Patterns reuse in the PASSI methodology

Massimo Cossentino[1], Luca Sabatucci[1], and Antonio Chella[1],[2]

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, Consiglio Nazionale delle Ricerche
Viale delle Scienze, 90128 Palermo, Italy
[2] Dipartimento di Ingegneria Informatica - University of Palermo
Viale delle Scienze 90128 Palermo, Italy
{cossentino, sabatucci}@pa.icar.cnr.it, chella@unipa.it
http://www.pa.icar.cnr.it

Abstract. Design patterns already proved successful in lowering the development time and number of errors of object-oriented software; now, they are, candidate to play a similar role in the MAS (multi-agent system) context. In this work we describe our experiences in the identification, production and application of patterns for agents. Some patterns are described together with the classification criteria and documentation approach we adopt. Upon them, we base a pattern reuse process that can be considered one of the distinguishing elements of the design methodology (PASSI) we use to develop MAS. Patterns can be applied to an existing agent or used to produce a new one with the support of a specific web based application that can read both the JAVA source code and XMI representation of the agent design documentation. After the successful application of the desired pattern(s), the source code and the design diagrams (usually a structural and dynamic diagram) of the agent can be exported. Some experimental results are reported in order to demonstrate the utility of this approach in automatically producing an interesting percentage of code lines.

1 Introduction

In the last years, multi-agent systems (MAS) achieved an unprecedent success and diffusion; as an example, e-commerce applications are growing up quickly, they are leaving the research field and the first experiences of industrial applications are appearing. These applicative contexts require high-level qualities of design as well as secure, affordable and well-performing implementation architectures. In our research we focus on the design process of agent societies considering that this activity implies not only modeling an agent in place of an object but also capturing the ontology of its domain, representing its interaction with other agents (social aspects), and providing it with the ability of performing intelligent behaviors. Several scientific works that address this topic can be found in literature; it is possible to note that they come from different research fields: some come from Artificial Intelligence (Gaia [1]) others from Software Engineering (MaSE [2], Tropos [3]) but there are also methodologies coming

directly from Robotics (Cassiopeia [4]). They give different emphasis to the different aspects of the process (for example the design of goals, communications, roles) but almost all of them deal with the same basic elements although in a different way or using different notations/languages. In the following, we will pursuit a specific goal: lowering the time and costs of developing a MAS application. In order to obtain this result, we think that a fundamental contribution could come by the automation of as many steps of the process as possible (or similarly by providing a strong automatic support to the designer). In pursuing these objectives we developed a design methodology (PASSI, "Process for Agent Societies Specification and Implementation" [5]) specifically conceived to be supported by a CASE tool that automatically compiles some models that are part of the process, using the inputs provided by the designer. PASSI 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 UML notation. We widely applied it in the design of robotics applications [6] but it also proved successful in designing information systems [7]. In PASSI (Fig. 1), the reuse of existing patterns has a great importance. Unlike other authors (Kendall [8]), we chose to introduce a pattern definition conceived for MAS belonging to a specific architecture (the FIPA one). As a consequence, all the agents we consider in our pattern, have a similar structure and their behavior refers to a finite state machine. All of these simplifications were useful in order to create a pattern that affects all the stages of the agent development process. Our pattern is a representation and implementation of some kind of (a part of) the system behaviors that solves a recurrent problem (for example a specific type of agents' interaction). During a PASSI design process, the designers will use a Rational Rose add-in that we have specifically produced. In this procedure they move gradually from the problem domain (described in the System Requirements Model and Agent Society Model) towards the solution domain (mainly represented by the Agent Implementation Model) and, passing through the coding phase, to the dissemination of the agents in their world. It is in this progress of activities that they can identify some problems that could be profitably solved reusing the patterns of our repository. The choice of the implementation platform is postponed to the final steps of the design and in order to support the localization of our patterns in both the most diffused FIPA [9] platforms (FIPA-OS [10] and JADE [11]) we represent the models and the code of each pattern using XML. In the case of the models we use the diffused XMI representation of the UML diagrams while for the code we introduced some intermediate levels of representation in XML from which we obtain the final Java code using several XSL transformation.
The remaining part of the paper is organized as follows: section 2 describes our approach to the classification of agent patterns; section 3 provides a discussion on the process used to construct the code for a specific execution environment starting from the platform-independent pattern; section 4 discusses where patterns can be identified during the design process and their impact in it; numerical

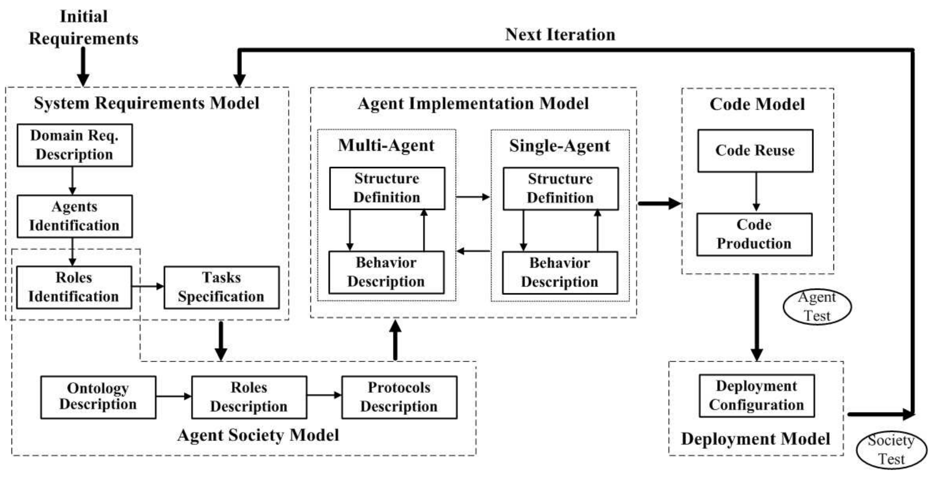

Fig. 1. The PASSI methodology

results about the amount of automatically generated lines of code are provided in section 5 and finally some conclusions are drawn in the final section.

2 Patterns Classification

A classic approach to patterns classification, can be found in [12], where patterns are classified by two criteria: purpose and scope. With purpose the authors refer to what the pattern does and according to this, they enumerate creational (dealing with the process of object creation), structural (dealing with the composition of classes and objects) or behavioral (describing the interactions of classes/objects) patterns. The scope classification is directed to separate patterns that apply to classes or object. While the first category (class patterns) deal with the relationships among classes (usually structural, for example inheritance), the second one includes relationships among object (usually dynamic and established at runtime).

A different, more agent-oriented, classification is proposed by Lind in [13]. Patterns are classified in accordance to the views defined by the author in the MASSIVE methodology [14]. Categories reported in this work are: Interaction, Role, Architecture, Society, System, Task, and Environment.

Another classification can be found in [15] where patterns are clustered in the traveling (dealing with agent mobility issues), task (regarding the breakdown of agents tasks and the delegation of them from one agent to another) and interaction (dealing with agents communications) categories. Recently another possible category (adaptability patterns [16]) has been proposed by other authors.

We think that the basic duality that exists in software between structure and dynamic behavior is someway captured by the Gamma's approach but we also

		Application Context			
		Action	*Behavior*	*Component*	*Service*
Functionality	*Access to local resources*	53		Generic Agent, Parallel Resource Sharing, Sequential Resource Sharing, Publish-Subscribe, Resource Caching	
	Communication	66	Request (I/P), Query (I/P), Inform (I/P), ContractNet (I/P)		Request, Query, Inform, Contract-Net
	Elaboration	44		Planner	
	Mobility	7			Explorer

Table 1. The classification of the patterns in our repository. (I/P) indicates the presence of both the Initiator and Participant role patterns. Action patterns are not listed by name because of their number.

consider the importance of enriching it in order to fit the specific context (agents not objects). For this reasons we decided to classify the patterns of our repository using two main criteria (like in Gamma et al. [12]) and to include agent specific subcategories in them. The first criterion is the *application context*; it regards the structural aspects of the pattern, whether it is to be applied to one agent, more agents or to their composing elements (tasks/behaviors). The second criterion is the *functionality* expressed by the pattern. We enumerated four kinds of patterns in the first category:

Action patterns. They address a functionality of the system; for instance they can be implemented as a method of either an agent class or a task class.

Behavior patterns. They address a specific behavior of an agent; we can look at each of them as a collection of actions.

Component patterns. They are entire agent patterns; these patterns propose a solution composed of the entire structure of an agent together with its tasks.

Service pattern. Concerned about the collaborations between two or more agents; they can be thought as an aggregation of component patterns.

Again, looking at the functionality of the patterns, we can consider four categories:

Access to local resources. They deal with information retrieval and manipulation of data source.

Communication. They represent the solution to the problem of making two agents communicate by an interaction protocol.

Elaboration. They are used to deal with the agent's functionality devoted to perform some kind of elaboration on its knowledge.

Mobility. These patterns describe the possibility for an agent to move from a platform to another, maintaining its knowledge.

In Table 1 we can find a classification of the patterns in our repository. Some of them have proved particularly useful for building up a totally new agent (the *GenericAgent* component pattern includes the base functionalities that an agent needs in order to register to the platform and yellow pages service) or enriching the capabilities of an existing one (the *Request Initiator* behavior pattern can be applied to an agent to give it the possibility of initiating a communication with the FIPA Request interaction protocol).

3 From design representation to multi-platform coding of patterns

Patterns can contribute to significantly enhance the quality of the software design and this is one of the reasons that justify their diffusion. In addiction to this argument, we think that under precise hypothesis, patterns can also provide another important contribution, to the development process: enhancing the amount of code produced with a CASE tool. The hypothesis we assume to achieve this goal regards the implementation domain.

If we suppose that all the agents will be implemented with a specific FIPA-compliant platform, then we define the structure of the implementation obtaining an extreme simplification. This is a very restrictive hypothesis that can be done in a specific company but it is too limiting for a wider research purpose. Therefore, we decided to study the possibility of reducing the implementation platforms to a little number in order to obtain some kind of generality for our approach. In this first phase we selected two different platforms (JADE [11] and FIPA-OS [10]) that, together represent a greater part of the installed platforms in the Agentcities EU initiative [17].

3.1 Differences in FIPA-compliant implementation platforms

In order to understand how the choice of coding our patterns for the JADE and FIPA-OS platforms effects the proposed solution we will now describe the main differences of two different implementations (FIPA-OS and JADE) of the same agent. We will initially propose a FIPA-OS implementation of a simple agent that once started invokes a task (behavior in the JADE platform implementation) in order to accomplish its duty. In so doing this task uses the results provided by another one. Both the presented implementations (fig. 2 and fig. 3) are documented using a class diagram that shows the structural differences between the two agents and an activity diagram that reports the flow of control.

In fig. 2 we can see the structure of the FIPA-OS agent, it is composed of the main agent class and two task classes. Its behavior consists of:

1. The execution of the agent's constructor and *setup* methods.

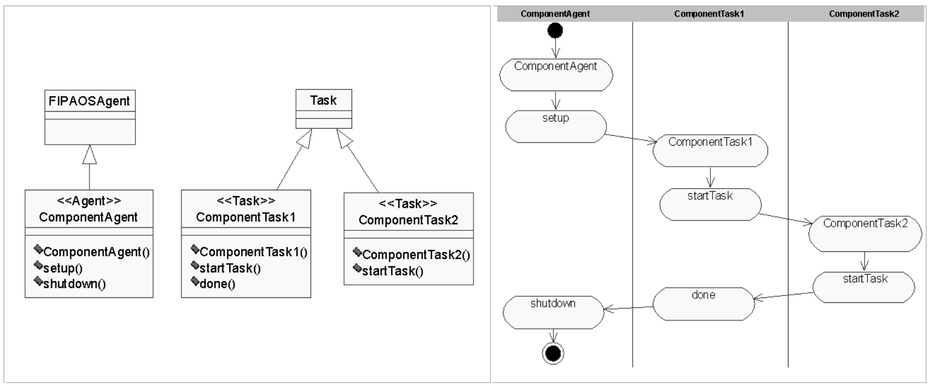

Fig. 2. A simple FIPA-OS agent (left) and its behavior (right)

2. The instantiation of the *ComponentTask1* task (a piece of behavior in the FIPA-OS terminology), invoked by the agent's *setup*. In the FIPA-OS platform at the moment of the instantiation of a new task, the execution of the constructor it is followed by the execution of its *startTask* method. While the constructor is often used to initialize variables, the *startTask* method is commonly used to actuate the specific initial behavior of the task.
3. The *startTask* method of *ComponentTask1* instantiates *ComponentTask2* (another task, for instance devoted to elaborate some kind of data). Again, after the constructor, the *startTask* method is executed.
4. The conclusion of the execution of *ComponentTask2* is stated by a *done* command in the last line of the code of its *startTask* method. The result is that the *done* event is sent to the task that invoked this one. This event is intercepted by the *done* method in *ComponentTask1*. One different *done* method is necessary, in FIPA-OS systems, in the calling task for each other invoked task (the link between the instantiated task and the related *done* method is based on the name of the method that is in fact composed by the *done* prefix and the invoked task name).

In fig. 3 we can see the JADE implementation of the agent described before. We can see that the structure is very similar (considering that task classes in FIPA-OS are named behavior in JADE). One of the differences, only formal, is in the name of the FIPA-OS *startTask* method, that is called *setup* in JADE. Another difference consists in the *done* method. In JADE we have one done method for each behavior class, its duty is related to the behavior that is hosting it, not to an invoked one. The platform scheduler verifies the value reported by the done method of each behavior and if it is true the class will not be rescheduled. There is no necessity of introducing a method in a behavior who calls another one. Another difference can be found in the *shutdown* method, present in the FIPA-OS agent that is not necessary in the JADE agent.

As we can see in fig. 2 and in fig. 3 these little differences are clearly shown in both the structural and dynamical diagrams. This fact concretely affects our

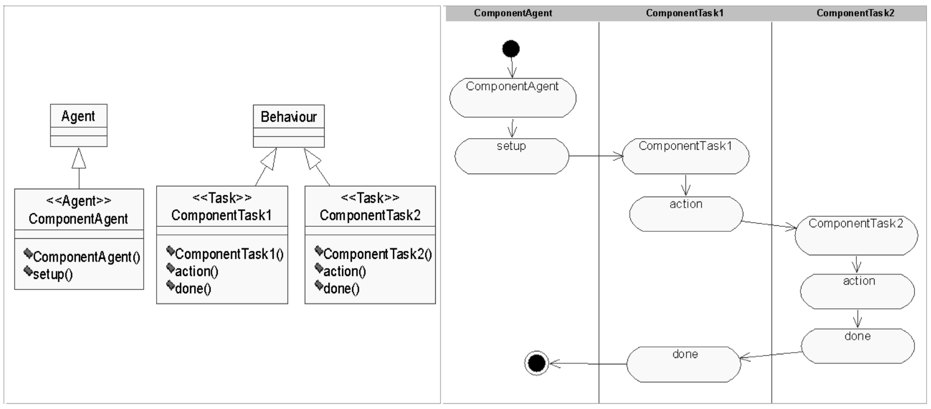

Fig. 3. A simple Jade agent (left) and its behavior (right)

work and we introduced in our patterns, a specific instrument (constraints, described in the next subsection) to deal with it.

3.2 Pattern Definition

These platforms use the same coding language (JAVA) and share several similarity. Starting from these working hypothesis, we produced a meta representation of all our patterns (see fig. 4) that describes the pattern not considering implementation specific issues. This meta-pattern can be derived from the repository and can be used to generate portions of the design diagrams through an XML-XMI transformation. From the same meta-pattern we instantiate the platform specific pattern and then we generate the related code. In the following sections we will briefly describe this process.

Meta-Patterns. The key element of our multi-platform implementation of patterns consists in the introduction of meta-patterns. They are platform independent and contain all the elements that are common to patterns of the different environments. For example meta-patterns refers to class constructors, mother-classes from which agents and their standard elements inherit their behavior, setup and shutdown methods and so on. Meta-patterns are described using XML and can be used to both generate portions of design diagrams and the platform-specific pattern description.

Patterns. Applying an XSL transformation we substitute the meta-level placeholders with the name used in the selected platforms and if the case the values introduced by the designer (for example the specific name of the agent or some parameters). In this XML file an agent is described inside an Agent tag. Agent properties such as attributes or tasks are represented as inner elements of the structure.

Patterns and constraints When a pattern is applied to a project it modifies the context in which it is placed, for instance introducing new functionalities

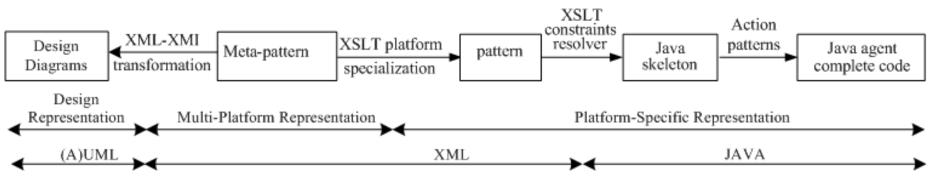

Fig. 4. The process used to obtain the platform specific pattern from its meta-representation

into the system. These additions need to satisfy some constraints (e.g. in FIPA-OS, when we insert a communication task pattern into an existing agent, the listener task should have a handleX method to catch performative acts of a particular type). This relationship between the pattern and the existing elements could be expressed with a constraint. A constraint is a rule composed of two elements: a target and a content. The target specifies what agent/task will be influenced by the rule. The content expresses the changes to be applied when the pattern is inserted into the project; it could be an aggregation of attributes, constructors or methods. After localizing the pattern for the specific platform we apply all the constraints related to it using another XSL transformation.

Multi-platform code generation. As briefly mentioned before, XSLT application grants to export an agent described with our meta-language into a specific programming language. This is possible because the pattern at this stage intrinsically represents an implementation viewpoint. As a matter of fact, UML classes (of the design representation) correspond to Java classes, and UML attributes and methods correspond to the Java classes' attributes and methods. This allows us to look at the source code as one of the possible views of an agent: we could imagine this agent representation as an intermediate layer between agent design and agent development. The use of XSLT enables code generation for both FIPA-OS and Jade frameworks by only changing the transformation sheet. At this stage the JAVA skeleton of the agent (and it's tasks) is complete. In order to (partially) fill the skeleton with the remaining code, action patterns are applied. An action pattern is a portion of code realizing some kind of behavior, for example the registration to the DF (Directory Facilitator) service, the yellow pages of the platform. Action patterns are stored in a database of pieces of code and the correct one for each method is selected referring to the value of the *Code* tag for the specific module.

3.3 Pattern Repository Population

The process of patterns identification has been divided in two different stages. In the first one, on the basis of our previous experiences with MASs, we prepared several common use, easily identifiable patterns. Among these patterns we have the *GenericAgent* one; this implements the basic behaviors of each agent (registration with the agent platform and yellow pages service). Other patterns that we identified in this phase are the communication related ones. We produced a set of patterns to deal with communications based on the most diffused FIPA standard protocols (AIP, Agent Interaction Protocols) at different levels (a behavior that can initiate or participate to a communication, an agent with the same ability or a couple of agents interacting according the same AIP).

In the next step we analyzed several existing applications in order to identify portions of the solution that can be generalized for reusing in other contexts. In this way we identified, for example, the *Planner* and the *Explorer* pattern.

The *Planner* pattern is a component pattern (i.e. it describes an entire agent) and it implements the structure necessary to perform some kind of planning activity for reaching a goal. It partitions the system architecture in almost three levels: a superior strategic level, an intermediate planner level and an inferior actuator level. We could explain the usefulness of this pattern by applying it in a classic robotic mission, the exploration of an unknown environment. The strategic level responsibility consists in selecting the goal to pursue from a set of possible choices, in the navigational example it selects the direction to explore from all the possible alternatives. When the strategic level has selected a goal (or sub-goal), it passes this information to the planner level; this has access to the already discovered map of the environment and builds the plan. When the plan is ready the last level provides commands to some actuator agents that enable the robot movements.

The *Explorer* pattern allows the exploration of remote sites with the intent of searching for some information. A typical scenario that illustrates the scope this pattern is represented by web searching. It is a service pattern and therefore it is composed of two agents: the *base* agent and the *explorer* agent. The first one is responsible for the searching activity and can create one or more *explorer* agents that will move to the target sites. Each *explorer* is associated to a destination and when created, it moves to reach it; once there it starts the searching activity and then reports the results using a communication based on some kind of interaction protocol.

The pattern repository is actually composed (see Table 1) of five service patterns (regarding problems solved by the participation of at least two agents), six component patterns (an entire agent is involved in the problem), eight behavior patterns (including behavior-level solutions) and a consistent number of action patterns (solutions at the class method level, some of them extracted from the previous patterns).

4 Reusing patterns

In order to better describe where we can introduce and reuse patterns with PASSI we should start defining what we mean by pattern. We accept the definition provided by Christopher Alexander [18]: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core solution to that problem". A pattern can be found (or applied) wherever a problem is present. If we consider our pattern classification we can conclude that we can reuse patterns when we face a local resource access, communication, elaboration of mobility problem. This reuse can have different scopes: a collaboration among two agents can be accomplished by a service pattern, a single agent with a particular vocation can be realized applying a component pattern, specific features of an agent can be delegated to a behavior pattern and simple actions can be done with action patterns. As the design process starts from an high level of abstraction and step by step goes down to the more detailed aspects of the solution we should expect that larger patterns (services, components) will be identified in the initial phases of the design while smaller ones (behaviors, actions) could arise during the final design choices. Now we will discuss these aspects with regard to the PASSI methodology in order to describe our experiences.

In PASSI from the functional requirements description (use case diagrams of the Domain Requirements Description phase), we obtain the agent's responsibilities in terms of the functionalities they will provide. This happens in the Agent Identification diagram that is an use case diagrams where packages are used to enclose use cases that will be under the responsibility of the same agent. Stereotypes of relationships among use cases belonging to different agents are changed to *communicate* (not standard in UML) since they represent interactions among different active elements (agents) of the system. In this diagram we could easily identify patterns of higher structural levels (service and component) as described in the following scenario.

Let us consider a supply chain whose purpose is to ensure the availability of raw materials for the production chain of some manufacturing company. In fig. 5 we can see a part of the A.Id. diagram of this application; the scenario we are dealing with involves the following agents (see the package name): *StockGuardian*, *PurchaseAgent* and *SupplierAgent*. The *StockGuardian* is responsible of looking at the amount of raw materials and starting the supplying process once they go below a defined level (depending on the scheduled work). It calculates the quantity of materials that is necessary to buy and asks to the *PurchaseAgent* of supplying them (*SupplyingRequest* use case). The *PurchaseAgent* starts an auction (in the proposed implementation not really an auction since the process will be based on the ContractNet interaction protocol) to buy at the best price. The possible suppliers are selected considering the previous experiences of the company with them (*CompileGoodSupplierList* and *SuppliersEvaluation* use cases. Each selected supplier receives a notice and then, can post his own bids (*IntroduceBidParameters* use case) for the auction, interacting (via web) with the instance of the *SupplierAgent* that has been devoted to him (the access is password protected). Suppliers achieve an highest level of security installing an agent platform in their company. In this case the *SupplierAgent* will move to his owner host.

Now suppose that we want to apply patterns to this diagram. We can decide to use an Explorer service pattern in order to realize the basic part of the *SupplierAgent*. This pattern allows to an agent the exploration of remote agent platforms with the intent to perform some kind of operation in them; it is com-

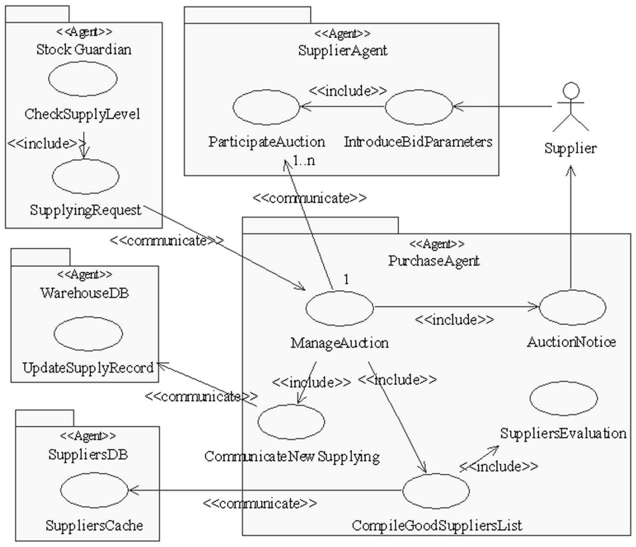

Fig. 5. Part of the Agent Identification diagram of a manufacturing plant software solution

posed of two agents: the base agent will create an explorer agent and will send it to the other platform(s). The explorer agent will perform the required (and not described in the pattern) operation and then will inform its base agent. We can apply the base agent part of the pattern to the *PurchaseAgent* and the explorer part to the *SupplierAgent*. The auction communications can be implemented by a ContractNet service pattern (the initiator part will be applied to the *PurchaseAgent* and the participant part to the *SupplierAgent*). Several communications can be identified in the diagram and these can be realized using the corresponding service pattern (for instance the Request service pattern that is a communication pattern providing the initiator and participant roles for a communication based on the FIPA Request interaction protocol).

Summarizing, in the agent identification diagram reported in fig. 5 we can identify six service patterns and one component pattern. It should be noted that many of them cannot arise from the analysis of a standard UML use case diagram; specifically, communications among agents (that are pointed out by the *communicate* relationship among use cases of different agents) are present in the A.Id. diagram since it shows a particular hypothesis of solution (an agent-based system).

4.1 Impact of patterns in the design

It is interesting to note how relevant can be the backlash of the previous identified patterns in the other phases of the PASSI design. In the following we will describe two different typologies of possible consequences: the guidance that the reused pattern gives to the designer in his/hers (manual) operations, and the support that can be automatically provided by a tool in reusing that pattern (some of these features are already supported by PTK, the Rational Rose add-in that we built in order to design with PASSI).

After the A.Id. phase, the designer performs the Roles Identification where he/she describes the scenarios arising from the use case diagrams using several sequence diagrams. When the scenarios involves the identified pattern, the designer receives a guidance from the sequence diagram that illustrates the collaborations of the pattern. During the exploitation of these scenarios, a first draft of the MASD (Multi-Agent Structure Diagram) is built. This is an high level representation of the agent society structure and interaction. It is a class diagram reporting each agent as a class and each agent behavior as a method of this class; relationships among agents represent their communications. The choice of reusing a pattern obviously has a direct (and tool-supported) effect on this diagram. In fact, new behavior and agents are introduced according to the pattern structure.

The Task Specification (T.Sp.) phase is the next one and this also offers an opportunity to identify behavior patterns. In this step, one different activity diagram is drawn for each agent. It represents a possible decomposition of the agent functionalities into a series of activities (that are candidate to be implemented as behaviors). As a consequence, it is possible to find behavior patterns in these diagrams. Their compilation is also strongly affected by previous introduced component and service patterns. Suppose that in the previous discussed manufacturing company scenario, we decide to introduce a *Resource caching* pattern in the *SuppliersDB* agent in order to implement a caching mechanism for minimizing accesses to the DB. In fig. 6 we can see the *Resource caching* pattern; this diagram can be largely reused in order to build the T.Sp. diagram of the *SuppliersDB* agent and PTK support this.

We have now completed the system requirements model and only minor effects we will find in the following agent society model (for example in the communication ontology description diagram we will complete the description of all the communication including the pattern supported ones).

The third PASSI model (agent implementation) is strongly related to the final coding activities and almost all of its phases receive an influence from selected patterns. This model is composed of a structural and behavioral definition of the MAS that is performed at both the multi and single agent level. As a result we have two structural diagrams and two behavioral diagrams: the multi-agent structure diagram (MASD, one unique diagram representing each agent of the society as a class and communications among agents as relationships, see fig. 7), the single-agent structure diagram (SASD, one different class diagram for each agent, reporting all the agent implementation structure, it is the nearest to the code), the multi-agent behavior diagram (MABD, an activity diagram representing the agents and their behaviors) and a single-agent behavior diagram (SABD, used to represent the algorithmic aspects of the solution). Finally the

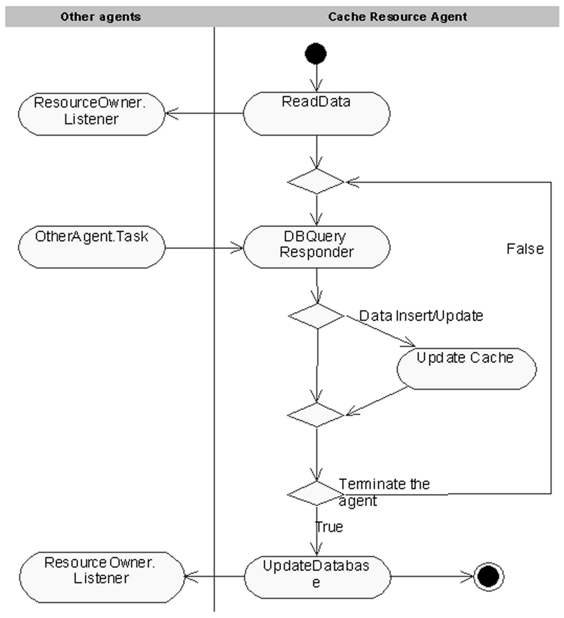

Fig. 6. The Resource cache pattern can store some data from a DB and provide it to the other interacting agents without repeatedly querying the DB

code is generated, also reusing patterns from the repository and the designer after having manually completed the agents, can deploy them.

In section 5 we will report some numerical results about the contribution that the automatic code generation can give in developing the cited manufacturing application.

5 Experimental Results

In this section we will report the results obtained applying our methodology and patterns to an application built to support the production and supply chains of a manufacturing company.This is not a theoretic exercise because the introduction of this software in the real productive environment is ongoing.

Because of the large dimension of the whole application we will here describe only a part of it and specifically, we will focus on the supply chain whose purpose is to ensure the availability of raw materials to the production chain.

The initial application has been designed following the PASSI methodology [5] but without a considerable use of patterns, since their repository was almost empty at that time. Now, in our experiment we will reproduce it applying the patterns with the support of PTK and the AgentFactory tool (the pattern reuse tool we integrated in PASSI and that is also available as a standalone or an on-

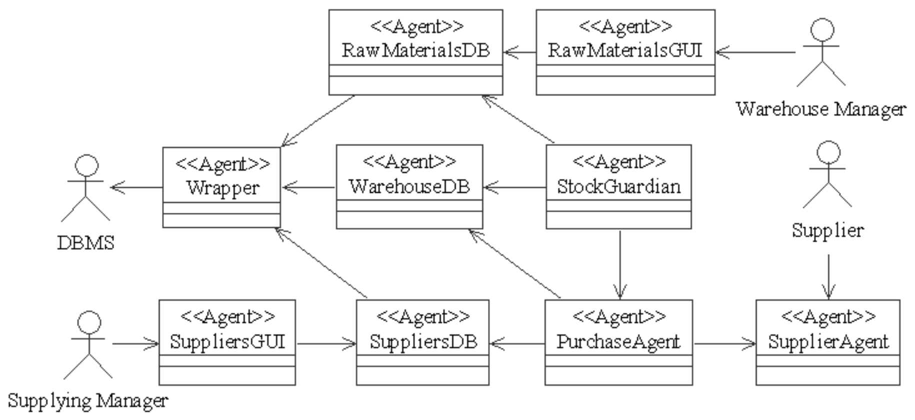

Fig. 7. A class diagram reporting the agents involved in the experiment and their relationships (communications)

line application[3]). The identification of the best suited patterns will be performed on the PASSI diagrams as discussed in section 4. Then we will compare the number of lines of code (LOC) of the original agents with the number of lines of code obtained by the patterns application.

In fig. 7 we can see a class diagram reporting the agents involved in the experiment. Each class in the diagram represents an agent, and its relationships stand for the agent's communications. Agents' attributes (knowledge) and operations (tasks) are not reported for clarity but they are present in the original diagram that is a Multi-Agent Structure Definition diagram of the PASSI methodology.

Here is a description of the way these agents interact to provide the production chain with the necessaries supplies:

The *Warehouse Manager* interacts with the *RawMaterialsGUI* agent running on his laptop or PDA in order to describe the raw materials needed for each artifact. This data is received by the *RawMaterialAgent* (responsible for everything concerning the raw materials) that registers it in the central DBMS asking for the collaboration of the *Wrapper* agent (responsible for the interactions among the agent-based application, the DBMS, and other existing OO software). The *StockGuardian* agent supervises the levels of supplies (also considering the quantity of products that the production chain is going to deal with). When the stock of some material goes under a specified level, this agent starts the supplying process with a message to the *PurchaseAgent* agent. The acquisition of new materials is an automatic process performed by the *PurchaseAgent* agent interacting with several *SupplierAgent* agents; each one of them is responsible for the interaction with one of the real suppliers that have been introduced by the *Supplying Manager* in a specific list. The agents involved in the commercial

[3] Website: http://mozart.csai.unipa.it/af/

Agents	Total LOC	Automatically Generated LOC	% of Total	Methods Body LOC	% of Autom. Generated
SuppliersDB	284	162	57	85	52
Wrapper	790	66	8	35	53
RawMaterialsDB	290	144	49	78	54
SuppliersAgent	124	43	34	26	60
PurchaseAgent	304	133	43	79	59
WarehouseDB	219	162	73	87	53
StockGuardian	109	98	89	49	50
RawMaterialsGUI	71	36	50	19	52
SuppliersGUI	69	36	52	19	52
Totale	**2260**	**880**	**39**	**477**	**54**

Table 2. Statistics of the experiment

transaction interact using the standard FIPA Contract Net interaction protocol [19].

Each agent of this scenario has been built using one or more patterns. In Table 2 we summarize the results of this experiment. Globally this portion of the system is composed of about 2000 lines of code subdivided into nine agents. The result of building the agents with patterns is that we obtain a base code of 880 lines: the 43% of the entire system. In this code, automatically generated, we can separate the class skeletons (generated by several CASE tools) from the methods body (not generated by conventional CASE tools). The methods body, represents the behavior and the actions of the agents. The amount of this code is the 54% of the automatically generated one; usually this should be entered and tested manually by skilled programmers. With the use of our approach this code (already tested) is automatically generated with a minimal amount of work and this allows a remarkable saving in time and a positive effect on the overall quality.

6 Conclusions

Our conviction is that pattern reuse is a very challenging and interesting issue in multi-agent systems as it has been in object-oriented ones. However we are aware that the problems arising from this subject are quite delicate and risky. Nonetheless, we believe, thanks to the experiences made in fields such as robotics, that it is possible to obtain great results with a correct approach. In the previous sections we discussed the impact of pattern reuse in PASSI, a complete design methodology for multi-agent systems that is supported by PTK (PASSI ToolKit), an add-in for Rational Rose, and AgentFactory, a pattern reuse tool. The use of this methodology and the related tools allowed us the construction of significant projects (as an example here we reported the rebuild of part a large application) with very good results in terms of autmatically generated code. Our multi-phase approach to the representation of patterns, based on XML, allows us the generation of agents' code for two different multi-agents FIPA-compliant operating environments (FIPA-OS and JADE) starting from platform-independent design patterns. In order to cover the entire process we use different representation languages (UML, XML and JAVA for the final code) and apply several transformations. Experimental results have demonstrated the goodness of the approach that is however strongly effected by the number of patterns in the repository and the support that the tool offers to designer in terms of automatically performed operations. We are now working in order to increase the number of patterns and to enhance the functionalities of our design tool.

References

1. Wooldridge, M., Jennings, N.R., Kinny, D.: The gaia methodology for agent-oriented analysis and design. Journal of Autonomous Agents and Multi-Agent Systems **3** (2000) 285–315
2. DeLoach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent systems engineering. International Journal on Software Engineering and Knowledge Engineering (**11**) 231–258
3. Castro, J., Kolp, M., Mylopoulos, J.: Towards requirements-driven information systems engineering: The tropos project. In: To appear in Information Systems, Elsevier, Amsterdam, The Netherlands (2002)
4. Collinot, A., Drogoul, A.: Using the cassiopeia method to design a soccer robot team. Applied Articial Intelligence (AAI) Journal **12** (1998) 127–147
5. Cossentino, M., Potts, C.: A case tool supported methodology for the design of multi-agent systems. In: The 2002 International Conference on Software Engineering Research and Practice, Las Vegas (NV), USA, SERP'02 (2002)
6. Chella, A., Cossentino, M., Pirrone, R., Ruisi, A.: Modeling ontologies for robotic environments. In: The Fourteenth International Conference on Software Engineering and Knowledge Engineering, Ischia, ITALY (2002)
7. Burrafato, P., Cossentino, M.: Designing a multi-agent solution for a bookstore with the passi methodology. In: Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002), Toronto (Ontario, Canada) (2002)
8. Kendall, E.A., Krishna, P.V.M., Pathak, C.V., Suresh, C.B.: Patterns of intelligent and mobile agents. In Sycara, K.P., Wooldridge, M., eds.: Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98), New York, ACM Press (1998) 92–99
9. O'Brien, P., Nicol, R.: Fipa - towards a standard for software agents. BT Technology Journal **16** (1998) 51–59
10. Poslad, S., Buckle, P., Hadingham, R.: The fipa-os agent platform: Open source for open standards. In: 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents, Manchester, UK (2000)
11. Bellifemine, F., Poggi, A., Rimassa, G.: Jade - a fipa2000 compliant agent development environment. In: Agents Fifth International Conference on Autonomous Agents (Agents 2001), Montreal, Canada (2001)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns Elements of Reusable Object Oriented Software. Addison-Wesley (1994)
13. Lind, J.: Patterns in agent-oriented software engineering. In: AOSE Workshop at AAMAS 2002, Bologna, Italy (2002)

14. Lind, J.: The Massive Development Method for Multiagent Systems. In Bradshaw, J., Arnold, G., eds.: Proceedings of the 5th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 2000), Manchester, UK, The Practical Application Company Ltd. (2000) 339–354
15. Aridor, Y., Lange, D.B.: Agent design patterns: Elements of agent application design. In: Autonomous Agents '98. (1998)
16. Dikenelli, O., Erdur, R.C.: Adaptability patterns of multi-agent organizations. In: this volume. (2003)
17. Agentcities.NET: (http://www.agentcities.net)
18. Alexander, C.: The Timeless Way of Building. Oxford University Press (1979)
19. Foundation for Intelligent Physical Agents: FIPA Interaction Protocol Library Specification. (2000)