Applying Proto-Frameworks in the Development of Multi-Agent Systems

Andrés Díaz Pace[1], Federico Trilnik, Marcelo Campo

ISISTAN Research Institute, UNICEN University
Campus Universitario, Paraje Arroyo Seco - (B7001BBO) Tandil, Argentina
[1]Also CONICET
{adiaz, ftrilnik, mcampo}@exa.unicen.edu.ar

Abstract

Current trends in software development are increasingly reasoning about software applications in terms of multi-agent systems (MAS). However, the development of multi-agent applications is still a technically difficult task. One of the main barriers is the lack of comprehensive design practices to move systematically from problem analysis and agent models to effective implementations. This work describes a design approach to assist multi-agent development based on the notion of proto-frameworks, which proposes the materialization of agent models expressed in terms of architectural models into object-oriented counterparts, enabling then different implementation alternatives for multi-agent frameworks according to specific developers´ needs.

Keywords: Multi-agent development, design approach, frameworks, architectural styles.

1. Introduction

The adoption of the agent paradigm [Demazeau91, Sycara98, Weiss99] in software development necessarily involves news ways of conceiving software systems emphasizing properties such as modularity, autonomy, distribution and adaptability. As current software applications get bigger both in size and complexity, multi-agent systems (MAS) appear as one of the most promising approaches to handle such continuous growing [Wooldridge96]. The research in the field includes several lines of work regarding agent languages [Finin97], organizational models [Gutknecht97], learning [Rao90, Nilsson96] and coordination [Lesser98], among others. Nonetheless, despite agent benefits, the development of multi-agent applications has been mostly homegrown [Bradshaw97] and it still requires a strong grasp of experience and design skills. This fact has somehow limited the application of MAS in practical cases. In contrast with conventional object-oriented technologies, MAS seem still not mature enough to provide comprehensive mechanisms that can be applied by developers in order to make a transition from problem analysis to design and implementation matters. Therefore, it is reasonable to argue that a combination of agents with existent object-oriented techniques can contribute to provide computational models promoting better modularity and adaptability in software products. In particular, object-oriented frameworks [Johnson97, Fayad99] can play a key role in this context. They can be useful means to express reusable designs and capture the essence of patterns, algorithms, components and architectures, even when some extension mechanisms may need to be considered in order to adapt frameworks to the agent requirements.

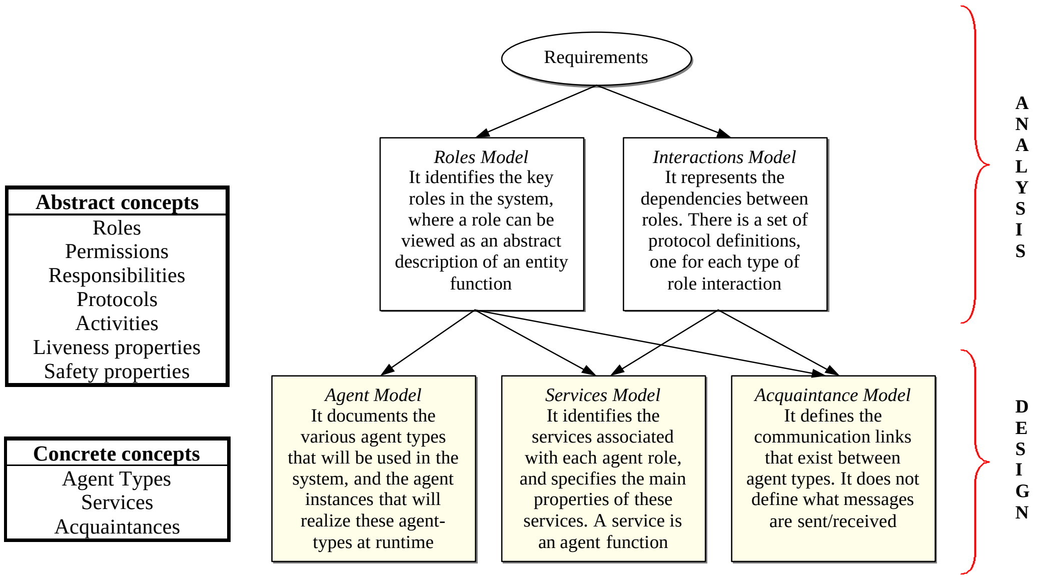
In this work, we describe a design approach to assist multi-agent development based on the application of a method for object-oriented framework design which combines architectural styles [Shaw96, Bass98] with object-oriented concepts. This design method relies on the notion of *object-oriented materialization of software architectures* derived from non-object-oriented architectural styles, but it is general enough to support the development of MAS as well. The roots of this work started with the development of a simple object-oriented framework called *Bubble*, initially designed for multi-agent simulation [DiazPace00]. This framework was used to support the implementation of several simulation applications on different domains, but the most relevant part of this process was that *Bubble* became also the basis for building other frameworks on top of it. After several refinements of *Bubble*, we realized that the framework actually provides a set of domain-independent architectural abstractions which made easier the mapping of domain concepts into computational components. Additionally, the object-oriented representation enabled the reuse of essential mechanisms provided by the architecture in the same way a normal framework does. However, the mechanisms provided by *Bubble* are more oriented towards a generic architectural behavior than a generic behavior of domain. This aspect made difficult to classify the framework into the different framework categories proposed by the literature. Essentially, *Bubble* is a framework reused by inheritance (i.e., a white-box framework), but from a functional point of view it cannot be considered as a horizontal or service framework because it is *not used* by different framework. Instead of this, different frameworks can be derived, by inheritance, from *Bubble* through the mapping of domain components into proper *Bubble* components. For this reason, we refer to *Bubble* as a *proto-framework* [Campo01], that is an object-oriented framework that provides the essential basis to build new frameworks that adopt the *Bubble* underlying architecture. As regards multi-agent systems, the notion of proto-frameworks can serve to the design of multi-agent applications by allowing the materialization of agent models expressed in terms of architectural models into object-oriented counterparts, and then enabling different implementation alternatives for multi-agent frameworks according to specific developer needs.

The paper presents our experience with development of the proto-framework concepts, and its application to the development of MAS. In particular, we take as case-study the development of an industrial framework for Enterprise Quality Management systems using multi-agent technology. The rest of the paper is organized into six sections. The first section discusses about frameworks and multi-agent development. The second section introduces the foundations of proto-frameworks. Then, the third section talks about the design of a particular proto-framework called Bubble. The fourth section describes the multi-agent case-study mentioned before. And finally, the last two sections present the lessons learned and the conclusions of this work.

2. The Role of Frameworks in the Development of MAS

The concepts of MAS are being spread everywhere. Many software applications are being viewed, more and more, as groups of autonomous entities collaborating with one another to collectively accomplish some goals. This popularity has lead some authors to talk about a so-called *multi-agent software engineering* [Wooldridge97, Jennings99, Garijo99]. These arguments are based on the suitability of multi-agent models to better capture the intrinsic complexity of software applications composed by a large number of parts with many interactions among them. However, as the research in the field is recently emerging, we cannot find too much quantitative evidence about these facts. Reality shows that the multi-agent paradigm still lacks of well-established design practices to move systematically from problem analysis and agent models to effective implementations. In recent times, there has been a number of attempts to provide methodologies for MAS [Burmeister96, Drogoul98, Wooldridge00]. To exemplify these issues, let's take for example the *Gaia* methodology [Wooldridge00], which proposes a set of models to deal with analysis and design of MAS. The methodology aims the development of multi-agent applications through successive refinements of several agent models, but it remains neutral with respect to both the target domain and the agent architecture. Figure 1 shows a summary of the *Gaia* models and its more important concepts. For the sake of our discussion, we will focus on the design phase proposed by the methodology, that is how to transform the abstract models derived during the analysis phase into more detailed specifications that can be directly mapped to agents. Readers looking for more information can consult the original work [Wooldridge00].

Very briefly, the *Gaia* design process comprises three models: an agent model, a services model and an acquaintance model. Firstly, an agent model is created, identifying the agent types that will make


Requirements
Roles Model
It identifies the key roles in the system, where a role can be viewed as an abstract description of an entity function
Interactions Model
It represents the dependencies between roles. There is a set of protocol definitions, one for each type of role interaction
Abstract concepts
Roles
Permissions
Responsibilities
Protocols
Activities
Liveness properties
Safety properties
Agent Model
It documents the various agent types that will be used in the system, and the agent instances that will realize these agent-types at runtime
Services Model
It identifies the services associated with each agent role, and specifies the main properties of these services. A service is an agent function
Acquaintance Model
It defines the communication links that exist between agent types. It does not define what messages are sent/received
Concrete concepts
Agent Types
Services
Acquaintances
ANALYSIS
DESIGN

Abstract notions are those used during analysis to conceptualize the system, but they do not necessarily have any direct realization within the system. Concrete notions, in contrast, are used within the design process, and will tipically have direct counterparts in the runtime system

Figure 1. Example of analysis and design models in the Gaia methodology [Wooldridge00]

the system work and the agent instances that will be generated from these types. Then, a services model should specify the main services that are required to materialize each agent's role. And finally, an acquaintance model derived from the interaction model and the agent model should document the lines of communication between the agents. *Gaia* is mainly concerned with how a society of agents cooperate to achieve the system-level goals, and what is required of each individual agent in order to do this. However, the methodology says nothing about how an agent should accomplish its services, because it will depend on the particular domain. *Gaia* relies on an organizational viewpoint and uses a mostly top-down approach based on a progressive decomposition of behaviors, but it does not provide any direct, explicit representation of such structures, for example through some kind of organizational patterns. We think these shortcomings may be better exploited if the design process is guided by technologies such as architectural styles and frameworks, in order to simplify the development of more reusable and adaptable multi-agent applications.

Presently, there is a diversity of generic multi-agent models and architectures, ranging from agent platforms for the development of applications (e.g., Madkit [Gutknecht97], Swarm [Minar96]) and specific environments for mobile agents (e.g., Aglets [Lange98], Ajanta [Tripathi98]), to more elaborated object-oriented multi-agent frameworks (e.g., Jafmas [Chauhan97], Jafima [Kendall99], BrainstormJ [Zunino00]). Some of them also support visual environments (e.g., Zeus [Nwana99], AgentBuilder [Reticular99]).

These tools provide different agent facilities, and often need a quite extensive tailoring to build agent applications. Nonetheless, the development of MAS is still a technically difficult task. One of the principal barriers is the absence of general agreement about what it is really meant by terms such as agent models, agent architectures and agent frameworks. Moreover, usually the vocabulary used by the agent community for frameworks and architectures is quite different from the one used by the object community. Therefore, we believe it is necessary to conciliate these views, because this misconception clearly affects both the widespread adoption of multi-agent technology and its integration with object-oriented techniques as well. In the following sections we introduce a design approach based on the notion of proto-frameworks, aiming to provide an intermediate stage in the transition from abstract agent models to agent implementations through object-oriented frameworks.

3. Framework Design using Proto-Frameworks

Object-oriented frameworks represent, perhaps, the current most successful technology to achieve both design and code reuse [Fayad97]. The benefits of a framework [Johnson97, Fayad99] are that it provides a general and reusable skeleton of classes and behavior patterns for a given domain, and relying on this support new application can be developed in a flexible and direct way, with additional savings of time and design effort. However, there is a growing feeling in the software community that this technology is not enough to cope with the problem of building highly reusable and adaptable software systems. On one hand, all these benefits need to be enforced during framework design. On the other hand, the influence of quality factors such as flexibility, extensibility, or interoperability, imposes additional conditions to the process. These properties are more inherent to the software itself than to the application domain. Therefore, alternative methods to overcome these problems are becoming more and more important and necessary. Our approach to framework development can be seen not as new method, but as a bridge between architectural styles and object orientation. The problem relies on how to break the tradeoff imposed by a pure functional decomposition versus a pure object-oriented decomposition of a problem. In our opinion, the notion of *architectural materialization* is a key aspect that can lead to the development of better frameworks, but also powerful *proto-frameworks*. These ideas are further explained in subsequent sections.

Architectural materialization

Architectural styles can be used to derive a domain-specific software architecture, that is a software architecture that fits the requirements of a given domain *prescribing* a generic solution for such domain. An architecture derived from architectural styles, however, does not always prescribe an object-oriented computational solution. According to [Shaw96], object orientation is considered as a particular architectural style. The style only indicates the usage of objects as architectural components and messages as connectors. Nonetheless this view, does not recognize the fact that object orientation can be a very convenient technology to implement, or *materialize*, different architectural styles.

By *materialization* we mean the process of producing a concrete computational representation from an abstract description using a given technology. Taking into account this aspect, the proposed approach is based on the notion of *object-oriented materialization* of domain-specific architectures derived from domain models. Our view differs from other approaches to object-oriented architecture design using patterns like [Buschmann96], in that these approaches intend to provide an object-oriented description of the architectural models, which not always can be directly mapped to a particular situation.

Our approach to framework development is depicted in Figure 2. Starting from an abstract domain model and one or more architectural styles (e.g. pipeline, blackboard, hierarchical layers, etc.) adequate for

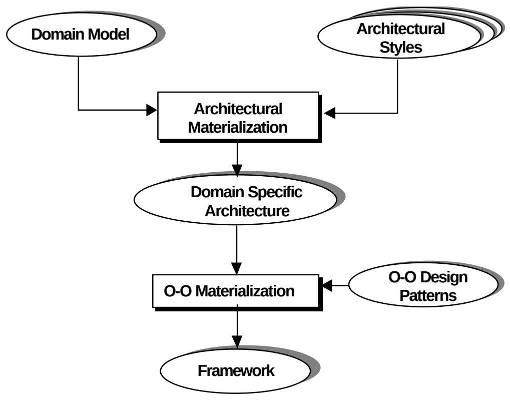


Figure 2. Materialization approach to framework design

the domain, we can obtain a first architectural materialization of the domain. This materialization is driven, essentially, by non-functional requirements that predominate in that domain.

The right formalization of this architecture can bring the possibility of early verification of essential properties that the architecture is supposed to satisfy. After that, the process should continue by refining and translating that organization to an object-oriented world. In such activity, design patterns may be applied to obtain a flexible design that is appropriate for a framework. Simultaneously, specific methods must be defined for representing the application domain control flow. In such process, some points that can not be generalized must be traduced in abstract and hooks methods.

The process of refinement of an architectural model to produce a first architectural design of the application can be obtained through one of the following ways:

- ***Direct Mapping*****:** Application requirements match the constraints prescribed by the pattern, so architecture components can be directly derived from the model.
- ***Constraint Relaxation*****:** Some constraints can be relaxed due to specific characteristics of the application that do not require the complete functionality prescribed by the pattern. In this case, several components can be combined in a single component.
- ***Constraint Strengthening*****:** In some cases, one or more non-functional requirements are more important or crucial than others. Depending on the type of requirement, the resulting mapping varies. In case of adaptability, for example, a given component predicted by the model can be split in several components. In case of efficiency, the predicted functionality of a component can be relaxed to behave more efficiently.

Functionality Attribution and Proto-Frameworks

Normally, the choice of architectural styles gives a set of abstract components and patterns of communication among them. But, depending on the styles, the specific number of components and their function will obviously vary according to the functionality implemented by the framework. In this process, also the interface of each resulting component is defined, as well as the data flow among the components.

This functionality attribution, however, can lead to different frameworks according to the design goals and, particularly, the specific target domain. That is, if we are designing a framework for Enterprise Quality Management Systems (EQM) probably one or more components will be in charge of managing structured documents, for example. A class hierarchy, implementing the different types of documents and the functions applicable to them will probably materialize these documents.

On the other hand, when the target domain is a paradigm of software organization, as MAS for example, the materialization of the architecture can produce an object-oriented framework implementing the mechanisms needed to support the concepts of the paradigm. In this case, there are two alternatives:

- ***Service-oriented frameworks***: The designers can decide to materialize a framework oriented to provide specific functional services for the domain, which can be used by specific applications in a mostly black-box fashion, as is the case of Madkit [Gutknecht97], for example.
- ***Proto-frameworks***: The designers can decide to materialize a framework implementing the essential abstractions of the domain, providing all the infrastructure needed for cooperation and communication of each component type. In this case, the framework provides very abstract hooks to map specific domain components into a class hierarchy in a white-box fashion. This mapping can produce a specific application, but more important yet, it can produce new domain-specific frameworks that adopt the underlying architectural model. In other words, a proto-framework represents an object-oriented materialization of a software architecture derived from non-object-oriented architectural styles.

There is subtle difference between both alternatives. In the first case, the framework does not necessarily provide structured guidance to build, or derive, specific applications or even frameworks for the target domain. In the second case, a proto-framework provides essential abstractions to derive new applications or frameworks by inheritance from the proto-framework classes. This aspect represents an important advantage because the designer has to concentrate on how to map specific entities to the proto-framework abstract entities which provide the essential architectural behavior and how to implement the specific functionality of this framework.

The next section presents the main characteristics of *Bubble*, an object-oriented proto-framework that showed the feasibility of the approach.

4. The Proto-Framework *Bubble*

Bubble is a framework implemented in Java [DiazPace00], originally conceived and designed to build multi-agent systems for the simulation domain. The current design of *Bubble* is the result of several design iterations, essentially due to strong flexibility limitations imposed by the initial fully

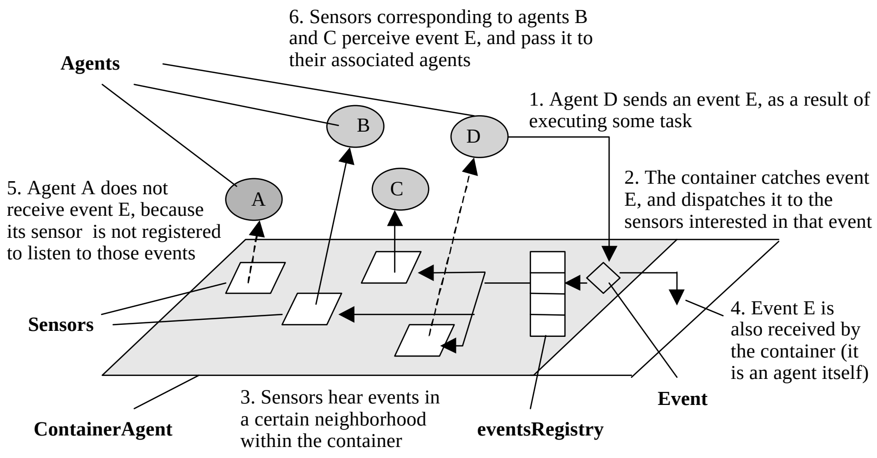

6. Sensors corresponding to agents B and C perceive event E, and pass it to their associated agents
Agents
1. Agent D sends an event E, as a result of executing some task
B
D
C
A
2. The container catches event E, and dispatches it to the sensors interested in that event
5. Agent A does not receive event E, because its sensor is not registered to listen to those events
4. Event E is also received by the container (it is an agent itself)
Sensors
Event
3. Sensors hear events in a certain neighborhood within the container
ContainerAgent
eventsRegistry

Figure 3. Conceptual model of *Bubble´*s architecture

object-oriented approach used for its design. After realizing the problem, we shift our attention to a pure architectural view of the problem, and we chose an implicit invocation style as the main design driver for the architecture, and to model domain entities (agents) separating state from actions using the notion of tasks. The interaction among these agents, that we call bubbles, is performed through events that they produce and receive. Bubble agents are equipped with associated sensors (like filters) that are registered to listen to certain kinds of events with a defined criterion of relevance (local, by group, by event strength, regional, etc.).

The behavior of any bubble agent is defined through tasks using a condition-action style, i.e. a task is a module composed by a series of actions to be executed by the agent (action part) when certain conditions are fulfilled (condition part). Conditions can be related either to the internal state of the agent or the incoming events. The framework also admits bubble agents containing groups of other agents, and tasks composed by groups of predefined tasks. In this way, complex interactions, structures and behaviors can be modeled combining primary blocks.

Figure 3 shows a diagram of the underlying framework architectural model, illustrating a typical event flow between bubble agents in a container and the role that sensors play in this process. Note that the outgoing events produced by an agent D are propagated only if the agent is attached to a container, but this relationship is not compulsory. When a agent receives an incoming event (the agents B, C and container, in the example), the processing depends on the current tasks associated with the agent.

Architectural Views

The *Bubble* architecture can be described from three different views: structural organization, communications, and agent tasks. The next subsections briefly describe each of these views.

- ***Structural organization:*** *Bubble* was designed pursuing the goal of having uniform decomposition. By uniform decomposition [Bass98] we mean the operation of separating a large component into two or more smaller ones, limiting the composition mechanisms to a restricted uniform set. Thus, integration of components and scaling of the system as a whole is achieved having besides modifiability and reusability properties. The aim is to represent the agent organization with a hierarchy of abstraction levels: bubble agents composed by other agents, which in turn are composed by others, and so on. The same structure is used to handle incoming and outgoing events (see next section about communications).
- ***Communication:*** Communications among different components in *Bubble* are performed through events. Every agent can be linked to a container agent, and this container is engaged to collect and dispatch incoming events to the sensors registered inside it. As we explained in a previous section, sensors act like filters and transmit only interesting events to their associated agents. An implicit-invocation mechanism [Shaw96] is used to achieve these notifications. The container agent is in

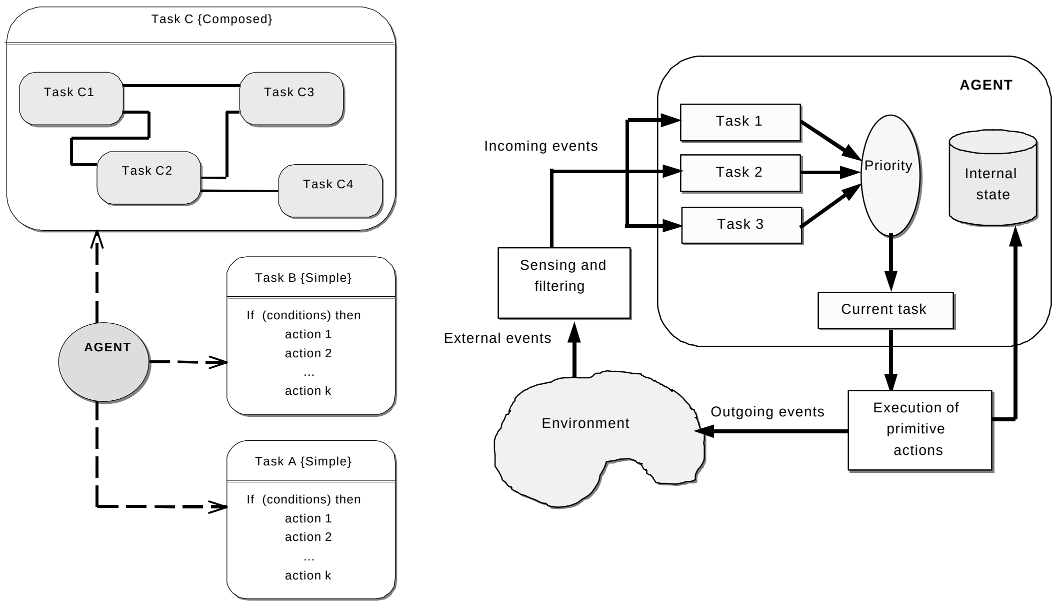


Figure 4. Competing tasks

charge of the event flow management among all the agents. An event represents a notification of any change occurring in the system. Sensors are responsible for reception and conditional transmission (filtering) of events. Container agents deliver events received from the bubble agents to the sensors. To illustrate this interaction, let's suppose we are modeling a market where buyers and sellers are free to perform transactions, any customer interested in buying certain items needs to specify a purchasing criterion and enroll itself with the market to listen to bids. In this context, the market can be a container agent, both sellers and buyers are simple agents, and the specific purchasing criterion corresponds with a sensor. Every time a new offer appears, our customer will be notified about that situation only if the offer fulfills its purchasing conditions.

- ***Behavior:*** All the agents of the framework can perform a set of tasks. A task is composed by one or more procedures with a set of input and output parameters. Tasks are triggered by predefined conditions, which can be related to the internal state of the agents or incoming events. In this way, the agent behavior is conceived as a set of competing tasks, where only one task is active at the same time [Drogoul92]. When a selected task is executed, it can generate either outgoing events and/or changes affecting the agent state. Figure 4 provides a picture of such situation. Different tasks can be dynamically assigned to a agent, and they compete to execute according to their priorities and activation requirements.

Object-Oriented Materialization of the Architecture

The process of materializing the architecture of *Bubble* on an object structure was accomplished in a stepped way, firstly we produced a preliminary design for the architecture and then we refined it including several design patterns [Gamma94] in order to provide more flexibility to the framework.

The initial design comprised the classes mapping the main components of *Bubble*, namely: bubble agents, tasks, sensors and events. The different types of entities defined in the architecture (agents, composed agents and container agents) appeared in the framework as subclass relationships, representing the extension in capabilities (behavior) present in such entities. In this way, a bubble agent models the basic entities of the framework (class *Agent*), a composed agent models a group of entities (class *ComposedAgent*), and a container agent, in turn, represents a set of entities and sensors working together within a container agent (class *ContainerAgent*). Sensors were directly implemented as wrapper objects (class *Sensor*), and their filtering criterion was coded as an abstract method. Finally, tasks were defined as separated

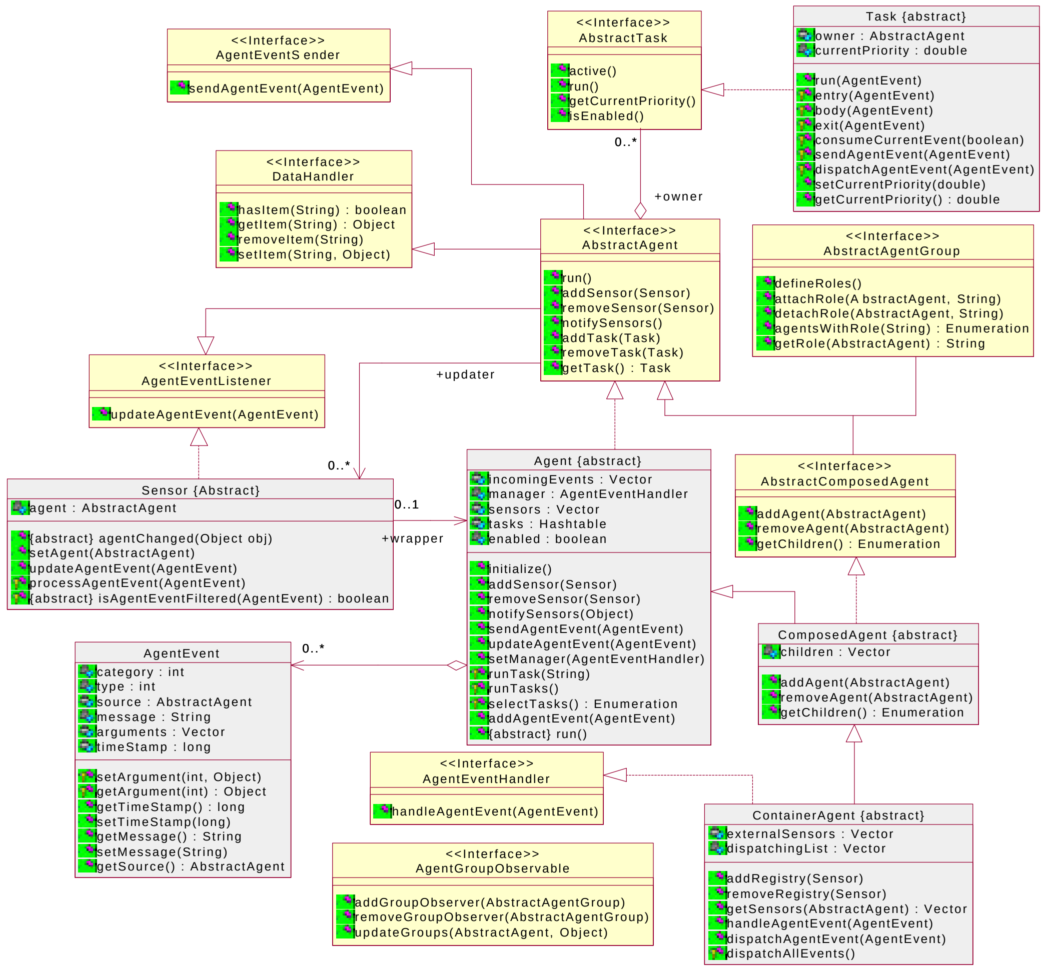


Figure 5. Bubble Class Diagram

objects that can be attached with different entities (class *Task*). Again, both the task body and the conditions to activate it were specified as abstract methods. To execute their tasks, bubble agents have a general implementation of the competing-tasks algorithm using template methods. However, developers are not constrained to this particular style of execution, and they can define other execution mechanisms according to specific needs. Event and implicit invocation features were spread out through agents and sensors. Any common agent defines entry and exit methods for events, whereas container agents implement mechanisms for event registry and dispatching.

Following the design approach, once we outlined a first abstraction of the *Bubble* framework, we proceeded to improve it with a more detailed design. This step involved, for instance, the splitting of behavior in some classes, class refactorizations, and the application of several design patterns. Essentially, the purpose of all these activities was to offer more possibilities of reuse and flexibility at framework level. The following reports on some of the refinements.

Bubble agents were restructured to permit a better separation between behavior interfaces and implementation of these behaviors. Basically, we used Java interfaces to define the abstract behavior of the agents (interfaces *AbstractAgent* and *AbstractComposedAgent*), and also included some concrete implementations of these interfaces (classes *Agent*, *ComposedAgent*, and *ContainerAgent*) that can be used by developers as default components. Additionally, we added a new interface to handle group interactions and roles (interface *AbstractAgentGroup*). In the same way, we split a task into an interface and a concrete implementation (interface *AbstractTask* and class *Task*). Regarding events, we moved this functionality to interfaces (interfaces *AgentEventSender*, *AgentEventListener*, and *AgentEventHandler*), in order to decouple event handling from the entities (agents) manipulating them.

All the agents supported by the framework were structured using a *Composite* pattern, this provides a uniform treatment of these entities favoring the adding/replacement of agents into the system. For example, we can initially have a container with simple agents inside, and then replace certain agents by other composed agents with a low influence on the overall design. We discovered that, sometimes, as agents change their respective sensors need to be updated to reflect these changes. Thus, we introduced an *Observer* pattern between agents and sensors to synchronize changes.

As we mentioned above, the execution of tasks in a given agents follows a competing-tasks scheme. This aspect was modeled using a *Strategy* pattern, so that tasks implement specific behaviors that can be dynamically associated with agents. Regarding tasks themselves, its internal structure follows a *Template Method* pattern to specify the required steps of processing a task should involves. Finally, the management of events in the framework acts like a *Mediator* pattern, this makes easier to define complex interaction protocols between agents and confers enough flexibility to update them.

On the whole, we can say that all these points of variability improved significantly the framework design while preserving the original architectural model. To illustrate these concepts, the following section briefly describes a multi-agent framework development based on the *Bubble* infrastructure

5. Case-Study: A Multi-Agent Framework for Enterprise Quality Systems

InQuality is a framework for Enterprise Quality Management systems (EQM) derived from the proto-framework Bubble, developed by Analyte Lab Information Technology. Essentially, *InQuality* is a framework for building structured document-based applications which is intended to support different types of control quality applications from, for example, ISO9000 compliance document management systems up to Laboratory Information Management Systems (LIMS).

Analyzing the different variants of the domain and having into account the dynamic configurability and flexibility requirements, we concluded that the Bubble architecture could fit very well these requirements and, simultaneously, serve as a guide for the development team to map the required functionality into computational components. Following this idea a first version of the framework was designed in a surprisingly short period.

The design process mainly involved two phases:

- Mapping of the functional specification to Bubble components. This phase involved the decision of what components would be mapped to agents, associated tasks and event types.
- Materialization of functional components through the design of the subclasses that implement the specific functionality of the EQM domain.

After the first iteration the design was evaluated, trying to identify potential bottlenecks and architecture tradeoffs regarding security and efficiency. The final architecture is depicted in Figure 6. Shaded components correspond to Bubble instantiations, whereas non-shaded ones, excepting the blackboard, correspond to normal object models that mainly provide static specifications for the rest of the components. Some components, like the blackboard, were incorporated to satisfy certain domain specifications. This aspect shows how the Bubble-based architecture can be complemented with another architectural style, without loosing the advantages of having an architectural guidance for the rest of the framework design.

Basically, *InQuality* is composed of the following components:

- **Message Server:** The message sever implements a blackboard, accepting messages from both external and internal agents. External agents are agents which may reside outside the application server space, and may be connected via HTTP, to the server. This can be browsers, Instrument Interface Agents and External Application Agents. Taking into account that many messages are addressed to specific users, the blackboard is able to maintain a persistent version of messages until the involved users log into the system. This capability is especially important for the workflow component.

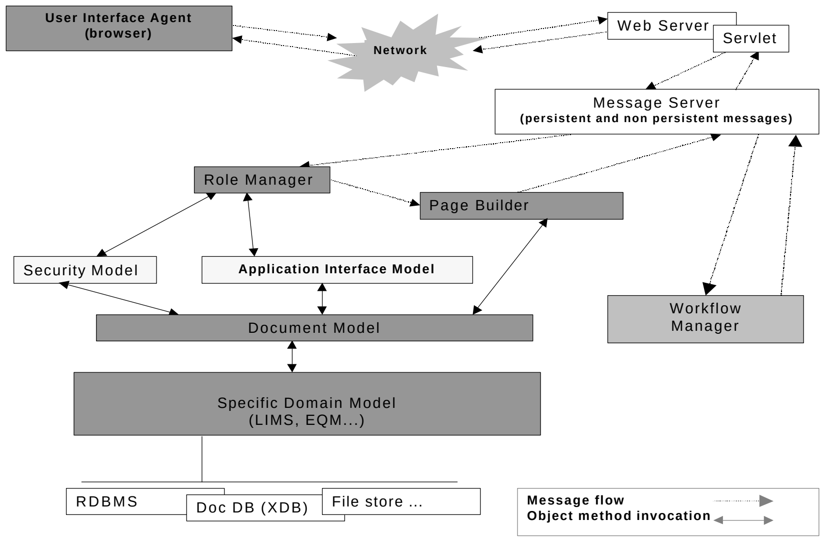


Figure 6. Architectural Scheme of InQuality

- **Role Manager:** Every user of an *InQuality* system is represented by a Bubble agent. Tasks associated with these agents determine the allowed behavior for the user depending on his/her *role*. These capabilities are represented by a role model, which defines the different rights of every type of user. Roles are also represented by agents able to dynamically activate allowed tasks to every new logged user. This is uniformly done by subscribing to events that may be of their interest and acting according to certain rules that determine the activation of a given task. These can be security rules, such as allowing a particular user to edit some sample information if the sample belongs to certain area of the plant or is in a certain state. It uses the Security Model to access the rules and actions allowed for a particular Role and Participant.

- **Page Builder:** This agent is responsible for converting documents, frames and interface objects to a representation acceptable for a given browser (such as HTML or XML+XSL). It isolates the Application Interface Model from the details required to convert to the different HTM dialects.

- **Event Notifier:** Participants and Roles, and other agents can decide that they are interested in certain messages (events) and want to be notified if (and only if) certain (user-defined) rule is met. These rules are expressed in a scripting language and evaluated by the Notification component, which then notifies the interested agent. So, we can say that this component acts as an event filter.

- **Clock and Scheduler:** It is usual for documents, tests, samples and activities to have time limits imposed on them. For example, if a particular sample has not been collected before an hour of its specified collection time, it will not be collected at all. A certain workflow activity, such as approving an ISO 9000 document for distribution, may have a given deadline. Agents will register this deadline with the Clock component, and will ask this component to send them a particular event when the time arrives.

- **WorkFlow Manager:** Almost every enterprise framework heavily relies on a workflow management system [Fayad00], and also this is the case of *InQuality*. Different documents, samples, test types, etc., may require different collection, processing, revision and approval cycles. This means they will have different associated workflows. This component is responsible for initiating, tracking and ending each of this workflow instances, also notifying *roles* and *participants* when they should perform a certain activity. This component represents an interesting application of Bubble's facilities, and it will be described in detail in the next section.

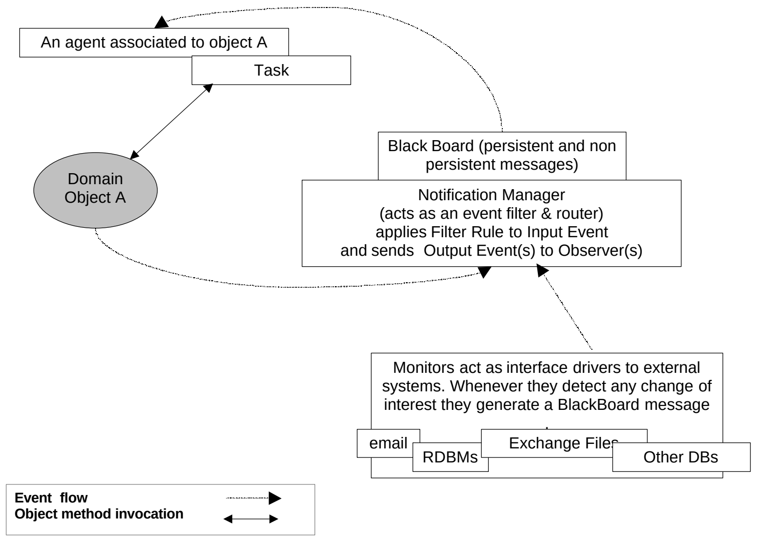

An agent associated to object A
Task
Domain Object A
Black Board (persistent and non persistent messages)
Notification Manager (acts as an event filter & router) applies Filter Rule to Input Event and sends Output Event(s) to Observer(s)
Monitors act as interface drivers to external systems. Whenever they detect any change of interest they generate a BlackBoard message
email
RDBMs
Exchange Files
Other DBs
Event flow
Object method invocation

Figure 7. *InQuality* mapping of domain objects and agents

- **The Roles & Security Model:** This is an object model which acts as a repository used by the Role manager and the system configuration tools to define and get/set security attributes on the user interface and specific domain objects. Generally speaking, each application document and attribute may have security rules imposed for each role or user.

- **The Application Interface Model:** This object model acts as a repository for all user interface information associated to a particular application, such as menus, frames, topic frames, button frames, etc. The Page Builder component uses this information to translate entities in this model to a browser dependent implementation. Generally speaking, it defines a particular application and the way it modifies objects in the specific domain model.

- **The Specific Domain Model:** This is the "real" object model, associated to a particular domain, such as the Lab Information System (LIMS), or the Entreprise Quality Management System (EQM). The objects modeled here are in fact "passive" wrappers for the database objects. Certain "business rules" and "state change" rules may also be included here. We refer to the objects belonging to this component as passive, because an associated Bubble agent manages the object's entire autonomous behavior (see Figure 7). In this way, a better separation of concerns is achieved leaving for the agent the dynamic control of the object behavior through Bubble tasks [1].

The Workflow and Role Manager Components

In this section we describe in more detail the design of the workflow component of *InQuality*. This component is an example of how the mapping to the Bubble architecture simplified the implementation of a complex component for workflow management and how it relates to the role manager.

Usually, a workflow represents a graph constituted by a set of activities (nodes) and paths (arcs) connecting these activities, related to the specific tasks to be developed in a work process over a product. In many cases, workflow activities are modeled using some variants of Petri-nets, and they are enacted by a workflow engine. Under this approach, this engine becomes, almost unavoidably, in one of the main functional components of an enterprise framework. *InQuality*, however, uses a different approach.

[1] Details of this core component can not be described in detail due to academic-industrial confidentiality agreements.

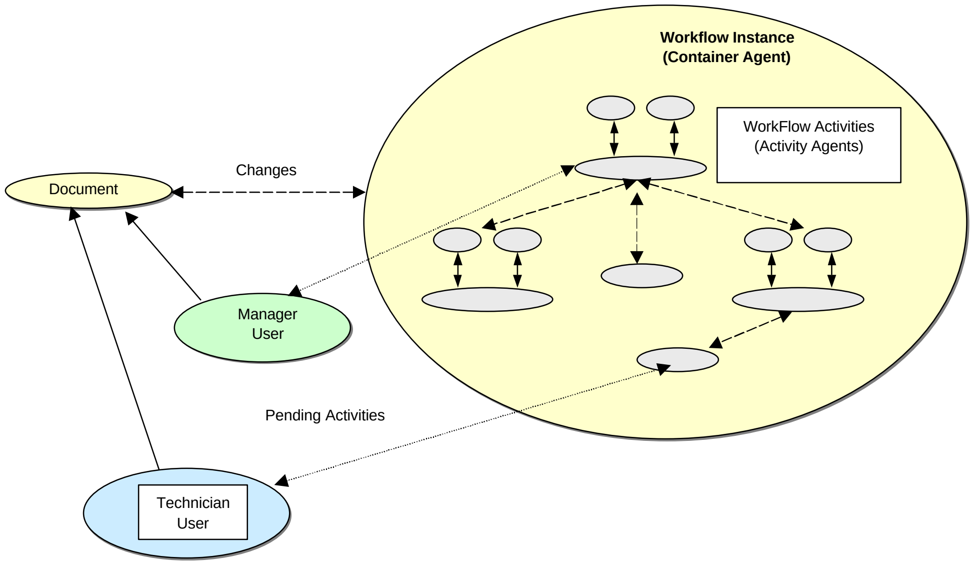


Figure 8. Relationship among Workflow Activities, Roles and Participants and Documents

In the *InQuality* case each document type has an associated workflow. Each workflow is produced through a Petri-net-based graphical notation using a graphical editor. The editor produces an specification of the net that will be used to configure the specific workflow instance. But, instead of having a centralized workflow engine, every particular workflow instance (or workflow process) associated with a document is represented as a container agent. Each activity within the workflow instance is represented as a single agent (see Figure 8).

The workflow instance agent encapsulates the knowledge related to the structure of the particular workflow and to create the corresponding agents and associated tasks that will implement the workflow. Activity agents listen to events generated by the execution of previous activities and reacts when these events arrive. Tasks associated with activity nodes of the workflow will produce messages directed, through the blackboard component, to the specific role or participant agent in charge of executing such activity.

When the "end of activity" event arrives, the corresponding activity agent evaluates each of the rules associated with each path originated from it, and generates the corresponding events that enable the next activities in the workflow. There are two workflow-specific cases of activity agents, AND activities and OR activities, which are used to coordinate the workflow process.

An AND activity agent waits until all the events coming from its upstream nodes have arrived (that is, the events sent by tasks T1 and T2) and then announces its own event to activate the next task (T3). In this way it acts as a synchronization node, not allowing task T3 to start until both tasks T1 and T2 are completed. The OR agent announces its event if any of its registered event has arrived, and ignores all other incoming events. So, if either task T1 or task T2 is completed, then task T3 will be started. By ignoring all other events after the first one, it can assure the next task (T3) is not started twice. These situations are shown in Figure 9.

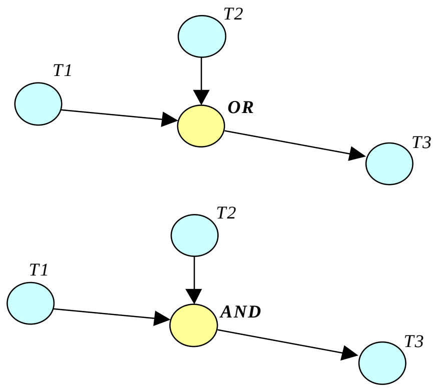


Figure 9. AND and OR activity agents

This scheme greatly simplifies the implementation of the workflow management component by mapping it to the simple conceptual model of agents and tasks. The dynamic nature of the task scheme makes easy to configure a Participant agent with the tasks that implement the activities it must execute. The Application Interface component is in charge of customizing the user interface of each specific participant to show the activities it is expected to accomplish, by simply asking the Participant Agent about its current tasks.

This organization is also used to control the access to system function for each user according to its role. In this way, a given user can only see authorized functions, which are mapped from the interface to the task that implement the selected function. For example, some type of user can only edit some parts of a document or other can only consult a document.

6. Lessons Learned

From the development of *InQuality*, we collected several experiences that are currently driving the development of new versions of the framework. One of the lesson learned was that, besides the fact that framework design is a difficult task, the design of a complex enterprise framework requires more than domain expertise. This kind of frameworks can be so complex that they require the capacity of combining many computational factors that transcend the domain itself. In this sense, to have a clear architectural guidance, as the one provided by the *Bubble* architecture, became a stronger key than to have an elegant but complex framework from which start the development.

The transition to object-oriented development, as several renowned authors have highlighted, requires more than just to know about object-oriented programming and design concepts [Goldberg95, Fayad98]. The provision of technology-independent architectural guidance to organize the development is almost indispensable. This last fact impacts positively in providing a smooth technology transition for a relatively inexperienced development team. In this case, the proto-framework concept represents an additional help to facilitate the reuse of object-oriented architectural materialization.

As regards the framework, there are good reasons to state the appropriateness of *Bubble* to assist multi-agent development. This can be traced from the intrinsic characteristics of MAS . As it is argued by [Jennings99], it is natural to modularize complex systems in terms of a number of related sub-systems organized in a hierarchical fashion, where the components of each sub-system work together to deliver a given functionality. As these systems get more complex, it is usually impossible to know about all the potential links in advance. The unpredictable nature of the relationships between components makes it difficult to deal with these kind of systems using just conventional object-oriented design techniques. In such cases, the policy of deferring to runtime decisions about component interactions provides a more realistic view. The power of agent organizations comes from the ability of agents to join groups, and by doing so, to acquire new abilities that they would have not obtained otherwise [Gutknecht97] All the aforementioned concepts are somehow reflected in the design of *Bubble*. Moreover, the framework also extends this uncoupling to behavioral issues in the form of tasks. In this way, problems associated with the coupling of components are reduced and adaptability is improved. Furthermore, it promotes delegation models, that is a component (agent) can generate requests for assistance to other components if it is asked to carry out some specific task it cannot solve by its own means.

7. Conclusions

In this paper, we have reported on our experience with the design of frameworks, and particularly the development of multi-agent frameworks. In this context, we introduced a new approach based on the notions of architectural materialization and proto-frameworks. We believe that the benefits of the approach are twofold. On one hand, it provides a smoother transition between architectural styles and application frameworks by inserting an intermediate stage of architectural materialization. On the other hand, and perhaps more significantly, the notion of proto-frameworks rises a new challenge regarding multi-agent development.

A proto-framework makes explicit certain essential architectural choices by means of object-oriented constructs, which can serve as basis for the development of traditional frameworks. Moreover, the tradeoffs between non-functional requirements selected by the framework developers can determine different alternatives of building frameworks on top of proto-frameworks, and these alternatives can be evaluated using the techniques inherited from the architecture-driven design approach.

We have also presented an example of a proto-framework, named *Bubble*, which relies mainly on an architecture composed by modular entities, associated tasks and events. According to the proposed approach, the initial conceptual architecture was mapped to an object-oriented framework and empowered to fulfill reusability and adaptability requirements. The resulting framework (i.e., the proto-framework) retains the benefits of the original architectural model. The implementation of an EQM framework on top of *Bubble* illustrates how new application frameworks can be derived from proto-frameworks, and also provided a valuable experience about industrial application of our architectural materialization approach. Another important aspect of the work was that the architectural guidance provided by *Bubble* much helped a team of developers with few or no experience in object-oriented design to produce an enterprise multi-agent framework that can be adapted to a broad range of applications.

Finally, this work hopes to contribute to the still incipient field of agent methodologies and design of MAS. We think that the ideas and results presented are worth to be deeply explored in order to obtain quantitative evidence of the benefits of combining object-oriented techniques and multi-agent systems.

Acknowledgements

The authors whish to thank Edgardo Belloni and Alfredo Teysyere for their efforts in the development of the first academic applications of the approach, and also to Alejandro Zunino for his valuable suggestions regarding multi-agent issues. Likewise, they are grateful to Mario Zito and the Analyte development team for their support and patience, during the project, as well as the critic enthusiasm that helped too much to improve the final results.

References

1. [Bass98] *Software Architecture in Practice*. L. Bass, P. Clement, and R. Kazman. Addison-Wesley. 1998
2. [Bradshaw97] *Software Agents.* J. Bradshaw. AAAI Press, Menlo Park, USA, 1997.
3. [Burmeister96]*Models and Methodologies for Agent-Oriented Design and Analysis*. B. Burmeister. In Working Notes of the KI´96 Workshop on Agent-Oriented Programming and Distributed Systems. K. Fisher (Ed.), 1996.
4. [Buschmann96] *Pattern-Oriented Software Architecture. A System of Patterns*. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal John Wiley & Sons. 1996.
5. [Campo01] *Developing Object-Oriented Enterprise Quality Frameworks using Proto-Frameworks.* A. Díaz Pace, M. Campo and M. Zito. Technical Report RR001-2001, ISISTAN Research Institute, Universidad Nacional del Centro de la Provincia de Buenos Aires. February, 2001.
6. [Chauhan97] *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation*. D. Chauhan. PhD thesis, ECECS Department, University of Cincinnati, 1997.
7. [Demazeau91] *From Reactive to Intentional Agents.* In Decentralized Artificial Intelligence 2, Y. Demazeau and J. P. Müller (Eds.), pp. 3-14. Elsevier/North-Holland, Amsterdam, 1991.
8. [DiazPace00] *Bubble: A Framework for Simulation of Collective Processes*. A. Diaz Pace., F. Trilnik, A. Clausse, and M. Campo. Technical Report, Isistan Research Institute, Faculty of Sciences, Universidad Nacional del Centro de la Provincia de Buenos Aires. 2000. (In Spanish).
9. [Drogoul92] *Multi-agent Simulation as Tool for Modeling Societies: Applications to Social Differentiation in Ant Colonies*. A. Drogoul and J. Ferber. In Proceedings of the 4th European Workshop on Modeling Autonomous Agent and Multi-agent World. Rome (Italy), 1992.
10. [Drogoul98] *Applying an Agent-Oriented Methodology to the Design of Artificial Organizations: A Case-Study in Robotic Soccer*. A. Drogoul and A. Collinot. Autonomous Agents and Multi-Agent Systems, vol.1, 1, 113-129. 1998.
11. [Fayad97] *Object-Oriented Application Frameworks.* M. Fayad and D. Schmidt. Communications of ACM, Vol. 40, No. 10, pp. 32-38, October 1997
12. [Fayad98] *Transition to Object-Oriented Software Development.* M. Fayad and M. Laitinen. John Wiley & Sons. 1998.
13. [Fayad99] *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. M. Fayad, D. Schmidt, and R. Johnson. Wiley Eds. 1999.
14. [Fayad00] *Enterprise Frameworks Characteristics, Criteria and Challenges*. M.

Fayad, D. Hamu, D. Brugali. Comunications of the ACM, Vol. 43, NO. 10, October 2000.

15. [Finin97] *KQML as an Agent Communication Language*. T. Finin, Y. Labrou, and J. Mayfield. In Software Agents, AAAI Press, Menlo Park, USA, 1997.

16. [Gamma94] *Design Patterns, Elements of Reusable Object-Oriented Software*. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Addison-Wesley. Massachussetts, 1994.

17. [Garijo99] *Multi-Agent System Engineering.* F. Garijo and M. Boman (Eds.). Proceedings 9th European Workshop on Modeling Autonomous Agents in a Multi-Agent World, MAAMAW´99. Valencia, Spain. June, 1999.

18. [Goldberg95] *Succeeding With Objects: Decision Frameworks for Project Management*. A. Goldberg and K. Rubin. Addison-Wesley. 1995

19. [Gutknecht97] *Madkit: Organizing Heterogeneity with Groups in a Platform for Multiple Multi-Agent Systems.* O. Gutknecht and J. Ferber. Technical Report RR97188, Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier, University Montpellier II C 09928, France. December, 1997.

20. [Jennings99] *Agent-Oriented Software Engineering.* N. Jennings. Proceedings of MAAMAW´99, Valencia, Spain. June, 1999.

21. [Jonhson97] *Frameworks = (components + patterns)*. R. Johnson. Communications of the ACM, theme issue on “Object-oriented application frameworks”, M. Fayad and D. Schmidt (Eds.), 40(10) pp. 39-42. 1997

22. [Kendall99] *A Framework for Agent Systems.* E. Kendall, P. Krishna, C. Pathak, and C. Suresh.. In Implementing Applications Frameworks: Object Oriented Frameworks atWork, M. Fayad, D. C. Schmidt, and R. Johnson, Eds. Wiley & Sons, 1999.

23. [Lange98] *Programming and Deploying Mobile Agents with Java Aglets.* D. Lange and M. Oshima. Addison-Wesley, Reading, MA, USA, September 1998.

24. [Lesser98] *Reflections on the Nature of Multi-Agent Coordination.* V. Lesser. Journal of Autonomous Agents and Multi-Agent Systems, 1(1), pp. 89-111, 1998.

25. [Minar96] *The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations.* N. Minar, R. Burkhart, C Langton and M. Askenazi. Internal Research Report. June 1996.

26. [Nilsson96] *Introduction to Machine Learning*. N. Nilsson. Draft. September 1996.

27. [Nwana99] *A Toolkit for Building Distributed Multi-Agent Systems.* H. Nwana, D. Ndumu, L. Lee and J. Collins. Applied Artifical Intelligence Journal 13, 1, 129–186. 1999.

28. [Rao90] *Deliberation and the Formation of Intentions*. A. Rao and M. Georfeff. Technical Report 10. Australian AI Institute, Carlton, Australia, 1990.

29. [Reticular99] *AgentBuilder: An integrated toolkit for constructing intelligent software agents*. Reticular Systems Inc. White Paper, February 1999.

30. [Shaw96] *Software Architecture, Perspectives on an Emerging Discipline*. M. Shaw and D. Garlan. Published by Prentice-Hall. 1996.

31. [Sycara98] *The many faces of agents.* K. Sycara. Artificial Intelligence Magazine, Vol. 19 Nro 2. 1998.

32. [Tripathi98] *Ajanta- A System for Mobile-Agent Programming.* A. Tripathi, N. Karnik, M. Vora and T. Ahmed. Technical Report, Department of Computer Science, University of Minnesota, Number TR98-016, 1998.

33. [Weiss99] *Multi-Agent Systems, a Modern Approach to Distributed Artificial Intelligence*. Edited by G. Weiss. MIT Press. 1999.

34. [Wooldridge96] *Software Agents.* M. Wooldridge and N. Jennings. IEEE Review 17-20. January, 1996.

35. [Wooldridge97] *Agent-based Software Engineering*. M. Wooldridge. Technical Report. September 1997.

36. [Wooldridge00] *The Gaia Methodology for Agent-Oriented Analysis and Design*. M. Wooldridge, N. Jennings and D. Kinny. Autonomous Agents and Multi-Agent Systems, vol.3, 3, 285-312. 2000.

37. [Zunino00] *Brainstorm/J: A Framework for Intelligent Agents*. A. Zunino. Master's Degree Dissertation. Universidad Nacional del Centro, Instituto de Sistemas ISISTAN. April 2000. (In Spanish).