5 CHAPTER 5: AGENT SYSTEM IMPLEMENTATION

5.1 Introduction

The systematic study of the development of agent systems has a recent history. Little time has elapsed since the scientific world perceived the promise of using the agent paradigm to solve a great variety of problems. This realization prompted many researchers to design, independently, their own infrastructures on which to activate their own agents. The result working proposals were often optimal, very efficient for a specific problem domain, but not devoid of some defects. The programming language, the communication paradigm, and other technical details generally made these frameworks unsuitable for purposes other than those for which a given approach was originally conceived. The total absence of genuine attention towards the system design and development process (and consequent documentation) often stymied the growth, scalability and maintenance of these applications. Furthermore, systems were developed without regard to compliance to any standard, thereby creating agents so significantly diverse that they were unable to interact with each other across different frameworks. Now that agent technology has come of age, these solutions, while good for a first experimental phase, , are inadequate for the true uptake of this paradigm.

The importance of standardization is such a pivotal issue that an international organization, the Foundation for Intelligent Physical Agents (FIPA), was founded to promote the intelligent agent industry by openly developing specifications supporting interoperability among agents and agent-based applications. A new and very active field, agent-oriented software engineering is now dealing with the problem of identifying the proper design method for a multi-agent systems.

In this chapter we deal with all of these themes, first discussing the key features of FIPA specifications in order to position and define widespread concepts like agent, behavior, and communication in a reference context, and then presenting a complete design process (adopting the PASSI methodology) applied to the PPS-Bikes' system case study. In more detail, the chapter is articulated as follows: in paragraph 5.2 the standard architecture designed by FIPA for an agent platform is examined, describing the mandatory components that each platform has to implement, then in paragraph 5.3, using the practical example of the PPS-Bikes' system, the fundamentals guiding the implementation of a multi agent system, starting from the initial design down to the code implementation, are illustrated.

5.2 The FIPA abstract architecture

The work of the FIPA focuses mainly on the definition of the agent platform (AP); this is defined as the physical infrastructure where agents can be deployed. Most of the standardization work, therefore, concerns the definition of some key-points that an AP has to comply with. Thanks to these standards, agents living in two or more FIPA compliant platforms are able to communicate and interoperate with each other.

The principal aspects defined by FIPA specifications are:

- The message level, which describes the composition of a message (expressed with the Agent Communication Language), a set of primitive messages with a specific semantic (referring to the speech acts theory [17]) and the sequence of speech acts that compose a correct communication (the Agent Interaction Protocol);
- The transport level, which details how a message has to be moved from a sender to a receiver;
- The service level, which defines the mechanism used by each agent to offer its own services and to discover the services offered by other agents in the platform.

5.2.1 Architecture overview

One of the main goals of FIPA specifications is to promote inter-operability between agent applications and agent systems and this is achieved by defining the Abstract Architecture Specification. This is a collection of architectural elements that characterize each FIPA-compliant platform. The term ‘abstract’ means that the architecture defines only some functional requirements but it is neutral about the technologies used to achieve them.

The agent-platform architecture (represented in Figure 1) is centered on three mandatory components:

- the DF (Directory Facilitator) component,
- the AMS (Agent Management System) component,
- the MTS (Message Transport System) component.

All of these elements will be examined more in detail in the paragraphs that follow.

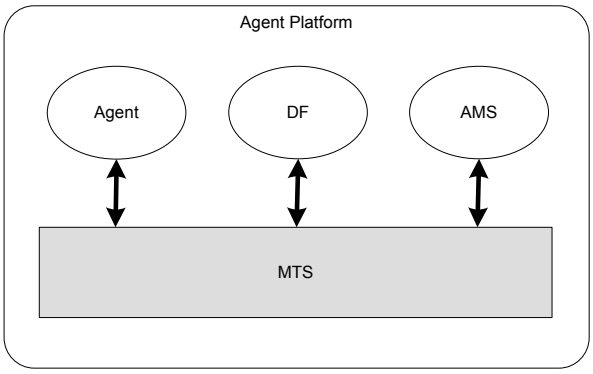

Figure 1. Overview of the FIPA abstract architecture

5.2.2 Infrastructures for agent interactions

The **DF** component of an AP provides the yellow pages service to agents 'living' on that platform. It defines the support for agents' collaborations centered on the concept of service where a service is defined as an activity that an agent performs on the request of another one belonging to the same community. Agents may interact with the DF in two different ways: registration and search. To advertise that a specific service is available to the community the provider agent can register it in the DF with a significant name. Generally an agent can provide more than one service, each one of them being registered in the DF with a different name. An agent has no a-priori knowledge about the other agents of the system. In order to discover if another can be of any help in reaching its own goal(s), the agent may search the DF. Consequently, the agent obtains a vector of DF-entries; each entry contains the univocal address of an agent of the system that performs that service. Generally speaking, the result is a vector, because more than one agent can provide the required service.

The **AMS** is responsible for managing the operation of an AP; the main functionalities of the AMS are the creation, deletion and life-cycle management of agents. The AMS may support other activities that are not mandatory, e.g., the migration of agents to and from other platforms (mobile agents). The AMS maintains the physical index (AID) of all the agents that are currently resident on an AP; this index is an address that univocally identifies all the agents of the system.

The **MTS** (Message Transport Service) is generally invisible to agents and their developers. It provides a mechanism for delivering messages among agents within a platform and to agents that are resident on other platforms. Messages are coded in a standard structure composed of an envelop and a payload. The envelop contains transport information needed for the correct delivery of the message. Transport information could specify a network protocol like HTTP or SMTP and the address of the agent if it is reachable using that protocol (something like *www.mysite.net/abc* or

agentname@host.domain.org). The payload record is coded in a language called Agent Communication Language (ACL) (see also paragraph 5.2.3.2), and it contains the information content that is to be delivered.

5.2.3 Agent Social relationships

Social relationships are among the most important characteristics of agents. A multi-agent system is composed of a number of autonomous and interacting agents and it is frequently represented as a well organized society of individuals. In this context each agent has its own personal goals and plays one or more different roles during its life to interact with other community members.

Agents interact through messages only and, most commonly, their interaction is composed of a series of messages, thus composing what we define as a conversation. It is more correct to think about an agent interaction as a conversation rather then one simple message. A conversation, and specifically a FIPA conversation, is essentially composed of one or more messages. As already mentioned, each message needs a transport infrastructure in order to be delivered. This allows the effective implementation of a conversation but does not ensure any usefulness for it. In order to add a semantic value five important concepts must be adhered to (see Figure 2): ontology, content,

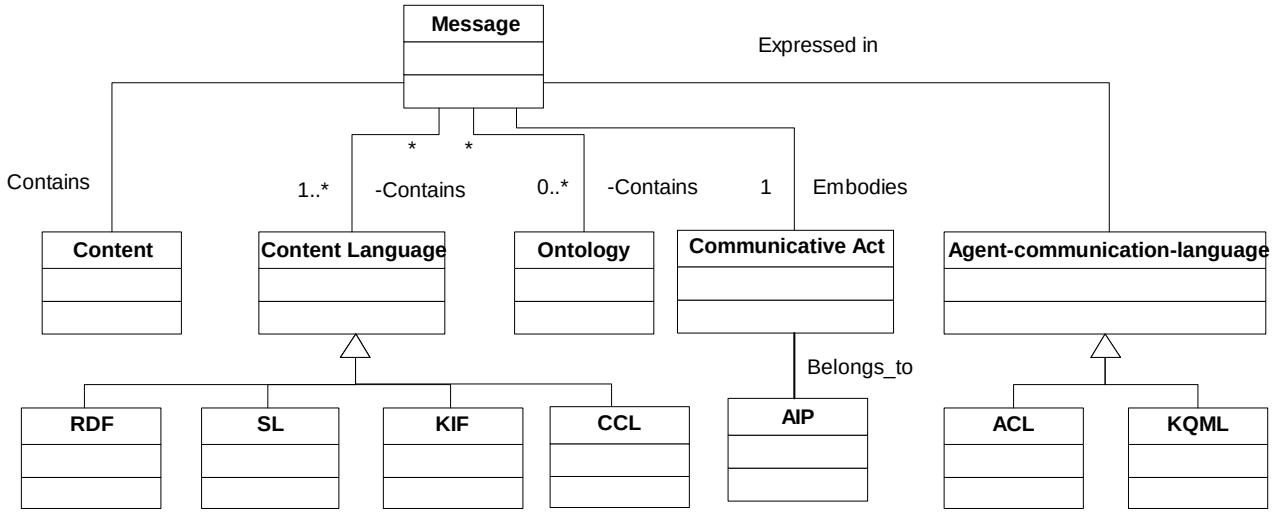

Figure 2. Structural diagram illustrating the elements constituting a FIPA Message and the relationship among them

content language, communicative act and agent interaction protocol (AIP).

5.2.3.1 Modeling the Communication Semantic with an Ontology

An ontology is a representation of the categories that exist in a specific domain; it is a vocabulary used to describe the terms and the relationships among them with a subject matter. An ontology allows the specification of the type of terms an agent may handle and what type of manipulation and reasoning it is able to perform on them. Referring to the same ontology, two agents can interact without the risk of a misunderstanding. They refer to the same set of concepts and, if they adopt the same (content) language, the communication will be meaningful for both of them. On the contrary, the lack of a common ontology introduces the risk that a term used by an agent with some specific significance will be interpreted by another in a different way, thereby jeopardizing agents' interaction and the entire system's performance.

Ontology defines the meaning of categories and the relationship among them but in order to manage it agents need a language that can represent both the ontology structure and content. In many approaches, the ontology structure is composed of three kinds of elements (concepts, predicates and actions), and the associations among them. Many authors have dealt with the representation of the ontology using Unified Modeling Language (UML) [18][19]. In this book we will adopt the PASSI notation that uses a UML class diagram. Concepts, predicates and actions are represented as classes characterized by a specific stereotype. Figure 3 reports a PASSI diagram representing a portion of the ontology designed for the PPS-Bikes' multi-agent system.

As an example, the *Order* class (Figure 3) represents a concept of the ontology; a concept stands for one of the categories of the specific domain, and in this example, *Order* represents the order issued by a customer for receiving some bicycles. It has some attributes, e.g., the *delivery_date*, that is the delivery date requested by the customer for the ordered goods. A concept may be related to other concepts; for example, an order is composed of one or more *OrderStock*, i.e., the number of bicycles of a certain model specified in the order). A concept may extend another concept, inheriting all the attributes and relationships of its super-concept. For example, a *Customer* is a specific *Company* with some supplementary characteristics (the *ID* attribute used to identify it in the bicycle production company).

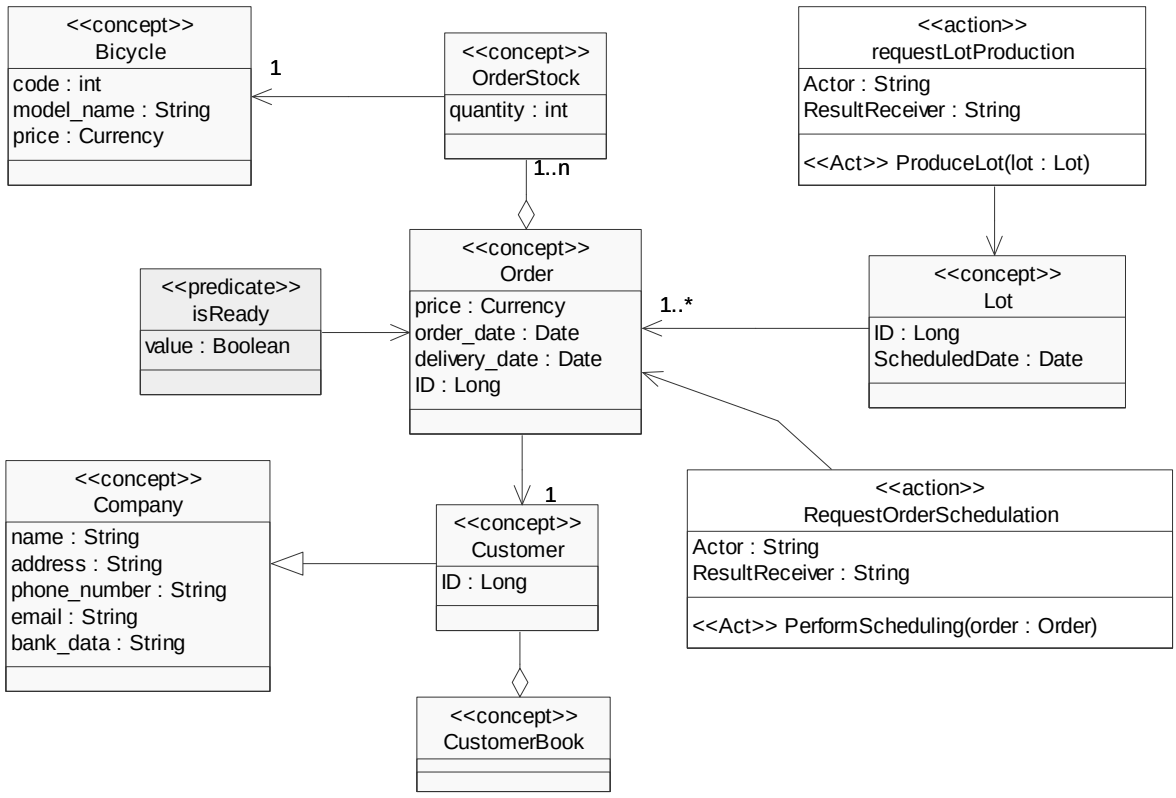

Figure 3. Example of ontology expressed using a UML class diagram (Domain Ontology Description diagram of the PASSI methodology)

A predicate represents a particular statement or belief surrounding some concept, as in the case of the *isReady* predicate shown in Figure 3; this is used to announce that some specific *Order* is ready to be delivered.

An action indicates the type of operation that can be performed on elements of the ontology, thus possibly provoking some changes to the internal knowledge of the agent. *RequestOrderSchedulation*, in Figure 3, is the example of an action specifying the request from one agent to another to schedule the production of the bikes for some specific order.

5.2.3.2 Message Content and Message Content Language

The MTS is the architectural level of a platform that performs the routing of a message from the sender to the receiver whether they are in the same or in different platforms. The life-cycle of a message from its initial creation by the sender to its reading by the receiver agent is reported in Figure 4. The basic information delivered by a message is taken from the ontology of the sender

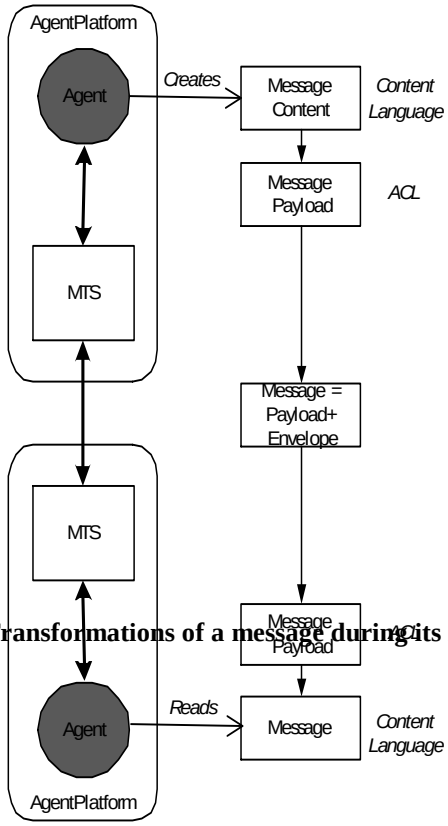

Figure 4. Transformations of a message during its life

agent. It could be a concept, a predicate or an action. The message content (that refers an element of the ontology) is expressed by the agent using a Content Language. FIPA specifications include four languages:

1. Semantic Language (SL)
2. Constraint Satisfaction Problems (CCL)
3. Knowledge Interchange Format (KIF)
4. Resource Description Framework (RDF)

These are born in different contexts and represent the solutions adopted in specific approaches or by some communities; each of them has its specific domain in which it is preferable. The RDF language was created for web applications, but, as previously alluded, it proved to be optimal for representing an ontology for many different applications. It is frequently used, alternatively to SL, as the Content Language of messages exchanged among FIPA agents. The other two languages,

CCL and KIF, were developed for Artificial Intelligence applications; they are very powerful at expressing actions and predicates, but they come with a complex grammar.

The RDF language enjoys very widespread use because i) it is both a W3C[1] and an FIPA[2] standard, ii) it has quite a simple syntax and iii) it allows a number of possible representations (e.g., it also exists in the form of a XML specification). The RDF description (expressed using XML) of the ontology element *Bicycle* reported in Figure 3 is shown in Figure 4.

```
<rdfs:Class rdf:ID="Bicycle">
        <rdf:type rdf:resource="rdfsx:concept"/>
</rdfs:Class>
<rdf:Property ID="Bicycle.model_name">
        <rdfs:domain rdf:resource="#Bicycle"/>
        <rdfs:range rdf:resource="rdfsx:String"/>
</rdf:Property>
<rdf:Property ID="Bicycle.price">
        <rdfs:domain rdf:resource="#Bicycle"/>
        <rdfs:range rdf:resource="#Currency"/>
</rdf:Property>
```

Figure 4. The RDF description of the Bycicle element of the ontology shown in Figure 3

Once the message content is expressed in a content language, it is necessary to encapsulate it into a structure called Message Payload. This structure is coded in a specific ACL that includes several other message parameters, the most relevant of which are:

- performative Type of communicative acts (inform, request, agree…), which depends on the AIP;
- sender ID of the agent that is playing the Sender role in the communication;
- receiver ID of the agent that is playing the Participant role in the communication;
- content The already discussed Message Content (express in a Content Language);
- language Language used for the Message Content;
- ontology name of the ontology element reported in the Message Content;
- protocol name of the AIP used in the communication;

The Message Payload, coded in ACL, is received by the MTS of the platform where the Sender agent is located. MTS encapsulates the payload into an Envelope including the transport information needed to deliver the message: sender and receiver transport-descriptions, plus additional information such as the encoding representation, security related data and whatever else needs to be visible to the MTS. The transport-descriptions describe what transport protocol is to be used (IIOP, HTTP and WAP are all examples of such protocols), and the physical address (e.g., an IP address) to which the message has to be delivered.

1World Wide Web Consortium RDF specifications: http://www.w3.org/RDF/
2FIPA RDF specifications: http://www.fipa.org/specs/fipa00011/

5.2.3.3 Agent Interaction Protocols

The FIPA Abstract Architecture places a great deal of importance to the interaction rules of agent conversations. These have been formalized primarily through two concepts: the communicative act and the AIP (also known simply as protocol in this context). According to the FIPA directive, each conversation has to respect a protocol and has to be made up of communicative acts (see also Figure 2). A **communicative act** is a way to associate a predefined semantic to the content of a message so that it can be univocally understood by agents. The FIPA is responsible for maintaining a consistent list of communicative acts. Some examples of communicative acts are reported in Figure 5; they are *request*, *refuse*, *agree*, *inform*, and *failure*.

A **protocol** univocally defines which communicative acts may be used in a conversation and in what order the related messages have to be sent to give the proper meaning to the communication. Therefore, a protocol compels the use of determined messages with a specific semantic according to a specific sequence. When an agent starts a conversation with another agent it has to specify a protocol; a conversation without a protocol is not possible. If a message does not respect the rules of the protocol or violates the prescribed order, then the conversation fails.

Until now, FIPA specifications use AUML diagrams [20][21] to describe protocols. This diagram is a modified version of the UML sequence diagram. The FIPA Request Interaction Protocol is reported in Figure 5. This may be used when one agent (the Initiator) asks another (the Participant) to perform some kind of action.

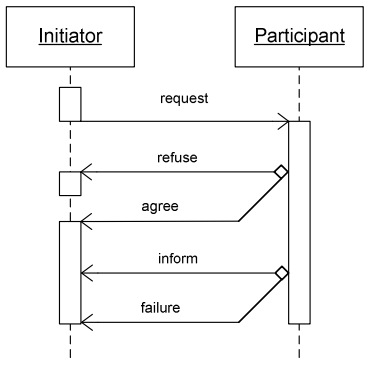

Figure 5. The FIPA Request Interacting Protocol

To start the conversation the Initiator sends a *request* communication act. The content of the message is a description, constructed in a language the receiver understands, of the action to be performed; if there is a common ontology the content may be an ontology action (as described in the previous paragraph).

The Participant processes the incoming *request* and it decides whether to accept or refuse it. The receiving agent makes a decision on the basis of a type of reasoning as could be expected given the principle of autonomy of agents. If the Participant agent agrees to perform the requested action, then it replies with an *agree* message; otherwise a *refuse* message is sent (the possibility of sending an *agree* or *refuse* response is represented in Figure 5 by the diamond).

Once the request has been accepted, the Participant has to fulfill the action and, according to the result obtained, reply with one of the following communicative acts:

- a *failure* message to notify that the action was not completed for some reason; this motivation is usually reported in the content of the message;
- an *inform* message to communicate that it successfully carried on the action to be done; some information on the action results may be reported in the content of the message (e.g., a link to a web site selected according to criteria passed on by the Initiator agent).

5.2.4 JADE: an Implementation Platform

The FIPA describes an abstract architecture that cannot be directly implemented; since the main focus of these specifications regards agent interoperability, not many details are provided on the platform implementation aspects.

On this basis a great number of different solutions have been proposed over the last years, a list of which can be found on the FIPA web site. Among the most widely used are FIPA-OS, JADE and Zeus. In this section the JADE AP is briefly analyzed in order to illustrate some of its specific implementation details.

JADE (Java Agent DEvelopment Framework) [12] was completely developed in Java language by Telecom Italia Lab with the collaboration of the University of Parma. The JADE platform has many interesting features; one of these is the support it provides for agent mobility, which allows its use for the creation of distributed applications where mobility plays an important role (e.g., searching).

A JADE agent is based on a class that extends the *Agent* super-class (a UML class diagram representing the *Administration* agent from the bicycle case study reported in the next subsection is shown in Figure 6). The agent class usually contains a constructor (required by Java and, by convention, in JADE used to initialize data structures) and the *setup* method, which, automatically invoked by the platform once the constructor ends, is often used to begin the agent activity. An agent can be instantiated only by the platform; when this happens, a univocal ID is assigned to the agent and the constructor followed by the *setup* method are executed. Often, the developer uses the constructor to initialize the agent's data structures and the *setup* method to start the activity of its agent.

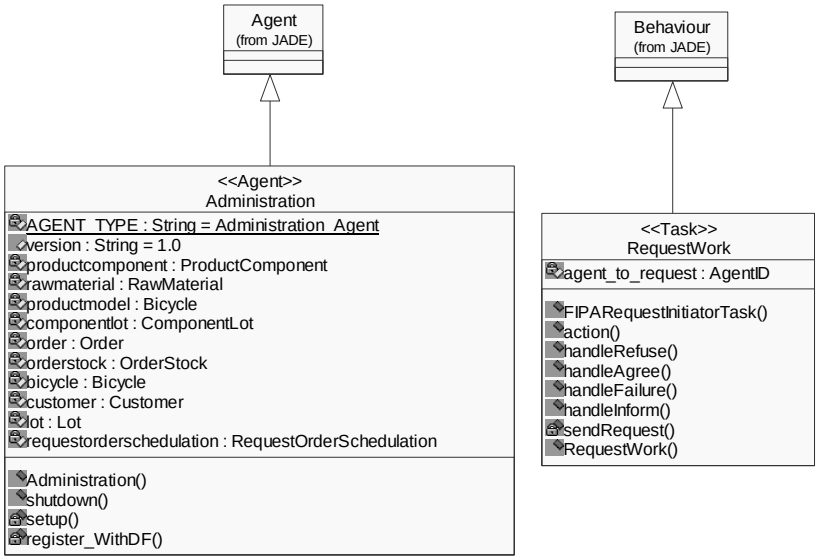

Figure 6. Structure of a Jade agent with a behavior

Another method automatically invoked by the platform is *shutdown*, which arises when an agent is about to terminate It contains the code needed to properly conclude the agent's activities and to reallocate the assigned resources; the JADE *Agent* class (the mother class of all the agents) already provides such a method and, in most cases, this is sufficient to successfully shutdown the agent.

Agent activities are typically not described in its base class methods, but are located in some sub-classes called *behaviors*. A *behavior* represents the atomic element of decomposition of the agent's tasks. Operations needed to reach a goal of the agent are partitioned among its behaviors. For instance, communication with another agent is delegated to a specific behavior (an example is the *RequestWork* class shown in Figure 6). Concretely, a behavior is a class that extends a JADE super-class called *Behavior*. As seen for the agent base class, a template structure exits for behavior classes. All the behaviors must contain an *action* method. Like the *setup* method, *action* is automatically invoked by the platform, after which the class constructor method is completed; the use is the same but at the behavior level (i.e., it is used to start the operations related to that behavior).

Obviously, a behavior class can contain several methods; a communication behavior is usually made up of a set of methods in order to catch all the incoming messages of a specific protocol. For instance, if a behavior is used to initiate a *Request* communication [1] (as the *RequestWork* behavior of Figure 6) it must contain the *handleRefuse*, *handleAgree*, *handleFailure*, *handleInform* methods.

5.3 A case study: designing the PS-Bikes system

Designing a multi-agent system is as complex as designing an object-oriented one. In order to achieve a sound design and to guarantee access to documentation that could be used to further enhance or maintain the software, a specific design methodology should be adopted. Several different approaches exist in the literature and some of them have been already discussed in the previous chapters. We will now describe an example of a design process, applying it to the construction of an application for the PPS-Bikes' case study. The adopted methodology is PASSI (Process for Agent Societies Specification and Implementation) [2][22] and, with the help of the supporting tool, PTK (PASSI ToolKit), the design documentation will be produced. The system will be implemented using JADE as deployment AP.

5.3.1 PPS-Bikes' case study: system requirements initial description

The first phase of the design in most methodologies entails the elicitation and analysis of requirements. A requirement is a feature that the system must exhibit: it can be a functional requirement, such as service, or a non-functional requirement such as a constraint or a performance issue. In UML [4] (functional) requirements are described with use case diagrams. According to UML [3] a use case represents a coherent unit of functionality provided by a system, a subsystem, or a class, as manifested by sequences of messages exchanged throughout the system (subsystem, class) and one or more outside interactors (called actors), together with actions performed by the system (subsystem, class). An actor defines a coherent set of roles that users of an entity can play when interacting with the entity.

In Figure 7 a use case diagram depicts the functionalities of a portion of the PPS-Bikes' system and the interactions with two actors: the customer department and the production supervisor.

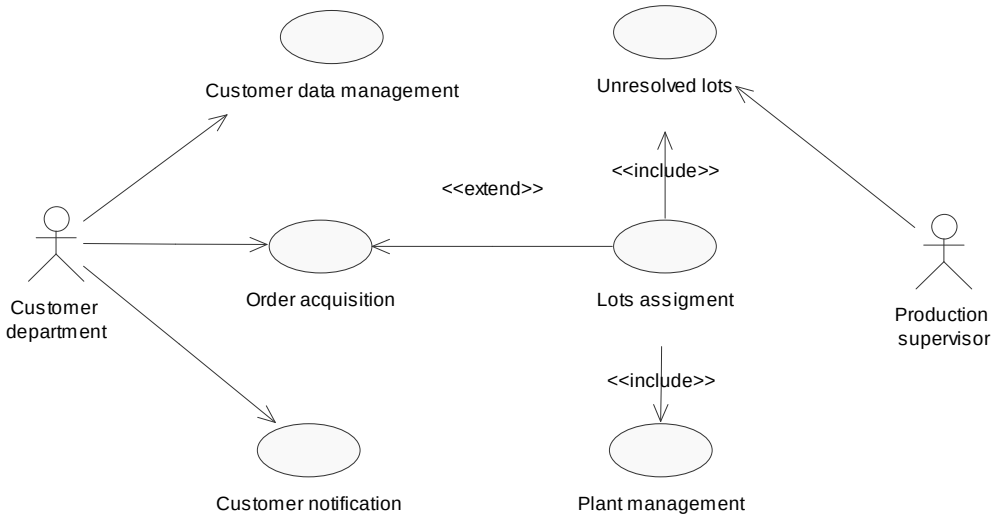

Figure 7. A portion of use case diagram representing the functionalities of the PPS-Bikes system

The company organizes its production on the basis of the received orders. The customers are both wholesalers and retailers of sporting goods; they interact with a figure called the customer department represented by an actor (a stick figure) in the diagram. When a customer wants to place an order for some bicycles, he contacts the customer department directly (e.g., sending the order by fax); using a graphical interface the customer department employee that receives the customer order may introduce the data into the system. This functionality is represented by the "Order acquisition" use case. The "customer data management" functionality allows the company to maintain an archive of customers. The administration department generates plans for the production phases of the two plants on the basis of forecasts of the demands and customers' orders. When an order is placed by the customer department, it has to be composed in lots and its production assigned to a specific plant. These operations are represented by the "Lot assignment" and "Plant management" use cases. The person responsible for interacting with the lot scheduler is the *Production Supervisor*.

5.4 Designing the solution with PASSI

It is well known that code production is a complex activity, and the agent oriented paradigm does not ignore this hurdle. A methodology to design and implement multi-agent systems is a pre-requisite approach to simplify this task. The PASSI methodology is a step-by-step requirements-to-code methodology for designing and developing multi-agent societies. It integrates design models and concepts from both OO software engineering and artificial intelligence approaches using the UML notation with some extensions.

As already mentioned, the methodology is supported by PTK (PASSI Toolkit), a Rational Rose plug-in, and also by a repository of patterns for agents. These tools are very useful in the design and development of the MAS (multi-agent system) because they introduce a level of automation in the process, thus enhancing the designer's productivity. This is particularly effective when entire portions of the model are taken from the patterns repository; this reuse, performed during the design phase, also affects the coding activity, since a significant portion of code is automatically generated starting from the pattern structure.

In the following sections, the PASSI methodology is synthetically analyzed in order to illustrate how a methodology specifically conceived for multi agent systems can support and simplify the designer's work. The methodology is applied to the design of a system for the PPS-Bikes' case study.

5.4.1 The PASSI Methodology

PASSI is composed of five models (Figure 8) regarding the different abstraction levels of the process:

1. **System Requirements Model**. The initial part of this model is similar to other common object-oriented methodologies (requirements analysis). An agent-based solution to the problem is thus drafted.
2. **Agent Society Model**. This describes the details of the system solution in terms of agent society concepts like ontology, communications and roles.
3. **Agent Implementation Model**. The previous models are used to obtain a detailed description of the agent society in terms of both structure and behavior that can be used to produce the code of the system.
4. **Code Model**. In order to streamline and speed up the development of a new system, code is partially obtained from the application of patterns. A conventional code completion activity is then carried out.
5. **Deployment Model**. Mobile agents require that specific attention be paid to the specification of their needs in terms of both software environments (e.g., libraries available in the host platform) and hardware capabilities and performance (e.g., amount of available network bandwidth); these are the issues defined in the deployment model.

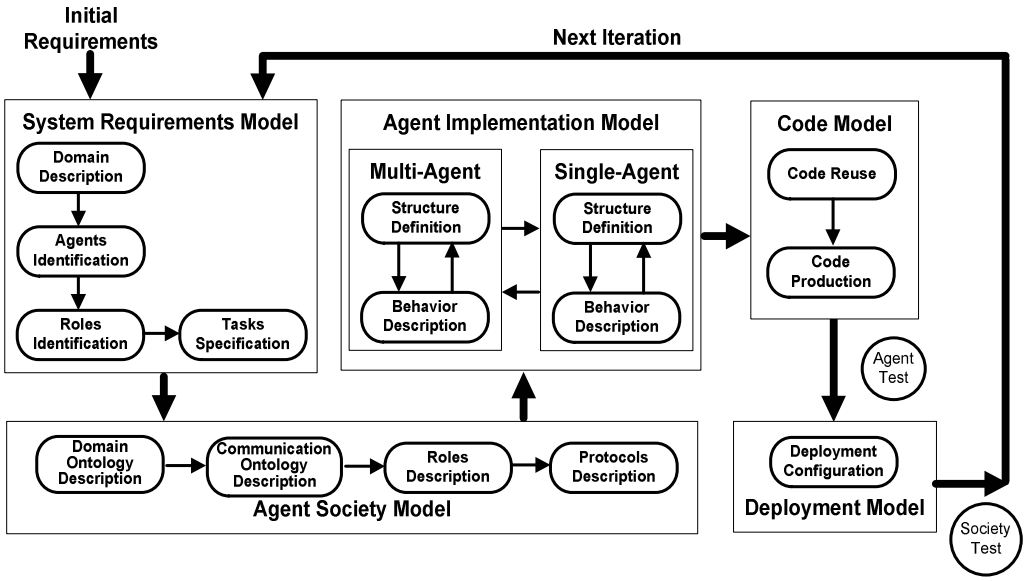

Figure 8. The different steps and models of the PASSI design process

5.4.2 The System Requirements Model

The **System Requirements Model** is a model of the system requirements in terms of agency and purpose. The methodology is use case driven and starts with the requirements analysis, where the designer models the system as a set of use case diagrams. Some of these diagrams, the Domain (Requirements) Description diagrams, are drawn to represent the actors and the use cases identified for the system. Figure 7 reports some of the use cases of our PPS-Bikes' system. In this kind of diagram the designer can identify the agents that will populate the solution. In PASSI, each agent receives the responsibility for a part of the functionalities of the whole system; this is represented in a use case diagram (called Agent Identification diagram) by grouping some of the use cases within a package and giving it the name of the agent.

Figure 9 depicts a portion of the Agent Identification diagram for the PPS-Bikes' system. It describes only two agents, the *Customer* and the *Administration*; these are displayed as two packages containing some use cases from Figure 7. Each agent is responsible for accomplishing the functionalities associated with the use cases included in its package. For example the *Customer* agent responsibilities include: "*Customer Data Management*", "*Order Acquisition*" and "*Customer Notification*". All of these have a direct interaction with the *Customer department* actor that represents one of the users of the application.

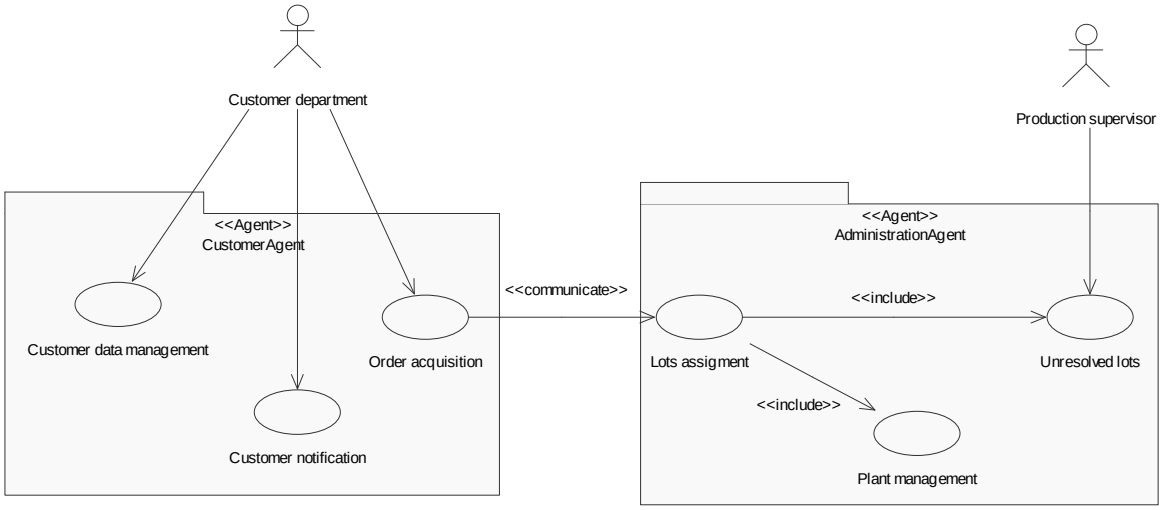

Figure 9. A portion of the agent identification diagram for the PPS-Bikes case study

When two use cases are assigned to different agents and are related by an *include* relationship (showing that the included use case offers some kind of functionality to the including one) or *extend* relationship (showing that the extended use case profits from the extending one to tackle some specific situation triggered by a guard condition), then the involved agents have a dependency and will communicate to achieve the collaboration requested by the relationship between the two use cases.

In this phase an agent is only an aggregation of functionalities. In the example, the *Order Acquisition* and the *Lots Assignment* use cases are connected (see Figure 7) with an *extend* association: in the Agent Identification diagram, this turns into a *communicate* relationship (representing an agent conversation) between the two agents.

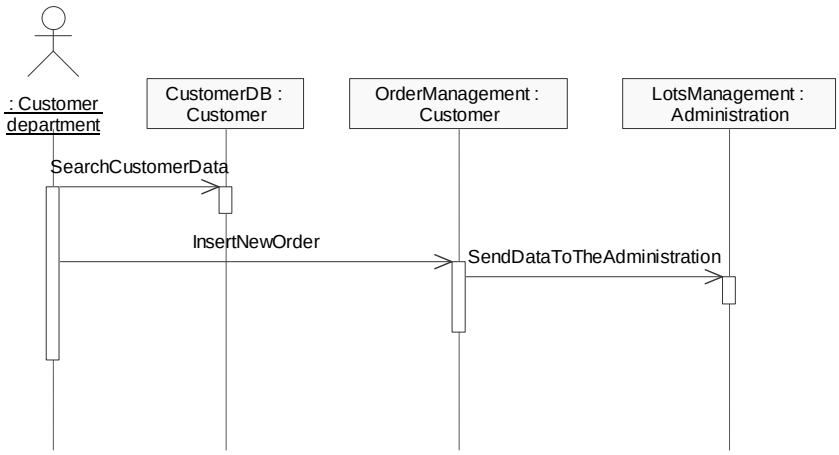

Figure 10. The Role Identification diagram for the "insert new order" scenario

When all the agents are identified, the next step is to explore the scenarios in which they are involved. This is done using a set of UML sequence diagrams; in these diagrams each agent may be involved in many activities and may appear more than once in each single scenario, thus meaning that an agent plays more than one role in that scenario. The identification of agent roles is one of the main outcomes of these diagrams, which are therefore called Role Identification diagrams in PASSI. An example of a Role Identification diagram is shown in Figure 10. Here the *Customer* agent appears twice: in the first instance, it searches for information about a customer in the company database (role *CustomerDB*) and then, in the second, it archives a new customer's order (role *OrderManagement*).

The last step of this first model (the System Requirements Model) is to begin the description the dynamic behavior of each agent. This phase is performed with a set of Task Specification Diagrams (one for each identified agent). According to FIPA definitions [10], a task is "*the observable effect of an operation or an event, including its results. It specifies the computation that generates the effects of the behavioral feature*". Starting from this definition, PASSI considers a task as an entity that is somehow similar to the *Behavior* defined in the JADE agent structure. The Task Specification Diagram is a UML activity diagram representing agents in a swim-lane and their tasks as activities. Each diagram is drawn to detail one agent and only two swim-lanes are present in it (see Figure 11): the right-hand one contains a collection of activities symbolizing the current agent's tasks, while the left-hand one reports some activities from other agents involved in interactions with this specific agent.

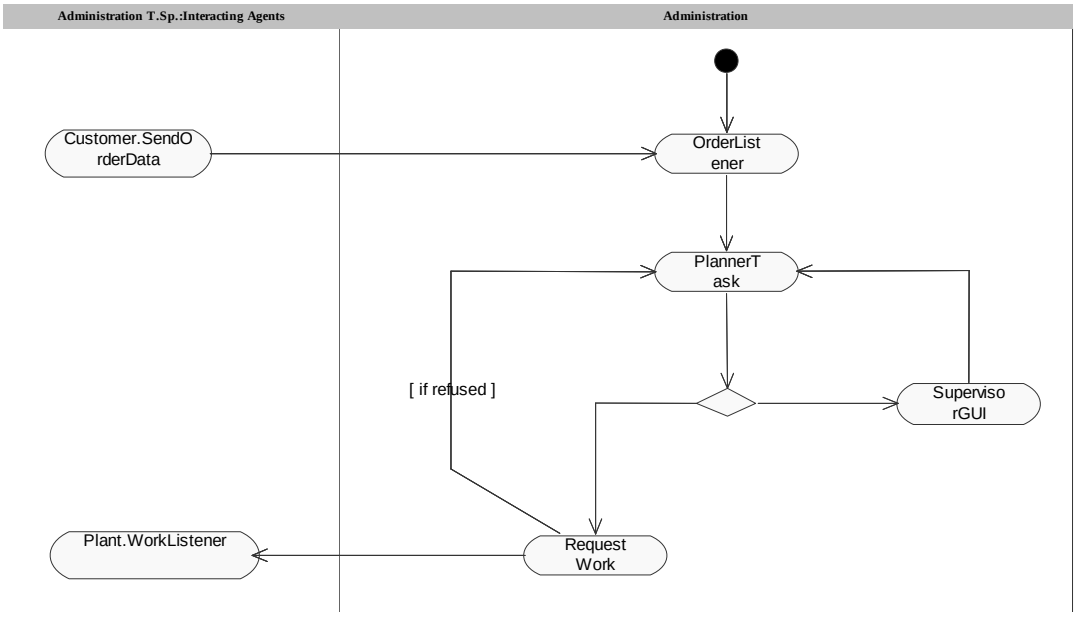

Figure 11. The Task Specification diagram for the *Administration* Agent of the PPS-Bikes' case study

An example of a Task Specification Diagram for the *Administration* agent is reported in Figure 11. This agent is involved in the introduction of a new order from a *Customer* agent. It receives this communication with the *OrderListener* task. After that, the agent plans the bike production with the *PlannerTask* and *RequestWork* tasks. The *SupervisorGUI* task is activated if a problem is found in the planning phase; the task is responsible for notifying the production of the need to manually adjust the plan.

5.4.3 The Agent Society Model

The next PASSI model is the **Agent Society Model** that represents social interactions and dependencies among agents involved in the solution. This model is composed of four phases:

- the **Domain Ontology Description,** where the domain is explored and its distinguishing concepts are identified together with actions and propositions related to them;
- the **Communication Ontology Description,** used to detail agent communications in terms of ontology, content language and interaction protocol;
- the **Roles Description,** which consists of a diagram representing agents with their roles, the tasks involved in those roles and the dependencies among agents/roles in terms of resources to be shared and services to be provided;
- the **Protocols Description**, which constitutes a phase that is frequently skipped by the designer. It is necessary to define a new protocol only if the existing FIPA protocols are insufficient to model the specific communication, and this happens rarely.

In the PASSI methodology the design of ontology is performed in the Domain Ontology Description (DOD) phase and a class diagram is used. Several works can be found in the literature about the use of UML for modeling ontology (6-8). Figure 3 reports an example of a PASSI DOD diagram; it describes the ontology in terms of concepts (categories, entities of the domain), predicates (assertions on properties of concepts) and actions (performed in the domain). 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 [5] and FIPA (FIPA RDF Content Language) [9] specifications.

In Figure 3, the PPS-Bikes system ontology is described by classes and their relationships. Elements of the ontology are related using three UML standard relationships:

- Generalization, permits the ‘generalize’ relation between two entities, which is one of the essential operators for constructing an ontology;
- Association models the existence of some kind of logical relationship between two entities and allows specifying the role of the involved entities in order to clarify the structure;

- Aggregation can be used to construct sets where value restrictions can be explicitly specified; in the W3C RDF specification 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). For our purposes we consider a bag as an aggregation without an explicit restriction, a sequence as being qualified by the *ordered* attribute, while the alternative is identified with the *only_one* attribute of the relationship.

The example (Figure 3) shows that each *Order* concept is characterized by a *price*, *order_date* *delivery_date* and *ID*. Each order aggregates several *OrderStock*s, each one of them describing the number of bikes of a specific type that are part of the order. The bicycle model is described in the homonymous concept. One agent can ask another if an order has been completed, and this instance is stated by the Boolean value of the *isReady* predicate. The *ScheduleManifacturing* action introduces the order (and therefore the specified number of bicycles) in the manufacturing scheduling of the different machine tools.

The Communication Ontology Description (COD) (Figure 12) 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. 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 (coming from the entities described in the DOD.) 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. As already mentioned, each communication is characterized by three attributes, which we group into an association class. This is the characterization of the communication itself (a communication 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.

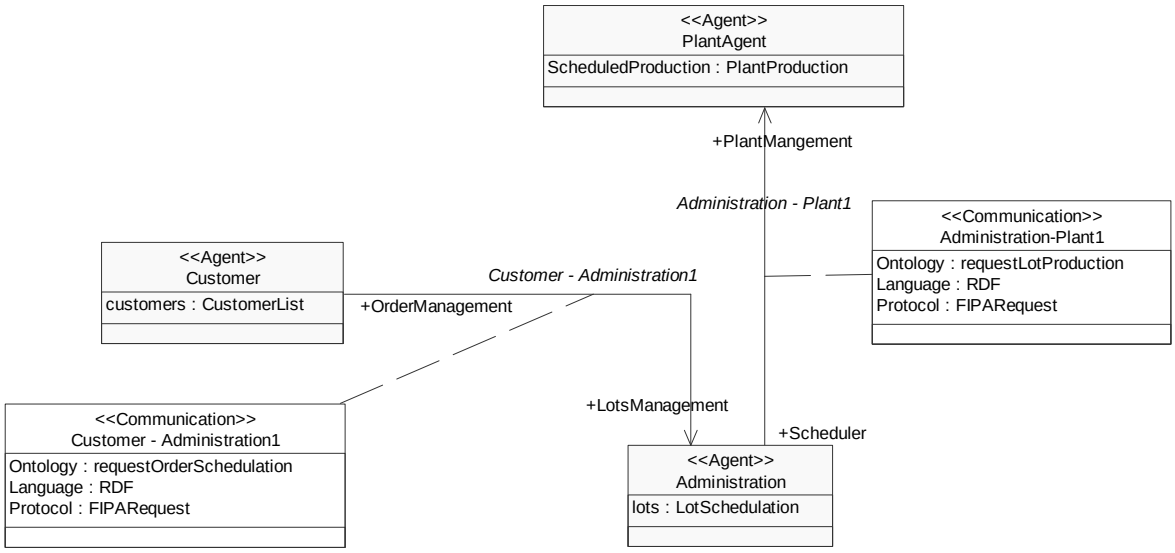

Figure 12. The Communication Ontology Description (COD) diagram for the PPS-Bikes case study

Figure 12 illustrates the communication between the *Customer* and *Administration* agents (the unique communication name is: *Customer-Administration1*). The first initiates the interaction in order to ask the other about the production scheduling of an order for some bikes. The referred ontology is an action (*requestOrderSchedulation*) and the interaction protocol is the FIPA Request that is dedicated to dealing with requests for some kind of service. RDF is the content language.

The FIPA Methodology Glossary [10] defines a role as "*a portion of the social behaviour of an agent that is characterized by some specificity such as a goal, a set of attributes (for example responsibilities, permissions, activities, and protocols) or providing a functionality/service*". In PASSI, roles are initially identified in the already discussed A.Id. diagrams. Their definition is the completed in the Role Description (RD) diagram, i.e., a UML class diagram in which classes are used to represent roles. Agents are represented by packages containing classes of roles (see Figure 13). Each role is achieved by grouping several elementary tasks into a resulting complex behavior; for this reason tasks are shown in the operation compartment of each role's class. An agent during its life can take on several different roles, and this dynamic evolution in its behavior is represented by a dashed line with the name [ROLE CHANGE] that connects its different roles in the expected order. Conversations between roles are indicated by solid lines (as we have depicted in the COD), using exactly the same relationships names.

We have also considered dependencies between agents. Because agents are autonomous and may refuse to provide a service or a resource to another, the design needs a schema that expresses such matters and explores alternative ways to achieve 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, where one role depends on another to bring about a goal (indicated by a dashed line with the *service* stereotype);
- Resource dependency, where one role depends on another for the availability of an entity (indicated by a dashed line with the *resource* stereotype);
- Soft-Service and Soft-Resource dependency, where the requested service/resource is helpful or desirable, but not essential to bring about a role's goal (indicated by a dashed line with the *soft-service* and *soft-resource* stereotypes).

In the example of Figure 13, the *Customer* agent plays the *CustomerDB* role while dealing with the customer data and the *OrderManagement* role while managing customer orders. We can see that several tasks are involved in the exploitation of the second role (e.g., graphical interfaces like *OrderDataGUI* are used to interact with the user that introduces the customer order data). We can also note that this agent initially plays a role related to the compilation of the customer data archive, and then changes its vocation (*Role Change* relationship) towards order-oriented operations. The communication with the *Administration* agent already discussed in the COD. diagram (Figure 12) is also reported in order to simplify the analysis of the interactions among the different roles.

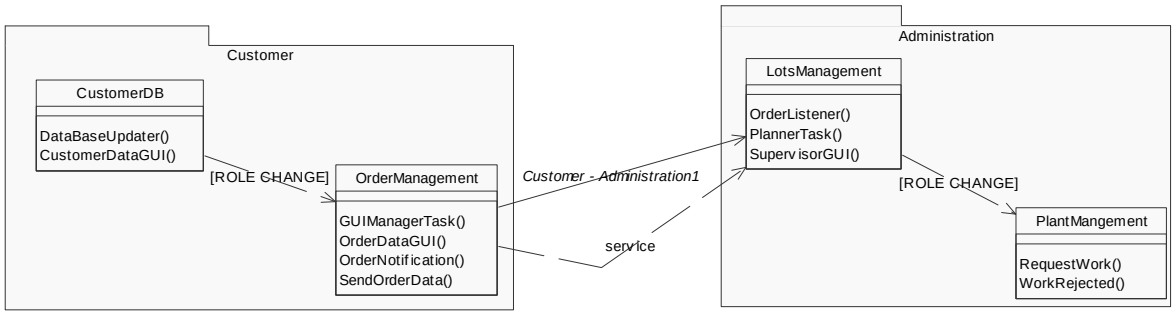

Figure 13. The Role Description (RD) diagram for the PPS-Bikes case study

As we have seen in the DOD phase and as specified by the FIPA architecture, a protocol is used for each communication. All of them are standard FIPA protocols in our case study. Usually, the related documentation is given in form of AUML sequence diagrams [11]. Hence, designers do not need to specify protocols on their own. In some cases, however, existing FIPA protocols are not adequate. If this happens, some specific protocols have to be properly designed (Protocol Description phase); this can be done using the same FIPA documentation's approach (with an AUML sequence diagram as in Figure 5).

5.4.4 The Agent Implementation Model

The **Agent Implementation Model** is a model of the solution architecture. It is composed of two different phases, each performed at both the multi- and single-agent level of abstraction. The multi-agent level deals with the agent society and is therefore detailed to a low degree as regards the agent implementation specifications; however, it fittingly documents the overall structure of the system (behaviors of each agent, communications, etc.). The single-agent level of abstraction focuses on the implementation details of each agent and specifies whatever is needed in order to prepare the coding phase. The two phases are:

- Agent Structure Definition (ASD) uses conventional class diagrams to describe the structure of solution agent classes;
- Agent Behavior Description (ABD); uses activity diagrams or statecharts to describe the behavior of individual agents.

This model is characterized by an iterative process and, specifically, by a double level of iteration (see the Agent Implementation Model box in Figure 8). This model needs to be viewed as being composed of two views: the multi-agent and single-agent views related by two iterations. The outer level of iteration concerns the dependencies between these two views. In each we can find an ASD (representing the agents' structure at the social or inner agent granularity) and an ABD (describing the agents' behaviors again from the social or single agent perspective). An inner level of iteration takes place at both the multi-agent and single-agent views and concerns the dependencies between the structural and behavioral matters. As a consequence of this double level of iteration, the Agent Implementation Model is composed of two steps (ASD and ABD), but still yields four kinds of diagrams taking into account the multi- and the single-agent views.

In the **Multi-Agent Structure Definition** (MASD) diagram, attention is centered on the general architecture of the system. The MASD is an overview of the results obtained from the previous phases from the structural point of view. In this diagram (Figure 14), agents are represented as classes with their behaviors in the operations compartments; attributes specify the agent knowledge. Building this diagram is not an effort for the designer, since PTK (the tool that supports the design with the PASSI methodology) automatically builds it using information coming from previous diagrams.

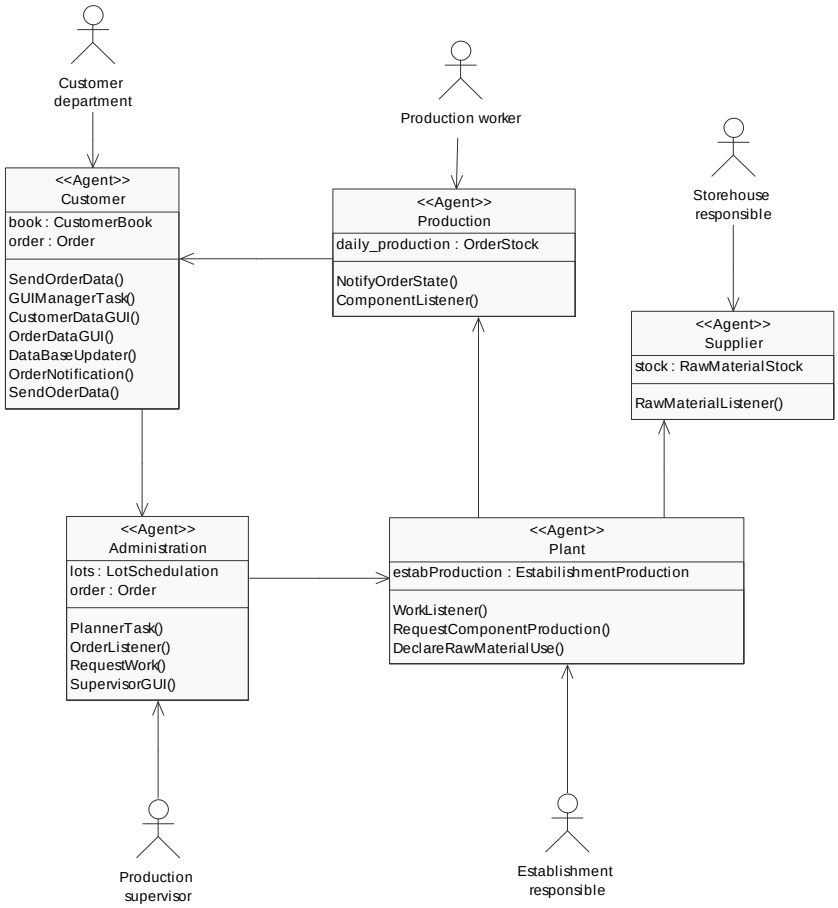

Figure 14. The Multi-Agent Structure Definition diagram for the PPS-Bikes case study

At this point, a new diagram, the **Single-Agent Structure Definition** (SASD) diagram is drawn for each agent in order to explore its internal composition and all of its tasks at a level of detail that is sufficient to generate the implementation code. This diagram is a UML class diagram and reports the agent main class and each agent task as a class, resembling the structure of the most common AP specifications (Jade [12], FIPA-OS [13],). At this point, we set up attributes and methods of the agent class (e.g., the constructor and the shutdown method required by the FIPA-OS platform or just the constructor in JADE) and the task classes (e.g., the methods required to deal with communication events when the agent receives/sends a communicative act).

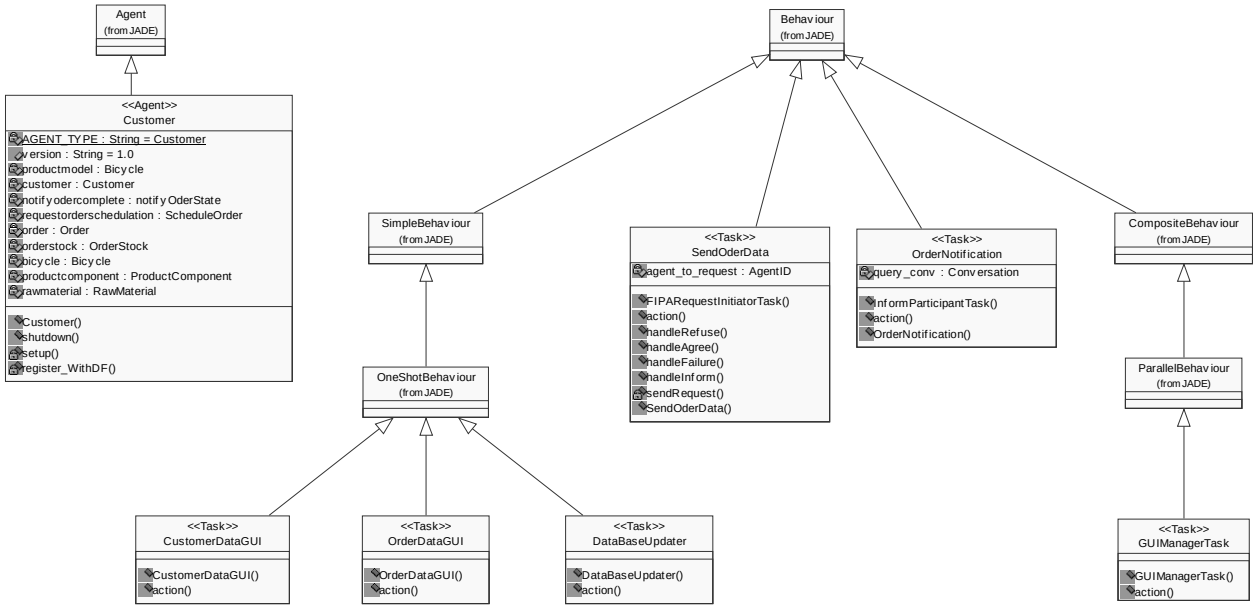

Figure 15. The Single-Agent Structure Definition (SASD) diagram for Customer of the PPS-Bikes case study

An example of an SASD diagram is reported in Figure 15 and describes the internal structure of the *Customer* agent of the PPS-Bikes' case study to be implemented in the JADE platform. The *Customer* main class is derived from the *Agent* base class of JADE. Among its attributes we find *AGENT_TYPE*, which usually contains the name of the agent type (*Customer* in this case), while in the operations compartment we find the *register_WithDF* method that contains the code necessary to register with the yellow pages service of the platform (*Directory Facilitator*).

As regards the agent's tasks (called *Behaviors* in JADE), we can consider *SendOrderData* and *OrderNotification*, which are represented as two classes extending the JADE *Behavior* super class, and whose duties entail dealing with the agent communications (as can be seen in Figure 14, this agent has relationships with both the *Production* and *Administration* agents); for example, *SendOrderData* adopts a Request protocol to delegate the *Administration* to take care of the introduction of a new order in the manufacturing schedule.

A different structure is proposed for *CustomerDataGUI*, *OrderDataGUI* and *DataBaseUpdater*, which are inherited from the JADE *OneShotBehavior* (a behavior that performs a single operation and then terminates its existence). This kind of solution is a valid option for controlling graphical interfaces, i.e., once the interaction with the user is completed, there is no reason for the behavior to remain active.

The agent behavior at the multi-agent level is described by the Multi-Agent Behavior Description (MABD) diagram. This is a UML activity diagram used to illustrate the dynamics of the system during the agents' life. Figure 16 reports an example of MABD; it illustrates the activities occurring during the *Request* communication between the *Customer* and *Administration*

agents. In the diagram, all the involved classes (both of agents and tasks) are represented with swim-lanes (such as *Customer* and *Customer.SendOrderData*), while operations are displayed as an activity (rectangles with rounded corners, like *SendOrderData.PrepareRequest*, which is the constructor method of the *SendOrderData* behavior in Figure 16). In these diagrams, transitions among activities indicate an event as a method invocation (if relating activities in the same swim-lane), a new behavior instantiation (if relating activities of the same agent but in different swim-lanes) or a message (if two different agents are involved). The communication described in the example initiates a request message and then, according to a decision process (not described), the *Administration* agent replies with a *refuse* or *agree* message. Each message is detailed with the communication name and the communicative act.

The **Single Agent Behavior Description** (SASD) is the last phase of the Agent Implementation Model. The approach we use in this activity is quite common. The aim of this phase is to produce a design of the inner part of methods introduced in the SASD diagrams in order to prepare their implementation. The designer is free to describe these features as he/she sees most fitting and appropriate (e.g., using flow charts, state diagrams or semi-formal text descriptions). It should be noted that, because in many instances operations performed according to a method are not complex enough to justify so much attention, a textual description is often sufficient.

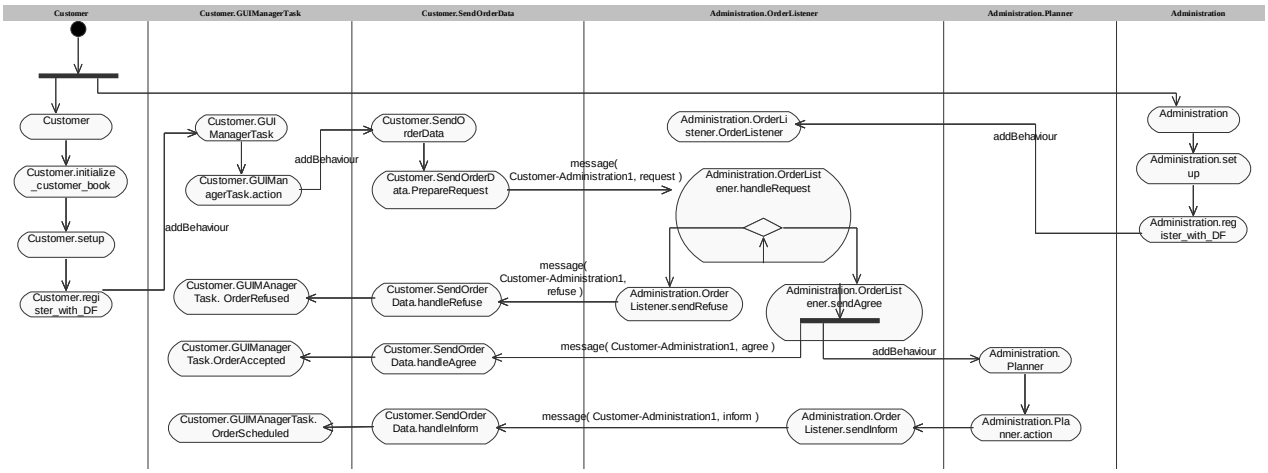

Figure 16. Multi-Agent Behavior Description (M.A.B.D.) diagram used to describe the interaction of two agents during a FIPA Request Communication

5.4.5 The Code Model

The **Code Model** is a model of the solution at the code level. In this phase the developer is aided by a tool (AgentFactory) developed in the order to grant the code reuse. AgentFactory may work inside PTK or as a standalone application, its key feature being that it allows the easy

construction of a substantial part of an MAS reusing elements of its pattern repository (specifically realized to solve agent-oriented problems and therefore different from a common object-oriented one).

An agent pattern, according to the PASSI conception, derives from object-oriented design patterns [14], and describes a tested solution for a recurrent design problem. This pattern [15][16] is presented as a set of diagrams of the PASSI methodology, each describing the different aspects of the problem at different abstraction levels and covering one or more phases of the design process. Typically, diagrams used to describe a pattern are classified in one of two categories: structural or behavioral, the most common diagrams used in the pattern description are the Task Specification, DOD, COD, SASD and MABD. Starting from these representations and from a description of the solution with an XML-based meta-language, AgentFactory can instantiate the implementation code for both the FIPA-OS and JADE platforms. Obviously, the code generation engine also considers the needs emerging from the composition of different parts to create a complex agent structure and can solve all the ensuing problems.

Communication patterns are among the most frequently used by the AgentFactory repository. As an example, the *FIPARequest* pattern introduces one possible solution to the recurrent problem to create a conversation among two agents according the FIPA Request agent interaction protocol (see subsection 5.2.3.3).

The structure of the two agents involved in the communication is described by two SASD diagrams (Figure 17), which illustrate what attributes and methods will be added to the initiator and participant agents when the pattern is applied to them. A plethora of methods are specifically related to protocol communicative acts; these methods have the preamble “handle” followed by the name of the communicative act, e.g., *handleAgree* or *handleInform* are the methods where messages containing the *Agree* of *Inform* performatives will be managed.

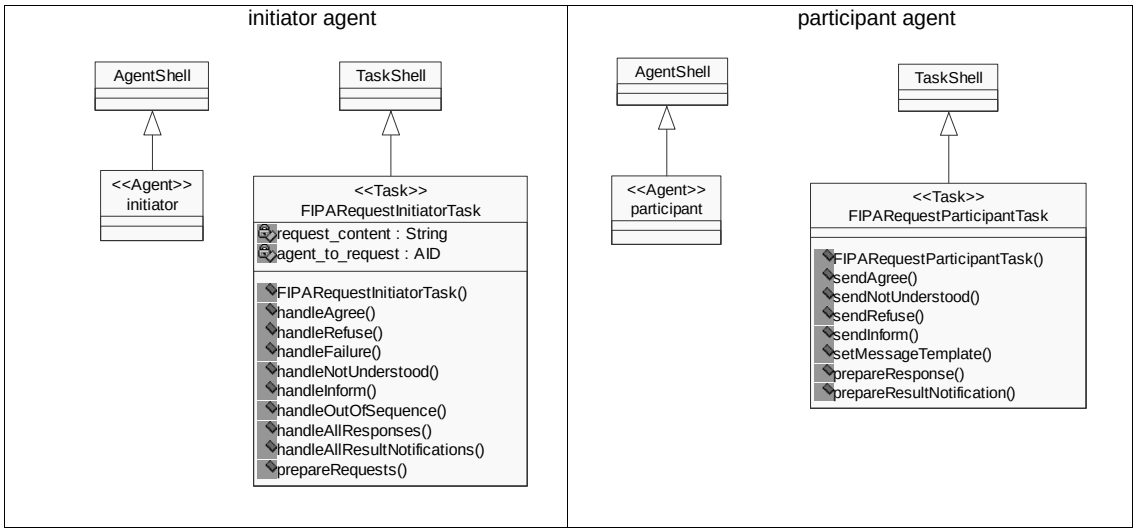

Figure 17. Two class diagrams representing the static structure of the agents involved in a FIPA Request communication

These two diagrams do not suffice to describe all the features of the FIPA protocol management, since they do not provide any dynamic representation. An MABD diagram is therefore needed to complete the pattern description: this is useful to describe the activities performed by the two agents involved in the communication (Figure 18) in a form that can be easily reused as a portion of the actual design of the system (in fact, once a pattern is applied to the project, PTK automatically introduces it in the corresponding diagrams).

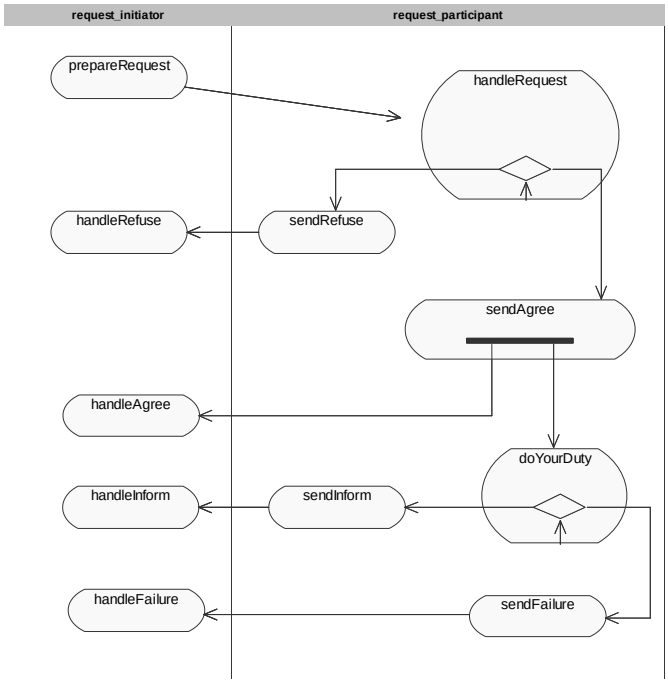

Figure 18. A Multi-Agent Behavior Description diagram used to describe the FIPA Request pattern

The MABD reported in Figure 18 illustrates that the *request_initiator* agent sends a message to the *request_participant* agent with the *prepareRequest* method (see also Figure 17). The responding agent receives it with the *handleRequest* method and according to its will responds with a message containing one of the Request interaction protocol performatives (*Refuse*, *Agree*,…) sent by the correspondent method (*sendRefuse*, *sendAgree*,…).

Since a significant part of the design and an even more substantial part of the code automatically descend from (depend/are contingent on) the appropriate choice of the right pattern for a specific situation, this activity becomes a strategic one and should not be neglected by the designer.

5.4.6 The Deployment Model

The **Deployment Model** is the response to the need to detail the position of the agents in a distributed system or in mobile-agent contexts. The Deployment Configuration diagram (Figure 19) is useful to depict where the agents will be located during their life (i.e., the processing units where they live), their movement and their communication support.

The standard UML notation is useful for representing the elaborating units, here shown as 3-D boxes, and the agents, which are depicted as components; since an agent may be instantiated more than once, agent (instance) names are in the form *agent-name:agent-class*.

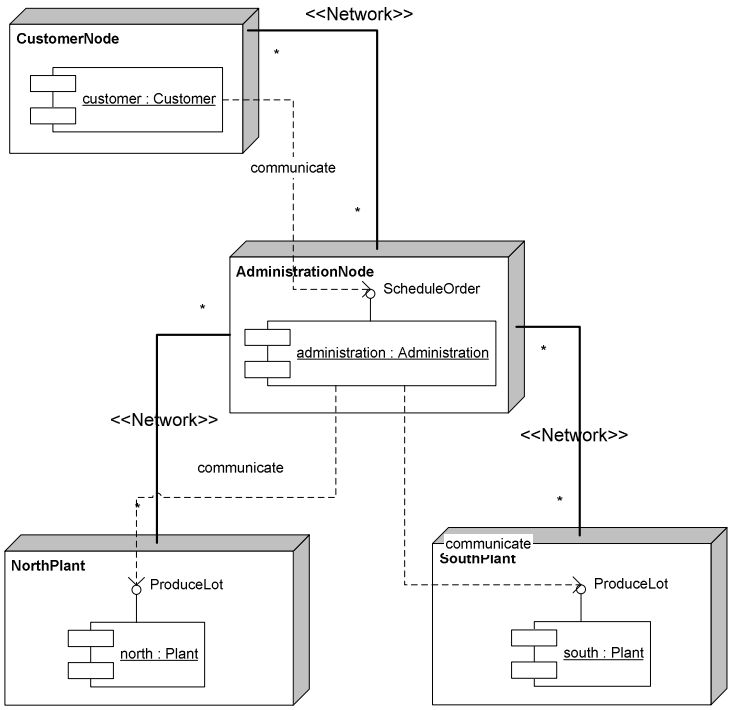

Figure 19. Deployment Configuration diagram for the agents of the PPS-Bikes' system

5.4.7 Agent and Society Test

The testing activity in PASSI has been split into two different steps: the (single) agent test and the society test. During the agent test, the aim is to verify whether each single agent respects its specifications as these can be derived from the different design steps. Most test cases can be derived from the use cases that constitute the agent functionality specification as described in the Agent Identification diagram.

In the society test, the validation of the correct interaction of the agents is performed to ascertain that they concur in solving problems requiring cooperation. Only at this stage is it possible to verify whether the expected social behavior is achieved and the agents interoperate correctly without any problems. This is also the moment for evaluating the system performance in terms of:

- the results provided by the different agents making it up (i.e., if they are able to offer the required services, or to deal with the required amount of data) while interacting with the others in the real operating configuration;
- the effect that the operating environment (network capabilities, host platforms elaboration power and configurations) has on the system.

5.5 Agent implementation

A distinguishing feature of the PASSI methodology is that it covers the whole development process from requirements analysis to code implementation. The aim of this section is to conclude the overview of the agent modeling process with a concrete realization of an agent , starting with the requirements analysis (System Requirements Model) and continuing up to the social representations (Agent Society Model) of the agents involved and their architectural implementation details (Agent Implementation and Deployment models).

In this section a brief description of the programming code derived naturally from the process diagrams will be given. Part of this code has been generated by PTK, and only a few lines have been added manually. The solution presented is an implementation in JADE of a portion of the *Administration* agent already described in the previous design phases (namely, the part dealing with the main agent class and a behavior that initiates a Request communication).

```
1     public class Customer extends Agent {
2        private final String AGENT_NAME = "customer" ;
3        private Order order;
4        private CustomerBook book;
5
6        public Customer() {
7             initialize_customer_book();
8        }
9
10       public void setup ( ) {
11            register_to_df();
12            GUIManagerTask gui = new GUIManagerTask(this);
13            addBehavior(gui);
14       }
15
16       public void register_to_df ( ) {
17               /* this block enables DF registration */
18               try {
19                       // create the agent description of itself
20                       DFAgentDescription dfd = new DFAgentDescription();
21                       dfd.setName(getAID());
22
23                       // register the description with the DF
24                       DFService.register(this, dfd);
25               } catch (FIPAException e) {
26                       e.printStackTrace();
27               }
28       }
29
30       public class RequestWork extends AchieveREInitiator {
...              ...
62       }
63
...       ...
268   }
```

Figure 20. A portion of the code for the *Customer* agent base class

Figure 20 shows a portion of the code for the *Customer* agent. the internal structure of the *RequestWork* behavior has by now been omitted (at lines 30-62) because in a first phase we focus on another issue, that is the agent inner structure represented by its base class.

Line 1 is the declaration of the *Customer* agent as a Java class inherited by the JADE *Agent* class (i.e., the mother class of all JADE agents; see also section 5.2.4). Line 2 defines an agent attribute, called *AGENT_NAME*; there is no difference between an agent attribute and a class attribute, since both of them follow the same (Java) syntax. This attributeknown (a String constant), has been introduced in the agent to contain its name; this value may be used in order to register some agent services to the local Directory Facilitator (DF).

The constructor method (Customer) in this case is used to call another method where the customer book is initialized (not dealt with by our example). Agents' constructors are often used only to initialize data structures, while the agent behavior is delegated to the methods that follow.

The agent *setup* method is declared at line 10. An agent may contain several methods, but some of them are reserved for specific goals. The *setup* method is one of them and it is a mandatory element, since it represents the starting point for all agent activities. Once an agent instance has been created (and its base class constructor executed), the platform registers it automatically to the local Agent Management Service (AMS); it then invokes the agent *setup* method.

In our example, the *Customer* agent *setup* method contains only three instructions:

- (line 11) an invocation to the *register_to_df* method defined some lines later (lines 16-28). This method inserts a new record in the local DF register. The instruction used for this operation is at line 24 (*DFService.register(this, dfd)*); it is put inside a *try – catch* construct to intercept possible exceptions arising during the registration. The *dfd* parameter is a *DFAgentDescription* object and represent the record used to describe the agent to the community. At line 21 this record is initialized with the agent ID value.
- (line 12,13) the *GUIManagerTask* behavior is created and then scheduled with the *addbehaviour* instruction. This is the classic way to start a new agent behavior in JADE. As can be deduced from the MABD diagram reported in Figure 16, this behavior will interact with the user and then call another behavior (the *RequestWork* behavior) that is described more in detail below.

Now we can analyze the structure of the *RequestWork* behavior, which was omitted in Figure 20 (lines 30-62); the complete code is reported in Figure 21. This is not the only behavior of the *Customer* agent (see also the agent structure described in Figure 14 and Figure 15), but it has been chosen because it is a classic communication task. The behavior is declared as a *Customer* agent inner class, and it inherits a JADE core super-class whose name is not univocally defined (as it was for the JADE *Agent* class used to define the agent); in fact, a complex hierarchy of behavior types is provided by this implementation platform and the choice is left to the developer. Each element of the hierarchy has its specific functionalities; for example, the *CyclicBehavior* may be used to create a behavior that cyclically repeats an operation, the *SequentialBehavior* to execute some activities in the specified order, and the *FSMBehavior* to implement a complex finite state machine.

```
30      public class RequestWork extends AchieveREInitiator {
31          private String request_content ;
32          private AID agent_to_request ;
33          private GUIManagerTask gui;
34
35          public RequestWork( Agent owner, AID id, String content, GUIManagerTask gui) {
36              super(owner, new ACLMessage(ACLMessage.REQUEST) );
37              agent_to_request = id;
38              request_content = some_service;
39          }
40          public void handleAgree ( ACLMessage msg ) {
41              gui.notifyOrderAccepted();
42          }
43          public void handleRefuse ( ACLMessage msg ) {
44              gui.notifyOrderRefused();
45          }
46          public void handleInform ( ACLMessage msg ) {
47              gui.notifyOrderSheduled();
48          }
49          public Vector prepareRequests ( ACLMessage msg ) {
50              //automatically invoked by the platform after the class constructor
51              msg.setPerformative(ACLMessage.REQUEST);
52              msg.setProtocol( FIPANames.InteractionProtocol.FIPA_REQUEST );
53              msg.setSender(myAgent.getAID());
54              msg.addReceiver(agent_to_request);
55              msg.setContent(request_content);
56
57              Vector l = new Vector();
58              l.addElement(msg);
59              return l;
60          }
61      }
62
```

Figure 21. Portion of the code for the *RequestWork* behavior of the *Customer* agent

The *RequestWork* behavior starts a Request conversation with the purpose to obtain some service from the *Administration* agent. The JADE API offers an off-the-shelf behavior to initiate a communication by adopting several communication protocols, the *AchieveREInitiator* and the *AchieveREResponder* to deal with the consequent incoming messages.

Line 30 defines the behavior as a class (*RequestWork*) that extends the *AchieveREInitiator* super class. It also has some attributes defined at lines 31-33: *request_content* is a String containing the message content (coded in a specific content language, e.g., RDF) for the initial “request” communicative act. The other attribute, *agent_to_request* (used to address the receiver agent), is an instance of the AID class belonging to the JADE API framework; this is a container for the univocal identifier used to locate an agent within a specific platform. The *gui* attribute is used to store a

reference to the behavior that calls this (*GUIManagerTask*, see Figure 18) in order to notify it with the results of the communication.

The *RequestWork* constructor is defined at lines 35-39. It requires four parameters: the *owner* (a reference to the agent), the *AID* (the receiver agent's unique ID), the *request_content* (the content of the message to be sent) and the *gui* reference to the caller behavior (see above). The first command of this method is a call to the super class constructor that is invoked by specifying, with the first parameter, the owner agent and, with the second parameter, that the message to be used to initiate the protocol is a *request* communicative act. This last parameter is not of paramount importance, since the request message is better defined in the following *prepareRequest* method.

Once the constructor is completed, the *prepareRequest* method (lines 49-60) is automatically invoked for all the *AchieveREIntiator* type behaviors. It returns a vector of *ACLMessage* objects used to initiate the communications with n different agents. The *ACLMessage* class represents the data structure used to contain the message payload of a message (in ACL language) as illustrated in subsection 5.2.3.2. In this method, the *performative*, *protocol*, *sender*, *receiver* and *content* fields of the message are filled in with necessary data. Then, at lines 57-59 the vector *l* is filled in with the message, and the method terminates by returning this vector as a result. At this point the *AchieveREIntiator* super-class actually sends the message to the receiver agent.

Lines 40 to 48 show the definitions of the methods devoted to handling the incoming messages sent by the receiver agent during this communication. It is possible to observe a *handleX* method for each expected communicative act, where the X is the name of the performative (*inform*, *agree*,…).In this way, when an *agree* message reaches the agent the *handleAgree* method is invoked with this message as a parameter.

What can be derived from the code described in this section is that coding FIPA agents under the JADE platform is essentially JAVA coding. The most important difference is not in the actual agent code, but in the communication infrastructure offered by the platform that acts like a middleware, enabling agents of our system to interact easily and relieving the designer of many decisions regarding details. For instance, the designer does not need to know where a mobile agent is at a given moment to code a message for it; the simple agent unique name is sufficient, and the AP will then take care of correctly delivering the message. This, in essence, is the mission of FIPA: to enable the interoperability of heterogeneous software agents.

References

[1] FIPA Request Interaction Protocol Specification. FIPA document n. 00026. 06-12-2002. http://www.fipa.org/specs/fipa00026/

[2] M. Cossentino - "Different perspectives in designing multi-agent systems" - AGES '02 workshop at NODe02 - 8-9 October 2002 - Erfurt, Germany

[3] OMG Unified Modeling Language Specification. Version 1.5. OMG Document formal/03-03-01. March 2003.

[4] The Unified Modeling Language Reference Manual. J. Rumbaugh, I. Jacobson, G. Booch. Addison-Wesley. 1999

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

[6] Bergenti F., Poggi A., "Exploiting UML in the design of multi –agent systems", ESAW Worshop at ECAI 2000.

[7] S. Cranefield and M. Purvis. UML as an ontology modelling language. In Proc. of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99), 1999

[8] Modeling XML applications with UML. D. Carlson. Addison-Wesley. 2001.

[9] FIPA RDF Content Language Specification. Foundation for Intelligent Physical Agents, Document FIPA XC00011B (2001/08/10). http://www.fipa.org/specs/fipa00011/XC00011B.html

[10] FIPA Methodology Glossary. http://www.pa.icar.cnr.it/~cossentino/FIPAmeth/glossary.htm.

[11] J.Odell, H. Van Dyke Parunak, B. Bauer. Representing Agent Interaction Protocols in UML, Agent-Oriented Software Engineering, P. Ciancarini and M. Wooldridge eds., Springer-Verlag, Berlin (2001), 121–140.

[12] Bellifemine, F., Poggi, A., Rimassa, G.: JADE - A FIPA2000 Compliant Agent Development Environment. In Proc. Agents Fifth International Conference on Autonomous Agents (Agents 2001), pp. 216-217, Montreal, Canada, 2001

[13] Poslad S., Buckle P., Hadingham R.: The FIPA-OS Agent Platform: Open Source for Open Standards. Proc. of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents. Manchester, UK, April 2000, 355-368

[14] Gamma, E. Helm, R. Johnson, R. Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995

[15] M. Cossentino, L. Sabatucci, A. Chella - A Possible Approach to the Development of Robotic Multi-Agent Systems - IEEE/WIC Conf. on Intelligent Agent Technology (IAT'03). October, 13-17, 2003. Halifax (Canada)

[16] M.Cossentino, L.Sabatucci, S.Sorace, A.Chella, *"Pattern reuse in the PASSI methodology"* – ESAW'03 workshop – 29-31 October 2003, Imperial College London, UK (EU)

[17] Searle, J.R., Speech Acts. Cambridge University Press, 1969.

[18] Cranefield, S., and Purvis, M. UML as an ontology modeling language. Proc. of the Workshop on Intelligent Information Integration, IJCAI-99 (Stockholm, Sweden, July 1999).

[19] F. Bergenti, A. Poggi. Exploiting UML in the Design of Multi-Agent Systems. In A. Omicidi, R. Tolksdorf, F. Zambonelli, eds., Engineering Societies in the Agents World - Lecture Notes on Artificial Intelligence, volume 1972, pp 106-113, 2000. Berlin, Germany, Springer Verlag Publ.

[20] Bernhard Bauer, Jörg P. Müller, James Odell, Agent UML: A Formalism for Specifying Multiagent Interaction. Agent-Oriented Software Engineering, Paolo Ciancarini and Michael Wooldridge eds., Springer, Berlin, pp. 91-103, 2001.

[21] H. Van Dyke Parunak and James Odell, Representing Social Structures in UML, Proc. of Agent-Oriented Software Engineering (AOSE) 2001, Agents 2001, Montreal, pp. 17-31.

[22] M. Cossentino, C. Potts - "A CASE tool supported methodology for the design of multi-agent systems" - The 2002 International Conference on Software Engineering Research and Practice (SERP'02) - June 24 - 27, 2002 - Las Vegas (NV), USA