJ**". Submitted to ECOOP'2001, Budapest, Hungary, 2001.
25. B. Kristensen. "**Roles: Conceptual Abstraction Theory and Practical Language Issues.**" Special Issue of Theory and Practice of Object Systems (TAPOS) on Subjectivity on Object-Oriented Systems, 1996.
26. D. Lange and M. Oshima. **"Programming and Developing Java Mobile Agents with Aglets."** Addison-Wesley, August 1998.
27. T. Lehman, S. McLaughry, and P. Wyckoff. "**TSpaces: The Next Wave**". Hawaii International Conference on System Sciences (HICSS-32), January 1999.
28. T. Nelson, D. Cowan, P. Alencar. "**A Model for Describing Object-Oriented Systems from Multiple Perspectives**". *Lecture Notes on Computer Science*, Springer-Verlag, (1783) 237-248, 2000.
29. H. Nwana. "**Software Agents: An Overview**". Knowledge Engineering Review, 11(3): 1-40, September 1996.
30. Object Management Group – Agent Platform Special Interest Group. "**Agent Technology – Green Paper**". Version 1.0, September 2000.
31. A. Oliva and L. Buzato. "**The Design and Implementation of Guaraná**". 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99), May 3-7, 1999, San Diego, CA, USA.
32. C. Petrie. "**Agent-Based Software Engineering**". Lecture Notes in IA, Springer-Verlag, 2000.
33. C. Petrie. "**Agent-Based Engineering, the Web, and Intelligence**". IEEE Expert, Special Issue on Intelligent Systems and their Applications, 11(6), December 1996:24-29.
34. 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.
35. Y. Shoham. "**Agent-Oriented Programming**". Artificial Intelligence, 60(1993): 24-29.
36. O. Silva, D. Orlean, F. Ferreira, C. Lucena. "**A Shared-Memory Agent-Based Framework for Business-to-Business Applications**". Proceedings of the 2001 Information Resources Management Association International Conference (IRMA'2001), Toronto, May 2001.
37. P. Tarr, H. Ossher, W. Harrison and S. Sutton. "**N Degrees of Separation: Multi-Dimensional Separation of Concerns.**" Proceedings of 21st International Conference on Software Engineering (ICSE'99), May 1999.
38. TecComm Group. "**Frameworks and New Technologies to E-Commerce**". URL: www.teccomm.les.inf.puc-rio.br
39. M. Wooldridge, N. Jennings. "**Agent Theories, Architectures, and Language: A Survey**". In: M. Wooldridge & N. Jennings (editors), *Intelligent Agents: ECAI-94 Workshop on Agent Theories, Architectures, Languages*, Berlin, Spring-Verlag (pp. 1-39).