Chapter IV

From Requirements to Code with the PASSI Methodology

Massimo Cossentino
ICAR-CNR, Italy

Abstract

A Process for Agent Societies Specification and Implementation (PASSI) is a step-by-step requirement-to-code methodology for designing and developing multi-agent societies, integrating design models and concepts from both object-oriented (OO) software engineering and artificial intelligence approaches using the UML notation. The models and phases of PASSI encompass representation of system requirements, social viewpoint, solution architecture, code production and reuse, and deployment configuration supporting mobility of agents. The methodology is illustrated by the well-known Bookstore case study.

Introduction

At present, several methods and representations for agent-based systems have been proposed (Aridor & Lange, 1998; Bernon, Camps, Gleizes, & Picard, 2004; Bresciani, Giorgini, Giunchiglia, Mylopoulos, & Perini, 2004; DeLoach & Wood,

2001; Jennings, 2000; Kendall, Krishna, Pathak, & Suresh, 1998; Zambonelli, Jennings, & Wooldridge, 2001, 2003). In order to explore them, we shall consider a relevant aspect in modelling software, that is, fidelity. Robbins, Medvidovic, Redmiles, and Rosenblum (1998) have defined fidelity as the distance between a model and its implementation. This means that low fidelity models are problem-oriented, while high fidelity models are more solution-oriented.

Since agents are still a forefront issue, some researchers have proposed methods involving abstractions of social phenomena and knowledge (Bernon et al., 2004; Bresciani et al., 2004; Jennings, 2000; Zambonelli, Jennings, & Wooldridge, 2001, 2003) (low-fidelity models); others have proposed representations involving implementation matters (Aridor & Lange, 1998; DeLoach & Wood, & Sparkman, 2001; Kendall et al., 1998) (higher fidelity models).

There exists one response to these proposals, which is to treat agent-based systems the same as non-agent based ones. However, we reject this idea because we think it is more natural to describe agents using a psychological and social language. Therefore, we believe that there is a need for specific methods or representations tailored for agent-based software. This belief originates from the related literature. To give an example, Yu and Liu (2000) say that "an agent is an actor with concrete, physical manifestations, such as a human individual. An agent has dependencies that apply regardless of what role he/she/it happens to be playing." On the other hand, Jennings (2000) defines an agent as "an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives." Also, Wooldridge and Ciancarini (2001) see the agent as a system that enjoys autonomy, reactivity, pro-activeness, and social ability.

Therefore, multi-agent systems (MAS) differ from non-agent based ones because agents are meant to be autonomous elements of intelligent functionality. Consequently, this requires that agent-based software engineering methods encompass standard design activities and representations as well as models of the agent society.

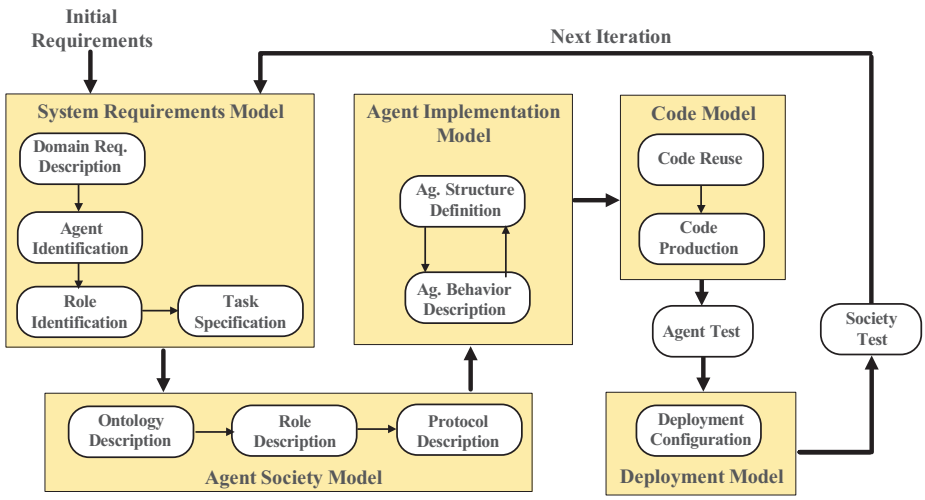
Two more responses exist. They both argue that agents differ from other software but disagree about the differences. The first, proposed by supporters of low-fidelity representations, is that agents are distinguished by their social and epistemological properties, only these need different abstractions. The second, proposed by supporters of high-fidelity representations, is that the difference is in the deployment and interaction mechanisms. With regard to the agent notion, DeLoach, Wood, and Sparkman (2001) argue that "an agent class is a template for a type of agent in the system and is analogous to an object class in object-orientation. An agent is an actual instance of an agent class," and "… agent classes are defined in terms of the roles they will play and the conversations in which they must participate." This definition in some way conjugates the social-

(conversational) and deployment- (implementation) oriented theories and positions DeLoach, Wood, and Sparkman in the middle.

We also reject these two views in their extreme forms. A designer may want to work at different levels of detail when modeling a system. This requires appropriate representations at all levels of detail or fidelity and, crucially, systematic mappings between them. Because such issues are, at present, not addressed by any of the existing MAS analysis and design methodologies, we have decided to create a brand new one.

The methodology we are going to illustrate is named a Process for Agent Societies Specification and Implementation (PASSI) or "steps" in the Italian language. It is our attempt at solving the scientific problem arising from the above considerations. In fact, it is a step-by-step requirement-to-code methodology for designing and developing multi-agent societies integrating design models and concepts from both object-oriented (OO) software engineering and MAS, using the Unified Modeling Language (UML) notation. It is closer to the argument made above for high-fidelity representations, but addresses the systematic mapping between levels of detail and fidelity. The target environment we have chosen is the standard, widely implemented Foundation for Intelligent Physical Agents (FIPA) architecture (O'Brien & Nicol, 1998; Poslad, Buckle, & Hadingham, 2000). PASSI is the result of a long period of theoretical studies and experiments in the development of embedded robotics applications (Chella, Cossentino, & LoFaso, 2000; Cossentino, Sabatucci, & Chella, 2003).

Figure 1. The models and phases of the PASSI methodology



The remainder of this chapter is structured as follows. The next section gives a quick presentation of the methodology's models and provides a justification for PASSI. The third section presents the application of PASSI to the "Juul Møller Bokhandel A/S" case study (Andersen, 1997), giving a detailed description of the steps and the use of UML notations within each of them. A comparison of PASSI with the Gaia (Zambonelli, Jennings, & Wooldridge, 2003) and MaSE (DeLoach, Wood, & Sparkman, 2001) is then given, and some conclusions are presented in the final section.

A Quick Overview of the PASSI Methodology

In conceiving this design methodology, we followed one specific guideline: the use of standards whenever possible. This justifies the use of UML as modeling language, the use of the FIPA architecture for the implementation of our agents, and the use of XML in order to represent the knowledge exchanged by the agents in their messages.

PASSI (Process for Agent Societies Specification and Implementation) is a step-by-step requirement-to-code methodology for developing multi-agent software that integrates design models and philosophies from both object-oriented software engineering and MAS using (more properly extending) the UML notation (OMG, 2003b). Because of the specific needs of agent design, the UML semantics and notation will be used as reference points, but they will be extended, and UML diagrams will be often used to represent concepts that are not considered in UML and/or the notation will be modified to better represent what should be modeled in the specific artifact. The PASSI process is composed of five process components: System Requirements, Agent Society, Agent Implementation, Code, and Deployment, and several distinct work definitions within each of them (Figure 1). Code production is strongly supported by the automatic generation of a large amount of code thanks to the PASSI ToolKit (PTK) used to design the system and a library of reusable patterns of code and pieces of design managed by the AgentFactory application.

In what follows, the five process components will be referred to as models and the work definitions as phases; in order to clarify the meaning of these terms, we will provide a parallelism with the Software Process Engineering Metamodel (SPEM) concepts (SPEM, 2002). Referring to SPEM, we could say that a process is composed of process components; each process component could be made by phases (a kind of work definition) that are in turn decomposable into activities and steps (both activities and steps are again work definitions). In the

PASSI process, the element that corresponds to the SPEM process component is called model, and it is composed of phases (for instance, in Figure 1, we can see that the System Requirements model is composed of the Domain Requirements Description, Agents Identification, … phases). The "models" and "phases" of PASSI are:

1. **System Requirements Model:** a model of the system requirements in terms of agency and purpose. It is composed of four phases:
   (a) Domain Requirements Description (D.R.D.): a functional description of the system using conventional use case diagrams;
   (b) Agent Identification (A.Id.): the phase of attribution of responsibilities to agents, represented as stereotyped UML packages;
   (c) Role Identification (R.Id.): a series of sequence diagrams exploring the responsibilities of each agent through role-specific scenarios; and
   (d) Task Specification (T.Sp.): specification of the capabilities of each agent with activity diagrams.
2. **Agent Society Model**: a model of the social interactions and dependencies among the agents involved in the solution. Developing this model involves three steps:
   (a) Ontology Description (O.D.): use of class diagrams and OCL constraints to describe the knowledge ascribed to individual agents and their communications;
   (b) Role Description (R.D.): class diagrams are used to show the roles played by agents, the tasks involved, communication capabilities, and inter-agent dependencies; and
   (c) Protocol Description (P.D.): use of sequence diagrams to specify the grammar of each pragmatic communication protocol in terms of speech-act performatives.
3. **Agent Implementation Model**: a classical model of the solution architecture in terms of classes and methods; the most important difference with the common object-oriented approach is that we have two different levels of abstraction, the social (multi-agent) level and the single-agent level. This model is composed of the following steps:
   (a) Agent Structure Definition (A.S.D.): conventional class diagrams describe the structure of solution agent classes; and
   (b) Agent Behavior Description (A.B.D.): activity diagrams or statecharts describe the behavior of individual agents.
4. **Code Model**: a model of the solution at the code level requiring the following steps to produce it:

(a) generation of code from the model using one of the functionalities of the PASSI add-in. It is possible to generate not only the skeletons but also largely reusable parts of the method's implementation based on a library of reused patterns and associated design descriptions; and

(b) manual completion of the source code.

5. **Deployment Model**: a model of the distribution of the parts of the system across hardware processing units and their migration between processing units. It involves one step: Deployment Configuration (D.C.): deployment diagrams describe the allocation of agents to the available processing units and any constraints on migration and mobility.

   Testing: the testing activity has been divided into two different steps: the single-agent test is devoted to verifying the behavior of each agent regarding the original requirements for the system solved by the specific agent (Caire, Cossentino, Negri, Poggi, & Turci, 2004). During the Society Test, integration verification is carried out together with the validation of the overall results of this iteration. The Agent Test is performed on the single agent before the deployment phase, while the society test is carried out on the complete system after its deployment.

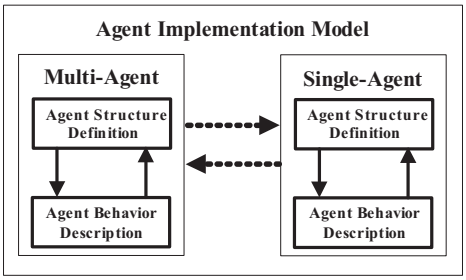
In the following, each of the above cited models will be discussed in details in a specific subsection.

The Agent in PASSI

The concept of agent will be central to our discussion and therefore a definition of what we mean by an agent will be helpful before proceedings.

In PASSI, we consider two different aspects of the agent: during the initial steps of the design, it is seen as an autonomous entity capable of pursuing an objective through its autonomous decisions, actions, and social relationships. This helps in preparing a solution that is later implemented, referring to the agent as a significant software unit. An agent may undertake several functional roles during interactions with other agents to achieve its goals. A role is a collection of tasks performed by the agent in pursuing a sub-goal or offering some service to the other members of the society. A task, in turn, is defined as a purposeful unit of individual or interactive behavior. Each agent has a representation of the world in terms of an ontology that is also referred to in all the messages that the agents exchange.

Figure 2. The agents' implementation iterations



Iterations

PASSI is iterative, as are most widely accepted object-oriented methods. There occur two types of iterations in it. The first one is led by new requirements and involves all the PASSI models.

The second iteration occurs, involving only modifications to the Agent Implementation Model. It is characterized by a double level of iteration (see Figure 2). We need to look at this model as characterized by two views: the multi-agent and single-agent views. The outer level of iteration (dashed arrows) concerns the dependencies between multi-agent and single-agent views. The first (multi-agent) view relates to the agents' structure (in terms of cooperation and tasks involved) and behaviors (flows of events depicting cooperation). The second one instead relates to the single-agent structure (attributes, methods, inner classes) and behavior (specified in an appropriate way). The inner level of iteration (Agent Structure Definition – Agent Behavior Description) takes place in both the multi-agent and single-agent views and concerns the dependencies between structural and behavioral matters.

As a consequence of this double level of iteration, the Agent Implementation Model is composed of two steps (A.S.D. and A.B.D.) but yields four kinds of diagrams, taking into account the multi- and the single-agent views.

A More Detailed Description of PASSI

Throughout the following subsections, we refer to the "Juul Møller Bokhandel A/S" Case Study (Andersen, 1997) that describes the problems of a small bookstore coping with rapidly expanding Internet-based book retailers. The bookstore has a strong business relationship with the Norwegian School of

Management. Nevertheless, there are communication gaps between them. As a consequence, the bookseller is in trouble, for example, when pricing the books (due to a lack of information about the number of attendees of some courses) or when the School changes the required literature. In addition, there are problems with the distribution chain. This requires a strong knowledge of distributors' and publishers' processes and practices.

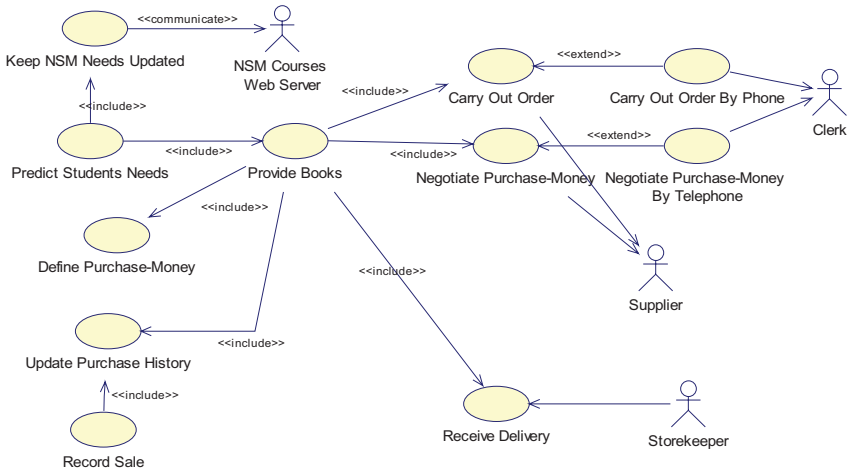
Domain Requirements Description Phase

Although many authors make use of goals in requirements engineering (Antón & Potts, 1998, Potts, 1999), we prefer the approach coming from Jacobson, Christerson, Jonsson, and Overgaard (1992), and we describe requirements in terms of use case diagrams. The Domain Requirements Description Phase, as a result, is a functional description of the system composed of a hierarchical series of use case diagrams. Scenarios of the detailed use case diagrams are then explained using sequence diagrams. Figure 3 shows part of the Domain Requirements Description diagram depicting our analysis for the bookstore case study. Stereotypes used here come from the UML standard.

Throughout this chapter, we will only examine one scenario—the one that takes place every time that the bookstore needs to purchase some books (*Provide Books* use case in Figure 3). This may happen, for example, before the beginning of every semester, so as to provide the store with the requested books and therefore anticipate the students' needs; or when some faculty has changed the required literature or switched a book from "recommended" to "required." The scenario begins with the prediction of the students' needs in order to establish whether there is a sufficient number of copies of that book in the store or not. If not, and if the book is needed, a new purchase must be made; this in turn includes (see Figure 3):

- Definition of the desired quotation (*Define Purchase-Money* use case) by the use of an expert system that holds the history of previous purchases, especially with regard to courses, teachers, number of attendees, books purchased and books sold, suppliers, time elapsed for negotiation and delivery, and so forth.
- Negotiation of the price (*Negotiate Purchase-Money* use case).
- Execution of the order (*Carry Out Order*).
- Updating of the purchase history archive (*Update Purchase History*) in order to increase the knowledge of the purchase expert system.
- Receiving delivery information about the purchase (*Receive Delivery*) in order to close the case related to it.

Figure 3. A portion of domain requirements description diagram



Agent Identification Phase

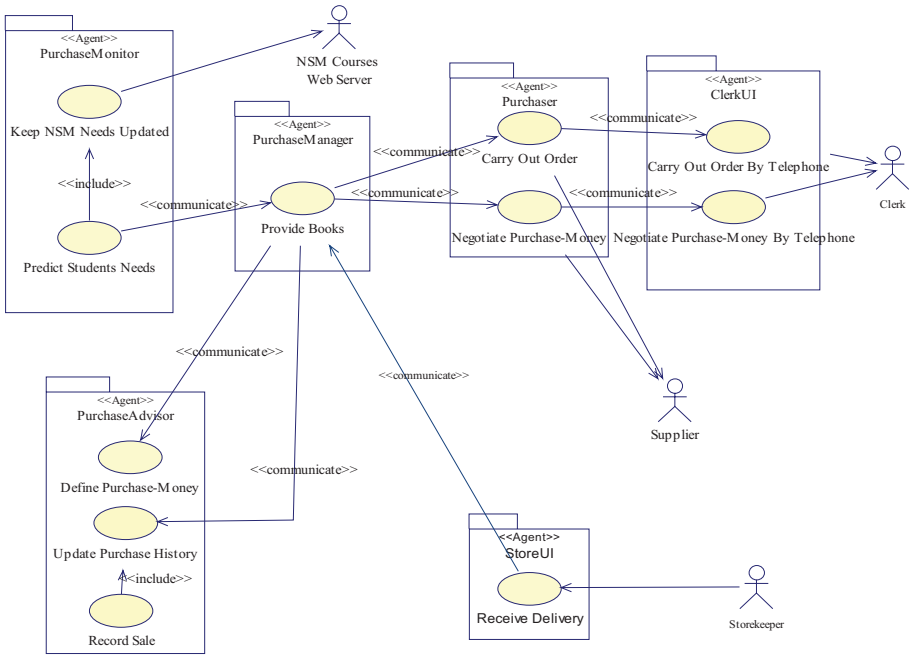
If we look at a MAS as a heterogeneous society of intended and existent agents that in Jackson's terminology can be "bidden" or influenced but not deterministically controlled (Jackson, 2001), it is more reasonable to locate required behaviors into units of responsibility from the start. That is why we have put this phase in the System Requirements Model.

Agents' identification starts from the use case diagrams of the previous step. Using our definition of agent, it is possible to see identification as a use case or a package of use cases in the functional decomposition of the previous phase. Starting from a sufficiently detailed diagram of the system functionalities (Figure 3), we group one or more use cases into stereotyped packages so as to form a new diagram (Figure 4). In so doing, each package defines the functionalities of a specific agent.

Relationships between use cases of the same agent follow the usual UML syntax and stereotypes (see the "include" relationships in the *Purchase Monitor* and *Purchase Advisor* agents in Figure 4), while relationships between use cases of different agents are stereotyped as "communicate."

The convention adopted for this diagram is to direct communication relationships between agents from the initiator towards the participant.

Figure 4. The agents identification diagram obtained from the requirements described in the previous phase



Note, for example, how the "include" relationship between the use cases *Provide Books* and *Receive Delivery* (Figure 3) turned from "include" into "communication" and also changed the navigability direction. This reflects the fact that in an autonomous organization of agents in a distributed system, we can organize things in a departmental manner, so as to have a *StoreKeeper* actor that records any stock's delivery that occurs. The *StoreUI* agent may then notify the *Purchase Manager* agent of that delivery. In so doing, the *Purchase Manager* does not need to keep bothering about the delivery of a stock, but rather it continues to work while another agent is taking care of this task.

The selection of the use cases that will be part of each agent should be done pursuing the criteria of functionality coherence and cohesion. These are important attributes of the design, and if the adopted agent identification does not produce a satisfactory result from this point of view, a change in it is strongly advised. In a limited number of cases (for instance when relevant limits in communication bandwidth are predictable, as occurs for agents deployed in small

and mobile devices), agents should be composed also considering how big are the information banks they exchange, although this cannot be evaluated at this stage. The iterative and incremental nature of PASSI provides great help in solving this problem; an initial hypothesis for agent identification is done and, if problems occur, it can be changed in successive iterations.

Roles Identification Phase

This phase occurs early in the requirements analysis since we now deal more with an agent's externally visible behavior than its structure – only approximate at this step.

Roles identification (R. Id.) is based on exploring all the possible paths of the Agents Identification diagram involving inter-agent communication. A path describes a scenario of interacting agents working to achieve a required behavior of the system. It is composed of several communication paths. A communication path is simply a "communicate" relationship between two agents in the above diagram. Each of them may belong to several scenarios, which are drawn by means of sequence diagrams in which objects are used to symbolize roles.

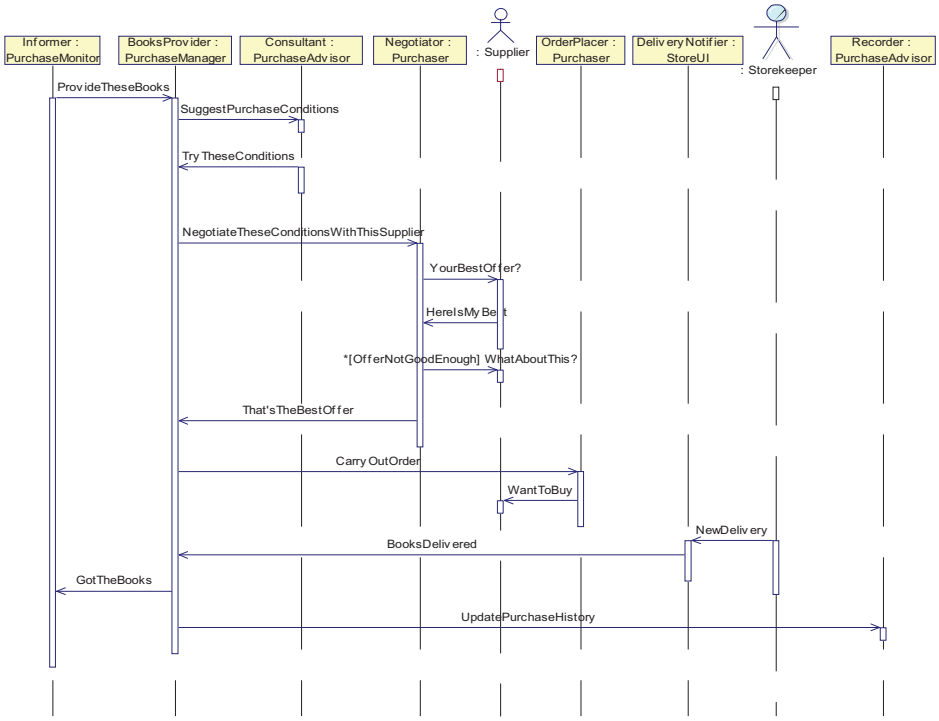
Figure 5 shows the already presented scenario, arising when a new purchase is required from the role *Informer* of the *PurchaseMonitor* agent to the role *BooksProvider* of the *Purchase Manager* agent. Although the diagram resembles an UML sequence diagram, the syntax is a bit different. Each object in the diagram represents an agent's role, and we name it with the following syntax:

<role_name> : *<agent_name>*

An agent may participate in different scenarios playing distinct roles in each. It may also play distinct roles in the same scenario (as happens to the *Purchaser* and the *Purchase Advisor* agents in Figure 5). Usually, UML sequence diagrams begin because of some actor's action; in PASSI, being agents autonomous and active, they can trigger a new scenario and actors can appear later (or not). For this reason, the *PurchaseMonitor* agent (while playing its *Informer* role) can be the first element of this diagram and can fire it.

The messages in the sequence diagram may either signify events generated by the external environment or communication between the roles of one or more agents. A message specifies what the role is to do and possibly the data to be provided or received.

Figure 5. The roles identification diagram for the scenario in which the Purchase Monitor agent announces the need for a book purchase



We can describe the scenario as follows:

- The *Informer* informs the *BooksProvider* that the bookstore needs to purchase a specified stock of books.
- Given a list of suppliers for the needed books, the *BooksProvider* requests that the *Consultant* suggest purchase conditions (number of stocks, purchase money, etc.) on the basis of past business.
- Whether the *Consultant* has returned any advice or not, the *BooksProvider* gives the *Negotiator* the data about the supplier with which to negotiate and the conditions to be negotiated; at the same time, it requests the negotiation to be started. The *BooksProvider* is then ready to take care of other requests that may come from the cooperating agents' roles.

- The *Negotiator* negotiates via fax or e-mail (this is the case of the present scenario) and gets the best offer. It then returns it to the *BooksProvider*.
- The *BooksProvider* establishes whether the offer is good enough or not, according to its budget and considerations such as the pricing of the book and the number of students that would then buy it. In this scenario, we assume that the offer is good enough and so the *BooksProvider* proposes that the *OrderPlacer* buys the books. Therefore, the *BooksProvider* is then ready to take care of other requests.
- When the books are delivered, a notification is then forwarded from the *DeliveryNotifier* to the *BooksProvider*.

The rest of the scenario is straightforward. Data contained in the messages of the above sequence diagram are specified more in details later in the Ontology Description phase.

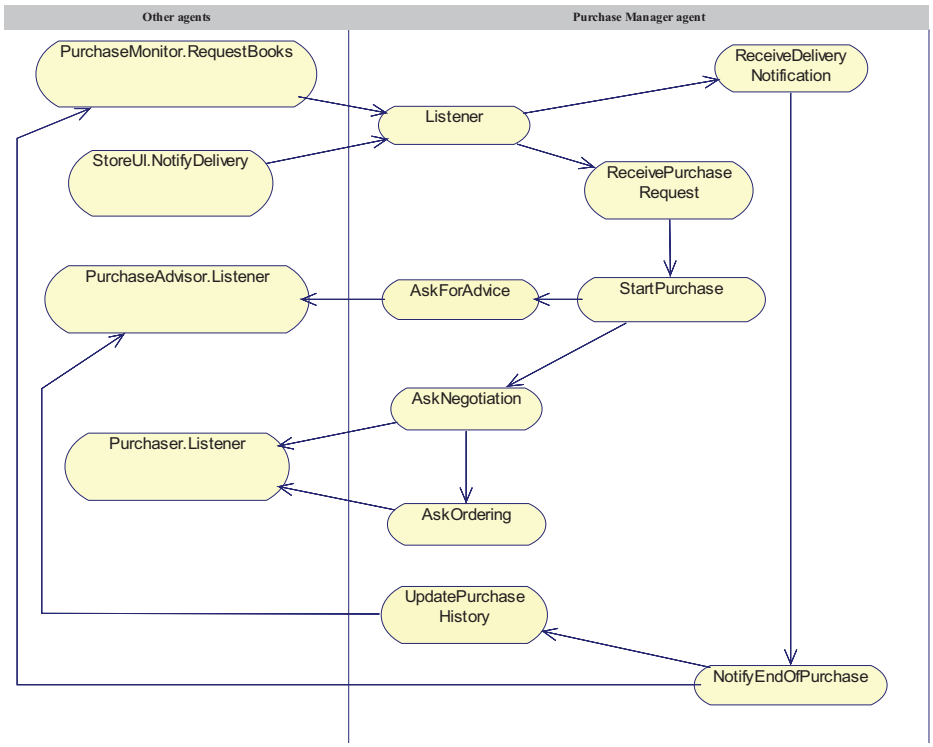
Task Specification Phase

At this step, we focus on each agent's behavior in order to conceive of a plan that could fulfil the agent's requirements by delegating its functionalities to a set of tasks. Tasks generally encapsulate some functionality that forms a logical unit of work. For every agent in the model, we draw an activity diagram that is made up of two swimlanes. The one from the right-hand side contains a collection of activities symbolizing the agent's tasks, whereas the one from the left-hand side contains some activities representing the other interacting agents.

A Task Specification diagram (T.Sp.) (see Figure 6) summarizes what the agent is capable of doing, ignoring information about roles that an agent plays when carrying out particular tasks. Relationships between activities signify either messages between tasks and other interacting agents or communication between tasks of the same agent. The latter are not speech acts, but rather signals addressing the necessity of beginning an elaboration, that is, triggering a task execution or delegating another task to do something. In order to yield an agent's T.Sp. diagram, we need to look at all of the agent's R.Id. diagrams (i.e., all of the scenarios in which it participates). We then explore all of the interactions and internal actions that the agent performs to accomplish a scenario's purpose. From each R.Id. diagram, we obtain a collection of related tasks. Grouping them all together appropriately then results in the T.Sp. diagram.

Because drawing a Task Specification diagram for each agent would require too much space in this chapter, we proceed from now on by focusing on a single agent: the *Purchase Manager*. In Figure 6, we can see its T.Sp. diagram. In this

Figure 6. The tasks of the purchase manager agent



example, we suppose that a *Listener* task is needed in order to forward incoming communication to the proper task; this is common in many MAS platforms (for example, in FIPA-OS [Poslad, Buckle, & Hadingham, 2000]), while this is not necessary in some others. We decided to present this situation because all the others can be reduced to this one. Further tasks are needed to handle all the incoming messages of the R.Id. scenario (see *ReceivePurchaseRequest* and *ReceiveDeliveryNotification* tasks in Figure 6 that correspond to the R.Id. messages coming from the *Purchase Monitor* and *StoreUI* agents, respectively, in Figure 5). Likewise, a task is introduced for each outgoing message (or series of messages that could be unified in one communication) of the R.Id. scenario (see *AskForAdvice*, *AskNegotiation*, *AskOrdering*, *UpdatePurchaseHistory*, and *NotifyEndOfPurchase* in Figure 6). In this way, we dedicate one task to deal with each communication and, if necessary, with minor other duties (for example, simple elaboration of received data). If a relevant activity follows/prepares the incoming/outgoing communication, extra tasks may be introduced to face a better decomposition of the agent (see *StartPurchase* task in Figure 6).

Ontology Description Phase

In the PASSI methodology, the design of ontology is performed in the Domain Ontology Description (D.O.D.) Phase and a class diagram is used. Several works can be found in the literature about the use of UML for modeling ontology (Bergenti & Poggi, 2000, Cranefield & Purvis, 1999). Figure 7 reports an example of a PASSI D.O.D. diagram; it describes the ontology in terms of concepts (categories, entities of the domain), predicates (assertions on properties of concepts), actions (performed in the domain), and their relationships. This diagram represents an XML schema that is useful to obtain a Resource Description Framework (RDF) encoding of the ontological structure. We have adopted RDF to represent our ontologies, since it is part of both the W3C (1999) and FIPA (2001) specifications.

Elements of the ontology are related using three UML standard relationships:

- *Generalization*, which permits the "generalize" relation between two entities, which is one of the essential operators for constructing an ontology;
- *Association,* which models the existence of some kind of logical relationship between two entities and allows the specification of the role of the involved entities in order to clarify the structure; and
- *Aggregation,* which can be used to construct sets where value restrictions can be explicitly specified; this originates from the W3C RDF specification where three types of container objects are enumerated, namely the *bag* (an unordered list of resources), the *sequence* (an ordered list of resources), and the *alternative* (a list of alternative values of a property), and is therefore not UML-compliant.

The example in Figure 7 shows that each *Purchase* is related to a *SuccessfulNegotiation*, a predicate that reports if an order has been issued (attribute *orderIssued* is true in this case) as a consequence of a negotiation. It includes a request from the library (*ourRequest*) for a specific *Stock* and an offer from the supplier (*theirBestOffer*) for that *Stock*. *Delivery* is an example of action—it describes the activity done by the *Supplier* of delivering to the *Storekeeper* some books listed in an ordered stock.

The Communication Ontology Description (C.O.D.) diagram (Figure 8) is a representation of the agents' (social) interactions; this is a class diagram that shows all agents and all their interactions (lines connecting agents). In designing this diagram, we start from the results of the A.Id. (Agent Identification) phase. A class is introduced for each identified agent, and an association is then

introduced for each communication between two agents (ignoring, for the moment, distinctions about agents' roles). Clearly, it is also important to introduce the proper data structure (selected from elements of the Domain Ontology Description) in each agent in order to store the exchanged data. The association line that represents each communication is drawn from the initiator of the conversation to the other agent (participant) as can be deduced from the description of their interaction performed in the Role Identification (R.Id.) phase. According to FIPA standards, communications consist of speech acts (Searle, 1969) and are grouped by FIPA in several interaction protocols that define the sequence of expected messages. As a consequence, each communication is characterized by three attributes, which we group into an association class. This is the characterization of the communication itself (a communication

Figure 7. The domain ontology diagram

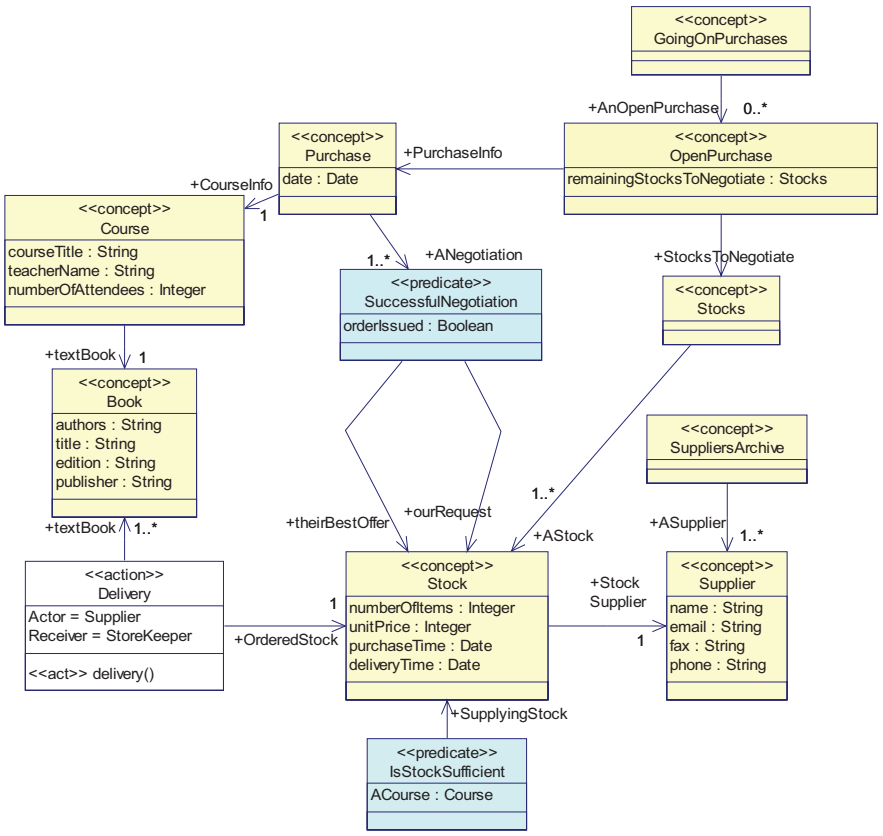
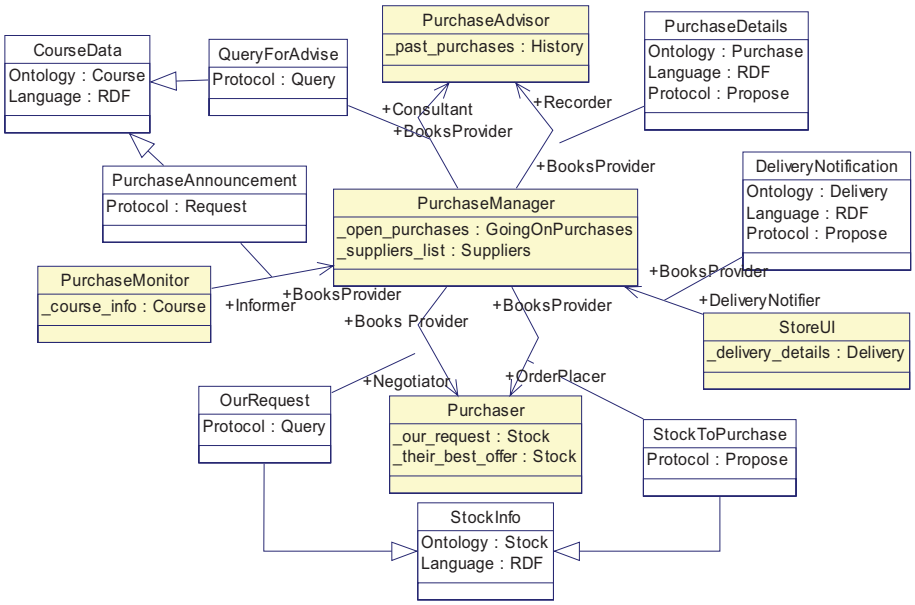


Figure 8. The communication ontology diagram



with different ontology, language, or protocol is certainly different from this one) and its knowledge is used to uniquely refer this communication (which can have, obviously, several instances at runtime, since it may arise more than once). Roles played by agents in the interaction (as derived from the R.Id. diagrams) are reported at the beginning and the end of the association line.

In Figure 8, the *PurchaseManager* agent starts a conversation (see *QueryForAdvice* association class) with the *PurchaseAdvisor* agent. The conversation contains the *Course* ontology, the *Query* protocol, and the RDF language. This means that the *PurchaseManager* wants to perform a speech act based on the FIPA's query protocol in order to ask the *PurchaseAdvisor* for advice on how to purchase (supplier, number of stocks, number of items per each, purchase-money) provided the Course information.

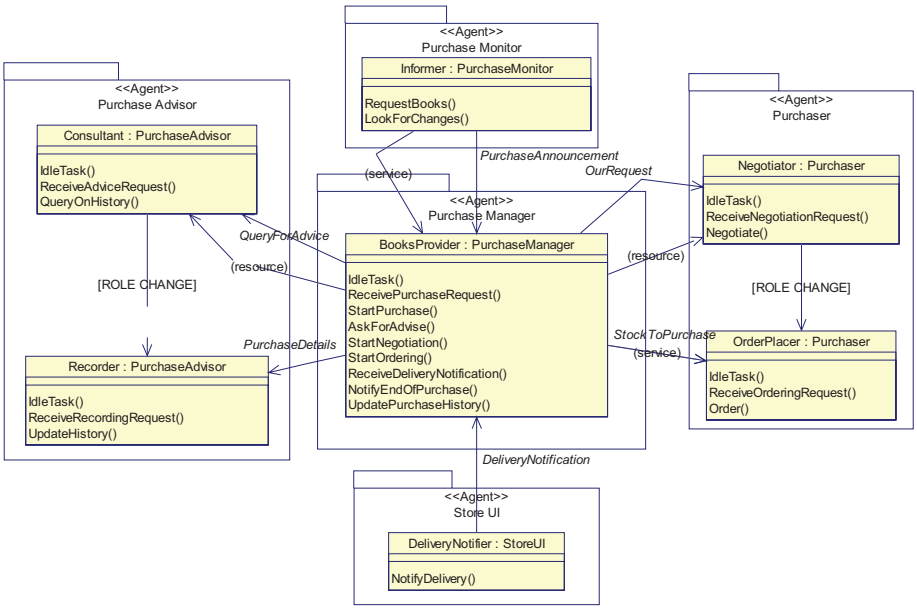
Roles Description Phase

This phase models the lifecycle of an agent taking into account its roles, the collaborations it needs, and the conversations in which it is involved. In this phase, we can also introduce the social rules of the society of agents (organizational rules) (Zambonelli, Jennings, & Wooldridge, 2001) and the behavioral laws as

considered by Newell (1982) in his "social level." These laws may be expressed in OCL or other formal or semi-formal manner depending on our needs.

The Roles Description (R.D.) phase yields a class diagram in which classes are used to represent roles (the UML syntax and notation is here modified slightly in order to represent agents' related concepts). Each agent is symbolized by a package containing its roles' classes (see Figure 9). Each role is obtained by composing several tasks in a resulting behavior. In order to show which tasks are necessary to compose the desired behavior, in this diagram, we put tasks in the operation compartment of the related role's class. Each task is related to an action or a set of actions, and therefore the list of tasks describes what a role is able to do; it can also be helpful in the identification of reusable patterns. An R.D. diagram can also show connections between roles of the same agent, representing changes of role (dashed line with the name [*ROLE CHANGE*]). This connection is depicted as a dependency relationship because we want to signify the dependency of the second role on the first. Sometimes the trigger condition is not explicitly generated by the first role, but its precedent appearance in the scenario justifies the consideration that it is necessary to prepare the situation that allows the second role to start. Conversations between roles are indicated by solid lines, as we did in the Communication Ontology Diagram, using exactly the same relationships names; this consistency, like other quality aspects of

Figure 9. The roles description diagram for our scenario



design, is ensured by the use of PTK (PASSI ToolKit, an open source add-in for Rational Rose™) that automatically builds portions of several diagrams and performs several checks on the inputs provided by the designer to verify their correctness with regards to the other parts of the design.

We have also considered dependencies between agents (Yu & Liu, 2000). Agents are autonomous, so they could refuse to provide a service or a resource. For this reason, the design needs a schema that expresses such matters so as to explore alternative ways to achieve the goals. In order to realize such a schema, we have introduced in the Roles Description diagram some additional relationships that express the following kinds of dependency:

- Service dependency: a role depends on another to bring about a goal (indicated by a dashed line with the *service* name).
- Resource dependency: a role depends on another for the availability of an entity (indicated by a dashed line with the *resource* name).

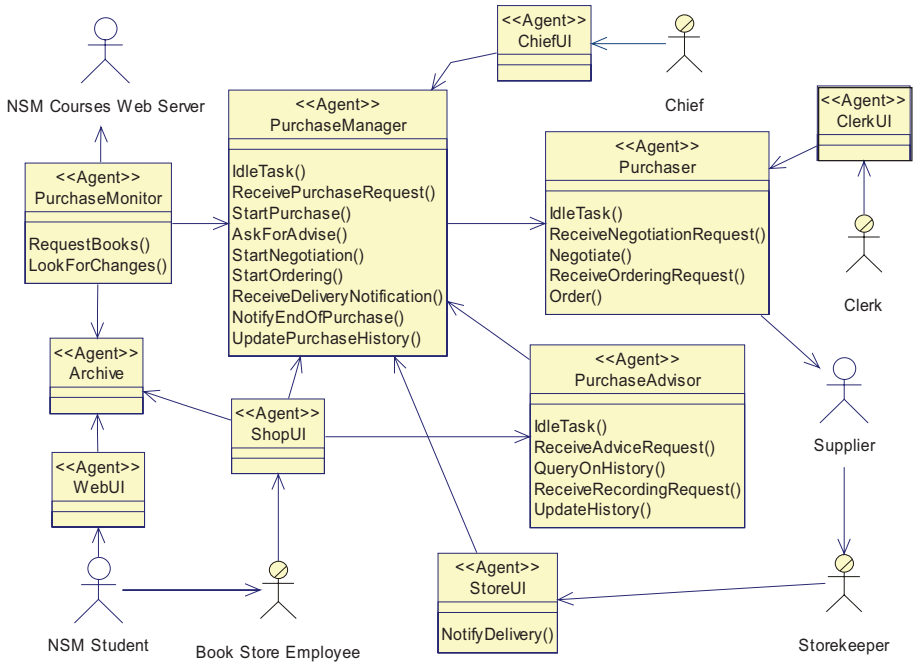
Protocols Description Phase

As we have seen in the Ontology Description phase and as specified by the FIPA architecture, an Agent Interaction Protocol has been used for each communication. In our example, all of them are FIPA standard protocols (FIPA, 2000). Usually the related documentation is given in the form of AUML sequence diagrams (Odell, Van Dyke Parunak, & Bauer, 2001). Hence, the designer does not need to specify protocols on his own. In some cases, however, existing protocols are not adequate and, subsequently, some dedicated ones need to be properly designed; this can be done using the AUML diagrams.

Agents Structure Definition Phase

As argued in subsection "Iterations," this phase influences and is influenced by the Agent Behavior Description phase as a double level of iteration occurs between them. The Agent Structure Definition phase produces several class diagrams logically subdivided into two views: the multi-agent and the single-agent views. In the former, we call attention to the general architecture of the system and so we can find agents and their tasks. In the latter, we focus on each agent's internal structure, revealing all the attributes and methods of the agent class together with its inner tasks' classes (the FIPA-platform classes that will be coded).

Figure 10. The multi-agent structure definition diagram for the bookstore case study



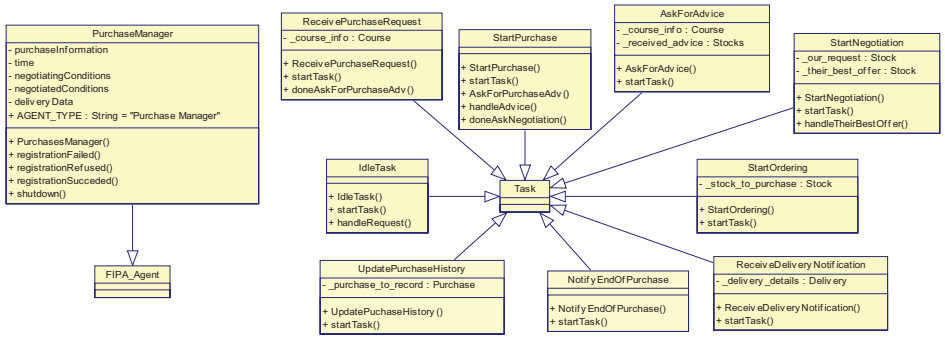
Multi-Agent Structure Definition (MASD)

At this stage, one diagram represents the MAS as a whole (Figure 10). The diagram shows classes, each symbolizing one of the agents identified in the A.Id. phase. Actors are reported in order to represent significant agents' interactions with the environment (for instance through sensing devices or actuators). Attributes compartments can be used to represent the knowledge of the agent as already discussed in the Communication Ontology diagram, whereas operations compartments are used to signify the agent's tasks.

Single-Agent Structure Definition (SASD)

Here one class diagram (Figure 11) is used for each agent to illustrate the agent's internal structure through all of the classes making up the agent, which are the

Figure 11. The single-agent structure definition diagram for the purchase manager agent



agent's main class together with the inner classes identifying its tasks. At this point, we set up attributes and methods of both the agent class (e.g., the constructor and the shutdown method when required by the implementation platform) and the tasks' classes. The result of this stage is to obtain a detailed structure of the software, ready to be implemented almost automatically.

Agents Behavior Description Phase

As was seen in the previous phase, this phase influences and is influenced by the Agent Structure Definition phase in a double level of iterations. The Agent Behavior Description phase produces several diagrams that are subdivided into the multi-agent and the single-agent views. In the former, we draw the flow of events (internal to agents) and communications (among agents) by representing method invocations and the message exchanges. In the latter, we detail the above methods.

Multi-Agent Behaviour Description (MABD)

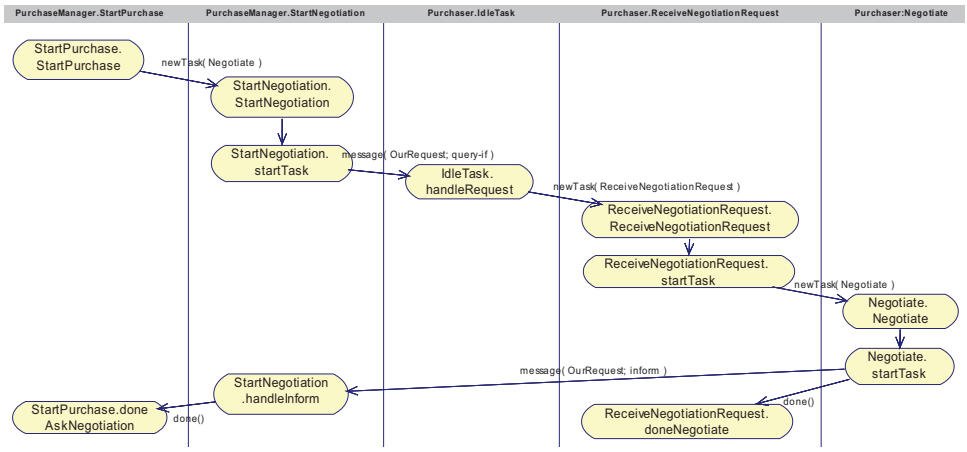
At this stage, one or more activity diagrams are drawn to show the flow of events between and within both the main agent classes and their inner classes (representing their tasks). We depict one swimlane for each agent and for each task. The activities inside the swimlanes indicate the methods of the related class. Unlike DeLoach, Wood, and Sparkman (2001), we need not introduce a specific diagram for concurrency and synchronization since UML activity diagrams' syntax already supports it.

The usual transitions of the UML standard are depicted here as signifying either events (e.g., an incoming message or a task conclusion) or invocation of methods. A transition is drawn for each message recognized in the preceding phases (e.g., from the R.Id. diagram). In this kind of transition, we indicate the message's performative as it is specified in the Communication Ontology Description diagram and the message's content as described in the Domain Ontology Description diagram. This results in having a comprehensive description of the communication including the exact methods involved.

Figure 12 shows an example of a multi-agent behavior description. The *StartPurchase* task of the *PurchaseManager* agent instantiates the *StartNegotiation* task by invoking the *newTask* super-class method. This has to be done in order to ask the *Purchaser* agent to perform a negotiation with a supplier. The invocation of the *StartNegotiation* task implies its *startTask* method to be invoked (according to the FIPA-OS implementation platform we have used). What the *startTask* method does is send a message to the *Purchaser* agent. This contains the *Request* performative (as required by the FIPA Request protocol) and the content *OurRequest* (coming from the D.O.D. diagram, Figure 7). The *handleRequest* method of the *Purchaser's IdleTask* task receives the incoming communication and sends it to the *ReceiveNegotiationRequest* task after this one has been instantiated as above. When a task completes its job, the *done* method is invoked.

This kind of diagram often becomes very huge and difficult to draw/read. In order to deal with this problem, an extended version of it has been presented in Caire et al. (2004) where the revised syntax supports different levels of detail.

Figure 12. An example of multi-agent behaviour description diagram



Single-Agent Behaviour Description (SABD)

This phase is quite a common one as it involves implementation of methods, exactly the ones introduced in the SASD diagrams. Designers are free to describe them in the most appropriate way (for example, using flow charts, state diagrams, or semi-formal text descriptions).

Code Reuse Phase

In this phase, we try to reuse predefined patterns of agents and tasks. With the term pattern, we do not only mean code but also design diagrams. As a matter of fact, the reuse process typically takes place in some CASE tool environment, where the designer looks more at diagrams detailing a pattern's libraries than rough code. So we prefer to look at patterns as pieces of design and code to be reused in the process of implementing new systems.

We have extended the Rational Rose UML CASE tool by developing an add-in supporting PASSI (PTK) and a specific pattern reuse application (AgentFactory). PTK and AgentFactory are complementary and responsible for two different stages of the design-implementation activities: initially, PTK helps in compiling the PASSI diagrams, then AgentFactory is used to generate the agents' code when patterns have been used in the design. PTK initial releases were able to generate agents' code, but this duty has been, more recently, assigned to the AgentFactory application. It works in this way: the PTK (PASSI ToolKit) add-in can export the multi-agent system model to AgentFactory or generate the code for just the skeletons of the designed agents, behaviors, and other classes included in the project. AgentFactory code generation capabilities (Cossentino, Sabatucci, Sorace, & Chella, 2003) are much more advanced than similar functions of PTK; AgentFactory can, very quickly, create complex multi-agent systems by using patterns from a large repository and can also provide the design documentation of the composed agents. The tool can work online as a Web-based application, but can also be used as a stand-alone application. This approach has proven quite flexible (Cossentino, Sabatucci, & Chella, 2003) in reusing patterns, thanks to its binding of design elements to code.

Due to the most common FIPA-compliant implementation platforms that delegate a specific task for each specific communication, it has turned out that in our applications, which are mainly JADE or FIPA-OS based, some of the most useful patterns are the ones that could be categorized as interaction patterns.

Our patterns (whose discussion is out of the scope of this chapter) result from the composition of three different aspects of a multi-agent system:

1. the static structure of one or more agent(s) or parts of them (i.e. behaviors);
2. the description of the dynamic behavior expressed by the previously cited elements; and
3. the program code that realizes both the static structure (skeletons) and the dynamic behavior (inner parts of methods) in a specific agent platform context (for example JADE).

In reusing patterns from our repository, the designer can select the *generic agent* pattern (that has the capability of registering itself to the basic platform services), and he/she can introduce it in the actual project. In this way, with a few mouse clicks, he/she created a totally new agent, the design diagram has been updated (although with some limitations due to the actual level of integration between Rational Rose and AgentFactory), and the agent's code is properly functional.

The repository also includes a list of behaviors that can be applied to existing agents. For example, we have behaviors dedicated to deal with the initiator/participant roles in the most common communications. When a pattern is introduced in the design, not only are some diagrams (like the structural and behavioral one of the implementation level) updated. but the resulting code also contains large portions of inner parts of methods; the result is a highly affordable and quick development production process.

Code Completion Phase

This phase is the classical work of the programmer, who just needs to complete the body of the methods yielded to this point, by taking into account the design diagrams.

Deployment Configuration Phase

The Deployment Configuration (D.C.) phase has been thought to comply with the requirements of detailing the agents' positions in distributed systems or more generally in mobile-agents' contexts.

The Deployment Configuration diagram is a UML deployment diagram and illustrates the location of the agents (the implementation platforms and processing units where they live), their movements, and their communication support. The standard UML notation is useful for representing processing units (by boxes), agents (by components), and the like. What is not supported by UML is the representation of the agent's mobility, which we have done by means of a

syntax extension consisting of a dashed line with a *move_to* stereotype connecting an agent reported in both its initial and final positions.

Discussion

Methodologies differ in commitments about the target agent architecture. PASSI is a requirement-to-code analysis and design methodology characterized by an iterative step-by-step refinement of the system, producing at its final stage a concrete design and implementation based on the FIPA architecture. Gaia, by contrast, regards the output of the analysis and design process as an abstract specification that necessitates being further developed by extra lower-level design methodologies. So does MaSE, but, on the other hand, it goes further in the design process if compared with Gaia. Now, one might think that a general approach such as Gaia is more advantageous, given the present proliferation of agent technologies. However, PASSI does not lead to a narrow scope concrete technology but rather actually yields executable code for a concrete and increasingly utilized standard architecture such as FIPA.

A key issue in modeling multi-agent system is the conversation among agents. In order to obtain a proper model of conversation, it would be desirable to have an ontology description of the system. Excluding PASSI, none of the other methodologies compared throughout this book specifically addresses such a matter (to be more precise, Dileo, Jacobs, and DeLoach [2002] have recently proposed a method to introduce ontology in MaSE). The PASSI Ontology Description phase describes the society of agents taking into account its ontological point of view. As counterpart, in MaSE, there is a detailed description of conversations by means of complementary state automata (couples of Communication Class Diagram) representing agents' state involved in communication. Together, the complementary sides of conversation make up a protocol definition. As for Gaia, a definition of protocols is provided in the Interaction Model.

Conclusion and Further Work

The methodology proposed here has proved successful with multi-agent and distributed systems, both in robotics and information systems. It has been used in several research projects and in the Software Engineering course at the University of Palermo for final assignments. Students and researchers appreci-

ated the step-by-step guidance provided by the methodology and have found it rather easy to learn and to use. Among the most appreciated features, we can list: (1) the ease of transition for designers coming from the object-oriented world, since the initial parts of PASSI adopt concepts of requirements analysis that are very common in that context; (2) the multiple views that permit an easy analysis of complex systems from many different aspects; (3) the support of a specific design tool (PTK, an add-in for Rational Rose), and (4) the patterns reuse that allows a rapid development of MASs. The implementation environments that we have used were based on the FIPA architecture in accordance with the aim of adopting standards whenever possible. We are now working on the enhancement of the CASE tool supporting PASSI and on the enlargement of the pattern repository in order to further increase the productivity of the PASSI developer.

References

Andersen, E. (1997). Juul Møller Bokhandel A/S. *Norwegian School of Management*. Available online *http://www.espen.com/papers/jme.pdf*

Antón, A.I. & Potts, C. (1998). The use of goals to surface requirements for evolving systems. In *Proceedings of International Conference on Software Engineering (ICSE '98)* (pp. 157-166).

Aridor, Y. & Lange, D.B. (1998). Agent design patterns: Elements of agent application design. In *Proceedings of the Second International Conference on Autonomous Agents* (pp. 108-115).

Bergenti, F. & Poggi A. (2000). Exploiting UML in the design of multi-agent systems. In *Proceedings of First International Workshop Engineering Societies in the Agents World.*

Bernon, C., Camps, V., Gleizes, M-P., & Picard, G. (2004). Tools for self-organizing applications engineering. In *Proceedings of the First International Workshop on Engineering Self-Organising Applications (ESOA)*. Springer-Verlag.

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., & Perini, A. (2004). TROPOS: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, *8*(3), 203-236.

Caire, G., Cossentino, M., Negri, A., Poggi, A., & Turci, P. (2004). Multi-agent systems implementation and testing. In *Proceedings of the Agent Technology to Agent Implementation Symposium (AT2AI-04*).

Chella, A., Cossentino, M., & Lo Faso, U. (2000). Designing agent-based systems with UML. In *Proceedings of International Symposium on Robotics and Automation ISRA2000.*

Cossentino, M., Sabatucci, L., & Chella, A. (2003). A possible approach to the development of robotic multiagent systems. In *Proceedings of IEEE/WIC IAT'03 Conference.*

Cossentino, M., Sabatucci, L., Sorace, S., & Chella, A. (2003). Patterns reuse in the PASSI methodology. *Fourth International Workshop Engineering Societies in the Agents World.*

Cranefield, S. & Purvis, M. (1999). UML as an ontology modelling language. In *Proceedings of the Workshop on Intelligent Information Integration at 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*.

DeLoach, S. A., & Wood, M. (2001). Developing multi-agent systems with agentTool. *Intelligent Agents VII - Proceedings of the 7th International Workshop on Agent Theories, Architectures, and Languages (ATAL2000)*. Springer Lecture Notes in AI. Berlin: Springer Verlag.

DeLoach, S.A., Wood, M.F., & Sparkman, C.H. (2001). Multi-agent systems engineering. *International Journal on Software Engineering and Knowledge Engineering, 11*(3), 231-258.

DiLeo, J., Jacobs, T. & DeLoach, S. (2002). Integrating ontologies into multiagent systems engineering. In *Proceedings of the Fourth International Conference on Agent-Oriented Information Systems (AIOS-2002).*

FIPA. (2000). Communicative Act Library Specification. *FIPA Document #FIPA00037*. Available online *http://www.fipa.org/specs/fipa00037/*

FIPA. (2001). FIPA RDF Content Language Specification. *FIPA Document FIPA XC00011B*. Retrieved from *http://www.fipa.org/specs/ fipa00011/ XC00011B.html*

Jackson, M. (2001). *Problem frames: Analyzing and structuring software development problems.* Reading, MA: Addison Wesley.

Jacobson, I., Christerson, M., Jonsson, P., & Overgaard, G. (1992). *Object-oriented software engineering: A use case driven approach*. Reading, MA: Addison-Wesley.

Jennings, N.R. (2000). On agent-based software engineering. *Artificial Intelligence*, *117*, 277-296.

Kendall, E. A., Krishna, P. V. M., Pathak, C. V., & Suresh, C. B. (1998). Patterns of intelligent and mobile agents. In *Proceedings of the Second International Conference on Autonomous Agents* (pp. 92-99).

Newell, A. (1982). The knowledge level. *Artificial Intelligence*, *18*, 87-127.

O'Brien, P. & Nicol, R. (1998). FIPA - Towards a standard for software agents. *BT Technology Journal,* 16(3), 51-59.

Odell, J., Van Dyke Parunak, H., & Bauer, B. (2001). Representing agent interaction protocols in UML. In *Agent-Oriented Software Engineering* (pp. 121-140). Berlin: Springer-Verlag.

OMG. (2003a). Software Process Engineering Metamodel Specification. Version 1.0.

OMG. (2003b). Unified Modeling Language Specification. Version 1.5.

Poslad S., Buckle, P., & Hadingham, R. (2000). The FIPA-OS agent platform: Open source for open standards. In *Proceedings of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents* (pp. 355-368).

Potts, C. (1999). ScenIC: A strategy for inquiry-driven requirements determination. In *Proceedings of IEEE Fourth International Symposium on Requirements Engineering (RE'99)* (pp. 58-65).

Robbins, J., Medvidovic, N., Redmiles, D., & Rosenblum, D. (1998). Integrating architecture description languages with a standard design method. In *Proceedings of the Twentieth International Conference on Software Engineering (ICSE '98)* (pp. 209-218).

Searle, J.R. (1969). *Speech acts*. Cambridge, UK: Cambridge University Press.

SPEM (2002). Software Process Engineering Metamodel Specification. Version 1.0. OMG document 02-11-04.

W3C. (1999). Resource Description Framework. (RDF), Model and Syntax Specification. *W3C Recommendation 22-02-1999*. Available online *http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/*

Wooldridge, M. & Ciancarini, P. (2001). Agent-oriented software engineering: The state of the art. In P. Ciancarini & M. Wooldridge (Eds.), *Agent-oriented software engineering,* No.1957 in LNCS (pp. 1-28). Berlin: Springer-Verlag.

Yu, E. & Liu, L. (2000). Modelling trust in the i* strategic actors framework. In *Proceedings of the 3rd Workshop on Deception, Fraud and Trust in Agent Societies at Agents 2000.*

Zambonelli, F., Jennings, N., & Wooldridge, M. (2001). Organizational rules as an abstraction for the analysis and design of multi-agent systems. *Journal of Knowledge and Software Engineering*, *11*(3), 303-328.

Zambonelli, F., Jennings, N., & Wooldridge, M. (2003). Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, *12*(3), 417-470.